Detecting Shadows from Moving Cameras for Mobile Robots

Final Report for CS39440 Major Project

Author: Owain Daniel Jones (odj@aber.ac.uk)
Supervisor: Dr. Hannah Dee (hmd1@aber.ac.uk)

May 6, 2014
Version: 1.0 (Release)

This report was submitted as partial fulfilment of a BSc degree in Artificial Intelligence & Robotics (GH7P)

Department of Computer Science
Aberystwyth University
Aberystwyth
Ceredigion
SY23 3DB
Wales, UK

Acknowledgements

A huge thanks to my supervisor Hannah, who suggested this really cool project in the first place, and who has nudged me in the right directions every week and kept me calm during my many periods of panic. Thanks to Hannah also for patiently supervising me for hours on weekends whilst I scratch my head at nonfunctional robots.

Thanks to fellow student and friend Christina, for helping me capture data using the Kondo robot she revived for her own project, and for having some good suggestions during my project.

Also thanks to fellow student Sam Nicholls who had some great advice about the machine learning sections of my project. Most of it went over my head, but what I did understand was very helpful.

Thanks to my parents for proof-reading this report. Especially my mum, who found me plenty of New Scientist articles for my background reading, and helped me track down copies of papers when federated access failed to.

Thanks Courtney, for the source of amusement and fluffy animals whenever I was getting frustrated with my code.

Author's Note

Throughout this report, URLs of interest have been included as footnotes. These URLs were all correct and accessible at the time that this report was finalized (Tuesday, the 6th May, 2014).

Abstract

The aim of this computer vision research project was to experiment with different methods of detecting shadows in moving images, where the camera itself was in motion – typical of many mobile robotics applications.

Shadow detection for static cameras (which are assumed to not change their viewpoint) has been studied thoroughly. However, there has yet to be a methodology that can robustly handle the rapidly changing scenery, low resolution and high noise levels that are typical of consumer-grade webcams.

Videos of scenes with simple shapes and well-defined shadows were captured using webcams attached to two small mobile robots. Using these videos, experiments were carried out with a number of existing shadow detection methods, proven to work well with static scenes, to determine their performance at correctly detecting shadows for a camera in motion.

It was found that these methods performed poorly on the videos captured in this project, due to the nature of the input data as well as assumptions they make about the physical properties of shadows, which proved not to be true for the indoors scenery used in this project's video captures.

Edge features were explored, with the hypothesis that shadow edges will have different characteristics to object edges and can therefore be classified as such, given a well-performing learning algorithm and good training data.

Different methods of reducing noise within the input images were found to have a positive effect on edge detection; creating longer and more connected contours that closer resemble actual physical boundaries in the image, however the learning algorithms tested performed badly.

The output of the detection methods was compared against "ground truth" images: These were used to evaluate their performance via True Positive and False Positive rates.

CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

Background & Objectives

1.1 The Importance of Shadows in Human Nature

Shadows often go unnoticed in our day-to-day lives, left to our subconscious brain to deal with. They provide us with a wealth of spatial information which we use without even knowing. Shadows can be used to determine the shape, size, order and distance of objects in a scene – even when such objects are hidden from view. Shadows in motion are particularly useful, as they can be used to quickly estimate the trajectory and position of moving objects without requiring stereoscopic vision.

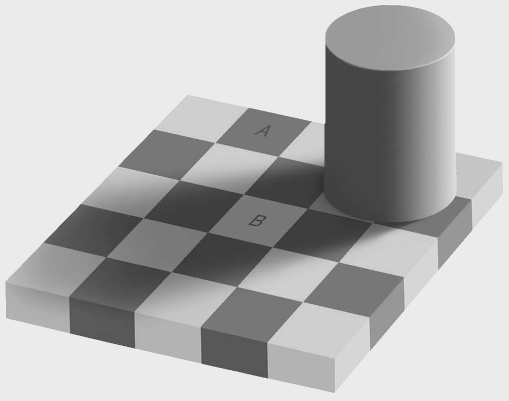


Figure 1.1: Adelson's Checker Shadow Illusion. Notice how squares A and B are the same shade, but we perceive B as lighter implicitly due to the shadow being cast around it. Further examples and explanation of this phenomena can be found in the paper "Lightness Perception and Lightness Illusions" [1] [2].

To best demonstrate the importance of shadows within the human visual system, a number of people have created optical illusions which make use of shadows and shading to trick the observer. A good example of this is Adelson's "Checker Shadow Illusion" (figure 1.1). Adelson's illusion includes two regions which are exactly the same shade, but usually the one in shadow is perceived as darker. Examples of shadow motion being used to infer object motion can be seen in the work by Kersten et al. [4], who showed that shadows are a very strong source of spatial information in the human visual system, often overriding other stimuli, even when the shadows themselves do not make sense.

It is interesting to note that the shape of a shadow does not have to follow the shape of the casting object; the use of shadows to convey depth in paintings is a good example of this. "The Birth of the Virgin" by Fra Carnevale (figure A.1) has a number of shadows that do not follow convention, yet it is not immediately obvious. The shadows being cast upwards in this painting would suggest the scene was lit from beneath, despite any sources of light doing so. They still help to create depth in the scene, and so their surreal nature goes unnoticed.

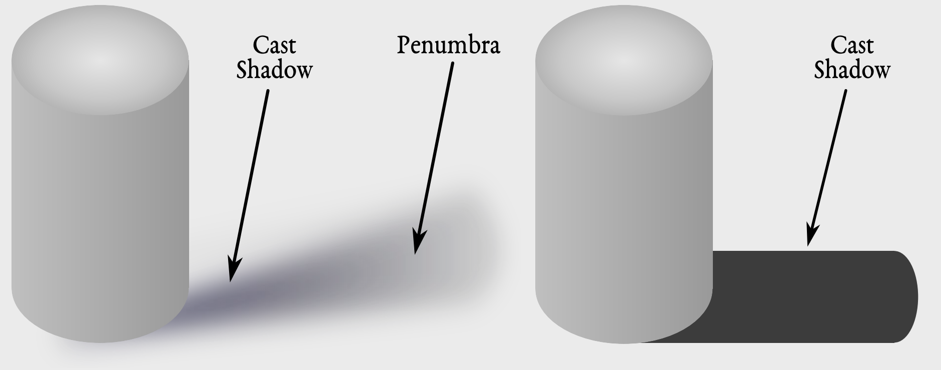


Figure 1.2: Cast shadows demonstrating different characteristics. The left is a very soft shadow with a lot of penumbra and which falls off gradually. The shadow on the right is a lot simpler, with no penumbra and hard edges. The former occurs when an object is lit from a low angle, the latter occurs when lit from above (such as in direct sunlight)

One of the first groups to notice this independence of shadow shape and perceived depth were Cavanagh & Leclerc, in their 1989 paper [5]:

> "In the stimuli that we have studied, the only requirements for the perception of depth due to shadows were that shadow regions be darker than the surrounding, nonshadow regions and that there be consistent contrast polarity along the shadow border. Three-dimensional shape due to shadows was perceived when shadow areas were filled with colours or textures that could not occur in natural scenes, when shadow and nonshadow regions had textures that moved in different directions, or when they were presented on different depth planes."

A formal description of shadows can be found in "The perception of cast shadows" by Mamaissan et al. [6] They describe five different areas of shadow: *Shading, Attached Shadow, Inter-reflection, Cast shadow and Penumbra*. These different types create different cues in the human visual system: Shading is used to determine object texture, and attached shadow gives cues about an objects shape. Cast shadows are the most important for depth perception. The presence of penumbra around a cast shadow gives an indication of the distance between shadow caster and light source: The more penumbra there is, the further away the light source is.

1.2 The Importance of Shadows in Computer Vision

Shadows have classically been treated as noise in vision tasks, as they interfere with object detection, creating False Positives and reducing the accuracy of detectors. A recent survey of shadow detection techniques by Sanin, Sanderson & Lovell (2013) [3] shows that object detection does improve when shadows can be detected and accounted for (figure 1.3).

Few have investigated the potential of shadows as *features* of vision, rather than noise. Recent studies by Dee, Santos et al. looked at how shadows could be used qualitatively, to allow a robot to reason about its environment [7]. They suggest that, given a good enough detection method, shadows could be used to infer the order of objects in scenes, and shape and positional information. They could be used to perceive depth in a scene, as an alternative or complementary method to stereopsis – which is more computationally expensive than shadow detection, and relies on highly textured surfaces to infer depth using multiple cameras, whereas shadow can be detected on smooth and reflective surfaces.

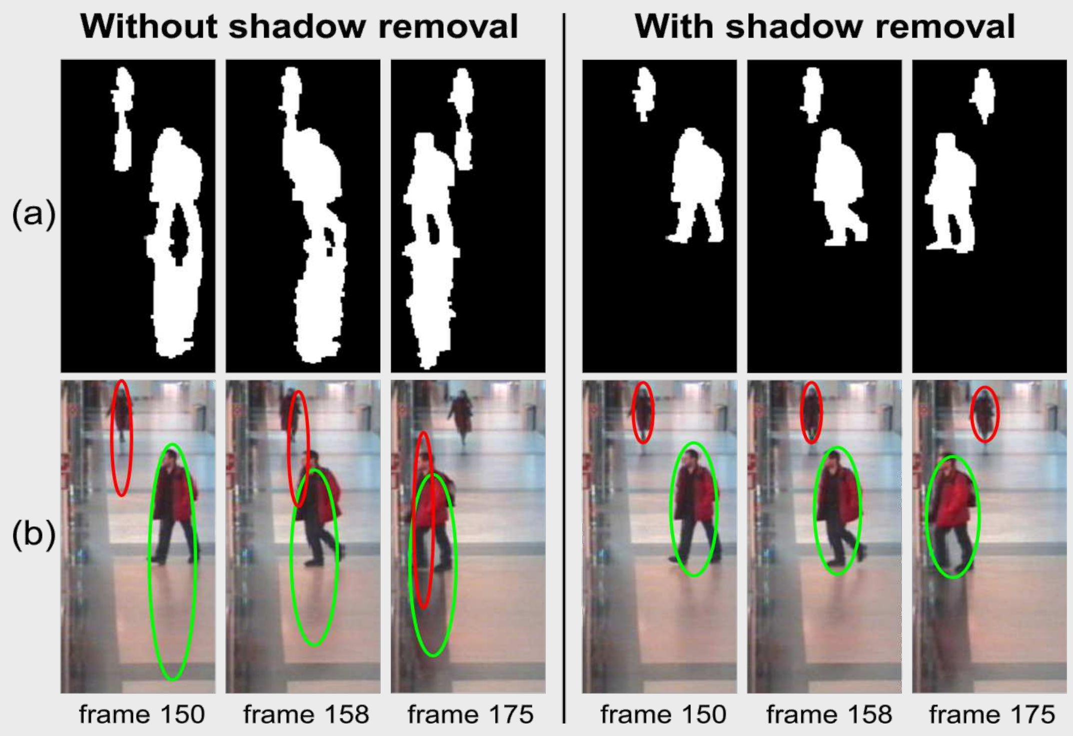


Figure 1.3: Figure from the Sanin, Sanderson & Lovell paper [3] which demonstrates how the accuracy of an object detector improves when shadows can be accounted for.

Dee & Santos used awareness of shadows to allow a robot to self-localise itself in its environment. They used shadows to estimate thresholds for image segmentation, to segment objects from the scene background. The size and position of these objects and the angles of their shadows, from the robot's viewpoint, were then used to correct odometry errors and allow the robot to more confidently know where it was in its environment. Compared to simple fixed thresholding and Otsu's method (which are explained in the following sections), their knowledge-based approach had a marked increase in accuracy.

1.3 Computer Vision principles

This project makes use of some common computer vision algorithms. This section gives a very brief introduction to some of the core concepts used.

1.3.1 Colour Spaces

Colour Space is a mathematical concept referring to the representation of colour through a distribution of numbers.

Used in this project are the RGB (Red, Green, Blue) colour space, and several HSL (Hue, Saturation, Lightness) colour spaces. The RGB space provides a simple mapping of a colour to three values (three channels). In the 8-bit RGB colour space commonly used on desktop computers, bright red is represented as $(255, 0, 0)$ and bright blue is represented as $(0, 0, 255)$. Shades of grey range from $(0, 0, 0)$ to $(255, 255, 255)$. RGB is a simple representation, but has some shortcomings, such as having no way of separating chroma (the colour) from luminance (lightness).

When dealing with shadows, it is important to consider the luminance of a colour separately from its chroma – which can be done using the HSL colour spaces [8]. In order to use them in the project, colours must be converted from RGB, which means colours represented in HSL spaces will still inherit some of the shortcomings of the RGB space [9]. However, from a conceptual standpoint they make it easier to extract brightness information from colours. The simplest HSL space is the HSV (Hue, Saturation, Value) space. Other HSL colour spaces include YUV, XYZ and L*UV.

Greyscale versions of RGB images are generated by converting the RGB image to a colour space which has a lightness/luminance channel – greyscale images are usually one or more components of a colour combined together. In OpenCV [1], the equation to get a greyscale channel (Y) from an RGB image is:

$$Y \leftarrow 0.299 \cdot R + 0.587 \cdot G + 0.114 \cdot B$$

Which seems to imply that the green component will affect the greyscale image more than red and blue – the default OpenCV implementation is biased towards the green channel.

[1] `http://docs.opencv.org/modules/imgproc/doc/miscellaneous_transformations.html#cvtcolor`

1.3.2 Thresholding

When referring to “thresholding an image” in this report, thresholding means clamping the colour values of pixels in an image such that all values are above or below the given value (threshold). Thresholding can provide a simple way of segmenting an image (dividing an image up into a small group of discrete colours, or ‘segments’), to find blobs that are certain colours – a very simple and fast way of tracking coloured objects – for example [2].

The simplest way of thresholding is using a fixed value: Pixel intensities that are below or above a given threshold value are replaced with another given value – for instance, for detecting shadows:

```
if intensity < 25:
    intensity = 0
else:
    intensity = 255
```

In the above example, pixels which have an intensity of less than 25 (very dark) are thresholded to 0, otherwise they are set to 255. This creates a *binary mask* of the input image.

Fixed-value thresholding fails when dealing with a sequence of images, or even a single image, which have a lot of noise or drastic changes in lighting. It is difficult to manually find threshold values which work well across different images. Automatically finding the best threshold is something which has been studied exhaustively, as shown in a 2004 survey by Sezgin & Sankur [10].

Some methods of automatic thresholding are quite complex. Apart from fixed value thresholding, this project also makes use of a relatively simple histogram-based approach called ‘Otsu thresholding‘, named after its author [11]. Otsu thresholding finds the best threshold for an image based on its histogram – it finds the two highest peaks in an image histogram and sets the threshold to be in the centre of them, such that the threshold maximizes the margin between the peaks.

1.3.3 Edge Detection

Edges in image processing are regions of pixels which have a change in colour/intensity/value. The greater the change, the steeper the gradient and the more defined the edge becomes. Strong edges in images can indicate the boundaries of objects – or in this project, the boundaries of shadows. As edges are usually based on gradients within an image, there will be a lot of edges wherever there are highly-textured surfaces or patterns. Often these are noise – *weak edges* – and should be filtered out somehow, leaving just *strong edges*. This project makes use of the Canny Edge Detector [12] to detect potential shadow and object edges in images. The Canny detector mitigates noise in images by using two thresholds; a lower threshold and an upper

[2] `http://www.aishack.in/2010/07/tracking-colored-objects-in-opencv/`

threshold. The upper threshold is used to mark where an edge begins, and an edge only ends when the gradient dips below the lower threshold. With a good set of thresholds, the Canny detector performs well at finding natural edges in images containing noise.

1.3.4 Noise Removal

In natural imagery, such as that captured from digital cameras (like Webcams), noise is often an image. In this project, noise comes in two types; sensor noise (the film grain and colour spots often seen in digital photographs)[3] and artefacts created by lossy compression algorithms, such as JPEG compression.[4]

Noise reduction algorithms aim to reduce noise in natural images without also removing too much useful information.

Gaussian Blurring was tested as a fast and simple way of removing noise. By convolving images with a Gaussian kernel, noise is averaged out of the image. Applying a Gaussian blurring to an image before edge detection can help remove unwanted, noisy edges from the result, as demonstrated in the Laplacian of Gaussian approach. [5] The drawback of simply blurring an image to remove noise is that it also smooths out strong edges and removes details in the image – details which may be important.

Later on in the project, Bilateral Filtering was used to remove noise from images. Bilateral Filtering filters out noise by averaging pixels that are close to each other, both in colour value and in physical distance. This results in smooth surfaces, but with edges and strong textures retained [13] – which is useful for when the image needs to be put through an edge detector.

1.4 Machine Learning

Towards the end of the project, it became clearer that attempting to classify shadows based on their physical models may not be the best approach – even with formal definitions of how shadows appear and what causes them, they did not always follow these definitions. Therefore, a statistical approach may be better. The last experiment in the project attempted to teach several different machine learning algorithms by example. The algorithms would then be able to predict whether new examples were shadow or not. This section describes three supervised learning algorithms that were used in the project very briefly due to time constraints.

[3]`http://www.dpreview.com/glossary/digital-imaging/noise`
[4]`http://www.stat.columbia.edu/~jakulin/jpeg/artifacts.htm`
[5]`http://homepages.inf.ed.ac.uk/rbf/HIPR2/log.htm`

1.4.1 Features, Instances (or Examples) and Labels

A 'feature', in the context of machine learning, is a single, usually numeric, value. Typical features used in vision could be:

- The brightness of a pixel (in the range 0-255);
- The length of an edge (distance in pixels);
- The ratio of colour (between blue and green) of a pixel.

Features can also be known as 'heuristics'.

An instance is a collection of different features, which represents a single concept (such as an object in an image). In vision, an instance could be:

- An edge;
- A single pixel;
- A region of the image.

Labels describe what class of concept or object an instance belongs to. In this project, instances are labelled as *Background*, *Object*, *Penumbra* and *Shadow*.

Labelled instances can be used to teach a learning algorithm via example: Instances that share the same label should have a set of features common to them all – a pattern should emerge. A supervised machine learning algorithm will essentially generate hypotheses such as "instances where `feature X` is around `3.14` belong to the class `Foo`".

It has been difficult to find papers that give formal definitions of features/heuristics, instances and examples. A very good reference is the Russel & Norvig book, 'Artificial Intelligence: A Modern Approach'. [14]

1.4.2 Support Vector Machines

The best introduction to Support Vector Machines found for this project has been a paper by Burges (1998) [15]. The following brief explanation of SVMs makes significant reference to information given in this paper.

Support Vector Machines (SVMs) look for *linear separability* in data. Given a set of examples and labels, they attempt to fit a line of best fit which separates the examples based on their labels. The best-fitting line is the one which has the *maximum margin* between the differently-labeled examples. This margin is known as the *support vector*.

These work best for data which only has two labels, and which can easily be separated by a straight line – however, by using *different kernels*, such as a polynomial kernel, a SVM can be trained on non-linear data.

Support Vector Machines have been shown to perform well given some fairly complex tasks such as digit recognition. Unlike other learning algorithms, they do not suffer from "the curse of dimensionality". 'Dimensions' refers to features in this case – 'High-dimensional spaces' are essentially large sets of features.

However, the performance depends on the kernel chosen. It is difficult to estimate which kernel (ie. linear or polynomial) would be a good choice for the data it is given.

1.4.3 Neural Networks

An excellent description of Neural Networks can be found in the Russel & Norvig book [14].

Neural Networks are learning algorithms inspired by biology. They are a simplified mathematical model of the human brain. A Neural Network is composed of *neurons* – units which perform simple mathematical or logical operations on a single input. The neurons are connected in a network, and the connections can be weighted – similar to how neurons in a brain communicate via synapses. A number of input neurons accept single values – such as pixel intensities from a greyscale image – and feed them through a number of 'hidden layers' of neurons, which are connected to output neurons.

Given their ability to take a small number of simple inputs and deduce complex patterns from them, they have found much use as autonomous control systems, such as in self-driving robots. In 1995, researchers at CMU successfully trained a Neural Network to steer a van on public roads, using data from a low-resolution digital camera as input to the network [16], along with steering data. The network was taught by collecting images of the road and steering data as a human driver drove the vehicle. After training, the network could then steer the vehicle itself with reasonable accuracy.

Careful consideration must be taken of the design of a neural network – how many input neurons it has, how many hidden layers it has, what weights are given for each hidden layer – all affect its performance. There are a number of parameters which must be chosen carefully – which can be done via testing and cross-validation – but training can take a long time, compared to other algorithms, such as Random Forests, which are explained next.

1.4.4 Random Forests

Information on how Random Forests work was taken from the original Breiman paper [17]. This paper assumes the reader has prior expert knowledge on learning systems, especially decision trees, 'bagging' and other ensemble learning methods such as Adaboost. It is possible that I have misinterpreted how Random Forests work, but I will attempt to explain them as best as I can below.

Random Forests are an *ensemble learning* method – that is, they are a collection of different learning algorithms combined to get better performance than any of the methods used individually.

Random Forests use a collection of decision tree learners to create a "forest" of small decision trees, based on random selections of features and instances. Each tree generated has its error rate tested. Decision trees are created that use a small, random selection of features (2 or 3 features is quoted as empirically being a good number, according to the paper). The forest is grown until the combined error rate of all the trees converges.

The paper claims that Random Forests work well on data that is composed of many 'weak' features: Features that may not have strong patterns or correlations within them. It is also robust against outliers in the data. Compared to Neural Networks or SVMs, Random Forests (and decision trees in general) can be trained very quickly.

1.5 Statistical Evaluation

1.5.1 Ground Truthing

In the context of statistics, the 'ground truth' is the 'gold standard' data that can be used to compare an algorithm's performance against. Ground truth data is the expected, best possible output from any given image processing algorithm. Ground Truth can be generated automatically using a method that is proven to work well – this could be a vision algorithm proven to detect specific objects in a scene with 99% accuracy – or can be obtained manually. This project uses manually obtained ground truth data, where input images are taken and pixels are given different labels such as *Object* or *Shadow*.

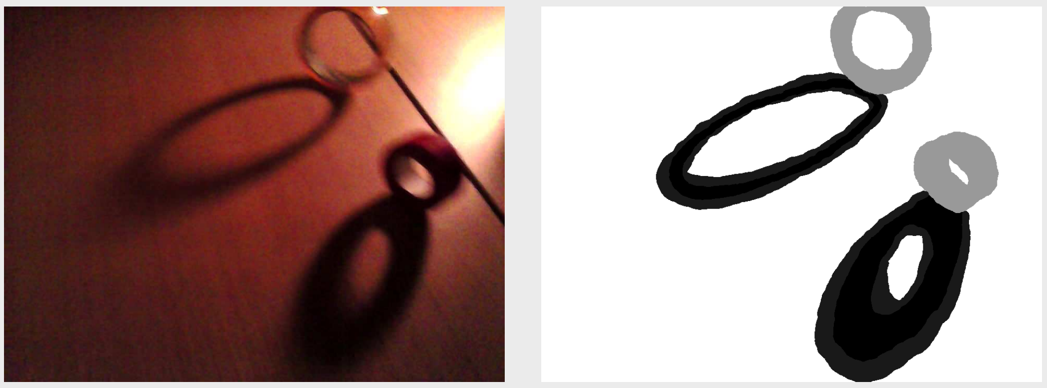


Figure 1.4: An example input image, and its corresponding ground truth image – which is also the expected output of a shadow detector (if it had the best possible accuracy). The pixel values in the ground truth image on the right correspond to specific classes – grey pixels are Object, black pixels are Shadow.

1.5.2 ROC Curves

‘Receiver Operating Characteristic‘ (ROC) Charts are the primary method by which different shadow detection algorithms are evaluated in this project. They provide a fast way of visualizing large amounts of performance data, to quickly see if an algorithm is performing well over large and varied sets of data.

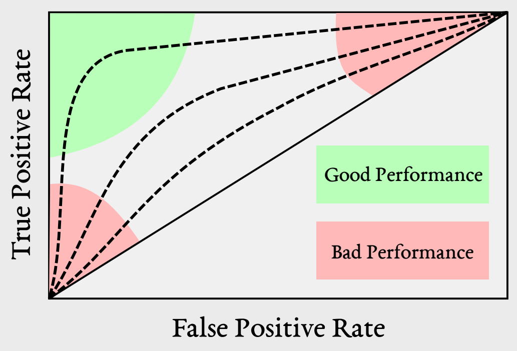


Figure 1.5: What an ROC chart looks like. The dashed lines denote the True Positive/False Positive rates of different things. Most things will not have points plotted beneath the diagonal line, as that would mean they perform worse than chance.

Figure 1.5 gives an example of an ROC Curve Chart. The solid line denotes chance – algorithms that perform the same as a random distribution would create data along this line. The bottom left corner is for algorithms which produce very few true or False Positives, meaning that their True/False Positive Rates are both low. This indicates that they generally tend to under-detect things in images. Anything in the top right corner of the graph indicates an algorithm is over-detecting the given class, as it has both a high True Positive rate and a high False Positive rate. Anything in the top left corner indicates an algorithm is performing well, as it has a high True Positive rate and a low False Positive rate. There should not be anything in the bottom right corner, as that would indicate that the True/False Positive/Negatives are being counted wrong.

The True Positive Rate (TPR, otherwise known as Sensitivity) is calculated as:

$$TPR = \frac{TruePositives}{(TruePositives + FalseNegatives)}$$

And the False Positive Rate (FPR, or Fall-Out):

$$FPR = \frac{FalsePositives}{(FalsePositives + TrueNegatives)}$$

More useful metrics which can be plotted on an ROC chart are documented in “An introduction to ROC analysis” by Fawcett (2006). [18]

1.6 Motivation

The method of detecting shadows in Dee & Santos’ experiments was very simple: thresholding on the Value channel of the Hue/Saturation/Value (HSV) colour space. They suggest in the “open issues” section of their paper that a more refined approach to detecting shadows may improve the performance of their qualitative spatial reasoning methodology.

The aim of this project was to implement some of the shadow detection algorithms listed in the Shadow Survey paper [3], and test them on a data-set similar to the one used in Dee & Santos' experiments, with the final outcome being an analysis of existing detection methods on this specific type of imagery (moving cameras, instead of static cameras). Whichever method performed the best over a varying dataset could be integrated into their PQRS framework.

This project was also an opportunity to learn C/C++ and how to use the OpenCV library, which is a very popular library for image processing.

1.7 Analysis

This is an exploratory research project, in which the overarching hypothesis is: *"Given good data and a suitable approach, shadows can be detected from cameras on moving platforms."* The overall plan for the project was to investigate as many things relating to shadow detection as was possible until time ran out.

Datasets exist for detecting shadows in static scenes (where the camera's viewpoint never changes)[6]. No prior datasets could be found for the specific problem of detecting shadows from a moving camera, so it was decided that the first objective of the project was to capture such a dataset, using mobile robots available in the university.

Evaluating the accuracy of shadow detection methods was to be done by comparing their outputs against ground truth images and calculating the True/Negative False/Positive rates – as ground truthing could be done by hand as soon as the input data was obtained.

This project focuses mostly on hard shadows, attached to objects. Existing work mentioned in the previous section has shown that these are easier to detect as they have well defined edges.

The detection methods listed in the Shadow Survey paper [3] have all been tested with static scenes, where the camera did not move. A large number of them used background subtraction to help segment shadows and objects in the scene. As this project focuses on images from constantly moving cameras, it was accepted that any implementation of the methods mentioned in the survey may perform very poorly, as subtracting background from moving camera images is a very difficult problem. Some work has been done on background subtraction for moving observers [19], but it relies on the objects in the scene being static. This problem was considered out of the scope of this project, due to its complexity, and time constraints.

The output of the detectors could be given in several ways – one would be a list of contours (lines with multiple segments), or bounding boxes (the rectangle that a shadow area fits into). A pixel-based representation was chosen, as it would be the easiest to visualize, as regions of shadow would map directly on to the input images. It would also be easiest to compare, as any detectors' output could be compared against the ground truth on a per-pixel basis.

The first hypothesis was that simple greyscale image thresholding would provide the baseline

[6]`http://arma.sourceforge.net/shadows/`

performance, and that most methods would perform better. Therefore the first detection method to be implemented would be simple greyscale thresholding, with the test harness and ground truth analysis programs being developed in parallel.

This project did not intend to implement any shadow detection methods in real-time, instead opting for an offline approach, where data was captured and processed retrospectively. However, it was decided that the average execution times of each detection method should be recorded, as it may have occurred that two methods performed with similar accuracy but differed in speed – and whichever was fastest would be useful to know for future implementations.

For data collection, we decided to create some simple scenes with one or two light sources and several "casters" – objects to cast shadow. We opted for controlled indoors office environments where we had full control over the lighting.

1.8 Planning

After initial background reading and analysis, it was decided that this project should consist of the following tasks:

- Gather video recorded from a mobile robot with a camera attached;
- Create ground-truth images for frames of the captured video, manually labelling image regions as "object", "shadow", "penumbra" or "background";
- Implement existing shadow detection methods, starting with the simplest to implement;
- Chart the performance of each method during and after implementation, to compare against the others;
- Iterate the above two tasks for the remainder of the project time, with the aim of finding an approach that has an acceptable level of accuracy when used on the collected data.

From these tasks, the following deliverables:

- An experimental setup for testing shadow detectors (the test harness), allowing others to repeat the tests;
- A set of videos recorded from one or more controlled environments using cameras on board mobile robots (we had several suitable robots available to us);
- A set of ground truth images for representative samples of some of the captured videos, where visual estimation has been used to label image regions, which is reasonably true to reality (areas labelled as shadow are shadows in the input image);
- Performance data for each of the detection methods tested.

At the beginning of the project, an initial Gantt Chart was sketched out (see appendix figure A.8) which the project would attempt to follow. The project was split into four stages:

1. **Data Collection**: Nothing could be tested without data to test it on, so data collection was the first priority.
2. **Development of the Test Harness**: Developing the test harness (and ground truth comparison) programs first meant that all the implementation code would be forced to use a standard set of input and output conventions.
3. **Implementation of detection methods**: The Shadow Survey paper categorizes different methods based on the types of image features they are looking at. Initially, investigating three features was planned, spending two weeks developing code for each. The three features were *Chomacity, Edges and Texture*. It was expected that this would entail some machine learning towards the end, when the best features were found for training a classifier.
4. **Final analysis**: Optimistically, by this point some methods would be implemented which clearly outperformed the others. The analysis would consist of comparing the algorithms through ROC charts and also comparing their execution speed.

1.9 Research Methodology

This is a very challenging computer vision problem with no clear solution. Because of this, no major up-front planning was done. Weekly meetings were held between my supervisor and I, in which results of the week were discussed and work for the following week planned.

I maintained a blog throughout the project[7], in which I aimed to post at least once a week, usually on a Sunday. This would force me to do some analysis and reflection of the results of that week. It also served as another channel of communication between my supervisor and I, as they would read it and have some suggestions for me at the next meeting.

This was very much an AI/Engineering project, focusing more on evaluating hypotheses than designing and implementing concrete software tools. The first stage of development was the test harness – with a small set of requirements sketched out, this was developed using rapid prototyping. Configuration files were written first, and the harness developed around these.

For the implementation of shadow detection methods, a test-driven methodology was chosen. Specific outputs were expected (the ground truth images), and each method had to reproduce these outputs as accurately as possible. Instead of having a suite of discrete "Pass/Fail" tests, a methods performance and accuracy would be tested using the test harness. Any bugs or programming errors would be obvious when programs were ran by the test harness. The output of each program was to be compared against ground truth images, and their "true positive rate" versus their "False Positive rate" plotted on a ROC chart.

[7] `http://shadows.odj.me`

1.10 Initial Design Decisions

I decided that I would implement my shadow detection programs in C++. All other code would be in Python, as I am comfortable with Python and already have working knowledge of a number of Python libraries. Whilst a lot slower than compiled languages like C++, Python is popular in scientific communities, as demonstrated through toolkits such as SciPy and numpy[8], the latter of which is used throughout the code of this project. Python also lends itself naturally to being a "shell scripting" language – as the test harness is intended to simply execute other programs on the system, it made sense to implement the test harness in Python. Using C++ code for the actual vision algorithms would make them a lot more lightweight and much faster than if they were written in Python, hence why I chose to implement that part specifically in C++.

1.10.1 Development Tools

Whilst an IDE may have helped me with the C++ side of my project, I struggled to find a free IDE for Linux in which I could write Python code as well as C++. Eclipse[9] has plugins for both Python and C++, but is bulky and combersome.

Being comfortable with Vim[10], I elected to write my code in Vim. Plugins exist to provide intelligent auto-completion[11] and syntax checking/highlighting of errors[12], which helped me write in a language, and using libraries that, I was not familiar with.

For source control management, I kept all my code in git repositories[13]. All my files were also backed up on Dropbox[14].

1.10.2 Choice of libraries (C++)

As this project intended to teach me some C++, it was decided at the start that all shadow detection algorithms should be implemented in C++ using the OpenCV library.[15] Had I not restricted myself to C++, I could have used ImageJ (in Java)[16] or possibly the Python Imaging Library[17]. However, OpenCV provides the largest code-base, with implementations of many computer vision algorithms, as well as machine learning algorithms. It also has Python bindings, allowing me to also use it for the analysis part of my code (which is all written in

[8]`http://www.numpy.org`
[9]`https://www.eclipse.org`
[10]`http://www.vim.org`
[11]`https://github.com/Valloric/YouCompleteMe`
[12]`https://github.com/scrooloose/syntastic`
[13]`http://git-scm.com/`
[14]`http://www.dropbox.com`
[15]`http://www.opencv.org`
[16]`http://imagej.nih.gov/ij/`
[17]`http://www.pythonware.com/products/pil/`

Python). OpenCV also appears to have the most comprehensive documentation, with an abundance of tutorials demonstrating how to use different parts of the library.

The programs I implemented later in C++ may require a large number of command-line parameters. For this reason, they would require a library with robust command line argument parsing. I found the Boost[18] `Program_Options`[19] suitable for this. Being familiar with the `argparse` module in Python, it allowed me to consider command line arguments first before implementing the rest of the program – a form of rapid prototyping.

For compilation of the C++, I required a build system which would compile my code and link the OpenCV and Boost libraries to it correctly. As I have been working entirely in a Unix environment for my project, I could write this myself using GNU Makefiles[20] – but that could be time consuming, especially if wanting to compile within different Operating Systems (such as Windows or OSX). Another option was `automake`[21]. However, as OpenCV and Boost both make use of CMake, I chose CMake instead[22]. CMake is intended for cross-platform compilation. It can automatically find a number of libraries (including Boost and OpenCV) on the machine it is compiling on, and I was able to write a `CMakeFile` very quickly.

1.10.3 Configuration files

When developing the test harness, the configuration files were written first – inspired by previous students' personal projects[23] in which they wrote out how data would be stored in their system represented through JSON (JavaScript Object Notation), before implementing the code to generate or use such data.

The configuration files should be self-documenting and self-explanatory; both through simple syntax and ability to leave comments in the files.

JSON was my first choice, as there are a huge number of JSON libraries for most popular programming languages.[24] However, JSON has some setbacks, such as lack of proper commenting.

YaML (YAML Ain't Markup Language) was discovered, after some searching. YaML also has libraries for many popular languages[25]. YaML is described as "human friendly data serialization", which made it a good fit for my project.

[18]`http://www.boost.org`
[19]`http://www.boost.org/doc/libs/1_55_0/doc/html/program_options.html`
[20]`http://www.gnu.org/software/make/manual/make.html`
[21]`http://www.gnu.org/software/automake/`
[22]`http://www.cmake.org/`
[23]`http://www.pieratnine.com/shut-up-and-jam`
[24]`http://www.json.org/`
[25]`http://www.yaml.org/`

1.10.4 Requirements for Test Harness

As the test harness was going to be used to test the shadow detector implementations, some thought about its design was needed before the first prototype was written.

The test harnesses job was to, given a list of possible parameters for a program or group of programs, run these programs with all possible combination of the given parameters. It was expected that this may generate a lot of data.

It was decided that the test harness should run different processing 'chains': Groups of programs, executed one after the other, which would work on images copied to a temporary directory, with the ground truth program being ran last.

The best way to test every possible combination of program parameters in a chain was to build a tree of the parameters. Traversing the tree depth-first until it terminated (reached a leaf node) would result in one full list of parameters for the programs in a chain. Doing this whilst also moving over the tree breadth-first would result in a list of lists of all parameter combinations.

The following functional requirements were outlined:

- Read all configuration from human-readable file – make configuration as simple to read as possible, by having default paths/program arguments, variable expansion and shell globbing [26];
- Accept command line arguments, for selecting which chains to run, which image sets to test on, as well as 'verbose' and 'test' modes for programs (to help aid debugging);
- Create a directory tree structure in which all possible combinations of 'chain' program parameters (specified in configuration files) are represented;
- (Optional) Make programs run asynchronously (fork to background) and run in parallel – but kept in a process pool limited by number of CPUs available.

[26] `http://www.linuxjournal.com/content/bash-extended-globbing`

1.10.5 Requirements for Shadow Detection Methods

If the detection methods were to be tested in a test harness, it made sense to have a set of command line arguments and inputs/outputs that would be common to all of the methods. The requirements for these programs was also outlined:

- Accept any image format accepted by OpenCV's `imread` [27] as input (given via the command line parameter `inputs`);
- Accept a list of input files and operate on them all;
- Output of the methods must be in greyscale PNG format. Given the command-line parameter `output_dir`, output files must have the same name as the inputs, except with the "`.png`" extension, and stored within the `output_dir` directory;
- Output of the methods must use the same values for labels as the ground truth, so they can be compared;
- All parameters that are adjustable on the command line must follow the double-dash (`--argument parameter`) format.

1.10.6 Format for the ground truth / output data

The best way of representing the ground truth was as greyscale, 8-bit, 1-channel images – with the same width and height as the input images. Pixel values correspond to different labels:

- Background: 0
- Penumbra: 25
- Object: 153
- Background: 255
- Unknown: 250

It was decided to save these images in the PNG format, as this provided good lossless compression of the data. Lossy formats (such as JPEG) would have introduced artefacts which would reduce the accuracy of the ground truth.

These standards were decided on at the start of the project. The different pixel shades provide a good visual representation of the scene, as shadows are dark and objects stand out against the white background.

[27] `http://docs.opencv.org/modules/highgui/doc/reading_and_writing_images_and_video.html?highlight=imread`

Chapter 2

Implementation

This chapter documents the actual implementation of code throughout the course of this project, including problems that were encountered. Actual results of the shadow detection methods implemented are discussed in the next chapter.

2.1 Data Collection

Two data-sets of natural imagery were captured during the first half of the project.

2.1.1 Pioneer datasets

2.1.1.1 Technical Details

The Pioneer P3-DX[1] robot is a small, two-wheeled mobile robotics platform. It has sonar range sensors and a small-factor X86 computer on-board running Linux. Its movements can be controlled, and its sensors read, via any program on it that uses the ARIA library.

This particular Pioneer had an analogue camera attached to it, connected to TV capture card. The resolution of the captured images was 768×512 pixels. The images were captured as 8-bit RGB colour images.

The processor on-board was too slow to save images in compressed formats such as PNG or JPEG – taking several seconds to capture and save each image. PPM format images were used, which are an uncompressed bitmap format. As such, they were very large; roughly 2.5MB each. The Pioneer had limited storage space, so care had to be taken over the length of the data captures. Even with PPM format images, the rate of data capture was roughly 1 image every half a second (2 frames per second). The robot was moved in very small increments to compensate for the low frame-rate, so a smooth movement is still seen when the images are played back at a higher frame-rate.

[1] `http://www.mobilerobots.com/ResearchRobots/PioneerP3DX.aspx`

Due to problems with degraded batteries and initially getting software to compile on the robot, there was limited time to actually capture any data with the Pioneer. Nonetheless, five different sets of images were captured. When each capture was finished, the robot was carried back to its starting position and left to charge whilst the images were copied via SFTP to a laptop with a bigger hard drive and then converted to maximum-quality JPEG (using the ImageMagick `convert` command[2]).

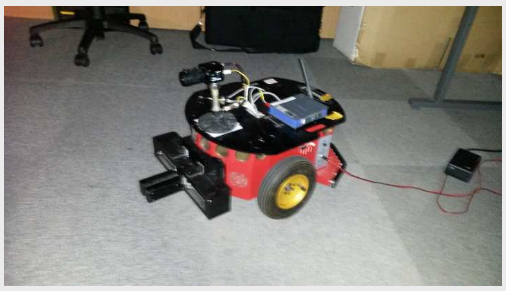


Figure 2.1: Pioneer Robot in the ISL

The range values reported by all the sonar sensors on the robot were recorded for each image frame captured. The robot's estimated position and heading per frame were also recorded – although these are only best-guesses by the robot.

1.7GB of PPM format data was recorded overall. Each image set contained between 186 and 289 images. After conversion to JPEG, the dataset was 100MB large. Further noise removal reduced the dataset to 75MB.

2.1.1.2 Environment

The Pioneer my supervisor and I had access to was in the Intelligent Systems Laboratory on Penglais Campus. This is a large room with lots of open floor space, allowing these Pioneers room to roam. It has the advantage of having only small windows with blinds, which allowed us to make the room completely dark and then light it using two high-power lamps. The rooms overhead lighting was divided into four sections which can be switched on and off independently.

[2]`http://www.imagemagick.org/script/convert.php`

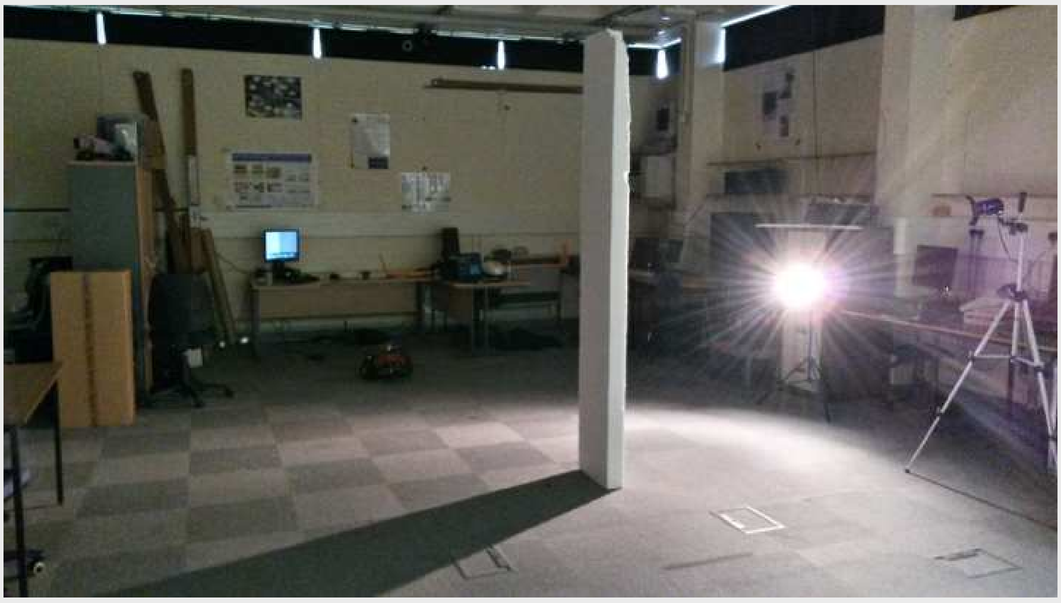


Figure 2.2: The Intelligent Systems Laboratory environment.

The lamps were at roughly desk-hight, and as such cast much longer shadows than the overhead lights would. As they were the only source of light in an enclosed environment and were relatively bright, the shadows they cast were hard shadows (little or no penumbra).

We chose to use one object as the "caster" in this scene: A tall white polystyrene block which was able to stand upright in the centre of the room. The robot took a circular path around the block, and then took a similar path in the opposite direction – to capture the cast shadows from the block at all possible angles.

Five different sets of images were captured:

1. One floodlight on, no overhead lights;
2. One floodlight on, background overhead lights on;
3. One floodlight on, all overhead lights on;
4. Both floodlights on, no overhead;
5. Both floodlights on, background overhead lights on.

("Background" referring to the section of the room which we were not using)

2.1.1.3 Quality of data

The images from this dataset ultimately proved too difficult to use. The images were very dark, making it difficult to discern between cast shadow and simply darkness – especially as shadows would fade into the distance. This made it very challenging to ground truth, and only 6 images from the first set of images were ground-truthed before it was decided it was an exercise in futility.

The images also suffered from multiple sources of noise – sensor noise from the camera, combined with the noise induced in the analogue signal from camera to TV capture card. The lens of the camera was also a cheap wide-angle lens which suffered from chromatic aberration around the edges, due to the distortion of the lens – creating some phantom edges in the image.

The background of the image was far more detailed than the foreground, and the cluttered edges of the environment cast complex shadows. This all proved too time-consuming to accurately ground-truth, and all shadows but those cast by the polystyrene block were ignored in the ground-truth – which means the ground truth itself was far from true.

The carpet in the lab also suffered from self-shadowing: The carpet tiles were all oriented at right-angles to one another, creating a checker-board pattern in the images, which was quite prominent under some of the lighting conditions.

The odometry and sonar data captured by the robot was also unreliable. The odometry was a best-guess estimate that was made increasingly inaccurate as the robot's wheels slipped or moved slower than normal. The sonar suffers from a lot of interference from the reflective metal surfaces which fill the lab (such as metal table legs). This telemetry data never found use in this project.

The following page has an example image from each set, from the robot's starting position, to demonstrate the effects of different lighting on the shadow casting object.

2.1.1.4 Post-Processing

As the images had a lot of grainy noise, especially in dark regions, some noise removal software was tested as a post-processing stage.

The software used was Neat Image[3], commercial software which claims to be able to remove a number of different noise types from natural images. As this is proprietary software, no documentation exists on what it does internally to remove noise, although it does seem to make use of 'camera profiles' – where a pattern of noise is detected over a sequence of images taken on the same camera, which can then be subtracted from the image. This 'camera profile' feature was not available to me at the time of testing as the shareware, free version of the software had restricted functionality.

[3]`http://www.neatimage.com/`

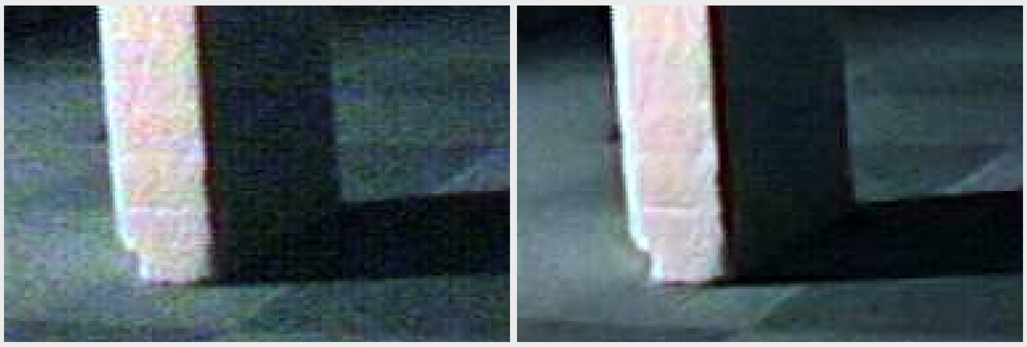


Figure 2.4: Original image (left) and image filtered through Neat Image (right).

The software did reduce the noise in the images, as shown in figure 2.4 – but it also sharpened edges, creating a 'halo' around highly contrasting areas, which could interfere with edge detection algorithms later. It also took a long time to process the individual images, and taxed the computer it was running on quite heavily. The noise reduction was promising, but I decided against using it in the future as it may be removing information which I couldn't see, especially in regions of shadow. It also wouldn't be realistic; any shadow detection algorithm running on a robot wouldn't have time to spend several seconds processing each frame to filter noise. I decided to focus on noise reduction implementations directly available in the OpenCV library, which were explained extensively in papers available for free online – unlike Neat Image, which is completely proprietary.

2.1.1.5 Concluding remarks

My analysis of the Pioneer data has been very pessimistic. The data may be a poor fit for this project, but the images could find use in other experiments. Especially when their resolution is reduced (which is a very cheap way of removing noise from images).

2.1.2 Kondo datasets

Several weeks later into the project, a fellow student let us borrow the small, bipedal robot they were using for their own dissertation. This was a Kondo KHR-3HV Humanoid Robot[4] which was much smaller than the Pioneer, and could be walked across table tops. This allowed us to create a simpler, more controlled scene for it to walk through, using desks against a wall in an empty office, with some simple household items as props and a single desk lamp for lighting.

[4]`http://www.robotshop.com/en/kondo-khr-3hv-humanoid-robot-kit.html`

2.1.2.1 Technical Details

Kondo was directly tethered to a laptop which used the `libkondo4` library from Python to send it movement commands. Kondo was able to execute a constant walk across the table whilst a webcam taped to its torso captured continuous video at around 15 frames per second average. The webcam was connected to the laptop via another USB cable.

The `guvcview`[5] software was used to configure the webcam and then capture video. White balancing and auto-exposure features were disabled on the webcam so that they would remain constant throughout the captured videos, even as lighting levels in the images changed.

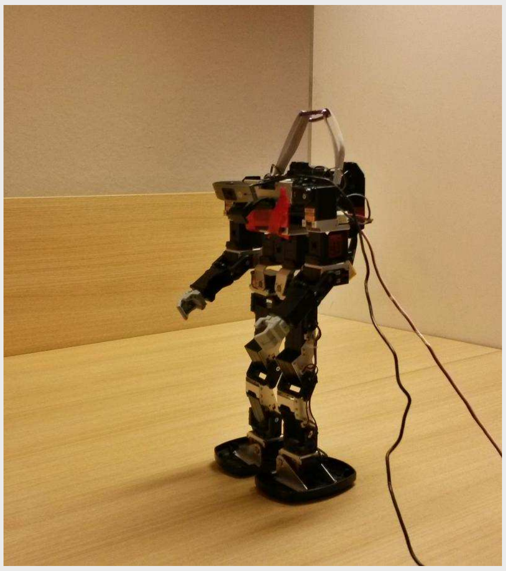


Figure 2.5: Kondo KHR-3HV robot.

`guvcview` captured the videos in the H.264 (MPEG4) video codec. The videos were at a 960×720 resolution; slightly higher than the image frames captured from the Pioneer camera. However, as these were image frames being sent over USB (therefore, already compressed) and converted to a lossless video format, compression artifacting was introduced to the captured data. The low light levels of the scene meant that the webcam's exposure had to be set quite high, which introduced a lot of sensor noise.

Three videos were captured – ranging from 45 seconds to 1 minute 30 seconds long. In total, 325MB of video data was captured.

2.1.2.2 Environment

A confined environment was created in the corner of an empty office – desks were joined together against the walls to create a surface over which Kondo could walk. The camera attached to Kondo was pointed slightly downwards, so that the rest of the room would never come into view, keeping the scene simple.

Some objects from around the room were placed in the scene – a fizzy drinks can, a whiteboard eraser and a remote control (we got creative with our very limited supply of props). A single desk-lamp was used to light the scene. This cast some strong shadows in the scene, but the shadows were considerably softer than those in the Pioneer imagery. This was not necessarily a bad thing, as now the penumbra of the shadows could be investigated also.

In each of the videos, Kondo walked through the scene past the various objects in a straight

[5]`http://guvcview.sourceforge.net/`

line. Considerable body movement of the robot was captured on the camera, creating an interesting effect – there are constant small changes in the camera viewpoint's pitch, roll and yaw. This could possibly be used as a way of inferring depth in the 2D images [20], which could help determine what is "foreground" and what is "background" – information which could be used by shadow detection methods.

2.1.2.3 Selection of video frames

As this dataset was in compressed video format, there was a need to extract frames from the videos in order to ground truth against them and use them in the detection methods (which, as mentioned earlier, took lists of images as inputs).

The fact that the video was in a format which used key-frames was exploited to extract a selection of frames that were reasonably different from each other.

To extract the frames as JPEG images, `ffmpeg`[6] was used. `ffmpeg` contains a very powerful command-line tool for interacting with video files. Using the `select` filter, 'I-frames' that were at least 3 seconds apart were extracted. 'I-frames' are the frames within a video stream which contain a full image – as opposed to 'B-' and 'P-frames' which contain only differences in motion.[7] This is the command that was used:

```
ffmpeg -v debug -i kondo1.mkv -vsync 0\
    -vf select='eq(pict_type\,PICT_TYPE_I)*(isnan(prev_selected_t)'\
               '+gte(t-prev_selected_t\,3))'\
    -f image2 '%4d.jpg' 2>&1 | fgrep 'select:1'
```

This was done for the first data-set only. 32 frames were extracted and subsequently ground truthed against. For this dataset, a ground truthing tool was developed – which is described later in this chapter.

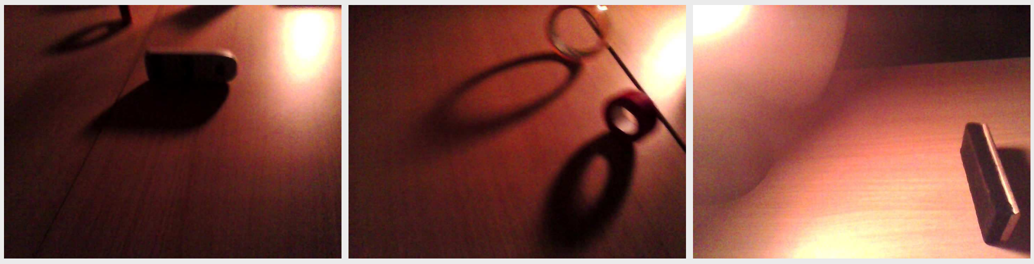


Figure 2.6: Example frames from the `kondo1` image set.

[6] `http://www.ffmpeg.org`

[7] MPEG Key-frames explained: `http://nickyguides.digital-digest.com/keyframes.htm`

2.1.2.4 Noise removal

For this dataset, Bilateral filtering was tested on the extracted images. This was implemented in C++ using OpenCV's `bilateralFilter` function. The pixel neighbourhood, colour sigma and space sigma were all set empirically at 33, which had a strong smoothing effect on the images. In this data set, textures are not particularly important, so this was acceptable. Strong object and shadow edges were still retained.

Bilateral Filtering, whilst taking a few seconds to iterate over , was considerably faster than the methods in use in the Neat Image software. It could potentially be used as part of an image processing pipeline running in near- real-time.

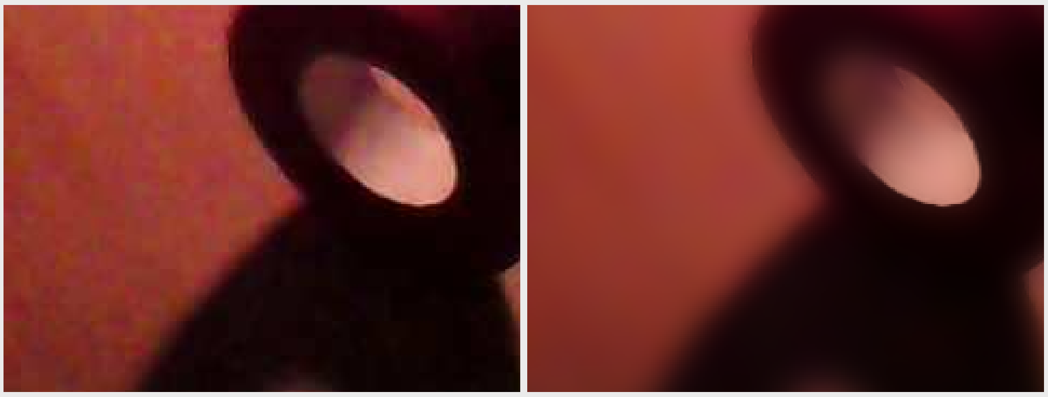


Figure 2.7: Results of Bilateral Filtering (right) on an image from the `kondo1` image set (left). Note how the noise is completely gone, but strong object edges have been retained.

2.1.2.5 Concluding remarks

This seems to be a better quality data-set when it comes to shadow detection. The shadows are much darker than the surface they are being cast on. The simple objects in the scene made it easier to draw ground-truths for the extracted frames. The whole scene has a strong red component in the RGB channel, due to the warm colour of the light emitted by the desk lamp. For testing shadow detection methods, the `kondo1` image set was used far more than the others.

2.1.3 Artificial datasets

Shortly after collecting the Kondo data, the thought occurred that some artificial data may be a good idea – to provide a source of 'perfect' data on which the algorithms could be tested and debugged first.

Using 3D modelling and raytracing software, it would be possible to create environments over which we had full control of lighting and shadowing, without the issue of noise, and which had perfect ground truthing for every image frame.

With this in consideration, two days were devoted to learning how to use Blender in order to create and render some artificial image sets. Blender[8] was chosen because it is Free, Open Source Software (similar software is mostly commercial with a price attached, such as 3DS Max[9]). Blender comes with a very capable ray-tracing renderer, which can render scenes with very complex lighting and shadowing, taking into account ambient occlusions, reflectivity, surface emissions and sub-scattering. However, all shadows could only be rendered as hard shadows with no penumbra.

Three different environments were created. Each one had the camera move around the scene and return to its original position. There were 250 frames in each, and the frames were saved as PNG images. All the images were rendered at a 960×720 resolution.

The first environment was very simple – three primitive shapes with solid black shadows and primary colours against a background, with a camera rotating around them. The second one was an intentionally dark scene, with volumetric lighting and some highly-contrasting textures. It also had shadows that faded into the black background. This was intentionally similar to the Pioneer data. The final one made use of coloured lighting and contained more objects and multiple lights.

To generate full sets of ground truths for each of the environments, nearly all the rendering options were disabled. Object materials were set to a uniform grey colour (`rgb(153, 153, 153)`) with lighting disabled. The sky/background was set to bright white. Anti-aliasing was switched off. This produced very accurate ground truth data. What couldn't be disabled was the gradual fall-off of shadows: So instead of being uniform black in the ground truth, some shadows would gradually fade out. The solution to this problem was simply to make all code which made use of the ground truth treat values that did not correspond to any of the labels (Background, Shadow, Object, ...) as either shadow or background, depending on what they were using it for.

2.2 Ground Truthing

Ground Truthing can be a time-consuming task: labelling complicated shapes accurately within dozens of images is not an easy feat. In this project, the ground truth images were obtained by opening an input image in graphics editing software, creating a new empty layer overlaid on the original image, and then manually painting the ground truth onto this layer using the paint tool with the mouse. The ground truth layer was then saved as a separate image. This was done for every image which needed to be ground truthed.

[8] `http://www.blender.org`

[9] `http://www.autodesk.com/products/autodesk-3ds-max/overview`

2.2.1 Comparison of different tools

2.2.1.1 Existing Tools

GIMP was the first tool to be used – for ground-truthing the Pioneer images. It is a very powerful graphics editing package, often considered the free and open-source alternative to Adobe's Photoshop. It has tools for selecting image regions with complex shapes (including some 'intelligent' edge fitting tools), which can then be filled in with a solid colour. It also has tools for editing images at the per-pixel level, useful for making small adjustments which could make a ground truth more accurate. The concept of "layers" in GIMP is useful as the ground truth could be kept as a semi-transparent layer above the original image, making it easier to follow the natural boundaries in the original.

Due to GIMP's complexity as a graphics package, it also has a complicated user interface which is mostly mouse-driven. Whilst some keyboard shortcuts exist, too much of my work-flow involved navigating through mouse-driven interfaces. Opening, editing and then saving each ground truth image was taking too much time (one of the reasons that only nine of the Pioneer images were ground truthed). Because of this, it was decided to investigate whether any software existed specifically for the task of labelling complex regions in images for the purposes of ground truthing.

Some tools do exist, such as GEDI[10] – GEDI was not tested as it can only output image annotations in XML format. It also seemed to only allow for simple image regions to be defined: rectangles and simple polygons.

ICY[11] also looked like promising software for this purpose. ICY is an image analysis tool aimed at biological imagery analysis. It contains tools for annotating images. However, it was found that the interface was very "clunky", with a steep learning curve – it took too long to learn, so was not used.

There seems to be a conceptual difference between what is considered 'ground truth' in this project, and what is used in others. This project uses ground truth where images are labelled at the *per-pixel* level, but in vision projects such as LabelMe[12], images are commonly annotated at a higher level: usually bounding boxes (rectangles or polygons) defined around regions of interest.

2.2.2 Implementation

Having had experience with writing simple painting programs before, it was decided that a tool ground-truth painting tool should be written specifically for this project. One day was spent on this. Python and the Pygame framework were used, as they allowed for a working

[10] `http://gedigroundtruth.sourceforge.net/`
[11] `http://icy.bioimageanalysis.org/`
[12] `http://labelme.csail.mit.edu/Release3.0/`

prototype to be developed very quickly. Pygame[13] contains libraries for loading and displaying images, drawing primitive shapes, and input (keyboard and mouse) handling.

The first prototype was written to implement the following functionality:

- Accept a list of input images via the command-line only: No time-consuming "Open File" graphical dialogues;
- Display the input image as a 'display-only' layer beneath the ground truth layer;
- Have a single tool for painting, which paints using adjustable-radius circles;
- Switch between different label classes (Shadow, Object, Penumbra) using a mouse button, erase via holding right-click;
- Save the ground-truth layer for the currently displayed image to another directory as greyscale PNG by pressing the S key;
- Be able to cycle through these images with the keyboard arrow keys (and re-load existing ground truth image if it exists for an image).

This first working prototype was implemented within several hours. It proved a lot quicker to use than GIMP, due to the fact all interaction was done via the mouse or single keyboard keys – the only graphical interface was the window in which the image (and mouse cursor) were displayed.

To make shadows easier to ground truth, OpenCV's Canny edge detection was then used to detect edges in the input images and display them as an extra display-only layer when the E key was held down. The mouse cursor was intended to 'snap' to the closest edge in this mode, to make it easier to follow edges – but in practise this does not work very well if at all. That being said, simply displaying edges in the images does help with painting a more accurate ground truth.

This tool was used to ground-truth the 32 images of the `kondo1` image-set. This task took roughly one day to finish. There are some inaccuracies in the resultant ground truths due to having to draw around shadows using the mouse, but most inaccuracies were from human error rather than problems with the tool.

2.3 Test Harness

The test harness software was a major part of this project. It took roughly a week to implement the functionality outlined in Chapter 1.10.4, *Requirements for Test Harness*. The configuration files were written first. Command-line argument parsing was then implemented using the Python `argparse` module, which provides a sophisticated way of parsing these arguments.

[13] `http://www.pygame.org`

With sensible configuration files and command-line arguments, the rest of the script could be implemented fairly quickly.

2.3.1 Parameter Tree building

A challenging part of writing the test harness was implementing the parameter list generation. Given this configuration, for example:

```
chain:
  program1:
    argument1: ['foo', 'bar', 'baz']
    argument2: [1, 2]
  program2:
    argument1: ['a', 'b', 'c']
```

The test harness had to generate a list of command-lines:

```
[
  'program1 --argument1 foo --argument2 1; program2 --argument1 a',
  'program1 --argument1 foo --argument2 1; program2 --argument1 b',
  'program1 --argument1 foo --argument2 1; program2 --argument1 c',
  ...
  'program1 --argument1 baz --argument2 2; program2 --argument1 c'
]
```

Which the test harness could then loop through and pass to a shell (`bash`) for execution. The parameters in the above configuration example would result in a list of $3 \times 2 \times 3 = 18$ unique shell statements in which `program1` would be executed followed by `program2`.

It was known that this could achieved by building a tree structure of the parameters, but how exactly to construct this tree was a difficult thing to conceptualize. The tree was implemented using Python's `dict` (Dictionary) structures, which are essentially hash-tables (dynamic arrays where items are referenced by key rather than index; this key is usually a string).

Getting the tree structure right took a lot of trial and error. Luckily, since Python is an interpreted language, it is easy to experiment with. The interactive mode of `ipython` was used to prototype functions for building a tree as well as generating the command line strings from branches of the tree.

Building command strings from the tree was fairly simple. The tree is recursed in a depth-first manner. Every node down a branch is a dictionary of dictionaries. Dictionary keys are appended to a string. When a dictionary contains only one value (and that value is `null`, or `None` in Python), this is considered a termination condition for the recursion. The resulting string is added to a list of strings.

The difficult part is knowing which strings are program names and which are program arguments, as this information is not encoded in the string. The simple way of solving this was to replace instances of "`--program_name`" in a string with "`; program_name`". The preceding semicolon is inserted to let the shell executing the command statement know that this is a separate program. Obviously this approach would cause problems if `program_name` was also an argument or parameter to one of the other programs. These issues could be avoided had proper tree data structure classes been implemented.

Once the tree structure was implemented, getting the test harness to run chains full of programs was easy to implement using the `subprocess` module.

Development of the test harness continued throughout the majority of the project – new features being implemented only when they were needed. Focus was on making the configuration files for the harness simple to read and change, so new shadow detection methods could be tested with minimal effort.

2.3.2 Process Pools

Mode	**Time**
Serial (Default)	$3m32.757s$
Parallel (`-p` flag)	$0m53.802s$

Table 2.1: Execution times for the test harness in serial and parallel execution modes (times averaged over 10 runs each). This was tested on a laptop with 8 processors.

Once the first shadow detection methods was properly implemented, the test harness started to take quite a while to test the chains. The harness executed all the programs serially, and the time it took to execute each program was CPU-bound rather than Input/Output-bound (CPU usage whilst the harness ran would spike, but there was little hard disk activity). Because of this, it was decided to add a 'parallel' mode to the test harness, which would run the chains in parallel on $N - 1$ CPUs, (where N is the total number of processors on the system). One CPU was kept free as using all CPUs caused desktop applications to stall whilst the harness was running. Introducing a parallel execution mode had a definite improvement on the execution time of the test harness, which came in very useful later in the project as testing more complicated shadow detection programs with a greater number of parameters could take hours to finish.

The parallel mode is implemented using a "process pool". This is a queue structure. How this pool works is best explained through the pseudo-code in the appendices (algorithm A.11). Jobs can be ran asynchronously to the test harness by creating `subprocess.Popen` objects in Python, which forks the processes by default. Determining when jobs have finished is done by polling the queue every 100 milliseconds.

2.3.3 Scalability

Since the test harness was able to produce so many unique sets of parameters for any given chain, there was concern that the output of the chains would use up considerable memory (if kept in `/tmp` as is default; a directory which is commonly stored in memory on modern Linux systems) or disk space, or the large number of directories would cause problems (e.g. be slow to enumerate).

However, this has proven not to be the case. The output images produced by the chains, being highly-compressed greyscale PNG images, are very small – around 4 kilobytes maximum. The largest image sets (the artificial data) contain 250 images each. One of the more complicated shadow detection programs has a configuration with 5 different parameters with 2–4 possible values each. This results in 216 different chains ($3 \times 3 \times 4 \times 3 \times 2$). $250\,images \times 216\,chains \times 4KB = 5,4000KB = 52.7MB$. This is still a very conservative amount of memory consumed.

There were several occasions where running the test harness on all image sets using all chains exhausted the memory of the machine it was running on – but this could be attributed to the large number of large desktop programs also open at these times.

One problem left unsolved is the error handling of the test harness. It had no means of detecting when a program crashes due to a segmentation fault (segfault) rather than a normal error. When testing a new shadow detection method in parallel, there were points where they would segfault rapidly. This would eventually cause the whole system to grind to a halt and then throw a kernel panic (It was a proud moment when innocuous userland code brought the whole system under).

This was later solved by adding a prompt to the test harness – when the number of errors reaches a threshold amount, the script pauses and waits for user input.

2.4 Simple Thresholding program

Once the first functioning prototype of the test harness was written, with Pioneer images and some ground truth images readily available, development began on the C++ shadow detection programs. Before the ground truth comparison code could be written, there needed to be an algorithm with output images to compare it to. A simple thresholding program which converted input images to greyscale and then thresholded based on command line parameters was written.

2.4.1 Configuring CMake, finding libraries

In order to compile anything, CMake had to be configured, which meant creating a `CMakeFile`. This was fairly easy to achieve by following the documentation for CMake, but it had some problems finding the libraries for OpenCV. This was solved by finding a copy of `FindOpenCV.cmake`, a module for CMake which automatically found OpenCV libraries.

CMake had some trouble detecting the configuration of my system and incorrectly determined that my machine couldn't link to libraries dynamically. It statically linked all of OpenCV and the Boost `Program_Options` library to my programs at first. This was not discovered until later on in the project, when the static linking started to cause problems with memory leaks in OpenCV functions. Once this problem was discovered and fixed (by forcing CMake to link libraries dynamically), some of the memory leaks and segmentation faults went away. It also meant that my programs loaded faster (not having to load all of OpenCV into memory per-program, every time they were ran) which had a noticeable impact on the execution time of the test harness.

2.4.2 A minimal design

The first thing to be written was `shadows.hpp`; a header file containing the pixel values for the different classes output by all the shadow detectors. By including this header, all programs could make use of the global variables `C_SHADOW`, `C_BACKGROUND`, `C_PENUMBRA` and `C_UNKNOWN`. If it was decided that the values of these variables should be changed in the future, only this header would have to be altered.

2.4.3 Command-line parsing

Before a list of images could be processed, the program needed a way of obtaining such a list. Boost's `Program_Options` library provides a powerful command-line parser, which allows 'multi-token' arguments. The library parses command-line arguments into the correct data-types (if the '`-threshold`' argument is passed to an `int`, the parser converts it to an integer first). It also handles unknown arguments well; it can be made to exit with errors if an unknown argument is passed to a program, or simply ignore it.

2.4.4 Using OpenCV

Learning to use OpenCV was very easy. Thresholding an image can take as little as four lines of code:

```
Mat image = cv2::imread("image.jpg", cv2::CV_LOAD_IMAGE_GRAYSCALE);
Mat output = cv2::Mat(image);
cv2::threshold(image, output, 50, 255, cv2::THRESH_BINARY);
cv2::imwrite(output, "output.png");
```

It has extensive documentation and tutorials, the former of which make it very easy to start using without having to have much experience with C++. As most of the library functions use the `cv2::Mat` (Matrix) data structures, very rarely do arrays, pointers or manual memory allocation have to be used.

Occasionally, it will have some issues with memory allocation which cause it to crash programs with segmentation faults.

It also has some conceptual differences which can be confusing at first. Instead of using the RGB colour space by default, it uses BGR – the red and blue channels flipped. When used with other image libraries, without converting to RGB and back, this can lead to strangely-coloured images. Images are also accessed in $(row, column)$ format, or (y, x), instead of $(column, row)$, or (x, y). This can be confusing when experienced with other libraries or software which refer to images *column-first*.

After several days of learning how to use OpenCV and the Boost libraries, a working thresholding program was ready to be ran through the test harness. It could load a list of images and then output PNG images to a directory specified by the `--output_dir` command-line argument.

2.5 Ground Truth comparison program

In order to test the simple thresholding program in the test harness, the program for comparing program output with the ground truth had to be developed. The code to load files and loop through the input images was copied and pasted from the thresholding program, with the thresholding options removed and a `-ground_truths` option added. The program assumes that the input images and ground truths are in the same order – it does not do any sorting. This could be a problem if running it manually from the command-line, but the test harness sorts files before passing them to the ground truth program.

As this program was keeping counts of true/false positives/negatives for four different classes (Shadow, Object, Penumbra, Background), some C++ classes were defined for storing data. The `ROC` class for storing the counts for each taxonomical class. A `Statistics` class stores four `ROC` instances for the taxonomical classes.

The ROC class was also responsible for performing the calculations relating to ROC curves: True Positive Rate (Sensitivity), False Positive Rate (Fallout), along with some other calculations. These calculations were taken from the Wikipedia entry for ROC curves[14], under the section *"Terminology and derivations from a confusion matrix"*.

Output of the ROC data was in Comma-Separated Value (CSV) format, as it is a very portable format which can be pasted directly into most Spreadsheet applications and read easily by most tools and languages. It is also very easy to output CSV from a C++ program: In this case, a `ofstream` (output file stream) was opened – `roc.csv` in the directory specified by `--output_dir` was created and opened – and values from the ROC classes output directly to the stream:

```
// s == an instance of the Statistics class
// csv == std::ofstream('roc.csv') in the current working dir (output_dir)

csv << s.shadows.true_positives << ',' << s.shadows.true_negatives << ','
    << s.shadows.false_positives << ',' << s.shadows.false_negatives << ','
    ...
    << s.objects.false_positives << ',' << s.shadows.false_negatives
    << std::endl;
```

This approach becomes more difficult to manage when the number of columns in the CSV is high. When including all possible derived ROC statistics for each of the four classes for every image, the number of columns can be very large. Each of those columns also requires a header. A lot of care has to be taken that the headers of the CSV file align with the columns of data. A good data structure which stores headers as well as columns of data in the same order may solve this problem. As only the True Positive and False Positive rates have been used for evaluating shadow detection performance so far, time was not spent exploring this.

Counting the positive/negatives for each class is fairly trivial; each input image is looped through by row first and then column. The pixel at (y, x) in the input is compared with the pixel at (y, x) in the ground truth. The logic for this is demonstrated in figure A.10.

To verify that the ground truth program was keeping correct counts of positives and negatives for each class and correctly calculating the true positive and false positive rates, some very small simple 3×3 pixel images were created. The pixels in these images were very easy to count by hand and calculate the correct true positive and false negative rates. The CSV output was pasted into a Spreadsheet application (LibreOffice Calc) and an ROC curve charted using the charting tool available in that program.

2.6 Graphing the Results

Now that there was a detection method which could be compared against the ground truth using a program whose output could be trusted, a means of plotting the resulting data on an

[14] http://en.wikipedia.org/wiki/Sensitivity_and_specificity

ROC curve from a large number of CSV files was needed. Importing all the CSV files into a spreadsheet application would be difficult, especially as there may be hundreds of files created per chain tested by the harness.

There existed several ways of automating the process of converting a large number of CSV files into a single graph. GNUPlot[15] could have been ran from command-line, or an Octave script could have been written to plot the data [16]. Having plenty of experience with Python, it was easiest to load the CSV files in Python using `numpy.recfromcsv`. Then `matplotlib`[17] was used to plot the graphs; `matplotlib` is able to plot large amounts of data efficiently and save the resulting graph in a number of formats (including PDF – useful for this report).

A script to graph results from the test harness was therefore written in Python using `numpy` to read in the CSVs and `matplotlib` to display the graphs. An annotation class was written – any function in the Python script with the `@graph` annotation could be picked as a command-line argument. This allowed a number of different graphs to be defined in the script, which shared common data.

Importing the CSV files into a `numpy` "record array" allowed for some advanced filtering, as `numpy` arrays can be indexed using arbitrary boolean expressions:

```
array[array['True Positive Rate'] > 0.5]
```

would return a view of an array where all the array column named "True Positive Rate" had values above 0.5.

Later on in the project, it was decided to store all of a chain's parameters in the ROC CSV files – so ROC curves could be coloured in a number of ways: Instead of comparing different chains (in different colours), the ROC charts could display different values of the `input_blur` parameter in different colours. This allowed to test how different parameters of programs affected the true positive and false positive rates.

Getting the graphs right also took a lot of trial-and-error development, and like the test harness, extra features were implemented throughout the project. ROC curves are plotted as scatter plots rather than line graphs, as the sheer number of data points created very crowded graphs which took a long time to draw when lines were used.

2.7 Different colour spaces & Otsu thresholding

Once the test harness, ground truth comparison and graphing programs had functioning prototypes, a testing framework was therefore in place which would allow the evaluation of other methods of detecting shadows.

[15] `http://www.gnuplot.info/`

[16] `https://www.gnu.org/software/octave/doc/interpreter/Two_002dDimensional-Plots.html`

[17] `http://matplotlib.org/`

The second method to implement was Otsu thresholding. This required a very small modification to the original greyscale thresholding code – to enable Otsu thresholding within OpenCV means simply passing the `THRESH_OTSU` flag to the `threshold` function. An extra command line parameter was added, `--otsu_mode`, which could be true or false (false being the default). Then a new chain was defined in the test harness configuration, `otsu_test`.

The next step was to test converting the input images into a number of different HSL colour spaces and threshold on the luminance (brightness) channel of those colour spaces. This was only slightly more complex, requiring another command-line argument, `--colour_space`, and OpenCV's `cvtColor` ("convert colour") function called in an if-statement in the main body of the program.

Next, Gaussian blurring was added to the simple thresholding program, as part of an experiment to see whether blurring the input images would help to reduce noise. This was yet another simple change requiring an extra command-line argument and a call to OpenCV's `GaussianBlur` function.

These three pieces of functionality were implemented very quickly and were also tested very quickly using the test harness.

2.8 Implementing existing methods from a paper

Now having learnt the basics of OpenCV and becoming more confident with C++, it was time to implement some existing shadow detection methods. The simple chromacity based approach to shadow detection used in the Shadow Survey paper [3] (section 3.1 *Chromacity-based Method*) was chosen. This method was originally described in a 2003 paper by Cucchiara et al. [21]. It is a per-pixel method which uses the saturation and value components of the HSV colour space to determine whether a pixel is shadow based on three different conditions:

1. $\beta_1 \leq \left(F_p^V / B_p^V\right) \leq \beta_2$
2. $\left(F_p^S - B_p^S\right) \leq \tau_S$
3. $\left|F_p^H - B_p^H\right| \leq \tau_H$

Where F_p is the *foreground pixel* and B_p is the *background pixel*. *H, S* and *V* are those pixels' values in the HSV colour space. β_1, β_2, τ_S and τ_H are all thresholds which must be chosen empirically.

In the original paper, the shadow detection step is part of a longer image processing pipeline. The concepts of *background* and *foreground* pixels come from an earlier step in that pipeline, when what the paper describes as "Moving Visual Objects" (MVOs) are detected. The background image is a rolling average of the pixels wherever the scene is static, and the foreground image contains only those pixels which are different to the background average.

This approach works well for fixed-camera systems such as surveillance cameras, where the camera is assumed not to move often, or with very predictable motion (such as a CCTV camera panning to one side). However, background subtraction would perform very badly when the camera is in motion, as the rolling average of pixel values will be constantly changing and the concepts of "foreground" and "background" will degrade – as shown in figure 2.12.

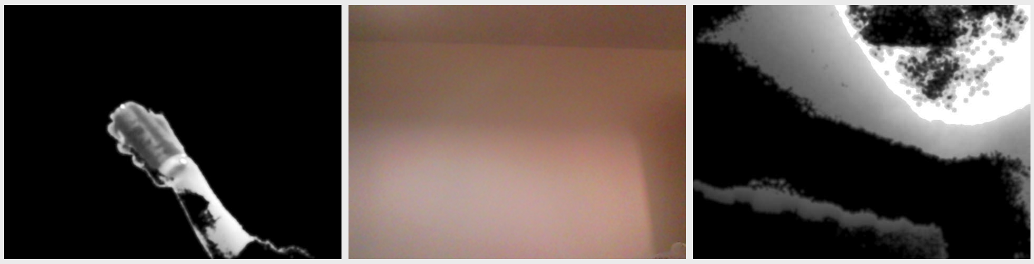


Figure 2.12: Testing background subtraction using a laptop's webcam. The middle image is the background image as seen by the camera. The image on the left is a correctly detected foreground object, as the background remains static. The image on the right is the result when the camera itself is panned upwards towards the ceiling. This experiment was performed using OpenCV's `BackgroundSubtractor` class.

Without the concept of foreground and background, it was decided that the *sliding-window* approach used in the Shadow Survey paper should be adapted. In that paper, the shadow detection method was implemented with a 5×5 window of pixels rather than looking at every pixel of an image separately.

This project's implementation made use of such a window as a substitute for foreground and background images. The centre pixel of the window was treated as the foreground pixel. A mean average of the pixels surrounding it (the window) was used for the background. This resulted in an implementation which detected edges within images but did not fill in the regions that these edges bordered.

2.8.1 Filling in the gaps

A simple way (in theory) of getting shadow *regions* instead of just shadow *edges* was to detect the edges output by the method and then "flood-fill" whichever side of the edge had the darker pixels.

To get a list of edges and the (y, x) coordinates of their pixels, the OpenCV function `findContours` was used. `findContours` uses an implementation of the contour-finding algorithms originally described in a 1985 paper by Suzuki [22]. This returns a list (`std::vector` in the C++ implementation) of contours. Each contour itself is a list of (y, x) pixel coordinates.

How to sample pixels from either side of an edge was the next investigation in the project, discussed in the following section.

2.9 Edge Analysis

Nearly all of the methods described in the Shadow Survey paper used the concept of background and foreground images. This meant that different approaches would have to be explored. Detecting shadows based on edges in an image looked promising, so an investigation into whether shadow edges and object edges exhibited different properties was conducted.

This required a means of seeing the changes in pixel values across edges in an image, or a set of images – which was the next thing to be implemented. Another Python tool which used OpenCV's Python bindings and `matplotlib` was written to plot the values across edges, with hue, saturation and value channels plotted separately. This tool was implemented in Python rather than in C++ as it involved some complicated functions and data structures which would be easier to write in Python.

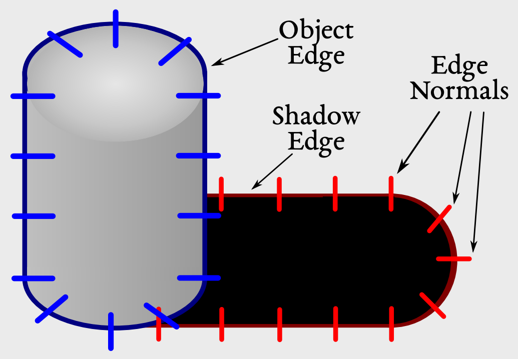


Figure 2.13: Diagram demonstrating the concept of *edge normals*.

To get values across an edge required sampling lines of pixels that were perpendicular to the line segments within the edge – getting the *normals* of the edge.

A normal of a line is the line itself rotated by 90°, which is a very trivial calculation. A normal will be of equal length to the original line. When investigating changes across an edge, a fixed-length normal was desired, which requires slightly more complicated trigonometry, but is still very easy to implement.

2.9.1 Calculating normals

The steps for getting a normal of a fixed length N are as follows:

1. Find the centre point of the line:
   $CENTRE_{x,y} = A_{x,y} + \frac{(B_{x,y} - A_{x,y})}{2}$
   (Where A and B are either ends of the line)
2. Calculate the difference between point A and B:
   $DIFF_{x,y} = B_{x,y} - A_{x,y}$
3. Calculate the angle of the line:
   $ANGLE = atan2(DIFF_x, DIFF_y)$
4. Using this information, create a new line which has it's origin at the centre of the original line, but is of fixed length N and is rotated by 90° to $ANGLE$:
   $NORMAL_a = \left(CENTRE_x - \frac{N}{2} \times sin(ANGLE), CENTRE_y - \frac{N}{2} \times cos(ANGLE)\right)$,
   $NORMAL_b = \left(CENTRE_x + \frac{N}{2} \times sin(ANGLE), CENTRE_y + \frac{N}{2} \times cos(ANGLE)\right)$
   (Where $NORMAL_a$ and $NORMAL_b$ are the end-points of the new line)

Care had to be taken here, as OpenCV returns points in (y, x) form rather than (x, y) form. Python's trigonometry functions also work in radians rather than degrees.

2.9.2 Simplifying edges

To confirm that normals were being calculated correctly, they were displayed on the original image (as seen in figure 2.14). This visualization of the normals was useful for debugging. The contours detected by OpenCV for these natural images were shown to be very complex, consisting of many line segments that are only a few pixels long each. Many of them crossed each other due to the changes in line angle, which meant that the same pixels were sampled repeatedly, resulting in very crowded graphs and skewing the averages.

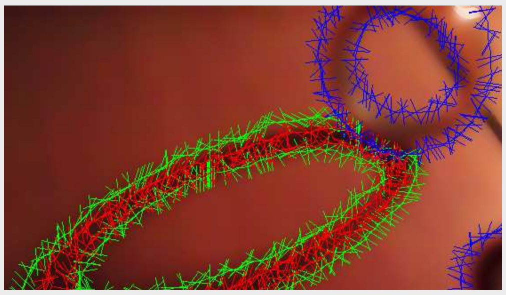


Figure 2.14: Normals of edges detected in the ground truth for an image in the `kondo1` image set, overlaid on the original image.

Without a clear way of getting an equal distribution of normals across an edge (e.g. every 10 pixels), it was decided to try some methods of "simplifying" contours by strategically removing points from them, resulting in contours which will be less accurate but should follow the general shape of the original contour. After some experimentation, a method of simplifying the contours by removing line segments that were close to each other was implemented. If the distance between a point and its previous point in the list of points that made up a contour was below a given threshold, the previous point was removed from the list.

This was a very naive implementation that did not take into account the shape of the contour. It introduced occasional bugs where "closed" contours (those which make a concrete shape, like a circle, rather than being open lines) would become open. Occasionally it would change the shape of the contour itself.

OpenCV's own "polygon approximation" function, `approxPolyDP`, was also tested, but had little effect on the contours – even with a high $epsilon$ value (the maximum allowed difference between the original contour and the simplified one), points that were very close together were not removed.

Due to the bugs in implementation, contours were not simplified in the graph tool. This resulted in more data being sampled in the images (as shown in figure 2.14). The "minimum distance" function is used in the feature extraction tool (explained in the following section) to reduce the number of instances however.

2.9.3 Getting pixel values across normals

The Python bindings for OpenCV represent images as `numpy` arrays. This means that all `numpy` functions can be used with loaded images. As these arrays support 2-dimensional indexing, diagonal lines of any angle can be used to index an array (using `numpy.hypot`) and get a 1-dimensional list of values.

`numpy` does, however, interpolate these values. Should a line cross two pixels at once, the result is the average of those two pixels. This does not have much of an effect on natural imagery where even sharp edges change gradually at the per-pixel level, but this results in strange plots when tested on the artificial image sets. It is important to be aware of this sub-pixel interpolation, as it is essentially synthesizing new data, and may not always be accurate.

2.10 Shadow Edge Detection using Machine Learning

Once the edge edge analysis tool was written, interesting patterns did appear in the resulting graphs, which indicated that shadow edges did differ from other edges in the images. It was decided that the final method of detecting shadows should incorporate some machine learning, using edge normals as features. These features could be used to train a learning algorithm. Given a good training set, such an algorithm could be used to classify new edges in an image as "shadow edge" or "not shadow edge".

The idea of using edges as machine learning features came from the approach to detecting shadows demonstrated by Lalonde et al. [23]. Their approach proved a high success rate on outdoor photographs taken from public sources on the internet (photos outside of any controlled environment). They trained a decision tree classifier to discern between detected edges in an image using a large number of different features (such as the changes in brightness and saturation over an edge at different angles).

This proved to work with high accuracy on consumer photographs, but there were some things which would make a full implementation of their approach unsuitable for this project:

- *Image Quality*: The images used to test their approach were of considerably higher resolutions than the images used in this project – coming from still-image digital cameras, as opposed to the video-capturing cameras used in this project. This meant that noise may have been much less of an issue.
- *Outdoors Photography*: Their approach focuses on outdoors photographs, in which sunlight is the only source of light, and which casts well-defined hard shadows. The shadows in this projects image sets have more varied characteristics than "outdoors ground shadows".

However, as their approach was shown to work well on a varied set of images, it was decided to use some of the concepts described in their paper. The implementation used in this project is described in the following sub-sections.

The feature extraction was implemented in Python first, being easier to debug than C++ and having access to numpy. Should it result in a successfully trained classifier, it was planned to be re-implemented in C++ for the performance benefits – but with only a week allocated remaining in which to write code, this was not a priority.

2.10.1 Detection of edges

First, a large number of "weak" edges are detected in an input image. This is achieved by computing gradient magnitudes on the input image. This can be achieved by applying two Sobel operators to the image – one in the horizontal direction and one in the vertical direction. Then the gradient magnitudes can be calculated. This is simple to do in OpenCV:

```
Mat gradient_boundaries(Mat input) {
    Mat Sx, Sy, mag, mag_uint;
    Sobel(input, Sx, CV_32F, 1, 0, 3);  // 1, 0 == horizontal (1 x, 0 y)
    Sobel(input, Sy, CV_32F, 0, 1, 3);  // 0, 1 == vertical
    magnitude(Sx, Sy, mag);
    mag.convertTo(mag_uint, CV_8UC3)  // convert back to an 8-bit image
    return mag_uint;
}
```

Their approach then uses the "watershed" algorithm on the gradient map. Their reasons for using watershed are not explained in the paper, but it appears to have the effect of strengthening strong edges and weakening weak edges (essentially increasing the contrast between them).

The next step is to run the Canny edge detector with a small difference between its thresholds, to filter out weak edges, leaving only strong edges in the image.

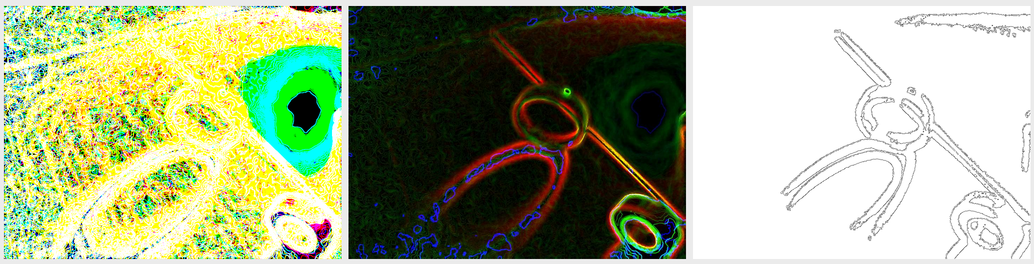


Figure 2.15: The Lalonde et al. approach to detecting edges. The image on the left shows the gradient magnitudes in an image in the HSV colour space. The middle image shows the effect of applying the watershed algorithm to the magnitudes. The image on the right demonstrates Canny edge detection performed on a (grayscale) version of the watershed output.

2.10.2 Extraction of features

Lalonde et al. use "oriented gaussian derivative filters" to sample pixels on either side of an edge. A clear way of doing this was not obvious during the project, so this project makes use of edge normals instead, as it was known how to calculate these thanks to the image analysis done previously.

Normals were divided into two sides (either side of the edge which the normal was centred around). Using these sides, the following features were computed:

- Colour ratios, calculated as:
  $\frac{min(f_1(p),f_2(p))}{max(f_1(p),f_2(p))}$
  Where f_1 and f_2 are either side of the edge.
- Mean, minimum and maximum values for both edges.

This was done independently for each of the channels in the Hue, Saturation, Value colour space, as well as each channel in the Red, Green, Blue colour space.

Other features were tested (such as ratios *between channels* rather than *between edges*), but due to the already high amount of features these were left out at the end of the available implementation time.

This resulted in 32 different types of features in total. These features were extracted for every normal of every edge detected of every image. The results were output as comma-separated value format (CSV) files, which would be easy to import into a number of other packages, such as Weka (which was used later to test the data). Columns in the data corresponded to individual features, and rows corresponded to instances. Some images contained a very high number of edge normals. The `kondo1` dataset contained around $3,000$ normals per image, for example. The feature extraction script takes several minutes to execute when given the entire `kondo1` image set.

2.10.3 Labelling instances

Using the ground-truth, instances were labelled as 'shadow' or 'background', based on the class which was most prominent across a normal:

$$VALUE = argmax(ground_{normal})$$

$$LABEL = \{'shadow'\ if\ VALUE = 0;\ 'background'\ otherwise.\}$$

Where $ground$ is the greyscale ground-truth image corresponding to the input image currently being looked at, and $normal$ is the list of (x, y) coordinates corresponding to a generated normal.

2.10.4 Reducing data size

A huge amount of data was produced, which had to be reduced in order to fit into the machine's available memory. Before reductions, the largest CSV file (features extracted from the `kondo1` dataset) was approximately $150MB$ in size. The files were reduced in size to approximately $120MB$ by rounding down all the floating point numbers they contained to 3 decimal places. This resulted in a loss of precision in the data, but did not have a noticeable effect on it when used in Weka. The data was also compressed using the GZip compression, as Python's `gzip` module makes this very easy to implement.

This shrank the files considerably ($120MB$ files becoming $20MB$), which was helpful for storage, but did not solve the problems that occurred when decompressing that data in memory.

Upon inspecting the CSV files, it was found that a considerable number of instances were *negative examples* – examples labelled as "background" – at a far higher rate than those labelled as "shadow". Background examples made up 90% of the data. This itself could introduce bias in learning algorithms, as they would have far more negative examples than positive ones, making them far less likely to predict a new example as shadow. It was decided that the data could further be reduced by including a much lower ratio of background examples to shadow examples.

This was achieved at the CSV output stage of the script, when instances were still stored as a list in memory. For every image processed, the following was done:

1. Get a list of the extracted edge instances labelled as 'background';

2. Shuffle this list randomly;
3. For all instances in the original list:

   If the instance is labelled 'shadow' (positive example), print to file;

   If the instance is labelled 'background', print to file *only if* $\frac{|background\ instances|}{|shadow\ instances|} > N$, where N is an adjustable ratio.

With an enforced ratio of 3× as many background (negative) examples as there were shadow (positive) examples, the total number of instances was greatly reduced.

2.11 Using Weka to experiment with learning algorithms

The extracted features needed to be analysed somehow, to see if any of the features contained useful information about shadow edges which could be used to train a learning algorithm.

A suitable learning algorithm also had to be chosen. Lalonde et. al.'s approach used Boosted Decision Trees, but it was decided to test the training data on a selection of different algorithms. Neural Networks, Support Vector Machines and Random Forests were chosen for this.

Several frameworks exist for the evaluation of learning algorithms given different data. Weka[18] was suggested by the project supervisor. Others have suggested the Python module `scikit-learn`[19]. Weka was chosen, as no code had to be implemented in order to use it, which was good given the limited project time left by this point. Whilst it had a complicated user interface, Weka also had less of a learning curve and had reasonable defaults for evaluating learning algorithms.

Weka was able to read the CSV files generated by the feature extraction scripts. Once the learning curve of the user interface had been overcome, it was easy to configure the "Experimenter" to perform 10-fold Cross Validation tests using the three different learning algorithms on the edge normals data.

The training by Weka was quite intensive and took hours to complete, so was only performed three times during the project (left to run overnight).

2.11.1 Reading results from Weka

When testing learning algorithms via the Experimenter, Weka outputs a lot of statistics in ARFF format. ARFF is an extension of the CSV format, where the data is comma-separated values, but the file also contains more descriptive headers which define the columns – the types of data they contain (for examples: a variable string, a variable integer, a value which is one of three discrete classes), as well as their associations to other columns.

[18]`http://www.cs.waikato.ac.nz/ml/weka/`

[19]`http://scikit-learn.org/stable/`

This can be interacted with within Weka, through the Experimenter's "Analyse" tab. It can also be read into Python using the `arff` module[20].

However, the best output came from Weka's "Classifier output" field in the Explorer. This gave more detailed statistics, such as the confusion matrix seen in the second Weka screenshot (figure 2.16). No means of getting this confusion matrix seems to be available from the data produced by the Experimenter.

[20]`http://code.google.com/p/arff/`

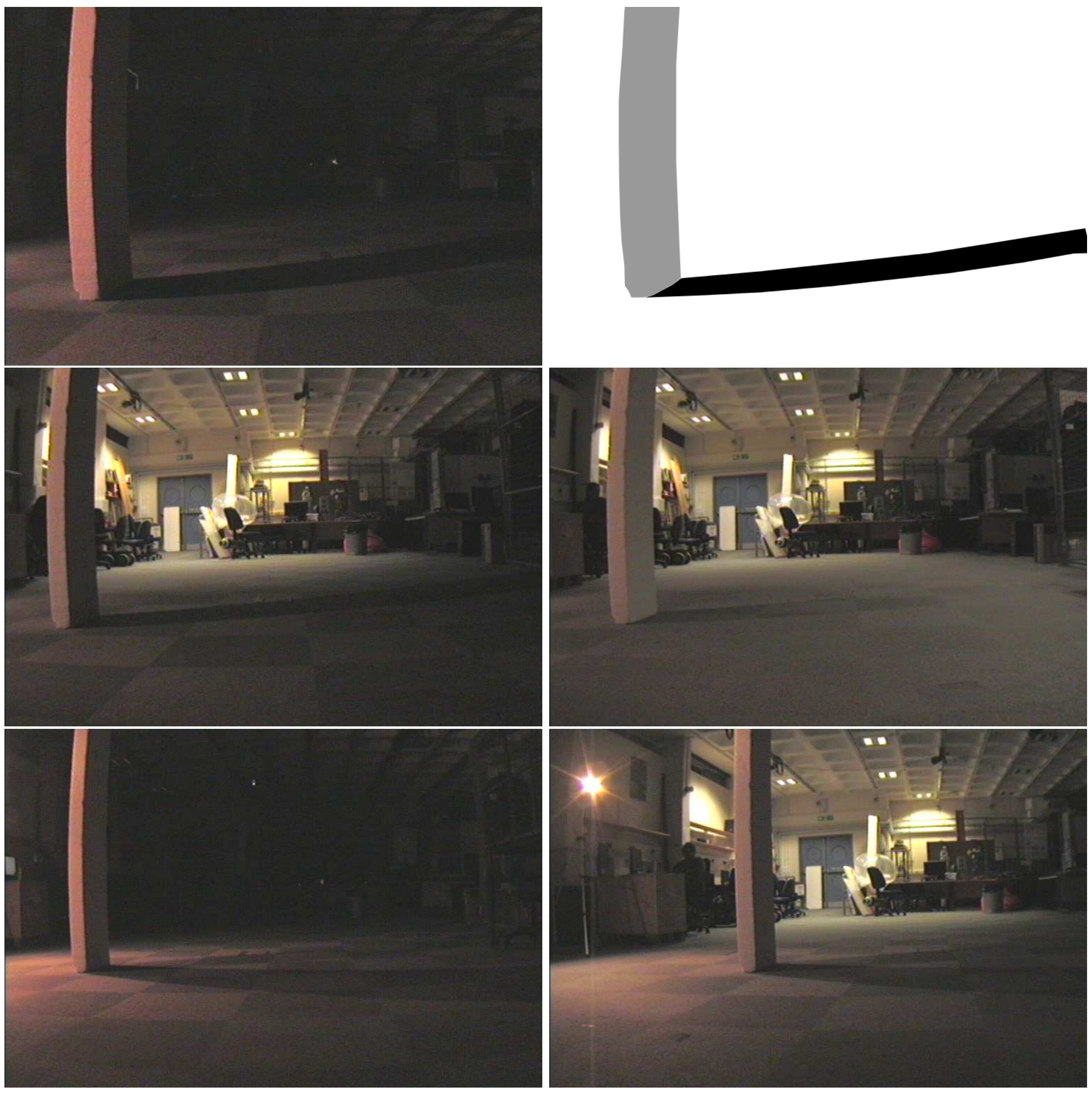


Figure 2.3: The different lighting conditions used in the Pioneer dataset. The image in the top-right is the ground truth for the top-left image – to illustrate the difficulty in ground-truthing this particular dataset. Many background objects have been ignored in the ground-truth.

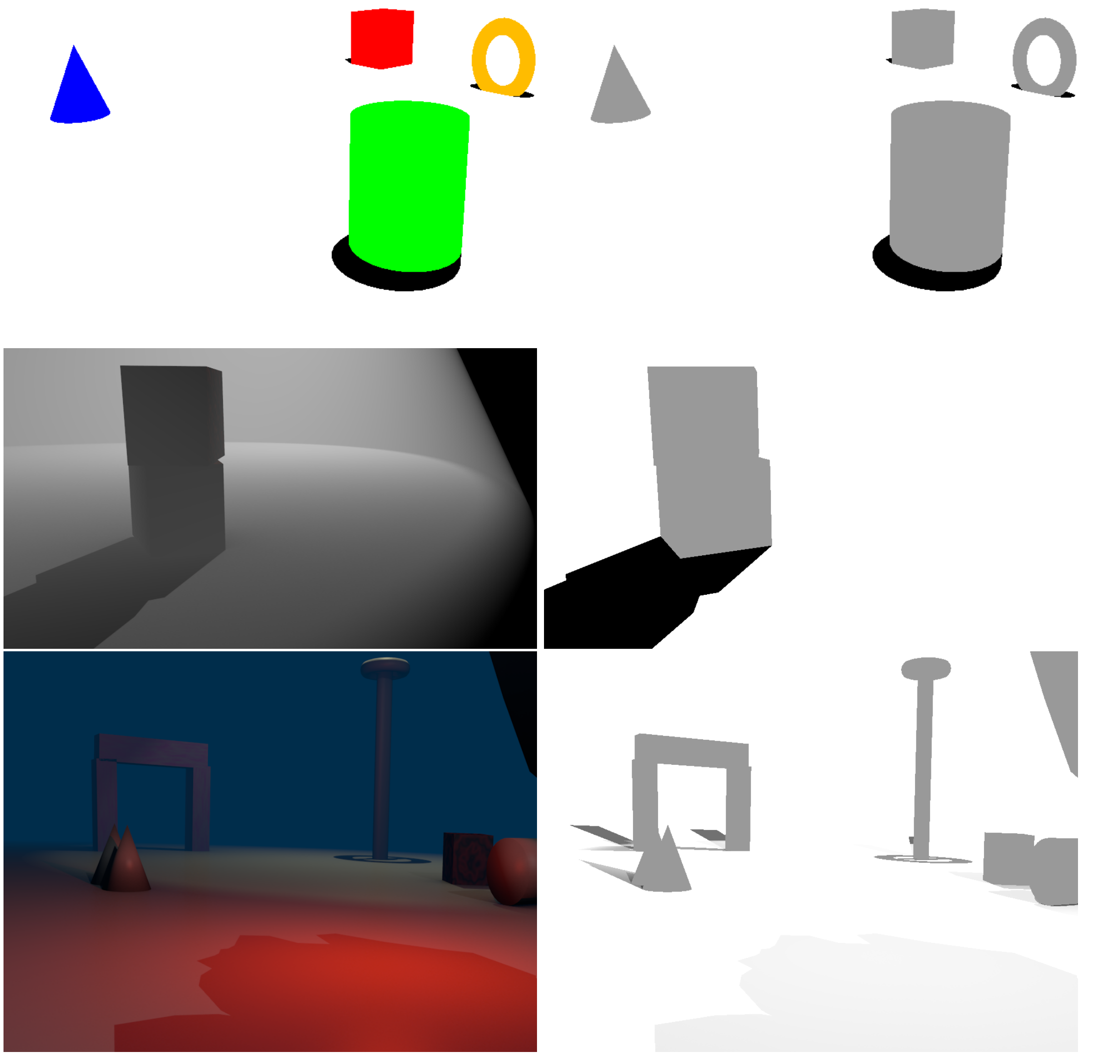


Figure 2.8: Example images from the artificial data sets (left) and their corresponding ground truth images (right).

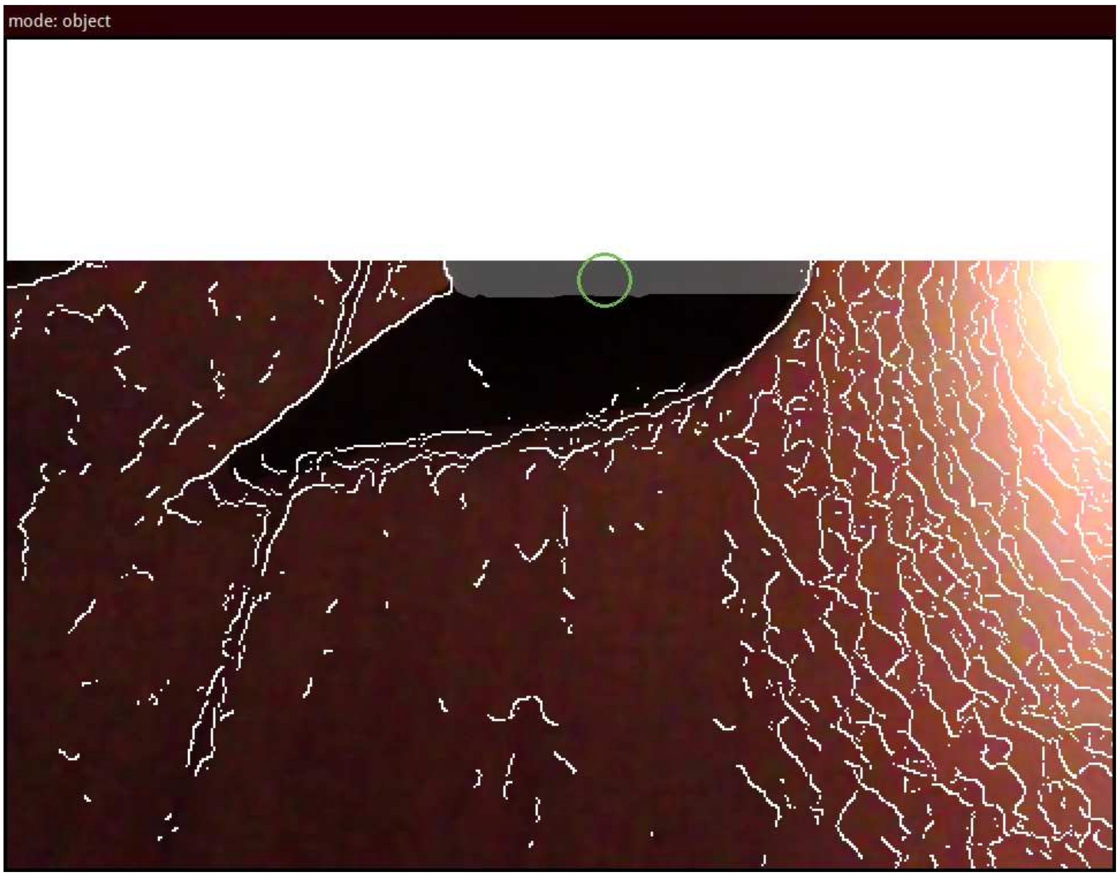


Figure 2.9: Screenshot of the final iteration of the Ground Truth Painting Tool. The image being displayed is at a 2× magnification, and the viewport has been panned to the top left of the image. The green ring in the image is the mouse cursor – the colour of the ring indicates which label class it will paint with (green for 'Object'), and the radius of the ring is the size of the circles it uses to paint with. The original image is shown beneath the ground truth. The ground truth can be seen underneath the cursor – the grey and black regions are already-painted ground truth. The edges detecged in the input image are displayed as the top layer.

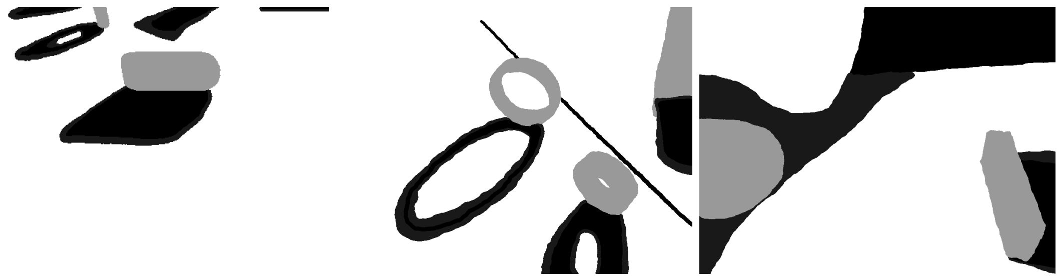
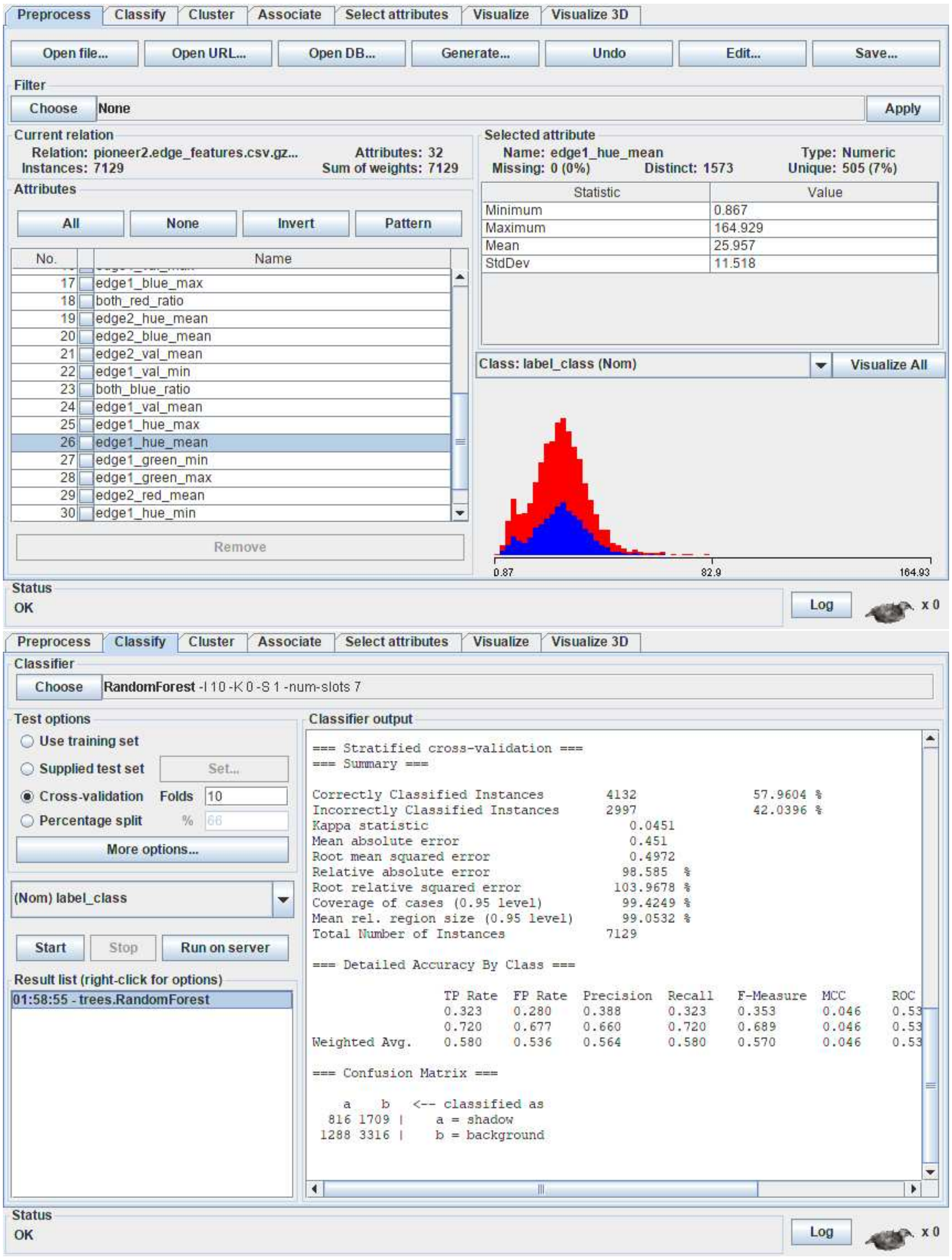


Figure 2.10: Three examples of ground truths produced using this tool.

```
program1
        \
         argument1
                  \
                   + foo
                   |    \
                   |     argument2
                   |              \
                   |               + 1
                   |               |  \
                   |               |   program2
                   |               |           \
                   |               |            argument1
                   |               |                     \
                   |               |                      + a
                   |               |                      + b
                   |               |                      + c
                   |               + 2
                   |                  \
                   |                   program2
                   |                           \
                   |                            argument1
                   |                                     \
                   |                                      + a
                   |                                      + b
                   |                                      + c
                   |
                   + bar
                        \
                         ...
                   + baz
                        \
                         ...
```

```
program1 --argument1 foo --argument2 1 --program2 --argument1 a
program1 --argument1 foo --argument2 2 --program2 --argument1 c
```

Figure 2.11: Graphical representation of the tree structure built from the configuration file example. The different colours represent different branches of the tree being followed.



Filter: None

Current relation
Relation: pioneer2.edge_features.csv.gz... Attributes: 32
Instances: 7129 Sum of weights: 7129

Attributes

No.	Name
17	edge1_blue_max
18	both_red_ratio
19	edge2_hue_mean
20	edge2_blue_mean
21	edge2_val_mean
22	edge1_val_min
23	both_blue_ratio
24	edge1_val_mean
25	edge1_hue_max
26	edge1_hue_mean
27	edge1_green_min
28	edge1_green_max
29	edge2_red_mean
30	edge1_hue_min

Selected attribute
Name: edge1_hue_mean Type: Numeric
Missing: 0 (0%) Distinct: 1573 Unique: 505 (7%)

Statistic	Value
Minimum	0.867
Maximum	164.929
Mean	25.957
StdDev	11.518

Class: label_class (Nom)

Status
OK

Classifier
Choose RandomForest -I 10 -K 0 -S 1 -num-slots 7

Test options
Use training set
Supplied test set
[✓ Cross-validation] Folds 10
Percentage split % 66

(Nom) label_class

Result list (right-click for options)
01:58:55 - trees.RandomForest

Classifier output

```
=== Stratified cross-validation ===
=== Summary ===

Correctly Classified Instances        4132               57.9604 %
Incorrectly Classified Instances      2997               42.0396 %
Kappa statistic                          0.0451
Mean absolute error                      0.451
Root mean squared error                  0.4972
Relative absolute error                 98.585  %
Root relative squared error            103.9678 %
Coverage of cases (0.95 level)          99.4249 %
Mean rel. region size (0.95 level)      99.0532 %
Total Number of Instances             7129

=== Detailed Accuracy By Class ===

                 TP Rate  FP Rate  Precision  Recall   F-Measure  MCC      ROC
                 0.323    0.280    0.388      0.323    0.353      0.046    0.53
                 0.720    0.677    0.660      0.720    0.689      0.046    0.53
Weighted Avg.    0.580    0.536    0.564      0.580    0.570      0.046    0.53

=== Confusion Matrix ===

    a    b   <-- classified as
  816 1709 |    a = shadow
 1288 3316 |    b = background
```

Status
OK

Figure 2.16: Screenshots of Weka in use on features extracted from the `pioneer2` image set.

Chapter 3

Results

This chapter gives the results of testing the different shadow detection methods described in the previous chapter. The methods based around thresholding and chromacity were evaluated using the test harness, and as such the metrics of "True Positive Rate" and "False Positive Rate" are used to evaluate them, as well as visual analysis of the the physical images they have produced.

For the method based on edge-features, the Weka outputs are analysed. These consist of Confusion matrices and error rates.

Throughout this chapter, the same image has been used for comparative purposes – Image number 20 from the `kondo1` image set. This image was chosen as it contains two objects as well as two complex shadow shapes with different levels of penumbra. All the methods were tested on all the images within the `pioneer2`, `kondo1` and `artificial0` image sets.

3.1 Simple Thresholding

The first hypothesis was that fixed-value thresholding on greyscale copies of images would produce a True-Positive Rate and False-Negative Rate with a high variability, that was slightly better than chance.

Thresholds of 0, 25, 50, 100 and 200 were tested (pixels whose brightnesses were beneath these thresholds were counted as shadow). Thresholds between 25–50 gave the best ratio of true-positives versus false-positives, however this is very variable, differing greatly between images in the same set.

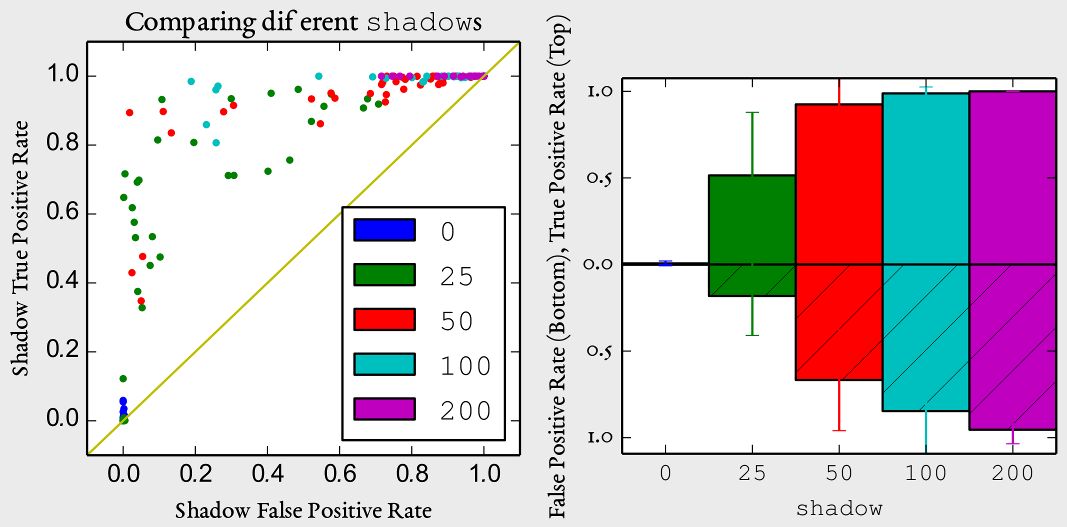


Figure 3.1: Results of shadow detection based on fixed-value, greyscale thresholding. Values between 25–50 gave the most accurate results.

3.1.1 Otsu Thresholding

The next hypothesis was that Otsu thresholding would actually perform *worse* than with a fixed-value threshold. This was based on the results of the work by Dee & Santos. Again, this hypothesis proved to be true. Otsu thresholding performed poorly because it had a high false-positive rate. It appears to set the threshold for many images too high, meaning that large areas of images are incorrectly marked as shadow. This can be seen in figure 3.2.

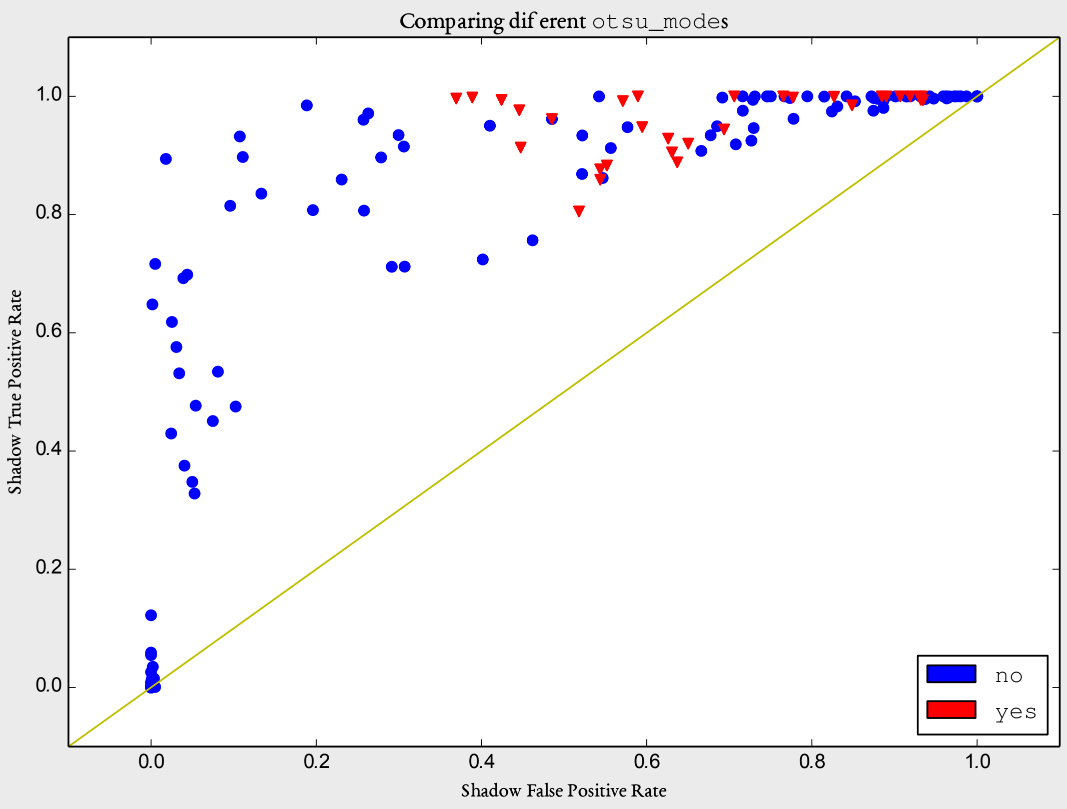


Figure 3.2: Comparing the performance of fixed-value thresholding (blue) versus Otsu thresholding (red).

3.1.2 Using different colour spaces

Disappointingly, using different colour spaces made little difference to the thresholding. Only minor differences in output could be seen. The best way of showing this is in the following ROC chart, which demonstrates little relation between overall accuracy and the colour space used.

For this experiment, both fixed thresholding and Otsu thresholding were tested. The fixed thresholds were 0, 25, 50, 100 and 200.

When the results are viewed as a bar-chart, there is very little difference in performance. It does show the greyscale method performing equally to the LAB and LUV colour spaces. HSV has a slightly lower True Positive rate than the others, but also has the lowest False Positive rate.

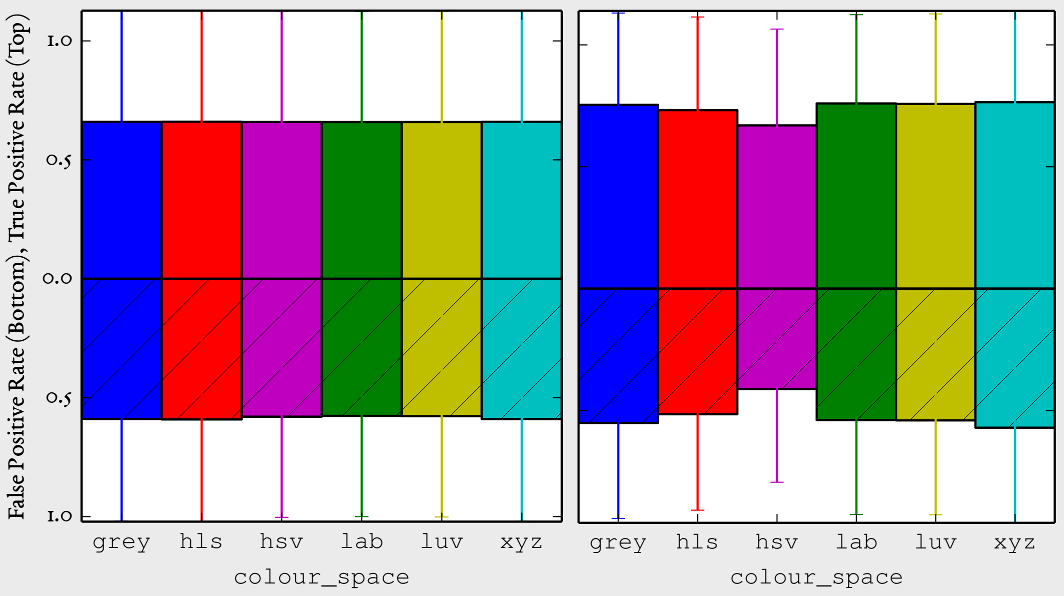


Figure 3.3: Comparison of different colour spaces tested on the `pioneer2` and `kondo1` image sets, respectively.

Converting to 3-channel colour HSL colour space is useful for more complex methods (where saturation and hue information may be needed), so it is promising that the discrepancies between the greyscale colour space and the HSL colour spaces are very small.

Because of this lack of variability between the colour spaces when used on these image sets, HSV was ultimately chosen for the later methods as it had the lowest False Positive rate overall and was conceptually the simplest to understand.

3.1.3 Gaussian Blur

The next hypothesis was that Otsu thresholding had a high false-detection rate due to the noisy histograms of the original images – perhaps it was finding a threshold between the wrong peaks in the histograms. Applying different levels of Gaussian blur could have a 'smoothing' effect on the histograms; removing particularly bright or dark spots which may have occurred due to unfiltered sensor noise.

This was tested on the greyscale space only, using Otsu thresholding. The blur values tested (size of the Gaussian kernel used) were `0`, 3, 9, 15 and 33. Applying even the heaviest Gaussian blur made very little difference in this method's performance, as shown by table 3.1.

Image Set	Mean TPR; `blur`= 0	= 3	= 9	= 15	= 33
pioneer2	0.98	0.98	0.98	0.97 (-0.01)	0.96 (-0.02)
kondo1	0.96	0.96	0.96	0.96	0.96
artificial0	0.92	0.92	0.91 (-0.01)	0.91 (-0.01)	0.89 (-0.03)

Table 3.1: Mean True Positive Rates across the three image sets with different levels of Gaussian blur. Values in brackets are the differences between the value in that column compared to the value in column 1 (where `blur` = 0).

3.2 Chromacity-based method

The chromacity-based method had five different adjustable parameters: `window_size` (the size of the frame surrounding each pixel as it is checked), `thresh_hue` and `thresh_sat` (the maximum differences in hue and saturation allowed to count a pixel as shadow – assuming that shadows have a similar saturation and hue to their casting surface), and `beta2` (maximum allowed ratio between frame's average value and pixel value [1]).

The hypothesis here was that, provided a selection of values for each parameter, a good set of parameters could be found that generalized well across the image sets tested. This would be seen as a clustering of points on a ROC chart, hopefully with a high True-Positive Rate and a low False-Negative rate.

This approach had very poor results; nearly all of the images were blank, meaning a very low detection rate. This chromacity-based method mostly followed what was documented in the original paper [21] – but the significance of the four parameters was not clear.

After some visual experimentation, a better set of parameter values was found. The `thresh_sat` and `thresh_hue` were set at 50, meaning that, in the HSV colour space, a maximum difference of 50 was allowed for the hue and saturation of a pixel compared to its surrounding window. As this projects implementation dealt with 8-bit images, all values should be in the range 0–255. `beta1` and `beta2` seemed to be the most significant. Also being scaled to the 0–255 range, it was found that only values over 200 had any appreciable effect; values below that would result in white images. These two parameters were both tested with the values 200, 220, 240, 250 and 254. The `window_size` parameter was also significant – as seen in figure 3.4, a large window size could have an "inversion" effect, where areas of shadow and areas of background were reversed.

True Positive and False Positive rates remained below 0.5. `beta2` appears to be the most significant parameter as it exhibits clusters when plotted on an ROC chart.

This approach performs poorly on the data-sets. According to the ROC graphs it performs worse than chance 50% of the time – much worse than the fixed-value thresholding.

[1]'value' referring to the Value channel of the HSV colour space

Given its best combination of parameters (`beta1` = 220 and `beta2` = 250), it appears to be falsely detecting a lot of small spots of shadow within the images. These appear to be symptoms of artifacting from the image compression.

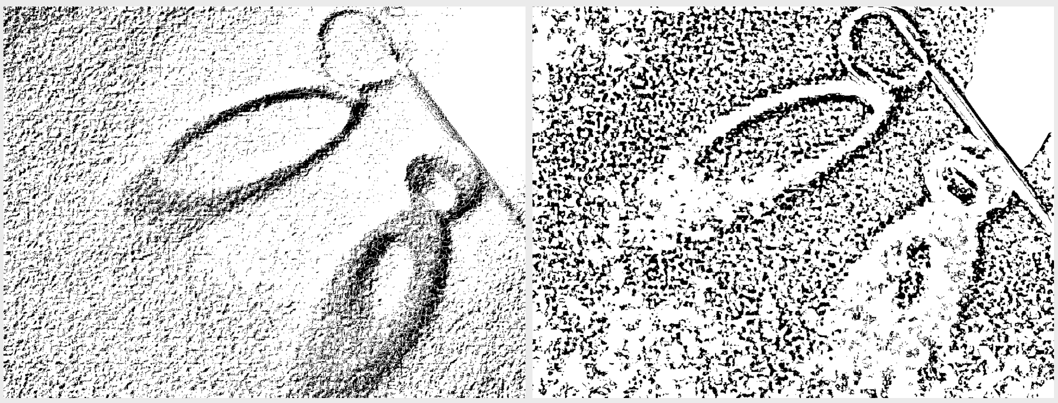
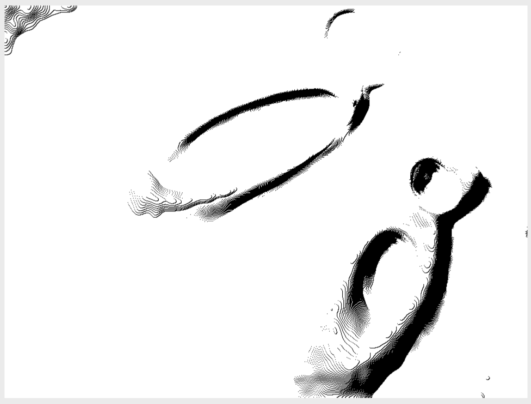


Figure 3.4: Comparison of different values of `window_size` on the Chromacity-based method. (Left: `window_size` = 1, right: `window_size` = 17)

Gaussian blurring the input images before running this method had a positive effect on noise on the image which was visually inspected (figure 3.5). No patterns emerged on the ROC chart for this, but when visualized as a bar chart, the False Positive Rate had a definite decrease. When testing Gaussian blur, all other parameters were fixed – `beta1` of 220, `beta2` of 250, and a `window_size` of 1 and 17. The `window_size` of 1 consistently performed the best. A window of 1 actually means the program samples a 3×3 window around each pixel. This was not known until the results of this method were analysed. The next section explains why this happens.



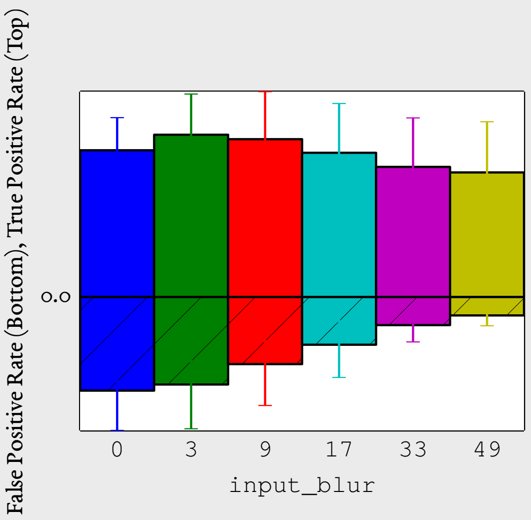


Figure 3.5: Effect of different levels of Gaussian blur. Left: Gaussian blur of 49, showing a clear reduction of noise. Right: Bar chart showing mean True Positive rate (top) and False Positive rate (bottom).

3.2.0.1 Investigation into `window_size`

Here is a section of the code which calculates the pixel coordinates for the frame around each pixel:

```
int half_window;
if(window_size % 2) {  // if window_size is an odd number
    half_window = 1 + (window_size / 2);
} else {  // if window_size is an even number
    half_window = window_size / 2;
}
int x_start = x - half_window; int x_end = x + half_window;
int y_start = y - half_window; int y_end = y + half_window;
```

When `window_size` is 1, `half_window` is 1 plus (1 divided by 2). As these variables are all integers, C++ rounds the results of these divisions down ('floors') to the nearest whole number – which will be `0`, in this case. It adds 1 to the result so that `half_window` will always be an odd number; creating a square in which the pixel at (x, y) is always the centre of. When `window_size` is 2, `half_window` is also 1.

This means that the reported window size is slightly wrong; a `window_size` of 3 actually creates a 5×5 window around pixels, as `half_window` $= 1 + \frac{3}{2} = 2$ (in C++ integer division), As this was not known until the final results of this method were analysed, it was decided to mention this in the report. The effects of fixing this implementation bug at this stage were not known.

To summarize, a `window_size` of 1 actually uses a 3×3 window around each pixel.

3.3 Dice Coefficients

To compare the two thresholding approaches against the chromacity-based sliding-window approach, a simple metric that can be used is the Dice Coefficient. This is a single value which indicates how much a set of results "agrees" with the ground truth.[2]

The Dice Coefficient for a set of results is calculated as follows:

$$Dice = \frac{TP}{(FP + TP) + (TP + FN)}$$

Values of the coefficient range between 0 and 1. A coefficient of 0 indicates no overlap between a results set and the ground truth. A coefficient of 1 indicates perfect overlap.

Table 3.2 shows the Dice Coefficients of the three different approaches for each of the image sets. The results are very varied. For images with a high contrast between shadow and background areas (as is the case with the `artificial0` image set), Otsu's method outperforms the others by a fair margin. The chromacity-based approach works best on natural imagery where lighting conditions are good (the `kondo1` image set).

Image Set	Fixed Threshold	Otsu's Method	Chromacity
pioneer2	0.063	0.061	0.043
kondo1	0.093	0.102	0.130
artificial0	0.173	0.479	0.025
Combined	0.113	0.241	0.056

Table 3.2: Comparison of three different detection methods, on three different image sets. For the fixed-value thresholding method, a threshold of 50 was tested. The chromacity-based approach was tested with the following parameters: `beta1` of 220, `beta2` of 250, `window_size` of 1, `input_blur` of 49, thresholds set at 50.

3.4 Investigation into Edges

As the Chromacity-based method seemed to be better at detecting the *edges* of shadows rather than shadows as a whole, it was decided to investigate how colour values changed across an edge.

[2]`http://sve.bmap.ucla.edu/instructions/metrics/dice/`

It was hypothesized that shadow edges would have a much more pronounced change in luminosity than other edges (for instances, the boundaries of objects). This hypothesis is visualized in figure 3.6.

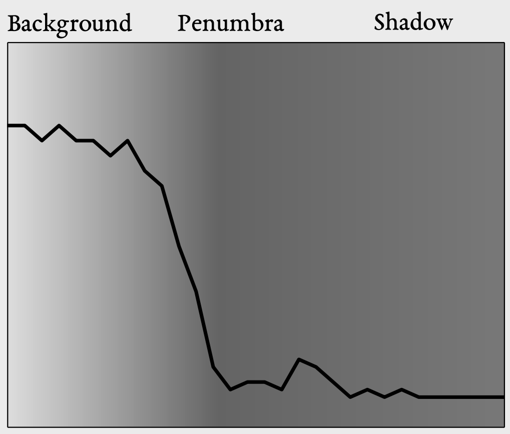


Figure 3.6: Hypothesis of how luminance would change across a shadow boundary.

As mentioned in the previous chapter, this hypothesis was tested using an analysis tool to plot pixel brightnesses across edge normals on a line graph. A normal length of 30 pixels was used in initial tests. The ground truth of an image was used to get edges – the ground truth was put through a Canny filter, and the detected edges were used to populate a list of normals from which to sample pixels from the original image.

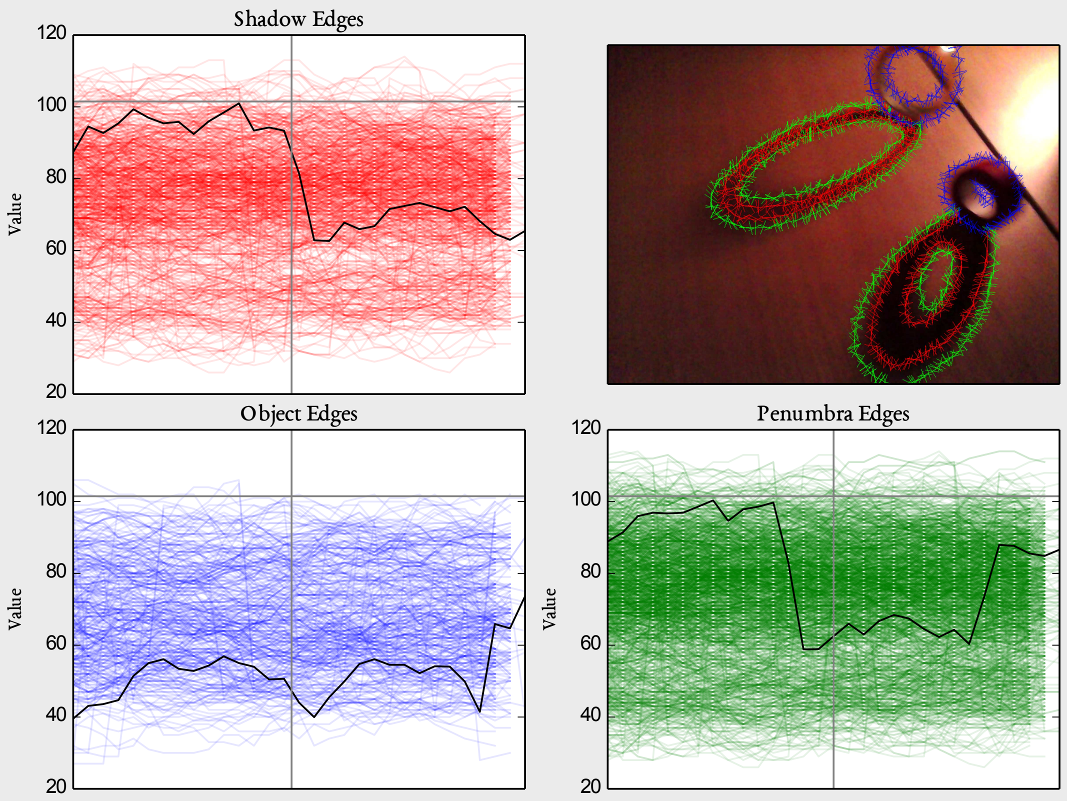


Figure 3.7: Changes in value across different classes of edge in an image from the `kondo1` images. Values sampled from edge normals with a length of 30 pixels, with a minimum distance of 3 pixels between each normal. The vertical grey line is the image's mean intensity, and the horizontal grey line is the centre of the normal.

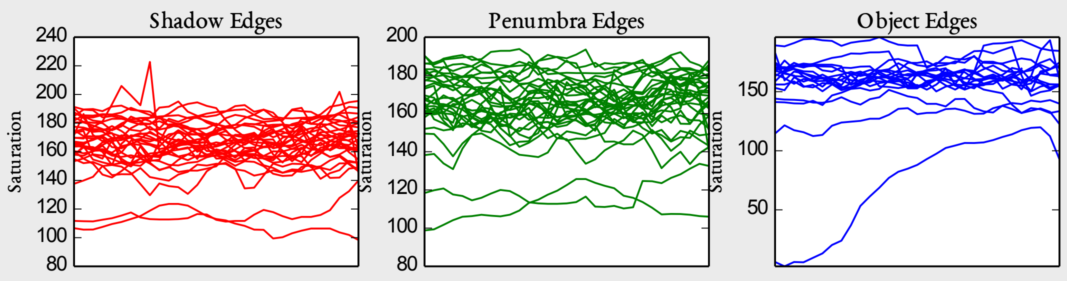


Figure 3.8: Mean saturation values for all edges within the `kondo1` images.

Figure 3.7 shows the change in pixel brightnesses (intensities) across all detected edges in the `kondo1` image set. There is high variability in the brightnesses across the edges, creating very noisy graphs. Calculating the mean pixel values for every horizontal step in the graph makes a pattern emerge however. This is denoted by the black line on the graphs. For shadows and penumbra, it does appear as though there is a sudden drop in pixel intensities across the edges.

There was also assumed to be a change in saturation across edges which would be characteristically different in areas of shadow, penumbra and object. This turned out to be true for the `kondo1` images, as shown in figure 3.8. Shadows have the lowest saturation on average, and penumbra has the most distributed levels of saturation. The objects in the `kondo1` image set are highly-coloured, which is likely why object edges have the highest saturation.

This hypothesis could not be tested on the `pioneer1` or `artificial0` image sets, as they lacked enough ground truth data (appendix figure A.5) and were too simple (appendix figure A.4), respectively.

Adjusting the length of the normals also had a strong effect on the results. With normals set at 10 pixels long, the patterns in the data broke down. This can be seen in appendix figure A.6, where little or no changes across edges can be seen. The patterns discovered across edges using the value and saturation channels at normal length 30 also break down at longer lengths – appendix figure A.7 shows the results of sampling with normals that are 50 pixels long.

3.5 Learning algorithms evaluation

The `kondo1` images had different characteristics between the three different edge types. The hypothesis arising from this was that, given a good set of features extracted from all of these edges, a learning algorithm could be chosen which would correctly classify new examples with reasonable accuracy.

Features were extracted from the normals of detected edges in the `kondo1` image set. Normals that were at least 3 pixels apart were selected. This generated 3MB of compressed CSV-format data. This data contained 51,675 training examples, with a roughly 1-to-1 ratio between negative examples (normals classed as 'in background') and positive examples (normals classed as 'in shadow').

3.5.1 Attribute selection

Before evaluating any of the learning algorithms, Weka was used to evaluate the attributes (features) to get an initial idea of which features were useful. Information Gain Ratio was chosen as an Attribute Evaluator, and attributes were ranked by their entropy best-first (using Weka's "Ranker" search method).

Table 3.3 shows that even the best 10 features are very low in entropy (informational content), indicating that patterns in the data may not be found by any learning algorithms tested later.

The attributes ranked "best" were also unexpected – all relating to the colour channels. It was predicted that the attributes ranked the most highly would be features relating to the brightness (HSV 'Value' channel) of pixels across edges, as this is where patterns emerged during the image analysis.

Gain Ratio	Attribute
0.004202	edge2_blue_mean
0.004129	edge1_sat_mean
0.004114	edge1_blue_min
0.004042	edge1_sat_max
0.003566	edge1_blue_mean
0.003549	edge1_green_min
0.003533	edge1_green_mean
0.00332	edge1_red_mean
0.003299	edge2_green_mean
0.003258	edge1_red_max

Table 3.3: Best 10 features according to the Information Gain Ratio attribute evaluator in Weka.

3.6 Random Forest evaluation

The "Random Forest" learning algorithm was tested first, as it was the quickest to train. It was trained using Weka's default parameters – Random Forests of 10 trees each, each tree testing 6 randomly-chosen features.

The algorithm was trained and tested 10 times. The data was divided into 10 sets of instances. Each test took 9 sets at random for training, and the remaining 10% of instances were used to test (this is known as *10-fold cross-validation*[3].

Random Forests performed slightly better than chance, with a high True Positive Rate for the shadow class (0.739) but also a high False Positive Rate (0.630). This indicates that, with the data it has been trained on, the Random Forest with 10 trees would be likely to over-classify new edges as shadow edges. This is verified in the confusion matrix 3.4.

[3]`http://www.cs.cmu.edu/~schneide/tut5/node42.html`

	Classified as Shadow	Classified as Background
Actually Shadow	**20708**	7298
Actually Background	14908	**8761**

Table 3.4: Confusion Matrix showing what number of instances were correctly and incorrectly classified as each class.

3.7 Neural Network evaluation

The Neural Network tested was a back-propagating network with multiple hidden layers – otherwise known as a *Multi-Layer Perceptron* (MLP). This was tested using the default parameters supplied by Weka – one hidden layer, a learning rate of 0.3 and a momentum of 0.2, and 500 training iterations.

This was also evaluated using 10-fold cross-validation. Similarly to Random Forests it had a low rate of accuracy, as it over-classified the test examples as shadow. The confusion matrix and resulting True Positive and False Positive rates were nearly identical to those of Random Forests.

3.8 Support Vector Machine evaluation

The SVM was tested using the default parameters it was supplied with: a "radial basis" kernel, C-SVC, with a cost ('slack variable') of 1. The SVM implementation tested in Weka was provided by LibSVM[4]. It has a wide range of adjustable parameters. There was not enough time to investigate what these parameters did, so they were left at their defaults.

Out of the three algorithms, the SVM took the longest to train and was also the most memory-intensive. Roughly 10 hours were spent waiting for the SVM to train multiple times in the cross-validation environment.

The results show a complete failure of the SVM to classify anything as background, classifying all test data as shadow. This can be seen in the Confusion Matrix results for this algorithm (table 3.5).

[4]`http://www.csie.ntu.edu.tw/~cjlin/libsvm/`

	Classified as Shadow	Classified as Background
Actually Shadow	**28006**	0
Actually Background	23669	0

Table 3.5: Confusion Matrix for the Support Vector Machine.

3.9 Comparison of the algorithms

Algorithm	Correct	Incorrect	Relative Error	Time Taken
Forests	57.0276%	42.9724%	94.5247%	2.93 seconds
MLP	56.9811%	43.0189%	96.457%	3 minutes
SVM	54.1964%	45.8036%	92.257%	51.8 minutes

Table 3.6: Performance of the three learning algorithms. Shows correct and incorrect classification of instances as a percentage. "Time Taken" is the amount of time it takes to train the algorithm once (for a single iteration out of the 10 iterations done in the 10-fold cross-validation).

The results in table 3.6 show that all the algorithms performed poorly at predicting the classes of new examples. The rate of correctly identified instances being slightly over 50% may be attributed to the fact that there is a roughly 50/50 split between shadow and background instances.

Chapter 4

Conclusion and Evaluation

In this chapter, the previous results section is summarized and discussed. This is followed by a more informal evaluation of the project as a whole.

4.1 Discussion of the results

All the shadow detection methods performed fairly poorly at their task, performing slightly better than chance. There are many reasons why this could be. The first three (fixed thresholding, Otsu's thresholding and the sliding-window chromacity based approach) were very simple and contained a lot of fixed parameters for which the best values had to be empirically derived. It was shown that these algorithms did not generalize well, varying greatly in their performance depending on the dataset used.

Simply thresholding the brightness of an image could be a good way of detecting shadows as a "first pass"; it is likely to pick up a lot of false positives (dark objects) but is also likely to pick up a lot of true positives also, assuming a good threshold is found. A good way of automatically finding the best threshold is needed. Otsu's method of determining the best threshold based on the image histogram proved not to be fit for this purpose. A better method of automatic thresholding could make use of the fact that there are multiple images in sequence and use it to determine the general lighting conditions of a scene – in the images captured in this project, shadows are generally within a specific range of values and saturations, due to being cast on a single surface. Anything dark but outside the ranges could be considered an object instead.

The analysis of edges showed characteristic differences between shadow, penumbra and object edges in the value and saturation channels of the HSV space. This is promising and, given the past work by Lalonde et al., could mean that an approach that uses suitable machine learning techniques would have a high probability of detecting shadows in this project's images.

One does not simply dive into using Machine Learning algorithms, as this project did, however. The two weeks given to thinking of a method of extracting features, implementing it, and evaluating the data in Weka, was too short. Not enough time was given for background reading, meaning that the best parameters for each of the algorithms were not fully understood; how these algorithms perform on specific datasets can vary greatly depending on how they are configured.

The features extracted from the images may have been an important factor when the algorithms failed to pick out any interesting patterns in the data – it was very hard to know whether the right features were being used. Possibly taking the minimum, maximum and mean values of colour channels from edges in the images was not the right approach, or using normals instead of oriented Gaussian filters was the wrong thing to do. Certainly, Lalonde's methods were shown to work very well in their method, so perhaps this was an implementation issue.

Had there been more time, a "grid-based" approach was to be tested, which divided the images into a uniform grid of $N \times N$ pixel squares and used those squares for getting features instead of edge normals. This simpler approach would have less problems with detecting the correct edges. This was partially implemented but time ran out before it could be finished.

Noise was also an issue throughout the project: even after efforts to reduce sensor noise in the images via proprietary software and later a strong bilateral filter, the natural images still contained a lot of noise – the webcam used for the Kondo images had sensor noise in the form of blotchy colours which were too large to be filtered out without also removing a lot of the information encoded in the images. This could have contributed to the poor quality of the features extracted, and would explain why features based on colour were chosen as the best attributes by Weka.

The ground truth images were also difficult to create and could only give a rough estimation of the correctness of an algorithm's output. Differentiating between areas of shadow and penumbra was difficult to do with simply visual estimation. This was considered acceptable for ground-truthing as the output of the algorithms were compared against the ground truth at the per-pixel level, but the inaccurate ground truth images were also used during the feature extraction to label edges – this may also be a reason why the learning algorithms trained on the data so poorly.

This project has shown that detecting shadows in moving-camera images is very challenging. The "state-of-the-art" methods which exist work well for specific environments, such as ground shadows outdoors (Lalonde et al.) or from fixed cameras where the background can be subtracted (Shadow Survey). They did not function well on this project's type of imagery.

Future work could consider combining the approaches tested into a more accurate method – thresholding is a good way of getting an initial shadow mask. A thresholded binary image could also be used as input to an edge detector, reducing the number of edges that an edge classifying method has to process.

Given more time and better camera equipment, it would have been good to capture some better data-sets: Filming a scene filled with simple objects in direct sunlight using higher-resolution video cameras would likely have resulted in natural images with less noise. Having more other people spend time producing more accurate ground truth images could also improve things –

using a service such as LabelMe[1], the ground truths could be crowd-sourced.

4.2 Self-Evaluation

4.2.1 What I would do differently

Having done very little with vision before this project, at the beginning I wasn't sure what to expect. Were I to do this project again I would implement the shadow detection methods in reverse – focusing first on the machine learning elements, and then finding ways of improving the performance of the algorithms by reducing the features through thresholding and noise reduction.

I would also get some people to help me with the ground truthing – 2^n eyes are better than 2. The time it took to ground truth 32 of the Kondo images (roughly 6 hours split over 2 evenings) made me decide against trying to capture more data in a brighter environment (to get images with less sensor noise), as the ground truthing would take too long. Having other people help me with the ground truthing was a suggestion from my mid-project demo feedback. Writing my own crowd-sourcing web interface for ground truthing my images wouldn't actually be too difficult, but would have taken too much time at that point in the project.

4.2.2 Design and Planning

My code was written in a very "I need this right now" way. A lot of my Python code lacked any form of design or planning beforehand. I wrote what I needed when I needed it for the experiments I was doing. Due to this, there are some Python scripts with very long-winded functions. Had this project been focused more on developing software than research, I would have separated out this code into smaller functions across multiple classes and files, creating proper Python modules for things. As the focus of this project was on research, I think this is acceptable. I have tried to keep to the Python style guide[2] where possible. I feel as though my C++ code could be a lot better written; once I discovered compiler macros I used them frugally to hide the boiler-plate code which was the same across my shadow methods. I now realize that this is obfuscating the code and would make it harder for other people to debug.

Looking back at the Project Specification I wrote at the beginning of the project, I have certainly produced all of the deliverables mentioned in it. I have followed the project outline fairly dilligently, although what methods I implemented did change. I wanted to experiment with methods of detecting shadow via textures (a shadow cast on a surface should have the same texture as the surface, only darker), but spent more time than expected on the simpler colour-based method, so texture comparison ended up as a hastily-implemented afterthought

[1]`http://labelme.csail.mit.edu/Release3.0/`

[2]`http://docs.python-guide.org/en/latest/writing/style/`

in the feature extraction code of my edge-based method (I used Local Binary Patterns; they didn't tell me anything useful and I likely implemented wrong).

I think the initial design that I did was sensible, and the test harness, image analysis and ground truth creation tools have proven to be very helpful. So whilst I failed to implement a good detection algorithm, I now have a framework in which my results can be recreated very easily, and new methods added to the test harness very quickly. The test harness is deliberately generic and could be used for any experiments involving sets of input images and ground truth images. This could be useful – at the time of writing, I can only find two other generic test harnesses for vision projects: HATE[3] and MVTH[4]. MVTH appears to be lacking documentation and according to their website, HATE is essentially vapourware (its "new version" slated for release in February 2004).

4.2.3 Implementation issues

The test harness is not without fault. Its error handling could be improved, and there are a few problems with its implementation which I haven't been able to fix. The commands it generates are very long-winded, as it passes multiple lists of files to each command. This has been fine in practise, as the size of the command-line argument (`argv`) buffers used on modern UNIX systems are very large (roughly 2MB on my Linux system) – but this would not scale well when there are thousands of images with very long file paths. It works fine for my purposes, so time wasn't taken to fix this. The fix would be to pass directory paths to programs the harness ran, rather than explicitly pass them every input and ground truth image.

4.2.4 Time management

My supervisor Hannah and I planned how I should spend time on different parts of the project at the beginning, as seen in the Gantt Chart we initially sketched out (appendix figure A.8). I managed to keep to that mostly, although more time was spent implementing each detection method than was hoped.

I should have anticipated this – time was wasted on debugging strange bugs in C++ code which turned out to be because CMake was linking libraries with my code wrong – putting my code through GDB and Valgrind was showing my code breaking in places that weren't in my code (the statically linked OpenCV libraries, which had no debugging information themselves).

Overall I think my time management was reasonable. Having group meetings and individual meetings with Hannah once a week helped keep me on track. Because the results of my implementation have been poor I feel as though I could have done a lot more, but have been reminded that I did a lot of experiments – I am very close to the 20,000 word limit writing

[3]`http://peipa.essex.ac.uk/hate/index.html`

[4]`http://gna.org/projects/mvth`

about them in what I consider is still quite a brief way.

4.2.5 Challenges

The most difficult part of this project was probably this report. Remembering everything that I have done in the project and talking about it in academic passive third-person was hard. Keeping a blog throughout the project has helped – although some weeks of the project are missing blog posts. If I were to do the project again I would keep a diary full of shorter notes so I could remember exactly what I did and in what order.

The second most challenging part was definitely the experiments with edges and normals and extracting features from normals. There were a lot of edges, which created a lot of data. There were a lot of numbers and it was very difficult to make sense of it all. Weka was meant to make sense of the data and pick out good features, but I couldn't understand the outputs of Weka enough to find out why the extracted features were so bad.

4.2.6 What I've learnt

It is easy for me to be critical of this project's results. It was disappointing that by the end of the project I didn't have a shadow detection method implemented that works better than simple thresholding. That said, I have definitely gained a lot of useful, practical knowledge when it comes to actually implementing computer vision algorithms. I now have good working knowledge of some fundamental computer vision tools, such as HLS colour spaces, morphological filters, Gaussian kernels, different algorithms for detecting edges – things that are core components to more complicated image processing chains.

This was my first real experience with computer vision projects, as well as my first experience with using machine learning algorithms on "real world" data (as opposed to studying them in theory and testing them on the conveniently prepared datasets of the Stanford class online). It was also my first experience using C++, and I hadn't touched C since an assignment in my second year. A lot of learning new tools happened in this project, which is great.

Overall I am pleased with what I've achieved with this project, despite the disappointing results from the experiments. I was warned straight from the start that this would be a challenging project, being a vision project using real-world images. I have definitely gained some insight into just how challenging real-world imagery is when you want to do computer vision things with it.

Appendices

Appendix A

Additional Figures

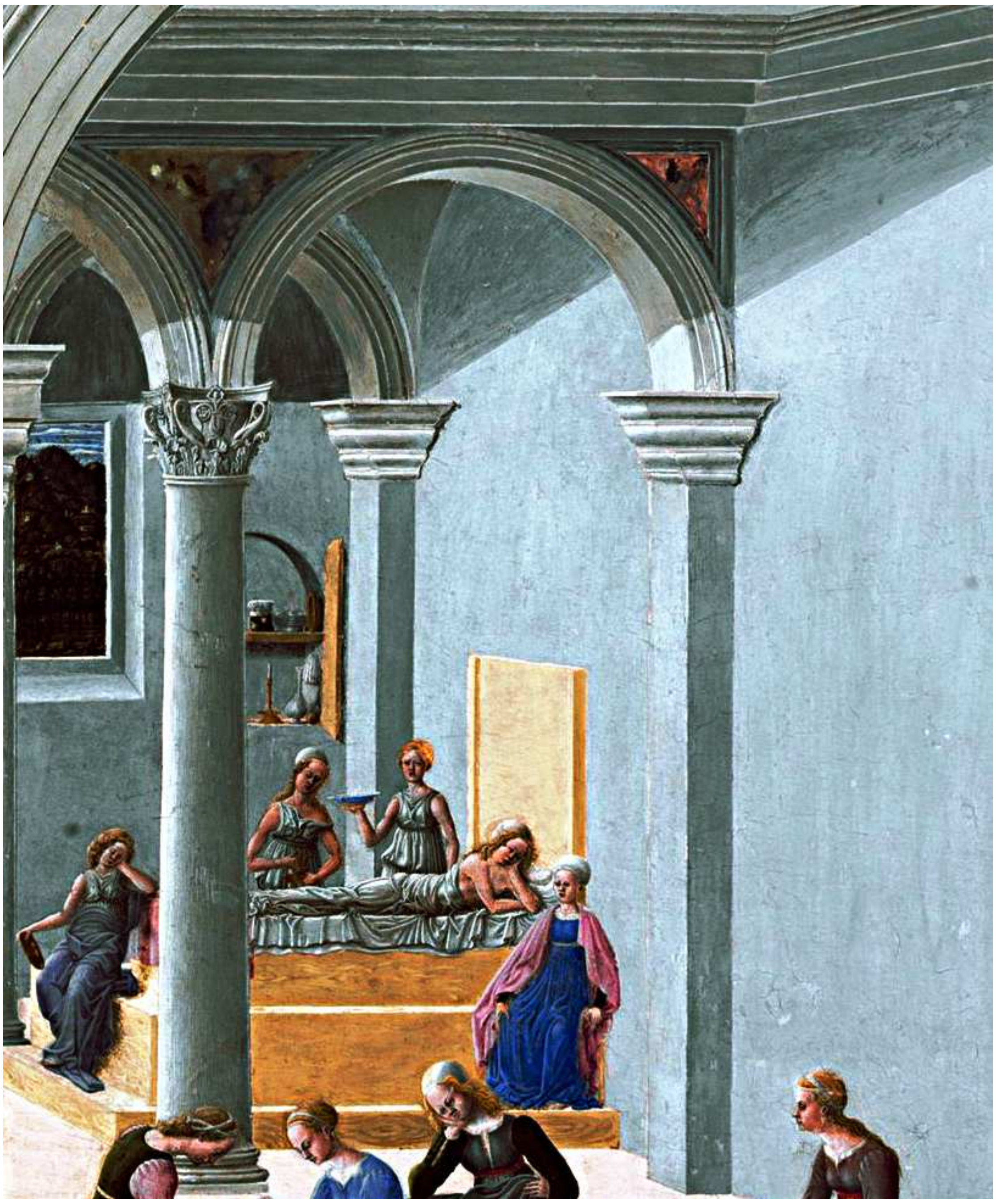


Figure A.1: Excerpt of the Carnevale painting, showing shadows being cast upwards.

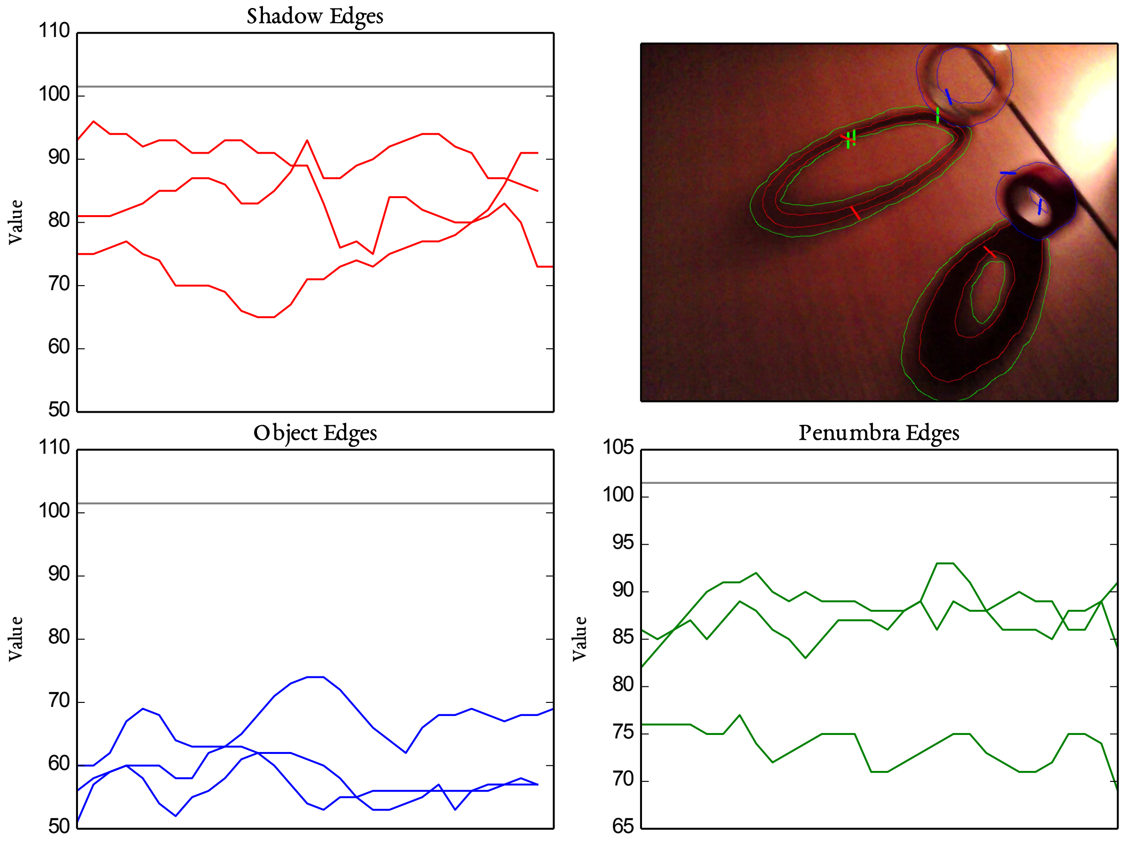


Figure A.2: Three random edge normals chosen per class. This random selection exhibits differences between edge classes better than the whole set of edges.

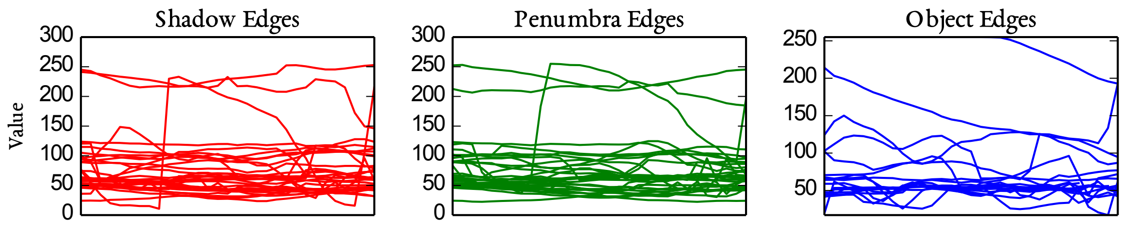


Figure A.3: Mean normal values for every image within the Kondo dataset (total of 32 images).

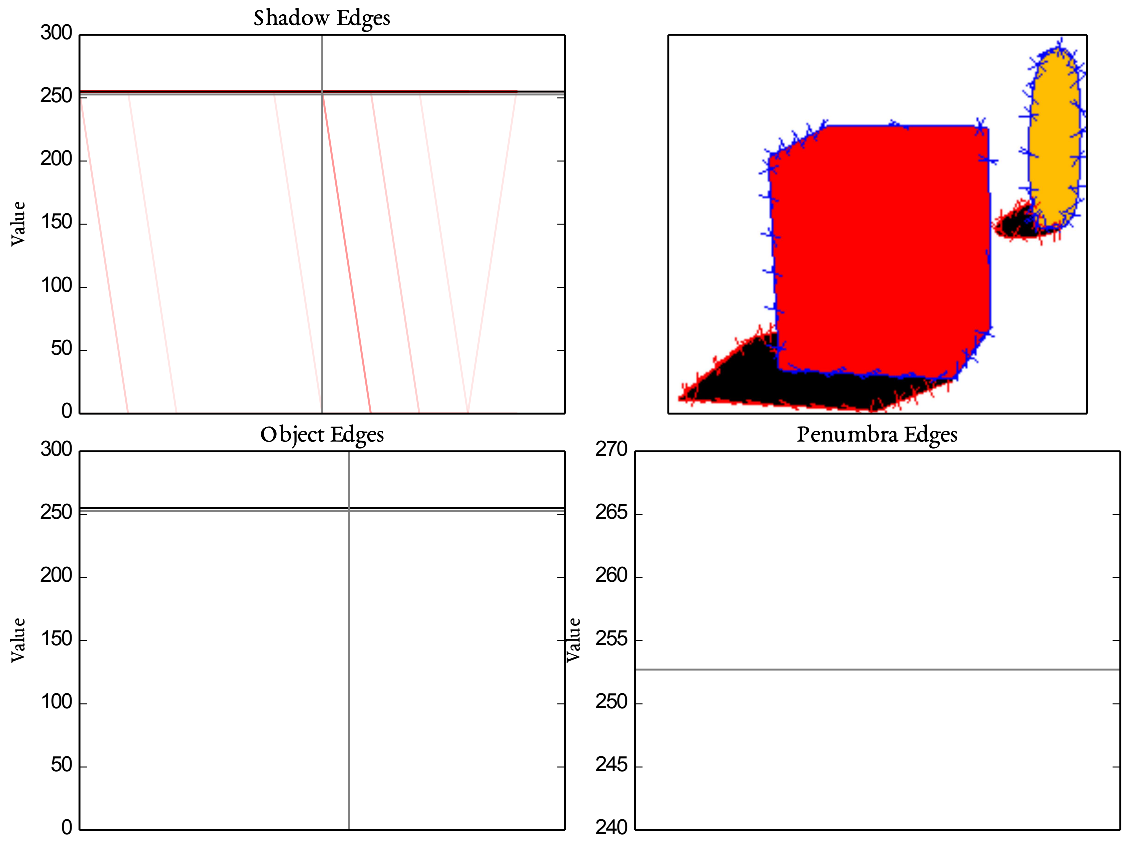


Figure A.4: Normal values for an image in the `artificial0` dataset. On this simple artificial data, edges are too sharp for any meaningful patterns to be discerned.

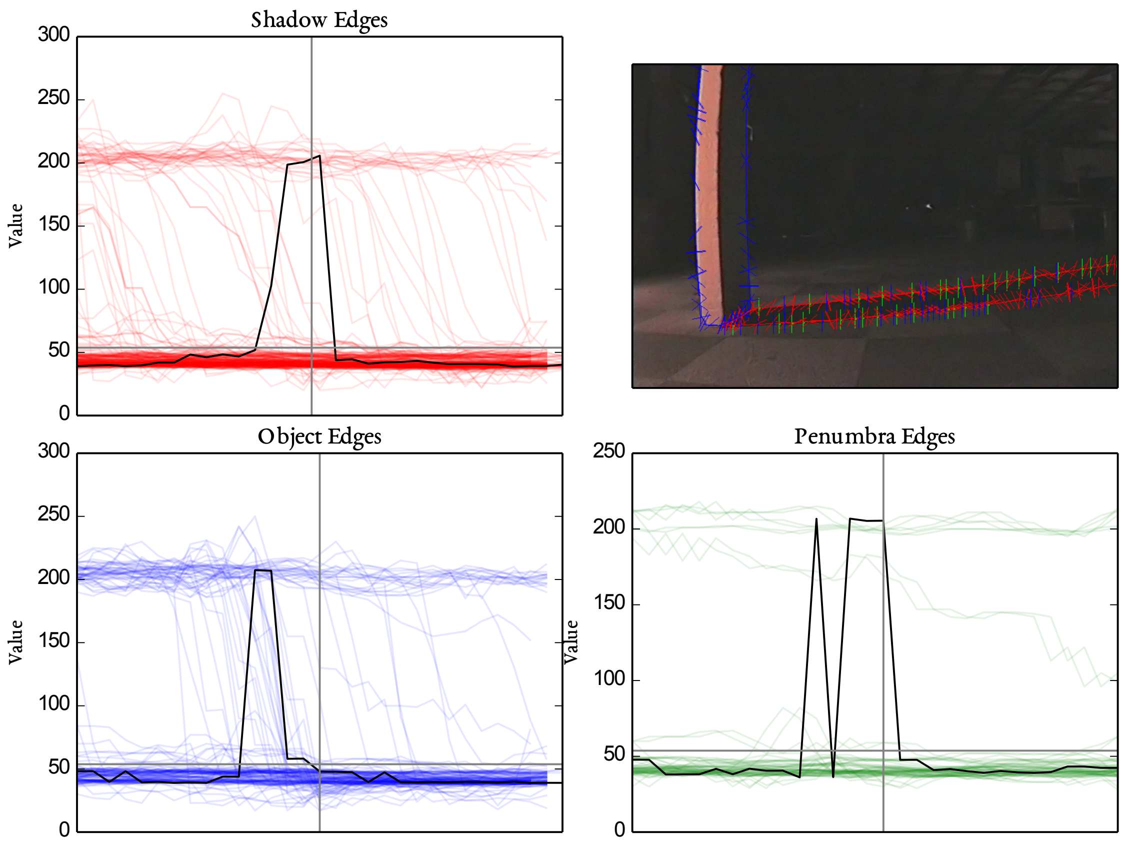


Figure A.5: Normal values for an image in the `pioneer2` dataset (edge normal length of 30).

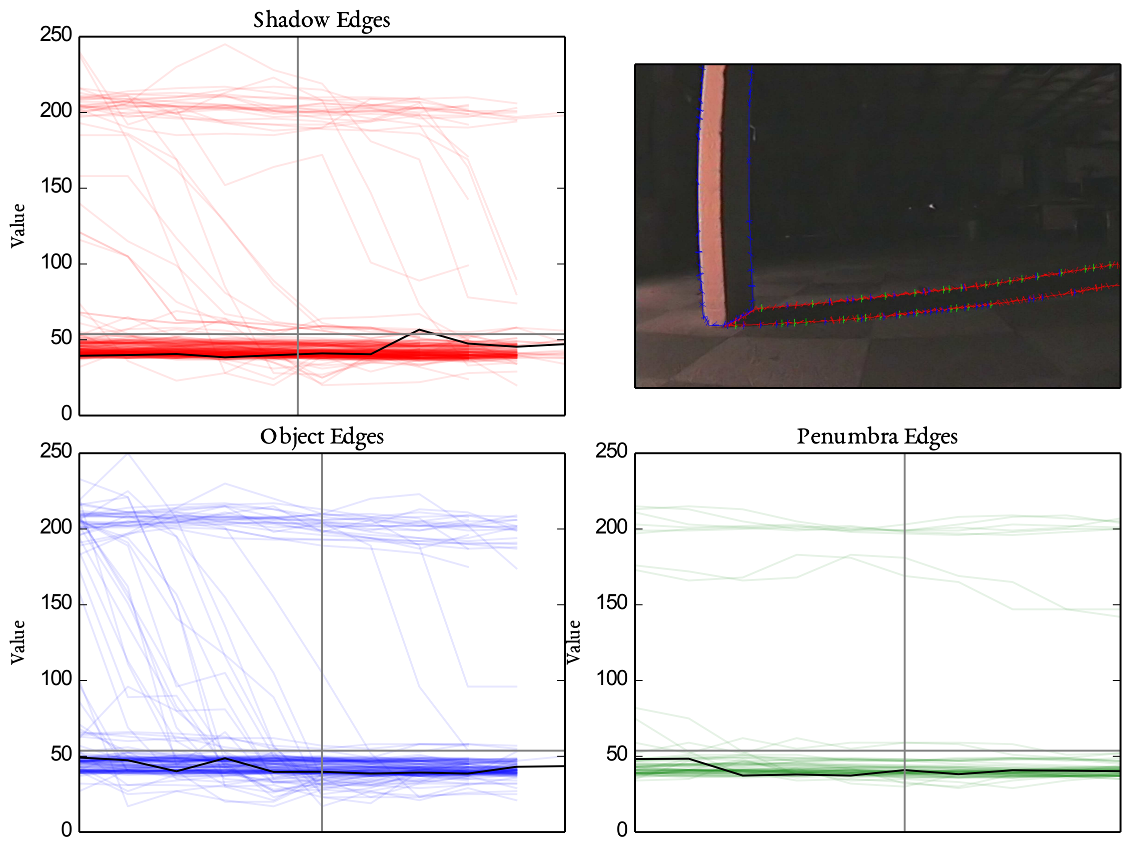


Figure A.6: Normal values for an image in the `pioneer2` dataset (edge normal length of 10).

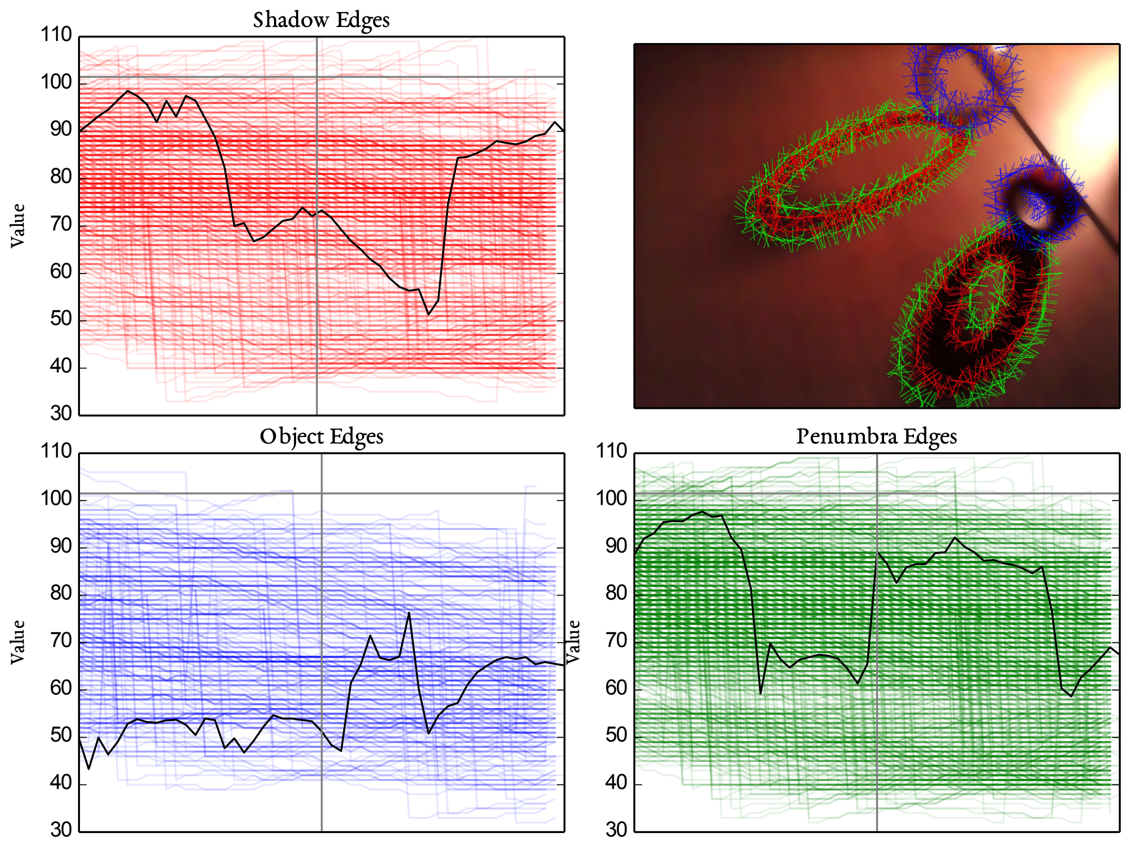
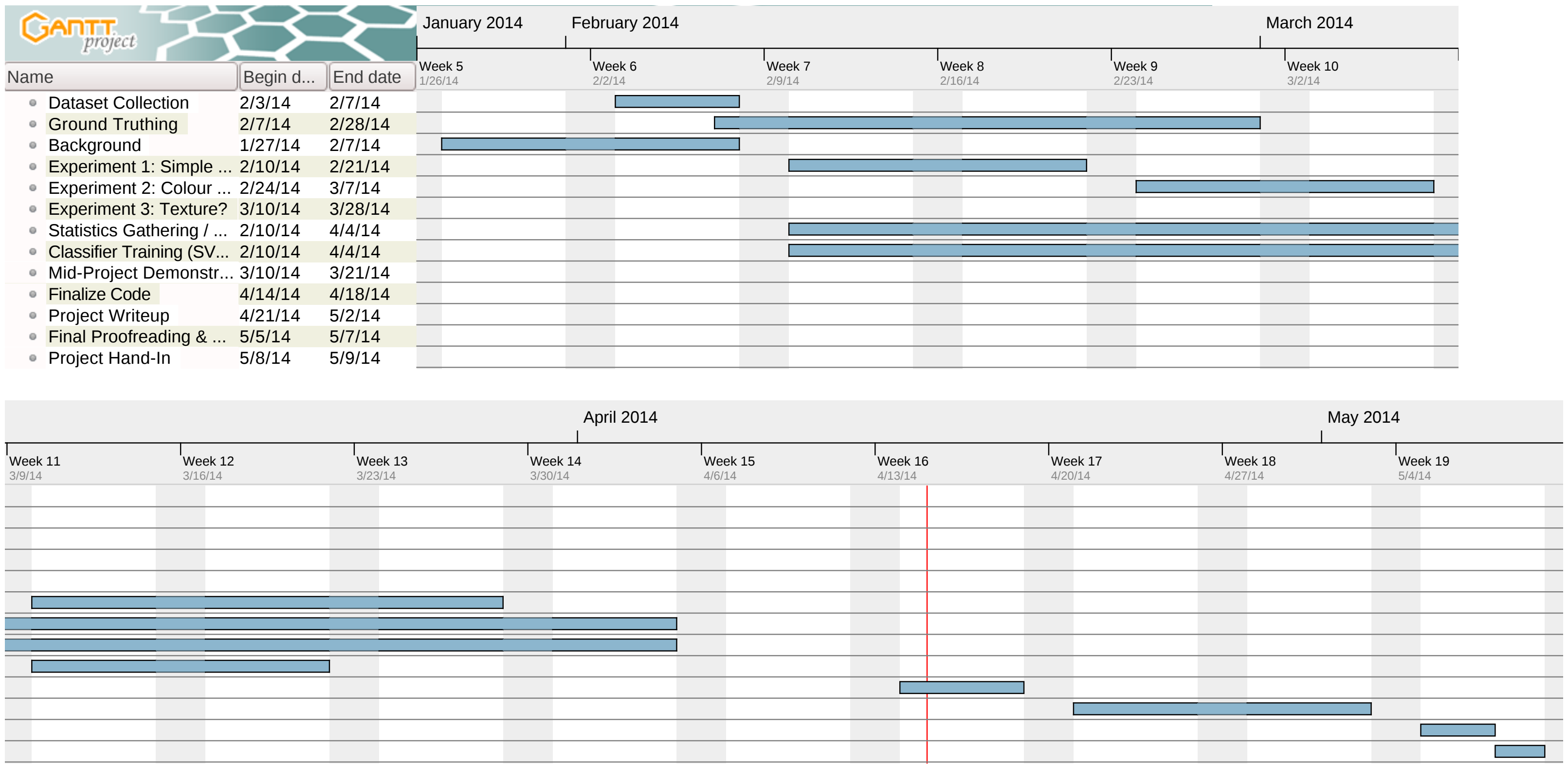


Figure A.7: Normal values for an image in the `kondo1` dataset (edge normal length of 50).



Name	Begin d...	End date
Dataset Collection	2/3/14	2/7/14
Ground Truthing	2/7/14	2/28/14
Background	1/27/14	2/7/14
Experiment 1: Simple ...	2/10/14	2/21/14
Experiment 2: Colour ...	2/24/14	3/7/14
Experiment 3: Texture?	3/10/14	3/28/14
Statistics Gathering / ...	2/10/14	4/4/14
Classifier Training (SV...	2/10/14	4/4/14
Mid-Project Demonstr...	3/10/14	3/21/14
Finalize Code	4/14/14	4/18/14
Project Writeup	4/21/14	5/2/14
Final Proofreading & ...	5/5/14	5/7/14
Project Hand-In	5/8/14	5/9/14

Figure A.8: Initial Gantt Chart for the project.

```
# Directory to save all the output of the test harness to.
# Also used as the working directory -- input images are copied here before any
# programs are ran on them, to prevent the original data accidentally being
# overwritten.
#
# Special variables:
# {image_set}: The name of the image set being tested
# {chain}: The processing chain being ran
# {tunables}: The parameters for every program in the current chain. Expands to
# a large number of subdirectories.
scratch: '/tmp/test_harness/{image_set}/{chain}/{tunables}'

# Defaults for programs
program_defaults:
    path: '~/Dissertation/code/detection_methods/build/'  # where to find it
    argument_format: '{parameters} --input {input_images}'  # argument format

# Here we define some chains
chains:
    simple_test:  # name of the chain
        simple_threshold:  # name of the program (also its filename)
            shadow_threshold: [0, 10, 25, 50, 100]  # --shadow_threshold param
            object_threshold: [255, 200, 150]  # --object_threshold param

# And also some image sets
image_sets:
    pioneer1: '~/robot_images/pioneer2/*.jpg'

ground_truths:
    pioneer1: '~/robot_images/ground_truth/pioneer2/*gt.png'
```

Figure A.9: Initial configuration file used to develop first prototype of test harness.

```
Statistics s = Statistics();  // plain old data object class
uchar iv, gt;  // uchar = unsigned 8 bit integer
for(int i=0; i<image.rows; i++) {
    for(int j=0; j<image.cols; j++) {
        iv = image.at<uchar>(i, j);  // pixel value from input image
        gt = ground_truth.at<uchar>(i, j);  // pixel value from ground truth

        // If input pixel == ground truth pixel, and both == shadow value (0),
        // count as true positive
        if(iv == shadow_value && gt == shadow_value)
            s.shadows.true_positive += 1;

        // If input pixel and ground truth pixel both != shadow,
        // count as true negative
        if(iv != shadow_value && gt != shadow_value)
            s.shadows.true_negative += 1;

        // If input pixel == shadow, and ground truth != shadow,
        // that's a false positive
        if(iv == shadow_value && gt != shadow_value)
            s.shadows.false_positive += 1;

        // Conversely, if input value != shadow, but ground truth == shadow,
        // that's a false negative.
        if(iv != shadow_value && gt == shadow_value)
            s.shadows.false_negative += 1;
    }
}
```

Figure A.10: Code used to count true/false positive/negative shadow pixels in images

```
$N$ = number of CPUs $-1$
while there are jobs to run do
    while pool size $>=$ N do
        for all jobs in pool do
            if job has finished then
                get job's return code
                if return code = 0 then
                    success $+= 1$
                else
                    error $+= 1$
                end if
                remove job from pool
            end if
        end for
    end while
    add new job to pool
    run job
end while
```

Figure A.11: Process pooling algorithm

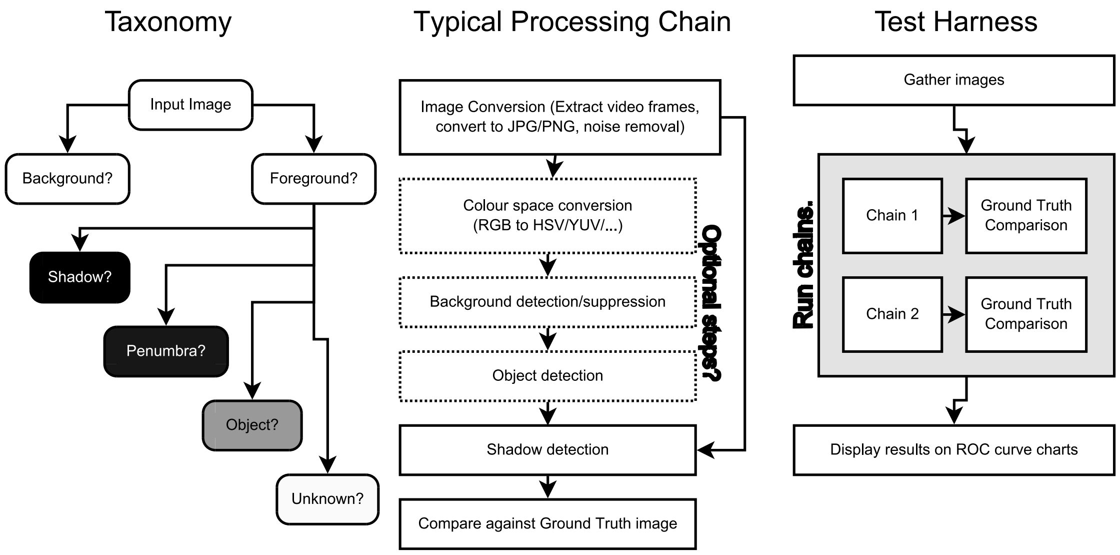


Figure A.12: Initial design: Diagrams on which the test harness software was based.

Appendix B

Third-Party Code and Libraries

All of my image processing code, in both C++ and Python, makes extensive use of the OpenCV libraries (`http://www.opencv.org`). OpenCV is open sourced under the BSD license. The library was linked dynamically and used without modification.

My C++ programs also use elements of the Boost library for command line argument parsing and general utility functions (`http://www.boost.org`). Boost is open sourced under their own license, the Boost Software License. This is a very permissive license similar to the BSD/MIT licenses.

Within my Python code, I have used the following additional libraries:

- numpy (`http://www.numpy.org`): Used for mathematical functions and multidimensional arrays. Is open sourced under the BSD license.
- matplotlib (`http://www.matplotlib.org`): Used for plotting graphs and charts of data from numpy arrays. The graphs and charts in this paper are outputs from matplotlib. It is open source under its own license, similar to the Python Software Foundation License, and compatible with BSD licenses.
- PyYAML (`http://www.pyyaml.org`): Used by the test harness scripts, for parsing of configuration files. PyYAML is under the MIT license.
- Pygame (`http://www.pygame.org`): Used for the ground truth painting tool. Provided graphics and image libraries suitable for the rapid development of a simple painting tool. Pygame is released under the GNU Lesser GPL (LGPL) license.

For data collection using the Pioneer robot, the ARIA SDK
(`http://robots.mobilerobots.com/wiki/ARIA`)
was used to gather telemetry data and control the robot's movements, and imalib by Frédérick Labrosse (`ffl@aber.ac.uk`) was used for image capture. The ARIA SDK is under the

GNU GPL license. imalib is internal to Aberystwyth University and is not distributed with a license.

Control of the Kondo robot was done using libkondo4
(`https://bitbucket.org/vo/libkondo4/wiki/Home`)
which uses the Apache License, version 2.0.

Appendix C

The MIT License (MIT)

All of the code written by myself in this project is under the MIT License (MIT), unless otherwise specified.

Annotated Bibliography

[1] E. H. Adelson, “Checker shadow illusion.” [Online]. Available: http://web.mit.edu/persci/people/adelson/checkershadow_illusion.html

[2] ——, “Lightness perception and lightness illusion,” 1999. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.23.2182

> The paper behind the “Checker-Shadow Illusion” image. Shows how object shading can induce the feeling of depth in an otherwise 2D image.

[3] A. Sanin, C. Sanderson, and B. C. Lovell, “Shadow detection: A survey and comparative evaluation of recent methods,” *Pattern Recognition*, vol. 45, no. 4, pp. 1684–1695, Apr. 2013. [Online]. Available: http://dx.doi.org/10.1016/j.patcog.2011.10.001

> This paper was one of the main sources of background reading for my project. It contains a list of existing shadow detection methods and evaluates their performance (on fixed-camera scenes).

[4] D. Kersten, P. Mamassian, and D. C. Knill, “Moving cast shadows induce apparent motion in depth.” *Perception*, vol. 26, no. 2, pp. 171–192, 1997. [Online]. Available: http://view.ncbi.nlm.nih.gov/pubmed/9274752

> Another paper making use of strong optical illusions to demonstrate how shadow motion is used in the human visual system for depth perception.

[5] P. Cavanagh and Y. G. Leclerc, “Shape from shadows,” in *Journal of Experimental Psychology: Human Perception & Performance*, 1990, pp. 3–27. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.309.6891

> Background reading: A study into the human visual system. Quoted in the first chapter of this report.

[6] P. Mamassian, D. C. Knill, and D. Kersten, *The perception of cast shadows*. Elsevier Science„ Aug. 1998, vol. 2, no. 8, pp. 288–295. [Online]. Available: http://linkinghub.elsevier.com/retrieve/pii/S1364661398012042

This paper demonstrates how the human visual system makes use of shadows to infer depth. It contains some images showing shadows in motion and the strong effects they have on scene perception.

[7] V. Fenelon, P. Santos, H. Dee, and F. Cozman, "Reasoning about shadows in a mobile robot environment," *Applied Intelligence*, vol. 38, no. 4, pp. 553–565, Sept. 2013. [Online]. Available: http://dx.doi.org/10.1007/s10489-012-0385-5

One of the original pieces of background reading for this project. This describes a higher-level framework, PQRS, that could make use of the low-level binary shadow masks that this kind of project produces.

[8] *On the separation of luminance from colour in images*, 2005. [Online]. Available: http://cadair.aber.ac.uk/dspace/handle/2160/12906

Explains some colour spaces in detail and compares their performance at separating luminance (brightness) from colour.

[9] M. Tkalcic and J. Tasic, "Colour spaces: perceptual, historical and applicational background, EUROCON 2003, computer as a tool," *IEEE Region. v8 i1*, pp. 304–308. [Online]. Available: http://ldos.fe.uni-lj.si/docs/documents/20030929092037_markot.pdf

Introduces some colour spaces commonly used in image processing, and explains their underlying concepts and shortcomings.

[10] M. Sezgin and B. Sankur, "Survey over image thresholding techniques and quantitative performance evaluation," *Journal of Electronic Imaging*, vol. 13, no. 1, pp. 146–168, 2004. [Online]. Available: http://dx.doi.org/10.1117/1.1631315

This paper provides thorough deconstructions of a number of different thresholding methods. Whilst I decided to use the simple histogram thresholding available in OpenCV, it was interesting to see what other approaches were available.

[11] "A threshold selection method from Gray-Level histograms," *Systems, Man and Cybernetics, IEEE Transactions on*, vol. 9, no. 1, pp. 62–66, Jan. 1979. [Online]. Available: http://dx.doi.org/10.1109/tsmc.1979.4310076

This is the original paper describing Otsu's thresholding method.

[12] J. Canny, "A computational approach to edge detection," *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, vol. PAMI-8, no. 6, pp. 679–698, 1986. [Online]. Available: http://dx.doi.org/10.1109/TPAMI.1986.4767851

The original paper on Canny Edge Detection, which explains in detail how the algorithm works, including how they mitigate noise using Gaussian smoothing and multi-scale image operators.

[13] C. Tomasi and R. Manduchi, "Bilateral filtering for gray and color images," in *Computer Vision, 1998. Sixth International Conference on*. IEEE, Jan. 1998, pp. 839–846. [Online]. Available: http://dx.doi.org/10.1109/iccv.1998.710815

This paper describes the Bilateral Filtering algorithm in depth, giving examples of its performance on various natural imagery.

[14] S. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach (2nd Edition)*, 2nd ed., ser. Prentice Hall series in artificial intelligence. Prentice Hall, Dec. 2002. [Online]. Available: http://www.amazon.com/exec/obidos/redirect?tag=citeulike07-20&path=ASIN/0137903952 0137903952.

This book was used towards the end of the project to help understand key machine learning concepts such as features/examples, and also as a reference on neural networks. Being an old edition, however, it does lack any information on the newer algorithms this project has used (Random Forests and Support Vector Machines). Despite this, it is a very comprehensive manual on Machine Learning.

[15] C. Burges, "A tutorial on support vector machines for pattern recognition," vol. 2, no. 2, pp. 121–167, 1998. [Online]. Available: http://dx.doi.org/10.1023/a%253a1009715923555

This paper aims to teach the reader everything they need to know about Support Vector Machines, including how they circumvent related ML problems such as "the curse of dimensionality".

[16] D. Pomerleau, "Neural network vision for robot driving," in *Intelligent Unmanned Ground Vehicles*, ser. The Springer International Series in Engineering and Computer Science, M. Hebert, C. Thorpe, and A. Stentz, Eds. Springer US, 1997, vol. 388, pp. 53–72. [Online]. Available: http://dx.doi.org/10.1007/978-1-4615-6325-9_4

Description of the CMU NavLab project, which successfully trained neural networks to steer a real vehicle on public roads.

[17] L. Breiman, "Random forests," vol. 45, no. 1, pp. 5–32, 2001. [Online]. Available: http://dx.doi.org/10.1023/a%253a1010933404324

The original paper describing Random Forests. Whilst it gives a thorough explanation of how Random Forests work, it is difficult to read without already understanding all of the statistical concepts mentioned relating to decision trees, bagging and ensemble learning, for example.

[18] T. Fawcett, "An introduction to ROC analysis," *Pattern Recogn. Lett.*, vol. 27, no. 8, pp. 861–874, June 2006. [Online]. Available: http://dx.doi.org/10.1016/j.patrec.2005.10.010

Demonstrates how to generate ROC curves and Confusion Matrices, as well as different ways of representing and interpreting them.

[19] E. Hayman and J. O. Eklundh, "Statistical background subtraction for a mobile observer," in *Computer Vision, 2003. Proceedings. Ninth IEEE International Conference on*. IEEE, Oct. 2003, pp. 67–74 vol.1. [Online]. Available: http://dx.doi.org/10.1109/iccv.2003.1238315

> Provides a method of subtracting background pixels from an image when the camera itself is in motion. Relies on objects in the scene also being in motion, so whilst it works for tracking moving objects, I did not attempt to use this approach in my own work as my image sets contained no moving objects (only moving cameras).

[20] A. Bruckstein, R. Holt, I. Katsman, and E. Rivlin, "Head movements for depth perception: Praying mantis versus pigeon," vol. 18, no. 1, pp. 21–42, 2005. [Online]. Available: http://dx.doi.org/10.1023/b%253aauro.0000047302.46654.e3

> This isn't particularly related to my project, but as some background reading whilst I was writing about the Kondo data captures, I looked into depth perception from monocular cameras via head motion, and I found this interesting.

[21] R. Cucchiara, C. Grana, M. Piccardi, and A. Prati, "Detecting moving objects, ghosts, and shadows in video streams," *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, vol. 25, no. 10, pp. 1337–1342, Oct. 2003. [Online]. Available: http://dx.doi.org/10.1109/tpami.2003.1233909

> This is the original paper upon which the "chromacity" method in my project was based.

[22] S. Suzuki and K. Be, "Topological structural analysis of digitized binary images by border following," *Computer Vision, Graphics, and Image Processing*, vol. 30, no. 1, pp. 32–46, Apr. 1985. [Online]. Available: http://dx.doi.org/10.1016/0734-189x(85)90016-7

> This is the original paper behind the contour-finding implementation used in OpenCV.

[23] J.-F. Lalonde, A. Efros, and S. G. Narasimhan, "Detecting ground shadows in outdoor consumer photographs," in *Computer Vision – ECCV 2010 Lecture Notes in Computer Science*, vol. 6312, 2010. [Online]. Available: http://repository.cmu.edu/robotics/787/

> This paper is what I loosely based my "edge normal features" shadow detection approach upon.

[24] C. Sutton and A. McCallum, "An introduction to conditional random fields," Nov. 2010. [Online]. Available: http://arxiv.org/abs/1011.4088

> This is a very comprehensive tutorial (75 pages long) that introduces conditional random fields in the context of general machine learning. It is also a very complex explanation, requiring other background reading. When searching for

descriptions of CRFs, this paper gave the clearest in its introduction section. As CRFs were not implemented in this project due to time constraints, this paper was only skim-read, but appears to cover most aspects of CRFs.

[25] H. H. Bülthoff, "Optical illusions." [Online]. Available: http://www.kyb.tuebingen.mpg.de/research/dep/bu/more-on-this-topic/optical-illusions.html

This webpage contains three videos which demonstrate very clearly how shadows affect our perceptions of objects in motion.

[26] A. Elqursh and A. Elgammal, "Online moving camera background subtraction," in *Computer Vision – ECCV 2012*, ser. Lecture Notes in Computer Science, A. Fitzgibbon, S. Lazebnik, P. Perona, Y. Sato, and C. Schmid, Eds. Springer Berlin Heidelberg, 2012, vol. 7577, pp. 228–241. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-33783-3_17

This paper documents an impressive method of extracting foreground from moving camera images, using camera trajectory modelling. Since background subtraction was decided against in this project it wasn't looked into much, but the results in the paper look good.

[27] H. M. Dee and P. E. Santos, "The perception and content of cast shadows: An interdisciplinary review," *Spatial Cognition & Computation*, vol. 11, no. 3, pp. 226–253, Aug. 2011. [Online]. Available: http://dx.doi.org/10.1080/13875868.2011.565396

A comprehensive literature review on shadows which covers art, the human vision system and computer vision.