Перевод Максима Каверина (ahinar@list.ru)
Источник - http://www.progz.ru/forum/index.php?showforum=91

OpenGL Red Book (русская версия)

Содержание

Глава 1. Введение в OpenGL

1.1 Что такое OpenGL?

OpenGL – это программный интерфейс к графической аппаратуре. Этот интерфейс состоит приблизительно из **250** отдельных команд **(**около **200** команд в самой **OpenGL** и еще **50** в библиотеке утилит**),** которые используются для указания объектов и операций, которые необходимо выполнить, чтобы получить интерактивное приложение, работающее с трехмерной графикой.

Библиотека **OpenGL** разработана как обобщенный, независимый интерфейс, который может быть реализован для различного аппаратного обеспечения. По этой причине сама **OpenGL** не включает функций для создания окон или для захвата пользовательского ввода; для этих операций вы должны использовать средства той операционной системы, в которой вы работаете. По тем же причинам в **OpenGL** нет высокоуровневых функций для описания моделей трехмерных объектов. Такие команды позволили бы вам описывать относительно сложные фигуры, такие как автомобили, части человеческого тела или молекулы. При использовании библиотеки **OpenGL** вы должны строить необходимые модели при помощи небольшого набора геометрических примитивов **–** точек, линий и многоугольников **(**полигонов**).**

Тем не менее, библиотека, предоставляющая описанные возможности может быть построена поверх **OpenGL.** Библиотека утилит **OpenGL (OpenGL Utility Library -- GLU)** предоставляет множество средств для моделирования, например, квадрические поверхности, кривые и поверхности типа **NURBS. GLU –** стандартная часть любой реализации **OpenGL.** Существуют также и более высокоуровневые библиотеки, например, **Fahrenheit Scene Graph (FSG),** которые построены с использованием **OpenGL** и распространяются отдельно для многих ее реализаций.

В следующем списке коротко описаны основные графические операции, которые выполняет **OpenGL** для вывода изображения на экран.

1. Конструирует фигуры из геометрических примитивов, создавая математическое описание объектов **(**примитивами в **OpenGL** считаются точки, линии, полигоны, битовые карты и изображения**).**
2. Позиционирует объекты в трехмерном пространстве и выбирает точку наблюдения для осмотра полученной композиции.
3. Вычисляет цвета для всех объектов. Цвета могут быть определены приложением, получены из расчета условий освещенности, вычислены при помощи текстур, наложенных на объекты или из любой комбинации этих факторов.
4. Преобразует математическое описание объектов и ассоциированной с ними цветовой информации в пиксели на экране. Этот процесс называется растеризацией **(**или растровой разверткой**).**

В течение всех этих этапов **OpenGL** может производить и другие операции, например, удаление частей объектов, скрытых другими объектами. В дополнение к этому, после того, как сцена растеризована, но до того, как она выводится на экран, вы можете производить некоторые операции с пиксельными данными, если это необходимо.

В некоторых реализациях **(**например, в реализации для системы **X Window)**, **OpenGL** разработана таким образом, чтобы работать даже в том случае, если компьютер, которые отображает графику не то же самый, на котором запущена ваша графическая программа. Это может происходить в случае, если работа происходит в сетевом окружении, состоящем из множества компьютеров, соединенных между собой. В данной ситуации компьютер, на котором функционирует программа, и вызываются команды **OpenGL**, является клиентом, в то время как компьютер, осуществляющий отображение, является сервером. Формат пересылки команд **OpenGL** от клиента серверу **(**или протокол**)** всегда один и тот же, так что программа может работать по сети даже в том случае, если клиент и сервер – совершенно различные компьютеры. В несетевом случае один и тот же компьютер является и клиентом, и сервером.

1.2 Немного OpenGL кода

Поскольку при помощи библиотеки **OpenGL** можно делать так много вещей, программа, использующая ее, может быть весьма сложна. Однако базовая структура полезной программы может быть достаточно простой: ее задачами являются инициализация нескольких переменных или переключателей, контролирующих, как **OpenGL** осуществляет визуализацию изображения и, далее, указание объектов для отображения.

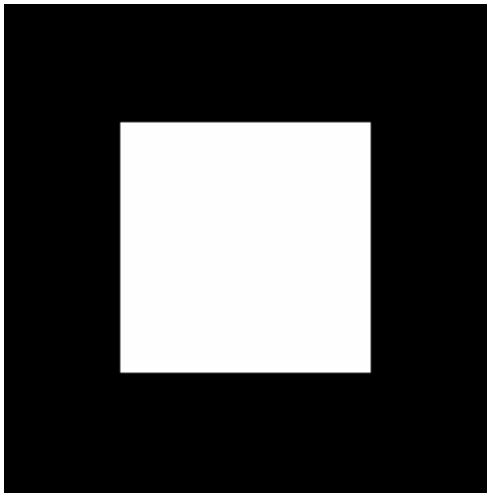
До того, как будет приведен небольшой пример, определимся с некоторыми терминами. Визуализация **(rendering)** – это процесс, с помощью которого компьютер создает изображения из моделей. Эти модели или объекты строятся из геометрических примитивов – точек, линий и полигонов, которые в свою очередь, определяются своими вершинами **(vertices).**

Результирующее изображение состоит из пикселей отображенных на экране. Пиксель – это самый маленький видимый элемент, который монитор может поместить на свой экран.

Информация о пикселях **(**например, какого цвета они должны быть**)** организована в памяти в виде битовых поверхностей **(bitplanes).** Битовая плоскость – это область памяти, содержащая один бит информации на каждый пиксель экрана. Этот бит может определять, например, насколько красным должен быть конкретный пиксель. Сами битовые плоскости организованы в буфер кадра, который содержит всю информацию необходимую монитору, чтобы контролировать цвет и интенсивность всех пикселей на экране.

Теперь посмотрим, на что может быть похожа **OpenGL –** программа. Пример **1-1** отображает белый квадрат на черном фоне, показанный на рисунке **1.**

Рисунок **1.1.** Белый квадрат на черном фоне



Пример **1-1.** Фрагмент **OpenGL –** кода

```
#include <все_что_необходимо.h>

main()
{
ИнициализироватьОкно();

glClearColor(0.0,0.0,0.0);
glClear(GL_COLOR_BUFFER_BIT);
glColor3f(1.0,1.0,1.0);
glOrtho(0.0,1.0,0.0,1.0,-1.0,1.0);
glBegin(GL_POLYGON);
glVertex3f(0.25,0.25,0.0);
glVertex3f(0.75,0.25,0.0);
glVertex3f(0.75,0.75,0.0);
glVertex3f(0.25,0.75,0.0);
glEnd();
glFlush();

ОбновитьОкно();
}
```

Первая строка функции **main()** инициализирует окно на экране. **ИнициализироватьОкно()** обозначает то место в тексте, куда надо поместить функции создания окна, зависимые от оконной системы, в которой вы работаете и не являющиеся командами **OpenGL.** Две следующих строки являются командами **OpenGL,** окрашивающими окно в черный цвет: **glClearColor()** задает цвет, в который окно будет окрашиваться при его очистке, **glClear()** как раз и очищает окно. После того, как «очищающий цвет» задан, окно всегда будет окрашиваться именно в него при вызовах **glClear().** Этот цвет может быть изменен повторным вызовом **glClearColor().** Похожим же образом **glColor3f()** задает цвет, который будет использоваться для рисования объектов – в данном случае белый. Начиная с этого момента, все объекты будут рисоваться белыми до тех пор, пока не последует замена этого цвета на другой при помощи одной из команд **OpenGL.**

Следующая **OpenGL –** команда**,** использованная в программе**, glOrtho()**, определяет координатную систему**,** на которую полагается **OpenGL** при отрисовке финального изображения и проецировании изображения на экран**.** Вызовы**,** заключенные в команды **glBegin()** и **glEnd()**, определяют объект**,** который следует нарисовать **–** в данном случае полигон с **4** вершинами**. «**Углы**»** полигона определяются командами **glVertex3f()**. Как можно догадаться по значениям аргументов**,** которые являются координатами **(x, y, z),** полигон представляет собой квадрат в плоскости **z=0.**

Наконец**,** вызов **glFlush()** позволяет быть уверенным в том**,** что команды **OpenGL** действительно выполнились**,** а не были сохранены в буфере в ожидании дальнейших команд**. ОбновитьОкно() –** это опять**-**таки метка для функций**,** зависящих от оконной системы**.**

На самом деле этот фрагмент кода не слишком хорошо структурирован**.** Могут возникнуть вопросы**: «**Что если мы попытаемся изменить размер окна или переместить его**?» «**Необходимо ли устанавливать координатную систему каждый раз при отрисовке квадрата**?»** Позже **ИнициализироватьОкно()** и **ОбновитьОкно()** будут заменены на реально работающие вызовы**,** которые**,** однако**,** требуют реструктуризации кода**,** что сделает его более эффективным**.**

1.3 Синтаксис команд OpenGL

Как можно было заметить из простой программы**,** приведенной в предыдущем разделе**,** в названии команд **OpenGL** используется префикс **gl** и заглавные буквы для отделения слов**,** составляющих название команды **(**вспомним**,** например**, glClearColor()).** Подобным же образом **OpenGL** определяет константы**,** начинающиеся с **GL_** со всеми заглавными буквами и символом подчеркивания для отделения слов **(**например**, GL_COLOR_BUFFER_BIT).**

Кроме того**,** можно было обратить внимание на казалось бы лишние буквы и цифры в названии некоторых команд **(**например**, 3f** в **glColor3f()** и **glVertex3f()).** Действительно**,** слова **Color (**цвет**)** в названии **glColor3f()** достаточно**,** чтобы определить команду меняющую текущий цвет**.** Однако в библиотеке определено несколько версий такой команды**,** принимающих данные в различных форматах**.** Более конкретно**, 3** в названии команды **glColor3f()** означает**,** что она получает три аргумента**,** существует и другая версия**,** получающая четыре аргумента**.** Буква **f** означает**,** что аргументы имеют формат числа с плавающей точкой**.** Подобное соглашение об именовании команд позволяет программисту использовать желаемый**,** более удобный для него формат аргументов**.**

Некоторые команды воспринимают до восьми различных типов данных**.** Буквы**,** используемые в качестве суффиксов для указания конкретного типа данных в реализации **OpenGL** для **ISO -** стандарта языка **C** приведены в таблице **1-1.** Отдельные реализации **OpenGL (**например**,** для языка **C++** или **Ada)** могут не следовать описанной схеме точно**.**

Таблица **1-1.** Суффиксы команд и соответствующие им типы аргументов

Суффиксы	Тип данных	Типично соответствующий тип языка C	Тип, определенный в OpenGL
b	целое **8** бит	**signed char**	**GLbyte**
s	целое **16** бит	**short**	**GLshort**
i	целое **32** бита	**int** или **long**	**GLint, GLsizei**
f	число с плавающей точкой **32** бита	**float**	**GLfloat, GLclampf**
d	число с плавающей точкой **64** бита	**double**	**GLdouble, GLclampd**
ud	беззнаковое целое **8** бит	**unsigned char**	**GLubyte, GLboolean**

us	беззнаковое целое **16** бит	**unsigned short**	**GLushort**
ui	беззнаковое целое **32** бита	**unsigned int** или **unsigned long**	**GLuint, GLenum, GLbitfield**

Таким образом, следующие две команды

```
glVertex2i(1,3);
glVertex2f(1.0,3.0);
```

эквивалентны за тем исключением, что первая принимает координаты вершины в виде **32-**разрядного целого, а вторая **–** в виде числа с плавающей точкой одинарной точности.

Замечание: Производители реализаций OpenGL имеют право выбирать, какие типы данных языка C использовать, для представления типов OpenGL. Если в тексте программы использовать типы, определенные OpenGL, вместо явного указания типов языка C, можно избежать проблем при переносе приложения на другую реализацию OpenGL.

Некоторые команды последней буквой в своем имени могут иметь **v**, это означает, что команда принимает указатель на вектор **(**или массив**)** величин, а не несколько отдельных аргументов. Многие команды имеют и векторную, и не векторную версии, но среди тех, которые не попадают в это число, одни работают только с индивидуальными аргументами, в то время как другие в обязательном порядке требуют указатель на вектор величин в качестве части или всех своих аргументов. Следующий фрагмент показывает пример использования векторной и не векторной версии команды установки текущего цвета.

```
glColor3f(1.0,0.0,0.0);
GLfloat color_array[]={1.0,0.0,0.0};
glColor3fv(color_array);
```

Кроме всего прочего, **OpenGL** определяет тип **GLvoid**, который чаще всего применяется в векторных версиях команд.
Далее в этом пособии будем мы ссылаться на команды по их базовому имени и звездочке в конце **(**например, **glColor*())**, что означает, что приводимая информация относится ко всем версиям определенной команды. Если информация специфична только для подмножества версий одной команды, это будет отмечено при помощи добавления к вышеописанной записи части суффикса **(**например, **glVertex*v()** означает все векторные версии команды установки вершины**)**.

1.4 OpenGL как машина состояния (state machine)

OpenGL – это машина состояния. Вы задаете различные переменные состояния, и они остаются в действии, сохраняя свое состояние, до тех пор, пока вы же их не измените. Как вы уже видели, текущий цвет **–** это переменная состояния. Вы можете установить цвет в красный, синий, белый и так далее, и после этого все объекты будут рисоваться этим цветом до тех пор, пока вы его не измените на что–либо другое. Текущий цвет **–** это только одна из многих переменных состояния, имеющихся в **OpenGL**. Другие управляют такими аспектами, как текущая видовая и проекционная трансформации, шаблоны для линий и полигонов, режимы отображения полигонов, соглашения об упаковке пикселей, позиции и характеристики источников света, параметры материалов для объектов и многое другое. Многие переменные относятся к возможностям **OpenGL**, которые можно включать или выключать командами **glEnable()** или **glDisable()**.

Каждая переменная состояния имеет свое значение по умолчанию, и в любой момент вы можете опросить систему на предмет ее текущего значения. Обычно, чтобы это

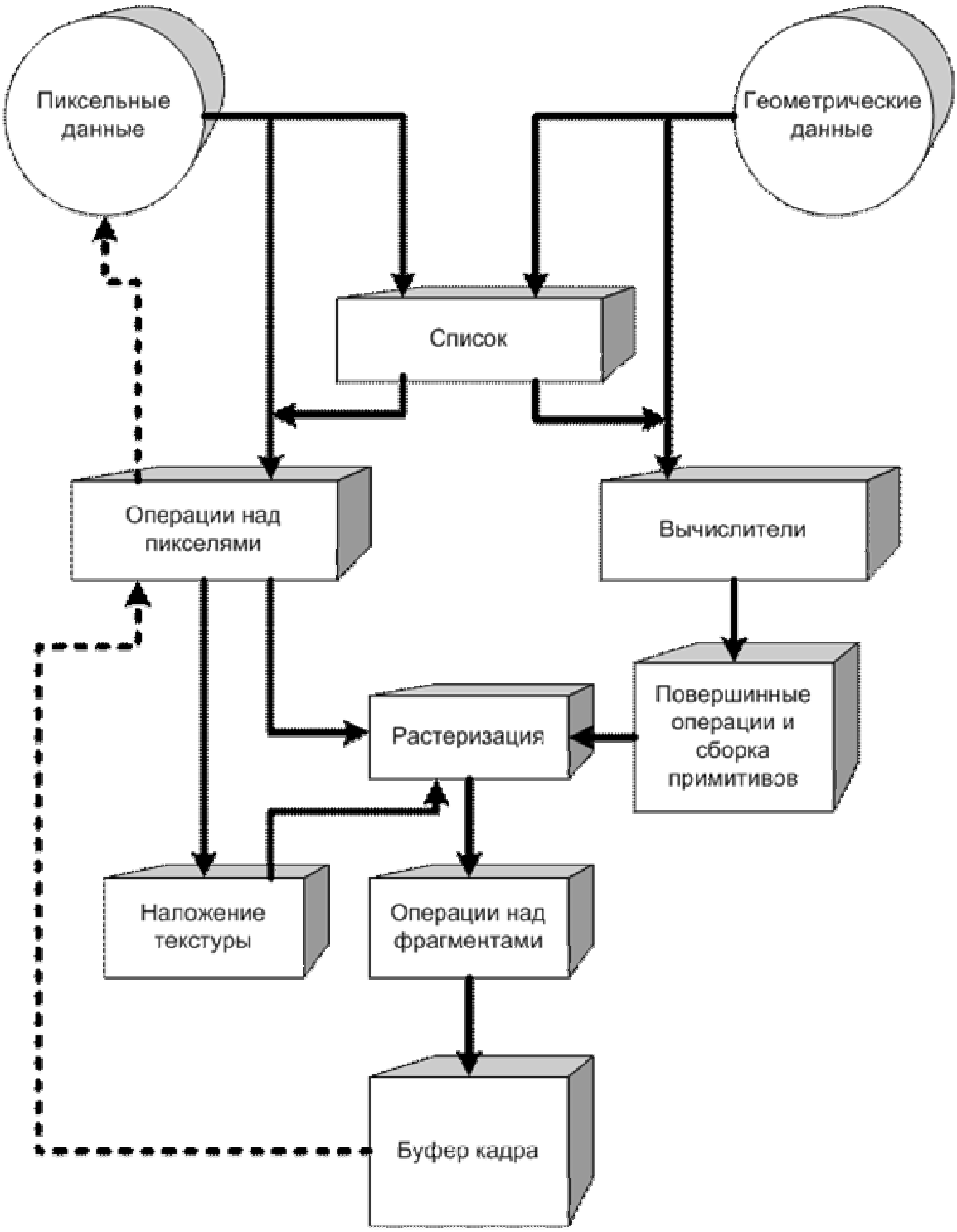
сделать, используется одна из следующих **6** команд: **glGetBooleanv()**, **glGetDoublev()**, **glGetFloatv()**, **glGetIntegerv()**, **glGetPointerv()**, **glIsEnabled()**. Выбор конкретной команды зависит от того, в каком формате вы ожидаете ответ на запрос. Для некоторых переменных состояния имеются более специфические команды опроса (такие как, **glGetLight*()**, **glGetError()** или **glGetPolygonStipple()**). Кроме того, вы можете сохранять наборы значений переменных состояния в стеке атрибутов командами **glPushAttrib()** или **glPushClientAttrib()**, временно изменять их значения и позже восстанавливать из стека командами **glPopAttrib()** или **glPopClientAttrib()**. Для временного изменения переменных состояния и восстановления их исходных значений следует пользоваться именно этими командами, так как они реализованы более эффективно по сравнению с отдельными командами запросов.

1.5 Конвейер визуализации OpenGL

Большинство реализаций **OpenGL** имеют сходный порядок операций или этапов обработки, называемый конвейером визуализации **OpenGL (OpenGL rendering pipeline)**. Этот порядок показан на рисунке **1-2** и, хотя он не является жестким правилом для всех реализаций, тем не менее, дает представление о том, что делает **OpenGL**.

Диаграмма демонстрирует конвейер (в духе сборочных линий Генри Форда), который используется **OpenGL** для обработки данных. Геометрические данные (вершины, линии и полигоны) проходят путь, включающий оценку и повершинные операции, в то время как пиксельные данные (пиксели, изображения и битовые карты) в части общего процесса обрабатываются иначе. Оба типа данных проходят через одинаковые финальные шаги (растеризация и операции над фрагментами) до того, как результирующие пиксельные данные записываются в буфер кадра.

Рисунок **1.2.** Порядок операций



Теперь более детально рассмотрим ключевые этапы конвейера визуализации **OpenGL.**

1.5.1 Списки

Все данные**,** геометрические или пиксельные могут быть сохранены в списках **(display lists)** для текущего или последующего использования**. (**Альтернативой занесению данных в списки является их немедленная обработка**,** известная как непосредственный режим**.)** Когда исполняется список**,** сохраненные в нем данные обрабатываются также**,** как если бы они поступали в непосредственном режиме**.**

1.5.2 Вычислители

Все геометрические примитивы описываются своими вершинами. Параметрические кривые и поверхности могут быть изначально описаны с помощью контрольных точек и полиномиальных функций, называемых базисными функциями. Вычислители предоставляют метод для определения реальных вершин, представляющих поверхность, по ее контрольным точкам. Этот метод – полиномиальная аппроксимация, он позволяет получить нормаль поверхности, координаты текстуры, цвета и значения координат в пространстве.

1.5.3 Повершинные операции

Для геометрических данных следующим этапом является выполнение повершинных операций **(per-vertex operations)**. В течение этого этапа вершины преобразуются в примитивы. Некоторые типы вершинных данных трансформируются матрицами чисел с плавающей точкой размерности **4**x**4**. Пространственные координаты проецируются с позиции в **3D** мире в позицию на вашем экране.

Если активизированы дополнительные возможности библиотеки **OpenGL**, то этот этап становится еще более сложным. Если используется текстурирование, координаты текстуры могут быть сгенерированы и изменены на этом шаге. Если используется освещение, вычисления, связанные с ним, производятся с использованием трансформированных вершин, нормалей поверхностей, позиций источников света, свойств материала, а также другой информации, позволяющей вычислить цветовую величину.

1.5.4 Сборка примитивов

Отсечение – большая часть сборки примитивов – это уничтожение частей геометрии, выпадающих за полупространство, определенное плоскостью. Отсечение точек просто отвергает или не отвергает вершину; отсечение линий или полигонов может добавить дополнительные вершины в зависимости от ситуации (того, как именно линия или полигон отсекаются).

В любом случае, после этого процесса производится перспективное разделение, которое заставляет более дальние объекты выглядеть меньше, чем ближние. Затем выполняются операции с портом просмотра **(viewport)** и глубиной (координатой **z**). Если включено распознавание лицевых граней, и примитив является полигоном, то на этом шаге грань может быть отвергнута в зависимости от теста на лицевые грани. В зависимости от режима рисования полигонов, они могут быть нарисованы в виде точек или линий.

Результатом этого этапа являются завершенные примитивы, то есть трансформированные и отсеченные вершины со связанными цветом, глубиной и, иногда, координатами текстуры.

1.5.5 Операции над пикселями

В то время как геометрические данные движутся по конвейеру своим путем, пиксельные данные двигаются иным маршрутом. Первым делом массивы данных из системной памяти распаковываются, то есть преобразуются из какого-либо формата, в формат с необходимым числом компонент. Далее данные масштабируются, базируются и обрабатываются пиксельными картами. Результат сжимается и записывается в текстурную память или отправляется на этап растеризации.

Если пиксельные данные читаются из буфера кадра, над ними выполняются пиксельные операции **(pixel-transfer operations)**. Затем результаты упаковываются в соответствующий формат и возвращаются в массив в системной памяти.

Существуют специальные операции копирования пикселей **(pixel copy operations)** для копирования данных из одной части буфера кадра в другие или из буфера кадра в текстурную память.

1.5.6 Наложение текстуры

Приложения **OpenGL** могут накладывать текстурные изображения на геометрические объекты, чтобы заставить их выглядеть более реалистично. Если используется несколько изображений текстур, разумно поместить их в объекты текстуры, чтобы можно было легко переключаться между ними.

Некоторые реализации **OpenGL** могут иметь дополнительные ресурсы для ускорения операций с текстурами. Например, может существовать специализированная быстрая текстурная память. Если такая память присутствует, текстурным объектам могут быть назначены приоритеты, чтобы управлять использованием этого ограниченного и весьма ценного ресурса.

1.5.7 Растеризация

Растеризация – это процесс преобразования геометрических и пиксельных данных во фрагменты. Каждый фрагмент соответствует пикселю в буфере кадра. Шаблоны линий и полигонов, толщина линии, размер точек, модель заливки, вычисления связанные с наложением для поддержки сглаживания принимаются в расчет при развертке двух вершин в линию или вычислении внутренних пикселей полигона. Каждый фрагмент имеет ассоциированные с ним значения цвета и глубины.

1.5.8 Операции над фрагментами

До того, как величины будут сохранены в буфере кадра, над ними производится серия операций, которые могут изменить или даже выбросить некоторые фрагменты. Все эти операции могут быть включены или выключены.

Первая операция, которая может быть произведена – это текстурирование, когда из текстурной памяти для каждого фрагмента генерируется и накладывается на него тексел **(**элемент текстуры**)**. Также могут производиться **(**в порядке выполнения**)** вычисления тумана, тест отреза **(scissor test)**, альфа-тест, тест трафарета **(stencil test)** и тест буфера глубины **(**для удаления невидимых поверхностей**)**. Если фрагмент не проходит один из включенных тестов, это может закончить его путь по конвейеру. Далее могут быть произведены наложение, смешивание цветов **(dithering)**, логические операции и маскирование с помощью битовой маски. Наконец, фрагмент заносится в соответствующий буфер, где становится пикселем.

1.6 Библиотеки, связанные с OpenGL

OpenGL предоставляет мощный, но примитивный набор команд и все высокоуровневое рисование должно производиться в терминах этих команд. Кроме того, программы **OpenGL** должны использовать нижележащие механизмы оконной системы. Существует несколько библиотек, которые могут облегчить программирование. Среди них имеются следующие:

- **OpenGL Utility Library (GLU)** содержит несколько функций, которые используют низкоуровневые команды **OpenGL** для выполнения таких операций, как установка специфических матриц видовой ориентации и проекций, триангуляции полигонов и визуализации поверхностей. Эта библиотека предоставляется как часть любой реализации **OpenGL.**
- Для каждой оконной системы существует библиотека, расширяющая возможности этой оконной системы для поддержки **OpenGL.** Для машин, где используется системы **X**

Window расширения **OpenGL (GLX)** предоставляются в виде добавочных функций с префиксом **glX.** Для **Microsoft Windows 95/98/Me/NT/200/XP** функции **WGL** предоставляют интерфейс от **Windows** к **OpenGL.** Почти все они имеют префикс **wgl.** Для **IBM OS/2** функции менеджера презентаций имеют префикс **pgl.** Для **Apple** существует интерфейс **AGL,** чьи функции имеют соответствующий префикс **(agl).**

- **OpenGL Utility Toolkit (GLUT) –** это независимая от оконной системы библиотека, написанная Марком Килгардом, чтобы скрыть сложности **API** различных оконных систем. Все функции библиотеки имеют префикс **glut.** Исходный код библиотеки **GLUT** для систем **Microsoft Windows 95/98/NT/Me/XP** и **X Window** может быть получен по Интернет – адресу **http://reality.sgi.com/opengl/glut3/glut3.html.** На данной странице помимо самого кода содержится информация о текущей версии **GLUT.**
- **Fahrenheit Scene Graph (FSG) –** объектно-ориентированный набор, основанный на **OpenGL,** он предоставляет объекты и методы для создания интерактивных трехмерных графических приложений. **FSG,** написанный на **C++,** предоставляет заранее построенные объекты и встроенную модель событий для взаимодействия с пользователем, высокоуровневые компоненты приложений для создания и редактирования трехмерных сцен, а также возможность обмена данными в различных форматах. **FSG** распространяется совершенно отдельно от **OpenGL.**
- Используя **OpenGL,** компания **Silicon Graphics** создала вспомогательную библиотеку для упрощения написания программ – примеров **(OpenGL Auxiliary Library -- GLAUX).** Код этой библиотеки поставляется в составе **Microsoft Platform SDK** и может быть использован в пользовательских программах или в образовательных целях.

1.7 Заголовочные файлы

Для всех **OpenGL** приложений необходимо включать заголовочный файл **gl.h.** Также большинство приложений используют **GLU** и должны включать файл **glu.h.** Таким образом, практически каабой исходный файл приложения **OpenGL** начинается со следующих строк:

```
#include <GL/gl.h>
#include <GL/glu.h>
```

Для систем **Microsoft Windows** требуется включение файла **windows.h** до включения **gl.h** или **glu.h,** так как некоторые макросы, используемые в этих файлах, определены внутри **windows.h.**
Если вы хотите получить доступ к библиотеке поддержки **OpenGL** оконной системой, например **GLX, AGL, PGL** или **WGL,** должны быть включены дополнительные файлы. Например, для вызовов функций **GLX,** требуется добавить в код следующие строки:

```
#include <X11/Xlib.h>
#include <GL/glx.h>
```

Для **Microsoft Windows** доступ к функциям **WGL** можно получить включением строки:

```
#include <windows.h>
```

Если предполагается использовать **GLUT** для управления задачами, связанными с окнами, следует добавить ее заголовочный файл:

```
#include <GL/glut.h>
```

Замечание: glut.h гарантирует, что включены также gl.h и glu.h, так что нет необходимости включать все три файла. glut.h также гарантирует, что все специфичные для оконной системы макросы определены должным образом, до включения gl.h и glu.h. Для повышения переносимости GLUT-программ, включайте только glut.h и не включайте gl.h или glu.h.

Многие приложения **OpenGL** также используют стандартную библиотеку языка **C**, поэтому является частой практикой включать в исходный текст заголовочные файлы не связанные с графикой:

```
#include <stdlib.h>
#include <stdio.h>
```

1.8 Сборка проекта

Помимо включения в исходный текст директив компилятора для добавления заголовочных файлов необходимо также проследить за тем, что во время сборки проекта к нему будут добавлены нужные библиотеки импорта. Например, в операционных системах **Microsoft Windows** библиотека **OpenGL (**в любой ее реализации**)** представлена динамической библиотекой **opengl32.dll**, а **GLU –** файлом **glu32.dll.** Библиотека импорта, присоединенная к проекту вашего приложения, позволяет ему во время выполнения загружать нужные динамические библиотеки и вызывать их функции. Для названных динамических библиотек соответствующими библиотеками импорта являются **opengl32.lib** и **glu32.lib**, находящиеся, как правило, в одном из подкаталогов компилятора.

Если помимо средств операционной или оконной системы вы используете еще какие-либо библиотеки, то, возможно, придется добавлять в проект и другие библиотеки импорта. Иногда **(**как, например, в случае с **GLUT)** в самом заголовочном файле содержится директива компилятору включить в проект нужные библиотеки импорта, но так бывает не всегда. Сигналом к тому, что нужные ссылки на библиотеки отсутствуют, чаще всего, является то, что компилятор просто отказывается собирать проект.

Кроме того, нужные динамические библиотеки должны быть в зоне досягаемости вашего готового приложения. Как правило, это означает, что они должны находиться либо в одном каталоге с исполняемым файлом, либо **(**и это случается чаще**)** они должны быть помещены в системную директорию **Microsoft Windows (**конкретный путь зависит от конкретного компьютера, но в общем виде этот путь можно записать как **[**Каталог **Windows]\System).**

1.9 GLUT

Как вы уже знаете, **OpenGL** содержит набор команд, но разработана как независимая от оконной или операционной системы. Как следствие, в ней нет команд для открытия окон или чтения событий от клавиатуры или мыши. К несчастью нельзя создать полноценное приложение без того, чтобы, как минимум, открыть окно, а наиболее интересные приложения требуют взаимодействия с пользователем посредством устройств ввода или используют иные средства операционной или оконной системы. Во многих случаях завершенные программы представляют собой наиболее интересные примеры, поэтому мы будем использовать **GLUT** для упрощения открытия окон, захвата пользовательского ввода и так далее. Если у вас имеется реализация **OpenGL** и **GLUT** для вашей системы, все примеры будут работать без изменений **(**или почти без изменений**).**

Кроме того, в отличие от **OpenGL**, чьи команды ограничены рисованием примитивов, **GLUT** содержит функции для рисования более сложных трехмерных объектов, таких как сфера, куб, торус **(**бублик**)** и чайник. **(**Обратите внимание на то, что **GLU** также содержит функции для рисования этих объектов, а также цилиндров, конусов и многого другого.**)**

Библиотеки **GLUT** может быть недостаточно для построения полномасштабного **OpenGL** приложения **(**например, в силу отсутствия полного контроля над форматом пикселей или созданием контекста **OpenGL)**, но это хорошая начальная точка для изучения

OpenGL. Остальная часть данного раздела посвящена наиболее часто применяемым группам функций **GLUT.** Функции неуказанные здесь, рассмотрены в приложении **A** данного пособия.

1.9.1 Управление окном

```
void glutInit (int argc, char **argv);
```

glutInit() должна быть вызвана до любых других **GLUT –** функций, так как она инициализирует саму библиотеку **GLUT. glutInit()** также обрабатывает параметры командной строки, но сами параметры зависят от конкретной оконной системы. Для системы **X Window**, примерами могут быть **–iconic**, **-geometry** и **–display. (**Параметры, передаваемые **glutInit()**, должны быть теми же самыми, что и параметры, передаваемые в функцию **main()).**

```
void glutInitDisplayMode (unsigned int mode);
```

Указывает режим отображения **(**например, **RGBA** или индексный, одинарной или двойной буферизации**)** для окон, создаваемых вызовами **glutCreateWindow().** Вы также можете указывать, имеет ли окно ассоциированные с ним буфер или буферы глубины, трафарета и аккумуляции. Аргумент **mask –** это битовая комбинация, полученная при помощи операции **OR** и следующих констант**: GLUT_RGBA** или **GLUT_INDEX (**для указания цветового режима**), GLUT_SINGLE** или **GLUT_DOUBLE (**для указания режима буферизации**),** а также константы для включения различных буферов **GLUT_DEPTH, GLUT_STENCIL, GLUT_ACCUN.** Например, для окна с двойной буферизаций, **RGBA –** цветовым режимом и ассоциированными буферами глубины и трафарета, используйте **GLUT_DOUBLE | GLUT_RGBA | GLUT_DEPTH | GLUT_STENCIL.** Значение по умолчанию **– GLUT_RGBA | GLUT_ SINGLE (**окно с однократной буферизацией в **RGBA -** режиме**).**

```
void glutInitWindowSize (int width, int height);
void glutInitWindowPosition (int x, int y);
```

Запрашивают окно определенного размера и в определенном месте экрана соответственно. Аргументы **(x, y)** определяют, где будет находиться угол окна относительно всего экрана. **width** и **height** определяют размер окна **(**в пикселях**).** Начальные размеры и место размещения окна могут быть перекрыты последующими вызовами.

```
int glutCreateWindow (char *name);
```

Открывает окно с предварительно установленными характеристиками **(**режимом отображения, размером и так далее**).** Строка **name** может быть отображена в заголовке окна, но это зависит от конкретной оконной системы. Окно не отображается до того, как произведен вход в **glutMainLoop()**, поэтому до вызова этой функции нельзя рисовать что-либо в окно.
Возвращаемая целая величина представляет собой уникальный идентификатор окна. Этот идентификатор может быть использован для управления несколькими окнами в одном приложении **(**каждое со своим контекстом **OpenGL)** и рисования в них.

1.9.2 Функции управления событиями

После того, как окно создано, но до входа в главный цикл программы, вы должны зарегистрировать функции обратного вызова, используя следующие функции **GLUT.**

```
void glutDisplayFunc (void (*func)(void));
```

Позволяет указать функцию **(**аргументом **func),** которая будет вызываться каждый раз**,** когда содержимое окна требует перерисовки**.** Это может случиться**,** когда окно открывается**,** разворачивается**,** освобождается его часть**,** ранее перекрытая другим окном**,** или вызывается функция **glutPostRedisplay().**

```
void glutReshapeFunc (void (*func)(int width, int height));
```

Позволяет указать функцию**,** которая вызывается каждый раз при изменении размера окна или его позиции на экране**.** Аргумент **func –** это указатель на функцию**,** которая принимает два параметра**: width –** новая ширина окна и **height –** новая высота окна**.** Обычно **func** вызывает **glViewport()** для отсечения графического вывода по новым размерам окна**,** а также настраивает проекционную матрицу для сохранения пропорций спроецированного изображения в соответствии с новыми размерами порта просмотра**.** Если **glutReshapeFunc()** не вызывается или ей передается **NULL (**для отмены регистрации функции обратного вызова**),** вызывается функция изменения метрик по умолчанию**,** которая вызывает **glViewport (0,0,width,height).**

```
void glutKeyboardFunc (void (*func)(unsigned int key, int x, int y));
```

Задает функцию **func,** которая вызывается**,** когда нажимается клавиша**,** имеющая **ASCII-**код**.** Этот код передается функции обратного вызова в параметре **key.** В параметрах **x** и **y** передается позиция курсора мыши **(**относительно окна**)** в момент нажатия клавиши**.**

```
void glutMouseFunc (void (*func)(int button, int state, int width, int height));
```

Указывает функцию**,** которая вызывается при нажатии или отпускании кнопки мыши**.** Параметр **button** может иметь значения **GLUT_LEFT_BUTTON**, **GLUT_MIDDLE_BUTTON** или **GLUT_RIGHT_BUTTON.** Параметр **state** может иметь значения **GLUT_UP** или **GLUT_DOWN** в зависимости от того отпущена или нажата кнопка мыши**.** В параметрах **x** и **y** передаются координаты курсора мыши **(**относительно окна**)** в момент наступления события**.**

```
void glutMotionFunc (void (*func)(int x, int y));
```

Указывает функцию**,** которая будет вызываться при движении мыши внутри окна в то время**,** как на ней нажата одна или несколько клавиш**.** В параметрах **x** и **y** передаются координаты курсора мыши **(**относительно окна**)** в текущий момент**.**

```
void glutPostRedisplay (void);
```

Помечает**,** что текущее окно требует перерисовки**.** После этого при любой возможности будет вызвана функция перерисовки окна**,** зарегистрированная вызовом **glutDisplayFunc().**

1.9.3 Загрузка палитры

Если вы работаете в индексном режиме**,** то можете к своему удивлению обнаружить**,** что в **OpenGL** нет команд для загрузки цвета в цветовую таблицу**.** Дело в том**,** что процесс загрузки палитры целиком зависит от оконной системы**.** В **GLUT** существует обобщенная функция для загрузки одного цветового индекса с соответствующим **RGB** значением**.**

```
void glutSetColor (Glint index, GLfloat red, GLfloat green, GLfloat blue);
```

Загружает в палитру по индексу **index, RGB-**значение**,** определенное параметрами **red, green** и **blue.** Последние три параметра нормализуются до диапазона **[0.0, 1.0].**

1.9.4 Рисование трехмерных объектов

Многие программы примеры используют простые трехмерные объекты для иллюстрации различных методов и техник визуализации изображения**. GLUT** содержит несколько функций для рисования таких объектов**.** Все эти функции работают в непосредственном режиме**.** Каждая из них имеет два варианта**:** первый рисует объект в виде проволочного каркаса и не генерирует нормалей**,** второй рисует объект сплошным и генерирует нормали поверхности **(**для чайника помимо этого генерируются координаты текстуры**).** Если используется освещение**,** следует выбирать сплошную версию объекта**.** Все объекты рисуются с учетом текущих параметров**,** например**,** цвета и характеристик материала**.** Кроме того**,** все объекты рисуются центрированными относительно текущих модельных координат**.**

```
void glutWireSphere (GLdouble radius, GLint slices, GLint stacks);
void glutSolidSphere (GLdouble radius, GLint slices, GLint stacks);
```

Рисуют проволочную или сплошную сферу с радиусом **radius,** количеством частей **(**полигонов из которых состоит сфера**) slices –** вокруг оси **z** и **stacks –** вдоль оси **z.** Для того**,** чтобы понять**,** что означает вокруг оси **z** и вдоль нее**,** представьте себе**,** что вы смотрите в длинную трубу**.** В данном случае направление вашего обзора совпадает с осью **z** трубы**.** Она может быть мысленно разделена как вдоль **(**на длинные фрагменты**),** так и поперек **(**на кольца**).** После таких разбиений труба фактически состоит из множества мелких кусочков**.** В случае сферы количество разбиений поперек задается параметром **stacks,** а количество разбиений вдоль **–** параметром **slices.** Из этого следует**,** что чем больше разбиений**,** тем более гладкой выглядит сфера на экране**,** но тем больше вычислений требуется для ее рисования**.**

```
void glutWireCube (GLdouble size);
void glutSolidCube (GLdouble size);
```

Рисуют проволочный или сплошной куб с длиной ребра **size.**

```
void glutWireTorus (GLdouble innerRadius, GLdouble outerRadius, GLint nsides,
GLint rings);
void glutSolidTorus (GLdouble innerRadius, GLdouble outerRadius, GLint nsides,
GLint rings);
```

Рисуют проволочный или сплошной торус **(**бублик**)** с внешним радиусом **outerRadius** и внутренним радиусом **innerRadius.** Параметр **rings** задает желаемое число колец из которых будет состоять торус**,** параметр **nsides –** из скольких частей будет состоять каждое кольцо**.**

```
void glutWireCone (GLdouble radius, GLdouble height, GLint slices, GLint stacks);
void glutSolidCone (GLdouble radius, GLdouble height, GLint slices, GLint
stacks);
```

Рисуют проволочный или сплошной конус радиусом **radius,** высотой **height.** Значение параметров **slices** и **stacks** аналогично таким же параметрам для сферы**.**

```
void glutWireIcosahedron (void);
void glutSolidIcosahedron (void);
void glutWireOctahedron (void);
```

```
void glutSolidOctahedron (void);
void glutWireTetrahedron (void);
void glutSolidTetrahedron (void);
void glutWireDodecahedron (GLdouble radius);
void glutSolidDodecahedron (GLdouble radius);
```

Рисуют проволочные или сплошные икосаэдр, октаэдр, тетраэдр и додекаэдр соответственно (единственный параметр последний пары функций задает радиус додекаэдра).

```
void glutWireTeapot (GLdouble size);
void glutSolidTeapot (GLdouble size);
```

Рисуют проволочный или сплошной чайник размера **size.**

1.9.5 Управление фоновым процессом

Вы можете указать функцию, которая будет вызываться в том случае, если нет других сообщений, то есть во время простоя приложения. Это может быть полезно для выполнения анимации или другой фоновой обработки.

```
void glutIdleFunc (void (*func)(void));
```

Задает функцию, выполняемую в случае, если больше приложению делать нечего (отсутствуют сообщения). Выполнение этой функции обратного вызова можно отменить передачей **glutIdleFunc()** аргумента **NULL.**

1.9.6 Запуск программы

После того, как все настройки выполнены, программы **GLUT** входят в цикл обработки сообщений функцией **glutMainLoop().**

```
void glutMainLoop (void);
```

Вводит программу в цикл обработки сообщений. Функции обратного вызова будут выполняться в случае наступления соответствующих событий.

Пример **1-2** показывает, как с помощью **GLUT** можно заставить работать программу, показанную в примере **1-1.** Обратите внимание на реструктуризацию кода. Для увеличения эффективности операции, которые нужно выполнить только один раз (установка цвета фона и координатной системы), теперь помещены в функцию **init().** Операции по визуализации (и пересчету) сцены находятся в функции **display()**, которая зарегистрирована в качестве дисплейной функции обратного вызова.

Пример **1-2.** Простая **OpenGL –** программа с использованием **GLUT: hello.cpp**

```
#include <GL/glut.h>

void init(void)
{
//Выбрать фоновый (очищающий) цвет
glClearColor(0.0,0.0,0.0,0.0);

//Установить проекцию
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
```

```
glOrtho(0.0,1.0,0.0,1.0,-1.0,1.0);
}

void display(void)
{
//Очистить экран glClear(GL_COLOR_BUFFER_BIT);

//Нарисовать белый полигон (квадрат) с углами //в (0.25, 0.25, 0.0) и (0.75,
0.75, 0.0)
glColor3f(1.0,1.0,1.0);
glBegin(GL_POLYGON);
glVertex3f(0.25,0.25,0.0);
glVertex3f(0.75,0.25,0.0);
glVertex3f(0.75,0.75,0.0);
glVertex3f(0.25,0.75,0.0);
glEnd();

//Не ждем. Начинаем выполнять буферизованные
//команды OpenGL
glFlush();
}

//Установить начальные характеристики окна,
//открыть окно с заголовком «hello».
//Зарегистрировать дисплейную функцию обратного вызова
//Войти в главный цикл
int main(int argc, char **argv)
{
glutInit(&argc,argv);
glutInitDisplayMode(GLUT_SINGLE|GLUT_RGB);
glutInitWindowSize(250,250);
glutInitWindowPosition(100,100);
glutCreateWindow("hello");
init();
glutDisplayFunc(display);
glutMainLoop();

return 0;
}
```

1.9.7 Типы проектов

Поскольку данное пособие предназначено в основном для тех, кто программирует приложения для среды **Microsoft Windows**, требуется сделать несколько важных замечаний.

Существует два основным типа приложения для этой операционной системы: консольные и оконные.

Консольные приложения по своему строению практически не отличаются от программ **DOS** запущенных под **Windows** (однако являются **32**-разрядными приложениями защищенного режима). При их запуске система автоматически создает окно, в которое можно выводить текстовую информацию языковыми средствами **C** без привлечения функций **API** операционной системы. Тем не менее, само приложение может также создавать дополнительные окна, если разработчику это требуется. Именно так и происходит при создании консольных приложений с использованием **GLUT** и **OpenGL**. Система при запуске создает консоль, а **GLUT** своим методом **glutCreateWindow()**

создает еще одно окно. В этом случае весь графический вывод **OpenGL** направляется в окно, созданное **GLUT**, а текстовый вывод функциями стандартной библиотеки **C** (например, **printf("текстовая строка")** или **cout<<"текстовая строка")** будет осуществляться в консольное окно. Это может быть весьма удобно, если помимо графического вывода программа выполняет некоторые вычисления над полученным изображением (определение минимальных и максимальных значений компонент цвета, использование обратного режима **OpenGL** и так далее). Стартовой точкой выполнения консольной программы является функция **main()**, в которую системой передаются те самые параметры командной строки (если они есть), которые позже должны быть переданы в **glutInit()**.

В случаях, когда лишнее консольное окно приложению не нужно, лучше изначально создавать оконное приложение. При его запуске вся ответственность по созданию окон ложится на программиста и средства, которыми он пользуется, в нашем случае опять-таки на **GLUT**. И вот здесь может возникнуть потенциальная проблема, которая происходит из разницы между форматами передачи параметров командной строки в функцию **main()** и **WinMain()**. В первом случае параметры передаются в функцию в виде массива строк, тогда как во втором – в виде одной единственной строки, причем в этом случае количество параметров неизвестно. В свою очередь **glutInit()** ожидает параметры в формате **main()** с указанием их числа. Существует несколько вариантов решения этой проблемы. Если параметры вам нужны, следует создать (или позаимствовать) функцию разборки единой командной строки на составляющие с выяснением их количества (это довольно просто). Если же у вас нет нужды в параметрах командной строки можно передать в **glutInit()** подделку следующего вида:

```
char* argv="";
int argc=0;
glutInit(&argc,&argv);
```

Конечно, существуют и другие варианты.

1.10 Анимация

Одна из самых впечатляющих вещей, которую вы можете делать на компьютере – это рисование движущихся картинок. Являетесь ли вы инженером, пытающимся увидеть разрабатываемую деталь со всех сторон, пилотом, обучающимся летать на самолете с помощью симулятора или профессионалом в области создания компьютерных игр, вам должно быть абсолютно ясно, что анимация – это важная часть компьютерной графики.

В кинотеатре движение достигается за счет проектирования серии картинок на экран со скоростью **24** кадра в секунду. Каждый кадр выдвигается на позицию перед лампой, открывается шторка и кадр показывается на экране. Шторка моментально закрывается, пленка передвигается к следующему кадру, он отображается на экране и так далее. Таким образом, когда вы просматриваете **24** кадра за одну секунду, ваш мозг объединяет их в непрерывную плавную анимацию. (Разрывы между кадрами были хорошо заметны в старых фильмах Чарли Чаплина, так как в то время фильмы снимались и показывались со скоростью **16** кадров в секунду.) Современные мониторы в среднем перерисовывают изображение от **60** до **76** раз в секунду (хотя скорость может достигать и **120** раз в секунду). Очевидно, что **60** кадров в секунду выглядят более гладко, чем **30**, а **120** намного лучше, чем **60**. Однако, частота обновления выше **120** кадров в секунду может находиться за гранью человеческого восприятия (это зависит, в основном, от индивидуальных особенностей).

Ключевая причина, по которой кинопроектор способен создать эффект движения заключается в том, что каждый кадр уже готов на момент его показа. Предположим, что вы пытаетесь воспроизвести анимацию, состоящую из миллиона кадров, с помощью программы вроде следующей:

```
Открыть_окно();
for(i=0;i<1000000;i++)
{
Очистить_окно();
Нарисовать_кадр(i);
Подождать_пока_пройдет_одна_24я_часть_секунды();
}
```

Если вы сложите время, которое требуется вашей программе, чтобы нарисовать кадр на экране и время, необходимое для очистки экрана, то выяснится, что программа выдает все более и более неприятные результаты по мере того, как это суммарное время приближается к **1/24** части секунды. Предположим, что рисование занимает практически всю **1/24** часть секунды. Тогда элементы изображения рисуемые первыми будут видны на экране практически постоянно, а те, что рисуются в конце, будут похожи на приведение, так как сразу после их отображения программа начнет стирать все изображение с экрана и, таким образом, большую часть времени наблюдатель будет видеть на месте этих элементов пустой фон. Проблема в том, что приведенная программа не воспроизводит готовые кадры, а рисует каждый из них и наблюдатель это видит.

Большинство реализаций **OpenGL** предоставляет двойную буферизацию – аппаратный или программный механизм, который обеспечивает наличие двух полноценных цветовых буферов. Один отображается на экране, пока второй отрисовывается. Когда рисование кадра окончено, два буфера переключаются: тот, что только что был на экране, теперь используется для рисования и наоборот. Этот процесс похож на отображение проектором в кинотеатре пленки всего с двумя кадрами: пока один проецируется на экран, художник стирает и перерисовывает другой – невидимый. До тех пор, пока художник достаточно быстр, наблюдатель не заметит разницы между подобной техникой и той, где все кадры уже готовы и просто показываются друг за другом. С двойной буферизацией каждый кадр показывается только после того, как он полностью готов. Наблюдатель никогда не увидит частично нарисованное изображение.

Модифицированная для использования двойной буферизации версия предыдущей программы может выглядеть следующим образом:

```
Открыть_окно_в_режиме_двойной_буферизации();
for(i=0;i<1000000;i++)
{
Очистить_окно();
Нарисовать_кадр(i);
Переключить_буферы();
}
```

1.11 Частота обновления экрана

В некоторых реализациях **OpenGL**, функция **Переключить_буферы()** помимо простого переключения буферов, ждет, пока закончится текущий период обновления экрана, чтобы предыдущий кадр был полностью отображен и следующий был нарисован с самого начала. Предположим, что ваша система обновляет экран **60** раз в секунду. Это означает, что наивысшая скорость смены кадров, которую вы можете достичь – **60** кадров в секунду **(frames per second -- fps)** и если все ваши кадры будут очищаться и перерисовываться менее чем за **1/60** долю секунды, ваша анимация будет на этой скорости гладкой.

Часто случается, что кадр слишком сложен, чтобы нарисовать его за **1/60** секунды, таким образом, каждый кадр показывается на экране более чем один раз. Если, например, рисование кадра занимает **1/45** секунды, вы получаете скорость **30 fps** и ничем не занятое время в размере **1/30-1/45=1/90** секунды, то есть треть времени рисования кадра на каждый кадр.

Кроме того, частота обновления монитора постоянна, что может вызвать неожиданные последствия для быстродействия. Например, при частоте обновления монитора **60** раз в секунду, анимация может идти со скоростями **60 fps**, **30**, **fps**, **20 fps**, **15 fps**, **12 fps** и так далее. Это означает, что если пишете приложение и постепенно добавляете детали, сначала каждая деталь, которую вы добавляете, не оказывает воздействия на общее быстродействие – анимация по-прежнему идет со скоростью **60 fps.** Затем вы добавляете еще одну деталь, и, система уже не может рисовать кадр за **1/60** долю секунды, вследствие чего, неожиданно, быстродействие падает до **30 fps**, поскольку не нельзя переключить буферы при первой возможности. То же происходит, если время рисования, затрачиваемое на каждый кадр, перешагивает порог в **1/30** долю секунды – быстродействие снижается до **20 fps.**

Если сложность сцены близка к одному из волшебных чисел **(1/60** секунды, **1/30** секунды, **1/20** секунды и так далее**)**, то из-за некоторых вариаций некоторые кадры могут рисоваться чуть быстрее, а некоторые немного не укладываться в график. В этом случае скорость анимации непостоянна, что может вести к неприятному внешнему виду. В таких случаях, если не удается упростить медленные кадры, возможно, стоит добавить небольшую паузу в быстрые, чтобы сохранять скорость анимации постоянной. Если различные кадры вашей анимации драматически отличаются друг от друга по сложности, может потребоваться более изощренный подход.

1.12 Движение = Перерисовка + Переключение

Структура реальных программ анимации не слишком отличается от описания, приведенного в заголовке этого раздела. Обычно намного легче перерисовать весь буфер, чем выяснять какие его части нужно изменить. Это особенно верно для таких приложений как симуляторы полетов, где даже небольшое изменение в ориентации самолета ведет к масштабным изменениям вида за окнами кабины.

В большинстве анимационных программ объекты сцены просто перерисовываются с учетом различных трансформаций – изменяется точка обзора или положение наблюдателя, или машина немного проезжает по дороге, или объект немного поворачивается. Если требуются обширные вычисления не связанные с рисованием, то они могут ощутимо понизить быстродействие в целом и скорость анимации в частности. Имейте в виду однако, что часто пауза после вызова Переключить_буферы**()** может быть использована для таких операций.

В **OpenGL** нет функции Переключить_буферы**()**, так как данная возможность может не обеспечиваться отдельными аппаратными средствами и, в любом случае, она сильно зависит от оконной системы. Например, при использовании системы **X Window (**и работе без дополнительных библиотек**)** можно использовать следующую **GLX** функцию:

```
void glXSwapBuffers (Display *dpy, Window window);
```

В аналогичном случае для системы **Microsoft Windows** функция будет выглядеть следующим образом:

```
BOOL SwapBuffers (HDC hdc);
```

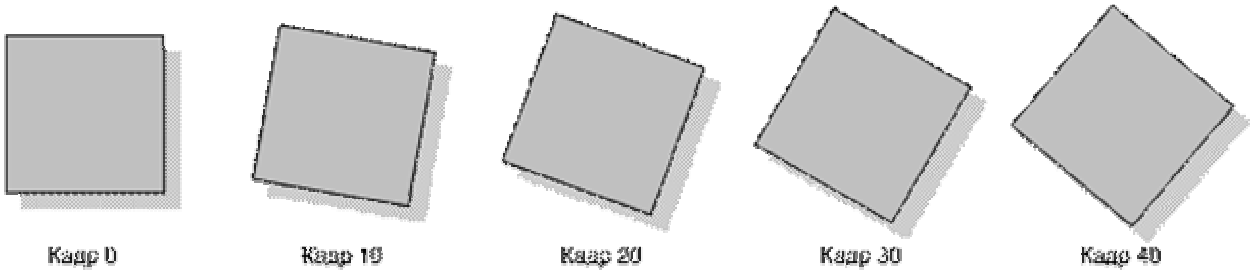
При использовании **GLUT** следует вызывать функцию:

```
void glutSwapBuffers (void);
```

Пример **1-3** иллюстрирует использование **glutSwapBuffers()** для рисования вращающегося квадрата, показанного на рисунке **1-3.** Этот пример также показывает, как использовать **GLUT** для захвата пользовательского ввода и включения**/**выключения

функции фоновой обработки. В данном примере левая и правая кнопки мыши соответственно включают и выключают вращение.

Рисунок **1.3.** Вращающийся квадрат в режиме двойной буферизации



Пример **1.3.** Программа, использующая двойную буферизацию: **double.cpp**

```
#include <GL/glut.h>

GLfloat spin=0.0;

void init(void)
{
glClearColor(0.0,0.0,0.0,0.0);
glShadeModel(GL_FLAT);
}

void display(void)
{
glClear(GL_COLOR_BUFFER_BIT);
glPushMatrix();
glRotatef(spin,0.0,0.0,1.0);
glColor3f(1.0,1.0,1.0);
glRectf(-25.0,-25.0,25.0,25.0);
glPopMatrix();
glutSwapBuffers();
}

void spinDisplay(void)
{
spin=spin+1.0;

if(spin>360.0) spin=spin-360.0;

glutPostRedisplay();
}

void reshape(int w, int h)
{
glViewport(0,0,(GLsizei) w, (GLsizei) h);
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
glOrtho(-50.0,50.0,-50.0,50.0,-1.0,1.0);
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
}

//При нажатии левой кнопки зарегистрировать
```

```
//функцию фоновой обработки (поворота)
//При нажатии правой – отменить регистрацию
void mouse(int button,int state,int x,int y)
{
switch(button)
{
case GLUT_LEFT_BUTTON:
if (state==GLUT_DOWN) glutIdleFunc(spinDisplay); break;

case GLUT_RIGHT_BUTTON: if (state==GLUT_DOWN) glutIdleFunc(NULL); break;
}
}

//Запросить режим двойной буферизации
//Зарегистрировать функции обработки мышиного ввода
int main(int argc, char **argv)
{
glutInit(&argc,argv);
glutInitDisplayMode(GLUT_DOUBLE|GLUT_RGB);
glutInitWindowSize(250,250);
glutInitWindowPosition(100,100);
glutCreateWindow("Двойная буферизация");
init();
glutDisplayFunc(display);
glutReshapeFunc(reshape);
glutMouseFunc(mouse);
glutMainLoop();

return 0;
}
```

Глава 2. Управление состоянием и рисование геометрических объектов

Хотя вы можете рисовать сложные и интересные картины, используя **OpenGL**, все они конструируются из небольшого набора примитивных графических элементов. Это не должно быть новостью – ведь Леонардо да Винчи работал лишь карандашами и кистями.

2.1 Необходимые приготовления

В данном разделе будет объяснено, как очистить окно до того как рисовать в нем, как устанавливать цвета объектов, которые будут нарисованы, и как заставить процесс рисования завершиться. Ни одна из этих операций не имеет прямого отношения к графическим объектам, но любая программа, рисующая объекты, вынуждена иметь с ними дело.

2.1.1 Очистка окна

Рисование на экране компьютера отличается от рисования на бумаге, так как бумага изначально является белой и все, что вам нужно сделать – это нарисовать изображение. В случае компьютера, память, хранящая изображение, обычно заполнена последней нарисованной картинкой, так что нужно предварительно очистить ее некоторым цветом фона, до начала того, как вы начнете рисовать новую сцену. Используемый цвет фона обычно зависит от типа приложения. Для текстового редактора вы можете использовать белый цвет (цвет бумаги) перед тем как выводить

текст. Если вы создаете вид из космического корабля, вы очищаете экран черным цветом перед тем, как рисовать на нем звезды, планеты и корабли пришельцев. В некоторых случаях вам вообще нет необходимости очищать экран, например, если вы изображаете некоторую сцену внутри комнаты, все графическое окно будет заполнено после того, как вы нарисуете все стены.К этому моменту вы можете задать вопрос: почему мы постоянно говорим об *очистке* окна – почему просто не нарисовать прямоугольник нужного цвета и такого размера, чтобы он перекрыл все окно? Во-первых, специализированная команда очистки экрана может быть намного более эффективно реализована, чем команды общего назначения. Кроме того, как вы увидите позже, **OpenGL** позволяет устанавливать координатную систему, положение наблюдателя и направление обзора, так что может быть достаточно сложно выяснить нужный размер и положение для очищающего прямоугольника. И, наконец, на многих компьютерах графическая аппаратура включает несколько буферов, а не только тот, в котором содержатся пиксели текущего изображения. Эти буферы должны также очищаться время от времени, и довольно удобно иметь одну единую команду, которая может очищать любую их комбинацию.Вы также должны знать, как цвета пикселей хранится в графической аппаратуре, называемой *битовыми плоскостями*. Существует два метода хранения. Либо непосредственно красное, зеленое, синее и альфа – значения **(RGBA)** пикселя сохраняются в битовых плоскостях, либо там сохраняется единственный индекс, ссылающийся на цветовую таблицу. **RGBA** – режим в наше время используется более часто. До поры до времени вы можете игнорировать присутствие альфа – значения и не пытаться понять, что оно означает.Как пример, следующие строки кода очищают окно в **RGBA** – режиме в белый цвет:

```
glClearColor(1.0,1.0,1.0,0.0);
glClear(GL_COLOR_BUFFER_BIT);
```

Первая строка белый цвет в качестве очищающего, а следующая – очищает все окно этим цветом. Единственный параметр **glClear()** указывает, какие буферы надо очистить. В нашем случае программа очищает только цветовой буфер, где находится текущее изображение, отображаемое на экране **(**или где находится предыдущее изображение, если вы используете двойную буферизацию**)**. Обычно, вы устанавливаете очищающий цвет единожды, где-то в начале программы, а затем чистите буферы так часто, как это необходимо. **OpenGL** сохраняет значение очищающего цвета в виде переменной состояния и не требует указывать его каждый раз при очистке буферов.Например, чтобы очистить не только буфер цвета, но и буфер глубины, следует использовать следующую последовательность команд:

```
glClearColor(1.0,1.0,1.0,0.0);
glClearDepth(1.0);
glClear(GL_COLOR_BUFFER_BIT|GL_DEPTH_BUFFER_BIT);
```

В этом случае вызов **glClearColor()** идентичен предыдущему, команда **glClearDepth()** указывает значение, которое будет присвоено каждому пикселю в буфере глубины, и параметр команды **glClear()** теперь состоит из двух констант, указывающих, какие буферы нужно очищать, объединенных логическим ИЛИ. Следующая информация по команде **glClear()** включает таблицу с именами буферов, которые можно очищать и соответствующие им константы.

```
void glClearColor (GLclampf red, GLclampf green, GLclampf blue, GLclampf alpha);
```

Устанавливает текущий цвет очистки буфера цвета в **RGBA** – режиме. Если необходимо, значения ***red***, ***green***, ***blue*** и ***alpha*** приводятся к диапазону **[0, 1]**. Цвет очистки по умолчанию черный, то есть **(0.0,0.0,0.0,0.0)**.

```
void glClear (GLbitfield mask);
```

Очищает указанные буферы указанными для них величинами очистки. Аргумент ***mask*** – это битовая комбинация констант, перечисленных в таблице **2-1**, полученная операцией ИЛИ.

Таблица **2-1.** Буферы, которые можно очищать

Название буфера	**Соответствующая константа**
Буфер цвета	**GL_COLOR_BUFFER_BIT**
Буфер глубины	**GL_DEPTH_BUFFER_BIT**
Буфер аккумуляции (аккумулятор)	**GL_ACCUM_BUFFER_BIT**
Буфер трафарета	**GL_STENCIL_BUFFER_BIT**

До того, как выполнять команду очистки нескольких буферов, вы должны установить очищающие значения для каждого из них, если хотите получить что-то кроме значений по умолчанию для **RGBA** – цвета, значения глубины, цвета аккумуляции и индекса трафарета. В дополнение к командам **glClearColor()** и **glClearDepth()** установки значений очистки для буфера цвета и буфера глубины, **glClearIndex()**, **glClearAccum()** и **glClearStencil()** указывают *цветовой индекс*, цвет аккумуляции и индекс трафарета, которые будут использоваться для очистки соответствующих буферов.

OpenGL позволяет указывать в одной команде несколько буферов потому, что очистка, как правило, процесс медленный, поскольку затрагивается каждый пиксель (а их могут быть миллионы), в то же время некоторая графическая аппаратура позволяет запараллелевать процессы очистки нескольких буферов. Аппаратура, которая не позволяет этого, выполняет процессы очистки последовательно. Разница между

```
glClear(GL_COLOR_BUFFER_BIT|GL_DEPTH_BUFFER_BIT);
```

и

```
glClear(GL_COLOR_BUFFER_BIT);
glClear(GL_DEPTH_BUFFER_BIT)
```

заключается в том, что хотя финальный эффект и одинаков, первый фрагмент на многих машинах может выполняться быстрее и уж в любом случае не будет работать медленнее.

2.1.1 Установка цвета

В **OpenGL** описание контура рисуемого объекта не зависит от описания его цвета. Когда бы ни рисовался геометрический объект, он рисуется с использованием текущей установленной цветовой схемы. Цветовая схема может быть простой («рисовать все красным цветом») или очень сложной («предположить, что объект сделан из голубого пластика, на нем два ярко-желтых отблеска здесь и здесь и по всей сцене распространяется слабый зеленый свет»). Обобщенно, программист сначала устанавливает цвет или цветовую схему и затем рисует объекты. Пока цвет или цветовая схема не изменены, все объекты рисуются с использованием установленного цвета или цветовой схемы. Этот метод позволяет **OpenGL** – программам достигать большего быстродействия, чем в том случае, если бы библиотека не отслеживала текущий цвет.

Например, следующий псевдокод

```
Установить_текущий_цвет(красный);
Нарисовать_объект(A);
```

```
Нарисовать_объект(B);
Установить_текущий_цвет(зеленый);
Установить_текущий_цвет(синий);
Нарисовать_объект(C);
```

рисует объекты **A** и **B** красным цветом, а **C –** синим. Команда на **4** строке, которая устанавливает текущий цвет в зеленый не имеет эффекта, так как сразу за ней **(**до каких-либо команд рисования**)** в **5** строке текущий цвет вновь меняется, на этот раз, на синий.

Чтобы установить цвет, используйте команду **glColor3f()**. Она принимает **3** параметра, каждый из которых является числом с плавающей точкой в диапазоне между **0.0** и **1.0**. Эти параметры, по порядку, указывают красный, зеленый и синий *компоненты* цвета. Вы можете думать об этих трех величинах как о задающих смесь из красного, зеленого и синего цветов: **0.0** означает, что мы совсем не используем данный компонент, **1.0** означает максимальное использование компонента. Таким образом, код

```
glColor3f(1.0,0.0,0.0);
```

делает текущий цвет ярко красным настолько, насколько его может изобразить ваша система, а зеленый и синий компоненты полностью отсутствуют. Если задать все нули, получится черный; если задать все единицы, получится белый, если установить три величины равными **0.5 –** получится серый **(**полпути между черным и белым**)**. Далее приведены **8** команд и цвета, которые они устанавливают.

```
glColor3f(0.0,0.0,0.0);      черный
glColor3f(1.0,0.0,0.0);      красный
glColor3f(0.0,1.0,0.0);      зеленый
glColor3f(1.0,1.0,0.0);      желтый
glColor3f(0.0,0.0,1.0);      синий
glColor3f(1.0,0.0,1.0);      фиолетовый
glColor3f(0.0,1.0,1.0);      голубой
glColor3f(1.0,1.0,1.0);      белый
```

2.1.2 Приказ завершить рисование

Как вы видели в разделе **«**Конвейер визуализации **OpenGL»** главы **1**, о большинстве современных графических систем можно думать как о линии сборки. Центральный процессор **(Central Processing Unit – CPU)** выполняет команду рисования. Возможно, иная аппаратура производит геометрические трансформации. Производится отсечение, заливка и**/**или текстурирование. Наконец, значения записываются на битовые плоскости для отображения. В **high-end** архитектурах каждая из этих операций производится отдельной частью аппаратуры, и каждая часть разработана таким образом, чтобы очень быстро выполнять свою специфическую задачу. В таких архитектурах **CPU** не нужно ждать пока завершится предыдущая команда рисования, чтобы начать выполнение следующей. Пока **CPU** посылает вершину на конвейер, аппаратура трансформации обрабатывает предыдущую, в то же время та вершина, что была до нее, проходит этап отсечения и так далее. В таких системах ожидание **CPU** завершения каждой команды, до начала выполнения следующей может привести к изрядной потере быстродействия.

Кроме того, приложение может выполняться более, чем на одном компьютере. Предположим, например, что программа запущена на какой-нибудь машине **(**называемой клиентом**)**, а вы просматриваете результат рисования на своей рабочей станции или терминале **(**называемом сервером**)**, который подключен к клиенту через сеть. В этом случае отправление каждой команды по сети по одной за раз может быть чудовищно неэффективным, поскольку с каждой сетевой пересылкой могут быть связаны серьезные затраты процессорных ресурсов, а также временные затраты. Обычно, клиент объединяет несколько команд в один сетевой пакет перед тем, как его

отправлять. К несчастью, сетевой код на клиентской стороне обычно не знает (и у него нет пути узнать), что графическая программа завершила рисование кадра или сцены. В худшем случае этот код будет вечно дожидаться дополнительных команд, которые нужны ему, чтобы заполнить пакет, и вы никогда не увидите финальное изображение.

По этой причине **OpenGL** предоставляет команду **glFlush()**, которая заставляет клиента отправить сетевой пакет даже в том случае, если он не полон. Если сети нет, и все команды действительно выполняются немедленно на сервере, **glFlush()** может не иметь никакого эффекта. Однако, если вы пишете программу и хотите, чтобы она работала верно и в отсутствие и при наличии сети, вам следует включить вызов **glFlush()** в конце каждого кадра или сцены. Обратите внимание на то, что **glFlush()** не ждет, пока закончится рисование – она просто заставляет начать выполнение рисования, таким образом, гарантируя, что все предыдущие команды будут *выполнены* за конечное время, даже если больше команд рисования не поступит.

- Существуют и другие ситуации, в которых **glFlush()** может быть полезной:
- Программные построители изображений, строящие их в системной памяти и не желающие постоянно обновлять экран.
- Реализации, которые амортизируют цену старта процесса за счет аккумулирования нескольких команд. Пример этого – описанная выше ситуация с сетью.

```
void glFlush(void);
```

Заставляет начать исполнение вызванных команд **OpenGL**, гарантируя, тем самым, что они выполнятся за некоторое конечное время.

Некоторые команды – например, команды переключения буферов в режиме двойной буферизации – автоматически выполняют **glFlush()** (или иную схожую операцию) до того, как сами могут выполниться.

Если вам недостаточно **glFlush()** попробуйте использовать **glFinish()**. Эта команда действует аналогично **glFlush()**, а затем ждет от аппаратуры или сети подтверждения о том, что рисование в буфере кадра окончено. **glFinish()** может понадобиться, если вы хотите синхронизировать задачи – например, убедиться, что ваша трехмерная картина на экране, чтобы потом с помощью **Display PostScript** нарисовать поверх нее заметки. Другой пример, если вы хотите убедиться в том, что картинка нарисована, чтобы начать принимать пользовательский ввод. После вызова **glFinish()** программа блокируется до тех пор, пока не поступит подтверждение от аппаратуры, что рисование окончено. Имейте ввиду, что чрезмерно частое использование **glFinish()** может сильно понизить быстродействие приложения особенно в том случае, если оно запускается по сети, так как **glFinish()** требует двустороннего сетевого обмена между клиентом и сервером. Если для ваших нужд достаточно **glFlush()**, используйте ее вместо **glFinish()**.

```
void glFinish(void);
```

Заставляет завершиться все ранее вызванные команды **OpenGL**. Выхода из этой команды не происходит, пока все ранее вызванные команды не отработают полностью.

2.2 Необходимые знания о координатных системах

Если вы только что создали окно, или передвинули имеющееся, или изменили его размер – во всех этих случаях оконная система посылает сообщение уведомления в программу. Если в программе используется **GLUT**, обработка уведомления автоматизирована – при получении сообщения будет вызвана любая функция,

зарегистрированная с помощью **glutReshapeFunc().** Обычно такая функция обратного вызова должна:

- Заново установить прямоугольный регион, в который будет осуществляться вывод.
- Определить координатную систему, которая будет использоваться при рисовании объектов.

Функция в следующем примере **2-1** определяет простую двумерную систему координат.

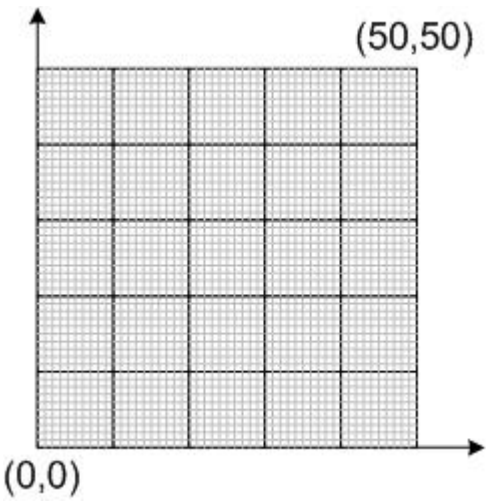
Пример **2-1.** Определение простой двумерной координатной системы

```
void reshape(int w, int h)
{
      glViewport(0,0,(GLsizei)w, (GLsizei)h);
      glMatrixMode(GL_PROJECTION);
      glLoadIdentity();
      gluOrtho2D(0,0,(GLdouble)w, (GLdouble)h);
}
```

Внутренние части **GLUT** передают этой функции два аргумента: длину и высоту **(**в пикселях**)** измененного окна. **glViewport()** подгоняет размеры прямоугольника, в который осуществляется графический вывод под размер всего окна. Три следующих вызова определяют координатную систему для рисования таким образом, чтобы левый нижний угол имел координаты **(0,0),** в правый верхний **(*w*,*h*) (**рисунок **2-1).**

Представьте себе лист бумаги – миллиметровки. В этом случае значения ***w*** и ***h*** в функции **reshape()** представляют собой количество квадратов по строкам и столбцам листа. Далее вам нужно определиться с осями координат. **gluOrtho2D()** помещает начало координат в самый левый нижний квадрат листа и указывает, что каждый квадрат представляет собой одну единицу. Теперь, когда вы будете рисовать точки, линии и полигоны, они будут появляться на бумаге в легко предсказуемых квадратах.

Рисунок **2-1.** Координатная система, определенная величинами ***w*=50** и ***h*=50**



2.3 Описание точек, линий и полигонов

Этот раздел объясняет, как описывать геометрические примитивы в **OpenGL.** Все геометрические примитивы описываются в терминах своих *вершин* – координат, которые определяют сами точки примитива, конечные точки линии или углы полигона. Следующая секция объясняет, как отображаются эти примитивы, и как вы можете контролировать их отображение.

2.3.1 Что такое точки, линии и полигоны?

Вероятно, у вас имеется некоторое представление о математическом смысле терминов точка, линия и полигон. Их смысл в **OpenGL** почти тот же самый, но не идентичный.

Одно из различий проистекает из ограничений на компьютерные расчеты. В любой реализации **OpenGL** числа с плавающей точкой имеют конечную точность и, следовательно, могут возникать ошибки, связанные с округлением. Как следствие, координаты точек, линий и полигонов в **OpenGL** страдают этим же недостатком.

Более важное различие проистекает из ограничений растровых дисплеев. На таких дисплеях наименьшим отображаемым элементом является пиксель и, хотя пиксель по размеру может быть меньше **1/100** дюйма, он все равно значительно больше, чем математические понятия бесконечно малого элемента (для точек) и бесконечно короткого (для линий). Когда **OpenGL** производит вычисления, она предполагает, что точки представлены в виде векторов с координатами в формате с плавающей точкой. Однако точка, как правило (но не всегда), отображается на дисплее в виде одного пикселя, и несколько точек, имеющих слегка различающиеся координаты, **OpenGL** может нарисовать на одном и том же пикселе.

2.3.1.1 Точки

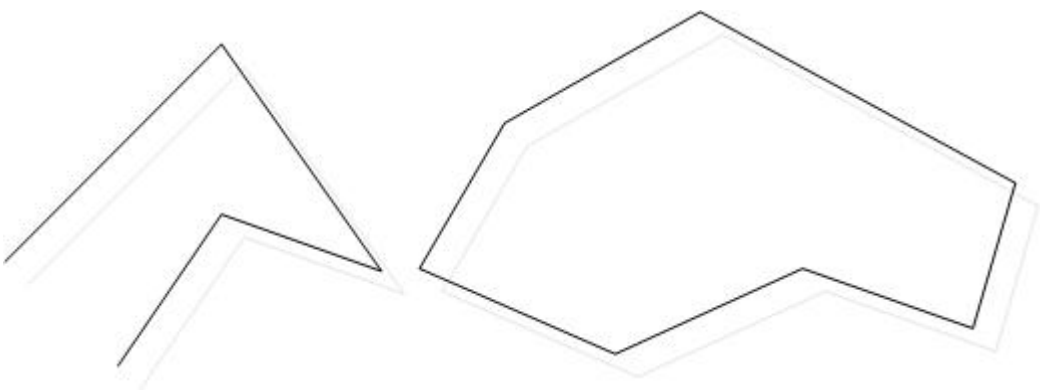
Точка определяется набором чисел с плавающей точкой, называемым вершиной. Все внутренние вычисления производятся в предположении, что координаты заданы в трехмерном пространстве. Для вершин, которые пользователь определил как двумерные (то есть задал только **x-** и **y-**координаты), **OpenGL** устанавливает **z-**координату равной **0.**

Дополнительно: **OpenGL** работает в *однородных координатах* трехмерной проекционной геометрии, таким образом, во внутренних вычислениях все вершины представлены четырьмя числами с плавающей точкой **(x,y,z,w).** Если **w** отлично от **0**, то эти координаты в Евклидовой геометрии соответствуют точке **(x/w,y/w,z/w).** Вы можете указывать **w-**координату в командах **OpenGL**, но это делается достаточно редко. Если **w-**координата не указана, она принимается равной **1.0.**

2.3.1.2 Линии

В **OpenGL** термин линия относится к сегменту прямой (отрезку), а не к математическому понятию прямой, которая предполагается бесконечной в обоих направлениях. Достаточно просто задавать серию соединенных отрезков или даже закрытую последовательность отрезков (рисунок **2-2).** В любом случае отрезки описываются в терминах вершин на их концах.

Рисунок **2-2.** Два примера соединенных сегментов линий (ломаная и замкнутая ломаная)

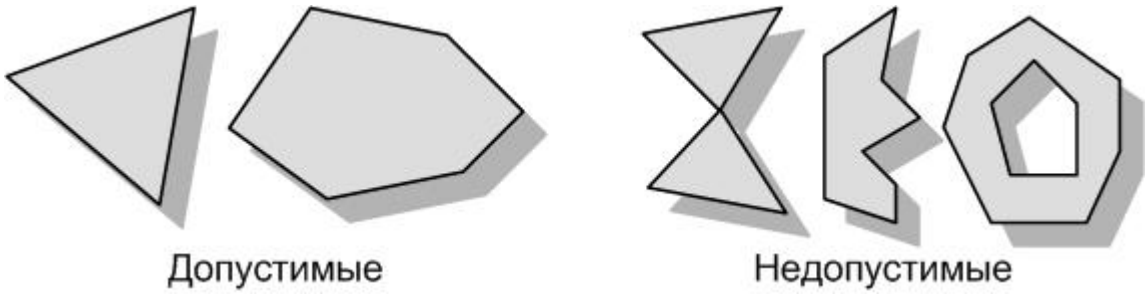


2.3.1.3 Полигоны

Полигон (или многоугольник) – это область, ограниченная одной замкнутой ломаной, при этом каждый отрезок ломаной описывается вершинами на его концах (вершинами полигона). Обычно полигоны рисуются заполненными, но вы можете также рисовать полигоны в виде их границ или в виде точек на концах отрезков, образующих полигон.

В общем случае полигоны могут быть достаточно сложны, поэтому **OpenGL** накладывает очень серьезные ограничения в отношении того, что считается полигональным примитивом. Во-первых, ребра полигонов в **OpenGL** не могут пересекаться (в математике полигон, удовлетворяющий этому условию, называется *простым*). Во-вторых, полигоны **OpenGL** должны быть выпуклыми. Полигон является выпуклым, если отрезок, соединяющий две точки полигона (точки могут быть и внутренними, и граничными) целиком лежит внутри полигона (то есть все его точки также принадлежат полигону). На рисунке **2-3** приведены примеры нескольких допустимых и недопустимых полигонов. **OpenGL** не накладывает ограничений на количество вершин полигона (и, как следствие, на количество отрезков, определяющих его границу). Заметьте, что полигоны с дырами не могут быть описаны, так как они не являются выпуклыми, и могут быть заданы при помощи одной замкнутой ломаной. Если вы поставите **OpenGL** перед вопросом о рисовании *невыпуклого* полигона, результат может быть не таким, какой вы ожидаете. Например, на большинстве систем в ответ будет заполнена только выпуклая оболочка полигона (на многих системах не произойдет и этого).

Рисунок **2-3.** Допустимые и недопустимые полигоны



Причина, по которой **OpenGL** накладывает подобные ограничения, заключается в том, что в аппаратуру намного проще заложить функции визуализации некоторого определенного класса полигонов. Простые полигоны визуализировать легко. Трудные случаи тяжело обнаружить быстро. Для увеличения быстродействия **OpenGL** «скрещивает пальцы» и предполагает, что все полигоны простые.

Многие реальные поверхности состоят их непростых полигонов, невыпуклых полигонов или полигонов с дырами. Поскольку каждый из таких полигонов может быть представлен в виде набора простых выпуклых полигонов, в библиотеке **GLU** имеется набор методов для построения более сложных объектов. Эти методы принимают описание сложного полигона и выполняют *тесселяцию* (***teselation***), то есть разбиение сложного полигона на группы более простых **OpenGL** – полигонов, которые в последствие могут быть визуализированы и отображены.

Поскольку вершины в **OpenGL** всегда трехмерны, точки, формирующие границу каждого конкретного полигона, не обязательно лежат на одной плоскости пространства. (Конечно, во многих случаях они лежат на одной плоскости – например, если **z**-координаты всех точек равны **0** или если полигон является *треугольником*.)

Если вершины полигона не лежат в одной плоскости, то после различных пространственных трансформаций, изменения точки наблюдения и проецирования на экран дисплея может оказаться, что эти точки уже не определяют простой выпуклый полигон. Представим себе четырехугольник, в котором точки немного отклоняются от

единой плоскости, и посмотрим на него со стороны ребра. В таких условиях может оказаться, что полигон не является простым и, как следствие, может быть не визуализирован верно (рисунок **2-4**). Ситуация не является такой уж необычной в случае если вы аппроксимируете изогнутые поверхности с помощью четырехугольников. Однако вы всегда можете избежать подобных проблем, применяя треугольники, так как три отдельно взятые точки всегда лежат в одной плоскости.

Рисунок **2-4.** Пример неплоского полигона, трансформированного в непростой

2.3.1.4 Квадраты

Поскольку квадраты часто встречаются в графических приложениях, **OpenGL** предоставляет примитив, рисующий заполненный квадрат и его команду **glRect*()**. Вы можете нарисовать квадрат как полигон, но в вашей конкретной реализации **OpenGL** **glRect*()** для квадратов может быть реализована более рационально.

```
void glRect{sifd} (TYPE x1, TYPE y1, TYPE x2, TYPE y2);
void glRect{sifd}v (TYPE *v1, TYPE *v2);
```

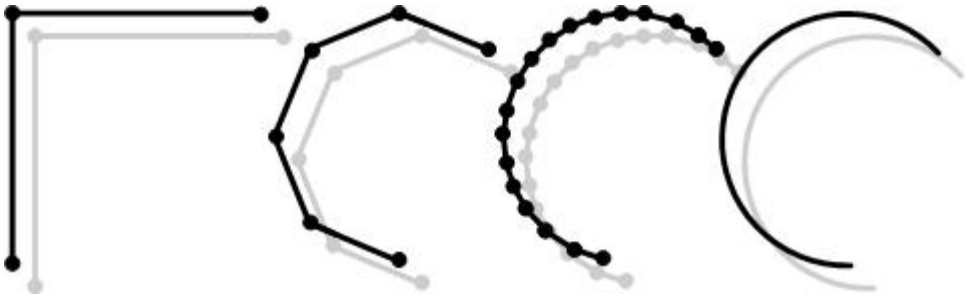
Рисует квадрат, определяемый угловыми точками ***(x1,y1)*** и ***(x2,y2)***. Квадрат лежит в плоскости ***z=0*** и его стороны параллельны осям абсцисс и ординат. Если используется векторная форма функции, углы задаются в виде двух указателей на массивы, каждый из которых содержит пару ***(x,y)***.

Заметьте, что хотя по умолчанию квадрат имеет определенную ориентацию в трехмерном пространстве, вы можете изменить это за счет применения вращения или других трансформаций.

2.3.1.5 Кривые и изогнутые поверхности

Любая плавная кривая или поверхность может быть аппроксимирована с любой степенью точности короткими отрезками или полигональными регионами. Таким образом, достаточное разделение кривых или поверхностей на прямые отрезки или плоские полигоны может привести к тому, что они по-прежнему будут выглядеть плавно изогнутыми (рисунок **2-5**). Если у вас есть сомнения относительно того, что это действительно работает, представьте такое разбиение, при котором каждый отрезок или полигон короче или меньше пикселя.

Рисунок **2-5.** Аппроксимация кривых



2.3.2 Указание вершин

При использовании **OpenGL** каждый геометрический объект однозначно описывается в виде упорядоченного набора вершин. Для указания вершины используется команда **glVertex*()**.

```
void glVertex{234}{sifd}(TYPE coords);
void glVertex{234}{sifd}v(const TYPE *coords);
```

Указывает одну вершину для использования в описании геометрического объекта. Для каждой вершины вы можете указывать от **2 (x,y)** до **4** координат **(x,y,z,w),** выбирая соответствующую версию команды. Если используется версия, где в явном виде не задаются **z** или **w,** то **z** принимается равным **0,** а **w –** равным **1.** Вызовы **glVertex*()** имеют силу только между вызовами команд **glBegin()** и **glEnd()**.

В примере **2-2** приводятся некоторые варианты использования **glVertex*()**.

Пример **2-2.** Варианты допустимого использования **glVertex*()**

```
glVertex2s(1,5);
glVertex3d(0.7,5.224,3.1415926535898);
glVertex4f(2.1,1.3,-2.0,2.0);

GLdouble dvect[3]={5.0,9.0,1992.0};
glVertex3dv(dvect);
```

Первый пример представляет собой определение вершины с координатами в трехмерном пространстве **(1,5,0). (**Вспомните, что если **z** не указано явно, оно принимается равным **0.)** Координаты во втором примере **(0.7,5.224,3.1415926535898) (**числа с плавающей точкой двойной точности**).** В третьем примере указывается вершина с трехмерными координатами **(1.05, 0.65, -1.0). (**В этом примере координаты указаны в однородном виде и для приведения их к более привычному следовало разделить первые три цифры на последнюю ***w*=2.0.)** В последнем примере ***dvect*** **–** это указатель на массив, содержащий три дробных числа двойной точности.

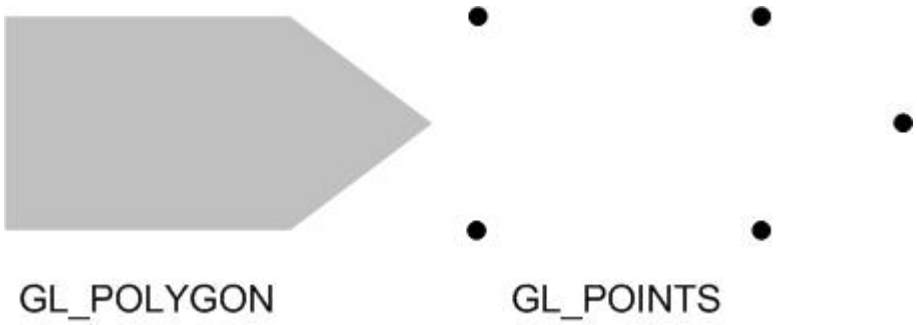
На некоторых машинах векторная форма **glVertex*()** работает более эффективно, поскольку в графическую подсистему требуется передать только один параметр. Отдельная аппаратура может быть способна отправлять целые серии команд в одном пакете. Если ваша машина одна из таких, вам может быть выгодно организовывать данные таким образом, чтобы координаты вершин были последовательно упакованы в памяти. В этом случае можно достичь некоторого увеличения быстродействия за счет использования операций с массивами вершин **OpenGL.**

2.3.3 Рисование геометрических примитивов

Теперь, когда вы знаете, как задавать вершины, вам все еще нужно узнать, как заставить **OpenGL** создать из этих вершин набор точек, линию или полигон. Для того

чтобы это сделать, вам следует обрамить каждый набор вершин вызовами команд **glBegin()** и **glEnd()**. Аргумент, передаваемый **glBegin()**, определяет, какие графические примитивы создаются на основе вершин. Пример **2-3** задает вершины для полигона на рисунке **2-6.**

Рисунок **2-6.** Рисуем полигон или набор точек.



Пример **2-3** Конструирование полигона

```
glBegin(GL_POLYGON);
      glVertex2f(0.0,0.0);
      glVertex2f(0.0,3.0);
      glVertex2f(4.0,3.0);
      glVertex2f(6.0,1.5);
      glVertex2f(4.0,0.0);
glEnd();
```

Если бы вы указали **GL_POINTS** вместо **GL_POLYGON** в качестве примитивов, были бы построены пять точек, показанных на рисунке **2-6.** В таблице **2-2** перечислены возможные аргументы команды **glBegin()** и соответствующие им типы примитивов.

```
void glBegin(GLenum mode);
```

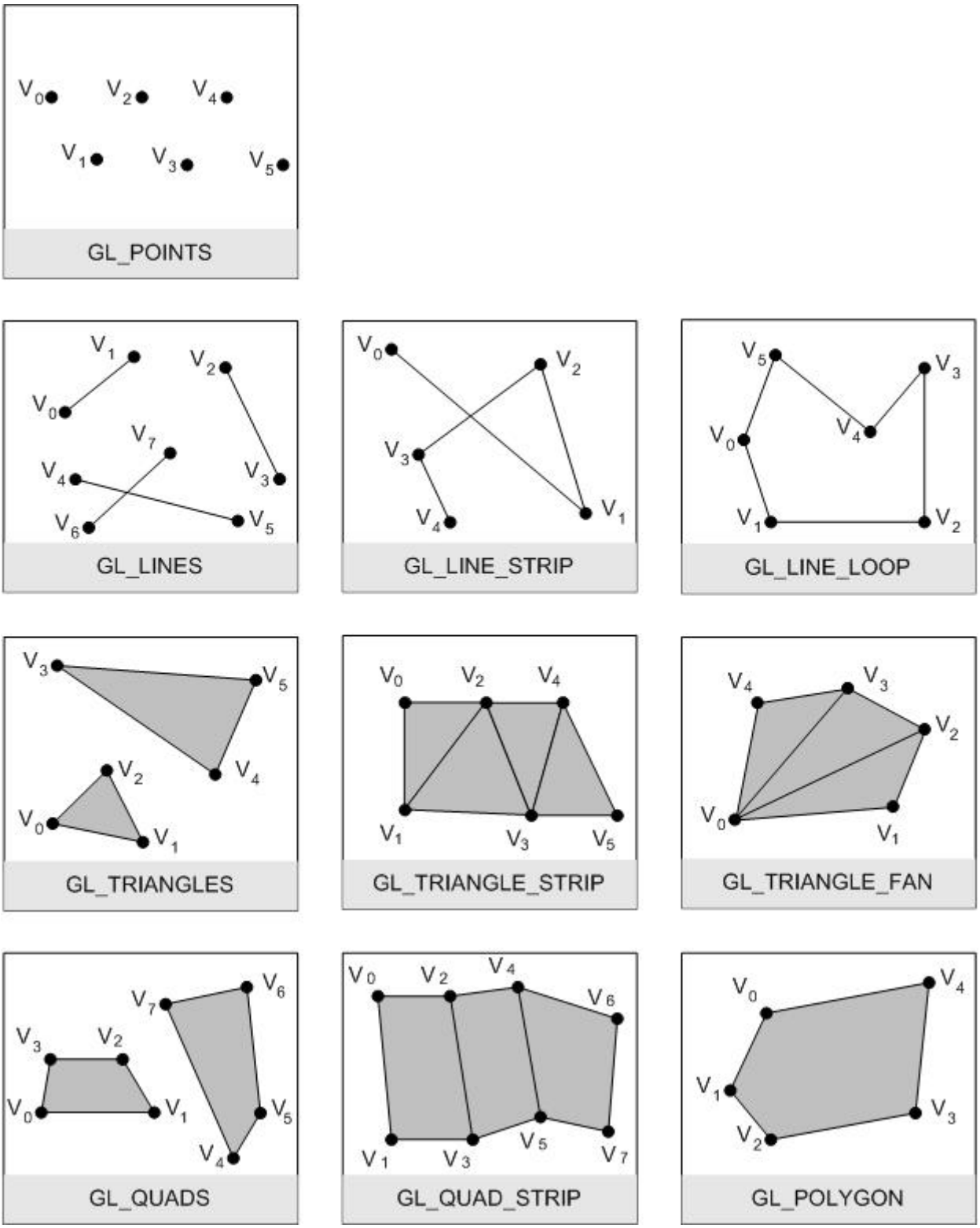
Отмечает начало блока вершинных данных, описывающего геометрические примитивы. Тип примитивов задается аргументом ***mode***, который может принимать значения, перечисленные в таблице **2-2.**

Таблица **2-2.** Значения аргумента ***mode*** **(**имена примитивов**)** для **glBegin()** и их смысл

Значение	**Соответствующие примитивы**
GL_POINTS	индивидуальные точки
GL_LINES	вершины попарно интерпретируются как самостоятельные отрезки
GL_LINE_STRIP	серия соединенных отрезков **(**ломаная**)**
GL_LINE_LOOP	аналогично предыдущему, но, кроме того, автоматически добавляется отрезок, соединяющий первую и последнюю вершины **(**замкнутая ломаная**)**
GL_TRIANGLES	каждая тройка вершин интерпретируется как треугольник
GL_TRIANGLE_STRIP	цепочка соединенных треугольников
GL_TRIANGLE_FAN	веер из соединенных треугольников
GL_QUADS	каждая четверка вершин интерпретируется как четырехугольный полигон
GL_QUAD_STRIP	цепочка соединенных четырехугольников
GL_POLYGON	граница простого выпуклого полигона

На рисунке **2-7** показаны примеры всех геометрических примитивов, перечисленных в таблице **2-2.** Обратите внимание, что в дополнение к точкам определено также несколько типов линий и полигонов. Безусловно, вы можете нарисовать один и тот же примитив различными методами. Как правило, выбираемый путь зависит от характера имеющихся вершинных данных.

Рисунок **2-7.** Типы геометрических примитивов



Когда вы будете читать следующее описание, предполагайте, что между парой **glBegin()** и **glEnd()** задано ***n*** вершин **(V[0], V[1], V[2], ..., V[*n*-1]).**

GL_POINTS	Рисует точку на месте каждой из ***n*** вершин
GL_LINES	Рисует серию несоединенных между собой отрезков. Рисуются отрезки между **V[0]** и **V[1]** , между **V[2]** и **V[3],** и так далее. Если ***n*** нечетно, то последний отрезок рисуется между **V[*n*-3]** и **V[*n*-2],** а вершина ***n*-1** игнорируется.
GL_LINE_STRIP	Рисует отрезок между **V[0]** и **V[1],** между **V[1]** и **V[2],** и так далее. Последний отрезок

	рисуется между **V[*n*-2]** и **V[*n*-1]**. Таким образом, создается ломаная из ***n-1*** отрезков. Если ***n* < 2** не отображается ничего. Не существует ограничений на вершины, описывающие ломаную (или замкнутую ломаную). Отрезки могут любым образом пересекаться.
GL_LINE_LOOP	То же, что и **GL_LINE_STRIP**, но, кроме того, рисуется отрезок между **V[*n*-1]** и **V[0]**, замыкающий ломаную.
GL_TRIANGLES	Рисует серию треугольников (полигонов с тремя сторонами) используя вершины **V[0]**, **V[1]** и **V[2]**, затем **V[3]**, **V[4]** и **V[5]**, и так далее. Если ***n*** не кратно **3**, последние **1** или **2** вершины игнорируются.
GL_TRIANGLE_STRIP	Рисует серию треугольников, используя вершины **V[0]**, **V[1]** и **V[2]**, затем **V[2]**, **V[1]** и **V[3]** (обратите внимание на порядок), затем **V[2]**, **V[3]**, **V[4]**, и так далее. Такой порядок гарантирует, что все треугольники будут иметь одинаковую ориентацию и, таким образом, соединенные треугольники могут сформировать часть поверхности. Сохранение ориентации очень важно для некоторых операций (например, для отсечения нелицевых граней). Для того, чтобы нарисовался хотя бы один треугольник ***n*** должно быть больше или равно **3**.
GL_TRIANGLE_FAN	То же, что и **GL_TRIANGLE_STRIP**, но порядок вершин следующий: **V[0]**, **V[1]**, **V[2]**, затем **V[0]**, **V[2]**, **V[3]**, затем **V[0]**, **V[3]**, **V[4]**, и так далее.
GL_QUADS	Рисует серию четырехугольников (полигонов с четырьмя сторонами), используя вершины **V[0]**, **V[1]**, **V[2]**, **V[3]**, затем **V[4]**, **V[5]**, **V[6]**, **V[7]**, и так далее. Если ***n*** не кратно **4**, последние **1**, **2** или **3** вершины игнорируются.
GL_QUAD_STRIP	Рисует серию прямоугольников, используя следующий порядок вершин: **V[0]**, **V[1]**, **V[3]**, **V[2]**, затем **V[2]**, **V[3]**, **V[5]**, **V[4]**, затем **V[4]**, **V[5]**, **V[7]**, **V[6]** и так далее. ***n*** должно быть не меньше **4**, иначе ничего не будет нарисовано. Если ***n*** нечетно, последняя вершина игнорируется.
GL_POLYGON	Рисует полигон, используя точки **V[0]**, **V[1]**, **V[2]**, ..., **V[*n*-1]** в качестве вершин. ***n*** должно быть не меньше **3** или ничего нарисовано не будет. Кроме того, описываемый полигон не должен иметь самопересечений и должен быть выпуклым. Если вершины не удовлетворяют этим условиям, результат непредсказуем.

2.3.3.1 Ограничения на использование glBegin() и glEnd()

Наиболее важная информация о вершинах – это их координаты, которые указываются командой **glVertex*()**. Вы также можете указывать для каждой вершины специфические данные, такие как цвет, вектор нормали, координаты текстуры или любую их комбинацию, используя специальные команды. Кроме того, некоторые другие команды допустимы между **glBegin()** и **glEnd()**. Таблица **2-3** содержит полный список этих команд.

Таблица **2-3**. Команды, допустимые между вызовами **glBegin()** и **glEnd()**.

Команда	Назначение
glVertex*()	установка координат вершины
glColor*()	установка текущего цвета
glIndex*()	установка текущего цветового индекса
glNormal*()	установка координат вектора нормали
glTexCoord*()	установка координат текстуры
glMultiTexCoord*ARB()	установка координат текстуры при мультитекстурировании
glEdgeFlag*()	контролирует рисование ребер
glMaterial*()	установка свойств материала
glArrayElement()	выделяет данные из массива вершин
glEvalCoords*(), **glEvalPoint*()**	генерирует координаты
glCallList(), **glCallLists()**	выполняет список отображения

Никакие другие команды недопустимы между **glBegin()** и **glEnd()**, и вызов большинства других команд **OpenGL** вызовет ошибку. Некоторые команды, связанные с массивами вершин, такие как **glEnableClientState()** и **glVertexPointer()** при вызове внутри **glBegin()** и **glEnd()** ведут к неопределенным результатам, но не обязательно генерируют ошибку. (Аналогично, функции оконной системы, связанные с **OpenGL**, такие как функции **glX*()** ведут себя неопределенным образом между **glBegin()** и

glEnd().) Ошибки, связанные с употреблением недопустимых команд, достаточно трудно обнаруживать и поэтому таких употреблений следует избегать.

Обратите внимание, однако, что ограничения накладываются только на команды **OpenGL** и функции, связанные с ней. Вы можете свободно помещать другие языковые конструкции и вызовы между **glBegin()** и **glEnd()**. Пример **2-4**, иллюстрирующий это, рисует окружность.

Пример **2-4.** Языковые конструкции между **glBegin()** и **glEnd()**

```
#define PI 3.1415926535898
GLint circle_points=100;
glBegin(GL_LINE_LOOP);
      for(i=0;i<CIRCLE_POINTS;I++)
      {
            angle=2*PI*i/circle_points;
            glVertex2f(cos(angle),sin(angle));
      }
glEnd();
```

Замечание: В примере приведен не самый эффективный способ рисования окружности, особенно, если вы планируете делать это часто. Используемые графические команды обычно достаточно быстры, но в приведенном коде для каждой вершины рассчитывается угол, и вызываются функции **sin()** и **cos()**. Кроме того, здесь присутствует наложение конца ломаной на ее начало. Если вам необходимо рисовать множество окружностей, вычислите координаты вершин единожды и создайте список отображения или используйте массив вершин.

Если только **glVertex*()** не используются для компиляции списка отображения, все они должны вызываться только внутри командных скобок **glBegin()** и **glEnd()**, иначе **glVertex*()** игнорируются. Если **glVertex*()** используются внутри списка отображения, то и в этом случае командные скобки должны присутствовать: они должны обрамлять либо обращения к **glVertex*()** внутри списка, либо команды исполнения готового списка (например, **glCallList()**).

Несмотря на то, что между **glBegin()** и **glEnd()** допустимо достаточно много команд, вершина создается только при вызове команды **glVertex*()**. В момент вызова **glVertex*()** **OpenGL** ассоциирует с создаваемой вершиной текущий цвет, координаты текстуры, информацию о векторе нормали и так далее. Чтобы убедиться в этом, посмотрите на следующий код. Первая точка рисуется красным цветом, вторая и третья – синим, несмотря на дополнительные команды изменения текущего цвета.

```
glBegin(GL_POINTS);
glColor3f(0.0,1.0,0.0); //зеленый
glColor3f(1.0,0.0,0.0); //красный
      glVertex(...);
      glColor3f(1.0,1.0,0.0); //желтый
      glColor3f(0.0,0.0,1.0); //синий
      glVertex(...);
      glVertex(...);
glEnd();
```

Вы можете использовать любую комбинацию **24** версий команды **glVertex*()** внутри **glBegin()** и **glEnd()**, хотя в реальных приложениях вершинные данные имеют тенденцию сохранять свой формат и поэтому, обычно достаточно одной версии. Если в вашем приложении вызовы команд установки параметров вершин имеют достаточно постоянную последовательность (например, **glColor***, **glVertex***, **glColor***, **glVertex*** и так далее), вы можете получить выигрыш по производительности путем использования массивов вершин.

2.4 Основы управления состоянием

В предыдущей секции вы видели пример переменной состояния – текущего **RGBA –** цвета, и как он может быть назначен примитиву. **OpenGL** управляет множеством переменных состояния. Объект может быть отображен с учетом освещения, текстурирования, удаления невидимых поверхностей, тумана, других *механизмов* **(states)**, влияющих на его внешний вид, и каждый из этих механизмов имеет множество, ассоциированных с ним, переменных состояния.

По умолчанию, большинство из этих механизмов **OpenGL** неактивно, так как они могут быть весьма дороги в смысле производительности. Например, включение текстурирования практически наверняка замедлит процесс визуализации примитива. Однако изображение будет лучше по качеству, и будет выглядеть более реалистично.

Для включения или выключения многих механизмов используются **2** простые команды:

```
void glEnable (GLenum cap);
void glDisable (GLenum cap);
```

glEnable() включает механизм, **glDisable()** выключает его. Существует более **40** величин, которые могут передаваться в качестве параметров **glEnable()** или **glDisable()**. Вот некоторые примеры: **GL_BLEND (**контролирует наложение **RGBA –** величин**), GL_DEPTH_TEST (**контролирует сравнение по глубине и обновление буфера глубины**), GL_LINE_STIPPLE (**применение шаблонов для линий**)** и **GL_LIGHTING (**освещение**).**

Вы также можете проверить включен или выключен определенный механизм:

```
GLboolean glIsEnabled (GLenum capability);
```

Возвращает **GL_TRUE** или **GL_FALSE** в зависимости от того включен или выключен запрашиваемый механизм.

Механизмы, которые вы только что видели, имеют два положения: включен или выключен. Однако большинство команд **OpenGL** устанавливает значения более сложных переменных состояния. Например, команда **glColor3f()** устанавливает **3** величины, которые являются частью механизма **GL_CURRENT_COLOR**. Существует **5** команд опроса, используемых для получения значений переменных из различных механизмов:

```
void glGetBooleanv (GLenum pname, GLboolean *params);
void glGetIntegerv (GLenum pname, GLint *params);
void glGetFloatv (GLenum pname, GLfloat *params);
void glGetDoublev (GLenum pname, GLdouble *params);
void glGetPointerv (GLenum pname, GLvoid *params);
```

Извлекают значения переменных состояния булевского типа, целого типа, дробного типа одинарной и двойной точности и типа указателя. Аргумент ***pname*** **–** символическая константа, определяющая запрашиваемую переменную состояния, а ***params*** **–** указатель на массив соответствующего типа, в который следует поместить возвращаемые данные. Например, чтобы получить текущий **RGBA –** цвет используйте **glGetIntegerv(GL_CURRENT_COLOR, *params*)** или **glGetFloatv(GL_CURRENT_COLOR, *params*)**. Конверсия типов **(**если она необходима**)** автоматически производится для того, чтобы вернуть значения запрашиваемого типа. Эти команды опроса годятся для извлечения значений почти всех **(**но не всех**)** переменных состояния.

2.5 Отображение точек, линий и полигонов

По умолчанию, точка отображается как один пиксель на экране, линия рисуется сплошной, в один пиксель толщиной, полигоны рисуются со сплошной заливкой. Следующие параграфы разъясняют, как можно изменить эту ситуацию.

2.5.1 Подробно о точках

Для управления размером рисуемых точек используйте **glPointSize()** и укажите желаемый размер **(**в пикселях**)** как аргумент.

```
void glPointSize (GLfloat size);
```

Устанавливает длину и высоту **(**в пикселях**)** для визуализируемой точки, ***size*** должен быть больше **0.0** и, по умолчанию, равен **1.0.**

Реальный набор пикселей на экране, который будет нарисован для различных размеров точки зависит от того, включен или выключен антиалиасинг. **(**Антиалиасинг **–** это техника сглаживания рисуемых точек и линий.**)** Если антиалиасинг выключен **(**а он выключен по умолчанию**),** дробные размеры точек округляются до ближайшего целого и для каждой точки на экране будет нарисован квадратный регион пикселей. Таким образом, если размер точки равен **1.0**, будет нарисован квадрат с размерами **1 x 1** пиксель, если размер точки **2.0** будет нарисован квадрат с размерами **2 x 2** пикселя и так далее.

Когда антиалиасинг включен, рисуется циркулярная группа пикселей, и пиксели по границе обычно имеют цвет с пониженной интенсивностью для придания границе гладкого вида. В этом режиме дробные размеры точек до целого не округляются **(**рисунок **2-8).**

Рисунок **2-8.** Точки разного размера в разных режимах

Большинство реализаций **OpenGL** поддерживают точки очень большого размера. Вы можете запрашивать минимальный и максимальный размеры для несглаженных точек, используя аргумент **GL_ALIASED_POINT_SIZE_RANGE** в команде **glGetFloatv().** Точно так же вы можете запрашивать минимальный и максимальный размеры сглаженных точек, используя в той же команде аргумент **GL_SMOOTH_POINT_SIZE_RANGE.** Допустимые размеры сглаженных точек дискретно меняются в диапазоне от минимума до максимума. Вызвав **glGetFloatv()** с параметром **GL_SMOOTH_POINT_SIZE_GRANULARITY,** вы узнаете, с какой точностью сглаженные точки отображаются на экране, то есть величину минимального изменения размера точки внутри диапазона от минимума до максимума, имеющую видимый эффект **(**величину гранулирования**).** Например, если вы запрашиваете **glPointSize(*2.34*),** а величина гранулирования равна **0.1,** размер точки будет округлен до **2.4.**

Замечание: В некоторых случаях довольно трудно получить указанные выше параметры, поскольку соответствующие константы могут быть не определены в заголовочных файлах, поставляемых с вашим компилятором или реализацией OpenGL. Однако, как правило, в этих файлах определена константа GL_POINTS_SIZE_RANGE (по смыслу соответствующая GL_SMOOTH_POINT_SIZE_RANGE) и GL_POINT_SIZE_GRANULARITY (соответствующая GL_SMOOTH_POINT_SIZE_GRANULARITY). То же касается и параметров линий (максимальной и минимальной толщины и гранулярности). Вполне возможна ситуация, когда в заголовочных файлах определены лишь константы GL_LINE_WIDTH_RANGE (по результату аналогичная GL_SMOOTH_LINE_WIDTH_RANGE) и GL_LINE_WIDTH_GRANULARITY (соответствующая GL_SMOOTH_LINE_WIDTH_GRANULARITY). Вообще подобные сложности могут возникать и в других случаях из-за различий в реализациях, версиях библиотеки, средствах разработки и так далее. Благо, они не настолько часты.

2.5.2 Подробно о линиях

В **OpenGL** вы можете рисовать линии различной толщины, а также линии, имеющие шаблон – пунктирные, штриховые, штрих – пунктирные и так далее.

2.5.2.1 Толщина линий

```
void glLineWidth (GLfloat width);
```

Устанавливает толщину линии в пикселях, ***width*** должно быть больше **0.0** и по умолчанию равно **1.0**.

Реальная визуализация линий зависит от механизма антиалиасинга по аналогии с точками. Без антиалиасинга значения толщины равные **1**, **2** и **3** задают линии на экране в **1**, **2** и **3** пикселя толщиной. При включенном антиалиасинге для толщины допустимы дробные значения и пиксели по границе линии обычно рисуются с меньшей интенсивностью. Как и в случае с точками, конкретная реализация **OpenGL** может ограничивать толщину несглаженной линии, максимальной толщиной сглаженной линии, округленной до ближайшего целого. Вы можете получить диапазон поддерживаемых значений толщины несглаженных линий посредством команды **glGetFloatv()** с аргументом **GL_ALIASED_LINE_WIDTH_RANGE**. Чтобы получить минимум и максимум значений толщины для сглаженных линий и величину гранулирования, поддерживаемые вашей реализацией **OpenGL**, следует употребить с той же командой аргументы **GL_SMOOTH_LINE_WIDTH_RANGE** и **GL_SMOOTH_LINE_WIDTH_GRANULARITY** соответственно.

Замечание: Имейте в виду, что по умолчанию линии имеют толщину в 1 пиксель, так что они кажутся толще при низком разрешении монитора и тоньше при высоком. Для мониторов это, как правило, не проблема, но если вы используете OpenGL для вывода изображения на другое устройство, например, плоттер высокого разрешения, линия в 1 пиксель толщиной может быть практически неразличима. В подобных случаях приходится вычислять значения толщины для каждого разрешения, принимая в расчет физический размер пикселя на экране или устройстве.

Дополнительно: При рисовании несглаженных линий следует учесть, что их толщина измеряется вовсе не перпендикулярно самой линии. Вместо этого толщина измеряется в направлении оси ординат при условии, что **|y2-y1|<|x2-x1|(**где **(x1,y1)** и **(x2,y2) -** координаты концов отрезка**)** и в направлении оси абсцисс в иных случаях. Рисование сглаженного отрезка определенной толщины полностью эквивалентно рисованию закрашенного прямоугольника соответствующей высоты, центрированного отрезку в один пиксель.

2.5.2.2 Шаблоны отрезков

Чтобы задать шаблон отрезка (например, для получения пунктирных или штриховых отрезков) следует использовать команду **glLineStipple()** и затем включить шаблонирование командой **glEnable()**.

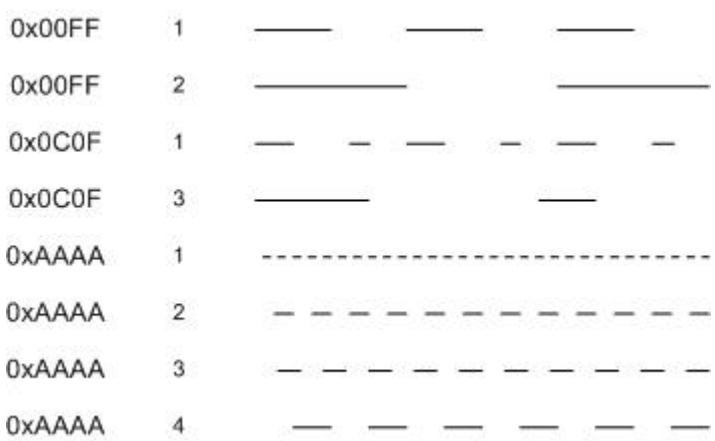
```
glLineStipple(1,0x3F07);
glEnable(GL_LINE_STIPPLE);
```

```
void glLineStipple (Glint factor, GLushort pattern);
```

Устанавливает текущий шаблон для отрезка. Аргумент ***pattern*** – это **16**-битная серия из нулей и единиц, определяющая, как будет рисоваться отрезок. Она повторяется по необходимости для шаблонирования всего отрезка. Единица означает, что соответствующая точка отрезка будет нарисована на экране, ноль означает, что точка нарисована не будет (на попиксельной основе). Шаблон применяется, начиная с младшего бита аргумента **pattern**. Шаблон может быть растянут с учетом значения фактора повторения ***factor***. Каждый бит шаблона при наложении на отрезок расценивается как ***factor*** битов того же значения, идущих друг за другом. Например, если в шаблоне встречаются подряд три единицы, а затем два нуля и ***factor*** равен **3**, то шаблон будет трактоваться как содержащий **9** единиц и **6** нулей. Допустимые значения аргумента ***factor*** ограничены диапазоном от **1** до **256**. Шаблонирование должно быть включено передачей аргумента **GL_LINE_STIPPLE** в функцию **glEnable()**. Оно блокируется передачей того же аргумента в **glDisable()**.

Итак, в предыдущем примере с шаблоном равным **0x3F07** (что в двоичной системе счисления соответствует записи **0011111100000111**) отрезок будет выведен на экран, начинаясь (по порядку) с **3** нарисованных пикселей, **5** отсутствующих, **6** нарисованных и **2** отсутствующих (если вам кажется, что мы применили шаблон задом - наперед вспомните, что он применяется, начиная с младшего бита). Если длина нашего отрезка на экране больше **16** пикселей, начиная с **17**-го, шаблон будет применен заново и так далее до конца отрезка. Если бы ***factor*** был равен **2**, шаблон был бы растянут, и отрезок выглядел бы следующим образом: вначале **6** нарисованных пикселей, затем **10** отсутствующих, **12** нарисованных и **4** отсутствующих. На рисунке **2-9** показаны отрезки, нарисованные с применением различных шаблонов и факторов повторения шаблона. Если шаблонирование заблокировано, все отрезки рисуются таким же образом, как если бы шаблон был установлен в **0xFFFF**, а фактор повторения в **1**. Обратите внимание, что шаблонирование может применяться в комбинации с линиями различной толщины.

Рисунок **2-9**. Шаблонированные отрезки



Если рисуется ломаная (с помощью **GL_LINE_STRIP** или **GL_LINE_LOOP**) то шаблон накладывается на нее непрерывно, независимо от того, где кончается один сегмент и начинается другой. В противовес этому для каждой индивидуальной линии (рисуемой с помощью **GL_LINES**) шаблон начинается заново, даже если все команды указания вершин вызываются внутри одного блока **glBegin() – glEnd()**.

Пример **2-5** иллюстрирует результаты экспериментов с различными шаблонами и значениями толщины отрезков**.** Он также показывает разницу между рисованием ломаной и отдельных отрезков**.** Результаты работа программы показаны на рисунке **2-10.**

Рисунок **2-10.** Эксперименты с шаблонами и толщиной отрезков

Пример **2-5.** Шаблоны отрезков**.** Файл **lines.cpp**

```
#include
#define drawOneLine(x1,y1,x2,y2)
glBegin(GL_LINES);glVertex2f((x1),(y1));glVertex2f((x2),(y2));glEnd();

void init(void)
{
   glClearColor(1.0,1.0,1.0,0.0);
   glShadeModel(GL_FLAT);
}

void display(void)
{
   int i;

   glClear(GL_COLOR_BUFFER_BIT);

   //Черный цвет для всех линий
   glColor3f(0.0,0.0,0.0);

   glEnable(GL_LINE_STIPPLE);

   //В первом ряду три линии с разными шаблонами
   glLineWidth(1.0);
   glLineStipple(1,0x0101); //Пунктирная
   drawOneLine(50.0,125.0,150.0,125.0);
   glLineStipple(1,0x00FF); //Штриховая
   drawOneLine(150.0,125.0,250.0,125.0);
   glLineStipple(1,0x1C47); //Штрих-пунктирная
   drawOneLine(250.0,125.0,350.0,125.0);

   //Во втором ряду три толстые линии с аналогичными шаблонами
   glLineWidth(5.0);
   glLineStipple(1,0x0101); //Пунктирная
   drawOneLine(50.0,100.0,150.0,100.0);
   glLineStipple(1,0x00FF); //Штриховая
   drawOneLine(150.0,100.0,250.0,100.0);
   glLineStipple(1,0x1C47); //Штрих-пунктирная
   drawOneLine(250.0,100.0,350.0,100.0);

   //В третьем ряду шесть штрих-пунктирных линий, объединенных в ломаную
   glLineWidth(1.0);
   glLineStipple(1,0x1C47); //Штрих-пунктирная
   glBegin(GL_LINE_STRIP);
   for (i=0;i<7;i++)
```

```
      glVertex2f(50.0+((GLfloat)i*50.0),75.0);
   glEnd();

   //В четвертом ряду шесть независимых линий того же шаблона
   for (i=0;i<6;i++)
   {
      drawOneLine(50.0+((GLfloat)i*50.0),50.0,50.0+((GLfloat)(i+1)*50.0),50.0);
   }

   //В пятом ряду 1 штрих-пунктирная линия с фактором повторения=5
   glLineStipple(5,0x1c47);
   drawOneLine(50.0,25.0,350.0,25.0);

   glDisable(GL_LINE_STIPPLE);
   glFlush();
}

void reshape(int w,int h)
{
   glViewport(0,0,(GLsizei) w, (GLsizei) h);
   glMatrixMode(GL_PROJECTION);
   glLoadIdentity();
   gluOrtho2D(0.0,400.0,0.0,150.0);
}

int main(int argc, char **argv)
{
   glutInit(&argc,argv);
   glutInitDisplayMode(GLUT_SINGLE|GLUT_RGB);
   glutInitWindowSize(400,150);
   glutInitWindowPosition(100,100);
   glutCreateWindow("Line Stipple Patterns");
   init();
   glutDisplayFunc(display);
   glutReshapeFunc(reshape);
   glutMainLoop();
   return 0;
}
```

2.5.3 Подробнее о полигонах

Обычно полигоны рисуются в виде закрашенной области пикселей, лежащих внутри их границ, но также они могут быть нарисованы в виде самой границы (ломаной) или просто в виде точек, соответствующих их вершинам. Возможна как полная заливка, так и применение различных шаблонов. Если смежные полигоны разделяют ребро или вершину, то пиксели, соответствующие на экране этому ребру или вершине рисуются только один раз, то есть они включаются только в один из полигонов. Это делается для того, чтобы для частично прозрачных полигонов ребра не рисовались дважды – это может привести к тому, что такие ребра будут выглядеть темнее или ярче (в зависимости от их цвета). Заметьте, что это может привести к тому, что узкие полигоны при рисовании будут иметь дыры в одном или более рядов или столбцов пикселей. Антиалиасинг для полигонов более сложен, чем для точек и отрезков.

2.5.3.1 Закрашенные полигоны, полигоны в виде линий или точек

У полигона две стороны или грани—лицевая и обратная, и он может быть визуализирован по-разному в зависимости от того, которая из сторон видна наблюдателю. Это позволяет вам создавать изображения таким образом, как если бы вы заглядывали внутрь объекта у которого лицевые и обратные части рисуются по-разному (например, сфера, разрезанная пополам). Полигон также считается лицевым или обратным в зависимости от того, какой из граней он повернут к наблюдателю. По умолчанию, лицевые и обратные грани изображаются одинаково. Чтобы это изменить, а также, чтобы рисовать только вершины или границы полигона, используйте команду **glPolygonMode()**.

```
void glPolygonMode (GLenum face, GLenum mode);
```

Управляет режимом отображения для лицевых и обратных граней полигонов. Параметр ***face*** указывает, для каких граней изменяется режим отображения и может принимать значения **GL_FRONT_AND_BACK (**режим меняется и для лицевых и для обратных граней**), GL_FRONT (**только для лицевых**), GL_BACK (**только для обратных**).** Параметр ***mode*** может быть равен **GL_POINT, GL_LINE** или **GL_FILL** в зависимости от желаемого режима отображения: точки, линии или заливка. По умолчанию оба типа граней рисуются в виде заполненных областей пикселей **(**заливки

Например, если вы хотите отображать лицевые грани в виде заливки, а обратные в виде линии по границе, используйте две следующие команды:

```
glPolygonMode(GL_FRONT,GL_FILL);
glPolygonMode(GL_BACK,GL_LINE);
```

2.5.3.2 Указание лицевых граней и удаление нелицевых граней

По принятому соглашению, грань полигона **(**и он сам на экране**)** считается лицевой, если ее вершины, начиная с первой, отображаются на экране против часовой стрелки. Вы можете сконструировать поверхность любого **«**разумного**»** тела **(**или ориентируемого тела, например, сфера, торус, чашка – ориентируемы, лист Мёбиуса – нет**)** из полигонов постоянной ориентации. Иными словами, вы можете сконструировать их с использованием только полигонов, чьи вершины идут по часовой стрелке или только полигонов, чьи вершины идут в обратном направлении **(**тело, собранное из полигонов постоянной ориентации, и называется ориентируемым**).**

Представьте себе, что вы долгое время описывали модель некоторой ориентируемой поверхности и вдруг обнаружили, что по ошибке разместили полигоны обратными гранями наружу **(**то есть гранями, вершины которых на экране идут по часовой стрелке**).** Вы можете указать **OpenGL**, что считать лицевыми и обратными гранями с помощью команды **glFrontFace().**

```
void glFrontFace (GLenum mode);
```

Контролирует то, какие грани **OpenGL** считает лицевыми. По умолчанию ***mode*** равняется **GL_CCW**, что соответствует ориентации против часовой стрелки упорядоченного набора вершин спроецированного полигона. Если в параметре ***mode*** указать **GL_CW**, то лицевыми будут считаться грани, чьи вершины расположены по часовой стрелке.

Если сконструированная поверхность, состоящая из непрозрачных полигонов постоянной ориентации, скажем лицевых, полностью замкнута, ни одна из обратных граней полигонов никогда не будет видна – они всегда будут закрыты лицевыми гранями. Если вы наблюдаете снаружи поверхности, вы можете включить механизм удаления нелицевых **(**обратных**)** граней, для того, чтобы **OpenGL** игнорировала и не рисовала грани, определенные ею как обратные. Аналогичным образом, если вы находитесь внутри замкнутой поверхности, вам видны только обратные грани, и вы можете заставить **OpenGL** рисовать только их. Чтобы заставить **OpenGL** игнорировать лицевые или обратные грани полигонов, используйте команду **glCullFace()** и включите удаление невидимых граней командой **glEnable().**

```
void glCullFace (GLenum mode);
```

Задает, какие полигоны должны игнорироваться и удаляться до их преобразования в экранные координаты. Параметр ***mode*** может принимать значение **GL_FRONT, GL_BACK**

или **GL_FRONT_AND_BACK**, что соответствует лицевым, обратным или всем полигонам соответственно. Для того, чтобы команда имела эффект, требуется включить механизм удаления нелицевых граней с помощью параметра **GL_CULL_FACE** и команды **glEnable()**. Чтобы выключить его тот же параметр должен быть передан в команду **glDisable()**.

Дополнительно: Технически решение о том, является ли грань полигона лицевой или обратной зависит от знака его площади вычисленной в оконных координатах. Один из путей рассчитать ее следующий: $a = \frac{1}{2}\sum_{i=0}^{n-2} x_i y_{i\oplus 1} - x_{i\oplus 1} y_i$, где x_i и y_i **i**-ой вершины, а $i \oplus 1 = (i+1) \bmod n$. Предполагая, что указан режим **GL_CCW**, если **a>0**, полигон считается лицевым. В режиме **GL_CW**, полигон считается лицевым, при **a<0**, иначе он считается обратным (повернутым к наблюдателю своей обратной гранью).

2.5.3.3 Шаблонирование полигонов

По умолчанию, если полигон рисуется в виде закрашенной области, он закрашивается одинаково по всей своей площади, без просветов. Эту ситуацию можно изменить, определив шаблон заливки полигона в виде матрицы **32 x 32** бита и установив его командой **glPolygonStipple()**.

```
void glPolygonStipple (const GLubyte *mask);
```

Определяет текущий рисунок шаблона заливки полигона. Параметр ***mask*** – это указатель на битовую карту размером **32 x 32** бита, интерпретируемую в качестве маски, накладываемой на полигон при рисовании (и при необходимости повторяемой). Там, где стоит **1** – соответствующий пиксель полигона будет нарисован, а там, где появляется **0** – пиксель нарисован не будет. Рисунок **2-12** показывает, как рисунок шаблона конструируется из символов в параметре ***mask***. Шаблонирование полигонов включается и выключается с помощью аргумента **GL_POLYGON_STIPPLE** функций **glEnable()** и **glDisable()**. Интерпретация ***mask*** зависит от **GL_UNPACK*** параметров команды **glPixelStore*()**.

Кроме того, вы должны включить шаблонирование полигонов:

```
glEnable(GL_POLYGON_STIPPLE);
```

Используйте **glDisable()** с аналогичным аргументом для выключения шаблонирования.

На рисунке **2-11** показаны результаты рисования трех полигонов: одного нешаблонированного и двух шаблонированных, каждый со своим рисунком. Сама программа приводится в примере **2-6**. Инверсия готового полигона (рисунок **2-11**) относительно шаблона (рисунок **2-12**) происходит потому, что программа рисует полигоны белым цветом на черном фоне (шаблон не влияет на цвет рисуемых точек, он просто определяет, какие точки полигона будут нарисованы, а какие нет).

Рисунок **2-11**. Шаблонированные полигоны

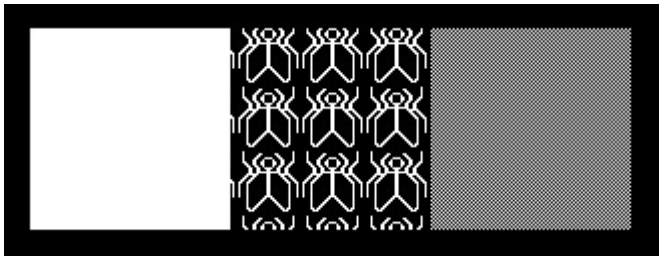
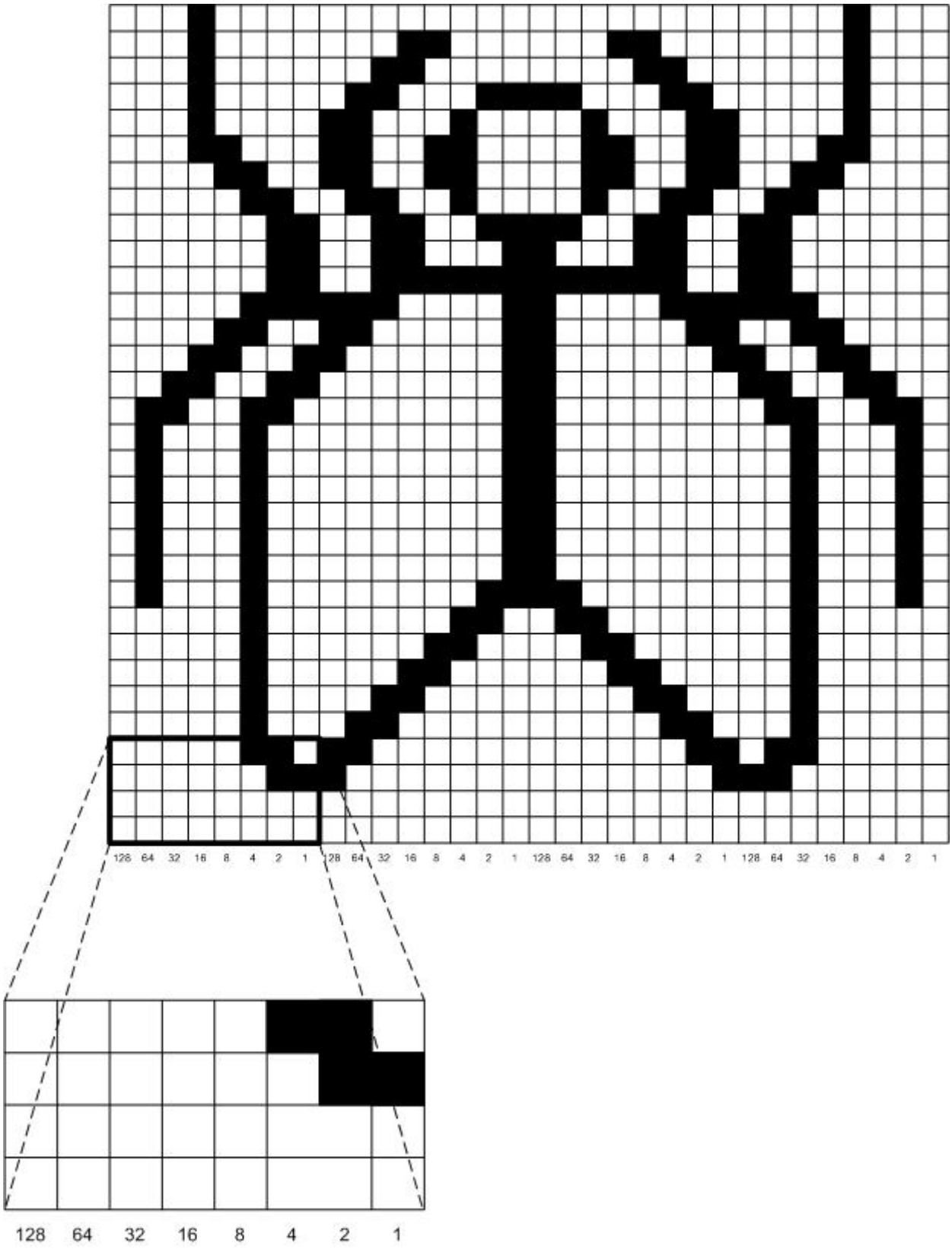


Рисунок **2-12.** Интерпретация рисунка шаблона заливки полигонов



По умолчанию для каждого байта наиболее значимым битом считается младший. Порядок битов может быть изменен вызовом **glPixelStore*()**.

Пример **2-6.** Шаблоны полигонов**.** Файл **polys.cpp**

```
#include
```

```
void init(void)
{
   glClearColor(0.0,0.0,0.0,0.0);
   glShadeModel(GL_FLAT);
}

void display(void)
{
   GLubyte fly[]= {
      0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,
      0x03,0x80,0x01,0xC0,0x06,0xC0,0x03,0x60,
      0x04,0x60,0x06,0x20,0x04,0x30,0x0C,0x20,
      0x04,0x18,0x18,0x20,0x04,0x0C,0x30,0x20,
      0x04,0x06,0x60,0x20,0x44,0x03,0xC0,0x22,
      0x44,0x01,0x80,0x22,0x44,0x01,0x80,0x22,
      0x44,0x01,0x80,0x22,0x44,0x01,0x80,0x22,
      0x44,0x01,0x80,0x22,0x44,0x01,0x80,0x22,
      0x66,0x01,0x80,0x66,0x33,0x01,0x80,0xCC,
      0x19,0x81,0x81,0x98,0x0C,0xC1,0x83,0x30,
      0x07,0xE1,0x87,0xE0,0x03,0x3F,0xFC,0xC0,
      0x03,0x31,0x8C,0xC0,0x03,0x33,0xCC,0xC0,
      0x06,0x64,0x26,0x60,0x0C,0xCC,0x33,0x30,
      0x18,0xCC,0x33,0x18,0x10,0xC4,0x23,0x08,
      0x10,0x63,0xC6,0x08,0x10,0x30,0x0C,0x08,
      0x10,0x18,0x18,0x08,0x10,0x00,0x00,0x08
   };

   GLubyte halftone[] = {
      0xAA,0xAA,0xAA,0xAA,0x55,0x55,0x55,0x55,
      0xAA,0xAA,0xAA,0xAA,0x55,0x55,0x55,0x55,
      0xAA,0xAA,0xAA,0xAA,0x55,0x55,0x55,0x55,
      0xAA,0xAA,0xAA,0xAA,0x55,0x55,0x55,0x55,
      0xAA,0xAA,0xAA,0xAA,0x55,0x55,0x55,0x55,
      0xAA,0xAA,0xAA,0xAA,0x55,0x55,0x55,0x55,
      0xAA,0xAA,0xAA,0xAA,0x55,0x55,0x55,0x55,
      0xAA,0xAA,0xAA,0xAA,0x55,0x55,0x55,0x55,
      0xAA,0xAA,0xAA,0xAA,0x55,0x55,0x55,0x55,
      0xAA,0xAA,0xAA,0xAA,0x55,0x55,0x55,0x55,
      0xAA,0xAA,0xAA,0xAA,0x55,0x55,0x55,0x55,
      0xAA,0xAA,0xAA,0xAA,0x55,0x55,0x55,0x55,
      0xAA,0xAA,0xAA,0xAA,0x55,0x55,0x55,0x55,
      0xAA,0xAA,0xAA,0xAA,0x55,0x55,0x55,0x55,
      0xAA,0xAA,0xAA,0xAA,0x55,0x55,0x55,0x55,
      0xAA,0xAA,0xAA,0xAA,0x55,0x55,0x55,0x55
   };

   glClear(GL_COLOR_BUFFER_BIT);
   glColor3f(1.0,1.0,1.0);

   //Рисуем один нешаблонированный прямоугольник
   //и два шаблоннированных
   glRectf(25.0,25.0,125.0,125.0);

   glEnable(GL_POLYGON_STIPPLE);
   glPolygonStipple(fly);
   glRectf(125.0,25.0,225.0,125.0);

   glPolygonStipple(halftone);
   glRectf(225.0,25.0,325.0,125.0);
   glDisable(GL_POLYGON_STIPPLE);

   glFlush();
}

void reshape(int w, int h)
{
   glViewport(0,0,(GLsizei) w, (GLsizei) h);
   glMatrixMode(GL_PROJECTION);
   glLoadIdentity();
   gluOrtho2D(0.0,350.0,0.0,150.0);
}

int main(int argc, char **argv)
{
```

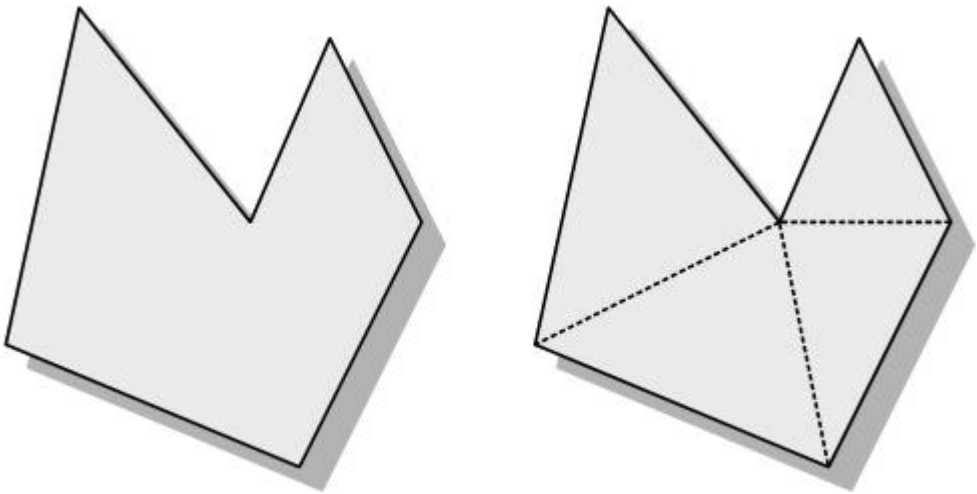
```
    glutInit(&argc,argv);
    glutInitDisplayMode(GLUT_SINGLE|GLUT_RGB);
    glutInitWindowSize(350,150);
    glutCreateWindow("Polygon Stipple Patterns");
    init();
    glutDisplayFunc(display);
    glutReshapeFunc(reshape);
    glutMainLoop();
    return 0;
}
```

Ради повышения эффективности можно использовать списки отображения **OpenGL** для хранения рисунков заливки полигонов**.**

2.5.3.4 Указание граничных ребер

Дополнительно: OpenGL может визуализировать только выпуклые полигоны**,** однако на практике часто требуется нарисовать полигон**,** не являющийся выпуклым**.** Для этого вы обычно разделяете его на выпуклые полигоны **–** чаще всего треугольники**,** как показано на рисунке **2-13** и рисуете эти треугольники**.** К несчастью**,** после декомпозиции вы не можете использовать **glPolygonMode()** для рисования полигона в виде его границы**,** поскольку помимо границы самого полигона**,** вы в этом случае получите еще и границы всех треугольников**,** из которых он состоит**.** Для решения этой проблемы вы можете указать **OpenGL,** является ли каждая конкретная вершина**,** начинающей граничное ребро или нет **(**из двух вершин**,** образующих ребро**,** начинающей его считается вершина с меньшим номером**). OpenGL** отслеживает эту информацию**,** храня и передавая вместе с каждой вершиной соответствующий бит**,** показывающий**,** является ли вершина границеобразующей **(**начинает ли она граничное ребро**).** При таком подходе**,** во время рисования полигона в режиме **GL_LINE** ребра**,** не являющиеся граничными**,** не рисуются**.** На рисунке **2-13,** ребра**,** добавленные при триангуляции и не являющиеся граничными**,** выделены пунктиром**.**

Рисунок **2-13.** Декомпозиция невыпуклого полигона



По умолчанию**,** все вершины помечены как образующие границу**,** но вы можете контролировать установку флага ребра с помощью команды **glEdgeFlag*().** Эта команда используется между **glBegin()** и **glEnd ()** и влияет на все вершины**,** перечисленные после нее до следующего вызова **glEdgeFlag*().** Команда влияет только на вершины**,** указываемые для полигонов**,** треугольников и прямоугольников и не влияет на вершины**,** перечисляемые для цепочек треугольников или прямоугольников **(GL_TRIANGLE_STRIP, GL_QUAD_STRIP).**

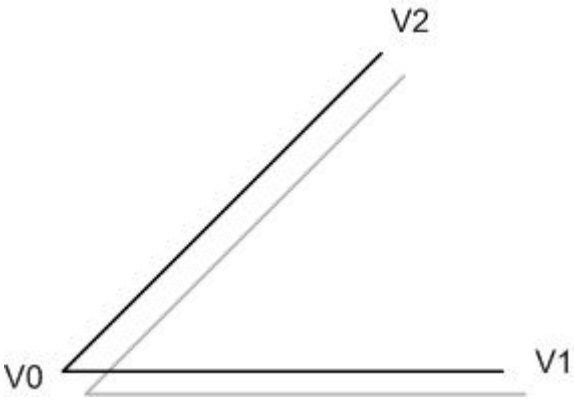
```
void glEdgeFlag (GLboolean flag);
```

```
void glEdgeFlagv (const GLboolean *flag);
```

Указывает, должна ли вершина считаться границеобразующей (начинающей одно из граничных ребер). Если ***flag*** равен **GL_TRUE**, флаг ребра для вершины устанавливается в **TRUE** (значение по умолчанию) и все перечисляемые далее вершины считаются, начинающими граничные ребра. Это происходит до тех пор, пока функция не вызвана еще раз с параметром **GL_FALSE**. Вершины, следующие за этим вызовом, начинающими граничные ребра не считаются и в режиме рисования полигонов **GL_LINE** на экране не отображаются.

Например, в листинге **2-7** рисуются линии, показанные на рисунке **2-14**.

Рисунок **2-14.** Полигон, нарисованный в виде границы при использовании флага ребра.



Пример **2-7.** Пометка границеобразующих вершин

```
glPolygonMode(GL_FRONT_AND_BACK,GL_LINE);
glBegin(GL_POLYGON);
   glEdgeFlag(GL_TRUE);
   glVertex3fv(V0);
   glEdgeFlag(GL_FALSE);
   glVertex3fv(V1);
   glEdgeFlag(GL_TRUE);
   glVertex3fv(V2);
glEnd();
```

2.6 Вектор нормали

Вектором нормали (или просто *нормалью*) называется вектор, который указывает в направлении перпендикулярном поверхности. Для плоской поверхности это перпендикулярное направление одинаково во всех точках, но в общем случае, для изогнутой поверхности направление нормали может быть разным в каждой точке поверхности. При использовании **OpenGL** вы можете указывать нормаль для всего полигона или для каждой вершины. Вершины одного полигона могут разделять одну и ту же нормаль (для плоских поверхностей) или иметь разные нормали (для изогнутых поверхностей). Однако вы можете назначать нормали только в вершинах.

Вектора нормали объекта определяют его ориентацию в пространстве – в частности, его ориентацию по отношению к источникам света. Эти вектора нужны **OpenGL** для определения количества света, падающего на вершины объекта (метод закраски Гуро). Вы определяете вектора нормалей объекта в то же время, когда задаете его геометрию.

Вы используете **glNormal*()**, чтобы установить текущую нормаль в значения параметров, передаваемых команде. При последующих вызовах **glVertex*()** текущая

нормаль ассоциируется с каждой вершиной. часто, каждая вершина имеет свою нормаль, что требует серии дополнительных вызовов, как показано в примере **2-8.**

Пример **2-8.** Вектора нормали в вершинах

```
glBegin(GL_POLYGON);
   glNormal3fv(n0);
   glVertex3fv(v0);
   glNormal3fv(n1);
   glVertex3fv(v1);
   glNormal3fv(n2);
   glVertex3fv(v2);
   glNormal3fv(n3);
   glVertex3fv(v3);
glEnd();
```

```
void glNormal3{bsidf}(TYPE nx, TYPE ny, TYPE nz);
void glNormal3{bsidf}v(const TYPE *v);
```

Устанавливает текущий вектор нормали, определяемый аргументами. Невекторная версия команды **(**без **v)** принимает три аргумента, определяющие вектор **(*nx*,*ny*,*nz*)**, принимаемый в качестве нормали. В качестве альтернативы вы можете использовать векторную версию этой команды **(**с **v)** и передавать в качестве параметра массив, содержащий три элемента, определяющих желаемую нормаль. При использовании версий команды, работающих с типами **b**, **s** и **i**, параметры линейно масштабируются до диапазона **[-1.0**, **1.0]**.

Имейте в виду, что в каждой точке существует не один, а два вектора перпендикулярных поверхности, и они указывают в противоположных направлениях. По принятому соглашению, нормалью считается тот из них, который указывает **«**наружу**»** поверхности **(**а не **«**вовнутрь**»). (**Если вы в программе перепутали понятия **«**наружу**»** и **«**вовнутрь**»**, просто измените каждый вектор нормали с **(*x*, *y*, *z*)** на **(*-x*, *-y*, *-z*))**.

Также имейте в виду, что поскольку нормали нужны только для указания направления, их длина по большей части не имеет значения. Вы можете указывать нормали любой длины, но до того, как смогут быть произведены вычисления, связанные с освещением, все нормали должны быть преобразованы к длине равной **1. (**Вектор, имеющий единичную длину, называется *нормализованным***)**. В общем, вам следует указывать нормализованные нормали. Для нормализации вектора поделите каждую из трех его координат на его длину:

$$\sqrt{x^2 + y^2 + z^2}$$

Вектора нормалей остаются нормализованными до тех пор, пока вы используете только повороты и переносы своих объектов. Если же вы производите масштабирование, используете специфическую матрицу преобразования или изначально указываете ненормализованные вектора нормалей – вы должны заставить **OpenGL** нормализовать векторы после преобразований. Чтобы это сделать, вызовите **glEnable(GL_NORMALIZE).**

Если вы устанавливаете вектора нормали единичной длины и применяете только равномерное масштабирование **(**с сохранением одинаковых пропорций по всем осям**)**, то для нормализации вы можете также использовать **glEnable(GL_RESCALE_NORMAL).** В этом случае для возвращения нормалей к единичной длине **OpenGL** будет делить все их координаты на одинаковую величину изменения масштаба, взятую из матрицы модельного преобразования.

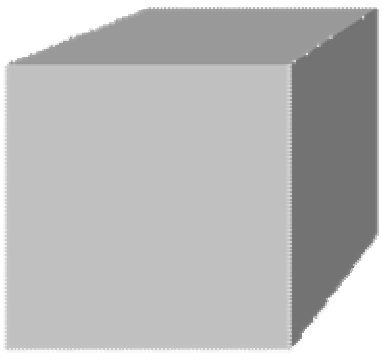
Заметьте, что автоматическая нормализация или масштабирование обычно требуют дополнительных расчетов, которые могут снизить быстродействие вашего приложения. Равномерное масштабирование нормалей с помощью **GL_RESCALE_NORMAL** обычно менее расточительно в смысле производительности, чем полномасштабная нормализация с помощью **GL_NORMALIZE**. По умолчанию оба описанных механизма (и нормализация, и масштабирование) выключены.

2.7 Вершинные массивы

Как вы могли заметить, для визуализации геометрических примитивов **OpenGL** требует вызовов достаточно большого числа функций. Например, рисование **20**-стороннего полигона требует **22** вызовов функций: сначала **glBegin()**, за тем по одному вызову **glVertex*()** для каждой из **20**-ти вершин и, в завершении, **glEnd()**. В двух предыдущих примерах кода ввод дополнительной информации (флагов ребра и нормалей к поверхности) также приводил к дополнительным вызовам. В такой ситуации количество вызываемых команд для рисования одного объекта может быстро удвоиться или даже утроиться. Для некоторых систем дополнительные вызовы функций могут привести к значительному снижению быстродействия вашего приложения.

Еще одной проблемой является излишняя обработка вершин, являющихся общими для нескольких смежных полигонов. Например, куб на рисунке **2-15** состоит из **6**-ти граней и **8**-ми общих вершин. К несчастью, при использовании стандартных методов описания этого объекта, каждая вершина должна быть описана **3** раза: по одному разу для каждой грани, которая ее использует. Таким образом, должны быть обработаны **24** вершины, хотя в принципе достаточно всего **8**-ми.

Рисунок **2-15**. Шесть сторон, восемь общих вершин



Механизм вершинных массивов **OpenGL** позволяет вам указывать разнообразные данные, связанные с вершинами, при помощи всего нескольких массивов и получать доступ к этим данным при помощи всего нескольких вызовов команд. При использовании вершинных массивов все **20** вершин двадцатистороннего полигона могут быть помещены в один массив и переданы для визуализации с помощью одной функции. Если кроме этого каждая вершина имеет вектор нормали, все эти вектора могут быть помещены в другой массив и тоже вызваны одной функцией.

Помещение данных в вершинные массивы может увеличить быстродействие вашего приложения. Использование вершинных массивов снижает количество вызовов функций, что увеличивает быстродействие. Кроме того, использование вершинных массивов может устранить избыточность при обработке общих вершин (но не обязательно – совместное использование вершин поддерживается не всеми реализациями **OpenGL**).

Замечание: Вершинные массивы являются частью стандарта OpenGL версии 1.1, но не были частью спецификации OpenGL 1.0. Тем не менее, в версии 1.0 некоторые поставщики реализовывали вершинные массивы в виде расширений.

Для визуализации геометрии с помощью вершинных массивов необходимо выполнить три шага:

1. Активизировать (включить) от одного до шести массивов, каждый из которых применяется для хранения различных данных: координат вершин, **RGBA** – цветов, цветовых индексов, нормалей, координат текстуры или флагов ребра.
2. Поместить данные в массив или массивы. Доступ к массивам производится с помощью указателей на их местонахождение в памяти. В клиент-серверной модели эти данные сохраняются в адресном пространстве клиента.
3. Нарисовать геометрию с помощью имеющихся данных. **OpenGL** разрешает указатели и одновременно извлекает данные, касающиеся каждой вершины (координаты, цвет и так далее), из всех активизированных массивов. (В клиент-серверной модели данные передаются в адресное пространство сервера.) Существует три варианта извлечения данных:
    a. Доступ к индивидуальным элементам массива (элементом вершинного массива считается фрагмент данных, относящихся к одной вершине – две координаты, тройка значений компонентов цвета и так далее)
    b. Создание списка индивидуальных элементов массива, которые нужно выбрать
    c. Обработка последовательностей элементов массива.

Метод доступа к данным может зависеть от типа решаемой вами задачи.

Смешанные вершинные массивы данных представляют собой другой часто используемый метод организации. Вместо того, чтобы использовать до шести массивов, содержащих разные типы данных, вы можете смешивать различные данные в одном единственном массиве.

2.7.1 Шаг 1: Включение массивов

Первый шаг заключается в вызове **glEnableClientState()** с параметром, определяющим выбранный для активизации массив. Теоретически, вам может понадобиться до шести вызовов этой функции для активизации всех шести имеющихся массивов. На практике, вы, вероятно, будете использовать от одного до четырех массивов. Например, очень маловероятно, что вы будете одновременно использовать массивы **GL_COLOR_ARRAY** (массив цветов в режиме **RGBA**) и **GL_INDEX_ARRAY** (массив индексов цветов в палитровых режимах), поскольку режим дисплея, используемый вашей программой, поддерживает либо **RGBA** – цвета, либо цветовые индексы, но вряд ли оба одновременно.

```
void glEnableClientState (GLenum array);
```

Применяется для активизации массива. Единственный параметр может принимать значения **GL_VERTEX_ARRAY** (массив координат вершин), **GL_COLOR_ARRAY** (массив цветов в режиме **RGBA**), **GL_INDEX_ARRAY** (массив цветовых индексов для соответствующего режима), **GL_NORMAL_ARRAY** (массив координат векторов нормалей), **GL_TEXTURE_COORD_ARRAY** (массив координат текстуры) или **GL_EDGE_FLAG_ARRAY** (массив значений флага ребра).

Если в программе вы используете освещение, вам может понадобиться определить вектор нормали для каждой вершины. Чтобы сделать это с помощью вершинных массивов необходимо включить не только массив координат вершин, но и массив координат векторов нормали^

```
glEnableClientState(GL_VERTEX_ARRAY);
glEnableClientState(GL_NORMAL_ARRAY);
```

Представьте себе, что в какой-то момент вы хотите выключить освещение и рисовать все геометрические объекты одним цветом. Вам понадобится **glDisable()** для выключения расчетов, связанных с освещением. Кроме того, вы, вероятно, захотите отключить присваивание нормалей вершинам, так как без света – это лишняя трата времени. Чтобы это сделать, вызовите:

```
glDisableClientState(GL_NORMAL_ARRAY);
```

```
void glDisableClientState (GLenum array);
```

Указывает массив, который надо выключить (то есть массив, из которого не нужно забирать данные при обработке). Возможные значения параметра аналогичны значениям параметра **glEnableClientState()**.

Возможно, вы спрашиваете себя, зачем архитекторам **OpenGL** понадобилось создавать эти новые длинные имена команд **(gl*ClientState())**, почему нельзя было просто использовать пару **glEnable() / glDisable()?** Одна из причин заключается в том, что **glEnable() / glDisable()** могут входить в список отображения, а спецификация вершинных массивов – нет, так как данные остаются на клиентской стороне.

Для реализаций **OpenGL**, поддерживающих расширение мультитекстурирования, включение и выключение вершинных массивов влияет только на текущий элемент текстуры.

2.7.2 Шаг 2: Указание данных для массивов

Передача данных для вершинных массивов является достаточно прямолинейным процессом. Одна команда задает один массив в пространстве клиента. Существует **6** различных команд для передачи данных в **6** типов массивов. Кроме того, существует команда, позволяющая указать в качестве источника данных для нескольких типов вершинных массивов, один физический перекрестный массив.

```
void glVertexPointer (Glint size, GLenum type, GLsizei stride, const GLvoid
*pointer);
```

Позволяет указать, где находятся данные для массива вершинных координат. ***pointer*** – это указатель на область памяти, где содержится первая координата первой вершины. ***type*** задает тип данных для каждой координаты в массиве **(GL_SHORT, GL_INT, GL_FLOAT** или **GL_DOUBLE). *size*** – количество координат на одну вершину, должен принимать значения **2, 3** или **4. *stride*** – промежуток в памяти (в байтах) между первой координатой предыдущей и первой координатой следующей вершины. Если ***stride*** равен **0** (специальное значение), считается, что вершины плотно упакованы в памяти.

Для указания данных к другим пяти массивам, существует еще пять похожих команд:

```
void glColorPointer (Glint size, GLenum type, GLsizei stride, const GLvoid
*pointer);
void glIndexPointer (GLenum type, GLsizei stride, const GLvoid *pointer);
void glNormalPointer (GLenum type, GLsizei stride, const GLvoid *pointer);
void glTexCoordPointer (Glint size, GLenum type, GLsizei stride, const GLvoid
*pointer);
void glEdgeFlagPointer (GLsizei stride, const GLvoid *pointer);
```

Главное различие этих команд заключается в том, что для каких-то из них необходимо указывать тип данных и количество элементов данных на каждую вершину, а для каких-то нет. Например, у нормали к поверхности всегда три координаты, таким образом, нет смысла указывать число элементов физического массива на одну нормаль. Флаг ребра всегда имеет булевский тип, поэтому нет смысла указывать число и тип

элементов физического массива на каждую вершину. В таблице **2-4** перечислены возможные значения параметров ***size*** и ***type*** для всех команд загрузки массивов, в которых эти параметры присутствуют, а также значения, принимаемые **OpenGL** изначально, для команд, где нет этих параметров в явном виде.

Таблица **2-4.** Типы и количество элементов **(**на одну вершину**)** вершинных массивов

Команда	**Количество элементов на вершину**	**Тип элементов**
glVertexPointer	**2,3, 4**	**GL_SHORT, GL_INT, GL_FLOAT, GL_DOUBLE**
glNormalPointer	не указывается – всегда **3**	**GL_BYTE, GL_SHORT, GL_INT, GL_FLOAT, GL_DOUBLE**
glColorPointer	**3, 4**	**GL_BYTE, GL_UNSIGNED_BYTE, GL_SHORT, GL_UNSIGNED_SHORT, GL_INT, GL_UNSIGNED_INT, GL_FLOAT, GL_DOUBLE**
glIndexPointer	не указывается – всегда **1**	**GL_UNSIGNED_BYTE, GL_SHORT, GL_INT, GL_FLOAT, GL_DOUBLE**
glTexCoordPointer	**1, 2, 3, 4**	**GL_SHORT, GL_INT, GL_FLOAT, GL_DOUBLE**
glEdgeFlagPointer	не указывается – всегда **1**	не указывается – всегда булевский тип

Для реализаций **OpenGL,** поддерживающих мультитекстурирование, указание массива координат текстуры с помощью **glTexCoordPointer()** влияет только на текущий элемент текстуры.

В примере **2-9** вершинные массивы используются одновременно для хранения координат вершин и их цветов в режиме **RGBA.** Значения с плавающей точкой для **RGB –** цветов и соответствующие им целые координаты вершин **(*x*, *y*)** загружаются в **GL_COLOR_ARRAY (**массив цветов в режиме **RGBA)** и **GL_VERTEX_ARRAY (**массив координат вершин**).**

Пример **2-9.** Включение и загрузка вершинных массивов

```
GLint vertices[]= {25, 25,
                   100, 325,
                   175, 25,
                   175, 325,
                   250, 25,
                   325, 325};
GLfloat colors[]= {1.0, 0.2, 0.2,
                   0.2, 0.2, 1.0,
                   0.8, 1.0, 0.2,
                   0.75, 0.75, 0.75,
                   0.35, 0.35, 0.35,
                   0.5, 0.5, 0.5};

glEnableClientState(GL_COLOR_ARRAY);
glEnableClientState(GL_VERTEX_ARRAY);

glColorPointer(3,GL_FLOAT, 0, colors);
glVertexPointer(2,GL_INT, 0, vertices);
```

2.7.2.1 Смещение

Параметр ***stride*** команд **gl*Pointer()** говорит **OpenGL,** как получать доступ к данным предоставляемого вами физического массива. Этот параметр должен быть равен числу байт памяти между первым элементом массива, относящимся к предыдущей вершине, и первым элементом, относящимся к следующей вершине. Кроме того, параметр может быть равен **0 –** эта величина имеет специальное назначение. Например, предположим, что вы сохранили **RGB –** цвета вершин и их координаты **(*x*, *y*, *z*)** в одном физическом массиве:

```
GLfloat interwined[]={
      1.0, 0.2, 1.0, 100.0, 100.0, 0.0,
      1.0, 0.2, 0.2, 0.0, 200.0, 0.0,
      1.0, 1.0, 0.2, 100.0, 300.0, 0.0,
      0.2, 1.0, 0.2, 200.0, 300.0, 0.0,
      0.2, 1.0, 1.0, 300.0, 200.0, 0.0,
      0.2, 0.2, 1.0, 200.0, 100.0, 0.0};
```

Замечание: Важно четко понимать разницу между физическим массивом и вершинным массивом. Физический массив представляет собой некоторый диапазон памяти, где хранится некоторое количество элементов одинакового типа. Например, вы можете объявить массив типа GLubyte. Элементом массива при этом считается одна величина указанного при объявлении типа. Любой же вершинный массив – понятие, не связанное прямо с физической памятью компьютера, поскольку любая реализация OpenGL может реализовывать вершинные массивы тем способом, который ее разработчики, считают эффективным. Кроме того, элементом любого вершинного массива считаются все данные, относящиеся к одной вершине, независимо от реального типа этих данных и количества компонент. Например, в качестве источника данных для вершинного массива RGB-цветов вы можете задать объявленный и определенный ранее физический массив типа GLubyte. Если вы укажете, что каждый цвет состоит из трех компонент, то одним элементом вершинного массива RGB – цветов будут считаться 3 байта, определяющих эти компоненты для одной вершины. То есть в данном случае одним элементом вершинного массива являются 3 элемента физического массива. Более того, источник данных для вершинных массивов может и не являться физическим массивом с простым типом элементов. Можно объявить массив сложных структур и использовать его. В общем случае, описываемый здесь Шаг 2 как раз и нужен для того, чтобы указать OpenGL, откуда брать данные для вершинных массивов и как их трактовать.

Чтобы сослаться только на цветовые значения массива ***interwined***, следующий вызов начинает забирать данные из начала массива (указатель на которое передается в виде ***&interwined[0]***), затем сдвигается на **6*sizeof(GLfloat)**, что является размером в байтах трех компонент цвета и трех координат одной вершины. Этого прыжка достаточно, чтобы попасть на начало цветовых данных для следующей вершины.

```
glColorPointer(3,GL_FLOAT,6*sizeof(GLfloat),&interwined[0]);
```

Для получения координат вершин, необходимо начать с **4**-го элемента физического массива ***interwined*** (помните, что в **C** нумерация элементов массива начинается с **0**).

```
glVertexPointer(3,GL_FLOAT, 6*sizeof(GLfloat),&interwined[3]);
```

Если ***stride*** равен **0**, считается, что данные плотно упакованы в памяти и массив является гомогенным (или однородным), то есть в нем только цвета, только координаты вершин, только нормали и так далее.

2.7.3 Шаг 3: Разрешение данных

До тех пор, пока не произошло разрешение данных вершинных массивов, они остаются на клиентской стороне и можно легко менять их содержимое. На третьем шаге данные извлекаются из массивов и отправляются на сервер и далее обрабатываются конвейером визуализации.

Данные могут быть извлечены из одного элемента массива (по его индексу), из упорядоченного списка элементов массива (который может определять какие именно данные следует обрабатывать, а какие нет) или из последовательности элементов массива.

2.7.3.1 Разрешение одного элемента массива

```
void glArrayElement (Glint ith);
```

Извлекает данные для одной вершины с индексом ***ith*** из всех включенных в текущий момент массивов. Для массива с координатами вершин соответствующей сгенерированной командой будет **glVertex[*size*][*type*]v()**, где ***size*** будет равен **2, 3** или **4**, а ***type*** – **s, i, f** или **d (**для **GLshort, Glint, GLfloat** и **GLdouble** соответственно**).** И ***size***, и ***type*** были ранее определены вызовом **glVertexPointer().** Для других типов массивов вызов **glArrayElement()** приведет к генерации команд **glEdgeFlagv(), glTexCoord[*size*][*type*]v(), glColor[*size*][*type*]v(), glIndex[*type*]v()** и **glNormal[*type*]v().** Если включен массив вершинных координат, команда **glVertex*v()** будет вызвана последней, после исполнения до пяти других команд, если включены соответствующие массивы.

Обычно **glArrayElement()** вызывается между командами **glBegin()** и **glEnd() (**если **glArrayElement()** вызывается за пределами этих командных скобок, он устанавливает текущее значение для всех переменных состояния, соответствующих включенным массивам, например, текущий цвет, текущий вектор нормали и так далее; если при этом включен массив вершинных координат, то его данные ни на что не влияют, поскольку не существует понятия текущей вершины**).** В примере **2-10** треугольник рисуется с помощью данных, относящихся к третьей, четвертой и шестой вершинам **(**следует опять-таки помнить, что в языке **C** нумерация массивов начинается с **0).**

Пример **2-10.** Использование **glArrayElement()** для задания координат вершин и их цветов

```
glEnableClientState(GL_COLOR_ARRAY);
glEnableClientState(GL_VERTEX_ARRAY);

glColorPointer(3,GL_FLOAT,0,colors);
glVertexPointer(2,GL_INT,0,vertices);

glBegin(GL_TRIANGLES);
      glArrayElement(2);
      glArrayElement(3);
      glArrayElement(5);
glEnd();
```

В период выполнения последние пять строк кода будут иметь такой же эффект, как

```
glBegin(GL_TRIANGLES);
   glColor3fv(colors+(2*3));
   glVertex2iv(vertices+(2*2));
   glColor3fv(colors+(3*3));
   glVertex2iv(vertices+(3*2));
   glColor3fv(colors+(5*3));
   glVertex2iv(vertices+(5*2));
glEnd();
```

Поскольку при использовании **glArrayElement()** происходит только один вызов на одну вершину, это может сократить общее количество вызовов функций и тем самым увеличить быстродействие.

Имейте в виду, что если содержимое массива изменяется между вызовами **glBegin()** и **glEnd(),** нет гарантии, что вы получите оригинальные или измененные данные для запрошенного элемента. Для большей предсказуемости результата не изменяйте данные тех элементов массивов, доступ к которым может понадобиться до завершения примитива.

2.7.3.2 Разрешение списка элементов массива

Команду **glArrayElement()** удобно применять для случайных выборок данных из вершинных массивов. Похожие команды **glDrawElements()** и **glDrawRangeElements()** удобны для проведения случайных выборок данных в более упорядоченной манере.

```
void glDrawElements (GLenum mode, GLsizei count, GLenum type, void *indices);
```

Определяет последовательность геометрических примитивов, используя ***count*** элементов вершинных массивов, индексы которых находятся в массиве ***indices***. ***type*** может принимать значение **GL_UNSIGNED_BYTE, GL_UNSIGNED_SHORT** или **GL_UNSIGNED_INT,** задавая тип данных в массиве ***indices***. Параметр ***mode*** указывает, какие примитивы следует построить, и может принимать те же значения, что и единственный параметр **glBegin(),** например **GL_POLYGON, GL_LINE_LOOP, GL_LINES, GL_POINTS** и так далее.

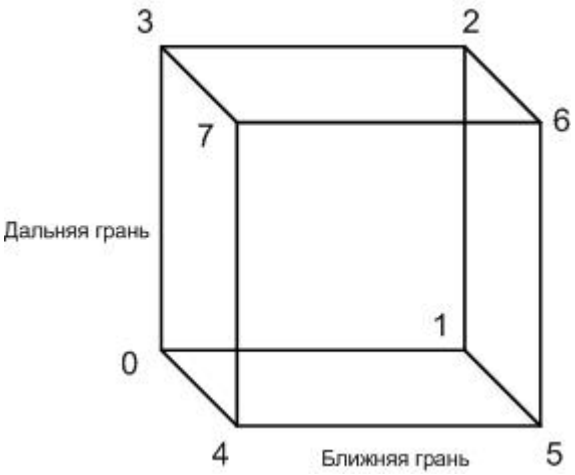
Эффект от вызова **glDrawElements()** практически то же, что и от следующей последовательности команд:

```
int i;
glBegin(mode);
   for(i=0;i<COUNT;I++)
      glArrayElement(indices[i]);
glEnd();
```

В дополнение **glDrawElements()** проверяет ***mode***, ***count*** и ***type*** на корректность значений. Также, в отличие от предыдущих строк, вызов **glDrawElements()** оставляет некоторые переменные в неопределенном состоянии. После исполнения **glDrawElements()** текущие **RGB –** цвет, цветовой индекс, координаты нормали, координаты текстуры и флаг ребра не определены **(**или неизвестны, непредсказуемы**)** если во время вызова были включены соответствующие массивы.

При использовании **glDrawElements()** индексы всех вершин каждой грани куба могут быть помещены в массив индексов. Пример **2-11** демонстрирует два пути использования **glDrawElements()** для визуализации куба. На рисунке **2-16** показана нумерация вершин, используемая в примере **2-11.**

Рисунок **2-16.** Куб с пронумерованными вершинами



Пример **2-11.** Два способа использования **glDrawElements()**

```
GLubyte frontIndices[] = {4,5,6,7};
GLubyte rightIndices[] = {1,2,6,5};
GLubyte bottomIndices[] = {0,1,5,4};
GLubyte backIndices[] = {0,3,2,1};
GLubyte leftIndices[] = {0,4,7,3};
GLubyte topIndices[] = {2,3,7,6};

glDrawElements(GL_QUADS,4,GL_UNSIGNED_BYTE,frontIndices);
glDrawElements(GL_QUADS,4,GL_UNSIGNED_BYTE,rightIndices);
glDrawElements(GL_QUADS,4,GL_UNSIGNED_BYTE,bottomIndices);
glDrawElements(GL_QUADS,4,GL_UNSIGNED_BYTE,backIndices);
glDrawElements(GL_QUADS,4,GL_UNSIGNED_BYTE,leftIndices);
glDrawElements(GL_QUADS,4,GL_UNSIGNED_BYTE,topIndices);
```

Или другой вариант – собираем все индексы вместе:

```
GLubyte allIndices[] ={4,5,6,7,1,2,6,5,
                       0,1,5,4,0,3,2,1,
                       0,4,7,3,2,3,7,6};

glDrawElements(GL_QUADS,24,GL_UNSIGNED_BYTE,allIndices);
```

Замечание: Вызов **glDrawElements()** между **glBegin()** и **glEnd()** является ошибкой.

Как и **glDrawElements()**, команда **glDrawRangeElements()** удобна для случайной выборки данных из массивов и визуализации этих данных. **glDrawRangeElements()** также позволяет определить ограничения на диапазон допустимых значений индексов, что может увеличить быстродействие программы. Для оптимального быстродействия, некоторые реализации **OpenGL** могут еще до визуализации извлекать из вершинных массивов ограниченное количество данных. **glDrawRangeElements()** позволяет указать диапазон вершин, который будет кэширован таким образом.

```
void glDrawRangeElements (GLenum mode, GLuint start, GLuint end, GLsizei count,
GLenum type, void *indices);
```

Создает последовательность геометрических примитивов, так же как и **glDrawElements()**, однако накладывает дополнительные ограничения на значения принимаемых параметров. Часть параметров **glDrawRangeElements()**, включая ***mode*** (тип примитивов), ***count*** (количество элементов), ***type*** (тип данных) и ***indices*** (массив индексов), имеет то же значение, что и аналогичные параметры **glDrawElements()**. В **glDrawRangeElements()** присутствуют еще два параметра: ***start*** и ***end***, которые указывают диапазон допустимых значений для ***indices***. Для того, чтобы быть допустимыми, значения элементов массива **indices** должны лежать в диапазоне между ***start*** и ***end*** включительно.

Указание в массиве ***indices*** значений индексов вне диапазона **[*start*, *end*]** является ошибкой. Однако реализация **OpenGL** не обязательно обнаруживает эту ошибку или рапортует о ней. То есть, недопустимая величина индекса может генерировать ошибку **OpenGL**, а может и нет. Что будет происходить в каждом конкретном случае, зависит от каждой конкретной реализации.

Вы можете использовать **glGetIntegerv()** с параметрами **GL_MAX_ELEMENTS_VERTICES** и **GL_MAX_ELEMENTS_INDICES** для того, чтобы выяснить, соответственно, рекомендуемые максимум количества кэшируемых вершин и максимальное количество индексов (то есть общее количество вершин, которые должны быть визуализированы). Если ***end*-*start*+1** больше, чем рекомендованный максимум кэшируемых вершин, или если ***count*** больше рекомендуемого максимума индексов, **glDrawRangeElements()** должна по-прежнему работать корректно, но быстродействие может быть снижено.

Не является обязательной ссылка на все вершины из диапазона **[*start*, *end*]**. Однако для многих реализаций указание чрезмерно широкого диапазона вызовет ненужную обработку большого числа вершин, на которые нет ссылок в массиве ***indices***.

Возможно, что при использовании **glArrayElements()**, **glDrawElements()** и **glDrawRangeElements()**, ваша реализация **OpenGL** кэширует недавно обработанные (то есть преобразованные, освещенные и так далее) вершины, позволяя вашему приложению повторно использовать их, не передавая их на конвейер визуализации еще раз. Например, означенный выше куб состоит из **6**-ти граней (полигонов) и всего **8**-ти вершин. Каждая вершина используется **3**-мя гранями. Без команд **gl*Elements()** визуализация всех **6**-ти граней потребовала бы обработки **24**-х вершин, несмотря на то, что обработка **16**-ти из них являлась бы избыточной. Ваша реализация **OpenGL** может минимизировать избыточную обработку, визуализировав всего **8** вершин. (Повторное использование вершин может быть ограничено всеми вершинами, указанными в одном вызове **glDrawElements()** или **glDrawRangeElements()**, или в случае **glArrayElement()**, всеми вершинами, указанными внутри одного блока **glBegin()/glEnd()**.)

2.7.3.3 Разрешение последовательности элементов массива

В отличие от команд **glArrayElements()**, **glDrawElements()** и **glDrawRangeElements()**, позволяющих осуществлять случайную выборку элементов вершинных массивов, команда **glDrawArrays()** осуществляет последовательную выборку.

```
void glDrawArrays (GLenum mode, Glint first, GLsizei count);
```

Конструирует последовательность геометрических примитивов, используя элементы включенных вершинных массивов, начиная с индекса ***first*** и заканчивая индексом ***first+count*-1**. Параметр ***mode*** указывает, какие примитивы следует построить, и может принимать те же значения, что и единственный параметр **glBegin()**, например **GL_POLYGON**, **GL_LINE_LOOP**, **GL_LINES**, **GL_POINTS** и так далее.

Эффект от вызова **glDrawArrays()** аналогичен следующей последовательности команд:

```
int i;
glBegin(mode);
   for(i=0;i<COUNT;I++)
      glArrayElement(first+i);
glEnd();
```

Как и **glDrawElements()**, **glDrawArrays()** производит проверку значений своих параметров на допустимость и оставляет текущие **RGB**-цвет, цветовой индекс, координаты нормали, координаты текстуры и флаг ребра в неопределенном состоянии, если соответствующий массив был включен во время ее вызова.

2.7.4 Смешанные массивы

Дополнительно: Ранее мы рассматривали специальный случай смешанного массива. Тогда в массиве ***interwined*** чередовались данные **RGB –**цветов и **3D –** координат вершин. Доступ к ним был получен с помощью команд **glColorPointer()** и **glVertexPointer()** и правильного выбора значений параметра ***stride***.

```
GLfloat interwined[]={1.0, 0.2, 1.0, 100.0, 100.0, 0.0,
                      1.0, 0.2, 0.2, 0.0, 200.0, 0.0,
                      1.0, 1.0, 0.2, 100.0, 300.0, 0.0,
                      0.2, 1.0, 0.2, 200.0, 300.0, 0.0,
```

```
                    0.2, 1.0, 1.0, 300.0, 200.0, 0.0,
                    0.2, 0.2, 1.0, 200.0, 100.0, 0.0};
```

Существует также команда **glInterleavedArrays(),** которая позволяет задавать несколько вершинных массивов за один вызов**.** Кроме того**, glInterleavedArrays()** включает и выключает выбранные массивы**,** таким образом**,** объединяя действия на Шаге **1:** Включение массивов и Шаге **2:** Указание данных для массивов**.** На самом деле массив ***interwined*** отвечает одной из **14** возможных конфигураций данных смешанных массивов**,** поддерживаемых **glInterleavedArrays().** Таким образом**,** для того**,** чтобы использовать данные ***interwined*** в массиве **RGB-**цветов и массиве координат вершин**,** а также для включения обоих этих массивов**,** вызовите**:**

```
glInterleavedArrays(GL_C3F_V3F, 0, interwined);
```

Этот вызов **glInterleavedArrays()** включает массивы **GL_COLOR_ARRAY** и **GL_VERTEX_ARRAY** и выключает массивы **GL_INDEX_ARRAY, GL_TEXTURE_COORD_ARRAY, GL_NORMAL_ARRAY** и **GL_EDGE_FLAG_ARRAY.**

Кроме того**,** эта строка имеет тот же эффект**,** что и вызовы **glColorPointer()** и **glVertexPointer(),** задающие значения для шести вершин в каждом из указанных включаемых массивов**.** После этого можно переходить к Шагу **3,** вызывая **glArrayElement(), glDrawElements(), glDrawRangeElements()** или **glDrawArrays()** для разрешения данных массивов**.**

```
void glInterleavedArrays (GLenum format, GLsizei stride, void *pointer);
```

Инициализирует все шесть вершинных массивов**,** выключая не указанные и включая указанные в параметре ***format*****.** Параметр ***format*** должен быть равен одной из **14** символических констант**,** соответствующих **14** конфигурациям данных**.** Значения ***format*** и их смысл перечислены в таблице **2-5.** Параметр ***stride*** позволяет указать байтовое смещение между данными смежных вершин**.** Если ***stride*** равен **0,** считается**,** что вершинные данные плотно упакованы в массиве**.** Параметр ***pointer*** **–** это адрес памяти**,** где хранится первый элемент данных физического массива**,** относящийся к первой вершине **(**указатель на начало физического массива в памяти**).**

Для реализаций **OpenGL,** поддерживающих расширение мультитекстурирования **glInterleavedArrays()** влияет только на активный элемент текстуры**.**

Заметьте**,** что **glInterleavedArrays()** не поддерживает флаги ребра**.**

Объяснение механики **glInterleavedArrays()** достаточно сложно и требует ссылок на пример **2-12** и таблицу **2-5.** В этих примере и таблице**,** вы увидите несколько переменных**.** Переменные ***et***, ***ec*** и ***en***, содержат булевские значения и представляют соответственно включенное или выключенное состояние массивов текстурных координат**,** цвета и координат нормалей**.** Переменные ***st***, ***sc*** и ***sv*** представляют собой размеры **(**количество компонент**)** одного элемента в массивах координат текстуры**,** цвета и координат вершин соответственно**.** ***tc*** представляет собой тип данных для **RGBA –** цвета**,** поскольку это единственный массив**,** который при применении **glInterleavedArrays()** может содержать недробные значения**.** ***pc***, ***pn*** и ***pv*** представляют собой вычисленные значения смещения **(**в байтах**)** от начала данных в физическом массиве**,** относящихся к определенной вершине**,** до начала значений цвета**,** координат нормали и самих координат той же вершины**,** а ***s*** **--** это вычисленное смещение между началом данных физического массива**,** относящихся к предыдущей вершине и началом данных**,** относящихся к следующей за ней вершиной **(**применяется в том случае**,** если пользователь не указал свое смещение**).**

Эффект от вызова **glInterleavedArrays()** аналогичен эффекту от выполнения последовательности команд в примере **2-12.** Для простоты понимания считайте ***et***, ***ec***, ***en***, ***st***, ***sc***, ***sv***, ***tc***, ***pc***, ***pn***, ***pv*** и ***s*** **–** функциями значений параметра ***format***, перечисленных в таблице **2-5.** Вся векторная арифметика производится в единицах равных **sizeof(GLubyte)** байт **(**теоретически это не обязательно **1** байт**).**

Пример **2-12.** Эффект от вызова **glInterleavedArrays(format, stride, pointer)**

```
int str;

str=stride;
if(str==0)
   str=s;

glDisableClientState(GL_EDGE_FLAG_ARRAY);
glDisableClientState(GL_INDEX_ARRAY);

if (et)
{
   glEnableClientState(GL_TEXTURE_COORD_ARRAY);
   glTexCoordPointer(st,GL_FLOAT,str,pointer);
}
else
   glDisableClientState(GL_TEXTURE_COORD_ARRAY);

if (ec)
{
   glEnableClientState(GL_COLOR_ARRAY);
   glColorPointer(sc, tc, str,pointer+pc);
}
else
   glDisableClientState(GL_COLOR_ARRAY);

if (en)
{
   glEnableClientState(GL_NORMAL_ARRAY);
   glNormalPointer(GL_FLOAT,str,pointer+pn);
}
else
   glDisableClientState(GL_NORMAL_ARRAY);

glEnableClientState(GL_VERTEX_ARRAY);
glVertexPointer(sv, GL_FLOAT,str,pointer+pv);
```

В таблице **2-5, T** и **F** соответствуют **True** и **False. f** равно **sizeof(GLfloat).** c **– sizeof(GLubyte)*4 (**округленному до ближайшего произведения **f,** если необходимо**).**

Таблица **2-5.** Конфигурации данных**,** поддерживаемые командой **glInterleavedArrays()**

Format	et	ec	en	st	sc	sv	tc	pc	pn	pv	s
GL_V2F	F	F	F			2				0	2f
GL_V3F	F	F	F			3				0	3f
GL_C4UB_V2F	F	T	F		4	2	GL_UNSIGNED_BYTE	0		c	c+2f
GL_C4UB_V3F	F	T	F		4	3	GL_UNSIGNED_BYTE	0		c	c+3f
GL_C3F_V3F	F	T	F		3	3	GL_FLOAT	0		3f	6f
GL_N3F_V3F	F	F	T			3			0	3f	6f
GL_C4F_N3F_V3F	F	T	T		4	3	GL_FLOAT	0	4f	7f	10f
GL_T2F_V3F	T	F	F	2		3				2f	5f
GL_T4F_V4F	T	F	F	4		4				4f	8f
GL_T2F_C4UB_V3F	T	T	F	2	4	3	GL_UNSIGNED_BYTE	2f		c+2f	c+5f
GL_T2F_C3F_V3F	T	T	F	2	3	3	GL_FLOAT	2f		5f	8f
GL_T2F_N3F_V3F	T	F	T	2		3			2f	5f	8f
GL_T2F_C4F_N3F_V3F	T	T	T	2	4	3	GL_FLOAT	2f	6f	9f	12f
GL_T4F_C4F_N3F_V4F	T	T	T	4	4	4	GL_FLOAT	4f	8f	11f	15f

Начинайте с использования простых форматов, таких как **GL_V2F**, **GL_V3F** или **GL_C3F_V3F**. Если вы хотите использовать форматы, включающие **C4UB** – часть, возможно, что для упаковки **4** беззнаковых байтов в одно **32**-битовое слово вам придется использовать структурные типы данных, приведение типов или арифметику над указателями.

В некоторых реализациях **OpenGL** использование смешанных массивов может увеличить быстродействие приложения. При использовании смешанных массивов известен точный формат ваших данных. Вы точно знаете, что ваши данные плотно упакованы, и доступ к ним может быть произведен очень быстро. Без использования смешанных массивов, информация о смещениях и размерах должна дополнительно проверяться, чтобы обнаружить, плотно ли данные упакованы в памяти.

Замечание: Команда **glInterleavedArrays()** включает и выключает определенные вершинные массивы и задает данные для включенных ей вершинных массивов. Эта команда ничего не визуализирует и не рисует. Вы все еще должны выполнить Шаг 3: Разрешение данных и вызвать **glArrayElement()**, **glDrawElements()**, **glDrawRangeElements()** или **glDrawArrays()** для разрешения указателей и визуализации графики.

2.8 Группы атрибутов

Ранее вы видели, как запрашивать или устанавливать значения отдельных переменных состояния. Вы также можете сохранять и восстанавливать значения наборов связанных по смыслу переменных состояния с помощью одной команды.

OpenGL объединяет связанные по смыслу переменные состояния в *группы атрибутов*. Например, группа атрибутов **GL_LINE_BIT** состоит из **5** переменных состояния: толщины линии, статуса механизма шаблонирования линии **GL_LINE_STIPPLE**, рисунка шаблона линии, счетчика повтора шаблона и статуса механизма сглаживания линий **GL_LINE_SMOOTH**. С помощью команд **glPushAttrib()** и **glPopAttrib()** вы можете сохранять и восстанавливать значения всех этих пяти переменных состояния одновременно.

Некоторые переменные состояния относятся более чем к одной группе атрибутов. Например, переменная состояния **GL_CULL_FACE** является одновременно частью группы атрибутов полигона и группы включенных атрибутов.

В **OpenGL** версии **1.1** существует два различных стека атрибутов. В добавление к старому стеку **(**в котором сохраняются и из которого восстанавливаются переменные состояния сервера**)**, существует также стек атрибутов клиента, доступ к которому можно получить посредством команд **glPushClientAttrib()** и **glPopClientAttrib()**.

В общем случае, эти команды работают быстрее, чем индивидуальные команды получения переменных состояния. Некоторые величины могут обрабатываться и храниться внутри аппаратуры, и их получение может быть весьма затратным в смысле производительности. Кроме того, если вы работаете на удаленном клиенте, все данные атрибутов для сохранения и восстановления должны быть переданы по сетевому соединению и обратно. Однако **OpenGL** имеет стек атрибутов на самом сервере, что позволяет избежать ненужных сетевых задержек.

Существует около **20** различных групп атрибутов, которые могут быть сохранены и восстановлены командами **glPushAttrib()** и **glPopAttrib()**. Также существует **2** клиентские группы, сохраняемые и восстанавливаемые командами **glPushClientAttrib()** и **glPopClientAttrib()**. И на сервере, и на клиенте атрибуты сохраняются в стеке, глубина которого должна быть достаточной для сохранения как минимум **16** групп атрибутов **(**Реальная глубина стеков в вашей реализации **OpenGL** может быть получена с помощью параметров **GL_MAX_ATTRIB_STACK_DEPTH** и

GL_MAX_CLIENT_ATTRIB_STACK_DEPTH команды **glGetIntegerv().)** Сохранение данных в полный стек или их извлечение из пустого приведет к генерации ошибки.

```
void glPushAttrib (GLbitfield mask);
void glPopAttrib (void);
```

glPushAttrib() сохраняет все атрибуты, указанные битами в параметре **mask,** помещая их в стек атрибутов. **glPopAttrib()** восстанавливает значения тех переменных состояния, которые были сохранены командой **glPushAttrib().** В таблице **2-6** перечислены возможные значения параметра ***mask*** команды **glPushAttrib().** Каждое значения представляет определенный набор переменных состояния. **(**Эти значения могут комбинироваться с помощью логического ИЛИ.**)** Например, **GL_LIGHTING_BIT** объединяет все переменные, связанные с освещением. К ним относятся текущий цвет материала, все параметры света, список включенных источников света, и направления для тех из них, которым они присущи. Когда вызывается команда **glPopAttrib()** все эти переменные восстанавливаются в сохраненные значения.

Специальная маска **GL_ALL_ATTRIB_BITS** используется для сохранения и восстановления всех переменных состояния во всех группах атрибутов.

Таблица **2-6.** Группы атрибутов

Битовая маска	**Группа атрибутов**
GL_ACCUM_BUFFER_BIT	аккумулятор
GL_ALL_ATTRIB_BITS	**---**
GL_COLOR_BUFFER_BIT	цветовой буфер
GL_CURRENT_BIT	текущие
GL_DEPTH_BUFFER_BIT	буфер глубины
GL_ENABLE_BIT	включенные
GL_EVAL_BIT	вычислители
GL_FOG_BIT	туман
GL_HINT_BIT	комплексные установки
GL_LIGHTING_BIT	освещение
GL_LINE_BIT	линия
GL_LIST_BIT	список
GL_PIXEL_MODE_BIT	пиксели
GL_POINT_BIT	точка
GL_POLYGON_BIT	полигон
GL_POLYGON_STIPPLE_BIT	шаблон полигона
GL_SCISSOR_BIT	отрез
GL_STENCIL_BUFFER_BIT	буфер трафарета
GL_TEXTURE_BIT	текстура
GL_TRANSFORM_BIT	трансформация
GL_VIEWPORT_BIT	порт просмотра

```
void glPushClientAttrib (GLbitfield mask);
void glPopClientAttrib (void);
```

glPushClientAttrib() сохраняет все атрибуты, указанные битами в параметре **mask,** помещая их в стек атрибутов на клиентской стороне. **glPopClientAttrib()** восстанавливает значения тех переменных состояния, которые были сохранены последним вызовом **glPushClientAttrib().** В таблице **2-7** перечислены все возможные битовые маски, которые могут быть совмещены в параметре **mask** команды **glPushClientAttrib()** для сохранения любой комбинации клиентских атрибутов. Маска **GL_ALL_CLIENT_ATTRIB_BITS** означает сохранение всех возможных групп атрибутов. Существуют две клиентские группы атрибутов **(**обратного режима и выбора**),** которые не могут быть сохранены или восстановлены с помощью механизма стека.Таблица **2-7.** Клиентские группы атрибутов

Битовая маска	Группа атрибутов
GL_CLIENT_PIXEL_STORE_BIT	режимы хранения пикселей
GL_CLIENT_VERTEX_ARRAY_BIT	вершинные массивы
не может быть сохранена или восстановлена	обратного режима
не может быть сохранена или восстановлена	выбор

2.9 Советы по построению полигональных моделей и поверхностей

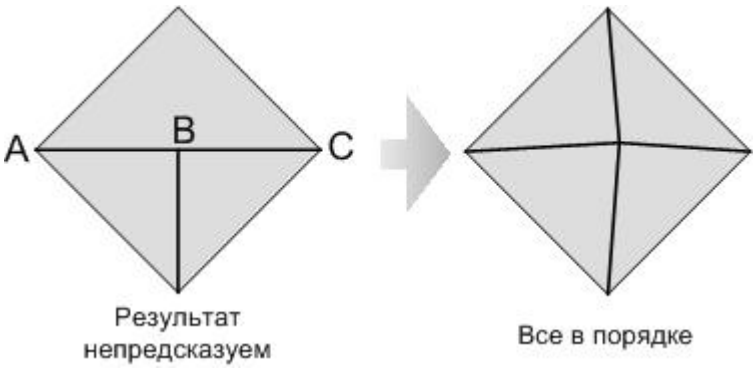
Далее перечислено несколько техник, которые вы можете найти полезными при построении полигональных аппроксимаций поверхностей. параметры освещения влияют на то, как будет выглядеть модель после рисовании, кроме того, некоторые из перечисленных техник намного более эффективны при использовании в соединении со списками отображения. Имейте в виду, что если включен механизм расчетов, связанных с освещением, то для получения ожидаемых результатов тем или иным путем должны быть определены вектора нормалей.

Построение полигональных моделей – это искусство, здесь нет предела опыту. Однако здесь перечислено несколько советов, которые помогут сделать процесс построения моделей намного более простым.

- Сохраняйте ориентацию всех полигонов одинаковой. Убедитесь, что при взгляде со стороны, все полигоны поверхности ориентированы в одном направлении (вершины всех полигонов расположены по часовой стрелке или против нее). Постоянство ориентации важно для отсечения нелицевых граней и двухстороннего освещения. Попытайтесь следовать этому правилу с самого начала, поскольку проблемы, связанные с его несоблюдением, очень сложно исправлять в дальнейшем. (Если вы используете **glScale*()** для отражения геометрии относительно какой-либо оси симметрии, вы можете изменить ориентацию с помощью **glFrontFace()** для сохранения ее постоянства.)
- При делении поверхности отслеживайте любые нетреугольные полигоны. Три вершины треугольника гарантированно лежат в одной плоскости; любой полигон с четырьмя и более вершинами не обязательно является плоским. При просмотре с определенных точек ребра такого полигона могут пересекаться, и **OpenGL** может неправильно его отобразить.
- Помните, что всегда существует противоречие между скоростью графического вывода и качеством изображения. Если вы разделите поверхность на небольшое количество полигоны, они будут отображаться очень быстро, но такая поверхность будет иметь негладкий, угловатый вид; если вы разделите ее на миллионы крошечных полигонов, она будет очень хорошо выглядеть, но ее вывод займет намного больше времени. В идеале вы можете передавать в функцию или процедуру деления поверхности параметр, определяющий насколько качественное дробление вы хотите осуществить. В этом случае, если поверхность находится далеко от наблюдателя, вы можете использовать более грубое разделение поверхности. Кроме того, используйте большие полигоны для относительно плоских частей поверхности и большее количество маленьких полигонов для сильно изогнутых ее областей.
- Для высококачественных изображений неплохой идеей является разделение контура поверхности на большее число фрагментов, чем разделение внутренних частей. Это сложнее сделать, если предполагается, что поверхность будет вращаться, поскольку в этом случае ее видимый контур будет изменяться. Контурные ребра можно выявить с помощью векторов нормалей – в любой точке такого ребра вектор нормали перпендикулярен вектору из этой точки к точке обзора, то есть их скалярное произведение равно **0**. Ваш алгоритм разделения поверхности может быть написан с учетом возможности выбора более частого дробления там, где произведение этих вектором приближается к **0**.
- Попытайтесь избежать **T**-пересечений в ваших моделях (рисунок **2-17**). Нет гарантии, что сегменты **AB** и **BC** будут отображены на тех же пикселях, что и

сегмент **AC.** Иногда это так**,** а иногда **–** нет**,** в зависимости от применяемых трансформаций и ориентации поверхности**.** Это может привести к появлению видимых **«**трещин**»** в поверхности**.**

Рисунок **2-17.** Проблема **T-**пересечения и ее решение



- Если вы строите замкнутую поверхность**,** при замыкании границы убедитесь**,** что используете точно те же значения координат для точек ее начала и конца**,** в противном случае**,** вы можете обнаружить видимые трещины в поверхности из**-**за арифметических округлений**.** Далее приводится неверных пример кода для рисования круга**:**

```
//Не используйте этот код
#define PI 3.14159265

//Количество отрезков, на которое мы подразделяем
//гладкую границу круга
#define EDGES 30

glBegin(GL_LINE_STRIP);
   for(i=0;i<=EDGES;i++)
      glVertex2f(cos((2*PI*i)/EDGES), sin((2*PI*i)/EDGES));
glEnd();
```

В действительности граница круга замкнется только в том случае**,** если ваш компьютер сможет при расчете синуса и косинуса от **0** и **(2*PI*EDGES/EDGES)** получить совершенно одинаковые величины**.** В противном случае начало и конец окружности**,** ограничивающей круг**,** могут не совпасть по координатам и позже по рисуемым пикселям и фигура не будет замкнутой**.** Однако на подобную точность расчетов с плавающей точкой рассчитывать не стоит**.** Для исправления кода убедитесь**,** что когда **i==EDGES**, вы используете для вычисления синуса и косинуса не **(2*PI*EDGES/EDGES)**, а **0. (**Существует более простой способ **–** используйте **GL_LINE_LOOP** вместо **GL_LINE_STRIP** и измените условие завершения цикла на **i<EDGES).**

2.9.1 Пример: Построение Икосаэдра

Для иллюстрации некоторых вопросов**,** возникающих при аппроксимации поверхностей**,** рассмотрим простой пример кода**.** Этот код создает обычный икосаэдр **(**который является Платоновым телом**,** состоящим из **12** вершин и **20** граней**,** каждая из которых является треугольником**).** Икосаэдр можно считать грубым приближением сферы**.** Пример **2-13** задает вершины и треугольники**,** образующие икосаэдр и затем рисует икосаэдр**.**

Пример **2-13.** Рисование икосаэдра

```
#define X .525731112119133606
```

```
#define Z .850650808352039932

GLfloat vdata[12][3] = {
      {-X,0.0,Z},{X,0.0,Z},{-X,0.0,-Z},{X,0.0,-Z},
      {0.0,Z,X},{0.0,Z,-X},{0.0,-Z,X},{0.0,-Z,-X},
      {Z,X,0.0},{-Z,X,0.0},{Z,-X,0.0},{-Z,-X,0.0}
};

GLuint tindices[20][3] = {
      {1,4,0},{4,9,0},{4,5,9},{8,5,4},{1,8,4},
      {1,10,8},{10,3,8},{8,3,5},{3,2,5},{3,7,2},
      {3,10,7},{10,6,7},{6,11,7},{6,0,11},{6,1,0},
      {10,1,6},{11,0,9},{2,11,9},{5,2,9},{11,2,7}
};

int i;

glBegin(GL_TRIANGLES);
      for(i=0;i<20;i++)
{
      //Здесь помещается информация о цвете

      glVertex3fv(&vdata[tindices[i][0]][0]);
      glVertex3fv(&vdata[tindices[i][1]][0]);
      glVertex3fv(&vdata[tindices[i][2]][0]);
}
glEnd();
```

Странные числа **X** и **Z** выбраны таким образом, чтобы расстояние от точки начала координат до любой из вершин было равно **1.0**. Координаты **12**-ти вершин задаются в массиве ***vdata***, где координаты нулевой вершины **{-X,0.0,Z}**, координаты первой **{X,0.0,Z}** и так далее. Массив ***tindices*** определяет, как нужно соединять вершины для получения нужных треугольников. Например, вершинами первого треугольника являются нулевая, четвертая и первая вершины из массива ***vdata***. Если вершины для каждого треугольника будут выбираться согласно указанному порядку, все треугольники будут иметь одинаковую ориентацию.

Строка комментария о цветовой информации должна быть заменена командой, устанавливающей цвет для **i**-ой грани. Если это не будет сделано, все грани будут нарисованы одним цветом, и это приведет к невозможности визуального определения того, что объект является трехмерным. Вместо изменения цветов можно определить вектора нормалей и использовать освещение, как описано в следующем подразделе.

Замечание: Во всех примерах данного раздела, если поверхность должна быть нарисована более одного раза, вероятно, стоит сохранить вычисленные координаты вершин и нормалей, чтобы не производить вычисления каждый раз при рисовании поверхности. Вы можете сделать это посредством собственных структур данных или списков отображения.

2.9.1.1 Расчет нормалей для поверхности

Если предполагается, что поверхность будет освещена, вам необходимо определить вектора нормалей к поверхности. Вектор нормали можно получить, вычислив нормализованное векторное произведение двух векторов на этой поверхности. Для каждой плоской части икосаэдра (то есть для каждого треугольника) все три вершины имеют одинаковый вектор нормали. Таким образом, нормаль должна быть определена единожды для каждого набора из трех вершин. Комментарий в предыдущем примере рисования икосаэдра может быть заменен на код из примера **2-14**.

Пример **2-14**. Генерируем вектора нормалей к поверхности

```
GLfloat d1[3], d2[3], norm[3];
for(j=0;j<3;j++)
{
```

```
d1[j]=vdata[tindices[i][0]][j]-vdata[tindices[i][1]][j];
d2[j]=vdata[tindices[i][1]][j]-vdata[tindices[i][2]][j];
}
normcrossprod(d1,d2,norm);
glNormal3fv(norm);
```

Функция **normcrossprod()** вычисляет нормализованное векторное произведение двух векторов, как показано в примере **2-15.**

Пример **2-15.** Вычисление нормализованного векторного произведения двух векторов

```
void normalize(float v[3])
{
   GLfloat d=sqrt(v[0]*v[0]+v[1]*v[1]+v[2]*v[2]);
   if(d==0.0)
   {
      error("Длина вектора равна 0");
      return;
   }
   v[0]/=d;
   v[1]/=d;
   v[2]/=d;
}

void normcrossprod(float v1[3], float v2[3], float out[3])
{
   out[0]=v1[1]*v2[2]- v1[2]*v2[1];
   out[1]=v1[2]*v2[0]- v1[0]*v2[2];
   out[2]=v1[0]*v2[1]- v1[1]*v2[0];
   normalize(out);
}
```

Если вы используете икосаэдр в качестве приближения равномерно закрашенной сферы, вам, возможно, потребуется использовать нормали, которые перпендикулярны истинной поверхности сферы, а не граням икосаэдра. Вектора нормалей для сферы вычисляются элементарно, поскольку в каждой точке вектор нормали совпадает с вектором из центра сферы в эту точку. Поскольку в приведенном примере икосаэдр имеет радиус равный **1**, данные для нормалей и данные координат вершин совпадают. Далее приводится код, рисующий двадцатистороннюю аппроксимацию (икосаэдр) равномерно закрашенной сферы (в коде предполагается, что кроме прочего включено освещение):

```
glBegin(GL_TRIANGLES);
   for(i=0;i<20;i++)
   {
      glNormal3fv(&vdata[tindices[i][0]][0]);
      glVertex3fv(&vdata[tindices[i][0]][0]);
      glNormal3fv(&vdata[tindices[i][1]][0]);
      glVertex3fv(&vdata[tindices[i][1]][0]);
      glNormal3fv(&vdata[tindices[i][2]][0]);
      glVertex3fv(&vdata[tindices[i][2]][0]);
   }
glEnd();
```

2.9.1.2 Улучшаем модель

20-сторонняя аппроксимация сферы выглядит не лучшим образом, если только она не находится далеко от наблюдателя, однако существует простой способ улучшить качество изображения. Представьте себе, что икосаэдр вписан в сферу (то есть все вершины икосаэдра лежат на поверхности сферы) и разбейте каждый из его треугольников на **4**, как показано на рисунке **2-18** (заметьте, что в правой части рисунка **AD=DB=AB/2**, **BE=EC=DC/2** и **AF=FC=AC/2**).

Замечание: Вообще говоря, это только один из вариантов разбиения треугольников. Например, можно разбить каждый треугольник не на 4 новых, а на 3, добавив новую вершину в точке пересечения биссектрис исходного треугольника или в центре его тяжести.

Появившиеся в результате новые вершины находятся внутри сферы, и мы должны притянуть их к поверхности сферы путем нормализации (делением координат вершин на некоторый коэффициент, в результате чего вектор из центра сферы к этой вершине будет иметь длину равную **1).** Подобный процесс разбиения может быть повторен несколько раз для получения приближения требуемой точности. Объекты на рисунке **2-19** состоят из **20**, **80** и **320** треугольников соответственно.

Пример **2-16** производит одно дополнительное разбиение, создавая **80**-стороннюю сферическую аппроксимацию.

Рисунок **2-18.** Разбиение треугольников

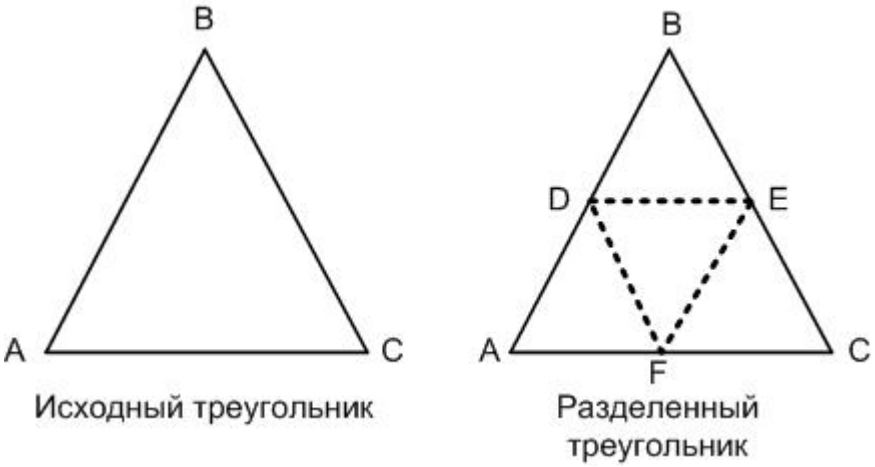
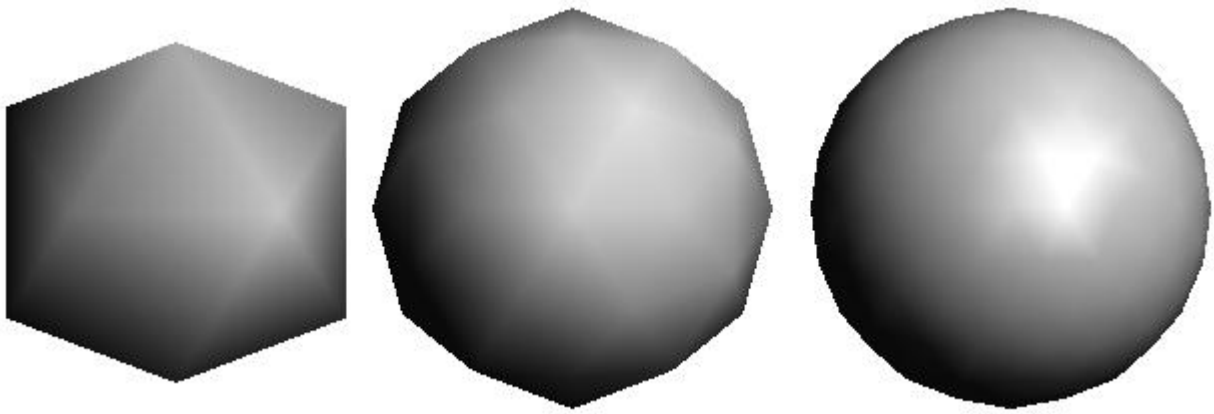


Рисунок **2-19.** Разбиение увеличивает гладкость полигональной модели



Пример **2-16.** Единичное разбиение

```
void drawTriangle(float *v1, float *v2, float *v3)
{
   glBegin(GL_TRIANGLES);
      glNormal3fv(v1);
      glVertex3fv(v1);
      glNormal3fv(v2);
      glVertex3fv(v2);
      glNormal3fv(v3);
      glVertex3fv(v3);
```

```
   glEnd();
}

void subdivine(float *v1, float *v2, float *v3)
{
   GLfloat v12[3], v23[3], v31[3];
   GLint i;

   for (i=0;i<3;i++)
   {
      v12[i]=(v1[i]+v2[i])/2.0;
      v23[i]=(v2[i]+v3[i])/2.0;
      v31[i]=(v3[i]+v1[i])/2.0;
   }
   normalize(v12);
   normalize(v23);
   normalize(v31);
   drawTriangle(v1,v12,v31);
   drawTriangle(v2,v23,v12);
   drawTriangle(v3,v31,v23);
   drawTriangle(v12,v23,v31);
}

for(i=0;i<20;i++)
{
   subdivine(&vdata[tindices[i][0]][0],
      &vdata[tindices[i][1]][0],
      &vdata[tindices[i][2]][0]);
}
```

Пример **2-17 –** это модификация примера **2-16,** осуществляющая рекурсивное разбиение треугольников до заданной глубины**.** Если глубина **(**параметр ***depth*****)** равна **0,** разбиение не производится**,** и треугольники рисуются **«**как есть**».** Если задана глубина равная **1,** разбиение производится **1** раз и так далее**.**

Пример **2-17.** Рекурсивное разбиение

```
void subdivine(float *v1, float *v2, float *v3, long depth)
{
   GLfloat v12[3], v23[3], v31[3];
   GLint i;

   if(depth==0)
   {
      drawTriangle(v1,v2,v3);
      return;
   }

   for (i=0;i<3;i++)
   {
      v12[i]=(v1[i]+v2[i])/2.0;
      v23[i]=(v2[i]+v3[i])/2.0;
      v31[i]=(v3[i]+v1[i])/2.0;
   }
   normalize(v12);
   normalize(v23);
   normalize(v31);
   subdivine(v1,v12,v31,depth-1);
   subdivine(v2,v23,v12,depth-1);
   subdivine(v3,v31,v23,depth-1);
   subdivine(v12,v23,v31,depth-1);
}
```

2.9.1.3 Обобщенное разбиение

Рекурсивное разбиение **(**например**,** то**,** которое приведено в примере **2-17)** обычно заканчивается**,** если достигнут заданный уровень глубины рекурсии**,** или если степень

кривизны поверхности удовлетворяет некоторому критерию **(**особенно кривые части поверхностей выглядят лучше при большем числе разбиений**).**

Рассмотрим более обобщенное решение проблемы разбиения**,** считая**,** что некоторая поверхность параметризирована двумя переменными ***u[0]*** и ***u[1]*****.** Предположим**,** что существует две функции**:**

```
void surf(GLfloat u[2],GLfloat vertex[3],GLfloat normal[3]);
float curv(GLfloat u[2]);
```

При передаче ***u[]*** функции **surf()**, возвращаются соответствующие трехмерные координаты вершины и вектора нормали единичной длины**.** Если передать ***u[]*** в функцию **curv()** вычисляется и возвращается кривизна поверхности в данной точке **(**за дополнительными сведениями об измерении кривизны поверхности обратитесь к литературе по аналитической геометрии**).**

В примере **2-18** представлена рекурсивная функция**,** которая разбивает треугольники до достижения максимального уровня вложенности или до тех пор**,** пока максимальная кривизна в трех вершинах треугольника не станет меньше некоторого заданного значения**.**

Пример **2-18.** Обобщенное разбиение

```
void subdivine(float u1[2],float u2[2],float u3[2],float cutoff,long depth)
{
   GLfloat v1[3],v2[3],v3[3],n1[3],n2[3],n3[3];
   GLfloat u12[2],u23[2],u31[2];
   GLint i;

   if(depth==maxdepth || (curv(u1)<cutoff))
   {
      surf(u1,v1,n1);
      surf(u2,v2,n2);
      surf(u3,v3,n3);
      glBegin(GL_POLYGON);
         glNormal3fv(n1); glVertex3fv(v1);
         glNormal3fv(n2); glVertex3fv(v2);
         glNormal3fv(n3); glVertex3fv(v3);
      glEnd();
      return;
   }

   for(i=0;i<2;i++)
   {
      u12[i]=(u1[i]+u2[i])/2.0;
      u23[i]=(u2[i]+u3[i])/2.0;
      u31[i]=(u3[i]+u1[i])/2.0;
   }
   subdivine(u1,u12,u31,cutoff,depth+1);
   subdivine(u2,u23,u12,cutoff,depth+1);
   subdivine(u3,u31,u23,cutoff,depth+1);
   subdivine(u12,u23,u31,cutoff,depth+1);
}
```

Глава 3. Вид

В этой главе объясняется**,** как проинструктировать **OpenGL** о том**,** каким образом она должна рисовать геометрические модели**,** отображаемые на сцене**.** Вы должны решить**,** как объекты будут расположены**,** и где будет находиться точка обзора**.** Можно оставить позицию и точку обзора по умолчанию**,** но это делается достаточно редко**.**

Главной целью компьютерной графики (или, более точно, машинной графики) является создание двумерного изображения трехмерных объектов (изображение должно быть двумерным, поскольку оно выводится на плоский экран). Однако, при принятии большинства решений, связанных с тем, что и как будет выводиться на экран, следует думать о трехмерных координатах. Обычной человеческой ошибкой при создании трехмерных сцен является то, что они слишком рано начинают думать о двумерной плоскости экрана. Не думайте о пикселях! Вместо этого попытайтесь представить свою сцену в трехмерном пространстве. Представьте, что вы создаете сцену в некотором трехмерном пространстве, находящемся за экраном, внутри компьютера и оставьте компьютеру все вычисления, связанные с тем, какие пиксели подсвечивать.

Трехмерные координаты объекта преобразуются в позиции пикселей на экране серией из трех операций.

- Трансформации (преобразования), представленные перемножением матриц, включают модельные, видовые и проекционные операции. К таким операциям относятся вращение, перенос, масштабирование, отражение, ортографическое и перспективное проецирование. Обычно для отображения сцены используется комбинация из нескольких трансформаций.
- Поскольку сцена отображается в прямоугольном окне, объекты (или их части), находящиеся вне окна должны быть отсечены. В трехмерной компьютерной графике отсечение производится путем отбрасывания объектов с одной стороны отсекающей плоскости.
- Наконец, должно быть установлено соответствие между преобразованными координатами и экранными пикселями. Этот процесс известен, как трансформация порта просмотра.

3.1 Аналогия с фотокамерой

Процесс трансформаций, используемых для создания сцены, аналогичен получению фотографии с помощью камеры. Как показано на рисунке **3-1** этапы для получения фотографии (или сцены на экране монитора) должны быть следующими.

1. Установить треногу и направить камеру на сцену (видовые трансформации).
2. Расположить фотографируемые объекты нужным образом (модельные трансформации).
3. Выбрать линзы для камеры и масштаб (проекционные трансформации).
4. Определить насколько большой должна быть результирующая фотография – например, может понадобиться ее увеличение (трансформация порта просмотра).
5. После произведения этих шагов фотография может быть отпечатана, а изображение сцены – нарисовано.

Рисунок **3-1.** Аналогия с фотокамерой

Заметьте, что нумерация шагов соответствует порядку, в котором вы задаете желаемые трансформации в вашей программе, но не обязательно порядку, в котором над вершинами объектов производятся соответствующие математические вычисления.

В коде программы видовые трансформации должны быть определены до модельных, однако проекцию и порт просмотра вы можете задавать в любой момент до самого

рисования. Рисунок **3-2** демонстрирует порядок, в котором эти операции производятся вашим компьютером.

Рисунок **3-2.** Этапы преобразования вершин

Чтобы задать видовое, модельное или проекционное преобразование, вы заполняете матрицу **M** размерностью **4x4** элемента. Затем эта матрица множится на координаты каждой вершины, в результате чего и происходит преобразование этих координат ***v`*=M*v*.** (Помните, что вершина всегда имеет **4** координаты **(*x*, *y*, *z*, *w*)**, хотя в большинстве случаев ***w=1***, а для двумерных изображений ***z=0*.)** Заметьте, что видовые и модельные преобразования наряду с координатами вершин автоматически применяются к нормалям к поверхности в этих вершинах. **(**Нормали используются только в видовых координатах.**)** Это делается для того, чтобы сохранить связь между вектором нормали и вершинными данными.

Видовые и модельные преобразования, указанные вами, комбинируются в видовой матрице, применяемой к входящим *объектным координатам* для получения *видовых координат*. Если вы задаете дополнительные плоскости отсечения для удаления

определенных объектов со сцены или для создания разрезанного вида объектов – эти плоскости используются на следующем шаге. После этого **OpenGL**применяет матрицу проекции для получения *усеченных координат*. Это преобразование определяет объем видимости; объекты вне этого объема полностью или частично отсекаются (отбрасываются) и не рисуются в финальной сцене. После этого производится перспективное деление, то есть, деление координат ***x***, ***y***, ***z*** на ***w***, для получения *нормализованных координат устройства*. В заключении, преобразованные координаты конвертируются в *оконные координаты* с помощью трансформации порта просмотра. Вы можете манипулировать размерами порта просмотра для растяжения или сжатия финального изображения.

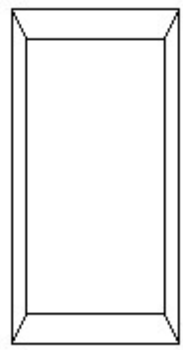
Вы можете правильно предположить, что для определения того, какие пиксели рисовать на экране, достаточно координат ***x*** и ***y***. Однако все преобразования производятся также и над ***z*** – координатой. Таким образом, в конце процесса преобразований величина соответствующей ***z*** – координаты верно отражает глубину каждой вершины (исчисляемую в дистанции от экрана). Одно из применений для значений глубины – избежать ненужного рисования. Например, предположим, что две вершины имеют равные ***x*** и ***y*** – координаты, но разную глубину. **OpenGL** может использовать эту информацию для определения того, какие поверхности перекрыты другими поверхностями и, таким образом, не рисовать невидимые поверхности. Кроме того, координата ***z*** в некоторых проекционных матрицах при перемножении может влиять на результирующие значения координат ***x*** и ***y***.

Для указания всех описанных трансформаций, вы должны знать, как математически производятся операции над матрицами. За дополнительной информацией обратитесь к литературе по линейной алгебре.

3.1.1 Простой пример: рисуем куб

Пример **3-1** рисует куб, который масштабируется модельной трансформацией. Видовая трансформация позиционирует и наводит камеру на то место, где будет нарисован куб. Проекционная трансформация и трансформация порта просмотра также указаны. После примера следуют разделы, разбирающие пример **3-1**. В них дается краткое описание команд трансформаций.

Рисунок **3-3.** Трансформированный куб



Пример **3-1.** Трансформированный куб. Файл **cube.cpp**

```
#include <glut.h>

//Инициализация
void init(void)
{
```

```
    glClearColor(0.0,0.0,0.0,0.0);
    glShadeModel(GL_FLAT);
}

//Отображение
void display(void)
{
    glClear(GL_COLOR_BUFFER_BIT);
    glColor3f(1.0,1.0,1.0);

    //Очистить матрицу
    glLoadIdentity();

    //Видовая трансформация(камера)
    gluLookAt(0.0,0.0,5.0,0.0,0.0,0.0,0.0,1.0,0.0);

    //Модельная трансформация
    glScalef(1.0,2.0,1.0);
    glutWireCube(1.0);
    glFlush();
}

//Изменение размеров окна
void reshape(int w, int h)
{
    glViewport(0,0,(GLsizei) w, (GLsizei) h);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    glFrustum(-1.0,1.0,-1.0,1.0,1.5,20.0);
    glMatrixMode(GL_MODELVIEW);
}

int main(int argc, char** argv)
{
    glutInit(&argc,argv);
    glutInitDisplayMode(GLUT_SINGLE|GLUT_RGB);
    glutInitWindowSize(500,500);
    glutInitWindowPosition(100,100);
    glutCreateWindow("Transformed Cube");
    init();
    glutDisplayFunc(display);
    glutReshapeFunc(reshape);
    glutMainLoop();
    return 0;
}
```

3.1.1.1 Видовая трансформация

Помните о том, что видовая трансформация аналогична позиционированию и наведению камеры. В приведенном примере кода до того как будет определена видовая трансформация, *текущая видовая матрица* должна быть установлена в единичную (четыре элемента на главной диагонали равны **1**, остальные **0**). Этот шаг необходим, поскольку большинство команд преобразований множат текущую матрицу на свою (указываемую) и оставляют результат в качестве новой текущей матрицы. Если вы не очистите текущую матрицу, загрузив в нее единичную, вы продолжите комбинировать матрицы предыдущих преобразований с новыми, определяемыми вами. Иногда, это вам и нужно, но иногда текущую матрицу следует очищать.

В примере **3-1**, после инициализации матрицы, видовая трансформация задается командой **gluLookAt()**. Аргументы этой команды определяют, где находится камера (точка обзора), куда она направлена, и какое направление считать верхним. Значения аргументов, использованные здесь, помещают камеру в точку **(0, 0, 5)**, наводят ее на точку **(0, 0, 0)** и задают вектор направления вверх как **(0, 1, 0)**. Вектор верхнего направления задает ориентацию камеры.

Если бы **gluLookAt()** не была вызвана, камера имела бы позицию и ориентацию по умолчанию. По умолчанию камера расположена в начале координат **(0, 0, 0)**,

направлена вдоль отрицательного направления оси **z**, и вектором верхнего направления имеет **(0, 1, 0)**. Таким образом суммарный эффект от вызова команды **gluLookAt()** в примере **3-1** заключается в перемещении камеры по оси **z**на **5** единиц.

3.1.1.2 Модельная трансформация

Вы используете модельную трансформацию для позиционирования и ориентирования модели. Например, вы можете поворачивать, переносить или масштабировать модель, а также производить комбинации этих операций. В примере **3-1**, использована трансформация **glScalef()**. Аргументами этой команды являются коэффициенты масштабирования модели по трем осям. Если все три коэффициента равны **1.0 –** команда не дает никакого эффекта. В примере **3-1** куб рисуется с удвоенным масштабом по оси **y**. Таким образом, если бы один из углов куба изначально имел координаты **(3.0, 3.0, 3.0)**, он был бы нарисован в точке **(3.0, 6.0, 3.0)**. Эффект от этого преобразования в примере **3-1** трансформирует куб в параллелепипед.

Заметьте, что вместо того, чтобы передвигать камеру, вы могли бы отодвинуть куб от камеры **(**с помощью модельного преобразования**)**. Такая двойственность природы видовых и модельных трансформаций заставляет думать об эффекте обоих типов преобразований одновременно. В общем, не имеет смысла разделять эти эффекты, но иногда проще думать о них тем или иным путем. Эта двойственность также одна из причин, по которой видовые и модельные трансформации объединяются в *видовой матрице* до воздействия на координаты.

Также заметьте, что видовые и модельные трансформации включены в функцию **display()** вместе с вызовом функции **glutWireCube()**, рисующей куб. Таким образом, **display()** может использоваться несколько раз для перерисовки содержимого окна, например, если оно было перемещено, и вы уверены, что каждый раз куб рисуется верно, с применением нужных преобразований. Преимущества повторного использования **display()** перевешивают недостаток, связанный с необходимостью загрузки единичной матрицы до указания преобразований, особенно если иные преобразования могут быть заданы между вызовами **display()**.

3.1.1.3 Проекционная трансформация

Определение проекционной трансформации походит на выбор линз для камеры. Вы можете думать об этом преобразовании как о задающем поле просмотра или объем видимости и, таким образом, определяющем какие объекты лежат внутри этого объема и как они выглядят **(**до некоторой степени**)**. Это эквивалентно выбору между широкоугольными, нормальными или телефото – линзами. С широкоугольными линзами вы можете охватить большее пространство, однако телефото – линзы позволяют фотографировать объекты таким образом, как будто они ближе к вам, чем на самом деле. В компьютерной графике вам нет необходимости платить **2000** долларов за **2000** – миллиметровые телефото - линзы – как только вы включили свою рабочую станцию, все, что вам нужно сделать – это указать меньшее число для объема видимости.

В дополнение к решениям об объеме видимости, проекционная трансформация определяет, **(**как и следует из ее имени**)** как объекты будут *проецироваться* на экран. Два основных типа проекций реализовано в **OpenGL** в явном виде. Вы можете задавать связанные с ними параметры различными способами, применяя несколько специальных команд. Первый из этих типов проекций – перспективная проекция соответствует тому, как вы видите в реальной жизни. Перспектива заставляет удаленные от вас объекты выглядеть меньше. Например, если посмотреть на рельсовую дорогу, кажется, что где-то далеко рельсы сходятся, чего на самом деле конечно не происходит. Если вы пытаетесь создать реалистичное изображение, вероятно, вы захотите использовать именно перспективную проекцию, которая в предыдущем примере кода задается командой **glFrustum()**.

Проекции другого типа – ортографические переносят объекты на экран без воздействия на их относительный размер. Ортографическая проекция используется в архитектурных приложениях и САПРах (Системах Автоматизированного Проектирования), где результирующее изображение должно скорее отражать реальные размеры объектов, а не то, как они выглядят. Архитекторы создают перспективные изображения для демонстрации того, как конкретное здание или интерьер будут выглядеть с разных точек зрения, но они используют ортографические проекции для создания чертежей.

До вызова **glFrustum()**, задающего проекционную трансформацию, должны быть сделаны некоторые приготовления. Как показано в функции **reshape()** примера **3-1**, вначале используется команда **glMatrixMode()** с параметром **GL_PROJECTION**. В результате текущей матрицей становится *проекционная матрица*, и все последующие задаваемые преобразования будут влиять именно на нее. Как вы можете заметить, через несколько строк кода команда **glMatrixMode()** вызывается снова, на этот раз с параметром **GL_MODELVIEW**. Таким образом, сообщается о том, что с этого момента последующие преобразования будут влиять на видовую матрицу, а не на проекционную.

Заметьте, что в коде используется команда **glLoadIdentity()** для инициализации текущей проекционной матрицы. В результате только задаваемая проекционная трансформация будет иметь эффект. Теперь может быть вызвана **glFrustum()** с аргументами, определяющими параметры проекционного преобразования. В предыдущем примере и проекционное преобразование, и преобразование порта просмотра содержатся в функции **reshape()**, которая вызывается, когда окно создается, а также каждый раз при перемещении или изменении размеров окна. Это имеет смысл, поскольку проецирование (соотношение ширина к высоте объема видимости проекции) и воздействие орта просмотра непосредственно связаны с экраном или, более конкретно с отношением размером окна на экране.

3.1.1.4 Трансформация порта просмотра

Вместе проекционное преобразование и преобразование порта просмотра определяет, каким образом изображение будет перенесено на экран компьютера. Проекционное преобразование задает механику того, как будет происходить перенос, преобразование порта просмотра задает форму и размеры доступной области на экране, куда будет перенесено изображение. Поскольку порт просмотра определяет регион, который изображение будет занимать на экране, вы можете думать о трансформации порта просмотра как об определяющей размер и место результирующей фотографии (фотография, например, может быть увеличена или уменьшена).

Аргументы команды **glViewport()** (задающей порт просмотра) задают начальную точку доступного экранного пространства внутри окна (в данном примере **(0, 0)**), а также ширину и высоту доступной области на экране. Все эти величины измеряются в экранных пикселях. Вот почему эта команда должна быть вызвана из **reshape()**: если изменяется размер окна, размер порта просмотра должен также измениться. Заметьте, что ширина и высота задаются с использованием реальной ширины и высоты окна; достаточно часто вам понадобится задавать порт просмотра именно таким образом, вместо того, чтобы применять абсолютные значения.

3.1.1.5 Рисуем сцену

После того, как все необходимые преобразования были заданы, вы можете нарисовать сцену (или снять фотографию). Во время рисования сцены **OpenGL**преобразует каждую вершину каждого объекта на сцене с помощью видовых и модельных преобразований. Далее каждая вершина преобразуется в соответствии с проекционной трансформацией и отсекается, если она лежит вне объема видимости, заданного проекционной трансформацией. Наконец, оставшиеся преобразованные вершины делятся на **w**и накладываются на порт просмотра.

3.1.2 Команды общего назначения для преобразований

В этом разделе обсуждаются некоторые команды **OpenGL**, которые вы можете счесть полезными для указания преобразований. Вы уже видели две из этих команд: **glMatrixMode()** и **glLoadIdentity()**. Две другие, описанные здесь команды – **glLoadMatrix*()** и **glMultMatrix*()** – позволяют задать любую матрицу любого преобразования непосредственно и умножить на нее текущую матрицу. Более специфические команды преобразований, такие как, **gluLookAt()** и **glScale*()**, будут рассмотрены позже.

Как описано в предыдущем разделе, вам следует определиться с тем, какую из матриц вы будете изменять (видовую или проекционную) и лишь затем вызывать команды преобразований. Вы выбираете матрицу с помощью **glMatrixMode()**. Если вы в разных частях кода используете множество преобразований – важно не запутаться с тем, какая матрица выбрана текущей в каждый момент времени. (Команда **glMatrixMode()** может также использоваться для выбора текстурной матрицы в качестве текущей.)

```
void glMatrixMode (GLenum mode);
```

Указывает, какая из матриц, а именно видовая, проекционная или текстурная, будет изменена. Аргумент ***mode*** может принимать значения **GL_MODELVIEW** (видовая матрица), **GL_PROJECTION** (проекционная матрица) или **GL_TEXTURE** (текстурная матрица). Последующие команды преобразований влияют на указанную матрицу. Заметьте, что в каждый момент времени может быть изменена только одна матрица. По умолчанию видовая матрица является текущей (подлежащей изменению) и все три матрицы являются единичными.

Вы используете команду **glLoadIdentity()** для очистки текущей матрицы для будущих преобразований, поскольку эти преобразования изменяют текущую матрицу. В большинстве случаев эта команда всегда вызывается перед указанием проекционных и видовых преобразований, но вам также может понадобиться очистить матрицу перед модельным преобразованием.

```
void glLoadIdentity (void);
```

Устанавливает текущую (подлежащую модификации) матрицу в единичную с размерностью **4 x4**.

Если вы хотите загрузить в качестве текущей некоторую специфическую матрицу (например, чтобы задать какую-либо особенную проекцию), используйте **glLoadMatrix*()**. Похожим образом используйте **glMultMatrix*()** для умножения текущей матрицы на матрицу, переданную в качестве аргумента. Аргументом для обеих этих команд выступает вектор, содержащий **16** величин ($m_1, m_2, \ldots, m_{16}$), задающих матрицу **M** следующим образом:

$$[M] = \begin{bmatrix} m_1 & m_5 & m_9 & m_{13} \\ m_2 & m_6 & m_{10} & m_{14} \\ m_3 & m_7 & m_{11} & m_{15} \\ m_4 & m_8 & m_{12} & m_{16} \end{bmatrix}.$$

Помните, что вы можете повысить эффективность, применяя дисплейные списки (списки отображения) для наиболее часто применяемых матриц, вместо того, чтобы пересчитывать их каждый раз. (Часто **OpenGL**реализации должны вычислять обратную

матрицу видовой матрицы для корректного преобразования нормалей и отсекающих плоскостей в видовые координаты.)

Замечание: Если вы программируете на C и объявляете матрицу как *m[4][4]*, то элемент *m[i][j]* будет соответствовать элементу в *i*-ом столбце и *j*-ой строке матрицы преобразования OpenGL. Однако это не совсем обычно для языка C, поскольку по соглашению *m[i][j]* соответствует элементу в *i*-ой строке и *j*-ом столбце. Иными словами в Cпринято хранить матрицы по строкам, но OpenGLпри получении подобного указателя будет трактовать его как матрицу, хранимую по столбцам. Во избежание ошибок объявляйте матрицы в виде *m[16]*.

`void` **`glLoadMatrix`**`{fd} (const TYPE *m);`Умножает текущую матрицу на матрицу, заданную **16** величинами вектора ***m*** и сохраняет результат в качестве текущей матрицы.

Устанавливает **16** величин текущей матрицы в значения, содержащиеся в ***m***.

`void` **`glMultMatrix`**`{fd} (const TYPE *m);`Умножает текущую матрицу на матрицу, заданную **16** величинами вектора ***m*** и сохраняет результат в качестве текущей матрицы.

Умножает текущую матрицу на матрицу, заданную **16** величинами вектора ***m*** и сохраняет результат в качестве текущей матрицы.

Замечание: Все перемножения матриц в OpenGL производятся следующим образом. Обозначим текущую матрицу как **C**, а матрицу, заданную командой **glMultMatrix*()** или любой командой преобразования как **M**. После умножения результирующей матрицей всегда является **CM**. Поскольку в общем случае произведение матриц не коммутативно, порядок перемножения матриц имеет большое значение.

3.2 Видовые и модельные преобразования

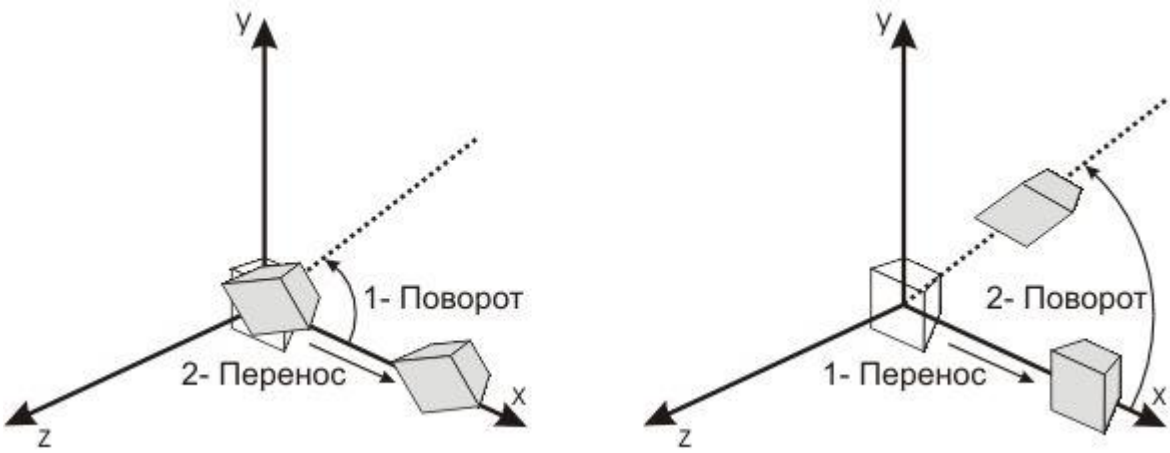
Видовые и модельные преобразования в **OpenGL**сильно связаны и, более того, объединены в одной видовой матрице. Одна из самых сложных проблем, с которой сталкиваются новички в компьютерной графике, состоит в понимании эффекта нескольких комбинированных трехмерных преобразований. Как вы уже видели, существуют альтернативы в понимании преобразований – вы можете передвинуть камеру в одном направлении или объект – в обратном. Каждый из способов понимания преобразований имеет свои недостатки и достоинства, но в определенных случаях один из них может более естественно соответствовать эффекту от произведенной трансформации, чем другой. Если вы найдете естественный подход для своего конкретного приложения, вам будет намного проще визуализировать необходимые трансформации и затем написать соответствующий код для манипуляций с матрицами. Первая часть этого раздела посвящена того, как следует понимать преобразования. Затем рассмотрим конкретные команды. Пока что мы будем использовать только команды матричных манипуляций, уже виденные вами. Помните о том, что вы должны вызвать команду **glMatrixMode()** с аргументом **GL_MODELVIEW**до того, как производить модельные или видовые преобразования.

3.2.1 Понимание преобразований

Начнем с простого случая включающего всего два преобразования: поворот против часовой стрелки на **45** градусов с центром в начале координат вдоль оси ***z*** и перенос

вдоль оси ***x*** в ее положительном направлении. Предположим, что объект достаточно мал по сравнению с преобразованием (то есть мы можем увидеть результат преобразования) и, что изначально он находится в начале координат. Если вы сначала повернете объект, а затем перенесете его, он в результате окажется на оси ***x***. Если же вы сначала перенесете его по оси ***x***, а затем повернете вокруг начала координат, объект окажется на линии ***y=x***, как показано на рисунке **3-4**. В общем случае порядок преобразования является весьма критичным. Если вы выполняете преобразование **A**, а затем – преобразование **B**, вы практически во всех случаях получите результат отличный от того, какой получили бы, производя преобразования в обратном порядке.

Рисунок **3-4**. Сначала поворачиваем или сначала переносим



Теперь поговорим о порядке, в котором вы задаете серии преобразований. Все видовые и модельные преобразования представлены в виде матриц размерностью **4x4**. Каждый успешный вызов команды **glMultMatrix*()** или другой команды преобразования умножает текущую видовую матрицу **C** на новую (заданную вами явно или неявно) матрицу **M**, в результате чего текущей матрицей становится **CM**. В итоге текущая видовая матрица умножается на передаваемые по конвейеру вершины **v**. Этот процесс означает, что последняя команда преобразования, вызванная в вашей программе, на самом деле будет первой применена к вершинам: **CMv**. Таким образом, можно смотреть на это как на необходимость задавать матрицы преобразований в обратном порядке. Как и во многих других случаях, однако, как только вы научитесь правильно думать об этом, обратный порядок начнет казаться прямым.

Рассмотрим следующий фрагмент кода, рисующий точку с учетом трех преобразований.

```
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
glMultMatrixf(N);    //Применить преобразование N
glMultMatrixf(M);    //Применить преобразование M
glMultMatrixf(L);    //Применить преобразование L
glBegin(GL_POINTS);
   glVertex3fv(v);   //Нарисовать преобразованную вершину v
glEnd();
```

В этом коде видовая матрица последовательно содержит матрицы **I** (единичную), **N**, **NM** и **NML**. Преобразованные вершины будут равны **NMLv**. Таким образом, процесс трансформации вершин можно записать в виде **N(M(Lv))** – то есть сначала **L** умножается на **v**, затем **M** – на **Lv** и, наконец, **N** – на **MLv**. Заметьте, что преобразования действительно применяются к вершинам в порядке обратном к тому, в котором вы их задавали. (На самом деле вершина участвует лишь в одном перемножении, так как матрицы отдельных преобразований – в данном примере **N**, **M** и **L**–предварительно перемножены между собой и получена одна результирующая матрица вида.)

3.2.1.1 Фиксированная система координат

Таким образом, если вам нравится думать о преобразованиях в терминах фиксированной координатной системы – в которой перемножение матриц влияет на позицию, ориентацию и масштабирование вашей модели – вы должны думать о перемножениях как о происходящих в обратном порядке относительно того, в котором они появляются в коде. Если бы вы хотели, чтобы после всех операций объект появился на оси ***x*** в простом примере, показанном в левой части рисунка **3-4** (вращение вокруг начала координат и перенос по оси ***x***), первым следовало бы произойти вращение, а затем перенос. Однако, чтобы это сделать, вам следует изменить порядок операций на обратный. В результате код должен выглядеть примерно следующим образом (в нем **R** обозначает матрицу поворота, а **T** – матрицу переноса):

```
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
glMultMatrixf(T);   //Перенос
glMultMatrixf(R);   //Поворот
нарисовать_объект();
```

3.2.1.2 Изменение локальной системы координат

Другой способ рассмотрения матричных перемножений заключается в том, чтобы забыть о фиксированной системе координат, в которой трансформируется модель и вместо этого представить себе локальную систему координат, привязанную к рисуемому объекту. Все операции происходят относительно этой изменяющейся системы координат. При таком подходе оказывается, что матричные преобразования появляются в коде в своем естественном порядке. (Независимо от точки зрения код всегда остается тем же самым, но изменяется то, как вы его осознаете.) Рассматривая пример на рисунке **3-4**, представьте себе координатную систему, привязанную к объекту. Операция переноса сдвигает объект и его координатную систему по оси ***x***. Затем поворот совершается вокруг теперь уже перенесенного начала координат. Таким образом, объект поворачивается на месте в своей позиции на оси.

Такой подход следует использовать для приложений, рисующих, например, руку робота, в которой присутствуют точки соединения объектов на плече, локте, запястье и на каждом из пальцев. Для выяснения того, где должны быть кончики пальцев относительно тела, следует начать с плеча, опуститься до запястья и далее, задавая необходимые повороты и переносы в каждой точке соединения. Если думать об этом процессе в терминах фиксированной системы координат и обратного порядка преобразований, можно очень быстро запутаться.

Этот второй подход, однако, может вызвать проблемы в тех случаях, когда применяется масштабирование, особенно, если оно неравномерное (то есть по разным осям используются разные коэффициенты растяжения/сжатия). После равномерного масштабирования с коэффициентом растяжения/сжатия ***k***перенос на ***(a, b, c)*** сдвигает вершину на расстояния ***(a*k,b*k,c*k)*** поскольку координатная система растянута или сжата. Неравномерное масштабирование вкупе с поворотами может вообще сделать оси локальной координатной системы неперпендикулярными друг другу.

Как было отмечено выше, обычно вы в своей программе задаете видовые трансформации перед всеми модельными. В этом случае вершина модели сначала преобразуется к выбранной ориентации, а затем преобразуется видовой операцией. Поскольку матричные перемножения должны быть заданы в обратном порядке, видовые команды должны быть заданы первыми. Заметьте, однако, что вы не обязаны задавать видовые или модельные преобразования, если вы удовлетворены тем, что определено по умолчанию. Если видовое преобразование не задано, то «камера» по умолчанию находится в начале координат и направлена вдоль отрицательного

направления оси ***z***. Если не заданы модельные трансформации – модель не двигается и сохраняет свою позицию, ориентацию и размер.

Поскольку команды для выполнения модельных преобразований могут быть использованы для выполнения видовых преобразований, модельные преобразования *рассматриваются* первыми, несмотря на то, что видовые преобразования должны первыми *задаваться*. Данный порядок рассмотрения также соответствует тому, как думают многие программисты при планировании своего кода. Часто сначала они пишут весь код, необходимый для композиции сцены, включая команды преобразований для верного расположения и ориентирования объектов сцены относительно друг друга. Далее они решают, где они хотят поместить точку наблюдения и пишут соответствующий код видового преобразования.

3.2.2 Модельные трансформации

Существует три команды **OpenGL** для модельных преобразований: **glTranslate*()**, **glRotate*()** и **glScale*()**. Как вы можете предположить, эти команды трансформируют объект (или координатную систему – в зависимости от того, как вы предпочитаете думать об этом), перенося, поворачивая, увеличивая, уменьшая или отражая его. Все эти команды эквивалентны созданию соответствующей матрицы переноса, поворота или масштабирования с последующим вызовом **glMultMatrix*()** с этой матрицей в качестве аргумента. Однако использование этих трех команд может быть быстрее, чем **glMultMatrix*() – OpenGL** автоматически вычисляет для вас нужные матрицы.

В следующем описании команд каждая матричная манипуляция рассматривается в терминах того, что она делает с вершинами геометрического объекта (при использовании подхода с фиксированной системой координат) и в терминах того, что она делает с локальной системой координат, привязанной к объекту (при использовании этого подхода).

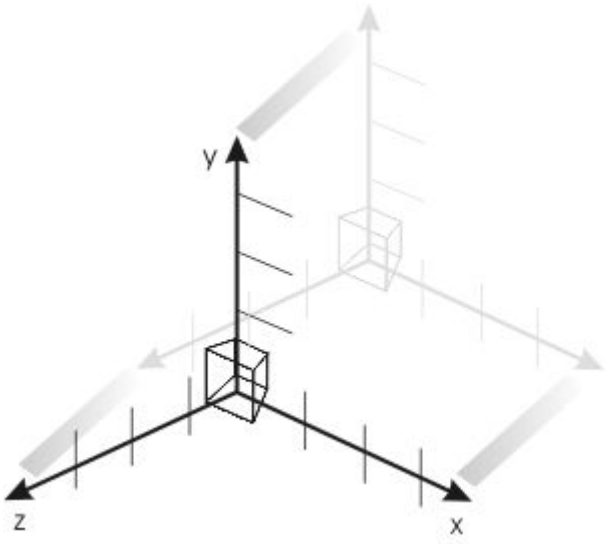
3.2.2.1 Перенос

```
void glTranslate{fd} (TYPE x, TYPE y, TYPE z);
```

Умножает текущую матрицу на матрицу, передвигающую (переносящую) объект на расстояния ***x***, ***y***, ***z***, переданные в качестве аргументов команды, по соответствующим осям (или перемещает локальную координатную систему на те же расстояния).

На рисунке **3-5** изображен эффект команды **glTranslate*()**.

Рисунок **3-5.** Перенос объекта



Обратите внимание на то, что использование **(*0.0*, *0.0*, *0.0*)** в качестве аргумента **glTranslate*()** – это единичная операция, то есть она не влияет на объект или на его координатную систему.

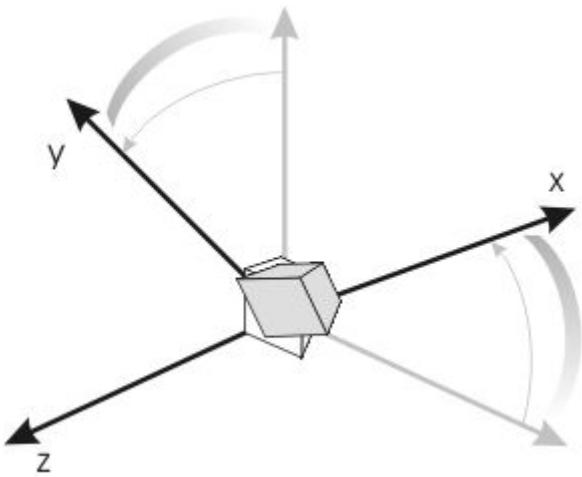
3.2.2.2 Поворот

```
void glRotate{fd} (TYPE angle, TYPE x, TYPE y, TYPE z);
```

Умножает текущую матрицу на матрицу, которая поворачивает объект (или локальную координатную систему) в направлении против часовой стрелки вокруг луча из начала координат, проходящего через точку **(*x*, *y*, *z*)**. Параметр ***angle***задает угол поворота в градусах.

Результат выполнения **glRotatef(*45.0*, *0.0*,*0.0*,*1.0*)**, то есть поворот на **45** градусов вокруг оси ***z***, показан на рисунке **3-6**.

Рисунок **3-6.** Поворот объекта



Заметьте, что чем дальше объект от оси вращения, тем больше орбита его поворота и тем заметнее сам поворот. Также обратите внимание на то, что вызов **glRotate*()** с параметром ***angle*** равным **0** не имеет никакого эффекта.

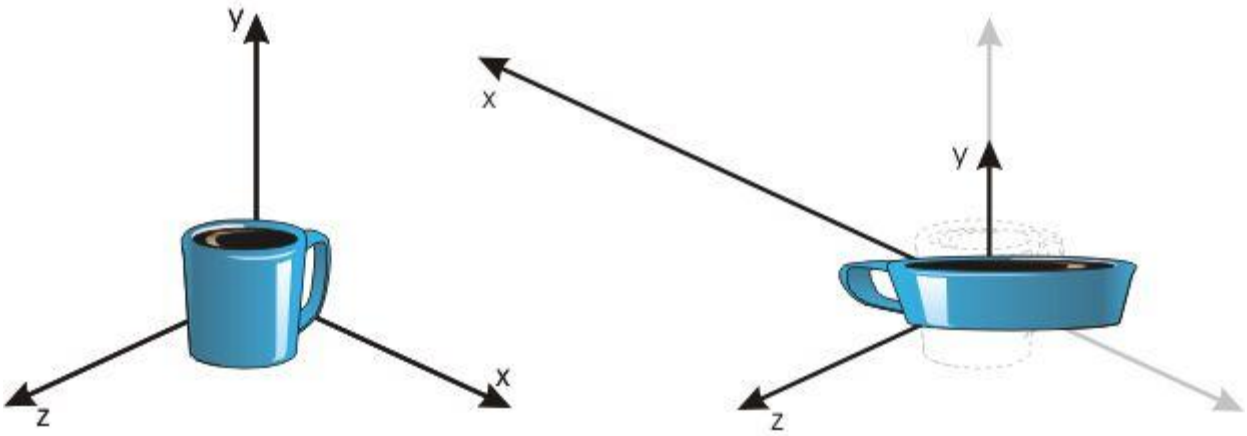
3.2.2.3 Масштабирование

```
void glScale{fd} (TYPE x, TYPE y, TYPE z);
```

Умножает текущую матрицу на матрицу, которая растягивает, сжимает или отражает объект вдоль координатный осей. Каждая ***x*** **-**, ***y*****-** и ***z*****-** координата каждой точки объекта будет умножена на соответствующий аргумент ***x***, ***y***или ***z***команды **glScale*()**. При рассмотрении преобразования с точки зрения локальной координатной системы, оси этой системы растягиваются, сжимаются или отражаются с учетом факторов ***x***, ***y*** и ***z***, и ассоциированный с этот системой объект меняется вместе с ней.

На рисунке **3-7** показан эффект команды **glScalef*(-2.0*,*0.5*,*1.0*)**;

Рисунок **3-7.** Масштабирование и отражение объекта



glScale*() – это единственная из трех команд модельных преобразований, изменяющая размер объекта: масштабирование с величинами более ***1.0*** растягивает объект, использование величин меньше ***1.0*** сжимает его. Масштабирование с величиной ***-1.0*** отражает объект относительно оси или осей. Единичными аргументами (то есть аргументами, не имеющими эффекта) являются (***1.0***, ***1.0***, ***1.0***). Вообще следует ограничивать использование **glScale*()** теми случаями, когда это действительно необходимо. Использование **glScale*()** снижает быстродействие расчетов освещенности, так как вектора нормалей должны быть нормализованы заново после преобразования.

Замечание: Величина масштабирования равная 0 приводит к коллапсу всех координат объекта по оси или осям до 0. Обычно это не является хорошей идеей, так как такая операция не может быть обращена. Говоря математически, матрица не может быть обращена, а обратные матрицы необходимы для многих расчетов, связанных с освещением. Иногда коллапс координат все же имеет смысл: расчет теней на плоской поверхности – это типичный пример применения коллапса. В общем, если координатная система должна быть подвергнута коллапсу, следует использовать для этого проекционную матрицу, а не видовую.

3.2.2.4 Пример кода с модельными трансформациями

Пример **3-2** – это фрагмент программы, которая рисует треугольник **4** раза:

- Сплошной треугольник в виде контура без модельных преобразований.
- Тот же треугольник, нарисованный штриховой линией и перенесенный по оси **x**.
- Треугольник, нарисованный линией с длинным штрихом, растянутый в **1.5** раза по оси **x** и сжатый в **1.5** раза по оси **y**.
- Повернутый треугольник, нарисованный пунктирной линией.

Пример **3-2.** Использование модельных преобразований

```
glLoadIdentity();
glColor3f(1.0,1.0,1.0);
draw_triangle();

glEnable(GL_LINE_STIPPLE);
glLineStipple(1,0xF0F0);
glLoadIdentity();
glTranslatef(-20.0,0.0,0.0);
draw_triangle();

glLineStipple(1,0xF00F);
glLoadIdentity();
glScalef(1.5,0.5,1.0);
draw_triangle();

glLineStipple(1,0x8888);
glLoadIdentity();
glRotatef(90.0,0.0,0.0,1.0);
draw_triangle();
glDisable(GL_LINE_STIPPLE);
```

Обратите внимание на использование **glLoadIdentity()** для изоляции эффектов модельных преобразований; инициализация матрицы единичными значениями предотвращает кумулятивный эффект последующих преобразований. Несмотря на то, что **glLoadIdentity()** позволяет добиться нужного результата, ее множественные вызовы могут быть неэффективны, так как после каждого из них вам необходимо заново задавать видовые и модельные преобразования. Для получения информации о более эффективном способе изоляции преобразований смотрите раздел «Манипулирование матричными стеками» далее в этой главе.

Замечание: Иногда программисты, пытаются заставить объект вращаться, комбинируя на каждом шаге текущую матрицу с матрицей вращения на небольшой угол (то есть как раз и используя кумулятивный эффект). Проблема в том, что из-за ошибок округления комбинация тысяч небольших поворотов все дальше уходит от реально желаемой величины поворота (этот процесс может даже привести к чему-либо отличному от вращения вообще). Вместо этой техники, на каждом шаге увеличивайте угол, инициализируйте видовую матрицу и исполняйте новую команду поворота.

3.2.3 Видовые трансформации

Видовое преобразование изменяет позицию и ориентацию точки обзора. Если вы вспомните аналогию с камерой, видовое преобразование устанавливает треногу для камеры и направляет камеру на модель. Видовое преобразование аналогично передвижению и поворотам камеры также обычно состоит из переносов и поворотов. Помните также, что для достижения определенной композиции в результирующем изображении или фотографии вы можете либо перемещать камеру, либо перемещать все объекты сцены в противоположном направлении. Таким образом, модельное преобразование, поворачивающее объекты сцены против часовой стрелки аналогично видовому преобразованию, которое поворачивает камеру по часовой стрелке. И, наконец, имейте в виду, что команды видового преобразования должны вызываться перед всеми командами модельных преобразований, чтобы модельные преобразования были применены к объектам первыми.

Вы можете производить видовые преобразования любым из нескольких способов, описанных далее. Вы также можете использовать расположение и ориентацию точки обзора по умолчанию, то есть ее расположение в начале координат и направление просмотра вдоль отрицательного направления оси ***z***.

- Используйте одну или несколько команд модельных преобразований **(glTranslate*()** или **glRotate*())**. Вы можете думать об эффекте этих

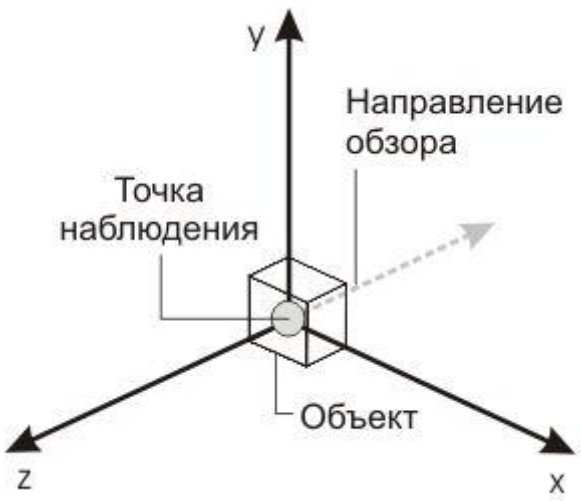
преобразований как перемещении камеры, или как о перемещении всех объектов сцены относительно стационарной камеры.

- Используйте команду библиотеки утилит **gluLookAt()** для определения точки и направления обзора. Эта команда инкапсулирует в себе серию поворотов и переносов.
- Создайте свою собственную (пользовательскую) функцию, инкапсулирующую повороты и переносы. В некоторых приложениях такая функция может понадобиться для того, чтобы можно было задавать видовую трансформацию каким-либо специфическим путем. Например, вам может понадобиться задавать преобразование в терминах полярных координат для камеры, вращающейся вокруг объекта или в терминах углов наклона самолета в полете.

3.2.3.1 Использование glTranslate*() и glRotate*()

Когда вы используете команды модельных преобразований для эмуляции видовых, вы пытаете передвинуть точку наблюдения в желаемом направлении, сохранив объекты неподвижными. Поскольку точка наблюдения изначально находится в начала координат, и поскольку объекты обычно конструируются там же (рисунок **3-8**), вам, в общем, необходимо выполнить некоторые преобразования, дабы объекты были видны. Заметьте, что изначально камера указывает в отрицательном направлении оси ***z***.

Рисунок **3-8**. Объект и точка обзора в начале координат

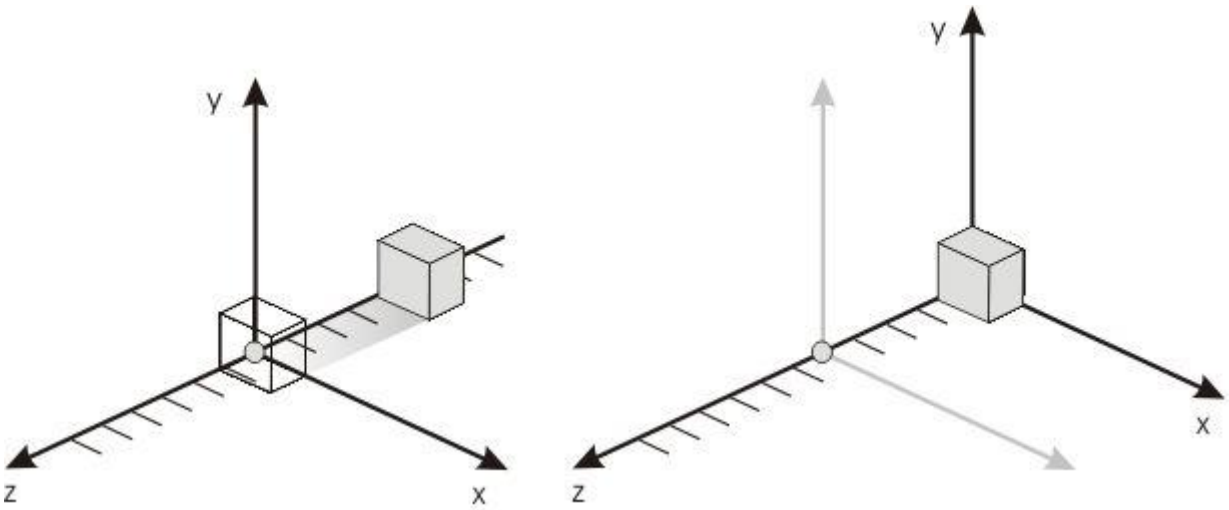


В простейшем случае вы можете передвинуть точку наблюдения назад, от объектов; эффект будет такой же, как если бы вы передвинули объекты вперед от точки наблюдения. Помните, что по умолчанию «вперед» – это значит в отрицательном направлении оси ***z***; если вы повернете точку наблюдения, «вперед» будет иметь другой смысл. Таким образом, чтобы поместить **5** единиц дистанции между точкой наблюдения и объектами, переместив точку наблюдения (как показано на рисунке **3-9**), используйте следующую команду:

```
glTranslatef(0.0,0.0,-5.0);
```

Эта команда передвигает объекты сцены на **-5** единиц вдоль оси ***z***. Она также эквивалентна передвижению камеры на **+5** единиц вдоль оси ***z***.

Рисунок **3-9**. Разделение точки наблюдения и объекта



Теперь предположим, что вы хотите наблюдать объекты со стороны. Должны ли вы в этом случае выполнить команду поворота до или после команды переноса? Если вы мыслите в терминах фиксированной системы координат, для начала представьте объекты и камеру в начале координат. Сначала вы должны повернуть объекты, а затем отодвинуть их от камеры, чтобы выбранная сторона была видна. Поскольку вы знаете, что при подходе с фиксированной системой координат команды должны вызываться в обратном порядке – в том, в котором они будут иметь эффект, вы знаете, что сначала должны написать команду переноса, а затем – поворота.

Теперь используем подход с локальной системой координат. В этом случае думайте о передвижении объекта и его локальной системы координат от начала координат, а затем о повороте с использованием теперь уже перенесенной системы координат. При таком подходе команды вызываются в том же порядке, в котором они применяются, так что перенос снова будет первым, а поворот – вторым. Таким образом, последовательность команд преобразований для получения желаемого результата должна быть следующей:

```
glTranslatef(0.0,0.0,-5.0);
glRotatef(90.0,0.0,1.0,0.0);
```

Если у вас возникают проблемы с понимаем матричных манипуляций, попробуйте разобрать оба подхода с фиксированной и локальной системами координат и оценить, имеет ли один из них смысл. Заметьте, что в фиксированной системе координат вращение всегда происходит вокруг фиксированного начала координат, а в локальной – вокруг изменяющегося начала координат. Вы также можете попробовать использовать команду библиотеки утилит **gluLookAt()**, которая описана далее.

3.2.3.2 Использование команды библиотеки утилит gluLookAt()

Часто программисты конструируют сцену в районе начала координат или в некотором другом месте, а затем хотят посмотреть на нее с определенной точки обзора для получения лучшего вида. Как и говорит ее имя, команда из библиотеки утилит **gluLookAt()** разработана как раз для подобных целей. Она принимает три набора аргументов, которые задают точку наблюдения, прицельную точку (точку, на которую направлена камера) и направление, которое следует считать верхним. Выберите точку обзора, чтобы получить желаемый вид сцены. Прицельная точка, как правило, находится где-то в середине сцены. (Если вы строите сцену в начале координат, то, вероятно, оно и будет прицельной точкой.) Несколько сложнее, видимо, задать верный вектор верхнего направления. Если вы строите сцену в или около начала координат и считаете, что положительное направление оси **y**указывает вверх, то это и есть ваш вектор верхнего направления для **gluLookAt()**. Однако если вы разрабатываете

симулятор полетов, верхним является направление перпендикулярное крыльям самолета.

Команда **gluLookAt()** может быть полезна, например, если вы хотите скользить по ландшафту. С объемом видимости симметричным по ***x***и ***y*** точка **(*eyex*, *eyey*, *eyez*)** может всегда задаваться как лежащая в центре изображения, и вы можете исполнять серию команд для незначительного изменения этой точки, таким образом, скользя по сцене.

```
void gluLookAt (GLdouble eyex, GLdouble eyey, GLdouble eyez, GLdouble centerx,
GLdouble centery, GLdouble centerz,
                GLdouble upx, GLdouble upy, GLdouble upz);
```

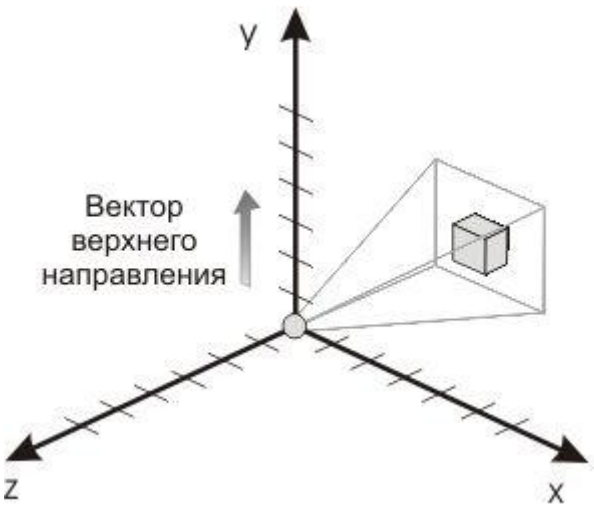
Задает видовую матрицу и умножает на нее текущую матрицу. Выбранная точка обзора задается аргументами ***eyex***, ***eyey*** и ***eyez***. Аргументы ***centerx***, ***centery***и ***centerz***задают любую точку на линии обзора, но обычно они задают точку где-то в середине обозреваемой сцены. Аргументы ***upz***, ***upy***и ***upz*** определяют, какое направление считается верхним (то есть направление от дна до вершины объема видимости).

По умолчанию камера находится в начале координат, направлена вдоль отрицательного направления оси ***z***, а вектор верхнего направления совпадает с положительным направлением оси ***y***. Таким образом, следующий вызов восстанавливает ситуацию по умолчанию:

```
gluLookAt(0.0,0.0,0.0,0.0,0.0,-100.0,0.0,1.0,0.0);
```

Величина ***z***-координаты прицельной точки здесь равна ***-100.0***, но на самом деле она может быть любой отрицательной величиной, поскольку в этом случае направление обзора останется неизменным. Для описанного случая нет необходимости вызывать **gluLookAt()**, поскольку это – установка по умолчанию (рисунок **3-10**). (Линии из точки наблюдения представляют собой объем видимости, задающий видимое пространство.)

Рисунок **3-10**. Позиция точки наблюдения по умолчанию



Заметьте, что **gluLookAt()** является частью библиотеки утилит, а не базовой командой **OpenGL**. Это произошло не потому, что **gluLookAt()** бесполезна, а потому, что она инкапсулирует несколько базовых команд **OpenGL**, а именно **glTranslate*()** и **glRotate*()**. Чтобы понять это представьте, что камера находится в выбранной точке обзора и направлена в соответствии с желаемым направлением обзора, как задано **gluLookAt()** и сцена находится в начале координат. Чтобы отменить действия **gluLookAt()** вам требуется поместить камеру в начале координат и установить направление обзора совпадающим с отрицательным направлением оси ***z*** (то есть

привести видовое преобразование к ситуации по умолчанию). Простой перенос передвинет камеру в начало координат. Вы легко можете представить себе серию поворотов вокруг осей фиксированной системы координат, которые в итоге ориентируют камеру в отрицательном направлении оси ***z***. Поскольку **OpenGL**позволяет задавать повороты вокруг любой выбранной оси, вы можете выполнить требуемый поворот с помощью всего одной команды **glRotate*()**.

Замечание: Вы можете иметь только одно активное видовое преобразование. Вы не можете комбинировать эффекты двух видовых преобразований (фотокамера не может быть установлена на двух треногах одновременно). Если вы хотите изменить позицию камеры, убедитесь, что вы вызвали **glLoadIdentity()** для стирания эффекта от текущего видового преобразования.

Дополнительно: Чтобы трансформировать любой вектор до совпадения (по направлению) с другим вектором (например, с отрицательным направлением оси ***z***) требуется произвести некоторые математические расчеты. Ось, вокруг которой следует производить вращение, получается как векторное произведение двух нормализованных векторов. Чтобы найти угол вращения нормализуйте оба данных вектора. Косинус нужного угла между векторами эквивалентен скалярному произведению между нормализованными векторами. Угол поворота вокруг оси, полученный как векторное произведение всегда находится между **0** и **180** градусами.

Заметьте, что вычисление угла между двумя нормализованными векторами посредством взятия арккосинуса их скалярного произведения дает весьма неточные результаты, особенно при малых углах. Однако этот метод работает.

3.2.3.3 Создание пользовательской функции

Дополнительно: Для некоторых специализированных приложений может понадобиться собственная функция преобразования. Поскольку это делается достаточно редко и, в любом случае, это достаточно сложный, не шаблонный процесс, он оставляется читателю в качестве упражнения. Далее приведены два примера таких функций.

Предположим, что вы разрабатываете симулятор полетов, и вам требуется изображать мир с точки зрения пилота в кабине самолета. Самолет находится в точке с координатами (***x***, ***y***, ***z***). Предположим также, что самолет характеризуется тремя углами своего наклона относительно центра своей гравитации – ***roll***, ***pitch*** и ***heading***. Для изображения мира глазами пилота может быть использована следующая функция:

```
void pilotView(GLdouble planex, GLdouble planey, GLdouble planez, GLdouble roll,
GLdouble pitch, GLdouble heading)
{
   glRotated(roll,0.0,0.0,1.0);
   glRotated(pitch,0.0,1.0,0.0);
   glRotated(heading,1.0,0.0,0.0);
   glTranslated(-planex,-planey,-planez);
}
```

Теперь предположим, что вашему приложению требуется вращать камеру вокруг объекта, находящегося в начале координат. В этом случае вы, вероятно, захотите задавать видовое преобразование в терминах полярной системы координат. Допустим, что переменная ***distance*** задает радиус орбиты, то есть расстояние от камеры до начала координат. (Вначале камера отодвигается на ***distance*** единиц вдоль положительного направления оси ***z***.) Переменная ***azimuth*** задает угол вращения камеры вокруг объекта в плоскости ***xy***, отмеряемый от положительного направления оси ***y***. Похожим образом, ***elevation*** – это угол вращения камеры в плоскости ***yz***, отмеряемый от положительного направления оси ***z***. Наконец, ***twist*** представляет собой угол вращения объема видимости вокруг линии обзора. В этом случае подойдет следующая функция.

```
void polarView(GLdouble distance, GLdouble twist, GLdouble elevation, GLdouble
azimuth)
{
   glTranslated(0.0,0.0,-distance);
   glRotated(-twist,0.0,0.0,1.0);
   glRotated(-elevation,1.0,0.0,0.0);
   glRotated(azimuth,0.0,0.0,1.0);
}
```

3.3 Проекционные трансформации

В предыдущем разделе объяснялось, как задать нужную видовую матрицу для применения требующихся модельных и видовых преобразований. В этом разделе объясняется, как определить нужную матрицу проекции, которая также используется для преобразования вершин в вашей сцене. Помните, что до выполнения любых команд преобразований, описанных в этом разделе, следует вызвать

```
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
```

чтобы следующие команды изменяли именно матрицу проекции, а не видовую матрицу и во избежание составных проекционных преобразований. Поскольку команда проекционного преобразования полностью описывает отдельную трансформацию, обычно вам не нужно комбинировать одну трансформацию с другой.

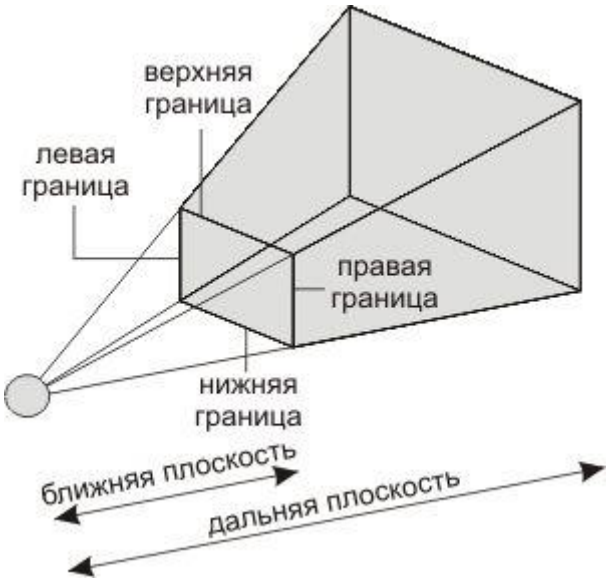
Назначение проекционного преобразования заключается в определении *объема видимости*, который используется двумя путями. Объем видимости определяет, как объект проецируется на экран (с использованием перспективной или параллельной проекции), он также определяет, какие объекты или части объектов будут отсечены в результирующем изображении. Точку наблюдения, о которой мы говорили раньше, вы можете представить себе находящейся на одном из концов объема видимости.

3.3.1 Перспективная проекция

Наиболее узнаваемой характеристикой перспективной проекции является уменьшение на расстоянии: чем дальше объект находится от камеры (точки наблюдения), тем меньше он будет в финальном изображении. Это происходит потому, что объем видимости перспективной проекции имеет форму усеченной пирамиды (пирамиды, верхушка которой отрезана плоскостью, параллельной ее основанию). Объекты, попадающие в объем видимости проецируются из вершины пирамиды, где находится точка наблюдения. Более близкие к точке наблюдения объекты получаются крупнее, поскольку они занимают пропорционально большее пространство объема видимости. Более далекие объекты оказываются меньше, поскольку они находятся в более широкой части усеченной пирамиды объема видимости. Данный метод проецирования используется для анимации, визуальной симуляции и в любых других приложениях, претендующих на некоторую долю реализма, так как перспективное проектирование похоже на то, как видит человеческий глаз (или камера).

Команда определения объема видимости в форме усеченной пирамиды **glFrustum()** вычисляет матрицу, выполняющую перспективное проецирование, и умножает на нее текущую матрицу проекции (обычно единичную). Вспомните, что объем видимости используется для отсечения объектов лежащих вне него; четыре стороны пирамиды, ее основание и вершина (точнее, верхняя сторона) соответствуют шести отсекающим плоскостям объема видимости, как показано на рисунке **3-11**. Объекты или части объектов вне этих плоскостей отсекаются и не выводятся в финальном изображении. Заметьте, что **glFrustum()** не требует от вас указания симметричного объема видимости.

Рисунок **3-11.** Объем видимости перспективной проекции, заданный командой **glFrustum()**



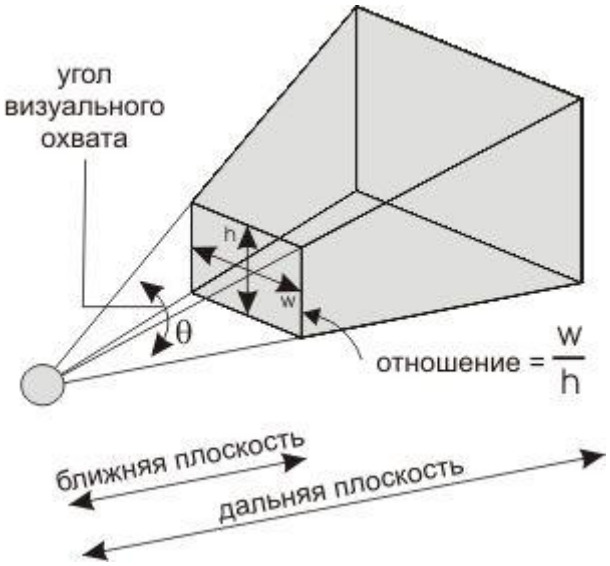

```
void glFrustum (GLdouble left, GLdouble right, GLdouble bottom, GLdouble top,
GLdouble near, GLdouble far);
```

Создает матрицу перспективного проецирования и умножает на нее текущую матрицу. Объем видимости задается параметрами **(*left*, *bottom*,*-near*)** и **(*right*, *top*,*-near*)** определяющими координаты **(*x*, *y*, *z*)** левого нижнего и правого верхнего углов ближней отсекающей плоскости; ***near***и ***far*** задают дистанцию от точки наблюдения до ближней и дальней отсекающих плоскостей **(**они всегда должны быть положительными**).**

Пирамида имеет ориентацию в пространстве по умолчанию. Вы можете производить повороты или переносы для управления ее положением, но это весьма сложный процесс, которого почти всегда можно избежать.

Дополнительно: Пирамида не обязана быть симметричной и ее центральная ось не обязательно должна совпадать с осью ***z***. Например, вы можете использовать **glFrustum()** для создания такого изображения, как если бы вы смотрели через прямоугольное окно, причем это окно находится выше и правее вас. Фотографы используют этот прием для создания ложной перспективы. Вы можете использовать его, чтобы аппаратно визуализировать изображения с разрешением значительно выше обычного **(**например, для вывода на принтер**).** Предположим, вам требуется изображение с разрешением вдвое большим разрешения вашего экрана. Нарисуйте изображение **4** раза, каждый раз используя пирамидальный объем видимости для покрытия всего экрана одной четвертью изображения. После того, как каждая четверть будет выведена на экран, вы можете считать пиксели, собрав, таким образом, данные для изображения высокого разрешения.

Рисунок **3-12.** Объем видимости перспективной проекции, заданный командой **gluPerspective**



Хотя **glFrustum()** концептуально ясна, ее использование не является интуитивно понятным. Вместо нее вы можете попробовать использовать функцию **gluPerspective()** из библиотеки утилит. Эта функция создает объем видимости той же формы, что и **glFrustum()**, но вы задаете его параметры иным путем. Вместо указания углов ближней отсекающей плоскости, вы задаете угол визуального охвата (**θ** или тета) в вертикальном направлении ***y*** и отношение ширины к высоте (***x/y***). (Для квадратной части экрана отношение ширины к высоте равно ***1.0***.) Этих двух параметров достаточно для определения неусеченной пирамиды вдоль направления обзора (рисунок **3-12**). Вы также задаете дистанцию между точкой наблюдения и ближней и дальней отсекающими плоскостями, таким образом, отсекая пирамиду.

Заметьте, что **gluPerspective()** ограничена созданием только пирамид симметричных вдоль линии обзора по ***x***- и ***y***-осям, но обычно это именно то, что и требуется.

```
void gluPerspective (GLdouble fovy, GLdouble aspect, GLdouble near, GLdouble
far);
```

Создает матрицу для пирамиды симметричного перспективного вида и умножает на нее текущую матрицу. Параметр ***fovy*** задает угол визуального охвата в плоскости ***yz***, его значение должно лежать в диапазоне ***[0.0, 180.0]***. Параметр ***aspect*** – это отношение ширины пирамиды к ее высоте. Параметры ***near*** и ***far*** представляют дистанции от точки наблюдения до ближней и дальней плоскостей отсечения вдоль отрицательного направления оси ***z***.

При использовании **gluPerspective()** вам необходимо выбрать подходящее значение для угла визуального охвата, иначе изображение будет выглядеть непропорциональным. Для получения наилучшего результата, заметьте, на каком расстоянии ваш глаз находится от монитора обычно и насколько велико окно, затем вычислите угол, на который распространяется окно в вашем поле зрения при данных дистанции и размере. Скорее всего, эта величина будет меньше, чем вы ожидали. Другой способ задуматься об этом – помнить, что **94**-ех градусный охват с **35**-миллиметровой камерой требует **20**-миллиметровых линз, которые считаются весьма широкоугольными.

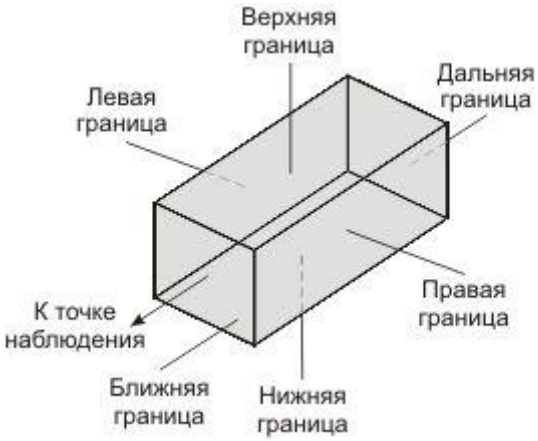
В данном разделе встречались упоминания о миллиметрах и дюймах – имеют ли они значение в **OpenGL**. Ответ в одно слово – нет. Проекционные и иные трансформации не имеют единиц измерения. Если вы предпочитаете думать, что ближняя и дальняя плоскости отсечения находятся на расстояниях **1.0** и **20.0** километров, метров, дюймов, миль и так далее – это ваше право. Единственное правило – используйте одинаковые

единицы для всех измерений в приложении, тогда изображение будет иметь нужные пропорции и масштаб.

3.3.2 Ортографическая проекция

При использовании ортографической проекции объем видимости представляет собой прямоугольный параллелепипед или коробку (рисунок **3-13**). В отличие от перспективной проекции размер объема видимости не изменяется от одного конца к другому, таким образом, дальность от камеры не влияет на размер объектов в результирующем изображении. Этот тип проекции используется для таких приложений, как системы автоматизированного проектирования, где важны реальные размеры объектов относительно друг друга и точность отображения углов между ними.

Рисунок **3-13.** Объем видимости ортографической проекции



Команда **glOrtho()** создает параллельный объем видимости ортографической проекции. Как и в случае **glFrustum()** вы задаете углы ближней отсекающей плоскости и расстояния до ближней и дальней плоскостей.

```
void glOrtho (GLdouble left, GLdouble right, GLdouble bottom, GLdouble top,
GLdouble near, GLdouble far);
```

Создает матрицу для параллельного объема видимости ортографической проекции и умножает на нее текущую матрицу. (***left, bottom, -near***) и (***right, top, -near***) задают точки на ближней плоскости отсечения, которые будут спроецированы соответственно на нижний левый и верхний правый углы порта просмотра. (***left, bottom, -far***) и (***right, top, -far***) это точки на дальней плоскости отсечения, которые будут спроецированы на те же углы порта просмотра. Значения для ***far*** и ***near*** могут быть положительными, отрицательными или даже нулевыми, однако они не должны быть равны между собой.

Если никакие другие преобразования не используются, направление проецирования совпадает с осью ***z***, а направление обзора – с ее отрицательным направлением.

Для специального случая, когда двумерное изображение проецируется на двумерный экран, используйте функцию **gluOrtho2D()** из библиотеки утилит. Эта функция идентична команде **glOrtho()**, но она предполагает, что все ***z*** – координаты объектов лежат в диапазоне от ***1.0*** до ***-1.0***. Если вы рисуете двумерные объекты с применением двумерных версий вершинных команд, все их ***z*** – координаты равны нулю, таким образом, ни один из объектов не отсекается из-за своих ***z*** – координат.

```
void gluOrtho2D (GLdouble left, GLdouble right, GLdouble bottom, GLdouble top);
```

Создает матрицу для проецирования двумерного изображения на экран и умножает на нее текущую матрицу. Область отсечения представляет собой прямоугольник с нижним левым углом в (***left, bottom***) и правым верхним углом в (***right, top***).

3.3.3 Отсечение по объему видимости

После того, как вершины объектов сцены преобразованы с помощью видовой и модельной матриц, любые примитивы, лежащие вне объема видимости отсекаются. В качестве плоскостей отсечения используются те шесть, которые задают стороны объема видимости. Вы также можете задавать дополнительные плоскости отсечения и располагать их там, где вы хотите. Имейте в виду также, что **OpenGL** реконструирует ребра отсекаемых полигонов.

3.4 Подробнее о порядке преобразований

Дополнительно: Ранее было отмечено, что все преобразования вершин в **OpenGL** (за исключением преобразования порта просмотра) имеют матричную природу. В теории компьютерной графики (и во многих приложениях) преобразования вершин производятся следующим образом. Предположим, требуется повернуть вершину с однородными координатами (***x***, ***y***, ***z***, ***1***) на угол χ вокруг оси аппликат (оси ***z***), причем центр поворота должен совпадать с началом координат. Соответствующей матрицей поворота будет:

$$[R_z] = \begin{bmatrix} \cos\chi & \sin\chi & 0 & 0 \\ -\sin\chi & \cos\chi & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

Затем требуется ортогонально спроецировать вершину вдоль оси ***z*** на плоскость ***xy***. Матрица этого преобразования имеет вид:

$$[P_z] = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

Для выполнения двух указанных преобразований необходимо, соблюдая их порядок, умножить вектор координат вершины на матрицу $[R_z]$, а затем результат умножить на матрицу $[P_z]$ (альтернативно можно умножить матрицу $[R_z]$ на , $[P_z]$ а затем умножить вектор координат на результат, но мы для простоты пойдем первым путем). После первого перемножения имеем:

$$[x \quad y \quad z \quad 1] * [R_z] = [x\cos\chi - y\sin\chi \quad x\sin\chi + y\cos\chi \quad z \quad 1].$$

После второго перемножения имеем (это очевидно, так как используемая матрица проекции лишь обнуляет ***z*** координату вершины):

$$[x\cos\chi - y\sin\chi \quad x\sin\chi + y\cos\chi \quad 0 \quad 1].$$

Вершина, определяемая результирующим вектором координат, является искомой.

Механизм расчета преобразований идентичен тому, что используется в **OpenGL** с двумя важными отличиями:

- в вычислениях мы использовали прямой порядок преобразований – **OpenGL**перемножила бы матрицы в обратном порядке (сначала проекционную матрицу на матрицу поворота, затем результат – на вектор координат) – с первого взгляда это не кажется логичным;
    - матрицы в **OpenGL** хранятся (и используются в вычислениях) в формате по столбцам, а не по строкам.

Как вы увидите далее, эти два отличия связаны между собой, но сначала зададимся вопросом: Почему они вообще имеют место? В чем корни различия между теорией и практикой?

Для начала рассмотрим упрощенно порядок наших действий при вычислениях:

1. Запомнить координаты вершины или вершин, над которыми будет производиться определенная группа преобразований. Перейти к шагу **2**.
2. Получить матрицу первого преобразования и умножить на нее имеющиеся в памяти векторы координат вершин. Перейти к шагу **3**.
3. Получить матрицу второго преобразования и умножить на нее результаты шага **2**. Перейти к шагу **4**...

Вспомним, что **OpenGL** в обращении с геометрическими данными действует подобно конвейеру. При работе конвейера по описанной схеме возникали бы следующие проблемы:

- Конвейер **OpenGL** должен был бы останавливаться всякий раз, когда в него поступила группа вершин, к которым нужно применить определенную группу преобразований. Он простаивал бы до тех пор, пока не поступило бы последнее преобразование группы и сигнал о том, что оно действительно последнее – только после этого можно было бы начать вычисления.
- С простоем связана еще одна проблема – все вершины одной группы необходимо было бы хранить в памяти до того, как поступит последняя матрица преобразования. Количество же вершин в приложении может быть сколь угодно велико (в большинстве приложений количество вершин на порядок больше количества преобразований), и, как следствие, для их хранения во время простоя потребовался бы сколь угодно большой объем памяти.
- Предположим, что к вершинам первого четырехугольника требуется применить поворот и перспективное проецирование, а к вершинам второго – перенос и то же самое перспективное проецирование. При работе по описанной схеме мы должны были бы передать на конвейер матрицу проецирования два раза и два раза выполнить полный пересчет координат вершин с двумя перемножениями, поскольку вершины двух четырехугольников будут находиться в разных группах.

Все эти недостатки вели бы к неэффективности в работе конвейера (если, конечно, его можно было бы назвать конвейером вообще). Вследствие этого конвейер **OpenGL** функционирует по-другому.

Преобразования задаются до поступления координат вершин. Матрица первого преобразования становится текущей. При поступлении матрицы второго преобразования текущая матрица умножается на нее, и результат становится новой текущей матрицей. Количество поступающих матриц преобразований в этом случае может быть сколь угодно велико, но **OpenGL**хранит только одну – текущую (или точнее две текущие, поскольку проекционные и видовые/модельные преобразования хранятся

в разных матрицах). Координаты же вершин могут поступать в любой момент между двумя командами преобразования, и для их трансформации будет использована только одна текущая матрица. Поскольку вычисления производятся по мере поступления вершин, конвейер не простаивает.

Теперь обратимся к вопросу о том, почему проекционные и видовые/модельные преобразования хранятся в **OpenGL**в двух разных матрицах и не комбинируются заранее. Следуя схеме вычислений, описанной в начале этого раздела, матрица проекции должна быть последней в цепочке матриц преобразований, соответствующих группе вершин. Рассмотрим, как должна была бы выглядеть типичная программа:

```
Задать модельное преобразование 1
Задать матрицу проекции
Задать группу вершин 1

Установить текущую матрицу в единичную
Задать модельное преобразование 2
Задать модельное преобразование 3
Задать матрицу проекции
Задать группу вершин 2

...
```

Иными словами если бы проекционная матрица в **OpenGL** не хранилась отдельно, ее нужно было бы поставлять на конвейер каждый раз до передачи туда группы вершин, а это неэффективно, так как матрица проекции, как правило, была бы одной и той же.

Кроме того, хранение и упрежденное комбинирование матрицы проекции с матрицами модельных или видовых преобразований сделало бы невозможным само накопление модельных преобразований. Рассмотрим следующий фрагмент:

```
Перейти к позиции дома
Нарисовать дом
Перейти к позиции окна относительно дома
Нарисовать окно
```

Запись является достаточно логичной: задаем матрицу переноса к точке, где надо нарисовать дом, рисуем его, еще одним переносом смещаемся относительно него в позицию окна (то есть перемножаем матрицу первого переноса на матрицу второго), рисуем окно. Есть, однако, одна проблема: чтобы отобразить вершины дома мы должны не только перенести их, но и спроецировать, то есть на момент второго переноса текущая матрица будет испорчена матрицей проекции и двойной комбинированный перенос не получится. От этой беды, кстати, не спасает и хранение проекционной матрицы отдельно от видовой.

В **OpenGL**проблема решается за счет того, что матрицы множатся в обратном порядке. Для того, чтобы результат был верен требуется обеспечить транзитивность произведения матриц (которой на самом деле не существует) – именно поэтому матрицы в **OpenGL** хранятся по столбцам. Рассмотрим, как в **OpenGL** производится расчет координат вершины из примера в начале раздела.

Первой задается матрица проекции, хранимая по столбцам (в данном случае она идентична хранимой по строкам):

$$\left[P_z^*\right]=\begin{bmatrix}1 & 0 & 0 & 0\\ 0 & 1 & 0 & 0\\ 0 & 0 & 0 & 0\\ 0 & 0 & 0 & 1\end{bmatrix}.$$

Затем поступает команда преобразования **(**поворота**),** формирующая матрицу поворота, хранимую по столбцам **(**эта матрица становится текущей**):**

$$\left[R_z^*\right]=\begin{bmatrix}\cos\chi & -\sin\chi & 0 & 0\\ \sin\chi & \cos\chi & 0 & 0\\ 0 & 0 & 1 & 0\\ 0 & 0 & 0 & 1\end{bmatrix}.$$

Наконец, поступает столбец вершинных координат, которые требуется преобразовать**:**

$$\left[P_z^*\right]*\left[R_z^*\right]*\begin{bmatrix}x\\ y\\ z\\ 1\end{bmatrix}=\begin{bmatrix}x\cos\chi - y\sin\chi & x\sin\chi + y\cos\chi & 0 & 1\end{bmatrix}.$$

Сравните результат с приведенным в начале раздела, чтобы убедиться в его правильности**.**

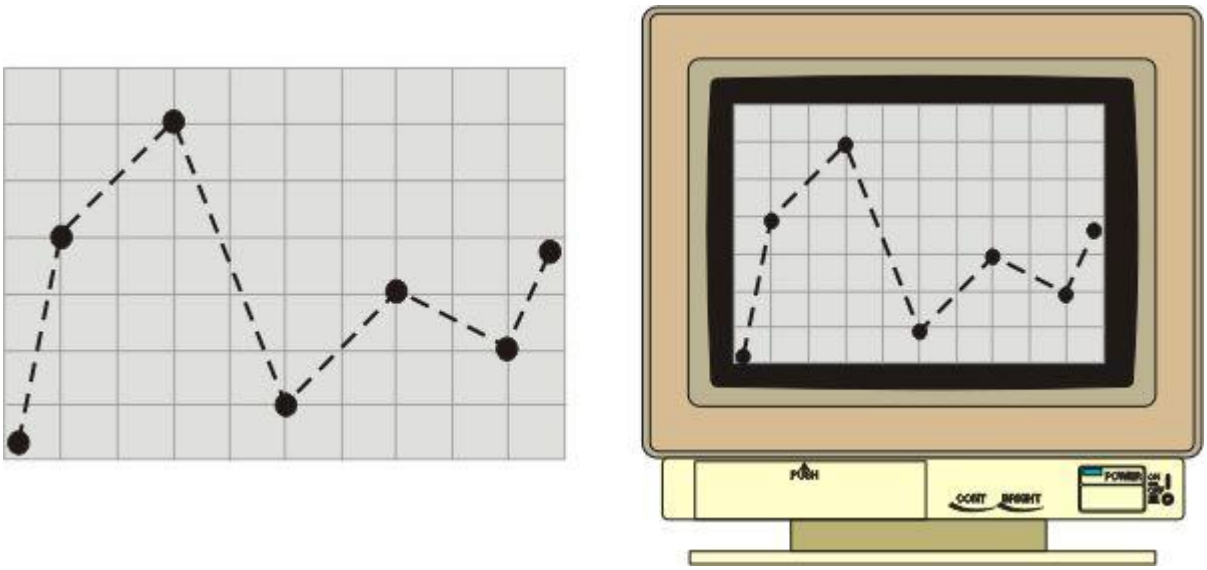
Резюмируем все вышесказанное**:**

- Текущая видовая и текущая проекционная матрицы хранятся в **OpenGL** отдельно, причем каждая из них хранится по столбцам**.**
- Хранение по столбцам необходимо для обеспечения видимой транзитивности произведения матриц, так как реально они перемножается в порядке поступления **(**Проекционная матрица на видовую матрицу, их произведение **–** на последовательно поступающие матрицы модельных преобразований и, в итоге, результат **–** на вектор координат вершины**).**
- Все это сделано из соображений эффективности, а также для обеспечения возможности комбинирования модельных преобразований, использования дисплейных списков **(**списков отображения**)** и так далее**.** Разработчики стандарта **OpenGL** действительно хорошо поработали**!**

3.5 Трансформация порта просмотра

Вспомните аналогию с камерой **–** вы знаете, что трансформация порта просмотра это тот этап, на котором выбирается размер результирующей фотографии**.** Поскольку мы работаем с компьютерной графикой, порт просмотра представляет собой прямоугольную область окна, где рисуется изображение**.** На рисунке **3-14** изображен порт просмотра, занимающий большую часть экрана**.** Порт просмотра измеряется в оконных координатах, которые отражают позиции пикселей на экране относительно нижнего левого угла окна**.** Имейте в виду, что к текущему моменту все вершины уже преобразованы с помощью видовой и проекционной матриц, и те из них, которые не попали в объем видимости, были отсечены**.**

Рисунок **3-14.** Прямоугольник порта просмотра



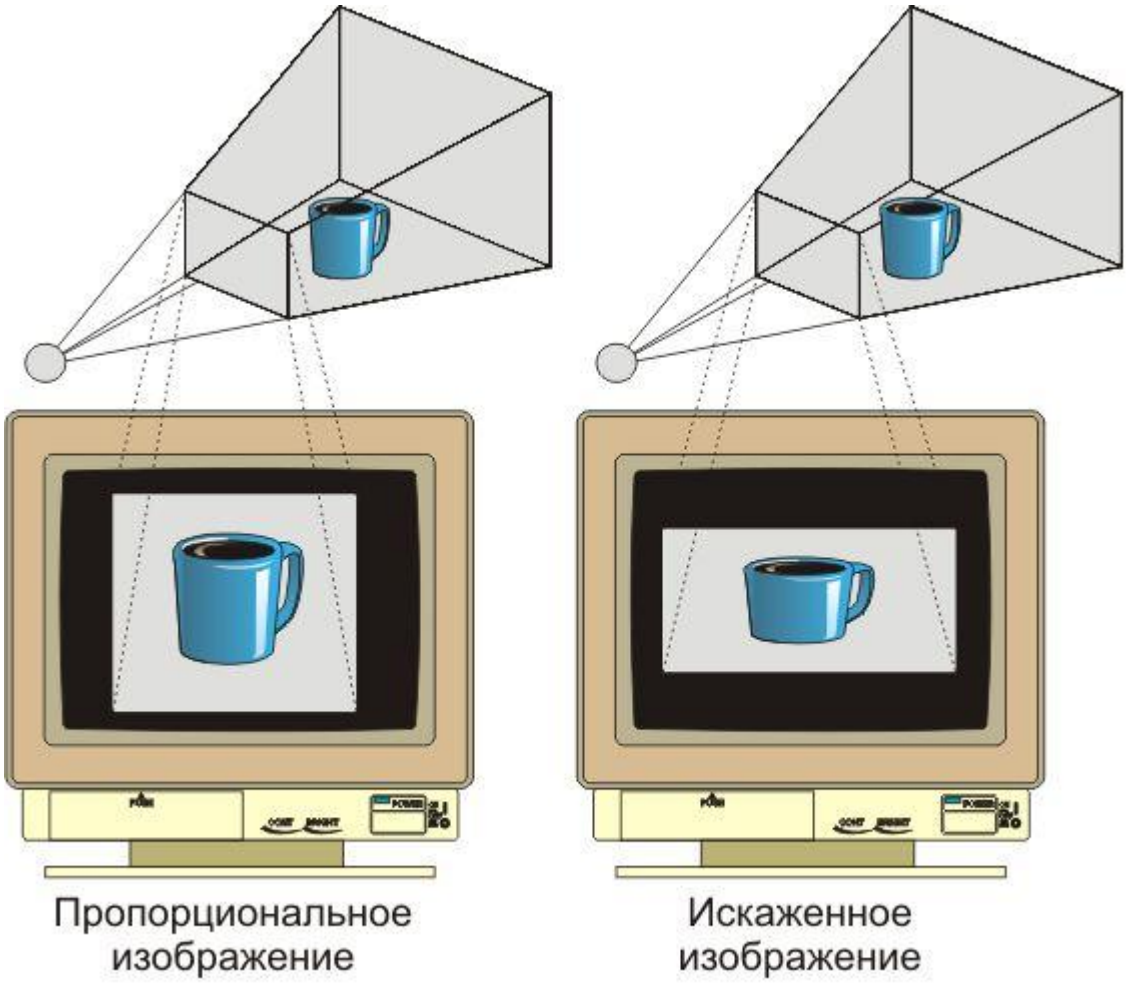
3.5.1 Задаем порт просмотра

```
void glViewport (GLint x, GLint y, GLint width, GLint height);
```

Задает прямоугольник пикселей в окне, в который будет перенесено финальное изображение. Параметры ***(x,y)*** задают нижний левый угол порта просмотра, а параметры ***width***и ***height***– размер прямоугольника порта просмотра. По умолчанию левый нижний угол порта просмотра находится в левом нижнем углу окна, а его размер совпадает с размерами окна.

Отношение ширины порта просмотра к его высоте обычно должно быть таким же, как и соответствующее отношение объема видимости используемой проекции. Если эти два отношения не совпадают, спроецированное изображение при отображении в порте просмотра будет искажено, как показано на рисунке **3-15**. Заметьте, что изменение размеров окна не влияет на порт просмотра. В своем приложении вы должны отслеживать события изменения размеров окна и изменять порт просмотра соответствующим образом.

Рисунок **3-15.** Отображение объема видимости в порт просмотра



Левая часть рисунка **3-15** демонстрирует проекцию, отображающую квадратное изображение в квадратный порт просмотра. Для этого необходимо использовать последовательность команд подобную этой:

```
gluPerspective(fovy, 1.0, near, far);
glViewport(0,0,400,400);
```

Однако, в правой части того же рисунка размеры и форма окна и порта просмотра были изменены, но проекция осталась той же. В результате изображение выглядит сжатым по оси ***y***:

```
gluPerspective(fovy, 1.0, near, far);
glViewport(0,0,400,200);
```

Во избежание искажений измените отношение ширины проекции к ее высоте, чтобы оно совпадало с аналогичным отношением порта просмотра:

```
gluPerspective(fovy, 2.0, near, far);
glViewport(0,0,400,200);
```

Замечание: В каждый момент времени вы можете рисовать только в одном порте просмотра. Однако вы можете задать один порт просмотра (например, в правой половине окна), нарисовать в нем некоторую сцену, затем задать второй порт просмотра (в левой половине окна) и нарисовать там ту же или другую сцену. Таким образом, вы получите две сцены в двух портах просмотра, но при этом в одном и том же окне.

3.5.2 Преобразованная глубина

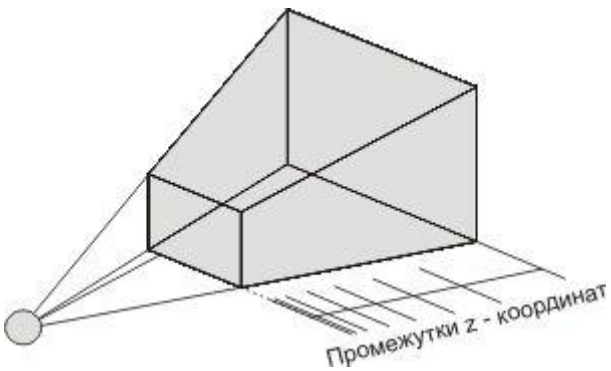
Глубинная координата **(*z*)** кодируется во время трансформации порта просмотра **(**и позже сохраняется в буфере глубины**).** С использованием команды **glDepthRange()** вы можете масштабировать ***z*** **–** величины для того**,** чтобы они лежали в определенном диапазоне**.** В отличие от ***x*** **–** и ***y*** **–** координат окна**,** оконные ***z*****–** координаты всегда трактуются **OpenGL**как лежащие в диапазоне от ***0.0*** до ***1.0*****.**

```
void glDepthRange (GLclampd near, GLclampd far);
```

Определяет кодирование**,** которое производится для ***z*****–** координат во время трансформации порта просмотра**.** Параметры ***near*** и ***far***представляют собой минимальную и максимальную границы величин**,** которые могут быть сохранены в буфере глубины**.** По умолчанию эти границы равны ***0.0*** и ***1.0*** соответственно**,** что подходит для большинства приложений**.** Параметры ***near*** и ***far*** должны лежать в диапазоне **[*0.0*, *1.0*].**

При перспективном проецировании преобразованная координата глубины **(**так же как ***x*** **-** и ***y*** **-** координаты**)** подвергается перспективному делению на координату ***w*****.** С увеличением дистанции между ближней отсекающей плоскостью и ***z*** **–** координатой**,** ее местоположение становится все менее и менее точным **(**рисунок **3-16).**

Рисунок **3-16.** Перспективное проецирование и преобразованная координата глубины



Таким образом**,** перспективное деление влияет на точность операций базирующихся на преобразованных координатах глубины**,** особенно на операции с буфером глубины**,** который используется для удаления невидимых поверхностей**.**

3.6 Наиболее частые проблемы**,** связанные с трансформациями

Достаточно легко нацелить реальную камеру в правильном направлении**,** однако при работе с компьютерной графикой**,** вы должны задавать позицию и направление с помощью координат и углов**.** Как мы все знаем**,** получить хорошо**-**известный эффект черного экрана при этом слишком легко**.** Можно сделать множество ошибок**,** в результате которых на экране не будет нарисовано абсолютно ничего **(**например**,** неправильно навести камеру или попытаться получить изображение сцены**,** находящейся за наблюдателем**).**

Если вы обнаружили**,** что приложили нечеловеческие усилия только для того**,** чтобы получить черный экран**,** попробуйте выполнить следующие диагностические шаги**.**

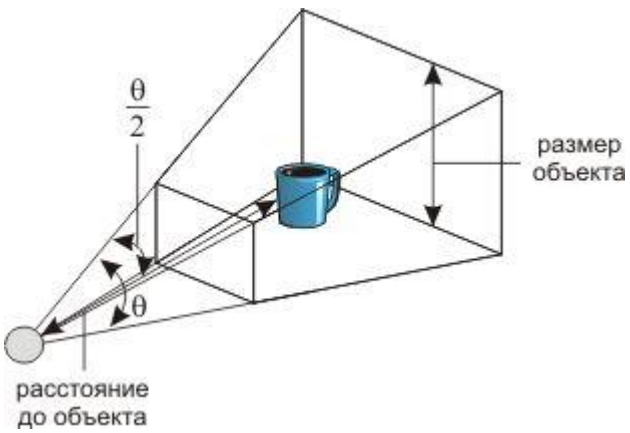
1. Проверьте очевидные факты**.** Убедитесь**,** что монитор включен**.** Убедитесь**,** что вы рисуете свои объекты цветом отличным от того**,** которым вы очищаете порт просмотра**.** Убедитесь**,** что любые используемые вами механизмы **(**такие как

освещение, текстурирование, альфа наложение или антиалиасинг) включены или выключены (в зависимости от того, что именно вам требуется).

2. Помните, что при использовании проекционных команд координаты ближней и дальней отсекающих плоскостей представляют собой дистанцию от точки наблюдения и направление обзора (по умолчанию) совпадает с отрицательным направлением оси ***z***. Таким образом, если ближняя плоскость находится на расстоянии ***1.0***, а дальняя – на расстоянии ***3.0***, ***z*** – координаты объектов должны лежать в диапазоне ***[-1.0, -3.0]*** для того, чтобы объекты были видимыми. Если вы хотите убедиться, что не отсекли все объекты сцены, временно установите параметры **near**и **far**проекционных команд в абсолютно абсурдные значения вроде ***0.001*** и ***1000000.0*** соответственно. Это полностью сведет на нет действие таких механизмов, как буфер глубины и туман, но зато позволит обнаружить ошибочно отсекаемые объекты.
3. Определите, где находится точка наблюдения, в каком направлении вы смотрите, и где находятся объекты. Возможно, в этом случае вам поможет построение реальной трехмерной модели (например, с помощью рук).
4. Убедитесь в том, что вы знаете, вокруг какой точки вы производите вращение объектов. Нет ничего страшного во вращении вокруг любой точки, если только вам не требуется вращение именно вокруг начала координат.
5. Проверьте ваш прицел. Используйте **gluLookAt()**, чтобы навести объем видимости на ваши объекты. Или нарисуйте объекты вблизи начала координат, а затем используйте **glTranslate*()** в качестве видовой трансформации, чтобы отодвинуть камеру от модели несколько дальше по оси ***z*** и объекты попадали в объем видимости. После того, как вам удалось сделать объекты видимыми, попробуйте инкрементально изменять объем видимости до получения именно тех результатов, каких вы хотите.
6. При использовании перспективных преобразований убедитесь, что ближняя отсекающая плоскость находится не слишком близко к наблюдателю (камере), так как это может сильно повлиять на точность глубинной буферизации.

Даже после того, как вы навели камеру в правильном направлении и увидели ваши объекты, они могут выглядеть слишком маленькими или слишком большими. Если вы используете **gluPerspective()**, возможно вам придется изменить угол, задающий визуальный охват, изменив значение первого параметра этой команды. С помощью тригонометрии вы можете вычислить нужный вам угол, опираясь на размер объекта и его дистанцию от точки наблюдения: тангенс половины нужного угла равен отношению половины размера объекта к его дистанции от точки наблюдения (рисунок **3-17**). Таким образом, вы можете использовать функцию вычисления арктангенса для получения значения половины нужного вам угла. В примере **3-3** предполагается наличие функции **atan2()**, которая вычисляет арктангенс угла между противолежащим и прилежащим катетами прямоугольного треугольника, длины которых передаются в качестве параметров. Затем результат, полученный в радианах, преобразуется в градусы.

Рисунок **3-17.** Использование тригонометрии для вычисления угла визуального охвата



Пример **3-3.** Вычисление угла визуального охвата

```
#definePI 3.1415926535>

double calculateAngle(double size, double distance)
{
   double radtheta, degtheta;
   radtheta = 2.0 * atan2(size/2.0,distance);
   degtheta=(180.0*radtheta)/PI;
   return degtheta;
}
```

Конечно, в большинстве случаев вы не знаете точного размера объекта, может быть выяснена только дистанция между точкой наблюдения и одной из точек вашей сцены. Чтобы получить размер объекта с достаточно точным приближением, постройте прямоугольный параллелепипед, содержащий в себе всю вашу сцену. Для этого определите максимальные и минимальные значения ***x*** **-**, ***y*** **-** и ***z*** **–** координат всех объектов вашей сцены. Затем вычислите радиус сферы, описывающей получившийся параллелепипед. Наконец, используйте центр сферы в качестве дистанции до объекта и радиус этой сферы в качестве половины размера объекта.

Предположим, например, что все координаты вашего объекта **(**или объектов**)** удовлетворяют следующим уравнениям: $-1 \le x \le 3$, $5 \le y \le 7$ и $-5 \le z \le 5$. Тогда центр параллелепипеда будет находиться в точке **(1, 6, 0)**, а радиус описывающей его сферы будет равен расстоянию из центра параллелепипеда до любого из его углов, скажем до **(3, 7, 5)**, а именно он будет равен:

$$\sqrt{(3-1)^2 + (7-6)^2 + (5-0)^2} = \sqrt{30} \approx 5.477.$$

Если точка наблюдения имеет координаты **(8, 9, 10)**, дистанция между ней и центром будет равна:

$$\sqrt{(8-1)^2 + (9-6)^2 + (10-0)^2} = \sqrt{158} \approx 12.57.$$

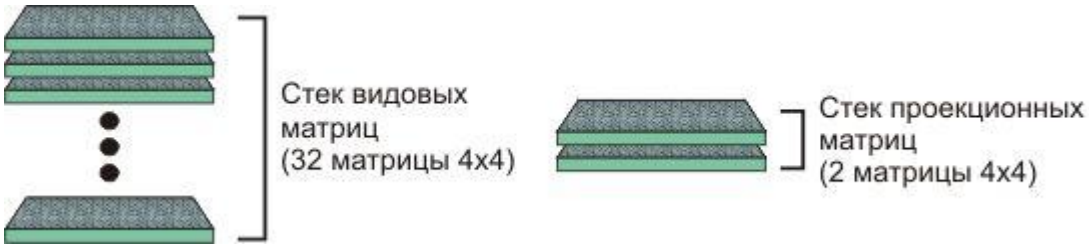
Тангенс половины нужного угла равен отношению **5.477** к **12.570**, которое составляет **0.4357**, то есть половина нужного угла равна **23.54** градусам.

Угол визуального охвата влияет на оптимальное положение точки наблюдения. Это следует помнить, если вы пытаетесь создать реалистичное изображение. Например, если ваши расчеты показывают, что вам требуется угол визуального охвата равный **179** градусам, точка наблюдения должна находиться на расстоянии доли дюйма от экрана для достижения реализма. Если вычисленный вами визуальный охват слишком велик, возможно, потребуется передвинуть точку наблюдения дальше от объекта.

3.7 Манипулирование матричными стеками

Видовая и проекционная матрицы, которые вы создавали, загружали и перемножали были на самом деле лишь вершиной айсберга. На самом деле каждая из этих матриц является верхним элементом матричного стека **(**рисунок **3-18).**

Рисунок **3-18.** Видовый и проекционный матричные стеки



Матричный стек полезен для построения иерархических моделей, в которых сложные объекты конструируются при помощи простых. Предположим, например, что вы рисуете автомобиль, у которого **4** колеса и каждое колесо прикреплено к автомобилю пятью болтами. У вас имеется функция, рисующая колесо и функция, рисующая болт, так как все колеса и болты выглядят одинаково. каждая из этих функций рисует колесо или болт в четко определенном месте, скажем в начале координат, и с определенной ориентацией, например, с центральной осью объекта, совпадающей с отрицательным направлением оси **z**. Когда вы рисуете машину, включая колеса и болты, вы захотите вызвать функцию, рисующую колесо **4** раза с применением различных трансформаций для правильного позиционирования каждого колеса. При рисовании каждого колеса вы захотите нарисовать болт пять раз, каждый раз перенося болт в нужное место относительно колеса.

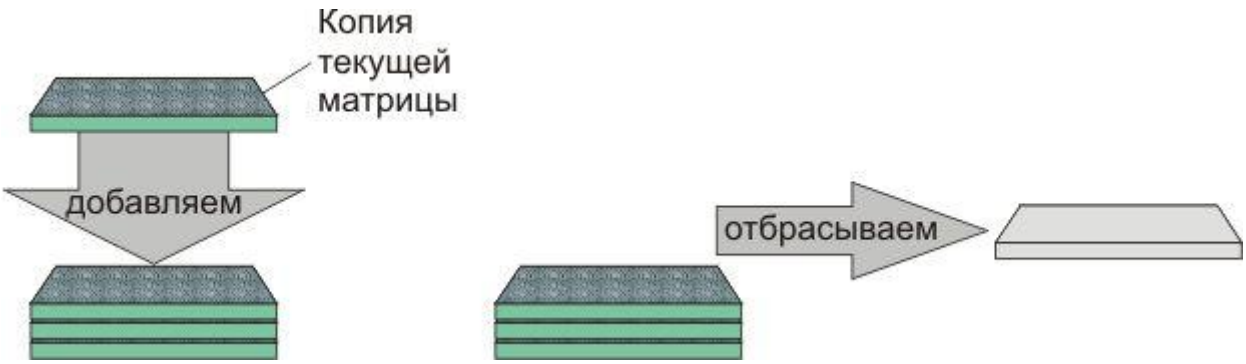
На секунду предположим, что все, что вы хотите сделать – это нарисовать корпус машины и колеса. В этом случае алгоритм процесса может быть описан следующим образом:

- Нарисовать корпус машины.
- Запомнить, где мы находимся, и выполнить перенос к переднему левому колесу.
- Нарисовать колесо и отбросить последний перенос, чтобы вернуться в начало координат относительно корпуса машины.
- Запомнить, где мы находимся, и выполнить перенос к левому заднему колесу...

Похожим образом для каждого колеса, нам следует нарисовать его, запомнить, где мы, и последовательно выполнять переносы к позиции каждого болта, отбрасывая преобразования после того, как каждый болт нарисован.

Поскольку преобразования сохраняются в матрицах, матричный стек предоставляет идеальный механизм для подобного рода запоминаний, переносов и отбрасываний. Все ранее описанные матричные операции **(glLoadMatrix()**, **glMultMatrix()**, **glLoadIdentity()** и команды, создающие специфические матрицы**)** работают с текущей матрицей, то есть с верхней матрицей стека. С помощью команд управления стеком вы можете управлять тем, какая матрица находится на вершине стека: **glPushMatrix()** копирует текущую матрицу и добавляет копию на вершину матричного стека, **glPopMatrix()** уничтожает верхнюю матрицу в стеке **(**рисунок **3-19). (**Помните, что текущей матрицей всегда является матрица на вершине**).** Говоря проще, **glPushMatrix()** означает **«**запомнить, где мы находимся**»**, а **glPopMatrix() – «**вернуться туда, где мы были**».**

Рисунок **3-19.** Помещение в матричный стек и извлечение из матричного стека




```
void glPushMatrix (void);
```

Опускает все имеющиеся в текущем стеке матрицы на один уровень. То, какой стек является текущим, задается с помощью вызова **glMatrixMode()**. Верхняя матрица при этом копируется, таким образом, ее содержимое продублировано в верхней и второй сверху матрице стека. Если добавлено слишком много матриц, будет сгенерирована ошибка.

```
void glPopMatrix (void);
```

Выкидывает верхнюю матрицу из стека, тем самым, уничтожая ее содержимое. Верхней (и, как следствие, текущей) становится матрица, которая занимала второе сверху место в стеке. Текущий стек задается командой **glMatrixMode()**. Если стек содержит только одну матрицу, вызов **glPopMatrix()** сгенерирует ошибку.

Пример **3-4** рисует автомобиль в предположении о наличие функций, рисующих корпус машины, колесо и болт.

Пример **3-4.** Помещение и извлечение матриц

```
нарисовать_колесо_и_болты()
{
   long i;

   нарисовать_колесо();
   for(i=0;i<5;i++)
   {
      glPushMatrix();
         glRotatef(72.0*i,0.0,0.0,1.0);
         glTranslatef(3.0,0.0,0.0);
         нарисовать_болт();
      glPopMatrix();
   }
}

нарисовать_корпус_колеса_и_болты()
{
   нарисовать_корпус_машины();
   glPushMatrix();

      //передвинуться к позиции первого колеса
      glTranslatef(40,0,30);
      нарисовать_колесо_и_болты();
   glPopMatrix();

   glPushMatrix();

      //передвинуться к позиции второго колеса
      glTranslatef(40,0,-30);
      нарисовать_колесо_и_болты();
   glPopMatrix();
```

```
    //похожим образом нарисовать еще два колеса
    ...
}
```

В данном коде предполагается, что ось колеса и болта совпадает с осью **z**, что болты располагаются на колесе каждые **72** градуса и находятся на расстоянии **3** единицы от центра колеса. Также предполагается, что передние колеса находятся на **40** единиц впереди и на **30** влево и вправо от центра корпуса.

Стек более эффективен, чем индивидуальная матрица, особенно в том случае, если он реализован аппаратно. Когда вы проталкиваете матрицу в стек, вам не нужно предварительно извлекать из нее данные и запоминать их в главном процессе, кроме того, аппаратура может обладать способностью одновременно копировать несколько элементов матрицы. Иногда стоит держать на дне стека единичную матрицу во избежание многократных вызовов **glLoadIdentity()**.

3.7.1 Стек видовых матриц

Как вы видели раньше, видовая матрица содержит кумулятивное произведение от перемножения матриц, представляющих отдельные видовые и модельные преобразования. Каждая видовая или модельная трансформация создает новую матрицу, на которую умножается текущая видовая матрица. Результат, который становится новой текущей видовой матрицей, представляет композитное преобразование. Стек видовых матриц может содержать как минимум **32** матрицы размерностью **4x4**. В самом начале верхней **(**и единственной**)** матрицей является единичная. Некоторые реализации **OpenGL**могут поддерживать больше чем **32** матрицы в видовом стеке. Чтобы выяснить максимально допустимое число матриц, используйте команду **glGetIntegerv (GL_MAX_MODELVIEW_STACK_DEPTH, GLint *params)**.

3.7.2 Стек проекционных матриц

Матрица проекции содержит матрицу для проекционного преобразования, описывающую объем видимости. В общем случае вам не нужно объединять проекционные матрицы, поэтому вы вызываете **glLoadIdentity()** перед выполнением проекционного преобразования. По этой же причине стек проекционных матриц должен быть всего два уровня в глубину. Некоторые реализации **OpenGL**могут позволять хранить больше двух матриц размерностью **4x4**. Для выяснения максимальной глубины проекционного стека вызовите **glGetIntegerv (GL_MAX_PROJECTION_STACK_DEPTH, GLint *params)**.

Одним из применений второго уровня стека может быть приложение, которому наряду со своим основным окном, содержащим трехмерную модель, требуется отображать окно помощи с текстом внутри него. Поскольку текст наиболее легко позиционируется с применением ортографической проекции, вы можете временно переключиться на ортографическую проекцию, отобразить помощь и затем вернуться к предыдущей проекции:

```
glMatrixMode(GL_PROJECTION);
glPushMatrix();//сохранить текущую проекцию
   glLoadIdentity();
   glOrtho(...);//настроиться на отображение текста
   отобразить_помощь();
glPopMatrix();
```

Обратите внимание на то, что, вероятно, вам потребуется также изменить должным образом и видовую матрицу.

Дополнительно: Если вы достаточно понимаете в математике, вы можете создавать свои собственные проекционные матрицы, выполняющие более сложные проекционные преобразования. Например, **OpenGL** и ее библиотека утилит не содержит встроенных механизмов для выполнения двухточечной перспективной проекции. Тем не менее, если вы хотите эмулировать, скажем, рисунки в виде набросков, вам может понадобиться такая проекционная матрица.

3.8 Дополнительные плоскости отсечения

Помимо шести плоскостей отсечения объема видимости (девой, правой, верхней, нижней, ближней и дальней) вы можете задавать до шести дополнительных плоскостей отсечения, дополнительно ограничивающих объем видимости. Эта техника удобна, например, для отображения «отрезанной» части некоторого объекта.

Каждая плоскость определяется коэффициентами своего уравнения: **A*x*+B*y*+C*z*+D=0**. Отсекающие плоскости автоматически трансформируются в соответствии с модельными и видовыми преобразованиями. Видимым усеченным объемом становится пересечение объема видимости и всех *полупространств* заданных дополнительными плоскостями отсечения. Помните, что **OpenGL** автоматически реконструирует ребра отсеченных полигонов.

```
void glClipPlane (GLenum plane, const GLdouble *equation);
```

Задает плоскость отсечения. Аргумент ***equation*** указывает на **4** коэффициента уравнения плоскости, **A*x*+B*y*+C*z*+D=0**. Все точки с видовыми координатами (x_e, y_e, z_e, w_e), удовлетворяющими условию:

$$(A \quad B \quad C \quad D)M^{-1}(x_e, y_e, z_e, w_e)^T \geq 0,$$

где **M –** видовая матрица являющаяся текущей на момент вызова **glClipPlane()** лежат в полупространстве определенном плоскостью. Все точки вне этого полупространства отсекаются. Аргумент ***plane***должен быть равен **GL_CLIP_PLANE*i***, где ***i*–** целое, показывающее, какую из имеющихся плоскостей мы определяем. ***i***должно лежать в диапазоне от **0** до числа на единицу меньшего, чем максимально допустимое количество дополнительных плоскостей отсечения.

Каждая из дополнительных плоскостей отсечения, кроме того, должна быть включена или выключена (соответственно) командами:

```
glEnable(GL_CLIP_PLANEi);
```

или

```
glDisable(GL_CLIP_PLANEi);
```

Все реализации **OpenGL** должны поддерживать как минимум **6** дополнительных плоскостей отсечения, хотя некоторые могут позволять и больше. Вы можете выяснить максимально допустимое количество дополнительных плоскостей отсечения в вашей реализации **OpenGL**, вызвав **glGetIntegerv()** с аргументом **GL_MAX_CLIP_PLANES**.

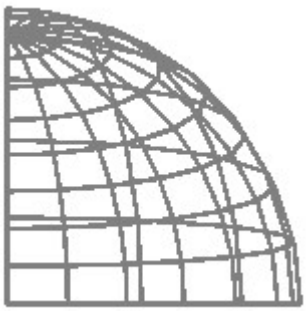
Замечание: Отсечение, производимое как результат вызова **glClipPlane()**, совершается в видовых, а не в усеченных координатах. Эта разница заметна в том случае, если матрица проекции является вырожденной (то есть, реальной проекционной матрицей, отображающей трехмерные координаты на двумерные).

Отсечение, производимое в видовых координатах, продолжает происходить в трех измерениях, даже когда матрица проекции вырождена.

3.8.1 Пример кода с дополнительными плоскостями отсечения

Пример **3-5** визуализирует проволочную сферу с двумя отсекающими плоскостями, которые отрезают три четверти этой сферы **(**рисунок **3-20).**

Рисунок **3-20.** Отсеченная проволочная сфера



Пример **3-5.** Проволочная сфера с двумя отсекающими плоскостями**:** файл **clip.cpp**

```
#include <glut.h>

//Инициализация
void init(void)
{
   glClearColor(0.0,0.0,0.0,0.0);
   glShadeModel(GL_FLAT);
}

//Изменение размеров окна
void reshape(int w, int h)
{
   glViewport(0,0,(GLsizei) w, (GLsizei) h);
   glMatrixMode(GL_PROJECTION);
   glLoadIdentity();
   gluPerspective(60.0,(GLfloat) w/(GLfloat) h,1.0,20.0);
glMatrixMode(GL_MODELVIEW);
}

//Отображение
void display(void)
{
   GLdouble eqn[4]={0.0,1.0,0.0,0.0};
   GLdouble eqn2[4]={1.0,0.0,0.0,0.0};

   glClear(GL_COLOR_BUFFER_BIT);
   glColor3f(1.0,1.0,1.0);
   glPushMatrix();
   glTranslatef(0.0,0.0,-5.0);

   //Отсечь нижнюю половину (y<0)
   glClipPlane(GL_CLIP_PLANE0,eqn);
   glEnable(GL_CLIP_PLANE0);

   //Отсечь левую половину (x<0)
   glClipPlane(GL_CLIP_PLANE1,eqn2);
   glEnable(GL_CLIP_PLANE1);
   glRotatef(90.0,1.0,0.0,0.0);
   glutWireSphere(2.0,20,16);
   glPopMatrix();
   glFlush();
}
```

```
int main(int argc, char **argv)
{
   glutInit(&argc,argv);
   glutInitDisplayMode(GLUT_SINGLE|GLUT_RGB);
   glutInitWindowSize(500,500);
   glutInitWindowPosition(100,100);
   glutCreateWindow("Wireframe Sphere with Two Clipping Planes");
   init();
   glutDisplayFunc(display);
   glutReshapeFunc(reshape);
   glutMainLoop();
   return 0;
}
```

3.9 Примеры комбинирования нескольких преобразований

В этом разделе демонстрируется, как комбинировать несколько преобразований для достижения определенных результатов. Приводятся два примера: солнечная система, в которой объекты должны вращаться относительно своих осей, а также по орбитам вокруг друг-друга и рука робота, в которой присутствует несколько сочленений, изменяющих координатную систему и двигающихся относительно друг-друга.

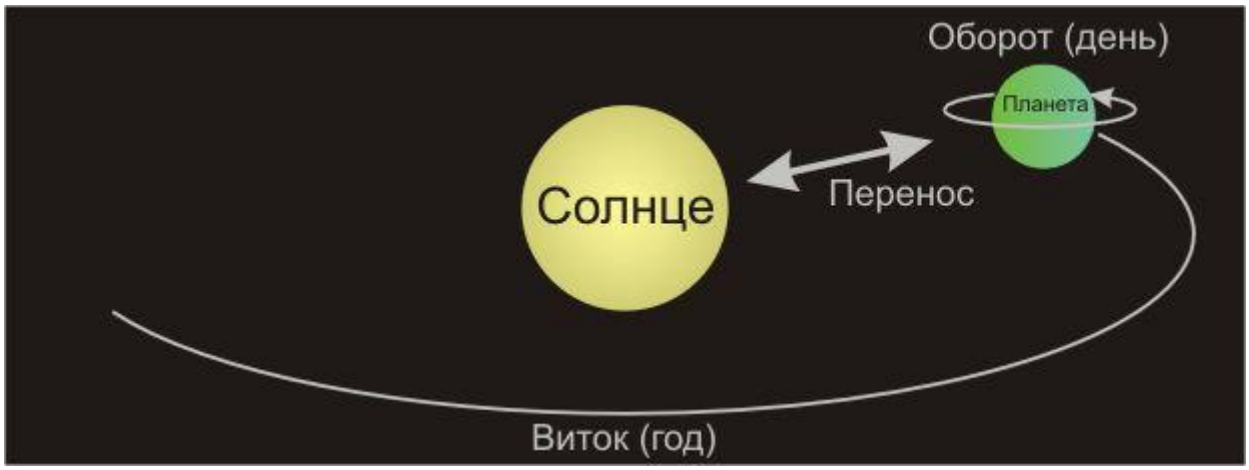
3.9.1 Строим солнечную систему

Программа, описанная в данном разделе, рисует упрощенную модель солнечной системы, состоящую из солнца и одной планеты. Оба объекта рисуются в виде сфер. В ходе написания этой программы потребуется команда **glRotate*()** для вращения планеты вокруг солнца и вокруг своей оси. Также потребуется **glTranslate*()** для перемещения планеты из начала координат на ее орбиту. Помните, что вы можете задавать нужные размеры сфер, передавая соответствующие аргументы функции **glutWireSphere()**.

Чтобы нарисовать солнечную системы, сначала требуется настроить проекцию и видовое преобразование. Для данного примера были использованы **gluPerspective()** и **gluLookAt()**.

Рисование солнца достаточно прямолинейно, поскольку оно должно находиться в начале координат фиксированной координатной системы, то есть там, где ее помещает рисующая ее функция. Таким образом, рисование сферы не требует никаких переносов. Вы можете использовать **glRotate*()**, чтобы заставить солнце вращаться вокруг своей оси. Чтобы нарисовать планету, вращающуюся вокруг солнца, как показано на рисунке **3-21**, требуется выполнить несколько модельных преобразований. Планета должна вращаться вокруг своей оси раз в день, кроме того, раз в год планета должна совершать виток вокруг солнца.

Рисунок **3-21.** Планета и солнце



Для определения порядка модельных преобразований представьте себе, что происходит с локальной координатной системой. Первый вызов **glRotate*()** поворачивает локальную координатную систему (изначально совпадающую с фиксированной). Затем **glTranslate*()** переносит локальную координатную систему в позицию на орбите планеты. Расстояние, на которое выполняется перенос, должно совпадать с радиусом орбиты планеты. Таким образом, начальный вызов **glRotate*()** на самом деле определяет, где именно на орбите находится планета (то есть определяет время года).

Второй вызов **glRotate*()** поворачивает локальную координатную системы вокруг одной из ее осей, таким образом, задавая для планеты время суток. После того, как выполнены все описанные преобразования, можно рисовать планету.

Обобщая вышесказанное, следующая последовательность команд **OpenGL** рисует солнце и планету, а полный текст программы приведен в примере **3-6.**

```
glPushMatrix();
glutWireSphere(1.0,20,16);   //Рисуем солнце
glRotatef((GLfloat) year,0.0,1.0,0.0);
glTranslatef(2.0,0.0,0.0);
glRotatef((GLfloat) day,0.0,1.0,0.0);
glutWireSphere(0.2,10,8);   //Рисуем планету
glPopMatrix();
```

Пример **3-6.** Планетарная система: файл **planet.cpp**

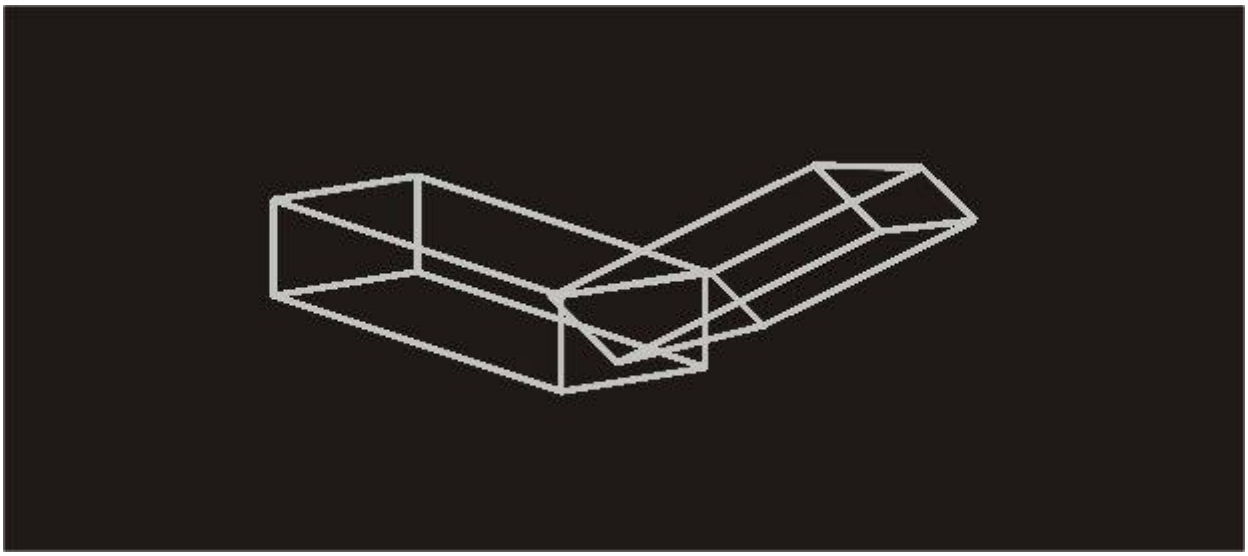
```
#include <glut.h>

int year=0, day=0;

//Инициализация
void init(void)
{
   glClearColor(0.0,0.0,0.0,0.0);
   glShadeModel(GL_FLAT);
}

//Изменение размеров окна
void reshape(int w, int h)
{
   glViewport(0,0,(GLsizei) w, (GLsizei) h);
   glMatrixMode(GL_PROJECTION);
   glLoadIdentity();
   gluPerspective(60.0,(GLfloat) w/ (GLfloat) h,1.0,20.0);
   glMatrixMode(GL_MODELVIEW);
   glLoadIdentity();
   gluLookAt(0.0,0.0,5.0,0.0,0.0,0.0,0.0,1.0,0.0);
}
```

```
//Отображение
void display(void)
{
   glClear(GL_COLOR_BUFFER_BIT);
   glColor3f(1.0,1.0,1.0);
   glPushMatrix();

   //Рисуем солнце
   glutWireSphere(1.0,20,16);
   glRotatef((GLfloat)year,0.0,1.0,0.0);
   glTranslatef(2.0,0.0,0.0);
   glRotatef((GLfloat)day,0.0,1.0,0.0);

   //Рисуем планету
   glutWireSphere(0.2,10,8);
   glPopMatrix();
   glutSwapBuffers();
}

//Реакция на клавиатуру
void keyboard(unsigned char key,int x, int y)
{
   switch(key)
   {
      case 'd':
         day=(day+10)%360;
         glutPostRedisplay();
         break;
      case 'D':
         day=(day-10)%360;
         glutPostRedisplay();
         break;
      case 'y':
         year=(year+5)%360;
         glutPostRedisplay();
         break;
      case 'Y':
         year=(year-5)%360;
         glutPostRedisplay();
         break;
      default:
         break;
   }
}

int main(int argc, char **argv)
{
   glutInit(&argc,argv);
   glutInitDisplayMode(GLUT_DOUBLE|GLUT_RGB);
   glutInitWindowSize(500,500);
   glutInitWindowPosition(100,100);
   glutCreateWindow("Planetary System");
   init();
   glutDisplayFunc(display);
   glutReshapeFunc(reshape);
   glutKeyboardFunc(keyboard);
   glutMainLoop();
   return 0;
}
```

3.9.2 Строим руку робота

В этом разделе обсуждается программа, создающая искусственную руку робота с двумя или более сегментами. Эти фрагменты соединены в точках плеча, локтя и так далее. На рисунке **3-22** показана одна такая точка соединения.

Рисунок **3-22.** Рука робота



В качестве сегментов руки вы можете использовать масштабированные кубы, но сначала следует применить нужные модельные преобразования, чтобы ориентировать каждый сегмент. Поскольку изначально начало локальной координатной системы совпадает с центром куба, требуется передвинуть ее к ребру куба, иначе он будет поворачиваться вокруг своего центра, а не вокруг точки соединения.

После вызова **glTranslate*()** для установки точки соединения и **glRotate*()** для поворота и присоединения куба, выполним перенос обратно к центру куба. До его рисования изменим его масштаб по осям. Вызовы **glPushMatrix()** и **glPopMatrix()** ограничат действие **glScale*()**. Для первого сегмента руки код будет выглядеть следующим образом **(**полный текст программы приводится в примере **3-7):**

```
glTranslatef(-1.0,0.0,0.0);
glRotatef((GLfloat) shoulder, 0.0,0.0,1.0);
glTranslatef(1.0,0.0,0.0);
glPushMatrix();
glScalef(2.0,0.4,1.0);
glutWireCube(1.0);
glPopMatrix();
```

Чтобы построить второй сегмент, требуется передвинуть локальную координатную систему к следующей точке соединения. Поскольку координатная система была предварительно повернута, ее ось **x**уже ориентирована вдоль повернутой руки. Таким образом, перенос вдоль оси **x** передвигает локальную координатную систему к следующей точке соединения. После перемещения к ней, вы можете использовать уже приводившийся выше код для рисования второго сегмента руки, так же как вы использовали его для рисования первого. Этот процесс может быть повторен для бесконечного числа сегментов **(**плечо, локоть, запястье, пальцы**).**

```
glTranslatef(1.0,0.0,0.0);
glRotatef((GLfloat) elbow,0.0,0.0,1.0);
glTranslatef(1.0,0.0,0.0);
glPushMatrix();
glScalef(2.0,0.4,1.0);
glutWireCube(1.0);
glPopMatrix();
```

Пример **3-7.** Рука робота**:** файл **robot.cpp**

```
#include <glut.h>

static intshoulder=0,elbow=0;
```

```
//Инициализация
void init(void)
{
   glClearColor(0.0,0.0,0.0,0.0);
   glShadeModel(GL_FLAT);
}

//Изменение размеров окна
void reshape(int w, int h)
{
   glViewport(0,0,(GLsizei) w, (GLsizei) h);
   glMatrixMode(GL_PROJECTION);
   glLoadIdentity();
   gluPerspective(65.0,(GLfloat) w/ (GLfloat) h,6.0,25.0);
   glMatrixMode(GL_MODELVIEW);
   glLoadIdentity();
   glTranslatef(0.0,0.0,-10.0);
}

//Отображение
void display(void)
{
   glClear(GL_COLOR_BUFFER_BIT);
   glPushMatrix();
      glTranslatef(-1.0,0.0,0.0);
      glRotatef((GLfloat)shoulder,0.0,0.0,1.0);
      glTranslatef(1.0,0.0,0.0);
      glPushMatrix();
         glScalef(2.0,0.4,1.0);
         glutWireCube(1.0);
      glPopMatrix();
      glTranslatef(1.0,0.0,0.0);
      glRotatef((GLfloat)elbow,0.0,0.0,1.0);
      glTranslatef(1.0,0.0,0.0);
      glPushMatrix();
         glScalef(2.0,0.4,1.0);
         glutWireCube(1.0);
      glPopMatrix();
   glPopMatrix();
   glutSwapBuffers();
}

//Реакция на клавиатуру
void keyboard(unsigned char key,int x, int y)
{
   switch(key)
   {
      case 's':
         shoulder=(shoulder+5)%360;
         glutPostRedisplay();
         break;
      case 'S':
         shoulder=(shoulder-5)%360;
         glutPostRedisplay();
         break;
      case 'e':
         elbow=(elbow+5)%360;
         glutPostRedisplay();
         break;
      case 'E':
         elbow=(elbow-5)%360;
         glutPostRedisplay();
         break;
      default:
         break;
   }
}

int main(int argc, char **argv)
{
   glutInit(&argc,argv);
   glutInitDisplayMode(GLUT_DOUBLE|GLUT_RGB);
   glutInitWindowSize(500,500);
```

```
    glutInitWindowPosition(100,100);
    glutCreateWindow("Robot Arm");
    init();
    glutDisplayFunc(display);
    glutReshapeFunc(reshape);
    glutKeyboardFunc(keyboard);
    glutMainLoop();
    return 0;
}
```

3.10 Трансформационный реверс

Конвейер геометрической визуализации весьма удобен для преобразования вершин объектов в оконные координаты, благодаря видовой и проекционной матрицам, а также порту просмотра. Однако бывают ситуации, когда вам требуется обратить этот процесс. Достаточно типичен случай, когда пользователь вашей программы использует мышь для выбора точки в трехмерном пространстве. Мышь возвращает двумерные координаты, соответствующие положению курсора на экране. В этом случае приложению нужно обратить трансформационный процесс, чтобы выяснить, какой точке в трехмерном пространстве соответствует это текущее положение курсора.

Такой трансформационный реверс выполняется функциями **gluUnProject()** и **gluUnProject4()** из библиотеки утилит. Получая трехмерные оконные координаты преобразованной вершины и данные обо всех преобразованиях, которые влияли на нее, **gluUnProject()** возвращает объектные координаты вершины в пространстве. (Если диапазон глубины в вашем приложении иной нежели ***[0, 1]***, используйте функцию **gluUnProject4().**)

```
int gluUnProject (GLdouble winx, GLdouble winy, GLdouble winz, const GLdouble
modelMatrix[16],
                  const GLdouble projMatrix[16], const GLint viewport[4],
                  GLdouble *objx, GLdouble *objy, GLdouble *objz);
```

Отображает заданные оконные координаты (***winx***, ***winy***, ***winz***) в объектные координаты, используя преобразования, заданные видовой матрицей (***modelMatrix***), проекционной матрицей (***projMatrix***) и портом просмотра (***viewport***). Результирующие объектные координаты возвращаются в параметрах ***objx***, ***objy*** и ***objz***. Функция возвращает значение **GL_TRUE**, индицируя успех операции, или **GL_FALSE**в случае неудачи (например, при наличии необратимой матрицы). Эта операция не пытается отсечь координаты по границе порта просмотра или уничтожить значения глубины, не заданные командой **glDepthRange()**.

В трансформационном реверсе существуют специфические сложности. Двумерная точка на экране реально может находиться где угодно на линии глубины в пространстве. Для возможности однозначного результата, **gluUnProject()** требует в качестве одного из своих параметров значение глубины в оконных координатах (***winz***), кроме того, это значение должно быть указано в терминах **glDepthRange()**. Для диапазона глубин по умолчанию вызов **gluUnProject()** с параметром ***winz*=0.0** вернет координаты соответствующей точки на ближней плоскости отсечения объема видимости, а при ***winz*=1.0** будет вычислена точка на дальней плоскости отсечения.

Пример **3-8** демонстрирует использование **gluUnProject()** для чтения позиции курсора в оконных координатах и распечатки на стандартном устройстве вывода соответствующих этой позиции трехмерных точек на ближней и дальней плоскостях отсечения. В самом окне приложения ничего не отображается.

Пример **3-8.** Обращение конвейера геометрической визуализации: файл **unproject.cpp**

```
#include <glut.h>
```

```
#include <stdlib.h>
#include <stdio.h>

//Изменение размеров окна
void reshape(int w, int h)
{
   glViewport(0,0,(GLsizei) w, (GLsizei) h);
   glMatrixMode(GL_PROJECTION);
   glLoadIdentity();
   gluPerspective(45.0,(GLfloat) w/ (GLfloat) h,1.0,100.0);
   glMatrixMode(GL_MODELVIEW);
   glLoadIdentity();
}

//Отображение
void display(void)
{
   glClear(GL_COLOR_BUFFER_BIT);
   glFlush();
}

//Реакция на мышь
void mouse(int button,int state,int x, int y)
{
   GLint viewport[4];
   GLdouble mvmatrix[16], projmatrix[16];

   //y – координата в OpenGL
   GLint realy;

   //Возвращаемые объектные x, y, z координаты
   GLdouble wx,wy,wz;

   switch(button)
   {
      case GLUT_LEFT_BUTTON:
         if (state==GLUT_DOWN)
         {
            glGetIntegerv(GL_VIEWPORT,viewport);
            glGetDoublev(GL_MODELVIEW_MATRIX,mvmatrix);
            glGetDoublev(GL_PROJECTION_MATRIX,projmatrix);

            //viewport[3] - высоте окна в пикселях
            realy=viewport[3]-(GLint)y-1;
            printf("Координаты в позиции курсора (%4d,%4d)\n",x,realy);

gluUnProject((GLdouble)x,(GLdouble)realy,0.0,mvmatrix,projmatrix,viewport,&wx,&wy,&wz);
            printf("Объектные координаты при z=0 (%f,%f,%f)\n",wx,wy,wz);

gluUnProject((GLdouble)x,(GLdouble)realy,1.0,mvmatrix,projmatrix,viewport,&wx,&wy,&wz);
            printf("Объектные координатыпри z=1 (%f,%f,%f)\n",wx,wy,wz);
         }
      break;
      case GLUT_RIGHT_BUTTON:
         if (state==GLUT_DOWN)
            exit(0);
      break;
   }
}

int main(int argc, char **argv)
{
   glutInit(&argc,argv);
   glutInitDisplayMode(GLUT_SINGLE|GLUT_RGB);
   glutInitWindowSize(500,500);
   glutInitWindowPosition(100,100);
   glutCreateWindow("Reversing the Geometric Processing Pipeline");
   glutDisplayFunc(display);
   glutReshapeFunc(reshape);
   glutMouseFunc(mouse);
   glutMainLoop();
   return 0;
}
```

В **GLU** версии **1.3** появилась модифицированная версия **gluUnProject(). gluUnProject4()** может обрабатывать нестандартные величины диапазона глубин и ***w–*** координаты не равные ***1*.**

```
int gluUnProject4 (GLdouble winx, GLdouble winy, GLdouble winz,
                   GLdouble clipw, const GLdouble modelMatrix[16], const GLdouble
projMatrix[16], const GLint viewport[4],
                   GLclampd znear, GLclampd zfar, GLdouble *objx, GLdouble *objy,
GLdouble *objz, GLdouble *objw);
```

В общем**,** эта команда выполняет операцию аналогичную **glUnProject().** Отображает заданные оконные координаты **(*winx*, *winy*, *winz*)** в объектные координаты**,** используя преобразования**,** заданные видовой матрицей **(*modelMatrix*)**, проекционной матрицей **(*projMatrix*)**, портом просмотра **(*viewport*)** и диапазоном глубин ***znear*** и ***zfar*.** Результирующие объектные координаты возвращаются в параметрах ***objx*, *objy*, *objz*** и ***objw*.**

Еще одной функцией из библиотеки утилит**,** связанной с **gluUnProject()** и **gluUnProject4()** является функция **gluProject(). gluProject()** имитирует действие конвейера визуализации**.** Получая трехмерные объектные координаты и все преобразования**,** которые влияют на них**, gluProject()** возвращает преобразованные оконные координаты**.**

```
int gluProject (GLdouble objx, GLdouble objy, GLdouble objz,
                const GLdouble modelMatrix[16], const GLdouble projMatrix[16],
const GLint viewport[4],
                GLdouble *winx, GLdouble *winy, GLdouble *winz);
```

Отображает заданные объектные координаты **(*objx*, *objy*, *objz*)** в оконные координаты с использованием преобразований**,** заданных видовой матрицей **(*modelMatrix*)**, проекционной матрицей **(*projMatrix*)** и портом просмотра **(*viewport*).** Результирующие оконные координаты возвращаются в параметрах ***winx*, *winy*** и ***winz*.** Функция возвращает **GL_TRUE** в случае успеха и **GL_FALSE** в противном случае**.**

Замечание: Матрицы, передаваемые **gluUnProject()**, **gluUnProject4()** и **gluProject()** задаются в формате стандартном для OpenGL, то есть по столбцам. Вы можете использовать **glGetDoublev()** и **glGetIntegerv()** с параметрами GL_MODELVIEW_MATRIX, GL_PROJECTION_MATRIX и GL_VIEWPORT для получения текущей видовой матрицы, матрицы проекции и порта просмотра соответственно, а затем использовать полученные величины при вызове **gluUnProject()**, **gluUnProject4()** или **gluProject()**.

Глава 4. Цвет

Целью практически всех приложений **OpenGL**является отображение цветного изображения в окне на экране**.** Окно **–** это прямоугольный массив пикселей**,** каждый из которых содержит и отображает свой собственный цвет**.** Таким образом**,** интуитивно**,** смысл всех вычислений**,** производимых реализацией **OpenGL –** вычислений**,** принимающих в расчет команды **OpenGL,** информацию о состоянии и значения параметров **–** определить результирующий цвет каждого пикселя**,** который должен быть нарисован в окне**.**

4.1 Цветовое восприятие

Физически**,** свет состоит из протонов **–** микроскопических световых частиц**,** каждая из которых движется по собственному маршруту и вибрирует со своей частотой **(**или длиной волны**,** или энергией **–** каждая из трех характеристик**:** частота**,** длина волны или энергия однозначно определяет две другие**).** Протон полностью характеризуется своим положением**,** направлением и частотой**/**длиной волны**/**энергией**.** Протоны с

длиной волны приблизительно от **390** нанометров **(nm) (**фиолетовый**)** до **720 nm (**красный**)** покрывают все цвета видимого спектра, формируя цвета радуги **(**фиолетовый, синий, голубой, зеленый, желтый, оранжевый и красный**).** Однако наши глаза воспринимают множество цветов, которых нет в радуге – например, белый, черный, коричневый, розовый и так далее. Каким образом это происходит**?**

В действительности наш глаз видит смесь протонов с различными частотами. Реальные источники света характеризуются распределением частот излучаемых ими протонов. Идеально белый свет состоит из равного количества света всех частот. Лазерный луч обычно очень плотный, и все его протоны практически идентичны по частоте **(**а также по направлению и фазе**).** В обычной лампочке больше света на желтой частоте. Свет от большинства звезд во вселенной имеет распределение, сильно зависящее от их температуры. Частота распределения света от большинства обыденных источников более сложна.

Человеческий глаз воспринимает цвет, когда несколько ячеек в сетчатке **(**называемых колбочками**)** возбуждаются вследствие того, что по ним бьют протоны. Три различных типа колбочек лучше реагируют на три различных длины световой волны: первый тип лучше реагирует на красный свет, второй – на зеленый, и третий – на синий. **(**У человека, не способного различать цвета, как правило, отсутствует один из типов колбочек.**)** Когда смесь протонов попадает в глаз, колбочки сетчатки регистрируют различные уровни возмущения в соответствии со своим типом. Если после этого в глаз попадает иная смесь протонов, возмущающая его с уровнем идентичным первой смеси, то цвет этих двух смесей неразличим.

Поскольку каждый цвет фиксируется глазом в виде уровней возмущения колбочек входящими протонами, глаз может воспринимать цвета, не являющиеся частью спектра, создаваемого призмой или радугой. Например, если направить смесь красных и синих протонов на сетчатку так, чтобы возмущение было зафиксировано красными и синими колбочками, ваш глаз определит это как лиловый цвет, которого нет в спектре. Другие комбинации могут дать коричневый цвет, цвет морской волны и другие, отсутствующие в спектре.

Графический монитор эмулирует видимые цвета, подсвечивая пиксели на экране комбинацией красного, зеленого и синего света в пропорциях, возбуждающих колбочки чувствительные к красному, зеленому и синему свету, таким образом, чтобы уровень их возмущения совпадал с уровнем, создаваемым естественной смесью протонов. Если бы у людей было больше типов колбочек, например, если были бы колбочки чувствительные к желтому свету, цветные мониторы имели бы, вероятно, еще и желтую пушку, и мы использовали бы четверку **RGBY (**красный, зеленый, синий, желтый**)** для указания цвета. И, наконец, если бы все люди не различали цветов, эта глава была бы намного проще.

Для отображения конкретного цвета, монитор посылает точное количество красного, зеленого и синего света **(red**, **green**, **blue – RGB)** должным образом стимулирующее различные типы колбочек в глазу. Цветной монитор может посылать свет с разными пропорциями красного, зеленого и синего в каждую точку экрана, и глаз видит миллионы световых точек, каждая из которых имеет свой собственный цвет.

Замечание: Помимо RGB существует множество других представлений цвета или цветовых моделей, обозначаемых как HLS, HSV, CMYK и так далее. Если вам требуется, чтобы цветовые данные были записаны в одном из этих форматов, вы всегда можете конвертировать их из RGB – формата или в RGB – формат.

В этой главе рассматривается только восприятие глазом комбинации входящих в него протонов. Ситуация, когда свет входит в глаз, отразившись от каких-либо материалов еще более сложна – например, белый цвет, отразившись от красного мяча, выглядит красным, а желтый свет, проходя через синее стекло, становится почти черным.

4.2 Цвет в компьютере

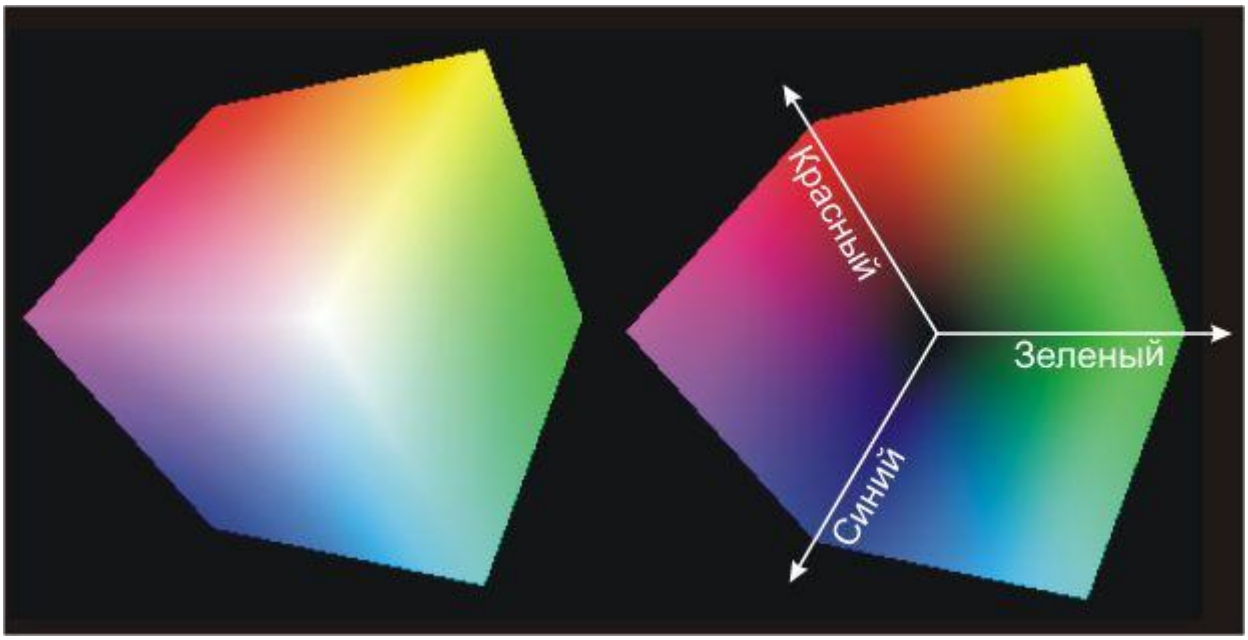
Аппаратура заставляет каждый пиксель экрана излучать различные количественные соотношения красного, зеленого и синего света. Эти количества называются **R** (красный), **G** (зеленый) и **B** (синий) – величинами. Они часто хранятся и упаковываются вместе (иногда вместе с четвертой – альфа – величиной или альфа – компонентой, называемой **A**), и упакованное значение называется **RGB** (или **RGBA**) величиной. Цветовая информация для каждого пикселя может храниться как в ***RGBA*** – *режиме* (при котором для каждого пикселя хранятся значения **R**, **G**, **B** и, возможно **A**), так и в *режиме цветовых индексов* (в этом случае для каждого пикселя хранится всего одно число, называемое *цветовым индексом* или *индексом в палитре*). Каждый цветовой индекс идентифицирует одно вхождение в таблицу, содержащую в каждом элементе набор из одной **R**, одной **G** и одной **B** – величины. Такая таблица называется *цветовой таблицей* (или *цветовой картой*, или просто *палитрой*).

В индексном режиме вам может понадобиться изменить величины в цветовой таблице. Поскольку цветовые таблицы управляются оконной системой, для этого не существует команд **OpenGL**. Все примеры в данном пособии инициализируют цветовой режим при открытии окна, применяя функции из библиотеки **GLUT**.

Между разными аппаратными графическими платформами существует множество различий, касающихся как размера пиксельного массива, так и количества цветов, которые могут быть отображены в каждом пикселе. В любой графической системе каждый пиксель имеет одинаковое количество памяти для хранения своего цвета, а вся эта память для всех пикселей вместе называется *цветовым буфером*. Размер буфера обычно измеряется в битах, таким образом, **8**-битовый буфер может содержать **8** бит данных (то есть одну из **256** возможных комбинаций, задающих конкретный цвет) для каждого пикселя. Размер возможных буферов меняется от компьютера к компьютеру.

Каждое из **R**, **G** и **B** – значений может изменяться от **0.0** (отсутствие интенсивности) до **1.0** (полная интенсивность). Например, комбинация **R=0.0**, **G=0.0** и **B=1.0** задает максимально возможный по яркости синий цвет. Если **R**, **G** и **B** равны **0.0**, цвет пикселя – черный; если все компоненты равны **1.0**, пиксель будет нарисован самым ярким белым цветом, который возможно отобразить на экране. Смешивание синего и зеленого цветов дает оттенки голубого. Комбинация синего и красного цветов дает лиловый. Красный и зеленый образуют желтый. Чтобы облегчить себе задачу по созданию желаемых цветов из **R**, **G** и **B** – компонент, взгляните на рисунок **4.1**. Оси куба на рисунке представляют интенсивности красного, зеленого и синего.

Рисунок **4-1**. Цветовой куб



Команды, используемые для указания цвета объекта (в данном случае точки), могут быть достаточно просты:

```
glColor3f(1.0,0.0,0.0);   //Текущий цвет - красный, без зеленого и без синего
glBegin(GL_POINTS);
   glVertex3fv(point_array);
glEnd();
```

В некоторых режимах (например, если производятся вычисления, связанные с освещением или текстурированием), назначенный цвет может преобразовываться другими операциями до того, как попадет в буфер кадра, представляя цвет пикселя на экране. Вообще говоря, цвет пикселя определяется посредством длинной последовательности операций.

Вначале исполнения программы цветовой режим устанавливается в **RGBA** или индексный. Как только цветовой режим установлен, он уже не может быть изменен. Во время выполнения программы цвет (и в индексном, и в **RGBA** - режиме) определяется на повершинном базисе для каждого геометрического примитива. Этот цвет может быть тем, который вы задали для вершины непосредственно, или, если включено освещение, тем, который определяется под влиянием взаимодействия трансформационных матриц с нормалями к поверхности и других свойств материала. Иными словами, красный мяч, освещенный синим светом, выглядит иначе, чем тот же мяч, но без освещения. После выполнения расчетов, связанных с освещением, применяется выбранная модель закраски. Вы можете выбрать плоскую или плавную закраску, каждая из которых оказывает специфическое влияние на цвет пикселя.

Далее, примитивы *растеризуются* (или преобразуются в двумерное изображение). Растеризация требует определения того, какие квадраты целочисленной решетки оконных координат заняты примитивом. Этим квадратам присваивается определенный цвет и другие величины. Квадрат решетки вместе с присвоенным ему значением цвета, ***z*** (глубиной) и координатами текстуры называется *фрагментом*. Пиксели являются элементами буфера кадра; фрагмент поступает из примитива и комбинируется с соответствующим ему пикселем для получения нового пикселя. После того, как фрагменты примитива построены, к ним применяются (если включены) текстурирование, туман и антиалиасинг. Затем над фрагментами и пикселями, уже находящимися в буфере кадра, выполняются альфа наложение, *микширование* (***dithering***– техника симуляции недостающих цветов) и побитовые логические операции (если какие-либо или все эти механизмы включены). Наконец, цветовая величина

фрагмента **(**и в индексном**,** и в **RGBA -** режиме**)** записывается в пиксель и отображается в окне с использованием цветового режима окна**.**

4.3 RGBA – режим против Индексного режима

В обоих режимах **(**индексном и **RGBA)** в каждом пикселе хранится определенное количество информации о цвете**.** Это количество определяется числом битовых плоскостей в буфере кадра**.** *Битовая плоскость* содержит **1** бит данных для каждого пикселя**.** Если присутствует **8** битовых плоскостей**,** в них суммарно хранится **8** бит на цвет и**,** следовательно**,** может существовать $2^8 = 256$ их различных комбинаций или цветов**,** которые могут быть сохранены в пикселе**.**

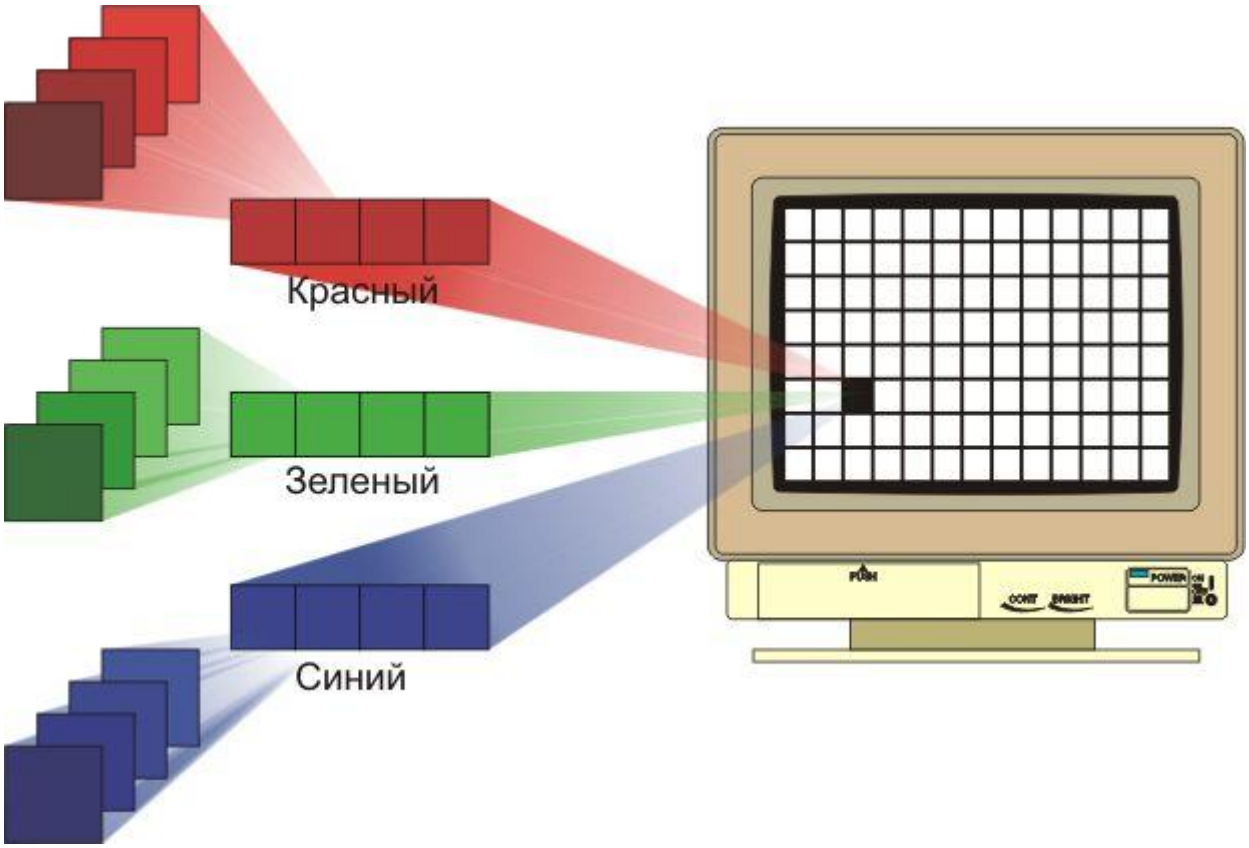
Битовые плоскости часто поровну разделяют на хранилища **R, G** и **B** компонент **(**то есть**,** система с **24** битовыми плоскостями содержит по **8** бит для красного**,** зеленого и синего**),** но это не всегда соответствует действительности**.** Для выяснения числа битовых плоскостей**,** присутствующих в вашей системе для величин красного**,** зеленого**,** синего цвета**,** альфа величин и величин цветовых индексов вызовите **glGetIntegerv() c** аргументами **GL_RED_BITS, GL_GREEN_BITS, GL_BLUE_BITS, GL_ALPHA_BITS** и **GL_INDEX_BITS** соответственно**.**

Замечание: Цветовые интенсивности на большинстве компьютерных экранов не воспринимаются человеческим глазом, как непрерывные. Предположим, что цвета состоят только из красного компонента с зеленым и синим, установленными в нуль. С увеличением интенсивности от 0.0 (полное отсутствие интенсивности) до 1.0 (полная интенсивность) увеличивается число электронов, ударяющихся в пиксель экрана. Однако возникает вопрос: будет ли цвет с интенсивностью 0.5 находится (по внешнему виду) ровно посередине между 0.0 и 1.0? Чтобы выяснить это, напишите программу, заполняющую прямоугольную область в шахматном порядке пикселями с интенсивностями красного равной 0.0 и 1.0, а рядом – прямоугольный регион, целиком заполненный пикселями с интенсивностью красного равной 0.5. На некоторой разумной дистанции от экрана должно казаться, что оба региона имеют одинаковую интенсивность. Если они выглядят сильно различающимися, вам следует использовать любой механизм коррекции, имеющийся в вашей конкретной системе. Например, во многих системах присутствует таблица настройки интенсивностей, позволяющая настроить внешний вид 0.5, таким образом, чтобы оно действительно выглядело серединой между 0.0 и 1.0. Обычно применяемый для этого механизм коррекции имеет экспоненциальный характер, а на экспоненту ссылаются как на гамму (отсюда термин гамма-коррекция). Использование одинаковой гаммы для красного, зеленого и синего компонент дает обычно неплохой результат, но три разные гаммы могут дать лучший.

4.3.1 RGBA – режим

В **RGBA –** режиме аппаратура выделяет некоторое число битовых плоскостей для каждого из **R, G, B** и **A** компонента **(**это число не обязательно является одинаковым для каждого компонента**),** как показано на рисунке **4-2. R, G** и **B –** величины чаще хранятся в виде целых чисел**,** чем в виде чисел с плавающей точкой и**,** следовательно**,** масштабируются согласно выделенному количеству битовых плоскостей**.** Например**,** если в системе имеется **8** бит для красного компонента**,** в них могут быть сохранены целые от **0** до **255.** Таким образом**, 0, 1, 2, ..., 255** в битовых плоскостях будут соответствовать **R –** величинам **0/255=0.0, 1/255, 2/255, ..., 255/255=1.0.** Независимо от числа битовых плоскостей **0.0** задает минимальную интенсивность**,** а **1.0 –** максимальную**.**

Рисунок **4-2. RGB –** величины из битовых плоскостей



Замечание: Значение альфа (A в RGBA) не имеет прямого действия на цвет, отображаемый на экране. Оно может быть использовано для множества вещей, включая цветовое наложение и прозрачность, а также для воздействия на записываемые в буфер величины R, G и B.

Количество различных цветов, которые могут быть отображены на месте одного пикселя, зависит от количества битовых плоскостей и возможностей аппаратуры по интерпретации этих плоскостей. Количество одновременно отображаемых цветов не может превышать 2^n, где ***n*** – количество битовых плоскостей. Таким образом, система с **24**-мя битовыми плоскостями может одновременно отображать до **16.77** миллионов различных цветов.

4.3.1.1 Цветовое микширование

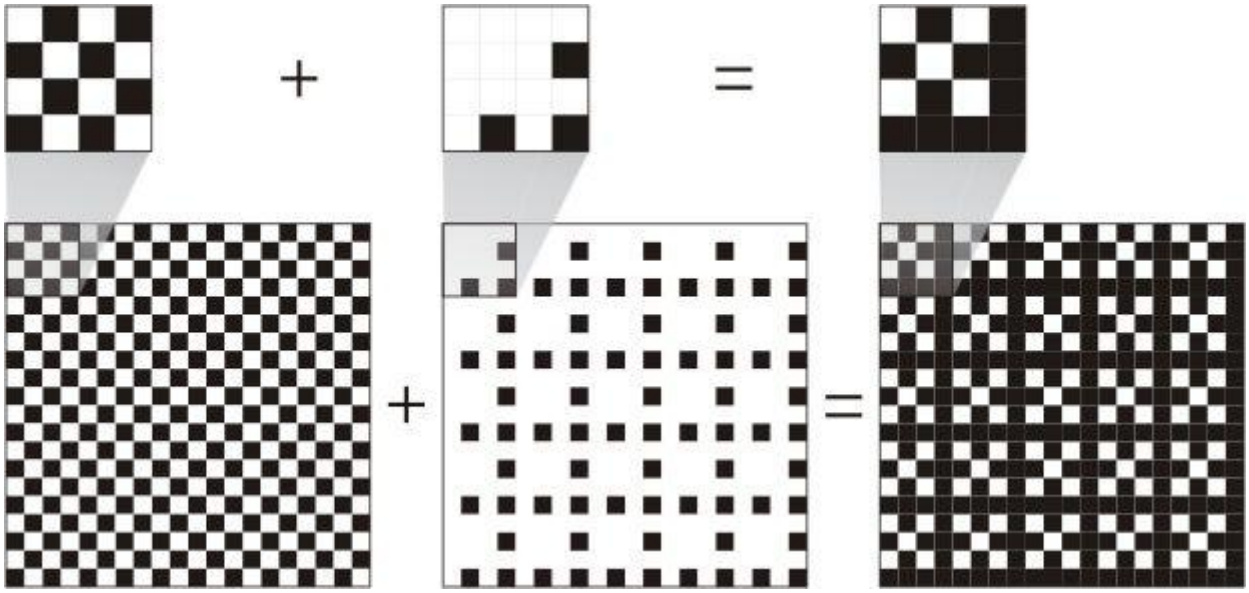
Дополнительно: Некоторые аппаратные средства используют микширование для увеличения числа видимых цветов. Микширование – это техника, заключающаяся в использовании комбинации нескольких цветов для создания видимости других цветов. Для иллюстрации того, как работает микширование, представим, что в вашей системе имеется всего по **1** биту для **R**, **G** и **B**, и она, таким образом, может отображать только **8** цветов: черный, белый, красный, синий, зеленый, желтый, голубой и фиолетовый. Чтобы отобразить область, закрашенную розовым цветом, аппаратура может заполнить ее красными и белыми пикселями по принципу шахматной доски (пиксели через один по вертикали и горизонтали имеют один и тот же цвет, а соседние пиксели – разные цвета). Если ваш глаз находится на достаточном расстоянии от экрана, на котором он не может различить отдельные пиксели – область будет казаться розовой, то есть средним цветом между белым и красным. Для увеличения «красноты» розового цвета можно изменить соотношение между цветами, увеличив количество красных пикселей и уменьшив количество белых. Для более «бледного» розового следует поступить наоборот – увеличить число белых пикселей и уменьшить количество красных.

При использовании этой техники на экране нет розовых пикселей вообще. Единственный способ увидеть розовый цвет – покрыть область, состоящую из

множества пикселей, в этом случае вы не сможете зрительно выделить отдельный пиксель. Если вы задаете **RGB-**величину отсутствующего, недоступного цвета и заполняете им полигон, аппаратура заполнит пиксели внутри полигона смесью доступных цветов, среднее между которыми наиболее близко к желаемому вами цвету. **(**Помните, однако, что если вы считаете пиксельную информацию из буфера кадра, вы получите именно красные и белые пиксели, так как розовых там нет.**)**

Рисунок **4-3** иллюстрирует несколько примеров микширования черного и белого цветов для получения трех оттенков серого. Слева направо в верхнем ряду фрагменты размером **4x4** пикселя представляют шаблоны микширования для **50** процентного, **19** процентного и **69** процентного серого цвета. Под каждым шаблоном вы можете видеть его уменьшенную и многократно повторенную копию, однако черные и белые квадраты на них все же больше, чем большинство пикселей. Если вы посмотрите на эти шаблоны с достаточно большого расстояния, вы обратите внимание, что отдельные квадратики на них сливаются и, таким образом, образуют три оттенка серого цвета.

Рисунок **4-3.** Микширование черного и белого для создания видимости серого



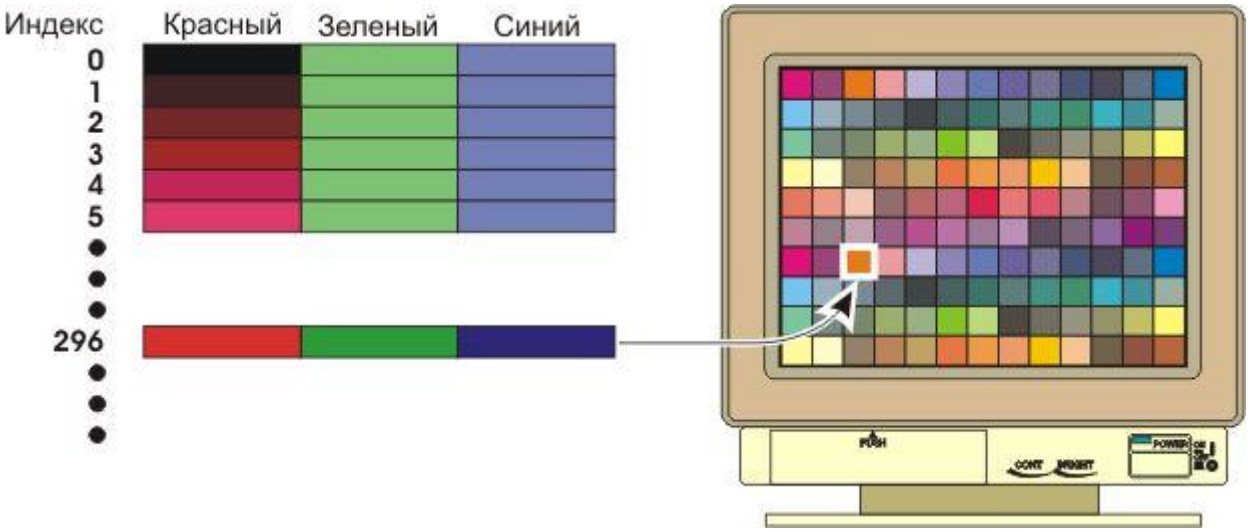
При наличии **8** бит на каждый из **R**, **G** и **B** компонентов вы можете получить достаточно высококачественное изображение и без микширования. Однако то, что на вашей машине имеется **24** битовых плоскости, еще не говорит о том, что микширование вам совсем не нужно. Например, если вы работаете в режиме двойной буферизации, битовые плоскости могут быть разделены на **2** набора по **12** плоскостей каждый, то есть на самом деле имеется только **4** бита для каждого из **R**, **G** и **B** компонент. Без микширования цвет с **4-**мя битами на компонент во многих ситуация может давать менее удовлетворительные результаты.

Вы включаете и выключаете микширование, передавая аргумент **GL_DITHER** командам **glEnable()** и **glDisable()**. Имейте в виду, что микширование, в отличие от большинства возможностей **OpenGL**, по умолчанию включено.

4.3.2 Индексный режим

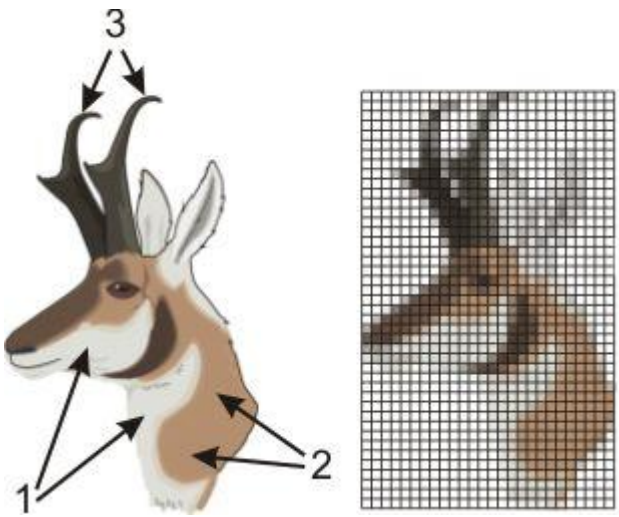
При работе в индексном цветовом режиме **OpenGL** использует цветовую таблицу, похожую на палитру, на которой заранее смешивают краски для того, чтобы потом нарисовать картинку определенным количеством цветов. В палитре художника имеются участки для смешивания красок, похожим образом компьютерная цветовая таблица предоставляет индексы ячеек, в которых хранятся цвета, заранее смешиваемые из красного, зеленого и синего компонентов **(**рисунок **4-4).**

Рисунок **4-4.** Цветовая таблица



Художник при рисовании ограниченным количеством цветов выбирает один из них на палитре и заполняет им область на картине. Компьютер сохраняет в битовых плоскостях цветовой индекс для каждого пикселя. Затем величины в битовых плоскостях разрешаются: каждый пиксель рисуется комбинацией красного, зеленого и синего цветов, извлеченной из ячейки цветовой таблицы, соответствующей хранимому индексу (рисунок **4-5).**

Рисунок **4-5.** Рисование картинки с использованием цветовой таблицы



При использовании индексного цветового режима количество одновременно доступных цветов ограничено размером цветовой таблицы и количеством имеющихся битовых плоскостей. Размер цветовой таблицы зависит от объема аппаратной поддержки. Размер цветовой таблицы всегда является степенью двойки и обычно варьируется от $256(2^8)$ до $4096\,(2^{12})$, где показатель представляет собой количество используемых битовых плоскостей. Если 2^n – количество индексов в цветовой таблице, а ***m*** – число доступных битовых плоскостей, то количество возможных к использованию элементов или вхождений в таблицу равно меньшему из 2^n и 2^m.

В **RGBA–** режиме цвет каждого пикселя не зависит от других пикселей. Однако в индексном режиме все пиксели с одинаковым индексом (хранящимся в битовых плоскостях) разделяют один и тот же элемент в цветовой таблице. Если изменяется

содержимое одного из элементов в таблице, то все пиксели, индексы которых ссылаются на этот элемент, изменяют свой цвет.

4.3.3 Выбор между RGBA – режимом и индексным режимом

В своем выборе между **RGBA** - режимом и индексным режимом вы должны основываться на том, какая имеется аппаратура, а также на том, что именно требуется вашему приложению. Большинство систем в **RGBA** – режиме позволяют одновременно отображать большее число цветов, чем в индексном. Кроме того, при использовании некоторых эффектов, таких как заливка, освещение, текстурирование и туман, **RGBA** предоставляет большую гибкость, чем индексный режим.

Индексный цветовой режим может быть предпочтительнее в следующих случаях:

- Если вы переносите (портируете) приложение, интенсивно использующее индексный режим, вероятно, будет проще не переделывать его под режим **RGBA**.
- Если у вас имеется в наличии небольшое количество битовых плоскостей, **RGBA** режим может создавать достаточно резкие оттенки цвета. Например, если у вас всего **8** битовых плоскостей, в **RGBA** режиме будет всего **3** бита для красного цвета, **3** для зеленого и **2** для синего (так как человеческий глаз наименее чувствителен именно к синему цвету). Таким образом, вы сможете отображать только **8** (2^3) оттенков красного, **8** оттенков зеленого и **4** оттенка синего цвета. Разрывы между цветовыми оттенками будут весьма заметными. В подобной ситуации, если у вас ограниченные требования, вы можете использовать цветовую таблицу, чтобы загрузить большее число цветовых оттенков. Например, если вам требуются только оттенки синего, вы можете использовать индексный режим и сохранить в цветовой таблице до **256** (2^8) оттенков синего цвета, что будет намного лучше **4**-ех оттенков в **RGBA** режиме. Естественно, в этом примере вы задействуете под оттенки синего всю цветовую таблицу, поэтому у вас не будет оттенков ни красного, ни зеленого цвета, ни каких-либо их комбинаций.
- Индексный режим может быть использован для различных специфических трюков, например, для анимации цветовой таблицы и рисования в слоях.

Вообще, используйте **RGBA** режим везде, где это возможно. Только он работает с текстурированием, а такие механизмы, как освещение, заливка, туман, антиалиасинг и цветовое наложение работают в **RGBA** режиме намного лучше. (Кроме того, индексный режим может быть недоступен на некоторых платформах, например, приложения **GLUT**, использующие индексный режим могут не запускаться в операционной системе **Microsoft Windows XP.**)

4.3.4 Изменение режима

В лучшем из всех возможных миров вам не пришлось бы делать выбор между **RGBA** и индексным режимами. Например, вы могли бы использовать индексный режим для эффектов, связанных с анимацией цветовой таблицы, а затем по необходимости мгновенно переключить сцену в **RGBA** режим для текстурирования.

Или, похожим образом, вы могли бы переключаться между режимами одинарной и двойной буферизации. Например, у вас всего несколько битовых поверхностей, скажем **8**. Вы можете использовать **256** (2^8) цветов в режиме одинарной буферизации, но если вы работаете в режиме двойной буферизации, чтобы устранить мигание в своей программе анимации, у вас может быть только **16** (2^4) цветов. Возможно, вы хотели бы нарисовать движущийся объект без мигания, то есть принести цвета в жертву возможности двойной буферизации (может быть, объект движется настолько быстро, что наблюдатель не заметит деталей). Однако, когда объект приходит в стационарное

состояние, вам захочется рисовать его в режиме одинарной буферизации, чтобы использовать большее количество цветов.

К сожалению, большинство оконных систем не предусматривает легкого переключения между режимами. Например, в **X Windows System** цветовой режим является атрибутом **X Visual. X Visual**, в свою очередь должен быть задан до того, как создано окно. Как только он задан, он не может быть изменен на протяжении времени существования окна. То есть если вы создали окно с двойной буферизацией и цветовым режимом **RGBA** – вы так с ними и останетесь до закрытия окна вашего приложения.

Достаточно хитрое решение этой проблемы заключается в создании более чем одного окна с различными режимами. Вы должны контролировать видимость каждого окна и рисовать объект в нужном – видимом окне.

4.4 Указание цвета и модели закраски

OpenGL управляет цветом через переменные состояния: текущий цвет (в **RGBA** режиме) и текущий цветовой индекс (в индексном режиме). Если только вы не используете более сложную модель закраски, например, применяя освещение или текстурирование, каждый объект рисуется с использованием текущего цвета (или текущего цветового индекса). Посмотрите на следующий псевдокод:

```
установить_цвет(красный);
нарисовать_объект (A);
нарисовать_ объект (B);
установить_цвет(зеленый);
установить_цвет(синий);
нарисовать_ объект (C);
```

Объекты **A** и **B** будут нарисованы красным цветом, а **C** – синим. Четвертая линия, устанавливающая текущий цвет в зеленый, не дает никакого результата (кроме задержки по времени). Если выключено освещение и текстурирование, то после того, как установлен текущий цвет, все объекты, рисуемые далее, рисуются именно этим цветом, и так происходит до того, как текущий цвет изменяется.

4.4.1 Указание цвета в RGBA режиме

В режиме **RGBA** для выбора текущего цвета используйте команды **glColor*()**.

```
void glColor3{b s i f d ub us ui} (TYPE r, TYPE g, TYPE b);
void glColor4{b s i f d ub us ui} (TYPE r, TYPE g, TYPE b, TYPE a);
void glColor3{b s i f d ub us ui}v (const TYPE *v);
void glColor4{b s i f d ub us ui}v (const TYPE *v);
```

Устанавливает текущие красную, зеленую, синюю и альфа величины. Эта команда может иметь до трех суффиксов, которые разделяют команды по вариациям принимаемых параметров. Первый суффикс (**3** или **4**) является индикатором того, передаете ли вы кроме красной, зеленой и синей величин еще и величину альфа. Если вы не задаете альфа величину, ей автоматически присваивается значение ***1.0***. Второй суффикс определяет тип данных для параметров: байт **(byte – b)**, короткое целое **(short – s)**, целое **(integer – i)**, число с плавающей точкой **(float – f)**, число с плавающей точкой двойной точности **(double – d)**, беззнаковый байт **(unsigned byte – ub)**, беззнаковое короткое целое **(unsigned short – us)** или беззнаковое целое **(unsigned integer – ui)**. Третий суффикс **(v)** является опциональным и индицирует, что аргументы передаются в виде указателя на массив величин указанного типа.

Для версий **glColor*()**, работающих с числами с плавающей точкой, значения параметров обычно должны принадлежать диапазону от ***0.0*** до ***1.0*** – диапазону от

минимума до максимума величин, которые могут быть сохранены в буфере кадра. Компоненты, указываемые в виде беззнаковых целых, линейно интерполируются в числа с плавающей точкой таким образом, чтобы максимально допустимая для этого типа величина была представлена числом ***1.0*** (полная интенсивность), а ***0*** – числом ***0.0*** (нулевая интенсивность). Компоненты задаваемые в виде знаковых целых линейно интерполируются до величин с плавающей точкой таким образом, чтобы максимально допустимое положительное значение для данного типа величина была представлена числом ***1.0***, а минимально допустимая – числом ***-1.0*** (таблица **4-1**).

Ни величины с плавающей точкой, ни целые не усекаются до диапазона **[*0, 1*]** до обновления текущего цвета или параметров материала. Результирующие значения цвета, выпадающие за границы диапазона **[*0, 1*]**, усекаются до него после проведения расчетов, связанных с освещением, до интерполяции или записи в буфер кадра. Даже если механизм освещения выключен, цветовые компоненты усекаются перед растеризацией.

Таблица **4-1.** Конверсия величин цветовых компонентов в числа с плавающей точкой

Суффикс	Тип данных	Минимально допустимая величина	Значение соответствующее минимальной величине	Максимально допустимая величина	Значение соответствующее максимальной величине
b	1-byte integer	-128	-1.0	127	1.0
s	2-byte integer	-32,768	-1.0	32,768	1.0
i	4-byte integer	-2,147,483,648	-1.0	2,147,483,648	1.0
ub	unsigned 1-byte integer	0	0.0	255	1.0
us	unsigned 2-byte integer	0	0.0	65,535	1.0
ui	unsigned 4-byte integer	0	0.0	4,294,967,295	1.0

4.4.2 Указание цветов в индексном режиме

В индексном режиме используйте команду **glIndex*()** для выбора одной величины индекса в качестве текущей.

```
void glIndex{s i f d ub} (TYPE c);
void glIndex{s i f d ub}v (const TYPE *c);
```

Устанавливает индекс текущего цвета равным ***c***. Первый суффикс этой команды индицирует тип данных для параметров: короткое целое **(short – s)**, целое **(integer – i)**, число с плавающей точкой **(float – f)**, число с плавающей точкой двойной точности **(double – d)**, беззнаковый байт **(unsigned byte – ub)**. Второй опциональный суффикс **(v)** показывает, что аргумент является массивом указанного типа (массив в данном случае содержит только одну величину).

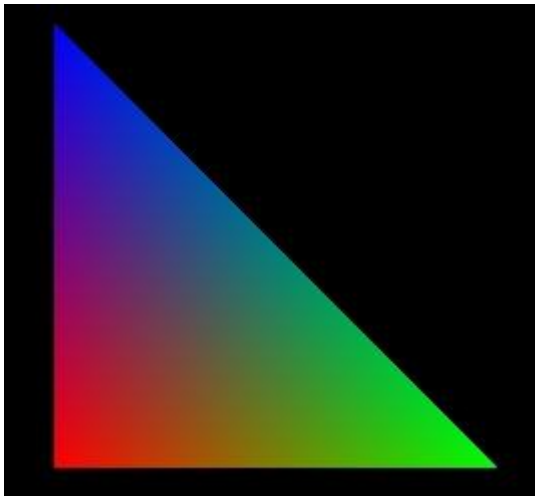
В разделе «Очистка окна» главы **2** вы видели спецификацию команды **glClearColor()**. Для индексного режима также имеется соответствующая команда **glClearIndex()**.

```
void glShadeModel (GLenum mode);
```

Устанавливает модель заливки. Параметр ***mode***может принимать значения **GL_SMOOTH (**плавная заливка **–** режим по умолчанию**)** или **GL_FLAT (**плоская заливка**).**

При использовании плоской заливки цвет одной отдельной вершины независимого примитива дублируется для всех остальных вершин при визуализации этого примитива. При плавной заливке цвет каждой вершины считается индивидуальным. Для линии цвет на протяжении отрезка интерполируется на основании цветов на его концах. Для полигона цвета его внутренней области интерполируются между цветами его вершин. В примере **4-1** рисуется плавно залитый треугольник, показанный на рисунке **4-6.**

Рисунок **4-6.** Плавно залитый треугольник



Пример **4-1.** Рисование плавно залитого треугольника: файл **smooth.cpp**

```
#include <glut.h>

//Инициализация
void init(void)
{
   glClearColor(0.0,0.0,0.0,0.0);
   glShadeModel(GL_SMOOTH);
}

//Изменение размеров окна
void reshape(int w, int h)
{
   glViewport(0,0,(GLsizei) w, (GLsizei) h);
   glMatrixMode(GL_PROJECTION);
   glLoadIdentity();
   if (w<=h)
      gluOrtho2D(0.0,30.0,0.0,30.0*(GLfloat)h/(GLfloat)w);
   else
      gluOrtho2D(0.0,30.0*(GLfloat)w/(GLfloat)h,0.0,30.0);
   glMatrixMode(GL_MODELVIEW);
}

//Рисуем треугольник
void triangle(void)
{
   glBegin(GL_TRIANGLES);
      glColor3f(1.0,0.0,0.0);
      glVertex2f(5.0,5.0);
      glColor3f(0.0,1.0,0.0);
      glVertex2f(25.0,5.0);
      glColor3f(0.0,0.0,1.0);
      glVertex2f(5.0,25.0);
   glEnd();
}
```

```
//Отображение
void display(void)
{
   glClear(GL_COLOR_BUFFER_BIT);
   triangle();
   glFlush();
}

int main(int argc, char** argv)
{
   glutInit(&argc,argv);
   glutInitDisplayMode(GLUT_SINGLE|GLUT_RGB);
   glutInitWindowSize(500,500);
   glutInitWindowPosition(100,100);
   glutCreateWindow("Drawing a Smooth-Shaded Triangle");
   init();
   glutDisplayFunc(display);
   glutReshapeFunc(reshape);
   glutMainLoop();
   return 0;
}
```

При использовании плавной заливки соседние пиксели имеют немного различающиеся цвета. В **RGBA** режиме соседние пиксели с немного различающимся цветом выглядят одинаково, таким образом, цвет плавно меняется по всей плоскости полигона. В индексном режиме соседние пиксели могут ссылаться на различные элементы в цветовой таблице, в результате они могут иметь совсем разные цвета. Соседние вхождения в цветовую таблицу могут содержать сильно отличающиеся цвета, и плавно-закрашенный в индексном режиме примитив может выглядеть психоделически.

Во избежание этой проблемы следует создавать цветовую таблицу с плавно-меняющимися на протяжении всей области индексов (или ее частей) цветами. Помните, что загрузка цветов в цветовую таблицу осуществляется функциями вашей оконной системы, а не командами **OpenGL**. Если вы используете **GLUT**, вы можете применять функцию **glutSetColor()** для загрузки одного индекса в цветовой таблице указанными значениями красного, зеленого и синего цветовых компонентов. Первым аргументов **glutSetColor()** является индекс, а остальными – значения красного, зеленого и синего. Чтобы загрузить **32** индекса, следующих один за другим (с **16** по **47**) немного отличающимися оттенками желтого, можете использовать следующий фрагмент:

```
for(i=0;i<32;i++)
{
   glutSetColor(16+i,1.0*(i/32.0), 1.0*(i/32.0),0.0);
}
```

Теперь, если вы визуализируете плавно-залитые полигоны, которые используют только цвета по индексам от **16** до **47**, эти полигоны действительно будут залиты плавно-меняющимися оттенками желтого цвета.

При использовании плоской заливки цвет одной вершины определяет цвет всего примитива. Цветом для линии в таком случае является цвет, являвшийся текущим на момент указания второй (конечной) вершины. Для полигона цвет определяется текущим цветом на момент указания определенной вершины в соответствии с таблицей **4-2**. В таблице предполагается, что вершины и полигоны нумеруются с **1**. **OpenGL** строго следует этим правилам, однако лучший способ избежать неопределенности относительно того, как будет выглядеть примитива с плоской заливкой – это задавать один цвет для всего примитива (для всех его вершин).

Таблица **4-2**. Как **OpenGL** выбирает цвет для **i**-ого полигона с плоской заливкой

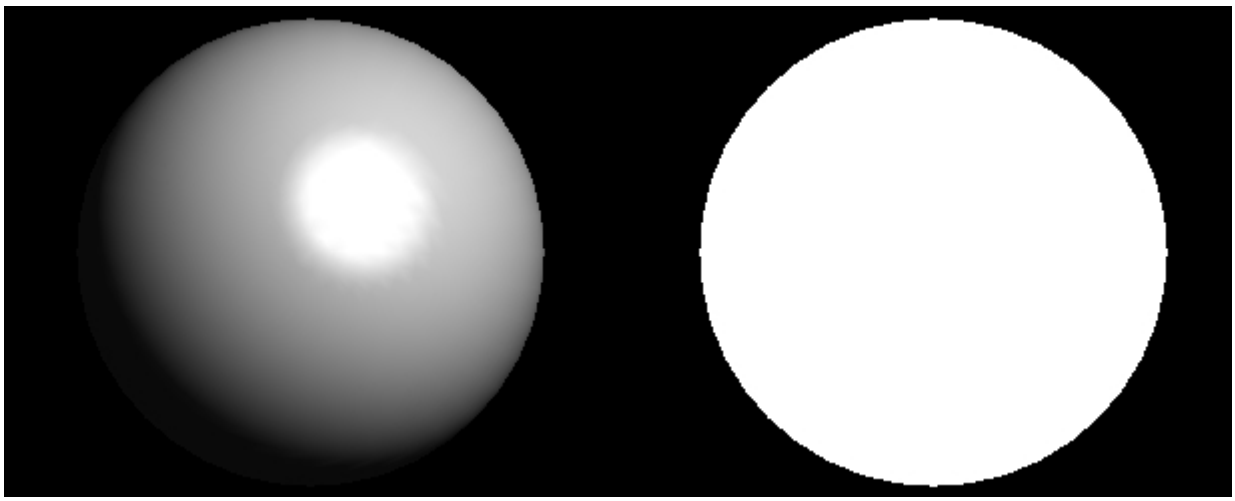
Тип полигона	**Вершина, используемая для выбора цвета *i*-го полигона**
единичный полигон	**1**

triangle strip	***i*+2**
triangle fan	***i*+2**
независимый треугольник	**3*i***
quad strip	**2*i*+2**
независимый прямоугольник	**4*i***

Глава 5. Освещение

Как вы могли видеть в предыдущей главе, **OpenGL** вычисляет цвет каждого пикселя в результирующей, отображаемой сцене, содержащейся в буфере кадра. Часть этого расчета зависит от того, какое освещение используется в сцене, и как объекты сцены отражают и поглощают свет. В качестве примера этому вспомните, что океан (или море, или река – вообще говоря, любой водоем) имеет различный цвет в солнечный или в облачный день. Присутствие света или облаков определяет, будет ли вода выглядеть ярко синей или грязно зеленой. По правде говоря, большинство объектов вообще не выглядят трехмерными, если они не освещены. На рисунке **5-1** изображено два варианта одной и той же сферы, однако слева она изображена с применением освещения, а справа – без.

Рисунок **5-1.** Освещенная и неосвещенная сферы



Как вы можете видеть, неосвещенная сфера ничем не отличается от двумерного круга. Это прекрасная демонстрация того, насколько важным является взаимодействие объектов и источников света при создании трехмерной сцены.

При использовании **OpenGL** вы можете манипулировать освещением и объектами сцены для создания различных эффектов.

5.1 Минимальные сведения об удалении невидимых граней

Предположим, что вы рисуете закрашенные трехмерные объекты. При использовании закрашенных полигонов задача рисования более близких к наблюдателю объектов и отбрасывания объектов, загороженных другими, становится очень важной.

Когда вы рисуете сцену, состоящую из трехмерных объектов, некоторые из них могут полностью или частично закрывать другие. Изменение точки наблюдения может изменить эти загораживающие связи между объектами. Например, если вы посмотрите на сцену с противоположного направления, все объекты, которые раньше были впереди других, окажутся позади них. Для создания реалистичных сцен следует

управлять подобными зависимостями. Предположим, что ваш код выглядит следующим образом:

```
while(1)
{
   получить_точку_наблюдения_через_позицию_мыши();
   glClear(GL_COLOR_BUFFER_BIT);
   нарисовать_трехмерный_объект_A();
   нарисовать_трехмерный_объект_B();
}
```

При определенных положениях мыши, объект **A** может закрывать объект **B**. При других положениях ситуация может быть обратной. Если не производить никаких специальных операций, предшествующий код всегда рисует объект **B**вторым (то есть поверх объекта **A**) вне зависимости от точки наблюдения. В худшем случае, если объекты **A**и **B** пересекаются друг с другом (то есть часть объекта **A** закрыта объектом **B**, и часть объекта **B**закрыта объектом **A**) изменение порядка рисования не решит проблему.

Уничтожение частей закрашенных объектов, загороженных другими, носит название *удаления невидимых поверхностей*. (Удаление невидимых линий, выполняющее ту же работу для объектов, представленных в виде каркаса, несколько сложнее и здесь не рассматривается.) Простейший способ добиться удаления невидимых поверхностей заключается в использовании буфера глубины (иногда называемого **z**-буфером).

Буфер глубины ассоциирует с каждым пикселем значение глубины (или дистанции) от плоскости наблюдения (обычно от ближней отсекающей плоскости). Изначально, командой **glClear()** с параметром **GL_DEPTH_BUFFER_BIT** глубина для всех пикселей устанавливается в значение наибольшей возможной дистанции (обычно дальней отсекающей плоскости). Затем объекты сцены могут рисоваться в любом порядке.

Графические вычисления, производимые аппаратурой или программно, конвертируют каждую рисуемую поверхность в набор тех пикселей окна, на которых поверхность должна появиться в случае, если она не загорожена чем-либо другим. Кроме того, вычисляется дистанция от плоскости наблюдения. При включенном механизме глубинной буферизации, до того как рисуется каждый пиксель, производится сравнение его глубины с той, которая уже хранится для данного пикселя в буфере глубины. Если новый пиксель находится ближе (то есть перед старым), его цвет и значение глубины заменяют имеющиеся. Если глубина нового пикселя больше, чем имеющаяся – он загорожен и его данные (цвет и глубина) отбрасываются.

Для того, чтобы использовать глубинную буферизацию, вы должны ее активизировать (включить). Это должно быть сделано только единожды. Каждый раз при отрисовке сцены, до начала фактического рисования, вы должны очистить буфер глубины, затем вы можете рисовать объекты в любом порядке.

Модификация предыдущего примера кода для использования в нем буфера глубины и выполнения удаления невидимых поверхностей должна выглядеть следующим образом:

```
glutInitDisplayMode(GLUT_DEPTH|...);
glEnable(GL_DEPTH_TEST);

...

while(1)
{
   glClear(GL_COLOR_BUFFER_BIT|GL_DEPTH_BUFFER_BIT);
   получить_точку_наблюдения_через_позицию_мыши();
   нарисовать_трехмерный_объект_A();
   нарисовать_трехмерный_объект_B();
}
```

Используемый для **glClear()** аргумент очищает и цветовой, и глубинный буферы.

Тест буфера глубины может повлиять на быстродействие вашей программы. Поскольку пиксельная информация часто отбрасывается, а не используется для рисования, удаление невидимых граней может несколько увеличить быстродействие. Однако наиболее большое значение для быстродействия имеет реализация буфера глубины. «Программный» буфер глубины (реализованный с использованием процессорной памяти) может быть намного медленнее, чем тот, который реализован с использованием специализированного аппаратного буфера.

5.2 Освещение в реальном мире и в OpenGL

Когда вы смотрите на физическую поверхность, ваше восприятие цвета зависит от распределения протонов, которые поступают в глаз и активизируют колбочки на сетчатке. Эти протоны исходят от источника света или комбинации источников. Часть этих протонов поглощается поверхностью, другая часть отражается ей. Вдобавок к этому разные поверхности могут иметь разные свойства – некоторые выглядят блестящими и отражают свет преимущественно в определенных направлениях, другие распределяют поступающий свет одинаково во всех направлениях. Большинство поверхностей являются чем-то средним между двумя описанными.

OpenGL рассчитывает свет и освещение так, как будто свет может быть разделен на красный, зеленый и синий компоненты. Таким образом, источник света характеризуется количеством красного, зеленого и синего света, которое он излучает, а материал поверхности характеризуется долями красного, зеленого и синего компонентов, которые он отражает в различных направлениях. Уравнения освещенности в **OpenGL** являются всего лишь аппроксимациями, но зато они работают достаточно хорошо и могут быть вычислены относительно быстро. Если вам требуется более точная (или просто другая) модель освещения, реализовывать ее самостоятельно. Такие расчеты могут быть весьма сложны – чтение книг по оптической физике в течение нескольких часов, возможно, убедит вас в этом.

В модели освещения **OpenGL** свет исходит от нескольких источников, которые могут включаться и выключаться индивидуально. Часть света обычно исходит из какого-либо определенного направления или позиции, часть распределена по всей сцене. Например, если вы включите лампочку в комнате, большая часть света будет исходить от нее, но часть света падает на поверхности предметов в комнате после того, как он отразился от одной, двух, трех или более стен. Считается, что этот многократно отраженный свет (называемый *фоновым* светом) распределен настолько сильно, что не существует никакого способа определить его исходное направление, однако он исчезает при выключении определенного источника света.

Наконец, в сцене может также присутствовать общий фоновый свет, у которого нет никакого конкретного источника, как будто он был отражен столько раз и распределен так сильно, что его оригинальный источник установить невозможно.

В модели **OpenGL** эффект от источника света присутствует только если есть поверхности поглощающие или отражающие свет. Считается, что каждая поверхность состоит из материала с несколькими свойствами. Материал может излучать свой собственный свет (например, фара автомобиля), он может распределять некоторое количество входящего света во всех направлениях, также он может отражать часть света в определенном направлении (например, зеркало или другая блестящая поверхность).

В модели освещения **OpenGL** предполагается, что освещение может быть разделено на **4** компонента: фоновое (***ambient***), диффузное (***diffuse***), зеркальное (***specular***) и исходящее (эмиссионное – ***emissive***). Все **4** компонента рассчитываются независимо и только затем суммируются.

5.2.1 Фоновый, диффузный, зеркальный и исходящий свет

Фоновоее излучение – это свет, который настолько распределен средой (предметами, стенами и так далее), что его направление определить невозможно – кажется, что он исходит отовсюду. Лампа дневного света имеет большой фоновый компонент, поскольку большая часть света, достигающего вашего глаза, сначала отражается от множества поверхностей. Уличный фонарь имеет маленький фоновый компонент: большая часть его света идет в одном направлении, кроме того, поскольку он находится на улице, очень небольшая часть света попадает вам в глаз после того, как отразится от других объектов. Когда фоновый свет падает на поверхность, он одинаково распределяется во всех направлениях.

Диффузный компонент – это свет, идущий из одного направления, таким образом, он выглядит ярче, если падает на поверхность под прямым углом, и выглядит тусклым, если касается ее всего лишь вскользь. Однако, когда он падает на поверхность, он распределяется одинаково во всех направлениях, то есть его яркость одинакова вне зависимости от того, с какой стороны вы смотрите на поверхность. Вероятно, любой свет, исходящий из определенного направления или позиции, имеет диффузный компонент.

Зеркальный свет исходит из определенного направления и отражается от поверхности в определенном направлении. При отражении хорошо сфокусированного лазерного луча от качественного зеркала происходит почти **100** процентное зеркальное отражение. Блестящий метал или пластик имеет высокий зеркальный компонент, а кусок ковра или плюшевая игрушка – нет. Вы можете думать о зеркальности как о том, насколько блестящим выглядит материал.

Помимо фонового, диффузного и зеркального цветов, материалы могут также иметь *исходящий* цвет, имитирующий свет, исходящий от самого объекта. В модели освещения **OpenGL**исходящий свет поверхности добавляет объекту интенсивности, но на него не влияют никакие источники света, и он не производит дополнительного света для сцены в целом.

Хотя источник света излучает единое распределение частот, фоновый, диффузный и зеркальный компоненты могут быть различны. Например, если в вашей комнате красные стены и белый свет, то этот свет, отражаясь от стен будет скорее красным, чем белым (несмотря на то, что падающий на стену свет -- белый). **OpenGL** позволяет устанавливать значения красного, зеленого и синего независимо для каждого компонента света.

5.2.2 Цвета материала

Модель освещения **OpenGL**делает допущение о том, что цвет материала зависит от долей падающего красного, зеленого и синего света, которые он отражает. Например, максимально красный шар отражает весь красный свет, который на него падает и поглощает весь зеленый и синий. Если вы посмотрите на такой мяч под белым светом (состоящим из одинакового количества красного, зеленого и синего), весь красный свет отразится, и вы увидите красный мяч. Если смотреть на мяч при красном свете, он также будет выглядеть красным. Если, однако, посмотреть на него под зеленым светом, он будет выглядеть черным (весь зеленый свет поглотится, а красного нет, то есть никакой свет отражен не будет).

Также как и свет, материалы имеют разные фоновый, диффузный и зеркальный цвета, которые задают реакцию материала на фоновый, диффузный и зеркальный компоненты света. Фоновый цвет материала комбинируется с фоновым компонентом всех источников света, диффузный цвет с диффузным компонентом, а зеркальный с зеркальным. Фоновый и диффузный цвета задают видимый цвет материала, они обычно близки, если не эквивалентны. Зеркальный цвет обычно белый или серый. Он задет

цвет блика на объекте (то есть он может совпадать с зеркальным компонентом источника света).

5.2.3 RGB– величины для света и материалов

Цветовые компоненты, задаваемые для источников света, означают совсем не то же самое, что для материалов. Для источника света число представляет собой процент от полной интенсивности каждого цвета. Если **R**, **G** и **B –** величины цвета источника света все равны **1.0**, свет будет максимально белым. Если величины будут равны **0.5**, свет все равно останется белым, но лишь с половиной интенсивности (он будет казаться серым). Если **R=G=1** и **B=0** (полный красный, полный зеленый, отсутствие синего), свет будет желтым.

Для материалов числа соответствуют отраженным пропорциям этих цветов. Так что, если для материала **R=1**, **G=0.5** и **B=0**, этот материал отражает весь красный свет, половину зеленого и совсем не отражает синего. Другими словами, если обозначить компоненты источника света как **(LR, LG, LB)**, а компоненты материала как **(MR, MG, MB)** и проигнорировать все остальные взаимодействия, то свет, который поступит в глаз можно определить как **(LR–MR, LG–MG, LB–MB)**.

Похожим образом, если два источника света с характеристиками **(R1, G1, B1)** и **(R2, G2, B2)** направлены в глаз, **OpenGL**сложит компоненты: **(R1+R2, G1+G2, B1+B2)**. Если какая-либо из сумм будет больше **1** (соответствуя цвету, который нельзя отобразить), компонент будет урезан до **1**.

5.3 Простой пример: отображение освещенной сферы

Чтобы добавить на вашу сцену освещение, требуется выполнить несколько шагов:

1. Определить вектор нормали для каждой вершины каждого объекта. Эти нормали задают ориентацию объекта по отношению к источникам света.
2. Создать, выбрать и позиционировать один или более источников света.
3. Создать и выбрать *модель освещения*, которая определяет уровень глобального фонового света и эффективное положение точки наблюдения (для вычислений, связанных с освещением).
4. Задать свойства материала для объектов сцены.

Эти шаги выполняются в примере **5-1**. Он отображает сферу, освещенную единственным источником света, отображенным ранее на рисунке **5-1**.

Пример **5-1**. Рисуем освещенную сферу: файл **light**.

```
#include <glut.h>

//Инициализация
void init(void)
{
   GLfloat mat_specular[]={1.0,1.0,1.0,1.0};
   GLfloat mat_shininess[]={50.0};
   GLfloat light_position[]={1.0,1.0,1.0,0.0};
   GLfloat white_light[]={1.0,1.0,1.0,1.0};

   glClearColor(0.0,0.0,0.0,0.0);
   glShadeModel(GL_SMOOTH);
   glMaterialfv(GL_FRONT,GL_SPECULAR,mat_specular);
   glMaterialfv(GL_FRONT,GL_SHININESS,mat_shininess);
   glLightfv(GL_LIGHT0,GL_POSITION,light_position);
   glLightfv(GL_LIGHT0,GL_DIFFUSE,white_light);
   glLightfv(GL_LIGHT0,GL_SPECULAR,white_light);

   glEnable(GL_LIGHTING);
```

```
   glEnable(GL_LIGHT0);
   glEnable(GL_DEPTH_TEST);
}

//Отображение
void display(void)
{
   glClear(GL_COLOR_BUFFER_BIT|GL_DEPTH_BUFFER_BIT);
   glutSolidSphere(1.0,40,16);
   glFlush();
}

//Изменение размеров окна
void reshape(int w, int h)
{
   glViewport(0,0,(GLsizei) w, (GLsizei) h);
   glMatrixMode(GL_PROJECTION);
   glLoadIdentity();
   if (w<=h)
      glOrtho(-1.5,1.5,-0.5*(GLfloat)h/(GLfloat)w,0.5*(GLfloat)h/(GLfloat)w,-
10.0,10.0);
   else
      glOrtho(-1.5*(GLfloat)w/(GLfloat)h,1.5*(GLfloat)w/(GLfloat)h,-1.5,1.5,-
10.0,10.0);
   glMatrixMode(GL_MODELVIEW);
   glLoadIdentity();
}

int main(int argc, char **argv)
{
   glutInit(&argc,argv);
   glutInitDisplayMode(GLUT_SINGLE|GLUT_RGB|GLUT_DEPTH);
   glutInitWindowSize(500,500);
   glutInitWindowPosition(100,100);
   glutCreateWindow("Rendering a Lit Sphere");
   init();
   glutDisplayFunc(display);
   glutReshapeFunc(reshape);
   glutMainLoop();
   return 0;
}
```

Вызовы команд, связанных с освещением помещены в функцию **init()**. Эти вызовы коротко обсуждают в следующих параграфах и более детально позже в этой главе. Один из моментов, связанных с примером **5-1**, который стоит отметить, заключается в том, что используется цветовой режим **RGBA**, а не индексный. **OpenGL** по-разному рассчитывает освещенность для этих двух режимах, и, вообще говоря, возможности освещения в индексном режиме весьма ограничены. Таким образом, если вам необходимо освещение, режим **RGBA**более предпочтителен, и во всех примерах этой главы используется именно он.

5.3.1 Определение вектора нормали для каждой вершины каждого объекта

Нормали объекта задают его ориентацию относительно источников света. **OpenGL**использует нормаль вершины для определения того, как много света эта вершина получает от каждого источника. В приведенном примере процесс определения нормалей происходит внутри функции **glutSolidSphere()**.

Для правильного освещения нормали поверхности должны иметь единичную длину. Вы должны быть осторожны и помнить, что матрица модельного преобразования не масштабирует вектора нормалей автоматически, и результирующие нормали могут уже не иметь единичную длину. Чтобы убедиться, что нормали имеют единичную длину, возможно, вам придется вызвать команду **glEnable()** с аргументами **GL_NORMALIZE** или **GL_RESCALE_NORMAL**.

GL_RESCALE_NORMALведет к тому, что каждый компонент вектора нормали к поверхности будет умножен на одно и то же число, извлеченное из матрицы модельных преобразований. Таким образом, операция работает корректно только в случае, если нормали масштабировались равномерно и изначально имели единичную длину.

GL_NORMALIZE– более сложная операция, чем **GL_RESCALE_NORMAL.** Когда активизирован механизм нормализации **(GL_NORMALIZE),** сначала вычисляется длина вектора нормали, а затем каждый из компонентов вектора делится на это число. Эта операция гарантирует, что результирующие нормали будут иметь единичную длину, но она может быть более затратной в смысле скорости, чем простое масштабирование нормалей.

Замечание: Некоторые реализации OpenGL могут выполнять операцию GL_RESCALE_NORMAL не за счет простого масштабирования, а путем все той же полной нормализации (GL_NORMALIZE). Однако не существует способа выяснить так ли это в вашей реализации, и, в любом случае, вы не должны полагаться на это.

5.3.2 Создание, позиционирование и включение одного или более источников света

В примере **5-1** используется всего один источник белого света. Его местоположение задается вызовом команды **glLightfv().** В этом примере для нулевого источника света **(GL_LIGHT0)** задается белый цвет для диффузного и зеркального отражения. Если вам требуется свет другого цвета, измените параметры **glLight*().**

Кроме того, вы можете добавить к вашей сцене как минимум **8** источников света различных цветов. **(**Конкретная реализация **OpenGL** может позволять и больше **8-**ми источников.**)** По умолчанию цвет всех источников света кроме **GL_LIGHT0 –** черный. Вы можете располагать источники света везде, где только захотите, например, близко к сцене **(**как настольную лампу**)** или бесконечно далеко за ней **(**симулирую солнечный свет**).** Кроме того, вы можете управлять тем, испускает ли источник сфокусированный, узкий луч или более широкий. Помните, что каждый источник света требует дополнительных **(**и немалых**)** расчетов для отображения сцены, так что быстродействие вашего приложение зависит от количества источников.

После настройки параметров источника света вы должны активизировать его командой **glEnable().** Кроме того, вы должны вызвать команду **glEnable()** с аргументом **GL_LIGHTING,** чтобы подготовить **OpenGL**к выполнению расчетов, связанных с освещением.

5.3.3 Выбор модели освещения

Как вы можете ожидать, параметры модели освещения описываются командой **glLightModel*().** В примере **5-1** единственным задаваемым параметром модели освещения является глобальное фоновое освещение. Модель освещения также позволяет указывать, где находится предполагаемое местоположение наблюдателя: бесконечно далеко или локально по отношению к сцене, и должны ли вычисления производиться по-разному для лицевых и обратных поверхностей объектов. В примере **5-1** используются значения по умолчанию для двух аспектов модели – наблюдатель находится бесконечно далеко **(**режим **«**бесконечно удаленного наблюдателя**»)** и одностороннее освещение. Использование режима **«**локального наблюдателя**»** заставляет производить намного больший объем сложных расчетов, так как **OpenGL** должна вычислить угол между точкой наблюдения и каждым объектом. В режиме **«**бесконечно удаленного наблюдателя**»,** однако, этот угол игнорируется, и результат может быть несколько менее реалистичным. Далее, поскольку в примере **5-1** обратная поверхность сферы никогда не видна **(**она находится внутри сферы**),** достаточно одностороннего освещения.

5.3.4 Определение свойств материала для объектов сцены

Свойства материала объектов определяют, как он отражает свет и, таким образом, из какого реального материала он сделан (в зрительном восприятии). Поскольку взаимодействие между поверхностью материала и входящим светом достаточно сложное, довольно трудно задать такие параметры материала, чтобы объект имел определенный, желаемый вид. Вы можете задавать фоновый, диффузный и зеркальный цвета материала и то, насколько блестящим он будет выглядеть. В приведенном примере с помощью команды **glMaterialfv()** были явно заданы только два свойства материала: зеркальный цвет материала и исходящий цвет.

5.3.5 Несколько важных замечаний

Когда вы пишете свою собственную программу с освещением, всегда помните, что для некоторых его параметров вы можете использовать значения по умолчанию, тогда как другие должны быть заданы явно. Кроме того, не забудьте включить все описанные вами источники света, а также сам механизм расчетов, связанных с освещением. Наконец, помните, что вы можете использовать дисплейные списки (списки отображения), чтобы максимизировать эффективность при изменении условий освещения.

5.4 Создание источников света

Источники света имеют несколько параметров, таких как цвет, позиция и направление. Следующие разделы объясняют, как контролировать эти свойства, на что будет похож результирующий источник света. Команда, используемая для указания всех параметров света – это **glLight*()**. Она принимает три аргумента: идентификатор источника света, имя свойства и желаемое для него значение.

```
void glLight{if} (GLenum light, GLenum pname, TYPE param);
void glLight{if}v (GLenum light, GLenum pname, TYPE *param);
```

Создает источник, задаваемый параметром ***light*** (который может принимать значения **GL_LIGHT0, GL_LIGHT1, ..., GL_LIGHT7**). Задаваемая характеристика света определяется аргументом ***pname*** в виде константы (таблица **5-1**). В параметре ***param*** задается значение или значения, в которые следует установить характеристику ***pname***. Если используется векторная версия команды, ***param*** представляет собой вектор величин, а если невекторная, то ***param*** – одно единственное значение. Невекторная версия команды может использоваться только для указания параметров, чье значение выражается одним числом.

Таблица **5-1.** Значения по умолчанию для свойств источника света

Имена параметров	Значения по умолчанию	Смысл
GL_AMBIENT	**(0.0,0.0,0.0,1.0)**	Интенсивность фонового света
GL_DIFFUSE	**(1.0,1.0,1.0,1.0)** или **(0.0,0.0,0.0,1.0)**	Интенсивность диффузного света (значение по умолчанию для **0**-го источника - белый свет, для остальных - черный)
GL_SPECULAR	**(1.0,1.0,1.0,1.0)** или **(0.0,0.0,0.0,1.0)**	Интенсивность зеркального света (значение по умолчанию для **0**-го источника - белый свет, для остальных - черный)
GL_POSITION	**(0.0,0.0,1.0,0.0)**	Положение источника света (***x***,***y***,***z***,***w***)
GL_SPOT_DIRECTION	**(0.0,0.0,-1.0)**	Направление света прожектора (***x***,***y***,***z***)
GL_SPOT_EXPONENT	**0.0**	Концентрация светового луча
GL_SPOT_CUTOFF	**180.0**	Угловая ширина светового луча
GL_CONSTANT_ATTENUATION	**1.0**	Постоянный фактор ослабления
GL_LINEAR_ATTENUATION	**0.0**	Линейный фактор ослабления

GL_QUADRATIC_ATTENUATION 0.0	Квадратичный фактор ослабления

Замечание: Значения по умолчанию для GL_DIFFUSEи GL_SPECULAR в таблице 5-1 различаются для GL_LIGHT0 и других источников света (GL_LIGHT1, GL_LIGHT2, ...). Для параметров GL_DIFFUSEи GL_SPECULAR источника света GL_LIGHT0 значение по умолчанию – (1.0, 1.0, 1.0, 1.0). Для других источников света значение тех же параметров по умолчанию – (0.0, 0.0, 0.0, 1.0).

В примере **5-2** показано, как использовать **glLight*()**:

Пример **5-2.** Указание цветов и позиции для источников света

```
GLfloat light_ambient[]={0.0,0.0,0.0,1.0};
GLfloat light_diffuse[]={1.0,1.0,1.0,1.0};
GLfloat light_specular[]={1.0,1.0,1.0,1.0};
GLfloat light_position[]={1.0,1.0,1.0,0.0};

glLightfv(GL_LIGHT0,GL_AMBIENT, light_ambient);
glLightfv(GL_LIGHT0,GL_DIFFUSE, light_diffuse);
glLightfv(GL_LIGHT0,GL_SPECULAR, light_specular);
glLightfv(GL_LIGHT0,GL_POSITION, light_position);
```

Как вы можете видеть, для величин параметров определены массивы, и для установки этих параметров несколько раз вызывается команда **glLightfv()**. В этом примере первые три вызова **glLightfv()** являются избыточными, поскольку они используются для указания таких же значений параметров **GL_AMBIENT**, **GL_DIFFUSE** и **GL_SPECULAR**, какие установлены по умолчанию.

Замечание: Помните, что каждый источник света нужно включить командой **glEnable()**.

Все параметры команды **glLight*()** и их возможные значения описываются в следующих разделах. Эти параметры взаимодействуют с параметрами модели освещения конкретной сцены и параметрами материала объектов.

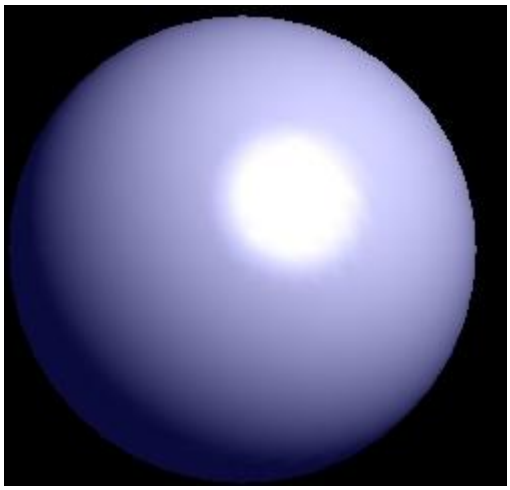
5.4.1 Цвет

OpenGL позволяет вам ассоциировать с каждым источником света три различных параметра, связанных с цветом: **GL_AMBIENT**, **GL_DIFFUSE**и **GL_SPECULAR**. Параметр **GL_AMBIENT** задает **RGBA** интенсивность фонового света, который каждый отдельный источник света добавляет к сцене. Как вы можете видеть в таблице **5-1**, по умолчанию источник сета не добавляет к сцене фонового света, так как значение по умолчанию для **GL_AMBIENT** равно **(0.0, 0.0, 0.0, 1.0)**. Именно эта величина была использована в примере **5-1.** Если бы для того же источника света был задан синий фоновый свет,

```
GLfloat light_ambient[]={0.0,0.0,1.0,1.0};
glLightfv(GL_LIGHT0,GL_AMBIENT,light_ambient);
```

то результат был бы таким, какой показан на рисунке **5-2.**

Рисунок **5-2.** Сфера, освещенная одним источником, добавляющим синий фоновый свет



Параметр **GL_DIFFUSE** наверное наиболее точно совпадает с тем, что вы привыкли называть «цветом света». Он определяет **RGBA** цвет диффузного света, который отдельный источник света добавляет к сцене. По умолчанию для **GL_LIGHT0** параметр **GL_DIFFUSE** равен **(1.0, 1.0, 1.0, 1.0)**, что соответствует яркому белому свету. Значение по умолчанию для всех остальных источников света **(GL_LIGHT1, GL_LIGHT2, ..., GL_LIGHT7)** равно **(0.0, 0.0, 0.0, 0.0)**.

Параметр **GL_AMBIENT** влияет на цвет зеркального блика на объекте. В реальном мире на объектах вроде стеклянной бутылки имеется зеркальный блик соответствующего освещению цвета (часто белого). Для создания эффекта реалистичности установите **GL_SPECULAR** в то же значение, что и **GL_DIFFUSE**. По умолчанию **GL_SPECULAR** равно **(1.0, 1.0, 1.0, 1.0)** для **GL_LIGHT0** и **(0.0, 0.0, 0.0, 0.0)** для остальных источников.

```
 Замечание: Альфа компонент всех этих цветов используется, только если
включено цветовое наложение.
```

5.4.2 Позиция и ослабление

Как было отмечено ранее, вы можете выбрать, следует ли считать источник света расположенным бесконечно далеко от сцены или близко к ней. На источники света первого типа ссылаются как на *направленные* **(directional)**: эффект от бесконечно далекого расположения источника заключается в том, что все лучи его света могут считаться параллельными к моменту достижения объекта. Примером реального направленного источника света может служить солнце. Источники второго типа называются *позиционными* **(positional)**, поскольку их точное положение на сцене определяет их эффект и, в частности, направление из которого идут лучи. Примером позиционного источника света является настольная лампа. Свет, используемый в примере **5-1**, является направленным:

```
GLfloat light_position[]={1.0,1.0,1.0,0.0};
glLightfv(GL_LIGHT0, GL_POSITION, light_position);
```

Как видно, для параметра **GL_POSITION** вы задаете вектор из четырех величин **(*x*, *y*, *z*, *w*)**. Если последняя величина ***w*** равна **0**, соответствующий источник света считается направленным, и величины **(*x*, *y*, *z*)** определяют его направление. Это направление преобразуется видовой матрицей. По умолчанию параметру **GL_POSITION** соответствуют значения **(0, 0, 1, 0)**, что задает свет, направленный вдоль отрицательного направления оси ***z***. (Заметьте, что никто не запрещает вам создать свет, направленный в **(0, 0, 0)**, однако такой свет не даст должного эффекта).

Если значение ***w*** не равно **0**, свет является позиционным, и величины **(*x*, *y*, *z*)** задают его местоположение источника света в однородных объектных координатах. Это

положение преобразуется видовой матрицей и сохраняется в видовых координатах. Кроме того, по умолчанию позиционный свет излучается во всех направлениях, но вы можете ограничить распространение света, создав конус излучения, определяющий прожектор.

Замечание: Помните, что цвета по всей плавно залитой грани полигона определяются цветами, вычисляемыми для его вершин. Возможно, из-за этого вам следует избегать применения больших полигонов вместе с локальным источником света. Если вы разместите источник света близко к центру полигона, вершины могут быть слишком далеки от него и не получат достаточно света. В результате весь полигон будет выглядеть темнее, чем вы ожидали. Во избежание подобной ситуации разбивайте крупные полигоны на более мелкие составляющие.

В реальном мире интенсивность света уменьшается с увеличением дистанции от его источника. Поскольку источник направленного света считается бесконечно удаленным, нет смысла ослаблять его интенсивность с увеличением дистанции, так что ослабление для направленных источников света не рассчитывается. Однако вам может понадобиться задать ослабление света от позиционного источника. **OpenGL** выполняет ослабление источника света, умножая его интенсивность на фактор ослабления:

$$F_{att} = \frac{1}{k_c + k_l d + k_q d^2},$$

где ***d*** **-** расстояние между позицией источника света и вершиной,

k_c **- GL_CONSTANT_ATTENUATION (**постоянный фактор ослабления**),**

k_l **- GL_LINEAR_ATTENUATION (**линейный фактор ослабления**),**

k_q **- GL_QUADRATIC_ATTENUATION (**квадратичный фактор ослабления**).**

По умолчанию k_c равно **1.0**, а k_l и k_q **- 0.0**, но вы можете задавать для этих коэффициентов другие значения:

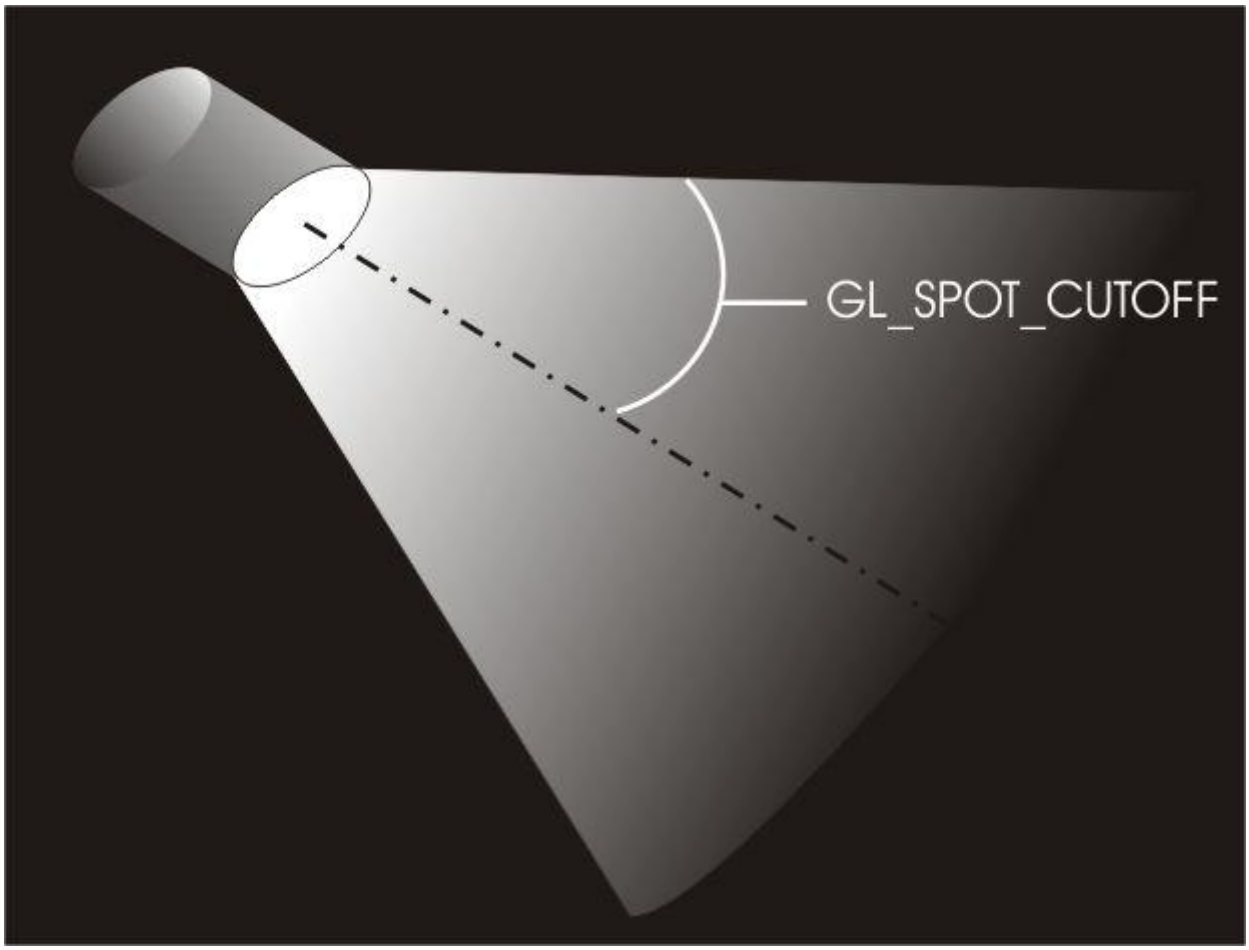
```
glLightf(GL_LIGHT0, GL_CONSTANT_ATTENUATION, 2.0);
glLightf(GL_LIGHT0, GL_LINEAR_ATTENUATION, 1.0);
glLightf(GL_LIGHT0, GL_QUADRATIC_ATTENUATION, 0.5);
```

Заметьте, что ослабляются и фоновая, и диффузная, и зеркальная интенсивности. Только исходящие от объектов интенсивности и глобальная фоновая интенсивность не подлежат ослаблению. Также отметьте, что поскольку расчет ослабления требует дополнительной операции деления **(**и возможно других операций**)** для каждого вычисляемого цвета, использование ослабляемых источников света может замедлить работу приложения.

5.4.3 Прожектора

Как было отмечено ранее, вы можете заставить позиционный источник света действовать в качестве прожектора, ограничив распространение света конусом излучения. Чтобы создать прожектор, вам требуется определить желаемую ширину светового конуса. **(**Помните, что поскольку прожектор является позиционным источником света, вам необходимо расположить его в желаемом месте. Также заметьте, что никто не запрещает вам создавать прожекторы из направленных источников света, но в этом случае вы наверняка не получите желаемый результат.**)** Чтобы задать угол между осью конуса и лучом, идущим вдоль, ребра конуса, используйте параметр **GL_SPOT_CUTOFF.** Тогда угол между ребрами конуса, образованными вершиной и концами диаметра основания, будет равен заданному вами углу, умноженному на **2.**

Рисунок **5-3.** Параметр **GL_SPOT_CUTOFF**



Заметьте, что свет не излучается за ребрами конуса. По умолчанию, работа с прожекторами заблокирована, так как параметр **GL_SPOT_CUTOFF** равен **180.0.** Эта величина означает, что свет излучается во всех направлениях **(**угол при вершине равен **360** градусам, что вообще не определяет конус**).** Значения для **GL_SPOT_CUTOFF** должны попадать в диапазон **[0.0, 90.0] (**значение также может быть равно специальной величине **180.0).** Следующая строка устанавливает параметр **GL_SPOT_CUTOFF** равным **45** градусам**:**

```
glLightf(GL_LIGHT0, GL_SPOT_CUTOFF, 45.0);
```

Вам также требуется задать направление прожектора, которое определяет центральную ось светового конуса**:**

```
GLfloat spot_direction[] = {-1.0, -1.0, 0.0};
glLightfv(GL_LIGHT0, GL_SPOT_DIRECTION, spot_direction);
```

Направление указывается в объектных координатах. По умолчанию направление **– (0.0, 0.0, -1.0),** так что если вы явно не изменяете параметр **GL_SPOT_DIRECTION,** источник света направлен вдоль отрицательного направления оси ***z*.** Также имейте в виду, что направление прожектора преобразуется видовой матрицей также, как если бы это был вектор нормали, и результат сохраняется в видовых координатах.

В дополнение к углу светового конуса и направлению прожектора, существует два способа управления распределением интенсивности света внутри конуса. Во-первых, вы можете задать фактор ослабления света, который перемножается с интенсивностью. Вы также можете установить параметр **GL_SPOT_EXPONENT (**который по умолчанию

равен **0)**, чтобы управлять концентрацией света. Свет имеет самую высокую интенсивность в центре конуса. При движении от центра к ребрам он ослабляется с коэффициентом равным косинусу угла между направлением света и направлением от источника света к освещаемой вершине, возведенному в степень **GL_SPOT_EXPONENT**. Таким образом, больший экспоненциальный коэффициент разброса света **(GL_SPOT_EXPONENT)** ведет к более фокусированному свету.

5.4.4 Несколько источников света

Как было отмечено, у вас на сцене может быть как минимум восемь источников света **(**возможно больше, но это зависит от реализации **OpenGL).** Поскольку **OpenGL** проводит вычисления для того, чтобы определить, сколько света получает каждая вершина от каждого источника, увеличение источников слета весомо влияет на быстродействие. Константы, используемые для ссылки на **8** источников света **–** это **GL_LIGHT0**, **GL_LIGHT1**, **GL_LIGHT2**, **GL_LIGHT3** и так далее. В предыдущих обсуждениях параметры устанавливались для источника **GL_LIGHT0**. Если вам требуются дополнительные источники света, вы должны задать и их параметры. Также помните, что значения параметров по умолчанию не совпадают для **GL_LIGHT0** и других источников. В примере **5-3** определяется белый ослабевающий прожектор:

Пример **5-3.** Второй источник света

```
GLfloatlight_ambient[]={0.2,0.2,0.2,1.0};
GLfloat light_diffuse[]={1.0,1.0,1.0,1.0};
GLfloat light_specular[]={1.0,1.0,1.0,1.0};
GLfloat light_position[]={-2.0,2.0,1.0,1.0};
GLfloat spot_direction[]={-1.0,-1.0,0.0};

glLightfv(GL_LIGHT1,GL_AMBIENT, light_ambient);
glLightfv(GL_LIGHT1,GL_DIFFUSE, light_diffuse);
glLightfv(GL_LIGHT1,GL_SPECULAR, light_specular);
glLightfv(GL_LIGHT1,GL_POSITION, light_position);
glLightf(GL_LIGHT1,GL_CONSTANT_ATTENUATION, 1.5);
glLightf(GL_LIGHT1,GL_LINEAR_ATTENUATION, 0.5);
glLightf(GL_LIGHT1,GL_QUADRATIC_ATTENUATION, 0.2);
glLightf(GL_LIGHT1,GL_SPOT_CUTOFF, 45.0);
glLightfv(GL_LIGHT1,GL_SPOT_DIRECTION, spot_direction);
glLightf(GL_LIGHT1,GL_SPOT_EXPONENT, 2.0);

glEnable(GL_LIGHT1);
```

Если эти строки добавить в пример **5-1**, сфера будет освещена двумя источниками: одним направленным и одним прожектором.

5.4.5 Управление позицией и направлением источников света

OpenGL обращается с позицией и направлением источника света так же, как с позицией геометрического примитива. Другими словами, источник света является субъектом тех же матричных преобразований, что и примитив. Говоря более определенно, когда команда **glLight*()** вызывается для указания позиции или направления источника света, эта позиция или направление преобразуются текущей видовой матрицей и сохраняется в видовых координатах. Это означает, что вы можете манипулировать позицией и направлением источников света, изменяя содержимое видовой матрицы. **(**Проекционная матрица не оказывает воздействия на позицию или направление источника света.**)** В этом разделе объясняется, как добиться трех перечисленных далее эффектов, изменяя место в программе **(**относительно видовых и модельных преобразований**)**, где задаются источники света:

- Позиция источника света остается фиксированной
- Источник света движется вокруг неподвижного объекта
- Источник света движется вместе с точкой наблюдения

5.4.5.1 Стационарный источник света

В простейшем случае, как в примере **5-1**, положение источника света не изменяется. Чтобы добиться этого эффекта, вам следует устанавливать позицию источника света после всех используемых видовых и/или модельных преобразований. Пример **5-4** демонстрирует, как может выглядеть соответствующий объединенный код из **init()** и **reshape()**.

Пример **5-4.** Стационарный источник света

```
glViewport(0,0,(GLsizei) w, (GLsizei) h);
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
if (w<=h)
   glOrtho(-1.5,1.5,-0.5*(GLfloat)h/(GLfloat)w,0.5*(GLfloat)h/(GLfloat)w,-
10.0,10.0);
else
   glOrtho(-1.5*(GLfloat)w/(GLfloat)h,1.5*(GLfloat)w/(GLfloat)h,-1.5,1.5,-
10.0,10.0);
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();

//далее функции init()
GLfloat light_position[]={1.0,1.0,1.0,0.0};
glLightfv(GL_LIGHT0,GL_POSITION,light_position);
```

Как вы видите, сначала устанавливается порт просмотра и проекционная матрица. Затем в видовую матрицу загружается единичная, и только после этого задается положение источника света. Поскольку используется единичная матрица, явно установленное положение источника света **– (1.0, 1.0, 1.0)** не изменяется умножением на него видовой матрицы. После этого, поскольку на тот момент ни позиция источника, ни видовая матрица не изменены, направление света остается прежним **– (1.0, 1.0, 1.0)**.

5.4.5.2 Независимо движущийся источник света

Теперь предположим, что вам нужно вращать источник света вокруг стационарного объекта или перемещать источник света вокруг него. Один из способов сделать это заключается в определении источника света после специфического модельного преобразования, которое изменяет позицию источника. Вы можете начать с вызовов тех же команд в **init()** как и в предыдущем разделе. Затем вам следует выполнить требуемое модельное преобразование (на стеке видовых матриц) и сбросить позицию источника (вероятно, в функции **display()**). Пример **5-5** демонстрирует возможный код функции **display()**.

Пример **5-5.** Независимое движение источника света

```
GLdouble spin;
void display()
{
   GLfloat light_position[]={0.0,0.0,1.5,1.0};
   glClear(GL_COLOR_BUFFER_BIT|GL_DEPTH_BUFFER_BIT);
   glPushMatrix();
      gluLookAt(0.0,0.0,5.0,0.0,0.0,0.0,0.0,1.0,0.0);
      glPushMatrix();
         glRotated(spin,1.0,0.0,0.0);
         glLightfv(GL_LIGHT0,GL_POSITION,light_position);
      glPopMatrix();
      glutSolidTorus(0.275,0.85,8,15);
   glPopMatrix();
glFlush();
}
```

spin **–** это глобальная переменная**,** значение которой**,** вероятнее всего**,** определяется устройством ввода**.** Функция **display()** вызывает перерисовку сцены с источником света обращенным вокруг неподвижного торуса на ***spin***градусов**.** Заметьте**,** что две пары команд **glPushMatrix()** и **glPopMatrix()** изолируют видовые и модельные преобразования**.** Поскольку в примере **5-5** точка наблюдения остается неизменной**,** текущая матрица проталкивается в стек и далее требуемое видовое преобразование задается командой **gluLookAt()**. Далее получившаяся матрица снова проталкивается в стек перед указанием преобразования поворота командой **glRotate()**. Затем задается положение источника света в новой повернутой системе координат**,** таким образом**,** источник света оказывается повернутым относительно своего первоначального положения**. (**Помните**,** что позиция источника света сохраняется в видовых координатах**,** получающихся после преобразования видовой матрицей**.)** Торус рисуется после того**,** как матрица выталкивается из стека**.** Пример **5-6** представляет собой листинг программы**,** поворачивающей источник света вокруг неподвижного объекта**.** При нажатии левой кнопки мыши угол поворота источника света увеличивается на **30** градусов**.** Позиция источника света представлена маленьким неосвещенным проволочным кубом**.**

Пример **5-6.** Перемещение источника света с помощью модельных преобразований**:** файл **movelight.cpp**

```
#include <glut.h>
int spin=0;

//Инициализация
void init(void)
{
   glClearColor(0.0,0.0,0.0,0.0);
   glShadeModel(GL_SMOOTH);
   glEnable(GL_LIGHTING);
   glEnable(GL_LIGHT0);
   glEnable(GL_DEPTH_TEST);
}

void display(void)
{
   GLfloat position[]={0.0,0.0,1.5,1.0};

   glClear(GL_COLOR_BUFFER_BIT|GL_DEPTH_BUFFER_BIT);
   glPushMatrix();
      glTranslatef(0.0,0.0,-5.0);
      glPushMatrix();
         glRotated((GLdouble)spin,1.0,0.0,0.0);
         glLightfv(GL_LIGHT0,GL_POSITION,position);
         glTranslated(0.0,0.0,1.5);
         glDisable(GL_LIGHTING);
         glColor3f(0.0,1.0,1.0);
         glutWireCube(0.1);
         glEnable(GL_LIGHTING);
      glPopMatrix();
      glutSolidTorus(0.275,0.85,40,40);
   glPopMatrix();
   glutSwapBuffers();
}

//Изменение размеров окна
void reshape(int w, int h)
{
   glViewport(0,0,(GLsizei) w, (GLsizei) h);
   glMatrixMode(GL_PROJECTION);
   glLoadIdentity();
   gluPerspective(40.0,(GLfloat)w/(GLfloat)h,1.0,20.0);
   glMatrixMode(GL_MODELVIEW);
   glLoadIdentity();
}

void light_moving(void)
{
```

```
   spin=(spin+1)%360;
   glutPostRedisplay();
}

void mouse(int button,int state,int x, int y)
{
   switch(button)
   {
      case GLUT_LEFT_BUTTON:
         if(state==GLUT_DOWN)
            glutIdleFunc(light_moving);
         else
            glutIdleFunc(NULL);
         break;
   }
}

int main(int argc, char **argv)
{
   glutInit(&argc,argv);
   glutInitDisplayMode(GLUT_DOUBLE|GLUT_RGB|GLUT_DEPTH);
   glutInitWindowSize(500,500);
   glutInitWindowPosition(100,100);
   glutCreateWindow("Moving a Light with Modeling Transformations");
   init();
   glutDisplayFunc(display);
   glutReshapeFunc(reshape);
   glutMouseFunc(mouse);
   glutMainLoop();
   return 0;
}
```

5.4.5.3 Перемещение источника света вместе с точкой наблюдения

Чтобы создать источник света, который передвигается вместе с точкой наблюдения, вам следует задать его позицию до видового преобразования. Тогда видовое преобразование будет одинаковым образом воздействовать и на источник света и на точку наблюдения. Помните, что положение источника света сохраняется в видовых координатах – это один из небольшого числа случаев, в которых видовые координаты весьма важны. В примере **5-7** позиция источника света задается в функции **init()** и сохраняется в видовых координатах **(*0*, *0*, *0*)**. Другими словами, свет излучается из линзы камеры.

Пример **5-7.** Источник света, перемещающийся вместе с точкой наблюдения

```
GLfloat light_position[]={0.0, 0.0, 0.0, 1.0};

glViewport(0,0,(GLsizei) w, (GLsizei) h);
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
gluPerspective(40.0, (GLfloat)w/(GLfloat)h,1.0,100.0);
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
glLightfv(GL_LIGHT0,GL_POSITION,light_position);
```

Теперь, если точка наблюдения переместится, источник света переместится вместе с ней, сохраняя дистанцию **(*0*, *0*, *0*)** относительно глаза. В продолжении примера **5-7**, следующим далее, глобальные переменные **(*ex*, *ey*, *ez*)** управляют положением точки наблюдения, а **(*upx*, *upy*, *upz*)** определяют вектор верхнего направления. Функция **display()**, вызываемая из цикла обработки сообщений для перерисовки сцены, может выглядеть следующим образом:

```
GLdlouble ex, ey, ez, upx, upy, upz;
```

```
void display()
{
   glClear(GL_COLOR_BUFFER_BIT|GL_DEPTH_BUFFER_BIT);
   glPushMatrix();
      gluLookAt(ex, ey, ez, 0.0, 0.0, 0.0, upx, upy, upz);
      glutSolidTorus(0.275, 0.85, 8, 15);
   glPopMatrix();
   glFlush();
}
```

Когда перерисовывается освещенный торус, и источник света, и точка наблюдения перемещаются в одно и то же место. С изменением величин, передаваемых в **gluLookAt() (**и перемещением наблюдателя**),** объект никогда не будет темным, поскольку он всегда освещается с позиции наблюдателя. Даже если вы не измените координаты позиции источника света явно, он все равно будет перемещаться, поскольку изменяется видовая координатная система.

Этот метод может быть весьма полезен при эмуляции фонаря на каске шахтера. Другим примером может быть свеча или лампа, которую несут в руке. Позиция источника света, задаваемая вызовом **glLightfv(GL_LIGHTi, GL_POSITION, *position*),** по смыслу будет соответствовать указанию расстояний источника от наблюдателя по ***x***, ***y*** и ***z***. Затем при изменении положения наблюдателя, источник света будет оставаться на том же относительном расстоянии.

5.5 Выбор модели освещения

В **OpenGL** понятие модели освещения разделяется на **4** компонента:

- Интенсивность глобального фонового света.
- Считается ли положение точки наблюдения локальным к сцене или бесконечно удаленным.
- Должен ли расчет освещенности производиться по-разному для лицевых и обратных граней объектов.
- Должен ли зеркальный цвет отделяться от фонового и диффузного и накладываться на объект после операций текстурирования.

Этот раздел объясняет, как задавать модель освещения. Здесь также обсуждается, как включать освещение – то есть, как сказать **OpenGL,** что она должна производить расчет освещенности.

glLightModel*() – это команда, используемая для задания всех параметров модели освещения. **glLightModel*()** принимает два аргумента: имя параметра модели освещения в виде константы и значение для этого параметра.

```
void glLightModel{if} (GLenum pname, TYPE param);
void glLightModel{if}v (GLenum pname, TYPE *param);
```

Устанавливает свойства модели освещения. Устанавливаемая характеристика модели освещения определяется аргументом ***pname*** (таблица **5-2**). ***param*** задает величину, в которую устанавливается ***pname***; если используется векторная версия команды, то это указатель на группу величин, если применяется невекторная версия – в ***param*** содержится сама величина. Невекторная версия команды может использоваться только для установки параметров, определяемых одной величиной (и не может применяться для **GL_LIGHT_MODEL_AMBIENT**).

Таблица **5-2.** Значения по умолчанию для параметра ***pname*** модели освещения

Имена параметров	Значения по умолчанию	Смысл

GL_LIGHT_MODEL_AMBIENT	**(0.2,0.2,0.2,1.0)**	**RGBA** интенсивность всей сцены
GL_LIGHT_MODEL_LOCAL_VIEWER	**0.0** или **GL_FALSE**	способ вычисления углов зеркального отражения
GL_LIGHT_MODEL_TWO_SIDE	**0.0** или **GL_FALSE**	выбор между односторонним и двухсторонним освещением
GL_LIGHT_MODEL_COLOR_CONTROL	**GL_SINGLE_COLOR**	вычисляется ли зеркальный цвет отдельно от фонового и диффузного

5.5.1 Глобальный фоновый свет

Как обсуждалось ранее, каждый источник света может добавлять к сцене фоновый свет. Кроме того, может присутствовать другой фоновый свет, не принадлежащий никакому конкретному источнику. Чтобы задать **RGBA** интенсивность такого глобального фонового света, используйте параметр **GL_LIGHT_MODEL_AMBIENT** следующим образом:

```
GLfloat lmodel_ambient[]={0.2,0.2,0.2,1.0};
glLightModelfv(GL_LIGHT_MODEL_AMBIENT, lmodel_ambient);
```

В этом примере значения для ***lmodel_ambient*** совпадают со значениями по умолчанию для **GL_LIGHT_MODEL_AMBIENT.** Поскольку эти числа задают небольшое количество белого фонового света, вы можете видеть объекты сцены даже в том случае, если отсутствуют какие-либо дополнительные источники света. На рисунке **5-4** изображены три чайника, нарисованные на сцене с разным количеством глобального фонового света **(**его интенсивность увеличивается слева направо**).**

Рисунок **5-4.** Чайники, освещенные разным количеством глобального фонового света

5.5.2 Локальная или бесконечно удаленная точка наблюдения

Положение точки наблюдения влияет на расчет блика, создаваемого зеркальным отражением. Более определенно, интенсивность блика в конкретной вершине зависит от вектора нормали в этой вершине, направления из вершины к источнику света и направления из вершины к точке наблюдения. Имейте в виду, что на самом деле точка наблюдения не перемещается никакими командами, связанными с освещением, однако расчеты могут быть различными на основании предположения о ее мнимом местонахождении.

Если предполагается, что точка наблюдения бесконечно удалена, то направления из нее ко всем вершинам считаются одинаковыми. Локальная точка наблюдения позволяет получать более реалистичный результат, но, поскольку должны быть рассчитаны направления между ней и каждой вершиной, среднее быстродействие приложения может значительно снизиться. По умолчанию используется бесконечно удаленная точка наблюдения. Вы можете сменить ее на локальную следующим вызовом:

```
glLightModeli(GL_LIGHT_MODEL_LOCAL_VIEWER, GL_TRUE);
```

Этот вызов помещает точку наблюдения в **(0, 0, 0)** в видовых координатах. Для переключения обратно к бесконечно удаленной точке наблюдения в предыдущей строке кода замените **GL_TRUE**на **GL_FALSE.**

5.5.3 Двухстороннее освещение

Расчет освещенности производится для всех полигонов, являются ли они лицевыми или обратными. Поскольку вы обычно настраиваете источники света, размышляя о лицевых полигонах, обратные могут быть неверно освещены. В примере **5-1**, где в качестве объекта используется сфера, всегда видны только лицевые грани, поскольку именно они находятся снаружи сферы. Таким образом, в данном случае неважно, как выглядят обратные. Однако, если часть сферы будет отсечена и ее внутренняя поверхность станет видимой, вам возможно захочется, чтобы эта поверхность была освещена в соответствии с заданными вами условиями освещения; возможно вам также потребуется задать для обратной поверхности иные свойства материала, чем для лицевой. Когда вы включаете двухстороннее освещение вызовом:

```
glLightModeli(GL_LIGHT_MODEL_TWO_SIDE, GL_TRUE);
```

OpenGL считает нормалями к обратным поверхностям нормали к лицевым, направленные в противоположном направлении. Обычно это означает, что вектора нормалей для видимых обратных полигонов **(**как и для лицевых**)** направлены на, а не от наблюдателя. В результате все полигоны освещаются правильно. Однако дополнительные операции по обеспечению двухстороннего освещения делают его более медленным, чем принятое по умолчанию одностороннее освещение.

Чтобы выключить двухстороннее освещение, измените **GL_TRUE** в предыдущей строке кода на **GL_FALSE.** Вы также можете управлять тем, какие грани **OpenGL** считает лицевыми с помощью команды **glFrontFace().**

5.5.4 Отделение зеркального цвета

При обычном расчете освещенности, фоновый, диффузный, зеркальный вклады источника света в общую освещенность и интенсивность исходящего света вычисляются и просто складываются вместе. По умолчанию наложение текстуры производится после расчета освещенности, в результате зеркальный блик может выглядеть мутным, а текстура может быть искажена. Если же вы произведете следующий вызов:

```
glLightModeli(GL_LIGHT_MODEL_COLOR_CONTROL, GL_SEPARATE_SPECULAR_COLOR);
```

OpenGL отделит расчет зеркального освещения от его применения. После этого вызова при расчете освещенности производится два цвета на каждую вершину: первичный, состоящий из суммы всех незеркальных вкладов всех источников света, и вторичный, состоящий из суммы всех зеркальных вкладов всех источников света. Во время текстурирования с цветами текстуры комбинируется только первичный цвет. Вторичный свет добавляется к комбинации первичного и текстурных цветов после операции текстурирования. Объекты, освещенные и текстурированные с применением отдельного зеркального цвета, как правило, имеют более четкий выделяющийся зеркальный блик.

Чтобы восстановить метод расчета освещенности по умолчанию, вызовите:

```
glLightModeli(GL_LIGHT_MODEL_COLOR_CONTROL, GL_SINGLE_COLOR);
```

После этого вызова первичный цвет будет состоять из всех вкладов источника света в освещенность вершины: фонового, диффузного и зеркального, а также из исходящего цвета материала самого объекта. Никакие дополнительные вклады в освещенность не добавляются после текстурирования.

Если вы не производите наложение текстуры на объект, отделение зеркального цвета от остальных не имеет никакого смысла.

5.5.5 Включение расчета освещенности

При использовании **OpenGL** вам необходимо включать (или выключать) механизм расчета освещенности. Если данный механизм не включен, текущий цвет просто ассоциируется с текущей вершиной, и не производится никаких вычислений, касающихся нормалей, источников света, модели освещения и свойств материала. Расчет освещенности включается командой:

```
glEnable(GL_LIGHTING);
```

и выключается командой:

```
glDisable(GL_LIGHTING);
```

Кроме того, вам требуется включать каждый источник света, после того, как вы определили его параметры. В примере **5-1** используется только один источник света – **GL_LIGHT0**:

```
glEnable(GL_LIGHT0);
```

5.6 Указание свойств материала

Вы видели, как создать источник света с определенными характеристиками и как задать нужную модель освещения. В этом разделе описано, как задать свойства материала для объектов на сцене: фоновый, диффузный и зеркальный цвета, степень сияния (насколько блестящим выглядит объект) и цвет исходящего от объекта света. Большинство свойств материала концептуально похожи на те, которые использовались при создании источников света. Механизм из указания также аналогичен установке параметров источника света за исключением того, что здесь используется команда **glMaterial*()**.

```
void glMaterial{if}(GLenum face, GLenum pname, TYPE param);
void glMaterial{if}v(GLenum face, GLenum pname, TYPE *param);
```

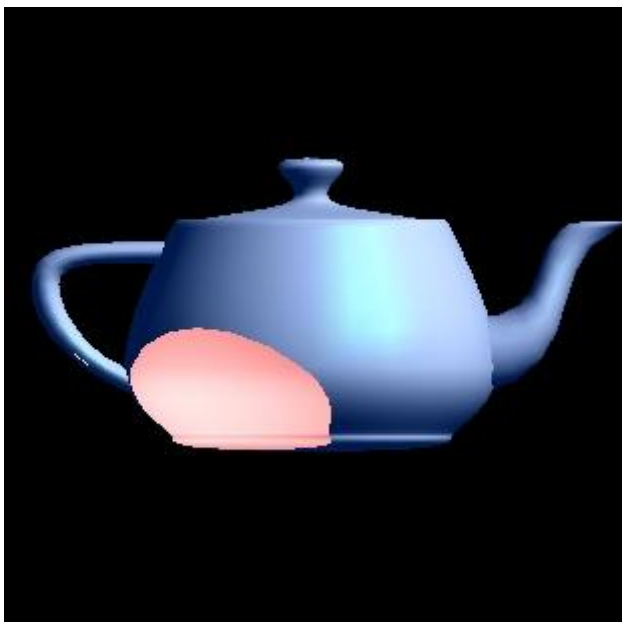
Задает свойство материала для использования при расчете освещенности. Аргумент ***face*** может принимать значения **GL_FRONT**, **GL_BACK**или **GL_FRONT_AND_BACK**, указывая для каких граней объекта задается свойство материала. Устанавливаемое свойство материала определяется значением аргумента ***pname***, а его значение содержится в ***param*** (в виде указателя на вектор величин в случае векторной версии команды или в виде самой величины при использовании невекторного варианта). Невекторная версия команды работает только для параметра **GL_SHININESS**. Возможные значения для аргумента ***pname*** перечислены в таблице **5-3**. Заметьте, что константа **GL_AMBIENT_AND_DIFFUSE** позволяет вам одновременно установить фоновый и диффузный цвета материала в одно и то же **RGBA**значение.

Таблица **5-3**. Значения по умолчанию для параметра материала ***pname***

Имена параметров	Значения по умолчанию	Смысл
GL_AMBIENT	**(0.2,0.2,0.2,1.0)**	фоновый цвет материала
GL_DIFFUSE	**(0.8,0.8,0.8,1.0)**	диффузный цвет материала
GL_AMBIENT_AND_DIFFUSE		фоновый и диффузный цвет материала
GL_SPECULAR	**(0.0,0.0,0.0,1.0)**	зеркальный цвет материала
GL_SHININESS	**0.0**	показатель зеркального отражения
GL_EMISSION	**(0.0,0.0,0.0,1.0)**	исходящий цвет материала
GL_COLOR_INDEXES	**(0,1,1)**	индексы фонового, диффузного и зеркального цветов

Как обсуждалось ранее, вы можете задать расчет освещенности для лицевых и обратных полигонов объекта. Если в вашем приложении обратные грани могут быть видимыми, вы можете по-разному задать параметры материала для лицевых и обратных граней объекта, используя аргумент ***face*** команды **glMaterial*()**. На рисунке **5-5** изображен объект, нарисованный с применением разных свойств материала для его лицевых и обратных граней.

Рисунок **5-5.** Разный материал для лицевых и обратных граней



В примере **5-8** приведен код, используемый для рисования чайника на рисунке **5-5.**

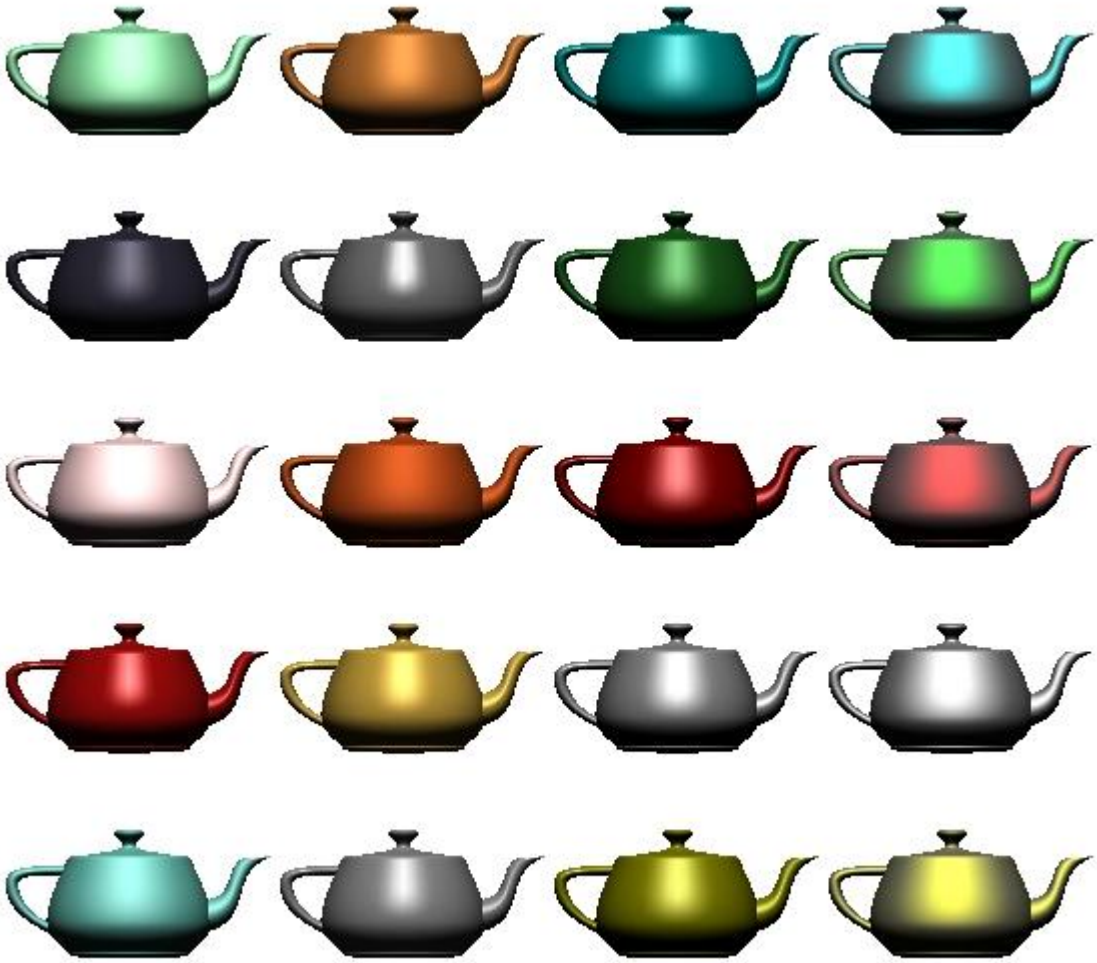
Пример **5-8.** Отсеченный чайник с разными материалами для лицевых и обратных граней: файл **two_side_lighting.cpp**

```
#include <glut.h>

//Инициализация
void init(void)
{
   glClearColor(0.0,0.0,0.0,0.0);
   glShadeModel(GL_SMOOTH);

   //Настраиваем источник света
   GLfloat light_position[]={1.0,1.0,1.0,0.0};
   GLfloat white_light[]={1.0,1.0,1.0,1.0};
   glLightfv(GL_LIGHT0,GL_POSITION,light_position);
   glLightfv(GL_LIGHT0,GL_DIFFUSE,white_light);
   glLightfv(GL_LIGHT0,GL_SPECULAR,white_light);

   //...и модель двухстороннего освещения
```

```
   GLfloat lmodel_ambient[]={1.0,1.0,1.0,1.0};
   glLightModelfv(GL_LIGHT_MODEL_AMBIENT,lmodel_ambient);
   glLightModeli(GL_LIGHT_MODEL_TWO_SIDE, GL_TRUE);

   //Включаем нужные механизмы
   glEnable(GL_LIGHTING);
   glEnable(GL_LIGHT0);
   glEnable(GL_DEPTH_TEST);

   //Задаем параметры материала для лицевых граней
   GLfloat mat_specular_front[]={0.0,0.3,0.9,1.0};
   GLfloat mat_ambient_front[]={0.0,0.1,0.3,1.0};
   glMaterialfv(GL_FRONT,GL_SPECULAR,mat_specular_front);
   glMaterialfv(GL_FRONT,GL_AMBIENT,mat_ambient_front);

   //... и для обратных
   GLfloat mat_specular_back[]={1.0,0.0,0.0,1.0};
   GLfloat mat_ambient_back[]={0.5,0.1,0.1,1.0};
   glMaterialfv(GL_BACK,GL_SPECULAR,mat_specular_back);
   glMaterialfv(GL_BACK,GL_AMBIENT,mat_ambient_back);

   //GL_SHININESS в данном случае одинаково для всех граней
   GLfloat mat_shininess[]={50.0};
   glMaterialfv(GL_FRONT_AND_BACK,GL_SHININESS,mat_shininess);

   //Задаем дополнительную плоскость отсечения
   GLdouble equation[]={0.4,0.5,-0.5,0.6};
   glClipPlane(GL_CLIP_PLANE1, equation);
   glEnable(GL_CLIP_PLANE1);

   //Задаем какие грани считать лицевыми
   glFrontFace(GL_CW);
}

//Отображение
void display(void)
{
   glClear(GL_COLOR_BUFFER_BIT|GL_DEPTH_BUFFER_BIT);
   glutSolidTeapot(0.9);
   glFlush();
}

//Изменение размеров окна
void reshape(int w, int h)
{
   glViewport(0,0,(GLsizei) w, (GLsizei) h);
   glMatrixMode(GL_PROJECTION);
   glLoadIdentity();
   if (w<=h)
      glOrtho(-1.5,1.5,-1.5*(GLfloat)h/(GLfloat)w, 1.5*(GLfloat)h/(GLfloat)w,-
10.0,10.0);
   else
      glOrtho(-1.5*(GLfloat)w/(GLfloat)h, 1.5*(GLfloat)w/(GLfloat)h,-1.5,1.5,-
10.0,10.0);
   glMatrixMode(GL_MODELVIEW);
   glLoadIdentity();
}

//Точка входа
int main(int argc, char **argv)
{
   glutInit(&argc,argv);
   glutInitDisplayMode(GLUT_SINGLE|GLUT_RGB|GLUT_DEPTH);
   glutInitWindowSize(310,310);
   glutInitWindowPosition(100,100);
   glutCreateWindow("Two-sided lighting");
   init();
   glutDisplayFunc(display);
   glutReshapeFunc(reshape);
   glutMainLoop();
   return 0;
}
```

Чтобы получить представление о том, каких эффектов можно достигнуть с помощью различных свойств материала, взгляните на рисунок **5-6**. На этом рисунке изображено несколько экземпляров одного и того же объекта, нарисованных с применением разных свойств материала. Каждый раз был использован один и тот же источник света и одинаковая модель освещения. В разделах следующих далее, обсуждаются отдельные свойства материалов.

Рисунок **5-6.** Чайники из разных материалов



Заметьте, что большинство параметров материала, устанавливаемых с помощью команды **glMaterial*()** являются **RGBA** величинами. Независимо от величины альфа, задаваемой в других параметрах, величиной альфа для конкретной вершины принимается та, которая была задана для диффузного цвета, то есть для параметра **GL_DIFFUSE**. Также заметьте, что в индексном цветовом режиме **RGBA** значения для свойств материала не используются.

5.6.1 Диффузное и фоновое отражение

Параметры **GL_DIFFUSE**и **GL_AMBIENT**, устанавливаемые командой **glMaterial*()** влияют на цвета диффузного и фонового света, отражаемого объектом. Диффузное отражение играет наиболее важную роль в определении того, что вы воспринимаете как цвет объекта. На ваше восприятие оказывает влияние цвет падающего диффузного света и угол между этим светом и вектором нормали к поверхности. (Диффузное отражение наиболее заметно, если диффузный свет падает перпендикулярно к

поверхности.) Положение точки наблюдения вообще не влияет на диффузное отражение.

Фоновое отражение одинаково влияет на все цвета объекта. Поскольку диффузное отражение наиболее заметно на непосредственно освещаемых частях объекта, фоновое отражение наиболее заметно на частях объекта, которые непосредственно не освещаются. Суммарное фоновое отражение объекта складывается из глобального фонового освещения и фонового света, излучаемого индивидуальными источниками. Также как и диффузное, фоновое отражение не зависит от точки наблюдения.

В реальном мире, диффузное и фоновое отражение от одного и того же объекта, как правило, имеет одинаковый цвет. По этой причине **OpenGL** предоставляет упрощенный путь для задания обоих параметров материала в команде **glMaterial*()**:

```
GLfloat mat_amb_diff[]={0.1,0.5,0.8,1.0};
glMaterialfv(GL_FRONT_AND_BACK,GL_AMBIENT_AND_DIFFUSE);
```

В этом примере **RGBA** цвет **(0.1, 0.5, 0.8, 1.0) –** глубокий синий цвет **–** задается в качестве текущих цветов фонового и диффузного отражения для лицевых и обратных граней полигонов.

5.6.2 Зеркальное отражение

Зеркальное отражение от объекта создает блик. В отличие от фонового и диффузного отражения, количество зеркального отражения, видимого наблюдателем, зависит от положения точки наблюдения. Чтобы это понять, представьте, что вы смотрите на металлический шарик под солнечным светом. При движении вашей головы, блик, создаваемый солнечным светом на шарике до некоторой степени будет передвигаться вместе с вами. Однако, если вы слишком далеко переместите голову, вы вообще перестанете видеть блик.

OpenGL позволяет вам задавать эффект, производимый материалом на отраженный свет **(**с помощью параметра **GL_SPECULAR)**, и контролировать размер и яркость блика **(**с помощью параметра **GL_SHININESS)**. **GL_SHININESS** может принимать значения в диапазоне **[0.0, 128.0]:** чем больше значение, тем меньше и ярче зеркальный блик **(**то есть при увеличении значения **GL_SHININESS** блик будет все более и более сфокусирован**).**

5.6.3 Исходящий (эмиссионный) цвет

Задавая **RGBA** цвет для параметра **GL_EMISSION**, вы можете изобразить объект так, как будто он излучает свет этого цвета. Поскольку большинство реальных объектов **(**помимо источников света**)** сами не излучают свет, вы, вероятно, будете использовать этот эффект для симуляции ламп и других источников света.

Имейте в виду, что хотя объекты с высокой интенсивностью исходящего света выглядят светящимися, на самом деле они не работают в качестве источников света, то есть не оказывают никакого воздействия на другие объекты сцены. Для того, чтобы имитировать лампу или фонарь, вам нужно создать дополнительный источник света и поместить его в ту же позицию, что и светящийся объект.

5.6.4 Изменение свойств материала

В примере **5-1** один и тот же материал использовался для всех вершин единственного объекта сцены **(**сферы**).** В иных ситуациях вам может потребоваться ассоциировать различные свойства материала с различными вершинами одного и того же объекта.

Более вероятно, что на сцене будет больше одного объекта. Например, посмотрите на выдержку кода в примере **5-9.**

Пример **5-9.** Разные свойства материала

```
GLfloat no_mat[]={0.0,0.0,0.0,1.0};
GLfloat mat_ambient[]={0.7,0.7,0.7,1.0};
GLfloat mat_ambient_color[]={0.8,0.8,0.2,1.0};
GLfloat mat_diffuse[]={0.1,0.5,0.8,1.0};
GLfloat mat_specular[]={1.0,1.0,1.0,1.0};
GLfloat no_shininess[]={0.0};
GLfloat low_shininess[]={5.0};
GLfloat high_shininess[]={100.0};
GLfloat mat_emission[]={0.3,0.2,0.2,0.0};
glClear(GL_COLOR_BUFFER_BIT|GL_DEPTH_BUFFER_BIT);

//Нарисовать первую сферу слева
glPushMatrix();
glTranslate(-3.75,0.0,0.0);
glMaterialfv(GL_FRONT, GL_AMBIENT, no_mat);
glMaterialfv(GL_FRONT, GL_DIFFUSE, mat_diffuse);
glMaterialfv(GL_FRONT, GL_SPECULAR, mat_specular);
glMaterialfv(GL_FRONT, GL_SHININESS, no_shininess);
glMaterialfv(GL_FRONT, GL_EMISSION, no_mat);
glutSolidSphere(1.0,16,16);
glPopMatrix();

//Нарисовать вторую сферу правее первой
glPushMatrix();
glTranslate(-1.25,0.0,0.0);
glMaterialfv(GL_FRONT, GL_AMBIENT, no_mat);
glMaterialfv(GL_FRONT, GL_DIFFUSE, mat_diffuse);
glMaterialfv(GL_FRONT, GL_SPECULAR, mat_specular);
glMaterialfv(GL_FRONT, GL_SHININESS, low_shininess);
glMaterialfv(GL_FRONT, GL_EMISSION, no_mat);
glutSolidSphere(1.0,16,16);
glPopMatrix();

//Нарисовать третью сферу правее первых двух
glPushMatrix();
glTranslate(1.25,0.0,0.0);
glMaterialfv(GL_FRONT, GL_AMBIENT, no_mat);
glMaterialfv(GL_FRONT, GL_DIFFUSE, mat_diffuse);
glMaterialfv(GL_FRONT, GL_SPECULAR, mat_specular);
glMaterialfv(GL_FRONT, GL_SHININESS, high_shininess);
glMaterialfv(GL_FRONT, GL_EMISSION, no_mat);
glutSolidSphere(1.0,16,16);
glPopMatrix();

//Нарисовать последнюю сферу справа
glPushMatrix();
glTranslate(3.75,0.0,0.0);
glMaterialfv(GL_FRONT, GL_AMBIENT, no_mat);
glMaterialfv(GL_FRONT, GL_DIFFUSE, mat_diffuse);
glMaterialfv(GL_FRONT, GL_SPECULAR, no_mat);
glMaterialfv(GL_FRONT, GL_SHININESS, no_shininess);
glMaterialfv(GL_FRONT, GL_EMISSION, no_mat);
glutSolidSphere(1.0,16,16);
glPopMatrix();
```

Как вы видите, **glMaterial*()** вызывается несколько раз для установки требуемого свойства материала для каждой сферы. Заметьте, что эту команду необходимо вызывать только для тех параметров, которые действительно изменяются. Вторая, третья и четвертая сферы используют одинаковые фоновое и диффузное свойство материала, так что на самом деле нет смысла задавать их каждый раз. Поскольку существуют некоторые временные затраты на выполнение **glMaterial*()**, пример **5-9** может быть переписан более эффективным образом (на самом деле из него нужно просто удалить некоторые строки).

5.6.5 Режим цвета материала

Другая техника снижения затрат производительности заключается в использовании команды **glColorMaterial().**

```
void glColorMaterial (GLenum face, GLenum mode);
```

Для граней, заданных аргументом ***face***, заставляет свойство (или свойства) материала, заданное аргументом **mode**, все время принимать значение текущего цвета. При изменении текущего цвета (командой **glColor*()**) указанные свойства материала также незамедлительно меняются. Аргумент ***face*** может принимать значения **GL_FRONT**, **GL_BACK**или **GL_FRONT_AND_BACK** (такое значение аргумент имеет по умолчанию). Аргумент ***mode*** может принимать значения **GL_AMBIENT**, **GL_DIFFUSE**, **GL_SPECULAR**, **GL_AMBIENT_AND_DIFFUSE** (значение по умолчанию) или **GL_EMISSION**. В каждый момент времени активным является только один из режимов. **glColorMaterial()** не работает при использовании освещения в индексном режиме.

Отметьте, что **glColorMaterial()** задает две независимые величины: первая определяет, свойства материала какого типа или типов граней изменяются с изменением текущего цвета, а вторая – какие именно свойства материала изменяются. Однако **OpenGL** не следит за разными значениями аргумента ***mode*** для разных типов граней!

Помимо вызова **glColorMaterial()** вам требуется вызвать **glEnable()** с аргументом **GL_COLOR_MATERIAL.** После этого во время рисования вы можете изменять текущий цвет командой **glColor*() (**а также по необходимости другие свойства материала командой **glMaterial*()).**

```
glEnable(GL_COLOR_MATERIAL);
glColorMaterial(GL_FRONT, GL_DIFFUSE);

//Теперь glColor*() будет изменять диффузное отражение
glColor3f(0.2,0.5,0.8);

//Здесь мы рисуем несколько объектов
glColorMaterial(GL_FRONT, GL_SPECULAR);

//Теперь glColor*() изменяет не диффузное,
//а зеркальное отражение
glColor3f(0.9,0.0,0.2);

//Рисуем еще несколько объектов
glDisable(GL_COLOR_MATERIAL);
```

Используйте **glColorMaterial()** в тех случаях, когда для множества вершин на сцене вам требуется изменять значение единственного параметра материала. Если вам нужно постоянно изменять более одного свойства – используйте **glMaterial*().** Когда функциональность команды **glColorMaterial()** вам не требуется, обязательно убедитесь, что вы выключили данный механизм командой **glDisable().** Это позволит вам избежать свойств материала, изменяющихся «по неизвестным причинам», а также снизит затраты производительности. Быстродействие приложения при работе с механизмом **glColorMaterial()** зависит от реализации **OpenGL.** Некоторые реализации могут оптимизировать обработку вершин для ускоренного изменения свойств материала на базе текущего цвета.

Пример **5-10** представляет собой интерактивную программу, использующую команду **glColorMaterial()** для изменения свойств материала. Нажатие на любую из трех кнопок мыши изменяет цвет диффузного отражения.

Пример **5-10.** Использование **glColorMaterial():** файл **colormat.cpp**

```
#include <glut.h>
GLfloat diffuseMaterial[4]={0.0,0.0,0.0,1.0};

//Инициализация
void init(void)
{
   GLfloat mat_specular[4]={1.0,1.0,1.0,1.0};
   GLfloat light_position[4]={1.0,1.0,1.0,0.0};

   glClearColor(0.0,0.0,0.0,0.0);
   glShadeModel(GL_SMOOTH);
   glEnable(GL_DEPTH_TEST);
   glMaterialfv(GL_FRONT,GL_DIFFUSE,diffuseMaterial);
   glMaterialfv(GL_FRONT,GL_SPECULAR,mat_specular);
   glMaterialf(GL_FRONT,GL_SHININESS,25.0);
   glLightfv(GL_LIGHT0,GL_POSITION,light_position);
   glEnable(GL_LIGHTING);
   glEnable(GL_LIGHT0);
   glColorMaterial(GL_FRONT,GL_DIFFUSE);
   glEnable(GL_COLOR_MATERIAL);
}

//Отображение
void display(void)
{
   glClear(GL_COLOR_BUFFER_BIT|GL_DEPTH_BUFFER_BIT);
   glutSolidSphere(1.0,20,16);
   glutWireCube(1.0);
   glFlush();
}

//Изменение размеров окна
void reshape(int w, int h)
{
   glViewport(0,0,(GLsizei) w, (GLsizei) h);
   glMatrixMode(GL_PROJECTION);
   glLoadIdentity();
   if (w<=h)
      glOrtho(-1.5,1.5,-1.5*(GLfloat)h/(GLfloat)w,1.5*(GLfloat)h/(GLfloat)w,-
10.0,10.0);
   else
      glOrtho(-1.5*(GLfloat)w/(GLfloat)h,1.5*(GLfloat)w/(GLfloat)h,-1.5,1.5,-
10.0,10.0);
   glMatrixMode(GL_MODELVIEW);
   glLoadIdentity();
}

//Реакция на мышь
void mouse(int button,int state, int x, int y)
{
   switch(button)
   {
      case GLUT_LEFT_BUTTON:
         if (state==GLUT_DOWN) // Изменяем красный
         {
            diffuseMaterial[0]+=0.1;
            if (diffuseMaterial[0] > 1.0)
               diffuseMaterial[0]=0.0;
            glColor4fv(diffuseMaterial);
            glutPostRedisplay();
         }
         break;
      case GLUT_MIDDLE_BUTTON:
         if (state==GLUT_DOWN) // Изменяем зеленый
         {
            diffuseMaterial[1]+=0.1;
            if (diffuseMaterial[1] > 1.0)
               diffuseMaterial[1]=0.0;
            glColor4fv(diffuseMaterial);
            glutPostRedisplay();
         }
         break;
      case GLUT_RIGHT_BUTTON:
         if (state==GLUT_DOWN) // Изменяем синий
```

```
            {
               diffuseMaterial[2]+=0.1;
               if (diffuseMaterial[2] > 1.0)
                  diffuseMaterial[2]=0.0;
               glColor4fv(diffuseMaterial);
               glutPostRedisplay();
            }
            break;
         default:
            break;
      }
   }

   int main(int argc, char **argv)
   {
      glutInit(&argc,argv);
      glutInitDisplayMode(GLUT_SINGLE|GLUT_RGB|GLUT_DEPTH);
      glutInitWindowSize(500,500);
      glutInitWindowPosition(100,100);
      glutCreateWindow("Using glColorMaterial()");
      init();
      glutDisplayFunc(display);
      glutReshapeFunc(reshape);
      glutMouseFunc(mouse);
      glutMainLoop();
      return 0;
   }
```

5.7 Математика освещения

Дополнительно: В этом разделе представлены уравнения, используемые **OpenGL** в расчетах освещенности, для определения цветов в **RGBA** режиме. Вам не требуется читать этот раздел, если вы хотите добиться определенного освещения путем экспериментов. Даже после чтения этого раздела, вам наверняка придется экспериментировать, но вы будете иметь лучшее представление о том, как значения определенных параметров влияют на цвет вершины. Помните, что если освещение выключено, цвет вершины просто принимает значение текущего цвета, если же освещение включено, расчеты, связанные с ним, производятся в видовых координатах.

В следующих уравнениях математические расчеты производятся отдельно для **R**, **G** и **B** компонент цвета. Таким образом, если в уравнении складываются три величины, следует понимать, что **R** величины, **G** величины и **B** величины складываются отдельно, формируя результирующий **RGB** цвет $(R_1 + R_2 + R_3, G_1 + G_2 + G_3, B_1 + B_2 + B_3)$. Когда же три величины перемножаются – результатом будет $(R_1 R_2 R_3, G_1 G_2 G_3, B_1 B_2 B_3)$. (Помните, что значение компонента **A** или альфа компонента цвета вершины равно значению диффузного альфа компонента материала этой вершины.)

Цвет освещенной вершины вычисляется следующим образом:

цвет вершины **=** исходящий цвет материала вершины **+**
глобальный фоновый цвет, умноженный на значение фонового свойства материала вершины **+**
фоновые, диффузные и зеркальные вклады всех источников света, ослабленные должным образом.

После расчета освещенности цветовые величины масштабируются в диапазон **[0, 1]** (в режиме **RGBA**).

Имейте в виду, что при расчете освещенности **OpenGL** не принимает в расчет возможность того, что один объект может загораживать свет для другого, как следствие

автоматически не отбрасываются тени. Также имейте в виду, что объекты, обладающие исходящим излучением, не добавляют света другим объектам.

5.7.1 Исходящий свет материала

Исходящий свет является простейшим для расчета. Его значение равно параметру **GL_EMISSION** материала вершины.

5.7.2 Масштабированный глобальный фоновый свет

Числовая величина этого света вычисляется путем перемножения глобального фонового света **(**определенного параметром **GL_LIGHT_MODEL_AMBIENT)** на значение фонового свойства материала **GL_AMBIENT**, определенного командой **glMaterial*()**:

$$ambient_{light\_model} * ambient_{material}$$

Каждая пара **R**, **G** и **B** величин для этих двух параметров перемножается независимо, формируя результирующий **RGB** цвет (R_1R_2, G_1G_2, B_1B_2).

5.7.3 Вклады источников света

Каждый источник света может вносить свой вклад в освещенность вершины, и все эти вклады от всех источников суммируются. Далее приведено уравнение, используемое для вычисления вклада каждого источника света:

вклад **=**фактор ослабления ***** прожекторный эффект *** (**фоновый вклад **+** диффузный вклад **+** зеркальный вклад**)**

5.7.3.1 Фактор ослабления

Фактор ослабления был описан ранее в этой главе.

$$F_{att} = \frac{1}{k_c + k_l d + k_q d^2},$$

где ***d*** **-** расстояние между позицией источника света и вершиной,

k_c **- GL_CONSTANT_ATTENUATION (**постоянный фактор ослабления**),**

k_l **- GL_LINEAR_ATTENUATION (**линейный фактор ослабления**),**

k_q **- GL_QUADRATIC_ATTENUATION (**квадратичный фактор ослабления**).**

Если источник света является направленным, фактор ослабления равен **1.**

5.7.3.2 Прожекторный эффект

Прожекторный эффект может вычисляться тремя разными способами в зависимости от того, является ли в действительности источник света прожектором и от того, находится ли вершина внутри или снаружи конуса излучения прожектора:

- **1**, если источник не является прожектором **(**параметр **GL_SPOT_CUTOFF**равен **180.0)**;
- **0**, если источник света является прожектором, но вершина лежит вне его конуса излучения;

- $(\max\{v \bullet d, 0\})^{GL\_SPOT\_EXPONENT}$, где $v = (v_x, v_y, v_z)$ -- единичный вектор из прожектора в вершину; $d = (d_x, d_y, d_z)$ -- направление прожектора **(GL_SPOT_DIRECTION)**, предполагается, что источник света является прожектором и вершина лежит внутри его конуса излучения. Скалярное произведение векторов ***v*** и ***d*** меняется в соответствии с косинусом угла между ними; таким образом, объекты на линии освещения получают максимум освещенности, объекты вне этой линии получают меньше света в соответствии с косинусом угла.

Для определения того, лежит ли вершина внутри конуса излучения прожектора **OpenGL** вычисляет величину $(\max\{v \bullet d, 0\})$. Если эта величина меньше косинуса угла при вершине конуса излучения прожектора **(GLL_SPOT_CUTOFF) –** вершина находится вне конуса излучения, иначе вершина находится внутри конуса.

5.7.3.3 Фоновый вклад

Фоновый вклад представляет собой фоновый цвет источника, умноженный на фоновое свойство материала вершины:

$ambient_{light} * ambient_{material}$

5.7.3.4 Диффузный вклад

Необходимо принимать в расчет диффузный вклад источника света независимо от того, падает ли свет непосредственно на вершину или нет. Он вычисляется как:

$(\max\{L \bullet n, 0\}) * diffuse_{light} * diffuse_{material}$, где

$L = (L_x, L_y, L_z)$ -- единичные вектор, указывающий из вершины в направлении положения источника света **(GL_POSITION)**.

$n = (n_x, n_y, n_z)$ -- вектор нормали единичной длины в вершине.

5.7.3.5 Зеркальный вклад

Зеркальный вклад зависит от того, падает ли свет непосредственно на вершину. Если $L \bullet n$ меньше или равно **0**, то в вершине не присутствует зеркального компонента. **(**Если эта величина меньше **0**, источник света находится с другой стороны поверхности.**)** Если же зеркальный вклад присутствует, он зависит от следующего:

- Вершинная нормаль единичной длины (n_x, n_y, n_z).
- Сумма двух векторов единичной длины: вектора между вершиной и положением источника света **(**или направлением источника света**)** и вектора между вершиной и точкой наблюдения **(**в случае если выбран режим **GL_LIGHT_MODEL_LOCAL_VIEWER**, если это не так, то в качестве второго вектора суммы берется **(0, 0, 1))**. Эта векторная сумма нормализуется **(**путем деления каждого компонента на размер получившегося вектора**)**. В результате получаем $s = (s_x, s_y, s_z)$.
- Зеркальный показатель **(GL_SHININESS)**.
- Зеркальный цвет источника света **(**$GL\_SPECULAR_{light}$**)**.

- Зеркальное свойство материала (${GL\_SPECULAR}_{material}$).

OpenGL использует следующее уравнение для расчета зеркального вклада источника света (с учетом приведенного выше описания):

$$(\max\{s \cdot n, 0\})^{shininess} * specular_{light} * specular_{material}.$$

Однако, если $L \bullet n = 0$, зеркальный вклад равен **0**.

5.7.4 Суммирование всех элементов

Ссылаясь на определения, данные в предыдущих разделах, мы можем записать полное уравнение освещенности, используемое **OpenGL** в следующем виде:

$$\text{цвет вершины} = emission_{material} + ambient_{light\_model} * ambient_{material} +$$
$$\sum_{i=0}^{n-1} \frac{1}{k_c + k_l d + k_q d^2} * spotlight\_effect_i *$$
$$[ambient_{light} * ambient_{material} +$$
$$(\max\{L \cdot n, 0\}) * diffuse_{light} * diffuse_{material} +$$
$$(\max\{s \cdot n, 0\})^{shininess} * specular_{light} * specular_{material}]_i$$

5.7.5 Отделение зеркального цвета

Если для модели освещения цветовой контроль установлен в значение **GL_SEPARATE_SPECULAR_COLOR**, то для каждой вершины вычисляется два цвета (первичный и вторичный) в соответствии со следующими формулами:

первичный цвет = исходящий свет материала вершины +
глобальное фоновое освещение, умноженное на
фоновое свойство материала вершины +
фоновые и диффузные вклады всех источников света,
ослабленные соответствующим образом.

вторичный цвет = зеркальные вклады всех источников света,
ослабленные соответствующим образом.

Следующие два уравнения применяются **OpenGL** для расчета первичного и вторичного цветов:

$$\text{первичный цвет} = emission_{material} + ambient_{light\_model} * ambient_{material} +$$
$$\sum_{i=0}^{n-1} \frac{1}{k_c + k_l d + k_q d^2} * spotlight\_effect_i *$$
$$[ambient_{light} * ambient_{material} +$$
$$(\max\{L \cdot n, 0\}) * diffuse_{light} * diffuse_{material}]_i$$

$$\text{вторичный цвет} = \sum_{i=0}^{n-1} \frac{1}{k_c + k_l d + k_q d^2} * spotlight\_effect_i *$$

$$[(\max\{s \cdot n, 0\})^{shininess} * specular_{light} * specular_{material}]_i$$

В процессе текстурирования, с цветами текстуры комбинируется только первичный цвет. После текстурирования вторичный цвет добавляется к комбинации первичного и текстурных цветов.

5.8 Освещение в индексном цветовом режиме

В индексном режиме параметры, задаваемые **RGBA**величинами, либо не имеют эффекта, либо интерпретируются особым образом. Поскольку в индексном режиме намного сложнее достигнуть определенных эффектов, связанных с освещением, следует использовать **RGBA**режим везде, где это возможно. Вообще говоря, в индексном режиме используется всего три параметра в форме **RGBA.** Более конкретно, используются параметры источника света **GL_DIFFUSE**и **GL_SPECULAR**, а также параметр материала **GL_SHININESS. GL_DIFFUSE**и **GL_SPECULAR (**d_l и s_l, соответственно**)** используются для вычисления цветовых индексов диффузной и зеркальной световых интенсивностей **(**d_{ci}и s_{ci}**)** следующим образом:

$$d_{ci} = 0.30R(d_l) + 0.59G(d_l) + 0.11B(d_l)$$

$$s_{ci} = 0.30R(s_l) + 0.59G(s_l) + 0.11B(s_l)$$, где

R(x), G(x) и **B(x)** представляют соответственно красный, зеленый и синий компоненты цвета **x.** Весовые коэффициенты **0.30, 0.59** и **0.11** отражают степень восприятия компонентов цвета человеческим глазом **–** ваши глаза наиболее чувствительных зеленому и наименее чувствительны к синему цвету.

Чтобы задать цвета материала в индексном режиме, используйте **glMaterial*()** со специальным параметром **GL_COLOR_INDEXES:**

```
GLufloat mat_colormap[]={16.0,47.0,79.0};
glMaterialfv(GL_FRONT, GL_COLOR_INDEXES, mat_colormap);
```

Три числа заданные для **GL_COLOR_INDEXES** задают соответственно цветовые индексы фонового, диффузного и зеркального цветов материала. Другими словами **OpenGL** считает цвет с индексом, заданным в первом числе **(**в данном случае **16.0)** чистым фоновым цветом, цвет с индексом, заданным во втором числе **(47.0)** чистым диффузным цветом, а цвет с индексом, заданным в третьем числе **(79.0) –** чистым зеркальным цветом. **(**По умолчанию фоновый цвет имеет индекс **0.0**, а диффузный и зеркальный цвета **–** индекс **1.0.** Отметьте, что команда **glColorMaterial()** не работает в индексном режиме**.)**

Во время рисования сцены **OpenGL** использует цвета, ассоциированные с индексами между этими числами для закраски объектов сцены. Таким образом, вы должны верно построить цветовую таблицу между этими индексами **(**в данном примере между **16** и **47** и затем между **47** и **79).** Часто цветовая карта строится плавной, но вы можете создавать ее из других соображений для достижения определенных эффектов. Далее приведен пример плавной цветовой карты, начинающейся с черного фонового цвета, проходящей через фиолетовый диффузный цвет к белому зеркальному цвету:

```
for (i=0;i<32;i++)
```

```
{
   glutSetColor(16+i,1.0*(i/32.0),0.0,1.0*(i/32.0));
   glutSetColor(48+i,1.0,1.0*(i/32.0),1.0);
}
```

Команда **glutSetColor()** библиотеки **GLUT** принимает **4** аргумента. Она ассоциирует индекс, задаваемый первым аргументом с тройкой **RGB,** задаваемой вторым, третьим и четвертым аргументами. При **i=0** цветовой индекс **16** ассоциируется с черным **RGB**цветом **(0, 0, 0).** Цветовая карта плавно строится до диффузного цвета материала с индексом **47 (**при **i=31),** которому соответствует фиолетовый **RGB** цвет **(1.0, 0.0, 1.0).** Вторая команда внутри цикла строит карту между фиолетовым диффузным цветом и белым зеркальным цветом **(1.0, 1.0, 1.0)** в индексе **79.**

5.8.1 Математика освещения в индексном режиме

Дополнительно: Как вы можете ожидать, поскольку в индексном режиме доступно другое подмножество параметром освещения, нежели в **RGBA**режиме, расчеты также производятся иначе. Поскольку в индексном режиме исходящий от материала свет, а также фоновое освещение не задаются, здесь представляют интерес только диффузный и зеркальный вклады источников света. Даже эти элементы рассчитываются иначе.

Начнем с диффузного и зеркального элементов из **RGBA**уравнений. В диффузном элементе заменим $diffuse_{light} * diffuse_{material}$ на d_{ci}, вычисленное в предыдущем разделе для индексного режима. Аналогично, для зеркального элемента, вместо $specular_{light} * specular_{material}$ подставим s_{ci} из предыдущего раздела. **(**Ослабление, прожекторный эффект и другие компоненты этих элементов вычисляются также как в **RGBA**режиме.**)** Назовем эти модифицированные элементы ***d*** и ***s*** соответственно. Теперь, пусть $s` = \min(s,1)$. Вычислим $c = a_m + d`(1 - s`) + s`(s_m - a_m)$, где a_m, d_m и s_m представляют собой индексы фонового, диффузного и зеркального цветов материала, заданные командой **glMaterial*()** с аргументом **GL_COLOR_INDEXES.** Индекс результирующего цвета будет равен $c` = \min\{c, s_m\}$.

После того, как расчет освещенности произведен, индексные величины конвертируются в формат с фиксированной точкой **(**с неопределенным количеством битов справа от двоичной точки**).** Далее целая часть индекса маскируется **(**с помощью побитового И**)** с числом $2^n - 1$, где ***n*–** количество битов на цвет в индексном буфере.

Глава 6. Цветовое наложение, сглаживание, туман и смещение полигонов

В данной главе обсуждаются **4** техники, которые позволят вам **«**отполировать**»** ваши трехмерные сцены и внести в них дополнительные детали. Каждую из этих техник достаточно легко использовать **–** по правде говоря, их значительно тяжелее объяснить, чем использовать.

6.1 Цветовое наложение

Вы уже видели значения альфа **–** компонента в четверке **RGBA,** но до этого момента они игнорировались. Альфа **–** значения задаются при использовании команды **glColor*()**, при указании очищающего цвета командой **glClearColor()**, а также при определении некоторых параметров освещения, например, свойств материала или интенсивности освещения. Пиксели на мониторе излучают красный, зеленый и синий свет, а его количество контролируется соответствующими значениями красного, зеленого и синего. Так каким же образом значение альфа влияет на то, что будет нарисовано в окне на экране**?**

При включенном механизме цветового наложения, значение альфа часто используется для комбинирования цвета обрабатываемого фрагмента с цветом соответствующего ему пикселя, уже сохраненного в буфере кадра. Цветовое наложение происходит после того, как ваша сцена была растеризована и преобразована во фрагменты, но до того, как пиксели записываются в буфер кадра. Значения альфа, кроме того, могут быть использованы при альфа – тестировании для принятия или отбрасывания отдельных фрагментов на основании их альфа – значения.

Без цветового наложения каждый новый фрагмент заменяет любые цветовые значения в буфере кадра, как будто этот фрагмент является непрозрачным. С использованием цветового наложения вы можете управлять тем как (и в какой степени) существующие цветовые величины должны комбинироваться с цветами новых поступающих фрагментов. Таким образом, вы можете использовать цветовое наложение для создания полупрозрачных фрагментов, сквозь которые вида часть ранее сохраненного в буфере кадра цвета. Цветовое наложение необходимо для таких техник как прозрачность и цифровая композиция.

6.1 Цветовое наложение

Вы уже видели значения альфа – компонента в четверке **RGBA**, но до этого момента они игнорировались. Альфа – значения задаются при использовании команды **glColor*()**, при указании очищающего цвета командой **glClearColor()**, а также при определении некоторых параметров освещения, например, свойств материала или интенсивности освещения. Пиксели на мониторе излучают красный, зеленый и синий свет, а его количество контролируется соответствующими значениями красного, зеленого и синего. Так каким же образом значение альфа влияет на то, что будет нарисовано в окне на экране?

При включенном механизме цветового наложения, значение альфа часто используется для комбинирования цвета обрабатываемого фрагмента с цветом соответствующего ему пикселя, уже сохраненного в буфере кадра. Цветовое наложение происходит после того, как ваша сцена была растеризована и преобразована во фрагменты, но до того, как пиксели записываются в буфер кадра. Значения альфа, кроме того, могут быть использованы при альфа – тестировании для принятия или отбрасывания отдельных фрагментов на основании их альфа – значения.

Без цветового наложения каждый новый фрагмент заменяет любые цветовые значения в буфере кадра, как будто этот фрагмент является непрозрачным. С использованием цветового наложения вы можете управлять тем как (и в какой степени) существующие цветовые величины должны комбинироваться с цветами новых поступающих фрагментов. Таким образом, вы можете использовать цветовое наложение для создания полупрозрачных фрагментов, сквозь которые вида часть ранее сохраненного в буфере кадра цвета. Цветовое наложение необходимо для таких техник как прозрачность и цифровая композиция.

```
Замечание: Альфа – величины не задаются в индексном цветовом режиме, как
следствие, цветовое наложение в этом режиме недоступно.
```

Наиболее близкий к реальности способ осознания операций с цветовым наложением заключается в том, чтобы думать о **RGB–** тройке, как о цвете фрагмента, а об альфа – величине как о степени его непрозрачности (цветовой плотности). Прозрачные и полупрозрачные поверхности имеют плотность ниже, чем непрозрачные и, таким образом, обладают меньшем значением альфа. Например, если вы рассматриваете объект через зеленое стекло, его цвет будет частично зеленым (от стекла) и частично цветом самого объекта. Процентное соотношение этих цветов зависит от того, насколько хорошо стекло пропускает свет. Если оно пропускает **80%** поступающего на него света (то есть стекло цветовую имеет плотность равную **20%**), то цвет, который вы увидите, будет состоять на **20%** из цвета стекла и на **80%** из цвета объекта за ним. Вы

легко можете представить себе ситуации с множеством полупрозрачных поверхностей. Например, при рассматривании автомобиля снаружи, его интерьер виден вам через одно стекло, тогда как некоторые объекты за автомобилем могут быть видны через два стекла.

6.1.1 Факторы источника и приемника

Во время цветового наложения цветовые величины входящего фрагмента (*источника*) комбинируются с цветовыми величинами соответствующего, сохраненного к текущему моменту пикселя (*приемника*) в два этапа. Сначала вы задаете, каким образом следует вычислять факторы источника и приемника. Эти факторы представляют собой четверки **RGBA**, которые умножаются на **R**, **G**, **B** и **A** величины источника и приемника соответственно. Затем соответствующие компоненты в двух наборах **RGBA** четверок комбинируются между собой. Чтобы показать этот процесс с математической точки зрения, обозначим факторы источника и приемника соответственно (S_r, S_g, S_b, S_a) и (D_r, D_g, D_b, D_a), а цветовые компоненты источника и приемника обозначим нижним индексом ***s*** или ***d*** соответственно. Тогда результирующие цветовые величины после цветового наложения могут быть получены из следующего уравнения:

$$(R_s S_r + R_d D_r, G_s S_g + G_d D_g, B_s S_b + B_d D_b, A_s S_a + A_d D_a).$$

Каждый компонент этой четверки приводится к диапазону **[0,1]**.

По умолчанию фрагменты приемника и источника комбинируются путем суммирования их величин. Однако, если ваша реализация **OpenGL** поддерживает подмножество функций работы с изображениями, вы можете использовать другие математические операции для комбинирования фрагментов.

Теперь разберемся с тем, как генерируются факторы наложения. Вы используете команду **glBlendFunc()**, чтобы задать две константы: одна из них задает метод вычисления фактора источника, вторая – метод расчета фактора приемника. Чтобы производить цветовое наложение, вам также следует включить соответствующий механизм:

```
glEnable(GL_BLEND);
```

Для выключения наложения следует использовать команду **glDisable()** с аргументом **GL_BLEND**. Также обратите внимание, что использование констант **GL_ONE** (для источника) и **GL_ZERO** (для приемника) дает те же результаты, что и при выключенном цветовом наложении (именно эти значения установлены для факторов по умолчанию).

```
void glBlendFunc (GLenum sfactor, GLenum dfactor);
```

Управляет тем, как цветовые величины обрабатываемого фрагмента (источника) комбинируются с теми, которые уже сохранены в буфере кадра (приемника). Аргумент ***sfactor*** задает метод вычисления фактора наложения источника; аргумент ***dfactor*** -- метод вычисления фактора наложения приемника. Возможные значения этих параметров перечислены в таблице **6-1**. Вычисляемые факторы наложения лежат в диапазоне **[0,1]**; после того, как цветовые величины источника и приемника комбинируются, они также приводятся к диапазону **[0,1]**.

Замечание: В таблице 6-1 RGBA величины источника, приемника и константных цветов индицируются нижним индексом *s*, *d* или *c* соответственно. Вычитание четверок, означает покомпонентное вычитание одного их набора из другого.

Колонка «Принадлежность» показывает, к чему может быть применена константа: к источнику или к приемнику.

Таблица **6-1.** Факторы наложения источника и приемника

Константа	Принадлежность	Вычисляемый фактор
GL_ZERO	источник или приемник	**(0,0,0,0)**
GL_ONE	источник или приемник	**(1,1,1,1)**
GL_DST_COLOR	источник	(R_d, G_d, B_d, A_d)
GL_SRC_COLOR	приемник	(R_s, G_s, B_s, A_s)
GL_ONE_MINUS_DST_COLOR	источник	$(1,1,1,1) - (R_d, G_d, B_d, A_d)$
GL_ONE_MINUS_SRC_COLOR	приемник	$(1,1,1,1) - (R_s, G_s, B_s, A_s)$
GL_SRC_ALPHA	источник или приемник	(A_s, A_s, A_s, A_s)
GL_ONE_MINUS_SRC_ALPHA	источник или приемник	$(1,1,1,1) - (A_s, A_s, A_s, A_s)$
GL_DST_ALPHA	источник или приемник	(A_d, A_d, A_d, A_d)
GL_ONE_MINUS_DSR_ALPHA	источник или приемник	$(1,1,1,1) - (A_d, A_d, A_d, A_d)$
GL_SRC_ALPHA_SATURATE	источник	$(f, f, f, 1) : f = \min(A_s, 1 - A_d)$
GL_CONSTANT_COLOR	источник или приемник	(R_c, G_c, B_c, A_c)
GL_ONE_MINUS_CONSTANT_COLOR	источник или приемник	$(1,1,1,1) - (R_c, G_c, B_c, A_c)$
GL_CONSTANT_ALPHA	источник или приемник	(A_c, A_c, A_c, A_c)
GL_ONE_MINUS_CONSTANT_ALPHA	источник или приемник	$(1,1,1,1) - (A_c, A_c, A_c, A_c)$

Замечание: GL_CONSTANT_COLOR, GL_ONE_MINUS_CONSTANT_COLOR, GL_CONSTANT_ALPHA и GL_ONE_MINUS_CONSTANT_ALPHA поддерживаются только в том случае, если ваша реализация OpenGL поддерживает подмножество функций работы с изображениями.

6.1.2 Примеры использования цветового наложения

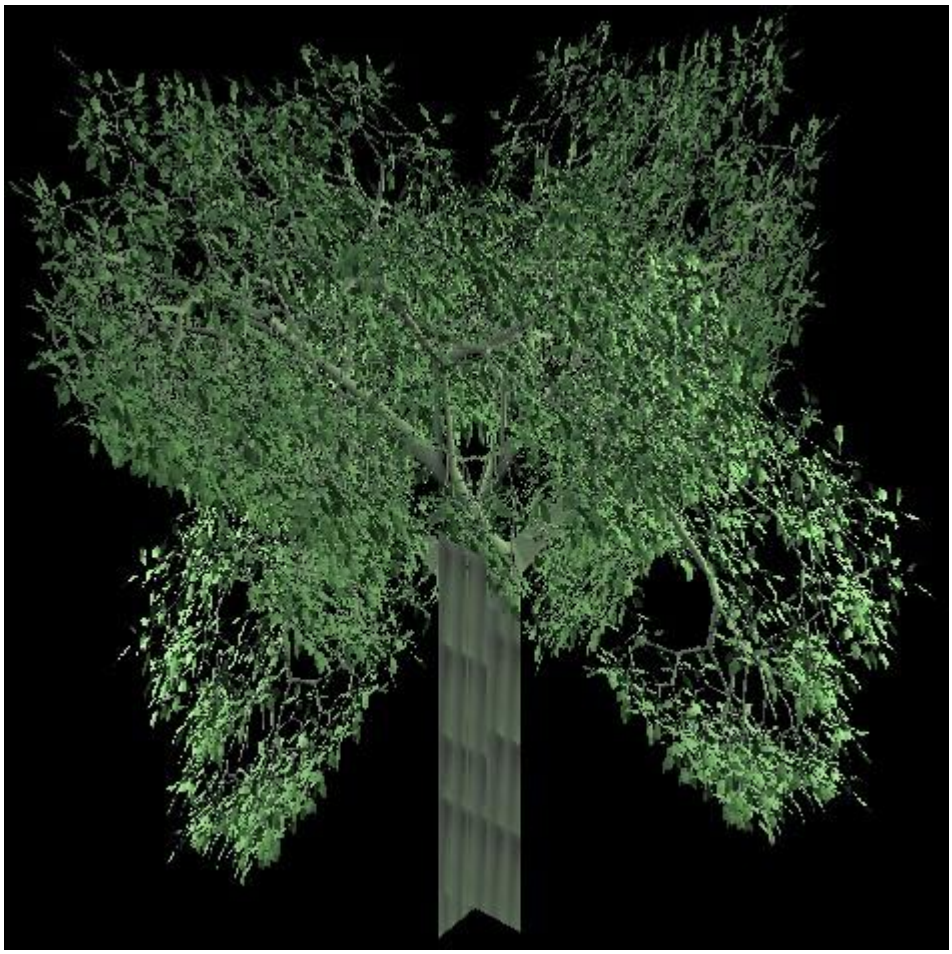
Не все комбинации факторов источника и приемника имеют смысл. Большинство приложений использует небольшое число этих комбинаций. Следующие параграфы описывают типичное использование конкретных комбинаций факторов источника и приемника. Некоторые из этих примеров используют только альфа – значение входящего фрагмента и, таким образом, могут работать даже в том случае, когда значения альфа не сохраняются в буфере кадра. Также имейте в виду, что часто существует более одного способа достичь определенного эффекта.

- Одним из способов нарисовать картинку, состоящую из половины одного изображения и половины второго, равномерно смешанных, является установка фактора источника в **GL_ONE**, а фактора приемника в **GL_ZERO** и отображение первого изображения. Затем следует установить фактор источника в **GL_SRC_ALPHA**, фактор приемника в **GL_MINUS_SRC_ALPHA** и нарисовать второе изображение с альфа равным **0.5**. Данная пара факторов, вероятно, представляет собой наиболее часто используемую операцию наложения. Если картинка должна быть смешана из **75%** первого изображения и **25%** второго, нарисуйте первое изображение как и раньше, затем нарисуйте второе с альфа равным **0.25**.
- Чтобы равномерно смешать три различных изображения, установите фактор приемника в **GL_ONE**, а фактор источника в **GL_SRC_ALPHA**. Нарисуйте каждое изображение с альфа равным **0.3333333**. При подобной технике каждое

изображение имеет только треть своей исходной яркости, что заметно в тех областях, где изображения не пересекаются.

- Предположим, что вы пишете программу рисования, и вам требуется создать кисть, постепенно добавляющую цвет таким образом, чтобы каждый росчерк кисти добавлял немного больше своего цвета к тому, что в данный момент находится на экране (например, каждый росчерк состоит из **10%** цвета и **90%** исходного изображения). Чтобы это сделать, нарисуйте изображение росчерка кисти с **10**-ти процентным альфа, используя **GL_SRC_ALPHA** для источника и **GL_ONE_MINUS_SRC_ALPHA** для приемника. Заметьте, что вы можете изменять альфа для разных частей росчерка кисти (например, чтобы цвет в центре росчерка был более ярким, чем по его границе). Похожим образом, ластик может быть реализован путем установки его цвета в цвет фона.
- Функции наложения, использующие цвета источника или приемника – **GL_DST_COLOR** или **GL_ONE_MINUS_DST_COLOR** для источника и **GL_SRC_COLOR** или **GL_ONE_MINUS_SRC_COLOR** для приемника – позволяют вам эффективно выделять отдельные цветовые компоненты. Данная операция эквивалентна наложению простого фильтра, например, умножение красного компонента на **0.80**, зеленого на **0.40**, а синего на **0.72** имитирует просмотр сцены через фотографический фильтр, блокирующий **20%** красного света, **60%** зеленого и **28%** синего.
- Предположим, что вам требуется отобразить три полупрозрачные поверхности на непрозрачном фоне. Части одних поверхностей закрывают другие. Предположим, что самая дальняя поверхность пропускает **80%** цвета, средняя – **40%** цвета, а ближняя – **90%**. Чтобы составить картинку, вначале нарисуйте задний фон с факторами источника и приемника по умолчанию, затем измените факторы на **GL_SRC_ALPHA** (для источника) и **GL_ONE_MINUS_SRC_ALPHA** (для приемника). Далее, нарисуйте самую дальнюю поверхность с альфа равным **0.2**, среднюю с альфа равным **0.6**, а самую ближнюю с альфа равным **0.1**.
- Если в вашей системе имеются альфа – плоскости, вы можете визуализировать объекты (включая их альфа - величины), считывать их обратно и затем производить интересные операции по их композиции с полностью визуализированными объектами. Обратите внимание, что объекты, используемые для композиции, могут приходить из любого источника – они могут быть построены с помощью команд **OpenGL**, получены с помощью других техник, например, трассировки лучей, реализованных в других графических библиотеках, или получены с помощью сканирования.
- Вы можете создать видимость непрямоугольного растрового изображения, назначив разные альфа – величины индивидуальным фрагментам изображения. В большинстве случаев вы назначаете альфа равную **0** каждому невидимому фрагменту и альфа равную **1.0** каждому непрозрачному фрагменту. Например, вы можете нарисовать полигон в форме дерева и наложить на него текстуру с изображением кроны; наблюдатель сможет видеть через части прямоугольной текстуры, не являющиеся частями дерева, если вы назначите им альфа равные **0**. Этот метод, иногда называемый ***billboarding*** (**billboard** – большая подсвечиваемая вывеска на здании), намного быстрее, чем создание дерева из трехмерных полигонов. Например, на рисунке **6-1** дерево создано из двух одинаковых полигонов повернутых под углом **90** градусов друг к другу. Буфер глубины выключен, при рисовании обоих полигонов использовались методы вычисления факторов **GL_SRC_ALPHA** (для источника) и **GL_ONE_MINUS_SRC_ALPHA** (для приемника).
- Цветовое наложение также используется для *антиалиасинга* – техники сокращения видимого лестничного эффекта в примитивах, рисуемых на растровом экране.

Рисунок **6-1.** Создание трехмерного дерева с помощью техники **billboarding:** файл **billboarding.cpp**

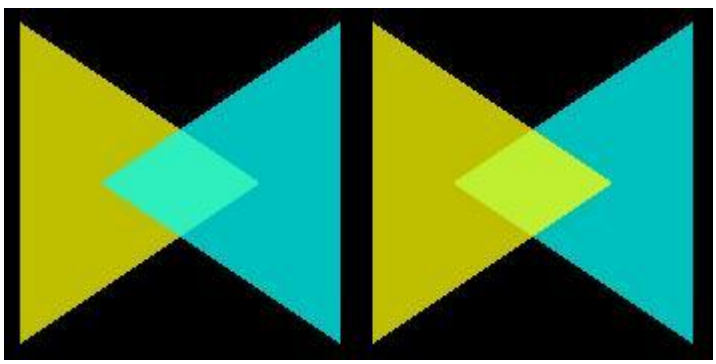


6.1.3 Пример цветового наложения

В примере **6-1** рисуются два перекрывающихся цветных треугольника, каждый из которых имеет альфа, равное **0.75.** Механизм цветового наложения включен, а факторы источника и приемника установлены в **GL_SRC_ALPHA**и **GL_ONE_MINUS_SRC_ALPHA** соответственно.

Когда программа стартует, слева отображается желтый треугольник, а справа – голубой. Таким образом, в центре окна, где треугольники пересекаются, голубой накладывается на желтый (рисунок **6-2**). Вы можете изменить порядок, в котором отображаются треугольники с помощью клавиши **'t'.**

Рисунок **6-2.** Цветовое наложение двух треугольников



Пример **6-1.** Пример цветового наложения: файл **alpha.cpp**

```
#include <GL/glut.h>

int leftFirst=GL_TRUE;

//Инициализация
void init(void)
{
   glEnable(GL_BLEND);
   glBlendFunc(GL_SRC_ALPHA,GL_ONE_MINUS_SRC_ALPHA);
   glShadeModel(GL_FLAT);
   glClearColor(0.0,0.0,0.0,0.0);
}

void drawLeftTriangle(void)
{
   //Нарисовать желтый треугольник на левой половине экрана
   glBegin(GL_TRIANGLES);
      glColor4f(1.0,1.0,0.0,0.75);
      glVertex3f(0.1,0.9,0.0);
      glVertex3f(0.1,0.1,0.0);
      glVertex3f(0.7,0.5,0.0);
   glEnd();
}

void drawRightTriangle(void)
{
   //Нарисовать голубой треугольник на правой половине экрана
   glBegin(GL_TRIANGLES);
      glColor4f(0.0,1.0,1.0,0.75);
      glVertex3f(0.9,0.9,0.0);
      glVertex3f(0.3,0.5,0.0);
      glVertex3f(0.9,0.1,0.0);
   glEnd();
}

void display(void)
{
   glClear(GL_COLOR_BUFFER_BIT);
   if(leftFirst)
   {
      drawLeftTriangle();
      drawRightTriangle();
   }
   else
   {
      drawRightTriangle();
      drawLeftTriangle();
   }
   glFlush();
}

void reshape(int w, int h)
{
   glViewport(0,0,(GLsizei) w, (GLsizei) h);
   glMatrixMode(GL_PROJECTION);
   glLoadIdentity();
   if (w<=h)
      gluOrtho2D(0.0,1.0,0.0,1.0*(GLfloat)h/(GLfloat)w);
   else
      gluOrtho2D(0.0,1.0*(GLfloat)w/(GLfloat)h,0.0,1.0);
}

void keyboard(unsigned char key,int x,int y)
{
   switch(key)
   {
      case 't':
      case 'T':
         leftFirst=!leftFirst;
         glutPostRedisplay();
         break;
      case 27:
         exit(0);
         break;
```

```
    }
}

int main(int argc, char** argv)
{
    glutInit(&argc,argv);
    glutInitDisplayMode(GLUT_SINGLE|GLUT_RGB);
    glutInitWindowSize(200,200);
    glutCreateWindow("Blending Example");
    init();
    glutDisplayFunc(display);
    glutReshapeFunc(reshape);
    glutKeyboardFunc(keyboard);
    glutMainLoop();
    return 0;
}
```

Порядок, в котором рисуются треугольники, влияет на цвет региона их перекрытия. Когда первым рисуется левый треугольник, голубые фрагменты (источник) смешиваются с желтыми фрагментами, которые уже находятся в буфере кадра (приемник). Когда первым рисуется правый треугольник, желтые фрагменты накладываются на голубые. Поскольку все альфа величины равны **0.75**, факторами наложения становятся значения **0.75** для источника и **1-0.75=0.25** для приемника. Другими словами фрагменты источника в некотором смысле прозрачны, но они оказывают больший эффект на результирующий цвет, чем фрагменты приемника.

6.1.4 Трехмерное цветовое наложение и буфер глубины

Как вы видели в предыдущем примере, порядок, в котором выводятся полигоны, значительно влияет на результат наложения. При рисовании трехмерных полупрозрачных объектов вы можете получить различный результат в зависимости от того, рисуете ли вы их от дальних к ближним или от ближних к дальним. Также при определении правильного порядка следует учитывать влияние буфера глубины. Буфер глубины отслеживает дистанцию между точкой наблюдения и частью объекта, занимающей определенный пиксель окна на экране; когда поступает следующий кандидат на тот же пиксель, он рисуется только если его объект ближе к точке наблюдения – в этом случае его величина глубины сохраняется в буфере глубины. При использовании этого метода загороженные (или скрытые) части поверхностей не обязательно будут нарисованы и, таким образом, не обязательно используются при наложении.

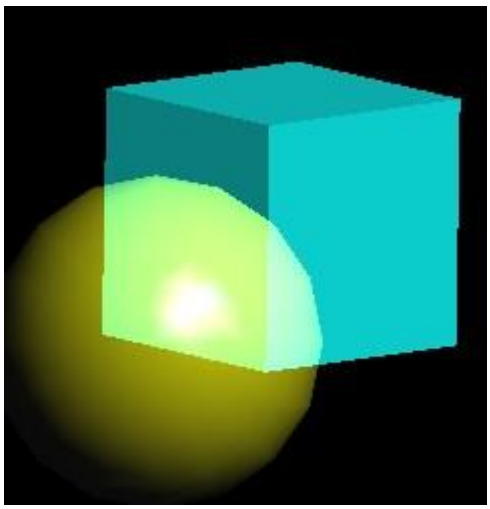
Если вам требуются и полупрозрачные и непрозрачные объекты в одной сцене, вам стоит использовать буфер глубины для удаления невидимых поверхностей любых объектов, лежащих за непрозрачными объектами. Если непрозрачный объект закрывает полупрозрачный объект или другой непрозрачный объект, стоит использовать буфер глубины для удаления более дальнего объекта. Однако, если полупрозрачный объект ближе, вам нужно выполнить цветовое наложение между ним и непрозрачным объектом. Довольно легко выяснить требуемый правильный порядок рисования полигонов в случае, когда объекты сцены стационарны, но проблема может быстро стать очень сложной, если точка наблюдения или объект перемещается.

Решение заключается в том, чтобы включить глубинную буферизацию, но в период рисования полупрозрачных объектов сделать буфер глубины доступным только для чтения. Сначала вы рисуете все непрозрачные объекты (в это время буфер глубины находится в нормальном состоянии). Затем вы делаете величины глубины неизменяемыми, переводя буфер глубины в состояние «только для чтения». Во время рисования полупрозрачных объектов их величины глубины по-прежнему сравниваются с величинами, установленными непрозрачными объектами, таким образом, полупрозрачные объекты не рисуются, если они находятся за непрозрачными. Если же они находятся ближе к точке наблюдения, они не удаляют непрозрачные объекты, поскольку величины буфера глубины не могут быть изменены. Вместо этого

выполняется их цветовое наложение на непрозрачные объекты. Для управления тем, можно ли изменять величины буфера глубины, используйте команду **glDepthMask()**; если вы передадите в качестве аргумента **GL_FALSE**, буфер станет доступным только для чтения, в то время как аргумент **GL_TRUE** восстанавливает возможность записи в него.

Пример **6-2** демонстрирует использование данного метода для рисования непрозрачных и прозрачных объектов в трехмерном пространстве. Внутри программы, клавиша **'a'** запускает анимационную последовательность, в процессе которой полупрозрачный куб проходит через непрозрачную сферу (рисунок **6-3**). Нажатие клавиши **'r'** приводит объекты в начальное состояние. Когда полупрозрачные объекты пересекаются, для получения наилучших результатов рисуйте объекты от дальних к ближним.

Рисунок **6-3.** Полупрозрачный куб и непрозрачная сфера



Пример **6-2.** Трехмерное цветовое наложение: файл **alpha3D.cpp**

```
#include <GL/glut.h>

#define MAXZ 8.0
#define MINZ -8.0
#define ZINC 0.02

float solidZ=MAXZ;
float transparentZ=MINZ;
GLuint sphereList,cubeList;

void init(void)
{
   GLfloat mat_specular[]={1.0,1.0,1.0,0.15};
   GLfloat mat_shininess[]={100.0};
   GLfloat position[]={0.5,0.5,1.0,0.0};

   glMaterialfv(GL_FRONT,GL_SPECULAR,mat_specular);
   glMaterialfv(GL_FRONT,GL_SHININESS,mat_shininess);
   glLightfv(GL_LIGHT0,GL_POSITION,position);
   glEnable(GL_LIGHTING);
   glEnable(GL_LIGHT0);
   glEnable(GL_DEPTH_TEST);
   sphereList=glGenLists(1);
   glNewList(sphereList,GL_COMPILE);
      glutSolidSphere(0.4,16,16);
   glEndList();
   cubeList=glGenLists(1);
   glNewList(cubeList,GL_COMPILE);
      glutSolidCube(0.6);
   glEndList();
}
```

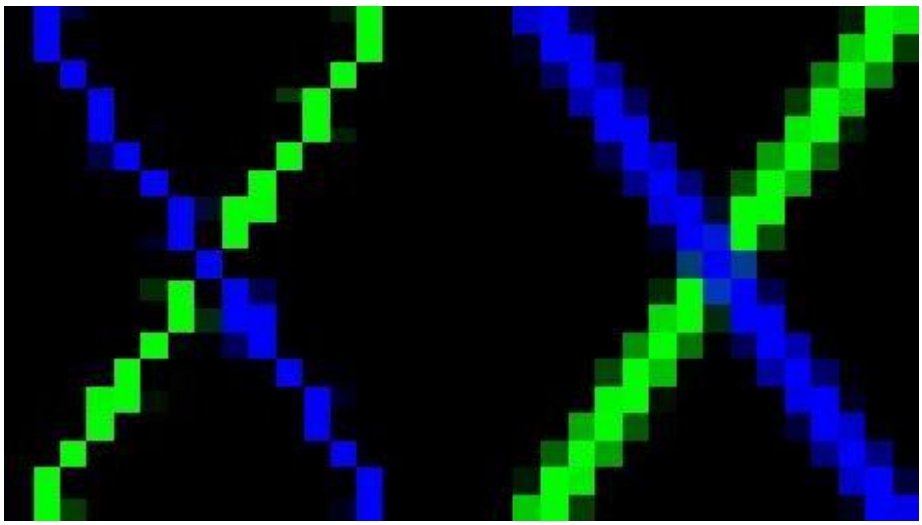
```
void display(void)
{
   GLfloat mat_solid[]={0.75,0.75,0.0,1.0};
   GLfloat mat_zero[]={0.0,0.0,0.0,1.0};
   GLfloat mat_transparent[]={0.0,0.8,0.8,0.6};
   GLfloat mat_emission[]={0.0,0.3,0.3,0.6};

   glClear(GL_COLOR_BUFFER_BIT|GL_DEPTH_BUFFER_BIT);
   glPushMatrix();
      glTranslatef(-0.15,-0.15,solidZ);
      glMaterialfv(GL_FRONT,GL_EMISSION,mat_zero);
      glMaterialfv(GL_FRONT,GL_DIFFUSE,mat_solid);
      glCallList(sphereList);
   glPopMatrix();
   glPushMatrix();
      glTranslatef(0.15,0.15,transparentZ);
      glRotatef(15.0,1.0,1.0,0.0);
      glRotatef(30.0,0.0,1.0,0.0);
      glMaterialfv(GL_FRONT,GL_EMISSION,mat_emission);
      glMaterialfv(GL_FRONT,GL_DIFFUSE,mat_transparent);
      glEnable(GL_BLEND);
      glDepthMask(GL_FALSE);
      glBlendFunc(GL_SRC_ALPHA,GL_ONE);
      glCallList(cubeList);
      glDepthMask(GL_TRUE);
      glDisable(GL_BLEND);
   glPopMatrix();
   glutSwapBuffers();
}

void reshape(int w, int h)
{
   glViewport(0,0,(GLsizei) w, (GLsizei) h);
   glMatrixMode(GL_PROJECTION);
   glLoadIdentity();
   if (w<=h)
      glOrtho(-1.5,1.5,-0.5*(GLfloat)h/(GLfloat)w,1.5*(GLfloat)h/(GLfloat)w,-
10.0,10.0);
   else
      glOrtho(-1.5*(GLfloat)w/(GLfloat)h,1.5*(GLfloat)w/(GLfloat)h,-1.5,1.5,-
10.0,10.0);
   glMatrixMode(GL_MODELVIEW);
   glLoadIdentity();
}

void animate()
{
   if(solidZ<=MINZ || transparentZ >=MAXZ)
      glutIdleFunc(NULL);
   else
   {
      solidZ-=ZINC;
      transparentZ+=ZINC;
      glutPostRedisplay();
   }
}

void keyboard(unsigned char key,int x, int y)
{
   switch (key)
   {
      case 'a':
      case 'A':
         solidZ=MAXZ;
         transparentZ=MINZ;
         glutIdleFunc(animate);
         break;
      case 'r':
      case 'R':
         solidZ=MAXZ;
         transparentZ=MINZ;
         glutPostRedisplay();
         break;
```

```
      case 27:
         exit(0);
   }
}

int main(int argc, char** argv)
{
   glutInit(&argc,argv);
   glutInitDisplayMode(GLUT_DOUBLE|GLUT_RGB|GLUT_DEPTH);
   glutInitWindowSize(500,500);
   glutCreateWindow("Three-Dimensional Blending");
   init();
   glutDisplayFunc(display);
   glutReshapeFunc(reshape);
   glutKeyboardFunc(keyboard);
   glutMainLoop();
   return 0;
}
```

6.2 Антиалиасинг

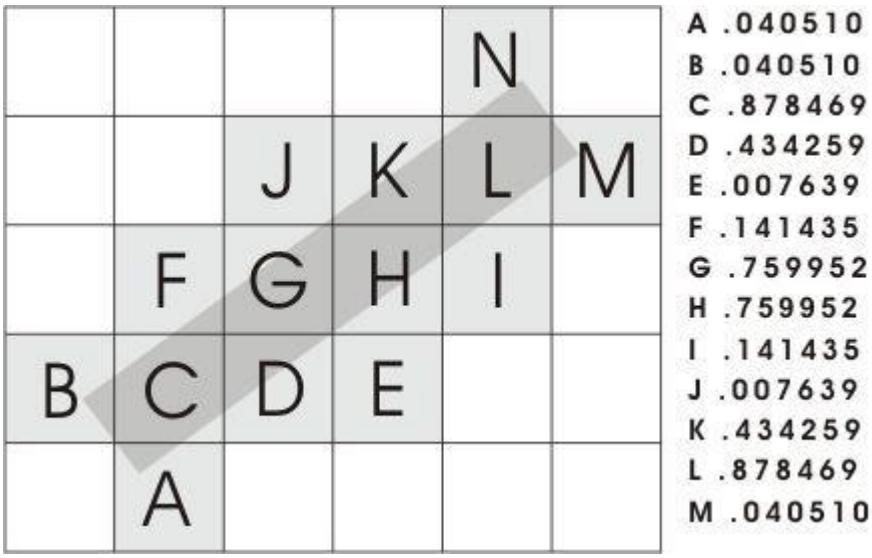
На некоторых изображениях, созданных с помощью **OpenGL** вы могли заметить, что линии выглядят «ступенчатыми», особенно если они близки к горизонтальным или вертикальным. Эти ступеньки появляются потому, что идеальная линия аппроксимируется сериями пикселей из пиксельной решетки экрана. «Ступенчатость» называется *алиасинг* (***aliasing***), и в данном разделе описывается техника антиалиасинга, позволяющая ее устранить. На рисунке **6-4** показаны две пары пересекающихся линий: слева пара несглаженных, а справа пара сглаженных путем антиалиасинга. Для демонстрации эффекта изображение было увеличено.

Рисунок **6-4.** Сглаженные и несглаженные линии



На рисунке **6-5** показано, что диагональная линия в **1** пиксель толщиной покрывает площадь некоторых пикселей в большей степени, чем других. Вообще говоря, когда производится антиалиасинг, **OpenGL** рассчитывает *величину покрытия* для каждого фрагмента, базируясь на доле покрываемой им площади пикселя на экране. Рисунок **6-5** демонстрирует эти величины для линии. В **RGBA** режиме **OpenGL** умножает эту величину на альфа величину фрагмента. Затем вы можете использовать получившееся альфа для цветового наложения фрагмента на соответствующий пиксель, уже находящийся в буфере кадра. В индексном цветовом режиме, **OpenGL** устанавливает **4** младших бита цветового индекса в зависимости от величины покрытия фрагмента **(0000 –** если фрагмент не попадает на пиксель и **1111 –** при полном покрытии фрагментом пикселя**).** Загрузка цветовой таблицы и ее верное применение для получения выгоды от этих величин покрытия лежит на вас.

Рисунок **6-5.** Определение величин покрытия



Подробности расчета величины покрытия довольно сложны, трудны для объяснения и вообще говоря, могут быть несколько различными в зависимости от вашей конкретной реализации **OpenGL.** Вы можете использовать команду **glHint()** для некоторого контроля над соотношением между качеством изображения и скоростью, однако не все реализации **OpenGL** реагируют на нее.

```
void glHint (GLenum target, GLenum hint);
```

Управляет определенными аспектами поведения **OpenGL.** Параметр ***target*** задает, какой именно аспект поведения контролируется, его возможные значения показаны в таблице **6-2.** Значениями для параметра ***hint*** может быть **GL_FASTEST (**для самого эффективного по скорости метода действий**), GL_NICEST(**для самого высококачественного метода**)** или **GL_DONT_CARE(**для метода по выбору библиотеки**).** Интерпретация этих установок зависит от реализации, конкретная реализация **OpenGL** может вообще игнорировать эти установки.

Целевая установка **GL_PERSPECTIVE_CORRECTION_HINT** относится к тому, как цветовые величины и координаты текстуры интерполируются внутри примитива: линейно в экранном пространстве **(**относительно простой метод вычислений**)** или путем перспективной коррекции **(**которая требует больших вычислений**).** Часто системы производят линейную цветовую интерполяцию, поскольку результаты хотя и не являются технически верными, но визуально приемлемы; однако текстуры в большинстве случаев требуют интерполяции с коррекцией перспективы для того, чтобы быть визуально приемлемыми. Следовательно, реализация **OpenGL** может интерпретировать этот параметр для управления тем, какой метод интерполяции использовать.

Таблица **6-2.** Значения, используемые в команде **glHint()**

Параметр	Смысл
GL_POINT_SMOOTH_HINT **GL_LINE_SMOOTH_HINT** **GL_POLYGON_SMOOTH_HINT**	задают желаемое количество сэмплов и следовательно качество сглаживания точек, линий и полигонов соответственно
GL_FOG_SMOOTH_HINT	задает, должен ли расчет тумана производиться для каждого пикселя **(GL_NICEST)** или для каждой вершины **(GL_FASTEST)**
GL_PERSPECTIVE_CORRECTION_HINT	задает желаемое качество интерполяции цветов и координат текстуры

6.2.1 Сглаживание точек и линий

Для сглаживания точек или линий вам нужно включить антиалиасинг командой **glEnable()**, передав ей в качестве аргумента **GL_POINT_SMOOTH**или **GL_LINE_SMOOTH** соответственно. Вы также можете задать параметр качества с помощью **glHint()**. **(**Помните, что вы можете задавать размер точек и толщину линий**.)** Далее следуйте процедурам, описанным в одном из следующих разделов в зависимости от того, работаете ли вы в **RGBA**или индексном цветовом режиме.

6.2.1.1 Антиалиасинг в RGBA режиме

В режиме **RGBA** вам нужно включить механизм цветового наложения. Факторы, которые вы, скорее всего, будете при этом использовать **– GL_SRC_ALPHA (**для источника**)** и **GL_ONE_MINUS_SRC_ALPHA (**для приемника**).** в качестве альтернативы вы можете задать фактор **GL_ONE(**для приемника**),** что сделает линии несколько более яркими в точках пересечения. Теперь вы можете рисовать любые точки или линии **–** они будут сглажены. Эффект от сглаживания наиболее заметен в случае использования достаточно высоких значений альфа. Помните, что поскольку вы используете цветовое наложение, вам, возможно, понадобится следить за порядком выбора фигур. Однако, в большинстве случаем, порядок вывода может быть проигнорирован без значительных визуальных последствий. В примере **6-3** инициализируются все необходимые режимы, и затем отображаются две пересекающиеся диагональные линии. Нажатие на клавишу **'r'** во время работы программы поворачивает линии, чтобы можно было рассмотреть эффект от сглаживания линии, нарисованных под разными углами. Заметьте, что в данном примере выключен буфер глубины.

Пример **6-3.** Сглаженные линии: файл **aargb.cpp**

```
#include <GL/glut.h>

float rotAngle=0;

void init(void)
{
   glEnable(GL_LINE_SMOOTH);
   glEnable(GL_BLEND);
   glBlendFunc(GL_SRC_ALPHA,GL_ONE_MINUS_SRC_ALPHA);
   glHint(GL_LINE_SMOOTH_HINT,GL_NICEST);
   glLineWidth(1.5);
   glClearColor(0.0,0.0,0.0,0.0);
}

void display(void)
{
   glClear(GL_COLOR_BUFFER_BIT);
   glColor3f(0.0,1.0,0.0);
   glPushMatrix();
      glRotatef(-rotAngle,0.0,0.0,0.1);
      glBegin(GL_LINES);
         glVertex2f(-0.5,0.5);
         glVertex2f(0.5,-0.5);
      glEnd();
   glPopMatrix();
   glColor3f(0.0,0.0,1.0);
   glPushMatrix();
      glRotatef(rotAngle,0.0,0.0,0.1);
      glBegin(GL_LINES);
         glVertex2f(0.5,0.5);
         glVertex2f(-0.5,-0.5);
      glEnd();
   glPopMatrix();
glFlush();
}

void reshape(int w, int h)
{
   glViewport(0,0,(GLsizei) w, (GLsizei) h);
   glMatrixMode(GL_PROJECTION);
   glLoadIdentity();
```

```
   if (w<=h)
      gluOrtho2D(-1.0,1.0,-1.0*(GLfloat)h/(GLfloat)w,1.0*(GLfloat)h/(GLfloat)w);
   else
      gluOrtho2D(-1.0*(GLfloat)w/(GLfloat)h,1.0*(GLfloat)w/(GLfloat)h,-1.0,1.0);
   glMatrixMode(GL_MODELVIEW);
   glLoadIdentity();
}

void keyboard(unsigned char key, int x, int y)
{
   switch(key)
   {
      case 'r':
      case 'R':
         rotAngle+=10;
         if (rotAngle>=360)
            rotAngle=0;
         glutPostRedisplay();
         break;
      case 27:
         exit(0);
         break;
   }
}

int main(int argc, char** argv)
{
   glutInit(&argc,argv);
   glutInitDisplayMode(GLUT_SINGLE|GLUT_RGB);
   glutInitWindowSize(200,200);
   glutCreateWindow("Antialiased Lines");
   init();
   glutDisplayFunc(display);
   glutReshapeFunc(reshape);
   glutKeyboardFunc(keyboard);
   glutMainLoop();
   return 0;
}
```

6.2.1.2 Антиалиасинг в индексном режиме

Самая хитрая часть антиалиасинга в индексном режиме заключается в загрузке и использовании цветовой таблицы. Поскольку последние **4** бита цветового индекса индицируют величину покрытия, вам следует загрузить **16** идущих один за другим индексов цветами градиентами от цвета фона до цвета объекта. **(**Каждый из градиентов должен начинаться с величины индекса кратной **16.)** Затем вы очищаете цветовой буфер с помощью первого индекса в градиенте и рисуете точки и линии с помощью других его цветов. Пример **6-4** демонстрирует, как конструировать градиент для рисования сглаженных линий в индексном режиме. В этом примере создается два цветовых градиента: один содержит оттенки зеленого, другой – оттенки синего.

Пример **6-4.** Сглаживание в индексном режиме: файл **aaindex.cpp**

```
#include <GL/glut.h>

#define RAMPSIZE 16
#define RAMP1START 32
#define RAMP2START 48

float rotAngle=0;

void init(void)
{
   int i;

   for (i=0;i<RAMPSIZE;i++)
   {
      GLfloat shade;
      shade=(GLfloat)i/(GLfloat)RAMPSIZE;
```

```
      glutSetColor(RAMP1START+(GLint)i,0.,shade,0.);
      glutSetColor(RAMP2START+(GLint)i,0.,0.,shade);
   }
   glEnable(GL_LINE_SMOOTH);
   glHint(GL_LINE_SMOOTH_HINT,GL_NICEST);
   glLineWidth(1.5);
   glClearIndex((GLfloat)RAMP1START);
}

void display(void)
{
   glClear(GL_COLOR_BUFFER_BIT);
   glIndexi(RAMP1START);
   glPushMatrix();
      glRotatef(-rotAngle,0.0,0.0,0.1);
      glBegin(GL_LINES);
         glVertex2f(-0.5,0.5);
         glVertex2f(0.5,-0.5);
      glEnd();
   glPopMatrix();
   glIndexi(RAMP2START);
   glPushMatrix();
      glRotatef(rotAngle,0.0,0.0,0.1);
      glBegin(GL_LINES);
         glVertex2f(0.5,0.5);
         glVertex2f(-0.5,-0.5);
      glEnd();
   glPopMatrix();
   glFlush();
}

void reshape(int w, int h)
{
   glViewport(0,0,(GLsizei) w, (GLsizei) h);
   glMatrixMode(GL_PROJECTION);
   glLoadIdentity();
   if (w<=h)
      gluOrtho2D(-1.0,1.0,-1.0*(GLfloat)h/(GLfloat)w,1.0*(GLfloat)h/(GLfloat)w);
   else
      gluOrtho2D(-1.0*(GLfloat)w/(GLfloat)h,1.0*(GLfloat)w/(GLfloat)h,-1.0,1.0);
   glMatrixMode(GL_MODELVIEW);
   glLoadIdentity();
}

void keyboard(unsigned char key,int x,int y)
{
   switch(key)
   {
      case 'r':
      case 'R':
         rotAngle+=10;
         if (rotAngle>=360) rotAngle=0;
            glutPostRedisplay();
         break;
      case 27:
         exit(0);
         break;
   }
}

int main(int argc, char** argv)
{
   glutInit(&argc,argv);
   glutInitDisplayMode(GLUT_SINGLE|GLUT_INDEX);
   glutInitWindowSize(200,200);
   glutCreateWindow("Antialiasing in Color-Index mode");
   init();
   glutDisplayFunc(display);
   glutReshapeFunc(reshape);
   glutKeyboardFunc(keyboard);
   glutMainLoop();
   return 0;
}
```

Поскольку градиент проходит от цвета фона до цвета объекта, сглаженные линии выглядят верно только в тех местах, где они рисуются поверх фона. Когда рисуется синяя линия, она затирает часть зеленой в точке их пересечения. Чтобы это исправить, вам требуется перерисовать область пересечения линий с помощью градиента, проходящего от зеленого (цвета в буфере кадра) до синего (цвета рисуемой линии). Однако, это требует дополнительных расчетов и обычно не стоит усилий, поскольку область пересечения линий достаточно мала. Заметьте, что это не является проблемой в **RGBA**режиме, поскольку цвета рисуемых линий накладываются на цвета, уже находящиеся в буфере кадра.

Вы также можете захотеть включить тест глубины при рисовании сглаженных точек и линий в индексном режиме. В предыдущем примере тест глубины выключен, поскольку обе линии рисуются в одной плоскости по **z**. Однако если вы захотите нарисовать трехмерную сцену, вам следует включить буфер глубины, чтобы итоговые цвета пикселей соответствовали ближайшим объектам.

Трюк, описанный в разделе «Трехмерное цветовое наложение» может быть также использован для смешивания сглаженных точек и линий с несглаженными полигонами в режиме с включенным буфером глубины. Чтобы это сделать, сначала нарисуйте полигоны, затем сделайте буфер глубины доступным только для чтения и нарисуйте точки и линии. Точку и линии будут пересекаться между собой, однако будут загораживаться ближайшими полигонами.

6.2.2 Сглаживание полигонов

Сглаживание ребер залитых полигонов похоже на сглаживание точек и линий. Когда разные полигоны накладываются друг на друга, вам нужно накладывать цвета соответствующим образом. Для сглаживания вы можете использовать метод, описываемый в этом разделе или буфер аккумулятора (сглаживающий всю сцену). Использование буфера аккумуляции может показаться более простым, но оно весьма интенсивно по производимым расчетам и, как следствие, весьма медленно. Однако, как вы увидите далее, метод, описываемый здесь, более сложен для воспроизведения.

Замечание: Если вы рисуете ваши полигоны в виде точек в вершинах или границы – то есть, передавая аргумент GL_POINT или GL_LINE команде **glPolygonMode()** – применяется техника сглаживания точек или линий, описанная ранее (если сглаживание включено). Оставшаяся часть данного раздела посвящена сглаживанию полигонов в режиме их отображения GL_FILL.

В теории, вы можете сглаживать полигоны и в **RGBA**, и в индексном цветовых режимах. Однако пересечение объектов оказывает на сглаживание полигонов значительно большее воздействие, чем оно влияет на сглаживание точек и линий, и, следовательно, порядок отображения и точность наложения играют критическую роль. На сомом деле они грают настолько критическую роль, что если вы сглаживаете более чем один полигон, вам нужно упорядочивать полигоны от передних к задним и использовать команду **glBlendFunc()** с параметрами **GL_SRC_ALPHA_SATURATE** (для источника) и **GL_ONE** (для приемника). Следовательно, сглаживание полигонов в индексном режиме не является практически применимым.

Чтобы сглаживать полигоны в **RGBA**режиме, вы используете альфа величины для представления величины покрытия ребер полигона. Вам следует включить сглаживание полигонов передачей аргумента **GL_POLYGON_SMOOTH** команде **glEnable()**. Это приведет к тому, что пикселям на ребрах полигона будут присвоены частичные альфа величины в зависимости от величины покрытия, так же как при сглаживании точек и линий. Также, если вы захотите, вы можете задать величину для **GL_POLYGON_SMOOTH_HINT**. Теперь вам следует должным образом наложить друг на друга перекрывающиеся ребра. Во-первых, выключите буфер глубины, чтобы иметь контроль над тем, как накладываются друг на друга перекрывающиеся пиксели. Затем установите факторы наложения в **GL_SRC_ALPHA_SATURATE** (для источника) и **GL_ONE**

(для приемника). С заданной таким образом функцией наложения результирующий цвет будет представлять собой сумму цвета приемника и масштабированного цвета источника. Фактор масштабирования будет одновременно меньше и входящей величины альфа и единицы минус альфа величины приемника. Это означает, что входящие пиксели имеют небольшой эффект на результирующий цвет для пикселей с большой величиной альфа, поскольку единица минус альфа приемника почти равна нулю. При использовании этого метода пиксель на ребре рисуемого полигона может быть совмещен с цветом полигона, нарисованного ранее. В итоге, до того чтобы рисовать, вам требуется упорядочить все полигоны сцены в порядке от более близких к более дальним.

6.3 Туман

Иногда компьютерные изображения могут казаться нереально четкими и резкими. Антиалиасинг делает объекты более реалистичными за счет сглаживания их ребер. Кроме того, вы можете увеличить реалистичность всего изображения, добавив *туман*, который постепенно размывает объекты в зависимости от дистанции до наблюдателя. «Туман» -- это довольно общий термин, описывающий несколько похожих форм атмосферных явлений (дым от горения, дымка, туман над болотом, смог и так далее). Туман просто необходим для различных программ визуальной симуляции, которым требуется имитация ограниченной видимости. Он часто применяется в симуляторах полетов.

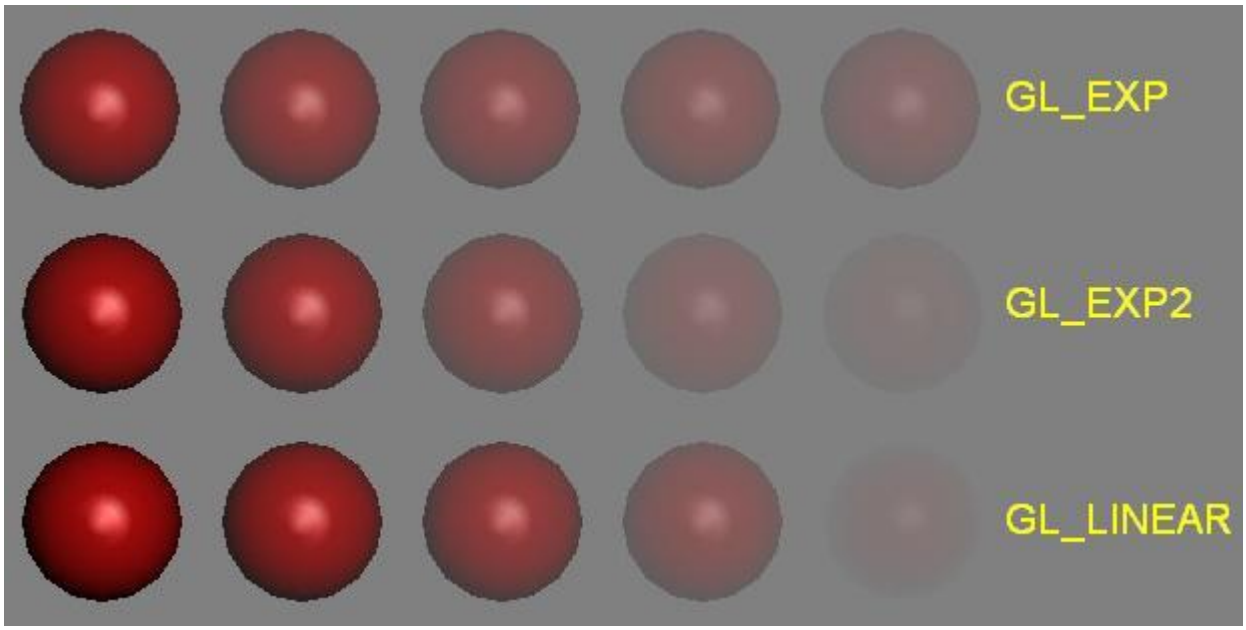
Когда включен туман, объекты, находящиеся дальше от наблюдателя как бы таят в цвете тумана. Вы можете управлять плотностью тумана, которая определяет уровень воздействия дистанции от наблюдателя на цвет и размытость объекта. Туман может быть использован и в **RGBA**, и в индексном режиме, однако расчеты, связанные с ним, в этих режимах несколько различаются. Поскольку расчет тумана производится после матричных преобразований, расчета освещенности и текстурирования, он воздействует на преобразованные, освещенные и текстурированные объекты. Заметьте, что применение тумана может увеличить быстродействие приложений симуляции, поскольку вы можете не рисовать объекты, которые находятся слишком далеко и слишком размыты туманом.

Все типы примитивов, включая точки и линии, могут быть затуманены. Использование тумана для точек и линий также называется ***depth – cuing*** (*глубинирование* – при использовании этой техники линии находящиеся дальше от наблюдателя отображаются более блеклыми) и весьма популярно в программах молекулярного моделирования и других приложениях.

6.3.1 Использование тумана

Использовать туман очень просто. Вы включаете его, передавая аргумент **GL_FOG**, команде **glEnable()**. Также вы выбираете цвет тумана и уравнение, управляющее его плотностью, с помощью команды **glFog*()**. Кроме того, вы можете установить качество расчета тумана с помощью аргумента **GL_FOG_HINT**команды **glHint()**. В примере **6-5** отображаются **5** красных сфер, все на разных дистанциях от наблюдателя. Клавиша **'f'** выбирает уравнение плотности тумана. Сами уравнения объясняются далее.

Рисунок **6-6**. Пять затуманенных сфер



Пример **6-5.** Пять затуманенных сфер в **RGBA** режиме**:** файл **fog.cpp**

```
#include <windows.h>
#include <GL/glut.h>

GLint fogMode;

void init(void)
{
   GLfloat position[]={0.5,0.5,3.0,0.0};
   glEnable(GL_DEPTH_TEST);
   glLightfv(GL_LIGHT0,GL_POSITION,position);
   glEnable(GL_LIGHTING);
   glEnable(GL_LIGHT0);
   {
      GLfloat mat[3]={0.1745,0.01175,0.01175};
      glMaterialfv(GL_FRONT,GL_AMBIENT,mat);
      mat[0]=0.61424;mat[1]=0.04135;mat[2]=0.04136;
      glMaterialfv(GL_FRONT,GL_DIFFUSE,mat);
      mat[0]=0.727811;mat[1]=0.626959;mat[2]=0.626959;
      glMaterialfv(GL_FRONT,GL_SPECULAR,mat);
      glMaterialf(GL_FRONT,GL_SHININESS,0.6*128.0);
   }
   glEnable(GL_FOG);
   {
      GLfloat fogColor[4]={0.5,0.5,0.5,1.0};
      fogMode=GL_EXP;
      glFogi(GL_FOG_MODE,fogMode);
      glFogfv(GL_FOG_COLOR,fogColor);
      glFogf(GL_FOG_DENSITY,0.35);
      glHint(GL_FOG_HINT,GL_NICEST);
      glFogf(GL_FOG_START,1.0);
      glFogf(GL_FOG_END,5.0);
   }
   glClearColor(0.5,0.5,0.5,1.0);
}

void renderSphere(GLfloat x ,GLfloat y, GLfloat z)
{
   glPushMatrix();
   glTranslatef(x,y,z);
   glutSolidSphere(0.4,16,16);
   glPopMatrix();
}

void display(void)
{
   glClear(GL_COLOR_BUFFER_BIT|GL_DEPTH_BUFFER_BIT);
```

```
   renderSphere(-2.,-0.5,-1.0);
   renderSphere(-1.,-0.5,-2.0);
   renderSphere(0.,-0.5,-3.0);
   renderSphere(1.,-0.5,-4.0);
   renderSphere(2.,-0.5,-5.0);
   glFlush();
}

void reshape(int w, int h)
{
   glViewport(0,0,(GLsizei) w, (GLsizei) h);
   glMatrixMode(GL_PROJECTION);
   glLoadIdentity();
   if (w<=h)
      glOrtho(-2.5,2.5,-2.5*(GLfloat)h/(GLfloat)w,2.5*(GLfloat)h/(GLfloat)w,-
10.0,10.0);
   else
      glOrtho(-2.5*(GLfloat)w/(GLfloat)h,2.5*(GLfloat)w/(GLfloat)h,-2.5,2.5,-
10.0,10.0);
   glMatrixMode(GL_MODELVIEW);
   glLoadIdentity();
}

void keyboard(unsigned char key,int x, int y)
{
   switch (key)
   {
      case 'f':
      case 'F':
         if(fogMode==GL_EXP)
         {
            fogMode=GL_EXP2;
            MessageBox(NULL,"Fog mode is GL_EXP2","Five Fogged Spheres in RGBA
mode",MB_OK);
         }
         else
            if(fogMode==GL_EXP2)
            {
               fogMode=GL_LINEAR;
               MessageBox(NULL,"Fog mode is GL_LINEAR","Five Fogged Spheres in
RGBA mode",MB_OK);
            }
            else
               if(fogMode==GL_LINEAR)
               {
                  fogMode=GL_EXP;
                  MessageBox(NULL,"Fog mode is GL_EXP","Five Fogged Spheres in
RGBA mode",MB_OK);
               }
         glFogi(GL_FOG_MODE,fogMode);
         glutPostRedisplay();
         break;
      case 27:
         exit(0);
         break;
   }
}

int APIENTRY WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, LPSTR
lpCmdLine, int nCmdShow)
{
   char* argv="";
   int argc=0;
   glutInit(&argc,&argv);
   glutInitDisplayMode(GLUT_SINGLE|GLUT_RGB|GLUT_DEPTH);
   glutInitWindowSize(500,500);
   glutCreateWindow("Five Fogged Spheres in RGBA mode");
   init();
   glutDisplayFunc(display);
   glutReshapeFunc(reshape);
   glutKeyboardFunc(keyboard);
   glutMainLoop();
   return 0;
}
```

6.3.2 Уравнения расчета плотности тумана

Туман накладывает свой цвет на цвет входящего фрагмента с учетом фактора наложения тумана. Этот фактор ***f***, вычисляется с помощью одного из трех следующих уравнений и затем усекается до диапазона **[0, 1]**.

$$f = e^{-(density \cdot z)} \qquad \text{(GL\_EXP)}$$

$$f = e^{-(density \cdot z)^2} \qquad \text{(GL\_EXP2)}$$

$$f = \frac{end - z}{end - start} \qquad \text{(GL\_LINEAR)}$$

В этих трех уравнениях ***z***– дистанция в видовых координатах от точки наблюдения до центра фрагмента. Величины ***density***, ***start*** и ***end*** задаются командой **glFog*()**. Фактор **f**используется по-разному в зависимости от того, работаете ли вы в **RGBA** или в индексном цветовом режиме.

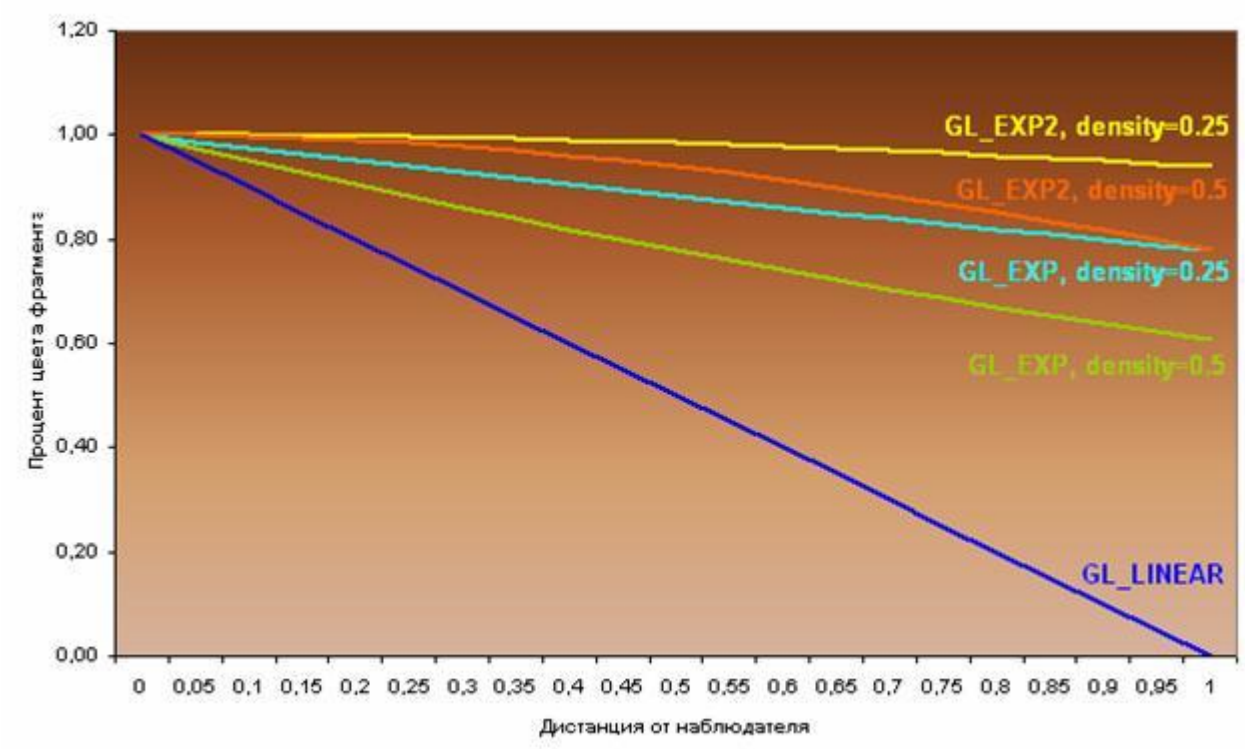
```
void glFog{if} (GLenum pname, TYPE param);
void glFog{if}v (GLenum pname, TYPE *params);
```

Задает параметры и функцию для вычисления тумана. Если ***pname*** имеет значение **GL_FOG_MODE**, то ***param*** может принимать значения **GL_EXP (**значение по умолчанию**)**, **GL_EXP2** или **GL_LINEAR** и задает метод вычисления фактора тумана. Если ***pname*** равен **GL_FOG_DENSITY**, **GL_FOG_START**или **GL_FOG_END**, то ***param*** содержит **(**или для векторной версии команды указывает на**)** величины ***density***, ***start*** или ***end*** для использования в уравнениях. **(**Значения по умолчанию **– 1**, **0** и **1** соответственно.**)** В **RGBA** режиме ***pname*** может также содержать значение **GL_FOG_COLOR**, в этом случае ***params*** указывает на четверку величин, задающую **RGBA** цвет тумана.

Соответствующее значение ***pname*** для индексного режима **– GL_FOG_INDEX**, в этом случае ***param*** должен содержать единственную величину, задающую цветовой индекс тумана.

На рисунке **6-7** изображены графики факторов наложения тумана, вычисленных с помощью различных уравнений и разных значений переменных.

Рисунок **6-7.** Уравнения плотности тумана



6.3.2.1 Туман в RGBA режиме

В **RGBA** режиме фактор тумана используется для вычисления финального затуманенного цвета следующим образом:

$$C = fC_i + (1-f)C_f,$$

где C_i представляет **RGBA** величины входящего фрагмента, а C_f -- цвет тумана, заданный аргументом **GL_FOG_COLOR.**

6.3.2.2 Туман в индексном режиме

В индексном режиме индекс финального затуманенного цвета вычисляется следующим образом:

$$I = I_i + (1-f)I_f,$$

где I_i представляет собой цветовой индекс входящего фрагмента, а I_f -- индекс цвета тумана, заданный аргументом **GL_FOG_INDEX.**

Чтобы использовать туман в индексном режиме, вам требуется должным образом загрузить величины в цветовую карту. Первый цвет в карте должен быть цветом объекта без тумана, а последний – цветом полностью затуманенного объекта. Возможно, вы захотите использовать **glClearIndex()**, чтобы инициализировать цвет заднего фона индексом последнего цвета в карте; таким образом, полностью затуманенные объекты будут совмещены с задним фоном. Похожим образом, до того, как объекты нарисованы, вам следует вызвать команду **glIndex*()** и передать ей индекс первого цвета в карте (цвета незатуманенного объекта). Кроме того, если вы хотите нарисовать несколько затуманенных объектов разного цвета, вам следует загрузить соответствующее количество цветовых карт и вызывать **glIndex*()** перед

рисованием каждого объекта. Пример **6-6** иллюстрирует инициализацию всех нужных аспектов для использования тумана в индексном режиме.

Пример **6-6.** Туман в индексном режиме: файл **fogindex.cpp**

```
#include <windows.h>
#include <GL/glut.h>

#define NUM_COLORS 32
#define RAMPSTART 16

void init(void)
{
   int i;

   glEnable(GL_DEPTH_TEST);
   for(i=0;i<NUM_COLORS;i++)
   {
      GLfloat shade;
      shade=(GLfloat)(NUM_COLORS-i)/(GLfloat)NUM_COLORS;
      glutSetColor(RAMPSTART+i,shade,shade,shade);
   }
   glEnable(GL_FOG);
   glFogi(GL_FOG_MODE,GL_LINEAR);
   glFogi(GL_FOG_INDEX,NUM_COLORS);
   glFogf(GL_FOG_DENSITY,0.35);
   glFogf(GL_FOG_START,1.0);
   glFogf(GL_FOG_END,6.0);
   glHint(GL_FOG_HINT,GL_NICEST);
   glClearIndex((GLfloat)(NUM_COLORS+RAMPSTART-1));
}

void renderSphere(GLfloat x ,GLfloat y, GLfloat z)
{
   glPushMatrix();
   glTranslatef(x,y,z);
   glutWireSphere(0.4,16,16);
   glPopMatrix();
}

void display(void)
{
   glClear(GL_COLOR_BUFFER_BIT|GL_DEPTH_BUFFER_BIT);
   glIndexi(RAMPSTART);
   renderSphere(-2.,-0.5,-1.0);
   renderSphere(-1.,-0.5,-2.0);
   renderSphere(0.,-0.5,-3.0);
   renderSphere(1.,-0.5,-4.0);
   renderSphere(2.,-0.5,-5.0);
   glFlush();
}

void reshape(int w, int h)
{
   glViewport(0,0,(GLsizei) w, (GLsizei) h);
   glMatrixMode(GL_PROJECTION);
   glLoadIdentity();
   if (w<=h)
      glOrtho(-2.5,2.5,-2.5*(GLfloat)h/(GLfloat)w,2.5*(GLfloat)h/(GLfloat)w,-
10.0,10.0);
   else
      glOrtho(-2.5*(GLfloat)w/(GLfloat)h,2.5*(GLfloat)w/(GLfloat)h,-2.5,2.5,-
10.0,10.0);
   glMatrixMode(GL_MODELVIEW);
   glLoadIdentity();
}

int APIENTRY WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, LPSTR
lpCmdLine, int nCmdShow)
{
   char* argv="";
   int argc=0;
   glutInit(&argc,&argv);
```

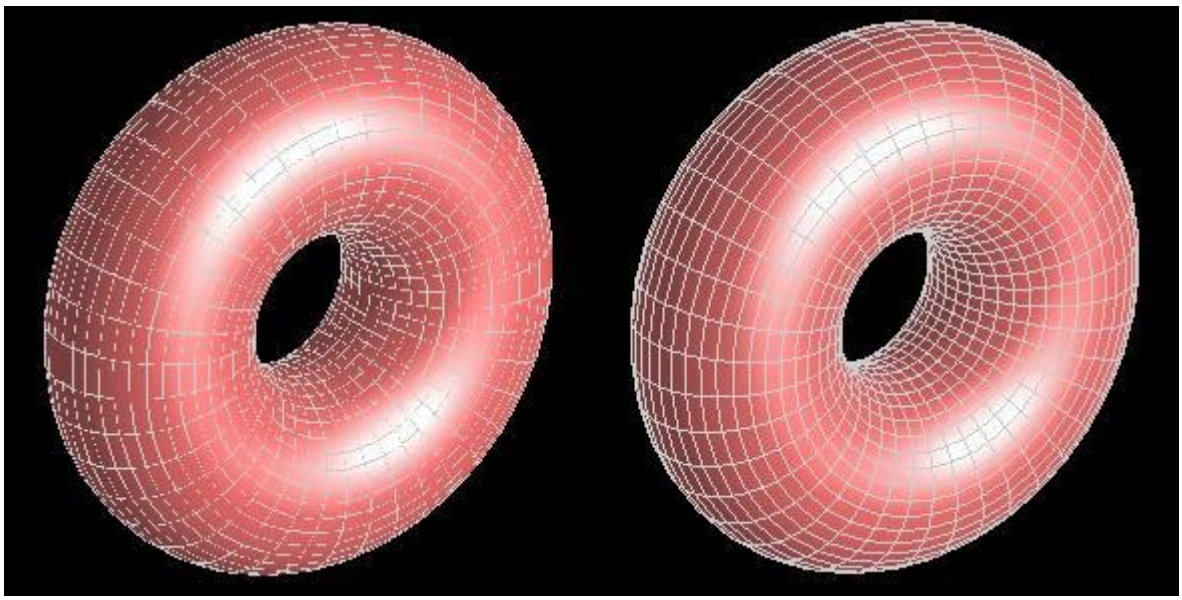
```
    glutInitDisplayMode(GLUT_SINGLE|GLUT_INDEX|GLUT_DEPTH);
    glutInitWindowSize(500,500);
    glutCreateWindow("Fog in Color-Index Mode");
    init();
    glutDisplayFunc(display);
    glutReshapeFunc(reshape);
    glutMainLoop();
    return 0;
}
```

6.4 Смещение полигонов

Если вы хотите выделить ребра залитого объекта, вы можете попробовать нарисовать его сначала в режиме **GL_FILL**, а затем перерисовать его снова, но другим цветом и в режиме **GL_LINE.** Однако поскольку линии и залитые полигоны растеризуются не совсем одинаковым образом, величины глубины, сгенерированные для линий не совсем те же самые, что сгенерированы для ребер полигона, даже между двумя одинаковыми вершинами. Выделяемые ребра могут входить и выходить из смежного полигона, который в результате выглядит недостаточно качественно и часто называется **«***простроченным***» («*stiching*»).**

Недостатки внешнего вида могут быть устранены путем использования полигонального смещения, которое вносит необходимое смещение, форсируя четкое разделение смежных ребер и линий по глубине. **(**Для этого также может быть использован буфер трафарета, но полигональное смещение, как правило, работает быстрее.**)** Полигональное смещение также весьма удобно использовать для добавления выделения к поверхностям, визуализации изображений с удалением невидимых линий. Помимо линий и залитых полигонов эта техника может быть применена к точкам. На обеих частях рисунка **6-8** сначала был нарисован освещенный торус в режиме заливки, а затем такой же торус, но другим цветом и в режиме контура, однако слева полигональное смещение выключено, а справа включено.

Рисунок **6-8.** Залитые и проволочные торусы с полигональным смещением **(**справа**)** и без него **(**слева**)**



Существует три различных способа включить полигональное смещение, по одному для каждого режима растеризации полигонов**: GL_FILL, GL_LINE, GL_POINT.** Вы включаете полигональное смещение, передавая соответствующий аргумент команде **glEnable() – GL_POLYGON_OFFSET_FILL, GL_POLYGON_OFFSET_LINE** или

GL_POLYGON_OFFSET_POINT. Вы также должны вызвать **glPolygonMode()**, чтобы выбрать текущий режим растеризации полигонов.

```
void glPolygonOffset (GLfloat factor, GLfloat units);
```

Если полигональное смещение включено, величина глубины каждого фрагмента добавляется к вычисленной величине смещения. Смещение добавляется до того, как производится тест глубины и до того, как величина глубины записывается в буфер глубины. Величина смещения вычисляется следующим образом:

$$o = m \cdot factor + r \cdot units,$$

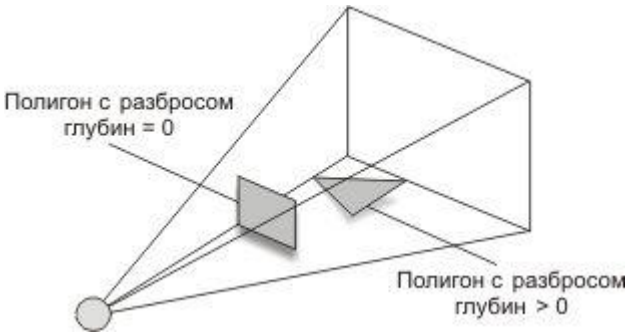
где ***m*** – максимальное разброс глубины вершин полигона, а ***r*** – минимальная величина, гарантирующая разумное различие по глубине в оконных координатах. Величина ***r*** – константа, зависимая от реализации.

Чтобы достигнуть приемлемых результатов и избежать визуальных артефактов, вы можете либо добавить положительное смещение к залитому объекту (отодвинуть его от себя), либо добавить отрицательное смещение к каркасу (придвинуть его к себе). Самый сложный вопрос заключается в том, насколько большое смещение стоит выбирать? К сожалению, нужное смещение зависит от многих факторов, в частности от максимального разброса в глубине вершин полигона и толщины линий каркаса.

OpenGL вычисляет разброс глубины вершин полигона самостоятельно, но важно, чтобы вы понимали, что есть этот разброс, поскольку это позволит вам выбирать осмысленные значения для аргумента ***factor***. Разброс глубины – это изменение в величине глубины, деленное на изменение в **x** или **y** – координатах, пересекающих полигон. Величины глубины при этом измеряются в оконных координатах, усеченных до диапазона **[0, 1]**. Чтобы вычислить разброс глубин полигона, используйте формулу

$$m = \max\left\{\left|\frac{\partial V}{\partial s}\right|, \left|\frac{\partial V}{\partial t}\right|\right\}.$$

Рисунок **6-9.** Полигоны и их разброс глубин



Для полигонов, которые параллельны ближней или дальней отсекающим плоскостям, разброс глубин равен **0**. Для полигонов в вашей сцене, разброс глубин которых близок к нулю, требуется лишь небольшая величина смещения. Чтобы создать небольшое, постоянное смещение, вы можете передать в команду **glPolygonOffset()** ***factor*=0.0** и ***units*=1.0**.

Для полигонов с большим углом к плоскостям отсечения, разброс глубин может быть значительно больше нуля, и может потребоваться большее смещение. Небольшая, неравная нулю величина для ***factor***, например **0.75** или **1.0** может быть вполне

достаточной для генерации величин глубины, необходимых для устранения визуальных артефактов.

Пример **6-7** демонстрирует часть кода с применением списка отображения **(**рисующим залитый объект**).** Сначала объект отображается с освещением, режимом полигонов **GL_FILL** и полигональном смещением с величинами ***factor*=1.0** и ***units*=1.0.** Данные величины гарантируют, что смещение достаточно для всех полигонов в вашей сцене, независимо от разброса глубины. **(**Эти величины могут быть даже больше минимально необходимых, однако чуть-чуть большее, чем нужно полигональное смещение заметно меньше, чем чуть-чуть меньшее.**)** Затем, для выделения ребер первого объекта, объект отображается еще раз в виде неосвещенного каркаса с выключенным смещением.

Пример **6-7.** Использование полигонального смещения для удаления визуальных артефактов

```
glEnable(GL_LIGHTING); glEnable(GL_LIGHT0);
glEnable(GL_POLYGON_OFFSET_FILL);
glPolygonOffset(1.0,1.0);
glCallList(list);
glDisable(GL_POLYGON_OFFSET_FILL);
glDisable(GL_LIGHTING);
glDisable(GL_LIGHT0);
glColor3f(1.0,1.0,1.0);
glPolygonMode(GL_FRONT_AND_BACK,GL_LINE);
glCallList(list);
glPolygonMode(GL_FRONT_AND_BACK,GL_FILL);
```

В небольшом числе ситуаций простейшие значения для величин ***factor*** и **units (1.0**, и **1.0)** не являются решениями проблемы. Например, если длина линий, выделяющих ребра больше **1**, может быть необходимо увеличение значения ***factor***. Также, поскольку величины глубины неравномерно преобразуются в оконные координаты при перспективном проецировании, меньшее смещение требуется для полигонов, которые находятся ближе к ближней отсекающей плоскости и большее для полигонов, находящихся дальше от нее. Таким образом, требуются эксперименты со значениями ***factor***.

Глава 7. Списки отображения

Список отображения (дисплейные списки) – это группа команд **OpenGL**, сохраненная для дальнейшего исполнения. Когда исполняется список отображения, команды, включенные в него, исполняются в том порядке, в котором они были заданы. Большинство команд **OpenGL** могут быть как сохранены в списке отображения, так и выполняться в непосредственном режиме, в котором они выполняются немедленно. В одной программе вы можете смело смешивать код непосредственного режима с использованием списков отображения. Во всех примерах программ до этого момента все команды исполнялись в непосредственном режиме. В данной главе обсуждаются списки отображения и то, как их лучше использовать.

7.1 Зачем использовать списки отображения

Списки отображения могут увеличить быстродействие, поскольку вы можете сохранять в них команды **OpenGL** для дальнейшего исполнения. Хорошей идеей является кэширование команд в списке отображения, если вы планируете рисовать какую-либо геометрию несколько раз или если несколько раз устанавливаете одно и то же состояние настроек. Используя списки отображения, вы можете определить геометрические данные или изменения в состоянии один раз и затем исполнять их столько раз, сколько вам нужно.

Чтобы разобраться с использование списков, представьте себе рисование трехколесного велосипеда. Два задних колеса имеют одинаковый размер, но смещены относительно друг друга. Переднее колесо больше по размеру, чем задние и находится в другом месте. Эффективным способом визуализации колес велосипеда будет сохранение геометрии для одного колеса в списке отображения, и затем его исполнение три раза. Вам следует устанавливать видовую матрицу каждый раз перед исполнением списка для вычисления правильного размера и положения каждого колеса.

Особенно важно кэшировать команды **OpenGL** в списках отображения, когда приложение исполняется на удаленной машине. В этом случае, сервер и хост физически являются разными машинами. Поскольку списки отображения являются частью состояния сервера и, таким образом, сохраняются на серверной машине, вы можете сократить время на постоянную передачу этих данных по сети, если сохраните часто используемые команды в списках отображения.

При локальном исполнении, вы часто также можете увеличить быстродействие путем сохранения часто используемых команд в списках отображения. Некоторая графическая аппаратура может сохранять списки отображения в выделенной для этого памяти или хранить данные в оптимизированной форме, которая более совместима с аппаратным или программным обеспечением.

7.2 Пример использования списка отображения

Список отображения это удобный и эффективный путь именования и организации набора команд **OpenGL**. Предположим, например, что вы хотите нарисовать торус и смотреть на него под разными углами. Наиболее эффективный способ сделать это заключается в том, чтобы сохранить торус в списке отображения. После этого, если вы хотите изменить угол обзора, все что вам нужно сделать, это изменить видовую матрицу и вызвать список отображения к исполнению.

Пример **7-1.** Создание списка отображения: файл **torus.cpp**

```
#include <glut.h>
#include <math.h>

GLuint theTorus;

//Отрисовка тора
void torus(int numc,int numt)
{
   int i,j,k;
   double s,t,x,y,z,twopi;
   twopi=2*(double) 3.1415;
   for(i=0;i<numc;i++)
   {
      glBegin(GL_QUAD_STRIP);
         for(j=0;j<=numt;j++)
         {
            for(k=1;k>=0;k--)
            {
               s=(i+k)%numc+0.5;
               t=j%numt;
               x=(1+.1*cos(s*twopi/numc))*cos(t*twopi/numt);
               y=(1+.1*cos(s*twopi/numc))*sin(t*twopi/numt);
               z=.1*sin(s*twopi/numc);
               glVertex3f(x,y,z);
            }
         }
      glEnd();
   }
}

void init(void)
```

```
{
   theTorus=glGenLists(1);
   glNewList(theTorus,GL_COMPILE);
      torus(8,25);
   glEndList();
   glShadeModel(GL_FLAT);
   glClearColor(0.0,0.0,0.0,0.0);
}

void display()
{
   glClear(GL_COLOR_BUFFER_BIT);
   glColor3f(1.0,1.0,1.0);
   glCallList(theTorus);
   glFlush();
}

void reshape(int w,int h)
{
   glViewport(0,0,(GLsizei)w,(GLsizei)h);
   glMatrixMode(GL_PROJECTION);
   glLoadIdentity();
   gluPerspective(30,(GLfloat)w/(GLfloat)h,1.0,100.0);
   glMatrixMode(GL_MODELVIEW);
   glLoadIdentity();
   gluLookAt(0,0,10,0,0,0,0,1,0);
}

//"x" -- повернуть вокруг x -оси;"y" -- повернуть вокруг y -оси;"i"--вернуться в
начальное состояние
void keyboard(unsigned char key,int x,int y)
{
   switch(key)
   {
      case 'X':
      case 'x':
         glRotatef(30.,1.0,0.0,0.0);
         glutPostRedisplay();
         break;
      case 'Y':
      case 'y':
         glRotatef(30.,0.0,1.0,0.0);
         glutPostRedisplay();
         break;
      case 'I':
      case 'i':
         glLoadIdentity();
         gluLookAt(0,0,10,0,0,0,0,1,0);
         glutPostRedisplay();
         break;
      case 27:
         exit(0);
   }
}

int main(int argc, char** argv)
{
   glutInit(&argc,argv);
   glutInitDisplayMode(GLUT_SINGLE|GLUT_RGB);
   glutInitWindowSize(200,200);
   glutCreateWindow("Creating a Display List");
   init();
   glutDisplayFunc(display);
   glutReshapeFunc(reshape);
   glutKeyboardFunc(keyboard);
   glutMainLoop();
   return 0;
}
```

Начнем с рассмотрения функции **init().** Она создает список отображения для торуса и инициализирует состояние **OpenGL.** Заметьте**,** что вызов функции рисования торуса **(torus())** заключена в командные скобки **glNewList()** и **glEndList(),** которые определяют список отображения**.** Аргумент ***listName*** команды **glNewList()**

представляет собой целочисленный индекс, сгенерированный командой **glGenLists()** и однозначно идентифицирующий конкретный список отображения.

Пользователь может поворачивать торус вокруг осей ***x***и ***y***, нажимая клавиши **'x'** и **'y'**, соответственно. Каждый раз, когда это случается, вызывается функция обратного вызова **keyboard()**, которая комбинирует текущую видовую матрицу с матрицей поворота вокруг соответствующей оси. Затем вызывается функция **glutPostRedisplay()**, которая заставляет **glutMainLoop()** вызвать функцию **display()** и вывести торус после того, как будут обработаны все остальные события. Когда нажимается клавиша **'i'**, функция **keyboard()** восстанавливает начальную видовую матрицу и возвращает торус к начальному положению.

Сама функция **display()** крайне проста. Она очищает окно и затем вызывает команду **glCallList()** для исполнения команд в списке отображения. Если бы мы не использовали списки отображения, в функции **display()** следовало бы вызывать все команды для отображения торуса.

Список отображения содержит только команды **OpenGL**. В примере **7-1** в списке отображения сохраняются только команды **glBegin()**, **glEnd()** и **glVertex()**. Параметры для этих вызовов *вычисляются* и после этого копируются в список отображения при его создании. Все тригонометрические расчеты для создания торуса выполняются только один раз, что увеличивает быстродействие. Однако величины в списке отображения не могут быть изменены в дальнейшем, и после того как команда была сохранена в списке отображения, не существует способа извлечь ее из него. Также невозможно добавлять новые команды в список отображения, после того как он был определен. Вы можете удалить список целиком и создать новый, но не можете изменять его.

Замечание: Списки отображения также отлично работают с командами из библиотеки утилит GLU, поскольку эти команды могут быть разбиты на низкоуровневые команды OpenGL и сохранены в списках отображения. Использование списков отображения совместно с командами GLU особенно важно для увеличения быстродействия при работе с тесселяцией и NURBS.

7.3 Философия разработки списков отображения

В целях оптимизации быстродействия, списки отображения сделаны больше похожими на кэш команд, чем на динамическую базу данных. Другими словами, как только список создан, он не может быть изменен. Если бы списки были бы изменяемыми, быстродействие снизилось бы за счет необходимости затрат на поиск по ним и управление памятью. Из-за изменения частей списка, пространство в памяти выделялось и освобождалось бы частями, что вело бы к фрагментации памяти. Кроме того, для модификации частей списка требовалось бы обратить все изменения, которые реализация **OpenGL** внесла в него в целях оптимизации. Помимо прочего, вероятно, что получить доступ к спискам было бы весьма трудно, так как они могут кэшироваться где-либо в сети или системной шине.

Способ, которым оптимизируются команды в списке отображения, может меняться от реализации к реализации. За счет базовых команд вроде **glRotate()** можно получить достаточно большой выигрыш по производительности, поместив их в списки отображения, поскольку вычисления, необходимые для построения матрицы поворота нетривиальны **(**и могут включать вычисление квадратных корней и тригонометрические функции**)**. Однако в списке отображения можно сохранить только результирующую матрицу поворота, поэтому команда поворота в списке отображения может быть выполнена настолько быстро, насколько быстро аппаратура может выполнить **glMultMatrix*()**. Наиболее продвинутые реализации **OpenGL** могут даже комбинировать смежные команды преобразований в одну финальную матрицу.

Хотя и не гарантируется, что ваша конкретная реализация **OpenGL** оптимизирует списки отображения для каких-либо конкретных нужд, в любом случае исполнение списка не может быть медленнее, чем непосредственное индивидуальное исполнение всех команд, содержащихся в нем. Некоторая потеря быстродействия может, однако, возникнуть из необходимости переходов от списка к списку. Если конкретный список достаточно мал, то эта потеря может свести на нет выгоду от его использования. Далее перечислены наиболее вероятные случаи, в которых можно оптимизировать быстродействие за счет списков отображения.

- Матричные операции. Большинство матричных операций **OpenGL** вычисления обратных матриц. Обе матрицы – расчетная и обратная к ней могут быть сохранены в списке отображения реализацией **OpenGL**.
- Битовые карты и изображения. Формат, в котором вы задаете растровые данные, наверняка не является тем, который идеален для аппаратуры. При компиляции списка отображения, **OpenGL** может преобразовать данные к представлению, предпочтительному для аппаратуры. Это может значительно повлиять на скорость рисования растровых символов, строки символов обычно состоят из серий небольших битовых карт.
- Источники света, свойства материалов и модели освещения. Когда вы рисуете сцену со сложными условиями освещения, вы можете изменять материалы для каждого элемента сцены. Настройка материалов может быть достаточно медленной, поскольку требует ряда вычислений. Если вы поместите определение материала в список отображения, эти вычисления не будут производиться каждый раз при переключении материала, поскольку необходимо сохранить только результаты вычислений; в результате визуализация и вывод освещенных сцен может совершаться значительно быстрее.
- Текстуры. Вы можете максимизировать эффективность при определении текстур скомпилировав их в список отображения, поскольку список отображения может кэшировать изображение текстуры в выделенной текстурной памяти. После этого не требуется копировать изображение текстуры каждый раз, когда оно понадобилось. Кроме того, аппаратный формат текстуры может отличаться от формата **OpenGL**, и конверсия может быть выполнена во время компиляции списка отображения, а не во время отображения сцены. В **OpenGL** версии **1.0** списки отображения были главным методом управления текстурами. Однако если вы работаете с **OpenGL** версии **1.1** или выше, вам следует сохранять текстуры в *объекте текстуры*. **(**Некоторые реализации версии **1.0** имеют специфичные для производителя расширения для поддержки объектов текстуры. Если ваша реализация поддерживает объекты текстуры, вам следует использовать их.**)**
- Рисунки шаблонов полигона.

Некоторые из команд, используемых для задания свойств, перечисленных здесь, являются контекстно-чувствительными, и это следует принимать в расчет для получения оптимального быстродействия. Например, когда включен режим цвета материала **(GL_COLOR_MATERIAL)**, некоторые из параметров материала будут отслеживать текущий цвет. Любые вызовы **glMaterial*()** для установки тех же параметров будут игнорироваться.

Сохранение установок переменных состояния вместе с геометрическими данными может увеличить быстродействие. Например, предположим, что вы хотите трансформировать некоторые геометрические объекты и затем вывести результат. Код для этого может быть таким:

```
glNewList(1, GL_COMPILE);
   нарисовать_некоторые_объекты();
glEndList();
glLoadMatrix(M);
glCallList(1);
```

Однако, если геометрический объект каждый раз преобразовывается одинаковым образом, лучше сохранить матрицу преобразования в списке отображения. Например, Если вы напишите код следующим образом, некоторые реализации **OpenGL** могут преобразовать объект по время его определения, а не во время его рисования:

```
glNewList(1, GL_COMPILE);
   glLoadMatrix(M);
   нарисовать_некоторые_объекты();
glEndList();
glCallList(1);
```

Более частая ситуация возникает при визуализации изображений. Вы можете изменять переменные состояния, управляющие переносом пикселей и управлять тем, как растеризуются изображения и битовые карты. Если команды, устанавливающие эти переменные, предшествуют определению изображения или битовой карты, реализация **OpenGL** может выполнить некоторые преобразования в момент создания списка и кэшировать результаты.

Помните, что списки отображения имеют и ряд недостатков. Очень маленькие списки могут работать недостаточно быстро, так как существует некоторая потеря времени при исполнении списка. Другим недостатком является неизменяемость данных внутри списка отображения. Для оптимизации быстродействия содержимое списков отображения **OpenGL** не может быть изменено или считано. Если приложению требуется управление данными отдельно от списков отображения **(**например, для отображения данных в реальном времени**)**, может потребоваться много дополнительной памяти.

7.4 Создание и исполнение списка отображения

Как вы уже видели, для начала и конца описания списка применяются команды **glNewList()** и **glEndList()**. Далее список может быть исполнен передачей его индекса в команду **glCallList()**. В примере **7-2** список создается в функции **init()**. Этот список отображения содержит команды **OpenGL** для рисования красного треугольника. Затем в функции **display()** список исполняется **10** раз. Кроме того, там же рисуется линия в непосредственном режиме. Имейте в виду, что для списка отображения выделяется память, в которой хранятся его команды и все необходимые переменные.

Пример **7-2.** Использование списка отображения: файл **list.cpp**

```
#include <glut.h>

GLuint listName;

void init(void)
{
   listName=glGenLists(1);
   glNewList(listName,GL_COMPILE);
      glColor3f(1.0,0.0,0.0);
      glBegin(GL_TRIANGLES);
         glVertex2f(0.0,0.0);
         glVertex2f(1.0,0.0);
         glVertex2f(0.0,1.0);
      glEnd();
      glTranslatef(1.5,0.0,0.0);
   glEndList();
   glShadeModel(GL_FLAT);
}

void drawLine()
{
   glBegin(GL_LINES);
      glVertex2f(0.0,0.5);
      glVertex2f(15.0,0.5);
   glEnd();
```

```
}

void display(void)
{
   GLuint i;
   glClear(GL_COLOR_BUFFER_BIT);
   glColor3f(0.0,1.0,0.0);
   for(i=0;i<=10;i++)
      glCallList(listName);

   //Эта линия НЕ будет зеленой, так как текущий цвет был изменен внутри ЛИСТА
   drawLine();
   glFlush();
}

void reshape(int w, int h)
{
   glViewport(0,0,w,h);
   glMatrixMode(GL_PROJECTION);
   glLoadIdentity();
   if (w<=h)
      gluOrtho2D(0.0,2.0,-0.5*(GLfloat)h/(GLfloat)w,1.5*(GLfloat)h/(GLfloat)w);
   else
      gluOrtho2D(0.0,2.0*(GLfloat)w/(GLfloat)h,-0.5,1.5);
   glMatrixMode(GL_MODELVIEW);
   glLoadIdentity();
}

int main(int argc, char** argv)
{
   glutInit(&argc,argv);
   glutInitDisplayMode(GLUT_SINGLE|GLUT_RGB);
   glutInitWindowSize(650,50);
   glutCreateWindow("Using a Display List");
   init();
   glutDisplayFunc(display);
   glutReshapeFunc(reshape);
   glutMainLoop();
   return 0;
}
```

Команда **glTranslatef()** в списке отображения сдвигает позицию следующего рисуемого объекта. Без нее два вызова списка к исполнению рисовали бы треугольники в одном и том же месте один поверх другого. Функция **drawLine()**, вызываемая в непосредственном режиме затронута влиянием **10**-го вызова **glTranslate()**. Таким образом, если вы вызываете команды преобразований внутри списка отображения, не забудьте учесть эффект от влияния этих команд на более поздние вызовы в вашей программе.

В каждый момент времени создается только один список отображения. Другими словами, за командой **glNewList()** должна следовать команда **glEndList()**, завершающая создание списка, и только после этого можно начинать создавать новый список. Как вы можете подозревать, вызов **glEndList()** без предшествующего вызова **glNewList()** сгенерирует ошибку **GL_INVALID_OPERATION**.

7.4.1 Именование и создание списка отображения

Каждый список отображения идентифицируется целым индексом. При создании списка следует быть осторожным, дабы случайно не задействовать уже используемый индекс, поскольку это приведет к перезаписи имеющегося списка. Во избежание этой проблемы используйте команду **glGenLists()** для генерирования одного или более незадействованных индексов.

```
GLuint glGenLists (GLsizei range);
```

Выделяет непрерывный диапазон незадействованных индексов, количеством ***range***. Возвращаемое значение – это первый индекс в этом блоке пустых индексов. Возвращаемые индексы помечаются как пустые и используемые, таким образом последующие вызовы **glGenLists()** не возвратят именно эти номера до тех пор, пока они не будут стерты. Если ***range*** равен **0** или блок свободных индексов нужного размера отсутствует, **glGenLists()** вернет **0**.

В следующем примере запрашивается единственный индекс, и, если он получен, создается новый список отображения.

```
listIndex=glGenLists(1);
if (listIndex!=0)
{
   glNewList(listIndex, GL_COMPILE);
      ...
   glEndList();
}
```

Замечание: 0 не является допустимым индексом.

```
void glNewList (GLuint list, GLenum mode);
```

Маркирует начало списка отображения. Вызываемые далее (до вызова **glEndList()**) команды **OpenGL** (кроме нескольких, запрещенных к использованию в списках) сохраняются в списке отображения. (Эти запрещенные команды, если они встречаются в описании списка, исполняются немедленно в непосредственном режиме.) Аргумент ***list*** – это ненулевое положительное целое число, уникально идентифицирующее список отображения. Возможными значениями для ***mode*** являются **GL_COMPILE** и **GL_COMPILE_AND_EXECUTE**. Используйте **GL_COMPILE**, если вы не хотите, чтобы команды **OpenGL** исполнялись немедленно при помещении в список. Если вы хотите, чтобы при создании списка команды, помещаемые в него, выполнились еще и в непосредственном режиме – используйте **GL_COMPILE_AND_EXECUTE**.

```
void glEndList (void);
```

Маркирует конец списка отображения.

При создании списка отображения, он сохраняется вместе с текущим контекстом **OpenGL**. То есть, при уничтожении контекста список также уничтожается. Некоторые оконные системы позволяют разделение списков несколькими контекстами. В этом случае, список уничтожается вместе с последним контекстом из *разделяющей группы*.

7.4.2 Что сохраняется в списке отображения

Когда вы строите список, в нем сохраняются только значения выражений. Если впоследствии элементы массива изменятся – величины в списке останутся нетронутыми. В следующем фрагменте кода список отображения содержит команды для установки текущего **RGBA** цвета в черный **(0.0, 0.0, 0.0)**. последующее изменение элементов массива ***color_vector*** на красный **(1.0, 0.0, 0.0)** не затрагивает список отображения, поскольку в нем сохранились величины, действовавшие на момент его создания.

```
GLfloat color_vector[3]={0.0, 0.0, 0.0};

glNewList(1, GL_COMPILE);
   glColor3f(color_vector);
glEndList();
color_vector[0]=1.0;
```

Не все команды **OpenGL** могут быть сохранены и исполнены из списка отображения. Например, команды, устанавливающие клиентское состояние, и команды, запрашивающие значения переменных состояния, в списках отображения не сохраняются. (Многие из несохраняемых в списках команд могут быть идентифицированы по тому признаку, что они возвращают значение непосредственно или через аргумент указатель.) Если эти команды вызываются при создании списка отображения, они выполняются немедленно.

Далее перечислены команды **OpenGL**, которые не сохраняются в списках отображения. (Заметьте также, что команда **glNewList()** сгенерирует ошибку, если будет вызвана во время создания списка отображения.)

glColorPointer()	**glFlush()**	**glNormalPointer()**
glDeleteLists()	**glGenLists()**	**glPixelStore()**
glDisableClientState()	**glGet*()**	**glReadPixels()**
glEdgeFlagPointer()	**glIndexPointer()**	**glRenderMode()**
glEnableClientState()	**glInterleavedArrays()**	**glSelectBuffer()**
glFeedbackBuffer()	**glIsEnabled()**	**glTexCoordPointer()**
glFinish()	**glIsList()**	**glVertexPointer()**

Чтобы более ясно понять, почему эти команды не сохраняются в списках, помните, что при использовании **OpenGL** по сети, клиент и сервер могут находиться на разных машинах. После создания списка отображения, он находится на сервере, и сервер не может полагаться на клиента в вопросах касающихся любой информации, связанной со списком. Если бы команды опроса, такие как **glGet*()** или **glIs*()** были допустимы в списках отображения, вызывающая программа была бы крайне удивлена данным, поступающим по сети в случайные моменты времени. Без разбора отосланного списка, программа вообще не будет знать, что делать с этими данными. Таким образом, любые команды, возвращающие значения, не могут быть сохранены в списке отображения. Кроме того, в списках не сохраняются команды, изменяющие состояние клиента, такие как **glPixelStore()** или **glSelectBuffer()**, и команды, определяющие вершинные массивы.

Действие некоторых команд **OpenGL** зависит от состояния клиента. Например, команды описания вершинных массивов (такие как **glVertexPointer()**, **glColorPointer()** и **glInterleavedArrays()**) устанавливают указатели в пространстве клиента и не могут сохраняться в списках отображения. Команды **glArrayElement()**, **glDrawArrays()** и **glDrawElements()** посылают данные состоянию серверу для построения примитивов из вершинных массивов, так что эти операции могут быть сохранены в списках отображения. Данные вершинных массивов, сохраненные в списке отображения извлекаются путем разрешения указателей, а не сохранением самого указателя. Таким образом, последующие изменения в данных вершинных массивов не повлияют на описание примитива в списке отображения.

Кроме того, любые команды, опирающиеся в своей работе на режимы сохранения пикселей, используют те режимы, которые были в действии на момент помещения этих команд в список. Другие команды, опирающиеся на состояние клиента – такие как **glFinish()** и **glFlush()** – не могут быть помещены в список отображения, так как они зависят от состояния клиента в момент их исполнения.

7.4.3 Исполнение списка отображения

После того, как вы создали список отображения, вы можете исполнять его, вызвав команду **glCallList()**. Естественно, вы можете исполнять один и тот же список много раз, и, как вы уже видели, вы можете смешивать в программе код для исполнения списков отображения и команды непосредственного режима.

```
void glCallList (GLuint list);
```

Эта команда исполняет список отображения, заданный аргументом ***list***. Команды в списке исполняются в том порядке, в котором они сохранялись, таким же образом, как если бы они выполнялись вне списка. Если список с индексом ***list*** не определен, не происходит ничего.

Вы можете вызывать **glCallList()** из любого места программы до тех пор, пока активным является контекст **OpenGL**, способный получить доступ к нужному списку отображения (то есть активным должен быть контекст, в котором список создавался, или контекст, входящий в ту же разделяющую группу). Список отображения может быть создан в одной функции, а и вызван к исполнению совершенно из другой, поскольку в любом случае он идентифицируется по своему индексу. Не существует методов для сохранения содержимого списка отображения в файл данных, а также методов для загрузки списков из файла. Таким образом, списки отображения разработаны для временного использования.

7.4.4 Иерархические списки отображения

Вы можете создать иерархический список отображения, то есть список отображения, который исполняет другой список отображения, вызывая **glCallList()** между парой **glNewList()** и **glEndList()**. Иерархический список отображения полезен для рисования объектов, состоящих из компонентов, особенно если эти компоненты используются более одного раза. Например, следующий список отображения для велосипеда исполняет списки отображения для рисования его частей:

```
glNewList(listIndex, GL_COMPILE);
   glCallList(handlebars);
   glCallList(frame);
   glTranslatef(1.0,0.0,0.0);
   glCallList(wheel);
   glTranslatef(3.0,0.0,0.0);
   glCallList(wheel);
glEndList();
```

Во избежание бесконечной рекурсии существует ограничение на уровень вложенности списков отображение; максимум уровня вложенности не может быть меньше **64**, но может быть больше в зависимости от реализации. Чтобы выяснить максимальный уровень вложенности в вашей реализации **OpenGL**, используйте:

```
glGetIntegerv(GL_MAX_LIST_NESTING, GLint *data);
```

OpenGL позволяет вам создавать списки, исполняющие другие списки, которые еще не созданы. Если один список пытается исполнить другой, еще не созданный, ничего не происходит.

Вы можете использовать иерархический список отображения для эмуляции редактируемого списка, оборачивая его вокруг нескольких списков более низкого уровня. Например, чтобы поместить полигон в список отображения и оставить за собой возможность редактировать его вершины, можно использовать код из примера **7-3**.

Пример **7-3**. Иерархический список отображения

```
glNewList(1, GL_COMPILE);
   glVertex3fv(v1);
glEndList();

glNewList(2, GL_COMPILE);
```

```
   glVertex3fv(v2);
glEndList();

glNewList(3, GL_COMPILE);
   glVertex3fv(v3);
glEndList();

glNewList(4, GL_COMPILE);
   glBegin(GL_POLYGON);
      glCallList(1);
      glCallList(2);
      glCallList(3);
   glEnd();
glEndList();
```

Чтобы отобразить полигон, вызовите список отображения номер **4**. Для редактирования вершины, вам нужно лишь заново создать единственный список отображения соответствующий этой вершине. Поскольку индекс однозначно идентифицирует список, создание нового списка, с индексом существующего автоматически удалит старый. Имейте в виду, что такая техника не гарантирует оптимального использования памяти или максимального быстродействия, но она приемлема и полезна в некоторых случаях.

7.4.5 Управление индексами списков отображения

До сих пор мы рекомендовали использовать **glGenLists()** для получения неиспользуемых индексов. Если вы хотите избежать использования **glGenLists()**, обязательно используйте **glIsList()** для выяснения того, находится ли конкретный индекс в использовании.

```
GLboolean glIsList (GLuint list);
```

Возвращает **GL_TRUE** если ***list*** уже используется в качестве идентификатора существующего списка отображения и **GL_FALSE** в противном случае.

Вы можете удалить конкретный список отображения или их непрерывный диапазон командой **glDeleteLists()**. После применения **glDeleteLists()**, индексы удаленных списков становятся доступны для дальнейшего использования.

```
void glDeleteLists (GLuint list, GLsizei range);
```

Удаляет последовательные **(**по индексам**)** списки отображения количеством ***range***, начиная с индекса ***list***. Попытки удалить несуществующие списки отображения игнорируются.

7.5 Исполнение нескольких списков отображения

OpenGL предоставляет эффективный механизм для последовательного исполнения нескольких списков отображения. Этот механизм требует, чтобы вы поместили индексы нужных списков в массив и вызвали **glCallLists()**. Этот механизм обычно используется тогда, когда индексы списков отображения имеют осмысленные значения. Например, если вы создаете шрифт, индекс каждого списка может соответствовать **ASCII** коду символа в шрифте. Чтобы создать несколько таких шрифтов, вам необходимо заранее установить разные начальные значения для индексов списков отображения каждого шрифта. Вы можете задать начальный индекс, вызвав **glListBase()** до вызова **glCallLists()**.

```
 void glListBase (GLuint base);
```

Задает смещение, которое добавляется к индексам списков в **glCallLists()** для получения финальных индексов. По умолчанию базовое смещение равно **0**. База не оказывает влияние ни на команду **glNewList()**, ни на **glCallList()**, исполняющую единственный список отображения.

```
void glCallLists (GLsizei n, GLenum type, const GLvoid *lists);
```

Исполняет **n**списков отображения. Индексы исполняемых списков отображения вычисляются путем сложения текущей базы списков **(**заданной с помощью **glListBase())** с знаковыми целыми значениями в массиве, на который указывает ***lists***.

Аргумент ***type*** задаеттипвеличинв ***lists*** иможет принимать значения **GL_BYTE**, **GL_UNSIGNED_BYTE**, **GL_SHORT**, **GL_UNSIGNED_SHORT**, **GL_INT**, **GL_UNSIGNED_INT** или **GL_FLOAT**, указывая на то, что массив ***lists*** должен интерпретироваться как массив байт, беззнаковых байт, коротких целых, беззнаковых коротких целых, целых, беззнаковых целых или чисел с плавающей точкой, соответственно. ***Type*** также может принимать значения **GL_2_BYTES**, **GL_3_BYTES**или **GL_4_BYTES**; в этом случае из массива считываются по **2**, **3** или **4** байта, сдвигаются и складываются вместе, байт за байтом, для вычисления смещения списка. Используется следующий алгоритм **(*byte[0]*–** это начало последовательности байтов**):**

```
/* b=2, 3 или 4; байты в массиве нумеруются 0, 1, 2, 3 и так далее */
offset=0;
for(i=0;i<b;i++)
{
   offset=offset << 8;
   offset+=byte[i];
}
index=offset+listbase;
```

Для многобайтовых данных первый, извлекаемый из массива байт, становится старшим.

В качестве примера использования нескольких списков отображения, рассмотрите фрагмент программы **7-4**, взятый из полной программы-примера **7-5**. Эта программа рисует символы линейным шрифтом **(**набор букв был создан из сегментов линий**).** Функция **initStrokedFont()** устанавливает индекс для каждого списка, чтобы индексы были равны значениям **ASCII**-кодов соответствующих символов.

Пример **7-4.** Определение нескольких списков отображения

```
void initStrokedFont(void)
{
   GLuint base;

   base=glGenLists(128);
   glListBase(base);
   glNewList(base+'A',GL_COMPILE);
      drawLetter(Adata);
   glEndList();

   glNewList(base+'E',GL_COMPILE);
      drawLetter(Edata);
   glEndList();

   glNewList(base+'P',GL_COMPILE);
      drawLetter(Pdata);
   glEndList();

   glNewList(base+'R',GL_COMPILE);
      drawLetter(Rdata);
   glEndList();

   glNewList(base+'S',GL_COMPILE);
```

```
        drawLetter(Sdata);
     glEndList();

     //Символ пробела
     glNewList(base+' ',GL_COMPILE);
        glTranslatef(8.0,0.0,0.0);
     glEndList();
  }
```

Команда **glGenLists()** выделяет **128** непрерывных индексов для списков отображения. Первый из этих индексов становится базой списков. Для каждой буквы создается свой список; индекс каждого списка представляет собой сумму базы и **ASCII**-кода символа. В этом примере создается только несколько символов и символ пробела.

После создания списков, может быть вызвана команда **glCallLists()** для их исполнения. Например, вы можете передать текстовую строку функции **printStrokedString()**:

```
  void printStrokedString(GLbyte *s)
  {
     GLint len=strlen(s);
     glCallLists(len, GL_BYTE, s);
  }
```

Значение **ASCII** для каждого символа в строке используется в качестве смещения в индексах списков отображения. Для определения финального индекса исполняемых списков списочная база складывается с **ASCII**-кодом каждого символа. Вывод, производимый примером **7-5**, показан на рисунке **7-1**.

Рисунок **7-1.** Линейный шрифт с символами **A, E, P, R** и **S**

Пример **7-5.** Несколько списков отображения, определяющие линейный шрифт: файл **stroke.cpp**

```
#include <glut.h>
#include <string.h>

#define PT 1
#define STROKE 2
#define END 3

typedef struct charpoint
{
   GLfloat x,y;
   int type;
} CP;

CP
Adata[]={{0,0,PT},{0,9,PT},{1,10,PT},{4,10,PT},{5,9,PT},{5,0,STROKE},{0,5,PT},{5,5,END}};
CP Edata[]={{5,0,PT},{0,0,PT},{0,10,PT},{5,10,STROKE},{0,5,PT},{4,5,END}};
CP Pdata[]={{0,0,PT},{0,10,PT},{4,10,PT},{5,9,PT},{5,6,PT},{4,5,PT},{0,5,END}};
CP Rdata[]={{0,0,PT},{0,10,PT},{4,10,PT},{5,9,PT},{5,6,PT},{4,5,PT},{0,5,STROKE},{3,5,PT},
{5,0,END}};
CP
Sdata[]={{0,1,PT},{1,0,PT},{4,0,PT},{5,1,PT},{5,4,PT},{4,5,PT},{1,5,PT},{0,6,PT},{0,9,PT},{
1,10,PT},{4,10,PT},{5,9,END}};

//Интерпретируем инструкции из массива для буквы и
//визуализируем букву с помощью сегментов линий
void drawLetter(CP *l)
```

```
{
   glBegin(GL_LINE_STRIP);
   while(1)
   {
      switch(l->type)
      {
         case PT:
            glVertex2fv(&l->x);
            break;
         case STROKE:
            glVertex2fv(&l->x);
            glEnd();
            glBegin(GL_LINE_STRIP);
            break;
         case END:
            glVertex2fv(&l->x);
            glEnd();
            glTranslatef(8.0,0.0,0.0);
            return;
      }
      l++;
   }
}

//Создаем список отображения для каждого из 6 символов (5 букв + пробел)
void init()
{
   glLineWidth(2.0);
   base=glGenLists(128);
   glListBase(base);
   glNewList(base+'A',GL_COMPILE); drawLetter(Adata); glEndList();
   glNewList(base+'E',GL_COMPILE); drawLetter(Edata); glEndList();
   glNewList(base+'P',GL_COMPILE); drawLetter(Pdata); glEndList();
   glNewList(base+'R',GL_COMPILE); drawLetter(Rdata); glEndList();
   glNewList(base+'S',GL_COMPILE); drawLetter(Sdata); glEndList();
   glNewList(base+' ',GL_COMPILE); glTranslatef(8.0,0.0,0.0); glEndList();
}

char *test1="A SPARE SERAPE APPEARS AS";
char *test2="APES PREPARE RARE PEPPERS";

void printStrokedString(char *s)
{
   GLsizei len=strlen(s);
   glCallLists(len,GL_BYTE,(GLbyte *) s);
}

void display()
{
   glClear(GL_COLOR_BUFFER_BIT);
   glColor3f(0.0,1.0,0.0);
   glPushMatrix();
      glScalef(2.0,2.0,2.0);
      glTranslatef(10.0,30.0,0.0);
      printStrokedString(test1);
   glPopMatrix();
   glPushMatrix();
      glScalef(2.0,2.0,2.0);
      glTranslatef(10.0,13.0,0.0);
      printStrokedString(test2);
   glPopMatrix();
   glFlush();
}

void reshape(int w,int h)
{
   glViewport(0,0,w,h);
   glMatrixMode(GL_PROJECTION);
   glLoadIdentity();
   gluOrtho2D(0.0,440,0.0,120);
}

void keyboard(unsigned char key,int x,int y)
{
```

```
   switch(key)
   {
      case ' ':
         glutPostRedisplay();
         break;
      case 27:
         exit(0);
         break;
   }
}

int main(int argc, char** argv)
{
   glutInit(&argc,argv);
   glutInitDisplayMode(GLUT_SINGLE|GLUT_RGB);
   glutInitWindowSize(440,120);
   glutCreateWindow("Multiple Display Lists to Define a Stroked Font");
   init();
   glutDisplayFunc(display);
   glutReshapeFunc(reshape);
   glutKeyboardFunc(keyboard);
   glutMainLoop();
   return 0;
}
```

7.6 Управление переменными состояния с помощью списков отображения

Список отображения может содержать команды, изменяющие значения переменных состояния **OpenGL.** Величины изменяются при исполнении списка отображения, как если бы команды выполнялись в непосредственном режиме. Изменения сохраняются и после завершения исполнения списка. Как было видно в примере **7-2** и будет показано в примере **7-6,** изменения текущего цвета и текущей матрицы, сделанные в процессе исполнения списка отображения, остаются в действии и после того, как он был выполнен.

Пример **7-6.** Постоянство изменения состояния после исполнения списка отображения

```
   glNewList(listIndex, GL_COMPILE);
      glColor3f(1.0,0.0,0.0);
      glBegin(GL_POLYGON);
         glVertex2f(0.0,0.0);
         glVertex2f(1.0,0.0);
         glVertex2f(0.0,1.0);
      glEnd();
      glTranslatef(1.5,0.0,0.0);
   glEndList();
```

Таким образом, если вы выполните следующую последовательность кода, линию, нарисованная после того, как список отображения рисовал красным цветом и выполнил смещение на **(1.5, 0.0, 0.0),** будет красной:

```
   glCallList(listIndex);
   glBegin(GL_LINES);
      glVertex2f(2.0,-1.0);
      glVertex2f(1.0,0.0);
   glEnd();
```

Иногда вам нужно, чтобы изменения в состоянии сохранялись, но иногда вам необходимо сохранить значения переменных до исполнения списка отображения и затем восстановить их после него. Помните, что в списках вы не можете использовать команды **glGet*(),** так что понадобится иной способ запроса и сохранения значений переменных состояния.

Вы можете использовать **glPushAttrib()** для сохранения групп переменных состояния и **glPopAttrib()** для их последующего восстановления. Чтобы сохранять и восстанавливать текущую матрицу, используйте **glPushMatrix()** и **glPopMatrix()**. Эти команды могут быть вполне законно кэшированы в списках отображения. Чтобы восстановить переменные состояния в примере **7-6**, можно использовать код примера **7-7**.

Пример **7-7.** Восстановление переменных состояния внутри списка отображения

```
glNewList(listIndex, GL_COMPILE);
   glPushMatrix();
      glPushAttrib(GL_CURRENT_BIT);
      glColor3f(1.0,0.0,0.0);
      glBegin(GL_POLYGON);
         glVertex2f(0.0,0.0);
         glVertex2f(1.0,0.0);
         glVertex2f(0.0,1.0);
      glEnd();
      glTranslatef(1.5,0.0,0.0);
      glPopAttrib();
   glPopMatrix();
glEndList();
```

Если вы используете список отображения из примера **7-7**, код в примере **7-8** нарисует зеленую, неперенесенную линию. При использовании списка отображения из примера **7-6 (**которые не сохраняет и не восстанавливает состояние**)**, линия будет нарисована красным сдвинутой **10** раз на **(1.5, 0.0, 0.0)**.

Пример **7-8.** Список отображение может влиять или не влиять на **drawLine()**

```
void display(void)
{
   GLint i;
   glClear(GL_COLOR_BUFFER_BIT);
   glColor3f(0.0,1.0,0.0);   //устанавливаем текущий цвет в зеленый
   for(i=0;i<10;i++)
      glCallList(listIndex); //список исполняется 10 раз
   drawLine();
   glFlush();
}
```

7.6.1 Инкапсуляция изменений режима

Вы можете использовать списки отображения для организации и сохранения групп команд, изменяющих различные режимы и устанавливающих различные параметры. Когда вы хотите переключиться от одной группы установок к другой, использование списков отображения может быть эффективнее непосредственных вызовов команд, так как команды в списке могут быть кэшированы в формате, более подходящим для вашей графической системы.

Списки могут быть эффективнее, чем команды непосредственного режима, при переключении различных условий освещения, моделей освещения и свойств материала. Вы также можете использовать списки для рисунков шаблонов, параметров тумана и уравнений отсекающих плоскостей. В общем, исполнение списков отображения, как минимум, не медленнее, чем вызовы команд в непосредственном режиме, однако помните, что с исполнением списков также связаны некоторые временные затраты.

Пример **7-9** показывает, как использовать списки отображения для переключения между тремя различными шаблонами линий. Сначала вызывается **glGenLists()**, чтобы

выделить индекс для списка каждого рисунка шаблона, и создаются сами списки. Затем для переключения между разными рисунками шаблона используется **glCallList().**

Пример **7-9.** Списки отображения, используемые для смены режимов

```
GLuint offset;

offset=glGenLists(3);
glNewList(offset,GL_COMPILE);
   glDisable(GL_LINE_STIPPLE);
glEndList();

glNewList(offset+1,GL_COMPILE);
   glEnable(GL_LINE_STIPPLE);
   glLineStipple(1,0x0F0F);
glEndList();

glNewList(offset+2,GL_COMPILE);
   glEnable(GL_LINE_STIPPLE);
   glLineStipple(1,0x1111);
glEndList();

#define drawOneLine(x1, y1, x2, y2) glBegin(GL_LINES); glVertex2f((x1),(y1));
glVertex2f((x2),(y2)); glEnd();

glCallList(offset);
drawOneLine(50.0,125.0,350.0,125.0);
glCallList(offset+1);
drawOneLine(50.0,100.0,350.0,100.0);
glCallList(offset+2);
drawOneLine(50.0,75.0,350.0,75.0);
```

Глава 8. Отображение пикселей, битовых карт, шрифтов и изображений

До сих пор большинство дискуссий в данном пособии касалось визуализации и отображения геометрических данных – точек, линий и полигонов. Два других важных типов данных, которые могут отображаться **OpenGL** это:

- Битовые карты, обычно используемые для символов шрифта
- Изображения, которые могут быть отсканированы или построены путем вычислений.

И битовые карты и изображения имеют форму прямоугольных массивов пикселей. Одно из различий, существующее между ними, заключается в том, что битовая карта хранит по одному биту на каждый, входящий в нее пиксель, а в изображении на каждый пиксель хранится, как правило, несколько элементов данных (например, полные значения красного, зеленого, синего и альфа компонентов цвета). Кроме того, битовые карты работают как маски, если перекрывают изображение, а данные изображения просто перезаписывают информацию в буфере кадра или накладываются на нее.

В этой главе описывается, как рисовать пиксельные данные (битовые карты и изображения) из памяти в буфер кадра и как считывать из буфера кадра обратно. Также описывается, как копировать пиксельные данные из одной позиции в другие, то есть из одного буфера в другой, или из буфера в него же.

Замечание: OpenGL не поддерживает ни считывание пикселей из графических файлов, ни сохранение пикселей в них.

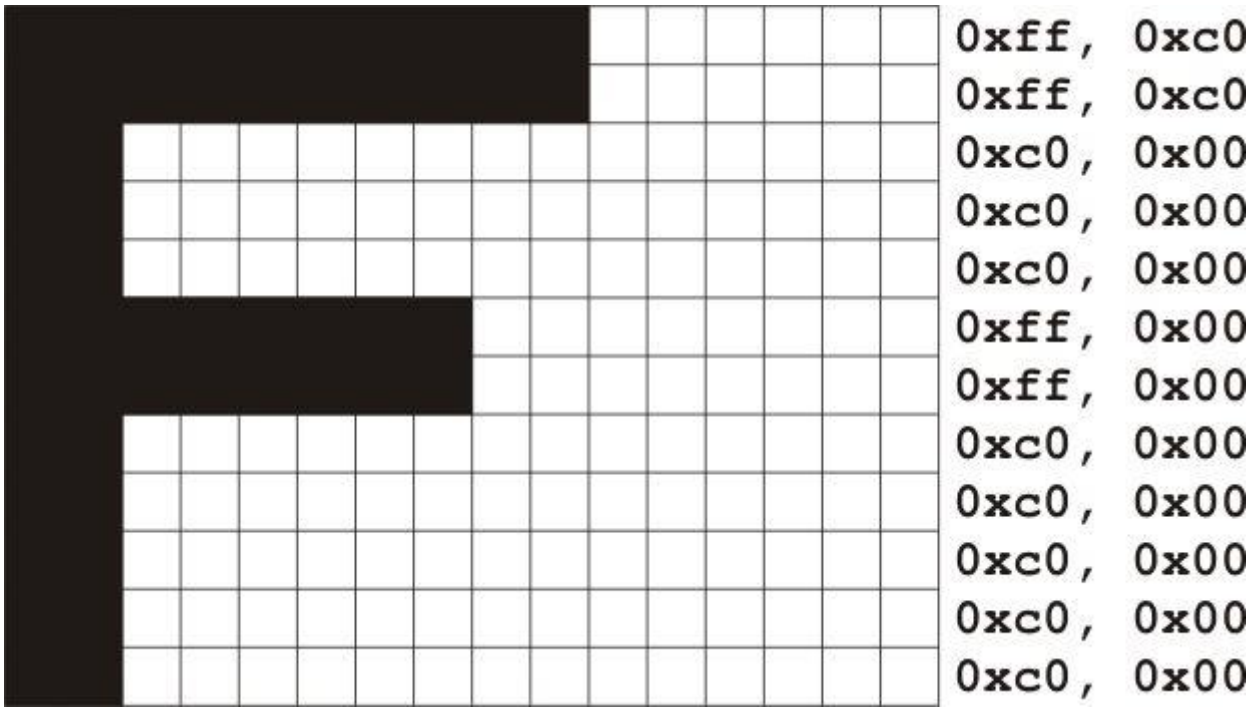
8.1 Битовые карты и шрифты

Битовая карта – это прямоугольный массив из нулей и единиц, который служит в качестве маски для прямоугольной части окна. Предположим, что вы рисуете битовую карту и текущий цвет – красный. Везде, где в битовой карте встречается **1**, соответствующий пиксель в буфере кадра замещается красным (или комбинируется с красным в зависимости от того, какие пофрагментные операции производятся). Там, где в битовой карте стоит **0**, фрагменты не генерируются и пиксели не изменяются. Битовые карты наиболее часто используются для рисования символов на экране.

OpenGL предоставляет только низкоуровневую поддержку для рисования строк символов и манипуляций со шрифтами. Команды **glRasterPos*()** и **glBitmap()** позиционируют и рисуют на экране одну битовую карту. Кроме того, с помощью механизма списков отображения вы можете использовать последовательности кодов символов для индексации соответствующих серий битовых карт, представляющих эти символы. Если для ваших манипуляций с битовыми картами, шрифтами и символами требуется иная поддержка, вам необходимо написать собственные функции.

Рассмотрим пример **8-1**, который трижды рисует на экране символ **F**. На рисунке **8-1** символ **F** показан в виде битовой карты и соответствующих ей данных.

Рисунок **8-1.** Битовая карта символа **F** и ее данные



Пример **8-1.** Рисование битовой карты символа: файл **drawf.cpp**

```
#include <glut.h>

GLubyte rasters[24]= { 0xC0,0x00,0xC0,0x00,0xC0,0x00,0xC0,0x00,
                       0xC0,0x00,0xFF,0x00,0xFF,0x00,0xC0,0x00,
                       0xC0,0x00,0xC0,0x00,0xFF,0xC0,0xFF,0xC0 };

void init(void)
{
   glPixelStorei(GL_UNPACK_ALIGNMENT,1);
   glClearColor(0.0,0.0,0.0,0.0);
}

void display(void)
{
   glClear(GL_COLOR_BUFFER_BIT);
   glColor3f(1.0,1.0,1.0);
```

```
   glRasterPos2i(20,20);
   glBitmap(10,12,0.0,0.0,11.0,0.0,rasters);
   glBitmap(10,12,0.0,0.0,11.0,0.0,rasters);
   glBitmap(10,12,0.0,0.0,11.0,0.0,rasters);
   glFlush();
}

void reshape(int w, int h)
{
   glViewport(0,0,(GLsizei) w, (GLsizei) h);
   glMatrixMode(GL_PROJECTION);
   glLoadIdentity();
   glOrtho(0,w,0,h,-1.0,1.0);
   glMatrixMode(GL_MODELVIEW);
}

void keyboard(unsigned char key,int x,int y)
{
   switch(key)
   {
      case 27:
      exit(0);
      break;
   }
}

int main(int argc, char **argv)
{
   glutInit(&argc,argv);
   glutInitDisplayMode(GLUT_SINGLE|GLUT_RGB);
   glutInitWindowSize(300,100);
   glutCreateWindow("Drawing a Bitmapped Character");
   init();
   glutDisplayFunc(display);
   glutReshapeFunc(reshape);
   glutKeyboardFunc(keyboard);
   glutMainLoop();
   return 0;
}
```

На рисунке **8-1** обратите внимание, что видимая часть символа **F**занимает **10** бит в ширину. Битовые карты всегда хранятся кусками, ширина которых кратна **8**, но реальная ширина битовой карты не обязана быть кратной **8**. Биты, задающие битовую карту, выводятся, начиная с левого нижнего угла: сначала рисуется нижний ряд, затем ряд над ним и так далее. Как вы можете видеть в коде, битовая карта хранится в памяти именно в этом порядке – массив начинается с чисел **0xc0**, **0x00**, **0xc0**, **0x00**, задающих два нижних ряда символа **F** и заканчивается числами **0xff**, **0xc0**, **0xff**, **0xc0**, задающими два верхних ряда.

В приведенном примере особый интерес представляют команды **glRasterPos2i()** и **glBitmap()**; они подробно рассматриваются в следующих разделах. Пока мы проигнорируем команду **glPixelStorei()**; она указывает на то, как данные битовой карты хранятся в памяти компьютера.

8.1.1 Текущая позиция растра

Текущая позиция растра – это некоторое положение на экране, где будет нарисована следующая битовая карта (или изображение). В примере с символом **F** текущая позиция растра была установлена командой **glRasterPos*()** с координатами **(20, 20)** в качестве параметров. Это то место, где был помещен нижний левый угол символа **F**:

```
glRasterPos2i(20, 20);
```

```
void glRasterPos[234]{sifd} (TYPE x, TYPE y, TYPE z, TYPE w);
void glRasterPos[234]{sifd}v (TYPE *coords);
```

Устанавливает текущую позицию растра. Аргументы ***x***, ***y***, ***z*** и ***w*** задают координаты позиции растра. Если используется векторная форма функции, массив ***coords*** содержит координаты позиции растра. Если используется версия **glRasterPos2*()**, то ***z***принимается равным **0**, а ***w*– 1**; похожим же образом, если используется **glRasterPos3*()**, то ***w*** принимается равным **1**.

Координаты позиции растра преобразуются в экранные координаты точно таким же образом, как и координаты, заданные командой **glVertex*()** (то есть они преобразуются с помощью видовой и проекционной матриц). После всех преобразований, текущие координаты растра либо определяют некоторую видимую на экране точку, либо отсекаются, если находятся вне объема видимости. Если преобразованная точка отсечена, текущая позиция растра считается незаконной.

Замечание: Если вы хотите задать позицию растра в экранных координатах, вам следует убедиться в том, что вы задали видовую и проекционную матрицу для простого 2D отображения. Для этого подойдет последовательность команд вроде следующей (*width* и *height* представляют собой размеры порта просмотра):

```
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
gluOrtho2D(0.0, (GLfloat) width, 0.0, (GLfloat) height);
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
```

Чтобы получить текущую позицию растра, вы можете использовать команду **glGetFloatv()** с **GL_CURRENT_RASTER_POSITION** в качестве первого аргумента. Второй аргумент должен быть указателем на массив, способный сохранить четверку (***x***, ***y***, ***z***, ***w***) в виде чисел с плавающей точкой. Если вы хотите выяснить, является ли текущая позиция растра легальной (допустимой), вызывайте **glGetBooleanv()** с константой **GL_CURRENT_RASTER_POSITION_VALID** в качестве первого аргумента.

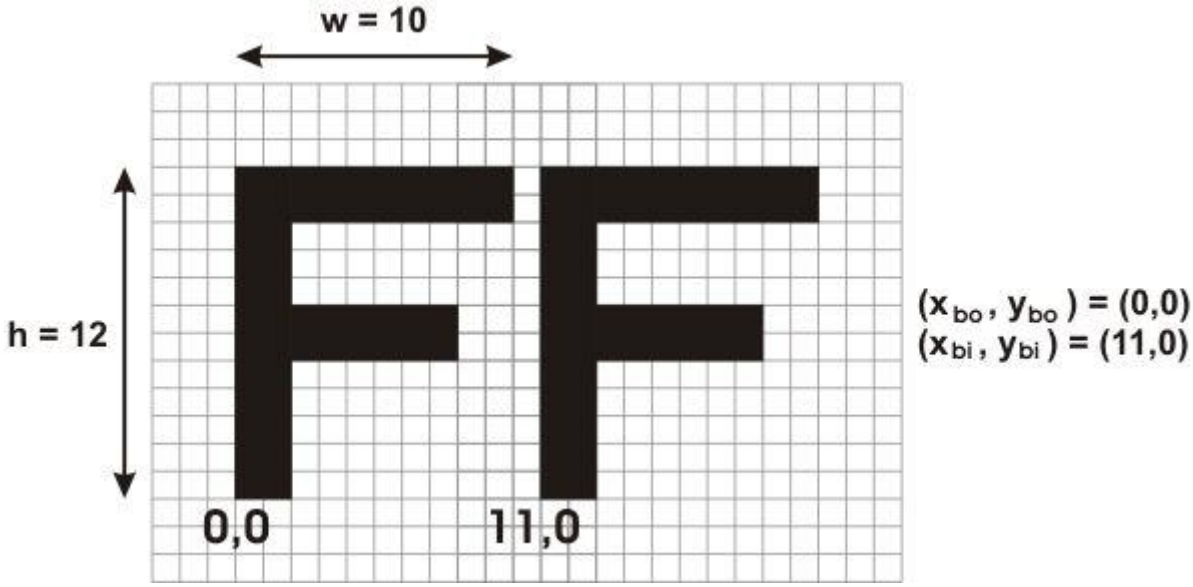
8.1.2 Отображение битовой карты

Как только вы установили желаемую позицию растра, вы можете использовать команду **glBitmap()**, чтобы вывести данные.

```
void glBitmap (GLsizei width,GLsizei height,GLfloat x_bo,GLfloat y_bo,GLfloat
x_bi,GLfloat y_bi,GLubyte *bitmap);
```

Рисует битовую карту, заданную аргументом ***bitmap***, который представляет собой указатель на данные карты. Начальная точка битовой карты помещается в текущую позицию растра. Если текущая позиция растра недопустима, ничего не рисуется, а позиция растра остается недопустимой. Аргументы ***width*** и ***height*** индицируют ширину и высоту битовой карты в пикселях. Ширина должна быть кратной **8**-ми, поскольку данные хранятся в беззнаковых символах по **8** бит каждый. (В примере с символом **F** не имеет значения, присутствует ли в данных за границами реальной карты какой-либо мусор, поскольку **glBitmap()** вызывается с аргументом ***width*** равным **10** и, как следствие, визуализируются только **10** бит в ряду.) Используйте x_{bo} и y_{bo}, чтобы задать начальную точку битовой карты, которая будет помещена в текущую позицию растра (положительные величины сдвигают начальную точку вверх и вправо от текущей позиции растра, отрицательные вниз и влево). Аргументы x_{bi} и y_{bi} задают смещение по ***x*** и ***y***, которое добавляется к текущей позиции растра после растеризации битовой карты (рисунок **8-2**).

Рисунок **8-2**. Битовая карта и ассоциированные с ней параметры



Свободное размещение начальной точки карты позволяет символам спускаться ниже начальной точки **(**обычно для символов**,** частично располагающихся ниже основной линии текста**,** например**, g, j**или **y)** или левее ее **(**например**,** для шрифтов**,** скошенных влево**,** и специальных символов**).**

После того**,** как битовая карта нарисована**,** текущая позиция растра сдвигается на x_{bi} и y_{bi} по ***x*** и ***y*** осям соответственно**. (**Если вы просто хотите сдвинуть позицию растра**,** ничего не рисуя**,** вызовите команду **glBitmap()** с аргументом ***bitmap*** установленным в **NULL** и параметрами ***width*** и ***height*** равными **0.)** Для стандартных латинских шрифтов y_{bi} обычно равно **0**, а x_{bi} **--** положительное число**,** так как последующие символы рисуются слева направо**.** Для Иврита**,** символы которого пишутся справа налево**,** x_{bi} обычно должно быть отрицательным числом**.** Для шрифта**,** символы которого выводятся вертикально в столбик**,** нужно использовать x_{bi} равное **0**, а y_{bi} отличное от **0.** На рисунке **8-2,** каждый раз при рисовании **F** текущая позиция растра сдвигается на **11** пикселей**,** оставляя между соседними символами пространство в **1** пиксель**.**

Поскольку x_{bo}, y_{bo}, x_{bi} и y_{bi} представляют собой числа с плавающей точкой**,** символы не должны занимать целое число пикселей**.** Реальные символы рисуются четко по пиксельным границам**,** однако текущая позиция растра хранится в виде дробного числа**,** благодаря чему каждый символ рисуется максимально близко к тому месту**,** где он должен находиться**.** Например**,** если в примере с символом **F** сделать x_{bi} равным **11.5** вместо **12,** и нарисовать больше символов**,** пространство между буквами было бы то **1**, то **2** пикселя**,** давая наилучшее к запрошенному пространству в **1.5** пикселя**.**

Замечание: Вы не можете поворачивать шрифты, созданные в виде битовых карт, поскольку битовые карты всегда рисуются выровненными по осям x и y буфера кадра. Кроме того, битовые карты нельзя масштабировать.

8.1.3 Выбор цвета для битовой карты

Вы уже знакомы с использованием команд **glColor*()** и **glIndex*()**, устанавливающих текущий цвет и текущий индекса для рисования геометрических примитивов**.** Те же самые команды используются для установки других переменных состояния**:** **GL_CURRENT_RASTER_COLOR** и **GL_CURRENT_RASTER_INDEX,** для рисования битовых карт**.** Переменная состояния**,** отвечающая за цвет растра устанавливается равной текущему цвету в момент вызова **glRasterPos*()**. Это может сыграть злую шутку**.** Как вы думаете**,** каков будет цвет битовой карты после выполнения следующего кода**?**

```
glColor3f(1.0,1.0,1.0);   /*белый*/
glRasterPos3fv(position);
glColor3f(1.0,0.0,0.0);   /*красный*/
glBitmap(...);
```

Вы удивлены, что битовая карта будет белой? Переменная **GL_CURRENT_RASTER_COLOR** устанавливается в белый цвет в момент вызова **glRasterPos3fv()**. Второй вызов **glColor3f()** меняет значение переменной **GL_CURRENT_COLOR** для визуализации будущих геометрических примитивов, но цвет, используемый для визуализации битовых карт, остается неизменным.

Чтобы выяснить текущий цвет или индекс растра, вы можете использовать опросные команды **glGetFloatv()** или **glGetIntegerv()** с аргументами **GL_CURRENT_RASTER_COLOR** или **GL_CURRENT_RASTER_INDEX** в качестве первого аргумента.

8.1.4 Шрифты и списки отображения

Списки отображения обсуждались ранее. Однако, некоторые из команд управления списками имеют особое отношение к выводу строк символов. Во время чтения данного раздела имейте в виду, что идеи, представленные здесь, в равной степени применимы и к символам, рисуемым с помощью битовых карт, и к символам, сконструированным из геометрических примитивов (точек, линий и полигонов).

Шрифт обычно состоит из набора символов, где каждый символ имеет идентификационный номер (как правило, свой **ASCII** код) и метод начертания. В стандартном наборе символов **ASCII** заглавная буква **A** (в латинице) имеет номер **65**, **B – 66** и так далее. Строка **«DAB»** может быть представлена тремя индексами **68**, **65** и **66**. В простейшем случае список отображения номер **65** рисует **A**, номер **66 – B** и так далее. То есть для того, чтобы вывести строку из символов **68**, **65**, **66** просто вызовите к исполнению соответствующие списки отображения.

Вы можете использовать команду **glCallLists()** следующим образом:

```
void glCallLists(GLsizei n, GLenum type, const GLvoid *lists);
```

Первый аргумент, ***n***, индицирует количество выводимых символов, ***type*** обычно равен **GL_BYTE**, а ***lists*** – это массив кодов символов.

Поскольку многим приложениям требуется отображать символы разных размеров и из разных шрифтов, простейший случай нас не устраивает. Было бы неплохо использовать **65** для символа **A** независимо от того, какой шрифт является активным на данный момент. Вы могли бы в шрифте **1** закодировать **A**, **B** и **C** номерами **1065**, **1066** и **1067**, а в шрифте **2** – номерами **2065**, **2066** и **2067**, но тогда любые номера больше **256** уже не будут укладываться в **8**-ми битовый байт. Лучшее решение заключается в том, чтобы добавлять некоторое смещение к каждому вхождению в строку до того, как производится выбор списка отображения. В этом случае символы **A**, **B** и **C** будут представлены номерами **1065**, **1066** и **1067** в шрифте **1**, а в шрифте **2** они могут быть представлены числами **2065**, **2066** и **2067**. Тогда, чтобы нарисовать символы шрифтом **1**, нужно установить смещение равным **1000** и вызвать к исполнению списки отображения **65**, **66** и **67**. Чтобы нарисовать ту же строку шрифтом **2**, нужно установить смещение равным **2000** и вызвать к исполнению те же списки.

Для того, чтобы установить смещение, используйте команду **glListBase()**. Для предыдущих примеров она должна быть вызвана с **1000** и **2000** в качестве единственного аргумента. Теперь все, что вам нужно – это непрерывный диапазон неиспользуемых индексов списков отображения, который может быть получен с помощью команды **glGenLists()**:

```
GLuint glGenLists (GLsizei range);
```

Эта функция возвращает блок, состоящий из ***range*** идентификаторов списков отображения. Возвращенные списки маркируются как «используемые» даже несмотря на то, что они пусты, так что последующие вызовы **glGenLists()** никогда не возвращают те же номера списков (если только вы не удалили их перед этим). Таким образом, если вы используете **4** в качестве аргумента и **glGenLists()** возвращает **81**, вы можете использовать идентификаторы списков отображения **81**, **82**, **83** и **84** для ваших символов. Если **glGenLists()** не может найти блок свободных индексов нужного размера, она возвращает **0**. (Заметьте, что команда **glDeleteLists()** позволяет удалить все списки отображения, связанные со шрифтом, за один вызов.)

Большинство американских и европейских шрифтов включают небольшое число символов (менее **256**), так что довольно легко представить каждый символ уникальным кодом, который может быть сохранен в одном байте. Азиатские шрифты (и некоторые другие) могут содержать намного большие наборы символов, и однобайтовой кодировки может быть недостаточно. **OpenGL** позволяет обрабатывать строки, состоящие из **1-**, **2-**, **3-** и **4-**ех байтовых символов с помощью параметра ***type*** команды **glCallLists()**. Этот параметр может иметь любое из следующих значений:

GL_BYTE	**GL_UNSIGNED_BYTE**
GL_SHORT	**GL_UNSIGNED_SHORT**
GL_INT	**GL_UNSIGNED_INT**
GL_FLOAT	**GL_2_BYTES**
GL_3_BYTES	**GL_4_BYTES**

8.1.5 Описание и использование целого шрифта

Команда **glBitmap()** и механизм списков отображений, описанный в предыдущем разделе, позволяет легко определить растровый шрифт. В примере **8-2** задаются символы верхнего регистра **ASCII**шрифта. В этом примере все символы имеют одинаковую длину, но в общем случае это не всегда так. Как только шрифт определен, программа выводит сообщение **«THEQUICKBROWNFOXJUMPSOVERALAZYDOG.»** (рисунок **8-3**).

Код примера **8-2** похож на пример с символом **F** за исключением того, что здесь битовая карта каждого символа сохраняется в своем собственном списке отображения. Комбинирование идентификатора списка со смещением, возвращенным командой **glGenLists()** равно **ASCII** коду символа.

Рисунок **8-3**. Фраза, написанная растровым шрифтом

Пример **8-2**. Отображение полного шрифта: файл **font.cpp**

```
#include <glut.h>
#include <string.h>

GLubyte space[]=
{0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00};
GLubyte letters[][13]={
{0x00,0x00,0xc3,0xc3,0xc3,0xc3,0xff,0xc3,0xc3,0xc3,0x66,0x3c,0x18},
{0x00,0x00,0xfe,0xc7,0xc3,0xc3,0xc7,0xfe,0xc7,0xc3,0xc3,0xc7,0xfe},
{0x00,0x00,0x7e,0x7e,0xc0,0xc0,0xc0,0xc0,0xc0,0xc0,0xc0,0xe7,0x7e},
```

```
{0x00,0x00,0xfc,0xce,0xc7,0xc3,0xc3,0xc3,0xc3,0xc3,0xc7,0xce,0xfc},
{0x00,0x00,0xff,0xc0,0xc0,0xc0,0xc0,0xfc,0xc0,0xc0,0xc0,0xc0,0xff},
{0x00,0x00,0xc0,0xc0,0xc0,0xc0,0xc0,0xc0,0xfc,0xc0,0xc0,0xc0,0xff},
{0x00,0x00,0x7e,0x7e,0xc3,0xc3,0xcf,0xc0,0xc0,0xc0,0xc0,0xe7,0x7e},
{0x00,0x00,0xc3,0xc3,0xc3,0xc3,0xc3,0xff,0xc3,0xc3,0xc3,0xc3,0xc3},
{0x00,0x00,0x7e,0x18,0x18,0x18,0x18,0x18,0x18,0x18,0x18,0x18,0x7e},
{0x00,0x00,0x7c,0xee,0xc6,0x06,0x06,0x06,0x06,0x06,0x06,0x06,0x06},
{0x00,0x00,0xc3,0xc6,0xcc,0xd8,0xf0,0xe0,0xf0,0xd8,0xcc,0xc6,0xc3},
{0x00,0x00,0xff,0xc0,0xc0,0xc0,0xc0,0xc0,0xc0,0xc0,0xc0,0xc0,0xc0},
{0x00,0x00,0xc3,0xc3,0xc3,0xc3,0xc3,0xc3,0xdb,0xff,0xff,0xe7,0xc3},
{0x00,0x00,0xc7,0xc7,0xcf,0xcf,0xdf,0xdb,0xfb,0xf3,0xf3,0xe3,0xe3},
{0x00,0x00,0x7e,0xe7,0xc3,0xc3,0xc3,0xc3,0xc3,0xc3,0xc3,0xe7,0x7e},
{0x00,0x00,0xc0,0xc0,0xc0,0xc0,0xc0,0xfe,0xc7,0xc3,0xc3,0xc7,0xfe},
{0x00,0x00,0x3f,0x6e,0xdf,0xdb,0xc3,0xc3,0xc3,0xc3,0xc3,0x66,0x3c},
{0x00,0x00,0xc3,0xc6,0xcc,0xd8,0xf0,0xfe,0xc7,0xc3,0xc3,0xc7,0xfe},
{0x00,0x00,0x7e,0xe7,0x03,0x03,0x07,0x7e,0xe0,0xc0,0xc0,0xe7,0x7e},
{0x00,0x00,0x18,0x18,0x18,0x18,0x18,0x18,0x18,0x18,0x18,0x18,0xff},
{0x00,0x00,0x7e,0xe7,0xc3,0xc3,0xc3,0xc3,0xc3,0xc3,0xc3,0xc3,0xc3},
{0x00,0x00,0x18,0x3c,0x3c,0x66,0x66,0xc3,0xc3,0xc3,0xc3,0xc3,0xc3},
{0x00,0x00,0xc3,0xe7,0xff,0xff,0xdb,0xdb,0xc3,0xc3,0xc3,0xc3,0xc3},
{0x00,0x00,0xc3,0x66,0x66,0x3c,0x3c,0x18,0x3c,0x3c,0x66,0x66,0xc3},
{0x00,0x00,0x18,0x18,0x18,0x18,0x18,0x18,0x3c,0x3c,0x66,0x66,0xc3},
{0x00,0x00,0xff,0xc0,0xc0,0x60,0x30,0x7e,0x0c,0x06,0x03,0x03,0xff}
};
GLuint fontOffset;

void makeRasterFont()
{
   GLuint i,j;

   glPixelStorei(GL_UNPACK_ALIGNMENT,1);
   fontOffset=glGenLists(128);
   for(i=0,j='A';i<26;i++,j++)
   {
      glNewList(fontOffset+j,GL_COMPILE);
         glBitmap(8,13,0.0,2.0,10.0,0.0,letters[i]);
      glEndList();
   }
   glNewList(fontOffset+' ',GL_COMPILE);
      glBitmap(8,13,0.0,2.0,10.0,0.0,space);
   glEndList();
}

void init(void)
{
   glShadeModel(GL_FLAT);
   makeRasterFont();
}

void printString(char *s)
{
   glPushAttrib(GL_LIST_BIT);
   glListBase(fontOffset);
   glCallLists(strlen(s),GL_UNSIGNED_BYTE,(GLubyte*)s);
   glPopAttrib();
}

void display(void)
{
   GLfloat white[3]={1.0,1.0,1.0};

   glClear(GL_COLOR_BUFFER_BIT);
   glColor3fv(white);
   glRasterPos2i(20,60);
   printString("THE QUICK BROWN FOX JUMPS");
   glRasterPos2i(20,40);
   printString("OVER A LAZY DOG");
   glFlush();
}

void reshape(int w, int h)
{
   glViewport(0,0,(GLsizei) 300, (GLsizei) 100);
   glMatrixMode(GL_PROJECTION);
```

```
    glLoadIdentity();
    glOrtho(0,w,0,h,-1.0,1.0);
    glMatrixMode(GL_MODELVIEW);
}

void keyboard(unsigned char key,int x,int y)
{
    switch(key)
    {
       case 27:
          exit(0);
          break;
    }
}

int main(int argc, char **argv)
{
    glutInit(&argc,argv);
    glutInitDisplayMode(GLUT_SINGLE|GLUT_RGB);
    glutInitWindowSize(300,100);
    glutCreateWindow("Drawing a Complete Font");
    init();
    glutDisplayFunc(display);
    glutReshapeFunc(reshape);
    glutKeyboardFunc(keyboard);
    glutMainLoop();
    return 0;
}
```

8.2 Изображения

Изображение похоже на битовую карту, но вместо одного бита на каждый пиксель прямоугольной области экрана, изображение может содержать значительно больше информации. Например, изображение может содержать информацию о полном цвете **(R, G, B, A)** для каждого пикселя. Изображения могут быть получены из нескольких источников, таких как:

- Оцифрованные сканером фотографии
- Изображение, которое первоначально было сгенерировано на экране графической программой с использованием графической аппаратуры, а затем считано в память пиксель за пикселем
- Программа, которая генерирует изображение в памяти пиксель за пикселем.
- Изображения, о которых вы обычно думаете, как о картинках происходят из цветовых буферов. Однако вы можете считывать или записывать области пиксельных данных из или в буфер глубины или буфер трафарета.

Помимо того, что изображение может просто отображаться на экране, оно может быть использовано для текстурирования. В этом случае изображение накладывается на полигон, который затем визуализируется обычным образом.

8.2.1 Считывание, запись и копирование пиксельных данных

OpenGL предоставляет три основные команды для манипуляции данными изображения:

- **glReadPixels() –** считывает прямоугольный массив пикселей из буфера кадра в процессорную память.
- **glDrawPixels() –** записывает прямоугольный массив пикселей из данных, хранящихся в процессорной памяти, в буфер кадра в текущую позицию растра, заданную командой **glRasterPos*()**.
- **glCopyPixels() –** копирует прямоугольный массив пикселей из одной части буфера кадра в другую. Эта команда ведет себя так же как пара вызовов **glReadPixels()** и **glDrawPixels()**, но данные никогда не попадают в процессорную память.

Порядок обработки пиксельных данных для вышеозначенных команд показан на рисунке **8-4.**

Рисунок **8-4.** Упрощенная диаграмма потоков пиксельных данных

Рисунок **8-4** отображает основной поток пикселей в процессе обработки. Координаты **glRasterPos*()**, которые задают текущую позицию растра и используются командами **glDrawPixels()** и **glCopyPixels()**, преобразуются геометрическим конвейером. Растеризация и пофрагментные операции влияют и на **glDrawPixels()** и на **glCopyPixels()**. **(**Однако, когда рисуется или копируется прямоугольник пикселей, как правило, нет причин для использования тумана или текстуры**.)**

Сложности возникают из-за того, что в буфере кадра существует совершенно различная информация, существует множество способов хранения пиксельной информации в памяти компьютера, а также различные операции преобразования данных могут производиться во время операций чтения, записи и копирования. Эти возможности отражаются в большом числе различных режимов операций. Если все, что делает ваша программа – это копирование изображений на экран или временное считывание изображений в память с тем, чтобы они могли быть скопированы позднее, вы можете игнорировать большинство этих режимов. Однако если вы хотите, чтобы ваша программа модифицировала данные, пока они находятся в памяти – например, если у вас есть изображение в одном формате, а окно требует его в другом формате, или вы хотите сохранить изображение в файл для дальнейшего восстановления в другой сессии или на другом типе машины с совершенно иными графическими возможностями – вам следует понимать различные режимы работы.

Остальная часть данного раздела описывает основные команды в деталях. Следующие разделы обсуждают детали серий операций с изображениями, входящими в состав конвейера изображений: режимы хранения пикселей, операции переноса пикселей и операции по преобразованию пикселей.

8.2.1.1 Чтение пикселей из буфера кадра в память процессора

```
void glReadPixels (GLint x, GLint y, GLsizei width, GLsizei height, GLenum
format, GLenum type, GLvoid *pixels);
```

Считывает пиксельные данные из прямоугольника буфера кадра, чей левый нижний угол находится в точке с оконными координатами **(*x*, *y*)**, а ширина и высота равны ***width*** и ***height***, и сохраняет его в массиве, на который указывает аргумент ***pixels***. Аргумент ***format*** задает характер элементов пиксельных данных, которые должны быть считаны (величина индекса или величины **R, G, B** или **A** в соответствии с таблицей **8-1**), а ***type*** задает тип данных для каждого элемента (таблица **8-2**).

Если вы используете **glReadPixels()**, чтобы получить информацию о **RGBA** величинах или величинах цветового индекса, вам, возможно, понадобиться уточнить, к какому буферу вы пытаетесь получить доступ. Например, если у вас окно с двойной буферизацией, вам нужно указать считываете ли вы данные из переднего или заднего буферов. Для управления тем, какой буфер выступает в качестве источника для чтения данных, вызывайте команду **glReadBuffer()**.

Таблица **8-1.** Форматы пикселей для команд **glReadPixels()** и **glDrawPixels()**

Константа формата	Формат пикселей
GL_COLOR_INDEX	единственный цветовой индекс
GL_RGB	красный компонент цвета, затем зеленый и синий компоненты
GL_RGBA	красный компонент цвета, затем зеленый, синий и альфа компоненты
GL_BGR	синий компонент цвета, затем зеленый и красный компоненты
GL_BGRA	синий компонент цвета, затем зеленый, красный и альфа компоненты
GL_RED	только красный компонент цвета
GL_GREEN	только зеленый компонент цвета
GL_BLUE	только синий компонент цвета
GL_ALPHA	только альфа компонент цвета
GL_LUMINANCE	только светлота
GL_LUMINANCE_ALPHA	светлота, а затем альфа компонент
GL_STENCIL_INDEX	только индекс из буфера трафарета
GL_DEPTH_COMPONENT	только компонент глубины

Замечание: Форматы пикселей GL_BGR и GL_BGRA появились в OpenGL версии 1.2.

Таблица **8-2.** Типы данных для **glReadPixels()** и **glDrawPixels()**

Константа типа	Тип данных
GL_UNSIGNED_BYTE	**8**-ми битовое беззнаковое целое
GL_BYTE	**8**-ми битовое знаковое целое
GL_BITMAP	по одному биту на пиксель, биты хранятся в **8**-ми битовых целых (тот же формат, что и для **glBitmap()**)
GL_UNSIGNED_SHORT	**16**-ми битовое беззнаковое целое
GL_SHORT	**16**-ми битовое знаковое целое
GL_UNSIGNED_INT	**32**-ми битовое беззнаковое целое
GL_INT	**32**-ми битовое знаковое целое
GL_FLOAT	число с плавающей точкой одинарной точности
GL_UNSIGNED_BYTE_3_3_2	упакованный в **8**-ми битовое беззнаковое целое
GL_UNSIGNED_BYTE_2_3_3_REV	упакованный в **8**-ми битовое беззнаковое целое
GL_UNSIGNED_SHORT_5_6_5	упакованный в **16**-ми битовое беззнаковое целое
GL_UNSIGNED_SHORT_5_6_5_REV	упакованный в **16**-ми битовое беззнаковое целое
GL_UNSIGNED_SHORT_4_4_4_4	упакованный в **16**-ми битовое беззнаковое целое
GL_UNSIGNED_SHORT_4_4_4_4_REV	упакованный в **16**-ми битовое беззнаковое целое
GL_UNSIGNED_SHORT_5_5_5_1	упакованный в **16**-ми битовое беззнаковое целое
GL_UNSIGNED_SHORT_1_5_5_5_REV	упакованный в **16**-ми битовое беззнаковое целое

GL_UNSIGNED_INT_8_8_8_8	упакованный в **32-**ми битовое беззнаковое целое
GL_UNSIGNED_INT_8_8_8_8_REV	упакованный в **32-**ми битовое беззнаковое целое
GL_UNSIGNED_INT_10_10_10_2	упакованный в **32-**ми битовое беззнаковое целое
GL_UNSIGNED_INT_2_10_10_10_REV	упакованный в **32-**ми битовое беззнаковое целое

Замечание: Пиксельные форматы GL_*_REV особенно удобны для применения в операционных системах MicrosoftWindows.

Помните, что в зависимости от формата, могут считываться ил записываться от одного до четырех элементов. Например, если формат равен **GL_RGBA**, и вы производите считывание в **32-**х разрядные целые **(**то есть ***type*** равен **GL_UNSIGNED_INT** или **GL_INT)**, то считывание данных для одного пикселя потребует **16** байт пространства в памяти **(4** компонента *** 4** байта на компонент**).**

Каждый элемент изображения сохраняется в памяти, как показано в таблице **8-2.** Если элемент представляет собой длинную **(**точную**)** величину такую как значение красного, зеленого, синего или величину освещенности, каждая величина масштабируется имеющегося количества бит. Предположим, например, что изначально красный компонент был задан в виде числа с плавающей точкой между **0.0** и **1.0.** Если требуется упаковать его в беззнаковый байт, то сохранятся только **8** бит точности, даже если для красного компонента в буфере кадра было выделено больше бит. **GL_UNSIGNED_SHORT** и **GL_UNSIGNED_INT** дадут **16** бит и **32** бита точности соответственно. Знаковые версии **GL_BYTE, GL_SHORT**и **GL_INT** дают **7, 15** и **31** бит точности, поскольку отрицательные величины, как правило, не используются.

Если элемент является индексом **(**цветовым индексом или индексом из буфера трафарета, например**)** и тип не равен **GL_FLOAT,** величина просто маскируется до доступного числа бит. Знаковые версии **– GL_BYTE, GL_SHORT** и **GL_INT –** имеют маски меньшие на **1** бит. Например, если цветовой индекс должен быть сохранен в знаковом **8-**ми разрядном целом, он сначала маскируется с **0x7f.** Если тип равен **GL_FLOAT,** индекс просто конвертируется в число с плавающей точкой одинарной точности **(**например, индекс **17** будет преобразован к дробному числу **17.0).**

Для упакованных типов данных **(**задаваемых константами, которые начинаются с **GL_UNSIGNED_BYTE_*, GL_UNSIGNED_SHORT_*** или **GL_UNSIGNED_INT_*),** все цветовые компоненты каждого пикселя ужимаются в один элемент данных, заданного беззнакового типа: или в байт, или в короткое целое, или в стандартное целое. Для каждого типа данных допустимы определенные форматы, показанные в таблице **8-3.** Если для упакованного типа данных используется нелегальный формат пикселей, генерируется ошибка **GL_INVALID_OPERATION.**

Замечание: Упакованные форматы пикселей появились в OpenGL версии 1.2.

Таблица **8-3.** Допустимые форматы пикселей для упакованных типов данных

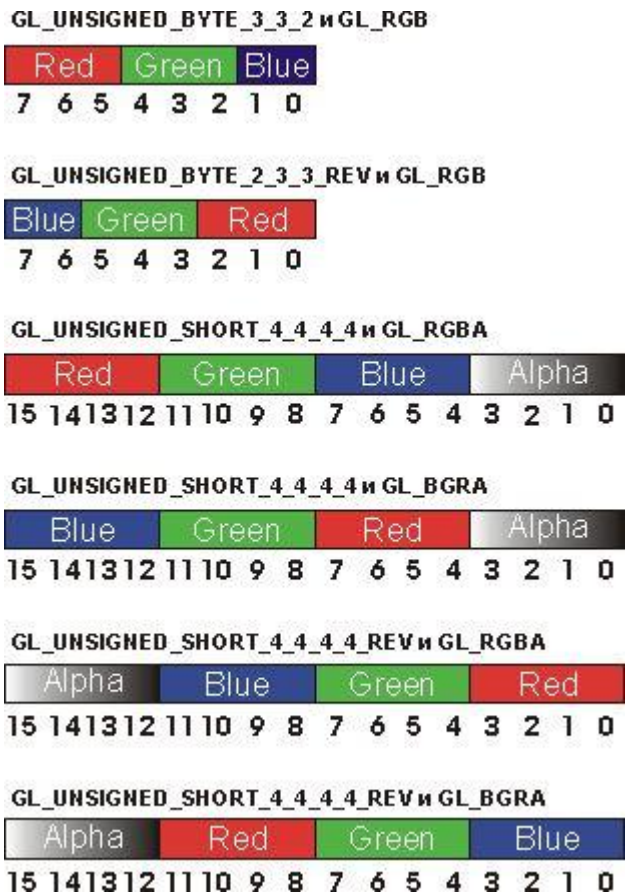
Константа упакованного типа	**Допустимые форматы пикселей**
GL_UNSIGNED_BYTE_3_3_2	**GL_RGB**
GL_UNSIGNED_BYTE_2_3_3_REV	**GL_RGB**
GL_UNSIGNED_SHORT_5_6_5	**GL_RGB**
GL_UNSIGNED_SHORT_5_6_5_REV	**GL_RGB**
GL_UNSIGNED_SHORT_4_4_4_4	**GL_RGBA, GL_BGRA**
GL_UNSIGNED_SHORT_4_4_4_4_REV	**GL_RGBA, GL_BGRA**
GL_UNSIGNED_SHORT_5_5_5_1	**GL_RGBA, GL_BGRA**
GL_UNSIGNED_SHORT_1_5_5_5_REV	**GL_RGBA, GL_BGRA**
GL_UNSIGNED_INT_8_8_8_8	**GL_RGBA, GL_BGRA**
GL_UNSIGNED_INT_8_8_8_8_REV	**GL_RGBA, GL_BGRA**
GL_UNSIGNED_INT_10_10_10_2	**GL_RGBA, GL_BGRA**

GL_UNSIGNED_INT_2_10_10_10_REV	GL_RGBA, GL_BGRA

Порядок цветовых величин в битовых полях упакованных пиксельных данных определяется по формату пикселей и по тому, содержит ли константа типа суффикс **_REV.** Без суффикса **_REV** цветовые компоненты располагаются таким образом, что первый цвет занимает старшие биты. С суффиксом **_REV** порядок упаковки компонентов становится обратным, то есть первый цвет начинается с самых младших битов.

Для иллюстрации этого, посмотрите на рисунок **8-5,** который показывает порядок битовых полей для типов **GL_UNSIGNED_BYTE_3_3_2, GL_UNSIGNED_BYTE_2_3_3_REV** и **4** допустимые комбинации **GL_UNSIGNED_SHORT_4_4_4_4 (**и **REV)** типов данных и пиксельных форматов **RGBA/BGRA.** Организация битовых полей для других **14-**ти допустимых типов данных с форматами пикселей принципиально не отличается от рассмотренной здесь.

Рисунок **8-5.** Порядок компонентов для нескольких типов данных и форматов пикселей



Старший бит каждого цветового компонента всегда упаковывается в старший бит типа данных. Хранение одного компонента не затрагивается никакими режимами хранения пикселей, хранение целого пикселя может затрагиваться режимом переключения байт.

8.2.1.2 Запись пиксельных данных из процессорной памяти в буфер кадра

```
void glDrawPixels (GLsizei width, GLsizei height, GLenum format, GLenum type,
const GLvoid *pixels);
```

Рисует прямоугольник пиксельных данных размерами ***width*** и ***height***. Пиксельный прямоугольник рисуется таким образом, что его левый нижний угол находится в текущей позиции растра. Аргументы ***format*** и ***type*** имеют то же значение, что и для команды **glReadPixels()**. (Смотрите таблицы **8-1** и **8-2**.) Массив, на который указывает ***pixels***, должен содержать пиксельные данные для рисования. Если текущая позиция растра недопустима, ничего не рисуется, а позиция растра остается недопустимой.

Пример **8-3 –** это часть программы, которая использует **glDrawPixels()** для рисования прямоугольника пикселей в левом нижнем углу окна. Функция **makeCheckImage()** создает **RGB** массив **64x64**, содержащий изображение шахматной доски. Команда **glRasterPos2i(0,0)** позиционирует левый нижний угол изображения. Пока игнорируйте вызов **glPixelStorei()**.

Пример **8-3.** Использование **glDrawPixels()**: файл **image.cpp**

```
#define checkImageWidth 64
#define checkImageHeight 64

GLubyte checkImage[checkImageHeight][checkImageWidth][3];

void makeCheckImage()
{
   int i,j,c;

   for (i=0;i<checkImageHeight;i++)
   {
      for (j=0;j<checkImageWidth;j++)
      {
         c=(((i&0x08)==0)^((j&0x8)==0))*255;
         checkImage[i][j][0]=(GLubyte)c;
         checkImage[i][j][1]=(GLubyte)c;
         checkImage[i][j][2]=(GLubyte)c;
      }
   }
}

void init(void)
{
   glClearColor(0.0,0.0,0.0,0.0);
   glShadeModel(GL_FLAT);
   makeCheckImage();
   glPixelStorei(GL_UNPACK_ALIGNMENT,1);
}

void display(void)
{
   glClear(GL_COLOR_BUFFER_BIT);
   glRasterPos2i(0,0);

glDrawPixels(checkImageWidth,checkImageHeight,GL_RGB,GL_UNSIGNED_BYTE,checkImage);
   glFlush();
}
```

При использовании **glDrawPixels()** для записи **RGBA** или цвето-индексной информации, вам может потребоваться управлять тем, какой буфер используется для рисования с помощью команды **glDrawBuffer()**.

8.2.1.3 Копирование пиксельных данных внутри буфера кадра

```
void glCopyPixels (GLint x, GLint y, GLsizei width, GLsizei height, GLenum
buffer);
```

Копирует пиксельные данные из прямоугольника в буфере кадра, чей левый нижний угол задается координатами **(*x*, *y*)**, а размеры – аргументами ***width*** и ***height***. Данные

копируются в новую позицию, чей левый нижний угол задается текущей позицией растра. Аргумент ***buffer*** может принимать значения **GL_COLOR, GL_STENCIL** или **GL_DEPTH,** задавая используемый буфер кадра. **glCopyPixels()** ведет себя так же как пара смежных вызовов **glReadPixels()** и **glDrawPixels()** со следующим отображением параметра ***buffer*** на значение параметра ***format***:

- Если ***buffer*** равен **GL_DEPTH** или **GL_STENCIL,** то ***format*** равен **GL_DEPTH_COMPONENT** или **GL_STENCIL_INDEX,** соответственно.
- Если задан **GL_COLOR,** используется **GL_RGBA** или **GL_COLOR_INDEX** в зависимости от того, работает ли система в **RGBA** или индексном режиме.

Заметьте, что для команды **glCopyPixels()** нет нужды в параметрах ***format*** или ***type***, поскольку данные не копируются в процессорную память. Буфер – источник и буфер – приемник для команды **glCopyPixels()** задаются командами **glReadBuffer()** и **glDrawBuffer()** соответственно.

Для всех трех описанных функций конверсия данных поступающих в или из буфера кадра зависит от того, какие режимы активны в момент проведения операции.

8.2.2 Конвейер обработки изображений

В этом разделе обсуждается конвейер обработки изображений: режимы хранения пикселей и операции по передаче пикселей, которые включают конверсию пиксельных данных. Вы также можете увеличивать или уменьшать пиксельный прямоугольник до его рисования, применяя команду **glPixelZoom().** Порядок операций показан на рисунке **8-6.**

Рисунок **8-6.** Конвейер обработки изображений

Когда вызывается команда **glDrawPixels()** данные сначала распаковываются из процессорной памяти в соответствии с активизированными режимами хранения пикселей, затем выполняются операции пиксельного переноса. Затем получившиеся пиксели растеризуются. Во время растеризации пиксельный прямоугольник может быть увеличен или уменьшен. Затем производятся пофрагментные операции, и пиксели заносятся в буфер кадра.

Когда вызывается команда **glReadPixels()**, данные считываются из буфера кадра, над ними производятся операции пиксельного переноса, и результирующие данные упаковываются в процессорную память.

glCopyPixels() производит все операции пиксельного переноса так же, как это делает **glReadPixels()**. Затем получившиеся данные записываются так же как в случае **glDrawPixels()**, но преобразования второй раз не производятся.

На рисунке **8-7** изображен процесс перемещения данных командой **glCopyPixels()** **(**процесс начинается с буфера кадра**).**

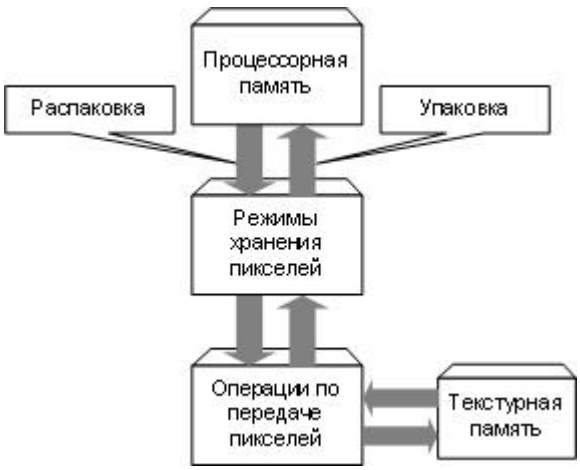
Рисунок **8-7.** Путь пикселей при вызове команды **glCopyPixels()**

Из рисунка **8-8**, отображающего процесс вывода битовой карты, вы можете сделать вывод о том, что визуализация битовых карт проще, чем визуализация изображений, поскольку ни операции пиксельного переноса, ни масштабирование не производится.

Рисунок **8-8.** Путь пикселей при вызове команды **glBitmap()**

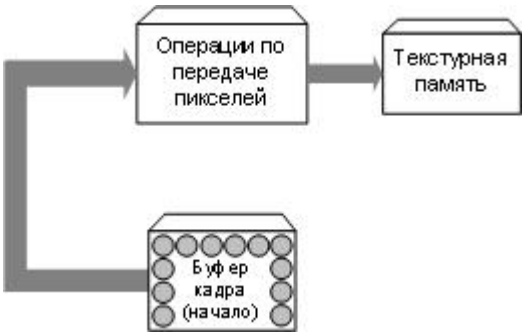
Обратите внимание, режимы хранения пикселей и операции пиксельного переноса также применяются к текстурам, во время того, как они считываются из текстурной памяти или записываются в текстурную память. Рисунок **8-9** демонстрирует эффект от команд **glTexImage*()**, **glTexSubImage*()** и **glGenTexImage*()**.

Рисунок **8-9.** Путь пикселей при вызове команд **glTexImage*()**, **glTexSubImage*()** и **glGenTexImage*()**



Как показано на рисунке **8-10,** когда данные из буфера кадра копируются в текстурную память **(**командами **glCopyTexImage*()** или **glCopyTexSubImage()),** производятся только операции пиксельного переноса**.**

Рисунок **8-10.** Путь пикселей при вызове команд **glCopyTexImage*()** и **glCopyTexSubImage*()**



8.2.2.1 Упаковка и распаковка пикселей

Упаковка и распаковка пикселей касается процесса записи данных в процессорную память и считывания их оттуда.

Изображение, хранимое в памяти, состоит из определенного количества фрагментов (от одного до четырех) на каждый пиксель. Эти фрагменты называются *элементами*. Данные могут состоять только из цветовых индексов или светлоты (светлота – это взвешенная сумма величин красного, зеленого и синего), но они также могут состоять из красного, зеленого, синего и альфа компонентов для каждого пикселя. Возможные варианты структуры пиксельных данных, или *форматы*, определяют число и порядок элементов для каждого пикселя.

Некоторые элементы (такие как цветовой индекс или индекс трафарета) являются целыми, а другие (такие как красный, зеленый, синий и альфа компоненты) – числами с плавающей точкой, обычно варьируясь между **0.0** и **1.0**. Дробные компоненты обычно сохраняются в буфере кадра с меньшим разрешением, чем требуется для сохранения полного числа с плавающей точкой (цветовые компоненты, например, могут храниться в **8**-ми битах). Точное число битов, используемых для представления компонентов, зависит от конкретной используемой аппаратуры. Таким образом, часто бессмысленно хранить компоненты в виде **32**-разрядных чисел с плавающей точкой, тем более, что изображение может состоять из миллионов пикселей.

Элементы могут храниться в памяти в виде различных типов данных от **8**-разрядных байтов до **32**-разрядных целых чисел или чисел с плавающей точкой. **OpenGL** определяет преобразование каждого компонента каждого формата к каждому из возможных типов данных. Имейте в виду, пытаясь сохранить компонент с высоким разрешением в тип данных, представляемый небольшим числом бит, вы можете потерять данные.

8.2.2.2 Управление режимами хранения пикселей

Данные изображения обычно хранятся в процессорной памяти в двух- или трехмерных прямоугольных массивах. Довольно часто вам требуется отобразить или сохранить часть изображения, соответствующую части прямоугольного массива. Кроме того, вам, возможно, придется учитывать, что различные машины используют разные соглашения о порядке байт. Наконец, некоторые машины могут включать аппаратуру, которая эффективнее перемещает данные в буфер кадра и из него, если данные в процессорной памяти выровнены по границе **2**-ух, **4**-ех или **8**-ми байт. Для таких машин может понадобиться управление выравниванием байт. Все вопросы, затронутые в данном абзаце, решаются путем использования режимов хранения пикселей. Вы задаете эти режимы, используя команду **glPixelStore*()**, которая уже использовалась в нескольких примерах.

Все режимы хранения пикселей, поддерживаемые **OpenGL**, управляются **glPixelStore*()**. Обычно производится несколько последовательных вызовов данной команды для установки значений различных параметров.

```
void glPixelStore{if} (GLenum pname, TYPE param);
```

Устанавливает режимы хранения пикселей, которые влияют на работу команд **glDrawPixels()**, **glReadPixels()**, **glBitmap()**, **glPolygonStipple()**, **glTexImage1D()**, **glTexImage2D()**, **glTexImage3D()**, **glTexSubImage1D()**, **glTexSubImage2D()**, **glTexSubImage3D()**, **glGetTexImage()**, а также, если присутствует подмножество команд обработки изображений **glGetColorTable()**, **glGetConvolutionFilter()**, **glGetSeparableFilter()**, **glGetHistogram()**, **glGetMinmax()**. Возможные имена параметров для ***pname*** перечислены в таблице **8-4.** Параметры **GL_UNPACK_*** управляют тем, как данные распаковываются из памяти командами **glDrawPixels()**, **glBitmap()**, **glPolygonStipple()**, **glTexImage1D()**, **glTexImage2D()**, **glTexImage3D()**, **glTexSubImage1D()**, **glTexSubImage2D()** и **glTexSubImage3D()**. Параметры **GL_PACK_*** управляют тем, как данные упаковываются в память командами **glReadPixels()**, **glGenTexImage()**, а также, если присутствует подмножество команд обработки изображений **glGetColorTable()**, **glGetConvolutionFilter()**, **glGetSeparableFilter()**, **glGetHistogram()**, **glGetMinmax()**. Параметры **GL_UNPACK_IMAGE_HEIGHT**, **GL_PACK_IMAGE_HEIGHT**, **GL_UNPACK_SKIP_IMAGES**, **GL_PACK_SKIP_IMAGES** влияют только на **3D** текстурирование **(glTexImage3D()**, **glTexSubImage3D()**, **glGenTexImage(GL_TEXTURE_3D,...))**.

Таблица **8-4.** Параметры **glPixelStore()**

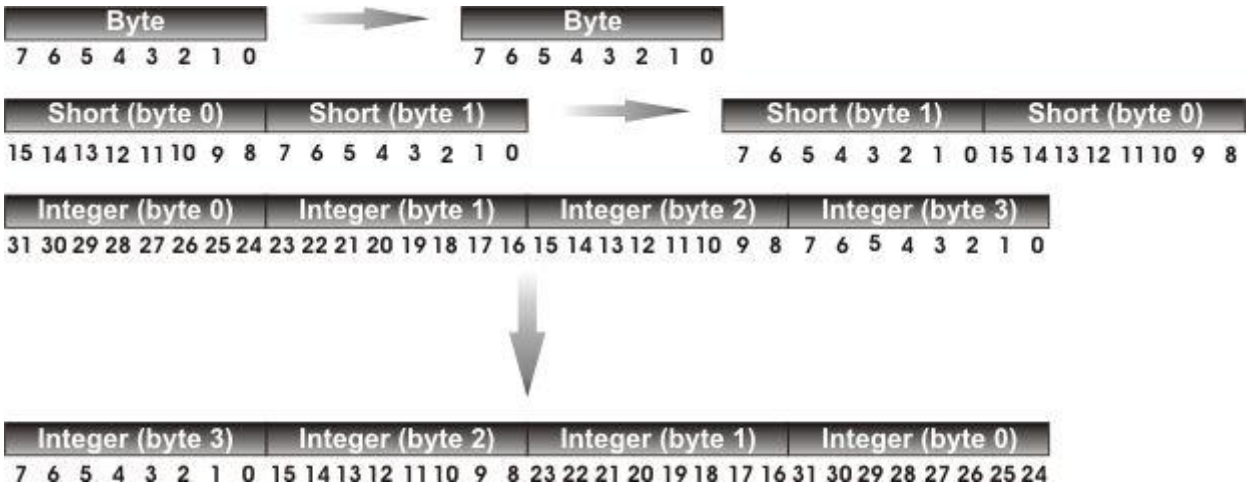
Имя параметра	Тип	Начальное значение	Допустимые величины
GL_UNPACK_SWAP_BYTES, GL_PACK_SWAP_BYTES	GLboolean	FALSE	TRUE / FALSE
GL_UNPACK_LSB_FIRST, GL_PACK_ LSB_FIRST	GLboolean	FALSE	TRUE / FALSE
GL_UNPACK_ROW_LENGTH, GL_PACK_ROW_LENGTH	GLint	0	любое неотрицательное целое
GL_UNPACK_SKIP_ROWS, GL_PACK_SKIP_ROWS	GLint	0	любое неотрицательное целое
GL_UNPACK_SKIP_PIXELS, GL_PACK_SKIP_PIXELS	GLint	0	любое неотрицательное целое
GL_UNPACK_ALIGNMENT, GL_PACK_ALIGNMENT	GLint	4	1, 2, 4, 8
GL_UNPACK_IMAGE_HEIGHT, GL_PACK_IMAGE_HEIGHT	GLint	0	любое неотрицательное целое
GL_UNPACK_SKIP_IMAGES, GL_PACK_SKIP_IMAGES	GLint	0	любое неотрицательное целое

Поскольку соответствующие параметры для упаковки и распаковки имеют одинаковый смысл, они обсуждаются вместе в оставшейся части этого раздела и ссылки на них производятся без префикса **GL_PACK** или **GL_UNPACK.** Например, ***SWAP_BYTES** относится и к **GL_UNPACK_SWAP_BYTES**, и к **GL_PACK_SWAP_BYTES.**

Если параметр ***SWAP_BYTES** равен **FALSE (**значение по умолчанию**)**, то порядок байтов в памяти такой же, как естественный порядок байтов для клиента **OpenGL**; если параметр равен **TRUE–** байты расположены в обратном порядке. Реверс байтов применяется для элементов любого размера, но имеет смысл только для элементов, состоящих из нескольких байт.

Эффект от реверса байтов может различаться в разных реализациях **OpenGL.** Рисунок **8-10** иллюстрирует эффект от байтового реверса для различных типов данных для реализации, в которой **GLubyte** состоит из **8** бит, **GLushort –** из **16**, а **GLuint –** из **32.** Заметьте, что обратный порядок байт не сказывается на однобайтовых данных.

Рисунок **8-11.** Обратный порядок байт для данных типа **Byte**, **Short** и **Integer**

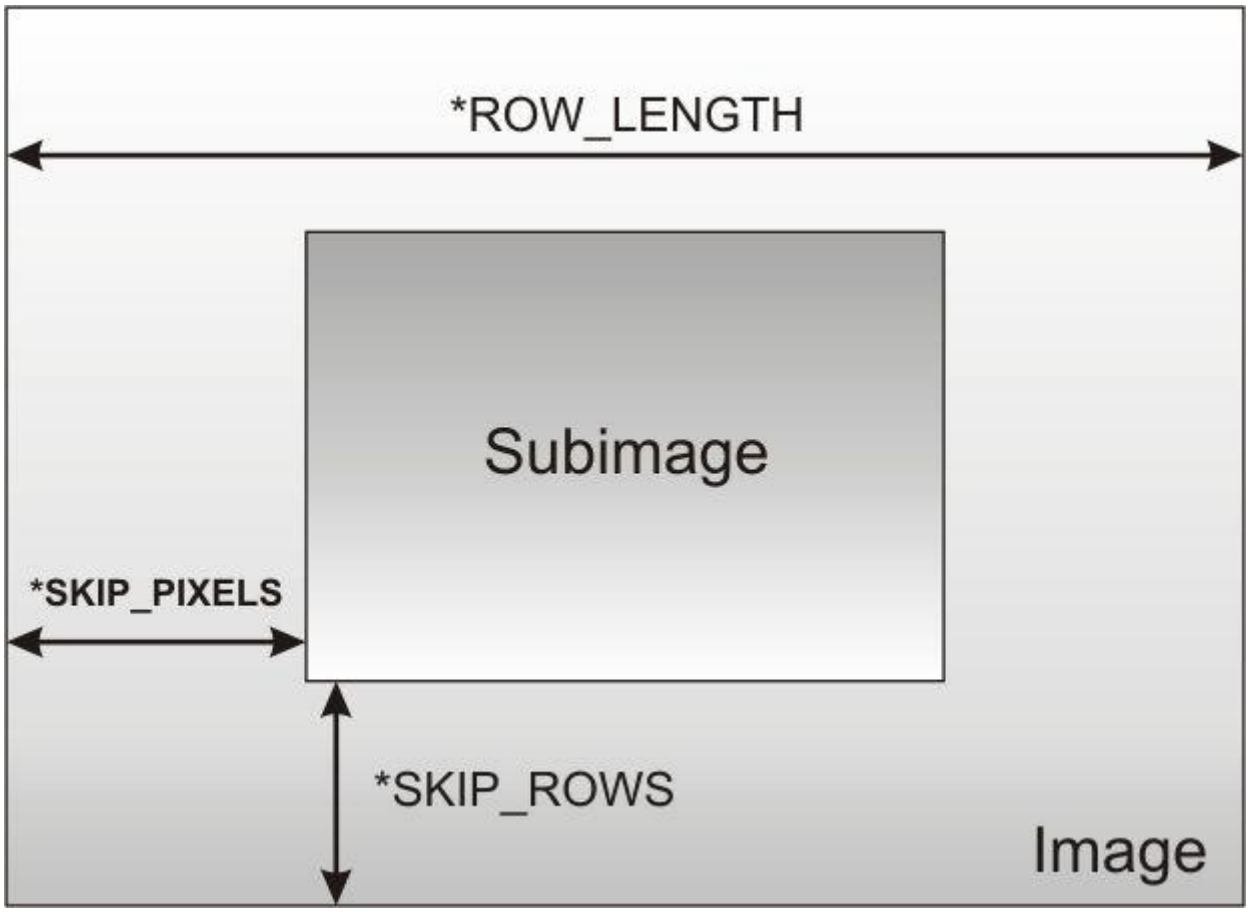


Замечание: Если ваше приложение OpenGL не разделяет изображения с другими машинами, вы можете игнорировать вопросы, связанные с порядком байт. Если ваша программа должна визуализировать изображение OpenGL, которое было создано на другой машине с другим порядком байт, этот порядок может быть изменен с помощью *SWAP_BYTES. Однако параметр *SWAP_BYTES не позволяет вам изменять порядок элементов (например, поменять местами красный и зеленый компоненты).

Параметр ***LSB_FIRST** имеет значение только при рисовании или считывании однобитовых изображений или битовых карт, для которых на каждый пиксель сохраняется или восстанавливается по одному биту данных. Если ***LSB_FIRST** равен **FALSE (**значение по умолчанию**)** биты берутся из байтов начиная со старшего, иначе они забираются в обратном порядке. Например, если ***LSB_FIRST**равен **FALSE,** а интересующий нас байт **– 0x31,** битами по порядку будут **{0, 0, 1, 1, 0, 0, 0, 1}.** Если же ***LSB_FIRST** равен **TRUE –** порядок будет следующим **{1, 0, 0, 0, 1, 1, 0, 0}.**

Иногда вам нужно нарисовать или считать только часть прямоугольника данных изображения **(**также прямоугольную**),** сохраненного в памяти. Если прямоугольник в памяти больше, чем прямоугольник, который нужно нарисовать или считать, вам нужно задать реальную длину **(**в пикселях**)** большего прямоугольник с помощью ***ROW_LENGTH.** Если ***ROW_LENGTH** равен **0 (**значение по умолчанию**)** длина ряда принимается такой же, как и ширина, задаваемая в командах **glReadPixels(), glDrawPixels()** или **glCopyPixels().** Вам также нужно задать количество рядов и пикселей, которые нужно пропустить до начала копирования данных для меньшего прямоугольника. Эти числа задаются при помощи параметров ***SKIP_ROWS** и ***SKIP_PIXELS,** как показано на рисунке **8-12.** По умолчанию оба параметра равны **0,** так что копирование начинается с нижнего левого угла.

Рисунок **8-12.** Параметры ***SKIP_ROWS, *SKIP_PIXELS** и ***ROW_LENGTH**



Часто аппаратура оптимизирована для перемещения пиксельных данных в память и из памяти, если данные сохранены в памяти с определенным выравниваем. Например, в машине с **32-**х разрядными словами, аппаратура часто намного быстрее извлекает данные, если они изначально выровнены по границе **32-**разрядов, что обычно означает адрес кратный **4.** Похожим образом, **64-**битовые архитектуры могут работать лучше, когда данные выровнены по границам в **8** байт.

Для примера, предположим, что ваша машина лучше работает, когда пиксельные данные выровнены по **4-**ех байтовым границам. Изображения могут быть сохранены более эффективно, если сделать так, чтобы данные для каждого ряда изображения начинались на **4-**ех байтовой границе. Если изображение имеет ширину в **5** пикселей, и каждый пиксель состоит из **1** байта на красный, **1** на зеленый и **1** на синий компоненты, для одного ряда требуется **5x3=15** байт данных. Максимальная эффективность может быть достигнута, если первый и последующие ряды будут начинаться с **4-**ех байтовой границы, то есть на каждый ряд будет тратиться **1** дополнительный байт памяти. Если ваши данные сохранены подобным образом, верно установите значение параметра ***ALIGNMENT (**в данном случае в значение **4).**

Если ***ALIGNMENT** равен **1,** используется следующий доступный байт. Если он равен **2,** в конце каждого ряда если это необходимо будет пропущен дополнительный байт, чтобы первый байт следующего ряда начинался с адреса кратного **2-**ум. В случае битовых карт **(**или **1-**битовый изображений**),** где для каждого пикселя хранится один бит, байтовое смещение также работает, но вам нужно подсчитывать отдельные биты. Например, если вы сохраняете по одному биту на пиксель, длина ряда составляет **75** пикселей, а выравнивание равно **4,** то каждый ряд потребует **75/8** или **9 3/8** байт. Поскольку число **12 –** это наименьшее число кратное **4** большее **9 3/8,** для каждого ряда будет использоваться **12** байт памяти. Если выравнивание равно **1,** то будет использовано **10** байт, то есть **9 3/8** округленное до целого числа байт.

Замечание: Значением по умолчанию для *ALIGNMENT является 4. Предположение о том, что данные изображения плотно упакованы в памяти (то есть *ALIGNMENT равно 1) является частой ошибкой.

Параметры ***IMAGE_HEIGHT** и ***SKIP_IMAGES** влияют только на определение и опрос трехмерных текстур.

8.2.2.3 Операции перемещения пикселей

В то время, как данные изображения передаются из памяти в буфер кадра или из буфера кадра в память, **OpenGL** может производить над ними некоторые операции. Например, могут быть изменены диапазоны вариации компонентов – обычно красный компонент варьируется в диапазоне от **0.0** до **1.0**, но вам может потребоваться хранить его в каком-либо другом диапазоне; или может быть данные, используемые вами, пришли из другой системы с другим диапазоном вариации красного компонента. Вы можете даже создать карты для конверсии цветовых индексов или цветовых компонентов во время переноса пикселей. Подобная конверсия, производимая во время передачи пикселей в буфер кадра или из него, называется операциями перемещения пикселей. Этими операциями управляют команды **glPixelTransfer*()**и **glPixelMap*()**.

Имейте в виду, что хотя цветовой, глубинный и трафаретный буфер имеют много общего, они не ведут себя одинаково во всех ситуациях, и некоторые из режимов имеют специальное назначение. Все детали о режимах обсуждаются в данном и последующих разделах, включая все специальные случаи.

Некоторые из характеристик функционирования операций пиксельного переноса задаются с помощью команды **glPixelTransfer*()**. Другие характеристики задаются с помощью **glPixelMap*()**, описанной в следующем разделе.

```
void glPixelTransfer{if} (GLenum pname, TYPE param);
```

Устанавливает режимы пиксельной передачи, которые влияют на функционирование команд **glDrawPixels())**, **glReadPixels()**, **glCopyPixels()**, **glTexImage1D()**, **glTexImage2D()**, **glTexImage3D()**, **glCopyTexImage1D()**, **glCopyTexImage2D()**, **glTexSubImage1D()**, **glTexSubImage2D()**, **glTexSubImage3D()**, **glCopyTexSubImage1D()**, **glCopyTexSubImage2D()**, **glCopyTexSubImage3D()**, **glGetTexImage()**. Аргумент ***pname*** должен принимать одно из значений, приведенных в первой колонке таблицы **8-5**, а ***param***– одно из допустимых для соответствующего параметра значений.

Таблица **8-5**. Параметры команды **glPixelTransfer*()**

Имя параметра	Тип	Начальное значение	Допустимый диапазон
GL_MAP_COLOR	GLboolean	FALSE	TRUE/FALSE
GL_MAP_STENCIL	GLboolean	FALSE	TRUE/FALSE
GL_INDEX_SHIFT	GLint	0	$(-\infty;+\infty)$
GL_INDEX_OFFSET	GLint	0	$(-\infty;+\infty)$
GL_RED_SCALE	GLfloat	1.0	$(-\infty;+\infty)$
GL_GREEN_SCALE	GLfloat	1.0	$(-\infty;+\infty)$
GL_BLUE_SCALE	GLfloat	1.0	$(-\infty;+\infty)$
GL_ALPHA_SCALE	GLfloat	1.0	$(-\infty;+\infty)$
GL_DEPTH_SCALE	GLfloat	1.0	$(-\infty;+\infty)$
GL_RED_BIAS	GLfloat	0.0	$(-\infty;+\infty)$
GL_GREEN_BIAS	GLfloat	0.0	$(-\infty;+\infty)$

GL_BLUE_BIAS	**GLfloat**	**0.0**	$(-\infty;+\infty)$
GL_ALPHA_BIAS	**GLfloat**	**0.0**	$(-\infty;+\infty)$
GL_DEPTH_BIAS	**GLfloat**	**0.0**	$(-\infty;+\infty)$
GL_POST_CONVOLUTION_RED_SCALE	**GLfloat**	**1.0**	$(-\infty;+\infty)$
GL_POST_CONVOLUTION_GREEN_SCALE	**GLfloat**	**1.0**	$(-\infty;+\infty)$
GL_POST_CONVOLUTION_BLUE_SCALE	**GLfloat**	**1.0**	$(-\infty;+\infty)$
GL_POST_CONVOLUTION_ALPHA_SCALE	**GLfloat**	**1.0**	$(-\infty;+\infty)$
GL_POST_CONVOLUTION_RED_BIAS	**GLfloat**	**0.0**	$(-\infty;+\infty)$
GL_POST_CONVOLUTION_GREEN_BIAS	**GLfloat**	**0.0**	$(-\infty;+\infty)$
GL_POST_CONVOLUTION_BLUE_BIAS	**GLfloat**	**0.0**	$(-\infty;+\infty)$
GL_POST_CONVOLUTION_ALPHA_BIAS	**GLfloat**	**0.0**	$(-\infty;+\infty)$
GL_POST_COLOR_MATRIX_RED_SCALE	**GLfloat**	**1.0**	$(-\infty;+\infty)$
GL_POST_COLOR_MATRIX_GREEN_SCALE	**GLfloat**	**1.0**	$(-\infty;+\infty)$
GL_POST_COLOR_MATRIX_BLUE_SCALE	**GLfloat**	**1.0**	$(-\infty;+\infty)$
GL_POST_COLOR_MATRIX_ALPHA_SCALE	**GLfloat**	**1.0**	$(-\infty;+\infty)$
GL_POST_COLOR_MATRIX_RED_BIAS	**GLfloat**	**0.0**	$(-\infty;+\infty)$
GL_POST_COLOR_MATRIX_GREEN_BIAS	**GLfloat**	**0.0**	$(-\infty;+\infty)$
GL_POST_COLOR_MATRIX_BLUE_BIAS	**GLfloat**	**0.0**	$(-\infty;+\infty)$

Замечание: Параметры GL_POST_CONVOLUTION_* и GL_POST_COLOR_MATRIX_* присутствуют только в том случае, если ваша реализация OpenGL поддерживает подмножество команд обработки изображений.

Если параметр **GL_MAP_COLOR** или **GL_MAP_STENCIL** равны **TRUE**, активизировано отображение **(mapping)** пикселей. Все остальные параметры непосредственно влияют на величины цветовых компонент пикселей.

Масштаб **(scale)** и скос **(bias)** могут быть применены к красному, зеленому, синему, альфа и глубинному компонентам. Например, вам нужно масштабировать красный, зеленый и синий компоненты, считанные из буфера кадра, перед конверсией их к формату светлоты в процессорной памяти. Светлота **(luminance)** вычисляется как сумма красного, зеленого и синего компонентов, то есть если вы используете значения по умолчанию для **GL_RED_SCALE**, **GL_GREEN_SCALE** и **GL_BLUE_SCALE**, все компоненты будут вносить одинаковый вклад в результирующую величину интенсивности или светлоты. Если вам нужно конвертировать **RGB** в светлоту в соответствии со стандартом **NTSC**, установите **GL_RED_SCALE** в **0.30**, **GL_GREEN_SCALE** в **0.59**, а **GL_BLUE_SCALE** в **0.11**.

Индексы **(**цветовой и трафаретный**)** также могут быть трансформированы. В случае индексов применяются сдвиг **(shift)** и смещение. Это полезно, если вам нужно управлять тем, какая часть цветовой таблицы используется.

8.2.2.4 Отображение пикселей

Все цветовые компоненты, цветовые индексы и трафаретные индексы могут быть изменены с помощью ссылочной таблицы до помещения в экранную память. Это делается при помощи команды **glPixelMap*()**.

```
void glPixelMap{ui us f}v (GLenum map, GLint mapsize, const TYPE *values);
```

Загружает пиксельную карту, заданную аргументом ***map***, с числом вхождений ***mapsize***, на величины которой указывает аргумент ***values***. Таблица **8-6** отображает все имена и

значения карт; все размеры по умолчанию равны **1,** а величины по умолчанию **– 0.** Размер каждой карты должен быть степенью **2.**

Таблица **8-6.** Имена и значения параметров **glPixelMap*()**

Имя карты	Что на входе	Что на выходе
GL_PIXEL_MAP_I_I	цветовой индекс	цветовой индекс
GL_PIXEL_MAP_S_S	индекс трафарета	индекс трафарета
GL_PIXEL_MAP_I_R	цветовой индекс	**R**
GL_PIXEL_MAP_I_G	цветовой индекс	**G**
GL_PIXEL_MAP_I_B	цветовой индекс	**B**
GL_PIXEL_MAP_I_A	цветовой индекс	**A**
GL_PIXEL_MAP_R_R	**R**	**R**
GL_PIXEL_MAP_G_G	**G**	**G**
GL_PIXEL_MAP_B_B	**B**	**B**
GL_PIXEL_MAP_A_A	**A**	**A**

Максимальный размер карты является машинно-зависимым**.** Вы можете выяснить размеры пиксельных карт на вашей машине с помощью команды **glGetIntegerv().** Для выяснения максимального размера всех пиксельных карт используйте аргумент **GL_MAX_PIXEL_MAP_TABLE,** а для выяснения текущего размера конкретной карты используйте аргументы **GL_PIXEL_MAP_*_TO_*.** Размеры **6** карт**,** касающихся цветового и трафаретного индекса**,** всегда должны быть степенью **2, RGBA**карты могут иметь любой размер от **1** до **GL_MAX_PIXEL_MAP_TABLE.**

Чтобы понять**,** как работают эти таблицы**,** рассмотрим простой пример**.** Предположим**,** что вы хотите создать таблицу в **256** вхождениями**,** которая отображает цветовые индексы на цветовые индексы **(GL_PIXEL_MAP_I_TO_I).** Вы создаете таблицу с одним вхождением для каждой величины от **0** до **255** и инициализируете ее с помощью **glPixelMap*().** Предположим**,** что вы используете таблицу**,** которая отображает все индексы меньше **101 (**индексы от **0** до **100** включительно**)** на **0,** а все остальные индексы **(**от **101** до **255** включительно**)** на **255.** В этом случае ваша таблица состоит из **101** нуля и **155** значений **255.** Пиксельная карта включается с помощью установки параметра **GL_MAP_COLOR** в **TRUE** командой **glPixelTransfer*().** Как только пиксельная карта загружена и активизирована входящие цветовые индексы меньше **101** будут выходить равными **0,** а большие или равные **101 –** равными **255.** Если входящий пиксель больше **255,** он сначала маскируется с **255 (**в процессе чего отбрасываются все биты старше восьмого**)** и в таблице ищется получившаяся величина**.** Если входящий индекс представляет собой число с плавающей точкой **(**например**, 88.14585),** он округляется до ближайшего целого **(**в нашем случае **88)** и это число ищется в таблице **(**результатом в нашем случае будет **0).**

Используя пиксельные карты вы также можете изменять индексы трафарета или конвертировать цветовые индексы в **RGB.**

8.2.2.5 Увеличение, уменьшение и отражение изображения

После того как применены режимы хранения пикселей и операции пиксельного переноса**,** изображения и битовые карты растеризуются**.** Обычно один пиксель изображения записывается в один пиксель экрана**.** Однако вы можете увеличивать**,** уменьшать и даже отражать изображения с помощью команды **glPixelZoom().**

```
void glPixelZoom (GLfloat zoom_x, GLfloat zoom_y);
```

Задает фактор увеличения или уменьшения для операций по записи пикселей **(glDrawPixels()** и **glCopyPixels())** по осям **x** и **y.** По умолчанию $zoom_x$ и $zoom_y$

равны **1.0**. Если оба фактора установить в **2.0**, каждый пиксель изображения будет нарисован на **4**-ех пикселях экрана. Заметьте, что допускаются дробные и отрицательные факторы увеличения или уменьшения изображения. Отрицательные значения факторов отражают изображение относительно текущей позиции растра и соответствующей оси.

Во время растеризации каждый пиксель изображения считается прямоугольником площадью $zoom_x \times zoom_y$, и фрагменты генерируются для всех пикселей экрана, чьи центры лежат внутри этого прямоугольника. Более конкретно, обозначим координаты текущей позиции растра как (x_{rp}, y_{rp}). Если отдельная группа элементов (индексов или компонент) является **n**-ой в ряду и принадлежит **m**-ной колонке, то в оконных координатах она будет покрывать регион, ограниченный прямоугольником с углами в

$$(x_{rp} + zoom_x \cdot n; y_{rp} + zoom_y \cdot m) \text{ и } (x_{rp} + zoom_x \cdot (n+1); y_{rp} + zoom_y \cdot (m+1)).$$

Любые фрагменты, чьи центры лежат внутри этого прямоугольника (а также на его нижней или левой границах) генерируются в соответствии с этой отдельной группой элементов.

Отрицательные факторы могут быть полезны для отражения изображения. В **OpenGL** изображения описываются снизу вверх и слева направо. Если у вас есть изображение, описанное сверху вниз, например, кадр видео, вам может пригодиться **glPixelZoom(1.0,-1.0)**; чтобы выправить изображение под формат **OpenGL**. Убедитесь, что вы должным образом установили текущую растровую позицию.

Пример **8-4** демонстрирует использование команды **glPixelZoom()**. В начале в нижнем левом углу окна рисуется изображение шахматной доски. Нажимая клавишу мыши и передвигая ее, вы выполняете команду **glCopyPixels()**, копируя нижний левый угол окна в текущую позицию курсора. (Если вы скопируете изображение само на себя, оно будет выглядеть довольно неприлично.) Копируемое изображение масштабируется, но изначально оно масштабируется на величину по умолчанию **(1.0)**, так что вы не обратите на это внимание. Клавиши **'z'** и **'Z'** увеличивают и уменьшают факторы масштаба на **0.5**. Любое повреждение содержимого окна, вызывает его перерисовку. Нажатие на клавишу **'r'** сбрасывает изображение и факторы масштаба.

Пример **8-4**. Рисование, копирование и масштабирование пиксельных данных: файл **image.cpp**

```
#include <glut.h>
#include <windows.h>
#include <stdio.h>

#define checkImageWidth 64
#define checkImageHeight 64

GLubyte checkImage[checkImageHeight][checkImageWidth][3];

GLdouble zoomFactor=1.0;
GLuint height;

void makeCheckImage()
{
      int i,j,c;

      for (i=0;i<checkImageHeight;i++)
      {
            for (j=0;j<checkImageWidth;j++)
            {
```

```
                  c=(((i&0x08)==0)^((j&0x8)==0))*255;
                  checkImage[i][j][0]=(GLubyte)c;
                  checkImage[i][j][1]=(GLubyte)c;
                  checkImage[i][j][2]=(GLubyte)c;
            }
      }
}

void init(void)
{
      glClearColor(0.0,0.0,0.0,0.0);
      glShadeModel(GL_FLAT);
      makeCheckImage();
      glPixelStorei(GL_UNPACK_ALIGNMENT,1);
}

void display(void)
{
      glClear(GL_COLOR_BUFFER_BIT);
      glRasterPos2i(0,0);
      glDrawPixels(checkImageWidth,checkImageHeight,
GL_RGB,GL_UNSIGNED_BYTE,checkImage);
      glFlush();
}

void reshape(int w, int h)
{
      glViewport(0,0,(GLsizei) w, (GLsizei) h);
      height=(GLint)h;
      glMatrixMode(GL_PROJECTION);
      glLoadIdentity();
      gluOrtho2D(0.0,(GLdouble)w,0.0,(GLdouble)h);
      glMatrixMode(GL_MODELVIEW);
      glLoadIdentity();
}

void motion(int x,int y)
{
      static GLint screeny;

      screeny=height-(GLint)y;
      glRasterPos2i(x,screeny);
      glPixelZoom(zoomFactor,zoomFactor);
      glCopyPixels(0,0,checkImageWidth,checkImageHeight,GL_COLOR);
      glPixelZoom(1.0,1.0);
      glFlush();
}

void outMsg(bool reset=false)
{
      char buffer[100];

      if (reset)
            sprintf(buffer,"zoomFactor is now %4.1f\n",zoomFactor);
      else
            sprintf(buffer,"zoomFactor reset to 1.0\n");
      MessageBox(NULL,buffer,"Drawing, Copying and Zooming Pixel
Data",MB_OK);
      return;
}

void keyboard(unsigned char key,int x,int y)
{
```

```
        switch(key)
        {
              case 'r':
              case 'R':
                    zoomFactor=1.0;
                    outMsg();
                    glutPostRedisplay();
                    break;
              case 'z':
                    zoomFactor+=0.5;
                    if (zoomFactor>=3.0)
                          zoomFactor=3.0;
                    outMsg(true);
                    break;
              case 'Z':
                    zoomFactor-=0.5;
                    if (zoomFactor<=0.5)
                          zoomFactor=0.5;
                    outMsg(true);

                    break;
              case 27:
                    exit(0);
                    break;
        }
  }

  int main(int argc, char **argv)
  {
        glutInit(&argc,argv);
        glutInitDisplayMode(GLUT_SINGLE|GLUT_RGB);
        glutInitWindowSize(250,250);
        glutInitWindowPosition(100,100);
        glutCreateWindow("Drawing, Copying and Zooming Pixel Data");
        init();
        glutDisplayFunc(display);
        glutReshapeFunc(reshape);
        glutKeyboardFunc(keyboard);
        glutMotionFunc(motion);
        glutMainLoop();
        return 0;
  }
```

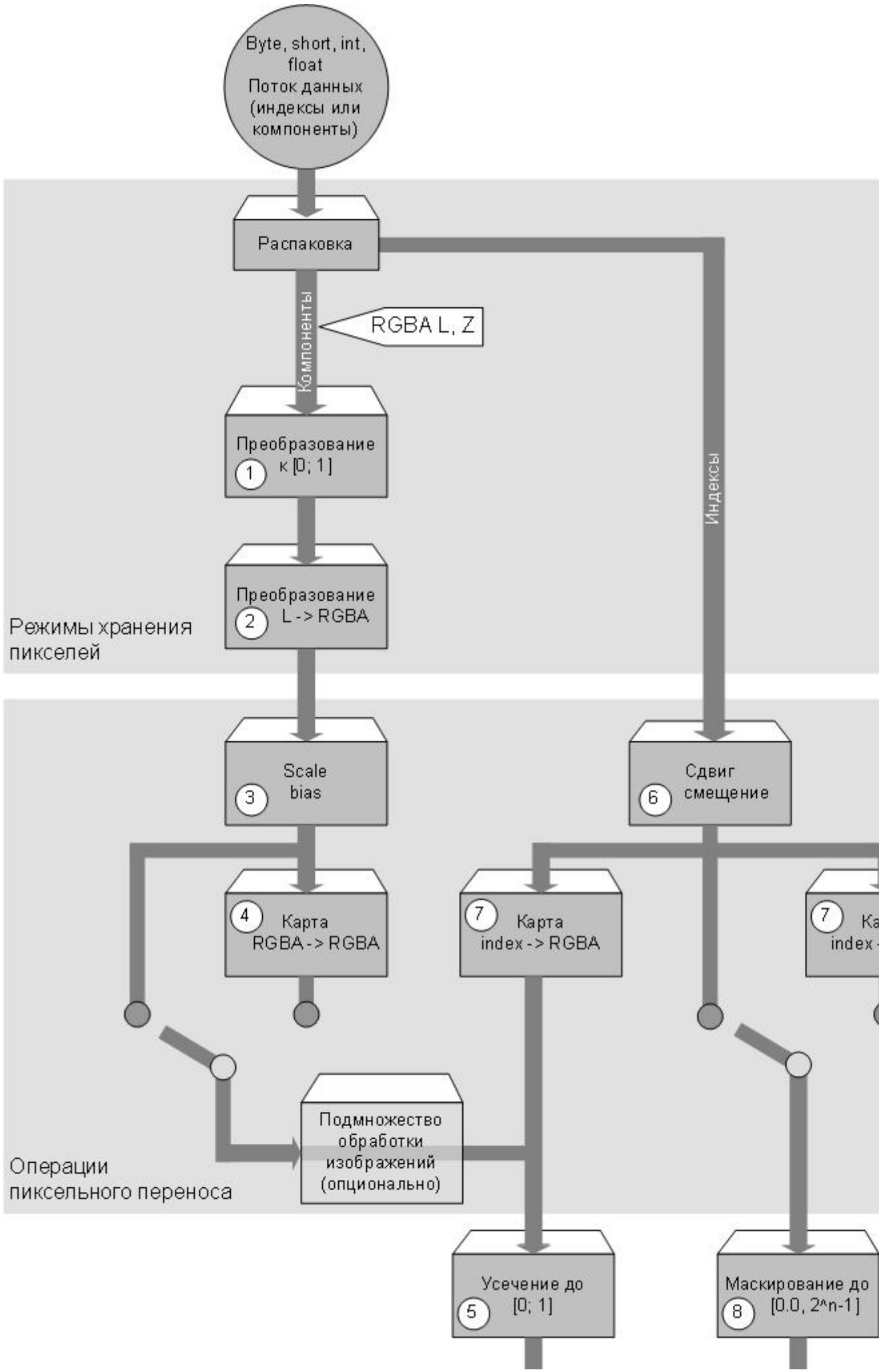
8.2.3 Считывание и рисование прямоугольников пикселей

В этом разделе процессы считывания и записи пикселей в деталях. Преобразования над пикселями во время их переноса из буфера кадра в память (считывание) похожи, но не идентичны тем, которые производятся при переносе пикселей в обратном направлении (записи). Вы можете пропустить этот раздел, если не собираетесь немедленно применять операции перемещения пикселей.

8.2.3.1 Процесс рисования пиксельного прямоугольника

Рисунок **8-13** и следующий за ним список описывают процесс рисования пикселей в буфер кадра.

Рисунок **8-13.** Рисование пикселей с помощью **glDrawPixels()**

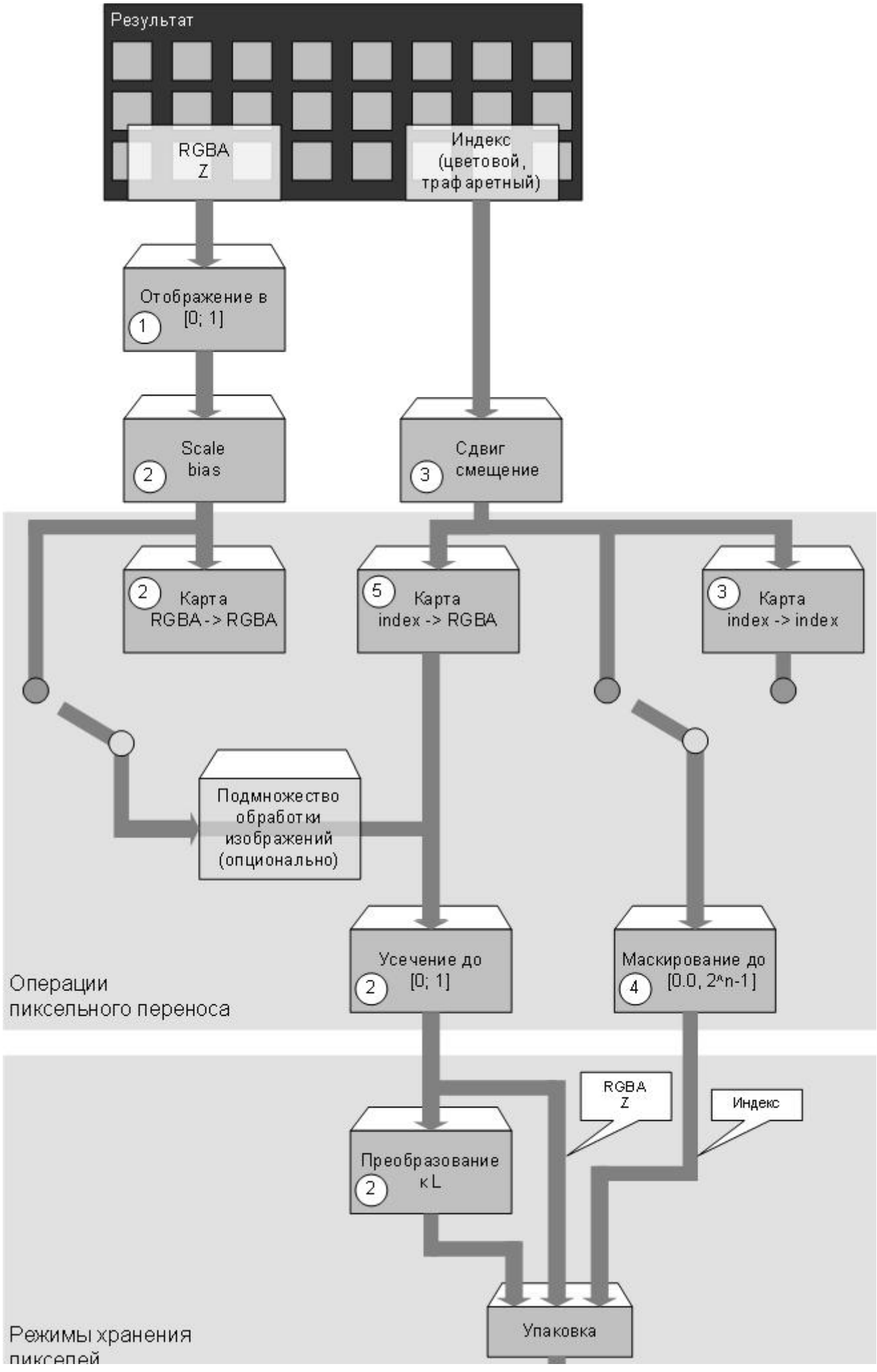

Byte, short, int, float
Поток данных (индексы или компоненты)
Распаковка
Компоненты
RGBA L, Z
Индексы
Преобразование к [0; 1]
1
Преобразование L -> RGBA
2
Режимы хранения пикселей
Scale bias
3
Сдвиг смещение
6
4
Карта RGBA -> RGBA
7
Карта index -> RGBA
7
Ка
index
Подмножество обработки изображений (опционально)
Операции пиксельного переноса
Усечение до [0; 1]
5
Маскирование до [0.0, 2^n-1]
8

1. Если пиксели представляют собой не индексы (то есть формат не равен **GL_COLOR_INDEX** и **GL_STENCIL_INDEX**), первым шагом является преобразование компонентов к формату с плавающей точкой, если это необходимо.
2. Если формат равен **GL_LUMINANCE** или **GL_LUMINANCE_ALPHA**, элемент светлоты конвертируется в **R, G** и **B** с использованием величины светлоты для каждого из **R, G** и **B** компонентов. В формате **GL_LUMINANCE_ALPHA** альфа величина напрямую переходит в компонент **A**. Если задан формат **GL_LUMINANCE** альфа устанавливается равной **1.0**.
3. Каждый компонент (**R, G, B , A** или глубина) умножается на соответствующую величину масштаба, и к ней прибавляется определенный скос. Например, компонент **R**умножается на величину **GL_RED_SCALE** и складывается с величиной **GL_RED_BIAS**.
4. Если **GL_MAP_COLOR** равно **TRUE**, каждый из **R, G, B** и альфа компонент интерполируется до диапазона **[0.0, 1.0]**, умножается на целое число на единицу меньшее размера цветовой таблицы, его дробная часть отбрасывается и ищется в таблице.
5. Далее **R, G, B** и **A** компоненты интерполируются до диапазона **[0.0, 1.0]** (если они не были интерполированы до этого) и конвертируются в формат с фиксированной точкой с максимально допустимым количеством бит слева от точки (в соответствии с возможностями буфера кадра для конкретного компонента).
6. Если вы работаете с индексными величинами (то есть с трафаретными или цветовыми индексами), величины конвертируются к формату с фиксированной точкой (если изначально они были числами с плавающей точкой). Индексы, которые изначально имели формат с фиксированной точкой, остаются таковыми, а все их биты справа от запятой устанавливаются в **0**. Получившиеся индексы сдвигаются влево или вправо на количество разрядов равное абсолютной величине **GL_INDEX_SHIFT**. Индекс сдвигается влево, если **GL_INDEX_SHIFT > 0** и вправо, если **GL_INDEX_SHIFT < 0**. Наконец, к индексу прибавляется значение **GL_INDEX_OFFSET**.
7. Следующий шаг при работе с индексами зависит от того, функционирует ли приложение в **RGBA** или в индексном цветовом режиме. В **RGBA** режиме индекс конвертируется в **RGBA** с использованием карт **GL_PIXEL_MAP_I_TO_R, GL_PIXEL_MAP_I_TO_G, GL_PIXEL_MAP_I_TO_B** и **GL_PIXEL_MAP_I_TO_A**. Иначе, если **GL_MAP_COLOR** равно **GL_TRUE**, индекс заменяется согласно таблице **GL_PIXEL_MAP_I_TO_I** (если **GL_MAP_COLOR** равно **GL_FALSE–** индекс остается неизменным). Если изображение состоит из трафаретных, а не из цветовых индексов, и если **GL_MAP_STENCIL** равно **GL_TRUE**, индекс изменяется согласно таблице **GL_PIXEL_MAP_S_TO_S**. Если **GL_MAP_STENCIL** равен **FALSE** трафаретный индекс не изменяется.
8. Наконец, если индексы не были конвертированы в **RGBA**, они маскируются до количества бит допустимых для цветового индекса или индекса трафарета.

8.2.3.2 Процесс считывания пиксельного прямоугольника

Многие из преобразований, совершаемых во время рисования пиксельного прямоугольника, также производятся и в процессе его считывания. Процесс считывания пиксельного прямоугольника показан на рисунке **8-14** и описан в следующим за ним списке.

Рисунок **8-14.** Считывание пикселей с помощью **glReadPixels()**


Результат
RGBA
Z
Индекс
(цветовой,
трафаретный)
Отображение в
[0; 1]
1
Scale
bias
2
Сдвиг
смещение
3
Карта
RGBA -> RGBA
2
Карта
index -> RGBA
5
Карта
index -> index
3
Подмножество
обработки
изображений
(опционально)
Усечение до
[0; 1]
2
Маскирование до
[0.0, 2^n-1]
4
Операции
пиксельного переноса
RGBA
Z
Индекс
Преобразование
к L
2
Упаковка
Режимы хранения
пикселей

1. Если считываемые индексы не являются индексами **(**то есть формат не равен **GL_COLOR_INDEX** и **GL_STENCIL_INDEX)**, компоненты отображаются в диапазон **[0.0, 1.0]**.
2. Далее, к каждому компоненту применяются масштаб и скос. Если **GL_MAP_COLOR** равен **GL_TRUE**, компоненты заменяются согласно таблице и снова интерполируются в диапазон **[0.0, 1.0]**. Если желаемый вами формат – освещенность, компоненты суммируются **(L=R+G+B)**.
3. Если пиксели являются индексами **(**цветовыми или трафаретными**)**, они сдвигаются, складываются со смещением и, если **GL_MAP_COLOR** равно **TRUE**, также заменяются с помощью таблицы.
4. Если формат хранения **GL_COLOR_INDEX** или **GL_STENCIL_INDEX**, индексы пикселей маскируются до числа бит в типе хранилища **(1, 8, 16** или **32)** и упаковываются в память.
5. Если формат хранения является одним из основанных на компонентах **(**таким как светлота или **RGB)**, пиксели всегда заменяются с помощью карт индекс-к-**RGBA**. Затем работа с ними ведется так, как если бы они были **RGBA** пикселями с самого начала **(**включая потенциальную конверсию в формат светлоты**)**.
6. Наконец, и индексные, и компонентные данные упаковываются в память в соответствии с режимами **GL_PACK*** установленными командой **glPixelStore*()**.

Величины масштаба, скоса, сдвига и смещения такие же как те, что использовались при рисовании пикселей, так что если вы и рисуете и считываете пиксели, не забывайте сбрасывать эти величины в нужные вам значения до чтения или рисования. Похожим образом, различные карты должны быть установлены в нужные значения, если вы планируете использовать их и для чтения, и для записи.

Замечание: Может показаться, что светлота обрабатывается неправильно и при чтении, и при записи. Например, обычно светлота зависит от R, G и B не в одинаковой степени, как можно предположить из рисунков 8-13 и 8-14. Если вам нужно, чтобы светлота вычислялась так, чтобы R, G и B вносили в нее 30, 59 и 11 процентов, вы можете установить GL_RED_SCALE в .30, GL_GREEN_SCALE в .59, а GL_BLUE_SCALE в .11. Тогда вычисленное L будет равно .30R+.59G+.11B.

8.2.4 Советы для увеличения скорости вывода пикселей

- Как вы можете видеть, у **OpenGL** весьма большие возможности для считывания, отображения и манипулирования пиксельными данными. Несмотря на то, что эти возможности часто бывают полезны, они также снижают быстродействие. Здесь приведено несколько советов для ускорения операций с пикселями.
- Для наилучшего быстродействия, установите все параметры, связанные с передачей пикселей в их значения по умолчанию, а факторы масштабирования в **(1.0, 1.0)**.
- Серии фрагментных операций применяются к пикселям во время их отображения в буфере кадра. Для оптимального быстродействия заблокируйте все операции над фрагментами.
- Когда производятся операции над пикселями, заблокируйте все остальные расточительные в смысле производительности механизмы, такие как текстурирование и освещение.
- Если вы используете формат изображения и тип, соответствующие буферу кадра, вы можете сократить работу, которую должна произвести реализация **OpenGL**. Например, если вы записываете изображение в буфер кадра, имеющий формат **RGB** и **8** бит на компонент, вызывайте **glDrawPixels()** с параметром ***format***, установленным в **GL_RGB**, и ***type***, установленным в **GL_UNSIGNED_BYTE**.
- Для некоторых реализаций беззнаковые форматы изображений работают быстрее знаковых.Обычно процесс рисования большого прямоугольника пикселей работает значительно быстрее, чем процесс рисования нескольких меньшего размера, поскольку цена переноса пиксельных данных может быть амортизирована для многих пикселей.

- Вполне возможно, что вам удастся снизить затраты на производительность, уменьшив количество перемещаемых данных. Например, вы можете уменьшить тип данных (в частности до **GL_UNSIGNED_BYTE**) или число компонент (в частности, использовав формат **GL_LUMINANCE_ALPHA**).
- Операции по переносу пикселей, включая отображение пикселей, а также масштабирование, скос, смещение и сдвиг, даже величины, заданные по умолчанию, могут снижать быстродействие.

8.2.5 Подмножество команд по обработке изображений

Подмножество обработки изображений представляет собой набор команд, предоставляющих дополнительные возможности по работе с пикселями, в частности, следующие:

- использование цветовых таблиц для замены величин пикселей
- использование фильтров для изображений
- использование цветовой матрицы для пространственной конверсии и других линейных преобразований
- сбор статистики в виде гистограмм, а также минимальных и максимальных цветовых значений
- использование уравнений цветового наложения для вычисления суммы, разницы, минимума и максимума двух изображений
- использование постоянных факторов цветового наложения для изменения цветов пикселей.

Вам следует использовать подмножество команд обработки изображений, если вам требуется большие возможности по работе с пикселями, чем те, что предоставляются командами **glPixelTransfer*()** и **glPixelMap*()**.

Подмножество команд по обработке изображений **(ImagingSubset)** – это расширение **OpenGL**. Если в библиотеке определен элемент **GL_ARB_imaging**, то подмножество присутствует, и вам доступна вся функциональность, описываемая в следующих разделах. Если элемент не определен, никакая часть описанной функциональности не присутствует в вашей библиотеке.

Независимо от того, передаются ли пиксели **OpenGL** или считываются из нее, они обрабатываются всеми активизированными механизмами подмножества. Команды, на которые влияет подмножество, включают следующие:

- команды, рисующие и считывающие пиксели: **glReadPixels()**, **glDrawPixels()**, **glCopyPixels()**
- команды, определяющие текстуры: **glTexImage1D()**,**glTexImage2D()**, **glCopyTextureImage*D()**, **glSubTexImage1D()**, **glSubTexImage2D()**, **glCopySubTexImage()**.

Рисунок **8-15** иллюстрирует операции, производимые подмножеством обработки изображений, над пикселями, которые поступают в **OpenGL**или считываются оттуда. Большинство из возможностей подмножества может быть активизировано или деактивировано (за исключением цветовой матрицы, которая всегда активизирована).

Рисунок **8-15**. Операции подмножества обработки изображений

Замечание: Константы и функции, связанные с различными расширениями OpenGL могут быть не объявлены в заголовочном файле gl.h, поставляемым с вашим средством разработки, поскольку их поставщики не могут быть заранее уверены, какие расширения поддерживает имеющаяся у вас реализация OpenGL. Чтобы решить проблему с константами, воспользуйтесь дополнительным заголовочным файлом glext.h (более новую его версию вы можете найти на сайте http://www.opengl.org/). Для получения адресов команд - расширений следует воспользоваться специфической для операционной системы функцией. В системах Windows, например, это функция **wglGetProcAddress()**. Примеры ее использования будут приведены дальше в этой главе.

8.2.5.1 Цветовые таблицы

Цветовые таблицы используются для замены цвета пикселя. В приложениях цветовые таблицы могут использоваться для настройки контраста, фильтрации и балансировки изображения.

Существует три отдельные цветовые таблицы. Они работают на разных этапах пиксельного конвейера. Таблица **8-7** перечисляет этапы, на которых цвет пикселя может быть изменен.

Таблица **8-7.** Цветовые таблицы и их место на конвейере

Цветовая таблица	**Воздействует на пиксели**
GL_COLOR_TABLE	когда они входят на конвейер
GL_POST_CONVOLUTION_COLOR_TABLE	после фильтрации
GL_POST_COLOR_MATRIX_COLOR_TABLE	после цветовой матрицы

Каждая таблица может быть включена отдельно с помощью команды **glEnable()** с соответствующим аргументом из таблицы **8-7.**

8.2.5.1.1 Настройка цветовых таблиц

Цветовые таблицы задаются тем же образом, что и одномерные изображения. Как показано на рисунке **8-15**, существует **3** цветовые таблицы для изменения цветовых величин, и все они задаются командой **glColorTable()**.

```
void glColorTable (GLenum target, GLenum internalFormat, GLsizei width, GLenum
format, GLenum type, const GLvoid *data);
```

Если ***target*** равно **GL_COLOR_TABLE, GL_POST_CONVOLUTION_COLOR_TABLE** или **GL_POST_COLOR_MATRIX_COLOR_TABLE,** настраивает соответствующую цветовую таблицу, а если **GL_PROXY_COLOR_TABLE, GL_PROXY_POST_CONVOLUTION_COLOR_TABLE** или **GL_ PROXY_POST_COLOR_MATRIX_COLOR_TABLE –** проверяет, что заданная цветовая таблица может быть размещена в доступных ресурсах. Аргумент ***internalFormat*** используется, чтобы указать внутреннее представление данных в **OpenGL.** Он может принимать следующие символические значения **– GL_ALPHA, GL_ALPHA4, GL_ALPHA8, GL_ALPHA12, GL_ALPHA16, GL_LUMINANCE, GL_LUMINANCE4, GL_LUMINANCE8, GL_LUMINANCE12, GL_LUMINANCE16, GL_LUMINANCE_ALPHA, GL_LUMINANCE4_ALPHA4, GL_LUMINANCE6_ALPHA2, GL_LUMINANCE_8ALPHA8, GL_LUMINANCE12_ALPHA4, GL_LUMINANCE12_ALPHA12, GL_LUMINANCE16_ALPHA16, GL_INTENSITY, GL_INTENSITY4, GL_INTENSITY8, GL_INTENSITY12, GL_INTENSITY16, GL_RGB, GL_R3_G3_B2, GL_RGB4, GL_RGB5, GL_RGB8, GL_RGB10, GL_RGB12, GL_RGB16, GL_RGBA, GL_RGBA2, GL_RGBA4, GL_RGB5_A1, GL_RGBA8, GL_RGB10_A2, GL_RGBA12, GL_RGBA16.** Параметр ***width***, который должен быть степенью **2**, индицирует количество пикселей в цветовой таблице. Аргументы ***format*** и ***type*** описывают формат и тип данных цветовой таблицы. Они имеют такое же значение, как и эквивалентные параметры команды **glDrawPixels().**

Внутренний формат таблицы определяет то, какие компоненты пикселей изображения будут заменены. Например, если вы задаете **GL_RGB** в качестве формата, то красный, зеленый и синий компоненты каждого входящего пикселя будут искаться в соответствующей цветовой таблице и заменяться значениями оттуда. Таблица **8-8** описывает, какие компоненты пикселей будут заменяться при различных внутренних форматах.

Таблица **8-8** Замена значений пикселей цветовой таблицей

Базовый внутренний формат	**Красный компонент**	**Зеленый компонент**	**Синий компонент**	**Альфа компонент**
GL_ALPHA	не изменяется	не изменяется	не изменяется	A_t
GL_LUMINANCE	L_t	L_t	L_t	не изменяется
GL_LUMINANCE_ALPHA	L_t	L_t	L_t	A_t
GL_INTENSITY	I_t	I_t	I_t	I_t
GL_RGB	R_t	G_t	B_t	не изменяется
GL_RGBA	R_t	G_t	B_t	A_t

В таблице **8-8** величины L_t представляют собой светлоту в цветовой таблице, которая влияет только на красный, зеленый и синий компоненты. I_t представляет табличные интенсивности, которые одинаково влияют на красный, зеленый, синий и альфа компоненты.

После того как таблица была применена к изображению, пиксели могут быть смасштабированы и скошены, после чего, их значения приводятся к диапазону **[0; 1].**

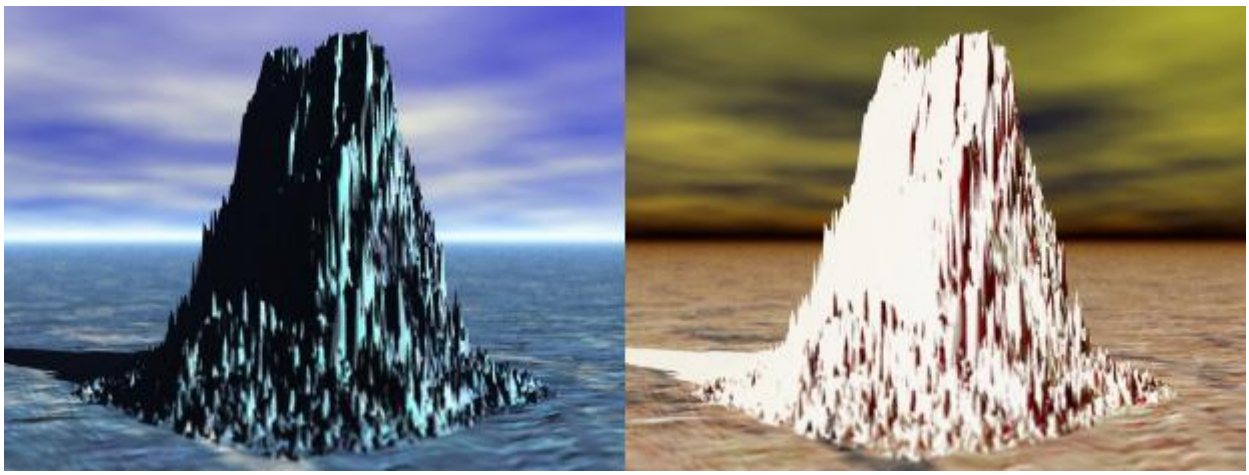
Факторы **GL_COLOR_TABLE_SCALE** и **GL_COLOR_TABLE_BIAS** для каждой таблицы устанавливаются с помощью команды **glColorTableParameter*().**

```
void glColorTableParameter{if}v (GLenum target, GLenum pname, TYPE *param);
```

Устанавливаетпараметры **GL_COLOR_TABLE_SCALE** и **GL_COLOR_TABLE_BIAS** для каждой таблицы**.** ***target*** может быть равно **GL_COLOR_TABLE, GL_POST_CONVOLUTION_COLOR_TABLE** или **GL_POST_COLOR_MATRIX_COLOR_TABLE** и задает целевую таблицу**,** для которой устанавливаются параметры**.** Возможными значениями для ***pnamee*** являются **GL_COLOR_TABLE_SCALE** и **GL_COLOR_TABLE_BIAS.** Величина ***param*** указывает на массив из **4-**ех величин**,** представляющих**,** соответственно**,** модификаторы для красного**,** зеленого**,** синего и альфа компонент**.**

Пример **8-5** показывает**,** как изображение может быть инвертировано посредством цветовой таблицы**.** Таблица настраивается таким образом**,** чтобы она заменяла каждый цвет на его инверсию**.** Пример инвертированного цветовой таблицей изображения приведен на рисунке **8-16.**

Рисунок **8-16.** Исходное изображение айсберга **(**слева**)** и то же изображение**,** инвертированное с помощью цветовой таблицы **(**справа**))**



Пример **8-5.** Замена пикселей с использованием цветовой таблицы**:** файл **colortable.cpp**

```
#include <glut.h>
#include <GL/glaux.h>
#include "glext.h"

//Массив для исходного изображения
AUX_RGBImageRec *image;

//Тип и указатель для функции цветовой таблицы (EXTENSION)
typedef void (APIENTRY * GLCOLORTABLE)
             (GLenum target,GLenum internalFormat,GLsizei width,
              GLenum format, GLenum type, const GLvoid *data);

GLCOLORTABLE glColorTable=NULL;

void init()
{
      int i;
      GLubyte colorTable[256][3];

      //Загрузить исходное изображение из 24-разрядного файла BMP
```

```
	//(библиотека GLAUX)
	image=auxDIBImageLoad("peak.bmp");

	//Получить адрес функции цветовой таблицы
	//с помощью функции специфической для реализации OpenGL для
Windows
	glColorTable=(GLCOLORTABLE)wglGetProcAddress("glColorTable");

	if (glColorTable==NULL)
	{
		MessageBox(NULL, "Function [glColorTable] from
[GL_ARB_imaging] extension not supported!",
			"Pixel Replacement Using Color Tables",
			 MB_OK|MB_ICONSTOP);
		exit(1);
	}

	glPixelStorei(GL_UNPACK_ALIGNMENT,1);
	glClearColor(0,0,0,0);

	//Инвертирующая цветовая таблица
	for(i=0;i<256;++i)
	{
		colorTable[i][0]=255-i;
		colorTable[i][1]=255-i;
		colorTable[i][2]=255-i;
	}

	glColorTable(GL_COLOR_TABLE,GL_RGB,256,
			GL_RGB,GL_UNSIGNED_BYTE,colorTable);
	glEnable(GL_COLOR_TABLE);
}

void deinit()
{
	//Очислитьпамять
	delete image;
}

void reshape(int w, int h)
{
	glViewport(0,0,(GLsizei) w, (GLsizei) h);
	glMatrixMode(GL_PROJECTION);
	glLoadIdentity();
	gluOrtho2D(0,500,0,500);
}

void display()
{
	glClear(GL_COLOR_BUFFER_BIT);
	glRasterPos2i(1,1);
	glDrawPixels(image->sizeX, image->sizeY,
			GL_RGB,GL_UNSIGNED_BYTE,image->data);
	glFlush();
}

int APIENTRY WinMain(HINSTANCE hInstance,
		     HINSTANCE hPrevInstance,
		     LPSTR     lpCmdLine,
		     int       nCmdShow)
{
	char* argv="";
	int argc=0;
```

```
        glutInit(&argc,&argv);
        glutInitDisplayMode(GLUT_SINGLE|GLUT_RGB);
        glutInitWindowSize(540,405);
        glutCreateWindow("Pixel Replacement Using Color Tables");
        init();
        glutDisplayFunc(display);
        glutReshapeFunc(reshape);
        glutMainLoop();
        deinit();
        return 0;
}
```

Замечание: Поскольку загрузка изображения должна производиться способом, зависящим от вашей оконной системы, а также потому, что в GLUT нет подобных функций, в данном примере используется функция **auxDIBImageLoad()** из библиотеки GLAUX для Windows.

Вместо того, чтобы задавать цветовую таблицу непосредственно из вашего приложения, вам может понадобиться использовать в качестве нее изображение, созданное в буфере кадра. Команда **glCopyColorTable()** позволяет вам задать ряд пикселей, которые считываются из буфера кадра и используются в качестве цветовой таблицы.

```
void glCopyColorTable (GLenum target, GLenum internalFormat, GLint x, GLint y,
GLsizei width);
```

Создает цветовую таблицу, используя данные из буфера кадра в качестве данных для цветовой таблицы. Пиксели считываются из текущего буфера для чтения **(GL_READ_BUFFER)** и обрабатываются так же как при вызове **glCopyPixels()**, но без последней конверсии. Учтите, что производятся также операции, заданные **glPixelTransfer*()**. Параметры ***target*** и ***internalFormat*** имеют то же значение, что и для **glColorTable()**. Цветовой массив будет состоять из ***width*** пикселей вправо, начиная с **(*x*, *y*)** буфера кадра.

8.2.5.1.2 Замена всей цветовой таблицы или ее части

Если вам требуется заменить часть цветовой таблицы, вы можете загрузить ее секцию с помощью команд **glColorSubTable()** и **glCopyColorSubTable()**.

```
void glColorSubTable (GLenum target, GLsizei start, GLsizei count, GLenum format,
GLenum type, const GLvoid *data);
```

Заменяет вхождения **[*start*;*start*+*count*-*1*]** цветовой таблица на величины, находящиеся в ***data***. Параметр ***target*** может принимать значения **GL_COLOR_TABLE, GL_POST_CONVOLUTION_COLOR_TABLE, GL_POST_COLOR_MATRIX_COLOR_TABLE.** Параметры ***format*** и ***type*** идентичны соответствующим параметрам **glColorTable()**.

```
void glCopyColorSubTable (GLenum target, GLsizei start, GLint x, GLint y, GLsizei
count);
```

Заменяет вхождения **[*start*;*start*+*count*-*1*]** цветовой таблица на ***count***цветовых величин пикселей из ряда буфера кадра, начинающегося с позиции **(*x*; *y*)**. Пиксели конвертируются в ***internalFormat*** оригинальной таблицы.

8.2.5.1.3 Запрос величин цветовых таблиц

Величины пикселей, сохраненные в цветовой таблице, могут быть получены с помощью команды **glGetColorTable().**

8.2.5.1.4 Проверка цветовых таблиц

Прокси цветовых таблиц позволяют заставить **OpenGL** выяснить, достаточно ли в системе ресурсов для размещения вашей цветовой таблицы. Если **glColorTable()** вызывается с одним из следующих аргументов:

- **GL_PROXY_COLOR_TABLE**
- **GL_PROXY_POST_CONVOLUTION_COLOR_TABLE**
- **GL_PROXY_POST_COLOR_MATRIX_COLOR_TABLE**

OpenGL проверяет, есть ли ресурсы для размещения соответствующей таблицы. Если таблица не влезает, ее величины ***width***, ***format*** и ***type*** устанавливаются в **0.** Если вы хотите выяснить, достаточно ли ресурсов – проверьте одну из этих величин. Например, так:

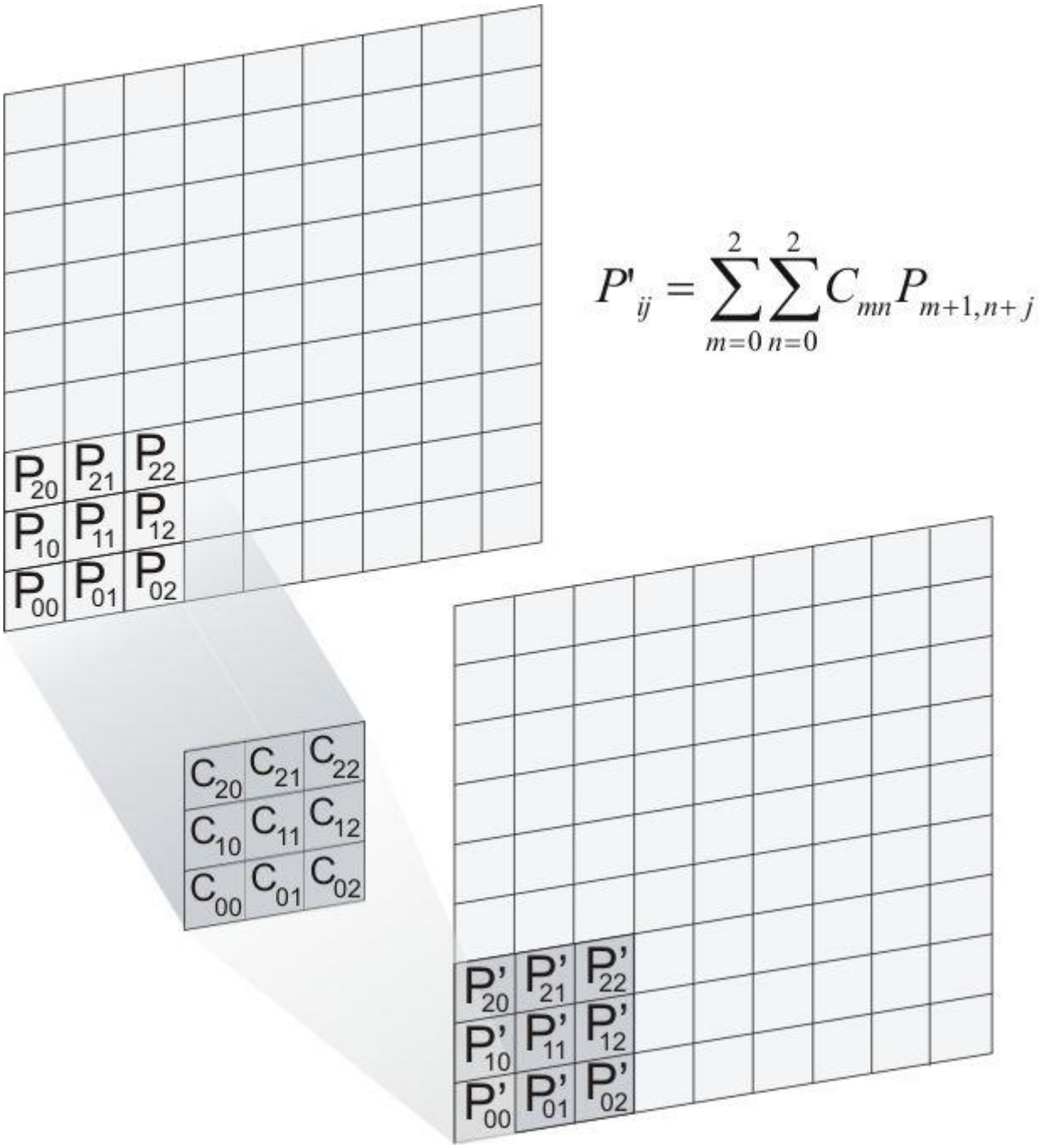
```
glColorTable(GL_PROXY_COLOR_TABLE, GL_RGB, 1024,
             GL_RGB, GL_UNSIGNED_BYTE, NULL);
glGetColorTableParameteriv(GL_PROXY_COLOR_TABLE,
                           GL_COLOR_TABLE_WIDTH, &width);
if(width==0)
      /*цветовая таблица не умещается*/
```

8.2.5.2 Весовые модификаторы

Весовые модификаторы **(convolutions)** работают в качестве фильтров, заменяя каждый пиксель изображения на взвешенное среднее соседствующих с ним пикселей и его самого. Размытие изображения, нахождение ребер и изменение контраста являются примерами применения весовые модификаторы.

На рисунке **8-18** показано, как пиксель P_{00} и соседние с ним обрабатываются весовым модификатором размера **3x3** для получения пикселя $P`_{00}$.

Рисунок **8-17.** Операция пиксельной фильтрации



Весовые модификаторы **–** это массивы весов пикселей**,** и работают они только на **RGBA** пикселях**.** Фильтр**,** также известный как ***kernel***, является простым двумерным массивом весов**.** Каждый пиксель изображения **–** результата создается умножением группы входящих пикселей на весовые коэффициенты фильтра и суммированием получившихся произведений**.** Например**,** на рисунке **8-17** пиксель P'_{00} вычисляется суммированием произведений **9-**ти пикселей из входящего изображения на **9** весовых коэффициентов из фильтра**.**

```
void glConvolutionFilter2D (GLenum target,GLenum internalFormat,GLsizei
width,GLsizei height,
                            GLenum format,GLenum type,const GLvoid *image);
```

Задает двумерный фильтр**.** Параметр ***filter*** должен быть равен **GL_CONVOLUTION_3D.** ***internalFormat*** задает**,** над какими компонентами пикселей будет производиться фильтрация и может принимать одно из **38** значений**,** перечисленных для аргумента ***internalFormat*** команды **glColorTable().** Аргументы ***width*** и ***height*** задают размеры

фильтра в пикселях. Максимальная ширина и высота для фильтров может быть получена с помощью команды **glGetConvolutionParameter*()**. Как и в случае **glDrawPixels()**, аргументы ***format*** и ***type*** задают формат пикселей, сохраненных в массиве, на который указывает ***image***.

Как и для цветовых таблиц, внутренний формат фильтра определяет то, с какими компонентами пикселей требуется работать. Таблица **8-9** описывает, как различные форматы влияют на пиксели. R_s, G_s, B_s и A_s представляют цветовые компоненты исходных пикселей. L_f представляет собой величину светлоты фильтра **GL_LUMINANCE**, а I_f -- величину интенсивности фильтра **GL_INTENSITY**. Наконец, R_f, G_f, B_f и A_f представляют красный, зеленый, синий и альфа компоненты фильтра.

Таблица **8-9.** Влияние фильтров на **RGBA** компоненты пикселей

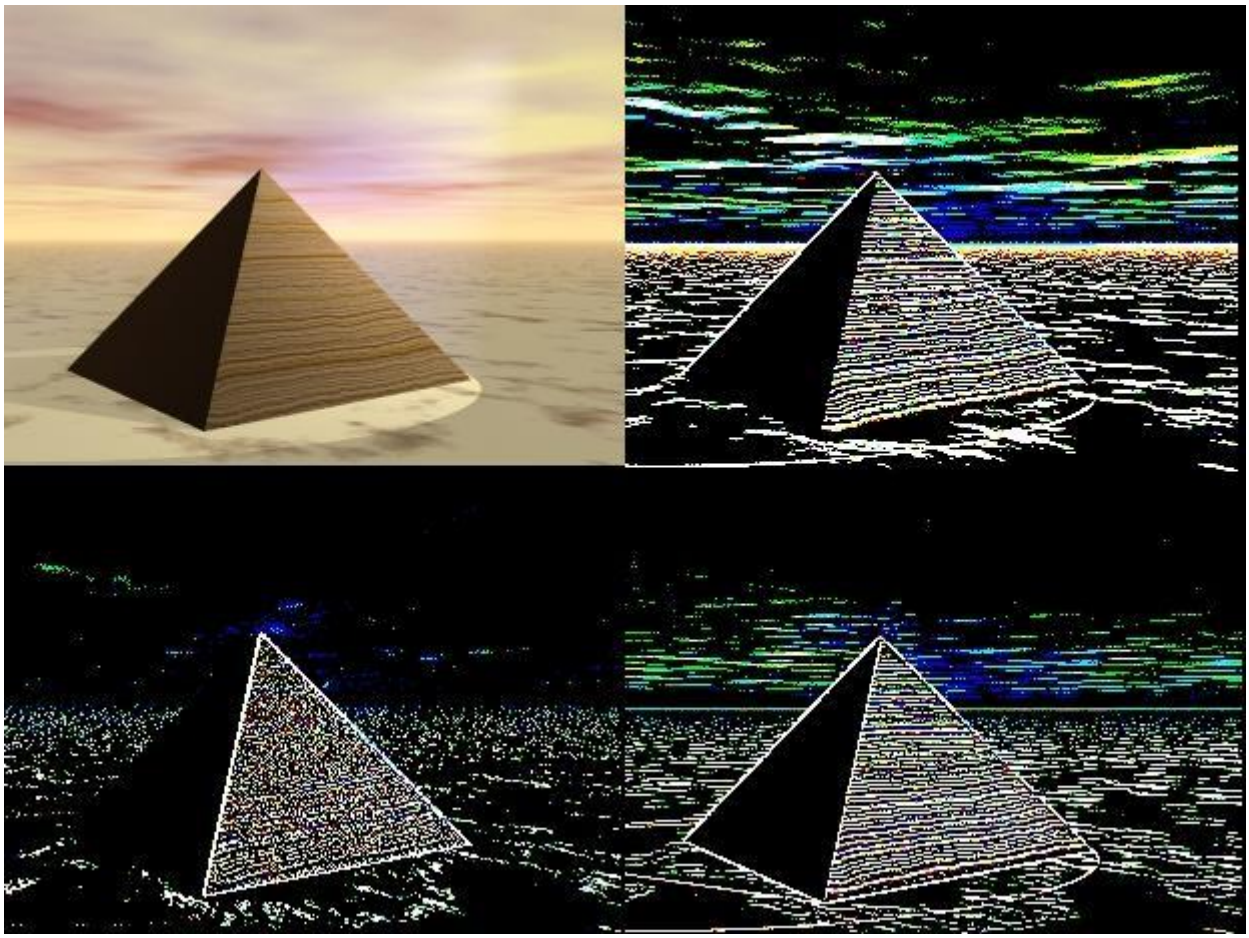
Базовый формат фильтра	**Результат по красному компоненту**	**Результат по зеленому компоненту**	**Результат по синему компоненту**	**Результат по альфа компоненту**
GL_ALPHA	не изменяется	не изменяется	не изменяется	$A_s \cdot A_f$
GL_LUMINANCE	$R_s \cdot L_f$	$G_s \cdot L_f$	$B_s \cdot L_f$	не изменяется
GL_LUMINANCE_ALPHA	$R_s \cdot L_f$	$G_s \cdot L_f$	$B_s \cdot L_f$	$A_s \cdot A_f$
GL_INTENSITY	$R_s \cdot I_f$	$G_s \cdot I_f$	$B_s \cdot I_f$	$A_s \cdot I_f$
GL_RGB	$R_s \cdot R_f$	$G_s \cdot G_f$	$B_s \cdot B_f$	не изменяется
GL_RGBA	$R_s \cdot R_f$	$G_s \cdot G_f$	$B_s \cdot B_f$	$A_s \cdot A_f$

Чтобы активизировать **2D** фильтрацию, используйте **glEnable (GL_CONVOLUTION_2D).**

Пример **8-6** демонстрирует использование нескольких **2D** фильтров **GL_LUMINANCE** размером **3x3** для поиска ребер в **RGB**изображении. Клавиши **'h', 'v'** и **'l'** переключают режимы.

Результат работы программы можно увидеть на рисунке **8-18 (**яркость и контрастность изображений увеличена для наглядности**).** Слева вверху находится исходное изображение пирамиды, справа вверху – то же изображение, обработанное фильтром, позволяющим обнаружить горизонтальные ребра, внизу слева – похожий фильтр, но на этот раз вертикальный, справа внизу – изображение обработанное фильтром Лапласа.

Рисунок **8-18.** Пример обработки изображения двумерным фильтром для обнаружения ребер



Пример **8-6.** Использование двумерных фильтров**:** файл **convolution.cpp**

```
#include <windows.h>
#include <glut.h>
#include <GL/glaux.h>
#include "glext.h"

//Указатель на память, где будет содержаться картинка
AUX_RGBImageRec *image;

//Определить фильтры
GLfloat horizontal[3][3]={{0,-1,0},{0,1,0},{0,0,0}};
GLfloat vertical[3][3]=  {{0,0,0},{-1,1,0},{0,0,0}};
GLfloat laplacian[3][3]= {{-0.125,-0.125,-0.125},
                          {-0.125,1,-0.125},
                          {-0.125,-0.125,-0.125}};

//Тип и указатель для фильтра (если конечно расширение
поддерживается)
typedef void (APIENTRY * GLCONVOLUTIONFILTER2D)
      (GLenum target,GLenum internalFormat,GLsizei width,
       GLsizei height,GLenum format, GLenum type, const GLvoid
*data);

GLCONVOLUTIONFILTER2D glConvolutionFilter2D=NULL;

void init()
{
      //Проверяем, присутствует ли расширение GL_ARB_imaging (imaging
subset)
      if (glutExtensionSupported("GL_ARB_imaging")==0)
```

```
	{
		MessageBox(NULL,"ARB_imaging extension not supported",
		"Using Two-Dimensional Convolution
Filters",MB_OK|MB_ICONHAND);
		exit(1);
	}

	//Получаем указатель на функцию расширения
	glConvolutionFilter2D=(GLCONVOLUTIONFILTER2D)

wglGetProcAddress("glConvolutionFilter2D");

	//Загружаем любую непалитровую картинку BMP
	image=auxDIBImageLoad("peak.bmp");
	glPixelStorei(GL_UNPACK_ALIGNMENT,1);
	glClearColor(0,0,0,0);
}

void deinit()
{
	//Освобождаемпамять
	delete image;
}

void keyboard(unsigned char key, int x, int y)
{
	switch(key)
	{
		case 'h':
			MessageBox(NULL,"Using horizontal filter",
				"Using Two-Dimensional Convolution
Filters",MB_OK);

glConvolutionFilter2D(GL_CONVOLUTION_2D,GL_LUMINANCE,3,3,

GL_LUMINANCE,GL_FLOAT,horizontal);
			glEnable(GL_CONVOLUTION_2D);
			break;
		case 'v':
			MessageBox(NULL,"Using vertical filter",
				"Using Two-Dimensional Convolution
Filters",MB_OK);

glConvolutionFilter2D(GL_CONVOLUTION_2D,GL_LUMINANCE,3,3,

GL_LUMINANCE,GL_FLOAT,vertical);
			glEnable(GL_CONVOLUTION_2D);
			break;
		case 'l':
			MessageBox(NULL,"Using laplacian filter",
				"Using Two-Dimensional Convolution
Filters",MB_OK);

glConvolutionFilter2D(GL_CONVOLUTION_2D,GL_LUMINANCE,3,3,

GL_LUMINANCE,GL_FLOAT,laplacian);
			glEnable(GL_CONVOLUTION_2D);
			break;
		case 'r':
			MessageBox(NULL,"No filter used",
				"Using Two-Dimensional Convolution
Filters",MB_OK);
			glDisable(GL_CONVOLUTION_2D);
```

```
                  break;
            case 27:
                  exit(0);
      }
      glutPostRedisplay();
}

void reshape(int w, int h)
{
      glViewport(0,0,(GLsizei) w, (GLsizei) h);
      glMatrixMode(GL_PROJECTION);
      glLoadIdentity();
      gluOrtho2D(0,540,0,405);
}

void display()
{
      glClear(GL_COLOR_BUFFER_BIT);
      glRasterPos2i(1,1);
      glDrawPixels(image->sizeX,image->sizeY,GL_RGB,
                  GL_UNSIGNED_BYTE,image->data);
      glFlush();
}

int APIENTRY WinMain(HINSTANCE hInstance,
                     HINSTANCE hPrevInstance,
                     LPSTR     lpCmdLine,
                     int       nCmdShow)
{
      char* argv="";
      int argc=0;
      glutInit(&argc,&argv);
      glutInitDisplayMode(GLUT_SINGLE|GLUT_RGB);
      glutInitWindowSize(540,405);
      glutCreateWindow("Using Two-Dimensional Convolution Filters");
      init();
      glutDisplayFunc(display);
      glutReshapeFunc(reshape);
      glutKeyboardFunc(keyboard);
      glutMainLoop();
      deinit();
      return 0;
}
```

Как и в случае с цветовыми таблицами, вы можете задать фильтр с помощью пиксельных величин, взятых из буфера кадра. Команда **glCopyConvolutionFilter2D()** копирует прямоугольник пикселей из текущего буфера для чтения **(GL_READ_BUFFER)** для использования в качестве данных фильтра. Если в качестве внутреннего формата используется **GL_LUMINANCE** или **GL_INTENSITY**, то в качестве величин фильтра берутся красные компоненты пикселей.

```
void glCopyConvolutionFilter2D (GLenum target, GLenum internalFormat, GLint x,
GLint y, GLsizei width, GLsizei height);
```

Задает двумерный фильтр, инициализируя его пикселями из цветового буфера кадра. Аргумент ***target*** должен иметь значение **GL_CONVOLUTION_2D**, а ***internalFormat*** должен быть установлен в одно из значений, допустимых для **glConvolutionFilter2D()**. Прямоугольник пикселей с левым нижним углом в точке **(*x*; *y*)** и размерами ***width*** и ***height*** считывается из буфера кадра и конвертируется в ***internalFormat***.

8.2.5.2.1 Разделяемые двумерные фильтры

Фильтры являются разделяемым, если они могут быть представлены в виде произведения двух одномерных фильтров.

Для создания двух одномерных фильтров, представляющих разделяемый двумерный фильтр, применяется команда **glSeparableFilter2D().** Также как и в случае **glConvolutionFilter2D()** внутренний формат определяет то, как будут обрабатываться пиксели.

```
void glSeparableFilter2D (GLenum target, GLenum internalFormat, GLsizei width,
GLsizei height,
                          GLenum format, GLenum type, const GLvoid *row, const
GLvoid *column);
```

Создает разделяемый двумерный фильтр. ***target*** должен быть установлен в значение **GL_SEPARABLE_2D.** Аргумент ***internalFormat*** может иметь те же значения, что и для команды **glConvolutionFilter2D().** Аргумент ***width*** задает количество пикселей в массиве ***row***. Похожим образом ***height*** задает число пикселей в массиве ***column***. ***type*** и ***format*** так же как в команде **glConvolutionFilter2D()** определяют формат хранения для массивов ***row*** и ***column***.

Чтобы включить обработку пикселей разделяемыми двумерными фильтрами, используйте **glEnable(GL_SEPARABLE_2D).** Если создан разделяемый фильтр, то он будет работать только в случае если активизированы оба режима: **GL_CONVOLUTION_2D** и **GL_SEPARABLE_2D.**

Например, вы можете создать фильтр **3x3**, задав одномерный фильтр **[-1/2, 1, -1/2]** и для ***row***, и для ***column*** разделяемого фильтра **GL_LUMINANCE. OpenGL** вычислит результирующее изображение с использованием двух одномерных фильтров так же, как она вычисляет его с использованием двумерного фильтра, вычислив следующее произведение:

$$\begin{bmatrix} -1/2 \\ 1 \\ -1/2 \end{bmatrix} \cdot \begin{bmatrix} -1/2 & 1 & -1/2 \end{bmatrix} = \begin{bmatrix} 1/4 & -1/2 & 1/4 \\ -1/2 & 1 & -1/2 \\ 1/4 & -1/2 & 1/4 \end{bmatrix}.$$

Использование разделяемых двумерных фильтров более эффективно с точки зрения вычислений, чем использование неразделимого двумерного фильтра.

8.2.5.2.2 Одномерные фильтры

Одномерные фильтры идентичны двумерным за тем исключением, что высота фильтра предполагается равной **1.** Однако одномерные фильтры влияют только на одномерные текстуры.

```
void glConvolutionFilter1D (GLenum target, GLenum internalFormat, GLsizei width,
GLenum format, GLenum type, const GLvoid *image);
```

Задает одномерный фильтр. Аргумент ***target*** должен быть установлен в значение **GL_CONVOLUTION_1D.** Аргумент ***width*** задает количество пикселей в фильтре. Аргументы ***internalFormat***, ***type*** и ***format*** имеют то же значение, что и соответствующие параметры **glConvolutionFilter2D().** ***image*** указывает на одномерное изображение, которое будет использоваться в качестве фильтра.

Для активизации одномерной фильтрации используйте команду **glEnablee(GL_CONVOLUTION_1D).**

Возможно, вам понадобится задать фильтр с помощью величин, взятых из буфера кадра. Команда **glCopyConvolutionFilterr1D()** копирует ряд пикселей из текущего буфера для чтения **(GL_READ_BUFFER),** конвертирует их в заданный внутренний формат и использует их в качестве фильтра.

```
void glCopyConvolutionFilter1D (GLenum target, GLenum internalFormat, GLint x,
GLint y, GLsizei width);
```

Создает одномерный фильтр с помощью пикселей взятых из буфера кадра. **glCopyConvolutionFilter1D()** копирует ***width*** пикселей, начиная с позиции **(*x*, *y*)** и конвертирует их в ***internalFormat*.**

Когда фильтр создан, он может быть смасштабирован и скошен. Масштаб и скос задаются с помощью команды **glConvolutionParameter*()**. Фильтр не приводится к какому-либо диапазону после масштабирования и скоса.

```
void glConvolutionParameter{if} (GLenum target, GLenum pname, TYPE param);
void glConvolutionParameter{if}v (GLenum target, GLenum pname, TYPE *params);
```

Устанавливают параметры, управляющие тем, как производится фильтрация. Аргумент ***target***должен быть установлен в значения **GL_CONVOLUTION_1D**, **GL_CONVOLUTION_2D** или **GL_SEPARABLE_2D**. Аргумент ***pname*** должен принимать одно из значений **GL_CONVOLUTION_BORDER_MODE**, **GL_CONVOLUTION_FILTER_SCALE** или **GL_CONVOLUTION_FILTER_BIAS**. Указание в качестве аргумента ***pname*** значения **GL_CONVOLUTION_BORDER_MODE** устанавливает режим границы фильтрации. В этом случае ***params*** должно иметь значение **GL_REDUCE**, **GL_CONSTANT_BORDER** или **GL_REPLICATE_BORDER**. Если ***pname*** установлен в **GL_CONVOLUTION_FILTER_SCALE** или **GL_CONVOLUTION_FILTER_BIAS**, ***params*** должен указывать на массив из **4**-ех цветовых величин для красного, зеленого, синего и альфа, соответственно.

8.2.5.2.3 Режимы границы

Пиксели изображения, находящиеся на верхней и левой границах, обрабатываются иначе, чем внутренние пиксели. Результат по ним подвергается модификации в зависимости от режима границы фильтра. Существует три варианта фильтрования граничных пикселей:

- режим **GL_REDUCE** приводит к тому, что изображение уменьшается в каждом из направлений. Ширина изображения уменьшается до **(*width*-1)**. Точно так же высота изображения сокращается **(*height*-1)**. Если такое сокращение приводит к тому, что изображение имеет нулевые или отрицательные размеры, выходное изображение не генерируется, но также не генерируется никаких ошибок.
- **GL_CONSTANT_BORDER** вычисляет фильтрованные граничные пиксели с использованием постоянной пиксельной величины для пикселей вне исходного изображения. Постоянная пиксельная величина задается с помощью команды **glConvolutionParameter*()**. Размер результирующего изображения совпадает с размерами исходного.
- **GL_REPLICATE_BORDER** производит вычисления способом аналогичным **GL_CONSTANT_BORDER** за тем исключением, что последний ряд и колонка пикселей используется для пикселей, лежащих вне исходного изображения. Размер результирующего изображения совпадает с размерами исходного.

8.2.5.2.4 Пост фильтрационные операции

После того, как операция фильтрации завершена, пиксели могут быть масштабированы и скошены. Затем они приводятся к диапазону **[0; 1]**. Величины масштаба и скоса задаются командой **glPixelTransfer*()**, с аргументами

GL_POST_CONVOLUTION_*_SCALEили **GL_POST_CONVOLUTION_*_BIAS.** Если задать цветовую таблицу командой **glColorTable()** с аргументом **GL_POST_CONVOLUTION_COLOR_TABLE,** то пиксели после фильтрации будут заменены с использованием этой таблицы.

8.2.5.3 Цветовая матрица

Для конверсии цветовых пространств и линейных преобразований пиксельных величин, подмножество обработки изображений поддерживает стек матриц **4x4,** выбираемый с помощью команды **glMatrixMode(GL_COLOR).** Например, чтобы преобразовать цветовое пространство **RGB** в **CMY (cyan –** голубой, **magenta –** фиолетовый, **yellow –** желтый**)** можно произвести следующие вызовы:

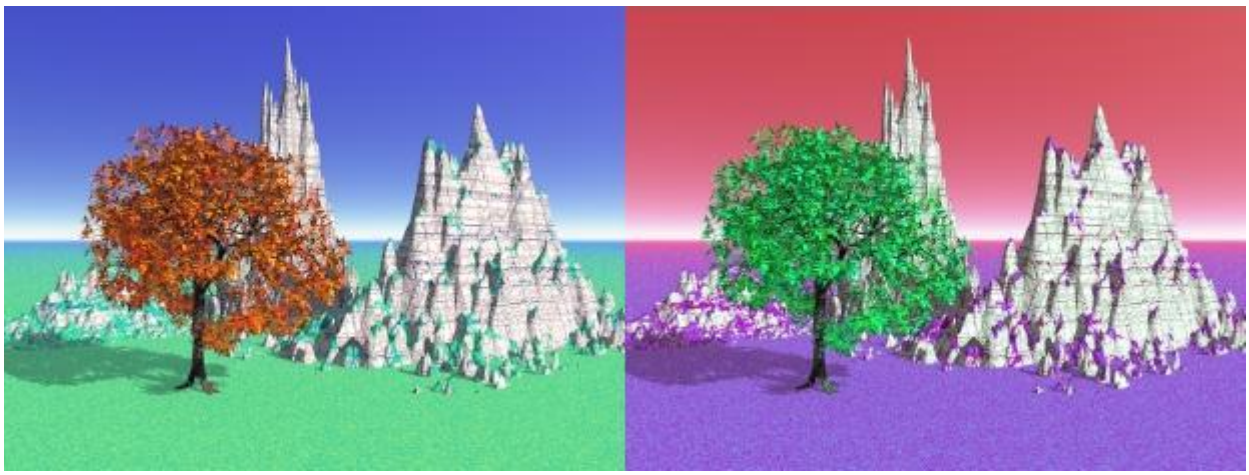
```
GLfloat rgb2cmy[16]= {-1,0,0,0,
                      0,-1,0,0,
                      0,0,-1,0,
                      1,1,1,1};

glMatrixMode(GL_COLOR);      //войти в режим цветовой матрицы
glLoadMatrixf(rgb2cmy);
glMatrixMode(GL_MODELVIEW);  //назад к видовой матрице
```

Замечание: помните о том, что матрицы в OpenGL хранятся по столбцам.

Стек цветовых матриц должен быть достаточного размера для хранения как минимум двух матриц. В отличие от других частей подмножества обработки изображений конверсия с помощью цветовой матрицы производится всегда и не может быть выключена.

Пример **8-7** иллюстрирует использование цветовой матрицы для перемены местами красного и зеленого компонентов изображения. Результат работы программы показан на рисунке **8-19.** Здесь слева изображено исходное изображение, а справа – обработанное с помощью цветовой матрицы.

Рисунок **8-19.** Результат обработки изображения цветовой матрицей, меняющей местами компоненты **R** и **G**



Пример **8-7.** Обмен цветовых компонент с помощью цветовой матрицы: файл **colormatrix.cpp**

```
#include <windows.h>
#include <glut.h>
```

```
#include <glaux.h>
#include "glext.h"

//Указатель на память, где будет содержаться картинка
AUX_RGBImageRec *image;

//Задать цветовую матрицу для изменения порядка цветов в пикселе
//изображения с RGB в GBR
GLfloat m[16]={
      0.0,1.0,0.0,0.0,
      0.0,0.0,1.0,0.0,
      1.0,0.0,0.0,0.0,
      0.0,0.0,0.0,1.0
};

//Флаг показывающий, используем мы матрицу или нет
bool reversing;

void init()
{
      //Проверяем, присутствует ли расширение GL_ARB_imaging (imaging
subset)
      if (glutExtensionSupported("GL_ARB_imaging")==0)
      {
            MessageBox(NULL,"ARB_imaging extension not supported",
            "Using Color Matrix",MB_OK|MB_ICONHAND);
            exit(1);
      }

      //Загружаем любую непалитровую картинку BMP
      image=auxDIBImageLoad("tree.bmp");
      glPixelStorei(GL_UNPACK_ALIGNMENT,1);
      glClearColor(0,0,0,0);

      reversing=false;
}

void deinit()
{
      //Освобождаем память
      delete image;
}

void reshape(int w, int h)
{
      glViewport(0,0,(GLsizei) w, (GLsizei) h);
      glMatrixMode(GL_PROJECTION);
      glLoadIdentity();
      gluOrtho2D(0,542,0,407);
}

void keyboard(unsigned char key,int x,int y)
{
      switch(key)
      {
            case 'r':
                  if (!reversing)
                  {
                        //Для расширения цветовой таблицы
                        //нужен только идентификатор GL_COLOR
                        //при отсутствии этого расширения
                        //код ниже не сработает
                        glMatrixMode(GL_COLOR);
```

```
                        glLoadMatrixf(m);
                        glMatrixMode(GL_MODELVIEW);
                  }
                  else
                  {
                        glMatrixMode(GL_COLOR);
                        glLoadIdentity();
                        glMatrixMode(GL_MODELVIEW);
                  }
                  reversing=!reversing;
                  glutPostRedisplay();
                  break;
            case 27:
                  exit(0);
                  break;
      }
}

void display()
{
      glClear(GL_COLOR_BUFFER_BIT);
      glRasterPos2i(1,1);
      glDrawPixels(image->sizeX,image->sizeY,GL_RGB,
                  GL_UNSIGNED_BYTE,image->data);
      glFlush();
}

int APIENTRY WinMain(HINSTANCE hInstance,
                     HINSTANCE hPrevInstance,
                     LPSTR     lpCmdLine,
                     int       nCmdShow)
{
      char* argv="";
      int argc=0;
      glutInit(&argc,&argv);
      glutInitDisplayMode(GLUT_SINGLE|GLUT_RGB);
      glutInitWindowSize(542,407);
      glutCreateWindow("Exchanging Color Components Using the Color
Matrix");
      init();
      glutDisplayFunc(display);
      glutReshapeFunc(reshape);
      glutKeyboardFunc(keyboard);
      glutMainLoop();
      deinit();
      return 0;
}
```

8.2.5.3.1 Пост матричные операции

Также как и в случае пост – фильтрационных операций, после применения цветовой матрицы пиксели могут быть масштабированы и скошены. Вызовы функции **glPixelTransfer*()** с аргументами **GL_POST_COLOR_MATRIX_*_SCALE** или **GL_POST_COLOR_MATRIX_*_BIAS** позволяют задать параметры масштабирования или скоса соответственно. После масштабирования и скоса пиксели приводятся к диапазону **[0; 1]**.

8.2.5.4 Гистограмма

Используя подмножество обработки изображений, вы можете собирать статистику по изображениям. Гистограмма позволяет увидеть распределение цветовых величин по

изображению, которое, например, может быть использовано для балансировки контраста изображения.

Команда **glHistogram()** позволяет задать, какие компоненты изображения вы хотите использовать для создания гистограммы, а также, хотите ли вы только собрать статистику или продолжить обработку изображения. Чтобы собрать гистограммную статистику, вы должны выполнить команду **glEnable(GL_HISTOGRAMM).**

Для гистограммы, так же как и для цветовых таблиц, прокси механизм доступен для того, чтобы определить достаточно ли в системе ресурсов для хранения требуемой гистограммы. Если ресурсов не хватает, длина гистограммы, ее формат и разрешения компонент устанавливаются в **0.** Вы можете запросить результаты прокси гистограммы, используя команду **glHistogramParameter().**

```
void glHistogram (GLenum target, GLsizei width, GLenum internalFormat, GLboolean
sink);
```

Задает, как должны сохраняться данные гистограммы изображения. Параметр ***target*** должен быть установлен в одно из значений **GL_HISTOGRAM** или **GL_PROXY_HISTOGRAM.** Параметр ***width*** задает количество вхождений в таблицу гистограммы. Его значение должно быть степенью **2.** Параметр ***internalFormat*** определяет, как должны сохраняться данные гистограммы. Допустимыми значениями являются: **GL_ALPHA, GL_ALPHA4, GL_ALPHA8, GL_ALPHA12, GL_ALPHA16, GL_LUMINANCE, GL_LUMINANCE4, GL_LUMINANCE8, GL_LUMINANCE12, GL_LUMINANCE16, GL_LUMINANCE_ALPHA, GL_LUMINANCE4_ALPHA4, GL_LUMINANCE6_ALPHA2, GL_LUMINANCE_8ALPHA8, GL_LUMINANCE12_ALPHA4, GL_LUMINANCE12_ALPHA12, GL_LUMINANCE16_ALPHA16, GL_RGB, GL_RGB2, GL_RGB4, GL_RGB5, GL_RGB8, GL_RGB10, GL_RGB12, GL_RGB16, GL_RGBA, GL_RGBA2, GL_RGBA4, GL_RGB5_A1, GL_RGBA8, GL_RGB10_A2, GL_RGBA12, GL_RGBA16.** В этом списке нет величин **GL_INTENSITY*.** Этот список отличается от величин допустимых для **glColorTable().** Параметр ***sink*** индицирует, должны ли пиксели проходить дальше на этап вычисления минимума*/*максимума или следует их отбросить**..**

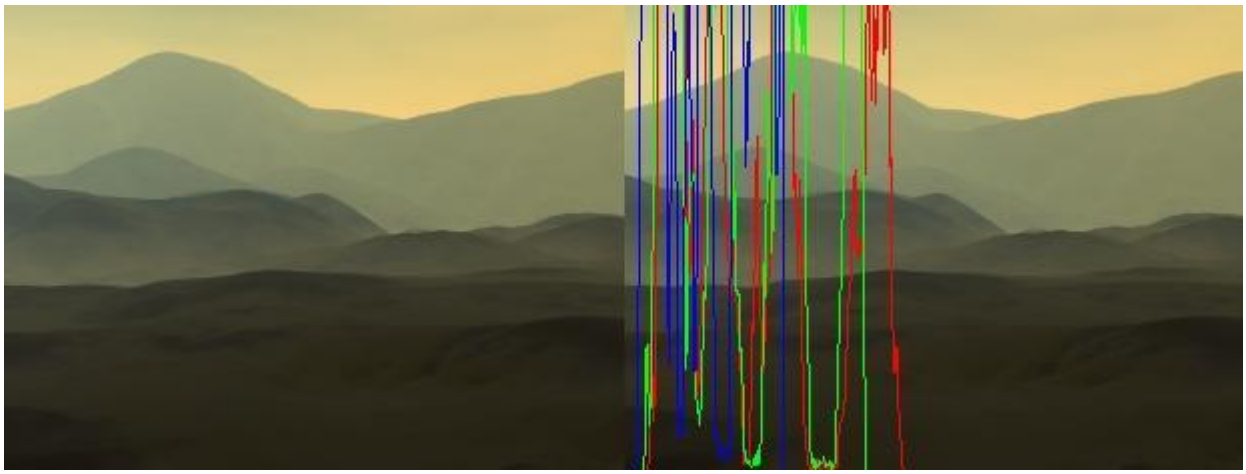
После того, как вы передали пиксели на конвейер с использованием **glDrawPixels()**, вы можете получить результаты гистограммы, используя **glGetHistogram().** Помимо получения значений гистограммы, **glGetHistogram()** может быть использована для сброса внутреннего хранилища гистограммы. Внутреннее хранилище также может быть сброшено с помощью функции **glResetHistogram().**

```
void glGetHistogram (GLenum target, GLboolean reset, GLenum format, GLenum type,
GLvoid *values);
```

Возвращает собранную статистику в виде гистограммы. Параметр ***target*** должен быть установлен в значение **GL_HISTOGRAM.** Параметр ***reset***задает, должно ли внутренне хранилище гистограммы быть сброшено. Параметры ***format*** и ***type*** задают формат хранения массива ***values*** и то, как данные гистограммы должны быть возвращены приложению. Эти аргументы принимают те же значения, что и соответствующие параметры **glDrawPixels().**

В примере **8-8** программа вычисляет гистограмму изображения и выводит результирующее распределение на экран. Клавиша **'s'** влияет на значение параметра ***sink***, контролирующего дальнейший путь пикселей исходного изображения **–** будут ли они пропущены на следующий этап конвейера или отброшены. Рисунок **8-20** демонстрирует исходное изображение **(**слева**),** а также изображение с наложенной на него гистограммой **(**справа**).**

Рисунок **8-20.** Исходное изображение и его гистограмма



Пример **8-8.** Вычисление гистограммы изображения**:** файл **histogram.cpp**

```
#include <windows.h>
#include <glut.h>
#include <glaux.h>
#include "glext.h"

#define HISTOGRAM_SIZE 256

GLboolean sink=GL_FALSE;

//Указатель на память, где будет содержаться картинка
AUX_RGBImageRec *image;

//Тип и указатель для функции определяющей как данные гистограммы
изображения будут сохранены
typedef void (APIENTRY * GLHISTOGRAM)
      (GLenum target,GLsizei width,
       GLenum internalFormat, GLboolean sink);
GLHISTOGRAM glHistogram=NULL;

//Тип и указатель для функции получения статистики по гистограмме
typedef void (APIENTRY * GLGETHISTOGRAM)
      (GLenum target,GLboolean reset, GLenum format,
       GLenum type, const GLvoid *values);
GLGETHISTOGRAM glGetHistogram=NULL;

void init()
{
      //Проверяем, присутствует ли расширение GL_ARB_imaging (imaging
subset)
      if (glutExtensionSupported("GL_ARB_imaging")==0)
      {
            MessageBox(NULL,"ARB_imaging extension not supported",
            " Compute and Diagram an Image's Histogram ",
            MB_OK|MB_ICONHAND);
            exit(1);
      }

      //Получаем указатель на функции расширения
      glHistogram=(GLHISTOGRAM)wglGetProcAddress("glHistogram");

glGetHistogram=(GLGETHISTOGRAM)wglGetProcAddress("glGetHistogram");

      //Загружаем любую непалитровую картинку BMP
      image=auxDIBImageLoad("pyr.bmp");
```

```
	glPixelStorei(GL_UNPACK_ALIGNMENT,1);
	glClearColor(0,0,0,0);

	glHistogram(GL_HISTOGRAM,HISTOGRAM_SIZE,GL_RGB,sink);
	glEnable(GL_HISTOGRAM);
}

void deinit()
{
	//Освобождаем память
	delete image;
}

void keyboard(unsigned char key, int x, int y)
{
	switch(key)
	{
		case 's':
			sink=!sink;

glHistogram(GL_HISTOGRAM,HISTOGRAM_SIZE,GL_RGB,sink);
			glutPostRedisplay();
			break;
		case 27:
			deinit();
			exit(0);
			break;
	}
}

void reshape(int w, int h)
{
	glViewport(0,0,(GLsizei) w, (GLsizei) h);
	glMatrixMode(GL_PROJECTION);
	glLoadIdentity();
	gluOrtho2D(0,540,0,405);
}

void display()
{
	int i;
	GLushort values[HISTOGRAM_SIZE][3];

	glClear(GL_COLOR_BUFFER_BIT);
	glRasterPos2i(1,1);
	glDrawPixels(image->sizeX,image->sizeY,GL_RGB,
		     GL_UNSIGNED_BYTE,image->data);

glGetHistogram(GL_HISTOGRAM,GL_TRUE,GL_RGB,GL_UNSIGNED_SHORT,values);

	//Plot histogram
	glBegin(GL_LINE_STRIP);
		glColor3f(1.0,0.0,0.0);
		for(i=0;i<HISTOGRAM_SIZE;i++)
			glVertex2s(i,values[i][0]);
	glEnd();
	glBegin(GL_LINE_STRIP);
		glColor3f(0.0,1.0,0.0);
		for(i=0;i<HISTOGRAM_SIZE;i++)
			glVertex2s(i,values[i][1]);
	glEnd();
	glBegin(GL_LINE_STRIP);
		glColor3f(0.0,0.0,1.0);
```

```
            for(i=0;i<HISTOGRAM_SIZE;i++)
                  glVertex2s(i,values[i][2]);
      glEnd();
      glFlush();
}


int APIENTRY WinMain(HINSTANCE hInstance,
                     HINSTANCE hPrevInstance,
                     LPSTR     lpCmdLine,
                     int       nCmdShow)
{
      char* argv="";
      int argc=0;
      glutInit(&argc,&argv);
      glutInitDisplayMode(GLUT_SINGLE|GLUT_RGB);
      glutInitWindowSize(540,405);
      glutCreateWindow("Compute and Diagram an Image's Histogram");
      init();
      glutDisplayFunc(display);
      glutReshapeFunc(reshape);
      glutKeyboardFunc(keyboard);
      glutMainLoop();
      deinit();
      return 0;
}
```

Команда **glResetHistogram()** сбрасывает гистограмму без извлечения ее данных.

```
void glResetHistogram (GLenum target);
```

Сбрасывает счетчики гистограммы в **0.** Параметр ***target*** должен быть установлен в **GL_HISTOGRAM.**

8.2.5.5 Минимум и максимум

Команда **glMinmax()** вычисляет минимальное и максимальное значения компонентов пикселя из пиксельного прямоугольника. Как и в случае **glHistogram()** после вычисления минимальной и максимальной величин вы можете либо передать пиксели дальше по конвейеру, либо прекратить процесс визуализации.

```
void glMinmax (GLenum target, GLenum internalFormat, GLboolean sink);
```

Вычисляет минимальное и максимальное значения пикселей изображения. Аргумент ***target*** должен быть установлен в **GL_MINMAX.** ***internalFormat*** задает для каких компонент должны быть вычислены значения минимума и максимума. Допустимыми значениями для ***internalFormat*** команды **glMinmax()** являются те же, что и для команды **glHistogram().** Если аргумент ***sink*** установлен в **GL_TRUE,** то после вычисления минимума и максимума путь пикселей по конвейеру прекращается, иначе их обработка продолжается.

Для получения вычисленных значений используется команда **glGetMinmax().** Так же как в случае **glGetHistogram()** внутренние значения могут быть сброшены после их получения.

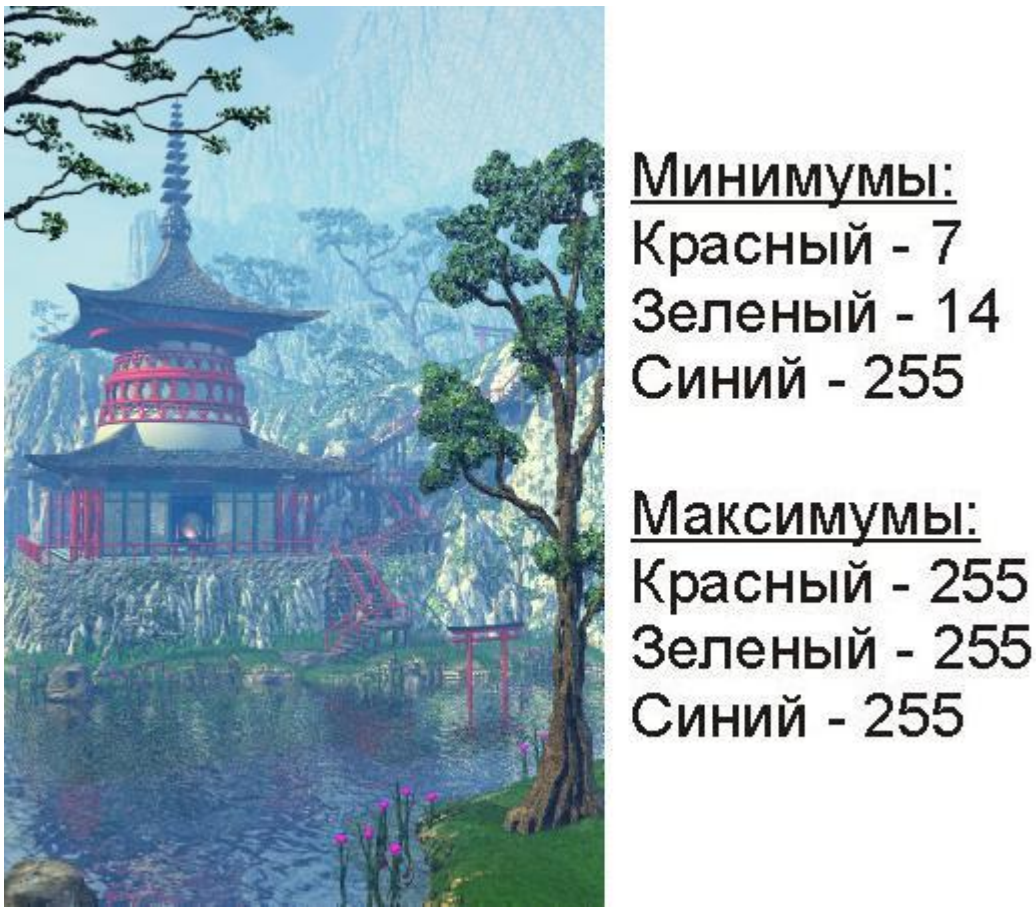
```
void glGetMinmax (GLenum target, GLboolean reset, GLenum format, GLenum type,
GLvoid *values);
```

Возвращает результаты вычисления минимума и максимума. Аргумент ***target*** должен быть установлен в **GL_MINMAX.** Если аргумент ***reset*** равен **GL_TRUE,** значения минимума и максимума сбрасываются в свои начальные значения. Аргументы ***format*** и ***type*** задают формат значений возвращаемых в массиве ***values*** и могут принимать те же значения, что и аналогичные параметры команды **glDrawPixels().**

Пример **8-9** демонстрирует использование команды **glMinMax()** для вычисления минимального и максимального значений компонент пикселей в формате **GL_RGB.** Пример исходного изображения и результатов работы программы можно увидеть на рисунке **8-21.** Операция вычисления минимума и максимума должна быть активизирована с использованием команды **glEnable(GL_MINMAX).**

В массиве, возвращенном командой **glGetMinmax()** первыми элементами идут вычисленные минимумы для всех трех цветовых компонент, а следом за ними – все максимумы.

Рисунок **8-21.** Изображение и экстремумы цветовых компонент его пикселей



Пример **8-9.** Вычисление минимума и максимума пиксельных величин: файл **minmax.cpp**

```
#include <windows.h>
#include <glut.h>
#include <glaux.h>
#include <stdio.h>
#include "glext.h"

//Указатель на память, где будет содержаться картинка
AUX_RGBImageRec *image;
```

```
typedef void (APIENTRY * GLMINMAX) GLenum target, GLenum
internalFormat, GLboolean sink);
GLMINMAX glMinmax=NULL;
typedef void (APIENTRY * GLGETMINMAX) GLenum target,GLboolean reset,
GLenum format,GLenum type, const GLvoid *values);
GLGETMINMAX glGetMinmax=NULL;

void init()
{
      //Проверяем, присутствует ли расширение GL_ARB_imaging (imaging
subset)
      if (glutExtensionSupported("GL_ARB_imaging")==0)
            exit(1);

      glMinmax=(GLMINMAX)wglGetProcAddress("glMinmax");
      glGetMinmax=(GLGETMINMAX)wglGetProcAddress("glGetMinmax");

      //Загружаем любую непалитровую картинку BMP
      image=auxDIBImageLoad("tree.bmp");
      glPixelStorei(GL_UNPACK_ALIGNMENT,1);
      glClearColor(0,0,0,0);
      glMinmax(GL_MINMAX,GL_RGB,GL_FALSE);
      glEnable(GL_MINMAX);
}

void deinit()
{
      //Освобождаем память
      delete image;
}

void reshape(int w, int h)
{
      glViewport(0,0,(GLsizei) w, (GLsizei) h);
      glMatrixMode(GL_PROJECTION);
      glLoadIdentity();
      gluOrtho2D(0,540,0,405);
}

void display()
{
      GLubyte values[6];
      glClear(GL_COLOR_BUFFER_BIT);
      glRasterPos2i(1,1);
      glDrawPixels(image->sizeX,image->sizeY,
                  GL_RGB,GL_UNSIGNED_BYTE,image->data);
      glGetMinmax(GL_MINMAX,GL_TRUE,GL_RGB,GL_UNSIGNED_BYTE,values);
      glFlush();

      printf("Red  : min = %d max = %d\n",values[0],values[3]);
      printf("Green: min = %d max = %d\n",values[1],values[4]);
      printf("Blue : min = %d max = %d\n",values[3],values[5]);
}

int main(int argc,char* argv[])
{
      glutInit(&argc,argv);
      glutInitDisplayMode(GLUT_SINGLE|GLUT_RGB);
      glutInitWindowSize(540,405);
      glutCreateWindow("Computing Minimum and Maximum Pixel Values");
      init();
      glutDisplayFunc(display);
      glutReshapeFunc(reshape);
```

```
        glutMainLoop();
        deinit();
        return 0;
}
```

Несмотря на то, что команда **glGetMinmax()** позволяет сбрасывать величины минимума и максимума в их начальные значений, вы также можете сделать это в любой момент при помощи команды **glResetMinmax()**.

```
void glResetMinmax (GLenum target);
```

Сбрасывает величины минимума и максимума в их начальные значения. Аргумент ***target*** должен быть установлен в значение **GL_MINMAX.**

8.2.5.6 Комбинирование пикселей с использованием уравнений наложения

При использовании цветового наложения цвета в буфере кадра складываются с поступающими фрагментами, создавая новые цвета в буфере кадра. Подмножество обработки изображений расширяет возможности цветового наложения, определяя дополнительные математические комбинации. С помощью команды **glBlendEquation()** можно указать, как модифицировать операцию, используемую для цветового наложения входящих фрагментов на цвета в буфере кадра.

```
void glBlendEquation (GLenum mode);
```

Задает как цвета в буфере кадра и цвета источника должны соединяться. Допустимыми значениями для ***mode*** являются **GL_FUNC_ADD, GL_FUNC_SUBTRACT, GL_FUNC_REVERSE, GL_MIN** и **GL_MAX.** Значением ***mode*** по умолчанию является **GL_FUNC_ADD (**сложение**),** которое, кроме того, является единственно возможным при отсутствии подмножества обработки изображений. Влияние различных режимов на результат цветового наложения отражен в таблице **8-10.**

В таблице **8-10** C_s и C_d представляют цвета источника и приемника, параметры **S** и **D** в таблице представляют факторы наложения источника и приемника, чье значение определяется в соответствии с аргументами команды **glBlendFunc()**.

Таблица **8-10.** Математические операции, связанные с режимами наложения подмножества обработки изображений

Режим наложения	Математическая операция
GL_FUNC_ADD	$C_s S + C_d D$
GL_FUNC_SUBTRACT	$C_s S - C_d D$
GL_FUNC_REVERSE_SUBTRACT	$C_d D - C_s S$
GL_MIN	$\min(C_s S, C_d D)$
GL_MAX	$\max(C_s S, C_d D)$

В примере **8-10** демонстрируются различные уравнения наложения. Для выбора режима наложения используются клавиши **‘a’, ’s’, ’r’, ’m’, ’x’.** В качестве источника используется синий квадрат, в качестве приемника используется желтый квадрат на заднем фоне. Фактор наложения для обоих цветов установлен в **GL_ONE**командой **glBlendFunc().** Результирующее изображение во всех режимах вы можете увидеть на рисунке **8-22.**

Рисунок **8-22.** Различные уравнения цветового наложения

Пример **8-10.** Демонстрация уравнений наложения: файл **blendeqn.cpp**

```
/*   'a' -> GL_FUNC_ADD
 *   's' -> GL_FUNC_SUBTRACT
 *   'r' -> GL_FUNC_REVERSE_SUBTRACT
 *   'm' -> GL_MIN
 *   'x' -> GL_MAX
 */

#include <glut.h>
#include "glext.h"

//Тип и указатель для функции смешивающего уравнения
typedef void (APIENTRY * GLBLENDEQUATION) (GLenum mode);
GLBLENDEQUATION glBlendEquation=NULL;

void init()
{
      //Проверяем, присутствует ли расширение GL_ARB_imaging (imaging
subset)
      if (glutExtensionSupported("GL_ARB_imaging")==0)
      {
            exit(1);
      }

      //Получить указатель на функцию

glBlendEquation=(GLBLENDEQUATION)wglGetProcAddress("glBlendEquation");

      glClearColor(1.0,1.0,0.0,0.0);
      glBlendFunc(GL_ONE,GL_ONE);
      glEnable(GL_BLEND);
}

void keyboard(unsigned char key, int x, int y)
{
      switch(key)
      {
            case 'a': case 'A':
                  glBlendEquation(GL_FUNC_ADD);
                  break;
            case 's': case 'S':
                  glBlendEquation(GL_FUNC_SUBTRACT);
                  break;
            case 'r': case 'R':
                  glBlendEquation(GL_FUNC_REVERSE_SUBTRACT);
                  break;
            case 'm': case 'M':
                  glBlendEquation(GL_MIN);
                  break;
            case 'x': case 'X':
```

```
                  glBlendEquation(GL_MAX);
                  break;
            case 27:
                  exit(0);
      }
      glutPostRedisplay();
}

void reshape(int w, int h)
{
      glViewport(0,0,(GLsizei) w, (GLsizei) h);
      glMatrixMode(GL_PROJECTION);
      glLoadIdentity();
      gluOrtho2D(-1,1,-1,1);
}

void display()
{
      glClear(GL_COLOR_BUFFER_BIT);
      glColor3f(0.0,0.0,1.0);
      glRectf(-0.5,-0.5,0.5,0.5);
      glFlush();
}

int main (int argc, char **argv)
{
      glutInit(&argc,argv);
      glutInitDisplayMode(GLUT_SINGLE|GLUT_RGB);
      glutInitWindowSize(500,500);
      glutCreateWindow("Demonstrating The Blend Equation Modes");
      init();
      glutDisplayFunc(display);
      glutReshapeFunc(reshape);
      glutKeyboardFunc(keyboard);
      glutMainLoop();
      return 0;
}
```

8.2.5.7 Постоянные факторы наложения

Подмножество обработки изображений предоставляет дополнительные факторы наложения**,** которые могут быть использованы в команде **glBlendFunc().** Эти дополнительные факторы описаны в таблице **8-11.**

Таблица **8-11.** Факторы наложения**,** предоставляемые подмножеством обработки изображений

Константа	**Принадлежность**	**Вычисляемый фактор**
GL_CONSTANT_COLOR	источник или приемник	(R_c, G_c, B_c, A_c)
GL_ONE_MINUS_CONSTANT_COLOR	источник или приемник	$(1,1,1,1) - (R_c, G_c, B_c, A_c)$
GL_CONSTANT_ALPHA	источник или приемник	(A_c, A_c, A_c, A_c)
GL_ONE_MINUS_CONSTANT_ALPHA	источник или приемник	$(1,1,1,1) - (A_c, A_c, A_c, A_c)$

Глава 9. Текстурирование

До сих пор мы рисовали все геометрические объекты либо одним цветом, либо с плавной закраской (когда внутренние цвета рассчитываются интерполяцией цветов в вершинах) – то есть объекты рисовались без текстурирования. Если вы хотите нарисовать массивную кирпичную стену без текстурирования, каждый кирпичик должен быть нарисован в виде отдельного полигона. Без текстурирования такая стена может потребовать тысяч полигонов, хотя сама она является прямоугольной, и даже тогда кирпичи будут выглядеть слишком гладкими и одинаковыми, то есть недостаточно реалистичными.

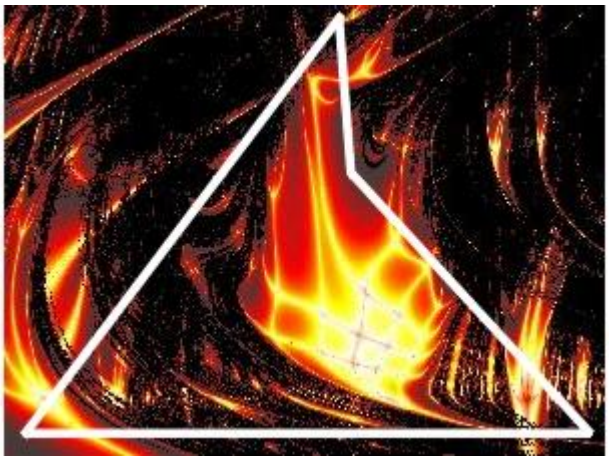
Текстурирование (или наложение текстур) позволяет прикрепить изображение кирпичной стены (полученное, возможно, сканированием реальной фотографии) к полигону и нарисовать всю стену в виде одного полигона. При использовании текстурирования можно быть уверенным в том, что с полигоном в процессе всех преобразований будут происходить правильные изменения. Например, если вы наблюдаете стену в перспективе, кирпичи, находящиеся дальше от вас, будут выглядеть меньше тех, что ближе. Другие примеры использования текстур включают симуляцию растительности на полигонах, представляющих землю; наложение обоев на стены или придание объектам вида физических материалов, например, мрамора, дерева или гранита. Хотя наиболее естественным является наложение текстур на полигоны, текстуру можно наложить на любой примитив – точку, линию, полигон, битовую карту или изображение.

Поскольку существует так много возможностей, текстурирование – это большой сложный предмет, и при работе с ним вам несколько раз придется делать выбор техники программирования. Например, большинство людей интуитивно понимают смысл двумерных текстур, но текстуры могут также быть одномерными и даже трехмерными. Вы можете накладывать текстуры на поверхности, созданные из нескольких полигонов, и на изогнутые поверхности, вы также можете повторять текстуру в одном, двух или трех направлениях (в зависимости от того в каком числе направлений текстура описана). Кроме того вы можете накладывать текстуру таким образом, чтобы она выделяла контуры других свойств рассматриваемого объекта. Блестящие объекты могут быть текстурированы таким образом, как будто они находятся в центре комнаты и отражают изображения окружающих предметов. Наконец, текстура может быть наложена на объект различными способами. Она может быть нарисована на объекте, использоваться для модуляции его цветов или комбинироваться с ними.

Текстуры – это просто прямоугольные массивы данных, например, цветовых, световых или цветовых и альфа. Индивидуальные элементы (значения) текстуры часто называются *тэкселями* (***texels***). Что делает текстурирование сложным, так это то, что прямоугольная текстура может быть наложена на непрямоугольный объект, и это должно быть сделано каким-либо разумным способом.

Рисунок **9-1** иллюстрирует процесс текстурирования. Левая часть рисунка представляет собой саму текстуру, а белая линия – четырехугольную форму, углы которой будут наложены на соответствующие точки текстуры. Когда четырехугольник отображается на экране, он может быть искажен за счет различных преобразований – поворотов, переносов, масштабирования и проецирования. Правая часть рисунка демонстрирует, как выглядит четырехугольник на экране после этих преобразований. (Заметьте, что четырехугольник является вогнутым и без предварительной тесселяции может быть неверно отображен **OpenGL**.)

Рисунок **9-1**. Процесс текстурирования



Обратите внимание на то, что текстура изменяется, чтобы соответствовать искажениям четырехугольника. В данном случае она слегка растянута в направлении **x**и сжата в направлении **y**. В зависимости от размера текстуры, степени искажения четырехугольника и размера изображения на экране. некоторые тэксели могут быть наложены более чем на один фрагмент, а некоторые фрагменты могут покрываться несколькими тэкселями. Поскольку текстура состоит из целого числа тэкселей (в данном случае **256x256**), для наложения тэкселей на фрагменты следует выполнить операцию фильтрации. Например, если несколько тэкселей соответствуют одному фрагменту, следует взять их среднее, которое уместится на фрагменте, если один тэксель попадает на несколько фрагментов, следует вычислить взвешенное среднее соседних тэкселей. Из-за этих расчетов текстурирование -- довольно дорогая операция с точки зрения объема расчетов, поэтому многие аппаратные средства включают встроенную поддержку наложения текстур.

Приложение может создавать текстурные объекты, каждый из которых содержит одну текстуру (и, возможно, связанные с ней *мипмапы* (***mipmaps***)). Некоторые реализации **OpenGL** поддерживают *рабочее подмножество* (***workingset***) текстурных объектов, которые работают быстрее, чем другие. Говорят, что эти объекты являются *резидентными* (***resident***) и могут иметь специальную аппаратную и*/*или программную поддержку. С помощью **OpenGL** вы можете создавать и уничтожать текстурные объекты, а также выяснять, какие из текстур могут быть размещены в рабочем подмножестве.

Несколько операций по наложению текстур появились в **OpenGL** версии **1.1**:

- Дополнительные внутренние форматы изображений текстуры.
- Текстурное прокси, позволяющее заранее выяснить, уместится ли нужная текстура в памяти.
- *Частичные текстуры* (***subtextures***), позволяющие заменять все изображение существующей текстуры или его часть. До их появления нужно было целиком удалить и создать текстуру заново.
- Создание текстуры из данных буфера кадра (а также из системной памяти)
- Объекты текстуры, включая резиденты и приоритеты.

Версия **1.2** добавила еще несколько операций, связанных с текстурированием:

- **3D** текстуры.
- Новый режим вычисления текстурных координат **GL_CLAMP_TO_EDGE**, отделяющий тэксели от ребра изображения текстуры, а не от его границы.
- Больший контроль над мипмапами, представляющими разные уровни детализации (**levels-of-details -- LOD**).
- Вычисление зеркального блика (от источника) после текстурирования.

Версия **1.2** также позволила производителям добавлять стандартизованные **(**принятые **ARB)** опциональные расширения**,** включая**:**

- Мультитекстурирование**,** позволяющее накладывать на один примитив несколько текстур**.**

Если вы пытаетесь использовать одну из перечисленных операций и не можете сделать этого**,** проверьте версию вашей реализации **OpenGL,** чтобы убедиться в том**,** что она действительно поддерживает требуемую операцию**.**

9.1 Введение и пример

В данном разделе перечисляются шаги**,** которые необходимо выполнить для наложения текстуры**.** В нем также приводится относительно простая программа**,** производящая наложение текстуры**.**

9.1.1 Наложение текстуры по шагам

Чтобы использовать наложение текстуры**,** вы должны выполнить следующие шаги**:**

1. Создать текстурный объект и задать текстуру для него
2. Задать**,** как текстура должна воздействовать на каждый пиксель
3. Активизировать механизм текстурирования
4. Нарисовать сцену**,** передавая на конвейер визуализации и геометрические координаты и координаты текстуры

Имейте в виду**,** что текстурирование работает только в **RGBA** режиме**.** Результат попытки применения текстурирования в индексном режиме не определен**.**

9.1.1.1 Создание текстурного объекта и указание текстуры для него

Обычно считается**,** что текстура является двумерной**,** как и большинство изображений**,** однако она может быть и одномерной**,** и трехмерной**.** Данные**,** описывающие текстуру могут состоять из одного**,** двух**,** трех или четырех элементов на тэксель и представлять все**,** что угодно от константы модуляции до четверки **(R**, **G**, **B**, **A)**.

В довольно простом примере **9-1** создается один текстурный объект**,** управляющий одной двумерной текстурой**.** Данный пример предварительно не выясняет**,** сколько имеется доступной памяти**.** Поскольку создается только одна текстура**,** не делается никаких попыток по оптимизации работы программы **(**нет приоритезации или другой работы с рабочим подмножеством текстур**).** Другие более сложные техники**,** такие как границы текстуры и мипмапы**,** в этом простом примере также не используются**.**

9.1.1.2 Задать способ**,** которым текстура должна накладываться на каждый пиксель

Вы можете выбрать любую из **4** функций для вычисления финальной **RGBA** величины на основании фрагмента и текстуры**.** Одна из возможностей заключается в использовании цвета текстуры в качестве результирующего цвета **(**этот режим называется режимом *замены* **–** ***replace***; текстура просто рисуется поверх фрагмента**)** **(**Этот режим как раз и используется в примере **9-1.)** Другой метод заключается в использовании текстуры для *модуляции* **(*modulate*)** или масштабирования цвета фрагмента**;** эта техника полезна при комбинировании эффектов освещения и текстурирования**.** Наконец**,** на основании величины текстуры на фрагмент может быть наложена константная величина цвета**.**

9.1.1.3 Активизация наложения текстур

До того, как вы нарисуете свою сцену, вам следует включить текстурирование. Текстурирование включается и выключается с использованием команд **glEnable()** и **glDisable()** с символическими константами **GL_TEXTURE_1D**, **GL_TEXTURE_2D** или **GL_TEXTURE_3D** в качестве аргумента для активизации одномерного, двумерного или трехмерного текстурирования соответственно. **(**Если активизированы два или все три режима, используется режим с большим числом измерений. Для лучшей вразумительности программы следует использовать только один режим.**)**

9.1.1.4 Нарисовать сцену, поставляя и текстурные, и геометрические координаты

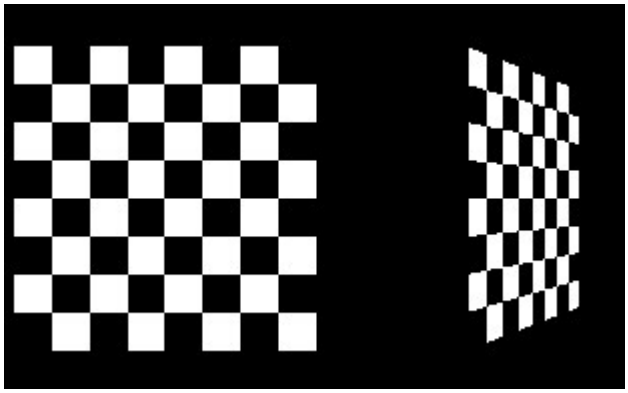
Вам нужно указать, как текстура должна быть выровнена относительно фрагментов до того, как она будет к ним прикреплена. То есть во время передачи объекта на конвейер, вам нужно поставлять не только геометрические, но и текстурные координаты. Для двумерной карты текстуры, например, координаты текстуры лежат в диапазоне **[0; 1]** в обоих направлениях, однако координаты текстурируемых элементов могут быть любыми. Например, чтобы наложить кирпичную текстуру на стену, предположим, что стена имеет квадратную форму и что должна использоваться только одна копия текстуры. Тогда в коде вам, вероятно, потребуется ассоциировать текстурные координаты **(0,0) (1, 0) (1, 1)** и **(1, 1)** с четырьмя углами стены. Если стена весьма велика, вы, возможно, захотите нарисовать на ней несколько копий текстуры. Чтобы сделать это, текстура должна быть создана таким образом, чтобы кубики на левой границе точно подходили к кубикам на правой границе, и то же должно относиться к кубикам на верхней и нижней границах.

Вы также должны определиться с тем, как будут трактоваться координаты текстуры за пределами диапазона **[0; 1]**. Должна ли текстура повторяться, чтобы накрыть объект, или координаты следует отсечь по допустимой границе.

9.1.2 Простая программа

Одной из проблем с примерами, демонстрирующими текстуры, является то, что интересные текстуры, как правило, велики. Обычно, текстуры считываются из файлов изображений, поскольку программное создание текстуры может вылиться в сотни строк кода. В примере **9-1** текстура, которая состоит из чередующихся белых и черных квадратиков, подобно шахматной доске, создается программно. Программа накладывает эту текстуру на два квадрата, которые затем выводятся в перспективе, при этом один из них перпендикулярен направлению обзора, а второй повернут на **45** градусов, как показано на рисунке **9-2.** В объектных координатах оба квадрата имеют одинаковый размер.

Рисунок **9-2.** Текстурированные квадраты



Пример **9-1.** Текстурированная шахматная доска**:** файл **checker.cpp**

```
#include <glut.h>

//Параметры текстуры шахматной доски
#define checkImageWidth 64
#define checkImageHeight 64

GLubyte checkImage[checkImageHeight][checkImageWidth][4];
GLuint texName;

void makeCheckImage()
{
int i,j,c;

for (i=0;i<checkImageHeight;i++)
{
for (j=0;j<checkImageWidth;j++)
{
c=(((i&0x8)==0)^((j&0x8)==0))*255;
checkImage[i][j][0]=(GLubyte)c;
checkImage[i][j][1]=(GLubyte)c;
checkImage[i][j][2]=(GLubyte)c;
checkImage[i][j][3]=(GLubyte)255;
}
}
}

void init(void)
{
glClearColor(0.0,0.0,0.0,0.0);
glShadeModel(GL_FLAT);
glEnable(GL_DEPTH_TEST);

makeCheckImage();
glPixelStorei(GL_UNPACK_ALIGNMENT,1);

glGenTextures(1,&texName);
glBindTexture(GL_TEXTURE_2D,texName);

glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_WRAP_S,GL_REPEAT);
glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_WRAP_T,GL_REPEAT);
glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MAG_FILTER,GL_NEAREST);
glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MIN_FILTER,GL_NEAREST);
glTexImage2D(GL_TEXTURE_2D,0,GL_RGBA,checkImageWidth,checkImageHeight,
0,GL_RGBA,GL_UNSIGNED_BYTE,checkImage);
}

void display(void)
{
glClear(GL_COLOR_BUFFER_BIT|GL_DEPTH_BUFFER_BIT);
glEnable(GL_TEXTURE_2D);
glTexEnvf(GL_TEXTURE_ENV,GL_TEXTURE_ENV_MODE,GL_REPLACE);
glBindTexture(GL_TEXTURE_2D,texName);

glBegin(GL_QUADS);
glTexCoord2f(0.0,0.0); glVertex3f(-2.0,-1.0,0.0);
glTexCoord2f(0.0,1.0); glVertex3f(-2.0,1.0,0.0);
glTexCoord2f(1.0,1.0); glVertex3f(0.0,1.0,0.0);
glTexCoord2f(1.0,0.0); glVertex3f(0.0,-1.0,0.0);

glTexCoord2f(0.0,0.0); glVertex3f(1.0,-1.0,0.0);
glTexCoord2f(0.0,1.0); glVertex3f(1.0,1.0,0.0);
```

```
glTexCoord2f(1.0,1.0); glVertex3f(2.41421,1.0,-1.41421);
glTexCoord2f(1.0,0.0); glVertex3f(2.41421,-1.0,-1.41421);
glEnd();

glFlush();
glDisable(GL_TEXTURE_2D);
}

void reshape(int w, int h)
{
glViewport(0,0,(GLsizei) w, (GLsizei) h);
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
gluPerspective(60.0,(GLfloat)w/(GLfloat)h,1.0,30.0);
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
glTranslatef(0.0,0.0,-3.6);
}

int main(int argc, char ** argv)
{
glutInit(&argc,&argv);
glutInitDisplayMode(GLUT_SINGLE|GLUT_RGB|GLUT_DEPTH);
glutInitWindowSize(250,250);
glutCreateWindow("Texture-Mapped Checkerboard");
init();
glutDisplayFunc(display);
glutReshapeFunc(reshape);
glutMainLoop();
return 0;
}
```

Текстура в виде шахматной доски генерируется в функции **makeCheckImage()**, а вся инициализация текстурирования производится в функции **init()**. Команды **glGenTextures()** и **glBindTexture()** именуют и создают текстурный объект для изображения текстуры. Единственная карта текстуры с полным разрешением задается командой **glTexImage2D()**, чьи параметры индицируют размер, тип, расположение и другие параметры изображения текстуры.

Четыре вызова команды **glTexParameter*()** задают, как текстура должна быть наложена, и как должны фильтроваться цвета, если количество тэкселей в текстуре и пикселей на экране точно не совпадают.

В функции **display()** команда **glEnable()** включает текстурирование. Команда **glTexEnv*()** устанавливает режим рисования в **GL_REPLACE**, чтобы текстурированные полигоны рисовались с использованием цветов карты текстуры (без принятия в расчет того цвета, которым они рисовались бы без текстурирования).

Затем рисуются два полигона. Заметьте, что координаты текстуры задаются вместе с координатами вершин. Команда **glTexCoord*()** ведет себя аналогично **glNormal*()**. Команда **glTexCoord*()** устанавливает текущие координаты текстуры; эти координаты ассоциируются со всеми последующими вызовами **glVertex*()** пока **glTexCoord*()** не будет вызвана снова.

Замечание: Изображение шахматной доски может неверно выглядеть на экране, если скомпилируете и запустите программу на своей машине. Например, вместо двух квадратов вы можете увидеть 4 треугольника с различно - спроецированным изображением. Если это произошло, попробуйте с помощью команды **glHint()** установить параметр GL_PERSPECTIVE_CORRECTION_HINT в значение GL_NICEST.

9.2 Создание текстуры

Команда **glTexImage2D()** определяет двумерную текстуру. Она принимает несколько аргументов, которые кратко описаны здесь и более подробно – в последующих разделах. Аналогичные команды для одномерных и трехмерных текстур – **glTexImage1D()** и **glTexImage3D()** описаны в соответствующих разделах.

```
void glTexImage2D (GLenum target, GLint level, GLint internalFormat, GLsizei
width, GLsizei height,
                   GLint border, GLenum format, GLenum type, const GLvoid
*texels);
```

Определяет двумерную текстуру. Аргумент ***target*** должен быть установлен в **GL_TEXTURE_2D** или **GL_PROXY_TEXTURE_2D.** Параметр ***level*** используется в том случае, если вы создаете несколько разрешений текстурной карты, в случае одного разрешения, ***level*** должен быть равен **0.** Следующий аргумент ***internalFormat*** определяет, какие величины – **R, G, B, A,** светлота или интенсивность выбраны для использования в описании тэкселей изображения. Значение для ***internalFormat***– это число от **1** до **4** или одна из **38** символических констант. Далее приводится список **38** допустимых констант: **GL_ALPHA, GL_ALPHA4, GL_ALPHA8, GL_ALPHA12, GL_ALPHA16, GL_LUMINANCE, GL_LUMINANCE4, GL_LUMINANCE8, GL_LUMINANCE12, GL_LUMINANCE16, GL_LUMINANCE_ALPHA, GL_LUMINANCE4_ALPHA4, GL_LUMINANCE6_ALPHA2, GL_LUMINANCE8_ALPHA8, GL_LUMINANCE12_ALPHA4, GL_LUMINANCE12_ALPHA12, GL_LUMINANCE16_ALPHA16, GL_INTENSITY, GL_INTENSITY4, GL_INTENSITY8, GL_INTENSITY12, GL_INTENSITY16, GL_RGB, GL_R3_G3_B2, GL_RGB4, GL_RGB5, GL_RGB8, GL_RGB10, GL_RGB12, GL_RGB16, GL_RGBA, GL_RGBA2, GL_RGBA4, GL_RGB5_A1, GL_RGBA8, GL_RGB10_A2, GL_RGBA12, GL_RGBA16.** Если значение ***internalFormat*** равно одной из **38** символических констант, это значит, что вы запрашиваете конкретные компоненты и, возможно, разрешение этих компонент. Например, если ***internalFormat*** равен **GL_R3_G3_B2,** вы требуете, чтобы в каждом тэкселе было **3** бита на красный компонент, **3** бита на зеленый и **2** бита на синий, однако **OpenGL** не гарантирует, что все будет именно так. Вместо этого, **OpenGL** попытается выбрать внутреннее представление данных, которое наиболее точно подходит к тому, что вы запросили. Точное совпадение обычно не требуется. Внутренние форматы **GL_LUMINANCE, GL_LUMINANCE_ALPHA, GL_RGB**и **GL_RGBA** по определению являются обобщенными, поскольку в них отсутствует запрос на конкретное разрешение. **(**Из-за требований совместимости с **OpenGL**версии **1.0** числовые величины для ***internalFormat***– **1, 2, 3** и **4 –** являются эквивалентами символических констант **GL_LUMINANCE, GL_LUMINANCE_ALPHA, GL_RGB** и **GL_RGBA** соответственно.**)** Аргументы ***width*** и ***height*** задают размеры изображения текстуры; ***border*** задает толщину границы и должно быть равно либо **0 (**граница отсутствует**),** либо **1.** И ***width***, и ***height*** должны иметь форму $2^m + 2b$, где **m –** неотрицательное целое **(**которое может иметь разное значение для ***width*** и ***height***), а **b**равно аргументу **border.** Максимальный размер текстуры зависит от реализации **OpenGL,** но он должен быть как минимум **64x64 (**то есть **66x66** вместе с границами**).** Аргументы ***format*** и ***type*** описывают формат и тип данных изображения текстуры. Они имеют то же значение, что и для команды **glDrawPixels().** Вообще говоря, данные текстуры имеют тот же формат, что и данные, используемые **glDrawPixels(),** таким образом, имеют значение установки команд **glPixelStore*()** и **glPixelTransfer*().** (В примере **9-1** вызов

```
glPixelsStorei(GL_UNPACK_ALIGNMENT,1);
```

производится потому, что данные не выравниваются специальным образом в конце каждого ряда тэкселей.**)** Аргумент ***format*** может принимать значения **GL_COLOR_INDEX, GL_RGB, GL_RGBA, GL_RED, GL_GREEN, GL_BLUE, GL_ALPHA, GL_LUMINANCE, GL_LUMINANCE_ALPHA–** то есть форматы могут быть такими же, как и у команды **glDrawPixels()** за исключением **GL_STENCIL_INDEX** и **GLL_DEPTH_COMPONENT.** Похожим образом аргумент ***type*** может принимать значения **GL_BYTE,**

GL_UNSIGNED_BYTE, GL_SHORT, GL_UNSIGNED_SHORT, GL_INT, GL_UNSIGNED_INT, GL_FLOAT, GL_BITMAP или одно из значений, означающих упакованный тип данных. Наконец, аргумент ***texels*** является указателем на данные изображения текстуры. Эти данные описывают само изображение и его границы.

Внутренний формат изображения текстуры может воздействовать на быстродействие текстурных операций. Например, некоторые реализации производят **GL_RGBA** текстурирование быстрее, чем **GL_RGB**, поскольку цветовые компоненты лучше выравниваются в процессорной памяти. Поскольку таких аспектов может быть множество, вам следует проверить информацию о своей реализации.

Внутренний формат изображения текстуры также определяет то, сколько памяти поглощается этим изображением. Например, текстура с внутренним форматом **GL_RGB8** использует **32** бита на тэксель, а текстура с внутренним форматом **GL_R3_G3_B2** использует только **8** бит на тэксель. Конечно, существует определенное противоречие между объемом поглощаемой памяти и цветовым разрешением.

Несмотря на то, что результат текстурирования в индексном режиме не определен, вы все-таки можете создать текстуру из изображения в формате **GL_COLOR_INDEX**. В этом случае до создания текстуры применяются операции пиксельного переноса для табличной конверсии индексов в **RGBA** величины.

Если ваша реализация **OpenGL** поддерживает подмножество обработки изображений и какие-либо механизмы этого подмножества активизированы, эти механизмы будут воздействовать на изображение текстуры. Например, если включена двумерная фильтрация, то изображение текстуры будет отфильтровано (фильтрация может изменить ширину и/или высоту изображения).

Число тэкселей в каждом ряду и каждом столбце изображения текстуры без опциональной границы должно быть степенью **2**. Если ваше исходное изображение не соответствует этому ограничению, вы можете использовать функцию **gluScaleImage()** из библиотеки утилит **GLU**, чтобы изменить размер ваших текстур.

```
int gluScaleImage (GLenum format, GLint widthin, GLint heightin, GLenum typein,
const *datain,
                   GLint widthout, GLint heightout, GLenum typeout, void
*dataout);
```

Масштабирует изображения, используя установленный режим хранения пикселей для распаковки данных из ***datain***. Аргументы ***format***, ***typein*** и ***typeout*** могут иметь любые значения форматов и типов данных, поддерживаемые командой **glDrawPixels()**. Масштабирование от размеров ***widthin*** x ***heightin*** до размеров ***widthout*** x ***heightout*** производится с использованием линейной интерполяции. Результирующее изображение записывается в ***dataout***с использованием текущих режимов хранения пикселей **GL_PACK***. Ответственность за выделение достаточного количества памяти для хранения результирующего изображения лежит на программисте. В случае успеха функция возвращает **0**, в случае неудачи – код ошибки **GLU**.

Замечание: В GLU версии 1.3 **gluScaleImage()** поддерживает упакованные пиксельные форматы (и связанные с ними типы данных), появившиеся в OpenGL 1.2.

Буфер кадра также может быть использован в качестве источника данных для текстуры. Команда **glCopyTexImage2D()** считывает прямоугольник пикселей из буфера кадра и использует его в качестве тэкселей новой текстуры.

```
void glCopyTexImage2D (GLenum target, GLint level, GLint internalFormat, GLint x,
GLint y, GLsizei width, GLsizei height, GLint border);
```

Создает двумерную текстуру, используя в качестве тэкселей данные из буфера кадра. Пиксели считываются из текущего буфера для чтения **(GL_READ_BUFFER)** и обрабатываются точно так же, как в случае вызова **glCopyPixels()**, однако вместо перемещения в буфер кадра, пиксели помещаются в текстурную память. Во время обработки применяются установки команды **glPixelTransfer*()** и другие операции переноса пикселей. Аргумент ***target*** должен быть установлен в значение **GL_TEXTURE_2D**. Аргументы ***level***, ***internalFormat*** и ***border*** имеют тот же эффект, что и для команды **glTexImage2D()**. Текстурный массив извлекается из экранного прямоугольника пикселей с нижним левым углом в точке **(*x*, *y*)**. Аргументы ***width*** и ***height*** задают размеры этого прямоугольника. И ***width***, и ***height*** должны иметь форму $2^m + 2b$, где **m –** неотрицательное целое **(**которое может иметь разное значение для ***width***и ***height*****)**, а **b** равно аргументу **border**.

В следующих разделах детали текстурирования, в том числе использование параметров ***target***, ***border*** и ***level***, излагаются более подробно. Параметр ***target*** может использоваться для запроса точного размера текстуры **(**для этого нужно создать текстурное прокси командой **glTexImage*D())**, а также для выяснения того, может ли конкретная текстура использоваться с учетом объема ресурсов реализации **OpenGL**. Существует возможность переопределения части текстуры. Детализируется использование границы текстуры. Параметр ***level*** может использоваться для создания текстур с различным разрешением, он также связан с техникой мипмаппинга, который в свою очередь требует понимания фильтрации текстур.

9.2.1 Текстурное прокси

Для программиста **OpenGL**, использующего текстуры, размер действительно имеет значение. Обычно текстурные ресурсы ограничены, а ограничения на форматы текстур меняются в зависимости от реализации. Существует механизм текстурного прокси, которое позволяет определить, способна ли ваша реализация **OpenGL** работать с текстурой конкретного формата и конкретного размера.

Команда **glGetIntegerv(GL_MAX_TEXTURE_SIZE, ...)**; выдаст вам нижнюю границу **(**ширину или высоту**)** максимально большой текстуры **(**без учета границы текстуры**)**. Обычно наибольшая текстура бывает квадратной. Константа **GL_MAX_3D_TEXTURE_SIZE** может быть использована для получения максимально допустимого измерения **3D** текстуры **(**ширины, высоты или глубины, без учета границ**)**.

Однако ни **GL_MAX_TEXTURE_SIZE**, ни **GL_MAX_3D_TEXTURE_SIZE** не заботится о внутреннем формате изображения или таких аспектах, как мипмаппинг. Изображение текстуры, сохраненное в формате **GL_RGB16** занимает **64** бита памяти под каждый тэксель, тогда как в формате **GL_LUMINANCE4** оно занимало бы в **16** раз меньше. **(**Кроме того, изображения, для которых требуется граница или мипмапы, могут еще быстрее сокращать объем доступной памяти.**)**

Специальный механизм **–** прокси изображения текстуры позволяет программе более точно выяснить может ли **OpenGL** принять текстуру желаемого внутреннего формата. Для использования текстурного прокси вы должны вызвать команду **glTexImage2D()** с аргументом ***target***, установленным в значение **GL_PROXY_TEXTURE_2D** и желаемыми аргументами ***level***, ***internalFormat***, ***width***, ***height***, ***border***, ***format*** и ***type***. **(**Для одномерных и трехмерных текстур используйте соответствующие **1D** и **2D** команды и константы.**)** При работе с прокси в качестве аргумента ***texels***вы должны передать **NULL**.

Чтобы выяснить, достаточно ли свободных ресурсов присутствует в системе для вашей текстуры, после создания прокси опросите переменные состояния текстуры с помощью команды **glGetTexLevelParameter*()**. Если ресурсов недостаточно, такие параметры как ширина и высота текстуры, ширина ее границы и разрешение ее компонентов будут равны **0**.

```
void glGetTexLevelParameter{if}v (GLenum target, GLint level, GLenum pname, TYPE
*params);
```

В аргументе ***params*** команда возвращает значения параметров текстуры для заданного с помощью аргумента ***level*** уровня детализации**.** Аргумент ***target*** задает целевую текстуру и может быть равен **GL_TEXTURE_1D, GL_TEXTURE_2D, GL_TEXTURE_3D, GL_PROXY_TEXTURE_1D, GL_PROXY_TEXTURE_2D, GL_PROXY_TEXTURE_3D.** Допустимыми значениями для аргумента ***pname*** являются**: GL_TEXTURE_WIDTH, GL_TEXTURE_HEIGHT, GL_TEXTURE_DEPTH, GL_TEXTURE_BORDER, GL_TEXTURE_INTERNAL_FORMAT, GL_TEXTURE_RED_SIZE, GL_TEXTURE_GREEN_SIZE, GL_TEXTURE_BLUE_SIZE, GL_TEXTURE_ALPHA_SIZE, GL_TEXTURE_LUMANANCE_SIZE, GL_TEXTURE_INTENSITY_SIZE.** Для аргумента ***pname*** также доступно значение **GL_TEXTURE_COMPONENTS,** но только из**-**за необходимости совместимости с версией **OpenGL 1.0.** Для последующих версий рекомендуется использовать константу **GL_TEXTURE_INTERNAL_FORMAT.**

Пример **9-2** демонстрирует технику использования текстурного прокси для того**,** чтобы выяснить достаточно ли в системе ресурсов под создание текстуры размером **64 x 64** тэкселя в формате **RGBA** с **8** битами на компонент**.** Если ресурсов достаточно**, glGetTexLevelParameteriv()** сохраняет внутренний формат **(**в данном случае **GL_RGBA8)** в переменной ***format*****.**

Пример **9-2.** Опрос текстурных ресурсов с помощью текстурного прокси

```
GLint width;

glTexImage2D(GL_PROXY_TEXTURE_2D, 0, GL_RGBA8, 64, 64, 0, GL_RGBA,
GL_UNSIGNED_BYTE, NULL);
glGetTexLevelParameteriv(GL_PROXY_TEXTURE_2D, 0, GL_TEXTURE_WIDTH,
&width);
```

Замечание: Существует одно серьезно ограничение на использование прокси: текстурное прокси отвечает на вопрос о том, может ли текстура быть загружена в текстурную память. Прокси дает один и тот же ответ вне зависимости от того, сколько ресурсов уже используется в текущий момент. Если ресурсы заняты другими текстурами, ответ на запрос может быть утвердительным, однако ресурсов может быть недостаточно для того, чтобы сделать вашу текстуру резидентной (то есть частью высокоскоростного рабочего подмножества текстур, которое поддерживается). Текстурное прокси, таким образом, не отвечает на вопрос о том, есть ли достаточно ресурсов для обработки текстуры на данный конкретный момент.

9.2.2 Замена всего изображения текстуры или его части

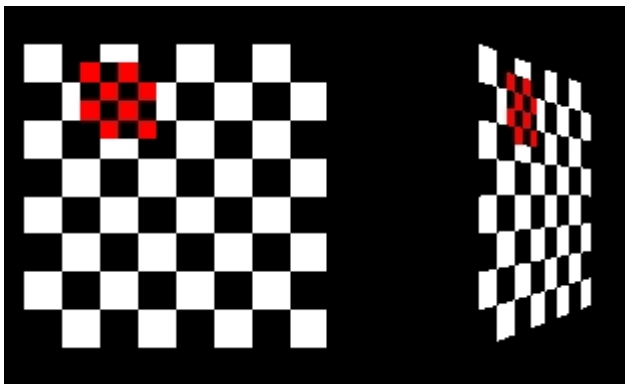
Создание новой текстуры с точки зрения вычислений может быть значительно дороже**,** чем модификация существующей**.** Начиная с **OpenGL** версии **1.1,** были введены несколько новых команд для замещения всего изображения текстуры или его части новой информацией**.** Это может быть весьма полезно для некоторых приложений**,** например**,** для тех**,** которые используют в качестве текстуры видеоизображение**,** захваченное в реальном времени**.** Для таких приложений имеет смысл создать одну текстуру и затем использовать команду **glTexSubImage2D()** для многократной замены данных текстуры новыми видеоизображениями**.** Кроме того**,** для команды **glTexSubImage2D()** нет ограничений на ширину или высоту **–** они не обязаны быть степенью **2. (**Обычно это полезно для обработки видеоизображений**,** размеры которых обычно не являются степенью **2.** Однако вы должны загружать изображения в первоначальную большую текстуру**,** размеры которой и по ширине**,** и по высоте должны быть 2^n**,** а также вам нужно настраивать координаты текстуры под новое изображение**.)**

```
void glTexSubImage2D (GLenum target, GLint level, GLint xoffset, GLint yoffset,
GLsizei width, GLsizei height,
                      GLenum format, GLenum type, const GLvoid *texels);
```

Задает двумерное текстурное изображение, которое заменяет непрерывный подрегион (в 2D это просто прямоугольник) текущего существующего текстурного изображения. Аргумент ***target*** должен быть установлен в значение **GL_TEXTURE_2D**. Аргументы ***level***, ***format*** и ***type*** имеют тот же смысл, что и аналогичные аргументы команды **glTexImage2D()**. ***level***– это номер уровня детализации. Указание параметров ***width*** и ***height*** установленных в **0** не является ошибкой, но в этом случае команда не имеет эффекта. ***format*** и ***type*** описывают формат и тип данных в изображении текстуры. на новое изображение также влияют установки команд **glPixelStore*()**, **glPixelTransfer*()** и другие операции передачи пикселей. ***texels*** содержит данные для подтекстуры. ***width*** и ***height*** являются размерами подрегиона, который заменяет все или часть текущего изображения текстуры. ***xoffset*** и ***yoffset*** задают смещение от левого нижнего угла текстуры в пикселях по **x** и **y** осям соответственно, то есть они задают, где в исходном изображении должны быть помещены новые данные. Подрегион не может включать тэксели вне оригинального изображения текстуры.

Пример **9-3** представляет собой модификацию примера **9-1.** Нажатие на клавишу **'s'** заменяет часть изображения текстуры новым изображением. (Результирующее изображение приведено на рисунке **9-3.**) Нажатие на клавишу **'r'** восстанавливает исходное изображение. В примере **9-3** были добавлены функции **makeCheckImages()** и **keyboard()).**

Рисунок **9-3.** Текстура с частично измененным изображением



Пример **9-3.** Замещение части текстуры: файл **texsub.cpp**

```
//Параметры текстуры шахматной доски
#define checkImageWidth 64
#define checkImageHeight 64
#define subImageWidth 16
#define subImageHeight 16
GLubyte checkImage[checkImageHeight][checkImageWidth][4];
GLubyte subImage[subImageHeight][subImageWidth][4];

void makeCheckImages()
{
int i,j,c;

for (i=0;i<checkImageHeight;i++)
{
for (j=0;j<checkImageWidth;j++)
{
c=(((i&0x8)==0)^((j&0x8)==0))*255;
checkImage[i][j][0]=(GLubyte)c;
```

```
checkImage[i][j][1]=(GLubyte)c;
checkImage[i][j][2]=(GLubyte)c;
checkImage[i][j][3]=(GLubyte)255;
}
}

for (i=0;i<subImageHeight;i++)
{
for (j=0;j<subImageWidth;j++)
{
c=(((i&0x4)==0)^((j&0x4)==0))*255;
subImage[i][j][0]=(GLubyte)c;
subImage[i][j][1]=(GLubyte)0;
subImage[i][j][2]=(GLubyte)0;
subImage[i][j][3]=(GLubyte)255;
}
}
}

void keyboard(unsigned char key, int x, int y)
{
switch (key)
{
case 's':
case 'S':
glBindTexture(GL_TEXTURE_2D,texName);
glTexSubImage2D(GL_TEXTURE_2D,0,12,44,subImageWidth,subImageHeight,GL_RGBA,GL
_UNSIGNED_BYTE,subImage);
glutPostRedisplay();
break;
case 'r':
case 'R':
glBindTexture(GL_TEXTURE_2D,texName);
glTexImage2D(GL_TEXTURE_2D,0,GL_RGBA,checkImageWidth,checkImageHeight,0,GL_RG
BA,GL_UNSIGNED_BYTE,checkImage);
glutPostRedisplay();
break;
}
}
```

Буфер кадра также может использоваться в качестве источника текстурных данных – на этот раз в качестве источника данных для подтекстуры. Команда **glCopyTexSubImage2D()** считывает прямоугольник пикселей из буфера кадра и замещает ими часть массива существующей текстуры. **(glCopyTexSubImage2D() –** это что-то среднее между **glCopyTexImage2D()**и **glTexSubImage2D().)**

```
void glCopyTexSubImage2D (GLenum target, GLint level, GLine xoffset, GLint
yoffset, GLint x, GLint y, GLsizei width, GLsizei height);
```

Использует данные изображения из буфера кадра для замещения целого или части непрерывного подрегиона текущей существующей двумерной текстуры. Пиксели считываются из текущего буфера для чтения **(GL_READ_BUFFER)** и обрабатываются так же как в случае **glCopyPixels(),** но вместо переноса в буфер кадра, пиксели помещаются в текстурную память. На переносимые пиксели влияют установки команд **glPixelStore*(), glPixelTransfer*()** и другие операции передачи пикселей. Аргумент ***target*** должен быть установлен в значение **GL_TEXTURE_2D.** ***level*** представляет собой уровень детализации мипмапа. ***xoffset*** и ***yoffset*** задают смещение от левого нижнего угла текстуры в пикселях по осям абсцисс и ординат соответственно, то есть они задают, где в исходном изображении должны быть помещены новые данные. Подизображение извлекается из экранного прямоугольника пикселей с левым нижним углом в точке **(*x*, *y*).** Аргументы ***width*** и ***height*** задают размер этого прямоугольника.

9.2.3 Одномерные текстуры

Иногда одномерная текстура просто необходима – например, если вы рисуете текстурированные отрезки, когда все вариации внешнего вида происходят только в одном направлении. Одномерная текстура ведет себя так же как двумерная текстура с высотой равной **1** и отсутствием границ сверху и снизу. Все команды создания двумерных текстур и подтекстур имеют свои аналоги в одномерном варианте. Чтобы создать простую одномерную текстуру, используйте команду **glTexImage1D()**.

```
void glTexImage1D (GLenum target, GLint level, GLint internalFormat, GLsizei
width, GLint border,
                   GLenum format, GLenum type, const GLvoid *texels);
```

Создает одномерную текстуру. Все параметры имеют то же назначение, что и аналогичные параметры **glTexImage2D()** за тем исключением, что аргумент ***texels*** в данном случае является одномерным массивом. Как и раньше значение ***width*** должно быть равно 2^m (или $2^m + 2$, если присутствует граница), где **m** – неотрицательное целое. Вы можете использовать мипмаппинг и текстурное прокси (для этого нужно вызвать команду с аргументом ***target***, установленным в **GL_PROXY_TEXTURE_1D**), также присутствуют аналогичные варианты фильтрации.

В качестве примера программы, использующей одномерную текстуру, обратитесь к примеру **9-8**.

Если ваша реализация **OpenGL** поддерживает подмножество обработки изображений и активизирована одномерная фильтрация **(GL_CONVOLUTION_1D)**, то изображение фильтруется. (Фильтрация может изменить ширину изображения текстуры.) Также могут применяться и другие пиксельные операции.

Чтобы заменить все или часть тэкселей одномерной текстуры, используйте команду **glTexSubImage1D()**.

```
void glTexSubImage1D (GLenum target, GLint level, GLint xoffset, GLsizei width,
GLenum format, GLenum type, const GLvoid *texels);
```

Создает одномерный текстурный массив, который заменяет весь или часть региона (то есть в **1D** – ряда) текущего существующего изображения одномерной текстуры. Аргумент ***target*** должен быть установлен в значение **GL_TEXTURE_1D**. Аргументы ***level***, ***format*** и ***type*** имеют то же значение, что и для команды **glTexImage1D()**. ***level*** – это уровень детализации мипмапа. ***format*** и ***type*** описывают формат и тип данных изображения текстуры. Частичное изображение подвержено влиянию режимов, установленных командами **glPixelsStore*()**, **glPixelsTransfer*()** и другим операциям пиксельного переноса. Аргумент ***texels*** содержит данные изображения частичной текстуры. ***width*** – это количество тэкселей, которые заменяют все изображение текущей текстуры или его часть. ***xoffset*** задает смещение в тэкселях от начала массива существующей текстуры, индицируя то место, с которого будут начинаться новые данные.

Для использования буфера кадра в качестве источника данных для новой одномерной текстуры или частичного изображения одномерной текстуры, которое заместит существующее, используйте команды **glCopyTexImage1D()** или **glCopyTexSubImage1D()** соответственно.

```
void glCopyTexImage1D (GLenum target, GLint level, GLint internalFormat, GLint x,
GLint y, GLsizei width, GLint border);
```

Создает одномерную текстуру, используя данные из буфера кадра в качестве тэкселей. Пиксели считываются из текущего буфера для чтения **(GL_READ_BUFFER)** и обрабатываются точно так же, как в случае вызова **glCopyPixels()**, однако вместо перемещения в буфер кадра, пиксели помещаются в текстурную память. Во время обработки применяются установки команды **glPixelTransfer*()** и другие операции переноса пикселей. Аргумент ***target*** должен быть установлен в значение **GL_TEXTURE_1D**. Аргументы ***level***, ***internalFormat*** и ***border*** имеют тот же эффект, что и для команды **glCopyTexImage2D()**. Текстурный массив извлекается из экранного ряда пикселей с нижним левым углом в точке **(*x*, *y*)**. Значение ***width*** должно быть равно 2^m **(**или $2^m + 2$ **,** если присутствует граница**),** где **m –** неотрицательное целое.

```
void glCopyTexSubImage1D (GLenum target, GLint level, GLint xoffset, GLint x,
GLint y, GLsizei width, GLint border);
```

Использует данные изображения из буфера кадра для замещения целого или части непрерывного подрегиона текущей существующей одномерной текстуры. Пиксели считываются из текущего буфера для чтения **(GL_READ_BUFFER)** и обрабатываются так же как в случае **glCopyPixels()**, но вместо переноса в буфер кадра, пиксели помещаются в текстурную память. На переносимые пиксели влияют установки команд **glPixelStore*()**, **glPixelTransfer*()** и другие операции передачи пикселей. Аргумент ***target*** должен быть установлен в значение **GL_TEXTURE_1D**. ***level*** представляет собой уровень детализации мипмапа. ***xoffset*** задает смещение внутри массива существующей текстуры, то есть оно задает, где в исходном изображении должны быть помещены новые данные. Подизображение извлекается из экранного ряда пикселей с левым нижним углом в точке **(*x*, *y*)**. Аргумент ***width*** задает число пикселей в этом ряду.

9.2.4 Трехмерные текстуры

Трехмерные текстуры чаще всего используются в медицинских и геологических приложениях. В медицинских приложениях трехмерная текстура может представлять многослойную компьютерную томографию или визуализацию магнитного резонанса. Для исследователей, связанных с нефтью и газом, трехмерная текстура может применяться для моделирования слоев камня. **(**Трехмерные текстуры являются неотъемлемой частью определенного класса приложений, связанных с *визуализацией объема* **– *volume rendering applications***. Наиболее продвинутые из этих приложений работают с *вокселями* **– *voxels***, которые представляют данные в виде сущностей, зависимых от объема.**)**

Из-за своего размера трехмерные текстуры могут поглощать большой объем текстурных ресурсов системы. Даже самая простая трехмерная текстура может занимать в **16** или даже **32** раза больше памяти, чем одна двумерная.

До версии **1.2** трехмерные текстуры были довольно распространенным расширением от нескольких производителей реализаций **OpenGL**. В версии **1.2** поддержка трехмерных текстур была включена в ядро **OpenGL**. Большинство команд для работы с двумерными текстурами и подтекстурами имеют свои аналоги для трехмерного варианта.

Можно представить себе трехмерную текстуру в виде слоев двумерных прямоугольников подизображений. В памяти эти прямоугольники выстроены в последовательность. Чтобы создать простую трехмерную текстуру используйте команду **glTexImage3D ()**.

```
void glTexImage3D (GLenum target, GLint level, GLint internalFormat, GLsizei
width, GLsizei height,
                   GLsizei depth, GLint border, GLenum format, GLenum type, const
GLvoid *texels);
```

Создает трехмерную текстуру. Все параметры имеют то же значение, что и для команды **glTexImage2D()**, за исключением того, что здесь ***texels*** представляет собой трехмерный массив. Кроме того, добавился параметр ***depth***. Величина ***depth*** должна иметь форму 2^m (или $2^m + 2$, если присутствует граница), где **m** – неотрицательное целое. Вы можете использовать мипмаппинг и текстурное прокси (для этого нужно вызвать команду с аргументом ***target***, установленным в **GL_PROXY_TEXTURE_3D**), также присутствуют аналогичные варианты фильтрации.

Замечание: В подмножестве обработки изображений отсутствуют трехмерные фильтры, однако вы можете использовать двумерные фильтры для воздействия на изображения трехмерной текстуры.

Пример **9-4** является частью программы, использующей трехмерные текстуры.

Пример **9-4.** Трехмерное текстурирование: файл **texture3d.cpp**

```
#define iwidth 16
#define iheight 16
#define idepth 16

GLubyte image [idepth][iheight][iwidth][3];
GLuint texName;

//Функция создает массив размерности 16x16x16x3, с различными
цветовыми
//величинами [r, g, b] в каждом элементе. Значения величин лежат в
диапазоне
//от 0 до 255
void makeImage(void)
{
int s, t, r;

for(s=0;s<16;s++)
for(t=0;t<16;t++)
for(r=0;r<16;r++)
{
image[r][t][s][0]=s*17;
image[r][t][s][1]=t*17;
image[r][t][s][2]=r*17;
}
}

//Инициализировать состояние: объект 3D текстуры и ее изображение
void init(void)
{
glClearColor(0.0,0.0,0.0,0.0);
glShadeModel(GL_FLAT);
glEnable(GL_DEPTH_TEST);

makeImage();
glPixelStorei(GL_UNPACK_ALIGNMENT,1);

glGenTextures(1, &texName);
glBindTexture(GL_TEXTURE_3D,texName);

glTexParameteri(GL_TEXTURE_3D, GL_TEXTURE_WRAP_S, GL_CLAMP);
glTexParameteri(GL_TEXTURE_3D, GL_TEXTURE_WRAP_T, GL_CLAMP);
glTexParameteri(GL_TEXTURE_3D, GL_TEXTURE_WRAP_R, GL_CLAMP);

glTexParameteri(GL_TEXTURE_3D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
glTexParameteri(GL_TEXTURE_3D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
```

```
glTexImage3D(GL_TEXTURE_3D,0,GL_RGB,iWidth,iHeight,iDepth,
0,GL_RGB,GL_UNSIGNED_BYTE,image);
}
```

Чтобы заменить все или часть тэкселей трехмерной текстуры, используйте команду **glTexSubImage3D().**

```
void glTexSubImage3D (GLenum target, GLint level, GLint xoffset, GLint yoffset,
GLint zoffset,
                      GLsizei width, GLsizei height, GLsizei depth, GLenum
format,
                      GLenum type, const GLvoid *texels);
```

Создает трехмерный текстурный массив, который заменяет весь или часть подрегиона текущего существующего изображения трехмерной текстуры. Аргумент ***target*** должен быть установлен в значение **GL_TEXTURE_3D.** Аргументы ***level***, ***format*** и ***type*** имеют то же значение, что и для команды **glTexImage3D()**. ***level***– это уровень детализации мипмапа. ***format*** и ***type*** описывают формат и тип данных изображения текстуры. Частичное изображение подвержено влиянию режимов, установленных командами **glPixelsStore*()**, **glPixelsTransfer*()** и другим операциям пиксельного переноса. Аргумент ***texels*** содержит данные изображения частичной текстуры. ***width***, ***height*** и ***depth*** представляют собой размеры подизображения в тэкселях. ***xoffset***, ***yoffset*** и ***zoffset*** задают смещение в тэкселях от начала массива существующей текстуры, индицируя то место, индицируя то место, куда нужно поместить новые данные.

Для использования буфера кадра в качестве источника данных для частичного изображения одномерной текстуры, которое заместит существующее, используйте команду **glCopyTexSubImage3D().**

```
void glCopyTexSubImage3D (GLenum target, GLint level, GLint xoffset, GLint
yoffset, GLint zoffset,
                          GLint x, GLint y, GLsizei width, GLint border);
```

Использует данные изображения из буфера кадра для замещения целого или части непрерывного подрегиона текущей существующей трехмерной текстуры. Пиксели считываются из текущего буфера для чтения **(GL_READ_BUFFER)** и обрабатываются так же как в случае **glCopyPixels()**, но вместо переноса в буфер кадра, пиксели помещаются в текстурную память. На переносимые пиксели влияют установки команд **glPixelStore*()**, **glPixelTransfer*()** и другие операции передачи пикселей. Аргумент ***target*** должен быть установлен в значение **GL_TEXTURE_3D.** ***level*** представляет собой уровень детализации мипмапа. Подизображение извлекается из экранного прямоугольника пикселей с левым нижним углом в точке **(*x*, *y*).** Аргументы ***width*** и ***height*** задают размеры этого прямоугольника. ***xoffset***, ***yoffset*** и ***zoffset*** задают смещение внутри массива существующей текстуры, то есть они определяют место, куда следует поместить извлеченные данные. Поскольку подизображение является двумерным, оно может заместить только часть или целое изображение в одном срезе (срезе со смещением ***zoffset***).

9.2.4.1 Режимы хранения пикселей для трехмерных текстур

Режимы хранения пикселей контролируют пропуски между рядами каждого слоя (иными словами, одного двумерного прямоугольника). Команда **glPixelStore*()** устанавливает режимы хранения пикселей с такими параметрами, как ***ROW_LENGTH**, ***ALIGNMENT**, ***SKIP_PIXELS** и ***SKIP_ROWS** (где * означает или **GL_PACK** или **GL_UNPACK**), что управляет разрешением части целого прямоугольника (также прямоугольной) пиксельных или тэксельных данных.

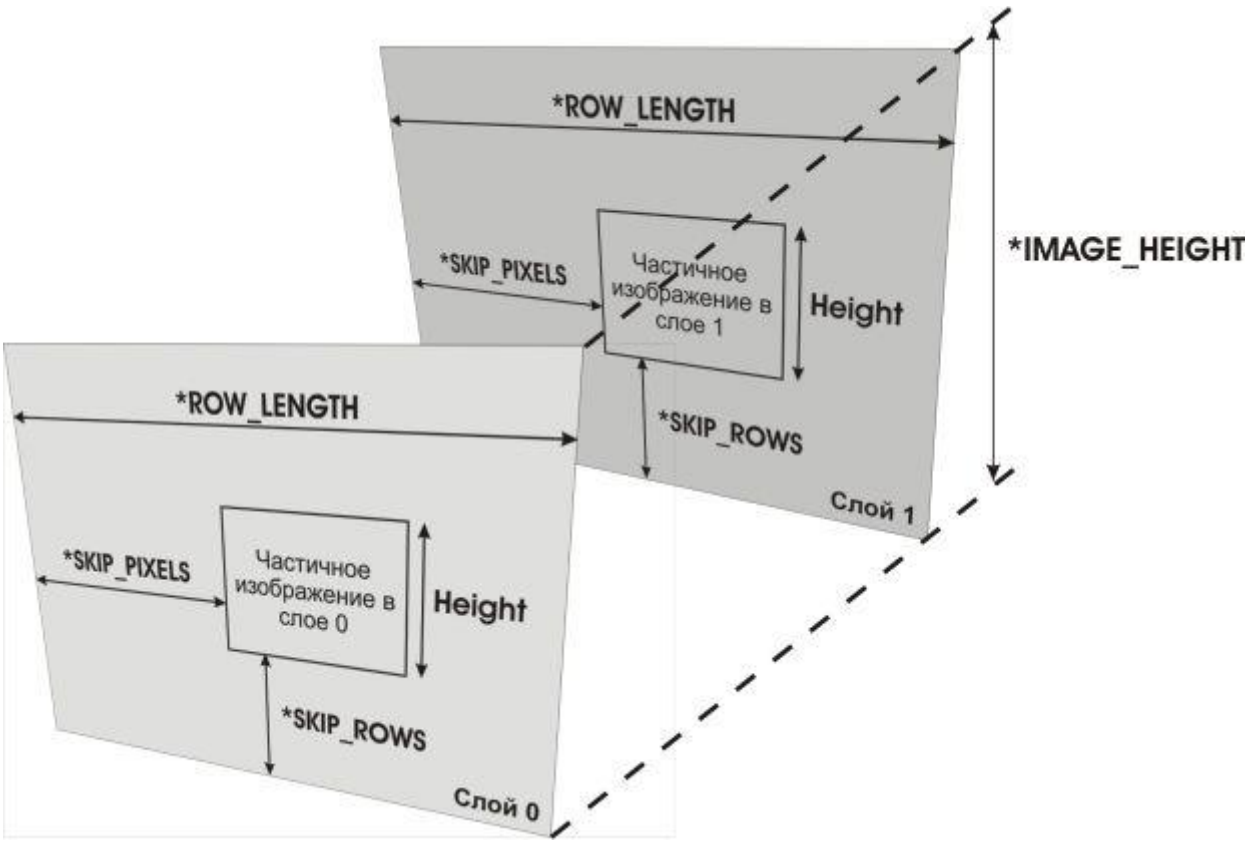
Перечисленные ранее режимы хранения пикселей остаются полезными для описания двух из трех измерений, но для поддержки разрешения частичных объемов изображения трехмерной текстуры требуются дополнительные режимы. Новые параметры ***IMAGE_HEIGHT** и ***SKIP_IMAGES** позволяют командам **glTexImage3D()**, **glTexSubImage3D()** и **glGetTexImage()** получать доступ к любому желаемому частичному объему.

Если трехмерная текстура в памяти больше, чем определенный частичный объем, вам необходимо указать высоту одного частичного изображения с помощью параметра ***IMAGE_HEIGHT**. Также, если частичный объем начинается не с самого первого слоя, следует установить параметр ***SKIP_IMAGES**.

IMAGE_HEIGHT –** это параметр хранения пикселей, который определяет высоту **(**количество рядов**)** одного слоя изображения трехмерной текстуры. Если значение ***IMAGE_HEIGHT** равно **0 (**отрицательная величина не является допустимой**)**, число рядов в каждом двумерном прямоугольнике имеет величину ***height, являющуюся аргументом команд **glTexImage3D()** или **glTexSubImage3D(). (**Эта ситуация встречается часто, поскольку величиной по умолчанию для ***IMAGE_HEIGHT** является именно **0.)** В ином случае высота одного слоя равна величине ***IMAGE_HEIGHT**.

Рисунок **9-4** показывает, как ***IMAGE_HEIGHT** задает высоту изображения **(**когда параметр ***height*** задает только высоту частичного изображения**)**. На рисунке изображена трехмерная текстура с двумя слоями.

Рисунок **9-4.** Режим хранения пикселей ***IMAGE_HEIGHT**

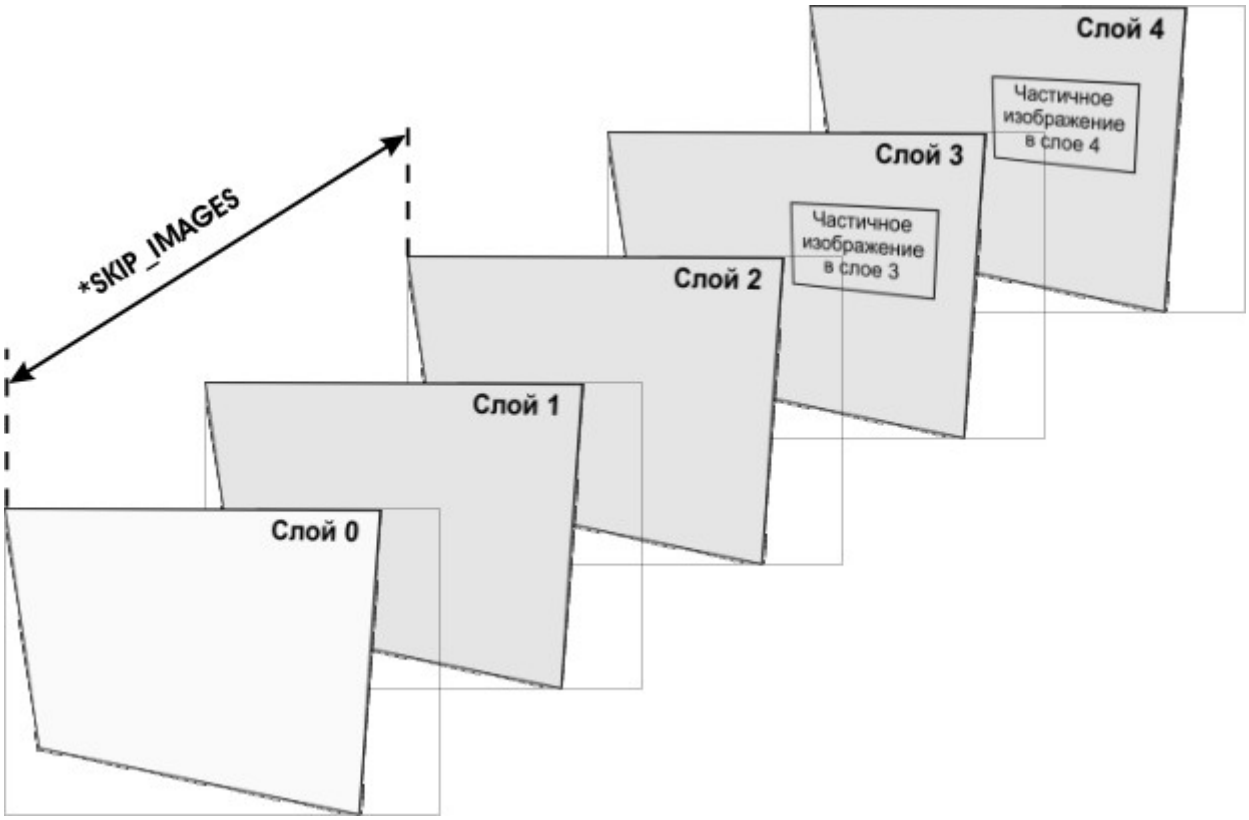


SKIP_IMAGES** определяет, сколько слоев нужно пройти до того, как можно будет получить доступ к данным частичного объема. Если ***SKIP_IMAGES** равно положительному целому числу **(**назовем его значением **n)**, то указатель в данных изображения текстуры сдвигается на это число слоев **(**то есть на **nразмер одного слоя тэкселей**)**. Результирующий частичный объем начинается со слоя **n** и продолжается на несколько слоев вглубь **–** на сколько вглубь, определяет аргумент ***depth***, передаваемый

командам **glTexImage3D()** или **glTexSubImage3D()**. Если ***SKIP_IMAGES** равно **0** (значение по умолчанию), доступ к данным тэкселей начинается с самого первого слоя, описанного в тэксельном массиве.

Рисунок **9-5** показывает, как параметр ***SKIP_IMAGES** проходит через несколько слоев, чтобы попасть туда, где в действительности расположен нужный частичный объем. В данном примере ***SKIP_IMAGES** равно **3**, и частичный объем начинается со слоя **3**.

Рисунок **9-5.** Режим хранения пикселей ***SKIP_IMAGES**



9.2.5 Использование границ текстуры

Дополнительно: Если вам нужно использовать большую текстуру, чем позволяет ваша реализация **OpenGL**, вы можете эффективно создавать большие текстуры, заполняя пространство текстурами меньшего размера. Например, если вам нужно наложить на квадрат текстуру размером вдвое большим, чем максимально допустимый, нарисуйте квадрат в виде **4**-ех квадратов меньшего размера, и на каждый из них наложите текстуру, представляющую собой четверть изображения исходной, большой текстуры (загружая новую четверть перед рисованием каждого из **4**-ех квадратов).

Поскольку в каждый конкретный момент доступна только одна карта текстуры, такой подход может привести к проблемам на ребрах текстур, особенно если используется одна из разновидностей линейной фильтрации. Значение текстуры, используемой для пикселей на ребрах, должно вычисляться с использованием чего-то за этими ребрами, в идеале, с чем-то, находящимся в соседней карте текстуры. Если для каждой текстуры вы определите границу, чьи величины тэкселей равны величинам тэкселей на ребре соседней карты текстуры, то при линейной фильтрации будет обеспечено корректное поведение.

Чтобы сделать это правильно, имейте в виду, что каждая текстура может иметь **8** соседей – по одной текстуре, прилегающей к каждому ребру и по одной, касающейся каждого из углов. Величины тэкселей в углах границы должны соответствовать

тэкселям на ребрах карт текстуры, касающихся углов. Если же ваша текстура является граничной для всей большой составной текстуры, вам нужно решить, какие значения помещать на ее границах. Простейшее разумное решение заключается в копировании соседних величин той же текстурной карты с помощью команды **glTexSubImage2D()**.

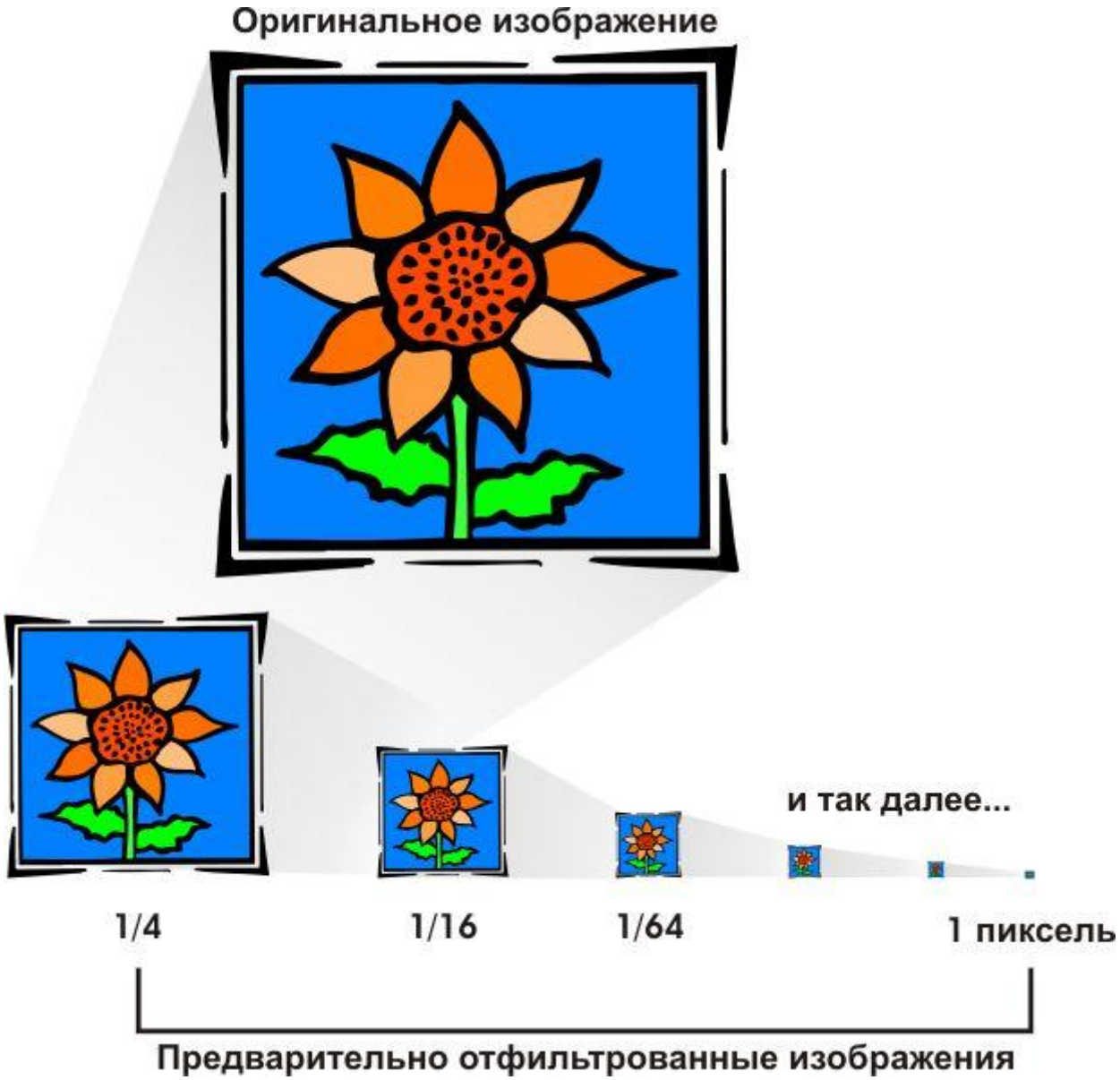
Цвет границы текстуры также используется в том случае, если текстура накладывается таким образом, что она только частично накрывает примитив.

9.2.6 Несколько уровней детализации

Дополнительно: Текстурированные объекты, как и все остальные объекты сцены, могут рассматриваться с разных дистанций от точки наблюдения. В динамической сцене в процессе перемещения текстурированного объекта дальше от точки наблюдения карта текстуры должна уменьшаться в размере вместе с размером проецируемого изображения. Чтобы достичь этого **OpenGL** должна фильтровать карту текстуры при ее уменьшении до размера, нужного для наложения на объект, таким образом, чтобы избежать появления визуальных артефактов. Например, чтобы визуализировать каменную стену, вы можете использовать большое изображение текстуры (скажем **128x128** тэкселей), если стена находится близко к наблюдателю. Однако если стена постепенно перемещается дальше от наблюдателя до тех пор, пока не будет размером с один пиксель, фильтрованная текстура может изменяться прерывисто, в определенные моменты.

Чтобы избежать подобных неприятностей, вы можете задавать серии предварительно отфильтрованных карт текстуры с разными разрешениями (**128x128**, **64x64**, **32x32** и так далее). Эти карты называются мипмапами и показаны на рисунке **9-6**. Термин *мипмап* (***mipmap***) был введен Лансом Вильямсом (**Lance Williams**), когда он представлял эту идею в своей статье «Пирамидальные параметрики» («**Pyramidal Parametrics**») (**SIGGRAPH1983**). «***Mip***» происходит от латинского «***multum in parvo***», что означает «много вещей в одном месте». Мипмаппинг использует некоторые ухищренные методы упаковки данных в памяти.

Рисунок **9-6**. Мипмапы




```
Замечание: Для полного понимания мипмапов, вам требуется понимание
уменьшающих фильтров.
```

При использовании мипмаппинга, **OpenGL** автоматически определяет, какую текстурную карту нужно использовать в зависимости от размера **(**в пикселях**)** текстурируемого объекта. При таком подходе уровень детализации текстурной карты соответствует изображению, рисуемому на экране – в то время, как объект на экране становится меньше, сокращается размер карты текстуры. Мипмаппинг требуется некоторых дополнительных расчетов и текстурной памяти; однако если его не использовать, текстуры, накладываемые на маленькие объекты, могут искажаться и мигать в процессе перемещения этих объектов.

Чтобы использовать мипмаппинг вы должны предоставить вашу текстуру во всех размерах, равных степеням **2**, от самого большого до размера **1x1**. Например, если наибольшее разрешение карты составляет **64x16**, вы должны также предоставить карты с размерами **32x8**, **16x4**, **8x2**, **4x1**, **2x1** и **1x1**. Меньшие карты обычно представляют собой фильтрованные и уменьшенные версии больших. Каждый тэксель в меньшей карте является средним между **4** соответствующими тэкселями в большей. **(**Поскольку **OpenGL** не накладывает никаких ограничений на метод вычисления меньших карт, карты текстуры разного размера могут быть абсолютно не связаны

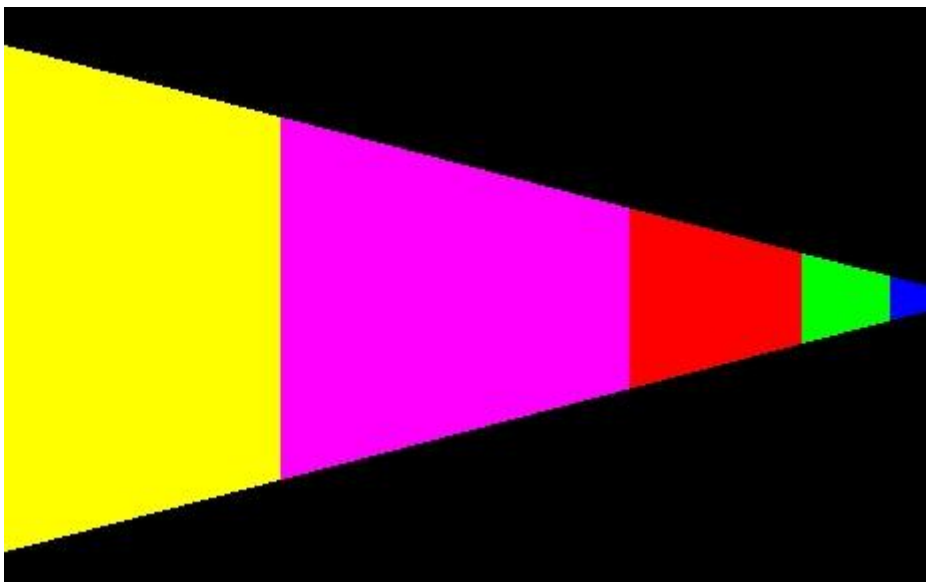
между собой. На практике несвязанные мипмапы могут сделать переходы между ними визуально весьма заметными, как на рисунке **9-7.)**

Чтобы задать все текстуры, вызовите команду **glTexImage2D()** по одному разу для каждого разрешения карты текстуры, каждый раз задавая новые значения для аргументов ***level***, ***width***, ***height*** и ***image***. Начиная с **0**, ***level*** идентифицирует, какая именно текстура в серии задается в текущий момент. В предыдущем примере тектура с самым высоким разрешением размера **64x16**, должна быть объявлена с аргументом ***level*=0**, текстура размера **32x8 –** с ***level*=1**, и так далее. Кроме того, чтобы мипмаппинг заработал, вам следует выбрать один из режимов фильтрации.

Замечание: В данном описании процесса мипмаппинга в OpenGL отсутствует подробное обсуждение фактора масштаба (известного как λ). Оно также приводится в предположении, что параметры GL_TEXTURE_MIN_LOD, GL_TEXTURE_MAX_LOD, GL_TEXTURE_BASE_LEVEL и GL_TEXTURE_MAX_LEVEL имеют значения по умолчанию. (Эти 4 параметра появились в OpenGL версии 1.2.) Объяснение фактора λ и воздействия параметров приводится далее в этой главе.

Пример **9-5** иллюстрирует использование серии из **6** текстурных карт, уменьшающихся от размера **32x32** до размера **1x1.** Эта программа рисует один длинный прямоугольник, который начинается на переднем плане и уходит вглубь до тех пор, пока не превращается в точку, как показано на рисунке **9-7.** Обратите внимание, что координаты текстуры ранжируются от **0.0** до **8.0**, таким образом, для покрытия всего прямоугольника требуется **64** копии текстуры **(**по **8** в каждом направлении**).** Для иллюстрации того, как одна карта текстуры продолжает другую, каждая из них имеет свой цвет.

Рисунок **9-7.** Пример применения мипмаппинга



Пример **9-5.** Мипмаппинг**:** файл **mipmap.cpp**

```
#include <glut.h>

GLubyte mipmapImage32[32][32][4];
GLubyte mipmapImage16[16][16][4];
GLubyte mipmapImage8[8][8][4];
GLubyte mipmapImage4[4][4][4];
GLubyte mipmapImage2[2][2][4];
GLubyte mipmapImage1[1][1][4];
```

```
GLuint texName;

void makeImages()
{
int i,j;

for (i=0;i<32;i++)
{
for (j=0;j<32;j++)
{
mipmapImage32[i][j][0]=255;
mipmapImage32[i][j][1]=255;
mipmapImage32[i][j][2]=0;
mipmapImage32[i][j][3]=255;
}
}
for (i=0;i<16;i++)
{
for (j=0;j<16;j++)
{
mipmapImage16[i][j][0]=255;
mipmapImage16[i][j][1]=0;
mipmapImage16[i][j][2]=255;
mipmapImage16[i][j][3]=255;
}
}
for (i=0;i<8;i++)
{
for (j=0;j<8;j++)
{
mipmapImage8[i][j][0]=255;
mipmapImage8[i][j][1]=0;
mipmapImage8[i][j][2]=0;
mipmapImage8[i][j][3]=255;
}
}
for (i=0;i<4;i++)
{
for (j=0;j<4;j++)
{
mipmapImage4[i][j][0]=0;
mipmapImage4[i][j][1]=255;
mipmapImage4[i][j][2]=0;
mipmapImage4[i][j][3]=255;
}
}
for (i=0;i<2;i++)
{
for (j=0;j<2;j++)
{
mipmapImage2[i][j][0]=0;
mipmapImage2[i][j][1]=0;
mipmapImage2[i][j][2]=255;
mipmapImage2[i][j][3]=255;
}
}
mipmapImage1[i][j][0]=255;
mipmapImage1[i][j][1]=255;
mipmapImage1[i][j][2]=255;
mipmapImage1[i][j][3]=255;
}
```

```
void init()
{
glEnable(GL_DEPTH_TEST);
glShadeModel(GL_FLAT);

glTranslatef(0.0,0.0,-3.6);
makeImages();
glPixelStorei(GL_UNPACK_ALIGNMENT,1);

glGenTextures(1,&texName);
glBindTexture(GL_TEXTURE_2D,texName);
glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_WRAP_S,GL_REPEAT);
glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_WRAP_T,GL_REPEAT);
glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MAG_FILTER,GL_NEAREST);
glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MIN_FILTER,
GL_NEAREST_MIPMAP_NEAREST);

glTexImage2D(GL_TEXTURE_2D,0,GL_RGBA,32,32,0,
GL_RGBA,GL_UNSIGNED_BYTE,mipmapImage32);
glTexImage2D(GL_TEXTURE_2D,1,GL_RGBA,16,16,0,
GL_RGBA,GL_UNSIGNED_BYTE,mipmapImage16);
glTexImage2D(GL_TEXTURE_2D,2,GL_RGBA,8,8,0,
GL_RGBA,GL_UNSIGNED_BYTE,mipmapImage8);
glTexImage2D(GL_TEXTURE_2D,3,GL_RGBA,4,4,0,
GL_RGBA,GL_UNSIGNED_BYTE,mipmapImage4);
glTexImage2D(GL_TEXTURE_2D,4,GL_RGBA,2,2,0,
GL_RGBA,GL_UNSIGNED_BYTE,mipmapImage2);
glTexImage2D(GL_TEXTURE_2D,5,GL_RGBA,1,1,0,
GL_RGBA,GL_UNSIGNED_BYTE,mipmapImage1);

glTexEnvf(GL_TEXTURE_ENV,GL_TEXTURE_ENV_MODE,GL_REPLACE);
glEnable(GL_TEXTURE_2D);
}

void display()
{
glClear(GL_COLOR_BUFFER_BIT|GL_DEPTH_BUFFER_BIT);
glBindTexture(GL_TEXTURE_2D,texName);

glBegin(GL_QUADS);
glTexCoord2f(0.0,0.0); glVertex3f(-2.0,-1.0,0.0);
glTexCoord2f(0.0,8.0); glVertex3f(-2.0,1.0,0.0);
glTexCoord2f(8.0,8.0); glVertex3f(2000.0,1.0,-6000.0);
glTexCoord2f(8.0,0.0); glVertex3f(2000.0,-1.0,-6000.0);
glEnd();
glFlush();
}

void reshape(int w,int h)
{
glViewport(0,0,w,h);
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
gluPerspective(60.0,w/h,1.0,30000);
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
}

int main(int argc, char **argv)
{
glutInit(&argc,argv);
glutInitDisplayMode(GLUT_SINGLE|GLUT_RGB|GLUT_DEPTH);
glutInitWindowSize(500,500);
```

```
glutInitWindowPosition(100,100);
glutCreateWindow("Mipmap Textures");
init();
glutDisplayFunc(display);
glutReshapeFunc(reshape);
glutMainLoop();
return 0;
}
```

Пример **9-5** иллюстрирует мипмаппинг с помощью мипмапов разного цвета. Таким образом, точки, в которых одна карта переходит в другую, очевидны. В реальных условиях мипмапы обычно создаются таким образом, чтобы переходы между ними были настолько плавными и незаметными, насколько это возможно. Таким образом, карты меньшего размера обычно представляют собой отфильтрованные и уменьшенные копии карты с большим разрешением. Построение таких изображений – это программный процесс и, следовательно, он не является частью **OpenGL**. Однако, поскольку конструирование мипмапов – это очень важный процесс, библиотека утилит **GLU** содержит три функции, которые помогают в манипуляциях с изображениями, используемыми в качестве мипмапов.

9.2.6.1 Автоматическое создание мипмапов

Предполагая, что вы построили текстурную карты с высочайшим разрешением уровня **0**, функции **gluBuild1DMipmaps()**, **gluBuild2DMipmaps()** и **gluBuild3DMipmaps()** конструируют и создают пирамиду мипмапов до разрешения **1x1** (или **1** для одномерной текстуры, или **1x1x1** для трехмерной). Если размеры вашего изображения не являются степенями **2**, функции **gluBuild*DMipmaps()** масштабируют его до ближайших степеней **2**. Кроме того, если ваша текстура слишком велика, **gluBuild*DMipmaps()** сократит ее до допустимого размера (с помощью механизма текстурного прокси).

```
int gluBuild1DMipmaps (GLenum target, GLint internalFormat, GLint width, GLenum
format, GLenum type, void *texels);
int gluBuild2DMipmaps (GLenum target, GLint internalFormat, GLint width, GLint
height, GLenum format, GLenum type, void *texels);
int gluBuild3DMipmaps (GLenum target, GLint internalFormat, GLint width, GLint
height,
                       GLint depth, GLenum format, GLenum type, void *texels);
```

Конструирует серию мипмапов и вызывает **glTexImage*D()** для загрузки изображений. Назначение аргументов ***target***, ***internalFormat***, ***width***, ***height***, ***depth***, ***format***, ***type*** и ***texels*** в точности соответствует тем, что передаются в **glTexImage1D()**, **glTexImage2D()** и **glTexImage3D()**. Если все мипмапы сконструированы успешно, функция возвращает **0**. В случае неудачи возвращается код ошибки **GLU**.

9.2.6.2 Дополнительные детали мипмаппинга

Расчет мипмаппинга зависит от фактора масштаба между изображением текстуры и размером текстурируемого полигона (в пикселях). Давайте назовем этот фактор ρ и определим вторую величину -- λ, причем $\lambda = \log_2 \rho$. (Поскольку изображения текстуры могут лежать в нескольких измерениях, важно прояснить, что ρ -- это максимальный фактор масштаба во всех измерениях.) Если $\lambda \leq 0.0$, то текстура меньше, чем полигон, и используется увеличивающий фильтр. Если же $\lambda > 0.0$, используется уменьшающий фильтр. Если выбранный уменьшающий фильтр использует мипмаппинг, то λ означает уровень используемого мипмапа.

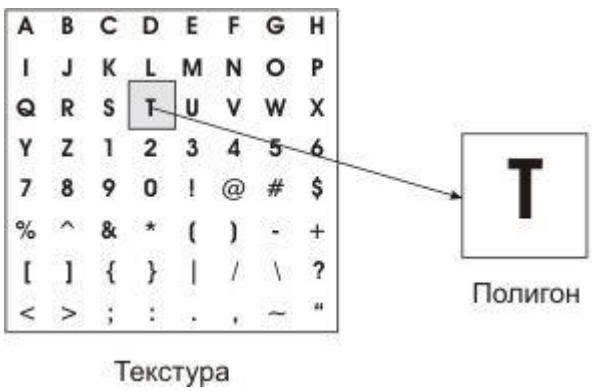
Например, если размер изображения текстуры **64x64** тэкселя, а размер полигона **-- 32x32** пикселя, то $\rho = 2.0$ **(**а не **4.0)** и $\lambda = 1.0$**.** Если изображение текстуры **64x32** тэкселя, а размер полигона **-- 8x16** пикселей, то $\rho = 8.0$ **(**масштаб по **x** равен **8.0**, а по **y –2.0)** и $\lambda = 3.0$**.**

Замечание: Точкой равновесия между применением уменьшающего фильтра и увеличивающего обычно является $\lambda = 0.0$, но это не всегда так. Если в качестве увеличивающего фильтра выбран GL_LINEAR, а в качестве уменьшающего – GL_NEAREST_MIPMAP_NEAREST или GL_NEAREST_MIPMAP_LINEAR, то эта точка находится в $\lambda = 0.5$. Это предотвращает ситуацию, когда уменьшенная текстура выглядит более резкой, чем ее увеличенная версия.

До **OpenGL** версии **1.2** применение мипмаппинга накладывало на программиста дополнительные ограничения. Например, нельзя было «на лету» производить такие изменения, как добавление новых мипмапов. Кроме того, в течение процесса уменьшения или увеличения полигона, текстура могла изменяться скачками в точках замены одного мипмапа другим, разрешение которого радикально выше.

В добавление к этому приходилось предоставлять мипмапы для всех уровней разрешения, даже для невероятно маленьких. При применении некоторых техник следует избегать представления данных в виде очень маленьких мипмапов. Например, вам может пригодиться техника *мозаики*, когда несколько небольших изображений помещены на одной текстуре. Один из примеров мозаики показан на рисунке **9-8**, где изображен набор символов, помещенный на одной текстуре. использование мозаики в данном случае более эффективно, чем создание отдельной текстуры для каждого символа. Для наложения на полигон одной буквы, вам следует произвести расчет координат текстуры, дабы выделить эту букву из всего изображения.

Рисунок **9-8.** Использование техники «мозаика»



Однако, если вы предоставите действительно маленькие мипмапы для мозаики, то на текстурах с низким разрешением детали из нескольких букв будут сливаться. В **OpenGL** версии **1.2** вы можете установить ограничение на самое низкое разрешение.

В **OpenGL 1.2** появились новые параметры для управления уровнями мипмаппинга: **GL_TEXTURE_MIN_LOD**, **GL_TEXTURE_MAX_LOD**, **GL_TEXTURE_BASE_LEVEL** и **GL_TEXTURE_MAX_LEVEL.** Первые два параметра **(**будем для краткости называть их **BASE_LEVEL** и **MAX_LEVEL)** управляют тем, какие уровни мипмаппинг используются и, следовательно, тем, какие из них должны быть предоставлены. Оставшиеся два параметра **(MIN_LOD** и **MAX_LOD)** управляют активным диапазоном упоминавшегося ранее фактора масштаба λ**.**

BASE_LEVEL и **MAX_LEVEL** используются для управления тем, какой диапазон мипмапов следует использовать. **BASE_LEVEL –** это уровень детализации с максимально высоким разрешением. Значением по умолчанию для **BASE_LEVEL** является **0.** Однако, впоследствии вы можете изменять это значение, если вы добавляете мипмап с еще более высоким разрешением **«**на лету**».** Похожим образом **MAX_LEVEL** ограничивает диапазон сверху, не давая использовать мипмапы с уровнем детализации больше **MAX_LEVEL (**то есть, не давая использовать мипмапы с более низкими разрешениями**).**

Для того, чтобы мипмаппинг заработал, следует загрузить все мипмапы между **BASEE_LEVEL** и **MAX_LEVEL.** Самый большой возможный уровень **–** это наименьшая из двух величин: **MAX_LEVEL** или уровень детализации, на котором мипмап содержит только **1** тэксель. Значение по умолчанию для **MAX_LEVEL – 1000,** что почти всегда означает, что наименьшее разрешение текстуры **– 1** тэксель.

Замечание: Если вы забудете загрузить один из необходимых мипмапов, текстурирование может таинственно деактивироваться. Если не работают ни мипмаппинг, ни просто текстурирование, проверьте загрузили ли вы все нужные мипмапы.

Чтобы установить базовый и максимальный уровни мипмапов, используйте команду **glTexParameter*()** с первым аргументом, установленным в **GL_TEXTURE_1D, GL_TEXTURE_2D** или **GL_TEXTURE_3D,** в зависимости от ваших текстур. Второй аргумент должен быть одним из параметров, описанных в таблице **9-1.** Третий аргумент представляет собой значение параметра.

Таблица **9-1.** Параметры управления уровнями мипмаппинга

Параметр	**Описание**	**Значения**
GL_TEXTURE_BASE_LEVEL	уровень текстуры с наивысшим используемым разрешением **(**наименьшее значение уровня по номеру**)**	любое неотрицательное целое
GL_TEXTURE_MAX_LEVEL	уровень текстуры с наименьшим используемым разрешением **(**наибольшее значение уровня по номеру**)**	любое неотрицательное целое

Код в примере **9-6** устанавливает базовый и максимальный уровни мипмапов в **2** и **5** соответсвенно. Поскольку изображение на базовом уровне **(**уровне **2)** имеет разрешение **64x32** тэкселя, мипмапы на уровнях **3, 4** и **5** должны иметь меньшее разрешение.

Пример **9-6.** Установки базового и максимального уровней мипмапов

```
glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_BASE_LEVEL,2);
glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MAX_LEVEL,5);
glTexImage2D(GL_TEXTURE_2D,2,GL_RGBA, 64,32,0,
GL_RGBA, GL_UNSIGNED_BYTE, image1);
glTexImage2D(GL_TEXTURE_2D,3,GL_RGBA, 32,16,0,
GL_RGBA, GL_UNSIGNED_BYTE, image2);
glTexImage2D(GL_TEXTURE_2D,4,GL_RGBA, 16,8,0,
GL_RGBA, GL_UNSIGNED_BYTE, image3);
glTexImage2D(GL_TEXTURE_2D,5,GL_RGBA, 8,4,0,
GL_RGBA, GL_UNSIGNED_BYTE, image4);
```

Позже в этой программе вам может понадобиться добавить дополнительные мипмапы с большим или меньшим разрешением. Например, вы можете добавить в этот набор текстуру с разрешением **128x64** тэкселя на уровне **1.** Однако не забудьте изменить значение параметра **BASE_LEVEL.**

9.2.6.3 Управление уровнем детализации мипмапа

MIN_LOD и **MAX_LOD** представляют минимальное и максимальное значения для λ **(**фактора масштаба текстурного изображения к полигону**)** для минификации и неявно задают**,** какой из мипмапов следует использовать**.**

Если ваш полигон занимает **64xx64** пикселя**,** и **MIN_LOD** равно своему значению по умолчанию**,** то есть **0.0,** то текстурная карта уровня **0** с разрешением **64x64** тэкселя может быть использована для минификации **(**заданный **BASE_LEVEL=0;** как правило**, BASE_LEVEL <= MIN_LOD).** Однако**,** если **MIN_LOD** равен **2.0,** то самая большая текстура**,** которая может быть использована для минификации имеет разрешение **16x16** тэкселей**,** что соответствует $\lambda = 2.0$**.**

MAX_LOD имеет влияние только в том случае**,** если он меньше максимального λ **(**которое равно либо **MAX_LEVEL,** либо уровню**,** на котором текстура имеет размер в **1** тэксель**).** В случае карты текстуры размером **64x64** тэкселя**,** $\lambda = 6.0$ соответствует мипмапу размером **1x1** тэксель**.** Если в той же ситуации **MAX_LOD** равен **4.0,** то ни один мипмап меньше чем **4x4** тэкселя не будет использоваться для минификации**.**

Избирательное использование **MIN_LOD** может уменьшить количество визуальных артефактов на текстурах с высоким разрешением**,** а использование **MAX_LOD –** на текстурах с низким**.** Вы обнаружите**,** что использование **MIN_LOD** незначительно большего **BASE_LEVEL** и **MAX_LOD** незначительно меньшего **MAX_LEVEL** дает наилучшие результаты по сокращению визуальных эффектов**,** связанных с переходами между мипмапами**.**

Как и в случае с **BASE_LEVEL** и **MAX_LEVEL,** для управления параметрами **MAX_LOD** и **MIN_LOD** применяется команда **glTexParameter*().** Возможные значения перечислены в таблице **9-2.**

Таблица **9-2.** Параметры управления уровнями детализации мипмаппинга

Параметр	Описание	Значения
GL_TEXTURE_MIN_LOD	минимальная величина для λ	любое значение
GL_TEXTURE_MAX_LOD	максимальная величина для λ	любое значение

Следующий код иллюстрирует пример установки параметров управления уровнями детализации**.**

```
glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MIN_LOD,2.5);
glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MAX_LOD,4.5);
```

9.2.6.4 Автоматическое создание подмножества мипмапов

При использовании описанного контроля над уровнями мипмапов**,** вам может понадобиться конструирование только некоторого их подмножества**. (**Например**,** вы возможно захотите остановиться на изображении размером **4x4** тэкселя**,** а не идти до наименьшего мипмапа размером **1x1).** Чтобы сконструировать и загрузить подмножество мипмапов**,** вы можете использовать функцию **gluBuild*DMipmapLevels().**

```
int gluBuild1DMipmapLevels (GLenum target, GLint internalFormat, GLint width,
GLenum format,
```

```
                                GLenum type, GLint level, GLint base, GLint max, void
*texels);
int gluBuild2DMipmapLevels (GLenum target, GLint internalFormat, GLint width,
GLenum height, GLenum format,
                                GLenum type, GLint level, GLint base, GLint max, void
*texels);
int gluBuild3DMipmapLevels (GLenum target, GLint internalFormat, GLint width,
GLenum height, GLenum depth,
                                GLenum format, GLenum type, GLint level, GLint base,
GLint max, void *texels);
```

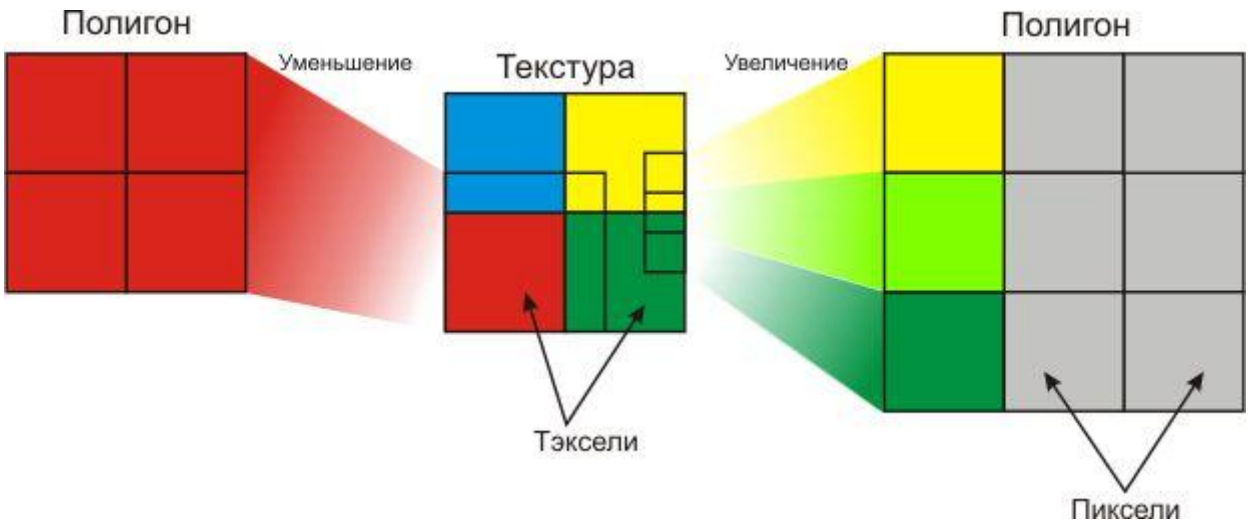
Конструируют серию мипмапов и вызывают **glTexImage*D()** для загрузки изображений**.** ***level*** индицирует уровень мипмапа изображения**,** заданного в аргументе ***texels*****.** Аргументы ***base*** и ***max*** задают нижнюю и верхнюю границы диапазона уровней мипмапов**,** которые будут произведены из изображения**,** заданного аргументом ***texels*****.** Остальные аргументы **–** ***target***, ***internalFormat***, ***width***, ***height***, ***depth***, ***format*** и ***type*** **–** имеют то же значение**,** что и для команд **glTexImage1D()**, **glTexImage2D()** и **glTexImage3D()****.** В случае успешного выполнения функции**,** она возвращает **0.** Функция возвращает код ошибки **GLU** в случае неудачи**.**

Замечание: В GLU версии 1.3 функции **gluBuild*DMipmaps()** и **gluBuild*DMipmapLevels()** поддерживают упакованные форматы пикселей (и связанные с ними типы данных), представленные в OpenGL 1.2.

9.3 Фильтрация

Текстурные карты бывают квадратными или прямоугольными**,** но после наложения на полигон или поверхность и трансформации в экранные координаты**,** индивидуальные тэксели текстуры редко соответствуют индивидуальным пикселям на экране**.** В зависимости от используемых преобразований и параметров текстурирования один пиксель на экране может соответствовать чему угодно от части тэкселя **(**увеличение**)** до большой группы тэкселей **(**уменьшение**),** как показано на рисунке **9-9.** В любом случае не совсем понятно**,** какие тэксели будут использоваться и как они должны интерполироваться**. OpenGL** позволяет вам выбирать одну из нескольких опций фильтрации**,** определяющих процесс означенных вычислений**.** Эти опции весьма варьируются по соотношениям скорость**/**качество изображения**.** Вы можете задавать режимы фильтрации отдельно и независимо для увеличения и уменьшения**.**

Рисунок **9-9.** Увеличение и уменьшение текстуры



В некоторых случаях ответ на вопрос о том**,** что использовать**:** увеличение или уменьшение**,** не является очевидным**.** Если текстурная карта должна быть растянута или сжата в обоих направлениях **(x** и **y),** то требуется увеличение или уменьшение

соответственно. Если же карту текстуры требуется растянуть в одном направлении и сжать в другом, **OpenGL** делает свой выбор между увеличением и уменьшением, и этот выбор в большинстве случаев дает наилучший результат. Однако лучше избегать ситуаций, используя координаты текстуры, позволяющие наложить ее на объект без подобных искажений.

Следующие строки являются примерами того, как использовать команду **glTexParameter*()** для выбора методов увеличивающей и уменьшающей фильтрации:

```
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
```

Первый аргумент команды должен быть установлен в **GL_TEXTURE_1D**, **GL_TEXTURE_2D** или **GL_TEXTURE_3D** в зависимости от вашей текстуры. Для данного обсуждения второй аргумент должен быть установлен либо в **GL_TEXTURE_MAG_FILTER,** либо в **GL_TEXTURE_MIN_FILTER,** определяя, задаете ли вы метод фильтрации для увеличения или для уменьшения. Третий аргумент задает метод фильтрации. Все возможные методы перечислены в таблице **9-3.**

Таблица **9-3.** Методы фильтрации при увеличении и уменьшении

Параметр	Значения
GL_TEXTURE_MAG_FILTER	GL_NEAREST, GL_LINEAR
GL_TEXTURE_MIN_FILTER	GL_NEAREST, GL_LINEAR, GL_NEAREST_MIPMAP_NEAREST, GL_NEAREST_MIPMAP_LINEAR, GL_LINEAR_MIPMAP_NEAREST, GL_LINEAR_MIPMAP_LINEAR

Если вы выберите **GL_NEAREST,** то для увеличения и уменьшения будет использоваться тэксель с ближайшими к центру пикселя координатами. Это может привести к нежелательным визуальным эффектам **(**иногда серьезным**).** Если же вы выберите **GL_LINEAR,** то для увеличения или уменьшения будет использоваться взвешенная сумма массива тэкселей **2x2,** которые находятся ближе всего к центру пикселя. **(**В случае трехмерных текстур используется массив размерностью **2x2x2,** а в случае одномерных используются два соседних пикселя.**)** Когда координаты текстуры близки к границе текстурной карты, в массив ближайших тэкселей могут попасть те из них, которые находятся все карты. В таких случаях величины используемых тэкселей зависят от того, какой из режимов **GL_REPEAT**, **GL_CLAMP** или **GL_CLAMP_TO_EDGE** находится в действии, а также от того, обозначили ли вы границу для текстуры. **GL_NEAREST** требует меньше расчетов, чем **GL_LINEAR** и, таким образом, может выполняться быстрее, однако **GL_LINEAR** дает визуально лучшие результаты.

При увеличении, независимо от того, предоставили ли вы мипмапы, всегда используется текстура базового уровня. При уменьшении вы можете выбрать метод фильтрации, который использует один или два наиболее подходящих мипмапа, как описано в следующем абзаце. **(**Если для уменьшения заданы методы **GL_NEAREST** или **GL_LINEAR,** то опять таки будет использована только текстура базового уровня.**)**

Как показано в таблице **9-3,** для уменьшения с применением мипмапов доступно **4** дополнительных метода фильтрации. Внутри индивидуального мипмапа вы можете выбрать ближайший тэксель **(GL_NEAREST_MIPMAP_NEAREST)** или линейную интерполяцию **(GL_LINEAR_MIPMAP_NEAREST).** Использование ближайших тэкселей быстрее, но приводит к недостаточно качественным результатам. Выбор того, какой конкретный мипмап использовать зависит от того, какое количество уменьшения требуется; существуют граничные значения, на которых производится переход от одного мипмапа к другому. Чтобы избежать неожиданных переходов, используйте **GL_NEAREST_MIPMAP_LINEAR** или **GL_LINEAR_MIPMAP_LINEAR** для линейной интерполяции величин тэкселей из двух ближайших мипмапов. **GL_NEAREST_MIPMAP_LINEAR** выбирает ближайшие тэксели в обоих мипмапах, а затем

производит линейную интерполяцию между их величинами. **GL_LINEAR_MIPMAP_LINEAR** производит линейную интерполяцию для вычисления величины тэкселей двух карт, а затем интерполирует еще и между ними. Как вы можете предположить, использование режима **GL_LINEAR_MIPMAP_LINEAR** дает лучшие визуальные результаты, но требует больших вычислений и работает медленнее.

Замечание: Если вы выберите один из фильтров, использующих мипмапы, но забудете загрузить один из необходимых мипмапов, OpenGL не выдаст никаких ошибок, но деактивирует текстурирование. Если не работают ни мипмаппинг, ни просто текстурирование, проверьте загрузили ли вы все нужные мипмапы.

Некоторые из описанных фильтров известны под более популярными именами. **GL_NEAREST** часто называется *точечной фильтрацией* (***point sampling***). **GL_LINEAR** известен как *билинейная фильтрация* (***bilinear sampling***), поскольку для двумерных текстур используется массив тэкселей размерности **2x2**. **GL_LINEAR_MIPMAP_LINEAR** иногда называют *трилинейной фильтрацией* (***trilinear sampling***), поскольку используется линейное среднее между двумя билинейно фильтрованными мипмапами.

9.4 Текстурные объекты

Текстурные объекты хранят данные текстуры и позволяют ими управлять. Вы можете контролировать одновременно несколько текстур и возвращаться к текстурам, которые были загружены ранее. Использование текстурных объектов обычно быстрейший способ работы с текстурами, что выражается в большом выигрыше по производительности, поскольку переключиться на существующую текстуру (повторно использовать ее) можно гораздо быстрее, чем загружать ее заново командой **glTexImage*D()**.

Кроме того, некоторые реализации поддерживают ограниченное рабочее подмножество быстродействующих текстур. Вы можете использовать текстурные объекты для загрузки самых часто используемых текстур в эту ограниченную область.

Чтобы использовать объекты текстуры для данных текстуры выполните следующие шаги.

- Сгенерируйте имена текстур.
- Привяжите объекты текстуры к данным текстуры (в частности к массивам изображений и свойствам).
- Если ваша реализация поддерживает рабочее подмножество быстродействующих текстур, проверьте, достаточно ли у вас пространства для всех ваших текстурных объектов. Если пространство недостаточно, возможно вам стоит установить приоритет для каждого текстурного объекта, чтобы наиболее часто используемые текстуры оставались в рабочем подмножестве.
- Выбирайте (повторно связывайте) ваши текстурные объекты, делая их текущими для визуализации текстурированных моделей.

Замечание: Текстурные объекты появились в OpenGL версии 1.1.

9.4.1 Именование текстурных объектов

В качестве имени текстуры может быть использовано любое ненулевое беззнаковое целое. Чтобы избежать случайного повторного использования имен, постоянно используйте команду **glGenTextures()** для получения неиспользуемых имен текстур.

```
void glGenTextures (GLsizei n, GLuint *textureNmes);
```

Возвращает ***n*** неиспользуемых имен текстурных объектов в массиве ***textureNames***. Имена, возвращаемые в массиве ***textureNames***, не обязаны составлять один непрерывный интервал. Возвращенные имена помечаются как используемые, однако следует помнить, что привязка имени к состоянию и размерности текстуры **(1D**, **2D** или **3D)** осуществляется только в момент его первой выдачи. Ноль – это зарезервированное имя текстуры, и оно никогда не возвращается командой **glGenTextures()**.

Команда **glIsTexture()** определяет, используется ли конкретное имя текстуры в данный момент. Если имя текстуры было возвращено командой **glGenTextures()**, но еще не было связано с данными **(**хотя бы однократным вызовом команды **glBindTexture())**, команда **glIsTexture()** возвратит **GL_FALSE**.

```
void glIsTexture (GLuint textureName);
```

Возвращает **GL_TRUE**, если ***textureName*** является именем текстуры, которое было связано и впоследствии не было удалено. Возвращает **GL_FALSE** в случае, если ***textureName*** равно **0**, или если оно не равно **0**, но не является именем существующей текстуры.

9.4.2 Создание и использование текстурных объектов

Одна и та же команда **glBindTexture()** используется и при создании, и при использовании текстурных объектов. При начальном связывании **(**с использованием **glBindTexture())** создается новый текстурный объект с величинами по умолчанию для изображения текстуры и ее свойств. Последующие обращения к командам **glTexImage*()**, **glTexSubImage*()**, **glCopyTexImage*()**, **glCopyTexSubImage*()**, **glTexParameter*()** и **glPrioritizeTextures()** сохраняются данные в текстурном объекте. Объект текстуры может содержать изображение текстуры и ассоциированные с ним мипмапы **(**если они присутствуют**)**, а также ассоциированные с текстурой данные, такие как ширина, высота, ширина границы, внутренний формат, разрешение компонент и свойства текстуры. Сохраняемые свойства текстуры включают уменьшающий и увеличивающий фильтры, режим наложения, цвет границы и приоритет.

Когда объект текстуры связывается впоследствии, данные, содержащиеся в нем, становятся текущим состоянием текстуры, замещая предыдущее состояние.

```
void glBindTexture (GLenum target, GLuint textureName);
```

glBindTexture() делает три вещи. При использовании в качестве ***textureName*** беззнакового целого неравного **0** в первый раз, создается новый объект текстуры, и переданное имя текстуры ассоциируется с ним. Если привязка осуществляется к предварительно созданному объекту текстуры, то этот объект становится текущим. Когда связывание производится при ***textureName*** равном **0**, **OpenGL** перестает использовать объекты текстуры и возвращается к безымянной текстуре по умолчанию. При начальном связывании текстурного объекта **(**то есть при его создании**)** с ним ассоциируется размерность текстуры, то есть **GL_TEXTURE_1D**, **GL_TEXTURE_2D** или **GL_TEXTURE_3D**, задаваемая в аргументе ***target***. Сразу после начального связывания состояние текстурного объекта совпадает с состоянием **GL_TEXTURE_1D**, **GL_TEXTURE_2D** или **GL_TEXTURE_3D (**в зависимости от размерности текстуры**)** по умолчанию при инициализации **OpenGL**. В начальном состоянии свойства текстуры, такие как уменьшающий и увеличивающий фильтры, режим наложения, цвет границы и приоритет установлены в свои значения по умолчанию.

В примере **9-7** в функции **init()** создаются **2** текстурных объекта. В функции **display()** каждый из объектов используется для визуализации одного четырехстороннего полигона. Результат работы программы изображен на рисунке **9-10**.

Рисунок **9-10.** Два полигона текстурированных с помощью объектов текстуры

```
#include <glut.h>

#define checkImageWidth 64
#define checkImageHeight 64

GLubyte checkImage[checkImageHeight][checkImageWidth][4];
GLubyte otherImage[checkImageHeight][checkImageWidth][4];

GLuint texName[2];

//Создание изображения текстуры
void makeCheckImage()
{
int i,j,c;

for(i=0;i<checkImageHeight;i++)
{
for(j=0;j<checkImageWidth;j++)
{
c=(((i&0x8)==0)^((j&0x8)==0))*255;
checkImage[i][j][0]=(GLubyte)c;
checkImage[i][j][1]=(GLubyte)0;
checkImage[i][j][2]=(GLubyte)0;
checkImage[i][j][3]=(GLubyte)1.0;
c=(((i&0x10)==0)^((j&0x10)==0))*255;
otherImage[i][j][0]=(GLubyte)0;
otherImage[i][j][1]=(GLubyte)c;
otherImage[i][j][2]=(GLubyte)0;
otherImage[i][j][3]=(GLubyte)1.0;
}
}
}

void init(void)
{
glClearColor(1.0,1.0,1.0,0.0);
glShadeModel(GL_FLAT);
glEnable(GL_DEPTH_TEST);

makeCheckImage();
glPixelStorei(GL_UNPACK_ALIGNMENT,1);

glGenTextures(2,texName);

glBindTexture(GL_TEXTURE_2D,texName[0]);
glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_WRAP_S, GL_CLAMP);
glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_WRAP_T, GL_CLAMP);
glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MAG_FILTER, GL_NEAREST);
glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MIN_FILTER, GL_NEAREST);
glTexImage2D(GL_TEXTURE_2D,0,GL_RGBA,checkImageWidth,checkImageHeight,
0,GL_RGBA,GL_UNSIGNED_BYTE,checkImage);

glBindTexture(GL_TEXTURE_2D,texName[1]);
glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_WRAP_S, GL_CLAMP);
glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_WRAP_T, GL_CLAMP);
glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MAG_FILTER, GL_NEAREST);
glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MIN_FILTER, GL_NEAREST);
glTexEnvf(GL_TEXTURE_ENV,GL_TEXTURE_ENV_MODE,GL_REPLACE);
glTexImage2D(GL_TEXTURE_2D,0,GL_RGBA,checkImageWidth,checkImageHeight,
0,GL_RGBA,GL_UNSIGNED_BYTE,otherImage);
```

```
glEnable(GL_TEXTURE_2D);
}

void display(void)
{
glClear (GL_COLOR_BUFFER_BIT|GL_DEPTH_BUFFER_BIT);
glBindTexture(GL_TEXTURE_2D,texName[0]);
glBegin(GL_QUADS);
glTexCoord2f(0.0,0.0); glVertex3f(-2.0,-1.0,0.0);
glTexCoord2f(0.0,1.0); glVertex3f(-2.0,1.0,0.0);
glTexCoord2f(1.0,1.0); glVertex3f(0.0,1.0,0.0);
glTexCoord2f(1.0,0.0); glVertex3f(0.0,-1.0,0.0);
glEnd();

glBindTexture(GL_TEXTURE_2D,texName[1]);
glBegin(GL_QUADS);
glTexCoord2f(0.0,0.0); glVertex3f(1.0,-1.0,0.0);
glTexCoord2f(0.0,1.0); glVertex3f(1.0,1.0,0.0);
glTexCoord2f(1.0,1.0); glVertex3f(2.41421,1.0,-1.41421);
glTexCoord2f(1.0,0.0); glVertex3f(2.41421,-1.0,-1.41421);
glEnd();
glFlush();
}

void reshape(int w, int h)
{
glViewport(0,0,(GLsizei) w, (GLsizei) h);
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
if (w<=h)
glOrtho(-3.0,3.0,-3.0*(GLfloat)h/(GLfloat)w,
3.0*(GLfloat)h/(GLfloat)w,-10.0,10.0);
else
glOrtho(-3.0*(GLfloat)w/(GLfloat)h,
3.0*(GLfloat)w/(GLfloat)h,-3.0,3.0,-10.0,10.0);
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
}

int main(int argc, char **argv)
{
glutInit(&argc,argv);
glutInitDisplayMode(GLUT_SINGLE|GLUT_RGB|GLUT_DEPTH);
glutInitWindowSize(620,620);
glutInitWindowPosition(100,100);
glutCreateWindow("Binding Texture Objects");
init();
glutDisplayFunc(display);
glutReshapeFunc(reshape);
glutMainLoop();
return 0;
}
```

Если текстурный объект становится активным **(**то есть вторично связывается**),** вы можете редактировать его содержимое**.** Любые вызываемые вами команды**,** изменяющие изображение текстуры или ее свойства**,** будут изменять содержимое текущего текстурного объекта**.**

В примере **9-7** после завершения функции **display()** вы все еще связаны с текстурой имеющей имя ***texName[1]*****.** Будьте осторожны**:** если вы спонтанно вызовите какую**-**либо команду**,** изменяющую текущее состояние текстуры**,** она также изменит и содержимое текущего текстурного объекта**.**

При использовании мипмапов, все мипмапы, связанные с одним текстурным изображением, должны быть помещены в один текстурный объект. В примере **9-5** уровни мипмапов **0-5** помещаются в один текстурный объект именуемый ***texName***.

9.4.3 Очистка текстурных объектов

В то время как вы производите первичное или вторичное связывание текстурных объектов, их данные по-прежнему находятся где-то в текстурных ресурсах. Если эти ресурсы ограничены, удаление текстур является единственным способом освобождения памяти.

```
void glDeleteTextures (GLsizei n, const GLuint *textureNames);
```

Удаляет ***n***текстурных объектов, чьи имена переданы в аргументе ***textureNames***. Освобожденные имена могут быть повторно использованы (например, возвращены командой **glGenTextures()**). Если удаляется текущая текстура, состояние переходит к текстуре по умолчанию, как если бы была вызвана команда **glBindTextures()** с аргументом ***textureName*** равным **0**. Попытки удаления имен несуществующих текстур игнорируются без генерации каких-либо ошибок.

9.4.4 Рабочее подмножество резидентных текстур

Некоторые реализации **OpenGL** поддерживают рабочее подмножество быстродействующих текстур, о которых говорят, что они резидентны. Обычно, в этих реализациях имеется специализированная аппаратура для выполнения текстурных операций и ограниченный аппаратный кэш для хранения изображений текстур. В таких случаях рекомендуется использовать текстурные объекты, поскольку вы можете загрузить несколько текстур в рабочее подмножество и управлять ими.

Если все текстуры, требуемые приложению, превышают размер кэша, некоторые из них не могут быть резидентными. Если вам нужно выяснить, является ли конкретная текстура резидентной, сделайте ее объект текущим, а затем используйте команду **glGetTexParameter*v()**, чтобы выяснить значение величины, связанной с переменной состояния **GL_TEXTURE_RESIDENT**. Если вам нужно выяснить резидентность состояния нескольких текстур, используйте команду **glAreTexturesResident()**.

```
GLboolean glAreTexturesResident (GLsizei n, const GLuint *textureNames, GLboolean
*residences);
```

Запрашивает текстурный статус резидентности ***n***текстурных объектов, чьи имена передаются в массиве ***textureNames***. ***residences*** – это массив, в котором статусы резидентности возвращаются для соответствующих текстурных объектов в массиве ***textureNames***. Если все текстуры, переданные в массиве ***textureNames***, являются резидентными, команда возвращает значение **GL_TRUE**, а содержимое массива ***residences*** не изменяется. Если хотя бы одна из текстур в ***textureNames***не является резидентной, команда возвращает **GL_FALSE**, и элементы массива ***residences***, соответствующие нерезидентным текстурам из массива ***textureNames***, также устанавливаются в **GL_FALSE**.

Обратите внимание на то, что **glAreTexturesResident()** возвращает текущий резидентный статус. Текстурные ресурсы очень динамичны, и этот статус может изменяться в любое время. Некоторые реализации кэшируют текстуры в момент их первого использования. Возможно, понадобится нарисовать хотя бы один текстурированный объект, а уже затем проверять резидентность его текстуры.

Если ваша реализация **OpenGL** не поддерживает рабочего подмножества текстур, то все текстурные объекты всегда считаются резидентными. В этом случае **glAreTexturesResident()**всегда будет возвращать **GL_TRUE.**

9.4.4.1 Стратегии резидентности текстур

Если вы можете создать рабочее подмножество текстур и хотите получить наилучшее возможное быстродействие при работе с текстурами, вам действительно нужно знать специфику вашей реализации и приложения. Например, если вы создаете визуальный симулятор или компьютерную игру, вам нужно следить за быстродействием во всех ситуациях. В этом случае вам никогда не следует обращаться к нерезидентной текстуре. Для таких приложений вам нужно загрузить все ваши текстуры на этапе инициализации и обеспечить их резидентность. Если вам не хватает текстурной памяти, возможно, вам следует сократить размер, разрешение или количество мипмапов для ваших текстур или использовать **glTexSubImage*()** для повторного использования одной и той же текстурной памяти.

Замечание: Если у вас есть несколько текстур одинакового размера, которые нужны лишь на короткий период, вы можете использовать **glTexSubImage*()** для загрузки новых данных в существующие объекты текстуры. Эта техника может работать быстрее, чем удаление и создание новых текстур.

Для приложений, которые создают текстуры «на лету», избежание нерезидентных текстур может быть трудной задачей. Если определенные текстуры используются чаще, чем другие, вы можете присвоить их текстурным объектам более высокий приоритет, что повысит их шансы на резидентность. Удаление текстурных объектов также освобождает пространство. Говоря коротко, присвоение текстурному объекту низкого приоритета может сделать его первым в очереди на удаление из рабочего подмножества. Для назначения приоритетов используется команда **glPrioritizeTextures().**

```
void glPrioritizeTextures (GLsizei n, const GLuint *textureNames, const GLclampf
*priorities);
```

Команда назначает **n** приоритетов из массива ***priorities*** для ***n*** соответствующих текстурных объектов из массива ***textureNames***. перед назначением величины приоритетов приводятся к диапазону **[0; 1]. 0** означает низший приоритет **(**такие текстуры вряд ли будут резидентными**),** а **1 –** наивысший**. glPrioritizeTextures()** не требует, чтобы какие-либо имена из ***textureNames*** были связаны с существующими текстурами. Однако приоритет не имеет никакого действия на объект до тех пор, пока он не связан.

Для назначения приоритета одной текстуре также может быть использована команда **glTexParameter*()**, но только в том случае, если имя текстуры уже связано. Кроме того, использование **glTexParameter*() –** это единственный способ назначить приоритет текстуре по умолчанию.

Если текстурные объекты имеют одинаковый приоритет, типичная реализация **OpenGL** при принятии решения о том, какую текстуру следует переместить из рабочего подмножества, применяет стратегию *наиболее давно использованного элемента* **(*least recently used* -- LRU).** Если вы уверены, что в вашей реализации действует именно этот алгоритм, тогда назначение одинаковых приоритетов всем объектам текстуры, создает разумную **LRU** систему по распределению текстурных ресурсов.

Если ваша реализация **OpenGL** не использует стратегию **LRU** для текстурных объектов с одинаковыми приоритетами **(**или если вы не знаете, какую стратегию она использует**),** вы можете разработать свою собственную стратегию **LRU,** аккуратно управляя приоритетами текстурных объектов. Когда текстура находится в использовании **(**когда

она связывается), вы можете установить максимальный приоритет, что соответствует наиболее недавнему использованию. Затем через регулярные промежутки времени вы можете уменьшать приоритеты всех объектов текстуры.

Замечание: Фрагментация текстурной памяти может быть проблемой, особенно если вы удаляете и создаете много новых текстур. Несмотря на то, что можно загрузить все текстуры в рабочее подмножество, произведя связывание последовательно в одной цепочке, связывание их в другой цепочке может оставить некоторые текстуры нерезидентными.

9.4.55 Функции текстурирования

В каждом из представленных в этой главе примеров величины в карте текстуры непосредственно использовались в качестве цветов, рисуемых на поверхности объектов. Вы также можете использовать текстуру для модуляции того цвета, которым объект был бы нарисован без текстурирования или комбинировать цвет в карте текстуры с оригинальным цветом поверхности. Вы выбираете одну из **4** функций текстурирования, передавая нужные аргументы команде **glTexEnv*()**.

```
void glTexEnv{if} (GLenum target, GLenum pname, TYPE param);
void glTexEnv{if}v (GLenum target, GLenum pname, TYPE *params);
```

Устанавливает текущую функцию текстурирования. Аргумент ***target*** должен быть установлен в значения **GL_TEXTURE_1D**, **GL_TEXTURE_2D** или **GL_TEXTURE_3D**. Если ***pname*** равен **GL_TEXTURE_ENV_MODE**, то ***param*** должен быть равен **GL_DECAL**, **GL_REPLACE**, **GL_MODULATE** или **GL_BLEND**, задавая метод, которым величины текстуры должны комбинироваться с обрабатываемыми фрагментами. Если ***pname*** равен **GL_TEXTURE_ENV_COLOR**, ***param*** представляет собой массив из **4** чисел с плавающей точкой, являющихся компонентами **R**, **G**, **B** и **A**. Эти величины используются только если текущей функцией текстурирования является **GL_BLEND**.

Комбинация функции текстурирования и базового внутреннего формата определяет, как применяется текстура для каждого ее компонента. Функции текстурирования оперируют выбранными компонентами текстуры и цветовыми величинами, которые использовались бы без текстурирования. (Заметьте, что этот выбор осуществляется после выполнения операций передачи пикселей.) Вспомните, что при создании ваших текстурных карт командой **glTexImage*D()**, вы передаете спецификатор внутреннего формата в третьем аргументе.

В таблицах **9-4** и **9-5** перечислены формулы текстурирования, используемые для каждого компонента текстуры. Существует **6** базовых внутренних форматов (буквы в скобках показывают, как они обозначены в таблице): **GL_ALPHA (A)**, **GL_LUMINANCE (L)**, **GL_LUMINANCE_ALPHA (L** и **A)**, **GL_INTENSITY (I)**, **GL_RGB (C)** и **GL_RGBA (C** и **A)**. Остальные внутренние форматы задают желаемое разрешение компонент текстуры, и работа с ними ведется так же как с одним из перечисленных выше форматов.

Замечание: В таблице 9-4 нижний индекс t означает величину текстуры, f – величину входящего фрагмента, c – величину, присвоенную переменной состояния GL_TEXTURE_ENV_COLOR, а отсутствие нижнего индекса – означает финальную вычисленную величину. Также в этих таблицах умножение цветовой тройки на скаляр означает умножение каждого из трех компонент R, G и B на этот скаляр; умножение (или суммирование) двух цветовых троек означает попарное умножение (или суммирование) соответствующих компонентов из каждой тройки.

Таблица **9-4**. Текстурные функции **GL_REPLACE** и **GL_MODULATE**

Базовый внутренний	Функция GL_REPLA	Функция GL_MODULA	Функция GL_DECAL	Функция GL_BLEND

формат	CE	TE		
GL_ALPHA	$C = C_f$ $A = A_t$	$C = C_f$ $A = A_f A_t$	не определена	$C = C_f$ $A = A_f A_t$
GL_LUMINANCE	$C = L_t$ $A = A_f$	$C = C_f L_t$ $A = A_f$	не определена	$C = C_f(1 - L_t) + C_c L$ $A = A_f$
GL_LUMINANCE_ALPHA	$C = L_t$ $A = A_t$	$C = C_f L_t$ $A = A_f A_t$	не определена	$C = C_f(1 - L_t) + C_c L$ $A = A_f A_t$
GL_INTENSITY	$C = I_t$ $A = I_t$	$C = C_f I_t$ $A = A_f I_t$	не определена	$C = C_f(1 - I_t) + C_c I_t$ $A = A_f(1 - I_t) + A_c I_t$
GL_RGB	$C = C_t$ $A = A_f$	$C = C_f C_t$ $A = A_f$	$C = C_t$ $A = A_f$	$C = C_f(1 - C_t) + C_c C$ $A = A_f$
GL_RGBA	$C = C_t$ $A = A_t$	$C = C_f C_t$ $A = A_f A_t$	$C = C_f(1 - A_t) + C_t A$ $A = A_f$	$C = C_f(1 - C_t) + C_c C$ $A = A_f A_t$

Замечание: Замещающая текстурная функция (GL_REPLACE) просто берет цвет, которым объект был бы нарисован без текстурирования (цвет фрагмента), отбрасывает его и замещает цветом текстуры. Функцию замещения следует использовать в случаях, когда на объект нужно наложить непрозрачную текстуру.

Функция **GL_DECAL (**переводная картинка**)** похожа на замещение за исключением того, что она работает только для внутренних форматов **RGB** и **RGBA** и обрабатывает альфа величины иначе. В формате **RGBA** результирующий цвет является результатом наложения цвета текстуры на цвет фрагмента, причем доли этих цветов определяются альфа компонентом текстуры, а альфа фрагмента в расчетах не участвует и не изменяется. Эта функция может использоваться для наложения таких текстур, как знак авиакомпании на крыле самолета **(**исходный цвет крыла виден везде, кроме того места, где нарисован знак**).**

При модуляции **(GL_MODULATE)** цвет фрагмента модулируется содержимым текстурной карты. Если базовый формат **– GL_LUMINANCE, GL_LUMINANCE_ALPHA** или **GL_INTENSITY,** цветовые величины умножаются на одинаковую величину, и цветовая карта модулирует между цветом фрагмента **(**если светлота или интенсивность равны **1)** и черным цветом **(**если они равны **0).** Для форматов **RGB** и **RGBA** каждый из компонентов входящего цвета умножается на соответствующую **(**и, возможно, разную для разных компонентов**)** величину в карте текстуры. Если присутствует альфа, ото умножается на альфа фрагмента. Модуляция хороша для использования с освещением, поскольку цвет освещенного полигона может быть использован для ослабления цвета текстуры. Белые зеркальные полигоны часто используются для визуализации освещенных объектов, а карта текстуры предоставляет диффузный цвет.

Текстурная функция наложения **–** это единственная функция, которая использует цвет, задаваемый переменной состояния **GL_TEXTURE_ENV_COLOR.** Величина светлоты, интенсивности или цвета в каком-то смысле используется в качестве значения альфа для наложения **GL_TEXTURE_ENV_COLOR** на цвет фрагмента.

9.4.5.1 Наложение зеркального цвета после текстурирования

По умолчанию операции, связанные с текстурированием выполняются после расчета освещенности. Однако наложение текстуры на зеркальный блик обычно снижает визуальный эффект освещения.

Вы можете отделить наложение зеркального цвета, чтобы он добавлялся к цвету фрагмента после текстурирования. Если зеркальный цвет отделяется, то при расчете освещенности получается два цвета на вершину: первичный, который представляет собой сумму всех незеркальных световых вкладов, и вторичный, представляющий собой сумму всех зеркальных вкладов. Во время наложения текстуры с ее цветами комбинируется только первичный цвет. Вторичный (зеркальный) цвет добавляется к цвету фрагмента после того, как его текстурный цвет вычислен и текстура наложена. В результате зеркальный блик на освещенном и текстурированном объекте получается более четким.

9.4.6 Назначение координат текстуры

Когда вы рисуете текстурированную сцену, вы должны предоставить для каждой вершины и объектные, и текстурные координаты. Объектные координаты после преобразований определяют место, где будет нарисована каждая вершина. Координаты текстуры задают, какой из тэкселей текстурной карты назначается этой вершине. Точно так же как цвета интерполируются между двумя вершинами плавно залитого полигона или линии, между двумя вершинами интерполируются координаты текстуры. (Помните, что текстуры являются прямоугольными массивами данных.)

Координаты текстуры могут включать в себя **1**, **2**, **3** или **4** элемента. Обычно на них ссылаются как на координаты ***s***, ***t***, ***r*** и ***q***, чтобы отделить от объектных координат (***x***, ***y***, ***z***, ***w***) и координат вычислителей (***u***, ***v***). Для одномерных текстур используется только координата ***s***; для двумерных используются ***s*** и ***t***; для трехмерных – ***s***, ***t*** и ***r***. Координате ***q***, как и объектной координате ***w***, обычно присваивается значение **1**. Она может быть использована для создания одномерных координат. Команда для передачи координат текстуры **glTexCoord*()** похожа на **glVertex*()**, **glColor*()** и **glNormal*()** – она имеет те же вариации и используется таким же образом между парами **glBegin()** и **glEnd()**. Обычно величины координат текстуры ранжируются от **0** до **1**, однако им можно присваивать значения и вне этого диапазона.

```
void glTexCoord{1234}{sifd} (TYPE coords);
void glTexCoord{1234}{sifd}v (TYPE *coords);
```

Устанавливает текущие координаты текстуры (***s***, ***t***, ***r***, ***q***). Последующие обращения к команде **glVertex*()** ассоциируют с передаваемыми вершинами текущие координаты текстуры. При использовании команды **glTexCoord1*()** координата ***s***устанавливается в передаваемую величину, ***t*** и ***r*** устанавливаются в **0**, а ***q***– в **1**. При использовании **glTexCoord3*()** ***q*** устанавливается в **1**, в ***s***, ***t*** и ***r*** в передаваемые значения. Все **4** координаты вы можете задать с помощью команды **glTexCoord4*()**. Используйте нужный суффикс (**s**, **i**, **f** или **d**) и соответствующее значение для **TYPE** (**GLshort**, **GLint**, **GLfloat** или **GLdouble**), чтобы задать тип данных для координат. Вы можете передавать координаты индивидуально или воспользоваться векторной версией команды, чтобы передать их в виде указателя на массив, который их содержит. До того, как будет выполнено наложение текстуры, текстурные координаты умножаются на текстурную матрицу размерности **4x4**. Обратите внимание на то, что целые координаты интерпретируются непосредственно, а не отображаются на диапазон **[-1; 1]** как координаты нормалей.

В следующем разделе объясняется, как вычислить правильные текстурные координаты. Вместо того, чтобы назначать их непосредственно, вы можете запросить у **OpenGL** автоматическое вычисление текстурных координат как функции от координат вершин.

9.4.7 Вычисление правильных координат текстуры

Двумерные текстуры являются квадратами или прямоугольниками изображений, которые обычно накладываются на полигоны, составляющие полигональную модель. В простейшем случае вы накладываете прямоугольную текстуру на прямоугольную же модель – например, если ваша текстура представляет собой отсканированное изображение каменной стены, а полигон, на который она должна быть наложена – прямоугольная стена здания. Предположим, что и стена и текстура имеют квадратную форму, и вы хотите наложить всю текстуру на всю стену. В порядке против часовой стрелки координаты текстуры будут **(0,0)**, **(1,0)**, **(1,1)**, **(1,0)**. Когда вы рисуете стену, просто передайте эти текстурные координаты в то же время, когда передаете координаты вершин в порядке против часовой стрелки.

Теперь предположим, что высота стены равна **2/3** ее ширины, а текстура по-прежнему квадратная. Чтобы избежать искажения текстуры, вам нужно наложить на стену только ее часть, сохранением пропорций изображения. Предположим, что для этого вы решили использовать нижние **2/3** изображения текстуры. В этом случае при передаче вершин в порядке против часовой стрелки используйте следующие координаты текстуры **– (0,0), (1, 0), (1, 2/3), (0, 2/3).**

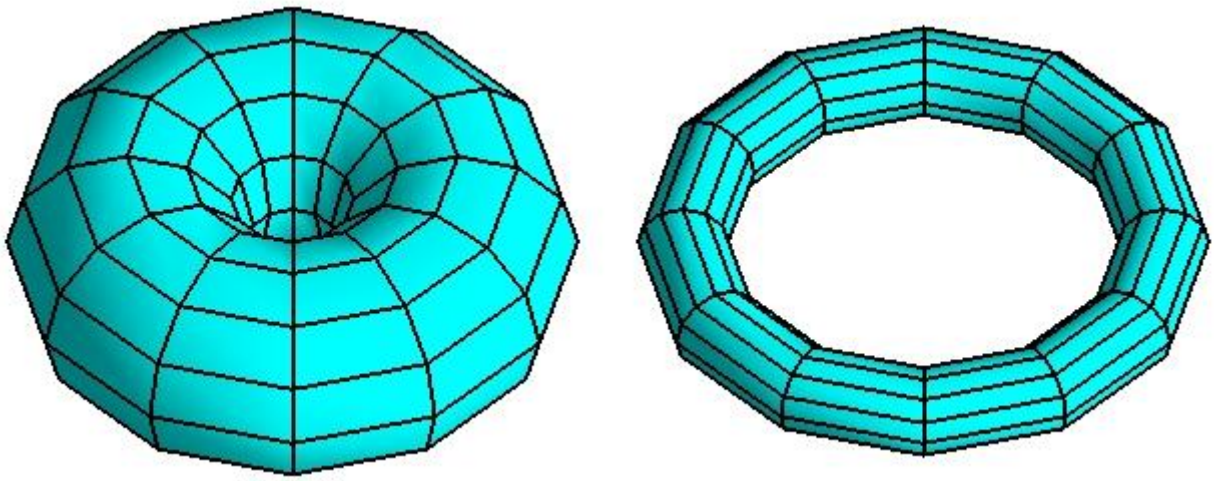
В качестве более сложного примера, предположим, что вы хотите нарисовать тонкую консервную банку и наложить на нее текстуру с изображением этикетки. Чтобы это сделать, вы покупаете консервную банку, отделяете этикетку и сканируете ее. Предположим, что она имеет **4** единицы в высоту и **12** в ширину, что соответствует пропорции размера **3** к **1**. Поскольку текстуры должны иметь пропорции 2^n к **1**, вы можете либо не использовать верхнюю треть текстуры, либо, скопировав, присоединить к левому (правому) краю текстуры ее часть с правого (левого) края шириной **4** единицы. Предположим, вы решили не использовать верхнюю треть. Пусть банка представляет собой цилиндр, аппроксимированный **30** полигонами по **4** единицы высотой (что составляет высоту банки) и шириной **12/30** (что составляет **1/30** длины окружности банки). Для каждого из аппроксимирующих прямоугольников вы можете использовать следующие координаты текстуры:

1:	**(0,0)**	**(1/30,0)**	**(1/30,2/3)**	**(0,2/3)**
2:	**(1/30,0)**	**(2/30,0)**	**(2/30,2/3)**	**(1/30,2/3)**
3:	**(2/30,0)**	**(3/30,0)**	**(3/30,2/3)**	**(2/30,2/3)**
...				
30:	**(29/30,0)**	**(1,0)**	**(1,2/3)**	**(29/30, 2/3)**

Только некоторые изогнутые поверхности, такие как конусы и цилиндры допускают наложение плоских поверхностей без геодезического искажения (геодезической дисторсия). Другие формы требуют некоторой доли искажения. Вообще, чем больше кривизна поверхности, тем больше искажений появляется в текстуре.

Если вы не озабочены искажением текстуры, часто довольно просто найти приемлемое решение. Например, рассмотрим сферу, чьи координаты заданы в форме $(\cos\theta\cos\varphi, \cos\theta\sin\varphi, \sin\theta)$, где $0 \le \theta \le 2\pi$ и $0 \le \varphi \le \pi$. Прямоугольная карта текстуры может быть непосредственно наложена на каждый прямоугольник θ - φ, но искажение текстуры будет увеличиваться при приближении к полюсам. Вся верхняя линия текстуры накладывается на северный полюс, а вся нижняя – на южный. Для других поверхностей, таких как торус (бублик) с большим отверстием координаты текстуры естественным образом ложатся на координаты поверхности, производя лишь небольшое искажение, которое можно считать приемлемым для большинства приложений. На рисунке **9-11** показаны два торуса, один с небольшим отверстием (и, как следствие, с большим искажением текстуры вблизи центра), другой с большим отверстием (с небольшим искажением).

Рисунок **9-11.** Искажение текстурной карты



Если вы текстурируете сплайновую поверхность, сгенерированную вычислителями, параметры поверхности ***u***и ***v*** иногда могут использоваться в качестве координат текстуры. Однако, в общем случае, при наложении текстуры на аппроксимированные полигонами или криволинейные поверхности большое значение имеет артистический компонент.

9.4.8 Повторение и подтягивание текстуры

Вы можете задавать текстурные координаты за пределами диапазона **[0,1]**, в результате чего текстура будет повторена или подтянута. В режиме повторения, если у вас есть большая плоскость, текстурные координаты на которой варьируются от **0.0** до **10.0** в обоих направления, вы получите **100** копий текстуры, расположенных на экране вплотную друг к другу **(**по аналогии с паркетом или кусками линолеума**)**. В процессе повторения целые части текстурных координат игнорируются, и копии заполняют поверхность. Для большинства приложений, использующих повторение текстуры, тэксели в верхнем и нижнем рядах текстуры должны совпадать, то же касается и левого и правого столбца.

Другой возможность является подтягивание текстурных координат: любые величины больше **1.0** устанавливаются равными **1.0**, любые величины меньше **0.0** устанавливаются в **0.0**. Подтягивание полезно для приложений, в которых требуется, чтобы одна копия текстуры покрывала большую площадь. Если текстурные координаты поверхности варьируются от **0.0** до **10.0** в обоих направлениях, появится только одна копия текстуры в левом нижнем углу поверхности.

Если вы используете текстуры с границами или задали цвет границы, то и режим прикрепления, и метод фильтрации оказывают влияние на то, будет ли использоваться информация о границе и, если будет, то каким образом. Если вы выбрали метод фильтрации **GL_NEAREST**, используется ближайший тэксель в текстуре, а граница **(**или цвет границы**)** всегда игнорируется.

Если вы выбрали **GL_LINEAR** в качестве метода фильтрации, для наложения текстуры используется взвешенная комбинация цветовых данных из массива **2x2 (**для двумерных текстур**)**. Если присутствует граница или цвет границы, то текстура и этот цвет используются вместе следующим образом:

В режиме прикрепления **GL_REPEAT** граница всегда игнорируется. Массив взвешенных тэкселей размерности **2x2** прикрепляется к противоположному краю текстуры. Таким образом, тэксели на правом краю усредняются с теми, что находятся на левом. То же касается тэкселей на верхнем и нижнем краях.

В режиме крепления **GL_CLAMP** тэксели на границе **(**или тэксели цвета **GL_TEXTURE_BORDER_COLOR)** используются в массиве взвешенных тэкселей **2x2.**

В режиме крепления **GL_CLAMP_TO_EDGE** граница всегда игнорируется**.** Для текстурных вычислений используются тэксели на краю **(**или близко к краю**)** текстуры**,** а не граница или ее цвет**. (**Режим **GL_CLAMP_TO_EDGE** появился в **OpenGL** версии **1.2.)**

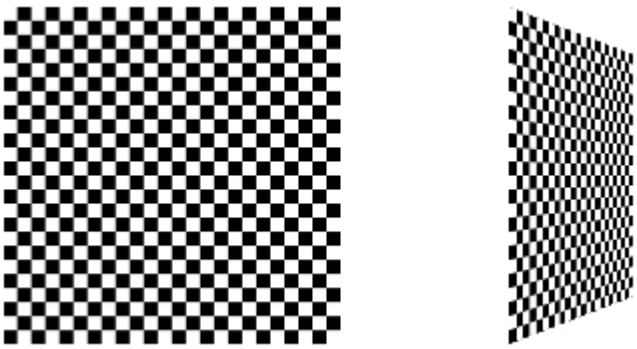
Заметьте**,** что если вы используете подтягивание**,** вы можете избежать воздействия текстуры на остаток поверхности**.** Чтобы добиться этого**,** используйте альфа равное **0** для тэкселей на краях текстуры **(**или на границе**).** Функция текстурирования **GL_DECAL** непосредственно использует альфа при вычислениях**.** Если вы используете другую текстурную функцию**,** вам**,** возможно**,** понадобиться использовать цветовое наложение**.**

Чтобы оценить эффекты разных режимов крепления текстуры**,** мы должны установить текстурные координаты за пределами диапазона **[0.0, 1.0].** Начнем с рассмотрения примера **9-1,** модифицировав код таким образом**,** чтобы при наложении текстуры на квадраты ее координаты изменялись от **0.0** до **3.0.**

```
glBegin (GL_QUADS);
glTexCoord2f (0.0,0.0); glVertex3f(-2.0,-1.0,0.0);
glTexCoord2f (0.0,3.0); glVertex3f(-2.0,1.0,0.0);
glTexCoord2f (3.0,3.0); glVertex3f(0.0,1.0,0.0);
glTexCoord2f (3.0,0.0); glVertex3f(0.0,-1.0,0.0);

glTexCoord2f (0.0,0.0); glVertex3f(1.0,-1.0,0.0);
glTexCoord2f (0.0,3.0); glVertex3f(1.0,1.0,0.0);
glTexCoord2f (3.0,3.0); glVertex3f(2.41421,1.0, -1.41421);
glTexCoord2f (3.0,0.0); glVertex3f(2.41421,-1.0, 2.41421);
glEnd();
```

Результат работы этого кода в режиме прикрепления **GL_REPEAT** показан на рисунке **9-12.**

Рисунок **9-12.** Повторение текстуры



В этом случае текстура повторяется в направлении ***s*****,** и в направлении ***t*****,** поскольку в тексте присутствуют два вызова**:**

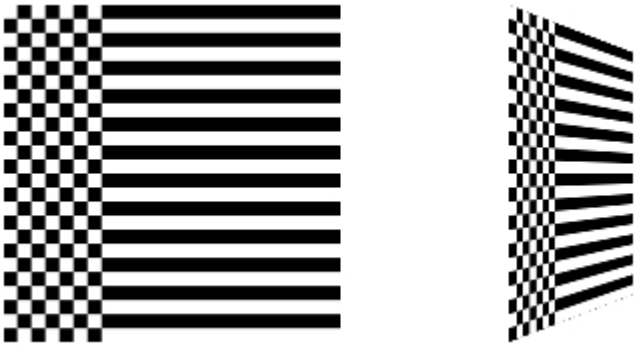
```
glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_WRAP_S,GL_REPEAT);
glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_WRAP_T,GL_REPEAT);
```

Если в каждом из направлений использовать **GL_CLAMP** вместо **GL_REPEAT,** вы получите нечто похожее на рисунок **9-13.**

Рисунок **9-13.** Подтягивание текстуры

Вы также можете повторять текстуру в одном направлении и подтягивать в другом, как на рисунке **9-14.**

Рисунок **9-14.** Повторение и подтягивание текстуры



Теперь вы изучили все аргументы команды **glTexParameter*().** Они перечислены в следующей таблице**.**

```
void glTexParameter{if} (GLenum target, GLenum pname, TYPE param);
void glTexParameter{if}v (GLenum target, GLenum pname, TYPE *param);
```

Устанавливает различные параметры**,** управляющие тем**,** как обрабатывается текстура в процессе наложения на фрагмент или сохранения в текстурном объекте**.** Аргумент ***target*** может принимать значения **GL_TEXTURE_1D, GL_TEXTURE_2D** или **GL_TEXTURE_3D,** индицируя размерность текстур**,** параметр которых изменяется**.** Возможные значения для ***pname*** и ***param*** перечислены в таблице **9-5.** Вы можете использовать векторную версию команды для установки параметра **GL_TEXTURE_BORDER_COLOR** или задавать индивидуальные значения других параметров с помощью невекторной версии**.** Если значения передаются как целые**,** они преобразуются в формат с плавающей точкой согласно таблице **4-1;** они также приводятся к диапазону **[0, 1].**

Таблица **9-5.** Параметры**,** устанавливаемые **glTexParameter*()**

Параметр	Возможные значения
GL_TEXTURE_WRAP_S	GL_CLAMP, GL_REPEAT, GL_CLAMP_TO_EDGE
GL_TEXTURE_WRAP_T	GL_CLAMP, GL_REPEAT, GL_CLAMP_TO_EDGE
GL_TEXTURE_WRAP_R	GL_CLAMP, GL_REPEAT, GL_CLAMP_TO_EDGE
GL_TEXTURE_MAG_FILTER	GL_NEAREST, GL_LINEAR
GL_TEXTURE_MIN_FILTER	GL_NEAREST, GL_LINEAR, GL_NEAREST_MIPMAP_NEAREST, GL_NEAREST_MIPMAP_LINEAR,GL_LINEAR_MIPMAP_NEAREST, GL_LINEAR_MIPMAP_LINEAR

GL_TEXTURE_BORDER_COLOR	любые **4** величины в диапазоне **[0.0, 1,0]**
GL_TEXTURE_PRIORITY	**[0.0, 1.0]** для текущего текстурного объекта
GL_TEXTURE_MIN_LOD	любое число с плавающей точкой
GL_TEXTURE_MAX_LOD	любое число с плавающей точкой
GL_TEXTURE_BASE_LEVEL	любое число с плавающей точкой
GL_TEXTURE_MAX_LEVEL	любое число с плавающей точкой

9.5 Автоматическое генерирование текстурных координат

Вы можете использовать текстурирование для создания контуров ваших моделей или для имитации отражения от окружающих предметов на блестящих поверхностях**.** Чтобы добиться этих результатов**,** позвольте **OpenGL** автоматически генерировать для вас координаты текстуры вместо того**,** чтобы задавать их непосредственно командой **glTexCoord*().** Для автоматического генерирования текстурных координат используйте команду **glTexGen*().**

```
void glTexGen{ifd} (GLenum coord, GLenum pname, TYPE param);
void glTexGen{ifd}v (GLenum coord, GLenum pname, TYPE *param);
```

Задает функцию для автоматического вычисления текстурных координат**.** Первый аргумент ***coord***может принимать значения **GL_S, GL_T, GL_R** или **GL_Q,** указывая на то**,** какая из координат**:** ***s***, ***t***, ***r*** или ***q*** должна быть вычислена**.** Аргумент ***pname*** может принимать значения **GL_TEXTURE_GEN_MODE, GL_OBJECT_PLANE, GL_EYE_PLANE, GL_SPHERE_MAP.** Если задано значение **GL_TEXTURE_GEN_MODE, *param*** должен быть целым числом **(**или указателем на целое число**,** если используется векторная версия команды**),** которое равно **GL_OBJECT_LINEAR, GL_EYE_LINEAR** или **GL_SPHERE_MAP**. Эти символические константы указывают на то**,** какая функция должна использоваться для вычисления координат текстуры**.** Со всеми остальными возможными значениями для ***pname***, ***param*** должен представлять собой указатель на массив величин **(**в векторной версии команды**),** задавая параметры функции вычисления текстурных координат**.**

Различные методы генерирования текстурных координат имеют различное назначение**.** Использование плоскости в объектных координатах лучше всего применять**,** если изображение текстуры должно оставаться фиксированным на движущемся объекте**.** Таким образом**, GL_OBJECT_LINEAR** может использоваться для наложения текстуры дерева на крышку стола**.** Использование плоскости в видовых координатах **(GL_EYE_LINEAR)** лучше всего использовать для создания динамических контурных линий на движущихся объектах**. GL_EYE_LINEAR** может быть использована специалистами по геодезии**,** работающими с нефтью и газом**.** По мере того как скважина уходит глубже в землю**,** она может быть нарисована разными цветами**,** чтобы показать слои камня на разных глубинах**. GL_SPHERE_MAP** главным образом используется для создания *наложения окружающей обстановки* **(*environmental mapping*).**

9.5.1 Создание контуров

Когда заданы **GL_TEXTURE_GEN_MODE** и **GL_OBJECT_LINEAR,** функция вычисления текстурных координат **–** это линейная комбинация объектных координат вершины (x_0, y_0, z_0, w_0)**:**

сгенерированная координата **=** $p_1x_0 + p_2y_0 + p_3z_0 + p_4w_0$

Величины $p_1,\ldots,p_4$ передаются в аргументе ***param*** команды **glTexGen*v()** при ***pname***, установленным в **GL_OBJECT_PLANE.** При правильно нормализованных $p_1,\ldots,p_4$

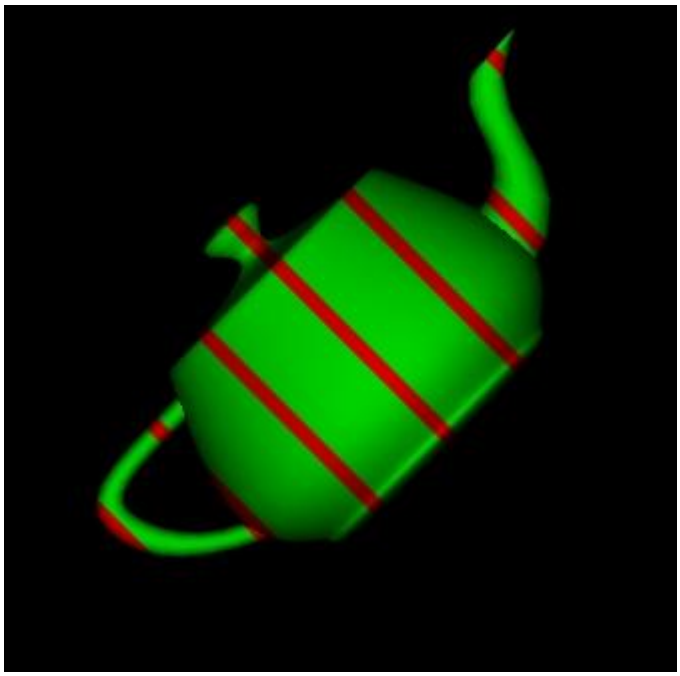
функция дает дистанцию от вершины до плоскости. Например, если $p_2 = p_3 = p_4$, а $p_1 = 1$, функция дает дистанцию между вершиной и плоскостью **x=0.** Дистанция положительна на одной стороне плоскости, отрицательна на другой и равна **0,** если вершина лежит в этой плоскости.

В самом начале в примере **9-8** на чайнике с равными пропусками рисуются контурные линии; линии показывают дистанцию от плоскости **x=0.** Коэффициенты для плоскости **x=0** находятся в массиве:

```
GLfloat xequalzero[]={1.0,0.0,0.0,0.0};
```

В данном случае вычисляется только **1** величина **(**дистанция от плоскости**),** что позволяет наложить одномерную текстуру. Текстура полностью зеленая, однако, через равные промежутки времени на ней помещены красные маркеры. Поскольку чайник как бы стоит на плоскости **xy,** все контуры перпендикулярны его основанию. Изображение нарисованное программой показано на рисунке **9-15.**

Рисунок **9-15.** Красные контуры параллельны плоскости **x=0**

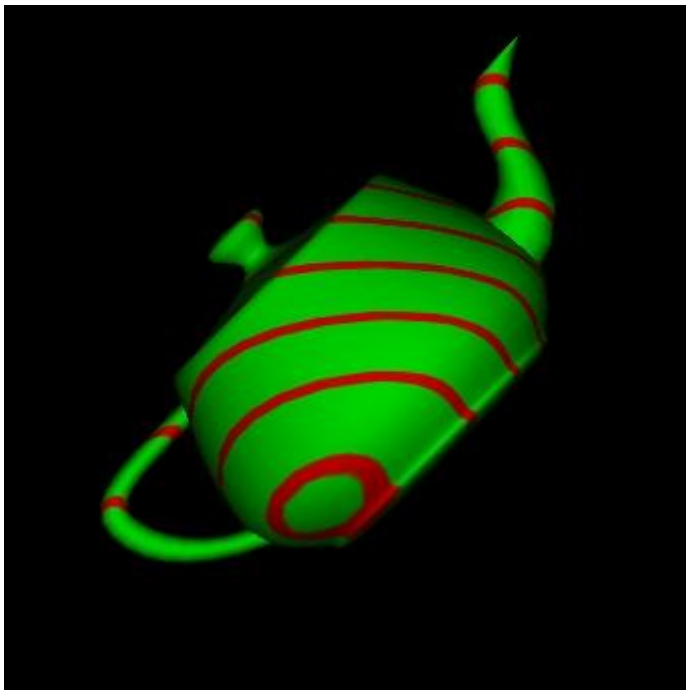


В этом же примере нажатие на клавишу **'s'** изменяет параметры плоскости:

```
GLfloat stanted[]={1.0,1.0,1.0,0.0};
```

Теперь красные контуры параллельны плоскости **x+y+z=0.** Соответствующее изображение показано на рисунке **9-16.** Чтобы восстановить уравнение, соответствующее плоскости **x=0,** нажмите клавишу **'x'.**

Рисунок **9-16.** Красные контуры параллельны плоскости **x+y+z=0**



Пример **9-8.** Автоматическое генерирование координат текстуры**:** файл **texgen.cpp**

```
#include <glut.h>

#define stripeImageWidth 32

GLubyte stripeImage[4*stripeImageWidth];
GLuint texName;

void makeStripeImage()
{
int j;

for(j=0;j<stripeImageWidth;j++)
{
stripeImage[4*j]=(GLubyte) ((j<=4) ? 255 : 0);
stripeImage[4*j+1]=(GLubyte) ((j>4) ? 255 : 0);
stripeImage[4*j+2]=(GLubyte) 0;
stripeImage[4*j+3]=(GLubyte) 255;
}
}

GLfloat xequalzero[]={1.0,0.0,0.0,0.0};
GLfloat slanted[]={1.0,1.0,1.0,0.0};
GLfloat *currentCoeff;
GLenum currentPlane;
GLint currentGenMode;

void init()
{
glClearColor(0.0,0.0,0.0,0.0);
glEnable(GL_DEPTH_TEST);
glShadeModel(GL_SMOOTH);

makeStripeImage();
glPixelStorei(GL_UNPACK_ALIGNMENT,1);

glGenTextures(1,&texName);
glBindTexture(GL_TEXTURE_1D,texName);
```

```
glTexParameteri(GL_TEXTURE_1D,GL_TEXTURE_WRAP_S,GL_REPEAT);
glTexParameteri(GL_TEXTURE_1D,GL_TEXTURE_MAG_FILTER,GL_LINEAR);
glTexParameteri(GL_TEXTURE_1D,GL_TEXTURE_MIN_FILTER,GL_LINEAR);

glTexImage1D(GL_TEXTURE_1D,0,GL_RGBA,stripeImageWidth,0,
GL_RGBA,GL_UNSIGNED_BYTE,stripeImage);

glTexEnvf(GL_TEXTURE_ENV,GL_TEXTURE_ENV_MODE,GL_MODULATE);

currentCoeff=xequalzero;
currentGenMode=GL_OBJECT_LINEAR;
currentPlane=GL_OBJECT_PLANE;
glTexGeni(GL_S,GL_TEXTURE_GEN_MODE,currentGenMode);
glTexGenfv(GL_S,currentPlane,currentCoeff);

glEnable(GL_TEXTURE_GEN_S);
glEnable(GL_TEXTURE_1D);
glEnable(GL_CULL_FACE);
glEnable(GL_LIGHTING);
glEnable(GL_LIGHT0);
glEnable(GL_AUTO_NORMAL);
glEnable(GL_NORMALIZE);
glFrontFace(GL_CW);
glCullFace(GL_BACK);
glMaterialf(GL_FRONT,GL_SHININESS,64.0);
}

void display()
{
glClear(GL_COLOR_BUFFER_BIT|GL_DEPTH_BUFFER_BIT);
glPushMatrix();
glRotatef(45.0,0.0,0.0,1.0);
glBindTexture(GL_TEXTURE_1D,texName);
glutSolidTeapot(2.0);
glPopMatrix();
glFlush();
}

void reshape(int w, int h)
{
glViewport(0,0,(GLsizei) w, (GLsizei) h);
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
if (w<=h)
glOrtho(-3.5,3.5,-3.5*(GLfloat)h/(GLfloat)w,
3.5*(GLfloat)h/(GLfloat)w,-3.5,3.5);
else
glOrtho(-3.5*(GLfloat)w/(GLfloat)h,
3.5*(GLfloat)w/(GLfloat)h,-3.5,3.5,-3.5,3.5);
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
}

void keyboard(unsigned char key,int x, int y)
{
switch(key)
{
case 'e':
case 'E':
currentGenMode=GL_EYE_LINEAR;
currentPlane=GL_EYE_PLANE;
glTexGeni(GL_S,GL_TEXTURE_GEN_MODE,currentGenMode);
```

```
glTexGenfv(GL_S,currentPlane,currentCoeff);
glutPostRedisplay();
break;
case 'o':
case 'O':
currentGenMode=GL_OBJECT_LINEAR;
currentPlane=GL_OBJECT_PLANE;
glTexGeni(GL_S,GL_TEXTURE_GEN_MODE,currentGenMode);
glTexGenfv(GL_S,currentPlane,currentCoeff);
glutPostRedisplay();
break;
case 's':
case 'S':
currentCoeff=slanted;
glTexGenfv(GL_S,currentPlane,currentCoeff);
glutPostRedisplay();
break;
case 'x':
case 'X':
currentCoeff=xequalzero;
glTexGenfv(GL_S,currentPlane,currentCoeff);
glutPostRedisplay();
break;
}
}

int main(int argc, char **argv)

{
glutInit(&argc,argv);
glutInitDisplayMode(GLUT_SINGLE|GLUT_RGB|GLUT_DEPTH);
glutInitWindowSize(350,350);
glutInitWindowPosition(100,100);
glutCreateWindow("Automatic Texture-Coordinate Generation");
init();
glutDisplayFunc(display);
glutReshapeFunc(reshape);
glutKeyboardFunc(keyboard);
glutMainLoop();
return 0;
}
```

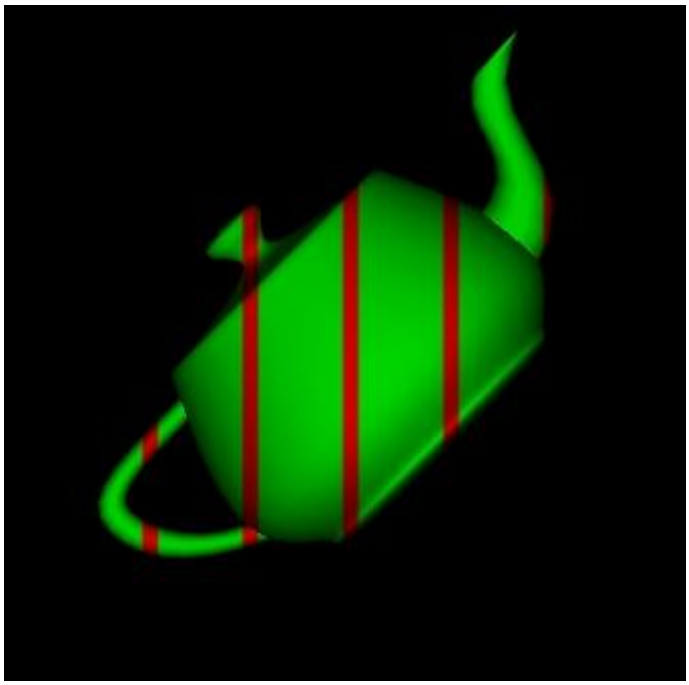
Вы можете включить автоматический расчет **s-**координат текстур**,** передав аргумент **GL_TEXTURE_GEN_S** команде **glEnable().** Для автоматической генерации координат **t, r** или **q** нужно передать той же команде аргументы **GL_TEXTURE_GEN_T, GL_TEXTURE_GEN_R** или **GL_TEXTURE_GEN_Q** соответственно**.** Для выключения генерации координат текстуры используйте **glDisable()** с теми же аргументами**.**

Функция **GL_OBJECT_LINEAR** вычисляет координаты текстуры в координатной системе модели**.** В примере **9-8** изначально используется функция **GL_OBJECT_LINEAR,** и линии контура остаются перпендикулярными основанию чайника независимо от того**,** как поворачивается чайник**,** и с какой стороны на него смотрят**.** Однако**,** если нажать клавишу **'e'** режим генерирования координат изменится с **GL_OBJECT_LINEAR** на **GL_EYE_LINEAR,** и координаты будут рассчитываться относительно видовой координатной системы**. (**Нажатие на клавишу **'o'** восстанавливает режим **GL_OBJECT_LINEAR.)** Если в качестве плоскости определена **x=0,** то линии контура на чайнике будут параллельны плоскости **yz** относительно точки наблюдения**,** как показано на рисунке **9-17.** Говоря математически**,** вектор (p_1, p_2, p_3, p_4) умножается на матрицу обратную к видовой для получения значений**,** используемых для вычисления дистанции до плоскости**.** Координаты текстуры генерируются следующей функцией**:**

сгенерированная координата = $p_1`x_e + p_2`y_e + p_3`z_e + p_4`w_e$, где $(p_1`, p_2`, p_3`, p_4`) = (p_1, p_2, p_3, p_4)M^{-1}$.

В этом случае (x_e, y_e, z_e, w_e) -- это видовые координаты вершины, а величины $p_1, \ldots, p_4$ передаются в аргументе ***param*** команды **glTexGen*v()** при ***pname***, установленным в **GL_EYE_PLANE.** Величины $(p_1`, p_2`, p_3`, p_4`)$ вычисляют только один раз в момент их передачи **glTexGen*v()**, так что эта операция вовсе не так затратна с точки зрения вычислений, как кажется с первого взгляда.

Рисунок **9-17.** Красные контуры параллельны плоскости **yz** в видовых координатах



Во всех приведенных примерах для создания контуров использовалась только одна сгенерированная координата. Однако координаты **s**, **t** и **(**если необходимо**) r** могут генерироваться независимо, отражая дистанцию между одной, двумя или тремя разными плоскостями. Если правильно построить двумерную или трехмерную текстурную карту, два или три набора контуров можно будет наблюдать одновременно. В качестве дополнительного усложнения вы можете смешивать генерирующие функции. Например, вы можете вычислять **s**координату с помощью **GL_OBJECT_LINEAR**, а ***t*–** с использованием **GL_EYE_LINEAR.**

9.5.2 Наложение изображения окружающихся предметов

Целью наложения окружающей обстановки является визуализация предмета, как будто он идеально блестящий и гладкий, и цвета на его поверхности – это цвета окружающих его объектов, отраженные в глаз наблюдателя. Другими словами, если вы смотрите на идеально отполированный, идеально отражающий серебряный объект в комнате, вы видите на нем отражение стен, пола и других предметов в комнате. **(**Классическим примером этой техники является изображение киборга **T1000** из фильма Терминатор **2.)** Объекты, отражения которых вы видите, зависят от точки наблюдения и углов поверхности серебряного объекта. Чтобы производить наложение окружающей обстановки, все, что вам нужно сделать – это создать подходящую карту текстуры, а затем заставить **OpenGL** автоматически сгенерировать для вас текстурные координаты.

Наложение *окружающей обстановки* **(*environmental mapping*) –** это аппроксимация, основанная на предположении о том, что элементы обстановки находятся далеко в сравнении с блестящим отражающим объектом – то есть маленький объект находится в большой комнате. При таком предположении для нахождения цвета точки на поверхности нужно выпустить луч из глаза наблюдателя в точку поверхности, а затем отразить его от поверхности. Направление отраженного луча полностью определяет цвет, который нужно нарисовать в точке отражения. Кодирование цвета для каждого направления на плоской текстурной карте эквивалентно помещению полированной сферы без изъянов в центр окружения и снятию с нее фотографии с помощью камеры, имеющей линзы с очень большой фокальной длиной и находящейся далеко от сферы. Говоря математически, линзы имеют бесконечно большую фокальную длину, а камера бесконечно удалена от сферы. Таким образом, кодирование покрывает циркулярный регион текстурной карты, находящийся вплотную к ее верхнему, нижнему, правому и левому краям. Величины вне этого круга не важны, поскольку при наложении обстановки доступ к ним никогда не осуществляется **(**они попросту не нужны**).**

Для получения абсолютно правильной карты обстановки, вам нужно раздобыть большую серебристую сферу, снять с нее фотографию в некоторой обстановке с помощью камеры, расположенной бесконечно далеко и имеющей линзы с бесконечно большой фокальной длиной, и отсканировать эту фотографию. Для получения приблизительно верного результата вы можете использовать отсканированную фотографию обстановки, снятую камерой, имеющей линзы с очень большим углом **(**рыбий глаз**).**

После того, как вы создали текстуру, разработанную для наложения окружающей обстановки, вам нужно активизировать соответствующий алгоритм **OpenGL.** Этот алгоритм находит точку на поверхности сферы, которая находится там же, где и точка на поверхности визуализируемого отражающего объекта и закрашивает точку объекта цветом, находящимся в соответствующей точке сферы.

Чтобы автоматически сгенерировать координаты текстуры для поддержки наложения окружающей обстановки, используйте в программе следующий код:

```
glTexGeni(GL_S, GL_TEXTURE_GEN_MODE, GL_SPHERE_MAP);
glTexGeni(GL_T, GL_TEXTURE_GEN_MODE, GL_SPHERE_MAP);
glEnable(GL_TEXTURE_GEN_S);
glEnable(GL_TEXTURE_GEN_T);
```

Константа **GL_SPHERE_MAP** создает правильные текстурные координаты для наложения обстановки. Как показано выше, вам нужно установить этот режим в обоих направлениях **(*s* и *t*).** Однако вам не требуется задавать никаких параметров для функции, генерирующей координаты текстуры.

Функция **GL_SPHERE_MAP** генерирует координаты текстуры в несколько шагов:

1. Пусть ***u*** **–** это вектор единичной длины направленный из начала координат к вершине **(**в видовых координатах**).**
2. Пусть ***n'*** **–** текущий вектор нормали после преобразования в видовые координаты.
3. Пусть ***r*** **–** это вектор отражения $(r_x, r_y, r_z)^T$, вычисляемый как $u - 2n`n`^T u$.
4. Затем вычисляет промежуточная величина $m = 2\sqrt{r_x^2 + r_y^2 + (r_z + 1)^2}$.
5. Наконец, координаты текстуры **s** и **t** вычисляются следующим образом: $s = r_x / m + \frac{1}{2}$ и $t = r_y / m + \frac{1}{2}$.

Естественно, что с помощью описанной техники на блестящих объектах можно получать не только изображения окружающих предметов, а вообще всего, чего угодно. Единственное условие – необходима сферическая карта текстуры. На рисунке **9-18** слева изображена одна из вариаций фрактала Мальденброда, а справа то же самое изображение, преобразованное к сферическому виду с помощью одного из графических редакторов. На рисунке **9-19** показан результат наложения этого изображения на чайник Юта.

Рисунок **9-18.** Изображение, преобразованное в сферическую карту текстуры

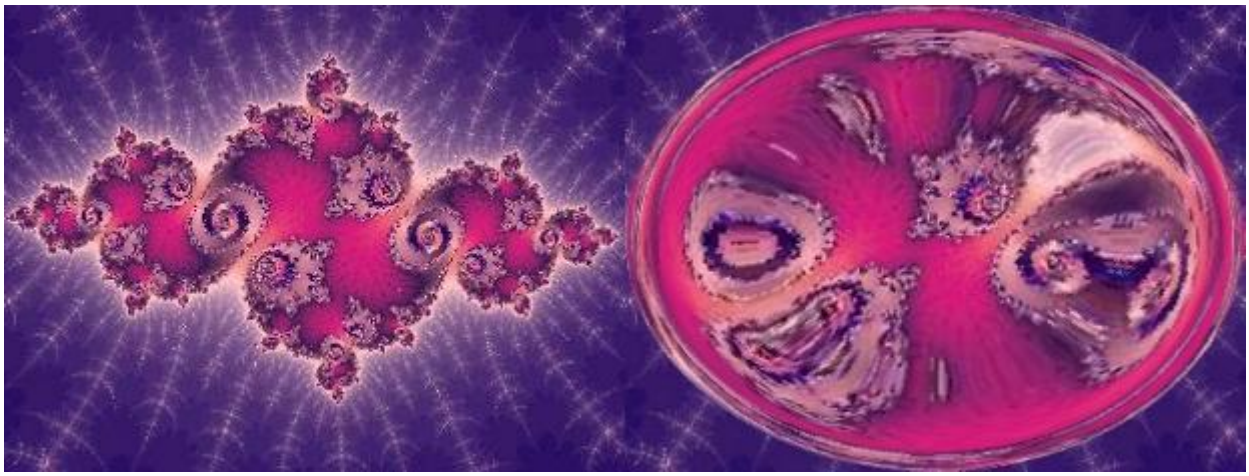


Рисунок **9-19.** Изображение множества Мальденброда на чайнике Юта

9.6 Дополнительные возможности

Дополнительно: В данном разделе описано, как использовать стек текстурных матриц (и связанную с ним координату ***q***), а также мультитекстурирование. Эти техники считаются весьма сложными и не понадобятся для большинства приложений.

9.6.1 Стек текстурных матриц

Точно так же как координаты вашей модели преобразуются матрицами до завершения визуализации, текстурные координаты умножаются на матрицу **4x4** до того как происходит наложение. По умолчанию текстурная матрица представляет собой

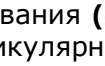
единичную, следовательно, заданные или автоматически сгенерированные координаты остаются неизменными. Однако, модифицируя текстурную матрицу в процессе перерисовки объекта, вы можете заставить текстуру двигаться по поверхности объекта, поворачиваться вокруг нее, растягиваться и сжиматься или формировать любую комбинацию всего перечисленного. Вообще говоря, поскольку текстурная матрица – это обычная матрица **4x4**, можно добиться даже эффекта перспективы.

На самом деле текстурная матрица является верхней матрицей текстурного стека, который, как минимум, должен иметь глубину **2** матрицы. Все стандартные команды манипуляций с матрицами – **glPushMatrix()**, **glPopMatrix()**, **glMultMatrix()**, **glRotate*()** и другие – могут применяться к текстурным матрицам. Чтобы изменить текущую текстурную матрицу, установите режим матрицы в **GL_TEXTURE** следующим образом:

```
glMatrixMode(GL_TEXTURE); /* войти в режим текстурной матрицы */
glRotated(...);
/*...другие манипуляции с матрицей...*/
glMatrixMode(GL_MODELVIEW);
```

9.6.1.1 Координата q

Математика четвертой текстурной координаты ***q*** похожа на математику координаты **w** из четверки **(*x*, *y*, *z*, *w*)** объектных координат. Когда четверка текстурных координат **(*s*, *t*, *r*, *q*)** умножается на матрицу текстуры, результирующий вектор **(*s'*, *t'*, *r'*, *q'*)** интерпретируется как четверка однородных координат текстуры. Другими словами текстурная карта индексируется величинами ***s'/q'***, ***t'/q'*** и ***r'/q'***.

Вы можете использовать ***q***в случаях, когда требуется более одного проекционного или перспективного преобразования. Например, предположим, что вы хотите смоделировать прожектор с неравномерным лучом, который ярче ближе к центру или имеет не круглую форму из-за линз. Вы можете эмулировать сияние такого луча на плоской поверхности, создав текстурную карту, которая соответствует форме и интенсивности света, и спроецировав ее на нужную поверхность с использованием проекционного преобразования. Проектирование конуса света на поверхности сцены требует перспективного преобразования (), поскольку свет может появляться на поверхностях, которые неперпендикулярны его источнику. Второе перспективное преобразование необходимо потому, что наблюдатель видит сцену с другой (но тоже перспективной) точки наблюдения.

Другой пример может возникнуть в ситуации, когда в качестве текстуры должна использовать фотография, сама имеющая перспективу. Как и в случае с источником света, результирующее изображение зависит от комбинации двух перспективных преобразований.

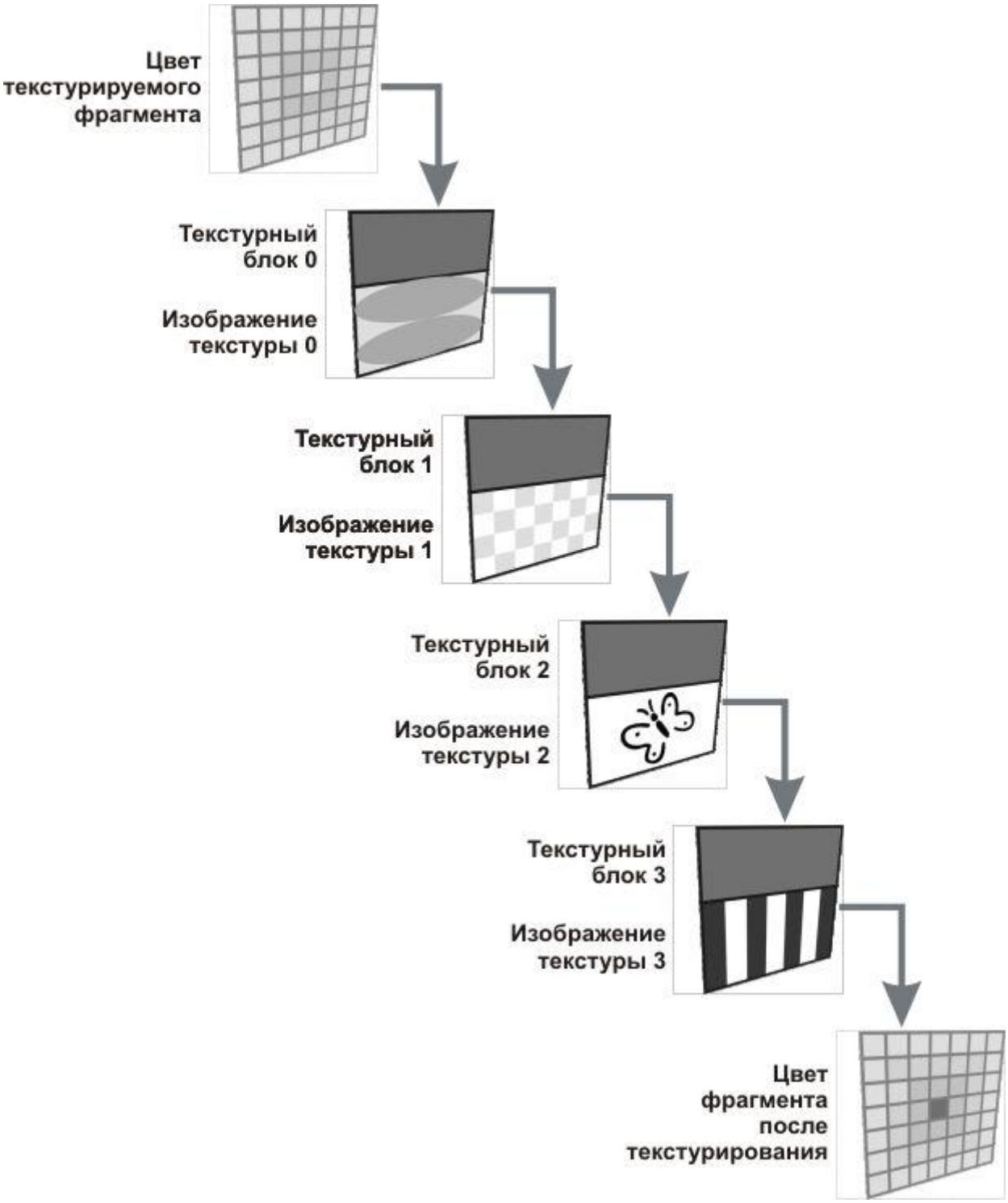
9.6.2 Мультитекстурирование

Во время стандартного текстурирования на полигон накладывается единственное изображение текстуры. В **OpenGL** версии **1.2** появилось мультитекстурирование, которое позволяет накладывать на один и тот же полигон несколько текстур одну за одной.

Мультитекстурирование – это опциональное расширение. Таким образом, несмотря на то, что оно было одобрено **OpenGL Architecture Review Board (ARB –** совет по рассмотрению архитектуры **OpenGL)**, который является управляющим органом по всем вопросам, связанным с **OpenGL**, реализация **OpenGL** версии **1.2 не** обязательно поддерживает мультитекстурирование.

Если ваша реализация поддерживает мультитекстурирование, у вас в распоряжении имеется серия *текстурных блоков или модулей* (***texture units***), каждый из которых производит одну операцию текстурирования и передает свой результат в следующий текстурный блок. Так происходит до тех пор, пока все определенные текстурные блоки не завершат свои операции. Рисунок **9-20** показывает, как фрагмент претерпевает **4** текстурные операции – по одной на каждый текстурный блок.

Рисунок **9-20.** Конвейер мультитекстурирования



Мультитекстурирование позволяет применять усложненные техники визуализации, такие как световые эффекты, композиция и детализированные текстуры.

9.6.2.1 Мультитекстурирование по шагам

Для написания кода, использующего мультитекстурирование, нужно выполнить следующие шаги:

1. Проверьте, поддерживается ли расширение мультитекстурирования. Если оно не поддерживается, возможно, вам удастся достигнуть тех же результатов с помощью нескольких проходов визуализации. Чтобы выяснить какое количество текстурных блоков присутствует в вашей реализации, вызовите **glGetIntegerv(GL_MAX_TEXTURE_UNITS_ARB, ...)**. При наиболее плохом сценарии возвращенное число будет равно **1**, что соответствует единственному текстурному блоку.

Замечание: Даже в случае, когда расширение поддерживается, в режиме отклика (feedback) результат мультитекстурирования не определен за пределами первого текстурного блока.

2. Установите нужное состояние текстурирования, включая изображение текстуры, фильтры, окружение, режим генерирования координат и матрицу, для каждого текстурного блока. Для смены текущего текстурного блока используйте команду **glActiveTextureARB()**.
3. Задавая вершины, используйте **glMultiTexCoord*ARB()** чтобы задавать несколько наборов координат для разных текстур. В каждом текстурном блоке используется свой набор координат. Каждый набор текстурных координат будет использован в период текстурирования определенным текстурным блоком **(**в одном из нескольких проходов**)**. Автоматическое генерирование координат и передача координат в виде вершинного массива являются специальными случаями.

9.6.2.2 Настройка текстурных блоков

Мультитекстурирование предоставляет множественные текстурные блоки, которые производят дополнительные проходы по наложению текстуры. Каждый из текстурных блоков имеет одинаковые возможности и свое собственное состояние, включая:

- изображение текстуры
- параметры фильтрации
- режимы наложения
- стек текстурных матриц
- функции автоматического генерирования текстурных координат
- спецификацию вершинного массива **(**если она требуется**)**.

Каждый текстурный блок комбинирует предыдущий цвет фрагмента со своим текстурным изображением в соответствии со своим состоянием. Результирующий цвет фрагмента поступает в следующий текстурный блок, если он активен.

Команда **glActiveTextureARB()** выбирает текущий модифицируемый текстурный блок, чтобы можно было изменять его состояние. После этого вызовы команд **glTexImage*()**, **glTexParameter*()**, **glTexEnv*()**, **glTexGen*()** и **glBindTexture()** воздействуют только на текущий текстурный блок. Запрос значений переменных состояния текстурирования также возвращает значения из текущего текстурного блока, также как запрос значений текущих координат текстуры и текущих растровых координат текстуры.

```
void glActiveTextureARB (GLenum texUnit);
```

Выбирает текстурный блок, который будет модифицироваться текстурными командами. ***texUnit*** – это символическая константа в форме **GL_TEXTURE*i*_ARB**, где ***i*** должно иметь значение в диапазоне от ***0*** до ***k-1***, где ***k***– максимальное число текстурных блоков.

Если вы используете объекты текстуры, вы можете связать текстуру с текущим текстурным блоком. Текущий текстурный блок будет иметь состояние, содержащееся в текстурном объекте (включая изображение текстуры).

Следующий пример кода в примере **9-9** имеет две части. В первой части создаются два обычных текстурных объекта (в предположении о том, что ***texelss0*** и ***texels1*** содержат изображения текстур). Во второй части два текстурных объекта используются для настройки двух текстурных блоков.

Пример **9-9.** Инициализация текстурных блоков для мультитекстурирования

```
/* Создание обычных текстурных объектов */
GLuint texNames[2];
glGenTextures(2, texNames);
glBindTexture(GL_TEXTURE_2D,texNames[0]);
glTexImage2D(GL_TEXTURE_2D,0,GL_RGBA,32,32,0,
GL_RGBA,GL_UNSIGNED_BYTE,texels0);
glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MIN_FILTER,GL_NEAREST);
glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MAX_FILTER,GL_NEAREST);
glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_WRAP_S,GL_REPEAT);
glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_WRAP_T,GL_REPEAT);
glBindTexture(GL_TEXTURE_2D,texNames[1]);
glTexImage2D(GL_TEXTURE_2D,0,GL_RGBA,16,16,0,
GL_RGBA,GL_UNSIGNED_BYTE,texels1);
glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MIN_FILTER,GL_LINEAR);
glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MAX_FILTER,GL_LINEAR);
glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_WRAP_S,GL_CLAMP_TO_EDGE);
glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_WRAP_T,GL_CLAMP_TO_EDGE);

/* Использование 2 текстурных объектов для настройки 2 текстурных
блоков, */
/* участвующих в мультитекстурировании */
glActiveTextureARB(GL_TEXTURE0_ARB);
glEnable(GL_TEXTURE_2D);
glBindTexture(GL_TEXTURE_2D,texNames[0]);
glTexEnvi(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_REPLACE);
glMatrixMode(GL_TEXTURE);
glLoadIdentity();
glTranslatef(0.5,0.5,0.0);
glRotatef(45.0,0.0,0.0,1.0);
glTranslatef(-0.5,-0.5,0.0);
glMatrixMode(GL_MODELVIEW);
glActiveTextureARB(GL_TEXTURE1_ARB);
glEnable(GL_TEXTURE_2D);
glBindTexture(GL_TEXTURE_2D,texNames[1]);
glTexEnvi(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_MODULATE);
```

Результатом будет то, что текстурированный полигон при обработке будет визуализирован с использованием **2** текстурных блоков. В первом блоке накладывается текстура с изображением ***texels0***, с фильтрацией **GL_NEAREST**, режимом присоединения **GL_REPEAT**, режимом наложения **GL_REPLACE** и поворачивающей текстурной матрицей. После завершения работы первого блока, уже текстурированный полигон направляется во второй блок текстуры **(GL_TEXTURE1_ARB)**, где на него накладывается текстура с изображением ***texels1***, режимом фильтрации **GL_LINEAR**, режимом присоединения **GL_CLAMP_TO_EDGE**, режимом наложения **GL_MODULATE** и текстурной матрицей по умолчанию.

Замечание: Операции по работе с группами атрибутов (с использованием команд **glPushAttrib()**, **glPushClientAttrib()**, **glPopAttrib()** или **glPopClientAttrib()**) сохраняют и восстанавливают состояние текстурирования *для всех блоков текстуры* (но не сохраняют стек текстурных матриц).

9.6.2.3 Указание вершин и их текстурных координат

При мультитекстурировании недостаточно задать один набор текстурных координат. Вам нужно иметь по одному набору для каждой вершины на каждый текстурный блок. Вместо команды **glTexCoord*()** вы должны использовать команду **glMultiTexCoord*ARB()**, которая позволяет задать координаты текстуры и текстурный блок, к которому они относятся.

```
void glMultiTexCoord{1234}{sifd}ARB (GLenum texUnit, TYPE coords);
void glMultiTexCoord{1234}{sifd}vARB (GLenum texUnit, TYPE *coords);
```

Устанавливает текстурные координаты **(*s*, *t*, *r*, *q*)** в значения, передаваемые в аргументе ***coords***. Координаты будут использованы в текстурном блоке ***texUnit***. Возможные значения для аргумента ***texUnit***такие же, как и в команде **glActiveTextureARB()**.

В примере **9-10** треугольнику присваивается два набора текстурных координат, необходимых для мультитекстурирования с двумя активными текстурными блоками.

Пример **9-10.** Указание вершин для мультитекстурирования

```
glBegin(GL_TRIANGLES);
glMultiTexCoord2fARB(GL_TEXTURE0_ARB,0.0,0.0);
glMultiTexCoord2fARB(GL_TEXTURE1_ARB,1.0,0.0);
glVertex2f(0.0,0.0);
glMultiTexCoord2fARB(GL_TEXTURE0_ARB,0.5,1.0);
glMultiTexCoord2fARB(GL_TEXTURE1_ARB,0.5,0.0);
glVertex2f(50.0,100.0);
glMultiTexCoord2fARB(GL_TEXTURE0_ARB,1.0,0.0);
glMultiTexCoord2fARB(GL_TEXTURE1_ARB,1.0,1.0);
glVertex2f(100.0,0.0);
glEnd();
```

Замечание: Если вы используете мультитекстурирование и вызываете **glTexCoord*()**, вы устанавливаете значения координат для первого текстурного блока. Другими словами, использование **glTexCoord*()** эквивалентно использованию **glMultiTexCoord*ARB(GL_TEXTURE0_ARB, ...)**.

В редких случаях, когда вы применяете мультитекстурирование к битовой карте или изображению, вам нужно ассоциировать несколько наборов текстурных координат с каждой позицией растра. То есть, вы должны вызвать **glMultiTexCoord*ARB()** несколько раз, по одному на каждый активный текстурный блок для каждого вызова **glRasterPos*()**. (Поскольку для каждого изображения и битовой карты существует только одна позиция растра, может быть задано лишь по одному набору координат для каждого текстурного блока, что сильно ограничивает эстетические возможности.)

Если вы применяете мультитекстурирование и используете автоматическую генерацию текстурных координат, **glActiveTextureARB()** задает текстурный блок, на который воздействуют следующие команды:

- **glTexGen*(...);**
- **glEnable(GL_TEXTURE_GEN_*);**
- **glDisable(GL_TEXTURE_GEN_*);**

Если вы применяете мультитекстурирование и передаете координаты текстуры в вершинном массиве, используйте команду **glClientActiveTextureARB()**, устанавливающую текстурный блок, для которого команда **glTexCoordPointer()** задает текстурные координаты.

```
void glCLientActiveTextureARB (GLenum texUnit);
```

Выбирает текстурный блок, для которого координаты текстуры задаются в вершинном массиве. ***texUnit***– это символическая константа в форме **GL_TEXTURE*i*_ARB**, могущая принимать те же значения, что и в команде **glActiveTextureARB()**.

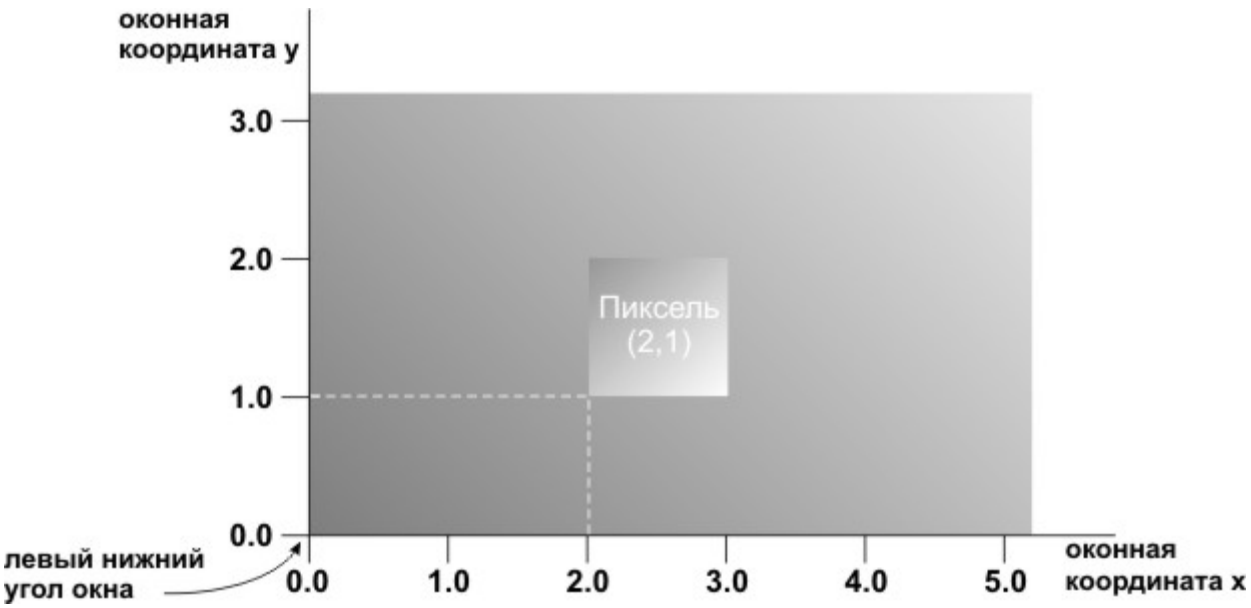
Глава 10. Буфер кадра

Целью практически каждой графической программы является отображение изображений на экране. Экран представляет собой прямоугольный массив пикселей, каждый из которых способен отобразить миниатюрный квадрат цвета, соответствующий цвету точки изображения. После этапа растеризации (включая текстурирование и туман) данные все еще не являются пикселями – они являются фрагментами. Каждый фрагмент имеет координаты, соответствующие пикселю, а также значения цвета и глубины. Далее каждый из фрагментов проходит через серию текстов и операций.

Если фрагмент переживает тесты и операции, он становится пикселем. Чтобы нарисовать пиксели, требуется знать какого они цвета. Эта информация сохраняется в цветовом буфере. Поскольку данные хранятся одинаково для каждого пикселя, хранилище всех пикселей и называется *буфером*. Разные буферы могут содержать различное количество информации на каждый пиксель, но внутри одного буфера на каждый пиксель приходится одинаковое количество информации. Буфер, который хранит по одному биту информации на каждый пиксель, называется *битовой плоскостью*.

Как показано на рисунке **10-1**, нижний левый пиксель **OpenGL**– это пиксель (***0***, ***0***), что соответствует оконным координатам нижнего левого угла региона размером **1x1**, занятого этим пикселем. Вообще пиксель с координатами (***x***, ***y***) заполняет область, ограниченную слева – ***x***, справа – ***x+1***, снизу – ***y***, сверху – ***y+1***.

Рисунок **10-1**. Область, занимаемая пикселем



В качестве примера буфера давайте подробнее рассмотрим цветовой буфер, содержащий цветовую информацию, которая должна быть отображена на экране. Предположим, что экран имеет ширину **1280** пикселей, высоту **– 1024** пикселя и **24** бита на цвет – то есть существует $2^{24}-1$ **(**или **16,777,216)** цветов, которые можно отобразить. Поскольку **24** бита являются тремя байтами **(**по **8** бит на **1** байт**)**, цветовой буфер из нашего примера должен хранить, как минимум, по **3** байта данных на каждый из **1,310,720 (1280*1024)** пикселей экрана. Конкретная аппаратная система может содержать больше или меньше пикселей на экране, а так же больше или меньше цветовых данных на пиксель. Однако любой цветовой буфер содержит одинаковое количество информации на каждый пиксель.

Цветовой буфер – это только один из нескольких буферов, хранящих информацию о пикселе. Например, информация о глубине для каждого пикселя хранится в буфере глубины. Сам цветовой буфер может состоять из нескольких буферов. Системный *буфер кадра* **(*framebuffer*)** объединяет **(**включает в себя**)** все буферы. Вы не видите содержимого буферов **(**за исключением цветового буфера или буферов**)** непосредственно; вместо этого они используются для таких целей, как удаление невидимых поверхностей, антиалиасинга всей сцены, трафарета, рисования плавного движения и других.

Эта глава описывается все буферы, которые могут существовать в **OpenGL** и методы их использования. Также обсуждаются серии тестов и операций, которые производятся до того, как какие-либо данные записываются в цветовой буфер. Наконец, здесь объясняется, как использовать аккумуляторный буфер, используемый для накопления изображений, отображаемых в цветовом буфере.

10.1 Буферы и их использование

Система **OpenGL** может манипулировать следующими буферами:

- Цветовые буферы: передний левый, передний правый, задний левый, задний правый, а также любое число дополнительных цветовых буферов.
- Буфер глубины
- Буфер трафарета
- Аккумулирующий буфер

Какие буферы поддерживаются, и сколько бит на пиксель хранит каждый буфер, определяет ваша реализация **OpenGL**. Кроме того, у вас может быть множество типов окон, поддерживающих различные буферы. Список всех параметров команды **glGetIntegerv()**, позволяющих опрашивать **OpenGL** на предмет количества бит на пиксель в буферах ее реализации приведен в таблице **10-1.**

Замечание: Если вы работаете в системе XWindow, вам гарантирован, как минимум, один визуальный объект с цветовым буфером в режиме RGBA, а также ассоциированные буферы трафарета, глубины и аккумуляции, имеющие ненулевые размеры цветовых компонент. Кроме того, если ваша система поддерживает визуальный объект Pseudo-Color вам также гарантировано наличие визуального объекта с цветовым буфером в режиме цветовых индексов и ассоциированных буферов глубины и трафарета. Для получения информации о системе, вам, возможно, придется использовать команду **glXGetConfig()**.

Таблица **10-1.** Параметры опроса для выяснения числа бит на пиксель в разных буферах

Параметр	**Значение**
GL_RED_BITS, GL_GREEN_BITS, GL_BLUE_BITS, GL_ALPHA_BITS	количество бит на **R, G, B** и **A**компоненты в цветовом буфере

GL_INDEX_BITS	количество бит на индекс в цветовом буфере
GL_DEPTH_BITS	количество бит на пиксель в буфере глубины
GL_STENCIL_BITS	количество бит на пиксель в буфере трафарета
GL_ACCUM_RED_BITS, GL_ACCUM_GREEN_BITS, GL_ACCUM_BLUE_BITS, GL_ACCUM_ALPHA_BITS	количество бит на **R, G, B** и **A**компоненты в буфере аккумулятора

10.1.1 Типы буферов

10.1.1.1 Цветовые буферы

Цветовые буферы – это те, что обычно рисуются на экране. Они содержат либо цветовые индексы, либо данные **RGB** цвета, а также могут содержать значения альфа. Реализация **OpenGL**, поддерживающая стереоскопический просмотр, имеет левый и правый цветовые буферы для левого и правого стереоскопических изображений. Если стерео не поддерживается, используются только левые буферы. Аналогично, системы с двойной буферизацией поддерживают передний и задний буферы, а системы с однократной буферизацией – только передние. Любая реализация **OpenGL** должна как минимум предоставлять левый передний цветовой буфер.

Опционально могут поддерживаться дополнительные цветовые буферы, которые не отображаются на экране. **OpenGL** не определяет конкретных нужд, для которых должны использоваться эти буферы, так что вы можете определять и использовать их как угодно. Например, вы можете использовать их для хранения часто применяемых изображений. Затем вместо того, чтобы перерисовывать изображение, вы можете просто скопировать его из дополнительного буфера в один из основных.

Для выяснения того, поддерживает ли ваша система стерео просмотр (то есть, присутствуют ли в ней левые и правые буферы) или двойную буферизацию (то есть, присутствуют ли в ней передний и задний буферы), вы можете использовать команду **glGetBooleanv()** с аргументами **GL_STEREO** или **GL_DOUBLEBUFFER** соответственно. Для того, чтобы выяснить количество дополнительных цветовых буферов в системе (если они вообще есть), используйте **glGetIntegerv()** с аргументом **GL_AUX_BUFFERS**.

10.1.1.2 Буфер глубины

Буфер глубины хранит значение глубины для каждого пикселя. Глубина обычно измеряется в виде дистанции от глаза наблюдателя, таким образом, пиксели с большими значениями глубины затираются пикселями с меньшими значениями. Однако, это всего лишь обычное соглашение, и поведение буфера глубины может быть изменено. Буфер глубины иногда называется ***z** буфером* (***z*** в этом термине берет свое начало от того факта, что ***x***и ***y*** определяют горизонтально и вертикальное положение на экране, а ***z***– дистанцию перпендикулярно плоскости экрана).

10.1.1.3 Буфер трафарета

Одно из назначений буфера трафарета заключается в том, чтобы блокировать рисование определенных областей на экране (по аналогии с реальным трафаретом). Например, если вы хотите нарисовать изображение так, как оно выглядело бы через железную решетку, вам нужно сохранить изображение решетки в буфере трафарета и нарисовать всю сцену. Буфер трафарета не позволит частям изображения, которые были бы не видны через решетку, быть нарисованными на экране. Таким образом, если ваше приложение – это симулятор вождения, вы можете нарисовать все детали и инструменты, находящиеся внутри машины, только единожды, а затем, в процессе движения, машины обновлять только объекты снаружи машины.

10.1.1.4 Аккумуляторный буфер

Аккумуляторный буфер цветовые данные **RGBA** точно так же как цветовые буферы в режиме **RGBA (**результат использования буфера аккумулятора в индексном режиме не определен**).** Обычно он используется для сборки серии последовательных изображений в одно составное изображение**.** С помощью этого метода вы можете использовать такую технику**,** как антиалиасинг всей сцены**.** Для этого нужно аккумулировать серию изображений одной и той же сцены **(**обычно под слегка разными углами**)** и усреднить их между собой для получения результирующих цветовых величин**.** Эти величины впоследствии рисуются в цветовом буфере**.** Вы не производите рисование в аккумуляторный буфер непосредственно**;** аккумуляторные операции**,** которые обычно передают данные в или из буфера аккумулятора**,** всегда производятся над прямоугольными блоками**.**

10.1.2 Очистка буферов

Очистка экрана **(**и любых буферов**)** это обычно одна из самых дорогих операций в графических программах **–** на мониторе с разрешением **1280x1024** она затрагивает более миллиона пикселей**.** В простых графических приложениях очистка экрана может занимать больше времени**,** чем требуется на все остальное рисование**.** Если вам нужно очистить не только цветовой буфер**,** но еще и буфер глубины**,** и буфер трафарета**,** время выполнения очистки увеличивается втрое**.**

Для решения этой проблемы некоторые машины имеют в своем составе аппаратуру**,** позволяющую очищать более одного буфера одновременно**.** Команды очистки **OpenGL** построены с учетом преимуществ таких архитектур**.** Сначала вы задаете для каждого буфера величины**,** которые будут записаны в него при очистке**.** Затем вы выполняете единственную команду**,** передавая ей список всех буферов**,** которые должны быть очищены**.** Если аппаратура способна производить одновременную очистку**,** все указанные буферы очищаются одновременно**;** если нет **–** они очищаются последовательно**.**

Следующие команды устанавливают очищающие значения для каждого буфера**.**

```
void glClearColor (GLclampf red, GLclampf green, GLclampf blue, GLclampf alpha);
void glClearIndex (GLfloat index);
void glClearDepth (GLclampd depth);
void glClearStencil (GLint s);
void glClearAccum (GLfloat red, GLfloat green, GLfloat blue, GLfloat alpha);
```

Задают очищающие значения для цветового буфера **(**в **RGBA** режиме**),** цветового буфера **(**в индексном режиме**),** буфера глубины**,** буфера трафарета и аккумуляторного буфера**.** Аргументы типов **GLclampf** и **GLclampd (**усеченное с плавающей точкой одинарной точности и усеченное с плавающей точкой двойной точности**)** усекаются по границам диапазона **[0.0**, **1.0] (**то есть должны лежать в нем**).** Очищающая глубина по умолчанию равна **1.0;** все остальные очищающие величины **– 0.0.** Величины установленные этими командами остаются в силе до тех пор**,** пока они не будут изменены теми же командами**.**

После выбора очищающих величин для буферов можно очистить их командой **glClear().**

```
void glClear (GLbitfield mask);
```

Очищает заданные буферы**.** Значение ***mask*–** это побитовое логическое ИЛИ некоторой комбинации значений **GL_COLOR_BUFFER_BIT**, **GL_DEPTH_BUFFER_BIT**, **GL_STENCIL_BUFFER_BIT** и **GL_ACCUM_BUFFER_BIT**, задающих нужные буферы**. GL_COLOR_BUFFER_BIT** очищает либо цветовой буфер **RGBA,** либо индексный цветовой буфер в зависимости от того**,** в каком режиме система находится в момент вызова**.** Когда вы очищаете цветовой или цветоиндексный буфер**,** очищаются все цветовые

буферы в момент вызова доступные для записи. Тест принадлежности пикселей, тест отреза и цветовое микширование, если они активизированы, влияют на работу очищающих команд. Маскирующие операции, такие как **glColorMask()** и **glIndexMask()** также работают. Альфа тест, тест трафарета и тест глубины не оказывают влияния на работу **glClear()**.

10.1.3 Выбор цветовых буферов для записи и чтения

Результаты операций рисования и считывания могут в любой цветовой буфер и из любого цветового буфера: переднего, заднего, переднего левого, переднего правого, заднего левого, заднего правого и любого из дополнительных буферов. Вы можете выбрать конкретный буфер, который будет источником для чтения или местом назначения для записи. Для операций рисования вы можете выбрать одновременно более одного буфера – рисование в них будет происходить одновременно. Используйте команду **glDrawBuffer()** для выбора буфера, в который буфет производиться рисование и **glReadBuffer()** для выбора буфера, который буфет выступать в качестве источника данных для команд **glReadPixels()**, **glCopyPixels()**, **glCopyTexImage*()** и **glCopyTexSubImage*()**.

Если вы используете двойную буферизацию, вам обычно следует рисовать только в задний буфер (и переключать буферы после завершения рисования). В некоторых случаях вам может потребоваться работать с дважды буферизированным окном так, как если бы оно было окном с однократной буферизацией. Это можно сделать с помощью команды **glDrawBuffer()**, активизировав для записи оба буфера – передний и задний.

glDrawBuffer() также используется при визуализации стерео изображений (**GL*LEFT**и **GL*RIGHT**) и для вывода в дополнительные цветовые буферы (**GL_AUX*i***).

```
void glDrawBuffer (GLenum mode);
```

Выбирает цветовые буферы доступные для записи и очистки и запрещает для записи и очистки буферы, выбранные предыдущим вызовом той же команды. В каждый конкретный момент для записи и очистки может быть выбрано несколько буферов. Значением аргумента ***mode*** может быть любая из следующих констант:

GL_FRONT	**GL_FRONT_LEFT**	**GL_AUX*i***
GL_BACK	**GL_FRONT_RIGHT**	**GL_FRONT_AND_BACK**
GL_LEFT	**GL_BACK_LEFT**	**GL_NONE**
GL_RIGHT	**GL_BACK_RIGHT**	

Аргументы, в которых не упоминается **LEFT** и **RIGHT**, обозначают и левый и правый буферы. Точно так же аргументы, в которых нет **FRONT** и **BACK**, относятся и переднему, и к заднему буферу. ***i*** в **GL_AUX*i*** идентифицирует конкретный дополнительный буфер. По умолчанию ***mode*** равно **GL_FRONT** для режима с однократной буферизацией и **GL_BACK** для режима с двойной буферизацией.

Замечание: Вы можете активизировать запись в несуществующие буферы, если в том же вызове команды **glDrawBuffer()** открывается для записи хотя бы один из существующих. Если же ни один из указанных в *mode* буферов не существует, будет сгенерирована ошибка.

```
void glReadBuffer (GLenum mode);
```

Выбирает цветовой буфер, который будет служить в качестве источника для чтения пикселей последующими вызовами команд **glReadPixels()**, **glCopyPixels()**,

glCopyTexImage*(), **glCopyTexSubImage*()** и **glCopyConvolutionFilter*()** и запрещает для чтения буфер, выбранный предыдущим вызовом той же команды. Значением аргумента ***mode*** может быть любая из следующих констант:

GL_FRONT	**GL_FRONT_LEFT**	**GL_AUX*i***
GL_BACK	**GL_FRONT_RIGHT**	
GL_LEFT	**GL_BACK_LEFT**	
GL_RIGHT	**GL_BACK_RIGHT**	

Буферы для **glReadBuffers()** те же самые, что и для команды **glDrawBuffers()**. По умолчанию ***mode*** равно **GL_FRONT** для режима с однократной буферизацией и **GL_BACK** для режима с двойной буферизацией.

Замечание: Вы можете активизировать чтение только из существующего буфера. В противном случае будет сгенерирована ошибка.

10.1.4 Маскирование буферов

До того, как **OpenGL** записывает данные в доступные цветовые, глубинный или трафаретный буферы, к данным применяется операция маскирования, заданная одной из следующих команд. Над записываемыми данными и маской производится побитовое логическое И.

```
void glIndexMask (GLuint mask);
void glColorMask (GLboolean red, GLboolean green, GLboolean blue, GLboolean
alpha);
void glDepthMask (GLboolean flag);
void glStencilMask (GLuint mask);
```

Устанавливают маски, управляющие записью в буферы. Маска, заданная командой **glIndexMask()** применяется только в индексном режиме. Если в маске в определенном разряде оказывается **1**, соответствующий разряд записывается в цветоиндексный буфер; если там оказывается **0** – бит не записывается. Точно также **glColorMask()** работает только в режиме **RGBA**. Величины ***red***, ***green***, ***blue*** и ***alpha*** управляют тем, будут ли записываться соответствующие компоненты **(GL_TRUE** означает, что компонент будет записан**)**. Если ***flag*** в **glDepthMask()** равен **GL_TRUE**, буфер глубины доступен для записи; иначе – нет. Маска для **glStencilMask()** используется с данными трафарета точно так же, как маска для индексного цветового буфера. Значения по умолчанию для всех булевских значений – **GL_TRUE**, а для двух целых масок – все единицы.

С помощью цветового маскирования в индексном режиме вы можете совершать массу трюков. Например, вы можете использовать каждый бит индекса, как отдельный слой и устанавливать взаимодействие между отдельными слоями с помощью определенных установок в цветовой карте. Вы можете создавать подслои и надслои, а также выполнять так называемую анимацию цветовой палитры. Маскирование в **RGBA** режиме используется несколько реже, но вы можете использовать его для загрузки трех разных изображений в красные, зеленые и синие битовые плоскости, например.

Запрещение буфера глубины для записи можно использовать в том случае, если для последовательности кадров используется один и тот же задний план, и вы хотите добавить несколько деталей, которые могут загораживаться частями заднего плана. Например, предположим, что ваш задний план – это лес, и вы хотите нарисовать серию кадров с одними и теми же деревьями, но с объектами, которые движутся среди них. После того, как деревья нарисованы, и их глубина записана в буфер глубины, сохраните изображение деревьев и нарисуйте новые объекты при заблокированном для записи буфере глубины. До тех пор, пока объекты не накладываются друг на друга,

картинка выглядит так, как нужно. Чтобы нарисовать следующий кадр, восстановите изображение деревьев и продолжайте. Вам не нужно восстанавливать значения в буфере глубины. Этот трюк особенно полезен, если задний план весьма сложен – настолько сложен, что его изображение намного быстрее скопировать в цветовой буфер, чем заново рассчитать из геометрических данных.

Маскирование буфера трафарета позволяет вам использовать многобитный буфер трафарета для хранения нескольких трафаретов (по одному на бит)..

Замечание: Маска, задаваемая с помощью **glStencilMask()** управляет тем, какие из битовых плоскостей трафарета доступны для записи. Эта маска никак не связана с маской, задаваемой в качестве третьего параметра **glStencilFunc()**. Та маска определяет, какие битовые плоскости должны приниматься в расчет трафаретной функцией.

10.2 Тестирование и операции над фрагментами

Когда вы рисуете геометрические фигуры, текст или изображения на экране, **OpenGL**производит определенные расчеты для поворотов, масштабирования, расчета освещенности, перспективного проецирования объектов, вычисления затрагиваемых пикселей и определения цвета, которым они должны быть нарисованы. После того, как **OpenGL**определит, что нужно сгенерировать отдельный фрагмент, а также определит его цвет, остается выполнить еще несколько этапов обработки, которые управляют тем, как должен отображаться фрагмент, и должен ли он вообще быть отображен в виде пикселя в буфере кадра. Например, если он лежит за пределами прямоугольной области или находится дальше от точки наблюдения, чем пиксель, уже находящийся в буфере кадра, он нарисован не будет. На другом этапе цвет фрагмента накладывается на цвет пикселя, находящегося в буфере кадра.

В этом разделе описывается полный набор тестов, которые фрагмент должен пройти, чтобы попасть в буфер кадра, и все возможные финальные операции, которые совершаются над фрагментом во время его записи в буфер. Тесты и операции выполняются в следующем порядке; если фрагмент не проходит один из ранних тестов, никакие более поздние тесты и операции над ним не производятся.

1. Тест отреза **(scissor test)**
2. Альфа тест **(alpha test)**
3. Тест трафарета **(stencil test)**
4. Тест глубины **(depth test)**
5. Цветовое наложение **(blending)**
6. Цветовое микширование **(dithering)**
7. Логические операции **(logical operations)**

Все эти тесты и операции подробно описаны в следующих подразделах.

10.2.1 Тест отреза

С помощью команды **glScissor()** вы можете задать прямоугольную область окна и ограничить рисование только этой областью. Фрагмент проходит тест, если он лежит внутри заданного прямоугольника.

```
void glScissor (GLint x, GLint y, GLsizei width, GLsizei height);
```

Устанавливает положение и размер прямоугольника отреза (также известного как **scissorbox**). Аргументы задают положение левого нижнего угла (***x***, ***y***) и размеры прямоугольника (***width***, ***height***). (Измерение производится в пикселях экрана.) Тест проходят пиксели, которые лежат внутри прямоугольника. Тест отреза активизируется

и деактивируется с помощью аргумента **GL_SCISSOR_TEST** команд **glEnable()** и **glDisable()**. По умолчанию тест деактивирован, а прямоугольник отреза совпадает с размерами окна.

Тест отреза – это просто версия теста трафарета, использующая прямоугольную область экрана. Очень просто создать ошеломляюще быструю аппаратную реализацию отреза, тогда как та же система может намного медленнее работать с трафаретами – может быть потому, что трафареты реализуются программно.

Дополнительно: Более сложное назначение отрезов заключается в возможности нелинейного проецирования. Сначала разделите окно на сетку областей, задавая такие параметры порта просмотра и отреза, которые в каждый конкретный момент ограничивают область рисования одним регионом. Затем проецируйте всю сцену в каждый регион с использованием различных проекционных матриц.

Чтобы выяснить, активизирован ли тест отреза, и получить значения, определяющие прямоугольник отреза, используйте **GL_SCISSOR_TEST** с командой **glIsEnabled()** и **GL_SCISSOR_BOX** с командой **glGetIntegerv()** соответственно.

10.2.2 Альфа тест

Альфа тест в **RGBA** режиме позволяет вам принимать или отвергать фрагмент в зависимости от его значения альфа. Альфа тест активизируется и деактивируется с помощью аргумента **GL_ALPHA_TEST** команд **glEnable()** и **glDisable()**. Чтобы выяснить, активизирован ли альфа тест, используйте аргумент **GL_ALPHA_TEST** с командой **glIsEnabled()**.

Если тест активизирован, он сравнивает входящую величину альфа с некоторым заданным значением. Фрагмент принимается или отвергается в зависимости от результата этого сравнения. И сравниваемое значение, и функция сравнения устанавливаются командой **glAlphaFunc()**. По умолчанию сравниваемое значение равно **0**, функция сравнения – **GL_ALWAYS**, а сам тест деактивирован. Чтобы выяснить текущую функцию сравнения или сравниваемое значение, используйте аргумент **GL_ALPHA_FUNC** или **GL_ALPHA_TEST_REF** с командой **glGetIntegerv()**.

```
void glAlphaFunc (GLenum func, GLclampf ref);
```

Устанавливает сравниваемое значение и функцию сравнения для альфа теста. Сравниваемое значение ***ref*** усекается до диапазона **[0; 1]**. Возможные значения для функции сравнения ***func*** и их смысл приводятся в таблице **10-2**.

Таблица **10-2.** Значения параметров **glAlphaFunc()**

Параметр	Значение
GL_NEVER	фрагмент никогда не принимается
GL_ALWAYS	фрагмент принимается всегда
GL_LESS	фрагмент принимается, если его альфа **<** заданного значения
GL_LEQUAL	фрагмент принимается, если его альфа **<=** заданного значения
GL_EQUAL	фрагмент принимается, если его альфа **=** заданному значению
GL_GEQUAL	фрагмент принимается, если его альфа **>=** заданного значения
GL_GREATER	фрагмент принимается, если его альфа **>** заданного значения
GL_NOTEQUAL	фрагмент принимается, если его альфа **! =** заданному значению

Одним из предназначений альфа теста является реализация алгоритма прозрачности. Визуализируйте всю вашу сцену дважды, первый раз принимая только фрагменты с альфа величиной равной **1**, а второй раз – только с альфа величиной не равной **1**.

Включите буфер глубины для обоих проходов, но при втором проходе запретите запись в него.

Кроме того, альфа тест, как и цветовое микширование, позволяет реализовать биллбоардинг.

10.2.3 Тест трафарета

Тест трафарета может производиться, только если имеется буфер трафарета. (Если буфера нет, фрагмент всегда проходит тест успешно.) Тест трафарета сравнивает некоторое заданное значение со значением, сохраненным для пикселя в буфере трафарета. В зависимости от результата сравнения величина в буфере трафарета изменяется. Вы можете выбрать функцию сравнения, сравниваемое значение и метод модификации с помощью команд **glStencilFunc()** и **glStencilOp()**.

```
void glStencilFunc (GLenum func, GLint ref, GLuint mask);
```

Устанавливает функцию сравнения (***func***), сравниваемое значение (***ref***) и маску (***mask***) для использования тестом трафарета. Значение ***ref*** сравнивается с величиной в буфере трафарета с использованием функции сравнения, однако сравнение применяется только к тем битам, которые остаются после маскирования маской ***mask***. В качестве функции сравнения могут быть выбраны **GL_NEVER**, **GL_ALWAYS**, **GL_LESS**, **GL_LEQUAL**, **GL_GEQUAL**, **GL_GREATER**, **GL_EQUAL** или **GL_NOTEQUAL**. Например, если выбрана **GL_LESS**, фрагмент проходит тест только в случае, если ***ref*** меньше, чем величина в буфере трафарета. Если буфер трафарета содержит ***s***битовых плоскостей, то над величиной в буфере трафарета и сравниваемым значением ***ref*** производится логическое И с ***s*** младших битов ***mask***. Маскирование производится до сравнения. Все маскированные величины интерпретируются как неотрицательные.

Тест трафарета активизируется и деактивируется с помощью аргумента **GL_STENCIL_TEST** команд **glEnable()** и **glDisable()**. По умолчанию тест деактивирован, ***func*** равно **GL_ALWAYS**, ***mask*** – всем единицам, а ***ref*** -- **0**.

```
void glStencilOp (GLenum fail, GLenum zfail, GLenum zpass);
```

Задает, как данные в буфере трафарета должны измениться в случае прохождения или непрохождения теста трафарета. Три функции ***fail***, ***zfail*** и ***zpass*** могут принимать значения **GL_KEEP**, **GL_ZERO**, **GL_REPLACE**, **GL_INCR**, **GL_DECR** или **GL_INVERT**. Эти функции соответствуют (по порядку) сохранению предыдущего значения, замене его **0**, замене его сравниваемой величиной, увеличению величины в буфере на **1**, уменьшению величины на **1** и побитовой инверсии. Результат функций инкремента и декремента усекается до отрезка между **0** и максимальным беззнаковым целым значением, которое может быть сохранено в буфере трафарета (если буфер трафарета содержит ***s*** битовых плоскостей, это значение равно $2^s - 1$). Функция ***fail*** применяется, если фрагмент не проходит тест трафарета, функция ***zfail***– если фрагмент прошел тест трафарета, но не прошел тест глубины, а ***zpass***– если фрагмент прошел и тест трафарета, и тест глубины (или если тест глубины деактивирован).

10.2.3.1 Опрос трафарета

Вы можете получить значения всех **6** параметров, связанных с тестом трафарета, используя команду **glGetIntegerv()** и одну из констант, перечисленных в таблице **10-3**. Выяснить, активизирован ли тест трафарета, вы можете с помощью аргумента **GL_STENCIL_TEST** команды **glIsEnabled()**.

Таблице **10-3.** Константы для запросов состояния теста трафарета

Константаb>	Значение
GL_STENCIL_FUNC	функция сравнения
GL_STENCIL_REF	сравниваемое значение
GL_STENCIL_VALUE_MASK	маска трафарета
GL_STENCIL_FAIL	действие при непрохождении теста трафарета
GL_STENCIL_PASS_DEPTH_FAIL	действие при прохождении тест трафарета и провале теста глубины
GL_STENCIL_PASS_DEPTH_PASS	действие при прохождении тест трафарета и прохождении теста глубины

10.2.3.2 Примеры применения трафарета

Наверное, самое типичное использование теста трафарета заключается в маскировании экранной области необычной формы для ограничения области рисования этой формой. Чтобы это сделать, заполните трафарет нулями, а затем нарисуйте в буфере трафарета нужную форму единицами. Вы не можете рисовать геометрию непосредственно в буфер трафарета, но вы можете добиться того же результата, рисуя в цветовой буфер и выбрав подходящее значение для функции ***zpass*** (например, **GL_REPLACE**). (Кроме того, для рисования пикселей непосредственно в буфер трафарета, вы можете использовать **glDrawPixels()**.) Во время рисования величина (в данном случае сравниваемая) записывается и в буфер трафарета. Чтобы избежать воздействия рисования в буфере трафарета на содержимое цветового буфера, установите цветовую маску в **0** (или **GL_FALSE**). Возможно, вы также захотите запретить запись в буфер глубины.

После того, как вы определили область трафарета, установите сравниваемое значение в **1**, а функцию сравнения в **GL_EQUAL**. Не модифицируйте содержимое буфера трафарета во время рисования.

Пример **10-1** демонстрирует подобное использование теста трафарета. Рисуются два торуса и вырез в середине сцены. Внутри выреза рисуется сфера. В этом примере рисование в буфер трафарета осуществляется только во время перерисовки окна, так что после создания формы трафарета цветовой буфер очищается.

Пример **10-1**. Использование теста трафарета: файл **stencil.cpp**

```
#include <glut.h>

#define YELLOWMAT 1
#define BLUEMAT 2

void init()
{
      GLfloat yellow_diffuse[]={0.7,0.7,0.0,1.0};
      GLfloat yellow_specular[]={1.0,1.0,1.0,1.0};

      GLfloat blue_diffuse[]={0.1,0.1,0.7,1.0};
      GLfloat blue_specular[]={0.1,1.0,1.0,1.0};

      GLfloat position_one[]={1.0,1.0,1.0,0.0};

      glNewList(YELLOWMAT,GL_COMPILE);
            glMaterialfv(GL_FRONT,GL_DIFFUSE,yellow_diffuse);
            glMaterialfv(GL_FRONT,GL_SPECULAR,yellow_specular);
            glMaterialf(GL_FRONT,GL_SHININESS,64.0);
      glEndList();

      glNewList(BLUEMAT,GL_COMPILE);
            glMaterialfv(GL_FRONT,GL_DIFFUSE,blue_diffuse);
            glMaterialfv(GL_FRONT,GL_SPECULAR,blue_specular);
            glMaterialf(GL_FRONT,GL_SHININESS,45.0);
```

```
	glEndList();

	glLightfv(GL_LIGHT0,GL_POSITION,position_one);

	glEnable(GL_LIGHT0);
	glEnable(GL_LIGHTING);
	glEnable(GL_DEPTH_TEST);

	glClearStencil(0x0);

	glEnable(GL_STENCIL_TEST);
}

void display()
{
	glClear(GL_COLOR_BUFFER_BIT|GL_DEPTH_BUFFER_BIT);

	//Синяя сфера, где трафарет =1
	glStencilFunc(GL_EQUAL,0x1,0x1);
	glStencilOp(GL_KEEP,GL_KEEP,GL_KEEP);
	glCallList(BLUEMAT);
	glutSolidSphere(0.5,15,15);

	//Желтый бублик, где трафарет <>1
	glStencilFunc(GL_NOTEQUAL,0x1,0x1);
	glPushMatrix();
	glRotatef(45,0.0,0.0,1.0);
	glRotatef(45,0.0,1.0,0.0);
	glCallList(YELLOWMAT);
	glutSolidTorus(0.275,0.85,15,15);
	glPushMatrix();
		glRotatef(90.0,1.0,0.0,0.0);
		glutSolidTorus(0.275,0.85,15,15);
	glPopMatrix();
	glPopMatrix();
}

void reshape(int w,int h)
{
	glViewport(0,0,(GLsizei)w,(GLsizei)h);

	//Создать алмазоподобную область трафарета
	glMatrixMode(GL_PROJECTION);
	glLoadIdentity();
	if (w<=h)
		gluOrtho2D(-3.0,3.0,-3.0*(GLfloat)h/(GLfloat)w,
				3.0*(GLfloat)h/(GLfloat)w);
	else
		gluOrtho2D(-3.0*(GLfloat)w/(GLfloat)h,
				3.0*(GLfloat)w/(GLfloat)h,-3.0,3.0);
	glMatrixMode(GL_MODELVIEW);
	glLoadIdentity();

	glClear(GL_STENCIL_BUFFER_BIT);
	glStencilFunc(GL_ALWAYS,0x1,0x1);
	glStencilOp(GL_REPLACE,GL_REPLACE,GL_REPLACE);
	glBegin(GL_QUADS);
		glVertex2f(-1.0,0.0);
		glVertex2f(0.0,1.0);
		glVertex2f(1.0,0.0);
		glVertex2f(0.0,-1.0);
	glEnd();
```

```
        glMatrixMode(GL_PROJECTION);
        glLoadIdentity();
        gluPerspective(45.0,(GLfloat)w/(GLfloat)h,3.0,7.0);
        glMatrixMode(GL_MODELVIEW);
        glLoadIdentity();
        glTranslatef(0.0,0.0,-5.0);
}

int main(int argc, char **argv)
{
        glutInit(&argc,argv);

glutInitDisplayMode(GLUT_SINGLE|GLUT_RGB|GLUT_DEPTH|GLUT_STENCIL);
        glutInitWindowSize(400,400);
        glutCreateWindow("Using the Stencil Test");
        init();
        glutDisplayFunc(display);
        glutReshapeFunc(reshape);
        glutMainLoop();
        return 0;
}
```

Далее приводятся другие примеры использования теста трафарета.

Предположим, что вы рисуете закрытый выпуклый объект (или несколько подобных объектов, при условии, что они не пересекают и не включают друг друга), созданный из нескольких полигонов, и у вас есть отсекающая плоскость, которая может отсекать или не отсекать часть этого объекта. Предположим, что в случае, когда плоскость пересекает объект, вы хотите закрыть его поверхностью постоянного цвета, а не рассматривать его внутренности. Чтобы это сделать, установите буфер трафарета в нули и начинайте рисование, активизировав тест трафарета и установив функцию сравнения в **GL_ALWAYS**. Инвертируйте значения в битовых плоскостях буфера трафарета каждый раз, когда фрагмент принимается. После того, как все объекты нарисованы, областям, не требующим закрытия, в буфере трафарета будут соответствовать **0**, а областям, где оно требуется **– 1**. Установите трафаретную функцию таким образом, чтобы рисование совершалось только так, где индекс трафарета не равен **0** и нарисуйте большой полигон закрывающего цвета размером с экран.

Предположим, что у вас есть полупрозрачная поверхность, состоящая из нескольких, слегка перекрывающихся полигонов. Если вы просто используете цветовое наложение, части нижележащих объектов будут закрываться более чем одной прозрачной поверхностью, что неправильно. Чтобы каждый фрагмент накрывался только одной частью поверхности, используйте буфер трафарета. Установите буфер трафарета в нули, рисуйте, только если плоскость трафарета имеет значение **0**, и при рисовании инкрементируйте значение в плоскости трафарета.

Предположим, что вам нужно отобразить шаблонированное изображение. Вы можете сделать это, записав рисунок шаблона в буфер трафарета, и производя рисование с учетом содержимого буфера трафарета. После того, как рисунок шаблона записан, во время рисования изображения буфер трафарета не изменяется.

10.2.4 Тест глубины

Для каждого пикселя на экране, буфер глубины следит за дистанцией между точкой наблюдения и объектом, занимающим этот пиксель. Затем, если тест глубины проходит, входящая величина глубины заменяет ту, что уже находится в буфере.

В общем случае буфер глубины используется для удаления невидимых поверхностей. Новый цвет для пикселя, рисуется только в том случае, если соответствующий объект ближе, чем предыдущий. Таким образом, после визуализации всей сцены на экране остаются только ничем не загороженные объекты. В самом начале очищающее значение для буфера глубины, это величина, равная наибольшей возможной дистанции от точки наблюдения, то есть любой объект ближе, чем эта величина. Если вы собираетесь использовать буфер глубины именно так, все, что вам нужно сделать, это активизировать тест глубины командой **glEnable()** с аргументом **GL_DEPTH_TEST** и не забывать очищать буфер глубины перед перерисовкой каждого кадра. Вы также можете выбрать иную функцию сравнения для теста на глубину.

```
void glDepthFunc (GLenum func);
```

Устанавливает функцию сравнения для теста глубины. Возможными значениями для ***func*** являются **GL_NEVER**, **GL_ALWAYS**, **GL_LESS**, **GL_LEQUAL**, **GL_EQUAL**, **GL_GEQUAL**, **GL_GREATER** и **GL_NOTEQUAL**. Входящий фрагмент проходит тест глубины в том случае, если его ***z*** координата относится к значению, хранимому в буфере глубины, указанным образом. Функция по умолчанию – это **GL_LESS**, что означает прохождение фрагментом теста в том случае, если его глубина меньше хранимой в буфере. В этом случае величина ***z*** представляет собой дистанцию от объекта до точки наблюдения, и меньшие величины означают, что соответствующий объект находится ближе к наблюдателю.

10.2.5 Цветовое наложение, цветовое микширование и логические операции

Как только входящий фрагмент прошел все тесты, его можно комбинировать с содержимым цветового буфера одним из нескольких методов. Простейшим методом является перезапись уже существующих значений. Альтернативно, если вы работаете в **RGBA** режиме и хотите сделать, чтобы фрагмент был полупрозрачным или сглаженным, вы можете усреднить его значение со значением, уже находящимся в буфере (цветовой наложение). В системах с ограниченным количеством доступных цветов вам может понадобиться микшировать цветовые величины для увеличения числа доступных цветов, ценой потери разрешения. На финальном этапе вы можете использовать простые логические операции для комбинирования цвета входящего фрагмента с цветом пикселя, уже записанного в цветовой буфер.

10.2.5.1 Цветовое наложение

Цветовое наложение комбинирует **R**, **G**, **B** и **A** величины фрагмента с компонентами пикселя, уже сохраненного в соответствующем месте цветового буфера. Могут применяться различные операции наложения, а результат зависит от альфа входящего фрагмента и альфа уже сохраненного пикселя.

10.2.5.2 Цветовое микширование

В системах с небольшим количеством цветовых битовых плоскостей, вы можете увеличить цветовое разрешение за счет пространственного разрешения, микшируя цвета в изображении. Микширование похоже на полутонирование в газетах. Несмотря на то, что практически в любой газете **2** цвета – черный и белый – в ней могут появляться фотографии, напечатанные оттенками серого цвета, полученного путем чередования белых и черных точек. Сравнение газетного изображения фотографии (без реальных оттенков серого) с оригинальной фотографией (с оттенками серого) свидетельствует об очевидных потерях пространственного разрешения. Похожим образом системы с малым количеством цветовых битовых поверхностей могут смешивать величины красного, зеленого и синего в соседних пикселях для получения более широкого диапазона цветов.

Операция микширования полностью зависит от аппаратуры; все, что позволяет вам сделать **OpenGL–** это активизировать или деактивировать эту операцию. На некоторых системах, однако, это может вообще ни к чему не приводить, что имеет смысл, если цветовых плоскостей и так достаточно. Чтобы активизировать и деактивировать микширование передайте аргумент **GL_DITHER** команд**tglEnable()** или **glDisable()** соответственно. Микширование по умолчанию активизировано.

Микширование работает и в **RGBA**, и в индексном режиме. Цвета или цветовые индексы изменяются каким-либо аппаратно-зависимым методом между двумя соседними значениями. Например, если микширование включено, и индекс, который нужно нарисовать, равен **4.4**, то **60%** пикселей могут быть нарисованы с индексом **4**, а **40% --** с индексом **5. (**Может существовать множество алгоритмов микширования, но результат любого алгоритма должен зависеть только от цвета входящего фрагмента и его координат **(*x*, *y*).)** В **RGBA** режиме микширование производится отдельно для каждого компонента **(**включая альфа**).** Чтобы использовать микширование в индексном режиме, вы, как правило, должны должным образом подготовить цветовую карту **(**чтобы цвета с соседними индексами выглядели близкими**),** в противном случае могут получаться очень странные изображения.

10.2.5.3 Логические операции

Последняя операция, которая применяется к фрагменту **–** это логическая операция, такая как **OR**, **XOR** или **INVERT**, которая применяется к значениям входящего фрагмента **(**источнику**)** и**/**или тем значениям, которые уже хранятся в цветовом буфере **(**приемнике**).** Такие логические операции особенно полезны в машинах, на которых первостепенной графической операцией является копирование прямоугольников данных из одной части окна в другую, из окна в процессорную память или из памяти в окно. Обычно копирование не записывает данные в память непосредственно, а позволяет вам производить логические операции на входящих данных и тех, что уже присутствуют; уже затем копирование замещает старые данные результатом этих операций.

Поскольку этот процесс может быть легко реализован аппаратно, имеется множество подобных систем. В качестве примера использования логической операции, можно рассмотреть использование **XOR** для рисования поверх изображения с восстановлением **–** просто нарисуйте то же изображение с помощью **XOR** второй раз, и исходная картинка восстановлена. В качестве другого примера, рассмотрим использование цветовых индексов в качестве битовых рисунков в индексном режиме. Вы можете составлять изображение в виде комбинации рисунков на разных слоях, использовать ограничение на рисование в определенном наборе слоев и производить логические операции для изменения различных слоев.

Вы включаете и выключаете логические операции с помощью аргументов **GL_INDEX_LOGIC** и **GL_COLOR_LOGIC** команд **glEnable()** и **glDisable()** для индексного и **RGBA** режимов соответственно. Вы также должны выбрать одну из поддерживаемых логических операций командой **glLogicOp()**. В противном случае вы получите результат по умолчанию **– GL_COPY. (**Для обратной совместимости с **OpenGL** версии **1.0**, **glEnable(GL_LOGIC_OP)** также включает логические операции в индексном режиме**.)**

```
void glLogicOp (GLenum opcode);
```

Выбирает производимую логическую операцию, выполняемую над входящим фрагментом **(**источником**)** и пикселем, находящимся в цветовом буфере **(**приемником**).** Все возможные значения для ***opcode*** и их значения перечислены в таблице **10-4.** ***s***означает источник **(source)**, а ***d*****–** приемник **(destination).** Значение по умолчанию **– GL_COPY.**

Таблица **10-4.** Логические операции

Параметр	Операция	Параметр	Операция
GL_CLEAR	0	GL_AND	$s \wedge d$
GL_COPY	*s*	GL_OR	$s \vee d$
GL_NOOP	*d*	GL_NAND	$\neg (s \wedge d)$
GL_SET	1	GL_NOR	$\neg (s \vee d)$
GL_COPY_INVERTED	$\neg s$	GL_XOR	$s \mathrm{XOR} d$
GL_INVERT	$\neg d$	GL_EQUIV	$\neg (s \mathrm{XOR} d)$
GL_AND_REVERSE	$s \wedge \neg d$	GL_AND_INVERTED	$\neg s \wedge d$
GL_OR_REVERSE	$s \vee \neg d$	GL_OR_INVERTED	$\neg s \vee d$

10.3 Аккумуляторный буфер

Дополнительно: Аккумуляторный буфер может использоваться для таких техник, как сглаживание всей сцены, размытое движение, симуляции фотографической глубины поля и вычисление мягких теней, получающихся от нескольких источников света. Можно применять и другие с техники, особенно если использовать еще и другие буферы.

Графические операции **OpenGL** не пишут в буфер аккумулятора непосредственно. Обычно в одном из цветовых буферов генерируется серия изображений, и эти изображения аккумулируются в буфере аккумулятора по одному за раз. Когда аккумуляция завершена, результат копируется обратно в цветовой буфер для просмотра. Во избежание ошибок округления буфер аккумуляции может иметь большее цветовое разрешение **(**больше бит на цвет**)**, чем цветовые буферы. Визуализация сцены несколько раз, очевидно, занимает больше времени, но результат имеет более высокое качество. Следовательно, вы должны решать, что важнее для вашего приложения – качество или скорость.

Вы можете использовать буфер аккумуляции так же, как фотограф использует один кадр для нескольких снимков одной и той же сцены. Если какой-либо объект сцены перемещается – он будет выглядеть размытым. Не является сюрпризом тот факт, что компьютер может делать с изображением намного больше трюков, чем фотограф с камерой. Например, компьютер может управлять точкой наблюдения в отличие от фотографа, который не может встряхнуть камеру предсказуемым образом.

```
void glAccum (GLenum op, GLfloat value);
```

Управляет буфером аккумуляции. Аргумент ***op***выбирает операцию, а ***value***– число, которое будет использоваться в этой операции. Возможными операциями являются **GL_ACCUM**, **GL_LOAD**, **GL_RETURN**, **GL_ADD** и **GL_MULT**.

GL_ACCUM считывает каждый пиксель из буфера выбранного для чтения командой **glReadBuffer()**, умножает **R**, **G**, **B** и **A** величины на число ***value*** и добавляет результаты в буфер аккумуляции.

GL_LOAD работает так же как **GL_ACCUM** за тем исключением, что результирующие значения заменяют уже хранимые в буфере, а не добавляются к ним.

GL_RETURN забирает величины из буфера аккумуляции, умножает их на ***value***и помещает результат в цветовой буфер, выбранный для записи.

GL_ADD и **GL_MULT** просто добавляют или умножают значения каждого пикселя в буфере аккумуляции к или на величину ***value***. Для **GL_MULT *value*** усекается до диапазона **[-1.0, 1.0]**. Для **GL_ADD** усечение не производится.

10.3.1 Сглаживание всей сцены

Чтобы произвести сглаживание сцены, сначала очистите аккумулятор и активизируйте передний цветовой буфер для записи и чтения. Затем несколько раз (скажем ***n***) выполните код, который сдвигает изображение (***jittering***– это перемещение изображения в слегка иную позицию) и рисует его, аккумулируя данные командой

```
glAccum(GL_ACCUM,1.0/n);
```

а затем в конце вызовите

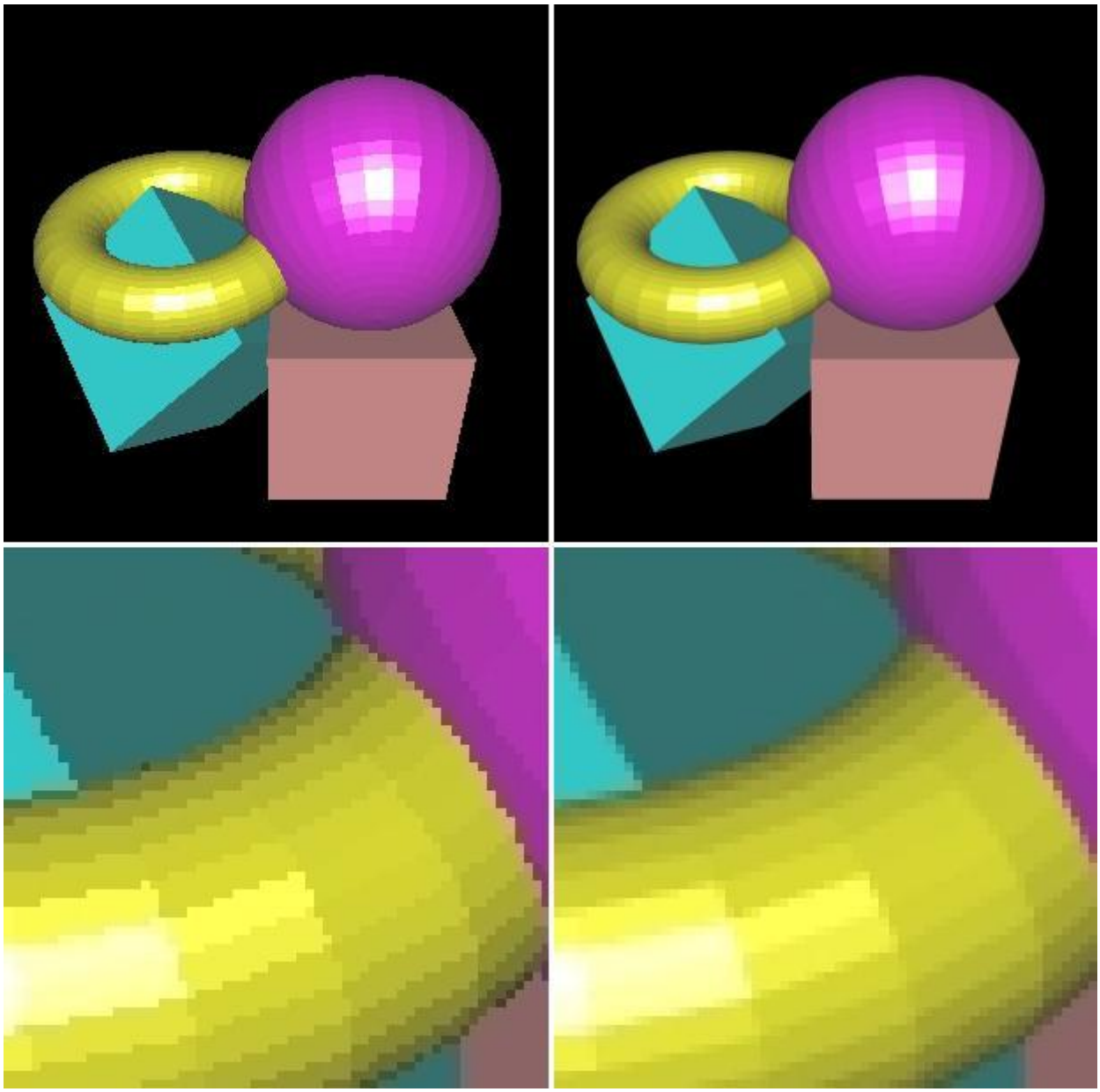
```
glAccum(GL_RETURN,1.0);
```

Заметьте, что этот метод работает несколько быстрее, если при первом проходе вместо **GL_ACCUM** применить **GL_LOAD**, а не делать предварительную очистку аккумулятора. С помощью этого кода изображение нужно нарисовать **n**раз, а затем нарисовать результирующее изображение. Если вы не хотите, чтобы наблюдатель видел промежуточные изображения, считывайте из невидимого цветового буфера, а результат записывайте в видимый (или в задний буфер, впоследствии переключив его с передним).

Вы также можете создать пользовательский интерфейс, который показывает, как рассматриваемое изображение улучшается с каждым шагом и позволяет пользователю остановить процесс, когда ему кажется, что изображение достигло нужного качества. Чтобы сделать это, в цикле, рисующем последовательные изображения, вызывайте **glAccum()** с первым аргументом **GL_RETURN**, а вторым равным **16.0/1.0**, **16.0/2.0**, **16.0/3.0**, ... В данной технике после одного прохода показывается **1/16** часть финального изображения, после двух проходов – **2/16** и так далее. После **GL_RETURN** программа должна проверять, не остановил ли пользователь процесс. Такой метод будет работать несколько медленнее, поскольку после каждого шага изображение должно быть скопировано в цветовой буфер.

Чтобы решить, каким должно быть ***n***, вы должны сделать выбор между скоростью (чем больше проходов вы сделаете, тем больше времени понадобится для получения финального изображения) и качеством (чем больше проходов вы сделаете, тем выше будет качество финального изображения). На рисунке **10-2** слева показано исходное изображение, справа то же изображение после **16** аккумулирующих итераций, а в нижней части увеличенные фрагменты первого и второго изображений соответственно.

Рисунок **10-2.** Сглаживание сцены с помощью буфера аккумулятора



Пример **10-2.** Функции для сдвига объема видимости

```
#include <math.h>

#define PI_20 3.14159265358979323846

void accFrustum(GLdouble left,GLdouble right,
                GLdouble bottom,GLdouble top,
                GLdouble zNear,GLdouble zFar,
                GLdouble pixdx,GLdouble pixdy,
                GLdouble eyedx, GLdouble eyedy,
                GLdouble focus)
{
      GLdouble xwsize, ywsize;
      GLdouble dx, dy;
      GLint viewport[4];

      glGetIntegerv(GL_VIEWPORT,viewport);

      xwsize=right-left;
      ywsize=top-bottom;
```

```
      dx=-(pixdx*xwsize/(GLdouble)viewport[2] + eyedx*zNear/focus);
      dy=-(pixdy*ywsize/(GLdouble)viewport[3] + eyedy*zNear/focus);

      glMatrixMode(GL_PROJECTION);
      glLoadIdentity();
      glFrustum(left+dx,right+dx,bottom+dy,top+dy,zNear,zFar);
      glMatrixMode(GL_MODELVIEW);
      glTranslatef(-eyedx,-eyedy,0.0);
}

void accPerspective(GLdouble fovy,GLdouble aspect,
                    GLdouble zNear,GLdouble zFar,
                    GLdouble pixdx,GLdouble pixdy,
                    GLdouble eyedx,GLdouble eyedy,
                    GLdouble focus)
{
      GLdouble fov2,left,right,bottom,top;
      fov2=((fovy*PI_20)/180.0)/2.0;

      top=zNear/(cos(fov2)/sin(fov2));
      bottom=-top;
      right=top*aspect;
      left=-right;

      accFrustum(left,right,bottom,top,zNear,zFar,
                 pixdx,pixdy,eyedx,eyedy,focus);
}
```

Пример **10-3** использует две функции из примера **10-2** для выполнения сглаживания сцены**.**

Пример **10-3.** Сглаживание сцены**:** файл **accpersp.cpp**

```
#include <glut.h>
#include "jitter.h"

#define ACSIZE 16
#define SPHERE_FACES 32
#define JIT j16

void init()
{
      GLfloat mat_ambient[]={1.0,1.0,1.0,1.0};
      GLfloat mat_specular[]={1.0,1.0,1.0,1.0};
      GLfloat light_position[]={0.0,0.0,10.0,1.0};
      GLfloat lm_ambient[]={0.2,0.2,0.2,1.0};

      glMaterialfv(GL_FRONT,GL_AMBIENT,mat_ambient);
      glMaterialfv(GL_FRONT,GL_SPECULAR,mat_specular);
      glMaterialf(GL_FRONT,GL_SHININESS,50.0);
      glLightfv(GL_LIGHT0,GL_POSITION,light_position);
      glLightModelfv(GL_LIGHT_MODEL_AMBIENT,lm_ambient);

      glEnable(GL_LIGHTING);
      glEnable(GL_LIGHT0);
      glEnable(GL_DEPTH_TEST);
      glShadeModel(GL_FLAT);

      glClearColor(0.0,0.0,0.0,0.0);
      glClearAccum(0.0,0.0,0.0,0.0);
}
```

```
void displayObjects()
{
      GLfloat torus_diffuse[]={0.7,0.7,0.0,1.0};
      GLfloat cube_diffuse[]={0.0,0.7,0.7,1.0};
      GLfloat sphere_diffuse[]={0.7,0.0,0.7,1.0};
      GLfloat octa_diffuse[]={0.7,0.4,0.4,1.0};

      glPushMatrix();
      glTranslatef(0.0,0.0,-5.0);
      glRotatef(30.0,1.0,0.0,0.0);

      glPushMatrix();
      glTranslatef(-0.80,0.35,0.0);
      glRotatef(100.0,1.0,0.0,0.0);
      glMaterialfv(GL_FRONT,GL_DIFFUSE,torus_diffuse);
      glutSolidTorus(0.275,0.85,SPHERE_FACES,SPHERE_FACES);
      glPopMatrix();

      glPushMatrix();
      glTranslatef(-0.75,-0.50,0.0);
      glRotatef(45.0,0.0,0.0,1.0);
      glRotatef(45.0,1.0,0.0,0.0);
      glMaterialfv(GL_FRONT,GL_DIFFUSE,cube_diffuse);
      glutSolidCube(1.5);
      glPopMatrix();

      glPushMatrix();
      glTranslatef(0.75,0.60,0.0);
      glRotatef(30.0,1.0,0.0,0.0);
      glMaterialfv(GL_FRONT,GL_DIFFUSE,sphere_diffuse);
      glutSolidSphere(1.0,SPHERE_FACES,SPHERE_FACES);
      glPopMatrix();

      glPushMatrix();
      glTranslatef(0.70,-0.90,0.25);
      glMaterialfv(GL_FRONT,GL_DIFFUSE,octa_diffuse);
      glutSolidCube(1.5);
      glPopMatrix();

      glPopMatrix();
}

void display()
{
      GLint viewport[4];
      int jitter;

      glGetIntegerv(GL_VIEWPORT,viewport);

      glClear(GL_ACCUM_BUFFER_BIT);
      for (jitter=0;jitter<ACSIZE;jitter++)
      {
            glClear(GL_COLOR_BUFFER_BIT|GL_DEPTH_BUFFER_BIT);

accPerspective(50.0,(GLdouble)viewport[2]/(GLdouble)viewport[3],
                           1.0,15.0,JIT[jitter].x,JIT[jitter].y,
                           0.0,0.0,1.0);
            displayObjects();
            glAccum(GL_ACCUM,1.0/ACSIZE);
      }
      glAccum(GL_RETURN,1.0);
      glFlush();
}
```

```
void reshape(int w,int h)
{
      glViewport(0,0,(GLsizei)w,(GLsizei)h);
}

int main (int argc, char **argv)
{
      glutInit(&argc,argv);

glutInitDisplayMode(GLUT_SINGLE|GLUT_RGB|GLUT_DEPTH|GLUT_ACCUM);
      glutInitWindowSize(310,310);
      glutInitWindowPosition(100,100);
      glutCreateWindow("Scene Antialiasing");
      init();
      glutDisplayFunc(display);
      glutReshapeFunc(reshape);
      glutMainLoop();
      return 0;
}
```

Для того, чтобы производить сглаживание сцены, вы не обязательно должны использовать перспективную проекцию. Если вы работаете с ортографической проекцией, для сдвига сцены вам достаточно использовать **glTranslate*()**. Имейте в виду, что **glTranslate*()** работает в мировых координатах, а вам нужно сдвигать сцену менее чем на **1** пиксель в оконных координатах. Таким образом, вам нужно обратить процесс преобразования мировых координат, вычислив величины сдвигающего переноса с использованием его ширины и высоты в мировых координатах, деленных на размер порта просмотра. Затем умножьте получившуюся величину в мировых координатах на количество сдвига, вычислив, таким образом, на сколько сцена должна быть перемещена в мировых координатах, чтобы получить предсказуемый сдвиг менее чем в **1** пиксель. Пример **10-4** показывает, как должны выглядеть функции **display()** и **reshape()**, если ширина и высота в мировых координатах равны **4.5**.

Пример **10-4.** Сдвиг в ортографической проекции: файл **accanti.cpp**

```
void display()
{
      GLint viewport[4];
      int jitter;

      glGetIntegerv(GL_VIEWPORT,viewport);

      glClear(GL_ACCUM_BUFFER_BIT);
      for (jitter=0;jitter<ACSIZE;jitter++)
      {
            glClear(GL_COLOR_BUFFER_BIT|GL_DEPTH_BUFFER_BIT);
            glPushMatrix();

            //Обратите внимание, что 4.5 - это дистанция в мировых
координатах
            //между левой и правой, а также нижней и верхней
границами.
            //Формула преобразует сдвиг на часть пикселя в мировые
координаты.
            glTranslatef(JIT[jitter].x*4.5/viewport[2],
                         JIT[jitter].y*4.5/viewport[3],0.0);
            displayObjects();
            glPopMatrix();
            glAccum(GL_ACCUM,1.0/ACSIZE);
      }
```

```
        glAccum(GL_RETURN,1.0);
        glFlush();
}

void reshape(int w,int h)
{
        glViewport(0,0,(GLsizei)w,(GLsizei)h);
        glMatrixMode(GL_PROJECTION);
        if(w<=h)
                glOrtho(-2.25,2.25,-2.25*h/w,2.25*h/w,-10.0,10.0);
        else
                glOrtho(-2.25*w/h,2.25*w/h,-2.25,2.25,-10.0,10.0);
        glMatrixMode(GL_MODELVIEW);
        glLoadIdentity();
}
```

10.3.2 Размытое движение

Похожие методы могут быть использованы для симуляции размытого движения, как показано на рисунке **10-3.** предположим, что в вашей сцене есть несколько стационарных и несколько движущихся объектов, и вы хотите показать движение размытием. Настройте буфер аккумуляции таким же образом, как и раньше, но вместо пространственного сдвига изображений, сдвигайте их по времени. Вся сцена может быть сделана более тусклой, если в процессе рисования в буфер аккумуляции вызывать

```
glAccum(GL_MULT, decayFactor);
```

где ***decayFactor*** – это число от **0.0** до **1.0.** Меньшие значения для ***decayFactor*** как бы заставляют объект двигаться быстрее. Вы можете перенести результирующее изображение с текущей позицией движущегося объекта и хвостом от его предыдущих позиций из аккумулятора в цветовой буфер, вызвав

```
glAccum(GL_RETURN,1.0);
```

Изображение выглядит правильно даже если объекты движутся с разными скоростями или если часть из них движется с ускорением. как и раньше, чем больше точек сдвига **(**в данном случае временных**)** вы используете, тем качественнее будет финальное изображение. Это верно как минимум до той точки, когда вы начнете терять разрешение, поскольку буфер аккумуляции имеет конечную точность. Вы можете комбинировать размытое движение и сглаживание, производя сдвиг и в пространстве и во времени, однако за более высокое качество изображения вы заплатите низкой скоростью работы программы.

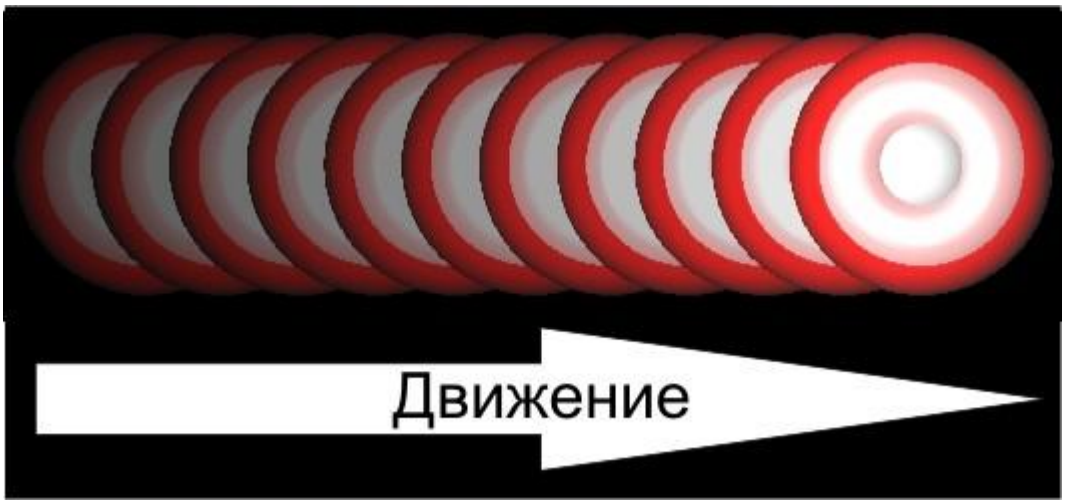
Описанная техника отображения размытого изображения с помощью буфера аккумуляции, конечно, не является единственной. Альтернативно, вы можете после отрисовки каждого кадра сохранять получившееся изображение в буфере аккумулятора **(**с ослаблением**)** и затем сразу же восстанавливать его **(**уже без ослабления**).** Это можно сделать, например, с помощью следующих команд:

```
glAccum(GL_LOAD, 0.9);
glAccum(GL_RETURN, 1.0);
```

После этого нужно переместить модель, нарисовать ее поверх уже имеющегося изображения, загрузить его в аккумулятор, восстановить и так далее. Этот метод применяется в примере **10-5**, который использовался для получения рисунка **10-3.**

Имейте в виду, что вы в любом случае должны согласовывать коэффициент итеративного ослабления и число шагов сдвига. Например, если коэффициент мал, а число шагов велико, то к последней итерации след от первой итерации может быть невидим на экране (поскольку будет слишком ослаблен). Если же коэффициент велик (близок к **1.0**), а число шагов небольшое – разница между изображением первой и последней итераций может быть настолько мала, что невозможно будет понять, где начало движения, а где его конец.

Рисунок **10-3.** Размытое движение



Пример **10-5.** Размытие движения с помощью буфера аккумулятора: файл **motiblur.cpp**

```
#include <glut.h>

#define SPHERE_FACES 64

//Горизонтальный размер проекции
#define PROJ_SIZE 8

//Количество шагов сдвига/поворота
#define TIME_JITTER_STEPS 10

//Угол на который за время сдвигов
//должна повернуться модель
#define FULL_ROT 0

//Фактор сжатия модели
#define SCALE_FACTOR 1.0

void init()
{
      GLfloat mat_ambient[]={0.8,0.8,0.8,1.0};
      GLfloat mat_specular[]={1.0,1.0,1.0,1.0};
      GLfloat light_position[]={0.0,0.0,10.0,1.0};
      GLfloat lm_ambient[]={0.2,0.2,0.2,1.0};

      glMaterialfv(GL_FRONT,GL_AMBIENT,mat_ambient);
      glMaterialfv(GL_FRONT,GL_SPECULAR,mat_specular);
      glMaterialf(GL_FRONT,GL_SHININESS,50.0);
      glLightfv(GL_LIGHT0,GL_POSITION,light_position);
      glLightModelfv(GL_LIGHT_MODEL_AMBIENT,lm_ambient);

      glEnable(GL_LIGHTING);
      glEnable(GL_LIGHT0);
```

```
	glEnable(GL_RESCALE_NORMAL);
	glEnable(GL_DEPTH_TEST);
	glShadeModel(GL_SMOOTH);

	glClearColor(0.0,0.0,0.0,0.0);
	glClearAccum(0.0,0.0,0.0,0.0);
}

void displayObjects(GLfloat xoffset,GLfloat yrot)
{
	GLfloat torus1_diffuse[]={1.0,0.0,0.0,1.0};
	GLfloat torus2_diffuse[]={1.0,0.7,0.7,1.0};
	GLfloat sphere_diffuse[]={1.0,1.0,1.0,1.0};

	glLoadIdentity();
	glTranslatef(xoffset,0.0,0.0);
	glRotatef(yrot,0.0,1.0,0.0);
	glScalef(SCALE_FACTOR,SCALE_FACTOR,SCALE_FACTOR);
	glMaterialfv(GL_FRONT,GL_DIFFUSE,torus1_diffuse);
	glutSolidTorus(0.3,0.55,SPHERE_FACES,SPHERE_FACES);
	glMaterialfv(GL_FRONT,GL_DIFFUSE,torus2_diffuse);
	glutSolidTorus(0.35,0.45,SPHERE_FACES,SPHERE_FACES);
	glMaterialfv(GL_FRONT,GL_DIFFUSE,sphere_diffuse);
	glutSolidSphere(0.4,SPHERE_FACES,SPHERE_FACES);
}

void display()
{
	GLfloat correctedProjSize=PROJ_SIZE-3;
	int step;
	GLfloat oneStepTrans=correctedProjSize/TIME_JITTER_STEPS;
	GLfloat oneStepRot=FULL_ROT/TIME_JITTER_STEPS;

	glClear(GL_COLOR_BUFFER_BIT|GL_ACCUM_BUFFER_BIT);
	for(step=0;step<=TIME_JITTER_STEPS;step++)
	{
		glClear(GL_DEPTH_BUFFER_BIT);
		displayObjects(-correctedProjSize/2+oneStepTrans*step,
				oneStepRot*step);
		if(step!=TIME_JITTER_STEPS)
			glAccum(GL_LOAD,0.9);
		else
			glAccum(GL_LOAD,1.0);
		glAccum(GL_RETURN,1.0);
	}
	glAccum(GL_RETURN,1.0);
	glFlush();
}

void reshape(int w,int h)
{
	glViewport(0,0,(GLsizei)w,(GLsizei)h);
	glMatrixMode(GL_PROJECTION);
	if(w<=h)
		glOrtho(-PROJ_SIZE/2,PROJ_SIZE/2,
			-(PROJ_SIZE/2)*h/w,(PROJ_SIZE/2)*h/w,-10.0,10.0);
	else
		glOrtho(-PROJ_SIZE/2*w/h,(PROJ_SIZE/2)*w/h,
			-(PROJ_SIZE/2),PROJ_SIZE/2,-10.0,10.0);
	glMatrixMode(GL_MODELVIEW);
	glLoadIdentity();
}
```

```
int main (int argc, char **argv)
{
      glutInit(&argc,argv);

glutInitDisplayMode(GLUT_SINGLE|GLUT_RGB|GLUT_DEPTH|GLUT_ACCUM);
      glutInitWindowSize(620,620);
      glutInitWindowPosition(100,100);
      glutCreateWindow("Motion blur with Accumulator");
      init();
      glutDisplayFunc(display);
      glutReshapeFunc(reshape);
      glutMainLoop();
      return 0;
}
```

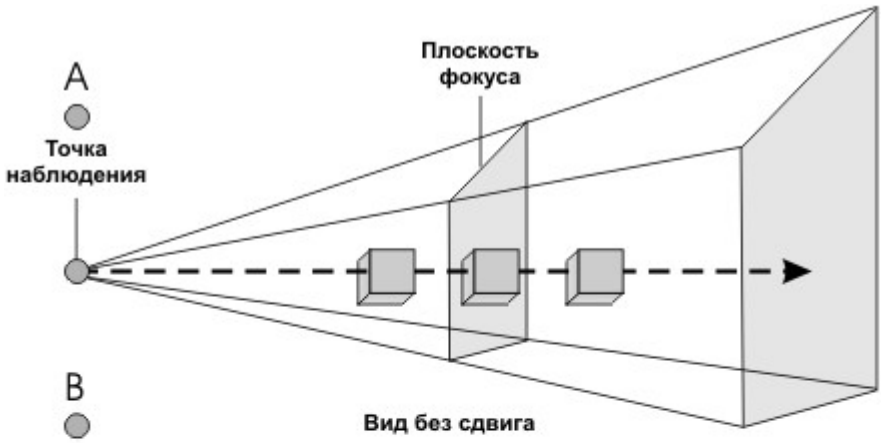
10.3.3 Глубина поля

Фотографии, сделанные с помощью камеры, имеют превосходный фокус только для объектов, лежащих в одной плоскости параллельной фотопленке на определенном расстоянии. Чем дальше объект от этой плоскости, тем более он размыт. Глубина поля для камеры – это область вокруг плоскости фокуса, где объекты сравнительно слабо выпадают из фокуса.

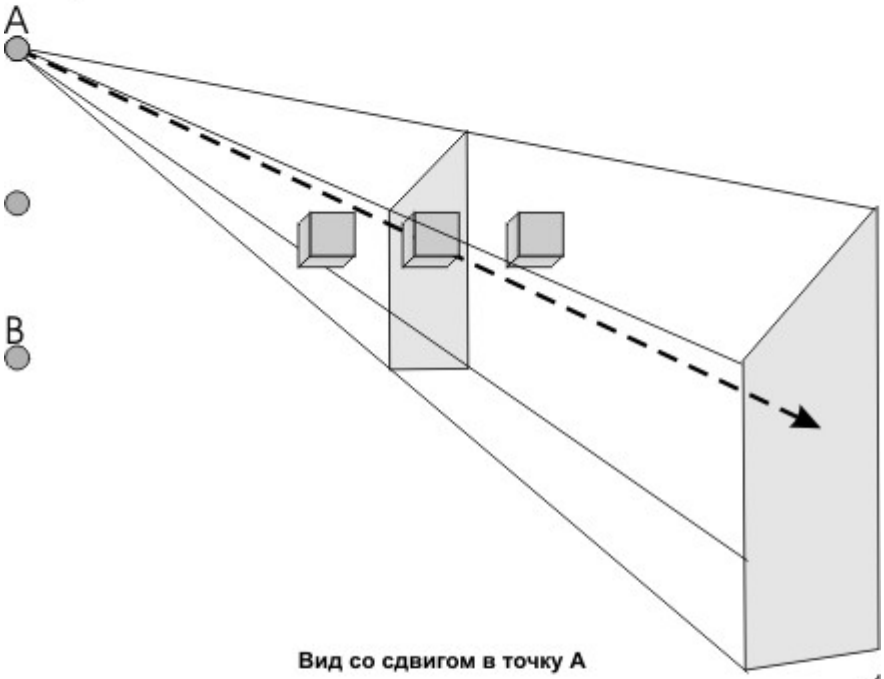
При обычных условиях, все, что вы рисуете с помощью **OpenGL**, находится в фокусе (если только у вас хороший монитор – в противном случае абсолютно все выглядит размытым). Буфер аккумуляции может быть использован для приближения к виду реальной фотографии, где объекты выглядят все более и более размытыми по мере удаления от плоскости фокуса. Это (как и во многих других случаях) не точная имитация того, что происходит внутри камеры, но результат похож на тот, который получается при помощи камеры.

Чтобы достигнуть этого результата, нарисуйте сцену несколько раз, вызывая **glFrustum()** с разными аргументами. Выбирайте аргументы таким образом, чтобы точка наблюдения перемещалась (на незначительное расстояние) вокруг своей оригинальной позиции, и, чтобы прямоугольник, лежащий в плоскости фокуса, был одинаковым для всех проекций (как показано на рисунке **10-4**). Результаты всех визуализаций должны быть усреднены обычным путем с помощью буфера аккумуляции.

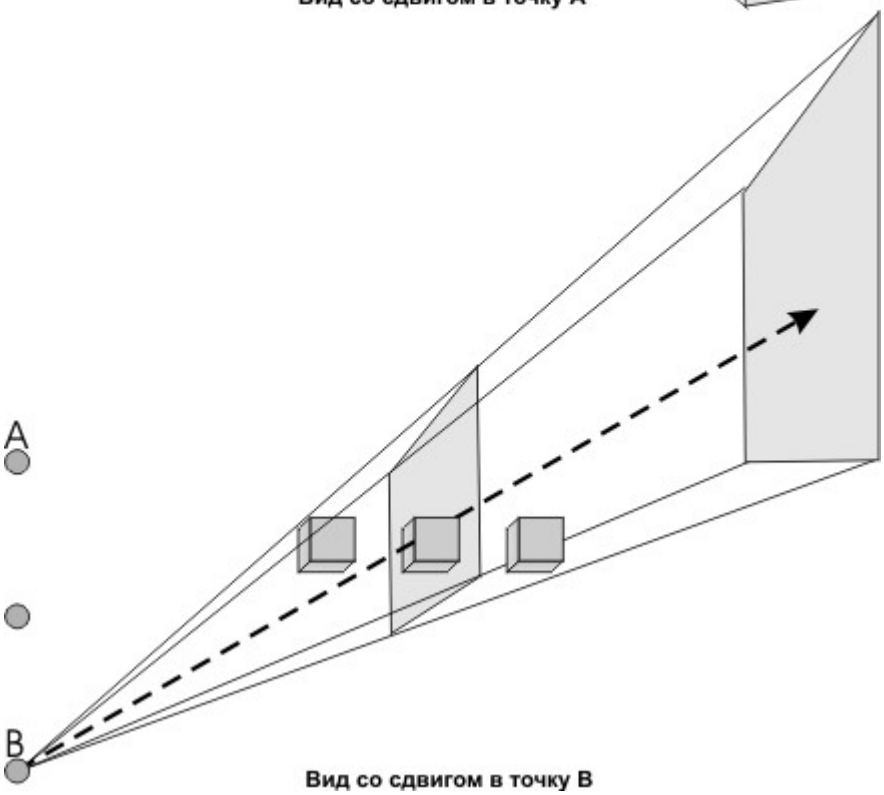
Рисунок **10-4.** Перемещение объема видимости для эффекта глубины поля



Вид без сдвига



Вид со сдвигом в точку A



Вид со сдвигом в точку B

На рисунке **10-5** изображено **5** чайников**,** нарисованных с применением эффекта глубины поля**.** Золотой чайник **(**второй слева**)** находится в фокусе**,** а остальные размываются в разной степени в зависимости от дистанции до плоскости фокуса **(**золотого чайника**).** Код**,** используемый для рисования этого изображения**,** приводится в примере **10-6.** Сцена рисуется **8** раз**,** каждый раз со слегка сдвинутым объемом видимости **(**сдвиг осуществляется с помощью функции **accPerspective()).** Как вы можете помнить**,** из метода сглаживании сцены**, 5-**ый и **6-**ой аргументы сдвигают объем видимости в направлениях ***x***и ***y*** соответственно**.** Однако для эффекта глубины поля нам нужно сдвигать объем видимости таким образом**,** чтобы он оставался на плоскости фокуса**.** Плоскость фокуса **–** это величина глубины**,** определяемая **9-**ым **(**последним**)** аргументом **accPerspective() (**например ***z*****=5.0).** Количество размытия определяется умножением величин сдвига по ***x***и ***y*****(7-**го и **8-**го аргументов**)** на константу**.** Определение такой константы не представляет сложности **–** экспериментируйте с ней до тех пор**,** пока изображение не будет иметь тот вид**,** который вам нужен**. (**Обратите внимание на то**,** что в примере **10-6 4-**ый и **5-**ый аргументы **accPerspective()** установлены в **0.0.** Таким образом**,** сглаживание сцены не производится**.)**

Рисунок **10-5.** Чайники с эффектом глубины поля

Пример **10-6.** Эффект глубины поля**:** файл **dof.cpp**

```
#include <glut.h>
#include "jitter.h"

#define ACSIZE 8
#define JIT j8

GLint teapotList;

void init()
{
      GLfloat ambient[]={0.0,0.0,0.0,1.0};
      GLfloat diffuse[]={1.0,1.0,1.0,1.0};
      GLfloat specular[]={1.0,1.0,1.0,1.0};
      GLfloat position[]={0.0,3.0,3.0,0.0};

      GLfloat lmodel_ambient[]={0.2,0.2,0.2,1.0};
      GLfloat local_view[]={0.0};

      glLightfv(GL_LIGHT0,GL_AMBIENT,ambient);
      glLightfv(GL_LIGHT0,GL_DIFFUSE,diffuse);
      glLightfv(GL_LIGHT0,GL_POSITION,position);

      glLightModelfv(GL_LIGHT_MODEL_AMBIENT,lmodel_ambient);
      glLightModelfv(GL_LIGHT_MODEL_LOCAL_VIEWER,local_view);

      glFrontFace(GL_CW);
      glEnable(GL_LIGHTING);
      glEnable(GL_LIGHT0);
```

```
	glEnable(GL_AUTO_NORMAL);
	glEnable(GL_NORMALIZE);
	glEnable(GL_DEPTH_TEST);

	glClearColor(0.0,0.0,0.0,0.0);
	glClearAccum(0.0,0.0,0.0,0.0);

	teapotList=glGenLists(1);
	glNewList(teapotList,GL_COMPILE);
	glutSolidTeapot(0.5);
	glEndList();
}

void renderTeapot(GLfloat x,GLfloat y,GLfloat z,
		  GLfloat ambr,GLfloat ambg,GLfloat ambb,
		  GLfloat difr,GLfloat difg,GLfloat difb,
		  GLfloat specr,GLfloat specg,GLfloat specb,
		  GLfloat shine)
{
	GLfloat mat[4];

	glPushMatrix();
	glTranslatef(x,y,z);
	mat[0]=ambr;mat[1]=ambg;mat[2]=ambb;mat[3]=1.0;
	glMaterialfv(GL_FRONT,GL_AMBIENT,mat);
	mat[0]=difr;mat[1]=difg;mat[2]=difb;
	glMaterialfv(GL_FRONT,GL_DIFFUSE,mat);
	mat[0]=specr;mat[1]=specg;mat[2]=specb;
	glMaterialfv(GL_FRONT,GL_SPECULAR,mat);
	glMaterialf(GL_FRONT,GL_SHININESS,shine);
	glCallList(teapotList);
	glPopMatrix();
}

void display()
{
	GLint viewport[4];
	int jitter;

	glGetIntegerv(GL_VIEWPORT,viewport);
	glClear(GL_ACCUM_BUFFER_BIT);

	for (jitter=0;jitter<ACSIZE;jitter++)
	{
		glClear(GL_COLOR_BUFFER_BIT|GL_DEPTH_BUFFER_BIT);
		glPushMatrix();

accPerspective(45.0,(GLdouble)viewport[2]/(GLdouble)viewport[3],
			1.0,15.0,0.0,0.0,

0.33*JIT[jitter].x,0.33*JIT[jitter].y,5.0);

		//Рубиновый, золотой, серебряный, изумрудный и голубой
чайники
		renderTeapot(-1.1,-0.5,-4.5,
			     0.1745,0.01175,0.01175,
			     0.61424,0.04136,0.04136,
			     0.727811,0.626959,0.626959,
			     0.6);
		renderTeapot(-0.5,-0.5,-5.0,
			     0.24725,0.1995,0.0745,
			     0.75164,0.60648,0.22648,
			     0.628281,0.555802,0.366065,
```

```
                        0.4);
            renderTeapot(0.2,-0.5,-5.5,
                        0.19225,0.19225,0.19225,
                        0.50754,0.50754,0.50754,
                        0.508273,0.508273,0.508273,
                        0.4);
            renderTeapot(1.0,-0.5,-6.0,
                        0.0215,0.1745,0.0215,
                        0.07568,0.61424,0.07568,
                        0.633,0.727811,0.633,
                        0.6);
            renderTeapot(1.8,-0.5,-6.5,
                        0.0,1.0,0.06,
                        0.0,0.50980392,0.50980392,
                        0.50196078,0.50196078,0.50196078,
                        0.25);
            glAccum(GL_ACCUM,0.125);
            glPopMatrix();
      }
      glAccum(GL_RETURN,1.0);
      glFlush();
}

void reshape(int w,int h)
{
      glViewport(0,0,(GLsizei)w,(GLsizei)h);
}

int main(int argc, char **argv)
{
      glutInit(&argc,argv);

glutInitDisplayMode(GLUT_SINGLE|GLUT_RGB|GLUT_DEPTH|GLUT_ACCUM);
      glutInitWindowSize(620,620);
      glutInitWindowPosition(100,100);
      glutCreateWindow("Depth-of-Field Effect");
      init();
      glutDisplayFunc(display);
      glutReshapeFunc(reshape);
      glutMainLoop();
      return 0;
}
```

10.3.4 Мягкие тени

Чтобы аккумулировать мягкие тени, получающиеся от нескольких источников света, визуализируйте их с одним включенным источником за раз и аккумулируйте вместе. Этот процесс можно комбинировать с пространственным сдвигом для одновременного сглаживания сцены.

10.3.5 Сдвиг

Если вам нужно снять **9** или **16** кадров для сглаживания изображения, вы возможно решите, что лучшие решение – это взять точки, распределенные по пикселю с равными промежутками. Удивительно, но это не обязательно так. По правде говоря, иногда является хорошей идеей взять точки на соседних пикселях. Иногда вам потребуется равномерное распределение точек, а иногда – нормальное, уплотняющееся вблизи центра пикселя. Кроме того, в таблице **10-5** приводится несколько наборов разумных величин сдвига, которые могут использоваться в некоторых случаях. Все величины в этой таблице лежат внутри пикселя, а большинство из них еще и равномерно распределены.

Таблица **10-5.** Примеры величин сдвига

Число	Величины
2	[0.25,0.75],[0.75,0.25]
3	[0.5033922635,0.8317967229],[0.7806016275,0.2504380877], [0.2261828938,0.4131553612]
4	[0.375,0.25],[0.125,0.75],[0.875,0.25],[0.625,0.75]
5	[0.5,0.5],[0.3,0.1],[0.7,0.9],[0.9,0.3],[0.1,0.7]
6	[0.4646464646,0.4646464646],[0.1313131313,0.7979797979], [0.5353535353,0.8686868686], [0.8686868686,0.5353535353], [0.7979797979,0.1313131313],[0.2020202020,0.2020202020]
8	[0.5625,0.4375],[0.0625,0.9375],[0.3125,0.6875],[0.6875,0.8125], [0.8125,0.1875], [0.9375,0.5625],[0.4275,0.0625],[0.1875,0.3125]
9	[0.5,0.5],[0.1666666666,0.9444444444], [0.5,0.1666666666],[0.5,0.8333333333], [0.1666666666,0.2777777777],[0.8333333333,0.3888888888], [0.1666666666,0.6111111111], [0.8333333333,0.7222222222], [0.8333333333,0.0555555555]
12	[0.4166666666,0.625],[0.9166666666,0.875],[0.25,0.375], [0.4166666666,0.125],[0.75,0.125], [0.0833333333,0.125], [0.75,0.625],[0.25,0.875],[0.5833333333,0.375], [0.9166666666,0.375], [0.0833333333,0.625],[0.5833333333,0.875]
16	[0.375,0.4375],[0.625,0.0625],[0.875,0.1875],[0.125,0.0625], [0.375,0.6875],[0.875,0.4375], [0.625,0.5625],[0.375,0.9375], [0.625,0.3125],[0.125,0.5625],[0.125,0.8125],[0.375,0.1875], [0.875,0.9375],[0.875,0.6875],[0.125,0.3125],[0.625,0.8125]

Глава 11. Тесселяция и квадрические поверхности

Библиотека **OpenGL (GL)** разработана для низкоуровневых операций, одновременно являющихся конвейерными и имеющими доступ к аппаратному ускорению. Библиотека утилит **OpenGL (GLU)** дополняет **OpenGL**, предоставляя высокоуровневые операции. Некоторые из операций **GLU (**такие как мипмаппинг функцией **gluBuild*DMipmaps()**, масштабирование изображений функцией **gluScaleImage()**, операции матричных преобразований функциями **gluOrtho2D()**, **gluPerspective()**, **gluLookAt()**, **gluProject()**, **gluUnProject()** и **gluUnProject4())** рассматриваются в других главах. Некоторые будут рассмотрены далее.

Для оптимизации быстродействия, ядро **OpenGL** визуализирует только выпуклые полигоны, однако **GLU** содержит функции для тесселяции **(**разбиения**)** вогнутых полигонов на выпуклые, которые могут обрабатываться **OpenGL.** Там, где **OpenGL** оперирует простыми примитивами, такими как точки, линии и закрашенные полигоны, **GLU** может создавать объекты более высокого уровня, такие, как поверхности сфер, цилиндры или конусы.

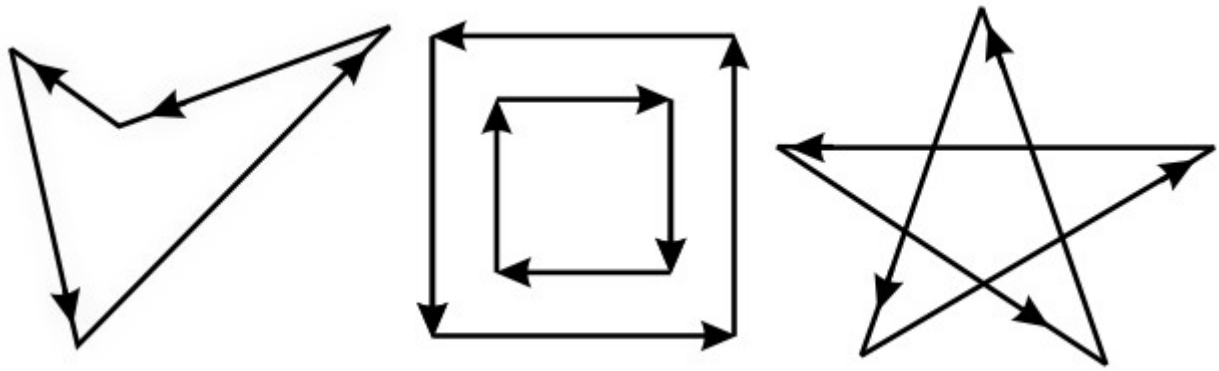
11.1 Тесселяция полигонов

OpenGL может непосредственно отображать только простые выпуклые полигоны. Полигон является простым, если его ребра пересекаются только в вершинах, если в каждой точке находится только одна вершина, и в каждой вершине соединяются только два ребра. Если вашему приложению требуется отображение вогнутых полигонов, полигонов с дырами или полигонов с пересекающимися ребрами, такие полигоны должны быть предварительно разделены на простые выпуклые полигоны до того, как

их можно будет отобразить. Такое разбиение называется *тесселяцией*, и **GLU** предоставляет ряд функций для осуществления тесселяции. Эти функции принимают информацию о контурах, описывающих трудный для отображения полигон, в качестве аргумента и возвращают комбинацию треугольников, фигур из треугольников, треугольных вееров или линий.

На рисунке **11-1** показаны контуры нескольких полигонов, нуждающихся в тесселяции: слева – направо показаны вогнутый полигон, полигон с дырой и полигон с пересекающимися ребрами.

Рисунок **11-1.** Контуры, нуждающиеся в тесселяции



Если вы считаете, что полигон нуждается в тесселяции, выполните следующие шаги:

1. Создайте новый объект тесселяции с помощью функции **gluNewTess()**.
2. Несколько раз используйте **gluTessCallback()** для регистрации функций обратного вызова, производящих операции во время тесселяции. Самый сложный случай при работе с возвратно-вызываемыми функция возникает, когда алгоритм тесселяции обнаруживает пересечение и должен вызвать функцию, зарегистрированную для события **GLU_TESS_COMBINE**.
3. Задайте свойства тесселяции с помощью функции **gluTessProperty()**. Наиболее важным свойством является правило оборота, определяющее какие регионы должны быть закрашены, а какие – остаться незакрашенными.
4. Создайте и визуализируйте тесселированные полигоны, задав контуры одного или более закрытых полигонов. Если данные объекта являются статическими, инкапсулируйте тесселированные полигоны в списке отображения. **(**Если вам не нужно пересчитывать тесселяцию снова и снова, использование списков отображения более эффективно.**)**
5. Если вам требуется тесселировать что-либо еще, вы можете использовать существующий объект тесселяции. Если вы закончили с тесселяцией, вы можете удалить объект функцией **gluDeleteTess()**.

Замечание: Тесселяция, описанная здесь, появилась в GLU версии 1.2. Если у вас более старая версия GLU, вы должны использовать функции, описанные в разделе «Описание ошибок GLU». Чтобы запросить версию используемой GLU, используйте функцию **gluGetString(GLU_VERSION)**, которая возвращает строку с номером версии GLU. Если в вашей GLU нет функции **gluGetString()**, значит это GLU версии 1.0 – в этой версии такая функция отсутствует.

11.1.1 Создание объекта тесселяции

Во время описания и тесселяции сложного полигона с ним ассоциируются некоторые данные, например, вершины, ребра и функции обратного вызова. Для выполнения тесселяции, ваша программа сначала должна создать объект тесселяции с помощью функции **gluNewTess()**.

```
GLUtesselator* gluNewTess (void);
```

Создает новый объект тесселяции и возвращает указатель на него. Если создать объект не удается, возвращается нулевой указатель.

Для всех тесселяций может использоваться один и тот же объект. Сам объект необходим исключительно потому, что функциям библиотеки могут потребоваться собственные тесселяции, и они должны иметь возможность выполнять их, не вмешиваясь в какие-либо тесселяции, выполняемые вашей программой. Несколько объектов полезно иметь также и в случае, если вы используете разные наборы возвратно вызываемых функций для разных тесселяций. Однако типичная программа создает только один объект и использует его для всех тесселяций. Освобождать объект, в общем-то, не имеет смысла, так как он требует очень небольшого объема памяти. С другой стороны, аккуратность еще никому не вредила.

11.1.2 Возвратно-вызываемые функции тесселяции

После того, как вы создали объект тесселяции, вы должны предоставить серию возвратно-вызываемых функций, которые будут вызываться в определенные моменты во время тесселяции. После определения этих функций, вы задаете контуры одного или нескольких полигонов с использованием функций **GLU**. После передачи описания контуров, механизм тесселяции вызывает ваши функции по необходимости.

Любые опущенные возвратно-вызываемые функции просто не вызываются в процессе тесселяции, и любая информация, которую они могут возвращать в вашу программу будет потеряна. Все возвратные функции задаются с помощью **gluTessCallback()**.

```
void gluTessCallback (GLUtesselator *tessobj, GLenum type, void (*fn)());
```

Ассоциирует возвратно-вызываемую функцию ***fn*** с объектом тесселяции ***tessobj***. Тип возвратной функции определяется аргументом ***type***, который может быть равен **GLU_TESS_BEGIN, GLU_TESS_BEGIN_DATA, GLU_TESS_EDGE_FLAG, GLU_TESS_EDGE_FLAG_DATA, GLU_TESS_VERTEX, GLU_TESS_VERTEX_DATA, GLU_TESS_END, GLU_TESS_END_DATA, GLU_TESS_COMBINE, GLU_TESS_COMBINE_DATA, GLU_TESS_ERROR** или **GLU_TESS_ERROR_DATA. 12** возможных возвратно-вызываемых функций имеют следующие прототипы:

GLU_TESS_BEGIN	**void begin (GLenum *type*);**
GLU_TESS_BEGIN_DATA	**void begin (GLenum *type*, void* *user_data*);**
GLU_TESS_EDGE_FLAG	**void edgeFlag (GLboolean *flag*);**
GLU_TESS_EDGE_FLAG_DATA	**void edgeFlag (GLboolean *flag*, void* *user_data*);**
GLU_TESS_VERTEX	**void vertex (void* *vertex_data*);**
GLU_TESS_VERTEX_DATA	**void vertex (void* *vertex_data*, void* *user_data*);**
GLU_TESS_END	**void end (void);**
GLU_TESS_END_DATA	**void end (void* *user_data*);**
GLU_TESS_COMBINE	**void combine (GLdouble *coords[3]*, void* *vertex_data[4]*, GLfloat *weight[4]*, void** *outData*);**
GLU_TESS_COMBINE_DATA	**void combine (GLdouble *coords[3]*, void* *vertex_data[4]*, GLfloat *weight[4]*, void** *outData*, void* *user_data*);**
GLU_TESS_ERROR	**void error (GLenum *errno*);**
GLU_TESS_ERROR_DATA	**void error (GLenum *errno*, void* *user_data*);**

Чтобы изменить возвратно-вызываемую функцию, просто вызовите **gluTessCallback()** с адресом новой функции. Чтобы устранить возвратную функцию без замещения ее новой, передайте **gluTessCallback()** нулевой указатель для соответствующего типа функции.

В течение тесселяции, возвратные функции вызываются в манере, похожей на ту, в которой вы используете команды **OpenGL glBegin()**, **glEdgeFlag*()**, **glVertex*()** и **glEnd()**. Функция комбинирования **(combine)** используется для создания новых вершин в точках пересечения ребер. Функция ошибок **(error)** вызывается в процессе тесселяции только тогда, когда что-то идет не так, как должно.

Для каждого созданного объекта тесселяции функция **GLU_TESS_BEGIN** вызывается с одним из **4** возможных параметров: **GL_TRIANGLE_FAN**, **GL_TRIANGLE_STRIP**, **GL_TRIANGLES** или **GL_LINE_LOOP**. Когда тесселятор декомпозирует (разбивает) полигоны, алгоритм тесселяции решает, какой тип треугольного примитива более эффективен для использования. (Если активизировано свойство **GLU_TESS_BOUNDARY_ONLY**, для визуализации используется **GL_LINE_LOOP**.)

Поскольку флаг ребра не имеет смысла в случаях **GL_TRIANGLE_FAN** или **GL_TRIANGLE_STRIP**, то, если существует заданная возвратная функция **GLU_TESS_EDGE_FLAG**, активизирующая флаги ребра, функция **GLU_TESS_BEGIN** вызывается только с аргументом **GL_TRIANGLES**. Функция **GLU_TESS_EDGE_FLAG** работает абсолютно аналогично вызову команды **OpenGL glEdgeFlag*()**.

После вызова функции, ассоциированной с **GLU_TESS_BEGIN** и до вызова функции, ассоциированной с **GLU_TESS_END**, вызывается некоторая комбинация функций **GLU_TESS_VERTEX** и **GLU_TESS_EDGE_FLAG** (обычно из-за обращений к функции **gluTessVertex()**). Ассоциированные флаги ребра и вершины интерпретируются точно таким же образом, как если бы они задавались между командами **OpenGL glBegin()** и **glEnd()**.

Если что-то идет не так, возвратной функции ошибки передается номер ошибки **GLU**. Символьная строка, описывающая ошибку, извлекается с использованием функции **gluErrorString()**.

Пример **11-1** демонстрирует часть кода файла **tess.cpp**, в котором создается объект тесселяции и регистрируется несколько функций обратного вызова.

Пример **11-1**. Регистрация возвратных функций тесселяции: файл **tess.cpp**

```
#ifndef CALLBACK
#define CALLBACK
#endif

/* часть функции init() */
tobj=gluNewTess();
gluTessCallback(tobj,GLU_TESS_VERTEX,glVertex3dv);
gluTessCallback(tobj,GLU_TESS_BEGIN,beginCallback);
gluTessCallback(tobj,GLU_TESS_END,endCallback);
gluTessCallback(tobj,GLU_TESS_ERROR,errorCallback);

/* возвратно вызываемые функции, зарегистрированные с помощью
gluTessCallback() */
void CALLBACK beginCallback(GLenum which)
{
      glBegin(which);
}

void CALLBACK endCallback(void)
{
      glEnd();
}

void CALLBACK errorCallback(GLenum errorCode)
```

```
{
      const GLubyte *estring;

      estring=gluErrorString(errorCode);
      fprintf(stderr, "Tesselation error: %s\n",estring);
      exit(0);
}
```

Замечание: Приведение типов возвратно вызываемых функций довольно сложно, особенно, если вы хотите создать код, который будет одинаково работать на платформах Microsoft Windows (95/98/NT) и UNIX. Чтобы верно работать на платформах Microsoft Windows, программам, использующим возвратно-вызываемые функции, таким как tess.cpp требуется наличие символа CALLBACK в объявлении функции. Трюк с использованием пустого определения для CALLBACK (как показано ниже) позволяет коду запускаться как на Microsoft Windows, так и в UNIX.

```
#ifndef CALLBACK
#define CALLBACK
#endif

void CALLBACK callbackFunction(..)
{
}
```

В примере **11-1,** зарегистрированная функция для **GLU_TESS_VERTEX,** представляет собой просто команду **glVertex3dv(),** в которую передаются только координаты каждой вершины. Однако, если вы желаете задавать больше информации в каждой вершине, например, цветовые величины, вектор нормали или координаты текстуры, вам следует создать более сложную функцию обратного вызова. Пример **11-2** демонстрирует начало другого тесселируемого объекта далее в программе **tess.cpp.** Зарегистрированная функция **vertexCallback()** ожидает параметра, являющегося указателем на **6** чисел с плавающей точкой двойной точности: координаты ***x***, ***y*** и ***z***, а также значения цветовых компонент красного, зеленого и синего для каждой вершины.

Пример **11-2.** Возвратно-вызываемые функции **GLU_TESS_VERTEX** и **GLU_TESS_COMBINE**

```
/* другая часть функции init() */
gluTessCallback(tobj,GLU_TESS_VERTEX,vertexCallback);
gluTessCallback(tobj,GLU_TESS_BEGIN,beginCallback);
gluTessCallback(tobj,GLU_TESS_END,endCallback);
gluTessCallback(tobj,GLU_TESS_ERROR,errorCallback);
gluTessCallback(tobj,GLU_TESS_COMBINE,combineCallback);

/* новые возвратно-вызываемые функции */
void CALLBACK vertexCallback(GLvoid* vertex)
{
      const GLdouble* pointer;

      pointer=(GLdouble*)vertex;
      glColor3dv(pointer+3);
      glVertex3dv(vertex);
}

void CALLBACK combineCallback(GLdouble coords[3],
                              GLdouble* vertex_data[4],
                              GLfloat weight[4], GLdouble** dataOut)
{
      GLdouble *vertex;
      int i;
```

```
        vertex=(GLdouble*) malloc(6*sizeof(GLdouble));
        vertex[0]=coords[0];
        vertex[1]=coords[1];
        vertex[2]=coords[2];
        for(i=3;i<6;i++)
              vertex[i]=weight[0]*vertex_data[0][i]+
                        weight[1]*vertex_data[1][i]+
                        weight[2]*vertex_data[2][i]+
                        weight[3]*vertex_data[3][i];
        *dataOut=vertex;
  }
```

Пример **11-2** также демонстрирует использование функции **GLU_TESS_CALLBACK.** Каждый раз, когда алгоритм тесселяции, анализирующий входящий контур, обнаруживает пересечение и решает, что должна быть создана новая вершина, вызывается функция обратного вызова зарегистрированная для **GLU_TESS_COMBINE.** Эта функция также вызывается в случае, если алгоритм решает объединить две вершины, которые очень близки друг к другу. Новая вершина является линейной комбинацией до четырех существующих вершин, на которые в примере **11-2** ссылаются как на ***vertex_data[0..3]***. Коэффициенты линейной комбинации передаются в ***weight[0..3]*** **(**чья сумма составляет ***1.0***). Аргумент ***coords*** задает положение новой вершины.

Зарегистрированная функция обратного вызова должна зарезервировать память для новой вершины, произвести взвешенную интерполяцию данных с использованием ***vertex_data*** и ***weight*** и возвратить указатель на новую вершину в аргументе ***data_out***. **combineCallback()** в примере **11-2** интерполирует цветовую величину. Функция резервирует массив из **6**-ти элементов, помещает ***x***, ***y*** и ***z*** в первые три элемента, а взвешенное среднее цветовых величин **RGB** в следующие три элемента.

11.1.2.1 Данные, определенные пользователем

Может быть зарегистрировано шесть типов возвратно-вызываемых функций. Поскольку существует два варианта каждой возвратной функции, существует всего **12** функций. Для каждой функции существует один вариант с данными, определенными пользователем, и один вариант без них. Данные, определенные пользователем передаются приложением в функцию **gluTessBeginPolygon()**, а затем без изменений передаются всем возвратным функциям ***DATA.** С функцией **GLU_TESS_BEGIN_DATA** данные, определенные пользователем могут использоваться в качестве данных для одного полигона. Если для определенного типа возвратно-вызываемой функции вы зададите оба варианта, будет использоваться вариант с аргументом ***user_data***. Таким образом, несмотря на то, что существует **12** возвратных функций, в каждый момент времени активными могут быть только **6.**

Например, в примере **11-2** используется плавная закраска, и **vertexCallback()** задает **RGB** цвет для каждой вершины. Если вы хотите использовать плавную заливку и освещение, функция обратного вызова должна задавать вектор нормали для каждой вершины. Однако, если вы хотите использовать освещение и плоскую закраску, вы можете задать только одни вектор нормали для каждого полигона, а не для каждой вершины. В таком случае вы можете использовать функцию **GLU_TESS_BEGIN_DATA** и передать координаты вершины и нормаль к поверхности в аргументе ***user_data***.

11.1.3 Свойства тесселяции

Перед тесселяцией и визуализацией вы можете использовать функцию **gluTessProperty()** для настройки некоторых свойств, влияющих на алгоритм

тесселяции. Наиболее важное и сложное из этих свойств – правило оборота, определяющее, что считается «внутренним», а что «внешним».

```
void gluTessProperty (GLUtesselator* tessobj, GLenum property, GLdouble value);
```

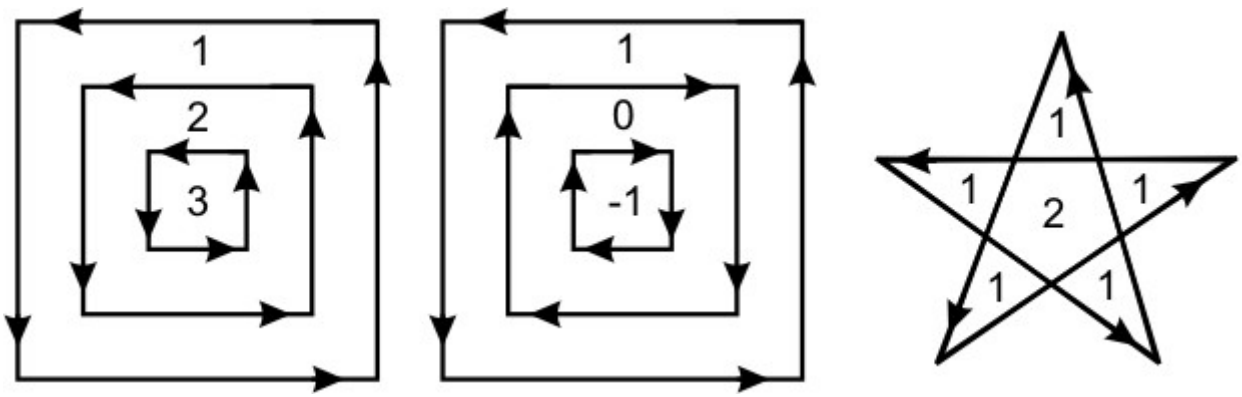
Значение свойства **property** объекта тесселяции ***tessobj*** устанавливается в значение **value.** Аргумент ***property*** может принимать значения **GLU_TESS_BOUNDARY_ONLY, GLU_TESS_TOLERANCE** или **GLU_TESS_WINDING_RULE.** Если ***property*** равно **GLU_TESS_BOUNDARY_ONLY, *value*** может принимать значения **GL_TRUE** или **GL_FALSE.** Если оно равно **GL_TRUE** полигоны не тесселируются на закрашенные полигоны – рисуются только замкнутые ломаные, показывающие границу контуров, разделяющих интерьер и экстерьер полигона. Значение по умолчанию – **GL_FALSE. (**Смотрите описание **gluTessNormal()** для понимания того, как управлять направлением перекрытия контуров.**)** Если ***property*** равно **GLU_TESS_TOLERANCE, *value*** представляет собой дистанцию, используемую при вычислении того, являются ли две вершины близкими в достаточной степени, чтобы их можно было объединить с помощью возвратной функции **GLU_TESS_COMBINE.** Величина толерантности умножается на максимальную разницу в координатах входящих вершин для определения максимальной дистанции, на которую может сместиться какой-либо фрагмент вследствие одной операции совмещения. Совмещение может не поддерживаться вашей реализацией **OpenGL,** а величина толерантности **(**терпимости**)** имеет только рекомендательный характер. Величина толерантности имеет нулевое значение по умолчанию. Свойство **GLU_TESS_WINDING_RULE** определяет, какие части полигона находятся внутри, а какие снаружи и не должны быть закрашены. ***value*** может принимать значения **GLU_TESS_WINDING_ODD (**значение по умолчанию**),** **GLU_TESS_WINDING_NONZERO, GLU_TESS_WINDING_POSITIVE, GLU_TESS_WINDING_NEGATIVE** или **GLU_TESS_WINDING_ABS_GEQ_TWO.**

11.1.3.1 Число оборотов и правило оборота

Для отдельного контура, число оборотов точки представляет собой знаковое целое число полных оборотов, которые мы совершаем вокруг этой точки, проходя вдоль контура **(**обороты против часовой стрелки добавляют **1** к этому числу, а обороты по часовой стрелке вычитают **1) .** Когда существует несколько контуров, индивидуальные числа оборотов суммируются. Данная процедура ассоциирует знаковое число со знаком с каждой точкой плоскости. Отметьте, что число оборотов является одним и тем же для всех точек одной ограниченной области.

На рисунке **11-2** показано три набора контуров и числа оборотов для точек внутри контуров. В наборе слева все три контура идут против часовой стрелки, так что каждый контур добавляет единицу к числам оборотов точек, находящихся внутри него. В наборе посередине два внутренних контура имеют направление по часовой стрелке, так что число оборотов уменьшается и в итоге становится меньше **0.**

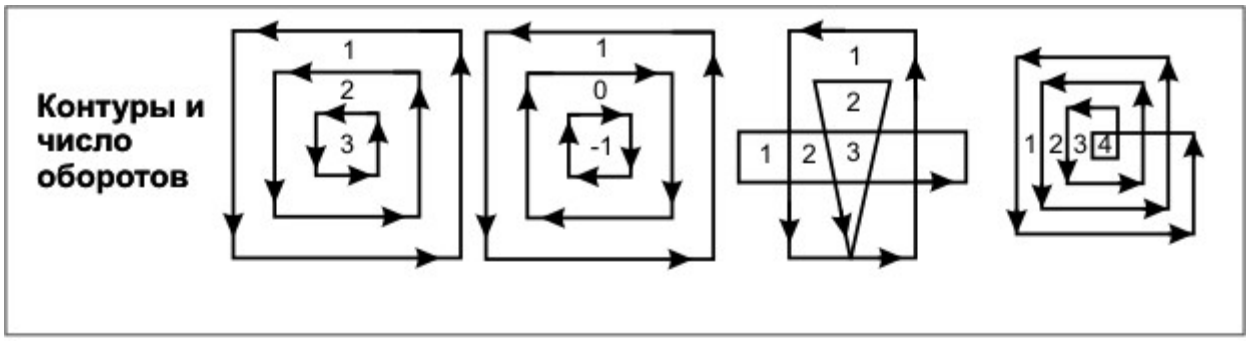
Рисунок **11-2.** Числа оборотов для простых контуров

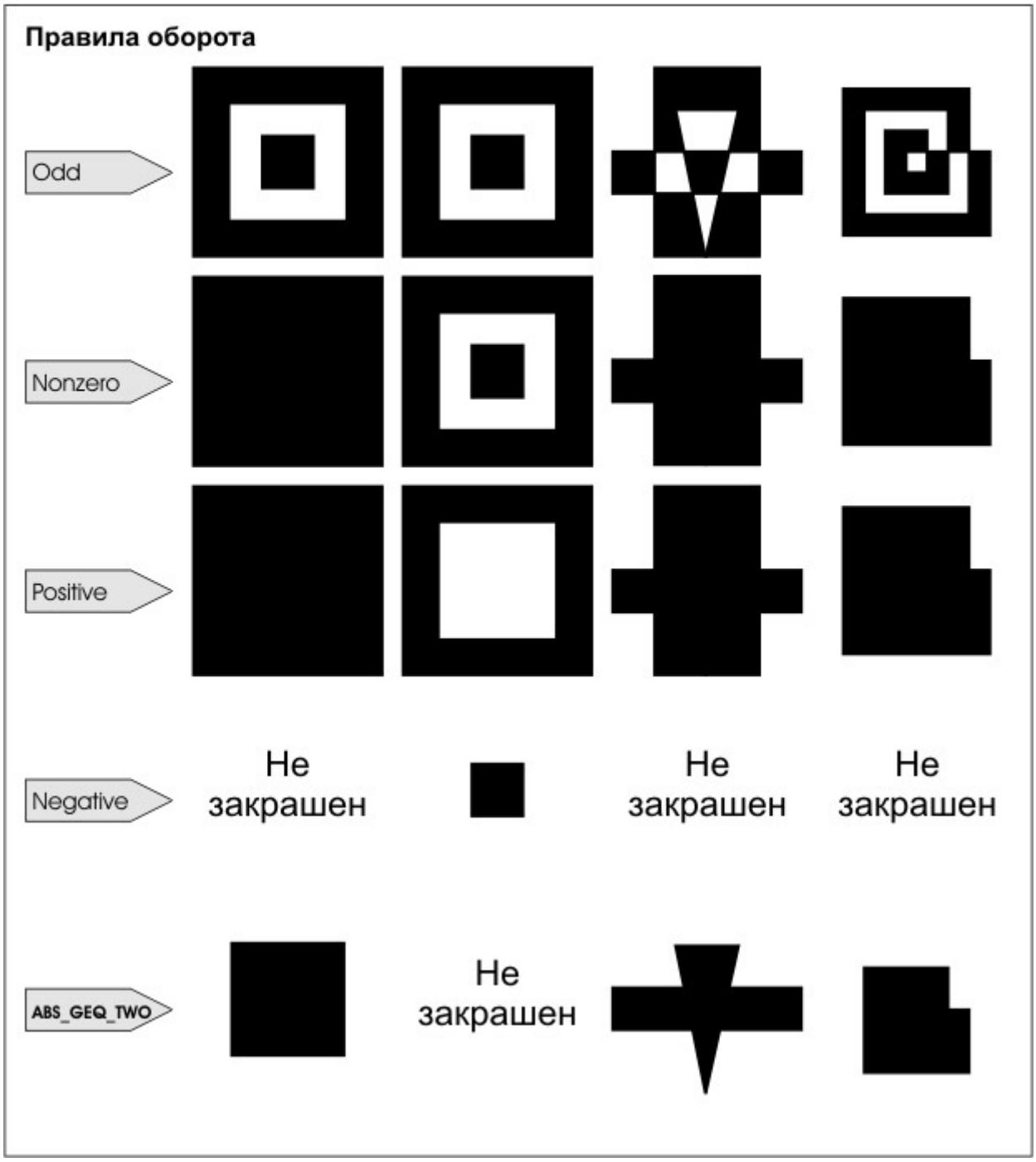


Правило оборота классифицирует регион, как внутренний, если его число оборотов принадлежит к определенной категории **(odd –** нечетное**, nonzero –** ненулевое**, positive –** положительное**, negative –** отрицательное или **abs – «**абсолютная величина больше или равная **2»).** Правила **GLU_TESS_WINDING_ODD** и **GLU_TESS_WINDING_NONZERO** часто используются для определения интерьера**.** Правила позитивное**,** негативное и **«**абсолютная величина**»** имеют ограниченное применение при выполнении операций над **CSG (computational solid geometry –** плоская вычисляемая геометрия**).**

Рисунок **11-3** демонстрирует влияние различных правил оборота на визуализацию контуров**.** Темные области на рисунке являются внутренними**.**

Рисунок **11-3.** Как правила оборота определяют интерьер





11.1.3.2 Использование правил оборота с CSG

Правила оборота **GLU_TESS_WINDING_ODD** и **GLU_TESS_WINDING_NONZERO** являются наиболее часто используемыми. Они работают в большинстве типичных случаев заливки.

Правила оборота были разработаны и для операций с **CSG** и позволяют легко находить объединение, разницу или пересечение **(**Булевские операции**)** нескольких контуров.

Вначале предположим, что каждый контур определен таким образом, что число оборотов для каждого внешнего региона равно **0**, а для каждого внутреннего **– 1. (**То есть ни один из контуров не пересекает сам себя.**)** Далее считаем, что контуры, проходящие против часовой стрелки, обозначают границы полигонов, а контуры, идущие по часовой стрелке, **–** дыры. Контуры могут быть вложенными, но внутренний контур должен иметь направление противоположное направлению содержащего его контура.

Если изначальные полигоны не удовлетворяют данным условиям, их можно привести к ним, выполним предварительную тесселяцию с активизированным параметром **GLU_TESS_BOUNDARY_ONLY.** Эта тесселяция возвратит список полигонов, удовлетворяющих описанным ограничениям. Если создать два объекта тесселяции, то возвратные функции первого из них могут направлять информацию непосредственно на вход второго.

При наличии двух или более полигонов в описанной форме, операции **CSG** могут быть реализованы следующим образом:

- **UNION (**объединение**) –** чтобы вычислить объединение нескольких контуров, нарисуйте все входящие контуры в виде одного полигона. Число оборотов каждой результирующей области **–** это сумма чисел оборотов входящих полигонов, которые ее покрывают. Объединение может быть получено с помощью правил **GLU_TESS_WINDING_NONZERO** или **GLU_TESS_WINDING_POSITIVE.** Заметьте, что в случае ненулевого правила, мы получим тот же результат, даже если обратим направление всех контуров.
- **INTERSECTION (**пересечение**) –** его можно получить только для двух контуров за один раз. Нарисуйте один полигон с использованием двух контуров. Результат получается с помощью правила **GLU_TESS_WINDING_ABS_GEQ_TWO**.
- **DIFFERENCE (**разница**) –** предположим, что вы хотите вычислить **A diff (B union C union D).** Нарисуйте единственный полигон, состоящий из неизмененного контура **A**, за которым следуют контуры **B**, **C** и **D** с обратным порядком вершин. Для получения результата используйте правило **GL_TESS_WINDING_POSITIVE. (**Если **B**, **C** и **D** являются результатом операции **GLU_TESS_BOUNDARY_ONLY**, то можно не изменять порядок вершин, а воспользоваться функцией **gluTessNormal()** для изменения знака поставляемой нормали.**)**

11.1.3.3 Другие функции для работы со свойствами тесселяции

Существует две функции, работающие совместно с **gluTessProperty().** **gluTessGetProperty()** позволяет получать текущие значения свойств тесселяции. Если тесселятор используется для генерирования проволочных каркасов, а не закрашенных полигонов, функция **gluTessNormal()** позволяет определить направление оборота для тесселируемых полигонов.

```
void gluTessGetProperty (GLUtesselator* tessobj, GLenum property, GLdouble*
value);
```

Возвращает текущее значение свойства ***property*** объекта тесселяции ***tessobj*** в переменной ***value***. Возможные значения для аргументов ***property*** и ***value*** те же самые, что и в функции **gluTessGetProperty().**

```
void gluTessNormal (GLUtesselator* tessobj, GLdouble x, GLdouble y, GLdouble z);
```

Задает вектор нормали для объекта тесселяции ***tessobj***. Вектор нормали задает направление оборота для генерируемых полигонов. Перед тесселяцией все входные данные проецируются на плоскость перпендикулярную вектору нормали. После все результирующие треугольники ориентируются против часовой стрелки с учетом нормали. (Направление по часовой стрелке может быть получено путем изменения знака задаваемой нормали.) Вектор нормали по умолчанию – (***0***, ***0***, ***0***).

Если у вас есть данные о положении и ориентации входных данных, использование **gluTessNormal()** может увеличить скорость тесселяции. Например, если вы знаете, что все полигоны лежат в плоскости ***xy***, вызовите **gluTessNormal(*tessobj*,*0*,*0*,*1*)**.

Вектор нормали по умолчанию (***0***, ***0***, ***0***), и его влияние не всегда очевидны. В подобном случае ожидается, что все входные данные лежат примерно в одной плоскости и плоскость охватывает все вершины независимо от того, как они на самом деле соединены. Знак нормали выбирается таким образом, чтобы сумма всех знаковых областей всех входных контуров была неотрицательной (там, где контур против часовой стрелки окружает положительную область). Заметьте, что если входные данные не лежат примерно в одной плоскости, проецирование перпендикулярно вычисленной нормали может существенно изменить геометрию.

11.1.4 Определение полигона

После того, как установлены все свойства тесселяции и зарегистрированы все функции обратного вызова, наступает время описывать вершины, составляющие входящие контуры, и тесселировать полигоны.

```
void gluTessBeginPolygon (GLUtesselator* tessobj, void* user_data);
void gluTessEndPolygon (GLUtesselator* tessobj);
```

Открывают и завершают спецификацию полигона, который нужно тесселировать и ассоциируют с ним объект тесселяции ***tessobj***. ***user_data*** указывает на данные определяемые пользователем, которые передаются всем связанным возвратным функциям **GLU_TESS_*_DATA**.

Вызовы **gluTessBeginPolygon()** и **gluTessEndPolygon()** обрамляют определение одного или более контуров. Когда вызывается **gluTessEndPolygon()**, выполняется алгоритм тесселяции, генерируются и визуализируются тесселированные полигоны. В течение этого алгоритма используются связанные функции обратного вызова и установленные свойства тесселяции.

```
void gluTessBeginContour (GLUtesselator* tessobj);
void gluTessEndContour (GLUtesselator* tessobj);
```

Открывают и завершают спецификацию замкнутого контура, являющегося частью полигона. Замкнутый контур определяется нулем или более обращениями к функции **gluTessVertex()**, задающей вершину. Последняя вершина каждого контура автоматически соединяется с первой.

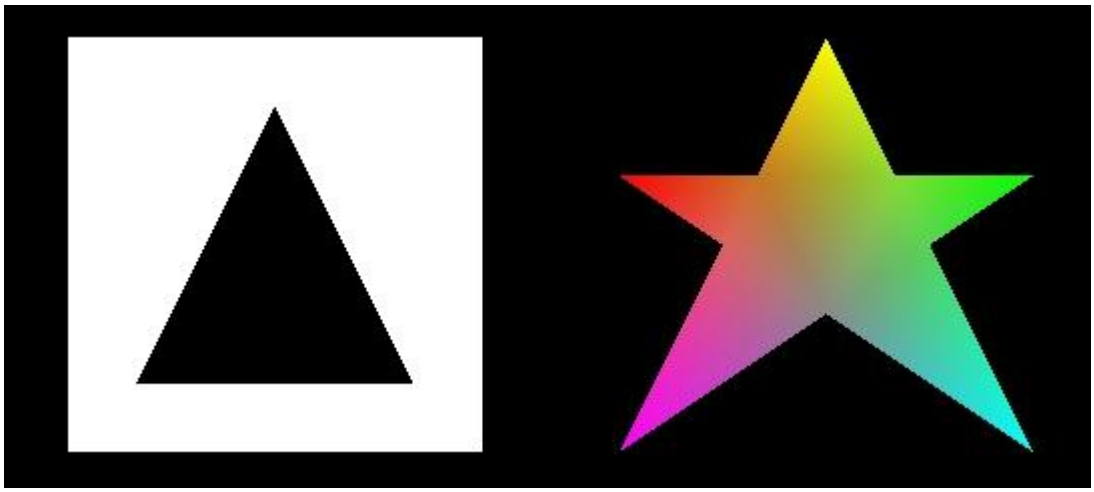
На практике для осмысленного контура требуется как минимум **3** вершины.

```
void gluTessVertex (GLUtesselator* tessobj, GLdouble coords[3], void*
vertex_data);
```

Задает одну вершину в текущем контуре для объекта тесселяции ***tessobj***. ***coords*** содержит трехмерные координаты вершины, а ***vertex_data*** – это указатель, отсылаемые функции обратного вызова **GLU_TESS_VERTEX** или **GLU_TESS_VERTEX_DATA**. Обычно, ***vertex_data*** содержит координаты вершины, нормали к поверхности, координаты текстуры, информацию о цвете или что-либо другое, нужное приложению.

В программе **tess.cpp**, часть которой приводится в примере **11-3**, определяются два полигона. Первый полигон состоит из прямоугольного контура с треугольной дырой внутри, а второй представляет собой плавно закрашенную самопересекающуюся пятиконечную звезду. Для большей эффективности оба полигона сохраняются в списке отображения. Первый полигон состоит из двух контуров, причем внешний ориентирован против часовой стрелки, а внутренний **(«**дыра**»)** – по часовой стрелке. Для второго полигона массив ***star*** содержит и координаты вершин и информацию о цвете, и функция обратного вызова **vertexCallback()** использует и то, и другое. Результат работы программы **tess.cpp** изображен на рисунке **11-4**.

Рисунок **11-4**. Результат работы тесселятора



Важно, чтобы каждая вершина находилась в своей области памяти, так как вершинные данные не копируются функцией **gluTessVertex() –** сохраняется только указатель **(*vertex_data*)**. Программа, которая использует одну и ту же память для нескольких вершин, может не получить ожидаемого результата.

Замечание: Может показаться, что в функции **gluTessVertex()** бессмысленно задавать координаты вершин дважды – в аргументах ***coords*** и ***vertex_data***. Однако иногда это необходимо – ***coords*** ссылается только на координаты вершины, а ***vertex_data*** также хранит координаты, но может содержать и другую информацию для каждой вершины.

Пример **11-3**. Определение полигона: файл **tess.cpp**

```
GLdouble rect[4][3]= { 50.0, 50.0, 0.0,
                       200.0,50.0,0.0,
                       200.0,200.0,0.0,
                       50.0,200.0,0.0 };
GLdouble tri[3][3]={ 75.0,75.0,0.0,
                     125.0,175.0,0.0,
                     175.0,75.0,0.0 };
GLdouble star[5][6]={ 250.0,50.0,0.0,1.0,0.0,1.0,
                      325.0,200.0,0.0,1.0,1.0,0.0,
                      400.0,50.0,0.0,0.0,1.0,1.0,
                      250.0,150.0,0.0,1.0,0.0,0.0,
                      400.0,150.0,0.0,0.0,1.0,0.0 };
```

```
startList=glGenLists(2);
tobj=gluNewTess();
gluTessCallback(tobj,GLU_TESS_VERTEX,glVertex3dv);
gluTessCallback(tobj,GLU_TESS_BEGIN,beginCallback);
gluTessCallback(tobj,GLU_TESS_END,endCallback);
gluTessCallback(tobj,GLU_TESS_ERROR,errorCallback);

glNewList(startList,GL_COMPILE);
      glShadeModel(GL_FLAT);
      gluTessBeginPolygon(tobj,NULL);
            gluTessBeginContour(tobj);
                  gluTessVertex(tobj,rect[0],rect[0]);
                  gluTessVertex(tobj,rect[1],rect[1]);
                  gluTessVertex(tobj,rect[2],rect[2]);
                  gluTessVertex(tobj,rect[3],rect[3]);
            gluTessEndContour(tobj);
            gluTessBeginContour(tobj);
                  gluTessVertex(tobj,tri[0],tri[0]);
                  gluTessVertex(tobj,tri[1],tri[1]);
                  gluTessVertex(tobj,tri[2],tri[2]);
            gluTessEndContour(tobj);
      gluTessEndPolygon(tobj);
glEndList();

gluTessCallback(tobj,GLU_TESS_VERTEX,vertexCallback);
gluTessCallback(tobj,GLU_TESS_BEGIN,beginCallback);
gluTessCallback(tobj,GLU_TESS_END,endCallback);
gluTessCallback(tobj,GLU_TESS_ERROR,errorCallback);
gluTessCallback(tobj,GLU_TESS_COMBINE,combineCallback);

glNewList(startList+1,GL_COMPILE);
      glShadeModel(GL_SMOOTH);
      gluTessProperty(tobj,GLU_TESS_WINDING_RULE,
GLU_TESS_WINDING_POSITIVE);
      gluTessBeginPolygon(tobj,NULL);
            gluTessBeginContour(tobj);
                  gluTessVertex(tobj,star[0],star[0]);
                  gluTessVertex(tobj,star[1],star[1]);
                  gluTessVertex(tobj,star[2],star[2]);
                  gluTessVertex(tobj,star[3],star[3]);
                  gluTessVertex(tobj,star[4],star[4]);
            gluTessEndContour(tobj);
      gluTessEndPolygon(tobj);
glEndList();
```

11.1.5 Удаление объекта тесселяции

Если вы более не нуждаетесь в объекте тесселяции**,** вы можете удалить его и освободить всю связанную с ним память с помощью функции **gluDeleteTess().**

```
void gluDeleteTess (GLUtesselator* tessobj);
```

Удаляет указанный объект тесселяции ***tessobj*** и освобождает всю связанную с ним память**.**

11.1.6 Советы по увеличению быстродействия тесселяции

Для наилучшего быстродействия соблюдайте следующие правила**.**

1. Кэшируйте результаты тесселяции в списке отображения или другой пользовательской структуре. Чтобы получить пост – тесселяционные координаты вершин, тесселируйте полигон, находясь в режиме отклика.
2. Используйте **gluTessNormal()**, чтобы задавать нормаль полигона.
3. Используйте для нескольких полигонов один и тот же объект тесселяции, а не создавайте каждый раз новый. (В многопоточном многопроцессорном окружении вы можете получить большее быстродействие за счет использования нескольких тесселяторов.)

11.1.7 Описание ошибок GLU

GLU предоставляет функцию для получения описательной строки кода ошибки. Эта функция не ограничена использованием только с тесселяторами, она также используется при работе с **NURBS** и квадрическими объектами, а также для описания ошибок **GL**.

11.1.8 Обратная совместимость

Если вы используете **GLU** версий **1.0** или **1.1**, в вашем распоряжении намного менее мощный тесселятор. Тесселятор версий **1.0/1.1** обрабатывает только простые невыпуклые полигоны или простые полигоны с дырами. Он не может правильно тесселировать пересекающиеся контуры (в нем отсутствует возвратная функция **COMBINE**) или обрабатывать по-полигонные данные. Тесселятор версий **1.0/1.1** все еще работает в **GLU** версий **1.2/1.3**, но его использование не рекомендуется.

В тесселяторе версии **1.0/1.1** есть несколько похожих черт с новым тесселятором. Функции **gluNewTess()** и **gluDeleteTess()** используются в обоих тесселятора. Главной функцией, задающей вершины, также является **gluTessVertex()**. Механизм возвратно-вызываемых функций также контролируется функцией **gluTessCallback()**, однако старый тесселятор работает только с **5** типами возвратных функций – подмножеством текущих **12**.

Вот прототипы функций для тесселятора версии **1.0/1.1**:

```
void gluBeginPolygon (GLUtriangulatorObj* tessobj);
void gluNextContour (GLUtriangulatorObj* tessobj, GLenum type);
void gluEndPolygon (GLUtriangulatorObj* tessobj);
```

Самый внешний контур должен быть задан первым, и эта процедура не требует предварительного вызова **gluNextContour()**. Для полигонов без дыр задается только один контур, а функция **gluNextContour()** не используется. Если полигон состоит из нескольких контуров (то есть в нем есть дыры или дыры внутри дыр), эти контуры задаются один за другим, причем каждому должен предшествовать вызов функции **gluNextContour()**. Функция **gluTessVertex()** вызывается для каждой вершины каждого контура. Значениями для аргумента ***type*** функции **gluNextContour()** могут быть **GLU_EXTERIOR**, **GLU_INTERIOR**, **GLU_CCW**, **GLU_CW** или **GLU_UNKNOWN**. Этот аргумент служит только в качестве рекомендации для тесселятора. Если вы выберете для него правильное значение, тесселяция может завершиться быстрее. Если же выбранное значение неверно, аргумент будет проигнорирован, тесселяция пройдет медленнее, но по-прежнему успешно. Для полигонов с дырами один контур является внешним (**GLU_EXTERIOR**), а все остальные – внутренними (**GLU_INTERIOR**). Предполагается, что первый контур будет иметь тип **GLU_EXTERIOR**. Выбор ориентации по или против часовой стрелки (**GLU_CW** или **GLU_CCW**) может быть произвольным в трехмерном пространстве. Однако на любой плоскости существует две различных ориентации, поэтому **GLU_CCW** и **GLU_CW** должно быть последовательным. Если вы не знаете, к какому типу относится контур, используйте **GLU_UNKNOWN**.

Строго рекомендуется приводить код для **GLU 1.0/1.1** к коду, использующему новый интерфейс тесселяции. Это можно сделать, выполнив следующие шаги.

1. Измените тип ссылок на объекты тесселяции с **GLUtriangulatorObj** на **GLUtesselator**. В **GLU** версии **1.2** эти типы определены одинаковым образом.
2. Конвертируйте **gluBeginPolygon()** в две команды: **gluTessBeginPolygon()** и **gluTessBeginContour()**. Все контуры должны быть заданы внутри, включая самый первый.
3. Конвертируйте **gluTessContour()** на **gluTessEndContour()** и **gluTessBeginContour()**. Вы должны завершить предыдущий контур до того, как начать следующий.
4. Замените **gluEndPolygon()** на **gluTessEndContour()** и **gluTessEndPolygon()**. Этим вы закроете последний контур.
5. В вызовах функции **gluTessCallback()** измените ссылки на константы. В **GLU 1.2** **GLU_BEGIN**, **GLU_VERTEX**, **GLU_END**, **GLU_ERROR** и **GLU_EDGE_FLAG** определены как синонимы для **GLU_TESS_BEGIN**, **GLU_TESS_VERTEX**, **GLU_TESS_END**, **GLU_TESS_ERROR** и **GLU_TESS_EDGE_FLAG**.

11.2 Квадрические объекты: визуализация сфер, цилиндров и дисков

Базовая библиотека **OpenGL** предоставляет поддержку только для моделирования и визуализации простых точек, линий и выпуклых закрашенных полигонов. Ни **3D** объекты, ни часто используемые **2D** объекты, такие как круги, непосредственно недоступны.

GLU предоставляет функции для моделирования и визуализации тесселированных полигональных аппроксимаций различных **2D** и **3D** фигур (сфер, цилиндров, дисков и частей дисков), которые могут быть вычислены с помощью квадратных уравнений (именно поэтому такие объекты иногда называют квадрическими). Предоставляемая поддержка включает функции для отображения квадрических поверхностей различными стилями и с различной ориентацией. Квадрические поверхности в общем виде определяются следующим квадратным уравнением:

$$a_1x^2 + a_2y^2 + a_3z^2 + a_4xy + a_5yz + a_6xz + a_7x + a_8y + a_9z + a_{10}$$

Процесс создания и визуализации квадрической поверхности похож на процесс использования тесселятора. Чтобы использовать квадрический объект, выполните следующие шаги:

1. Чтобы создать новый квадрический объект, используйте функцию **gluNewQuadric()**.
2. Задайте атрибуты визуализации для квадрического объекта (если, конечно, вы не удовлетворены атрибутами по умолчанию):
    a. Используйте **gluQuadricOrientation()** для настройки правила оборота и отделения интерьера от экстерьера.
    b. Используйте **gluQuadricDrawStyle()**, чтобы выбрать стиль визуализации – в виде точек, линий или закрашенных полигонов.
    c. Если объекты предполагается освещать, используйте **gluQuadricNormals()**, задавая по одной нормали на вершину или по одной нормали на грань. По умолчанию нормали вообще не генерируются.
    d. Если объекты предполагается текстурировать, используйте **gluQuadricTexture()**, чтобы настроить механизм автоматической генерации координат текстуры.

3. Зарегистрируйте возвратно-вызываемую функцию обработки ошибок с помощью **gluQuadricCallback()**. Возвратная функция будет вызвана, если в процессе визуализации возникнет ошибка.
4. Вызовите функцию визуализации желаемого квадрического объекта: **gluSphere()**, **gluCylinder()**, **gluDisk()** или **gluPartialDisk()**. Для увеличения скорости работы со статическими данными, инкапсулируйте квадрический объект в списке отображения.
5. После полного завершения работы с квадрическим объектом удалите его функцией **gluDeleteQuadric()**. Если вам нужно несколько фигур, лучше использовать для них всех один и тот же квадрический объект.

11.2.1 Управление квадрическими объектами

Квадрический объект состоит из параметров, аттрибутов и возвратно-вызываемых функций, сохраняемых в структуре данных типа **GLUquadricObj**. Квадрический объект может генерировать вершины, нормали, координаты текстуры и другие данные, которые могут использоваться либо непосредственно, либо сохраняться в списке отображения для более позднего использования. Следующие функции создают и уничтожают квадрический объект, а также устанавливают возвратную функцию обработки ошибок.

```
GLUquadricObj* gluNewQuadric (void);
```

Создает новый квадрический объект и возвращает указатель на него. В случае неудачи функция возвращает нулевой указатель.

```
void gluDeleteQuadric (GLUquadricObj* qobj);
```

Уничтожает квадрический объект ***gobj*** и освобождает всю память, используемую им.

```
void gluQuadricCallback (GLUquadricObj* qobj, GLenum which, void (*fn)());
```

Устанавливает, что функция ***fn*** будет вызвана в определенных ситуациях. Единственным допустимым значением для аргумента ***which*** является **GLU_ERROR**, то есть функция ***fn*** вызывается в случае ошибки. Если аргумент ***fn*** равен **NULL**, удаляется ссылка на текущую возвратную функцию.

В случае **GLU_ERROR** ***fn*** вызывается с одним аргументом, равным коду ошибки. Функция **gluErrorString()** может быть полезна для конверсии этого кода в **ASCII** строку.

11.2.2 Управление атрибутами квадрических объектов

Следующие функции воздействуют на то, какие данные генерируются квадрическими функциями. Эти функции следует использовать до создания примитивов.

Пример **11-4**, **quadric.cpp** демонстрирует изменение стиля рисования и характера генерируемых нормалей, а также создание квадрических объектов, обработку ошибок и рисование примитивов.

```
void gluQuadricDrawStyle (GLUquadricObj* qobj, GLenum drawStyle);
```

Аргумент ***drawStyle*** управляет стилем визуализации для объекта ***qobj***. Допустимыми значениями ***drawStyle*** могут быть **GLU_POINT**, **GLU_LINE**, **GLU_SILHOUETTE** и **GLU_FILL**.

GLU_POINT и **GLU_LINE** задают, что примитивы должны визуализироваться в виде точки в каждой вершине или в виде линий между парами соединяющихся вершин.

GLU_SILHOUETTE также задает режим отображения в виде линий, но ребра, разделяющие соседние грани, не рисуются. Этот режим чаще всего используется для дисков или их частей.

GLU_FILL задает визуализацию закрашенных полигонов там, где полигоны рисуются с ориентацией против часовой стрелки с учетом их нормалей. Данный параметр учитывает установки команды **gluQuadricOrientation()**.

```
void gluQuadricOrientation (GLUquadricObj* qobj, GLenum orientation);
```

Для объекта ***qobj*** аргумент ***orientation*** может принимать значения **GLU_OUTSIDE (**наружу, значение по умолчанию**)** или **GLU_INSIDE (**вовнутрь**)** и управляет направлением нормалей.

Для сфер и цилиндров определение понятий наружу и вовнутрь очевидно. Для дисков и их частей сторона диска с положительным ***z*** считается находящейся снаружи.

```
void gluQuadricNormals (GLUquadricObj* qobj, GLenum normals);
```

Для объекта ***qobj***, аргумент ***normals*** может принимать значения **GLU_NONE (**нормали не генерируются, значение по умолчанию**), GLU_FLAT (**генерация полигональных нормалей**)** или **GLU_SMOOTH (**генерация истинных нормалей**).**

gluQuadricNormals() используется для указания того, как генерировать нормали. **GLU_NONE** означает, что нормали вообще не генерируются, этот режим используется при работе без освещения. **GLU_FLAT** означает, что нормаль будет сгенерирована для каждой грани, что обычно используется при работе с освещением в режиме плоской заливки. **GLU_SMOOTH** означает, что нормаль будет сгенерирована для каждой вершины, этот режим дает наилучшие результаты при работе с освещением и плавной заливкой.

```
void gluQuadricNormals (GLUquadricObj* qobj, GLenum normals);
```

Для объекта ***qobj*** аргумент ***textureCoords*** может принимать значения **GL_TRUE** или **GL_FALSE (**значение по умолчанию**)** и управляет тем, следует ли генерировать для квадрического объекта координаты текстуры. Способ генерации текстурных координат зависит от типа квадрического объекта.

11.2.3 Квадрические примитивы

Следующие функции непосредственно генерируют вершины и другие данные, составляющие квадрический объект. В каждом случае ***qobj*** представляет собой указатель, созданный с помощью **gluNewQuadric()**.

```
void gluSphere (GLUquadricObj* qobj, GLdouble radius, GLint slices, GLint
stacks);
```

Рисует сферу с радиусом равным аргументу ***radius***, центр которой находится в начале координат. Вокруг оси ***z*** сфера состоит из частей, количеством ***slices (***что-то вроде географической долготы**)**, а вдоль ***z –*** из частей количеством ***stacks (***что-то вроде географической широты***).***

Если требуется генерировать координаты текстуры, то координата ***t*** меняется от **0.0** на плоскости ***z=-radius*** до **1.0** на плоскости ***z=radius***, то есть ***t*** линейно увеличивается вдоль линий долготы. В то же время ***s*** изменяется от **0.0** на оси ***+y***, к **0.25** на оси ***+x***, к **0.5** на оси ***–y***, к **0.75** на оси ***–x*** до **1.0** снова на оси ***+y***.

```
void gluCylinder (GLUquadricObj* qobj, GLdouble baseRadius, GLdouble
topRadius,GLdouble height, GLint slices, GLint stacks);
```

Рисует цилиндр, ориентированный вдоль ***z*** оси, с основанием цилиндра на плоскости ***z=0***, а вершиной на плоскости ***z=height***. Также как и сфера, цилиндр разделяется вокруг ***z*** оси на ***slices*** частей, а вдоль ***z*** оси на ***stacks*** частей. ***baseRadius*** задает радиус цилиндра у основания, а ***topRadius*** – радиус в вершине (то есть с помощью этой функции рисуются не только цилиндры в классически геометрическом понимании, но и конусы и части конусов). Если ***topRadius*** установлен в **0.0**, будет сгенерирован конус.

Если требуется генерировать координаты текстуры, то координата ***t*** линейно изменяется от **0.0** на плоскости ***z=0*** до **1.0** на плоскости ***z=height***. Координата ***s*** генерируется также как для сферы.

Замечание: У цилиндра не закрыты ни основание, ни вершина. Диски, представляющие собой основание и вершину, не рисуются.

```
void gluDisk (GLUquadricObj* qobj, GLdouble innerRadius, GLdouble outerRadius,
GLint slices, GLint rings);
```

Рисует диск на плоскости ***z=0***, с радиусом ***outerRadius*** и концентрической дырой в центре радиусом ***innerRadius***. Если ***innerRadius*** равен **0**, никакой дыры в центре нет. Вокруг оси ***z*** диск разделяется на ***slices*** частей (что похоже на нарезку пиццы). Кроме того, в плоскости ***z*** диск разделяется на ***rings*** концентрических кругов.

С учетом ориентации ***+z*** сторона диска считается находящейся снаружи: то есть все генерируемые нормали указывают вдоль положительного направления оси ***z***.

Если следует генерировать координаты текстуры, то они генерируются линейно следующим образом. Если **R=*outerRadius***, то значения ***s*** и ***t*** в точке **(R, 0, 0)** равны **(1, 0.5)**, в точке **(0, R, 0) – (0.5, 1)**, в точке **(-R, 0, 0) – (0, 0.5)**, и, наконец, в точке **(0, -R, 0) – (0.5, 0)**.

```
void gluPartialDisk (GLUquadricObj* qobj, GLdouble innerRadius, GLdouble
outerRadius,
                     GLint slices, GLint rings, GLdouble startAngle, GLdouble
sweepAngle);
```

Рисует часть диска на плоскости ***z=0***. В терминах ***outerRadius***, ***innerRadius***, ***slices*** и ***rings*** часть диска похожа на целый диск. Разница в том, что рисуется только часть диска, начинающаяся с угла ***startAngle*** до угла ***startAngle+sweepAngle*** (где ***startAngle*** и ***sweepAngle*** измеряются в градусах, и угол **0** соответствует положительному направлению оси ***y***, угол **90 –** положительному направлению оси ***x***, **180 –** отрицательному направлению оси ***y***, а **270 –** отрицательному направлению оси ***x***).

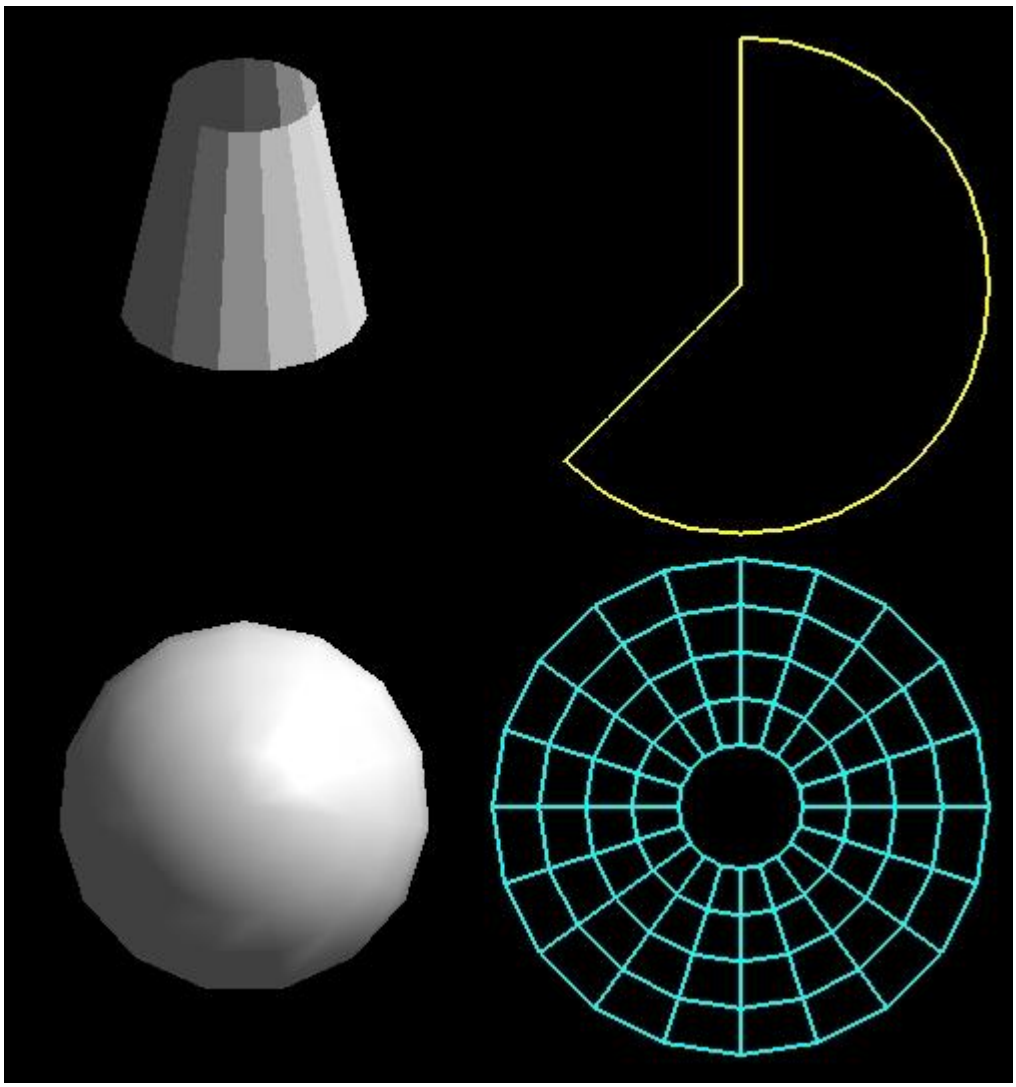
Ориентация и координаты текстуры для части диска обрабатываются так же как для целого.

Замечание: Для всех квадрических объектов в случае необходимости изменить их размер лучше использовать аргументы функций построения квадрических примитивов (*Radius, height и так далее) вместо команды **glScale()**, так как в последнем случае нормали единичной длины должны быть повторно нормализованы. Устанавливайте аргументы *rings* и *stacks* в значения отличные от 1, чтобы

расчеты освещенности производится с большей гранулярностью. Это особенно важно, если материал имеет высокий зеркальный коэффициент.

Пример **11-4** демонстрирует все типы квадрических примитивов, а также различные стили рисования. Результат работы программы приводится на рисунке **11-5.**

Рисунок **11-5.** Квадрические примитивы



Пример **11-4.** Квадрические объекты: файл **quadric.cpp**

```
#include
#include
#include

#ifndef CALLBACK
#define CALLBACK
#endif

GLuint startList;

void CALLBACK errorCallback(GLenum errorCode)
{
      printf("Quadric error: %s\n",gluErrorString(errorCode));
      exit(0);
}
```

```
//Инициализация
void init(void)
{
	GLUquadricObj *qobj;
	GLfloat mat_ambient[]={0.5,0.5,0.5,1.0};
	GLfloat mat_specular[]={1.0,1.0,1.0,1.0};
	GLfloat mat_shininess[]={50.0};
	GLfloat light_position[4]={1.0,1.0,1.0,0.0};
	GLfloat model_ambient[]={0.5,0.5,0.5,1.0};

	glClearColor(0.0,0.0,0.0,0.0);

	glMaterialfv(GL_FRONT,GL_AMBIENT,mat_ambient);
	glMaterialfv(GL_FRONT,GL_SPECULAR,mat_specular);
	glMaterialfv(GL_FRONT,GL_SHININESS,mat_shininess);
	glLightfv(GL_LIGHT0,GL_POSITION,light_position);
	glLightModelfv(GL_LIGHT_MODEL_AMBIENT,model_ambient);

	glEnable(GL_LIGHT0);
	glEnable(GL_DEPTH_TEST);

	//Создать 4 списка каждый с разным квадрик-объектом.
	//Используются различные стили и параметры нормалей
	startList=glGenLists(4);
	qobj=gluNewQuadric();
	gluQuadricCallback(qobj,GLU_ERROR,
			(void(__stdcall*)(void))errorCallback);

	//Плавно закрашенный
	gluQuadricDrawStyle(qobj,GLU_FILL);
	gluQuadricNormals(qobj,GLU_SMOOTH);
	glNewList(startList,GL_COMPILE);
		gluSphere(qobj,0.75,15,10);
	glEndList();

	//Плоско закрашенный
	gluQuadricDrawStyle(qobj,GLU_FILL);
	gluQuadricNormals(qobj,GLU_FLAT);
	glNewList(startList+1,GL_COMPILE);
		gluCylinder(qobj,0.5,0.3,1.0,15,5);
	glEndList();

	//Каркасный
	gluQuadricDrawStyle(qobj,GLU_LINE);
	gluQuadricNormals(qobj,GLU_NONE);
	glNewList(startList+2,GL_COMPILE);
		gluDisk(qobj,0.25,1.0,20,4);
	glEndList();

	//Силуэт
	gluQuadricDrawStyle(qobj,GLU_SILHOUETTE);
	gluQuadricNormals(qobj,GLU_NONE);
	glNewList(startList+3,GL_COMPILE);
		gluPartialDisk(qobj,0.0,1.0,20,4,0.0,225.0);
	glEndList();
}

//Отображение
void display(void)
{
	glClear(GL_COLOR_BUFFER_BIT|GL_DEPTH_BUFFER_BIT);
	glPushMatrix();
```

```
	glEnable(GL_LIGHTING);
	glShadeModel(GL_SMOOTH);
	glTranslatef(-1.0,-1.0,0.0);
	glCallList(startList);

	glShadeModel(GL_FLAT);
	glTranslatef(0.0,2.0,0.0);
	glPushMatrix();
	glRotatef(300.0,1.0,0.0,0.0);
	glCallList(startList+1);
	glPopMatrix();

	glDisable(GL_LIGHTING);
	glColor3f(0.0,1.0,1.0);
	glTranslatef(2.0,-2.0,0.0);
	glCallList(startList+2);

	glColor3f(1.0,1.0,0.0);
	glTranslatef(0.0,2.0,0.0);
	glCallList(startList+3);

	glPopMatrix();
	glFlush();
}

//Изменение размеров окна
void reshape(int w, int h)
{
	glViewport(0,0,(GLsizei) w, (GLsizei) h);
	glMatrixMode(GL_PROJECTION);
	glLoadIdentity();
	if (w<=h)
		glOrtho(-2.5,2.5,-2.5*(GLfloat)h/(GLfloat)w,
			2.5*(GLfloat)h/(GLfloat)w,-10.0,10.0);
	else
		glOrtho(-2.5*(GLfloat)w/(GLfloat)h,
			2.5*(GLfloat)w/(GLfloat)h,-2.5,2.5,-10.0,10.0);
	glMatrixMode(GL_MODELVIEW);
	glLoadIdentity();
}

void keyboard(unsigned char key,int x,int y)
{
	switch(key)
	{
		case 27:
			exit(0);
			break;
	}
}

int main (int argc, char** argv)
{
	glutInit(&argc,argv);
	glutInitDisplayMode(GLUT_SINGLE|GLUT_RGB|GLUT_DEPTH);
	glutInitWindowSize(500,500);
	glutInitWindowPosition(100,100);
	glutCreateWindow("Quadrics Objects");
	init();
	glutDisplayFunc(display);
	glutReshapeFunc(reshape);
	glutKeyboardFunc(keyboard);
```

```
        glutMainLoop();
        return 0;
}
```

Глава 12. Вычислители и NURBS

Дополнительно: На самом низком уровне графическая аппаратура рисует только точки, сегменты линий и полигоны, которые обычно являются треугольниками или четырехугольниками. Плавные кривые и поверхности рисуются с помощью аппроксимаций большим число маленьких сегментов линий или полигонов. Однако множество полезных кривых и поверхностей могут быть математически описаны небольшим числом параметров, например, несколькими *контрольными точками*. Сохранение **16** контрольных точек для поверхности требует значительно меньше места в памяти, чем сохранение **1000** треугольников вместе с вектором нормали для каждой вершины. Кроме того, **1000** треугольников всего лишь аппроксимирует поверхность, а контрольные точки определяют ее истинную форму.

Вычислители **(evaluators)** предоставляют возможность задать точки кривой или поверхности **(**или их частей**)** используя только контрольные точки. Затем кривая или поверхность может быть визуализирована с любой точностью. Вдобавок, вектора нормалей для поверхностей могут быть вычислены автоматически. Точки, сгенерированные вычислителем вы можете использовать по-разному **–** чтобы нарисовать часть точек поверхности, чтобы нарисовать ее каркасный вариант или чтобы отобразить полностью освещенную, закрашенную и даже текстурированную поверхность.

Вы можете использовать вычислители для описания любых полиномиальных или рациональных полиномиальных сплайнов или поверхностей. Допустимое множество включает все сплайны и сплайновые поверхности, используемые в наши дни: **B–** сплайны, **NURBS (Non-UniformRationalB-Spline–** рациональные **B -** сплайны, заданные на неравномерной сетке**)**, кривые и поверхности Безье и сплайны Эрмита. Поскольку вычислители обеспечивают только низкоуровневое описание точек кривой или поверхности, они обычно используются в качестве фундамента библиотеками утилит, предоставляющими программисту высокоуровневый интерфейс. Один из таких высокоуровневых интерфейсов предоставляется механизмом **NURBS** из состава **GLU –** этот механизм инкапсулирует в себе большой объем сложного кода. Большая часть финальной визуализации реализуется с помощью вычислителей, но для некоторых случаев **(**например, для отделки кривых**)** функции **NURBS** используют плоские полигоны.

12.1 Предварительные требования

Вычислители создают кривые и поверхности на базе кривых Безье. Определяющие формулы основных функций приведены в этой главе, однако в ней не приводятся выводы этих формул, также как и все их интересные математические свойства. Если вы хотите использовать вычислители для рисования кривых и поверхностей в другом базисе, вам нужно знать, как конвертировать ваш базис к базису Безье. Кроме того, когда вы визуализируете поверхность Безье или ее часть, вам следует определиться с гранулярностью поверхности. При принятии решения нужно принимать в расчет вечную сделку между качеством **(**малая гранулярность**)** и высокой скоростью. Определение нужной стратегии может быть достаточно сложным **–** слишком сложным, чтобы обсуждаться здесь.

Подобным же образом, в этой книге не обсуждаются все детали, касающиеся **NURBS**. Для программистов, которые уже разбираются в этом предмете в данной главе описывается интерфейс **GLU NURBS** и приводятся примеры его использования. Вы уже

должны знать что такое контрольные точки, узловые последовательности и декорирующие кривые.

Если вам не хватает знаний в одной из описанных областей, попробуйте обратиться к следующим источникам:

- **Farin, Gerald E., *Curves and Surfaces for Computer – Aided Geometric Design, Fourth Edition*. San Diego, CA: Academic Press, 1996.**
- **Farin, Gerald E., NURB *Curves and Surfaces: from Projective Geometry to Practical Use*. Wellesley, MA: A. K. Peters Ltd., 1995.**
- **Farin, Gerald E., editor, *NURBS for Curve and Surface Design*. Society for Industrial and Applied Mathematics, Philadelphia, PA, 1991.**
- **Hoschek, Josef and Dieter Lasserm *Fundamentals of Computer Aided Geometric Design*. Wellesley, MA: A. K. Peters Ltd., 1993.**
- **Piegl, Les and Wayne Tiller, *The NURBS Book*. New York, NY: Springer – Verlag, 1995.**

Замечание: Российским читателям можно порекомендовать книгу Е. В. Шикин, А. И. Плис «Кривые и поверхности на экране компьютера. Руководство по сплайнам для пользователя» -- М.: ДИАЛОГ – МИФИ, 1996.

Замечание: Некоторые термины в этой главе могут иметь несколько иной смысл, чем в других книгах, поскольку между практикующими в этой области не существует четкого соглашения. В общем, смысл терминов в OpenGL более четок и конкретен. Например, вычислители OpenGL всегда используют базис Безье, а в других источниках на вычислители могут ссылаться как на ту же концепцию, но использующую произвольный базис.

12.1 Вычислители

Кривая Безье – это векторная функция одной переменной

$$C(u) = [X(u) \quad Y(u) \quad Z(u)],$$

где ***u*** изменяется в некотором диапазоне (скажем **[*0*, *1*]**). Поверхность Безье – это векторная функция двух переменных

$$S(u,v) = [X(u,v) \quad Y(u,v) \quad Z(u,v)].$$

где обе переменные ***u*** и ***v*** изменяются в некотором диапазоне. Результирующий диапазон не обязательно должен быть трехмерным, как показано здесь. Возможно, вам понадобится двумерный вывод для кривых на плоскости или координат текстуры, или четырехмерный вывод для того, чтобы можно было задавать с помощью него **RGBA** информацию. Даже одномерный вывод имеет смысл при работе с оттенками серого цвета.

Для каждого ***u*** (или ***u*** и ***v***, в случае поверхности), формула **C()** (или **S()**) вычисляет тоску на кривой (или поверхности). Чтобы использовать вычислитель, сначала определите функцию **C()** или **S()**, активизируйте ее, а затем используйте команду **glEvalCoord1()** или **glEvalCoord2()** вместо **glVertex*()**. Таким образом, вершины кривой или поверхности могут использоваться также как и любые другие вершины – например, для формирования точек или линий. Кроме того, другие команды могут генерировать серии вершин, образующих фигуру, равномерно распределенную по ***u*** (или по ***u*** и ***v***). Одномерные и двумерные вычислители похожи, но одномерный случай проще для описания, поэтому начнем с него.

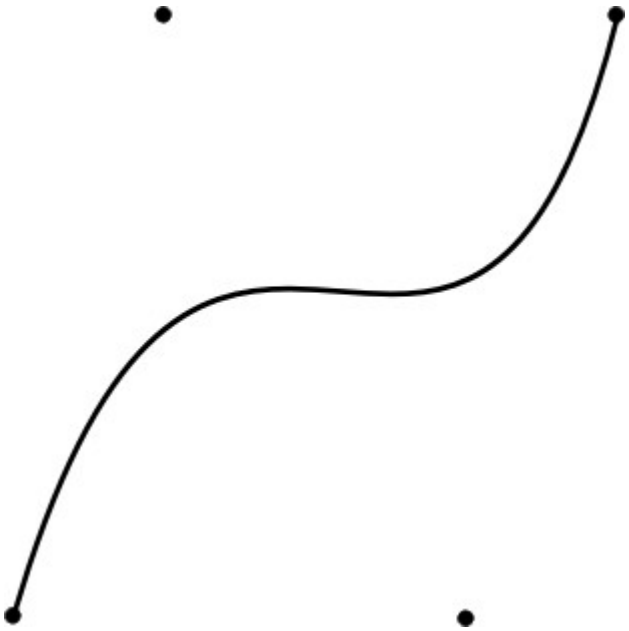
12.1.1 Одномерные вычислители

В данном разделе представлен пример использования одномерного вычислителя для рисования кривой. Далее описаны команды и уравнения, контролирующие вычислитель.

12.1.1.1 Одномерный пример: простая кривая Безье

Программа, представленная в примере **12-1** рисует кубическую кривую Безье с использованием **4** контрольных точек, как показано на рисунке **12-1.**

Рисунок **12-1.** Кривая Безье



Пример **12-1.** Кривая Безье с **4** контрольными точками: файл **bezcurve.cpp**

```
#include

GLfloat ctrlpoints[4][3]={   {-4.0,-4.0,0.0},
                             {-2.0,4.0,0.0},
                             {2.0,-4.0,0.0},
                             {4.0,4.0,0.0}
                         };

void init()
{
      glClearColor(0.0,0.0,0.0,0.0);
      glShadeModel(GL_FLAT);
      glMap1f(GL_MAP1_VERTEX_3,0.0,1.0,3,4,&ctrlpoints[0][0]);
      glEnable(GL_MAP1_VERTEX_3);
}

void display()
{
      int i;

      glClear(GL_COLOR_BUFFER_BIT);
      glColor3f(0.0,1.0,0.0);
      glBegin(GL_LINE_STRIP);
            for(i=0;i<=30;i++)
```

```
                    glEvalCoord1f((GLfloat)i/30.0);
        glEnd();

        //Контрольные точки
        glPointSize(5.0);
        glColor3f(1.0,1.0,0.0);
        glBegin(GL_POINTS);
              for(i=0;i<4;i++)
                    glVertex3fv(&ctrlpoints[i][0]);
        glEnd();
        glFlush();
  }

  void reshape(int w, int h)
  {
        glViewport(0,0,(GLsizei) w, (GLsizei) h);
        glMatrixMode(GL_PROJECTION);
        glLoadIdentity();
        if (w<=h)
              glOrtho(-5.0,5.0,-5.0*(GLfloat)h/(GLfloat)w,
                       5.0*(GLfloat)h/(GLfloat)w,-5.0,5.0);
        else
              glOrtho(-5.0*(GLfloat)w/(GLfloat)h,
                       5.0*(GLfloat)w/(GLfloat)h,-5.0,5.0,-5.0,5.0);
        glMatrixMode(GL_MODELVIEW);
        glLoadIdentity();
  }

  int main(int argc, char** argv)
  {
        glutInit(&argc,argv);
        glutInitDisplayMode(GLUT_SINGLE|GLUT_RGB);
        glutInitWindowSize(500,500);
        glutInitWindowPosition(100,100);
        glutCreateWindow("Bezier Curve with Four Control Points");
        init();
        glutDisplayFunc(display);
        glutReshapeFunc(reshape);
        glutMainLoop();
        return 0;
  }
```

Кубическая кривая Безье описывается **4** точками**,** которые фигурируют в примере в виде массива ***ctrlpoints[][]*****.** Этот массив является одним из аргументов для команды **glMap1f().** Полный список аргументов для этой команды следующий**:**

GL_MAP1_VERTEX3	Предоставлены трехмерные точки и должны быть сгенерированы трехмерные вершины
0.0	Нижняя граница для ***u***
1.0	Верхняя граница для ***u***
3	Число величин с плавающей точкой между началом данных для одной контрольной точки и началом данных для другой в массиве
4	Порядок сплайна**,** равный его степени плюс **1.** В данном случае степень равна **3 (**поскольку это кубический сплайн**)**
&ctrlpoints[][]	Указатель на данные первой контрольной точки

Обратите внимание на то**,** что второй и третий аргументы команды управляют параметризацией кривой **–** в то время как ***u***меняется от **0.0** до **1.0,** кривая проходит от своего начала до своего конца**.** Вызов команды **glEnable()** активизирует одномерный вычислитель для трехмерных вершин**.**

Сама кривая рисуется в функции **display()** между вызовами **glBegin()** и **glEnd()**. поскольку вычислитель активизирован, вызов команды **glEvalCoord1f()** аналогичен выполнению команды **glVertex()** с координатами вершины на кривой, соответствующими заданному параметру ***u***.

12.1.1.2 Определение и вычисление одномерного вычислителя

Многочлен Бернштейна степени ***n***(или порядка ***n+1***) вычисляется по формуле

$$B_i^n(u) = \binom{n}{i} u^i (1-u)^{n-i}$$

Если P_i представляет набор контрольных точек (одно-, двух-, трех- или даже четырехмерных), то уравнение

$$C(u) = \sum_{i=0}^{n} B_i^n(u) P_i$$

представляет кривую Безье в процессе изменения ***u***от ***0.0*** до ***1.0***. Чтобы представить ту же кривую, позволяя **u** меняться от u_1 до u_2, а не между ***0.0*** и ***1.0***, вычислите

$$C\left(\frac{u-u_1}{u_2-u_1}\right).$$

Команда **glMap1()** определяет одномерный вычислитель, использующий данные уравнения.

```
void glMap1{fd} (GLenum target, TYPE u1, TYPE u2, GLint stride, GLint order,
const TYPE *points);
```

Определяет одномерный вычислитель. Аргумент ***target*** определяет, что именно задается контрольными точками (смотрите таблицу **12-1**) и, как следствие, сколько величин должно быть задано в аргументе ***points***. Точки могут представлять вершины, цветовые данные **RGBA**, вектора нормалей или координаты текстуры. Например, если задать для **target** значение **GL_MAP1_COLOR_4**, вычислитель генерирует цветовые данные в четырехмерном **RGBA** пространстве вдоль кривой. Величины параметров, перечисленные в таблице **12-1**, также используются для активизации конкретного вычислителя до его вызова. Чтобы активизировать или деактивировать нужный вычислитель, передайте соответствующий аргумент командам **glEnable()** или **glDisable()** соответственно.

Следующие два параметра **glMap1*() – *u1*** и ***u2*** задают диапазон изменения переменной ***u***. Аргумент ***stride***– это количество величин с плавающей точкой одинарной или двойной точности в каждом блоке хранилища, то есть это величина смещения между началом предыдущей контрольной точки и началом следующей.

Аргумент ***order*** – это степень кривой плюс **1**, и это число должно согласовываться с количеством поставляемых контрольных точек. Аргумент ***points*** должен указывать на первую координату первой контрольной точки.

Таблица **12-1.** Типы контрольных точек для **glMap1*()**

Параметр	Смысл параметра
GL_MAP1_VERTEX_3	координаты вершины ***x***, ***y***и ***z***
GL_MAP1_VERTEX_4	координаты вершины ***x***, ***y***, ***z*** и ***w***
GL_MAP1_INDEX	цветовой индекс
GL_MAP1_COLOR_4	**R, G, B, A**
GL_MAP1_NORMAL	координаты нормали
GL_MAP1_TEXTURE_COORD_1	координата текстуры ***s***
GL_MAP1_TEXTURE_COORD_2	координаты текстуры ***s*** и ***t***
GL_MAP1_TEXTURE_COORD_3	координаты текстуры ***s***, ***t***и ***r***
GL_MAP1_TEXTURE_COORD_4	координаты текстуры ***s***, ***t***, ***r*** и ***q***

Одновременно может быть включено более одного вычислителя. Если у вас определены, например, два вычислителя **GL_MAP1_VERTEX_3** и **GL_MAP1_COLOR_4,** то вызов команды **glEvalCoord1()** сгенерирует и позицию и цвет. Одновременно может быть активизирован только один вершинный вычислитель, даже если определены оба. Также одновременно может быть включен только один текстурный вычислитель. В других случаях, однако, вычислители могут использоваться для генерирования любой комбинации вершин, нормалей, цветов и координат текстуры. Если вы определите и активизируете два или более вычислителей одного и того же типа, будет использован тот из них, в котором наибольшее число измерений.

Для вычисления заданной и активизированной одномерной таблицы используйте команду **glEvalCoord1*()**.

```
void glEvalCoord1{fd} (TYPE u);
void glEvalCoord1{fd}v (TYPE* u);
```

Вызывает к исполнению процесс вычисления одномерных таблиц. Аргумент ***u***задает одномерную координату на кривой.

Обращения к **glEvalCoord*()** не используют текущие величины цвета, цветового индекса, вектора нормали и координат текстуры. **glEvalCoord*()** оставляет эти величины неизмененными.

12.1.1.3 Определение одномерных доменных координат на кривой с равными промежутками

Вы можете использовать **glEvalCoord1()** с любыми значениями ***u***, но наиболее частой практикой является использование сетки величин с равными промежутками между ними, как показано ранее в примере **12-1.** Чтобы получить такие величины, определите сетку командой **glMapGrid1*()** и примените ее, используя **glEvalMesh1().**

```
void glMapGrid1{fd} (GLint n, TYPE u1, TYPE u2);
```

Определяет сетку величин изменяющихся от ***u1*** до ***u2*** за ***n*** равных шагов.

```
void glEvalMesh1 (GLenum mode, GLint p1, GLint p2);
```

Применяет определенную в текущий момент одномерную сетку величин ко всем активизированным вычислителям. Аргумент ***mode*** может принимать значения **GL_POINT** или **GL_LINE** в зависимости от того, как вы хотите изобразить кривую – точками вдоль нее или соединяющимися линиями. Вызов данной команды полностью эквивалентен

вызовам **glEvalCoord1()** для каждого шага от ***p1*** до ***p2*** включительно, где **0<=p1, p2<=n.** С точки зрения кода, это эквивалентно следующему фрагменту:

```
glBegin(GL_POINTS);     /* или glBegin(GL_LINES); */
      for(i=p1;i<=p2;i++)
            glEvalCoord1(u1+i*(u2-u1)/n);
glEnd();
```

за исключением того, что если ***i=0*** или ***i=n***, **glEvalCoord1()** вызывается непосредственно с параметрами ***u1*** или ***u2***.

12.1.2 Двумерные вычислители

Двумерный случай практически идентичен одномерному за тем исключением, что все команды должны принимать в расчет **2** параметра ***u*** и ***v***. Точки, цвета, нормали и координаты текстуры должны поставляться по поверхности, а не по кривой. Математически описание поверхности Безье задается в виде

$$S(u,v) = \sum_{i=0}^{n} \sum_{j=0}^{m} B_i^n(u) B_j^m(v) P_{ij}$$

где величины P_{ij} представляют собой набор из ***m*n*** контрольных точек, а функции **B –** это те же многочлены Бернштейна, что и в одномерном случае. Как и раньше величины P_{ij} могут являться вершинами, нормалями, цветами или координатами текстуры.

Процедура использования двумерного вычислителя идентична одномерному случаю.

1. Определите вычислитель (или вычислители) с помощью **glMap2*()**.
2. Активизируйте их, передав нужную величину (или величины) команде **glEnable()**.
3. Вызовите их к исполнению либо с помощью команд **glEvalCoord2()** между **glBegin()** и **glEnd()**, либо определив и применив сетку величин с помощью команд **glMapGrid2()** и **glEvalMesh2()**.

12.1.2.1 Определение и вычисление двумерного вычислителя

Используйте **glMap2*()** и **glEvalCoord2*()** для определения и выполнения двумерного вычислителя.

```
void glMap2{fd} (GLenum target, TYPE u1, TYPE u2, GLint ustride, GLint uorder,
TYPE v1, TYPE v2,
                 GLint vstride, GLint vorder, TYPE* points);
```

Параметр ***target*** может принимать любые значения из таблицы **12-1**, но в данном случае **MAP1** нужно изменить на **MAP2** в именах всех параметров. Как и раньше, эти же значения используются в командах **glEnable()** и **glDisable()** для активизации или деактивации нужных вычислителей. Минимальное и максимальное значения аргументов ***u*** и ***v***, задаются в виде аргументов ***u1***, ***u2***, ***v1*** и ***v2***, соответственно. Аргументы ***ustride*** и ***vstride*** задают количество чисел однократной или двойной точности между независимыми установками величин ***u*** и ***v***, позволяя пользователю выбирать подрегион контрольных точек из намного большего по размеру массива. Например, если данные заданы в форме

```
GLfloat ctrlpoints[100][100][3];
```

и вы хотите использовать только подмножество точек размером ***4x4***, начинающееся с точки ***ctrlpoints[20][30]***, установите ***ustride***в ***100*3***, а ***vstride*** в ***3***. Аргумент ***points*** в этом случае должен быть задан как ***&ctrlpoints[20][30][0]***. Наконец, аргументы, определяющие порядок – ***uorder*** и ***vorder***, могут иметь разные значения, позволяя создавать, например, поверхности квадратные в одном направлении и кубические в другом.

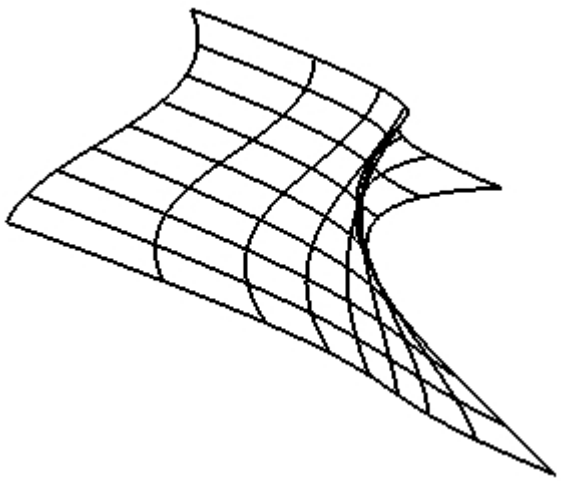
```
void glEvalCoord2{fd} (TYPE u, TYPE v);
void glEvalCoord2{fd}v (TYPE* u, TYPE* v);
```

Вызывает к исполнению заданные и активизированные двумерные вычислители. Аргументы ***u***и ***v*** являются величинами (или указателями на величины в случае векторной версии команды) доменных координат (координат на поверхности или кривой). Если активизирован один из вершинных вычислителей (**GL_MAP2_VERTEX_3** или **GL_MAP2_VERTEX_4**), то координаты нормалей к поверхности вычисляются автоматически. Если активизирован режим автоматической генерации нормалей (с помощью аргумента **GL_AUTO_NORMAL** в команде **glEnable()**), эта нормаль ассоциируется с вычисленной вершиной. Если этот режим выключен, для вычисления нормали используется текущий активизированный вычислитель. Если же таковой отсутствует, используется текущий вектор нормали.

12.1.2.2 Двумерный пример: поверхность Безье

Пример **12-2** отображает каркасную поверхность Безье, показанную на рисунке **12-2**, с использованием вычислителя. В этом примере поверхность рисуется в виде **9** изогнутых линий в каждом направлении. Каждая линия состоит из **30** сегментов. Для получения цельной программы, добавьте функции **reshape()** и **main()** из примера **12-1**.

Рисунок **12-2.** Поверхность Безье



Пример **12-2.** Поверхность Безье: файл **bezsurf.cpp**

```
GLfloat ctrlpoints[4][4][3]={
      {{-1.5,-1.5,4.0},{-0.5,-1.5,2.0},{0.5,-1.5,-1.0},{1.5,-
1.5,2.0}},
      {{-1.5,-0.5,1.0},{-0.5,-0.5,3.0},{0.5,-0.5,0.0},{1.5,-0.5,-
1.0}},
      {{-1.5,0.5,4.0},{-0.5,0.5,0.0},{0.5,0.5,3.0},{1.5,0.5,4.0}},
      {{-1.5,1.5,-2.0},{-0.5,1.5,-2.0},{0.5,1.5,0.0},{1.5,1.5,-1.0}}
};
```

```
void display()
{
      int i,j;

      glClear(GL_COLOR_BUFFER_BIT|GL_DEPTH_BUFFER_BIT);
      glColor3f(0.0,0.0,0.0);
      glLoadIdentity();
      glRotatef(85.0,1.0,1.0,1.0);
      for(j=0;j<=8;j++)
      {
            glBegin(GL_LINE_STRIP);
                  for(i=0;i<=30;i++)

glEvalCoord2f((GLfloat)i/30.0,(GLfloat)j/8.0);
            glEnd();
            glBegin(GL_LINE_STRIP);
                  for(i=0;i<=30;i++)

glEvalCoord2f((GLfloat)j/8.0,(GLfloat)i/30.0);
            glEnd();
      }
      glFlush();
}

void init()
{
      glClearColor(1.0,1.0,1.0,0.0);
      glLineWidth(2.0);

glMap2f(GL_MAP2_VERTEX_3,0,1,3,4,0,1,12,4,&ctrlpoints[0][0][0]);
      glEnable(GL_MAP2_VERTEX_3);
      glMapGrid2f(20,0.0,1.0,20,0.0,1.0);
      glEnable(GL_DEPTH_TEST);
      glShadeModel(GL_FLAT);
}
```

12.1.2.3 Определение двумерных доменных координат на поверхности с равными промежутками

В двух измерениях команды **glMapGrid2*()** и **glEvalMesh2()** используются так же как их одномерные версии**,** за тем исключением**,** что должна быть задана информация и о ***u***, и о ***v*.**

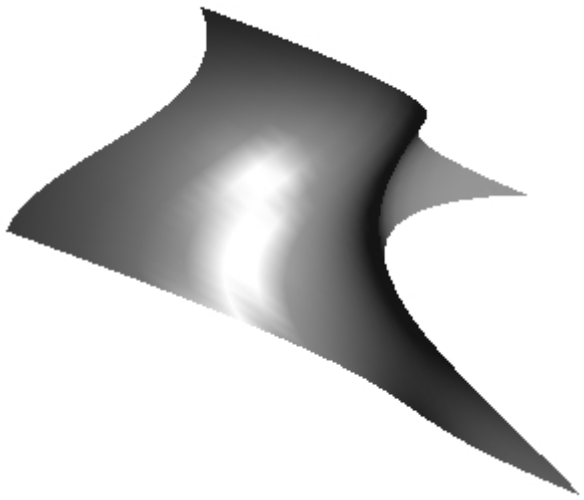
```
void glMapGrid2{fd} (GLint nu, TYPE u1, TYPE u2, GLint vn, TYPE v1, TYPE v2);
void glEvalMesh2 (GLenum mode, GLint i1, GLint i2, GLint j1, GLint j2);
```

Определяют двумерную сетку величин**,** проходящих от ***u1*** до ***u2*** за ***nu*** шагов с равными промежутками и от ***v1*** до ***v2*** за ***nv*** с равными промежутками **(glMapGrid2*())** и затем применяют эту сетку ко всем активизированным вычислителям **(glEvalMesh2()).** Единственное существенное отличие от одномерного случая заключается в том**,** что аргумент ***mode*** команды **glEvalMesh2()** помимо **GL_POINT** и **GL_LINE** может принимать и значение **GL_FILL. GL_FILL** генерирует закрашенные полигоны с помощью четырехугольников**.** Если говорить точно**,** вызов **glEvalMesh2()** практически эквивалентен одному из трех следующих блоков кода**. («**Практически**»** потому**,** что**,** когда ***i=nu*** или ***j=nv*****,** параметры равны ***u2*** или ***v2*****,** а не ***u1+nu*(u2-u1)/nu*** или ***v1+nv*(v2-v1)/nv–*** эти числа могут различаться из-за ошибок округления**.)**

```
glBegin(GL_POINTS);     /* режим GL_POINT */
for(i=nu1;i<=nu2;i++)
      for(j=nv1;j<=nv2;j++)
            glEvalCoord2(u1+i*(u2-u1)/nu, v1+j*(v2-v1)/nv);
glEnd();

или

for(i=nu1;i<=nu2;i++)   /* режим GL_LINE */
{
      glBegin(GL_LINES);
            for(j=nv1;j<=nv2;j++)
                  glEvalCoord2(u1+i*(u2-u1)/nu, v1+j*(v2-v1)/nv);
      glEnd();
}
for(j=nv1;j<=nv2;j++)
{
      glBegin(GL_LINES);
            for(i=nu1;i<=nu2;i++)
                  glEvalCoord2(u1+i*(u2-u1)/nu, v1+j*(v2-v1)/nv);
      glEnd();
}

или

for(i=nu1;i<=nu2;i++)   /* режим GL_FILL */
{
      glBegin(GL_QUAD_STRIP);
            for(j=nv1;j<=nv2;j++)
                  glEvalCoord2(u1+i*(u2-u1)/nu, v1+j*(v2-v1)/nv);
                  glEvalCoord2(u1+(i+1)*(u2-u1)/nu, v1+j*(v2-v1)/nv);
      glEnd();
}
```

Пример **12-3** показывает отличия**,** которые нужно сделать в примере **12-2,** чтобы нарисовать ту же поверхность**,** но с применением **glMapGrid2()** и **glEvalMesh2()** для разделения квадрата доменных координат на сетку величин размерностью ***8x8*****.** Пример **12-3** также добавляет освещение и закраску**,** как показано на рисунке **12-3.**

Рисунок **12-3.** Освещенная и закрашенная поверхность Безье**,** нарисованная по сетке доменных координат



Пример **12-3.** Освещенная и закрашенная поверхность Безье**,** нарисованная по сетке доменных координат**:** файл **bezmesh.cpp**

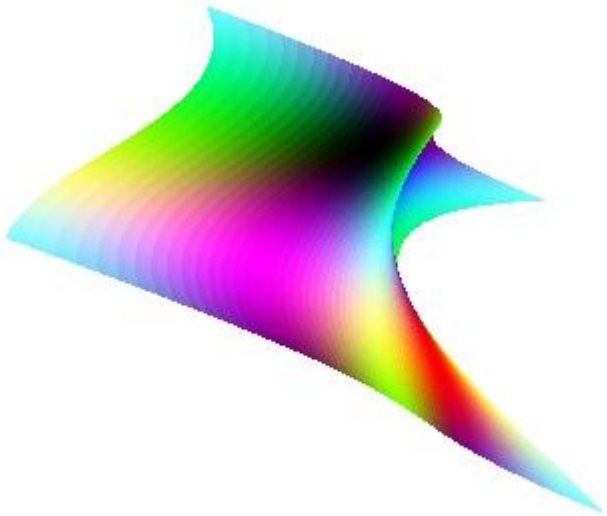
```
void display()
{
      glClear(GL_COLOR_BUFFER_BIT|GL_DEPTH_BUFFER_BIT);
      glLoadIdentity();
      glRotatef(85.0,1.0,1.0,1.0);
      glEvalMesh2(GL_FILL,0,20,0,20);
      glFlush();
}

void init()
{
      glClearColor(1.0,1.0,1.0,0.0);

glMap2f(GL_MAP2_VERTEX_3,0,1,3,4,0,1,12,4,&ctrlpoints[0][0][0]);
      glEnable(GL_MAP2_VERTEX_3);
      glMapGrid2f(20,0.0,1.0,20,0.0,1.0);
      glEnable(GL_DEPTH_TEST);
      glEnable(GL_AUTO_NORMAL);

      //Источники света
      GLfloat ambient[]={0.2,0.2,0.2,1.0};
      GLfloat position[]={0.0,0.0,2.0,1.0};
      GLfloat mat_diffuse[]={0.6,0.6,0.6,1.0};
      GLfloat mat_specular[]={1.0,1.0,1.0,1.0};
      GLfloat mat_shininess[]={50.0};

      glEnable(GL_LIGHTING);
      glEnable(GL_LIGHT0);
      glLightfv(GL_LIGHT0,GL_AMBIENT,ambient);
      glLightfv(GL_LIGHT0,GL_POSITION,position);
      glMaterialfv(GL_FRONT,GL_DIFFUSE,mat_diffuse);
      glMaterialfv(GL_FRONT,GL_SPECULAR,mat_specular);
      glMaterialfv(GL_FRONT,GL_SHININESS,mat_shininess);
}
```

12.1.3 Использование вычислителей для текстур

Пример **12-3** активизирует одновременно **2** вычислителя**:** первый генерирует трехмерные точки на той же поверхности Безье**,** что и пример **12-3,** а второй генерирует координаты текстуры**.** В данном случае в качестве координат текстуры выступают те же значения ***u***и ***v***, которые используются для вычисления вершин**,** но для того**,** чтобы применить их к поверхности**,** требуется определить отдельный вычислитель**.**

Второй вычислитель определяется на квадрате с углами в точках **(0,0), (0, 1), (1,0)** и **(1,1);** он генерирует **(0,0)** в углу **(0,0), (0,1)** в углу **(0,1)** и так далее**.** Поскольку он имеет порядок **2 (**являясь линейным **–** первая степень плюс **1),** вычисление этой текстуры в точке **(*u*,*v*)** дает координаты текстуры **(*s*,*t*).** Он активизируется в то же время**,** что и вершинный вычислитель**,** таким образом**,** на рисуемую поверхность воздействуют оба**,** что можно увидеть на рисунке **12-4.** Если вы хотите**,** чтобы текстура повторилась **3** раза**,** измените каждую **1.0** в массиве ***texpts[ ][ ][ ]*.**

Рисунок **12-4.** Текстурированная поверхность Безье



Пример **12-4.** Использование вычислителей для текстур**:** файл **texturesurf.cpp**

```
#include
#include

GLfloat ctrlpoints[4][4][3]={
      {{-1.5,-1.5,4.0},{-0.5,-1.5,2.0},{0.5,-1.5,-1.0},{1.5,-
1.5,2.0}},
      {{-1.5,-0.5,1.0},{-0.5,-0.5,3.0},{0.5,-0.5,0.0},{1.5,-0.5,-
1.0}},
      {{-1.5,0.5,4.0},{-0.5,0.5,0.0},{0.5,0.5,3.0},{1.5,0.5,4.0}},
      {{-1.5,1.5,-2.0},{-0.5,1.5,-2.0},{0.5,1.5,0.0},{1.5,1.5,-1.0}}
};

GLfloat
texpts[2][2][2]={{{0.0,0.0},{0.0,1.0}},{{1.0,0.0},{1.0,1.0}}};

void display()
{
      glClear(GL_COLOR_BUFFER_BIT|GL_DEPTH_BUFFER_BIT);
      glColor3f(1.0,1.0,1.0);
      glEvalMesh2(GL_FILL,0,20,0,20);
      glFlush();
}

#define imageWidth 64
#define imageHeight 64
GLubyte image[3*imageWidth*imageHeight];

void makeImage()
{
      int i,j;
      float ti,tj;

      for(i=0;i<="h)" glortho(-4.0,4.0,-4.0*(glfloat)h (glfloat)w,
4.0*(glfloat)h (glfloat)w,-4.0,4.0); else glOrtho(-4.0*(GLfloat)w
(GLfloat)h, 4.0*(GLfloat)w (GLfloat)h,-4.0,4.0,-4.0,4.0);
glMatrixMode(GL_MODELVIEW); glLoadIdentity();
glRotatef(85.0,1.0,1.0,1.0); } int main(int argc, char** argv) {
glutInit(&argc,argv);
glutInitDisplayMode(GLUT_SINGLE|GLUT_RGB|GLUT_DEPTH);
glutInitWindowSize(500,500); glutInitWindowPosition(100,100);
```

```
glutCreateWindow(?Using Evaluators for Textures?); init();
glutDisplayFunc(display); glutReshapeFunc(reshape); glutMainLoop();
return 0; }< pre>
```

12.2 Интерфейс GLU NURBS

Хотя единственными непосредственно доступными в **OpenGL** примитивами для рисования кривых и поверхностей являются вычислители, и даже, несмотря на то, что они могут быть весьма эффективно реализованы на аппаратном уровне, приложения часто обращаются к ним через высокоуровневые библиотеки. Библиотека утилит **GLU** предоставляет интерфейс **NURBS**, построенный поверх команд **OpenGL** для работы с вычислителями.

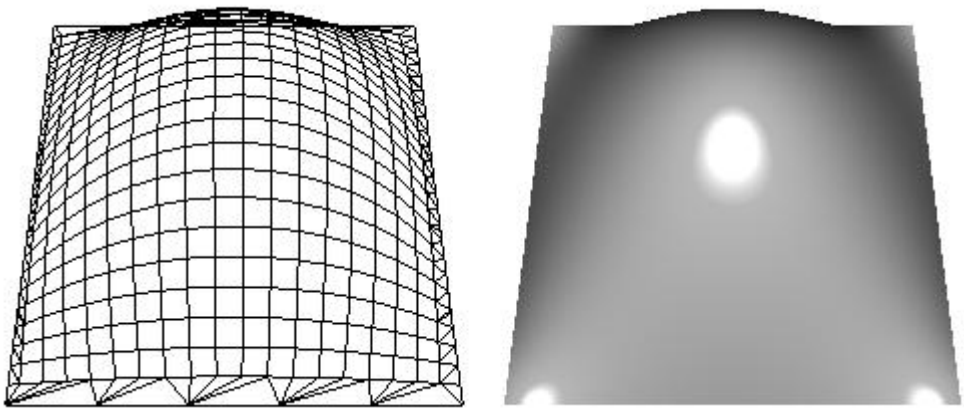
12.2.1Простой пример NURBS

Если вы разбираетесь в **NURBS**, написать код для манипулирования **NURBS** в **OpenGL** относительно просто, даже если вы хотите использовать освещение или текстурирование. Для того, чтобы нарисовать кривую или поверхность **NURBS** выполните следующие шаги.

1. Если вы намереваетесь использовать освещение на поверхности **NURBS**, вызовите **glEnable()** с аргументом **GL_AUTO_NORMAL** для активизации режима автоматического вычисления нормалей **(**вы также можете вычислить свои собственные**).**
2. Используйте функцию **gluNewNurbsRenderer()** для создания нового объекта **NURBS** и получения указателя на него. На этот объект вы будете ссылаться при построении кривой или поверхности **NURBS.**
3. Если хотите, вызовите **gluNurbsProperty()** для установки значений различных свойств, например, максимального размера линий или полигонов, используемых для визуализации кривой или поверхности. **gluNurbsProperty()** также позволяет активизировать режим, в котором тесселированные геометрические данные могут быть получены через интерфейс возвратно – вызываемых функций.
4. Если вы хотите принимать уведомления об ошибках, вызовите **gluNurbsCallback(). (**Проверка ошибок может снизить быстродействие программы, но, тем не менее, ее наличие настойчиво рекомендуется.**)** **gluNurbsCallback()** также позволяет задать возвратные функции, которые будут вызываться для извлечения тесселированных геометрических данных.
5. Начните описание вашей кривой или поверхности, вызвав **gluBeginCurve()** или **gluBeginSurface().**
6. Сгенерируйте и визуализируйте вашу кривую или поверхность. Вы должны хотя бы один раз вызвать функцию **gluNurbsCurve()** или **gluNurbsSurface(),** передавая им контрольные точки **(**рациональные или нерациональные**),** узловые последовательности и порядок полиномиальной базисной функции для вашего объекта **NURBS.** Для указания нормалей и*/*или координат текстуры вам могут понадобиться дополнительные вызовы этих функций.
7. Вызовите **gluEndCurve()** или **gluEndSurface(),** завершая описание кривой или поверхности.

Пример **12-5** визуализирует поверхность **NURBS** в форме симметричного холма с контрольными точками, изменяющимися от ***-3.0*** до ***3.0***. Базисная функция представляет собой кубический **B –** сплайн, но узловая последовательность является неравномерной, заставляя функцию вести себя в каждом направлении как кривая Безье. Поверхность освещена, имеет темно-серое диффузное отражение и белый зеркальный блик. Поверхность в каркасном и закрашенном виде можно увидеть на рисунке **12-5.**

Рисунок **12-5.** Поверхность **NURBS**



Пример **12-5.** Поверхность **NURBS:** файл **surface.cpp**

```
#include
#include

#ifndef CALLBACK
#define CALLBACK
#endif

GLfloat ctrlpoints[4][4][3];
int showPoints=0;
GLUnurbsObj *theNurb;

void init_surface()
{
      int u,v;
      for (u=0;u<4;u++)
      {
            for(v=0;v<4;v++)
            {
                  ctrlpoints[u][v][0]=2.0*((GLfloat)u-1.5);
                  ctrlpoints[u][v][1]=2.0*((GLfloat)v-1.5);

                  if((u==1||u==2)&&(v==1||v==2))
                        ctrlpoints[u][v][2]=3.0;
                  else
                        ctrlpoints[u][v][2]=-3.0;
            }
      }
}

void CALLBACK nurbsError(GLenum errorCode)
{
      char message[100];

      sprintf(message,"NURBS error: %s\n",gluErrorString(errorCode));
      MessageBox(NULL,message,"NURBS surface",MB_OK);
      exit(0);
}

void display()
{
      GLfloat knots[8]={0.0,0.0,0.0,0.0,1.0,1.0,1.0,1.0};
      int i,j;

      glClear(GL_COLOR_BUFFER_BIT|GL_DEPTH_BUFFER_BIT);
```

```
	glPushMatrix();
	glRotatef(330.0,1.0,0.0,0.0);
	glScalef(0.5,0.5,0.5);

	gluBeginSurface(theNurb);
	gluNurbsSurface(theNurb,8,knots,8,knots,4*3,3,
			&ctrlpoints[0][0][0],4,4,GL_MAP2_VERTEX_3);
	gluEndSurface(theNurb);

	if(showPoints)
	{
		glPointSize(5.0);
		glDisable(GL_LIGHTING);
		glColor3f(1.0,1.0,0.0);
		glBegin(GL_POINTS);
			for(i=0;i<4;i++)
				for(j=0;j<4;j++)
					glVertex3fv(&ctrlpoints[i][j][0]);
		glEnd();
		glEnable(GL_LIGHTING);
	}
	glPopMatrix();
	glFlush();
}

void init()
{
	GLfloat mat_diffuse[]={0.7,0.7,0.7,1.0};
	GLfloat mat_specular[]={1.0,1.0,1.0,1.0};
	GLfloat mat_shininess[]={100.0};

	glClearColor(0.0,0.0,0.0,0.0);
	glMaterialfv(GL_FRONT,GL_DIFFUSE,mat_diffuse);
	glMaterialfv(GL_FRONT,GL_SPECULAR,mat_specular);
	glMaterialfv(GL_FRONT,GL_SHININESS,mat_shininess);

	glEnable(GL_LIGHTING);
	glEnable(GL_LIGHT0);
	glEnable(GL_DEPTH_TEST);
	glEnable(GL_AUTO_NORMAL);
	glEnable(GL_NORMALIZE);

	init_surface();

	theNurb=gluNewNurbsRenderer();
	gluNurbsProperty(theNurb,GLU_SAMPLING_TOLERANCE,2.0);
	gluNurbsProperty(theNurb,GLU_DISPLAY_MODE,GLU_FILL);
	gluNurbsCallback(theNurb,GLU_ERROR,(void(__stdcall
*)(void))nurbsError);
}

void reshape(int w, int h)
{
	glViewport(0,0,(GLsizei) w, (GLsizei) h);
	glMatrixMode(GL_PROJECTION);
	glLoadIdentity();
	gluPerspective(45.0,(GLdouble)w/(GLdouble)h,3.0,8.0);
	glMatrixMode(GL_MODELVIEW);
	glLoadIdentity();
	glTranslatef(0.0,0.0,-5.0);
}

void keyboard(unsigned char key,int x,int y)
```

```
{
      switch(key)
      {
            case 'c':
            case 'C':
                  showPoints=!showPoints;
                  glutPostRedisplay();
                  break;
      }
}

int main(int argc, char** argv)
{
      glutInit(&argc,argv);
      glutInitDisplayMode(GLUT_SINGLE|GLUT_RGB|GLUT_DEPTH);
      glutInitWindowSize(500,500);
      glutInitWindowPosition(100,100);
      glutCreateWindow("NURBS surface");
      init();
      glutDisplayFunc(display);
      glutReshapeFunc(reshape);
      glutKeyboardFunc(keyboard);
      glutMainLoop();
      return 0;
}
```

12.2.2 Управление объектом NURBS

Как показано в примере **12-5,** функция **gluNewNurbsRenderer()** возвращает новый объект **NURBS,** чей тип определен в виде указателя на структуру **GLUnurbsObj.** Вы должны создать этот объект до вызова любой функции, связанной с **NURBS.** Когда вы завершите работу с объектом **NURBS,** вы можете использовать функцию **gluDeleteNurbsRenderer()** для освобождения всей используемой памяти.

```
GLUnurbsObj* gluNewNurbsRenderer (void);
```

Создает новый объект **NURBS – *nobj*** и возвращает указатель на него или ноль, если **OpenGL** не может выделить память для нового объекта **NURBS.**

```
void gluDeleteNurbsRenderer (GLUnurbsObj* nobj);
```

Уничтожает объект **NURBS *nobj*.**

12.2.2.1 Управление свойствами визуализации NURBS

Набор свойств, ассоциированных с объектом **NURBS,** влияет на то, как визуализируется объект. Эти свойства задают, как визуализируется поверхность (например, закрашенной или каркасной), должны ли тесселированные вершины отображаться или возвращаться в программу, а также точность тесселяции.

```
void gluNurbsProperty (GLUnurbsObj* nobj, GLenum property, GLfloat value);
```

Управляет атрибутами **NURBS** объекта ***nobj*.** Аргумент ***property*** задает конфигурируемое свойство и может принимать значения **GLU_DISPLAY_MODE, GLU_NURBS_MODE, GLU_CULLING, GLU_SAMPLING_METHOD, GLU_SAMPLING_TOLERANCE, GLU_PARAMETRIC_TOLERANCE, GLU_U_STEP, GLU_V_STEP** или

GLU_AUTO_LOAD_MATRIX. Аргумент ***value*** задает величину, в которую должно быть установлено свойство.

Для свойства **GLU_DISPLAY_MODE**, значение по умолчанию **– GLU_FILL,** в результате чего поверхность визуализируется с помощью полигонов. Если для свойства используется значение **GLU_OUTLINE_POLYGON,** тесселяцией создаются и отображаются только полигоны, окантовывающие поверхность. Если же используется **GLU_OUTLINE_PATCH,** визуализируется только полигон, обозначающий участок, занимаемый поверхностью и декорирующие кривые.

Свойство **GLU_NURBS_MODE** управляет тем, должны ли тесселированные вершины выводиться на экран **(**значение по умолчанию, **GLU_NURBS_RENDERER)** или тесселированные вершины должны быть возвращены в программу через интерфейс возвратно **–** вызываемых функций **(**если значение свойства **– GLU_NURBS_TESSELATOR).**

Параметр **GLU_CULLING** может значительно повысить быстродействие, если задать ему значение **GL_TRUE (**значение по умолчанию **– GL_FALSE).** В этом случае, расчет поверхности не будет производиться, если она целиком выпадает из объема видимости.

Поскольку объект **NURBS** визуализируется в виде примитивов, он строится по различным величинам своих параметров **(*u*** и ***v*)** и разбивается на небольшие сегменты линий или полигоны. Если аргумент ***property*** равен **GLU_SAMPLING_METHOD**, ***value*** может принимать значения **GLU_PATH_LENGTH (**значение по умолчанию**),** **GLU_PARAMETRIC_ERROR, GL_DOMAIN_DISTANCE, GLU_OBJECT_PATH_LENGTH** или **GLU_OBJECT_PARAMETRIC_ERROR,** что определяет способ тесселяции поверхности. Когда ***value*** установлено в значение **GLU_PATH_LENGTH,** поверхность визуализируется таким образом, что максимальная длина ребер полигонов в пикселях не больше, чем значение, заданное для свойства **GLU_SAMPLING_TOLERANCE**. Когда ***value*** установлено в **GLU_PARAMETRIC_ERROR,** величина, заданная для свойства **GLU_PARAMETRIC_TOLERANCE** представляет максимально допустимое расстояние в пикселях между аппроксимирующими полигонами и реальной поверхностью. Значение ***value*** равное **GLU_OBJECT_PATH_LENGTH** похоже на **GLU_PATH_LENGTH** за тем исключением, что величина, заданная для **GLU_SAMPLING_TOLERANCE** считается измеряемой в объектном пространстве, а не в пикселях. Точно так же **GLU_OBJECT_PARAMETRIC_ERROR** отличается от **GLU_PARAMETRIC_ERROR** тем, что значение **GLU_SAMPLING_TOLERANCE** измеряется в объектном пространстве, а не в пикселях.

Когда ***value*** установлено в значение **GLU_DOMAIN_DISTANCE,** приложение задает в параметрических координатах, сколько точек должно быть взято на единицу длины в направлениях ***u***и ***v***с помощью величин **GLU_U_STEP** и **GLU_V_STEP.**

Если ***property*** установлено в **GLU_SAMPLING_TOLERANCE** и режим построения равен **GLU_PATH_LENGTH** или **GLU_OBJECT_PATH_LENGTH**, ***value*** управляет максимальной длиной в пикселях или объектных координатах, соответственно, которая используется для результирующих полигонов. Например, значение по умолчанию **50.0** ограничивает максимальную длину сегмента линии или ребра полигона **50.0** пикселями или **50.0** единицами в объектном пространстве. Если ***property*** равно **GLU_PARAMETRIC_ERROR** и режим построения установлен в **GLU_PARAMETRIC_ERROR** или **GLU_OBJECT_PARAMETRIC_ERROR**, ***value*** управляет максимальной дистанцией в пикселях или объектных координатах между результирующими полигонами и исходной поверхностью. Для обоих режимов построения значение **GLU_PARAMETRIC_ERROR** по умолчанию равно **0.5.** Для **GLU_PARAMETRIC_ERROR** эта величина означает, что результирующие полигоны должны иметь такой размер и положение, чтобы отстоять от исходной аналитической поверхности не более, чем на **0.5** пикселя. **(**Смысл этого значения для **GLU_OBJECT_PARAMETRIC_ERROR** вовсе не очевиден.**)**

Если режим построения установлен в **GLU_DOMAIN_DISTANCE**, а ***property*** равно **GLU_U_STEP** или **GLU_V_STEP**, ***value*** является числом точек поверхности, которые нужно вычислить в направлениях ***u***или ***v*** на единицу длины в параметрических координатах. Значениями по умолчанию для этих параметров является **100.**

Свойство **GLU_AUTO_LOAD_MATRIX** определяет, должны ли видовая и проекционная матрица, а также размеры порта просмотра извлекаться из сервера **OpenGL (**в случае значения **GL_TRUE –** значение по умолчанию**)** или приложение должно загрузить эти матрицы с помощью функции **gluLoadSamplingMatrices() (GL_FALSE).**

Замечание: Некоторые свойства NURBS, такие как GLU_NURBS_MODE и его значение GLU_NURBS_TESSELATOR, а также режимы построения на основе объектных координат GLU_OBJECT_PATH_LENGTH и GLU_OBJECT_PARAMETRIC_ERROR появились только в GLU версии 1.3. До версии 1.3 эти режимы и свойства существовали только в качестве расширений отдельных производителей (если вообще существовали). Перед попыткой использовать эти свойства обязательно проверьте свою версию GLU.

```
void gluLoadSamplingMatrices (GLUnurbsObj* nobj, const GLfloat modelMatrix[16],
const GLfloat projMatrix[16], const GLint viewport[4]);
```

Если режим **GLU_AUTO_LOAD_MATRIX** выключен, порт просмотра, видовая и проекционная матрицы, заданные в **gluLoadSamplingMatrices()** используется для создания расчетной и отсекающей матриц для каждой кривой или поверхности **NURBS.**

Если вам нужно выяснить текущую величину одного из свойств **NURBS,** используйте функцию **gluGetNurbsProperty().**

```
void gluGetNurbsProperty (GLUnurbsObj* nobj, GLenum property, GLfloat* value);
```

Возвращает текущее значение свойства ***property*** объекта ***nobj*** в переменной ***value*.**

12.2.2.2 Обработка ошибок NURBS

Поскольку насчитывается **37** ошибок, специфических для работы с функциями **NURBS,** неплохой идеей является регистрация функции обратного вызова для обработки ошибок. Эта функция будет вызвана в случае, если вы наткнетесь на одну из специфических ошибок. В примере **12-5** функция обратного вызова регистрируется с помощью строки

```
gluNurbsCallback(theNurb, GLU_ERROR, (GLvoid (*)()) nurbsError);
```

```
void gluNurbsCallback (GLUnurbsObj* nobj, GLenum which, void (* fn)(GLenum
errorCode));
```

which задает тип возвратно вызываемой функции, для функции проверки ошибок ***which*** должно иметь значение **GLU_ERROR.** Когда механизм работы с **NURBS** сталкивается с одной из ошибок, он вызывает функцию ***fn*,** передавая ей код ошибки в качестве единственного аргумента. ***errorCode*** представляет собой одно из **37** значений кода ошибки, имеющих символические имена от **GLU_NURBS_ERROR1** до **GLU_NURBS_ERROR37.** Для получения строки с описание ошибки используйте функцию **gluErrorString().**

В примере **12-5** в качестве возвратной функции обработки ошибок была зарегистрирована функция **nurbsErrorr():**

```
void CALLBACK nurbsError(GLenum errorCode)
```

```
{
      char message[100];

      sprintf(message,"NURBS error: %s\n",gluErrorString(errorCode));
      MessageBox(NULL,message,"**NURBS surface",MB_OK);
      exit(0);
}
```

В **GLU** версии **1.3** были добавлены дополнительные функции обратного вызова, которые позволяют программе получать результирующие данные обратно, а не визуализировать их.

12.2.3 Создание кривой или поверхности NURBS

Чтобы визуализировать поверхность **NURBS**, функции **gluNurbsSurface()** обрамляются вызовами **gluBeginSurface()** и **gluEndSurface()**. Обрамляющие функции сохраняют и восстанавливают состояние вычислителя.

```
void gluBeginSurface (GLUnurbsObj* nobj);
void gluEndSurface (GLUnurbsObj* nobj);
```

После вызова **gluBeginSurface()** один или более вызовов функции **gluNurbsSurface()** задает атрибуты поверхности. Для генерирования вершин поверхности типа **GL_MAP2_VERTEX_3** или **GL_MAP2_VERTEX_4** требуется один и только один такой вызов. Функция **gluEndSurface()** используется в качестве маркера конца определения поверхности. Декорирование **NURBS** также происходит в обрамлении указанных функций.

```
void gluNurbsSurface (GLUnurbsObj* nobj, GLint uknot_count, GLfloat* uknot, GLint
vknot_count, GLfloat* vknot,
                      GLint u_stride, GLint v_stride, GLfloat* ctlarray, GLint
uorder, GLint vorder, GLenum type);
```

Описывает вершины (или нормали, или координаты текстуры) поверхности **NURBS** ***nobj***. Часть величин должна быть задана для обоих параметрических направлений ***u*** и ***v***, например, узловые последовательности (***uknot*** и ***vknot***), число узлов (***uknot_count*** и ***vknot_count***) и порядок полинома (***uorder*** и ***vorder***). Заметьте, что число контрольных точек не задается. Вместо этого число точек вдоль каждого параметрического направления вычисляется как количество узлов минус порядок полинома. Далее общее число контрольных точек получается как произведение двух найденных величин. Аргумент ***ctlarray*** указывает на массив контрольных точек.

Последний параметр ***type*** должен быть равен одной из констант, обозначающих тип двумерного вычислителя. Чаще всего используются **GL_MAP2_VERTEX_3** (для нерациональных вершин) и **GL_MAP2_VERTEX_4** (для рациональных). Вы также можете использовать другие типы, такие как **GL_MAP2_TEXTURE_COORD_*** или **GL_MAP2_NORMAL** для вычисления и ассоциирования координат текстуры и вектора нормали. Например, чтобы создать освещенную (с вычисленными векторами нормали) и текстурированную поверхность **NURBS**, вам может понадобиться выполнить следующую последовательность команд:

```
gluBeginSurface(nobj);
      gluNurbsSurface(nobj, ..., GL_MAP2_TEXTURE_COORD_2);
      gluNurbsSurface(nobj, ..., GL_MAP2_NORMAL);
      gluNurbsSurface(nobj, ..., GL_MAP2_VERTEX_3);
glEndSurface();
```

Аргументы ***u_stride*** и ***v_stride*** представляют количество дробных величин между контрольными точками в каждом параметрическом направлении**.** На значение аргументов ***u_stride*** и ***v_stride*** влияют тип вычислителя и его порядок**.** Так в примере **12-5** ***u_stride*** равен **12 (4*3),** поскольку задаются **3** координаты для каждой вершины **(GL_MAP2_VERTEX_3)** и четыре контрольные точки в параметрическом направлении ***v*****,** ***v_stride*** равен **3,** поскольку каждая вершина имеет **3** координаты и точки ***v*** расположены в памяти плотно одна за одной**.**

Рисование кривой **NURBS** похоже на рисование поверхности**,** за тем исключением**,** что вычисления производятся по одному параметру **–** ***u*****,** а не по двум**.** Кроме того**,** для кривых используются обрамляющие функции **gluBeginCurve()** и **gluEndCurve().**

```
void gluBeginCurve (GLUnurbsObj* nobj);
void gluEndCurve (GLUnurbsObj* nobj);
```

После вызова **gluBeginCurve()**, один или несколько вызовов **gluNurbsCurve()** определяют атрибуты кривой**.** Для генерирования вершин кривой типа **GL_MAP1_VERTEX_3** или **GL_MAP1_VERTEX_4** требуется один и только один такой вызов**.** Для завершения определения кривой**,** используйте функцию **gluEndCurve().**

```
void gluNurbsCurve (GLUnurbsObj* nobj, GLint uknot_count, GLfloat* uknot, GLint
u_stride, GLfloat* ctlarray, GLint uorder, GLenum type);
```

Определяет кривую **NURBS** для объекта ***nobj*****.** Аргументы имеют тот же смысл**,** что и для **gluNurbsSurface().** Заметьте**,** что эта функция требует только одной узловой последовательности и одного объявления порядка для объекта **NURBS.** Если эта кривая определяется внутри блока **gluBeginCurve() / gluEndCurve()**, аргумент ***type***может иметь любое значение**,** допустимое для типа одномерного вычислителя **(**например**,** **GL_MAP1_VERTEX_3** или **GL_MAP1_VERTEX_4).**

12.2.3.1 Получение примитивов из тесселятора NURBS обратно

По умолчанию тесселятор **NURBS** разбивает объект **NURBS** на геометрические линии и полигоны**,** и визуализирует их**.** В **GLU** версии **1.3** были добавлены дополнительные возвратные функции**,** так что вместо визуализации пост **–** тесселяционных величин**,** они могут быть возвращены для использования программе**.**

Чтобы это сделать**,** первым шагом является вызов функции **gluNurbsProperty()** для установки свойства **GLU_NURBS_MODE** в режим **GLU_NURBS_TESSELATOR.** Следующие шагом является выполнение нескольких вызовов функции **gluNurbsCallback()** для регистрации функций обратного вызова**,** которые будут вызываться тесселятором **NURBS.**

```
void gluNurbsCallback (GLUnurbsObj* nobj, GLenum which, void (*fn)());
```

nobj представляет собой тесселируемый объект **NURBS.** ***which*** **–** это перечислимое значение**,** идентифицирующее тип возвратной функции**.** Если свойство **GLU_NURBS_MODE** установлено в режим **GLU_NURBS_TESSELATOR,** то помимо функции **GLU_ERROR** можно зарегистрировать еще **12** функций**. (**В противном случае**,** активна только функция **GLU_ERROR.) 12** возвратных функций идентифицируются следующими константами и имеют следующие прототипы**:**

GLU_NURBS_BEGINtd>	**void begin (GLenum *type*);**
GLU_NURBS_BEGIN_DATA	**void begin (GLenum *type*, void* *userData*);**
GLU_NURBS_TEXTURE_COORD	**void texCoord (GLfloat* *tCrd*);**
GLU_NURBS_TEXTURE_COORD_DATA	**void texCoord (GLfloat* *tCrd*, void* *userData*);**

GLU_NURBS_COLOR	**void color (GLfloat* *color*);**
GLU_NURBS_COLOR_DATA	**void color (GLfloat* *color*, void* *userData*);**
GLU_NURBS_NORMAL	**void normal (GLfloat* *nml*);**
GLU_NURBS_NORMAL_DATA	**void normal (GLfloat* *nml*, void* *userData*);**
GLU_NURBS_VERTEX	**void vertex (GLfloat* *vertex*);**
GLU_NURBS_VERTEX_DATA	**void vertex (GLfloat* *vertex*, void* *userData*);**
GLU_NURBS_END	**void end (void);**
GLU_NURBS_END_DATA	**void end (void* *userData*);**

Чтобы изменить возвратную функцию, зарегистрируйте новую с тем же типом. Чтобы отменить регистрацию функции вообще, вызовите **gluNurbsCallback()** с нулевым указателем для соответствующего типа функции.

6 возвратных функций позволяют пользователю передавать в них некоторые данные. Чтобы задать пользовательские данные, вызовите **gluNurbsCallbackData()**.

```
void gluNurbsCallbackData (GLUnurbsObj* nobj, void* userData);
```

nobj представляет собой тесселируемый объект **NURBS**. ***userData*** это данные, которые должны быть переданы функции обратного вызова.

В течение тесселяции возвратно – вызываемые функции вызываются в той же манере, к какой вы используете команды **glBegin()**, **glTexCoord*()**, **glColor*()**, **glNormal*()**, **glVertex*()** и **glEnd()**. В режиме **GLU_NURBS_TESSELATOR** кривая или поверхность не визуализируются на экране, но вы можете захватить вершинные данные, передаваемые в качестве параметров возвратным функциям.

Чтобы продемонстрировать эти новые возвратные функции, обратимся к примерам **12-6** и **12-7**. В примере **12-6** показана часть функции **init()**, где создается объект **NURBS**, устанавливается режим **GLU_NURBS_TESSELATOR** для свойства **GLU_NURBS_MODE** и регистрируются функции обратного вызова.

Пример **12-6**. Регистрация возвратных функций тесселяции **NURBS**: файл **surfpoints.cpp**

```
void init(void)
{
      /* часть функции init() */
      theNurb=gluNewNurbsRenderer();
      gluNurbsProperty(theNurb, GLU_NURBS_MODE,
GLU_NURBS_TESSELATOR);
      gluNurbsProperty(theNurb,GLU_SAMPLING_TOLERANCE,100.0);
      gluNurbsProperty(theNurb,GLU_DISPLAY_MODE,GLU_FILL);
      gluNurbsCallback(theNurb,GLU_ERROR,nurbsError);
      gluNurbsCallback(theNurb,GLU_NURBS_BEGIN,beginCallback);
      gluNurbsCallback(theNurb,GLU_NURBS_VERTEX,vertexCallback);
      gluNurbsCallback(theNurb,GLU_NURBS_NORMAL,normalCallback);
      gluNurbsCallback(theNurb,GLU_NURBS_END,endCallback);
}
```

Пример **12-7** демонстрирует сами функции обратного вызова. В этих функциях **printf()** выступает в качестве инструмента диагностики, показывающего директивы и вершинные данные, возвращаемые из тесселятора. Кроме того, пост – тесселяционные данные посылаются на конвейер для обычной визуализации.

Пример **12-7**. Возвратно вызываемые функции тесселятора **NURBS**: файл **surfpoints.cpp**

```
void CALLBACK beginCallback(GLenum whichType)
```

```
{
      glBegin(whichType);     //Посылаем данные на конвейер

      printf("glBegin(");
      switch(whichType)//Вывести диагностическое сообщение
      {
            case GL_LINES:
                  printf(“GL_LINES)\n”);
                  break;
            case GL_LINE_LOOP:
                  printf(“GL_LINE_LOOP)\n”);
                  break;
            case GL_LINE_STRIP:
                  printf(“GL_LINE_STRIP)\n”);
                  break;
            case GL_TRIANGLES:
                  printf(“GL_TRIANGLES)\n”);
                  break;
            case GL_TRIANGLE_STRIP:
                  printf(“GL_TRIANGLE_STRIP)\n”);
                  break;
            case GL_TRIANGLE_FAN:
                  printf(“GL_TRIANGLE_FUN)\n”);
                  break;
            case GL_QUADS:
                  printf(“GL_QUADS)\n”);
                  break;
            case GL_QUAD_STRIP:
                  printf(“GL_QUAD_STRIP)\n”);
                  break;
            case GL_POLYGON:
                  printf(“GL_POLYGON)\n”);
                  break;
            default:
                  break;
      }
}

void CALLBACK endCallback()
{
      glEnd();
      printf(“glEnd()\n”);
}

void CALLBACK vertexCallback(GLfloat* vertex)
{
      glVertex3fv(vertex);
      printf("glVertex3f (%5.3f, %5.3f, %5.3f)\n",
              vertex[0],vertex[1],vertex[2]);
}

void CALLBACK normalCallback(GLfloat* normal)
{
      glNormal3fv(normal);
      printf(“glNormal3f (%5.3f, %5.3f, %5.3f)\n”,
             normal[0], normal[1], normal[2]);
}
```

12.2.4 Декорирование поверхностей NURBS

Чтобы создать декорированную поверхность **NURBS** с помощью **OpenGL,** начните так же как и при создании обычной поверхности**.** После вызовов **gluBeginSurface()** и

gluNurbsSurface(), но до вызова **gluEndSurface()** начните декорирование с вызова **gluBeginTrim()**.

```
void gluBeginTrim (GLUnurbsObj* nobj);
void gluEndTrim (GLUnurbsObj* nobj);
```

Маркируют начало и конец определения декорирующих кривых и поверхностей. Декорирующий круг – это набор ориентированных декорирующих сегментов кривой (формирующих замкнутую кривую), определяющих границы поверхности **NURBS**.

Вы можете создавать два вида декорирующих кривых: кусочную линейную кривую с помощью **gluPwlCurve()** или кривую **NURBS** с помощью **gluNurbsCurve()**. Кусочная линейная кривая не выглядит как то, что обычно называют кривой, поскольку состоит она из прямых линий. Кривая **NURBS** для декорирования должна лежать в единичном параметрическом пространстве (***u***, ***v***). Типом для декорирующей кривой **NURBS** обычно является **GLU_MAP1_TRIM_2**. Реже, типом является **GLU_MAP1_TRIM_3**, где кривая описывается в двумерном однородном пространстве (***u'***, ***v'***, ***w'***) как (***u***, ***v***)=(***u'/w'***, ***v'/w'***).

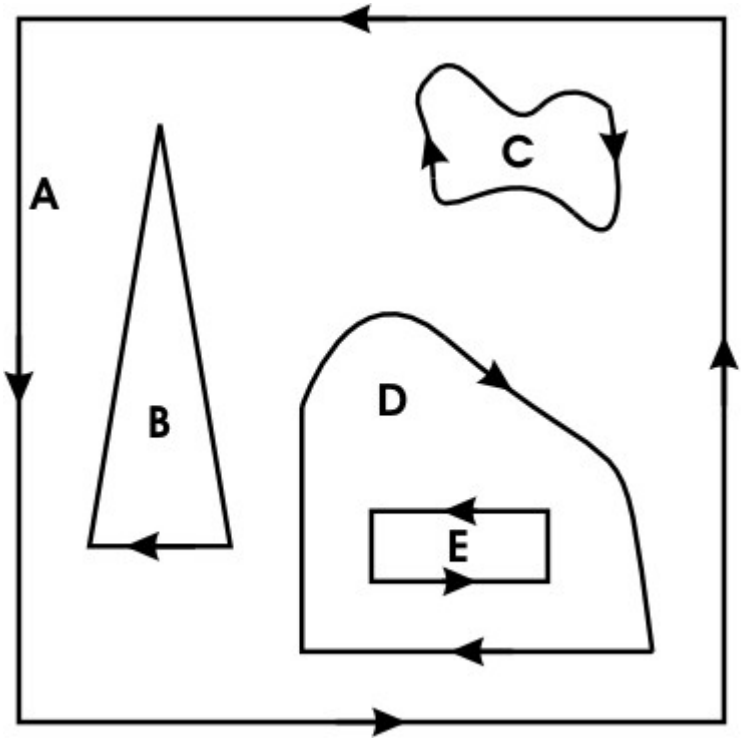
```
void gluPwlCurve (GLUnurbsObj* nobj, GLint count, GLfloat* array, GLint stride,
GLenum type);
```

Описывает кусочную линейную декорирующую кривую для **NURBS** Объекта ***nobj***. На кривой ***count*** точек, задаваемых в массиве ***array***. Аргумент ***type*** может принимать значения **GLU_MAP1_TRIM_2** (более часто) или **GLU_MAP1_TRIM_3** (однородное параметрическое пространство (***u***, ***v***, ***w***)). Значение ***type*** определяет значение ***stride***, которое может быть равно **2** или **3**. ***stride*** должен быть равен количеству величин между соседними вершинами в ***array***.

Вы должны определиться с ориентацией обрамляющих кривых – будет ли она против часовой стрелки или по часовой стрелке – чтобы быть уверенными в том, что вы включите нужную часть поверхности. Если вы представите себя идущим вдоль кривой, все, что находится слева от вас, будет нарисовано, а все, что справа, будет отброшено. Например, если ваша кривая состоит из одного цикла против часовой стрелки, все что находится внутри кривой, будет включено. Если декорирующая кривая состоит из двух непересекающихся циклов против часовой стрелки, имеющих непересекающиеся границы, будет включено все, что находится в обоих циклах. Если она состоит из одного цикла против часовой стрелки с двумя циклами по часовой стрелке внутри него, включаемая область будет иметь **2** дыры в ней. Самая внешняя кривая должна иметь ориентацию против часовой стрелки. Часто, вы запускаете декорирующую кривую по всему параметрическому пространству, чтобы включить все, что находится внутри. Этот результат соответствует тому, что вы получаете без декорирования вообще.

Декорирующие кривые должны быть закрытыми и непересекающимися. Вы можете комбинировать кривые, если конечные точки кривых формируют закрытую кривую. Вы можете вкладывать кривые одну в другую, создавая острова, плавающие в пространстве. Убедитесь в правильности ориентации кривой. Например, если вы зададите два декорирующих региона против часовой стрелки, причем так, что один находится внутри другого, произойдет ошибка, поскольку регион, находящийся между ними, находится слева от одной кривой и справа от другой, то есть он одновременно должен быть и включен и исключен. Рисунок **12-6** демонстрирует несколько допустимых случаев.

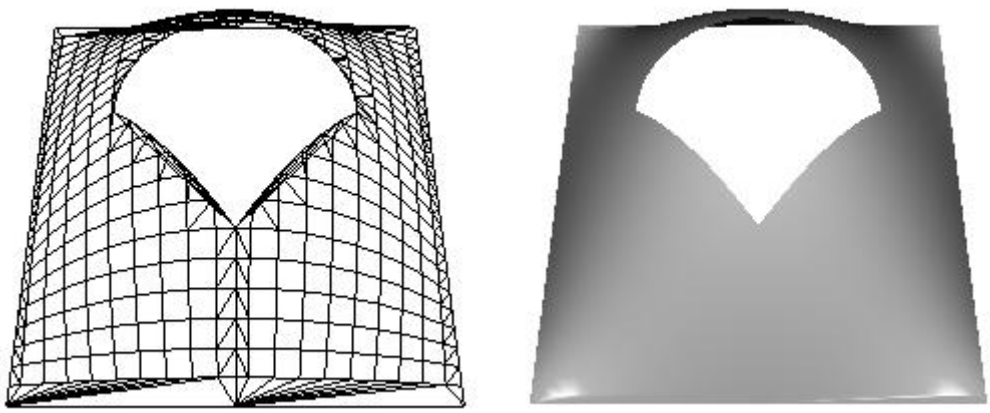
Рисунок **12-6**. Параметрические декорирующие кривые




```
gluBeginSurface();
gluNurbsSurface(...);
gluBeginTrim();
gluPwlCurve(...);    /*A*/
gluEndTrim();
gluBeginTrim();
gluPwlCurve(...);    /*B*/
gluEndTrim();
gluBeginTrim();
gluNurbsCurve(...); /*C*/
gluEndTrim();
gluBeginTrim();
gluNurbsCurve(...); /*D*/
gluPwlCurve(...);    /*D*/
gluEndTrim();
gluBeginTrim();
gluPwlCurve(...);    /*E*/
gluEndTrim();
gluEndSurface();
```

На рисунке **12-7** показан тот же холм, что и на рисунке **12-5**, но к нему добавлена декорирующая кривая, состоящая из кусочной кривой и кривой **NURBS.** Программа, которая создает это изображение аналогично примеру **12-5** за исключение фрагмента, показанного в примере **12-8.**

Рисунок **12-7.** Декорированная поверхность **NURBS**



Пример **12-7.** Декорирование поверхности **NURBS:** файл **trim.cpp**

```
void display()
{
      GLfloat knots[8]={0.0,0.0,0.0,0.0,1.0,1.0,1.0,1.0};

      GLfloat edgePt[5][2]= /* против часовой стрелки */
            {{0.0,0.0},{1.0,0.0},{1.0,1.0},{0.0,1.0},
             {0.0,0.0}};

      GLfloat curvePt[4][2]= /* по часовой стрелке */
            {{0.25,0.5}, {0.25,0.75}, {0.75,0.75}, {0.75,0.5}};
```

```
        GLfloat curveKnots[8]={0.0,0.0,0.0,0.0,1.0,1.0,1.0,1.0};

        GLfloat pwlPt[3][2]= /* по часовой стрелке */
              {{0.75,0.5}, {0.5,0.25}, {0.25,0.5}};

        glClear(GL_COLOR_BUFFER_BIT|GL_DEPTH_BUFFER_BIT);
        glPushMatrix();
        glRotatef(330.0,1.0,0.0,0.0);
        glScalef(0.5,0.5,0.5);

        gluBeginSurface(theNurb);
              gluNurbsSurface(theNurb,8,knots,8,knots,4*3,3,

  &ctrlpoints[0][0][0],4,4,GL_MAP2_VERTEX_3);
              gluBeginTrim(theNurb);

  gluPwlCurve(theNurb,5,&edgePt[0][0],2,GLU_MAP1_TRIM_2);
              gluEndTrim(theNurb);
              gluBeginTrim(theNurb);
                    gluNurbsCurve(theNurb,8,curveKnots,2,
                                &curvePt[0][0],4,GLU_MAP1_TRIM_2);

  gluPwlCurve(theNurb,3,&pwlPt[0][0],2,GLU_MAP1_TRIM_2);
              gluEndTrim(theNurb);
        gluEndSurface(theNurb);

        glPopMatrix();
        glFlush();
  }
```

В примере **12-8** вызовы **gluBeginTrim()** и **gluEndTrim()** обрамляют каждую декорирующую кривую. Первая из них с вершинами, определенными в массиве ***edgePt[ ][ ]***, проходит против часовой стрелки вокруг всего единичного квадрата параметрического пространства. Это позволяет быть уверенным в том, что будет нарисовано все (если, конечно, часть поверхности не будет удалена внутренней кривой с ориентацией по часовой стрелке). Вторая кривая – это комбинация кривой **NURBS** и кусочной линейной кривой. Кривая **NURBS** заканчивается в точках (***0.9***, ***0.5***) и (***0.1***, ***0.5***), где она соединяется с линейной кривой, формируя замкнутую кривую.

Глава 13. Режим выбора и обратный режим

Некоторые графические приложения просто рисуют статические изображения двумерных или трехмерных объектов. Другие приложения позволяют пользователю выбирать объект на экране, а затем двигать, изменять, удалять или как-либо еще манипулировать этим объектом. **OpenGL** разработана для поддержки именно таких интерактивных приложений. Поскольку объекты, рисуемые на экране обычно претерпевают множественные повороты, переносы и перспективные преобразования, для вас может быть достаточно сложно определить, какой объект в трехмерной сцене выбрал пользователь. Чтобы помочь вам, **OpenGL** предоставляет механизм выбора **(selection mechanism)**, который автоматически вычисляет и сообщает вам, какие объекты нарисованы внутри заданного региона окна. Вы можете использовать этот механизм совместно со специальной утилитарной функцией, определяющей внутри региона конкретный объект, который пользователь *указал* (***picking***) с помощью курсора.

На самом деле выбор – это один из режимов выполнения операций **OpenGL**; отклик **(feedback)** является еще одним таким режимом. В режиме отклика ваша графическая аппаратура и **OpenGL** используются для выполнения обычных расчетов для визуализации. Однако вместо того, чтобы использовать вычисленные величины для рисования изображения на экране, **OpenGL** возвращает информацию о рисовании вам. Например, если вы хотите отобразить трехмерную сцену на плоттере, а не на экране,

вы можете нарисовать объекты в режиме отклика, собрать инструкции, необходимые для рисования и преобразовать их в команды, которые плоттер может понять и выполнить.

И в режиме выбора, и в режиме отклика информация о рисовании возвращается приложению, а не отсылается в буфер кадра, как происходит в режиме визуализации. Таким образом, изображение на экране не меняется – ничего не рисуется, пока **OpenGL** находится в режимах выбора или отклика. В этих режимах цветовой буфер, буфер глубины, буфер трафарета и аккумуляторный буфер не используются. В данной главе каждый из указанных режимов описан в отдельном разделе.

13.1 Выбор

Обычно, когда вы планируете использовать механизм выбора **OpenGL**, вы сначала рисуете вашу сцену в буфере кадра, а затем переходите в режим выбора и перерисовываете сцену. Однако после того, как вы вошли в режим выбора, содержимое буфера кадра не меняется до тех пор, пока вы не покинете этот режим. Когда вы выходите из режима выбора, **OpenGL** возвращает список примитивов, пересекающих объем видимости (помните, что объем видимости определяется текущими видовой и проекционной матрицами, а также дополнительными плоскостями отсечения). Каждый примитив, имеющий пересечение с объемом видимости вызывает то, что называется *попаданием* (***hit***). На самом деле список примитивов возвращается в виде массива целочисленных *имен* и связанных данных – *записей о попаданиях* (***hitrecords***) – соответствующих текущему содержимому *стека имен*. Находясь в режиме выбора, вы конструируете стек имен, загружая в него имена одновременно с выполнением команд для рисования примитивов. Таким образом, когда список имен возвращается в программу, вы можете использовать его для определения тех примитивов, которые пользователь мог выбрать на экране.

Вдобавок к этому механизму выбора **OpenGL** предоставляет утилитарную функцию, разработанную для упрощения процесса выбора в некоторых случаях, путем ограничения рисования небольшой областью порта просмотра. Обычно эта функция используется для обнаружения объектов, нарисованных вблизи курсора, чтобы вы могли идентифицировать каждый объект, который указал пользователь. (Вы также можете ограничить область выбора за счет добавления дополнительных плоскостей отсечения. Помните, что эти плоскости действуют в мировом пространстве, а не в экранном.) Поскольку указание объектов (**picking**) это специальный случай выбора как такового, в данной главе выбор описывается первым, а указание за ним.

13.1.1 Основные шаги

Чтобы использовать механизм выбора, вам нужно выполнить следующие шаги.

1. С помощью команды **glSelectBuffer()** задайте массив, который будет использоваться для возвращения записей о попаданиях.
2. Перейдите в режим выбора, передав аргумент **GL_SELECT** команде **glRenderMode()**.
3. Инициализируйте стек имен с помощью команд **glInitNames()** и **glPushName()**.
4. Определите объем видимости, который вы хотите использовать для режима выбора. Обычно этот объем отличается от того, который изначально использовался для рисования сцены, так что, скорее всего, вам нужно будет сохранить и позже восстановить состояние преобразований с помощью команд **glPushMatrix()** и **glPopMatrix()**.
5. Чередуйте выполнение команд для рисования примитивов с командами для манипуляций со стеком имен, дабы быть уверенными в том, что каждый интересующий вас полигон получил соответствующее имя.
6. Выйдите из режима выбора и обработайте возвращенные данные о выборе (записи о попаданиях).

```
void glSelectBuffer (GLsizei size, GLuint* buffer);
```

Задает массив, который будет использоваться для возвращения информации о выбранных примитивах. Аргумент ***buffer*** должен быть указателем на массив беззнаковых целых, в который будут помещаться данные, а аргумент ***size***– задает максимальное количество величин, которые могут быть записаны в этот массив. Вы должны вызвать команду **glSelectBuffer()** до перехода в режим выбора.

```
GLint glRenderMode (GLenummode);
```

Управляет тем, в каком режиме приложение находится в текущий момент. Аргумент ***mode*** может принимать значения **GL_RENDER (**режим визуализации, значение по умолчанию**), GL_SELECTION (**режим выбора**)** или **GL_FEEDBACK (**режим отклика**).** Приложение остается в заданном режиме до того, как команда **glRenderMode()** не будет вызвана снова с другим аргументом. До перехода в режим выбора вы должны задать буфер выбора с помощью команды **glSelectBuffer().** Похожим образом, до перехода в режим отклика с помощью команды **glFeedbackBuffer()** должен быть задан массив отклика. Возвращаемое значение команды **glRenderMode()** имеет смысл, только если текущим режимом визуализации **(**именно текущим, а не заданным в аргументе**)** является **GL_SELECT** или **GL_FEEDBACK.** Возвращаемое значение представляет собой число записей о попаданиях или число величин, занесенных в массив отклика при выходе из режимов **GL_SELECT** или **GL_FEEDBACK** соответственно; отрицательная величина означает, что буфер выбора или отклика переполнен. Для выяснения того, в каком режиме вы находитесь в данный момент, используйте аргумент **GL_RENDER_MODE** в команде **glGetIntegerv().**

13.1.2 Создание стека имен

Как было указано ранее, стек имен формирует базис для информации о выборе, которая к вам возвращается. Чтобы создать стек имен, сначала инициализируйте его командой **glInitNames()**, которая просто очищает стек, а затем добавляйте в него целочисленные имена в процессе исполнения соответствующих команд рисования. Как вы можете ожидать, команды манипулирования стеком имен позволяют поместить имя на вершину стека с продвижением всех, содержащихся в нем, глубже **(glPushName())**, поднять имя из стека на вершину, потеряв при этом предыдущее имя на вершине **(glPopName())** и заменить имя на вершине стека каким-либо другим **(glLoadName()).** Пример кода, манипулирующего стеком имен, продемонстрирован в примере **13-1.**

Пример **13-1.** Создание стека имен

```
glInitStack();
glPushName(0);

glPushMatrix();   /* сохраняем текущее состояние преобразований */

/* здесь создается требуемый объем видимости */

      glLoadName(1);
      нарисовать_Какой_либо_Объект();
      glLoadName(2);
      нарисовать_Другой_Объект();
      glLoadName(3);
      нарисовать_Еще_Один_Объект();
      нарисовать_Последний_Объект();

glPopMatrix();    /* восстанавливаем предыдущее состояние
преобразований */
```

В этом примере первые два рисуемых объекта имеют свои собственные имена, а третий и четвертый объекты разделяют одно общее имя. При такой настройке, если третий или четвертый объект или оба сразу вызовут попадание, вам будет возвращена только одна запись о попадании. Вы можете заставить несколько объектов разделять одно общее имя, если вам не нужно разделять их при обработке записей.

```
void glInitNames (void);
```

Полностью очищает стек имен.

```
void glPushName (GLuint name);
```

Помещает **name** на вершину стека. Помещение имени, в полный стек генерирует ошибку **GL_STACK_OVERFLOW**. Размер стека имен может быть разным в разных реализациях **OpenGL**, но в любой реализации стек должен иметь вместимость, как минимум, для **64** имен. Для выяснения реальной глубины стека имен вы можете использовать аргумент **GL_NAME_STACK_DEPTH** в команде **glGetIntegerv()**.

```
void glPopName (void);
```

Поднимает имя из стека на вершину. Извлечение имени из пустого стека генерирует ошибку **GL_STACK_UNDERFLOW**.

```
void glLoadName (GLuint name);
```

Заменяет имя на вершине стека величиной ***name***. Если стек пуст, что вполне нормально после вызова **glInitNames()**, вызов команды **glLoadName()** сгенерирует ошибку **GL_INVALID_OPERATION**. Чтобы этого избежать, сразу после инициализации стека поместите что-либо на его вершину командой **glPushName()** до вызова **glLoadName()**.

Обращения к **glPushName()**, **glPopName()** и **glLoadName()** игнорируются, если программа не находится в режиме выбора. Вы можете обнаружить, что этот факт упрощает смешивание этих команд с командами рисования – вы можете использовать такой смешанный код для отображения объектов на экране, а затем использовать его же для получения информации о выборе.

13.1.3 Запись о попадании

В режиме выбора примитивы, имеющие пересечения с объемом видимости, вызывают попадание. Каждый раз, когда выполняется команда манипулирования стеком имен или команда **glRenderMode()**, **OpenGL** записывает запись о попадании в буфер выбора, если попадание было зафиксировано после последней манипуляции со стеком или выполнения команды **glRenderMode()**. В течение этого процесса объекты, разделяющие общее имя – например, объекты, состояние из нескольких примитивов – не генерируют множественных записей о попаданиях. Кроме того, не гарантируется, что записи о попаданиях будут записаны в буфер выбора до вызова команды **glRenderMode()**.

Замечание: Помимо примитивов попадание могут вызвать допустимые координаты, обозначенные в команде **glRasterPos()**. Кроме того, в случае полигонов попадания не происходит, если полигон отбрасывается механизмом удаления нелицевых граней.

Каждая запись о попадании состоит из **4** элементов **(**они перечислены по порядку**):**

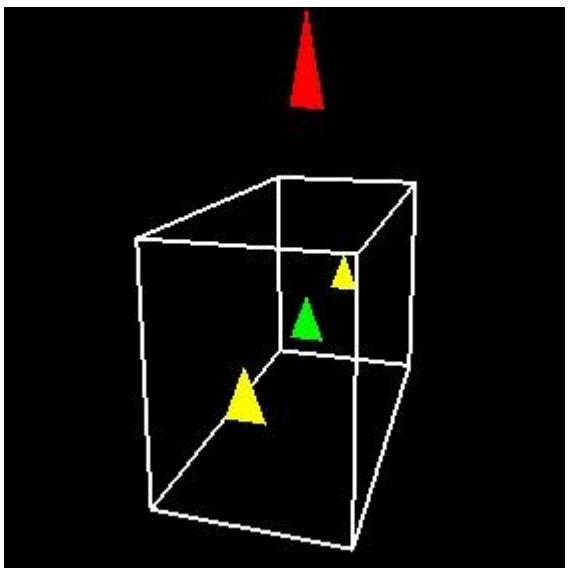
- Количество имен в стеке в момент попадания.
- Минимальная и максимальная координата ***z*** **(**оконная**)** всех вершин примитивов, пересекающих объем видимости с момента последнего попадания. Эти две величины, лежащие в диапазоне **[*0*, *1*]** умножаются на $2^{32}-1$ и округляются до ближайшего беззнакового целого.
- Содержимое стека имен на момент попадания. При этом первым элементом стоит элемент со дна стека.

Когда вы переходите в режим выбора, **OpenGL** инициализирует указатель на начало массива выбора. Каждый раз, когда запись о попадании записывается в массив, значение указателя обновляется. Если занесение записи о попадании должно привести к тому, что общее количество величин превысит аргумент **size** указанный в команде **glSelectBuffer()**, **OpenGL** записывает в буфер допустимую часть записи и устанавливает флаг переполнения. Когда вы выходите из режима выбора командой **glRenderMode()**, эта команда возвращает общее количество записей о попаданиях, занесенных в массив выбора **(**включая и частично записанную запись, если таковая была**)**, очищает стек имен, сбрасывает флаг переполнения и сбрасывает указатель стека. Если флаг переполнения был установлен, возвращаемое значение равно **-1**.

13.1.4 Пример реализации выбора

В примере **13-2** на экране с помощью функции **drawTriangle()** рисуются **4** треугольника: зеленый, красный и два желтых, а также каркасный параллелепипед, представляющий объем видимости **(**функция **drawViewVolume()).** Затем треугольники визуализируются еще раз, но на этот раз в режиме выбора **(**функция **selectObjects()).** Соответствующие записи о попаданиях обрабатываются в функции **processHits()** и массив выбора распечатывается в консольном окне. Первый треугольник генерирует попадание, второй не генерирует его, а третий и четвертый вместе генерируют одно попадание. Результат работы программы показан на рисунке **13-1.**

Рисунок **13-1.** Пример реализации выбора



Пример **13-2.** Пример реализации выбора: файл **select.cpp**

```
#include
#include
#include

void drawTriangle(GLfloat x1,GLfloat y1,GLfloat x2,GLfloat y2,GLfloat
```

```
x3,GLfloat y3,GLfloat z)
{
	glBegin(GL_TRIANGLES);
		glVertex3f(x1,y1,z);
		glVertex3f(x2,y2,z);
		glVertex3f(x3,y3,z);
	glEnd();
}

void drawViewVolume(GLfloat x1,GLfloat x2,GLfloat y1,GLfloat
y2,GLfloat z1,GLfloat z2)
{
	glColor3f(1.0,1.0,1.0);
	glBegin(GL_LINE_LOOP);
		glVertex3f(x1,y1,-z1);
		glVertex3f(x2,y1,-z1);
		glVertex3f(x2,y2,-z1);
		glVertex3f(x1,y2,-z1);
	glEnd();

	glBegin(GL_LINE_LOOP);
		glVertex3f(x1,y1,-z2);
		glVertex3f(x2,y1,-z2);
		glVertex3f(x2,y2,-z2);
		glVertex3f(x1,y2,-z2);
	glEnd();

	glBegin(GL_LINES);
		glVertex3f(x1,y1,-z1);
		glVertex3f(x1,y1,-z2);
		glVertex3f(x1,y2,-z1);
		glVertex3f(x1,y2,-z2);
		glVertex3f(x2,y1,-z1);
		glVertex3f(x2,y1,-z2);
		glVertex3f(x2,y2,-z1);
		glVertex3f(x2,y2,-z2);
	glEnd();
}

void drawScene()
{
	glMatrixMode(GL_PROJECTION);
	glLoadIdentity();
	gluPerspective(40.0,4.0/3.0,1.0,100.0);

	glMatrixMode(GL_MODELVIEW);
	glLoadIdentity();
	gluLookAt(7.5,7.5,12.5,2.5,2.5,-5.0,0.0,1.0,0.0);

	glColor3f(0.0,1.0,0.0);
	drawTriangle(2.0,2.0,3.0,2.0,2.5,3.0,-5.0);

	glColor3f(1.0,0.0,0.0);
	drawTriangle(2.0,7.0,3.0,7.0,2.5,9.0,-5.0);

	glColor3f(1.0,1.0,0.0);
	drawTriangle(2.0,2.0,3.0,2.0,2.5,3.0,0.0);
	drawTriangle(2.0,2.0,3.0,2.0,2.5,3.0,-10.0);

	drawViewVolume(0.0,5.0,0.0,5.0,0.0,10.0);
}

void processHits(GLint hits,GLuint buffer[])
```

```
{
      int i,j;
      GLuint names, *ptr;
      printf("hits=%d\n",hits);
      ptr=(GLuint*)buffer;
      for (i=0;i< pre>
```

Эта программа генерирует следующий вывод:

```
Green triangle name is 1
Red triangle name is 2
Yellow triangles name are 3
hits=2
 number of names for hit=1
 z1 is 0.999999; z2 is 0.999999;
 the names are 1
 number of names for hit=1
  z1 is 0; z2 is 2;
 the names are 3
```

13.1.5 Указание

В качестве расширения процесса, описанного в предыдущем разделе, вы можете использовать механизм выбора для определения того, какие объекты указаны пользователем. Чтобы это сделать, в сочетании с проекционной матрицей вы используете специальную матрицу указания, ограничивающую область рисования малым регионом порта просмотра, который обычно находится вблизи курсора. Затем вы выбираете какую-либо форму ввода, например, щелчок мышью, которая будет активизировать режим выбора. В режиме выбора с использованием матрицы указания попадание генерируют только объекты, которые нарисованы вблизи курсора. Таким образом, по время указания вы обычно определяете объекты, нарисованные вблизи курсора.

Указание настраивается практически так же, как и обычный выбор, но присутствуют и важные отличия:

- Указание обычно активизируется с помощью устройства ввода. В следующих примерах кода нажатие левой кнопки мыши запускает функцию, производящую указание.
- Вы используете функцию **gluPickMatrix()** для умножения текущей проекционной матрицы на специальную матрицу указания. Эта функция должна быть вызвана до установления стандартной проекционной матрицы (например, командами **glOrtho()** или **glFrustum()**). Вероятно перед этим процессом вам потребуется сохранить предыдущую матрицу проекции, так что ваш код будет похож на следующий:

```
glMatrixMode(GL_PROJECTION);
glPushMatrix();
glLoadIdentity();
gluPickMatrix(...);
gluPerspective, glOrtho, gluOrtho2D или glFrustum
      /* ... нарисовать сцену для указания, произвести указание ...
*/
glPopMatrix();
```

```
void gluPickMatrix (GLdouble x, GLdouble y, GLdouble width, GLdouble height,
GLint viewport[4]);
```

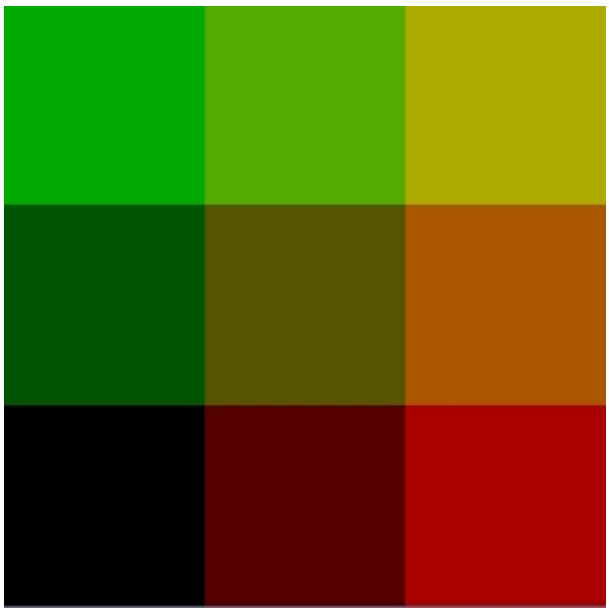
Создает проекционную матрицу, ограничивающую область рисования малым регионом порта просмотра и умножает текущую проекционную матрицу на нее. Центром региона указания становится точка (***x***, ***y***) в оконных координатах (обычно в это время там находится курсор). Аргументы ***width*** и ***height*** задают размеры (ширину и высоту) региона указания в экранных координатах. (Вы можете считать ширину и высоту области указания чувствительностью устройства ввода.) Аргумент ***viewport[]*** должен содержать текущие границы порта просмотра, которые могут быть получены с помощью следующего вызова

```
glGetIntegerv(GL_VIEWPORT, GLint *viewport);
```

Дополнительно: Действие матрицы, созданной с помощью **gluPickMatrix()** заключается в преобразовании отсекающего региона к единичному кубу **-1<=(*x*,*y*,*z*)<=1** (или **-*w*<=(*wx*,*wy*,*wz*)<=*w***). Матрица указания производит ортогональное преобразование, отображающее регион отсечения на единичный куб. Поскольку преобразование может быть произвольным, вы можете заставить указание работать с различными регионами – например, с вращающимися прямоугольными областями окна. Однако в некоторых случаях для определения области указания может быть проще создавать дополнительные плоскости отсечения.

Пример **13-3** иллюстрирует простое указание. Он также демонстрирует технику использования множественных имен для идентификации различных компонентов примитива, в данном случае – строки и столбца выбранного объекта. На экране рисуется сетка размером **33x3**, состоящая из квадратов разного размера. Массив ***board[3][3]*** хранит текущее количество синего в цвете каждого квадрата. Когда нажимается левая кнопка мыши, вызывается функция **pickSquares()**, определяющая квадраты, которые были указаны мышью. Каждый квадрат в сетке идентифицируется двумя именами – одно имя для строки и одно для столбца. Кроме того, когда нажимается левая кнопка мыши, изменяется цвет всех квадратов, находящихся вблизи курсора. Начальное изображение, генерируемое примером, показано на рисунке **13-2**.

Рисунок **13-2**. Простой пример указания



Пример **13-3**. Пример указания: файл **picksquares.cpp**

```
#include
#include
#include
```

```
int board[3][3]; //количество синего в цвете каждого квадрата

void init()
{
      int i,j;
      for(i=0;i<3;i++)
            for(j=0;j<3;j++)
                  board[i][j]=0;
      glClearColor(0.0,0.0,0.0,0.0);
}

void drawSquares(GLenum mode)
{
      GLuint i,j;
      for (i=0;i<3;i++)
      {
            if (mode==GL_SELECT)
                  glLoadName(i);
            for(j=0;j<3;j++)
            {
                  if(mode==GL_SELECT)
                        glPushName(j);

glColor3f((GLfloat)i/3.0,(GLfloat)j/3.0,(GLfloat)board[i][j]/3.0);
                  glRecti(i,j,i+1,j+1);
                  if (mode==GL_SELECT)
                        glPopName();
            }
      }
}

void processHits(GLint hits,GLuint buffer[])
{
      int i,j;

      GLuint ii, jj, names, *ptr;

      printf("hits=%d\n",hits);
      ptr=(GLuint*)buffer;
      for (i=0;i< pre this if(j="=0)" ii="*ptr;" else if (j="=1)"
jj="*ptr;" board[ii][jj]="(board[ii][jj]+1)%3;" pickSquares(int
button,int state,int x,int y) viewport[4];
if(button!="GLUT_LEFT_BUTTON" || state!="GLUT_DOWN)" return;
glGetIntegerv(GL_VIEWPORT,viewport); Create 5x5 pixel picking region
near cursor location
gluPickMatrix((GLdouble)x,(GLdouble)(viewport[3]-
y),5.0,5.0,viewport); drawSquares(GL_SELECT); glutPostRedisplay();
glClear(GL_COLOR_BUFFER_BIT); drawSquares(GL_RENDER); reshape(int w,
h) glViewport(0,0,w,h); gluOrtho2D(0.0,3.0,0.0,3.0);
glutInitDisplayMode(GLUT_SINGLE|GLUT_RGB);
glutInitWindowSize(300,300); glutCreateWindow(?Picking
glutMouseFunc(pickSquares); glutReshapeFunc(reshape);>
```

13.1.5.1 Указание с использованием множественных имен в иерархической модели

Множественные имена могут также использоваться для выбора частей иерархического объекта сцены. Например, если бы вы визуализировали сборочную линию автомобилей, вам могла бы потребоваться возможность, с помощью которой пользователь мог бы выбрать третий болт переднего левого колеса второй машины на линии. На каждом

уровне иерархии можно было бы применить свой набор имен для идентификации конкретного объекта на этом уровне. В качестве другого примера, одно имя могло бы идентифицировать всю молекулу среди других молекул, а другое – атомы внутри молекулы.

Пример **13-4** рисует автомобиль с четырьмя колесами, параллельно манипулируя стеком имен согласно объектной иерархии.

Пример **13-4.** Создание множественных имен

```
нарисовать_колесо_и_болты()
{
      long i;

      нарисовать_колесо();
      for(i=0;i<5;i++)
      {
            glPushMatrix();
                  glRotate(72.0*i,0.0,0.0,1.0);
                  glTranslate(3.0,0.0,0.0);
                  glPushName(i);
                        нарисовать_болт();
                  glPopName();
            glPopMatrix();
      }
}

нарисовать_корпус_колеса_и_болты()
{
      нарисовать_корпус();
      glPushMatrix();
            glTranslate(40,0,20);   /* позиция первого колеса */
            glPushName(1);          /* имя колеса №1 */
                  нарисовать_колесо_и_болты();
            glPopName();
      glPopMatrix();
      glPushMatrix();
            glTranslate(40,0,-20);  /* позиция второго колеса */
            glPushName(1);          /* имя колеса №2 */
                  нарисовать_колесо_и_болты();
            glPopName();
      glPopMatrix();

      /* нарисовать два последних колеса */
}
```

Пример **13-5** использует функции примера **13-4** для рисования трех разных автомобилей, пронумерованных **1**, **2** и **3**.

Пример **13-5.** Использование множественных имен

```
нарисовать_три_автомобиля()
{
      glInitNames();
      glPushMatrix();
            выполнить_перенос_к_позиции_первой_машины();
            glPushName(1);
                  нарисовать_корпус_колеса_и_болты();
            glPopName();
      glPopMatrix();
```

```
        glPushMatrix();
              выполнить_перенос_к_позиции_второй_машины();
              glPushName(2);
                    нарисовать_корпус_колеса_и_болты();
              glPopName();
        glPopMatrix();

        glPushMatrix();
              выполнить_перенос_к_позиции_третьей_машины();
              glPushName(3);
                    нарисовать_корпус_колеса_и_болты();
              glPopName();
        glPopMatrix();
  }
```

Если предположить, что производится указание, далее приводятся несколько примеров величин, которые могут быть возвращены, а также их интерпретация. В этих примерах возвращается максимум одна запись о попадании; ***d1*** и ***d2*** представляют значения глубины.

2 ***d1*** ***d2*** **2** **1** Машина **1**, колесо **1**

1 ***d1*** ***d2*** **3** Корпус машины **3**

3 ***d1*** ***d2*** **1** **1** **0** Болт **0** на колесе **1** машины **1**

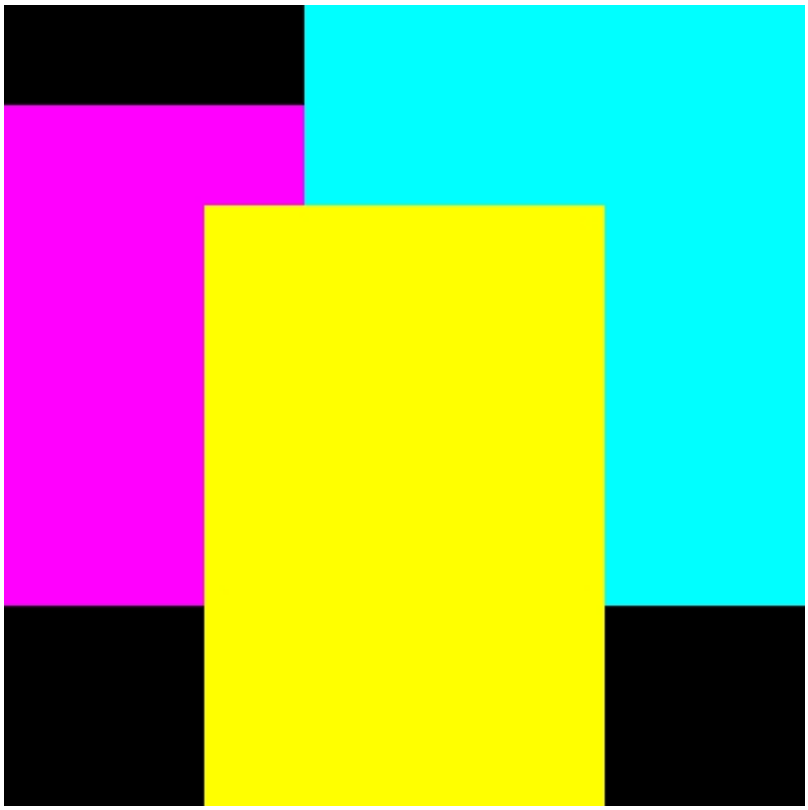
пусто Указание не коснулось ни одной из машин

В этих интерпретациях предполагается, что болт и колесо не оккупируют один и тот же регион указания. Пользователь же может легко выбрать и колесо и болт одновременно, в результате чего вы получите **2** попадания. Если вы получили несколько записей о попаданиях, вы должны решить, какую из них обрабатывать, базируясь, возможно, на величинах глубины, чтобы определить какое из попаданий находится ближе к точке наблюдения. Использование величин глубины исследуется в следующем разделе.

13.1.5.2 Указание и величина глубины

Пример **13-6** демонстрирует, как при указании использовать величины глубины для определения указанного объекта. В режиме визуализации эта программа рисует три перекрывающихся прямоугольника. При нажатии левой кнопки мыши, вызывается функция **pickRects()**. Эта функция получает позицию курсора, переходит в режим выбора, инициализирует стек имен и умножает матрицу указания на текущую матрицу ортографического проецирования. Попадание возникает для каждого треугольника, который находится под курсором мыши в момент нажатия левой кнопки. В конце, изучается содержимое буфера выбора для определения тех именованных объектов, которые находились внутри региона выбора вблизи курсора. Изображение, генерируемое программой, показано на рисунке **13-3**.

Рисунок **13-3**. Указание и величина глубины



В этой программе прямоугольники рисуются на разной глубине, то есть с разными величинами **z**. Поскольку для идентификации всех прямоугольников используется одно имя, может быть записано только одно попадание. Однако, в зависимости от того, сколько треугольников было указано и какие именно, это попадание имеет разные значения минимальной и максимальной координаты **z**.

Пример **13-6.** Указание и величина глубины: файл **pickdepth.cpp**

```
#include
#include
#include

void init()
{
      glClearColor(0.0,0.0,0.0,0.0);
      glEnable(GL_DEPTH_TEST);
      glShadeModel(GL_FLAT);
      glDepthRange(0.0,1.0);             /* отображение z по
умолчанию */
}

void drawRects(GLenum mode)
{
      if (mode==GL_SELECT)
            glLoadName(1);
      glBegin(GL_QUADS);
            glColor3f(1.0,1.0,0.0);
            glVertex3i(2,0,0);
            glVertex3i(2,6,0);
            glVertex3i(6,6,0);
            glVertex3i(6,0,0);
      glEnd();
      if (mode==GL_SELECT)
            glLoadName(2);
      glBegin(GL_QUADS);
```

```
            glColor3f(0.0,1.0,1.0);
            glVertex3i(3,2,-1);
            glVertex3i(3,8,-1);
            glVertex3i(8,8,-1);
            glVertex3i(8,2,-1);
      glEnd();
      if (mode==GL_SELECT)
            glLoadName(3);
      glBegin(GL_QUADS);
            glColor3f(1.0,0.0,1.0);
            glVertex3i(0,2,-2);
            glVertex3i(0,7,-2);
            glVertex3i(5,7,-2);
            glVertex3i(5,2,-2);
      glEnd();
}

void processHits(GLint hits,GLuint buffer[])
{
      int i,j;
      GLuint names, *ptr;

      printf("hits=%d\n",hits);
      ptr=(GLuint*)buffer;
      for (i=0;i< pre this pickSquares(int button,int state,int x,int
y) viewport[4]; if(button!="GLUT_LEFT_BUTTON" || state!="GLUT_DOWN)"
return; glGetIntegerv(GL_VIEWPORT,viewport); Create 5x5 pixel picking
region near cursor location
gluPickMatrix((GLdouble)x,(GLdouble)(viewport[3]-
y),5.0,5.0,viewport); reshape(int w, h) glViewport(0,0,w,h);
glutInitDisplayMode(GLUT_SINGLE|GLUT_RGB); glutCreateWindow(?Picking
glutMouseFunc(pickSquares); glutReshapeFunc(reshape);
drawRects(GL_SELECT); drawRects(GL_RENDER); glOrtho(0.0,8.0,0.0,8.0,-
0.5,2.5); glutInitWindowSize(400,400); with Depth Values?);>
```

13.1.6 Советы по написанию программ с использованием механизма выбора

Большинство программ, позволяющих пользователю интерактивно редактировать геометрические объекты, предоставляют механизм, позволяющий пользователю указать элемент или группу элементов для редактирования. Для программ двумерного рисования (вроде текстовых редакторов или программ форматирования страниц) может быть проще создать свой собственный механизм для обнаружения указанных объектов, чем использовать механизм выбора **OpenGL**. Часто, проще найти ограничивающие прямоугольники двумерных объектов и организовать их в подобие иерархии для ускорения поиска. Например, указание в стиле **OpenGL** может работать довольно медленно в программе, содержащей миллионы прямоугольников. Однако, если выравнивать прямоугольники на экране каким-либо образом и использовать только информацию об их границах, указание должно работать достаточно быстро. Кроме того, код для такого механизма проще писать и понимать.

Рассмотрим другой пример. Поскольку попадания происходят только для геометрических объектов, вам, возможно, понадобится собственная функция для указания символов текста. Установление текущей позиции растра является геометрической операцией, но создает только одну точку, которую можно выбрать. Эта точка находится, как правило, в нижнем левом углу текста. Если вашему редактору требуется возможность манипуляции с индивидуальными символами внутри текстовой строки, нужно использовать какой-либо другой механизм указания. Конечно, вы можете нарисовать небольшой прямоугольник вокруг каждого символа в режиме

выбора, но, скорее всего, будет проще обрабатывать текст каким-либо специальным образом.

Если вы решите использовать механизм указания **OpenGL**, организуйте программу и ее данные таким образом, чтобы рисовать списки объектов в режимах визуализации и выбора было достаточно просто. При таком подходе, когда пользователь выберет что-либо на экране, вы сможете использовать в режиме выбора тот же код, который использовался для фактического рисования сцены. Кроме того, заранее определитесь с тем, позволять ли пользователю выбирать несколько объектов одновременно. Одним из способов сделать это является хранение для каждого объекта бита информации, показывающего выбран объект или нет (однако при таком методе вам нужно пройти через весь массив битов для выявления выбранных объектов). Вы хранить список указателей на объекты для ускорения этого поиска. Хранение бита выбора может быть полезно и в случае, если вы хотите отображать выбранные объекты как-либо иначе (другим цветом или внутри параллепипеда выбора). Наконец, определитесь с пользовательским интерфейсом для выбора. Вы можете позволить пользователю:

- выбирать элемент
- выбирать группу элементов с помощью резинового контура
- добавлять элемент к группе выбранных
- добавлять группу элементов к выбранным
- удалить элемент из выбранных
- выбирать один элемент из группы перекрывающихся элементов

Типичное решение для двумерной программы может заключаться в следующем:

1. Весь выбор осуществляется с помощью курсора мыши и ее левой кнопки. В дальнейшем описании «курсор» всегда означает курсор мыши, а «кнопка» – левую кнопку мыши.
2. Кликанье на элементе выбирает его и отменяет выбор всех остальных выбранных в данный момент элементов. Если курсор находится поверх нескольких элементов, выбирает наименьший из них. (В трех измерения существует множество других стратегий для устранения неоднозначности выбора.)
3. Кликанье в точке экрана, где нет объектов, перемещение мыши с нажатой кнопкой и отпускание кнопки приводит к выбору всех объектов, находящихся внутри экранного прямоугольника, чьими углами являются точка, где кнопка была нажата, и точка, где кнопка была отпущена. Это называется выбором с помощью *резинового контура*. При этом отменяется выбор всех элементов находящихся вне прямоугольника резинового контура. (Вы должны решить, должен ли элемент быть выбран, только если он целиком попадает в прямоугольник или если любая его часть попадает в прямоугольник контура. Обычно первый вариант работает наилучшим образом.)
4. Если пользователь кликнул в объект, который не выбран, и одновременно держал нажатой кнопку **Shift**, объект добавляется к списку выбранных. Если указанный объект был выбранным, его следует удалить из списка выбранных.
5. Если резиновый контур растягивается при нажатой кнопке **Shift**, объекты внутри контура добавляются к выбранным.
6. Часто бывает трудно использовать резиновый конур в сильно загроможденных областях экрана. Когда нажимается кнопка (то есть происходит кликанье) курсор может находиться над каким-либо элементом, и в принципе, должен быть выбран этот элемент. Однако типичный интерфейс пользователя интерпретирует нажатие кнопки и перемещение мыши как растягивание резинового контура. Чтобы разрешить эту проблему, вы можете реализовать возможность принудительного запуска резинового контура с помощью удержания какой-либо кнопки (например, **Alt**). При таком подходе, для растягивания контура нужно выполнить следующие операции: нажать кнопку **Alt**, растянуть контур, отпустить кнопку **Alt**. В течение удержания **Alt** следует игнорировать элемент, на котором непосредственно произошел щелчок.

7. Если кнопка **Shift** нажата в течение выделения элементов резиновым контуром, эти элементы должны быть добавлены к уже выбранным.
8. и, наконец, если пользователь кликает на нескольких объектах, выделяйте только один из них. Если курсор не двигался (или сдвигался не более, чем на один пиксель), и пользователь кликает снова, в том же месте, отмените выбор первоначально выбранного элемента и выберите другой, из находящихся под курсором. Используйте повторные клики в одной точке для циклического перехода по всем возможностям.

В определенных ситуациях могут применяться разные правила. В текстовом редакторе вам, вероятно, не придется волноваться о том, что символы находятся один поверх другого, а выбор нескольких символов предусматривает только соседние символы в документе. Таким образом, вам нужно определить только первый и последний символы для идентификации всей выборки. Кроме того, в случае текста иногда легче идентифицировать промежутки между символами, а не сами символы. Такой подход позволяет вам реализовать пустую выборку, когда ее начало и конец находятся в одном и том же месте; он также позволяет вам поместить курсор перед первым символом текста и после последнего символа без дополнительного кодирования.

В трехмерном редакторе вы можете предоставить возможность поворачивать и перемещать выборки, так что нет смысла циклически проходить через их возможные варианты. С другой стороны выбор в трехмерном пространстве довольно сложен, поскольку координаты курсора на экране не предоставляют информации о глубине.

13.2 Отклик

Откат похож на выбор в том, что, когда вы находитесь в обоих режимах, реальный вывод пикселей не осуществляется, и изображение на экране остается неизменным. Рисования не происходит, вместо этого информация о примитивах, которые должны быть нарисованы, возвращается приложению. Ключевое отличие между режимами выбора и отката заключается в типе информации, которая отсылается приложению. В режиме выбора возвращают ассоциированные имена в виде массива целочисленных величин. В режиме отката информация о преобразованных примитивах возвращается в виде массива чисел с плавающей точкой. Величины, отсылаемые приложению в массиве отката, состоят из элемента, идентифицирующего тип обработанного и преобразованного примитива (точка, линия, полигон, изображение или битовая карта), за которым следуют вершины, цвета и другая информация об этом примитиве. Величины возвращаются после того, как они были преобразованы освещением и видовыми операциями. Режим отклика инициализируется вызовом команды **glRenderMode()** с аргументом **GL_FEEDBACK**.

Далее приводится процедура входа и выхода из режима отклика.

1. Вызовите **glFeedbackBuffer()**, чтобы задать массив, в котором будут сохраняться данные отклика. Аргументы этой команды задают тип и количество информации, которые будут сохраняться в массиве.
2. Вызовите **glRenderMode()**, чтобы войти в режим отклика. (На данный момент вы можете игнорировать значение, возвращенное **glRenderMode().**) После этого шага примитивы не будут растеризоваться в пиксели, и содержимое буфера кадра не изменяется до тех пор, пока вы не выйдите из режима отклика.
3. Нарисуйте ваши примитивы. Между вызовами команд рисования вы можете один или несколько раз вызвать команду **glPassThrough()** для добавления маркеров к данным отклика, чтобы упростить дальнейший разбор.
4. Выйдите из режима отклика, вызвав **glRenderMode()** с аргументом **GL_RENDER**, если вы хотите вернуться к режиму визуализации. Целочисленная величина, возвращенная **glRenderMode()**, соответствует числу записей, занесенных в буфер отклика.
5. Обработайте данные в массиве отклика.

```
void glFeedbackBuffer (GLsizei size, GLenum type, GLfloat* buffer);
```

Устанавливает буфер для данных отклика: ***buffer*** должен быть указателем на массив, где будут сохраняться данные. Аргумент ***size*** индицирует максимальное число величин, которые могут быть сохранены в массиве. Аргумент ***type*** задает тип информации, которая будет сохраняться в массиве для каждой вершины; его возможные значения и их смысл показаны в таблице **13-1**. **glFeedbackBuffer()** должна быть вызвана до входа в режим отклика. В таблице **13-1** ***k*** равно **1** в индексном режиме и **4 –** в режиме **RGBA**.

Таблица **13-1.** Значения аргумента **type** команды **glFeedbackBuffer()**

Аргумент *type*	**Координаты**	**Цвет**	**Текстура**	**Всего величин**
GL_2D	***x*, *y***	**-**	**-**	**2**
GL_3D	***x*, *y*, *z***	**-**	**-**	**3**
GL_3D_COLOR	***x*, *y*, *z***	***k***	**-**	**3+*k***
GL_3D_COLOR_TEXTURE	***x*, *y*, *z***	***k***	**4**	**7+*k***
GL_4D_COLOR_TEXTURE	***x*, *y*, *z*, *w***	***k***	**4**	**8+*k***

Замечание: Если поддерживается мультитекстурирование, режим отклика возвращает координаты текстуры только для текстурного блока 0.

13.2.1 Массив отклика

В режиме отклика каждый примитив, который должен быть растеризован (а также каждое обращение к командам **glBitmap()**, **glDrawPixels()** и **glCopyPixels()** в том случае если текущая позиция растра допустима) генерирует блок величин, которые копируются в буфер отклика. Количество величин определяется аргументом ***type***команды **glFeedbackBuffer()**. Используйте значение этого аргумента, соответствующее тому, что вы рисуете: **GL_2D** или **GL_3D** для освещенных двумерных или трехмерных примитивов, **GL_3D_COLOR** для освещенных трехмерных примитивов, **GL_3D_COLOR_TEXTURE** или **GL_4D_COLOR_TEXTURE** для освещенных, тектурированных трехмерных или четырехмерных примитивов.

Каждый блок величин отклика начинается с кода, позволяющего идентифицировать тип примитива. За этим кодом следуют величины, описывающие вершины примитива и ассоциированные с ними данные. Могут также присутствовать записи о прямоугольниках пикселей. Кроме того, в массиве могут возвращаться сквозные маркеры, созданные вами (эти маркеры описаны в следующем разделе). В таблице **13-2** приводится синтаксис массива отклика. Помните, что данные, ассоциированные с каждой возвращаемой вершиной, описаны в таблице **13-1**. Заметьте, что для полигона может быть возвращено ***n***вершин. Координаты ***x***, ***y***и ***z***, возвращаемые в массиве, являются оконными. Если же возвращается еще и ***w***, то координаты являются усеченными. Для битовых карт и изображений возвращаемые координаты являются текущей позицией растра на момент вывода соответствующего примитива. Обратите внимание, что **GL_LINE_RESET_TOKEN** возвращается только, когда ломаная прерывается.

Таблица **13-2.** Синтаксис массива отклика

Тип примитива	**Код**	**Ассоциированные данные**
Точка	**GL_POINT_TOKEN**	вершина
Линия	**GL_LINE_TOKEN** или **GL_LINE_RESET_TOKEN**	вершина вершина
Полигон	**GL_POLYGON_TOKEN**	**n** вершинавершина ... вершина
Битоваякарта	**GL_BITMAP_TOKEN**	вершина
Пиксельныйпрямоугольник	**GL_DRAW_PIXEL_TOKEN** или **GL_COPY_PIXEL_TOKEN**	вершина

Сквозной маркер	**GL_PASS_THROUGH_TOKEN**	число с плавающей точкой

13.2.2 Использование маркеров в режиме отклика

Отклик происходит после преобразований, расчета освещенности, удаления обратных граней и интерпретации полигонов в зависимости от команды **glPolygonMode()**. Также он происходит после того, как полигоны, имеющие более трех ребер, разбиваются на треугольники (если ваша реализация **OpenGL** визуализирует полигоны, производя такую декомпозицию). Таким образом, вам может быть сложно опознать те примитивы, что вы рисуете, в тех данных, которые были возвращены. Чтобы упростить процесс разбора данных отклика, в своем коде вставляйте маркеры с помощью команды **glPassThrough()**. Вы можете использовать маркеры для разделения величин отклика, относящихся к разным примитивам. Эта команда приводит к тому, что в массив отклика записывается код **GL_PASS_THROUGH_TOKEN**, за которым следует число с плавающей точкой, которое вы передали в качестве аргумента команды.

```
void glPassThrough (GLfloat token);
```

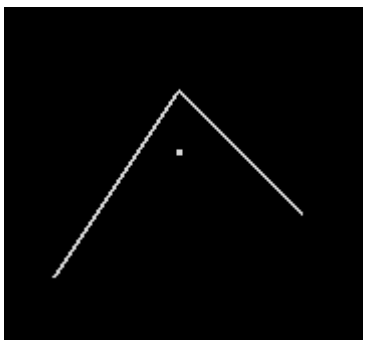
Если вызывается в режиме отклика, вставляет маркер в поток величин, записываемых в массив отклика. Маркер состоит из кода **GL_PASS_THROUGH_TOKEN**, за которым следует величина с плавающей точкой ***token***. Если команда вызывается не в режиме отклика, она ничего не делает. Вызов команды **glPassThrough()** между командами **glBegin()** и **glEnd()** генерирует ошибку **GL_INVALID_OPERATION**.

13.2.3 Пример использования отклика

Пример **13-7** демонстрирует использование режима отклика. Эта программа рисует освещенную трехмерную сцену в нормальном режиме визуализации. Затем она входит в режим отклика и перерисовывает сцену. Поскольку программа рисует освещенную нетекстурированную трехмерную сцену, типом данных отклика является **GL_3D_COLOR**, а поскольку она работает в режиме **RGBA**, каждая неотсеченная вершина генерирует **7** величин в массиве отклика: ***x***, ***y***, ***z***, ***r***, ***g***, ***b*** и ***a***.

В режиме отклика программа рисует две линии как часть ломаной, а затем вставляет сквозной маркер. Далее рисуется точка с координатами ***(-100.0***, ***-100.0***, ***-100.0)***, выпадающая из ортографического объема видимости и, таким образом, не генерирующая записей в массиве отклика. Наконец, вставляется еще один маркер и рисуется еще одна точка. Изображение, генерируемое программой, изображено на рисунке **13-4**.

Рисунок **13-4**. Пример использования режима отклика



Пример **13-7**. Режим отклика: файл **feedback.cpp**

```
#include
```

```
#include
#include

void init()
{
	glEnable(GL_LIGHTING);
	glEnable(GL_LIGHT0);
}

void drawGeometry(GLenum mode)
{
	glBegin(GL_LINE_STRIP);
		glNormal3f(0.0,0.0,1.0);
		glVertex3f(30.0,30.0,0.0);
		glVertex3f(50.0,60.0,0.0);
		glVertex3f(70.0,40.0,0.0);
	glEnd();

	if(mode==GL_FEEDBACK)
		glPassThrough(1.0);

	glBegin(GL_POINTS);
		glVertex3f(-100.0,-100.0,-100.0); //Будетотсечена
	glEnd();

	if(mode==GL_FEEDBACK)
		glPassThrough(2.0);

	glBegin(GL_POINTS);
		glNormal3f(0.0,0.0,1.0);
		glVertex3f(50.0,50.0,0.0);
	glEnd();
	glFlush();
}

void print3DcolorVertex(GLint size,GLint *count,GLfloat *buffer)
{
	int i;

	printf("   ");
	for(i=0;i<7;i++)
	{
		printf("%4.2f ",buffer[size-(*count)]);
		*count=*count-1;
	}
	printf("\n");
}

void printBuffer(GLint size,GLfloat *buffer)
{
	GLint count;
	GLfloat token;

	count=size;
	while(count)
	{
		token=buffer[size-count];
		count--;
		if(token==GL_PASS_THROUGH_TOKEN)
		{
			printf("GL_PASS_THROUGH_TOKEN\n");
			printf("   %4.2f\n",buffer[size-count]);
```

```
                  count--;
            }
            else
                  if(token==GL_POINT_TOKEN)
                  {
                        printf("GL_POINT_TOKEN\n");
                        print3DcolorVertex(size,&count,buffer);
                  }
                  else
                        if(token==GL_LINE_TOKEN)
                        {
                              printf("GL_LINE_TOKEN\n");
                              print3DcolorVertex(size,&count,buffer);
                              print3DcolorVertex(size,&count,buffer);
                        }
                        else
                              if(token==GL_LINE_RESET_TOKEN)
                              {
                                    printf("GL_LINE_RESET_TOKEN\n");

print3DcolorVertex(size,&count,buffer);

print3DcolorVertex(size,&count,buffer);
                              }
      }
}

void display()
{
      GLfloat feedBuffer[1024];
      GLint size;

      glMatrixMode(GL_PROJECTION);
      glLoadIdentity();
      glOrtho(0.0,100.0,0.0,100.0,0.0,1.0);

      glClearColor(0.0,0.0,0.0,0.0);
      glClear(GL_COLOR_BUFFER_BIT);
      drawGeometry(GL_RENDER);

      glFeedbackBuffer(1024,GL_3D_COLOR,feedBuffer);
      glRenderMode(GL_FEEDBACK);

      drawGeometry(GL_FEEDBACK);

      size=glRenderMode(GL_RENDER);
      printBuffer(size,feedBuffer);
}

int main(int argc, char* argv[])
{
      glutInit(&argc,argv);
      glutInitDisplayMode(GLUT_SINGLE|GLUT_RGB);
      glutInitWindowSize(100,100);
      glutInitWindowPosition(100,100);
      glutCreateWindow("Feedback Mode");
      init();
      glutDisplayFunc(display);
      glutMainLoop();
      return 0;
}
```

Запуск программы генерирует следующий вывод:

```
GL_LINE_RESET_TOKEN
30.00 30.00 0.00 0.84 0.84 0.84 1.0
50.00 60.00 0.00 0.84 0.84 0.84 1.0
GL_LINE_RESET_TOKEN
50.00 60.00 0.00 0.84 0.84 0.84 1.0
70.00 40.00 0.00 0.84 0.84 0.84 1.0
GL_PASS_THROUGH_TOKEN
1.00
GL_PASS_THROUGH_TOKEN
2.00
GL_POINT_TOKEN
50.00 50.00 0.00 0.84 0.84 0.84 1.0
```

Таким образом, ломаная, рисуемая следующими командами, отображается в два примитива:

```
glBegin(GL_LINE_STRIP);
      glNormal3f(0.0,0.0,1.0);
      glVertex3f(30.0,30.0,0.0);
      glVertex3f(50.0,60.0,0.0);
      glVertex3f(70.0,40.0,0.0);
glEnd();
```

Первый примитив начинается с кода **GL_LINE_RESET_TOKEN.** Это означает, что первым примитивом является сегмент линии, и что шаблон для линий сброшен. Второй примитив начинается с **GL_LINE_TOKEN,** то есть это тоже сегмент линии, но ломаная не прерывается и, следовательно, продолжается из той точки, где закончился предыдущий сегмент. Каждая из двух вершин этих линий генерирует по **7** величин в массиве отклика. Заметьте, что **RGBA** величины для всех четырех точек этих двух линий равны **(*0.84*, *0.84*, *0.84*, *1.0*)**, что является очень светлым оттенком серого цвета с максимальным значением альфа. Эти цветовые величины – результат взаимодействия вектора нормали и параметров освещения.

Поскольку между первым и вторым сквозными маркерами не было сгенерировано данных отклика, вы можете сделать вывод о том, что примитивы, нарисованные между первыми двумя вызовами **glPassThrough()** были отсечены по объему видимости. Наконец, рисуется точка с координатами **(*50.0*, *50.0*, *0.0*)**, и ассоциированные с ней данные заносятся в буфер отклика.

Замечание: И в режиме отклика, и в режиме выбора информация возвращается до выполнения всех тестов над фрагментами. Таким образом, данные об объектах, которые не были бы нарисованы из-за неудачи во время теста отреза, альфа, глубины или трафарета, тем не менее, обрабатываются и возвращаются в обоих режимах (выбора и отклика).

Глава 14. Трюки и советы

В этой главе обсуждаются различные техники, базирующиеся на командах **OpenGL,** которые иллюстрируют не очевидные примеры применения этих команд. Расположение этих примеров не имеет никакой закономерности, а сами примеры не связаны один с другим.

14.1 Обработка ошибок

Правда заключается в том, что ваша программа будет совершать ошибки. Использование функций обработки ошибок просто необходимо во время разработки и настоятельно рекомендуется для коммерческих приложений. (Если только вы не можете дать **100** процентной гарантии, что ваша программа не сгенерирует ошибку **OpenGL,** но

кто может это гарантировать**?) OpenGL** предоставляет простые функции обработки ошибок для базовой библиотеки **OpenGL** и библиотеки **GLU.**

Когда **OpenGL** обнаруживает ошибку**,** она записывает код текущей ошибки**.** Команда**,** сгенерировавшая ошибку игнорируется**,** так что она не оказывает никакого воздействия на состояние **OpenGL** или содержимое буфера кадра**. (**Однако**,** если записанной ошибкой была **GL_OUT_OF_MEMORY,** результат выполнения команды не определен**.)** После записи код текущей ошибки не изменяется**,** то есть в него другие ошибки не записываются до тех пор**,** пока вы не вызовите команду **glGetError(),** возвращающую код текущей ошибки**.** После того**,** как вы запросили и очистили код текущей ошибки или если ни одной ошибки не было**, glGetError()** будет возвращать значение **GL_NO_ERROR.**

```
GLenum glGetError (void);
```

Возвращает текущее значение флага ошибки**.** Когда ошибка происходит в **GL** или **GLU,** ее код записывается в флаг ошибки**.** Если возвращаемое значение равно **GL_NO_ERROR,** значит**,** ошибок не происходило с момента последнего вызова **glGetError()** или с момента инициализации **OpenGL.** Никакие другие ошибки не записываются до тех пор**,** пока не будет произведен вызов **glGetError(),** не вернется код ошибки**,** и флаг не будет сброшен в значение **GL_NO_ERROR.**

Настоятельно рекомендуется вызывать **glGetError()** хотя бы один раз в каждой функции **display()** вашего приложения**.** Основные коды ошибок **OpenGL** перечислены в таблице **14-1.**

Таблица **14-1.** Коды ошибок **OpenGL**

Код ошибкии	**Описание**
GL_INVALID_ENUM	аргумент типа **GLenum** вне допустимого диапазона
GL_INVALID_VALUE	численный аргумент вне допустимого диапазона
GL_INVALID_OPERATION	недопустимая операция для текущего состояния
GL_STACK_OVERFLOW	команда вызовет переполнение стека
GL_STACK_UNDERFLOW	команда вызовет извлечение или изменение значения в пустом стеке
GL_OUT_OF_MEMORY	недостаточно памяти для выполнения команды

Существует также **37** ошибок связанных с интерфейсом **GLU NURBS (**имеющих не описательные константные имена **GL_NURBS_ERROR1, GLU_NURBS_ERROR2** и так далее**); 14** ошибок тесселяции **(GLU_TESS_MISSING_BEGIN_POLYGON, GLU_TESS_MISSING_END_POLYGON, GLU_TESS_MISSING_BEGIN_CONTOUR, GLU_TESS_MISSING_END_CONTOUR, GLU_TESS_COORD_TOO_LARGE, GLU_TESS_NEED_COMBINE_CALLBACK,** а также еще **8** ошибок**,** начинающихся с префикса **GLU_TESS_ERROR*);** и **GLU_INCOMPATIBLE_GL_VERSION. GLU** также определяет коды ошибок **GLU_INVALID_ENUM, GLU_INVALID_VALUE** и **GLU_OUT_OF_MEMORY,** имеющие тот же смысл**,** что и связанные коды ошибок **OpenGL.**

Чтобы получить описательную строку**,** соответствующую коду ошибки **GL** или **GLU,** используйте функцию **GLU gluErrorString().**

```
const GLubyte* gluErrorString (GLenum errorCode);
```

Возвращает указатель на описательную строку**,** соответствующую коду ошибки **GL** или **GLU,** переданному в качестве аргумента ***errorCode*.**

В примере **14-1** показана простая функция обработки ошибок**.**

Пример **14-1.** Опрос и распечатка ошибки

```
GLenum errCode;
const GLubyte *errString;

if ((errCode=glGetError())!=GL_NO_ERROR
{
      errString=gluErrorString(errCode);
      fsprintf(stderr,”OpenGL error: %s\n”, errString);
}
```

Замечание: Приложение не должно изменять или удалять из памяти строку, возвращенную **gluErrorString()**.

14.2 Определение используемой версии

Переносимость приложений с использованием **OpenGL** является одним из привлекательнейших черт этого стандарта. Однако новые версии **OpenGL** предоставляют новые возможности, и, если вы используете эти возможности, у вас могут появиться проблемы с запуском вашей программы на более ранних версиях **OpenGL.** Кроме того, вы захотите реализовать ваше приложение таким образом, чтобы оно работало одинаково на различных реализациях **OpenGL.** Вы можете сделать текстурирование режимом по умолчанию на одной машине и ограничиться плоской закраской на другой. Для получения информации о версии **OpenGL** вы можете использовать команду **glGetString().**

```
const GLubyte* glGetString (GLenum name);
```

Возвращает указатель на строку, описывающий один из аспектов реализации **OpenGL.** Аргумент **name** может принимать значения **GL_VENDOR, GL_RENDERER, GL_VERSION** или **GL_EXTENSIONS. GL_VENDOR** возвращает название компании, ответственной за реализацию **OpenGL,** например:

```
NVIDIA Corporation
```

GL_RENDERER возвращает идентификатор визуализатора, обычно совпадающий с аппаратной платформой, например:

```
Voodoo550/AGP
```

или

```
GeForce4 Ti 4600/AGP/SSE2
```

О результате **GL_EXTENSIONS** можно подробно узнать в следующем разделе.

GL_VERSION возвращает строку, идентифицирующую номер версии реализации **OpenGL.** Строка версии состоит из следующих компонентов:

```
<номер версии><пробел><информация, зависящая от производителя>
```

Номер версии имеет форму

```
major_number.minor_number
```

или

```
major_number.minor_number.release_number
```

где каждое число может состоять из одной или более цифр. Информация, зависящая от производителя, является опциональной. Например, если реализация **OpenGL** создана компанией **XYZ**, возвращенная строка может быть следующей:

```
1.2.4 XYZ-OS 3.2
```

что означает, что это четвертый выпуск реализации **OpenGL** компании **XYZ**, и что эта реализация подчиняется спецификации **OpenGL** версии **1.2**. Это также может означать, что данный выпуск предназначен для какой-либо операционной системы версии **3.2**.

Другой способ получения информации о версии **OpenGL** заключается в использовании директивы препроцессора **#ifdef** для обнаружения символических констант **GL_VERSION_1_3**, **GL_VERSION_1_2** и **GL_VERSION_1_1**. Отсутствие константы **GL_VERSION_1_3** означает, что у вас версия **GL_VERSION_1_2**, **GL_VERSION_1_1** или **GL_VERSION_1_0**. Отсутствие константы **GL_VERSION_1_2** означает, что у вас версия **GL_VERSION_1_1** или **GL_VERSION_1_0** и так далее.

Замечание: Однако стоит помнить, что реальная версия используемой реализации OpenGL и версия, на которую рассчитан заголовочный файл (а константы определяются именно в нем) могут различаться. Например, на момент написания данного пособия в основном действовала спецификация OpenGL 1.3, однако корпорация Microsoft продолжала поставлять вместе со своими средствами разработки заголовочные файлы и библиотеки импорта, рассчитанные на OpenGL спецификации 1.1. Так что важно помнить, что команда **glGetString(GL_VERSION)** на этапе выполнения программы возвращает номер версии реальной реализации OpenGL, используемой в вашей системе, а метод с применением директивы #ifndef на этапе разработки позволяет определить только версию, на которую рассчитан заголовочный файл gl.h.

Замечание: При работе в клиент – серверной архитектуре (например, с OpenGL расширением системы X Window) клиент и сервер могут иметь разные версии. Если версия вашего клиента более поздняя, чем версия сервера, клиент может запросить операцию, которая сервером не поддерживается.

14.2.1 Номер версии GLU

Функция **gluGetString()** предназначена для опроса реализации **GLU** и похожа на команду **glGetString()**.

```
const GLubyte* gluGetString (GLenum name);
```

Возвращает указатель на строку, описывающий один из аспектов реализации **GLU**. Аргумент ***name*** может принимать значения **GLU_VERSION** или **GLU_EXTENSIONS**.

Обратите внимание, что функции **gluGetString()** не было в **GLU** версии **1.0**. Другим способом получения информации о **GLU** (точнее информации о версии, на которую рассчитан заголовочный файл и библиотека импорта) заключается в использовании директивы препроцессора **#ifndef** для поиска констант **GLU_VERSION_1_3**, **GLU_VERSION_1_2**, **GLU_VERSION_1_1** и **GLU_VERSION_1_0** по аналогии с константами **GL**.

Имейте в виду, что расширения **GLU** отличаются от расширений **OpenGL**.

14.2.2 Версия расширений оконной системы

Для расширений оконных систем, предназначенных для работы с **OpenGL**, таких как **GLX, WGL, PGL** и **AGL** существуют похожие функции **(**например, **glXQueryExtensionString()),** запрашивающие информацию о версии**.**

(Упомянутая **glXQueryExtensionString()** и связанные функции появились в **GLX 1.1** и не поддерживаются в **GLX1.0)**

14.3 Расширения стандарта

OpenGL имеет свою формальную напечатанную спецификацию, которая описывает операции, составляющие библиотеку**.** Индивидуальный производитель или группа таковых могут принять решение о добавлении к своим реализациям библиотеки дополнительной функциональности**.**

Имена функций и символических констант четко показывают, является ли конкретная возможность частью стандарта **OpenGL**, расширением специфичным для отдельного производителя или расширением **OpenGL** принятым **OpenGL ARB (Architecture Review Board –** Совет по Пересмотру Архитектуры**).**

Чтобы составить имя, специфичное для производителя, сам производитель добавляет идентификатор компании **(**заглавными буквами**)** и дополнительную информацию **(**например, тип компьютера**).** Например, если компания **XYZ** хочет добавить новую функцию и символическую константу, их имена могут быть в форме **glCommandXYZ()** и **GL_DEFINITION_XYZ,** соответственно**.** Если компания **XYZ** хочет разработать расширение доступное только на их **«**Специально Разработанной **(Special Developed)»** видеокарте, именами могут быть **glCommandXYZsd()** и **GL_DEFINITION_XYZ_SD.**

Если двое или более производителей согласны реализовать то же расширение, то к именам функций и констант добавляется более общий суффикс **EXT (glCommandEXT()** и **GL_DEFINITION_EXT).**

Похожим образом, если расширение было принято **OpenGL ARB,** к именам функций и констант добавляется суффикс **ARB (glCommandARB()** и **GL_DEFINITION_ARB).**

Если вы хотите выяснить, поддерживается ли конкретное расширение в вашей реализации, сначала вызовите **glGetString()** с аргументом **GL_EXTENSIONS,** а затем **– gluGetString()** с аргументом **GLU_EXTENSIONS.** Эти команды вернут полный список расширений, поддерживаемых вашей реализацией**.** В обеих строках индивидуальные имена расширений разделяются пробелами**.**

Первые расширения принятые **ARB** появились в **OpenGL** версии **1.2.** Если в реализации стандарта этой версии поддерживалось подмножество команд обработки изображений **(Imaging Subset),** в возвращенной строке расширений присутствовал фрагмент **GL_ARB_imaging.** Наличие фрагмента **GL_ARB_multitexture** свидетельствовало о том, что реализация поддерживает механизм мультитекстурирования, принятый **ARB.**

Если вы работаете с **GLU** версии **1.3** или выше, вы можете использовать функция **gluCheckExtension()** для определения того, поддерживается ли конкретное расширение**.**

```
GLboolean gluCheckExtension (char* extName, const GLubyte* extString);
```

Возвращает **GL_TRUE,** если строка ***extName*** обнаружена в ***extString***, и **GL_FALSE** в ином случае**.**

gluCheckExtension() может проверять как расширения **OpenGL**, так и **GLX**, и **GLU**.

Чтобы обнаружить поддерживается ли конкретное расширение при работе с **GLU** ранних версий, вы можете использовать код из примера **14-2**, который просматривает список расширений в поисках нужного. Функция **QueryExtension()** возвращает **GL_TRUE**, если расширение обнаружено и **GL_FALSE** в противном случае.

Пример **14-2.** Выявление наличия поддержки расширения **(**для кода с использованием **GLU 1.0**, **1.1** или **1.2)**

```
GLboolean QueryExtension(char *extName)

{
      char *p=(char*)glGetString(GL_EXTENSIONS);
      char *end;
      if(p==NULL)
            return GL_FALSE;
      end=p+strlen(p);
      while(p<end)
      {
            int n=strcspn(p," ");
            if((strlen(extName)==n) && (strncmp(extName,p,n)==0))
                  return GL_TRUE;
            p+=(n+1);
      }
      return GL_FALSE;
}
```

14.3.1 Расширения к стандарту для Microsoft Windows (WGL)

Если на платформе **Microsoft Windows** вы хотите получить доступ к функциям расширений, вы должны:

- создать условный код, базирующийся на символической константе, идентифицирующей расширение
- запросить обсуждаемую ранее строку расширений в период выполнения программы
- использовать функцию **wglProcAddress()** для обнаружения указателя на функцию расширения.

Пример **14-3** демонстрирует процесс обнаружения фиктивного расширения **glSpecialEXT()**. Заметьте, что условный код **(#ifndef)** проверяет наличие определения **GL_EXT_special**. Если расширение присутствует, то для указателя на функцию расширения определен специальный тип **PFNGLSPECIALEXTPROC**. Наконец, **wglGetProcAddress()** получает указатель на функцию.

Пример **14-3.** Обнаружение расширения **OpenGL** с помощью **wglGetProcAddress()**

```
#ifdef GL_EXT_special
      PFNGLSPECIALEXTPROC glSpecialEXT;
#endif

/* где-то в другом месте программы */
#ifdef GL_EXT_special
      glSpecialEXT=(
PFNGLSPECIALEXTPROC)wglGetProcAddress(“glSpecialEXT”);
      assert(glSpecialEXT!=NULL);
#endif
```

14.4 Пропускание света

Вы можете использовать шаблонирование полигонов для симуляции материала, пропускающего свет. Это решение особенно полезно для систем, в которых отсутствует аппаратура для цветового наложения. Поскольку размер рисунка шаблона полигона – **32** на **32** бита или **1024** бита в целом, вы можете пройти от непрозрачного до прозрачного материала за **1023** шага (на практике требуется значительно меньше шагов). Например, если вы хотите, чтобы поверхность пропускала **29** процентов света, создайте шаблон, в котором **29** процентов бит (это примерно **297** бит) будут установлены в **0**, а остальные в **1**. Даже если все ваши поверхности должны пропускать одинаковое количество света, не используйте один рисунок шаблона для всех, поскольку они накроют те же самые биты на экране. Создавайте новый рисунок для каждой поверхности, выбирая нужное количество бит, которые будут установлены в **0**, случайным образом.

Если вам не нравится эффект, получающийся при случайном выборе бит, вы можете использовать определенные рисунки, но они не работают так хорошо, в случае наложения прозрачных поверхностей друг на друга. Часто это не является проблемой, поскольку большинство сцен включают очень небольшое число прозрачных поверхностей. На изображении автомобиля с прозрачными окнами линия вашего обзора может пройти максимум через два окна, а чаще она проходит только через одно.

14.5 Эффект исчезающего изображения

Предположим, вы хотите, чтобы изображение постепенно растворилось в цвете фона. Определите серию рисунков полигональных шаблонов, в каждом из которых единице равно все больше и больше бит (то есть рисунки шаблона должны быть все плотнее и плотнее). Затем поочередно используйте эти шаблоны вместе с полигоном цвета фона достаточно большим, чтобы закрыть ту область, которая должна раствориться. Например, предположим, что вы хотите растворить изображение в черном цвете за **16** шагов. Сначала определите **16** различных массивов с рисунками шаблонов:

```
GLubyte stips[16][4*32];
```

Затем загрузите их таким образом, чтобы в первом рисунке в единицы была установлена **1/16** часть от **32x32** бит, а побитовое ИЛИ всех рисунков шаблона было равно всем единицам. Далее нужно применить следующий код:

```
нарисовать_изображение();
glColor3f(0.0,0.0,0.0);      /* установить текущий цвет в черный */
for(i=0;i<16;i++)
{
      glPolygonStipple(&stips[i][0]);
      нарисовать_большой_полигон_закрывающий_всю_область();
}
```

В некоторых реализациях **OpenGL** вы получите большее быстродействие, если предварительно скомпилируете рисунки шаблонов с списки отображения. Для этого во время инициализации сделайте нечто, вроде этого:

```
#define STIP_OFFSET 100
for(i=0;i<16;i++)
{
      glNewList(i+STIP_OFFSET,GL_COMPILE);
            glPolygonStipple(&stips[i][0]);
      glEndList();
```

```
}
```

Затем в предыдущем фрагменте кода замените строку

```
glPolygonStipple(&stips[i][0]);
```

на

```
glCallList(i);
```

Скомпилировав команды настройки рисунков шаблона в списки отображения, **OpenGL** может переработать данные в ***stips[ ][ ]*** в аппаратно-зависимую форму, требуемую для максимально быстрой работы с шаблонами.

Если и другое применение для этой техники. Предположим, что вы рисуете изменяющуюся картинку и хотите, чтобы предыдущие кадры не исчезали, а постепенно растворялись на фоне новых кадров. Например, вы имитируете планетарную систему и хотите, чтобы за планетами оставался постепенно исчезающий след, показывающий, где планета недавно проходила. Предположим, что след должен исчезать опять-таки за **16** шагов и используем те же рисунки шаблонов, что и в предыдущем примере **(**и те же списки отображения**)**. Тогда главный цикл может выглядеть следующим образом:

```
current_stipple=0;
while(1)                              /* вечный цикл */
{
      нарисовать_следующий_кадр();
      glCallList(current_stipple)++;
      if(current_stipple==16)
            current_stipple=0;
      glColor3f(0.0,0.0,0.0);      /* установить текущий цвет в
черный */
      нарисовать_большой_полигон_закрывающий_всю_область();
}
```

На каждой итерации цикла очищается **1/16** часть пикселей. Все пиксели, на которых планеты не будет в течение **16** итераций, будут закрашены черным цветом. Конечно, если в вашей системе есть аппаратная поддержка цветового наложения, будет значительно проще перед рисованием каждого кадра накладывать на изображение некоторое количество цвета фона.

14.6 Выбор объектов с помощью заднего буфера

Хотя **OpenGL** предоставляет мощный и гибкий механизм выбора, этот механизм иногда сложно использовать. Часто ситуация достаточно проста: ваше приложение рисует сцену, состоящую из некоторого количества объектов; пользователь указывает мышью на объект и приложение должно найти объект под курсором.

Один из способов сделать это требует, чтобы ваше приложение работало в режиме двойной буферизации. Когда пользователь кликает на объекте, приложение перерисовывает всю сцену в заднем буфере, но вместо использования обычных цветов объектов, оно кодирует идентификаторы отдельных объекта с помощью разных цветов. Затем приложение просто считывает пиксель под курсором и по значению цвета этого пикселя определяет номер выбранного объекта. Если для каждого статического изображения ожидается несколько кликов, вы можете считать все пиксели заднего буфера и изучить нужные, вместо того, чтобы индивидуально проверять пиксель на месте каждого клика.

Заметьте, что эта техника имеет над механизмом выбора то преимущество, что всегда выбирается только объект, находящийся ближе всего к наблюдателю, поскольку объекты рисуются один поверх другого на тех же пикселях. Поскольку изображение с фальшивыми цветами рисуется в заднем буфере, пользователь никогда его не увидит – вы можете снова перерисовать его (или скопировать из переднего буфера) до переключения буферов. В индексном цветовом режиме кодирование примитивно – передайте идентификатор объекта в качестве индекса. В режиме **RGBA** закодируйте биты идентификатора в компонентах **R**, **G** и **B**.

Имейте в виду, что если в вашей сцене очень много объектов, уникальные идентификаторы могут закончиться. Например, предположим, что вы работаете на системе с **4**-ех битными буферами для цветовых индексов (**16** возможных индексов) в обоих – переднем и заднем буферах, но в сцене присутствуют тысячи объектов. Чтобы решить такую проблему, можно осуществлять выбор в несколько проходов. Чтобы поговорить об этом конкретнее, предположим, что в вашей сцене менее **4096** объектов, то есть все идентификаторы могут быть закодированы **12** битами. Во время первого прохода нарисуйте сцену с использованием индексов, состоящих из **4**-ех старших бит идентификаторов, затем, во время второго и третьего проходов используйте индексы, состоящие из **4**-ех средних и младших битов идентификаторов. Во время каждого прохода, выясните значение индекса под курсором и, в конце, упакуйте их вместе – это даст вам идентификатор нужного объекта.

При таком подходе выбор занимает в три раза больше времени, но иногда это приемлемо. Заметьте, что после того, как вы получили **4** старших бита идентификатора, вы можете отбросить **15/16** всех объектов, так что на самом деле вам нужно нарисовать только **1/16** из них на втором проходе. Похожим образом после второго прохода можно отбросить **255/256** возможных объектов. Таким образом, первый проход занимает столько же времени, сколько рисование одного кадра, но второй и третий проходы могут пройти в **16** и **256** быстрее, соответственно.

Если вы пытаетесь создать переносимый код, работающий на различных системах, разбивайте ваши идентификаторы на фрагменты, которые могут уместиться в наименьший общий деноминатор этих систем (например, в байт). Также имейте в виду, что ваша система в **RGBA** режиме может производить автоматическое цветовое микширование. Если это так, вам придется его выключить.

14.7 Простые преобразования изображений

Если вы хотите нарисовать измененную версию битового изображения (возможно, просто растянутую или повернутую, а, возможно, значительно измененную с помощью математической функции), существует множество возможностей. Вы можете использовать изображение в качестве карты текстуры, позволяющей масштабировать, поворачивать и иначе искажать изображение. Если вам просто нужно изменить масштаб изображения, вы можете использовать команду **glPixelZoom()**.

Во многих случаях вы можете добиться хороших результатов, нарисовав изображение каждого пикселя в виде прямоугольника. Хотя эта методика не создает таких качественных изображений, как те, что получаются с помощью алгоритмов фильтрации, но она работает значительно быстрее.

Для конкретизации проблемы предположим, что оригинальное изображение имеет размеры ***m*** на ***n*** пикселей, с координатами между **[*0,m-1*]** и **[*0,n-1*]**. Пусть функциями искажения будут ***x(m,n)*** и ***y(m,n)***. Например, если искажение представляет собой просто изменение масштаба с фактором **3.2**, то ***x(m,n)=3.2m***, а ***y(m,n)=3.2n***. Следующий код рисует искаженной изображение:

```
glShadeModel(GL_FLAT);
glScale(3.2,3.2,1.0);
```

```
for(j=0;j<n;j++)
{
      glBegin(GL_QUAD_STRIP);
            for(i=0;i<=m;i++)
            {
                  glVertex2i(i,j);
                  glVertex2i(i,j+1);
                  set_color(i,j);
            }
      glEnd();
}
```

Этот код рисует каждый трансформированный пиксель цветом, равным цвету этого пикселя и масштабирует изображение на фактор **3.2**. Функция **set_color()** представляет собой функцию, устанавливающую цвет.

Далее приводится несколько более обобщенная версия, которая искажает изображение с использованием функций ***x(i,j)*** и ***y(i,j)***:

```
glShadeModel(GL_FLAT);
glScale(3.2,3.2,1.0);
for(j=0;j<n;j++)
{
      glBegin(GL_QUAD_STRIP);
            for(i=0;i<=m;i++)
            {
                  glVertex2i(x(i,j), y(i,j));
                  glVertex2i(x(i,j+1), y(i,j+1));
                  set_color(i,j);
            }
      glEnd();
}
```

И даже еще лучшее искаженное изображение может быть нарисовано с помощью следующего кода:

```
glShadeModel(GL_SMOOTH);
glScale(3.2,3.2,1.0);
for(j=0;j<n-1;j++)
{
      glBegin(GL_QUAD_STRIP);
            for(i=0;i<m;i++)
            {
                  set_color(i,j);
                  glVertex2i(x(i,j), y(i,j));
                  set_color(i,j+1);
                  glVertex2i(x(i,j+1), y(i,j+1));
            }
      glEnd();
}
```

Этот код плавно интерполирует цвет по всему четырехугольнику. Заметьте, что последняя версия производи на один четырехугольник меньше в каждом измерении, поскольку цвета изображения используются в вершинах полигонов. Кроме того, вы можете сглаживать изображение с помощью антиалиасинга и соответствующих факторов наложения **(GL_SRC_ALPHA, GL_ONE)**, чтобы еще больше улучшить качество изображения.

14.8 Отображение слоев

В некоторых приложениях (например, в программах создания полупроводников) вам требуется отобразить несколько слоев различных материалов и показать, где материалы пересекаются между собой.

В качестве простого примера, предположим, что у вас есть три различные субстанции, которые можно отобразить в виде слоев. В любой точке может оказаться одна из восьми комбинаций слоев, как показано в таблице **14-2**.

Таблица **14-2.** Восемь комбинаций слоев

	Слой 1	**Слой 2**	**Слой 3**	**Цвет**
0	отсутствует	отсутствует	отсутствует	черный
1	присутствует	отсутствует	отсутствует	красный
2	отсутствует	присутствует	отсутствует	зеленый
3	присутствует	присутствует	отсутствует	синий
4	отсутствует	отсутствует	присутствует	розовый
5	присутствует	отсутствует	присутствует	желтый
6	отсутствует	присутствует	присутствует	белый
7	присутствует	присутствует	присутствует	серый

Вы хотите, чтобы ваша программа отображала **8** разных цветов в зависимости от того, какие слои присутствуют в определенной точке. Одна из возможных цветовых схем показана в последней колонке таблицы **14-2**. Чтобы применить этот метод, используйте индексный цветовой режим и загрузите вхождения в палитру следующим образом: **0 –** черным цветом, **1 –** красным, **2 –** зеленым и так далее. Обратите внимание, что если числа от **0** до **7** перевести в двоичную систему, то бит **4** будет установлен везде, где присутствует слой **3**, бит **2 –** везде, где присутствует слой **2**, а бит **1 –** везде, где присутствует слой **1**.

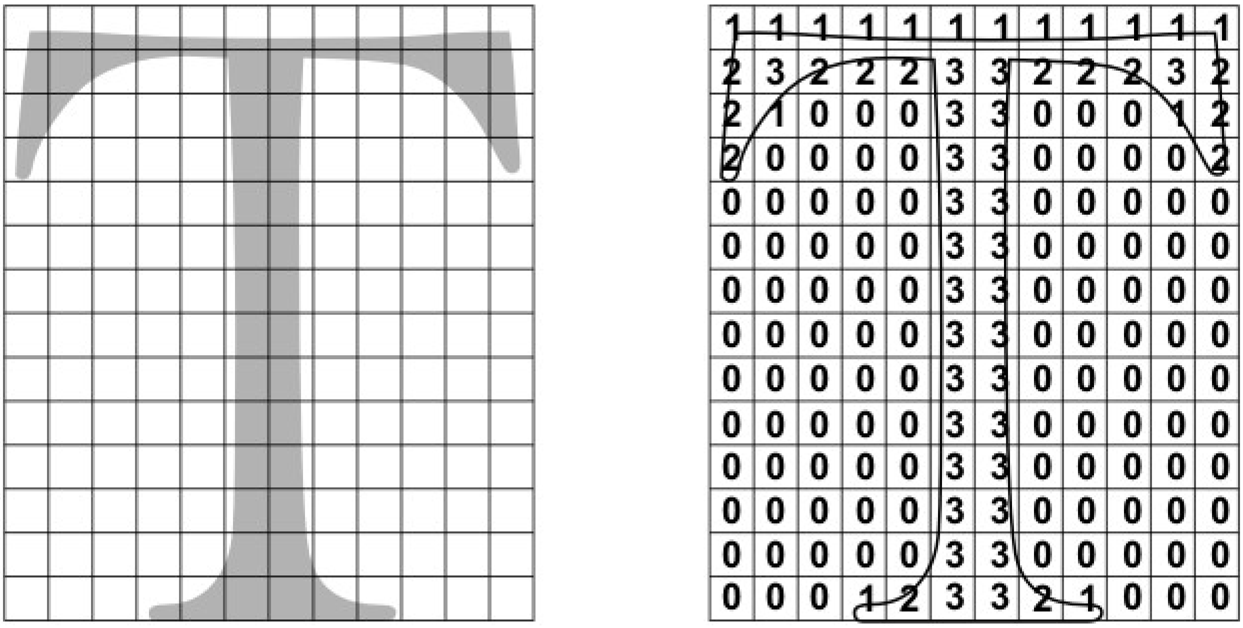
Для очистки окна, установите маску записи в **7** (все три слоя), а очищающий цвет в **0**. Чтобы нарисовать изображение, установите цвет в **7**, а маску записи в **n**, где **n –** это слой, в котором вы хотите рисовать. В других типах приложений может появиться необходимость удалить один из слоев. В этом случае вам нужно использовать маски записи так, как было оговорено ранее, но цвет устанавливать не в **7**, а в **0**.

14.9 Сглаженные символы

Использование стандартной техники отображения символов с помощью команды **glBitmap()** рисует каждый пиксель символа по принципу «все или ничего» -- либо пиксель рисуется, либо нет. Например, если вы рисуете черные символы на белом фоне, результирующие символы либо черные, либо белые, без намека на оттенок серого. Плавные изображения высокого качества могут быть получены при использовании промежуточных цветов (в данном примере оттенков серого.)

В предположении, что вы рисуете черные символы на белом фоне, представьте себе сильно увеличенное изображение пикселей, на которое наложено изображение символа с высоким разрешением, как показано на рисунке **14-1** слева.

Рисунок **14-1.** Сглаженные символы



Заметьте, что некоторые пиксели полностью содержатся внутри границы символа и должны быть нарисованы черным, некоторые полностью находятся вне границ символа и должны быть нарисованы белым, но многие пиксели было бы идеально нарисовать одним из оттенков серого, чтобы степень темноты оттенка соответствовала количеству черного на пикселе. Если использовать эту технику, изображение на экране будет иметь лучшее качество.

Если скорость и количество памяти не дают поводов для беспокойства, каждый символ можно нарисовать в виде небольшого изображения, а не битовой карты. Однако если вы работаете в режиме **RGBA**, этот метод потребует до **32** бит на каждый пиксель символа вместо **1** для стандартного символа. Альтернативно вы можете использовать для каждого пикселя **8**-битовый индекс и конвертировать его в **RGBA** с помощью цветовой таблицы во время переноса пикселей. Во многих случаях возможен компромисс, позволяющий вам рисовать символы с помощью нескольких оттенков серого цвета, и результирующее изображение при этом потребует всего **2** или **3** бита на пиксель.

Числа в правой части рисунка **14-1** показывают примерный проценты покрытия пикселя: **0** означает практическое отсутствие такового, **1** означает одну треть покрытия, **2** – две трети и **3** – полное покрытие пикселя символом. Если пиксели, отмеченные **0** рисовать белым, а пиксели, отмеченные **3** рисовать черным, а пиксели с номерами **1** и **2** можно рисовать оттенками **33%** и **66%** серого, соответственно, символы будут выглядеть достаточно хорошо. Для сохранения чисел **0**, **1**, **2** и **3** требуется только **2** бита, так что **2** бита позволяют сохранять **4** оттенка серого цвета.

Существует два основных метода реализации сглаженных символов, применимость которых зависит от того, в каком цветовом режиме вы работаете – в **RGBA** или в индексном.

В **RGBA** режиме создайте три различных битовых карты символа, соответствующих пикселям с номерами **1**, **2** и **3**. Установите цвет в белый и очистите экран. Установите цвет в **33%** оттенок серого **(RGB=(0.666, 0.666, 0.666))** и нарисуйте все пиксели, отмеченные **1** (с помощью одной из трех ранее подготовленных битовых карт). Затем установите **RGB=(0.333, 0.333, 0.333)** и нарисуйте вторую карту. Наконец, установите цвет в черный и нарисуйте третью карту. В принципе вы создали три разных шрифта и перерисовали строку символов три раза, причем каждый проход заполнил биты определенной цветовой плотности.

В индексном режиме вы можете сделать то же самое, но если вы решите использовать верно настроенную цветовую карту и маску записи, вы можете обойтись всего двумя битовыми картами и двумя проходами на строку. Вспомнив предыдущий пример, создайте первую битовую карту, содержащую все пиксели, отмеченные **1** или **3**. Создайте вторую битовую карту из пикселей, отмеченных **2** или **3**. Загрузите цветовую карту так, чтобы в **0** содержался белый, в **1** светло-серый, в **2 –** темно-серый, а в **3 –** черный. Установите цвет в **3**, а маску записи в **1** и нарисуйте первую карту. Затем измените маску на **2** и нарисуйте вторую. В тех местах, где на рисунке **14-1** стоят **0**, в буфере кадра ничего нарисовано не будет. Там, где стоят **1**, **2** и **3**, в буфере окажутся **1**, **2** и **3**.

Для данного примера с **4**-мя оттенками серого экономия достаточно мала **–** два прохода вместо трех. Если бы использовалось **8** оттенков вместо **4**, метод **RGBA** потребовал бы **7** проходов, а индексный только три. С **16** оттенками серого соотношение было бы **15** проходов к **4**.

14.10 Рисование круглых точек

Чтобы нарисовать почти круглые несглаженные точки, нужно активизировать сглаживание точек, выключить цветовое наложение и использовать альфа функцию **(**в альфа **–** тесте**)**, которая пропускает только фрагменты со значением альфа большим **0.5**.

14.11 Интерполяция изображений

Предположим, что у вас имеется два изображения **(**загруженных из файла или сгенерированных с помощью геометрии обычным путем**)**, и вы хотите, чтобы одно изображение плавно перешло в другое. Это можно легко сделать при помощи альфа компонента и соответствующих операций наложения. Скажем, вы хотите достигнуть полного перехода за **10** шагов, при этом изображение **A** будет полностью показано в кадре **0**, а изображение **B** будет полностью показано в кадре **9**. Очевидный подход заключается в том, чтобы в **i**-ом кадре рисовать изображение **A** с альфа равным **(9-*i*)/9**, а изображение **B** с альфа равным **i/9**.

Проблема этого метода заключается в том, что оба изображения должны рисовать в каждом кадре. Было бы быстрее нарисовать изображение **A** в кадре **0**. Затем для получения кадра **1** можно совместить **1/9** изображения **B** и **8/9** того изображение, которое уже нарисовано на экране. В кадре **2** нужно совместить **1/8** изображения **B** и **7/8** имеющегося изображения. Для кадра **3** нужно совместить **1/7** изображения **B** с **6/7** имеющегося изображения и так далее. На последнем шаге вам нужно нарисовать **1/1** изображения **B** и **0/1** имеющегося изображения, то есть вам нужно вывести только изображение **B**.

Чтобы понять, как это работает, предположим, что ваше изображение на кадре ***i*** представляет собой

$$\frac{(9-i)A}{9}+\frac{iB}{9}$$

и вы смешиваете **B/(9-i)** с **(8-i)/(9-i)** этого изображения. Тогда вы получите следующее:

$$\frac{B}{9-i}+\frac{8-i}{9-i}\left[\frac{(9-i)A}{9}+\frac{iB}{9}\right]=\frac{9-(i+1)A}{9}+\frac{(i+1)B}{9}.$$

14.12 Создание ярлыков

Предположим, что вы рисуете сложную трехмерную картинку с использованием буфера глубины для удаления невидимых поверхностей. Кроме того, предположим, что одна из частей вашей картинки состоит из двух объектов, лежащих на одной плоскости – **A** и **B**. При этом **B** является чем-то вроде ярлыка, который всегда должен появляться поверх **A**.

Первым, приходящим на ум подходом, является рисовать **B** после рисования **A**, установив функцию теста глубины таким образом, чтобы тест проходили фрагменты с глубиной большей или равной тому, что записано в буфере глубины **(GL_GEQUAL)**. Однако благодаря ограниченной точности дробного представления вершин, ошибки округления могут приводить к тому, что объект **B** иногда будет появляться перед объектом **A**, а иногда – за ним. Далее приводится одно из решений проблемы:

1. Запретить запись в буфер глубины и вывести объект **A**.
2. Разрешить запись в буфер глубины и вывести объект **B**.
3. Запретить запись в цветовой буфер и вывести объект **A**.
4. Разрешить запись в цветовой буфер.

Обратите внимание на то, что тест глубины активизирован и выполняется в течение всего процесса. На шаге **1 A** отображается там, где должно, но никакие величины не записываются в буфер глубины; следовательно, на шаге **2** гарантируется появление объекта **B**. Шаг **3** просто позволяет убедиться в том, что все значения глубины под **A** установлены в правильные значения, но, поскольку цветовой буфер закрыт для записи, пиксели на экране не изменяются. Наконец, шаг **4** возвращает системы к состоянию по умолчанию **(**оба буфера – цветовой и глубины – открыты для записи**)**.

Если имеется буфер трафарета, можно использовать более простую технику.

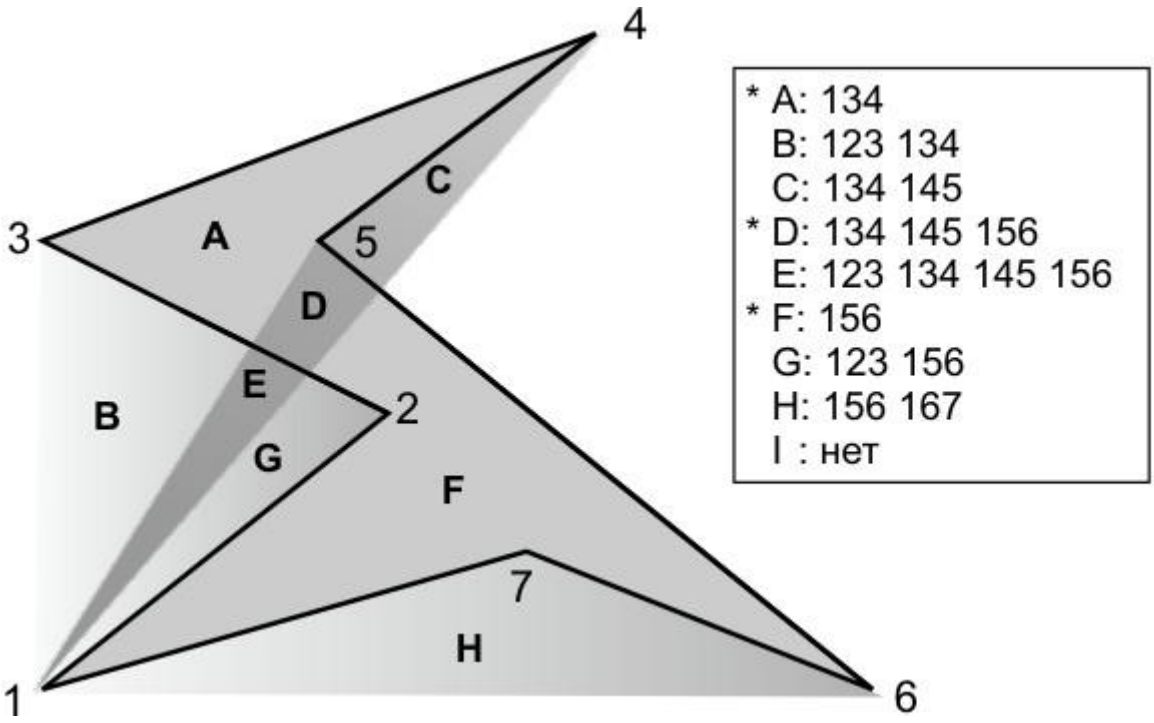
1. Настройте буфер трафарета, чтобы он записывал **1** в случае прохождения фрагментом теста глубины и **0** в противном случае. Нарисуйте **A**.
2. Настройте буфер трафарета, запретив изменение его величин и разрешив визуализацию только там, где значение в буфере трафарета равно **1**. Заблокируйте тест буфера глубины и изменение величин в нем. Нарисуйте **B**.

При использовании этого метода нет необходимости где-либо инициализировать содержимое буфера трафарета, поскольку значения трафарета всех интересующих пикселей **(**то есть тех, на которых будет отображаться **A)** будут установлены в процессе визуализации **A**. Не забудьте активизировать тест буфера глубины и деактивировать тест трафарета до того, как начнете рисовать остальные объекты сцены.

14.13 Рисование закрашенных вогнутых полигонов с помощью буфера трафарета

Возьмем вогнутый полигон ***1234567***, показанный на рисунке **14-2**. Представьте, что он рисуется в виде серии треугольников: ***123***, ***134***, ***145***, ***156*** и ***167***. Толстая линия показывает границу оригинального полигона. Рисование всех этих треугольников разделяет буфер на **9** областей ***A***, ***B***, ***C***, **...**, ***I***, причем ***I*** находится вне всех треугольников.

Рисунок **14-2.** Выпуклый полигон



В тексте к рисунку справа от имени каждой области приводится список треугольников, которые его накрывают. Области ***A***, ***D*** и ***F*** составляют оригинальный полигон; обратите внимание на то, что эти области накрыты нечетным количеством треугольников. Все остальные области накрыты четным (возможно нулевым) числом треугольников. Таким образом, чтобы отобразить вогнутый полигон, вам нужно вывести только области, ограниченные нечетным числом треугольников. Это может быть сделано с помощью буфера трафарета и двухпроходного алгоритма.

Сначала очистите буфер трафарета и запретите запись в цветовой буфер. Далее, по очереди нарисуйте каждый треугольник с использованием функции буфера трафарета **GL_INVERT. (**Для большего быстродействия используйте примитив **GL_TRIANGLE_FAN.)** Каждый раз, когда поверх пикселя буфет рисоваться треугольник, его значение трафарета буфет переключаться то в нелулевое, то в нулевое. После того, как все треугольники нарисованы, величина трафарета тех, пикселей, которые накрыты четным числом треугольников будет нулевой. Величина же трафарета пикселей накрытых нечетным числом треугольников будет не нулевой. Наконец, нарисуйте большой полигон над всей областью (или перерисуйте треугольники), но разрешите рисование только там, где значение в буфере трафарета не равно нулю.

Замечание: Существует обобщение приведенной техники, согласно которому вы не обязаны начинать с вершины полигона. Пусть в примере *1234567* точка *P* – любая точка, принадлежащая или не принадлежащая полигону. Нарисуйте треугольники *P12*, *P23*, *P34*, *P45*, *P56*, *P67* и *P71*. Области, накрытые нечетным числом треугольников, находятся внутри полигона, остальные области находятся снаружи полигона. Если *P* находится на одном из ребер полигона, один из треугольников будет пустым.

Эта техника может использоваться и для заполнения непростых полигонов (полигонов, чьи ребра пересекаются) и для заполнения полигонов с дырами. Следующий пример иллюстрирует обработку сложного полигона с двумя областями – одной четырехсторонней и одной пятисторонней. Кроме того, предположим, что в полигоне есть треугольная дыра и четырехугольная дыра (неважно в каких именно областях находятся дыры). Назовем две области ***abcd*** и ***efghi***, а дыры – ***jkl*** и ***mnop***. Пусть ***z*** – любая точка на плоскости. Нарисуем следующие треугольники:

zab zbc zcd zda zef zfg zgh zhi zie zjk zkl zlj zmn zno zop zpm

Пометьте области, накрытые нечетным числом треугольников, как находящиеся внутри полигона, а остальные – как находящиеся снаружи.

14.14 Нахождение областей пересечения

Если вы разрабатываете какую-либо механическую деталь, сделанную из меньших по размеру трехмерных частей, часто, вам необходимо отобразить те регионы, где части пересекаются. Во многих случаях наличие таких областей свидетельствует об ошибках дизайна (части машины соприкасаются там, где этого не должно быть). В случае движущихся частей этот процесс может быть еще более полезен, так как поиск регионов пересечения может производиться в течение всего цикла механической разработки. Метод, позволяющий сделать это, довольно сложен, его описание здесь приводится достаточно кратко.

Идея в том, чтобы пропустить произвольную отсекающую плоскость между объектами, которые вы хотите протестировать на пересечение, а далее определить те части этой плоскости, которые находятся внутри более чем одного объекта одновременно. Для статических изображений можно вручную сдвигать отсекающую плоскость для обнаружения областей пересечения; для динамических может быть проще использовать сетку отсекающих плоскостей для поиска всех возможных пересечений.

Нарисуйте каждый из проверяемых объектов и отсеките их плоскостью. Запомните, какие пиксели плоскости находятся внутри объекта, используя подсчет четное – нечетное в буфере трафарета, как объяснено в разделе «Рисование закрашенных вогнутых полигонов с помощью буфера трафарета». (Для правильно сформированных объектов точка находится внутри, если луч из этой точки в сторону наблюдателя пересекает нечетное число поверхностей объекта.) Чтобы найти пересечения, вам нужно найти пиксели в буфере кадра, где отсекающая плоскость находится внутри двух или более областей одновременно; другими словами на пересечении внутренних областей любой пары объектов.

Если на пересечение нужно протестировать множество объектов, сохраняйте один бит каждый раз, когда появляется пересечение, а другой бит – когда отсекающий буфер находится внутри любого из объектов (объединение внутренностей объектов). Для каждого нового объекта определите его внутреннюю область, найдите пересечение этой области с объединением внутренних областей уже протестированных объектов и учитывайте точки пересечения. Затем добавьте внутренние точки нового объекта к объединению внутренних областей других объектов.

Вы можете выполнить операции, описанные в предыдущем параграфе, используя разные биты буфера трафарета, совместно с разными операциями маскирования. На каждый пиксель требуется три бита буфера трафарета – один для переключения, определяющего внутреннюю область объекта; один для объединения всех внутренних областей, обнаруженных на данный момент; и один для областей, в которых на данный момент обнаружено пересечение. Чтобы сделать дискуссию более конкретной, предположим, что **1**-ый бит буфера трафарета используется для переключения между внутренними и внешними точками, **2**-ой – для объединения и **4**-ый – для пересечений. Для каждого выводимого объекта, очистите **1**-ый бит (используя маску трафарета **1** и очистив в **0**), затем переключайте **1**-ый бит, оставляя маску в состоянии **1** и, используя операцию трафарета **GL_INVERT**.

Вы можете найти пересечения и объединения битов в буфере трафарета, используя операции с трафаретом. Например, чтобы поместить в буфер **2** объединение битов в буферах **1** и **2**, маскируйте трафарет на эти **2** бита и нарисуйте что-либо поверх всего объекта, заставив функцию трафарета пропускать только фрагменты с ненулевым значением трафарета в буфере. Это произойдет, если установлен бит в буфере **1**, бит в буфере **2** или биты в обоих буферах. Если сравнение удается, записывайте **1** в буфер **2**. Кроме того, убедитесь в том, что запись в цветовой буфер заблокирована.

Пересечения рассчитываются аналогично – установите функцию на пропуск только тех фрагментов, чье значение в двух буферах трафарета равно **3 (**биты установлены и в буфере **1**, и в буфере **2)**. Запишите результат в нужный буфер.

14.15 Тени

Любую возможную проекцию трехмерного пространства на трехмерно пространство можно выполнить с помощью подходящей инвертируемой матрицы **4x4** и однородных координат. Если матрица не является инвертируемой, но имеет ранг **3**, она проецирует трехмерное пространство на двумерную плоскость. Любую проекцию такого типа можно выполнить с помощью подходящей матрицы **4x4**, имеющей ранг **3**. Чтобы найти тень произвольного объекта от произвольного источника света **(**возможно, лежащего в бесконечности**)** на произвольную плоскость, вам нужно найти матрицу, представляющую такую проекцию, умножить на нее матричный стек и нарисовать объект цветом тени. Имейте в виду, что вы должны спроецировать объект на каждую плоскость, которую вы называете **«**землей**»**.

В качестве простой иллюстрации, предположим, что источник света находится в начале координат, а плоскость **«**земли**»** представлена уравнением ***ax+by+cz+d=0***. Возьмем вершину ***S=(sx***, ***sy***, ***sz***, ***1)***. Луч из источника света, проходящий через точку ***S***, содержит все точки ξS, где ξ **--** произвольное вещественное число. точка, где луч пересекает плоскость, появляется тогда, когда:

$$\xi(a \cdot sx + b \cdot sy + c \cdot sz) + d = 0$$

то есть

$$\xi = \frac{-d}{a \cdot sx + b \cdot sy + c \cdot sz}.$$

Помещая последнюю формулу в уравнение линии, получаем:

$$\frac{-d(sx, sy, sz)}{a \cdot sx + b \cdot sy + c \cdot sz}$$

или

$$I = \left(\frac{-d \cdot sx}{a \cdot sx + b \cdot sy + c \cdot sz} \quad \frac{-d \cdot sy}{a \cdot sx + b \cdot sy + c \cdot sz} \quad \frac{-d \cdot sz}{a \cdot sx + b \cdot sy + c \cdot sz} \quad 1 \right)$$ где **I –** точка пересечения описанного луча с плоскостью **«**земли**»**, то есть одна из точек, составляющих тень.

Матрица, которая отображает точку **S** на точку **I** для любого **S**, равна:

$$\begin{bmatrix} -d & 0 & 0 & 0 \\ 0 & -d & 0 & 0 \\ 0 & 0 & -d & 0 \\ a & b & c & 0 \end{bmatrix}.$$

Эту матрицу можно использовать, если вы предварительно перенесете мировую систему координат, чтобы источник света оказался в начале координат.

Если свет исходит от источника, лежащего в бесконечности, все, что у вас есть – это точка **S** и направление **D=(*dx*, *dy*, *dz*)**. Точки вдоль луча получаются как $S + \xi D$.

Действуя, как и прежде, пересечение этого луча с плоскостью получается из уравнения

$$a(sx + \xi \cdot dx) + b(sy + \xi \cdot dy) + c(sz + \xi \cdot dz) + d = 0.$$

Разрешая это уравнение относительно ξ, помещая получившуюся формулу в уравнение луча и, далее, определяем проекционную матрицу

$$\begin{bmatrix} b \times dy + c \times dz & -b \times dx & -c \times dx & -d \times dx \\ -a \times dy & a \times dx + c \times dz & -c \times dy & -d \times dy \\ -a \times dz & -b \times dz & a \times dy + b \times dy & -d \times dz \\ 0 & 0 & 0 & a \times dx + b \times dy + c \times dz \end{bmatrix}.$$

При наличии уравнения плоскости и произвольного вектора направления, эта матрица работает без предварительных переносов чего-либо куда-либо.

14.16 Удаление невидимых линий

Если вы хотите нарисовать каркасный объект с удалением невидимых линий, один из подходов заключается в том, чтобы сначала нарисовать границы в виде линий, а затем закрасить внутренние области полигонов, приводя поверхности полигонов в соответствие с цветом фона. С включенным буфером глубины эта закраска внутренних областей накроет любые границы, которые должны быть загорожены гранями, находящимися ближе к наблюдателю. Этот метод работает до тех пор, пока внутренние области объекта полностью заключены в границы полигонов; на самом деле они могут накладываться друг на друга в различных местах **(**в результате будут получаться **«**прошитые полигоны**»).**

Существует простое двухпроходное решение, использующее либо полигональное смещение, либо буфер трафарета. Полигональное смещение обычно предпочтительнее, поскольку оно, как правило, работает быстрее, чем буфер трафарета. Далее описаны оба метода, так что вы можете увидеть, как работают оба подхода к решению проблемы.

14.16.1 Удаление невидимых линий с помощью полигонального смещения

Чтобы использовать полигональное смещение для удаления невидимых линий, объект рисуется дважды. Сначала выбранным цветом рисуются ребра полигонов с использованием полигонального режима **GL_LINE.** Затем те же полигоны рисуются цветом фона в режиме **GL_FILL** и с полигональным смещением, достаточным для того, чтобы сдвинуть их дальше от наблюдателя. Полигональное смещение позволяет избежать неприятных визуальных артефактов.

```
glEnable(GL_DEPTH_TEST);
glPolygonMode(GL_FRONT_AND_BACK,GL_LINE);
установить_цвет(передний_план);
```

```
нарисовать_объект_в_виде_закрашенных_полигонов();

glPolygonMode(GL_FRONT_AND_BACK,GL_FILL);
glEnable(GL_POLYGON_OFFSET_FILL);
glPolygonOffset(1.0,1.0);
установить_цвет(фон);
нарисовать_объект_в_виде_закрашенных_полигонов();
glDisable(GL_POLYGON_OFFSET_FILL);
```

Вам может понадобиться изменить настройки смещения **(**для более толстых линий**,** например**).**

14.16.2 Удаление невидимых линий с помощью буфера трафарета

Использование буфера трафарета для удаления невидимых линий **–** более сложная процедура**.** Для каждого полигона вы должны очистить буфер трафарета**,** а затем нарисовать его границу и в буфере кадра**,** и в буфере трафарета**.** Затем**,** при заполнении внутренних областей**,** разрешите рисование только там**,** где буфер трафарета по**-**прежнему чист**.** Во избежание очистки всего буфера трафарета для каждого полигона**,** можно записывать в буфер **0** с помощью той же границы полигона**.** В этом случае полную очистку буфера трафарета нужно будет произвести только один раз**.**

Например**,** следующий код представляет собой внутренний цикл**,** который вы можете использовать для удаления невидимых линий**.** Граница каждого полигона рисуется цветом переднего плана**,** закрашивается цветом фона**,** и затем граница рисуется еще раз**.** Буфер трафарета используется для предотвращения закрашивания границы цветом фона**.** Для оптимизации быстродействия параметры трафарета и цвета изменяются только дважды на итерацию оба раза с использованием одинаковых величин для рисования границ**.**

```
glEnable(GL_STENCIL_TEST);
glEnable(GL_DEPTH_TEST);
glClear(GL_STENCIL_BUFFER_BIT);
glStencilFunc(GL_ALWAYS,0,1);
glStencilOp(GL_INVERT,GL_INVERT,GL_INVERT);
установить_цвет(передний_план);
for(i=0;i<max;i++)
{
      нарисовать_границу_полигона(i);
      установить_цвет(фон);
      glStencilFunc(GL_EQUAL,0,1);
      glStencilOp(GL_KEEP, GL_KEEP, GL_KEEP);
      нарисовать_закрашенный_полигон(i);
      установить_цвет(передний_план);
      glStencilFunc(GL_ALWAYS,0,1);
      glStencilOp(GL_INVERT,GL_INVERT,GL_INVERT);
      нарисовать_границу_полигона(i);
}
```

14.17 Варианты применения текстурирования

Текстурирование является мощной техникой**,** которая может быть использована различными интересными способами**.** Вот несколько не очевидных вариантов применения текстурирования**.**

- Сглаженный текст – Создайте для каждого символа карту текстуры с относительно большим разрешением и наложите их на меньшие области (полигоны) с использованием текстурной фильтрации. Такой метод позволяет преобразовывать текст (поворачивать, масштабировать или искажать другими способами).
- Сглаженные линии – Метод работает так же, как и для текста: сделайте линии на текстуре шириной в несколько пикселей и используйте фильтрацию.
- Масштабирование и поворот изображений – Если вы поместите изображение в текстуру, а текстуру – на полигон, преобразования полигона будут отражаться и на изображении.
- Изгибание изображений – Сохраните изображение в карте текстуры, но наложите ее не на полигон, а на сплайновую поверхность (используйте вычислители). Изменение контрольных точек поверхности приведет к изменениям в изображении.
- Проектирование изображений – Поместите изображение в карту текстуры и спроецируйте его как прожектор, создав небольшой прожекторный эффект.

14.18 Рисование изображений, буферизованных по глубине

Для сложных статических изображений заднего плана время, требуемое на полную визуализацию из геометрических данных, может значительно превосходить время, необходимое для простого копирования того же плана, но в виде изображений. Если у вас фиксированный задний план и относительно простой изменяющийся передний план, вы можете решить, что стоит отобразить задний план и его величины глубины в виде изображения, а не просчитывать его из геометрических данных. Передний план также может состоять из элементов, которые долго просчитывать, но изображение и величины глубины которых известны. Вы можете визуализировать эти элементы в окружении с буфером глубины, используя двухпроходный алгоритм.

Например, если вы рисуете молекулярную модель, созданную из сфер, у вас может быть припасено изображение великолепно просчитанной сферы (и ассоциированные с ним значения глубины), вычисленной с использованием закраски Фонга или трассировки лучей или с помощью какого-либо другого метода не поддерживаемого **OpenGL** непосредственно. Чтобы нарисовать сложную сцену, вам могут потребоваться сотни таких сфер, совместно буферизуемых по глубине.

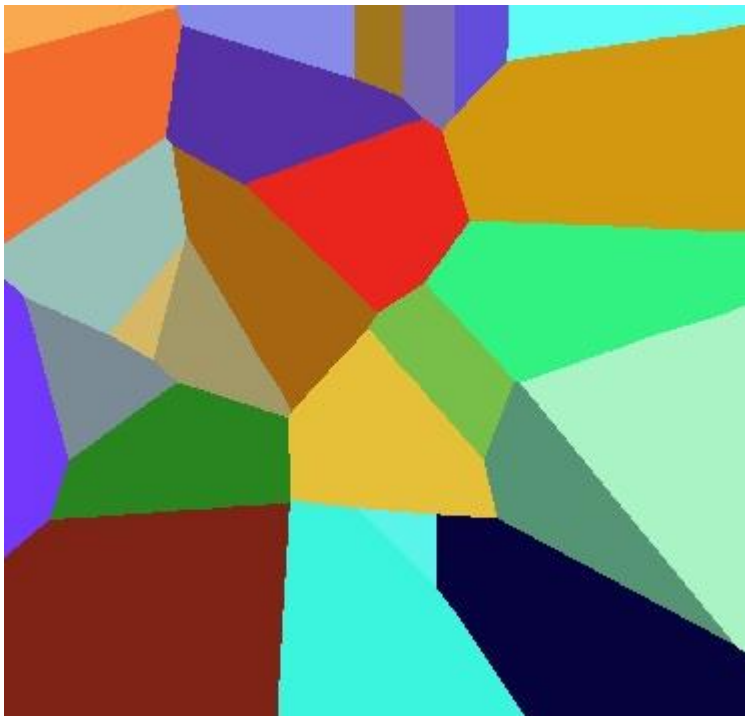
Чтобы добавить глубинированное изображение к сцене, сначала запишите его значения глубины в буфер глубины с помощью команды **glDrawPixels()**. Затем активизируйте тест глубины, установите маску записи в **0**, чтобы рисования не происходило, и активизируйте трафарет таким образом, чтобы рисование в буфер трафарета происходило всегда, когда происходит запись в буфер глубины.

Затем нарисуйте изображение в цветовом буфере, маскированном буфером трафарета (в который вы только что записали) таким образом, чтобы рисование происходило только в тех областях, где в буфере трафарета стоит **1**. Во время этого рисования установите функцию трафарета на обнуление буфера трафарета, чтобы он автоматически очищался к моменту добавления следующего изображения. Если объекты нужно придвинуть к наблюдателю или отодвинуть от него, нужно использовать ортографическую проекцию; в подобных случаях для перемещения глубинированного изображения используйте **GL_DEPTH_BIAS** и команду **glPixelTransfer*()**.

14.19 Многоугольники Вороного

Если взять множество **S** точек плоскости, многоугольником Вороного (или территорией Дирichлета) для произвольной точки **A** из **S** будет множество всех точек плоскости, находящихся ближе к точке **A**, чем к любой другой точке из **S**. Множество получившихся точек представляет собой решение многих проблем вычислительной геометрии. Многоугольники Вороного для набора точек показаны на рисунке **14-3**.

Рисунок **14-3.** Многоугольники Вороного



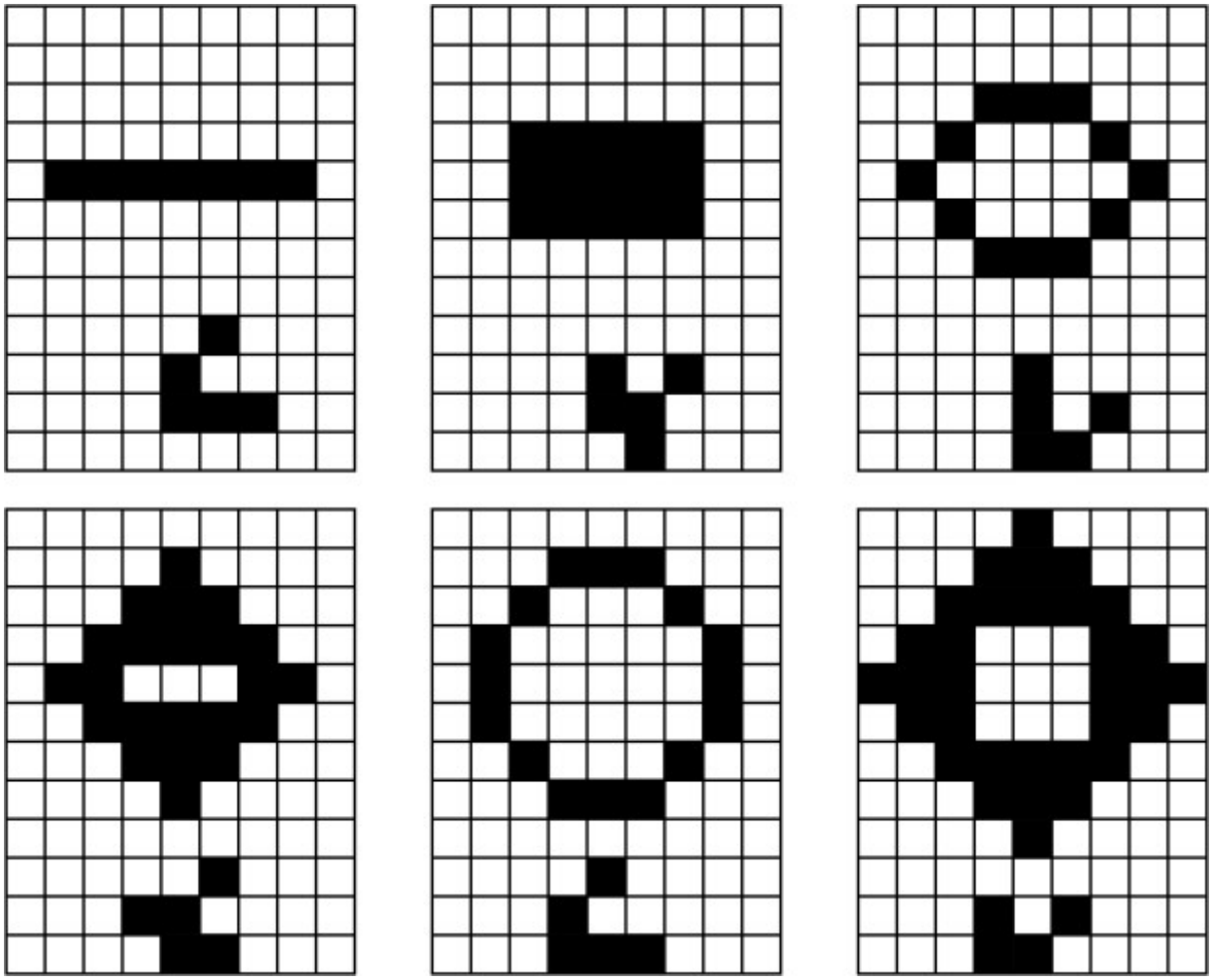
Если для каждой точки из **S** вы нарисуете глубинированный конус с вершиной в этой точке и цветом, отличающимся от цвета каждой из точек в **S**, территория Диричлета буфет нарисована этим цветом. Простейший путь сделать это заключается в предварительном расчете глубины конуса в виде изображения и использование этого изображения в качестве величин глубины, как описано в предыдущем разделе. Однако для рисования в буфер кадра изображение (в отличие от случая со сферами) вам не требуется. Пока вы рисуете в буфер глубины, используйте буфер трафарета для запоминания пикселей, на которых нужно рисовать: сначала очистите буфер, а затем записывайте ненулевые значения в тех местах, где тест глубины прошел. Чтобы нарисовать границы территорий Диричлета нарисуйте полигон размером со все окно, но разрешите рисование только там, где величины трафарета имеют ненулевое значение.

Вы можете сделать то же самое значительно проще, визуализировав конусы одного и того же цвета с простым буфером глубины, но хорошие конусы потребуют тысяч полигонов. Техника, описанная в этом разделе, позволяет визуализировать изображения высокого качества значительно быстрее.

14.20 «Жизнь» в буфере трафарета

«Игра в жизнь» («Game of Life») изобретенная Джоном Конвейем, проходит на прямоугольной сетке, где каждая клетка может быть «жива» или «мертва». Чтобы вычислить следующее поколение из текущего, сосчитайте количество живых соседей для каждой клетки (соседями каждой клетки являются **8** смежных клеток). Клетка жива в поколении ***n+1***, если она была жива в поколении ***n*** и имеет двух или трех живых соседей. Клетка также жива в поколении ***n+1***, если она была мертва в поколении ***n*** и имеет точно живых соседей. Во всех остальных случаях клетка мертва в поколении ***n+1***. В зависимости от начальной конфигурации эта игра генерирует потрясающе интересные рисунки. Рисунок **14-4** демонстрирует **6** поколений игры.

Рисунок **14-4. 6** поколений из «Игры в жизнь»



Одним из способов создания этой игры с помощью **OpenGL** является использование многопроходного алгоритма. Данные содержатся в цветовом буфере – по одному пикселю на клетку игровой сетки. Предположим, что цвет фона – черный (все нули), и, что цвет всех живых клеток – не черный (отличный от нуля). Очистите буферы глубины и трафарета в нули, установите маску записи буфера глубины в **0**, а функцию сравнения так, чтобы тест глубины проходили пиксели, имеющие значение глубины не равное хранимому в буфере. Для итерирования считайте изображение с экрана, разрешите запись в буфер глубины и установите функцию сравнения, чтобы она увеличивала значение в буфере трафарета везде, где проходит тест глубины, а иначе оставляла содержимое буфера трафарета неизменным. Запретите рисование в цветовой буфер.

Далее, нарисуйте изображение **8** раз, смещая его на один пиксель в каждом вертикальном, горизонтальном и диагональном направлениях. Когда вы закончите, буфер трафарета будет содержать числа, равные количеству живых соседей для каждого пикселя. Разрешите рисование в цветовой буфер, установите текущий цвет равным цвету живых клеток, а функцию трафарета установите таким образом, чтобы рисование происходило только там, где значение трафарета равно **3** (три живых соседа). Кроме того, если рисование происходит, уменьшайте величину в буфере трафарета. Затем нарисуйте прямоугольник, покрывающий все изображение – это окрасит каждую клетку, имеющую точно **3** живых соседа «живым» цветом.

На данный момент буфер трафарета содержит числа **0**, **1**, **2**, **4**, **5**, **6**, **7** и **8**. Клетки со значениями трафарета **0**, **1**, **4**, **5**, **6**, **7** и **8** должны быть очищены «мертвым» цветом. Установите функцию трафарета таким образом, чтобы рисование происходило только там, где значение трафарета не равно **2**, а операцию трафарета – на обнуление

значения в любом случае. Затем нарисуйте большой полигон «мертвого» цвета размером со все изображение. Итерация окончена.

Для того, чтобы получить пригодную версию такой программы, вам, возможно, стоит увеличить сетку, чтобы каждая клетка имела размер больший, чем один пиксель, поскольку с одним пикселем на клетку трудно различить рисунок.

14.21 Альтернативные варианты использования glDrawPixels() и glCopyPixels()

Вы можете считать, что **glDrawPixels()** предназначена только для рисования прямоугольных областей пикселей. Несмотря на то, что именно для этого команда применяется чаще всего, существуют и другие интересные варианты использования.

Видео – даже если ваша машина не обладает специальной видеоаппаратурой, вы можете отображать небольшие видео клипы, по очереди рисуя кадры с помощью **glDrawPixels()** в одной и той же области заднего буфера и затем переключая буферы. Размер кадра, при котором вы можете выводить их с допустимой скоростью, зависит от того, как быстро ваша аппаратура рисует, так что если вы хотите качественную картинку, вы можете быть ограничены размером **100x100 (**или меньше**).**

Кисть – в программе рисования форма вашей кисти может быть сымитирована с помощью величин альфа. Цвет краски представлен в виде цветовых величин. Чтобы нарисовать что-либо синей круглой кистью, рисуйте синий квадрат с помощью **glDrawPixels()**. При этом центр изображения квадрата должен иметь максимальное альфа, которое убывало бы с удалением от центра. Рисуйте, установив функцию наложения на использование значения альфа входящего цвета и **(1-**альфа**)** имеющегося в буфере цвета. Если все значения альфа кисти намного меньше единицы, то для получения четкого синего цвета вам придется проводить кистью по одному и тому же месту несколько раз. Если же все альфа величины кисти близки к **1**, первый же росчерк поглотит большую часть имеющегося рисунка.

Фильтрованное масштабирование – если вы изменяете размер пиксельного изображения с использованием дробного фактора, фильтрация, которую применит **OpenGL** может привести к серьезным визуальным неприятностям. Чтобы улучшить качество фильтрации, сдвигайте получившееся изображение на расстояние меньшее пикселя и перерисовывайте его несколько раз с использованием альфа – наложения для усреднения результирующих пикселей. Результатом будет фильтрованное масштабирование.

Транспозиция изображений – вы можете переключать изображения одинакового размера на месте с помощью **glCopyPixels()** и операции **XOR**. Этот метод позволяет избежать необходимости считывать изображения обратно в процессорную память. Если **A** и **B** представляют собой **2** изображения, операция выглядит так:

a. **A=A XOR B**
b. **B=A XOR B**
c. **A=A XOR B**

Приложение A. Переменные состояния

В этом приложении перечисляются переменные состояния **OpenGL**, их значения по умолчанию, их смысл и команды, с помощью которых можно получить их текущее значение.

Опросные команды

В дополнение к основным командам, позволяющим получить простые значения переменных состояния (таким как **glGetIntegerv()** или **glIsEnabled()**), существуют более специализированные команды для возврата более сложных значений. Ниже перечислены прототипы этих команд.

Для выяснения того, когда вы должны использовать эти команды и с какими константами, воспользуйтесь таблицами в следующем разделе.

```
void glGetClipPlane (GLenum plane, GLdouble *equation);
void glGetColorTable (GLenum target, GLenum pname, GLenum type,
GLvoid *table);
void glGetColorTableParameter{if}v (GLenum target, GLenum pname, TYPE
*params);
void glGetConvolutionFilter (GLenum target, GLenum format, GLenum
type, GLvoid *image);
void glGetConvolutionParameter{if}v (GLenum target, GLenum pname,
TYPE *params);
GLenum glGetError (void);
void glGetHistogram (GLenum target, GLboolean reset, GLenum format,
GLenum type, GLvoid *values);
void glGetHistogramParameter{if}v (GLenum target, GLenum pname, TYPE
*params);
void glGetLight{if}v (GLenum light, GLenum pname, TYPE *params);
void glGetMap{ifd}v (GLenum target, GLenum query, TYPE *v);
void glGetMaterial{if}v (GLenum face, GLenum pname, TYPE *params);
void glGetMinmax (GLenum target, GLboolean reset, GLenum format,
GLenum type, GLvoid *values);
void glGetMinmaxParameter{if}v (GLenum target, GLenum pname, TYPE
**params);
void glGetPixelMap {f ui us}v (GLenum map, TYPE *values);
void glGetPolygonStipple (GLubyte *mask);
void glGetSeparableFilter (GLenum target, GLenum format, GLenum type,
GLvoid *row, GLvoid *column, GLvoid *span);
const GLubyte* glGetString (GLenum name);
void glGetTexEnv{if}v (GLenum target, GLenum pname, TYPE *params);
void glGetTexGen{ifd}v (GLenum coord, GLenum pname, TYPE *params);
void glGetTexImage (GLenum target, GLint level, GLenum format, GLenum
type, GLvoid *pixels);
void glGetTexLevelParameter{if}v (GLenum target, GLint level, GLenum
pname, TYPE *params);
void glGetTexParameter{if}v (GLenum target, GLenum pname, TYPE
*params);
GLboolean glIsList (GLuint list);
GLboolean glIsTexture (GLuint texObject);
void gluGetNurbsProperty (GLUnurbsObj *nobj, GLenum property, GLfloat
*value);
const GLubyte* gluGetString (GLenum name);
void gluGetTessProperty (GLUtesselator *tess, GLenum which, GLdouble
*data);
```

Переменные состояния OpenGL

Следующие страницы содержат таблицы, в которых перечислены имена переменных состояния. Для каждой переменной в таблице есть описание, группа атрибутов, к которой она принадлежит, начальное или минимальное значение, а также рекомендуемая для получения значения переменной команда **glGet*()**. Для переменных состояния, значения которых могут быть получены с помощью команд **glGetBooleanv()**, **glGetIntegerv()**, **glGetFloatv()** или **glDoublev()** в таблице приводится только одна из этих команд – та, которая более всего подходит в соответствии с типом возвращаемого значения. (Некоторые переменные, связанные с вершинными массивами могут опрашиваться только с помощью **glGetPointerv().**) Эти

переменные состояния не могут быть получены с помощью команды **glIsEnabled().** Однако переменные, для которых **glIsEnabled()** указано в качестве команды опроса, могут быть получены так же и с помощью команды **glBooleanv()**, **glGetIntegerv()**, **glGetFloatv()** или **glDoublev().** Переменные состояния, для которых в таблице указана любая другая опросная команда, могут быть получены только с помощью этой команды.

Замечание: Когда вы опрашиваете состояние текстуры, например, GL_TEXTURE_MATRIX, в реализации OpenGL, поддерживающей расширение GL_ARB_multitexture, будут возвращены величины, относящиеся только к активному в данный момент текстурному блоку.

Если в таблице приводится одна или более групп атрибутов, это значит, что переменная состояния относится к этой группе или группам. Если группа атрибутов не указана – переменная не принадлежит ни к одной группе. Для сохранения и восстановления значений всех переменных состояния, относящихся к определенной группе или группам атрибутов, можно использовать команды **glPushAttrib()**, **glPopAttrib()**, **glPushClientAttrib()** и **glPopClientAttrib().**

Все опрашиваемые переменные состояния имеют начальное значение, однако, для переменных, зависящих от реализации, начальные значения могут не приводиться.

Таблица **A-1.** Текущие величины и связанные с ними данные

Переменная состояния	**Описание**	**Группа атрибутов**	**Начальное значение**	**Опросная команда**
GL_CURRENT_COLOR	Текущий цвет	текущие	**(1,1,1,1)**	**glGetIntegerv(), glGetFloatv()**
GL_CURRENT_INDEX	Текущий цветовой индекс	текущие	**1**	**glGetIntegerv(), glGetFloatv()**
GL_CURRENT_TEXTURE_COORDS	Текущие координаты текстуры	текущие	**(0,0,0,1)**	**glGetFloatv()**
GL_CURRENT_NORMAL	Текущая нормаль	текущие	**(0,0,1)**	**glGetFloatv()**
GL_CURRENT_RASTER_POSITION	Текущая позиция растра	текущие	**(0,0,0,1)**	**glGetFloatv()**
GL_CURRENT_RASTER_DISTANCE	Текущая дистанция растра	текущие	**0**	**glGetFloatv()**
GL_CURRENT_RASTER_COLOR	Цвет, ассоциированный с текущей позицией растра	текущие	**(1,1,1,1)**	**glGetIntegerv(), glGetFloatv()**
GL_CURRENT_RASTER_INDEX	Цветовой индекс, ассоциированный с текущей позицией растра	текущие	**1**	**glGetIntegerv(), glGetFloatv()**
GL_CURRENT_RASTER_TEXTURE_COORDS	Координаты текстуры, ассоциированные с текущей позицией растра	текущие	**(0,0,0,1)**	**glGetFloatv()**
GL_CURRENT_RASTER_POSITION_VALID	Бит допустимости позиции растра	текущие	**GL_TRUE**	**glGetBooleanv()**
GL_EDGE_FLAG	Флаг ребра **(**флаг границы**)**	текущие	**GL_TRUE**	**glGetBooleanv()**

Таблица **A-2.** Вершинные массивы

Переменная состояния	**Описание**	**Группа атрибутов**	**Начальное значение**	**Опросная команда**
GL_VERTEX_ARRAY	Активность массива координат вершин	вершинные массивы	**GL_FALSE**	**glIsEnabled()**

GL_VERTEX_ARRAY_SIZE	Число координат на вершину	вершинные массивы	**4**	**glGetIntegerv()**
GL_VERTEX_ARRAY_TYPE	Тип координат вершин	вершинные массивы	**GL_FLOAT**	**glGetIntegerv()**
GL_VERTEX_ARRAY_STRIDE	Смещение между вершинами	вершинные массивы	**0**	**glGetIntegerv()**
GL_VERTEX_ARRAY_POINTER	Указатель на массив координат вершин	вершинные массивы	**NULL**	**glGetPointerv()**
GL_NORMAL_ARRAY	Активность массива нормалей	вершинные массивы	**GL_FALSE**	**glIsEnabled()**
GL_NORMAL_ARRAY_TYPE	Тип координат нормали	вершинные массивы	**GL_FLOAT**	**glGetIntegerv()**
GL_NORMAL_ARRAY_STRIDE	Смещение между нормалями	вершинные массивы	**0**	**glGetIntegerv()**
GL_NORMAL_ARRAY_POINTER	Указатель на массив координат нормалей	вершинные массивы	**NULL**	**glGetPointerv()**
GL_COLOR_ARRAY	Активность массива **RGBA** цветов	вершинные массивы	**GL_FALSE**	**glIsEnabled()**
GL_COLOR_ARRAY_SIZE	Число компонент на вершину	вершинные массивы	**4**	**glGetIntegerv()**
GL_COLOR_ARRAY_TYPE	Тип цветовых компонентов	вершинные массивы	**GL_FLOAT**	**glGetIntegerv()**
GL_COLOR_ARRAY_STRIDE	Смещение между цветами	вершинные массивы	**0**	**glGetIntegerv()**
GL_COLOR_ARRAY_POINTER	Указатель на массив цветов	вершинные массивы	**NULL**	**glGetPointerv()**
GL_INDEX_ARRAY	Активность массива цветовых индексов	вершинные массивы	**GL_FALSE**	**glIsEnabled()**
GL_INDEX_ARRAY_TYPE	Тип индексов	вершинные массивы	**GL_FLOAT**	**glGetIntegerv()**
GL_INDEX_ARRAY_STRIDE	Смещение между индексами	вершинные массивы	**0**	**glGetIntegerv()**
GL_INDEX_ARRAY_POINTER	Указатель на массив индексов	вершинные массивы	**NULL**	**glGetPointerv()**
GL_TEXTURE_COORD_ARRAY	Активность массива координат текстуры	вершинные массивы	**GL_FALSE**	**glIsEnabled()**
GL_TEXTURE_COORD_ARRAY_SIZE	Число координат на элемент	вершинные массивы	**4**	**glGetIntegerv()**
GL_TEXTURE_COORD_ARRAY_TYPE	Тип координат текстуры	вершинные массивы	**GL_FLOAT**	**glGetIntegerv()**
GL_TEXTURE_COORD_ARRAY_STRIDE	Смещение между координатами	вершинные массивы	**0**	**glGetIntegerv()**
GL_TEXTURE_COORD_ARRAY_POINTER	Указатель на массив координат текстуры	вершинные массивы	**NULL**	**glGetPointerv()**
GL_EDGE_FLAG_ARRAY	Активность массива флагов ребра	вершинные массивы	**GL_FALSE**	**glIsEnabled()**
GL_EDGE_FLAG_ARRAY_STRIDE	Смещение между флагами	вершинные массивы	**0**	**glGetIntegerv()**
GL_EDGE_FLAG_ARRAY_POINTER	Указатель на массив флагов	вершинные массивы	**NULL**	**glGetPointerv()**
GL_CLIENT_ACTIVE_TEXTURE_ARB	Активный текстурный блок для спецификации массивом текстурных координат	вершинные массивы	**GL_TEXTURE0_ARB**	**glGetIntegerv()**

Таблица **A-3.** Преобразования

Переменная состояния	**Описание**	**Группа атрибутов**	**Начальное значение**	**Опросная команда**
GL_COLOR_MATRIX	Стек цветовых матриц	**-**	Единичная	**glGetFloatv()**
GL_MODELVIEW_MATRIX	Стек видовых матриц	**-**	Единичная	**glGetFloatv()**

GL_PROJECTION_MATRIX	Стек проекционных матриц	**-**	Единичная	**glGetFloatv()**
GL_TEXTURE_MATRIX	Стек текстурных матриц	**-**	Единичная	**glGetFloatv()**
GL_VIEWPORT	Начальная точка и размеры порта просмотра	порт просмотра	**-**	**glGetIntegerv()**
GL_DEPTH_RANGE	Разброс глубин	порт просмотра	**0, 1**	**glGetFloatv()**
GL_COLOR_STACK_DEPTH	Указатель стека цветовых матриц	**-**	**1**	**glGetIntegerv()**
GL_MODELVIEW_STACK_DEPTH	Указатель стека видовых матриц	**-**	**1**	**glGetIntegerv()**
GL_PROJECTION_STACK_DEPTH	Указатель стека проекционных матриц	**-**	**1**	**glGetIntegerv()**
GL_TEXTURE_STACK_DEPTH	Указатель стека текстурных матриц	**-**	**1**	**glGetIntegerv()**
GL_MATRIX_MODE	Текущий матричный режим	трансформация	**GL_MODELVIEW**	**glGetIntegerv()**
GL_NORMALIZE	Активность режима нормализации нормалей	трансформация/ включенные	**GL_FALSE**	**glIsEnabled()**
GL_RESCALE_NORMAL	Активность режима масштабирования нормалей	трансформация/ включенные	**GL_FALSE**	**glIsEnabled()**
GL_GET_CLIP_PLANE*i*	Коэффициенты ***i*-**ой пользовательской отсекающей плоскости	трансформация	**(0,0,0,0)**	**glGetClipPlane()**
GL_GET_CLIP_PLANE*i*	Активность ***i*-**ой пользовательской отсекающей плоскости	трансформация/ включенные	**GL_FALSE**	**glIsEnabled()**

Таблица **A-4.** Цвет

Переменная состояния	**Описание**	**Группа атрибутов**	**Начальное значение**	**Опросная команда**
GL_FOG_COLOR	Цвет тумана	туман	**(0,0,0,0)**	**glGetFloatv()**
GL_FOG_INDEX	Цветовой индекс тумана	туман	**0**	**glGetFloatv()**
GL_FOG_DENSITY	Плотность экспоненциальная тумана	туман	**1.0**	**glGetFloatv()**
GL_FOG_START	Начало линейного тумана	туман	**0.0**	**glGetFloatv()**
GL_FOG_END	Конец линейного тумана	туман	**1.0**	**glGetFloatv()**
GL_FOG_MODE	Режим тумана	туман	**GL_EXP**	**glGetIntegerv()**
GL_FOG	Активность тумана	туман/ включенные	**GL_FALSE**	**glIsEnabled()**
GL_SHADE_MODEL	Режим закраски	освещение	**GL_SMOOTH**	**glGetIntegerv()**

Таблица **A-5.** Освещение

Переменная состояния	**Описание**	**Группа атрибутов**	**Начальное значение**	**Опросная команда**
GL_LIGHTING	Активность расчета освещенности	освещение/ включенные	**GL_FALSE**	**glIsEnabled()**
GL_COLOR_MATERIAL	Активность режима цвета материала	освещение	**GL_FALSE**	**glIsEnabled()**
GL_COLOR_MATERIAL_PARAMETER	Параметр, совпадающий с текущим цветом для режима цвета материала	освещение	**GL_AMBIENT_ AND_DIFFUSE**	**glGetIntegerv()**

GL_COLOR_MATERIAL_FACE	Затрагиваемые грани для режима цвета материала	освещение	**GL_FRONT_ AND_BACK**	**glGetIntegerv()**
GL_AMBIENT	Фоновый цвет материала	освещение	**(0.2,0.2,0.2,1.0)**	**glGetMaterialfv()**
GL_DIFFUSE	Диффузный цвет материала	освещение	**(0.8,0.8,0.8,1.0)**	**glGetMaterialfv()**
GL_SPECULAR	Зеркальный цвет материала	освещение	**(0.0,0.0,0.0,1.0)**	**glGetMaterialfv()**
GL_EMISSION	Исходящий цвет материала	освещение	**(0.0,0.0,0.0,1.0)**	**glGetMaterialfv()**
GL_SHININESS	Зеркальная экспонента материала	освещение	**0.0**	**glGetMaterialfv()**
GL_LIGHT_MODEL_AMBIENT	Глобальный фоновый цвет	освещение	**(0.2,0.2,0.2,1.0)**	**glGetFloatv()**
GL_LIGHT_MODEL_LOCAL_VIEWER	Локален ли наблюдатель	освещение	**GL_FALSE**	**glGetBooleanv()**
GL_LIGHT_MODEL_TWO_SIZE	Включено ли двухстороннее освещение	освещение	**GL_FALSE**	**glGetBooleanv()**
GL_LIGHT_MODEL_COLOR_CONTROL	Отделяется ли зеркальный цвет	освещение	**GL_SINGLE_ COLOR**	**glGetIntegerv()**
GL_AMBIENT	Фоновая интенсивность источника ***i***	освещение	**(0.0,0.0,0.0,1.0)**	**glGetLightfv()**
GL_DIFFUSE	Диффузная интенсивность источника ***i***	освещение	**-**	**glGetLightfv()**
GL_SPECULAR	Зеркальная интенсивность источника ***i***	освещение	**-**	**glGetLightfv()**
GL_POSITION	Позиция источника ***i***	освещение	**(0.0,0.0,1.0,0.0)**	**glGetLightfv()**
GL_CONSTANT_ATTENUATION	Постоянный фактор ослабления	освещение	**1.0**	**glGetLightfv()**
GL_LINEAR_ATTENUATION	Линейный фактор ослабления	освещение	**0.0**	**glGetLightfv()**
GL_QUDRATIC_ATTENUATION	Квадратичный фактор ослабления	освещение	**0.0**	**glGetLightfv()**
GL_SPOT_DIRECTION	Прожекторное направление источника ***i***	освещение	**(0.0,0.0,-1.0)**	**glGetLightfv()**
GL_SPOT_EXPONENT	Прожекторная экспонента источника ***i***	освещение	**0.0**	**glGetLightfv()**
GL_SPOT_CUTOFF	Прожекторный угол источника ***i***	освещение	**180.0**	**glGetLightfv()**
GL_LIGHT*i*	Активность источника ***i***	освещение**/** включенные	**GL_FALSE**	**glIsEnabled()**
GL_COLOR_INDEXES	Цветовые индексы для освещения в индексном режиме	освещение**/** включенные	**0,1,1**	**glGetMaterialfv()**

Таблица **A-6.** Растеризация

Переменная состояния	Описание	Группа атрибутов	Начальное значение	Опросная команда
GL_POINT_SIZE	Размер точек	точка	**1.0**	**glGetFloatv()**
GL_POINT_SMOOTH	Активность режима сглаживания точек	точка**/**включенные	**GL_FALSE**	**glIsEnabled()**
GL_LINE_WIDTH	Толщина линий	линия	**1.0**	**glGetFloatv()**
GL_LINE_SMOOTH	Активность режима сглаживания	линия**/**включенные	**GL_FALSE**	**glIsEnabled()**

	линий			
GL_LINE_STIPPLE_PATTERN	Рисунок шаблона линий	линия	Единицы	**glGetIntegerv()**
GL_LINE_STIPPLE_REPEAT	Повторение шаблона линий	линия	**1**	**glGetIntegerv()**
GL_LINE_STIPPLE	Активность режима шаблонирования линий	линия**/** включенные	**GL_FALSE**	**glIsEnabled()**
GL_CULL_FACE	Активность режима удаления нелицевых граней	полигон**/** включенные	**GL_FALSE**	**glIsEnabled()**
GL_CULL_FACE_MODE	Какие грани нужно удалять	полигон	**GL_BACK**	**glGetIntegerv()**
GL_FRONT_FACE	Что считать лицевой гранью	полигон	**GL_CCW**	**glGetIntegerv()**
GL_POLYGON_SMOOTH	Активность режима сглаживания полигонов	полигон**/** включенные	**GL_FALSE**	**glIsEnabled()**
GL_POLYGON_MODE	Режим растеризации полигонов	полигон	**GL_FILL**	**glGetIntegerv()**
GL_POLYGON_OFFSET_FACTOR	Фактор полигонального смещения	полигон	**0**	**glGetFloatv()**
GL_POLYGON_OFFSET_BIAS	Скос полигонального смещения	полигон	**0**	**glGetFloatv()**
GL_POLYGON_OFFSET_POINT	Активность полигонального смещения для точек	полигон**/** включенные	**GL_FALSE**	**glIsEnabled()**
GL_POLYGON_OFFSET_LINE	Активность полигонального смещения для линий	полигон**/** включенные	**GL_FALSE**	**glIsEnabled()**
GL_POLYGON_OFFSET_FILL	Активность полигонального смещения для закрашенных областей	полигон**/** включенные	**GL_FALSE**	**glIsEnabled()**
GL_POLYGON_STIPPLE	Активность режима шаблонирования полигонов	полигон**/** включенные	**GL_FALSE**	**glIsEnabled()**
-	Рисунок шаблона полигона	шаблон полигона	Единицы	**glGetPolygonStipple()**

Таблица **A-7.** Текстурирование

Переменная состояния	**Описание**	**Группа атрибутов**	**Начальное значение**	**Опросная команда**
GL_TEXTURE_*x*	Активность текстурирования с размерностью ***x***	текстура**/**включенные	**GL_FALSE**	**glIsEnabled()**
GL_TEXTURE_BINDING_*x*	Объект текстуры прикреплен к **GL_TEXTURE_*x*** **(**где ***x* – 1D, 2D** или **3D)**	текстура	**GL_FALSE**	**glGetIntegerv()**
GL_TEXTURE	Изображение ***x*-D** текстуры на уровне ***i***	**-**	**-**	**glGetTexImage()**
GL_TEXTURE_WIDTH	Ширина ***i*-**го изображения	**-**	**0**	**glGetTexLevelParameter*()**

	***x*-D** текстуры			
GL_TEXTURE_HEIGHT	Высота ***i*-**го изображения ***x*-D** текстуры	**-**	**0**	**glGetTexLevelParameter*()**
GL_TEXTURE_DEPTH	Глубина ***i*-**го изображения ***x*-D** текстуры	**-**	**0**	**glGetTexLevelParameter*()**
GL_TEXTURE_BORDER	Ширина границы ***i*-**го изображения ***x*-D** текстуры	**-**	**0**	**glGetTexLevelParameter*()**
GL_TEXTURE_INTERNAL_FORMAT	Внутренний формат ***i*-**го изображения ***x*-D** текстуры	**-**	**1**	**glGetTexLevelParameter*()**
GL_TEXTURE_RED_SIZE	Разрешение красного компонента ***i*-**го изображения ***x*-D** текстуры	**-**	**0**	**glGetTexLevelParameter*()**
GL_TEXTURE_GREEN_SIZE	Разрешение зеленого компонента ***i*-**го изображения ***x*-D** текстуры	**-**	**0**	**glGetTexLevelParameter*()**
GL_TEXTURE_BLUE_SIZE	Разрешение синего компонента ***i*-**го изображения ***x*-D** текстуры	**-**	**0**	**glGetTexLevelParameter*()**
GL_TEXTURE_ALPHA_SIZE	Разрешение альфа компонента ***i*-**го изображения ***x*-D** текстуры	**-**	**0**	**glGetTexLevelParameter*()**
GL_TEXTURE_LUMINANCE_SIZE	Разрешение светлоты ***i*-**го изображения ***x*-D** текстуры	**-**	**0**	**glGetTexLevelParameter*()**
GL_TEXTURE_INTENSITY_SIZE	Разрешение интенсивности ***i*-**го изображения ***x*-D** текстуры	**-**	**0**	**glGetTexLevelParameter*()**
GL_TEXTURE_BORDER_COLOR	Цвет границы текстуры	текстура	**(0,0,0,0)**	**glGetTexParameter*()**
GL_TEXTURE_MIN_FILTER	Функция уменьшения текстуры	текстура	**GL_NEAREST_ MIPMAP_ LINEAR**	**glGetTexParameter*()**
GL_TEXTURE_MAG_FILTER	Функция увеличения текстуры	текстура	**GL_LINEAR**	**glGetTexParameter*()**
GL_TEXTURE_WRAP_*x*	Режим присоединения текстуры **(**где ***x* – S, T** или **R)**	текстура	**GL_REPEAT**	**glGetTexParameter*()**
GL_TEXTURE_PRIORITY	Приоритет текстурного объекта	текстура	**1**	**glGetTexParameter*()**
GL_TEXTURE_RESIDENT	Является ли текстура резидентной	текстура	по-разному	**glGetTexParameterfv()**
GL_TEXTURE_MIN_LOD	Минимальный уровень детализации	текстура	**-1000**	**glGetTexParameterfv()**

GL_TEXTURE_MAX_LOD	Максимальный уровень детализации	текстура	**1000**	**glGetTexParameterfv()**
GL_TEXTURE_BASE_LEVEL	Базовый текстурный массив	текстура	**0**	**glGetTexParameterfv()**
GL_TEXTURE_MAX_LEVEL	Максимальный уровень текстурного массива	текстура	**1000**	**glGetTexParameterfv()**
GL_TEXTURE_ENV_MODE	Режим наложения текстуры	текстура	**GL_MODULATE**	**glGetTexEnviv()**
GL_TEXTURE_ENV_COLOR	Цвет окружения текстуры	текстура	**(0,0,0,0)**	**glGetTexEnvfv()**
GL_TEXTURE_GEN_*x*	Активность автоматической генерации координат **(**где ***x*** **– S, T** или **R)**	текстура**/**включенные	**GL_FALSE**	**glIsEnabled()**
GL_EYE_PLANE	Коэффициенты уравнения плоскости для вычисления координат текстуры	текстура	**-**	**glGetTexGenfv()**
GL_OBJECT_PLANE	Коэффициенты уравнения плоскости в объектных координатах для вычисления координат текстуры	текстура	**-**	**glGetTexGenfv()**
GL_TEXTURE_GEN_MODE	Режим генерирования текстурных координат	текстура	**GL_EYE_LINEAR**	**glGetTexGeniv()**
GL_ACTIVE_TEXTURE_ARB	активный текстурный блок	текстура	**GL_TEXTURE0_ ARB**	**glGetIntegerv()**

Таблица **A-8.** Пиксельные операции

Переменная состояния	**Описание**	**Группа атрибутов**	**Начальное значение**	**Опросная команда**
GL_SCISSOR_TEST	Активность теста отреза	отрез**/** включенные	**GL_FALSE**	**glIsEnabled()**
GL_SCISSOR_BOX	Габариты прямоугольника для теста отреза	отрез	**-**	**glGetIntegerv()**
GL_ALPHA_TEST	Активность альфа теста	цветовой буфер**/**включенные	**GL_FALSE**	**glIsEnabled()**
GL_ALPHA_TEST_FUNC	Функция альфа теста	цветовой буфер	**GL_ALWAYS**	**glGetIntegerv()**
GL_ALPHA_TEST_REF	Сравниваемое значения для альфа теста	цветовой буфер	**0**	**glGetIntegerv()**
GL_STENCIL_TEST	Активность теста трафарета	буфер трафарета**/**включенные	**GL_FALSE**	**glIsEnabled()**
GL_STENCIL_FUNC	Функция для теста трафарета	буфер трафарета	**GL_ALWAYS**	**glGetIntegerv()**
GL_STENCIL_VALUE_MASK	Маска теста трафарета	буфер трафарета	Единицы	**glGetIntegerv()**
GL_STENCIL_REF	Сравниваемое	буфер трафарета	**0**	**glGetIntegerv()**

	значение теста трафарета			
GL_STENCIL_FAIL	Действие при провале теста трафарета	буфер трафарета	**GL_KEEP**	**glGetIntegerv()**
GL_STENCIL_PASS_DEPTH_FAIL	Действие при прохождении теста трафарета и провале теста глубины	буфер трафарета	**GL_KEEP**	**glGetIntegerv()**
GL_STENCIL_PASS_DEPTH_PASS	Действие при прохождении теста трафарета и прохождении теста глубины	буфер трафарета	**GL_KEEP**	**glGetIntegerv()**
GL_DEPTH_TEST	Активность теста глубины	буфер глубины**/**включенные	**GL_FALSE**	**glIsEnabled()**
GL_DEPTH_FUNC	Функция теста глубины	буфер глубины	**GL_LESS**	**glGetIntegerv()**
GL_BLEND	Активность цветового наложения	цветовой буфер**/**включенные	**GL_FALSE**	**glIsEnabled()**
GL_BLEND_SRC	Фактор источника цветового наложения	цветовой буфер	**GL_ONE**	**glGetIntegerv()**
GL_BLEND_DST	Фактор приемника цветового наложения	цветовой буфер	**GL_ZERO**	**glGetIntegerv()**
GL_BLEND_EQUATION	Уравнение цветового наложения	цветовой буфер	**GL_FUNC_ADD**	**glGetIntegerv()**
GL_BLEND_COLOR	Постоянный цвет наложения	цветовой буфер	**(0,0,0,0)**	**glGetFloatv()**
GL_DITHER	Активность микширования	цветовой буфер**/** включенные	**GL_TRUE**	**glIsEnabled()**
GL_INDEX_LOGIC_OP	Активность логических операций над цветовыми индексами	цветовой буфер**/** включенные	**GL_FALSE**	**glIsEnabled()**
GL_COLOR_LOGIC_OP	Активность логических операций над **RGBA** цветами	цветовой буфер**/** включенные	**GL_FALSE**	**glIsEnabled()**
GL_LOGIC_OP_MODE	Применяемая логическая операция	цветовой буфер	**GL_COPY**	**glGetIntegerv()**

Таблица **A-9.** Управление буфером кадра

Переменная состояния	**Описание**	**Группа атрибутов**	**Начальное значение**	**Опросная команда**
GL_DRAW_BUFFER	Буферы**,** выбранные для записи	цветовой буфер	**-**	**glGetIntegerv()**
GL_INDEX_WRITEMASK	Маска записи цветовых индексов	цветовой буфер	Единицы	**glGetIntegerv()**
GL_COLOR_WRITEMASK	Маска записи **RGBA** цветов	цветовой буфер	**GL_TRUE**	**glGetBooleanv()**
GL_DEPTH_WRITEMASK	Доступность буфера глубины для записи	буфер глубины	**GL_TRUE**	**glGetBooleanv()**
GL_STENCIL_WRITEMASK	Маска записи индексов трафарета	буфер трафарета	Единицы	**glGetIntegerv()**
GL_COLOR_CLEAR_VALUE	Очищающий цвет **(RGBA)**	цветовой буфер	**(0,0,0,0)**	**glGetFloat()**

GL_INDEX_CLEAR_VALUE	Очищающий цвет **(**индекс**)**	цветовой буфер	**0**	**glGetFloat()**
GL_DEPTH_CLEAR_VALUE	Очищающая величина для буфера глубины	буфер глубины	**1**	**glGetIntegerv()**
GL_STENCIL_CLEAR_VALUE	Очищающая величина для буфера трафарета	буфер трафарета	**0**	**glGetIntegerv()**
GL_ACCUM_CLEAR_VALUE	Очищающая величина для буфера аккумуляции	аккумулятор	**0**	**glGetFloat()**

Таблица **A-10.** Пиксели

Переменная состояния	**Описание**	**Группа атрибутов**	**Начальное значение**	**Опросная команда**
GL_UNPACK_SWAP_BYTES	Значение для **GL_UNPACK_SWAP_BYTES**	режимы хранения пикселей	**GL_FALSE**	**glGetBooleanv()**
GL_UNPACK_LSB_FIRST	Значение для **GL_UNPACK_LSB_FIRST**	режимы хранения пикселей	**GL_FALSE**	**glGetBooleanv()**
GL_UNPACK_IMAGE_HEIGHT	Значение для **GL_UNPACK_IMAGE_HEIGHT**	режимы хранения пикселей	**0**	**glGetIntegerv()**
GL_UNPACK_SKIP_IMAGES	Значение для **GL_UNPACK_SKIP_IMAGES**	режимы хранения пикселей	**0**	**glGetIntegerv()**
GL_UNPACK_ROW_LENGTH	Значение для **GL_UNPACK_ROW_LENGTH**	режимы хранения пикселей	**0**	**glGetIntegerv()**
GL_UNPACK_SKIP_ROWS	Значение для **GL_UNPACK_SKIP_ROWS**	режимы хранения пикселей	**0**	**glGetIntegerv()**
GL_UNPACK_SKIP_PIXELS	Значение для **GL_UNPACK_SKIP_PIXELS**	режимы хранения пикселей	**0**	**glGetIntegerv()**
GL_UNPACK_ALIGNMENT	Значение для **GL_UNPACK_ALIGNMENT**	режимы хранения пикселей	**4**	**glGetIntegerv()**
GL_PACK_SWAP_BYTES	Значение для **GL_PACK_SWAP_BYTES**	режимы хранения пикселей	**GL_FALSE**	**glGetBooleanv()**
GL_PACK_LSB_FIRST	Значение для **GL_PACK_LSB_FIRST**	режимы хранения пикселей	**GL_FALSE**	**glGetBooleanv()**
GL_PACK_IMAGE_HEIGHT	Значение для **GL_PACK_IMAGE_HEIGHT**	режимы хранения пикселей	**0**	**glGetIntegerv()**
GL_PACK_SKIP_IMAGES	Значение для **GL_PACK_SKIP_IMAGES**	режимы хранения пикселей	**0**	**glGetIntegerv()**
GL_PACK_ROW_LENGTH	Значение для **GL_PACK_ROW_LENGTH**	режимы хранения пикселей	**0**	**glGetIntegerv()**
GL_PACK_SKIP_ROWS	Значение для **GL_PACK_SKIP_ROWS**	режимы хранения пикселей	**0**	**glGetIntegerv()**
GL_PACK_SKIP_PIXELS	Значение для **GL_PACK_SKIP_PIXELS**	режимы хранения пикселей	**0**	**glGetIntegerv()**
GL_PACK_ALIGNMENT	Значение для **GL_PACK_ALIGNMENT**	режимы хранения пикселей	**4**	**glGetIntegerv()**
GL_MAP_COLOR	Активность отображения цвета	пиксели	**GL_FALSE**	**glGetBooleanv()**
GL_MAP_STENCIL	Активность отображения	пиксели	**GL_FALSE**	**glGetBooleanv()**

	индексов трафарета			
GL_INDEX_SHIFT	Значение для **GL_INDEX_SHIFT**	пиксели	**0**	**glGetIntegerv()**
GL_INDEX_OFFSET	Значение для **GL_INDEX_OFFSET**	пиксели	**0**	**glGetIntegerv()**
GL_*x*_SCALE	Значение для **GL_x_SCALE (**где ***x* – RED, GREEN, BLUE** или **ALPHA)**	пиксели	**1**	**glGetFloatv()**
GL_*x*_BIAS	Значение для **GL_*x*_BIAS (**где ***x* – RED, GREEN, BLUE** или **ALPHA)**	пиксели	**0**	**glGetFloatv()**
GL_COLOR_TABLE	Активность цветовой таблицы	пиксели**/** включенные	**GL_FALSE**	**glIsEnabled()**
GL_POST_CONVOLUTION_COLOR_TABLE	Активность пост фильтрационной цветовой таблицы	пиксели**/** включенные	**GL_FALSE**	**glIsEnabled()**
GL_POST_COLOR_MATRIX_COLOR_TABLE	Активность пост матричной цветовой таблицы	пиксели**/** включенные	**GL_FALSE**	**glIsEnabled()**
GL_COLOR_TABLE	Цветовые таблицы	**-**	Пусто	**glGetColorTable()**
GL_COLOR_TABLE_FORMAT	Формат цветовой таблицы	**-**	**GL_RGBA**	**glGetColorTableParameteriv()**
GL_COLOR_TABLE_WIDTH	Ширина цветовой таблицы	**-**	**0**	**glGetColorTableParameteriv()**
GL_COLOR_TABLE_*x*_SIZE	Разрешение цветовых компонент цветовой таблицы **(**где ***x* – RED, GREEN, BLUE, ALPHA, LUMINANCE** или **INTENSITY)**	**-**	**0**	**glGetColorTableParameteriv()**
GL_COLOR_TABLE_SCALE	Фактор масштаба цветовой таблицы	пиксели	**(1,1,1,1)**	**glGetColorTableParameteriv()**
GL_COLOR_TABLE_BIAS	Скос цветовой таблицы	пиксели	**(0,0,0,0)**	**glGetColorTableParameteriv()**
GL_CONVOLUTION_1D	Активность **1D** фильтрации	пиксели**/** включенные	**GL_FALSE**	**glIsEnabled()**
GL_CONVOLUTION_2D	Активность **2D** фильтрации	пиксели**/** включенные	**GL_FALSE**	**glIsEnabled()**
GL_SEPARABLE_2D	Активность разделяемой **2D** фильтрации	пиксели**/** включенные	**GL_FALSE**	**glIsEnabled()**
GL_CONVOLUTION_1D	**1D** фильтр	**-**	Пусто	**glGetConvolutionFilter()**
GL_CONVOLUTION_2D	**2D** фильтр	**-**	Пусто	**glGetConvolutionFilter()**
GL_SEPARABLE_2D	Разделяемый **2D** фильтр	**-**	Пусто	**glGetSeparableFilter()**
GL_CONVOLUTION_BORDER_COLOR	Цвет границы фильтра	пиксели	**(0,0,0,0)**	**glGetConvolutionParameterfv()**
GL_CONVOLUTION_BORDER_MODE	Режим границы фильтра	пиксели	**GL_REDUCE**	**glGetConvolutionParameteriv()**
GL_CONVOLUTION_FILTER_SCALE	Масштаб фильтра	пиксели	**(1,1,1,1)**	**glGetConvolutionParameterfv()**
GL_CONVOLUTION_FILTER_BIAS	Скос фильтра	пиксели	**(0,0,0,0)**	**glGetConvolutionParameterfv()**
GL_CONVOLUTION_FORMAT	Формат фильтра	**-**	**GL_RGBA**	**glGetConvolutionParameteriv()**
GL_CONVOLUTION_WIDTH	Ширина фильтра	**-**	**0**	**glGetConvolutionParameteriv()**
GL_CONVOLUTION_HEIGHT	Высота фильтра	**-**	**0**	**glGetConvolutionParameteriv()**
GL_POST_CONVOLUTION_*x*_SCALE	Пост фильтрационный масштаб **(**где ***x* – RED,**	пиксели	**1**	**glGetFloatv()**

	GREEN, BLUE или **ALPHA)**			
GL_POST_CONVOLUTION_*x*_BIAS	Пост фильтрационный скос **(**где ***x* – RED, GREEN, BLUE** или **ALPHA)**	пиксели	**0**	**glGetFloatv()**
GL_POST_COLOR_MATRIX_*x*_SCALE	Пост матричный масштаб **(**где ***x* – RED, GREEN, BLUE** или **ALPHA)**	пиксели	**1**	**glGetFloatv()**
GL_POST_COLOR_MATRIX_*x*_BIAS	Пост матричный скос **(**где ***x* – RED, GREEN, BLUE** или **ALPHA)**	пиксели	**0**	**glGetFloatv()**
GL_HISTOGRAM	Активность гистограммы	пиксели**/** включенные	**GL_FALSE**	**glIsEnabled()**
GL_HISTOGRAM	Таблица гистограммы	**-**	Пусто	**glGetHistogram()**
GL_HISTOGRAM_WIDTH	Ширина гистограммы	**-**	**0**	**glGetHistogramParameteriv()**
GL_HISTOGRAM_FORMAT	Формат гистограммы	**-**	**GL_RGBA**	**glGetHistogramParameteriv()**
GL_HISTOGRAM_*x*_SIZE	Разрешение цветовых компонент гистограммы **(**где ***x* – RED, GREEN, BLUE, ALPHA** или **LUMINANCE)**	**-**	**0**	**glGetHistogramParameteriv()**
GL_HISTOGRAM_SINK	Поглощение пиксельных групп	**-**	**GL_FALSE**	**glGetHistogramParameteriv()**
GL_MINMAX	Активность расчетов минимального и максимального цветовых значений	пиксели**/** включенные	**GL_FALSE**	**glIsEnabled()**
GL_MINMAX	Таблица минимальных и максимальных цветовых значений	**-**	Таблица минимумов устанавливается в максимально допустимые величины**,** а таблица максимумов **–** в минимально представляемые	**glGetMinmax()**
GL_MINMAX_FORMAT	Формат минимальных и максимальных цветовых значений	**-**	**GL_RGBA**	**glGetMinmaxParameteriv()**
GL_MINMAX_SINK	Поглощение пиксельных групп	**-**	**GL_FALSE**	**glGetMinmaxParameteriv()**
GL_ZOOM_X	Фактор масштаба по **X**	пиксели	**1.0**	**glGetFloatv()**
GL_ZOOM_Y	Фактор масштаба по **Y**	пиксели	**1.0**	**glGetFloatv()**
GL_PIXEL_MAP_*x*	Таблицы отображения пикселей	**-**	Нули	**glGetPixelMap*()**
GL_PIXEL_MAP_*x*_SIZE	Размер таблицы	**-**	**1**	**glGetIntegerv()**
GL_READ_BUFFER	Текущий буфер для чтения	пиксели	**-**	**glGetIntegerv()**

Таблица **A-11.** Вычислители

Переменная состояния	**Описание**	**Группа атрибутов**	**Начальное значение**	**Опросная команда**
GL_ORDER	порядок **1D** карты	**-**	**1**	**glGetMapiv()**
GL_ORDER	порядок **2D**	**-**	**1,1**	**glGetMapiv()**

	карты			
GL_COEFF	контрольные точки **1D** карты	**-**	**-**	**glGetMapfv()**
GL_COEFF	контрольные точки **2D** карты	**-**	**-**	**glGetMapfv()**
GL_DOMAIN	**1D** конечные точки	**-**	**-**	**glGetMapfv()**
GL_DOMAIN	**2D** конечные точки	**-**	**-**	**glGetMapfv()**
GL_MAP1_*x*	Активность **1D** карты **(**где ***x*** **–** тип карты**)**	вычислители**/**включенные	**GL_FALSE**	**glIsEnabled()**
GL_MAP2_*x*	Активность **2D** карты **(**где ***x*** **–** тип карты**)**	вычислители**/** включенные	**GL_FALSE**	**glIsEnabled()**
GL_MAP1_GRID_DOMAIN	Конечные точки **1D** сетки	вычислители	**0,1**	**glGetFloatv()**
GL_MAP2_GRID_DOMAIN	Конечные точки **2D** сетки	вычислители	**0,1;0,1**	**glGetFloatv()**
GL_MAP1_GRID_SEGMENTS	Разделители **1D** сетки	вычислители	**1**	**glGetFloatv()**
GL_MAP2_GRID_ SEGMENTS	Разделители **2D** сетки	вычислители	**1,1**	**glGetFloatv()**
GL_AUTO_NORMAL	Активность автоматической генерации нормалей	вычислители	**GL_FALSE**	**glIsEnabled()**
GL_PERSPECTIVE_CORRECTION_HINT	Комплексная установка перспективной коррекции	комплексные установки	**GL_DONT_CARE**	**glGetIntegerv()**
GL_POINT_SMOOTH_HINT	Комплексная установка сглаживания точек	комплексные установки	**GL_DONT_CARE**	**glGetIntegerv()**
GL_LINE_SMOOTH_HINT	Комплексная установка сглаживания линий	комплексные установки	**GL_DONT_CARE**	**glGetIntegerv()**
GL_POLYGON_SMOOTH_HINT	Комплексная установка сглаживания полигонов	комплексные установки	**GL_DONT_CARE**	**glGetIntegerv()**
GL_FOG_HINT	Комплексная установка качества тумана	комплексные установки	**GL_DONT_CARE**	**glGetIntegerv()**

Таблица **A-12.** Значения**,** зависящие от реализации

Переменная состояния	**Описание**	**Группа атрибутов**	**Минимальное значение**	**Опросная команда**
GL_MAX_LIGHTS	Максимальное количество источников света	**-**	**8**	**glGetIntegerv()**
GL_MAX_CLIP_PLANES	Максимальное количество отсекающих плоскостей	**-**	**6**	**glGetIntegerv()**
GL_MAX_MODELVIEW_STACK_DEPTH	Максимальная глубина стека	**-**	**32**	**glGetIntegerv()**

	видовых матриц			
GL_MAX_PROJECTION_STACK_DEPTH	Максимальная глубина стека проекционных матриц	**-**	**2**	**glGetIntegerv()**
GL_MAX_TEXTURE_STACK_DEPTH	Максимальная глубина стека текстурных матриц	**-**	**2**	**glGetIntegerv()**
GL_SUBPIXEL_BITS	Количество битов в ***x*** и ***y*****,** влияющих на положение соответствующего пикселя на экране	**-**	**4**	**glGetIntegerv()**
GL_MAX_3D_TEXTURE_SIZE	Для текстурного прокси	**-**	**16**	**glGetIntegerv()**
GL_MAX_TEXTURE_SIZE	Для текстурного прокси	**-**	**64**	**glGetIntegerv()**
GL_MAX_PIXEL_MAP_TABLE	Максимальный размер таблицы отображения пикселей **(glPixelMap())**	**-**	**32**	**glGetIntegerv()**
GL_MAX_NAME_STACK_DEPTH	Максимальная глубина стека имен для режима выбора	**-**	**64**	**glGetIntegerv()**
GL_MAX_LIST_NESTING	Максимальная вложенность вызовов списков отображения	**-**	**64**	**glGetIntegerv()**
GL_MAX_EVAL_ORDER	Максимальный порядок полинома вычислителя	**-**	**8**	**glGetIntegerv()**
GL_MAX_VIEWPORT_DIMS	Максимальные размеры порта просмотра	**-**	**-**	**glGetIntegerv()**
GL_MAX_ATTRIB_STACK_DEPTH	Максимальная глубина стека атрибутов	**-**	**16**	**glGetIntegerv()**
GL_MAX_CLIENT_ATTRIB_STACK_DEPTH	Максимальная глубина клиентского стека атрибутов	**-**	**16**	**glGetIntegerv()**
GL_AUX_BUFFERS	Количество дополнительных цветовых буферов	**-**	**0**	**glGetBooleanv()**
GL_RGBA_MODE	Текущим является режим **RGBA**	**-**	**-**	**glGetBooleanv()**
GL_INDEX_MODE	Текущим является индексный режим	**-**	**-**	**glGetBooleanv()**
GL_DOUBLEBUFFER	Существуют передний и задний буферы	**-**	**-**	**glGetBooleanv()**
GL_STEREO	Существуют левый и правый буферы	**-**	**-**	**glGetBooleanv()**
GL_ALIASED_POINT_SIZE_RANGE	Диапазон размеров несглаженных точек	**-**	**1,1**	**glGetFloatv()**
GL_SMOOTH_POINT_SIZE_RANGE	Диапазон размеров сглаженных точек	**-**	**1,1**	**glGetFloatv()**
GL_SMOOTH_POINT_SIZE_GRANULARITY	Гранулярность размеров сглаженных точек	**-**	**-**	**glGetFloatv()**
GL_ALIASED_LINE_WIDTH_RANGE	Диапазон толщины несглаженных линий	**-**	**1,1**	**glGetFloatv()**
GL_SMOOTH_LINE_ WIDTH_RANGE	Диапазон толщины сглаженных линий	**-**	**1,1**	**glGetFloatv()**

GL_SMOOTH_LINE_WIDTH_GRANULARITY	Гранулярность толщины сглаженных линий	-	-	**glGetFloatv()**
GL_MAX_CONVOLUTION_WIDTH	Максимальная ширина фильтра	-	**1**	**glGetConvolutionParameteriv()**
GL_MAX_CONVOLUTION_HEIGHT	Максимальная высота фильтра	-	**1**	**glGetConvolutionParameteriv()**
GL_MAX_ELEMENTS_INDICES	Рекомендуемое максимальное число индексов при использовании **glDrawRangeElements()**	-	-	**glGetIntegerv()**
GL_MAX_ELEMENTS_VERTICES	Рекомендуемое максимальное число вершин при использовании **glDrawRangeElements()**	-	-	**glGetIntegerv()**
GL_MAX_TEXTURE_UNITS_ARB	Максимальное число текстурных блоков	-	**1**	**glGetIntegerv()**

Таблица **A-13.** Глубина пикселей**,** зависящая от реализации

Переменная состояния	**Описание**	**Группа атрибутов**	**Минимальное значение**	**Опросная команда**
GL_RED_BITS	Число бит на красный компонент в цветовом буфере	-	-	**glGetIntegerv()**
GL_GREEN_BITS	Число бит на зеленый компонент в цветовом буфере	-	-	**glGetIntegerv()**
GL_BLUE_BITS	Число бит на синий компонент в цветовом буфере	-	-	**glGetIntegerv()**
GL_ALPHA_BITS	Число бит на альфа компонент в цветовом буфере	-	-	**glGetIntegerv()**
GL_INDEX_BITS	Число бит на индекс в цветовом буфере	-	-	**glGetIntegerv()**
GL_DEPTH_BITS	Число бит на глубину в буфере глубины	-	-	**glGetIntegerv()**
GL_STENCIL_BITS	Число бит на индекс в буфере трафарета	-	-	**glGetIntegerv()**
GL_ACCUM_RED_BITS	Число бит на красный компонент в буфере аккумулятора	-	-	**glGetIntegerv()**
GL_ACCUM_GREEN_BITS	Число бит на зеленый компонент в буфере аккумулятора	-	-	**glGetIntegerv()**
GL_ACCUM_BLUE_BITS	Число бит на синий компонент в буфере аккумулятора	-	-	**glGetIntegerv()**
GL_ACCUM_ALPHA_BITS	Число бит на альфа компонент в буфере аккумулятора	-	-	**glGetIntegerv()**

Таблица **A-14.** Дополнительные

Переменная состояния	**Описание**	**Группа атрибутов**	**Начальное значение**	**Опросная команда**
GL_LIST_BASE	Установка **glListBase()**	список	**0**	**glGetIntegerv()**
GL_LIST_INDEX	Номер конструируемого списка отображения**, 0 –**	-	**0**	**glGetIntegerv()**

	если такого нет			
GL_LIST_MODE	Режим конструируемого списка, **0** – если такого нет	**-**	**0**	**glGetIntegerv()**
GL_ATTRIB_STACK_DEPTH	Указатель на стек атрибутов	**-**	**0**	**glGetIntegerv()**
GL_CLIENT_ATTRIB_STACK_DEPTH	Указатель на клиентский стек атрибутов	**-**	**0**	**glGetIntegerv()**
GL_NAME_STACK_DEPTH	Глубина стека имен	**-**	**0**	**glGetIntegerv()**
GL_RENDER_MODE	Текущий режим визуализации **(glRenderMode())**	**-**	**GL_RENDER**	**glGetIntegerv()**
GL_SELECTION_BUFFER_POINTER	Указатель на буфер выбора	выбор	**0**	**glGetPointerv()**
GL_SELECTION_BUFFER_SIZE	Размер буфера выбора	выбор	**0**	**glGetIntegerv()**
GL_FEEDBACK_BUFFER_POINTER	Указатель на буфер отклика	отклик	**0**	**glGetPointerv()**
GL_FEEDBACK _BUFFER_SIZE	Размер буфера отклика	отклик	**0**	**glGetIntegerv()**
GL_FEEDBACK _BUFFER_TYPE	Тип буфера отклика	отклик	**GL_2D**	**glGetIntegerv()**
-	Код текущей ошибки **(**ошибок**)**	**-**	**0**	**glGetError()**

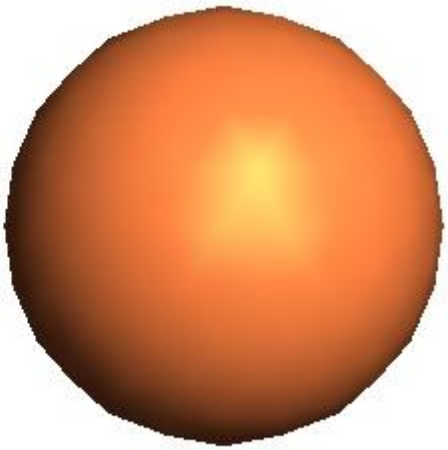
Приложение B. Вычисление векторов нормалей

В этом приложении описываются детали вычислений нормалей к поверхностям, необходимых для использования освещения в **OpenGL.**

Поскольку нормали являются перпендикулярами к поверхности, вы можете начать поиск нормали в конкретной точке с нахождения плоскости, которая касается вашей поверхности только в одной этой точке. Нормаль – это вектор перпендикулярный к этой плоскости. На идеальной сфере, например, нормаль к точке поверхности имеет то же направление, что и вектор из центра сферы в эту точку. Для других типов поверхностей существуют иные лучшие способы нахождения нормалей, зависящие от того, как задается поверхность.

Помните о том, что гладкие поверхности аппроксимируютсяя большим числом небольших плоских полигонов. Если векторы, перпендикулярные к этим полигонам используются в качестве нормалей аппроксимированных поверхностей, то сами поверхности выглядят сегментированными, поскольку пространство векторов не является непрерывным за границами полигонов. Однако во многих случаях для модели существует точное математическое описание, и в каждой точке может быть вычислен вектор истинной нормали. Использование истинных нормалей существенно улучшает результат визуализации, как показано на рисунке **B-1.** Даже если у вас нет математического описания поверхности, вы можете добиться лучшего результата, чем сегментированная поверхность.

Рисунок **B-1.** Истинные нормали **(**справа**)** против полигональных нормалей **(**слева**)**



Нахождение нормалей для аналитических поверхностей

Аналитические поверхности – это плавные поверхности, которые описываются математическим уравнением (или некоторым набором уравнений). Во многих случаях, нормали проще всего находить для аналитических поверхностей, для которых у вас есть исчерпывающее описание в следующей форме:

V(*s*, *t*) = [X(*s*, *t*) Y(*s*, *t*) Z(*s*, *t*)],

где ***s*** и ***t*** определены в некотором пространстве, а **X**, **Y** и **Z** – дифференцируемые функции **2**-ух переменных. Чтобы вычислить нормаль, найдите

$$\frac{\partial V}{\partial s} \text{ и } \frac{\partial V}{\partial t},$$

являющиеся векторами касательными к поверхности в направлениях ***s*** и ***t.*** Их векторное произведение

$$\frac{\partial V}{\partial s} \times \frac{\partial V}{\partial t}$$

перпендикулярно им обоим и, как следствие, перпендикулярно поверхности. Следующая формула отражает процесс вычисления векторного произведения **2**-ух векторов. (Следите за случаями, когда векторное произведение имеет нулевую длину.)

$$[v_x v_y v_z] \times [w_x w_y w_z] = [(v_y w_z - w_y v_z)(w_x v_z - v_x w_z)(v_x w_y - w_x v_y)]$$

Вероятно, вам потребуется нормализовать результирующий вектор. Для нормализации вектора **[*x y z*]**, вычислите его длину

$$\text{Длина} = \sqrt{x^2 + y^2 + z^2}$$

и разделите на нее каждый из компонентов вектора.

В качестве примера для этих расчетов, возьмем следующую аналитическую поверхность

$$V(s,t) = [s^2 \quad t^3 \quad 3 - st]$$

Отсюда имеем

$$\frac{\partial V}{\partial s} = [2s \quad 0 \quad -t], \frac{\partial V}{\partial t} = [0 \quad 3t^2 \quad -s], \frac{\partial V}{\partial s} \times \frac{\partial V}{\partial t} = [-3t^3 \quad 2s^2 \quad 6st^2]$$

Таким образом, если в данном примере ***s*=1** и ***t*=2**, соответствующей точкой поверхности является точка с координатами **(1, 8, 1)**, а вектор **(-24, 2, 24)** является перпендикуляром к поверхности в этой точке. Длина этого вектора равна **34**, следовательно, вектор нормали единичной длины равен

(-24/34, 2/34, 24/34) = (-0.70588, 0.058823, 0.70588).

В случае аналитических поверхностей, заданных в неявной форме

F(*x*, *y*, *z*)=0

найти решение значительно сложнее. В некоторых случаях вы можете разрешить это уравнение относительно одной из переменных, скажем ***z*=G(*x*, *y*)** и записать его в уже рассмотренной форме

V(*s*, *t*) = [*s t* G(*s*, *t*)].

В этом случае можно продолжать работу уже описанным способом.

Если вы не можете привести уравнение поверхности к явной форме, вам, возможно, поможет тот факт, что вектор нормали можно получить как градиент

$$VF = [\frac{\partial F}{\partial x} \quad \frac{\partial F}{\partial y} \quad \frac{\partial F}{\partial z}],$$

вычисленный в конкретной точке **(*x*, *y*, *z*)**. Вычислить градиента довольно просто, однако значительно сложнее найти точку, которая лежит на поверхности. В качестве примера неявно определенной аналитической функции рассмотрим уравнение сферы радиусом в **1** с центром в начале координат:

$$x^2 + y^2 + z^2 - 1 = 0$$

Это означает, что

$$F(x, y, z) = x^2 + y^2 + z^2 - 1.$$

Это уравнение может быть разрешено относительно ***z*** следующим образом:

$$z = \pm\sqrt{1 - x^2 - y^2}.$$

Таким образом, нормали можно рассчитать с помощью явного вида

$$V(s,t) = [s \quad t \quad \sqrt{1-x^2-y^2}]$$

как было описано ранее.

Если бы не могли разрешить уравнение относительно ***z***, вы могли бы воспользоваться градиентом

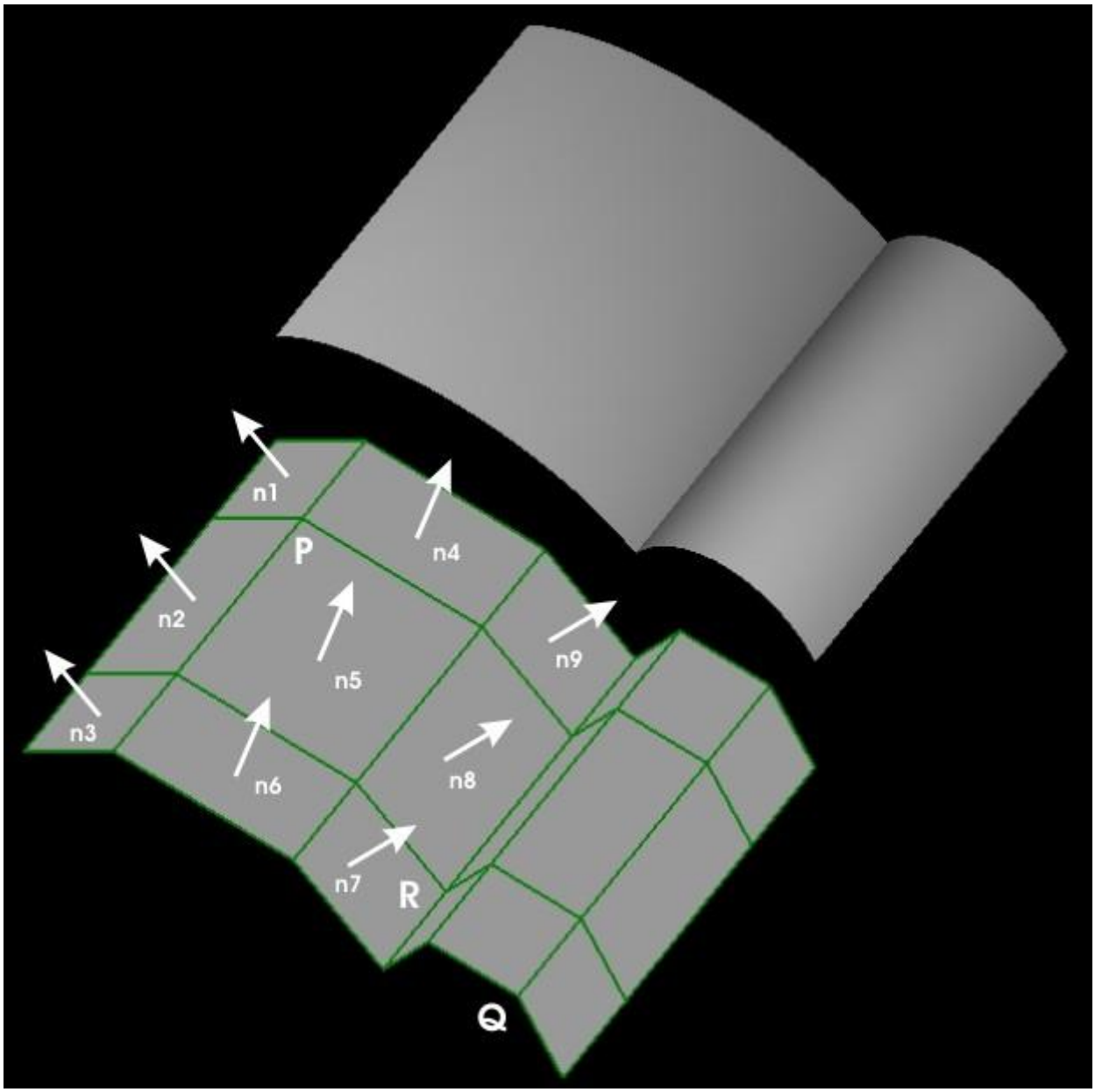
$$VF = [2x \quad 2y \quad 2z],$$

конечно, если бы смогли найти точку на поверхности. В данном случае это совсем не сложно – например, **(2/3, 1/3, 2/3)** лежит на поверхности. Вычисленная с помощью градиента нормаль в данной точке будет равна **(4/3, 2/3, 4/3).** Тогда нормаль единичной длины **– (2/3, 1/3, 2/3),** что, как и ожидалось, совпадает с точкой на поверхности.

Нахождение нормалей по полигональным данным

Как указывалось ранее, вам часто требуется находить нормали к поверхностям, заданным в виде полигональных данных, для того, чтобы они выглядели гладкими, а не сегментированными. В большинстве случаев простейший способ сделать это **(**однако вполне возможно не самый эффективный**) –** вычислить вектор нормали каждого полигонального фрагмента поверхности, а затем усреднить нормали соседних фрагментов. на рисунке **B-2** показана поверхность и ее полигональная аппроксимация. **(**Конечно, если полигоны представляют собой точную поверхность, а не являются просто аппроксимацией – не делайте усреднение.**)** Вычислите нормаль для каждого фрагмента, как описано в следующих параграфах, и используйте эту нормаль для всех вершин фрагмента.

Рисунок **B-2.** Усреднение векторов нормали



Чтобы найти вектор нормали для плоского полигона, возьмите три любые вершины полигона ***v1***, ***v2*** и ***v3***, не лежащие на одной прямой. Векторное произведение

$$[v_1 - v_2] \times [v_2 - v_3]$$

и будет перпендикулярно полигону. (Обычно результирующий вектор нужно нормализовать.) Затем вам нужно усреднить нормали соседствующих полигонов, чтобы не давать больше веса одному из них. Например, если в случае, показанном на рисунке **B-2**, ***n1***, ***n2***, ***n4*** и ***n5*** – нормали полигонов, соединяющихся в точке **P**, вычислите ***n1+n2+n4+n5***, а затем нормализуйте получившийся вектор. (Вы можете получить лучшее усреднение, если взвесите нормали величинами углов в общем пересечении.) Результирующий вектор может использоваться в качестве нормали в точке **P**.

Иногда вы должны модифицировать этот метод под конкретную ситуацию. Например, на границе поверхности (такой как точка **Q** на рисунке **B-2**) вы можете выбрать лучшую нормаль, основываясь на вашем знании о том, как поверхность должна выглядеть. Иногда лучшее, что вы можете сделать, это усреднить и нормали полигонов на границе. Кроме того, у некоторых моделей есть плавные части и острые углы (точка **R** на рисунке **B-2** находится на таком ребре). В этом случае не нужно усреднять нормали

соседствующих полигонов, даже наоборот – полигоны с одной стороны ребра должны быть нарисованы с использованием одной нормали, а с другой – с использованием другой нормали.

Приложение C. Основы GLUT

При использовании **GLUT** ваше приложение структурируется благодаря тому, что для обработки событий используются функции обратного вызова. (Этот метод похож на использование **Xt Toolkit**, также известного как **X Intristics.**) Например, сначала вы открываете окно и регистрируете функции обратного вызова для нужных событий. Затем вы создаете главный цикл, из которого нет выхода. Если в этом цикле происходит событие, вызывается ассоциированная с ним функция обратного вызова. По завершении этой функции поток управления возвращается в главный цикл.

Управление окном

Для инициализации окна существует минимальный набор из пяти функций.

```
void glutInit (int argc, char **argv);
```

glutInit() должна быть вызвана до любых других **GLUT –** функций, так как она инициализирует саму библиотеку **GLUT. glutInit()** также обрабатывает параметры командной строки, но сами параметры зависят от конкретной оконной системы. Для системы **X Window**, примерами могут быть **–iconic**, **-geometry** и **–display**. (Параметры, передаваемые **glutInit()**, должны быть теми же самыми, что и параметры, передаваемые в функцию **main()**).

```
void glutInitDisplayMode (unsigned int mode);
```

Указывает режим отображения (например, **RGBA** или индексный, одинарной или двойной буферизации) для окон, создаваемых вызовами **glutCreateWindow()**. Вы также можете указывать, имеет ли окно ассоциированные с ним буфер или буферы глубины, трафарета и аккумуляции. Аргумент ***mask*** **–** это битовая комбинация, полученная при помощи операции **OR** и следующих констант: **GLUT_RGBA** или **GLUT_INDEX** (для указания цветового режима), **GLUT_SINGLE** или **GLUT_DOUBLE** (для указания режима буферизации), а также константы для включения различных буферов **GLUT_DEPTH**, **GLUT_STENCIL**, **GLUT_ACCUN**. Например, для окна с двойной буферизаций, **RGBA –** цветовым режимом и ассоциированными буферами глубины и трафарета, используйте **GLUT_DOUBLE | GLUT_RGBA | GLUT_DEPTH | GLUT_STENCIL**. Значение по умолчанию **– GLUT_RGBA | GLUT_ SINGLE** (окно с однократной буферизацией в **RGBA -** режиме).

```
void glutInitWindowSize (int width, int height);
```

```
void glutInitWindowPosition (int x, int y);
```

Запрашивают окно определенного размера и в определенном месте экрана соответственно. Аргументы (***x***, ***y***) определяют, где будет находиться угол окна относительно всего экрана. ***width*** и ***height*** определяют размер окна (в пикселях). Начальные размеры и место размещения окна могут быть перекрыты последующими вызовами.

```
int glutCreateWindow (char *name);
```

Открывает окно с предварительно установленными характеристиками **(**режимом отображения**,** размером и так далее**).** Строка ***name*** может быть отображена в заголовке окна**,** но это зависит от конкретной оконной системы**.** Окно не отображается до того**,** как произведен вход в **glutMainLoop()**, поэтому до вызова этой функции нельзя рисовать что**-**либо в окно**.**

Возвращаемая целая величина представляет собой уникальный идентификатор окна**.** Этот идентификатор может быть использован для управления несколькими окнами в одном приложении **(**каждое со своим контекстом **OpenGL)** и рисования в них**.**

Функции управления событиями

После того**,** как окно создано**,** но до входа в главный цикл программы**,** вы должны зарегистрировать функции обратного вызова**,** используя следующие функции **GLUT.**

```
void glutDisplayFunc (void (*func)(void));
```

Позволяет указать функцию **(**аргументом ***func*****)**, которая будет вызываться каждый раз**,** когда содержимое окна требует перерисовки**.** Это может случиться**,** когда окно открывается**,** разворачивается**,** освобождается его часть**,** ранее перекрытая другим окном**,** или вызывается функция **glutPostRedisplay().**

```
void glutReshapeFunc (void (*func)(int width, int height));
```

Позволяет указать функцию**,** которая вызывается каждый раз при изменении размера окна или его позиции на экране**.** Аргумент ***func* –** это указатель на функцию**,** которая принимает два параметра**:** ***width* –** новая ширина окна и ***height* –** новая высота окна**.** Обычно ***func*** вызывает **glViewport()** для отсечения графического вывода по новым размерам окна**,** а также настраивает проекционную матрицу для сохранения пропорций спроецированного изображения в соответствии с новыми размерами порта просмотра**.** Если **glutReshapeFunc()** не вызывается или ей передается **NULL (**для отмены регистрации функции обратного вызова**),** вызывается функция изменения метрик по умолчанию**,** которая вызывает **glViewport (*0*,*0*,*width*,*height*).**

```
void glutKeyboardFunc (void (*func)(unsigned int key, int x, int y));
```

Задает функцию ***func*****,** которая вызывается**,** когда нажимается клавиша**,** имеющая **ASCII-**код**.** Этот код передается функции обратного вызова в параметре ***key*****.** В параметрах ***x*** и ***y*** передается позиция курсора мыши **(**относительно окна**)** в момент нажатия клавиши**.**

```
void glutMouseFunc (void (*func)(int button, int state, int width, int height));
```

Указывает функцию**,** которая вызывается при нажатии или отпускании кнопки мыши**.** Параметр ***button*** может иметь значения **GLUT_LEFT_BUTTON**, **GLUT_MIDDLE_BUTTON** или **GLUT_RIGHT_BUTTON.** Параметр ***state*** может иметь значения **GLUT_UP** или **GLUT_DOWN** в зависимости от того отпущена или нажата кнопка мыши**.** В параметрах ***x*** и ***y*** передаются координаты курсора мыши **(**относительно окна**)** в момент наступления события**.**

```
void glutMotionFunc (void (*func)(int x, int y));
```

Указывает функцию**,** которая будет вызываться при движении мыши внутри окна в то время**,** как на ней нажата одна или несколько клавиш**.** В параметрах ***x*** и ***y*** передаются координаты курсора мыши **(**относительно окна**)** в текущий момент**.**

```
void glutPostRedisplay (void);
```

Помечает, что текущее окно требует перерисовки. После этого при любой возможности будет вызвана функция перерисовки окна, зарегистрированная вызовом **glutDisplayFunc()**.

Загрузка палитры

Если вы работаете в индексном режиме, то можете к своему удивлению обнаружить, что в **OpenGL** нет команд для загрузки цвета в цветовую таблицу. Дело в том, что процесс загрузки палитры целиком зависит от оконной системы. В **GLUT** существует обобщенная функция для загрузки одного цветового индекса с соответствующим **RGB** значением.

```
void glutSetColor (Glint index, GLfloat red, GLfloat green, GLfloat blue);
```

Загружает в палитру по индексу ***index***, **RGB-**значение, определенное параметрами ***red***, ***green*** и ***blue***. Последние три параметра нормализуются до диапазона **[0.0, 1.0]**.

Рисование трехмерных объектов

Многие программы примеры используют простые трехмерные объекты для иллюстрации различных методов и техник визуализации изображения. **GLUT** содержит несколько функций для рисования таких объектов. Все эти функции работают в непосредственном режиме. Каждая из них имеет два варианта: первый рисует объект в виде проволочного каркаса и не генерирует нормалей, второй рисует объект сплошным и генерирует нормали поверхности **(**для чайника помимо этого генерируются координаты текстуры**)**. Если используется освещение, следует выбирать сплошную версию объекта. Все объекты рисуются с учетом текущих параметров, например, цвета и характеристик материала. Кроме того, все объекты рисуются центрированными относительно текущих модельных координат.

```
void glutWireSphere (GLdouble radius, GLint slices, GLint stacks);
```

```
void glutSolidSphere (GLdouble radius, GLint slices, GLint stacks);
```

Рисуют проволочную или сплошную сферу с радиусом **radius**, количеством частей **(**полигонов из которых состоит сфера**)** ***slices*** **–** вокруг оси **z** и ***stacks*** **–** вдоль оси **z**. Для того, чтобы понять, что означает вокруг оси **z** и вдоль нее, представьте себе, что вы смотрите в длинную трубу. В данном случае направление вашего обзора совпадает с осью **z** трубы. Она может быть мысленно разделена как вдоль **(**на длинные фрагменты**)**, так и поперек **(**на кольца**)**. После таких разбиений труба фактически состоит из множества мелких кусочков. В случае сферы количество разбиений поперек задается параметром ***stacks***, а количество разбиений вдоль **–** параметром ***slices***. Из этого следует, что чем больше разбиений, тем более гладкой выглядит сфера на экране, но тем больше вычислений требуется для ее рисования.

```
void glutWireCube (GLdouble size);
```

```
void glutSolidCube (GLdouble size);
```

Рисуют проволочный или сплошной куб с длиной ребра ***size***.

```
void glutWireTorus (GLdouble innerRadius, GLdouble outerRadius, GLint nsides,
GLint rings);
```

```
void glutSolidTorus (GLdouble innerRadius, GLdouble outerRadius, GLint nsides,
GLint rings);
```

Рисуют проволочный или сплошной торус **(**бублик**)** с внешним радиусом ***outerRadius*** и внутренним радиусом ***innerRadius*****.** Параметр ***rings*** задает желаемое число колец из которых будет состоять торус**,** параметр ***nsides*** **–** из скольких частей будет состоять каждое кольцо**.**

```
void glutWireCone (GLdouble radius, GLdouble height, GLint slices, GLint stacks);
```

```
void glutSolidCone (GLdouble radius, GLdouble height, GLint slices, GLint
stacks);
```

Рисуют проволочный или сплошной конус радиусом ***radius*****,** высотой ***height*****.** Значение параметров ***slices*** и ***stacks*** аналогично таким же параметрам для сферы**.**

```
void glutWireIcosahedron (void);
```

```
void glutSolidIcosahedron (void);
```

```
void glutWireOctahedron (void);
```

```
void glutSolidOctahedron (void);
```

```
void glutWireTetrahedron (void);
```

```
void glutSolidTetrahedron (void);
```

```
void glutWireDodecahedron (GLdouble radius);
```

```
void glutSolidDodecahedron (GLdouble radius);
```

Рисуют проволочные или сплошные икосаэдр**,** октаэдр**,** тетраэдр и додекаэдр соответственно **(**единственный параметр последний пары функций задает радиус додекаэдра**).**

```
void glutWireTeapot (GLdouble size);
```

```
void glutSolidTeapot (GLdouble size);
```

Рисуют проволочный или сплошной чайник размера **size.**

Управление фоновым процессом

Вы можете указать функцию**,** которая будет вызываться в том случае**,** если нет других сообщений**,** то есть во время простоя приложения**.** Это может быть полезно для выполнения анимации или другой фоновой обработки**.**

```
void glutIdleFunc (void (*func)(void));
```

Задает функцию, выполняемую в случае, если больше приложению делать нечего (отсутствуют сообщения). Выполнение этой функции обратного вызова можно отменить передачей **glutIdleFunc()** аргумента **NULL**.

Запуск программы

После того, как все настройки выполнены, программы **GLUT** входят в цикл обработки сообщений функцией **glutMainLoop()**.

```
void glutMainLoop (void);
```

Вводит программу в цикл обработки сообщений. Функции обратного вызова будут выполняться в случае наступления соответствующих событий.

Пример **1-2** показывает, как с помощью **GLUT** можно заставить работать программу, показанную в примере **1-1**. Обратите внимание на реструктуризацию кода. Для увеличения эффективности операции, которые нужно выполнить только один раз (установка цвета фона и координатной системы), теперь помещены в функцию **init()**. Операции по визуализации (и пересчету) сцены находятся в функции **display()**, которая зарегистрирована в качестве дисплейной функции обратного вызова.

Приложение D. Порядок операций

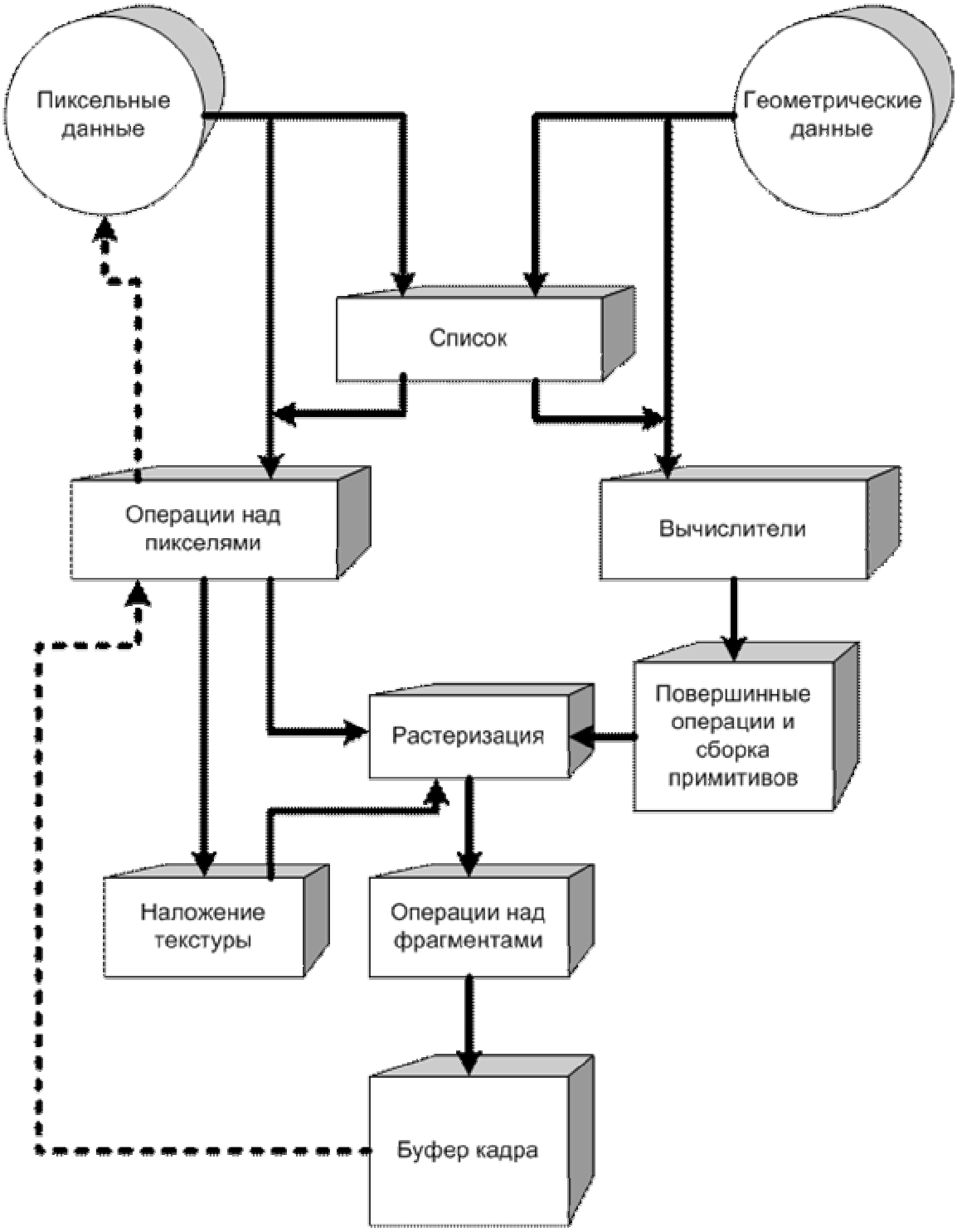
В этой книге описаны все операции, производимые между начальным указанием вершин и финальной записью фрагментов в буфер кадра. Главы книги расположены так, чтобы упростить изучение, их порядок не совпадает с порядком, в котором выполняются операции. Иногда точный порядок операций не имеет значения – поверхности могут быть преобразованы в полигоны и трансформированы или могут быть сначала трансформированы, а затем преобразованы в полигоны с идентичным результатом. Разные реализации **OpenGL** иногда имеют разный порядок операций.

В этом разделе описан возможный порядок. Любая реализация **OpenGL** должна выдавать эквивалентный результат.

Введение

В этом разделе дается введение в порядок операций, показанный на рисунке **D-1**. Геометрические данные (вершины, линии и полигоны) следуют по пути через ряд блоков, включающих вычислители и повершинные операции, в то время как пиксельные данные (пиксели, изображения и битовые карты) в определенной части процесса обрабатываются иначе. Оба типа данных подвергаются растеризации и пофрагментным операциям до того, как будут записаны в буфер кадра.

Рисунок **D-1**. Порядок операций



Все данные, описаны ли они геометрически или в виде пикселей, могут быть сохранены в списке отображения или обработаны немедленно. При исполнении списка отображения, данные из него пересылаются точно так же, как они пересылались бы из приложения непосредственно.

Все геометрические примитивы, в конце концов, описываются своими вершинами. Если используются вычислители, данные преобразуются к вершинам и, с момента преобразования, обрабатываются как вершины. Кроме того, вершинные данные могут быть сохранены и использованы из специализированных вершинных массивов. Для каждой вершины производятся повершинные вычисления, за которыми следует растеризация в фрагменты. С пиксельными данными производятся пиксельные

операции, после чего данные сохраняются в текстурной памяти, используются для полигонального шаблонирования или растеризуются в фрагменты.

Наконец, фрагменты подвергаются серии пофрагментных операций, после чего результирующие пиксельные значения записываются в буфер кадра.

Геометрические операции

Геометрические данные, появляются ли они из списка отображения, из вычислителя, из вершинного массива или в качестве вершин прямоугольника, состоят из вершин и типа примитива, который они описывают (точка, линия, полигон). Вершинные данные включают не только координаты (***x***, ***y***, ***z***, ***w***), но и вектор нормали, координаты текстуры, цвет **RGBA**, цветовой индекс, свойства материала и флаг ребра. За исключением координат вершины все эти элементы могут быть заданы в любом порядке, кроме того, существуют значения по умолчанию. Как только вызывается команда **glVertex*()**, компоненты расширяются до **4** измерений, если в том есть необходимость (с использованием ***z=0*** и ***w=1***), и текущие значения элементов ассоциируются с вершиной. Полный комплект вершинных данных уходит на обработку. (Если используются вершинные массивы, вершины могут обрабатываться группами, а обработанные группы могут использоваться многократно.)

Повершинные операции

На этапе повершинных операций геометрические координаты вершины преобразуются с помощью видовой матрицы, в то время как вектор нормали вершины преобразуется с использованием инвертированной и транспонированной видовой матрицы и заново нормализуется, если это требуется. Если активизирована автоматическая генерация координат текстуры, новые координаты текстуры генерируются с помощью преобразованных координат вершины и заменяют старые координаты текстуры для вершины. Далее координаты текстуры преобразуются с помощью текущей текстурной матрицы и передаются на этап сборки примитивов.

В то же время производятся расчеты, связанные с освещением (если освещение включено). В расчетах используются координаты вершины, координаты вектора нормали, параметры материала, текущие источники света, их параметры, а также текущие параметры модели освещения. С помощью этих расчетов генерируются новые цвета или индексы, которые усекаются или маскируются до необходимого диапазона и передаются на этап сборки примитивов.

Сборка примитивов

Сборка примитивов проходим по-разному в зависимости от того, является ли примитив точкой, линией или полигоном. Если активизирована плоская заливка, цвета или индексы всех вершин линии или полигона устанавливается в одно и то же значение. Если созданы и активизированы дополнительные плоскости отсечения, они используются для отсечения примитивов всех трех типов. (Уравнения отсекающих плоскостей сразу после определения преобразуются с помощью инвертированной транспозиции видовой матрицы.) Отсечение точек просто пропускает или отвергает вершины; отсечение линий или полигонов может создать дополнительные вершины в зависимости от того, как отсекаются эти примитивы. После этого пространственные координаты каждой вершины преобразуются с помощью проекционной матрицы, и результаты отсекаются по стандартными видовыми плоскостями $x = \pm w, y = \pm w, z = \pm w$.

Если активизирован режим выбора, любые примитивы, не устраненные отсечением, генерируют записи о попаданиях, а дальнейшая обработка не производится. В отсутствие режима выбора производится перспективное деление на **w**, производятся

операции порта просмотра и диапазона глубин. Кроме того, если примитив является полигоном, он подвергается тесту отсечения нелицевых граней (если таковой активизирован). Полигон может быть конвертирован в вершины или линии в зависимости от режима его отображения.

Операции над пикселями

Сначала пиксели из памяти хоста распаковываются в нужное количество компонент. Часть **OpenGL**, ответственная за распаковку, обрабатывает большое количество разных форматов. Далее данные масштабируются, скашиваются и обрабатываются с использованием пиксельной карты **(pixel map)**. Результаты усекаются до нужного диапазона в зависимости от типа данных и затем либо записываются в текстурную память для использования при текстурировании, либо растеризуются в фрагменты.

Если пиксельные данные считываются из буфера кадра, производятся операции пиксельного переноса (масштабирование и скос, отображение и усечение). Результаты упаковываются в нужный формат и возвращаются в процессорную память.

Операция копирования пикселей похожа на комбинацию распаковки и операций переноса, но упаковка и распаковка не нужны, и через операции пиксельного переноса делается только один проход перед тем, как данные записываются обратно в буфер кадра.

Текстурная память

OpenGL версии **1.1** предоставляет дополнительный контроль над текстурной памятью. Изображения текстуры могут извлекаться из текстурной памяти, так же как из процессорной. Все изображение текстуры или его часть могут быть заменены. Данные текстуры можно загружать в текстурные объекты, которые могут быть загружены в текстурную память. Если объектов так много, что все они не помещаются в текстурной памяти одновременно, в ней остаются текстуры, имеющие наивысшие приоритеты.

Операции над фрагментами

Если активизировано текстурирование, для каждого фрагмента генерируется тексель, который накладывается на фрагмент. Далее следует вычисление тумана, за которым следует вычисление величины покрытия фрагментами пикселей (антиалиасинг), если активизирован антиалиасинг.

Далее производится тест отреза, потом альфа тест (только в **RGBA** режиме), тест трафарета и тест глубины. Если работа ведется в режиме **RGBA**, производится цветовое наложение. За цветовым наложением производятся цветовое микширование и логические операции. Все описанные операции могут быть деактивированы.

Далее фрагменты маскируются с помощью цветовой маски или индексной маски, в зависимости от режима, и записываются в нужный буфер. Если фрагменты записываются в буфер трафарета или глубины, маскирование происходит после тестов трафарета или глубины, и результаты записываются в буфер кадра без проведения цветового наложения, микширования и логических операций.

Матричные операции

Матричные операции работают на текущем матричном стеке, коим может быть видовой, проекционный или текстурный. Команды **glMultMatrix*()**, **glLoadMatrix*()** и **glLoadIdentity()** применяются к верхней матрице стека, а **glTranslate*()**, **glRotate*()**, **glScale*()**, **glOrtho()** и **glFrustum()** используются для создания матрицы, на которую затем умножается верхняя матрица стека. Когда изменяется

видовая матрица, также генерируется ее инвертированная транспозиция для преобразований векторов нормали.

Команды, устанавливающие текущую растровую позицию, обрабатываются точно так же, как вершины до точки, в которой должна производиться растеризация. В этой точке величина сохраняется и используется при растеризации пиксельных данных.

Различные команды **glClear()** проходят мимо всех операций за исключением теста отреза, микширования и маскирования.

Приложение E. Однородные координаты и матрицы преобразований

В этом приложении представлено краткое обсуждение однородных координат. Здесь также приведены общие виды матриц, используемых для поворота, переноса, масштабирования, перспективной и ортогональной проекций. В следующем обсуждении термин однородные координаты всегда применяется к трехмерным координатам, хотя проективная геометрия существует для всех измерений.

Однородные координаты

Команды **OpenGL** обычно работают с двумерными и трехмерными вершинами, но на самом деле все вершины интерпретируются как трехмерные, состоящие из четырех координат. Любой вектор – столбец $(x, y, z, w)^T$ представляет собой однородную вершину, если хотя бы один из его элементов не равен ***0***. Если вещественное число **a** не равно ***0***, то $(x, y, z, w)^T$ и $(ax, ay, az, aw)^T$ представляют одну и ту же однородную вершину. Трехмерная точка Евклидового пространства $(x, y, z)^T$ переходит в однородную вершину с координатами $(x, y, z, 1.0)^T$, а двумерная точка Евклидового пространства $(x, y)^T$ -- в однородную вершину $(x, y, 0.0, 1.0)^T$.

До тех пор, пока **w** не равно **0**, однородная вершины $(x, y, z, w)^T$ соответствует трехмерной точке $(x/w, y/w, z/w)^T$. Если ***w=0.0***, вершина не соответствует ни одной точке в Евклидовом пространстве, а представляет собой некоторую идеализированную «точку в бесконечности». Чтобы понять, что такое «точка в бесконечности», рассмотрим вершину с координатами ***(1, 2, 0, 0)*** и заметим что последовательность точек ***(1, 2, 0, 1)***, ***(1, 2, 0, 0.01)*** и ***(1, 2, 0, 0.0001)***, соответствует Евклидовым точкам ***(1, 2)***, ***(100, 200)*** и ***(10,000, 20,000)***. Эта последовательность представляет точки, быстро смещающиеся в бесконечность вдоль прямой ***2x=y***. Таким образом, вы можете думать о точке ***(1, 2, 0, 0)*** как о точке в бесконечности, лежащей на этой прямой.

Замечание: OpenGL может неверно обрабатывать однородные усеченные координаты с *w<0*. Чтобы быть уверенными, что ваш код является переносимым на все реализации OpenGL, используйте только неотрицательные значения.

Преобразование вершин

Преобразования вершин (такие как повороты, переносы, масштабирования и отражения) и проекционные преобразования (такие как перспективное и ортографическое) могут быть представлены в виде применения соответствующей матрицы **4x4** к координатам, представляющим вершину. Если **v** представляет

однородную вершину, а **M –** трансформационная матрица **4x4**, то **Mv** является отображением **v** под воздействием **M. (**В приложениях компьютерной графики применяемые преобразования, как правило, не вырожденные **–** иными словами, матрица является обратимой**.** Это не является обязательным условием, но позволяет избежать ряда проблем с вырожденными преобразованиями**.)**

После преобразования все трансформированные вершины усекаются, чтобы ***x***, ***y*** и ***z*** лежали в диапазоне ***[-w, w]*** **(**в предположении, что ***w>0*****).** Заметьте, что этот диапазон соответствует Евклидовому ***[-1.0, 1.0]*****.**

Преобразование нормалей

Вектора нормалей преобразуются не так, как вершины или позиции вершин**.** Математически лучше думать о векторах нормалей не как о векторах, а как о плоскостях, перпендикулярным к этим векторам**.** Тогда можно описывать правила преобразования нормалей правилами преобразования перпендикулярных плоскостей**.**

Однородная плоскость определяется вектор **–** строкой ***((a, b, c, d)***, где как минимум один из компонентов не равен ***0*****.** Если ***q –*** вещественное число не равное ***0***, то ***(a, b, c, d)*** и ***(qa, qb, qc, qd)*** представляют одну и ту же плоскость**.** Точка $(x, y, z, w)^T$ лежит на поверхности ***((a, b, c, d)***, если ***ax+by+cz+dw=0*****. (**Если ***w=1***, то это стандартное описание Евклидовой плоскости**.)** Чтобы ***(a, b, c, d)*** представляло Евклидову плоскость, как минимум один из компонентов ***a***, ***b*** или ***c*** не должен быть равен ***0*****.** Если все они равны ***0***, то ***(0, 0, 0, d)*** представляет **«**плоскость в бесконечности**»**, содержащую все **«**точки в бесконечности**».**

Если **pp –** однородная плоскость, а **v –** однородная вершина, то утверждение **«v** лежит на **p»** математически записывается как **pv=0**, где **pv –** обычное произведение матриц**.** Если **M –** невырожденное вершинное преобразование **(**то есть матрица **4x4** имеющая обратную матрицу M^{-1}**)**, то **pv=0** эквивалентно $pM^{-1}Mv = 0$, то есть **Mv** лежит в плоскости pM^{-1}**.** Следовательно, pM^{-1} является отображением плоскости под воздействием вершинного преобразования **M.**

Если вам хочется думать о векторах нормалей именно как о векторах, а не как о плоскостях, перпендикулярным к ним, пусть **v** и **n –** такие векторы, что **v** перпендикулярно **n.** Тогда $n^T v = 0$**.** Следовательно, для произвольного невырожденного преобразования **M**, $n^T M^{-1}Mv = 0$, что означает, что $n^T M^{-1}$ является транспозицией преобразованного вектора нормали**.** Следовательно, преобразованный вектор нормали **--** $(M^{-1})^T n$**.** Иными словами вектор нормали преобразуется инвертированной транспозицией матрицы преобразования, преобразующей точки**.**

Матрицы преобразования

Несмотря на то, что любая невырожденная матрица **M** представляет допустимое проективное преобразование, несколько специальных случаев матриц особенно полезны**.** Эти матрицы перечислены в следующих разделах**.**

Перенос

Вызов команды **glTranslate*(*** ***x*****,** ***y*****,** ***z*****)** генерирует **T,** где

$$T=\begin{bmatrix}1&0&0&x\\0&1&0&y\\0&0&1&z\\0&0&0&1\end{bmatrix}$$ и $$T^{-1}=\begin{bmatrix}1&0&0&-x\\0&1&0&-y\\0&0&1&-z\\0&0&0&1\end{bmatrix}.$$

Масштабирование

Вызов команды **glScale*(*x*, *y*, *z*)** генерирует **S**, где

$$S=\begin{bmatrix}x&0&0&0\\0&y&0&0\\0&0&z&0\\0&0&0&1\end{bmatrix}$$ и $$S^{-1}=\begin{bmatrix}\frac{1}{x}&0&0&0\\0&\frac{1}{y}&0&0\\0&0&\frac{1}{z}&0\\0&0&0&1\end{bmatrix}.$$

Заметьте, что S^{-1} определена только если ***x*, *y*** и ***z*** одновременно не равны **0.**

Поворот

Вызов команды **glRotate*(*a*, *x*, *y*, *z*)** генерирует **R** следующим образом:

Пусть $v=(x,y,z)^T$, и $u=v/\|v\|=(x`,y`,z`)^T$.

Также пусть

$$S=\begin{bmatrix}0&-z`&y`\\z`&0&-x`\\-y`&x`&0\end{bmatrix}$$ и $$M=uu^T+(\cos a)(I-uu^T)+(\sin a)S.$$

Затем

$$R=\begin{bmatrix}m&m&m&0\\m&m&m&0\\m&m&m&0\\0&0&0&1\end{bmatrix},$$

где ***m*** представляет элементы матрицы из **M,** которая является матрицей **3x3,** определенной ранее**.** Матрица **R** определена всегда**.** Если ***x=y=z=0*, R** представляет собой единичную матрицу**.** Вы можете получить инверсию **R,** R^{-1}**,** заменив ***a*** на **–*a*,** или транспонировав матрицу**.**

Команда **glRotate*()** генерирует матрицу для поворота вокруг произвольной оси. Часто вы выполняете поворот вокруг одной из координатных осей, этим поворотам соответствуют следующие матрицы:

$$glRotate*(a,1,0,0) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos a & -\sin a & 0 \\ 0 & \sin a & \cos a & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$glRotate*(a,0,1,0) = \begin{bmatrix} \cos a & 0 & \sin a & 0 \\ 0 & 1 & 0 & 0 \\ -\sin a & 0 & \cos a & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$glRotate*(a,0,0,1) = \begin{bmatrix} \cos a & -\sin a & 0 & 0 \\ \sin a & \cos a & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

Как и раньше обратные матрицы получаются с помощью транспозиции.

Перспективная проекция

Обращение к **glFrustum(*l*,*r*,*b*,*t*,*n*,*f*)** генерирует **R**, где

$$R = \begin{bmatrix} \frac{2n}{r-l} & 0 & \frac{r+l}{r-l} & 0 \\ 0 & \frac{2n}{t-b} & \frac{t+b}{t-b} & 0 \\ 0 & 0 & \frac{-(f+n)}{f-n} & \frac{-2fn}{f-n} \\ 0 & 0 & -1 & 0 \end{bmatrix} \quad R^{-1} = \begin{bmatrix} \frac{r-l}{2n} & 0 & 0 & \frac{r+l}{2n} \\ 0 & \frac{t-b}{2n} & 0 & \frac{t+b}{2n} \\ 0 & 0 & 0 & -1 \\ 0 & 0 & \frac{-(f-n)}{2fn} & \frac{f+n}{2fn} \end{bmatrix}.$$

и

R определена до тех пор, пока $l \neq r, t \neq b$ и $n \neq f$.

Ортографическая проекция

Обращение к **glOrtho(*l*,*r*,*b*,*t*,*n*,*f*)** генерирует **R**, где

$$R=\begin{bmatrix} \frac{2}{r-l} & 0 & 0 & \frac{r+l}{r-l} \\ 0 & \frac{2}{t-b} & 0 & \frac{t+b}{t-b} \\ 0 & 0 & \frac{-2}{f-n} & \frac{f+n}{f-n} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$ и $$R^{-1}=\begin{bmatrix} \frac{r-l}{2} & 0 & 0 & \frac{r+l}{2} \\ 0 & \frac{t-b}{2} & 0 & \frac{t+b}{2} \\ 0 & 0 & \frac{f-n}{-2} & \frac{f+n}{2} \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

R определена до тех пор, пока $l \neq r, t \neq b$ и $n \neq f$.

Приложение F. Советы

В этом приложении представлены некоторые советы и наставления, которые могут быть для вас полезными. Имейте в виду, что эти советы базируются на рекомендациях разработчиков **OpenGL**, а не опыте работы с конкретными приложениями или реализациями.

Во избежание сбоев

- Постоянно производите контроль. Вызывайте **glGetError()** как минимум единожды каждый раз при отрисовке сцены, чтобы быть уверенным в том, что ошибка будет обнаружена.
- Не полагайтесь на поведение реализации **OpenGL** в отношении ошибок – оно может измениться в следующих версиях. Например, **OpenGL** версии **1.1** игнорирует матричные операции, производящиеся между вызовами команд **glBegin()/glEnd()**, но в будущих версиях это может быть и не так.
- Если вам нужно схлопнуть всю геометрию к одной плоскости, используйте проекционную матрицу. Использование видовой матрицы может привести к тому, что механизмы **OpenGL**, оперирующие в видовых координатах **(**такие как освещение и отсекающие плоскости**)** перестанут работать.
- Не вносите слишком интенсивные изменения в одну и ту же матрицу. Например, не следует создавать анимацию вращения, много раз вызывая **glRotate*()** с одним и тем же углом. Вместо этого используйте **glLoadIdentity()** для инициализации нужной матрицы каждый кадр, а затем вызывайте **glRotate*()** один раз с полным углом вращения для этого кадра.
- Вы можете рассчитывать на то, что при многократном прохождении через базу данных визуализации каждый раз будут сгенерированы одни и те же фрагменты, только в том случае, если такое поведение гарантировано правилами инвариантности, установленными для подчиняющейся реализации **OpenGL**. Иначе, при двух разных проходах через базу данных визуализации **(**например, при выполнении списков отображения**)** могут быть сгенерированы разные наборы фрагментов.
- Не рассчитывайте на то, что **OpenGL** будет рапортовать об ошибках. Команды внутри списка генерируют ошибки только тогда, когда список выполняется.
- Размещайте ближнюю плоскость отсечения перспективного объема видимости максимально далеко от точки наблюдения для оптимизации работы буфера глубины.
- Вызывайте **glFlush()** для форсирования всех ранее вызванных команд **OpenGL** к исполнению. Не рассчитывайте на то, что **glGet*()** или **glIs*()** выполнят формирование потока визуализации. Команды опроса вызывают исполнение только части потока, необходимой для возвращения корректных данных, но не гарантируют выполнения всех команд визуализации.
- Деактивируйте цветовое микширование при визуализации предопределенных изображений **(**например, при использовании **glCopyPixels()).**

- Используйте полный диапазон буфера аккумуляции. Например, если вы аккумулируете **4** изображения, во время аккумуляции берите по четверти от каждого из них.
- Если требуется именно двумерная растеризация, вы должны аккуратно задавать ортографическую проекцию и вершины примитивов, которые должны быть растеризованы. Ортографическая проекция должна быть задана с целыми координатами, как показано в следующем примере:

```
gluOrtho2D(0, width,0,height);
```

где ***width*** и ***height*** – это размеры порта просмотра. При такой проекционной матрице координаты примитивов и пиксельных изображений должны быть представлены целыми числами, чтобы растеризация была предсказуемой. Например, **glRecti(0,0,1,1)** точно закрасит нижний левый угол порта просмотра, а **glRasterPosi(0,0)** определенно позиционирует неизмененное изображение в нижний левый угол порта просмотра. Однако вершины точек, вершины линий и позиции битовых карт должны помещаться посередине между двумя целыми числами. Например, линия, нарисованная из **(*x1*, *0.5*)** в **(*x2*, *0.5*)** гарантированно будет отображена в нижнем ряду пикселей порта просмотра, а точка, нарисованная с координатами **(*0.5*, *0.5*)**, заполнит тот же пиксель, что и **glRecti(0,0,1,1)**.

- Оптимальный компромисс, позволяющий задавать все примитивы в целых локациях, получая предсказуемую визуализацию, заключается в том, чтобы перенести ***x*** и ***y*** на ***0.375***, как показано в следующем примере. Такой перенос держит полигоны и границы пиксельных изображений на безопасном расстоянии от центров пикселей, одновременно перемещая вершины линий достаточно близко к этим центрам.

```
glViewport(0,0,width,height);
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
gluOrtho2D(0,width,0,height);
gllMatrixMode(GL_MODELVIEW);
glLadIdentity();
glTranslatef(0.375,0.375,0.0);
/* визуализируйте все примитивы в целых локациях */
```

-
- Избегайте использования отрицательного ***w*** в координатах вершин и отрицательного ***q*** в координатах текстуры. **OpenGL** может не усекать такие вершины корректно, и может совершать ошибки интерполяции при закраске примитивов с такими вершинами.
- Не предполагайте, что точность операций **OpenGL** зависит от типов аргументов команд библиотеки. Например, если вы используете **glRotated()**, вы не должны ожидать, что геометрический конвейер сохраняет двойную точность чисел с плавающей точкой в процессе своей работы. Возможно, что до обработки аргументы **glRotated()** будут преобразованы к другому типу.

Для увеличения скорости работы OpenGL

- Если частым изменениям подвержено только одно свойство материала (например, в каждой вершине), используйте **glColorMaterial()**. Используйте **glMaterial()** для редких изменений или в тех случаях, когда частые изменения претерпевают несколько свойств материала.
- Вместо того, чтобы загружать свою копию единичной матрицы, используйте **glLoadIdentity()**.

- Используйте специфические команды, такие как **glRotate*()**, **glTranslate*()** и **glScale*()** вместо того, чтобы составлять свои собственные матрицы вращений, переносов и масштабирования.
- Используйте команды опроса только в случаях, когда вашему приложению требуется ряд значений переменных состояния для своих собственных вычислений. Если вашему приложению требуется несколько переменных состояния из одной группы атрибутов, для их сохранения и восстановления используйте **glPushAttrib()** и **glPopAttrib()**.
- Инкапсулируйте потенциально дорогие (с точки зрения быстродействия) изменения состояния в списках отображения.
- Инкапсулируйте в списках отображения вызовы команд визуализации объектов, которые используются часто.
- Для инкапсуляции текстурных данных используйте текстурные объекты. Поместите все вызовы команды **glTexImage*()** (включая мипмапы), необходимые для полного определения текстуры, а также ассоциированные вызовы **glTexParameter*()** в текстурный объект. Свяжите текстурный объект для выбора текстуры.
- Если ситуация позволяет это, используйте **gl*TexSubImage()** для замены всего изображения текстуры или его части, вместо того, чтобы выполнять более затратные операции по удалению и созданию новой текстуры.
- Если ваша реализация **OpenGL** поддерживает высокоскоростное подмножество резидентных текстур, попытайтесь сделать все ваши текстуры резидентными – то есть, сделать так, чтобы они уместились в текстурной памяти. Если это необходимо, сокращайте размеры или разрешение внутреннего формата ваших текстур до тех пор, пока они не влезут в память. Если такое сокращение приводит к недопустимо выглядящим текстурным объектам, вы можете назначить некоторым текстурам более низкий приоритет. В этом случае при нехватке памяти такие текстуры будут вытеснены из текстурной памяти.
- Для сокращения нагрузки на сеть в клиент – серверном окружении используйте вычислители даже для тесселяции даже самых простых поверхностей.
- Если возможно, поставляйте нормали единичной длины и избегайте частых обращений к **GL_NORMALIZE**. При использовании освещения избегайте вызовов **glScale*()**, поскольку масштабирование практически всегда требует перенормализации векторов нормалей.
- Если плавная закраска не требуется, установите режим **GL_FLAT** командой **glShadeModel()**.
- Если возможно используйте только один вызов **glClear()** на каждый кадр. Не используйте **glClear()** для очистки небольших областей буферов, используйте ее только для полной или близкой к полной очистке.
- Для рисования нескольких треугольников используйте один вызов **glBegin(GL_TRIANGLES)** (то же касается примитивов **GL_QUADS** и **GL_LINES**), а не множество таких вызовов или вызовов **glBegin(GL_POLYGON)**. Даже если должен быть нарисован только один треугольник, используйте **GL_TRIANGLES**, а не **GL_POLYGON**.
- Некоторые реализации **OpenGL** извлекают пользу из хранения вершинных данных в вершинных массивах. Использование вершинных массивов сокращает количество вызовов функций. Некоторые реализации извлекают пользу путем блочной обработки или повторного использования уже обработанных вершин.
- Вообще, следует использовать векторные версии команд для передачи заранее вычисленных данных и скалярные версии – для данных, которые были вычислены недалеко от момента вызова.
- Избегайте совершения ненужных изменений состояния, таких как установка цвета перед каждой вершиной в режиме плоской заливки.
- Убедитесь в том, что вы заблокировали растратные растеризующие и пофрагментные операции перед рисованием или копированием изображений. Если приказано, **OpenGL** будет накладывать текстуру даже на пиксельные изображения.
- Избегайте различия в полигональном режиме для лицевых и обратных гранях, кроме тех случаев, когда это абсолютно необходимо.

Советы по работе с GLX

- Используйте **glXWaitGL()** вместо **glFinish()**, чтобы команды визуализации **X** следовали за командами визуализации **GL**.
- Точно так же используйте **glXWaitX()** вместо **glFinish()**, чтобы команды визуализации **GL** следовали за командами визуализации **X**.
- Будьте осторожны при использовании **glXChooseVisual()**, поскольку Булевский выбор осуществляется по точному соответствию. Поскольку некоторые реализации не выполняют экспорта визуальных объектов со всеми Булевскими комбинациями возможностей, вы должны вызвать **glXChooseVisual()** несколько раз с разными Булевскими значениями до того, как сдадитесь. Например, если нет однократно буферизованного визуального объекта с нужными характеристиками, проверьте наличие визуального объекта с двойной буферизацией и теми же характеристиками. Он может существовать, и его легко использовать.

Приложение G. Инвариантность OpenGL

OpenGL не является точной пиксельной спецификацией. Это означает, что она не гарантирует точного совпадения между изображениями, сгенерированными разными реализациями. Однако **OpenGL** специфицирует ряд четких соглашений по некоторым моментам, касающимся изображений, сгенерированных одной и той же реализацией. В этом приложении описаны правила инварианта, определяющие эти случаи.

Самый очевидный и фундаментальный момент – это повторяемость. Любая удобоваримая реализация **OpenGL** генерирует один и тот же результат, каждый раз, когда специфическая последовательность команд выполняется при одних и тех же начальных условиях. Хотя такое заявление удобно для тестирования и проверок, оно часто не устраивает программистов, поскольку достаточно сложно приводить систему к одним и тем же начальным условиям. Например, рисование сцены дважды – второй раз после переключения переднего и заднего буферов – не соответствует требованию об одинаковых начальных условиях. Таким образом, повторяемость не может быть использована в качестве гарантии стабильного изображения в режиме двойной буферизации.

Рисование линии и стирание ее цветом фона – это простой и полезный алгоритм, рассчитанный на инвариантное исполнение. Он работает, только если в случае обоих линий будут сгенерированы фрагменты на одних и тех же (***x***, ***y***). **OpenGL** требует, чтобы координаты генерируемых растеризацией фрагментов не зависели от содержимого буфера кадра, того, разрешена ли запись в цветовой буфер, значений всех матриц, которые не находятся на вершинах матричных стеков, параметров теста отреза, всех масок записи, всех очищающих величин, текущих цвета, индекса, нормали, координат текстуры и флага ребра, текущих цвета растра, индекса растра и координат текстуры растра и свойств материала. Она также требует, чтобы в точности те же фрагменты, включая цветовые значения, генерировались вне зависимости от содержимого буфера кадра, того, разрешен ли цветовой буфер для записи, значений всех матриц, которые не находятся на вершинах матричных стеков, параметров теста отреза, всех масок записи и всех очищающих величин.

OpenGL также предполагает, но не требует, чтобы генерация фрагментов проходила независимо от матричного режима, глубин матричных стеков, параметров альфа теста (отличных от состояния активности), параметров трафарета (отличных от состояния активности), параметров цветового наложения (отличных от состояния активности), логических операций (но не от того, активны ли они), а также режимов хранения пикселей и параметров пиксельного переноса. Поскольку инвариантность по отношению к некоторых флагам активности не рекомендуется, вы должны использовать другие параметры для блокировки соответствующих механизмов, когда требуется инвариантная визуализация. Например, чтобы провести инвариантную визуализацию с

включенным и выключенным цветовым наложением, установите факторы наложения в **GL_ONE** и **GL_ZERO** вместо того, чтобы вызывать **glDisable(GL_BLEND)**. (Это может потребоваться, например, если в некоторых кадрах анимации должно работать цветовое наложение, а в некоторых – нет.) Альфа тестирование, тест трафарета, тест глубины и логические операции – все это может быть заблокировано подобным образом.

Наконец, **OpenGL** требует, чтобы пофрагментая арифметика, такая как цветовое наложение и тест глубины, была инвариантна по отношению к (независима от) всех элементов состоянию **OpenGL** за исключением тех, которые непосредственно ее определяют. Например, единственные параметры **OpenGL**, влияющие на то, как производятся арифметические операции для цветового наложения – это факторы источника и приемника, а также флаг активности цветового наложения. Цветовое наложение инвариантно по отношению ко всем остальным изменениям состояния. Такая инвариантность имеет место для теста отреза, альфа теста, теста глубины, теста трафарета, цветового наложения, цветового микширования, логических операций и масок записи.

Как результат всех этих требований инвариантности, **OpenGL** может гарантировать, что два изображения, одновременно или по очереди визуализированные в разных цветовых буферах с использованием одной и той же последовательности команд, являются попиксельно идентичными. Это касается всех цветовых буферов в буфере кадра, а также всех цветовых буферов во внеэкранном буфере, но это не относится к буферу кадра и внеэкранному буферу взятым вместе.

Приложение H. OpenGL и оконные системы

OpenGL доступна на множестве различных платформ и работает со множеством различных оконных систем. **OpenGL** разработана для дополнения оконных систем, а не для дублирования их функциональности. Вследствие этого **OpenGL** выполняет визуализацию геометрии и изображений в двумерном и трехмерном пространствах, но она не управляет окнами и не обрабатывает события пользовательского ввода.

Однако базовые определения большинства оконных систем не поддерживают такую сложную библиотеку, как **OpenGL**, с ее сложными и множественными форматами пикселей, буферами трафарета, глубины и аккумуляции, а также двойную буферизацию. Для большинства оконных систем были добавлены функции, расширяющие эти системы поддержкой **OpenGL**.

В этом приложении представлены расширения, определенные для нескольких оконных и операционных систем: **X Windows System**, **Apple Mac OS**, **IBM OS/2 Warp** и **Microsoft Windows 95/98/NT**. Для полного понимания приложения вам понадобятся некоторые знания об оконных системах.

GLX: Расширения OpenGL для X Window System

В системе **X Window** визуализация **OpenGL** реализована в качестве расширения к **X** в формальном понимании этого термина. **GLX** представляет собой расширение протокола **X** (и ассоциированного с ним **API**) для взаимодействия команд **OpenGL** с расширенным сервером **X**. Соединение и аутентификация производятся в соответствии с формальным механизмом **X**.

Как и в случае других расширений **XX**, команды **OpenGL** инкапсулируются внутри потока байт **X** согласно определенному сетевому протоколу для поддержки клиент – серверной визуализации **OpenGL**. Поскольку быстродействие является критическим для трехмерной визуализации, расширение **OpenGL** для **X** позволяет **OpenGL** избегать вмешательства сервера **X** в кодирование, копирование и интерпретацию данных и вместо этого осуществлять визуализацию непосредственно на графический конвейер.

GLX версии **1.3** добавил несколько новшеств, например, новую структуру данных **GLXFBConfig**, описывающую конфигурацию буфера кадра **GLX (**включая глубину компонент цветового буфера, а также типы и размеры буферов глубины, трафарета, аккумуляции и дополнительных буферов**).** Структура **GLXFBConfig** описывает эти атрибуты для поверхности визуализации **GLXDrawable. (**В **X**, поверхность визуализации называется **Drawable.)**

GLX 1.3 поддерживает **3** типа поверхностей **GLXDrawable: GLXWindow**, **GLXPixmap** и **GLXPbuffer. GLXWindow** является экранным, а остальные – внеэкранными. Поскольку поверхность **GLXPixmap** имеет ассоциированную с ней карту пикселей **X**, и **OpenGL** и **X** могут осуществлять визуализацию на этой поверхности. На поверхности **GLXPbuffer** может осуществлять визуализацию только **OpenGL**, такие поверхности предназначены для сохранения пиксельных данных в невидимой памяти буфера кадра. **(**Не гарантируется, что внеэкранная визуализация поддерживается непосредственными визуализаторами**.)**

X Visual является важной структурой данных для управления форматом пикселей окна **OpenGL.** Информация о формате пикселей отслеживается с помощью переменной типа **XVisualInfo**, включая тип пикселей **(**индексные или **RGBA)**, однократная или двойная буферизация, разрешение цветов, а также наличие буферов глубины, трафарета и аккумуляции. Стандартные объекты **X Visual (**например, **PseudoColor** или **TrueColor)** не описывают деталей пиксельного формата, так что каждая реализация должна расширять число объектов **X Visual.**

В **GLX 1.3 GLXWindow** имеет **X Visual**, ассоциированный с его **GLXFBConfig.** Для **GLXPixmap** и **GLXPBuffer** может быть, а может и не быть похожего ассоциированного **X Visual.** До **GLX 1.3** все поверхности **(**окна или пиксельные карты**)** были ассоциированы с **X Visual. (**До версии **1.3 P –** буферы не были частью **GLX.)**

Если вам требуется изучить внутренности **GLX**, его спецификацию можно найти по адресу**: ftp://sgigate.sgi.com/pub/opengl/doc/opengl1.2.**

Инициализация

Используйте функции **glXQueryExtension()** и **glXQueryVersion()** для установления того, присутствует ли расширение **GLX** для сервера **X** и, если да, то какой он версии. **glXQueryExtensionString()** возвращает информацию о клиент – серверном соединении. **glXGetClientString()** возвращает информацию о клиентской библиотеке, включая расширения и номер версии. **glXGetServerString()** возвращает такую же информацию о сервере.

glXChooseFBConfig() возвращает указатель на массив структур **GLXFBConfig**, описывающих все возможные конфигурации буфера **GLX**, удовлетворяющие атрибутам, заданным клиентом. Вы можете использовать **glXGetFBConfigAttrib()** для установления того, поддерживает ли конфигурация буфера кадра конкретный атрибут **GLX.** Также вы можете вызвать **glXGetVisualFromFBConfig()** для получения **X Visual**, ассоциированного с **GLXFBConfig.**

Процесс создания областей визуализации несколько различается в зависимости от типа поверхности. Для **GLXWindow** сначала создайте **X Window** с **X Visual**, соответствующим **GLXFBConfig.** Затем используйте это **X Window** при вызове функции **glXCreateWindow()**, которая возвращает **GLXWindow.** Для **GLXPixmap** сначала создайте **X Pixmap** с глубиной пикселей, соответствующей **GLXFBConfig.** Затем используйте эту **Pixmap** при вызове функции **glXCreatePixmap()** для создания **GLXPixmap. GLXPbuffer** не требует **X Window** или **X Pixmap –** просто вызовите **glxCreatePbuffer()** с соответствующей структурой **GLXFBConfig.**

Замечание: Если вы используете GLX 1.2 или меньше, у вас нет структуры GLXFBConfig. Вместо нее используйте функцию **glXChooseVisual()**, возвращающую указатель на структуру XVisualInfo, описывающую X Visual, соответствующий атрибутам, заданным клиентом. Вы можете опросить X Visual на предмет поддержки определенного атрибута OpenGL с помощью функции **glXGetConfig()**. Для визуализации на внеэкранную пиксельную карту, вы должны использовать функцию **glXCreateGLXPixmap()**.

Управление визуализацией

GLX предоставляет несколько функций для создания и управления контекстом визуализации **OpenGL.** Также предоставляются функции для таких задач**,** как обработка событий **GLX,** синхронизация потоков исполнения **X** и **OpenGL,** переключение переднего и заднего буферов**,** а также использование **X font.**

Управление контекстом визуализации OpenGL

Контекст визуализации **OpenGL** создается функцией **glXCreateNewContext().** Один из аргументов этой функции позволяет вам запросить непосредственный контекст визуализации**,** который минует **X Server,** как было описано ранее**. (**Для осуществления непосредственной визуализации соединение с сервером **X** должно быть локальным и реализация **OpenGL** должна поддерживать непосредственную визуализацию**.) glXCreateNewContext()** также позволяет нескольким контекстам разделять списки отображения и текстурные объекты**.** Вы можете проверить**,** является ли контекст **OpenGL** непосредственным с помощью функции **glXIsDirect().**

glXMakeContextCurrent() привязывает контекст визуализации к текущему потому визуализации и устанавливает две текущие поверхности для рисования**.** Вы можете рисовать на одной из текущих поверхностей для рисования и считывать пиксели с другой**.** Во многих ситуациях**,** обе этих поверхности являются одной и той же поверхностью **GLXDrawable. glXGetCurrentContext()** возвращает текущий контекст**.** Вы можете получить текущую поверхность для рисования с помощью **glXGetCurrentDrawable(),** текущую поверхность для считывания **–** с помощью функции **glXGetCurrentReadDrawable(),** а текущий **X Display –** с помощью **glXGetCurrentDisplay().** Вы можете использовать **glXQueryContext()** для выяснения текущих значений атрибутов контекста**.**

В любое время для одного потока может существовать только один текущий контекст**.** Если у вас создано несколько контекстов**,** вы можете копировать группы переменных состояния **OpenGL** из одного в другой с помощью функции **glXCopyContext().** Когда вы закончили работы с отдельным контекстом**,** уничтожьте его функцией **glXDestroyContext().**

Замечание: Если у вас GLX 1.2 или меньше, используйте **glXCreateContext()** для создания контекста визуализации и **glXMakeCurrent()** для установки его в качестве текущего. Вы не можете объявить поверхность в качестве отдельной поверхности для чтения, и у вас нет функции **glXGetCurrentReadDrawable()**.

Обработка событий GLX

События появились в **GLX 1.3.** Они возвращаются в стандартном потоке событий **X11.** Обработка событий **GLX** была добавлена специально для разрешение проблемы неоднозначности содержимого **GLXPbuffer.** В **GLX 1.3** вы можете использовать **glXSelectEvent()** для выбора только одного события **– GLX_PBUFFER_CLOBBER_MASK.** С помощью стандартного механизма обработки события **X** вы теперь можете засечь повреждение части **GLXPbuffer** и принять меры для восстановления**,** если это необходимо**. (**Вы также можете вызвать **glXGetSelectedEvent()** для выяснения того**,** проводите ли вы мониторинг этого события**.)**

Синхронизация исполнения

Во избежании запросов об исполнении **X** до того, как визуализация **OpenGL** закончится, вызовите **glXWaitGL().** В этом случае гарантируется, что все вызванные команды **OpenGL** будут выполнены до того, как начнут исполняться функции визуализации **X,** вызванные после **glXWaitGL().** Хотя тех же результатов можно достигнуть с помощью команды **glFinish(), glXWaitGL()** не требует дополнительного обращения к серверу и, таким образом, работает эффективнее в ситуациях, когда клиент и сервер **OpenGL** размещены на разных машинах.

Для предотвращения исполнения последовательностей команд **OpenGL** перед исполнением вызванных ранее функций визуализации **X,** используйте **glXWaitX().** В этом случае гарантируется, что все вызванные функции **X** будут выполнены до того, как начнут исполняться команды визуализации **OpenGL,** вызванные после **glXWaitX().**

Переключение буферов

Для режима двойной буферизации передний и задний буферы можно переключить вызовом **glXSwapBuffer(). glFlush()** выполняется как часть этой функции.

Использование X Font

Короткий путь к использованию **X Font** в **OpenGL** предоставляется функцией **glXUseXFont().** Эта функция строит списки отображения для каждого запрошенного шрифта и его размера, каждый из которых вызывает **glBitmap().**

Очистка поверхностей

После завершения визуализации, вы можете уничтожить поверхность для рисования с помощью функции **glXDestroyWindow(), glXDestroyPixmap()** или **glXDestroyPbuffer(). (**Эти функции отсутствуют в **GLX** версии **1.2** и ниже, хотя там есть функция **glXDestroyGLXPixmap()** по смыслу аналогичная **glXDestroyPixmap().)**

Прототипы GLX

Инициализация

Выяснить, определено ли расширение **GLX** для сервера **X:**

bool glXQueryExtension (Display *dpy, int *errorBase, int *eventBase);

Запросить версию и информацию о расширениях клиента и сервера:

bool glXQueryVersion (Display *dpy, int *major, int *minor);

const char* glXGetClientString (Display *dpy, int name);

const char* glXQueryServerString (Display *dpy, int screen, int name);

const char* glXQueryExtensionString (Display *dpy, int screen);

Получить доступные конфигурации буфера кадра:

GLXFBConfig* glXGetFBConfigs (Display *dpy, int screen, int* nelements);

GLXFBConfig* glXChooseFBConfig (Display *dpy, int screen, const int attribList, int *nelements);

Опросить буфер кадра на предмет атрибута **GLX** или информации **X:**

int glXGetFBConfigAttrib (Display *dpy, GLXFBConfig config, int attribute, int *value);

XVisualInfo* glXGetVisualFromFB (Display *dpy, GLXFBConfig config);

Создать поверхности для поддержки визуализации **(**экранной или внеэкранной**):**

GLXWindow glXCreateWindow (Display *dpy, GLXFBConfig config, Window win, const int* attribList);

GLXPixmap glXCreatePixmap (Display *dpy, GLXFBConfig config, Pixmap pixmap, const int* attribList);

GLXPbuffer glXCreatePbuffer (Display *dpy, GLXFBConfig config, const int* attribList);

Управление визуализацией

Управление и опрос контекста визуализации **OpenGL:**

GLXContext glXCreateNewContext (Display *dpy, GLXBFConfig config, int renderType, GLXContext shareList, bool direct);

bool glXMakeContextCurrent (Display *dpy, GLXDrawable drawable, GLXDrawable read, GLXContext context);

void glXCopyContext (Display *dpy, GLXContext source, GLXContext dest, unsigned long mask);

bool glXIsDirect (Display *dpy, GLXContext context);

GLXContext glXGetCurrentContext (void);

Display * glXGetCurrentDisplay (void);

GLXDrawable glXGetCurrentDrawable (void);

GLXDrawable glXGetCurrentReadDrawable (void);

int glXQueryContext (Display *dpy, GLXContext context, int attribute, int* value);

void glXDestroyContext (Display *dpy, GLXContext context);

Запрос событий **GLX:**

int glXSelectEvent (Display *dpy, GLXDrawable drawable, unsigned long eventMask);

int glXGetSelectedEvent (Display *dpy, GLXDrawable drawable, unsigned long* eventMask);

Синхронизация исполнения**:**

void glXWaitGL (void);

void glXWaitX (void);

Поменять местами передний и задний буферы:

void glXSwapBuffers (Display *dpy, GLXDrawable drawable);

Использовать **X Font:**

void glXUseXFont (Font font, int *first*, int count, int *listBase*);

Очистить поверхности:

void glXDestroyWindow (Display *dpy, GLXWindow win);

void glXDestroyPixmap (Display *dpy, GLXPixmap pixmap);

void glXDestroyPbuffer (Display *dpy, GLXPbuffer pbuffer);

Устаревшие прототипы

Следующие функции являются устаревшими по отношению к **GLX 1.3.** Если вы используете **GLX 1.2** или его предшественника, вам могут понадобиться некоторые из них.

Получить желаемый объект **X Visual:**

XVisualInfo* glXChooseVisual (Display *dpy, int screen, int* attribList);

int glXGetConfig (Display *dpy, XVisualInfo* visual, int attrib, int* value);

Управление контекстом **OpenGL:**

GLXContext glXCreateContext (Display *dpy, XVisualInfo *visual, GLXContext shareList, bool direct);

bool glXMakeCurrent (Display *dpy, GLXDrawable drawable, GLXContext context);

Внеэкранная визуализация:

GLXPixmap glXCreateGLXPixmap (Display *dpy, XVisualInfo *visual, Pixmap pixmap);

void glXDestroyGLXPixmap (Display *dpy, GLXPixmap pix);

AGL: Расширения OpenGL для Apple Macintosh

В этом разделе описаны функции, определенные в качестве расширений **OpenGL** для **Apple Macintosh (AGL).** Здесь требуется понимание того, как **Macintosh** производит графическую визуализацию **(QuickDraw).**

Для получения более подробной информации **(**включая способы получения программной библиотеки для **Macintosh)** вы можете получить на сайте **Apple: http://www.apple.com/opengl.**

Визуализация **OpenGL** для **Macintosh** реализуется через библиотеку, которая либо присоединяется к приложению, либо является резидентной для приложения, которое хочет ее использовать. Для систем, в которых отсутствует аппаратная поддержка,

OpenGL реализована на программном уровне. Если поддержка имеется, используются ее возможности, соответствующие требованиям конвейера **OpenGL**, а остальные реализуются программно.

Тип данных **AGLPixelFormat (**аналог **AGL** для **XVisualInfo)** хранит информацию о формате пикселей, включая тип пикселей **(**индексные или **RGBA)**, тип буферизации **(**однократная или двойная**)**, разрешение цветов, а также наличие буферов глубины, трафарета и аккумуляции.

В отличие от реализаций **OpenGL** для других систем **(**например, для **X Window System)** клиент – серверная модель не используется. Однако вам все равно может понадобиться команда **glFlush()**, поскольку некоторая аппаратура может буферизовать команды **OpenGL** и требовать толчка, чтобы их выполнить.

Инициализация

Используйте функцию **aglGetVersion()** для определения присутствующей версии **AGL** для **Macintosh.**

Соответствие возможностей нижележащих графических устройств и ваших требований к буферам визуализации устанавливается с помощью функции **aglChoosePixelFormat().** Она возвращает структуру **AGLPixelFormat** или **NULL** в зависимости от того, могут ли быть выполнены ваши требования.

Визуализация и контексты

Для создания и управления контекстом **OpenGL AGL** предоставляет несколько функций. Вы можете использовать контекст и для визуализации в окне и для внеэкранной визуализации. Также предоставляются функции для переключения переднего и заднего буферов, для использования шрифтов и для изменения настроек буфера в ответ на перемещение, изменение размера или событие от аппаратуры. Буферы для программной визуализации **(**и в некоторых случаях для аппаратной**)** создаются в системной памяти.

Управление контекстом визуализации OpenGL

Контекст **OpenGL** создается **(**должен быть как минимум один контекст на каждое окно, в которое осуществляется визуализация**)** с помощью функции **aglCreateContext().** Она принимает выбранный вами формат пикселей в качестве аргумента и использует его при инициализации буфера.

Используйте **aglSetDrawable()** для присоединения контекста к поверхности рисования и, далее, **aglSetCurrentContext()** для того, чтобы сделать контекст текущим. Только один контекст может быть текущим для одного потока управления в каждый конкретный момент. Это определяет, на какую поверхность производится визуализация, и какой контекст следует использовать совместно с ней. Для визуализации на одну и ту же поверхность может быть использовано более одного контекста **(**не одновременно**).** Определить текущий контекст визуализации и поверхность визуализации позволяют две функции: **aglGetCurrentContext()** и **aglGetDrawable().**

Если у вас создано несколько контекстов, вы можете копировать группы переменных состояния **OpenGL** из одного в другой с помощью функции **aglCopyContext().** Когда вы закончили работы с отдельным контекстом, уничтожьте его функцией **aglDestroyContext().**

Экранная визуализация

Для экранной визуализации сначала создайте формат пикселей. Затем создайте контекст на основании этого формата пикселей и присоедините его к окну с помощью функции **aglSetDrawable().** Прямоугольник буфера может быть изменен с помощью вызова **aglSetInteger(AGL_BUFFER_RECT, ...).**

Внеэкранная визуализация

Для внеэкранной визуализации создайте формат пикселей с атрибутом **AGL_OFFSCREEN.** Затем создайте контекст на основании этого формата пикселей и свяжите его с экраном с помощью функции **aglSetOffScreen().**

Полноэкранная визуализация

Для полноэкранной визуализации создайте формат пикселей с атрибутом **AGL_FULLSCREEN.** Затем создайте контекст на основании этого формата пикселей и свяжите его с экраном с помощью функции **aglSetFullScreen().**

Переключение буферов

Для поверхностей с двойной буферизацией **(**для формата пикселей текущего контекста**)** используйте **aglSwapBuffers()** для переключения переднего и заднего буферов. Переключаемый прямоугольник может быть настроен с помощью вызова **aglSetInteger(AGL_SWAP_RECT, ...). glFlush()** выполняется как часть этой функции.

Обновление буферов визуализации

Apple Macintosh требует от вас производить собственную обработку событий и не позволяет библиотекам автоматически подключаться к потоку событий. Чтобы поверхности под управлением **OpenGL** могли изменяться в размере, положении и глубине пикселей, **AGL** представляет функцию **aglUpdateContext().**

Эта функция должна вызываться вашим кодом обработки событий, как только одно из этих событий произойдет на текущей поверхности. В идеале, стоит перерисовать сцену после обновления для учета изменений в буфере визуализации.

Использование шрифтов Apple Macintosh

Простой путь использования шрифтов **Apple Macintosh** предоставляется с помощью функции **aglUseFont().**Эта функция строит списки отображения для каждого запрошенного шрифта и его размера, каждый из которых вызывает **glBitmap().**

Обработка ошибок

Для расширения **OpenGL** в системе **Apple Macintosh** предоставляется механизм обработки ошибок. Когда ошибка произошла, вы можете вызвать функцию **aglGetError()** для более точного описания того, что вызвало эту ошибку.

Прототипы AGL

Инициализация

Получить информацию о версии:

void aglGetVersion (GLint* *major*, GLint* *minor*);

Получить доступные форматы пикселей:

AGLPixelFormat aglChoosePixelFormat (const AGLDevice* *gdevs*, GLint *ndev*, onst GLint **attribs*);

void aglDestroyPixelFormat (AGLPixelFormat *pix*);

AGLPixelFormat aglNextPixelFormat (AGLPixelFormat *pix*);

GLboolean aglDescribePixelFormat (AGLPixelFormat *pix*, GLint *attrib*, GLint **value*);

AGLDevice* aglDevicesOfPixelFormat (AGLPixelFormat *pix*, GLint **ndevs*);

Информация о визуализаторе:

AGLRendererInfo aglQueryRendererInfo (const AGLDevice* *gdevs*, GLint *ndev*);

void aglDestroyRendererInfo (AGLRendererInfo *rend*);

AGLRendererInfo aglNextRendererInfo ((AGLRendererInfo *rend*);

GLboolean aglDescribeRenderer (AGLRendererInfo *rend*, GLint *prop*, GLint **value*);

Управление визуализацией

Управление контекстом визуализации **OpenGL:**

AGLContext aglCreateContext (AGLPixelFormat *pix*, AGLContext *share*);

GLboolean aglDestroyContext (AGLContext *ctx*);

GLboolean aglCopyContext (AGLContext *src*, AGLContext *dst*, GLuint *mask*);

GLboolean aglUpdateContext (AGLContext *ctx*);

Установка текущего состояния:

GLboolean aglSetCurrentContext (AGLContext *ctx*);

AGLContext aglGetCurrentContext (void);

Функции поверхностей:

GLboolean aglSetDrawable (AGLContext *ctx*, AGLDrawable *draw*);

GLboolean aglSetOffScreen (AGLContext *ctx*, GLsizei *width*, GLsizei *height*, GLsizei *rowbytes*, GLvoid **baseaddr*);

GLboolean aglSetFullScreen (AGLContext *ctx*, GLsizei *width*, GLsizei *height*, GLsizei *freq*, GLint *device*);

AGLDrawable aglGetDrawable (AGLContext *ctx*);

Функции виртуального экрана:

GLboolean aglSetVirtualScreen (AGLContext *ctx*, GLint *screen*);

GLint aglGetVirtualScreen (AGLContext *ctx*);

Конфигурирование глобальных опций библиотеки:

GLboolean aglConfigure (GLenum *pname*, GLuint *param*);

Функции переключения:

void aglSwapBuffers (AGLContext *ctx*);

Опции контекстов:

GLboolean aglEnable (AGLContext *ctx*, GLenum *pname*);

GLboolean aglDisable (AGLContext *ctx*, GLenum *pname*);

GLboolean aglIsEnabled (AGLContext *ctx*, GLenum *pname*);

GLboolean aglSetInteger (AGLContext *ctx*, GLenum *pname*, contst GLint **params*);

GLboolean aglGetInteger (AGLContext *ctx*, GLenum *pname*, contst GLint **params*);

Шрифтовые функции:

GLboolean aglUseFont (AGLContext *ctx*, GLint *fontID*, Style *face*, GLint *size*, GLint *first*, GLint *count*, GLint *base*);

Функции работы с ошибками:

GLenum aglGetError (void);

const GLubyte *aglErrorString (GLenum *code*);

Функция сброса:

void aglResetLibrary (void);

PGL: Расширения OpenGL для IBM OS/2 Warp

Визуализация **OpenGL** для **IBM OS/2 Warp** осуществляется с помощью функций **PGL,** добавленных с целью интеграции **OpenGL** в стандартный **Presentation Manager IBM. OpenGL** совместно с **PGL** поддерживают и непосредственный контекст **(**который часто работает быстрее**)** и опосредованный контекст **(**который позволяет некоторую долю интеграции Интерфейса программирования графики **(Graphics Programming Interface -- GPI)** и визуализации **OpenGL).**

Тип данных **VISUALCONFIG (**аналог **PGL** для **XVisualInfo)** хранит информацию о формате пикселей**,** включая тип пикселей **(**индексные или **RGBA),** тип буферизации **(**однократная или двойная**),** разрешение цветов**,** а также наличие буферов глубины**,** трафарета и аккумуляции**.**

Для получения более подробной информации **(**включая способы получения программной библиотеки для **IBM OS/2 Warp, Version 3.0)** вы можете получить на сайте **IBM: http://www.austin.ibm.com/software/opengl.**

Инициализация

Используйте функции **pglQueryCapability()** и **pglQueryVersion()** для определения того, поддерживается ли **OpenGL** на данной машине и, если так, как она поддерживается и какая версия присутствует. **pglChooseConfig()** возвращает указатель на структуру **VISUALCONFIG**, описывающую визуальную конфигурацию, наилучшим образом подходящую к заданным клиентом атрибутам. Список отдельных визуальных конфигураций, поддерживаемых графическим устройством можно получить с помощью **pglQueryConfigs()**.

Управление визуализацией

PGL предоставляет несколько функций для создания и управления контекстом визуализации **OpenGL**, захвата содержимого битовой карты, синхронизации потоков **Presentation Manager** и **OpenGL**, переключения буферов, использования цветовой палитры и использования логического шрифта **OS/2**.

Управление контекстом визуализации OpenGL

Контекст визуализации **OpenGL** создается с помощью функции **pglCreateContext()**. Один из аргументов этой функции позволяет запросить непосредственный контекст, который, минуя **GPI**, производит визуализацию прямо в окно **PM**, что обычно происходит быстрее. Вы можете определить, является ли контекст непосредственным с помощью функции **pglIsDirect()**.

Чтобы сделать контекст текущим, используйте **pglMakeCurrent()**; **pglGetCurrentContext()** возвращает текущий контекст. Вы также можете получить текущее окно с помощью **pglGetCurrentWindow()**. Вы можете копировать значения переменных состояния **OpenGL** с помощью **pglCopyContext()**. Когда вы закончите работы с контекстом **OpenGL**, вы можете удалить его с помощью **pglDestroyContext()**.

Доступ к битовой карте переднего буфера

Для доступа к битовому представлению содержимого переднего буфера используйте функцию **pglGrabFrontBitmap()**. В качестве части этой функции вызывается команда **glFlush()** после чего вы можете получить доступ к битовой карте, но она будет доступна только для чтения. Немедленно после того, как доступ завершен, вы должны вызвать функцию **pglReleaseFrontBitmap()** для восстановления режима записи в передний буфер.

Синхронизация исполнения

Во избежании запросов об исполнении функций **GPI** до того, как визуализация **OpenGL** закончится, вызовите **pglWaitGL()**. В этом случае гарантируется, что все вызванные команды **OpenGL** будут выполнены до того, как начнут исполняться функции визуализации **GPI**, вызванные после **pglWaitGL()**. Для предотвращения исполнения последовательностей команд **OpenGL** перед исполнением вызванных ранее функций визуализации **GPI**, используйте **pglWaitPM()**. В этом случае гарантируется, что все вызванные функции **GPI** будут выполнены до того, как начнут исполняться команды визуализации **OpenGL**, вызванные после **pglWaitPM()**.

```
Замечание: Визуализации OpenGL и GPI могут интегрироваться, только если
контекст является опосредованным.
```

Переключение буферов

Для окон с двойной буферизацией передний и задний буферы можно поменять местами с помощь функции **pglSwapBuffers(). glFlush()** выполняется как часть этой функции.

Использование цветовой палитры

Когда вы работаете в **8**-битном режиме **(**с **256** цветами**),** вы должны позаботиться об управлении цветовой палитрой. Для окон с индексной визуальной конфигурацией вызовите **pglSelectColorIndexPalette(),** чтобы сообщить **OpenGL** о том, какую палитру вы хотите использовать с вашим контекстом. Цветовая палитра должна быть выбрана до того, как контекст будет изначально связан с окном. В **RGBA** режиме **OpenGL** настраивает палитру автоматически.

Использование логического шрифта OS/2

Простой путь использования шрифтов **OS/2** предоставляется с помощью функции **pglUseFont().** Эта функция строит списки отображения для каждого запрошенного шрифта и его размера, каждый из которых вызывает **glBitmap().**

Прототипы PGL

Инициализация

Определить, поддерживается ли **OpenGL** и, если да, то какой версии:

long pglQueryCapability (HAB *hab*);

void pglQueryVersion (HAB *hab*, int **major*, int **minor*);

Выбор визуальной конфигурации:

PVISUALCONFIG pglChooseConfig (HAB *hab*, int **attribList*);

PVISUALCONFIG pglQueryConfigs (HAB *hab*);

Управление конфигурацией

Управление контекстами и их опрос:

HGC pglCreateContext (HAB *hab*, PVISUALCONFIG *pVisualConfig*, HGC *shareList*, bool *isDirect*);

bool pglDestroyContext (HAB *hab*, HGC *hgc*);

bool pglMakeCurrent (HAB *hab*, HGC *hgc*, HWND *hwnd*);

long pglIsDirect (HAB *hab*, HGC *hgc*);

HGL pglGetCurrentContext (HAB *hab*);

HWND pglGetCurrentWindow (HAB *hab*);

Доступ и освобождение битовой карты переднего буфера:

bool pglGrabFrontBitmap (HAB *hab*, HPS **hps*, HBITMAP **phbitmap*);

bool pglReleaseFrontBitmap (HAB *hab*);

Синхронизация исполнения:

HPS pglWaitGL (HAB *hab*);

void pglWaitPM (HAB *hab*);

Поменять местами передний и задний буферы:

void pglSwapBuffers (HAB *hab*, HWND *hwnd*);

Выбрать палитру:

void pglSelectColorIndexPalette (HAB *hab*, HPAL *hpal*, HGC *hgc*);

Использовать логический шрифт **OS/2:**

bool pglUseFont (HAB *hab*, HPS *hps*, FATTRS **fontAttribs*, long *logicalID*, int *first*, int *count*, int *listBase*);

WGL: Расширения OpenGL для Microsoft Windows 95/98/NT

Визуализация **OpenGL** поддерживается на системах с **Microsoft Windows 95, 98** и **NT.** Для инициализации формата пикселей, управления визуализацией и получения доступа к расширениям **OpenGL** необходимы функции библиотеки **Win32.** Для полной поддержки **OpenGL** был добавлен ряд функций с префиксом **wgl.**

Ключевой структурой данных для управления форматом пикселей окна **OpenGL** в системе **Win32/WGL** является **PIXELFORMATDESCRIPTOR.** Переменная типа **PIXELFORMATDESCRIPTOR** хранит информацию о формате пикселей, включая тип пикселей **(**индексные или **RGBA),** тип буферизации **(**однократная или двойная**),** разрешение цветов, а также наличие буферов глубины, трафарета и аккумуляции.

Для получения более подробной информации о **WGL** вам следует начать с технических статей, доступных через веб – сайт **Microsoft Developer Network.**

Инициализация

Для получения информации о версии используйте функцию **GetVersion()** или более новую **GetVersionEx(). ChoosePixelFormat()** пытается найти **PIXELFORMATDESCRIPTOR** с заданными атрибутами. Если совпадение найдено, следует вызвать **SetPixelFormat()** для перехода к использованию этого формата пикселей. Вы должны выбрать формат пикселей в контексте устройства до вызова **wglCreateContext().**

Если вы хотите получить подробную информацию о полученном формате пикселей, используйте **DecribePixelFormat()** или, для оверлеев, **wglDescribeLayerPlane().**

Управление визуализацией

Несколько функций **WGL** предоставлены для создания и управления контекстом **OpenGL,** визуализации на битовую карту, переключения буферов, установки цветовой палитры и использования растровых или векторных шрифтов.

Управление контекстом визуализации OpenGL

Для создания контекста **OpenGL**, рисующего на устройстве с выбранным в контекст устройства нужным форматом пикселей применяется функция **wglCreateContext()**. (Для создания контекста визуализации для окон – оверлеев используется **wglCreateLayerContext().**) Чтобы сделать контекст текущим, используйте **wglMakeCurrent()**; **wglGetCurrentContext()** возвращает текущий контекст. Вы также можете получить текущий контекст устройства с помощью **wglGetCurrentDC()**. Вы можете копировать некоторые переменные состояния **OpenGL** из контекста в контекст с помощью **wglCopyContext()** или разделять списки отображения или текстурные объекты с помощью **wglShareLists()**. Когда вы закончите работы с контекстом визуализации, вы можете уничтожить его функцией **wglDestroyContext()**.

Доступ к расширениям OpenGL

Для доступа к специфичным для реализации функциям расширений **OpenGL** используйте функцию **wglGetProcAddress()**. Для определения того, какие расширения поддерживаются реализацией **OpenGL**, используйте **glGetString(GL_EXTENSIONS)**. Если передать функции **wglGetProcAddress()** имя функции расширения (например **«glMinmax»** или **«glConvolution2D»**) она возвратит фактический указатель на эту функцию внутри библиотеки, если таковая функция там есть или **NULL**, если расширение не поддерживается.

Визуализация на битовой карте

Win32 включает несколько функций для выделения и освобождения памяти под битовые карты, на которые визуализация **OpenGL** может производиться непосредственно. **CreateDIBBitmap()** создает зависимую от устройства битовую карту **(Device Dependent Bitmap – DDB)** из независимой от устройства битовой карты **(Device Independent Bitmap – DIB). CreateDibSection()** создает независимую от устройства битовую карту, на которую приложение может писать непосредственно. Когда вы закончите работу с картой, вы можете освободить память с помощью **DeleteObject()**.

Синхронизация исполнения

Если вы хотите комбинировать визуализацию **GDI** и **OpenGL**, имейте в виду, что в **Win32** не функций аналогичных **glXWaitGL()**, **glXWaitX()** или **pglWaitGL()**. Хотя **glXWaitGL()** отсутствует в **Win32**, вы можете добиться того же эффекта с помощью **glFinish()**, которая заставляет приложение ждать, когда исполнятся все ожидающие команды **OpenGL** или **GdiFlush()**, которая заставляет приложение ждать, когда завершаться все ожидающие функции **GDI**.

Переключение буферов

Для окон с двойной буферизацией передний и задний буферы могут быть переключены с помощью **SwapBuffers()** или **wglSwapLayerBuffers()** последняя используется для оверлеев.

Управление цветовой палитрой

Для доступа к цветовой палитре стандартных (не оверлейных) битовых поверхностей используйте стандартные функции **GDI** для установки вхождений в палитру. Для оверлеев используйте **wglRealizeLayerPalette()**, которая отображает вхождения их палитры индексного слоя на физическую палитру или инициализирует палитру **RGBA** слоя. **wglGetLayerPaletteEntries()** и **wglSetLayerPaletteEntries()** используются для установки и получения вхождений в цветовую палитру слоев.

Использование растрового или векторного шрифта

В **WGL** есть две функции **wglUseFontBitmaps()** и **wglUseFontOutlines(),** для конвертирования системных шрифтов в формы пригодные для использования в **OpenGL.** Обе функции строят списки отображения для каждого символа запрошенного шрифта и его размера**.**

Прототипы WGL

Инициализация

Получить информацию о версии**:**

BOOL GetVersion (LPOSVERSIONINFO *lpVersionInformation*);

BOOL GetVersionEx (LPOSVERSIONINFO *lpVersionInformation*);

Выбор формата пикселей**:**

int ChoosePixelFormat (HDC *hdc*, CONST PIXELFORMATDESCRIPTOR **ppfd*);

BOOL SetPixelFormat (HDC *hdc*, CONST PIXELFORMATDESCRIPTOR **ppfd*);

int DescribePixelFormat (HDC *hdc*, int *iPixelFormat*, UINT *nBytes*, LPPIXELFORMATDESCRIPTOR *ppfd*);

BOOL wglDescribeLayerPlane (HDC *hdc*, int *iPixelFormat*, int *iLayerPlane*, UINT *nBytes*, LPLAYERPLANEDESCRIPTOR *plpd*);

Управление визуализацией

Создание и опрос контекстов устройства **OpenGL:**

HGLRC wglCreateContext (HDC hdc);

HGLRC wglCreateLayerContext (HDC hdc, int iLayerPlane);

BOOL wglShareLists (HGLRC hglrc1, HGLRC hglrc2);

BOOL wglDeleteContext (HGLRC hglrc);

BOOL wglCopyContext (HGLRC hglrcSrc, HGLRC hglrcDst, UINT mask);

BOOL wglMakeCurrent (HDC hdc, HGLRC hglrc);

HGLRC wglGetCurrentContext (void);

HDC wlgGetCurrentDC (void);

Доступ к функциям расширений**:**

PROC wglGetProcAddress (LPCSTR *lpszProc*);

Создание битовых карт**:**

HBITMAP CreateDIBitmap (HDC *hdc*, CONST BITMAPINFOHEADER **lpbmih*, DWORD *fdwInit*, CONST VOID **lpbInit*, CONST BITMAPINFO **lpbmi*, UINT *fuUsage*);

HBITMAP CreateDIBSection(HDC *hdc*, CONST BITMAPINFO **pbmi*, UINT *iUsage*, VOID ***ppvBits*, HANDLE *hSection*, DWORD *dwOffset*);

BOOL DeleteObject (HGDIOBJ *hObject*);

Поменять местами передний и задний буферы:

BOOL SwapBuffers (HDC *hdc*);

BOOL wglSwapLayerBuffers (HDC *hdc*, UINT *fuPlanes*);

Управление цветовой палитрой для оверлеев:

int wglGetLayerPaletteEntries (HDC *hdc*, int *iLayerPlane*, int *iStart*, int *cEntries*, CONST COLORREF **pcr*);

int wglSetLayerPaletteEntries (HDC *hdc*, int *iLayerPlane*, int *iStart*, int *cEntries*, CONST COLORREF **pcr*);

BOOL wglRealizeLayerPalette(HDC *hdc*, int *iLayerPlane*, BOOL *bRealize*);

Использование шрифтов:

BOOL wglUseFontBitmaps (HDC *hdc*, DWORD *first*, DWORD *count*, DWORD *listBase*);

BOOL wglUseFontOutlines(HDC *hdc*, DWORD *first*, DWORD *count*, DWORD *listBase*, FLOAT *deviation*, FLOAT *extrusion*, int *format*, LPGLYPHMETRICSFLOAT *lpgmf*);