An Exploration of the SIFT Operator

Module number: P00999

Supervisor: Prof. Philip H. S. Torr

Course code: CM79

Jonathan Rihan
Student number: 04073838

September 1, 2005

Dissertation Plan:
An Exploration of the SIFT Operator

Jonathan Rihan

Supervisor: Professor Philip H. S. Torr

September 1, 2005

Abstract

The SIFT operator developed by David Lowe (Lowe 1999, Lowe 2004) is an algorithm for object recognition in images.

This dissertation is an exploration of the SIFT operator, with the goal of identifying and exploring areas of possible improvement. These might be either in performance characteristics of the implementation of the algorithm or general improvements to the stability or robustness of the algorithm in analysing different images and detecting objects.

First the algorithm will be implemented in C++ on a Windows operating system, then once it has been successfully implemented, avenues of improvement will be identified and explored. The areas will be indentified by experimentation and further research during the course of the project.

1 Research

The SIFT algorithm was initially presented in a paper by David Lowe in 1999 (Lowe 1999) and recently summarised (Lowe 2004). It has since been analysed by a number of people with the goal of improving its performance and robustness.

The algorithm has been improved somewhat by Y. Ke and R. Sukthankar (Ke & Sukthankar 2003) by applying Principal Component Analysis instead of using smooth weighted histograms as presented in the original implementation. This has the effect of making keypoints more robust to changes in illumination and angle.

A SIFT implementation is available as MatLab source code, and this could possibly be used to help develop the initial implementation of the algorithm and aid it's migration to C++. This code is available via Standford University's project webpage for their computing module as part of a SIFT project. While the SIFT operator was explored in the project presented by students on that page (Gustavsson, Hui & Turitzin 2004), the focus of the project was changed to that of using SIFT features to find planes in consecutive images part way through and improvements to the SIFT algorithm were not developed.

Some areas of research that may yield improvements to the SIFT algorithm have been identified by my supervisor Professor Philip Torr. The first of these is that of possibly using kernel density estimation (Duong 2001) instead of the histograms used in SIFT. The other area of interest is researching whether or not filtering is necessary (Varma & Zisserman 2003) in the image being analysed.

Other areas of improvement may be identified and explored as the project develops once the algorithm has been implemented.

2 Objectives

- Implement SIFT in C++ on windows
- Identify and explore variations or improvements to the operator

3 Methods

Implement SIFT in C++ on windows

- Analyse the MatLab source code and references (Lowe 1999, Lowe 2004)
- Design program
- Implement design
- Test algorithm is implemented correctly by comparing to MatLab version, and redesign if not. (requires digital camera for building object database)
- Explore real time performance (requires web camera)
- Research methods of analysing real time video stream with SIFT
- Implement real time analysis mode in program
- Look in to implementing PCA-SIFT (Ke & Sukthankar 2003)

Identify and explore variations or improvements to parts of the operator

- Research the use of kernels (Duong 2001) instead of histograms in SIFT
- Research the issue of whether or not filters are required (Varma & Zisserman 2003) for the algorithm
- Research any other methods discovered during the project

Write up dissertation

- Write report
- Proof read draft

4 Resources

- **Web camera** for exploring real time object recognition performances
- **Digital camera** for building databases of object images used to detect objects

5 Schedule

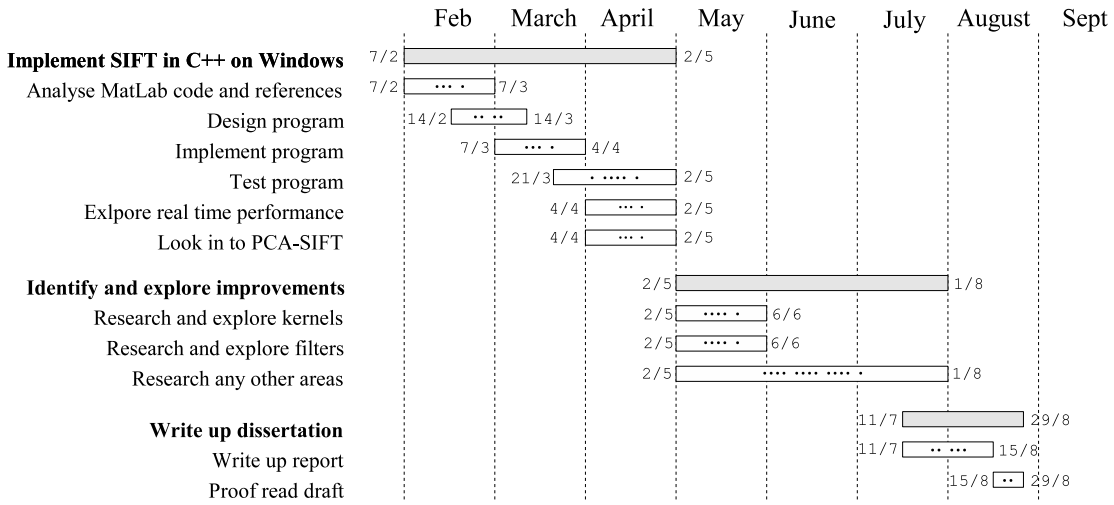

Figure 1: Gantt Chart for project plan

6 Marking Scheme

Evidence of research into the background to the topic: 20%
Analyitical content: 40%
Technical content: 40%

References

Duong, T. (2001), 'An introduction to kernel density estimation',
website: `http://www.maths.uwa.edu.au/~duongt/seminars/intro2kde/`.

Gustavsson, C., Hui, A. & Turitzin, M. (2004), 'Improving SIFT features / finding planes in hallways', website: `http://robots.stanford.edu/cs223b04/project9.html`.

Ke, Y. & Sukthankar, R. (2003), PCA-SIFT: A more distinctive representation for local image descriptors, Technical report IRP-TR-03-15, Intel.

Lowe, D. G. (1999), Object recognition from local scale-invariant features, *in* 'International Conference on Computer Vision', Corfu, Greece, pp. 1150–1157.

Lowe, D. G. (2004), 'Distinctive image features from scale-invariant keypoints', *International Journal of Computer Vision* **60**(2), 91–110.

Varma, M. & Zisserman, A. (2003), Texture classification: Are filter banks necessary?, *in* 'Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition'.

Abstract

The field of computer vision applies numerous algorithms to solve the problems of object recognition and classification. This dissertation explores two of these algorithms and presents a new human pose data set of challenging human poses for use with human detection algorithms.

The first of the algorithms that this dissertation explores is the Scale Invariant Feature Transform (or SIFT) first presented by Lowe (1999). This algorithm analyses an image across Gaussian scale-space and creates descriptors at minima and maxima in the difference-of-Gaussian function of two adjacent scale space images. This dissertation discusses a basic implementation of the SIFT algorithm.

The second algorithm is based on descriptors very similar to those used in SIFT, but distributes them in a dense grid of overlapping descriptor blocks. These blocks are known as Histograms of Oriented Gradients, or HOG (Dalal & Triggs 2005). There are two variants of this descriptor, one circular and the other rectangular in shape called C-HOG and R-HOG respectively. This dissertation tests out an implementation of the HOG algorithm by training with the database used by Dalal & Triggs (2005) and a new pose database to compare the results.

Acknowledgements

Thanks go to Prof. Philip H. S. Torr for his guidance in the field of computer vision, and M. Pawan Kumar from the Oxford Brookes University Vision Group for contribution of his original Histogram of Oriented Gradients program to this project.

Thanks to Ann Harvey for her advice and feedback during the writing of this report. Thanks also to Thomas El-Maraghi for creating the useful MATLAB SIFT resource used as a reference for the SIFT implementation in this dissertation.

The author also wishes to thank his family for their support while studying for the award of MSc in Computing at Oxford Brookes University.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Context

The field of computer vision employs numerous algorithms to extract interest features from images that can be used for applications such as object recognition. These algorithms try to search for features that are relatively invariant to changes in orientation and lighting conditions, so that the algorithms can find the same features in other images with different backgrounds or points of view.

The first algorithm that this dissertation is concerned with is called the SIFT operator (Scale Invariant Feature Transform), which has been developed by David Lowe. It was initially presented to the computer vision community in a paper a few years ago (Lowe 1999) and more recently with some improvements to the algorithm (Lowe 2004).

Features detected by this algorithm are invariant to small affine image transforms and small changes in lighting, so are quite robust compared to some of the other algorithms used to detect features for object recognition (Ke & Sukthankar 2003).

These features are used in object recognition to match features detected in a sample image to a large database of features extracted from various objects at different viewpoints. A match can be found by finding the most number of matches between keypoints that agree on an object's pose.

The SIFT algorithm analyses an image across scale-space (see Witkin 1983, Koenderink 1984) by creating an image pyramid with successive Gaussian blur filters, and then calculating the difference-of-Gaussian between two levels of the image scale space pyramid. It then finds maxima and minima across 3 adjacent difference-of-Gaussian levels to find potential keypoint locations. These keypoint locations are assessed for stability and descriptors are created at each of the stable locations. The descriptors represent the local image data around a keypoint in a way that is invariant to changes in scale, rotation, and small variations in illumination. This process is discussed in more detail in section 2.2.

The second algorithm that is explored in this dissertation uses descriptors very similar to those used by the SIFT algorithm. These descriptors are called Histograms of Oriented Gradients or HOG (Dalal & Triggs 2005), and are arranged within a window in a densely populated grid of overlapping descriptor blocks.

The HOG descriptor blocks are made up of a number of oriented histogram cells, and are arranged in one of two main configurations; square cells in a rectangular shaped descriptor called R-HOG, and angular cells arranged in a circular descriptor called C-HOG.

The histograms of each block from the window are combined together to form a HOG feature vector that is processed by a linear Support Vector Machine to classify the sample area within the HOG window. The SVM is trained using positive and negative examples from a training data set. Section 2.3 discusses this algorithm in more detail.

1.2 Outline

The aim of this project is to implement the SIFT algorithm using C++, and then explore possible improvements to it's performance or robustness.

The SIFT implementation discussed in Chapter 3 is implemented as a descriptor generator, and is not extended for use in real time object recognition applications as discussed in the original dissertation plan. The performance of the implementation was not sufficient enough to support a real-time detection scheme.

During the course of the dissertation a paper was published that presents an algorithm which uses feature descriptors very similar in construction to that of those used by the SIFT algorithm. This algorithm calls the class of descriptor it uses Histograms of Oriented Gradients (or HOG) (Dalal & Triggs 2005). It is shown to outperform the PCA-SIFT based descriptors presented by Ke & Sukthankar (2003), which were demonstrated to be more robust than the original SIFT descriptors.

Since the second phase of the project is to explore possible improvements to SIFT, the second phase focuses on exploring the performance of the new HOG based algorithm. Comparisons are made over the INRIA Comparisons presented in Dalal & Triggs (2005) and a new person database with more varied lighting conditions and poses. This new database is presented and discussed by this report in chapter 4.

1.3 Objectives

The first stage of the project is to implement Lowe's (2004) SIFT algorithm in C++. Once this is complete, comparisons between Lowe's (2004) demo program and the implementation will be made to assess correctness.

Once that has been completed, the next stage is to compare the performance of the PCA-SIFT descriptor to that of the HOG in the algorithm presented by Dalal & Triggs (2005). This

can be broken down in to a smaller set of goals.

- Generate a database of human poses using Poser
- Train both descriptors with the Poser and INRIA databases
- Compare results of classifying new images between descriptors trained with each database

To generate the human pose database, a software package called Poser will be used. This program is scriptable using a Python based language, so the final poses in the database will be automatically generated via script.

Training involves taking this database and training the two descriptors ready for classification. They will then be ready to classify new images that do not exist in the training data. The results of how well the descriptors performed in classifying new images will then be presented.

1.4 Resources

After the initial dissertation plan was submitted, another more accurate MATLAB resource (El-Maraghi 2004) than the one originally presented in the plan (Gustavsson, Hui & Turitzin 2004) was discovered. This second MatLab implementation more closely matches the results of David Lowe's own binary implementation of his algorithm, and better represents the algorithm described in the most recent SIFT paper (Lowe 2004).

In light of this, the new MatLab resource will be used as a reference to implementing the algorithm in C++ instead of the one originally specified in the dissertation plan.

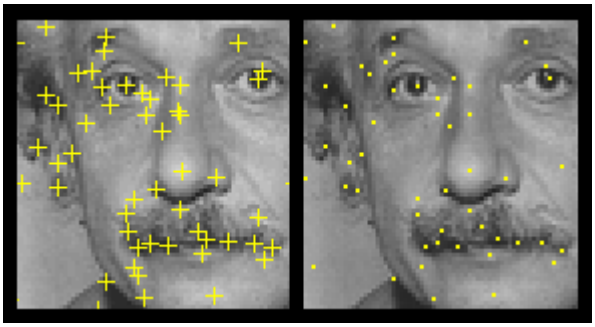

Figure 1.1: Comparison showing the MatLab keypoint locations (left) and the keypoint locations found by David Lowe's program (right). The keypoints on the right were plotted on top of the original image by using the locations in the keypoint file generated by the program.

Figure 1.1 shows the keypoint locations detected by the MatLab program in the left image, and on the right shows the location of keypoints found using David Lowe's keypoint demo program that was published along with his most recent SIFT paper (Lowe 2004). The figure shows that although the MatLab keypoint locations differ slightly from the keypoint locations found using David Lowe's program, their general positions are very similar.

The C++ implementation of the SIFT algorithm for this dissertation will attempt to be as close to the results of David Lowe's implementation as possible, so that there is a solid foundation to compare the effects of changes and improvements to the algorithm.

Source code for David Lowe's keypoint detection demo program is not publically available, so an exact reproduction of his implementation will not be possible. However, a reasonably close implementation should be able to be achieved using the MatLab source code and the two SIFT papers as reference material.

Chapter 2

Background Research

2.1 Image representation

2.1.1 Image Format

Pixels in the images processed in this report are all converted to a floating point grey level format in the range $[0, 1]$. All greyscale pixel values are assumed to be in this format and range throughout this report.

The original images processed however can be in various formats. Some images are greyscale, and others are full colour images with red, green and blue components. The images that are originally in colour are converted to their greyscale equivalent using the formula for calculating the Luminance component Y of the YIQ colour model (Hearn & Baker 2004), which creates a good greyscale representation of the image [1].

$$Y = 0.299\,R + 0.587\,G + 0.114\,B \tag{2.1}$$

2.1.2 Coordinate Systems

The coordinate system for image processing in this report is constructed from interpreting the image as a matrix with the horizontal axis as the columns of the matrix, and the vertical axis as the rows of the matrix. Pixels in this coordinate system will be referred to as either $(row, column)$ or in an abbreviated from (r, c).

One representation of image coordinate systems in computer graphics is to consider the horizontal axis as the x axis with the positive direction to the right, and the vertical axis as the

[1] The perception of 'correctness' seems to depend greatly on the use of colour in the source image. For instance an image with a large area of blue sky converted to greyscale using these coefficients will seem slightly dark, so there is no definitive simple transform to create greyscale levels that works for all images. However, this method is quite commonly used for creating greyscale representations of colour images and is used for image conversions in this report.

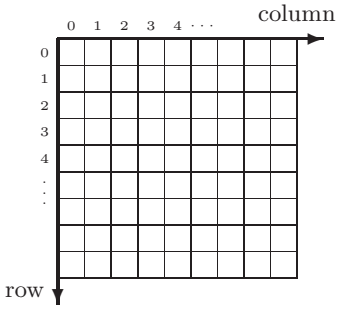

Figure 2.1: Row-column coordinate system for images.

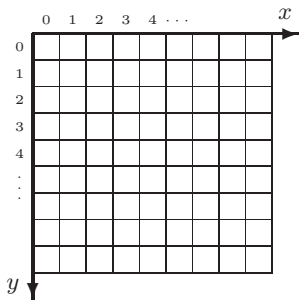

Figure 2.2: x-y coordinate system for images.

y axis with the positive direction downward. Most modern computer graphics libraries such as OpenGL® and DirectX® use this coordinate system representation.

This is similar to the system usually used for image processing mentioned above, as the vertical axis increases downward. The result of this is that given a linear buffer of memory, the offset to a given pixel at $(x, y)^T$ is found by evaluating the following C-style pseudo code:

$$\texttt{pixel} = \texttt{buffer}\,[\,(\texttt{y} * \texttt{width}) + \texttt{x}\,]$$

Where `width` is the width of the image in pixels. The same pixel specified in $(row, column)$ format would be indexed as:

$$\texttt{pixel} = \texttt{buffer}\,[\,(\texttt{row} * \texttt{width}) + \texttt{column}\,]$$

Therefore to convert from a pixel coordinate specified in $(row, column)$ form to (x, y) form, it's a simple case of mapping the row to y and $column$ to x, and vice versa to convert the other way.

In this report, when speaking in terms of (x, y) coordinates the coordinate system in figure 2.2 is being used, and for $(row, column)$ (or (r, c) in abbreviated form) the coordinate system in figure 2.1 is being used. However, on the whole (x, y) coordinates will be used when referring to image space locations.

It should be noted however, that Lowe's (2004) demo program appears to interpret images with the x-axis as vertical increasing downward, and the y-axis as horizontal increasing to the right. This is the transpose of the coordinate system used in this report.

2.2 The SIFT Operator

The Scale Invariant Feature Transform was developed by David Lowe (Lowe 1999, Lowe 2004) to find stable image features that can be used for object recognition. These features are invariant to rotation and scaling, as well as being relatively tolerant of small changes in illumination.

This kind of invariance means that these same features are also likely to be detected in images where the object has been rotated or is viewed from a slightly different perspective or distance.

There are four stages to the SIFT algorithm (Lowe 2004).

- Scale-space extrema detection
- Keypoint localization
- Orientation assignment
- Keypoint descriptor

2.2.1 Scale-space Extrema Detection

Scale-space Theory

Before this stage is discussed in any detail, a quick review of scale-space theory should be made.

Scale-space analysis is where a source signal I is convoluted with a variable scale Gaussian filter $G(\sigma)$ to project the signal in to what's known as scale-space. Varying the scale parameter σ moves the signal through scale space. Increasing σ has the effect of filtering out higher spatial frequencies in the signal so that only the lower spatial frequencies remain.

One of the immediate advantages of this is that low levels of noise are averaged out quite quickly as the scale parameter σ increases, so information in the remaining signal data can be analysed independently from small amounts of noise.

Scale-space theory has become popular since the works of Witkin (1983) and Koenderink (1984). However, research in scale-space theory was being done as early as 1959 by Taizo Iijima in a paper published in Japan in 1959 (Iijima 1959) and in following works, as discussed by Weickert, Ishikawa & Imiya (1999). This was independent of the work being done in the US and Europe later on.

Witkin (1983) discussed scale-space representation in a 1-dimensional case, but the principle holds for 2-dimensional case also (Koenderink 1984). Shown in figure 2.3 is a one-dimensional signal that has been smoothed by Gaussian kernels of increasing width (from Witkin 1983).

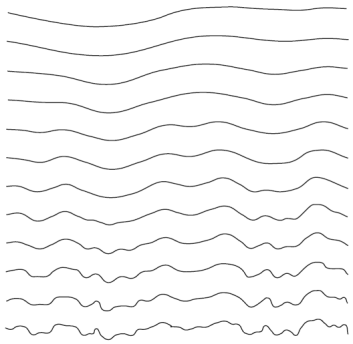

Figure 2.3: A one-dimensional signal smoothed by Gaussian kernels of increasing width (from Witkin 1983).

Extrema Detection

In this initial stage of the algorithm, a source image $I(x, y)$ is convoluted using a 2-dimensional Gaussian function $G(x, y, \sigma)$ to build an image pyramid. This creates a 3-dimensional representation with x and y representing the 2 image axis, and σ representing the scale axis of the image. Increasing σ moves the image plane upward in scale-space, and the image becomes increasingly more blurred as the higher spatial frequencies are filtered out.

The function $G(x, y, \sigma)$ is defined as follows:

$$G(x, y, \sigma) = \frac{1}{2\pi\sigma^2} \cdot e^{-\frac{(x^2+y^2)}{2\sigma^2}} \tag{2.2}$$

Convoluting the image $I(x, y)$ with $G(x, y, \sigma)$ creates the scale space representation $L(x, y, \sigma)$ of the image.

$$L(x, y, \sigma) = G(x, y, \sigma) * I(x, y) \tag{2.3}$$

The SIFT algorithm uses extrema found in the difference-of-Gaussian function from the convolved image pyramid that is computed from two adjacent scales in scale-space separated by a constant factor k. It has been shown that finding the value of this function is a close approximation to determining the scale-normalised Laplacian of Gaussian of the image (see Lindeberg 1994).

The difference-of-Gaussian function $D(x, y, \sigma)$ of the convoluted image $L(x, y, \sigma)$ is the subtraction of two adjacent scales in the Gaussian scale-space pyramid separated by constant factor k.

$$D(x, y, \sigma) = L(x, y, k\sigma) - L(x, y, \sigma) \tag{2.4}$$

To create the image pyramid, discrete intervals in scale-space are sampled by increasing the

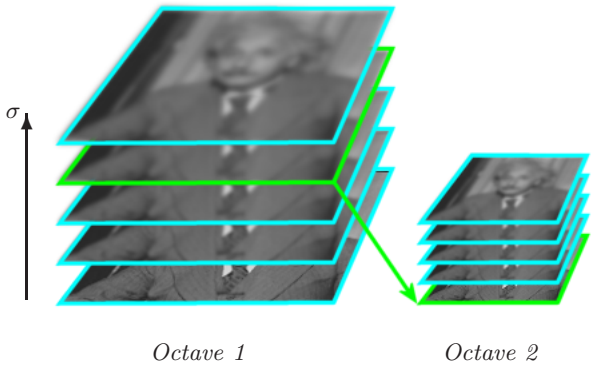

Figure 2.4: Two octaves of a Gaussian scale-space image pyramid with $s = 2$ intervals. The first image in the second octave is created by down sampling the second to last image in the previous octave by a factor of 2 (shown in green).

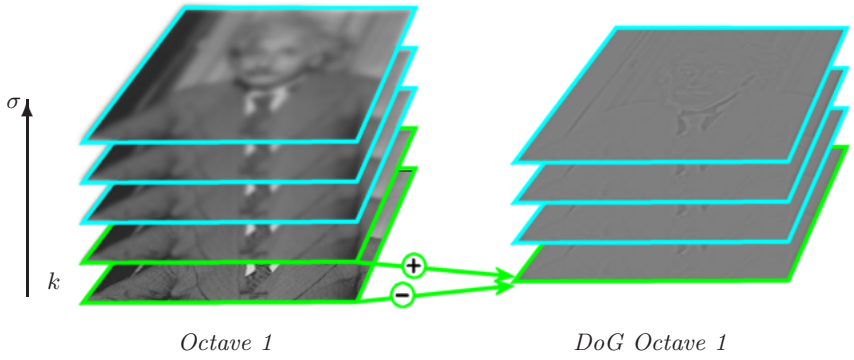

Figure 2.5: The difference of two adjacent intervals in the Gaussian scale-space pyramid create an interval in the difference-of-Gaussian pyramid (shown in green).

scale parameter σ by a constant amount.

In Lowe's (2004) paper, the image is down sampled by a factor of 2 when the scale parameter σ doubles so that an octave has half the previous octave's dimensions. This provides a great increase in processing speed with a negligible loss in accuracy.

Each of the octaves are split in to s intervals, and the factor k required to provide the correct number of intervals over an octave can be calculated using $2^{1/s}$. To search a complete octave for extrema detection $s+3$ intervals are required per octave for $s+2$ difference-of-Gaussian intervals.

For instance, if $s = 2$ then the number of Gaussian pyramid intervals per octave that would need to be created would be $2 + 3 = 5$ to create $2 + 2 = 4$ DoG intervals to search for extrema cover a complete octave. Of these DoG intervals, only 2 levels will be checked against the level above and below (for DoG intervals 1 and 4 there are no intervals to check below and above respectively, so they are only used during the search over intervals 2 and 3).

Extrema are found in the DoG intervals by comparing a sample to it's 8 neighbours on the

same level, 9 in the interval above and 9 in the interval below. If and only if the sample is either greater than or less than all of its 28 neighbours then it is considered as a potential keypoint.

2.2.2 Keypoint Localisation

Once all potential keypoint candidates have been identified, the keypoints need to be checked for stability.

The first test that needs to be made is that of contrast. If the value of $D(x, y, \sigma)$ at the keypoint location is less than a contrast threshold constant, then it is discarded as unstable and susceptible to low levels of noise. A value of $|D(x, y, \sigma)|$ less than 0.03 is used to filter out low contrast keypoints in Lowe's (2004) paper.

The most recent SIFT paper proposes a method of interpolating a keypoint by fitting a 3D parabola to the nearby sample data. If the calculated offset from the sample point is greater than 0.5 in any dimension, then the keypoint is moved to that position instead. Lowe (2004) proposes that this improves the stability of the keypoints found in the image.

The interpolated position is found by solving the following equation to find the offset:

$$\hat{\mathbf{x}} = -\frac{\delta^2 D}{\delta \mathbf{x}^2}^{-1} \frac{\delta D}{\delta \mathbf{x}} \tag{2.5}$$

Where $\hat{\mathbf{x}}$ is the offset vector from the sample point, and D and its derivatives are evaluated using local pixel differences around the sample point.

Lowe (2004) proposes that the function value at this final offset location $D(\hat{\mathbf{x}})$ is useful for rejecting unstable points with low contrast. This value can be found by evaluating:

$$D(\hat{\mathbf{x}}) = D + \frac{1}{2} \frac{\delta D}{\delta \mathbf{x}}^T \hat{\mathbf{x}} \tag{2.6}$$

Next keypoints that are situated along edges need to be rejected as they are poorly defined and are likely to be susceptible to small amounts of noise (Lowe 2004). Keypoints will have a small curvature along and edge, and a large curvature across the edge. These points can be rejected by finding the ratio of the principle curvatures at the sample point and rejecting ratios that are too large.

These principle curvatures can be found by calculating the Hessian matrix $\mathbf{H}$ at the keypoint location, using differences of neighbouring pixel samples to find the derivatives. Note that the matrix is symmetric so $D_{xy} = D_{yx}$.

$$\mathbf{H} = \begin{bmatrix} D_{xx} & D_{xy} \\ D_{xy} & D_{yy} \end{bmatrix} \tag{2.7}$$

The ratio of the principle curvatures can be found by calculating the trace $Tr(\mathbf{H})$ and determinant $Det(\mathbf{H})$ of the Hessian matrix:

$$\begin{aligned} Tr(\mathbf{H}) &= D_{xx} + D_{yy} \\ Det(\mathbf{H}) &= D_{xx}D_{yy} - D_{xy}^2 \end{aligned} \tag{2.8}$$

Then the following equation is evaluated to find the ratio of principle curvatures and check that it is below a desired threshold value r (Lowe 2004).

$$\frac{Tr(\mathbf{H})^2}{Det(\mathbf{H})} < \frac{(r+1)^2}{r} \tag{2.9}$$

Keypoints that have a curvature threshold less than r are rejected, as well as points that have a negative value for $Det(\mathbf{H})$. Keypoints with ratios greater than $r = 10$ were rejected in Lowe (2004).

2.2.3 Orientation Assignment

Once unstable keypoints have been discarded, orientations need to be assigned to each of the remaining keypoints so that local pixel data can be described relative to the orientation of a keypoint for rotational invariance.

The image gradient orientations and magnitudes of all the sample pixels on the same level as the keypoint that are within a Gaussian window of 1.5 times the scale of the keypoint are used to calculate a histogram. The histogram consists of 36 bins (one for each 10° step). Each sample contributes to the appropriate bin by its magnitude weighted by the Gaussian window.

For efficiency the orientations and magnitudes of the pixel data are precalculated during the creation of the image pyramid.

Let the finite central differences across x and y at pixel location $(x, y)^T$ be $\delta_x(x, y)$ and $\delta_y(x, y)$ respectively, then:

$$\begin{aligned} \delta_x(x, y) &= L(x+1, y) - L(x-1, y) \\ \delta_y(x, y) &= L(x, y+1) - L(x, y-1) \end{aligned} \tag{2.10}$$

The magnitude $m(x, y)$ at a sample pixel location can be found by evaluating the following equation:

$$m(x, y) = \sqrt{\delta_x(x, y)^2 + \delta_y(x, y)^2} \tag{2.11}$$

The orientation $\theta(x, y)$ of the gradient is relative to the image space x axis, and can be determined as follows:

$$\theta(x, y) = tan^{-1}\left(\frac{\delta_y(x, y)}{\delta_x(x, y)}\right) \tag{2.12}$$

Once the histogram has been calculated, keypoints are created for each orientation that has a value of 80% of the maximum histogram value or more. Lowe (2004) states that this contributes

significantly to stability of matching. For each of these peak orientations, a parabola is fit to the values of the 3 nearest bins to interpolate the orientation for better accuracy.

2.2.4 Keypoint Descriptor

Once all keypoint locations have been determined and have orientations assigned to them, the next stage is to create a descriptor to represent the image data around the keypoint in an invariant form.

Keypoint descriptors as used in Lowe (2004) are composed of a 4x4 grid of histograms formed from 4x4 pixel subregions from a larger 16x16 sample array. Histograms consist of 8 bins, one for each 45° step. The magnitudes of each of the sample points are weighted by a Gaussian window of width of half the keypoint's scale. To avoid boundary effects, a sample contributes to more than one bin in the histogram weighted by a factor of $1 - d$ where d is distance in histogram step units from a bin orientation. A sample also contributes to bins in adjacent histograms in the descriptor, again weighted by $1 - d$ where d is distance in descriptor histogram spacing units (4 for a 16x16 sample block).

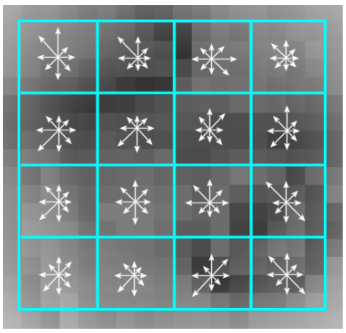

Figure 2.6: The layout 4x4 histogram bin SIFT descriptor. The descriptor covers a 16x16 pixel sample area, and each histogram cell in covers a 4x4 subregion within the 16x16 sample area. A sample gradient influences more than one bin using bilinear interpolation across cell spacing and across histogram bin spacing.

2.3 Histogram of Oriented Gradients

The 4x4 normalised gradient descriptors used in stage 4 of the SIFT algorithm are very similar in construction to a class of descriptors in an algorithm presented recently called Histogram of Oriented Gradients by Dalal & Triggs (2005). The algorithm has been shown to perform very well when applied to person detection, and outperforms the PCA based descriptors used in PCA-SIFT (Ke & Sukthankar 2003).

This section gives a brief overview of the operation of the HOG algorithm.

2.3.1 Technique

The principle behind the Histogram of Oriented Gradients (HOG) approach is quite simple. Whereas SIFT uses sparsely distributed descriptors positioned at extrema found in scale-space, HOG uses a dense array of overlapping histograms across a sample window.

The sample window is made up of a grid of overlapping blocks (the HOG descriptors themselves) and each of these blocks is made up of a number of 1D orientation histograms. The histograms within the descriptor are combined to form its representation, and the grid of descriptors are combined to form a feature vector that is used in a conventional SVM.

Each of the HOG descriptor blocks is normalised based on the 'energy' of the histograms contained within it. For example, a HOG vector $\mathbf{v}$ might be normalised using $\mathbf{v}/\sqrt{\|\mathbf{v}\|_2^2 + \epsilon^2}$ or a similar method. Dalal & Triggs (2005) discusses different methods of contrast normalisation, and the one used in this dissertation for the HOG implementation is the *L2-hys* method.

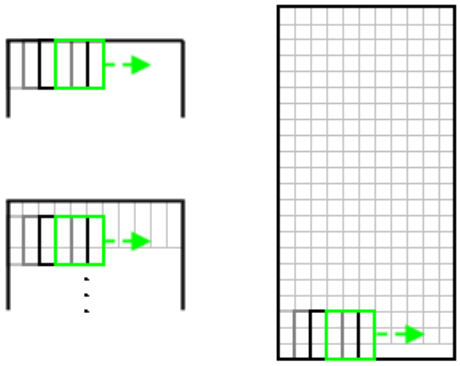

Figure 2.7: Construction of the HOG window with the HOG descriptor blocks. Blocks are created in order from the top left of the window moving to the right, then continue from the start of the left edge of the window for next row of blocks.

2.3.2 Linear Support Vector Machines

The linear Support Vector Machine (or SVM) used by the HOG algorithm is a binary classifier that when trained will classify a feature vector as either positive or negative using the equations in 2.13.

$$\begin{aligned} \mathbf{w}^T\mathbf{x} + b \geq 0 \quad & \text{for positive classification} \\ \mathbf{w}^T\mathbf{x} + b < 0 \quad & \text{for negative classification} \end{aligned} \tag{2.13}$$

Where $\mathbf{w}$ and b are the weight vector and bias determined during the training process that represent the decision hyperplane used to classify training examples, and $\mathbf{x}$ is the feature vector to be classified. The decision hyperplane surface is described by equation 2.14.

$$\mathbf{w}^T\mathbf{x} + b = 0 \tag{2.14}$$

The training problem of a linear SVM is to maximise the distance between the training examples from the decision hyperplane surface. This dissertation uses a program called SVM Light (Joachims 1999) to train the linear SVM using a set of positive and negative feature vector examples.

2.3.3 HOG Descriptors

There are two main types of HOG descriptor. These are a rectangular shaped R-HOG descriptor, and a circular shaped C-HOG descriptor. The layout of these two descriptors is shown in figure 2.8.

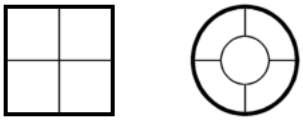

Figure 2.8: The layout of the two main descriptors used in the HOG algorithm. A 2x2 cell R-HOG descriptor is shown on the left, and a 5 cell C-HOG descriptor is shown on the right.

R-HOG

The R-HOG descriptor is a rectangular shaped grouping of histogram cells. Dalal & Triggs (2005) explores different configurations of this descriptor, but 2x2 blocks of 8x8 cells seem to give the best results on the INRIA database.

C-HOG

The R-HOG descriptor is a circular shaped group of histogram cells. The block is split in to a number of radial cells, and those are further split in to a number of angular cells. Several configurations were explored by Dalal & Triggs (2005) but a single central bin and 4 angular radial bins with a total descriptor radius of 8 samples seem to perform best.

2.3.4 Training

The HOG classifier uses a linear SVM to classify images, and Dalal & Triggs (2005) uses SVM Light (Joachims 1999) for training.

To train the SVM, HOG feature vectors are generated for a set of positive training images and randomly sampled patches from a negative image set. Once trained, the negative images are searched exhaustively in scale-space to find false positives to use as hard examples. These

hard examples are added to the initial positive and negative training set, and the SVM is trained again using this augmented set. Dalal & Triggs (2005) states that this improves the classification performance of the classifier.

In Dalal & Triggs's (2005) paper the exhaustive search phase of the training uses an image pyramid to search scale space. Each of the levels is separated by a factor of 1.2, and new levels are created until either the width of the pyramid level reaches 64 or the height reaches 128. The HOG windows are separated by a stride of 8 pixels at any level. To reduce the effects of boundary conditions however, a window size of 96x160 is used for this report.

If there is a margin left over after fitting the HOG windows, the whole window grid is shifted so that it is centred on the level. The formula used to calculate the offset to shift the window grid for HOG window H and image I in the pyramid is:

$$\begin{aligned} x &= \lfloor (I_w - n_x H_w)/2 \rfloor \\ y &= \lfloor (I_h - n_y H_h)/2 \rfloor \end{aligned} \tag{2.15}$$

Where I_w and I_h are the pyramid image width and height respectively, H_w and H_h are the HOG window width and height plus the 8 pixel stride respectively, and n_x and n_y are the number of HOG windows in the x and y directions with a window stride of 8 pixels.

In this report, the minimum pyramid level sizes for width and height are 90 and 160 respectively. This is so that the size for negative training images doesn't fall below the normalised input image size and avoids boundary conditions.

2.3.5 Comparisons

There are a few variations of the HOG descriptor discussed in Dalal & Triggs (2005). The two main categories are rectangular and circular HOG descriptors (R-HOG and C-HOG respectively), and the paper explores their performance compared to SIFT descriptors (as discussed in section 2.2.4), Shape context descriptors (simulated using C-HOG descriptors) and generalised Haar wavelet based descriptors.

Dalal & Triggs (2005) demonstrated that HOG based descriptors outperformed each of the other descriptor variations on two different person databases: the MIT pedestrian database, and a more challenging INRIA database created to test the HOG descriptor.

2.3.6 Parameters

The HOG implementation used in this dissertation is based on source code provided by M. Pawan Kumar, a PhD student from the Oxford Brookes Vision Group.

The HOG feature vectors used in the implementation are made from a 64x128 sample window consisting of a grid of overlapping 2x2 R-HOG descriptors of 8x8 sample histogram cells. The HOG descriptors are overlapped using a stride of 4 pixels and each of the HOG descriptor blocks

is normalised using the *L2-hys* method. The 1D histograms have 9 bins over a 180 degree range (the 'sign' of the gradient is ignored).

For the positive training set, the normalised input image set from the INRIA Person database is used. In these images the human figure is surrounded by approximately 16 pixels within the 64x128 detection window, and to avoid boundary conditions the detection window is surrounded by an additional 16 pixel border so that the final positive image size is 96x160. This normalisation method is also used to create training images in the pose database discussed in section 4.

Chapter 3

SIFT Implementation

3.1 Introduction

The SIFT algorithm was implemented from the ground up using C++ and Visual Studio .net 2003. The two SIFT papers (Lowe 1999, Lowe 2004) were used as a reference for the structure of the algorithm in addition to the MatLab resource identified (El-Maraghi 2004) in section 1.4.

The following sections describe the overall structure of the implementation and the reasons behind the implementation decisions made.

3.2 Design

To keep the problem domain focused on implementing the algorithm and not that of user interface design, the target application is a simple command line Win32 console application.

The program takes the name of the image to process and outputs several image files to show it's results, and a file containing the SIFT descriptors. The generated image files are of the Gaussian scale-space pyramid, the DoG pyramid with extrema marked in yellow, and separate copies of the original image: one to show detected extrema and another to show final keypoint locations and their dominant orientations.

While the program is running, progress information is output to the console window. Each stage of the SIFT algorithm that is being performed and how long it takes is displayed, in addition to other useful information for debugging purposes. This profiling information is useful in analysing the relative performance of the implementation when processing image files.

The preferred output image format of the program is that of PNG (Portable Network Graphics). This is an open-source image format with good lossless PK-ZIP based compression, so is ideal for outputting the somewhat large images generated by the program. The LibPNG library (PNG Development Group 2005) is used by this program to load and save PNG images.

3.2.1 Convolution

The program supports two methods of convolution that can be used to create the Gaussian scale-space pyramid. These two methods were implemented so that their effect on extrema detection can be compared. The differences between these two methods are shown in section 3.4.

Gaussian Box Filter

The first of these is a simple Gaussian box filter method that adds the weighted local neighbourhood of each pixel to create the blurred image. This is implemented efficiently using two passes of 1D Gaussian kernels in the x and y directions instead of applying a 2D kernel. This takes advantage of the separability property of convolution, and speeds up the filtering significantly.

Another optimisation is that instead of using the first level of the image and convoluting with different sized kernels for each level, each scale-space interval is created by convoluting the interval before it using a smaller kernel. The required standard deviation $\delta\sigma_{i+1}$ to calculate interval $i+1$ from interval i with σ_i can be calculated using equation 3.1 (where $k = 2^{1/s}$).

$$\delta\sigma_{i+1} = \sigma_i\sqrt{k^2 - 1} \tag{3.1}$$

This optimisation also speeds up the convolution required to create the Gaussian scale-space image by reducing the size of the kernels processed, and there for the number of calculations required to create each interval.

FFT Based Convolution

An alternate method provided by the program is a slower but more accurate Fast Fourier Transform based convolution. The program uses a library called 'Fastest FFT in the West' or FFTW (Frigo & Johnson 2005), and provides a fast implementation of the FFT algorithm.

For the FFT convolution approach, since the FFT transform speed is approximately constant regardless of kernel size the first interval in each octave is convoluted by different kernels to create the other interval levels.

Both the Gaussian kernel and source image are transformed in to the frequency domain using FFT, then they are multiplied together and the result is transformed back using the inverse FFT transform.

3.3 Structure

The general structure of the program is shown in table 3.3.

There are two main classes used in the SIFT implementation. One is a general representation of a floating point image, and the other is an encapsulation of the algorithms required by SIFT.

Since the Gaussian scale-space pyramid is the foundation of the SIFT algorithm, making this the central part of the SIFT keypoint detector seemed a logical choice. The class consists of

SIFT program

```
(01)  parse options
(02)  load image
(03)  resample image to double size
(04)  for each octave
(05)      create Gaussian blur intervals
(06)      create difference-of-Gaussian intervals
(07)      compute edges for each interval
(--)  end for
(08)  search each octave for stable extrema
(09)  create keypoints at dominant orientations of extrema
(10)  for each keypoint
(11)      rotate sample grid to keypoint orientation
(12)      sample region and create descriptor
(--)  end for
(13)  optionally save pyramid images
(14)  save output images
(15)  save descriptors
```

Table 3.1: Pseudo-code of the SIFT keypoint descriptor program.

several smaller data structures that manage the scale-space and difference-of-Gaussian representations in memory, as well as the precalculated image pixel gradients and magnitudes used by the algorithm.

The image class is a general purpose representation of a floating point image and has several image manipulation routines associated with it for use by the SIFT algorithm. It supports loading and saving different types of image formats, and converts RGB colour images in to their greyscale representations using the method described in section 2.1.1 using equation 2.1.

Both of these classes have a COM (Component Object Model) style interface to them, so that their lifetime is governed by reference counting. Auto-reference-counting structures are used to safely manage these classes during program operation. The reasoning behind this design choice was that by moving the memory management responsibility to the classes themselves, it would make memory management simpler to handle should the SIFT implementation code be used to create a general purpose library in the future.

3.3.1 SIFTImage

This class is responsible for providing an interface to a floating point image. Images can be loaded from file in to memory using this class, or the image can be given a pointer to image data already in memory (e.g. memory in a large buffer).

The ability to manipulate images indirectly allows the application to allocate a fixed size buffer for image manipulation routines and re-use it for consecutive image operations. This avoids having to re-allocate memory every time operations need to be performed.

In addition to loading an image from file or indirectly via memory pointer, the class can also make a copy of an image from a location elsewhere in memory. This is useful when displaying the results of the program on top of the original image, while leaving the source image intact.

3.3.2 SIFTPyramid

The SIFTPyramid class encapsulates and manages all the data structures required by the SIFT algorithm. Gaussian scale-space and DoG pyramid representations as well as the image pixel gradient and magnitudes for each pyramid level are managed by this class.

The image pyramid data is grouped in to octave data structures. Each octave contains a set of scale-space intervals, DoG intervals, and image pixel gradient information.

The memory for the all the octaves and intervals in the scale-space pyramid are allocated in a single pixel buffer, and SIFTImage objects are created to point to a specific area of memory within the buffer using the SIFTImage indirect referencing method. A second pixel buffer for the difference-of-Gaussian pyramid is also created and SIFTImages are used in indirect mode to provide an interface to the image data.

Gradient information for the scale-space pixel data across all octaves and intervals is stored in a buffer and indirectly referenced via a simple data structure within the data structure for each octave.

When an image is passed to the SIFTPyramid class for processing, the pyramid structures are resized if necessary to accommodate the image size and desired number of octaves and intervals. Once this has been done, the scale-space pyramid, DoG pyramid and scale-space pyramid pixel gradients are created using the source image.

Once the pyramid has been generated, SIFTPyramid can be used to find extrema, keypoints and create SIFT descriptors.

3.3.3 Other Classes

In addition to SIFTImage and SIFTPyramid, there are several modules that contain code for calculating image convolution and some general purpose mathematical and graphics related routines. These are used by the two main classes to create results and calculate the pyramid for the source image.

3.4 Program Results

3.4.1 Extrema Detection Comparison

To study the results of the extrema detection stage of the algorithm, an image was analysed using both the FFT and box filter methods then the extrema were compared to extrema found by the keypoint detector from Lowe (2004).

The results of extrema detection on the same image region from the keypoint program and the two convolution methods used by this SIFT implementation is shown in figure 3.1.

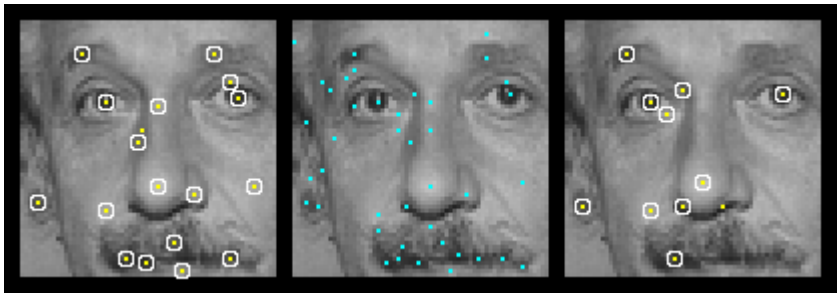

Figure 3.1: White circles mark extrema that are within a few pixels of extrema found by the keypoint program. Left: extrema found when using a box filter to create the image pyramid. Middle: extrema found by the keypoint program. Right: extrema found when using FFT to create the image pyramid.

The extrema found in both convolution methods that are in within a few pixels of extrema detected by the keypoint program from Lowe (2004) are marked with white circles.

This shows that although both convolution methods yield a different number of stable extrema, nearly all of the extrema detected are in a similar or near position to extrema found by the keypoint program.

This demonstrates that the extrema detection stage of the SIFT implementation finds extrema with a reasonable amount of accuracy, and differences are most likely due to numerical inaccuracy from floating point rounding errors in the methods used to create the image pyramid.

Due to the number of floating point operations involved in the box filter method, it is the least accurate of the two methods due to accumulated rounding errors over the intervals of the scale-space pyramid. The FFT method is very accurate due to the minimal number of floating point operations used for each convolution.

Hessian Curvature Rejection

When estimating the principle curvature ratios at an extrema point by estimating a Hessian matrix using local pixel differences, extrema on lines that are near 45 degrees are not rejected properly.

After experimentation, it was found that by estimating a second Hessian matrix using pixel offsets rotated by 45 degrees and filtering based on the largest curvature ratio from both Hessian matrices, the number of line rejections improves as shown in figure 3.2.

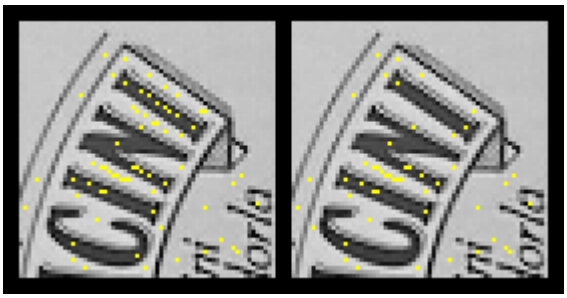

Figure 3.2: Left: extrema detected using a single curvature ratio check. Right: extrema detected using an additional curvature check using sample offsets rotated by 45 degrees.

3.4.2 Keypoint Matching

If the program is working correctly, then a number of similar keypoints should be detected in a source image and the same image modified by an affine transform such as a 45 degree rotation.

The result of keypoint detection on a test image is shown in figure 3.3, and the keypoints found in the same image rotated by 45 degrees is shown 3.4. These results demonstrate that similar keypoints are detected in both images that can be used for matching.

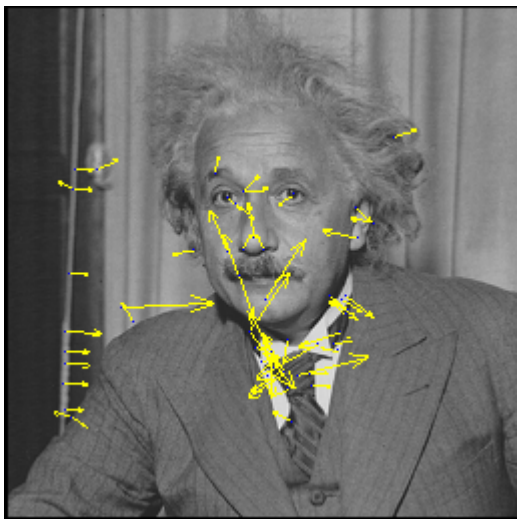

Figure 3.3: Keypoints found in a test image.

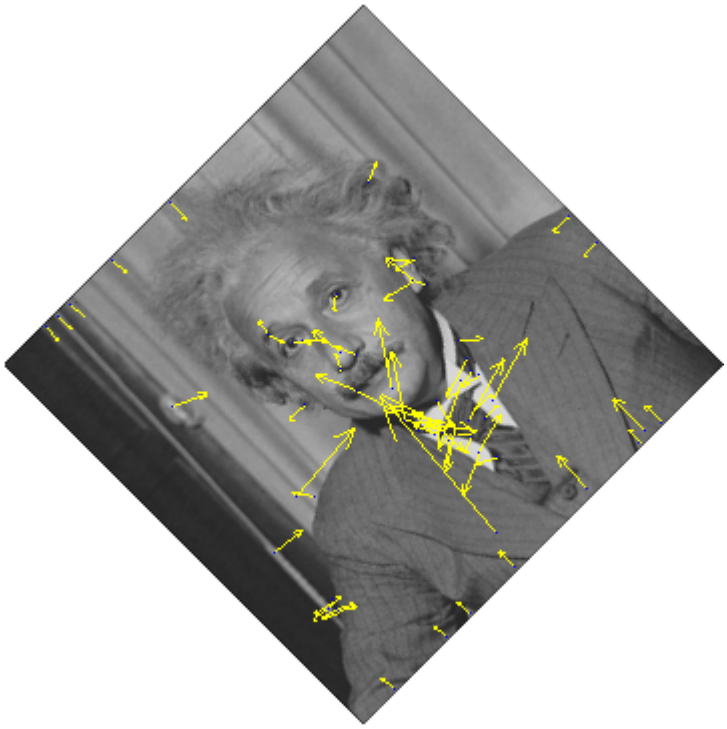

Figure 3.4: Keypoints found in a rotated version of the test image.

The `match` program from Lowe's (2004) SIFT demo was used to find matches between keypoints found in both images. The program draws lines between keypoints that are found to match in both images. This can be used with visual inspection to assess matching accuracy. The results from this matching program are shown in figure 3.5.

3.5 Discussion

The SIFT implementation presented in this chapter has been shown to detect extrema in scale-space and create a reasonable amount of similar keypoints between images that can be used for matching.

When compared to the performance of the keypoint program from Lowe (2004) the number of matchable keypoints is less in this implementation. This could be due to differences in descriptor construction and numerical accuracy.

Increasing the accuracy of the descriptor is something that could be studied further to improve robustness of matching keypoints created by this program.

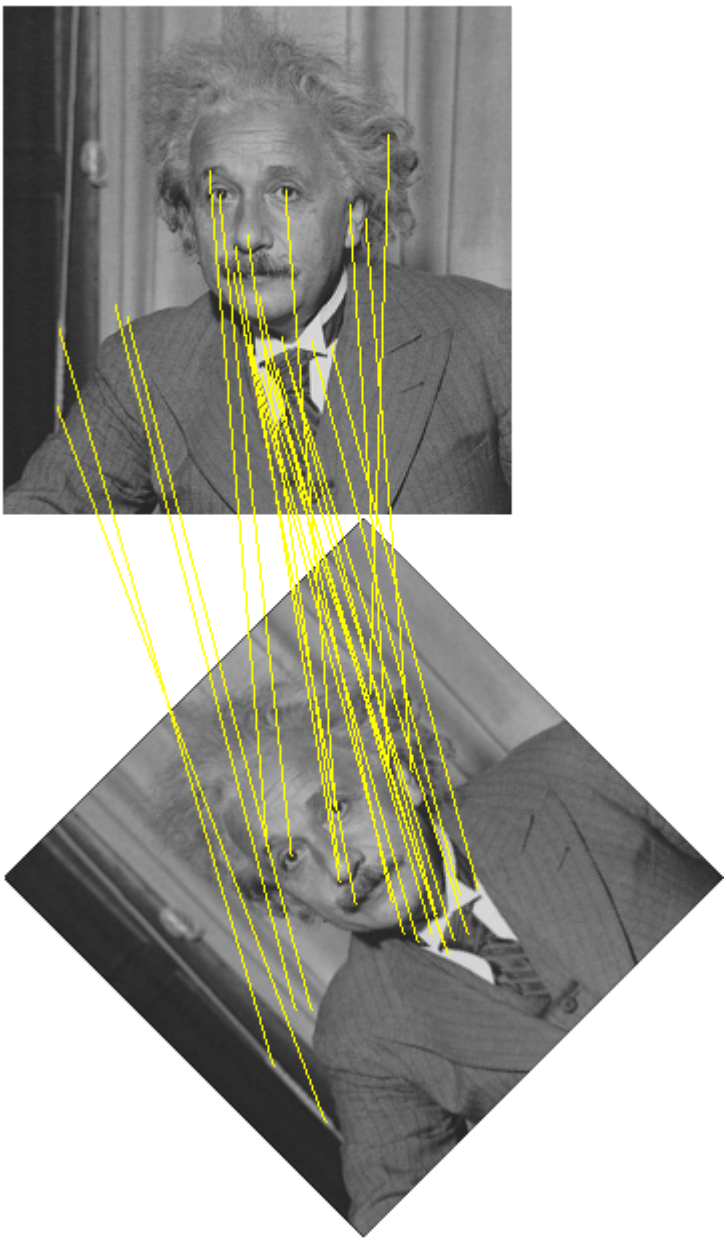

Figure 3.5: Results showing keypoint matches between both test images.

Chapter 4

Pose Database

This chapter describes the rationale behind the poses generated in the pose database used for this project. The image database consists of 5,000 images with random human poses, viewing angles and lighting conditions.

4.1 Curious Labs' Poser

The program used to generate the pose database is Curious Labs Poser 5. This program is a powerful animation tool specifically designed for animating human figures and rendering them in a realistic way.

The program offers a stylised user interface through which various aspects of a character's appearance and pose can be controlled. The interfaces of interest for this database are that of the 'Pose' and 'Material' screens.

The Pose screen allows the user to control the various joints using a parameter window, or alternatively the user may hold and drag a limb in the 3D view to animate the character. When the user moves a limb using the 3D view, the limb will influence other connected limbs based on the limbs relative size. For instance, moving the arm of a figure will move the shoulder and torso, but moving a finger will have less of an influence on the other joints due to it's relatively small size.

The Material interface provides a visual component based system for creating materials and textures that can be applied to a part of a figure. Different material components in the interface can be linked together to form simple materials, or many can be connected together to form more complex materials.

Using these interfaces a set of 10 male human figures were created with varied clothing and material types. To help vary the clothing appearance, some textures were used from a free online texture library available at `http://www.mayang.com/textures/` (Adnin & Smith 2005).

4.1.1 PoserPython

Although the program offers a well featured interface for creating different human poses, using this to create thousands of images manually would be unrealistically time consuming. To help relieve some of this work, the application provides an implementation of the Python scripting language called PoserPython that can be used to automate many aspects the program.

Using this scripting language a figure's limbs can be positioned, the view angle can be adjusted and the lighting can be changed. For the pose database, all these variables were changed randomly to give a set of many different poses.

4.2 Method

4.2.1 Range of Movement

To create the pose database, the default male figure was selected from the figures library and limbs were identified that could be repositioned over the set of poses. The movements that were identified were as follows.

- Arm side movements
- Torso left and right tilt
- Torso forward bend
- Various natural looking leg poses selected from Poser 5's pose database

In addition to these body movements, the camera viewing angle is rotated around the figure to give different viewpoints of a given pose. The movement ranges of the limbs are shown in figure 4.1, and some example base poses for standing and walking are shown in figure 4.2.

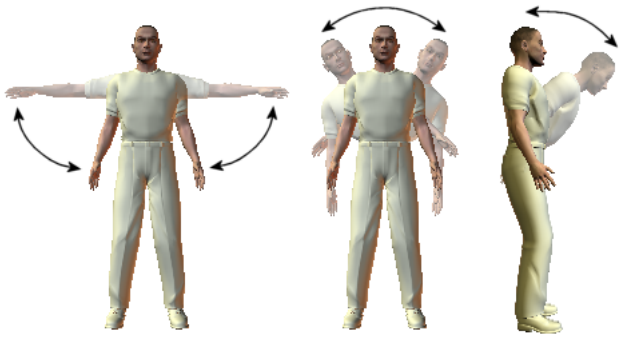

Figure 4.1: Range of movement used to adjust figure pose over the database. These were combined with a base walking or standing pose drawn from Poser 5's pose library.

Figure 4.2: Some of the base poses used from the Poser 5 pose library.

4.2.2 PoseLib

The application's PoserPython scripting language was used to generate the pose database. Using this language, several classes and utility functions were created to control the figure's pose in a logical way. These classes are grouped up in a Python package called PoseLib.

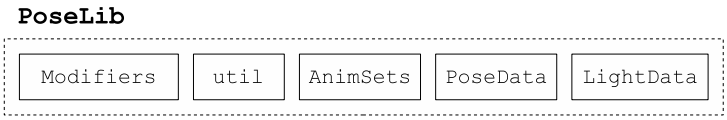

Figure 4.3: Modules contained within the PoseLib package.

Shown in figure 4.3 are the PoseLib modules used by the main pose generation program. The objects that control a pose or manipulate the scene in some way are known as Modifiers, and reside inside the Modifiers module. There are various types of modifiers to control the different limb movements, and each inherit from the Modifier class.

The main classes used to create the database reside in the Modifiers module. The other modules contain pose data and miscellaneous utility functions.

Modifiers

A Modifier is an object that defines some aspect of a figure's pose over a specified number of steps in a sequence. Modifiers can be initialised to an arbitrary step number in the sequence, and can be advanced incrementally through the sequence.

There are several different Modifier types that are responsible for controlling a certain aspect of a figure's pose.

Aggregate Types

In addition to the modification classes there are abstract aggregate types for grouping up more than one modifier for more complex control over limb movements.

The first of these is called Composite, and is used to apply a set of modifiers in parallel. For example, this class is useful for positioning a figure's legs as more than one limb needs to be adjusted to achieve a natural pose.

The other aggregate type is called Sequence, and allows the concatenation of other modifiers in to a larger sequence. This class is useful to chain together otherwise unrelated modifiers in to a single sequence, or simply link together shorter modifier sequences.

Both of these aggregate types can consist of other aggregates to create arbitrarily complex sequences for controlling a figure's pose.

Animation Builders

The third type of class within the PoseLib library is for generating a set of poses given a set of modifiers and frame range parameters. There are two of these classes within the Modifiers module.

The first is called AnimBuilder which constructs a pose set consisting of all possible combinations of the set of modifiers given to it. This is done by recursively applying the modifiers and advancing the frame number at the end of each modifier applied.

The second is RandomAnimBuilder which generates a set of random combinations of the modifiers given to it and applies them to the figure. The random vectors can either be generated internally by the class itself, or the class can be given a set of animation vectors containing the desired pose at any given state.

4.2.3 Pose Creation

A Python script was created to generate the database using the classes introduces above. This script controls the configuration of the animation and vector generation for the poses.

The script generates a set of random animation vectors for the entire 5,000 frame sequence using a constant seed for Python's random number generator (42 in the case of this database). Each element of an animation vector represents a step number for its corresponding modifier. For an animation vector $\mathbf{v}$ and a vector of modifiers $\mathbf{m}$, an animation vector's element v_i corresponds to the step number for the modifier m_i from the vector of modifiers.

The 5,000 images in the database are split in to equal length sub-sequences of 500 frames for each of the 10 figure models. For each of the image sub-sequences, the following steps are performed:

- Pose generation for the 500 frame sub-sequence (`generate_database.py`)
- Figure selection

- Render frames

Figure 4.4 shows some sample images generated by the script. The figure is segmented from the rest of the image using an alpha channel. This makes it easy to superimpose on top of a background image, as discussed in section 4.2.4.

Figure 4.4: Sample poses rendered by Poser 5 for the database.

Some of the poses created look quite natural, but others – although physically plausible – aren't particularly common poses for a human figure to take. This will make the database quite challenging.

1. Pose generation

To generate the pose for each image in the sub-sequence, the modifiers configured in the script are applied in order. The first of the modifiers change the whole pose of the figure and so need to be applied before the other more specialised modifiers. These initial modifiers are used to create a natural looking walking and standing poses. These complete poses are taken from the Pose libraries in Poser 5.

After the base pose has been set by the first modifier, the modifiers for controlling individual limb movements are applied.

Finally, a random lighting condition is applied to the frame.

2. Figure selection

Once the poses for sub-sequence have been generated, the next stage is to select the figure that will be used for this sub-sequence. The reason for selecting the figure after pose generation is due to the way the poses are defined in script.

3. Render frames

Finally, once the pose sub-sequence has been created and lit, the images are rendered to disk in PNG format. Each of the rendered images contain the figure pose on a transparent background created using an alpha channel.

4.2.4 Preparing the database

Once all 5,000 pose images have been rendered to PNG files, they are superimposed on a random background image region drawn from the negative training set in the INRIA Person database.

For training the classifier, the positive images need to be in a format that readily processed by the HOG algorithm to generate a feature vector for the SVM. This final image is constructed for a 64x128 window of HOG descriptors with a border of 16 pixels around the figure. Due to the varied set of arm positions in the pose database, the height of the figure was used to rescale the source pose to fit within the 16 pixel border.

To do this a program was developed to process a list of figures and a list of backgrounds to draw from, and prepare a final image of dimensions 96x160 (64x128 plus a 16 pixel border) ready for training. The design of this program is discussed in section 4.2.5.

Figure 4.5 shows a sample output from the program. The background regions used are randomly sampled from the background images input to the program so not all background images look entirely natural. However, they are sufficient for use as background clutter for the pose images.

Figure 4.5: Sample images from the superimpose program. Each image is 96x160 pixels. Each image has a 16 pixel margin around the centered 64x128 sample window and an approximate border size of 16 pixel between the figure and centered 64x128 sample window.

4.2.5 The `superimpose` program

The program used to rescale and subsample the main pose set takes as its input a list of figures, a list of backgrounds, an output directory and resampling and subregion options for processing the figures.

The pseudo-code in table 4.1 gives a high level view of the program operation. For each of the figures in the input list, the figure is processed based on the subsampling and resizing options passed to the program. After the figure has been processed, a random background is selected and

the figure is superimposed on top of the background image using the alpha channel information in the figure image generated by Poser 5.

superimpose program
(01) parse options
(02) load figure file list
(03) load background file list
(04) while (figurenumber < totalfigures)
(05) crop figure
(06) rescale figure
(07) pick random background
(08) superimpose figure on background
(09) save to output directory
(10) increment figurenumber
(--) end while

Table 4.1: Pseudo-code for the `superimpose` program.

Structure

The program consists of two main classes. The first is the `ImageProcessor` class that encapsulates the algorithm shown in figure 4.1.

The second is the `Image` class. This class represents a 32 bit ARGB image, and includes various methods to provide the image processing functionality required by the main program. Among these methods are image copying and resampling routines, loading and saving routines, as well as whole-pixel and (bilinear filtered) sub-pixel addressing methods.

Using the functionality of the `Image` class, the main program can process the figure and background images efficiently.

Figure 4.6 shows how a subregion of the original pose render is resized to fit within a 96x160 image. The green box on the left image shows the desired subregion and the dotted black box shows the desired inner border for the image. The distance between the green box and the outer black box is the scaled 16 pixel margin.

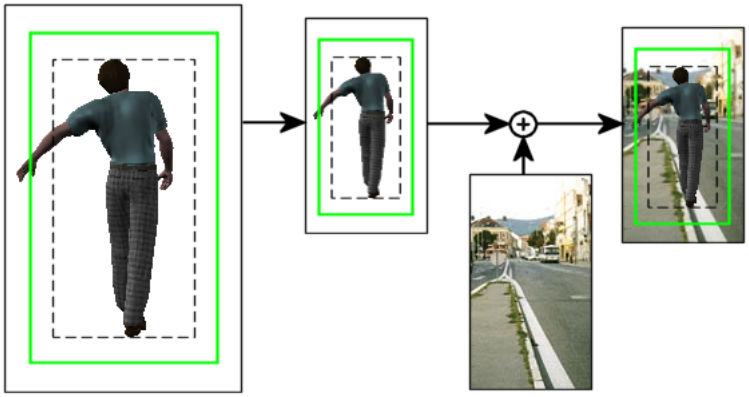

Figure 4.6: Pose image subregion with inner border resampled to fit within the required 96x160 image with a 16x16 border, and combined with a background image.

Chapter 5

HOG Implementation

The implementations of the HOG algorithm used for the training phase of the HOG experiments within this report are based on a modified version of an implementation in C provided by M. Pawan Kumar, a PhD student from the Oxford Brookes University Vision group.

The changes made to this implementation include the addition of bilinear cell interpolation to the R-HOG descriptor, the addition of the C-HOG descriptor and the separate program compilations for R-HOG and C-HOG respectively.

This chapter outlines the operation of the training programs, discusses the descriptor implementations used, and introduces a new program for classification across multiple scales.

For the training the linear Support Vector Machine used in the classifier, a program called SVM Light is used (Joachims 1999). This implementation of an SVM learner is optimised to learn training databases with large feature vectors.

5.1 Descriptors

The two descriptor types both cover a 16x16 pixel sample area, and are arranged in a dense grid across a 64x128 sample area window. The stride used to separate the descriptor blocks is 4 pixels, and the total number of descriptors in the x and y direction can be determined using equations 5.1 and 5.2.

$$n_x = \frac{w_w - b_w + b_s}{b_s} \tag{5.1}$$

$$n_y = \frac{w_h - b_h + b_s}{b_s} \tag{5.2}$$

Where w_w and w_h are the window width and height respectively, b_w and b_h are the block width and height respectively and b_s is the block stride.

The descriptors are created in order from the top left of the window moving to the right, and

then begin again on the next row (where $y' = y + b_s$), as discussed in section 2.3.1. As each descriptor is created it is appended to the end of the HOG window feature vector.

5.1.1 R-HOG Implementation

The R-HOG configuration used in the programs is the one identified in section 2.3.3, consisting of a 2x2 grid of 8x8 histogram sample bins. The descriptor covers a sample area in the window of 16x16 pixels, with each cell within the descriptor covering 8x8 samples.

Sample contributions are weighted by a Gaussian window positioned at the centre of the descriptor block, and contribute to adjacent histogram bins within a cell using a bilinear interpolation scheme. The contribution is weighted based on the distance from the sample to the centre of each histogram cell. The interpolation used in this dissertation is based on equations 5.3 and 5.4.

$$w_c(C_{ij}) = 1 - min\left[\frac{\|\mathbf{s} - \mathbf{c_{ij}}\|}{d_c}, 1\right] \tag{5.3}$$

$$w_\theta(h_b) = 1 - min\left[\frac{\|\theta(\mathbf{s}) - \theta_b\|}{d_\theta}, 1\right] \tag{5.4}$$

Where $w_c(\cdot)$ and $w_\theta(\cdot)$ determine the weighting for a cell and histogram angle bin respectively, $\mathbf{s}$ is the (x, y) position of the sample, $\mathbf{c_{ij}}$ is the centre of the cell C_{ij}, the function $\theta(\cdot)$ finds the gradient angle at a sample position, θ_b is the centre of histogram bin h_b, and d_c and d_θ are the cell spacing and histogram bin spacings respectively.

The gradient angle at the sample point is calculated using equation 5.5. This equation finds the gradient angle ignoring the sign of the gradient direction, *i.e.* the angles $+\frac{\pi}{2}$ and $-\frac{\pi}{2}$ both contribute to the same bins. The two functions $\delta_x(x, y)$ and $\delta_y(x, y)$ determine the central differences at (x, y) (as defined in equation 2.10 from section 2.2.3).

$$\theta(\mathbf{s}) = \cos^{-1}\left(\frac{\delta_x(s_x, s_y)}{\sqrt{\delta_x(s_x, s_y)^2 + \delta_y(s_x, s_y)^2}}\right) \tag{5.5}$$

The histogram bin contributions are determined by multiplying the gradient magnitude at the sample point $\mathbf{s}$ with the cell and histogram bin weights and a Gaussian function defined in equation 5.6 with a σ of half the descriptor window width[1].

$$G(d_s) = \frac{1}{\sqrt{2\pi}\,\sigma} \cdot e^{\frac{-d_s^2}{\sigma^2}} \tag{5.6}$$

Where d_s is the distance from the sample to the centre of the descriptor.

Once all the samples in the 16x16 area have been processed and their contributions to the descriptor histogram cells have been accumulated, the descriptor is normalized then the descriptor elements are thresholded so that none of the elements have a value greater than 0.2, and

[1] For the 16x16 sample area R-HOG descriptors used in this implementation, a value of 8 is used for σ.

then the descriptor is normalised again. This scheme is also used in the SIFT descriptors for normalisation and gives the descriptor more emphasis on orientation distributions rather than gradient strengths (Lowe 2004).

5.1.2 C-HOG Implementation

The C-HOG descriptor used in the experiments for this dissertation consist of a single central cell, and 4 equally spaced angular cells. The radius of the central bin is 3 pixels, and the radius of the outer cells is 5 pixels. The total radius of the descriptor is 8 pixels centred in the middle of a 16x16 sample area.

As with the R-HOG descriptor, samples contribute to adjacent histogram cells using a bilinear filtering based method. The sample position relative to the descriptor centre is used to determine the radial and angular cell weighting.

The angular cell c_a that the sample lies within is determined using equations 5.7 and 5.8.

$$c_a = \left\lfloor \frac{s_\phi}{d_\phi} \right\rfloor \tag{5.7}$$

$$s_\phi = \begin{cases} \cos^{-1}(\mathbf{n} \cdot \vec{\mathbf{x}}) & \text{if } \mathbf{n} \cdot \vec{\mathbf{y}} \geqslant 0 \\ 2\pi - \cos^{-1}(\mathbf{n} \cdot \vec{\mathbf{x}}) & \text{if } \mathbf{n} \cdot \vec{\mathbf{y}} < 0 \end{cases} \tag{5.8}$$

Where $\mathbf{n} = \mathbf{s}/\|\mathbf{s}\|$ and $\mathbf{s}$ is the sample (x, y) position. The vectors $\vec{\mathbf{x}}$ and $\vec{\mathbf{y}}$ are unit vectors in the x and y directions respectively, s_ϕ is the angle of the sample point from x axis relative to the centre of the descriptor, and d_ϕ is the spacing of the angular cells ($\frac{\pi}{2}$ for this descriptor configuration).

The radial weighting is calculated by using the distance from the sample to the descriptor centre. If distance from the sample is less than the central cell radius plus half the angular cell radius then it contributes to both the central cell and one or more angular cells, otherwise it contributes only to one or more angular cells.

$$w_r(\mathbf{s}) = 1 - \min\left[\frac{\|\mathbf{s} - \mathbf{o}\|}{r_c + 0.5 r_a}\ ,\ 1 \right] \tag{5.9}$$

Where $\mathbf{o}$ is the descriptor centre $\mathbf{s}$ is the sample point location, r_c is the radius of the central cell, and r_a is the radius of the angular cells.

The angular weighting is calculated based on the distance between the sample angle s_ϕ and the angle of the cell's centre ϕ_i. Equation 5.10 shows how the angular weight is determined.

$$w_a(A_i) = 1 - \min\left[\frac{|s_\phi - \phi_i|}{d_\phi}\ ,\ 1 \right] \tag{5.10}$$

Where ϕ_i is the centre of angular cell A_i, and d_ϕ is the spacing between angular cells.

The final contribution to each bin is the product of the two weights multiplied by the gradient

magnitude of the sample. This contribution is also interpolated across adjacent histogram bins within the cells.

As with the R-HOG descriptor, once all the samples in the 16x16 area have been processed the descriptor is normalised using a thresholded normalisation scheme as described in section 5.1.1.

5.2 Training

The training implementation is written in C and consists of two programs. The first is responsible for generating the positive and negative feature vector examples ready for training the support vector machine. The second program has a similar function, except that it creates feature vectors with the x and y coordinates of the top left corner of the area used to create the HOG feature vector [2].

These two programs facilitate the training of the SVM, the stages of which are as follows.

1. Create the positive and negative training examples using the training program.
2. Train the SVM using the training examples
3. Exhaustive search through negative examples across different scales to identify false positives.
4. Retrain the SVM with the original training examples along with the false positive examples.

The final two stages of training are to increase the accuracy of the classifier, as proposed by Dalal & Triggs (2005). However, since the increase is relatively small in proportion to the additional time taken to retrain the SVM, this dissertation follows the training up to step 2.

As demonstrated later, this causes some false positives to be identified by the SVM when testing the classifiers on sample video data. Retraining the algorithm with the false positives will reduce these false classifications.

The SVM is trained to create weight vectors for the R-HOG and C-HOG classifiers using the positive and negative training data from INRIA database, and positive training data from the Poser database combined with negative examples from the INRIA set. This results in 4 weight vectors: two for INRIA and two for the Poser database.

The performances of these weight vectors are explored in section 5.6.

5.3 Training program

There are two versions of this program. One is compiled to generate R-HOG descriptors, and the other is compiled to generate C-HOG descriptors. These are named `train_rhog` and `train_chog`

[2] This area is the top left of a 96x160 window consisting of the 64x128 HOG descriptor window plus a 16 pixel border, as discussed in section 2.3.6.

respectively.

The general operation of the program can be summarized as follows.

training program

```
(01)  parse options
(02)  load positive example file list
(03)  for each positive example
(04)      create hog descriptor
(05)      append to train_pos.txt
(--)  end for each
(06)  load negative example file list
(07)  for each negative example
(08)      create hog descriptor
(09)      append to train_neg.txt
(--)  end for each
```

Table 5.1: Pseudo-code for the HOG SVM training programs.

Once the descriptors have been created ready for training, the two training files are concatenated and the SVM Light's `svm_learn` program is run to find the support vectors for the classifier. This process takes around 6-7 hours for each descriptor type.

Once the `svm_learn` program has finished finding the support vectors a small program was written to build weight vectors for the linear SVM from each of the SVM Light model files. These are used for the multi-scale detector discussed later in this chapter.

5.4 Test program

As with the training program, there are two versions of this test example descriptor generator: `train_rhog` and `train_chog`.

The operation of this program is almost identical to that of the training program, save that instead of creating two separate descriptor output files all of the test examples are combined in to a single file, and the x and y coordinates of the training sample are appended to the end each of the descriptors. Since this is the only real difference between the two programs the details of the program's operation are omitted.

These training programs were used to create a set of testing samples for the INRIA classifiers. The samples were then passed to the SVM Light `svm_classify` program to assess the relative performance of the R-HOG and C-HOG descriptor implementations.

5.5 Multi-scale Classifier

To classify images effectively with HOG descriptors, it is necessary to search an image across different scales using the HOG window. This is accomplished by resampling the image to be searched to different scales using bilinear filtering. Dalal & Triggs (2005) stated that a separating factor of 1.2 between image scales was used to exhaustively search negative images for false positives so this implementation defaults to that scale separation factor.

A completely new program was written using descriptors based on the implementations used in the training and testing programs. The program takes a linear SVM weight vector extracted from an SVM model file created by the `svm_learn` program and uses it to scan across an image at multiple scales to find positive classifications.

A high level overview of the algorithm used in the program is shown in Table 5.2.

ScaleClassify program

```
(01)  parse options
(03)  initialise descriptor type
(02)  initialise the linear SVM
(04)  load image list
(05)  for each image in list
(06)      create set of scales from initial scale to minimum size
(07)      for each scale in search scales
(08)          resample image to new scale
(09)          compute edges
(10)          analyse image
(--)      end while
(11)      filter positives
(12)      optionally save output image
(13)      save metadata for search
(--)  end for
```

Table 5.2: Pseudo-code for the multi scale HOG classifier program.

The program first configures itself based on any options specified when the program was run. There are a few required options and number of optional ones that allow more control over the classifier program.

First the program selects the appropriate descriptor type to use (either R-HOG or C-HOG) then loads the linear SVM weight vector that was specified when the program was run. The size of the weight vector must match with the size of the HOG window feature vector, or classification cannot take place.

Next the program opens a file containing a list of images specified by the user for processing.

For each of these images, the program creates a set of scales ranging from the initial search scale configured by the user (the default is 1, *i.e.* start with the original image resolution) down to a minimum window size (the default is 96x160).

For each of these scales the image is resampled, image gradients are calculated, and the image is processed by the HOG sample window. The feature vector from the window is processed by the linear SVM and a classification is made. If the classification is positive, the bounding box and classification value are saved to a list for processing later.

The HOG window is moved across the image using a specified search stride (the default is 8 pixels). The total number of HOG windows in the x and y directions that are processed within a given scale can be determined using the equation 5.11.

$$n_x = \frac{I_w - w_w + s_s}{s_s} \tag{5.11}$$

$$n_y = \frac{I_h - w_h + s_s}{s_s} \tag{5.12}$$

Where I_w and I_h are the image width and height at the current scale, w_w w_h are the HOG sample window width and height respectively, and s_s is the window search stride (64x128 in the case of this implementation).

If there is any left over space in the image left over from fitting the grid of windows with the current search stride, then the whole grid of windows is moved by half the left over distances in the x and y directions. This effectively centres the grid of windows within the image being searched.

Once all window positions have been classified for all scales for the current image, the positive bounding boxes that were detected are processed using a non-maximum suppression scheme (by default). Since the bounding boxes found at different scales will have different relative sizes, the distance between box centres and the width of the bounding boxes are used to determine if an adjacent bounding box is to be considered a neighbour for non-maximum suppression.

Finally, unless the program was run with an option to disable it, an output image is saved with bounding boxes drawn to show all the positive classifications and their classification strengths.

A text file containing metadata information for the search is also saved so that the positive classifications that were rejected can be perhaps be reprocessed, or the experiment can be re-run using the same parameters for testing purposes. This metadata file contains the list of all positive classifications found, the search parameters, and the original command line options used to run the program.

5.5.1 Optimisation

The program contains an optimisation that speeds up the window classification greatly if the search stride used by the window is a multiple of the stride used to separate descriptor blocks within the sample window.

The basic idea behind this optimisation is that if the window were to shift 8 pixels to the right and the block stride is 4, then all but two columns within the HOG window can be reused as the area they cover is still within the window area and at a valid position for the next window. The same holds for a vertical shift of 8 pixels for the window, but in the case of a vertical shift all but two descriptor block rows can be reused.

When this optimisation is used, any invalid blocks from each row are moved to the opposite end for horizontal shifts, and any invalid rows are moved to the opposite end of the window for vertical window shifts. When these blocks are moved they are flagged as invalid to notify the window that they need to be recalculated before appending them to the HOG window feature vector.

To achieve this, each row of blocks within the window is represented by a linked list of descriptors (one descriptor for each column in the row), and linked list rows holding the blocks are in turn represented by a linked list of rows. This allows efficient moving of descriptor blocks from one side of the window to the other for horizontal shifts, and linked lists rows to the either end of the window for vertical shifts.

5.5.2 Structure

There are several classes used in the multi-scale classifier program. This section gives an overview each of the main classes and structures used by the program.

Pyramid

This class encapsulates the image pyramid searching algorithm shown in lines 6-13 of the program pseudo-code. It contains a `HOGWindow` and `HOGShiftWindow` class for each of the `RHOG` and `CHOG` descriptor structures, an instance of `LinearSVM` and various parameters to control the search process.

The class determines the best type of HOG window implementation to use based on the options passed to it when the program was initialised. When the search stride is a multiple of the window block stride then it uses a `HOGShiftWindow` by moving it only one search stride step in a direction at a time, otherwise the `HOGWindow` implementation is used.

The search is made from the top left of the image shifting the window to the right until it reaches the right edge of the image, then it shifts the window down one search step and moves toward the left side of the image. This is the most efficient search pattern for the shift window, s only a few rows or columns need to be recalculated at all but the first window position.

Image

The `Image` class is a slightly modified version of the one used for the superimpose program discussed in section 4.2.5, but only supports 24bit RGB colour images and the PPM image file

format. Reusing this class saved a lot of development time, as it contains several useful methods for image manipulation.

HOGWindow

This template class is a HOG descriptor window that is optimised for classifying random areas within an image. It contains a single descriptor class instance that it reuses as while processing the samples within the window area.

HOGShiftWindow

If an exhaustive search of an image scale is required and the search stride is a multiple of the HOG window block stride, then this class is very efficient for classifying adjacent areas of an image. This class is an implementation of the optimisation identified in section 5.5.1 so is efficient when the hog window only needs to shift by few block strides for each classification.

LinearSVM

This class contains a simple implementation of a linear Support Vector Machine, and provides a method to load weight vectors from file to be used in feature classification.

5.5.3 The `RenderMD` Program

In addition to the `ScaleClassify` program there is another program called `RenderMD`. This program is capable of processing metadata created by the scale classifier program to render bounding boxes on top of the metadata's original source image.

If necessary the classifications within the metadata can be refiltered to use a different method or all the results can be displayed without refiltering. This is possible because the metadata contains all the positive classifications detected by the scale classification program in addition to those rejected by any filtering that was being used at the time.

The real advantage to this program is that it can process results from up to 4 different metadata files so that comparisons can be made on the same image. This reprocessing is useful for varying filter settings without having to reprocess an image again using the `ScaleClassify` program.

However, the program can only display the classifications that were originally accepted by the scale classifier using the classification threshold parameter, so if a larger number of classifications are required than are available within the metadata file, then the image will need to be reprocessed by the `ScaleClassify` program.

5.6 Results

5.6.1 INRIA Classifier

Once the INRIA classifier had been trained, the `svm_classify` program from SVM Light was run to quantify the relative performance of the R-HOG and C-HOG descriptor implementations. The results of the program are shown in figure 5.1.

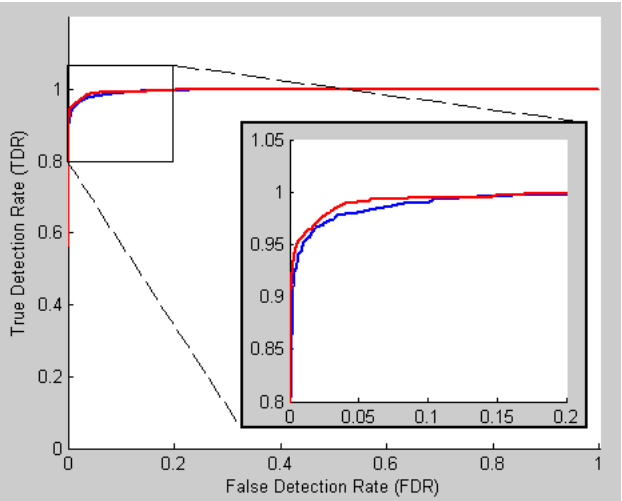

Figure 5.1: ROC curve showing relative performances of the R-HOG descriptor (shown in red) and C-HOG descriptor (shown in blue) on the INRIA database.

The graph in figure 5.1 shows that the R-HOG descriptor performs slightly better than the C-HOG descriptor.

5.6.2 Poser Database Performance

The estimated performance reported by the `svm_learn` program of the classifiers trained using the Poser Database is not as good as the classifiers trained using the INRIA database. Table 5.3 shows the estimates determined by `svm_learn` when each of the INRIA and Poser based classifiers were trained.

Classifier	Estimated Precision	Estimated Recall
INRIA C-HOG	99.20%	88.37%
INRIA R-HOG	96.68%	95.74%
Poser C-HOG	71.63%	84.09%
Poser R-HOG	71.96%	83.88%

Table 5.3: Estimates of recall and precision for the descriptor types of the INRIA and Poser classifiers.

This shows that the Poser Database is more difficult for the linear SVM to learn. The use of a non-linear kernel based Support Vector Machine might improve the performance of the Poser classifiers at the expense of higher computation times.

5.6.3 Multi-scale Detection Results

The multi scale detector was run on 5 images taken from the positive testing images in the INRIA database. These contain one or more people at varying scales and poses, and make a good test of the classifiers.

The results of the processing these images with the INRIA and Poser trained classifiers are listed in appendix A.

The INRIA trained classifiers performed best, so they were also used to try and find a human figure in a motion capture dataset from 4 different viewing angles (Sigal, Bhatia, Roth, Black & Isard 2004). Some sample frames from these results are shown in figure 5.2.

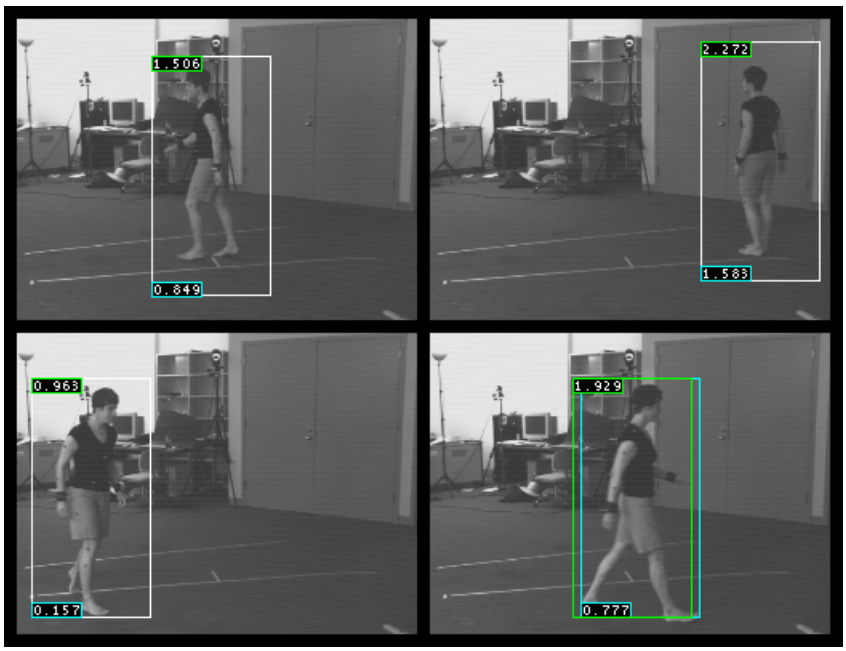

Figure 5.2: Sample frames from INRIA trained video classification. A green box shows the bounding box estimate from the INRIA R-HOG classifier and a blue box shows the estimate from the C-HOG classifier. A white box is when both classifiers agree on the bounding box position and size.

The green and blue bounding boxes in the images represent the bounding box predicted by the INRIA R-HOG and C-HOG classifiers respectively. Since the videos contain only one person and the scale doesn't vary much between images, frames were analysed at only one scale to speed up classification and the results were filtered by selecting the strongest positive classification

value. The scale was chosen to be where the person in the video fits comfortably within the HOG detection window: at a scale of around one third of the original frame image size.

5.7 Discussion

5.7.1 Descriptor Performance

The results presented in Dalal & Triggs (2005) show that their implementation of the C-HOG descriptor performs slightly better than their R-HOG descriptor implementations. However, the performance of the C-HOG and R-HOG descriptors in this implementation actually show the opposite of this.

Differences between the implementation is in Dalal & Triggs (2005) and the one presented in this dissertation may well be the cause of this, and it is also possible that the retraining of the SVM with false positives may increase the performance of the C-HOG descriptor to be better than that of the R-HOG descriptor.

The results on the 5 test images and the motion capture video sequence from Sigal et al. (2004) show that the R-HOG descriptor tends to give a 'best guess' but is susceptible to false positives, whereas the C-HOG descriptor is more accurate but at times will be able to classify anything at all.

This can be seen more clearly in the `Cam1` sequence from the video set where the C-HOG descriptor will fail to find any positive classifications over sets of frames, but the R-HOG still continues to classify areas of the image (albeit not always correct).

5.7.2 Poser Database Trained Classifier

The results in the 5 test images using the classifiers trained using the Poser Database (see appendix A.2) show that the performance is not particularly good. Both the Poser R-HOG and C-HOG based classifiers find more false positives than their INRIA equivalent. This is most likely due to the wide range poses in the positive training data making it difficult to train the SVM, as discussed previously in section 5.6.2.

Chapter 6

Conclusion

The results from the SIFT implementation demonstrate that the program works successfully, although the keypoints are not as robust as keypoints generated by the demo program from Lowe (2004). This is most likely due to differences in the convolution method used in and descriptor creation. However, the keypoints found by the SIFT program presented by this dissertation are sufficiently accurate for matching.

The number of extrema found by the two convolution methods used in the SIFT implementation were slightly different. The FFT based convolution method produced fewer keypoints overall, but this could be due to the minimum contrast threshold for extrema being too high for this method.

The similarities between the SIFT algorithm and the recently published Histograms of Oriented Gradients (HOG) algorithm are mainly in their descriptors. Both algorithms use blocks of threshold normalised several [1] edge orientation histograms and this seems to be a good way of representing local pixel information in a robust way suitable for object classification.

In most other respects however, the HOG algorithm's use of its descriptors are quite different. SIFT uses extrema in the difference-of-Gaussian function of the scale-space representation of an image to sparsely position its descriptors. In contrast, HOG uses a very dense grid of overlapping descriptors within a detection window to create a feature vector that can be classified by a Support Vector Machine.

Although the HOG descriptors are shown to perform better at classifying the positive examples than several other descriptor types (Dalal & Triggs 2005), the overall algorithm does not handle the issue of rotational invariance on the objects it attempts to classify. This means that if an image were rotated by 45 degrees for example, then the same objects that would normally by classified by the detection window would no longer be detected.

Another issue with the HOG algorithm is that the dense grid of descriptors in the detection window creates a very high dimension feature vector that makes training the Support Vector

[1] The histogram block is normalised, thresholded so that values are no larger than a specified maximum, then normalised again.

Machine a rather slow and resource demanding process. Once the SVM is trained however, the classification of feature vectors is quite fast.

The Poser Database presented by this dissertation has been shown to be a challenging set to classify, and suggests that the HOG algorithm in its current state might not be suited for classifying the more varied poses that the database contains. A parts based model as suggested by Dalal & Triggs (2005) might increase the HOG algorithm's performance with this database.

The algorithms explored in this dissertation both have advantages and disadvantages. The SIFT algorithm is quick at processing large images due to the way it places its descriptors at sparsely distributed interest points, but the algorithm is quite complicated to implement for object recognition compared to the HOG algorithm.

The simplicity of the HOG algorithm and its good performance at classification makes it a good choice for object classification, but its sensitivity to rotation limits the range of applications that it can be used in.

6.1 Future Work

The method of convolution used to create the image pyramid influences the overall number of keypoints created by the SIFT algorithm. It would be interesting to explore the effect of different convolution methods on the number of robust keypoint descriptors produced by the algorithm.

An efficient solution to the problem of rotational invariance for the HOG algorithm would be a challenging problem to solve, but would be one that could greatly broaden the number of applications the algorithm can be used in.

Exploring some of the possible areas of improvement identified by Dalal & Triggs (2005) would also be worthwhile and could speed up the algorithm significantly on classifying large areas. Methods that can quickly identify smaller areas of interest in a large image would be particularly useful and could reduce computation time enough for HOG to be used in real time classification tasks.

Appendix A

Classification Results

The classifiers were both run over 5 images selected from the INRIA database. They each contain different numbers of people and at different levels of occlusion. The first set shows the results from the classifier trained using the INRIA database, and the second set shows the results from the classifier trained using the Poser database.

A.1 INRIA Database Classifier Results

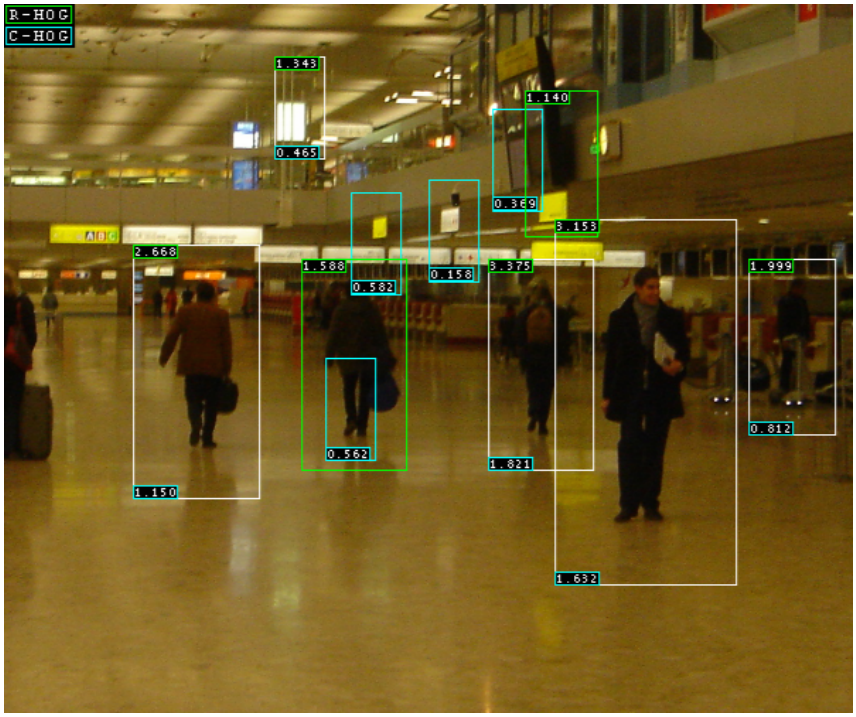

Figure A.1: Results of INRIA trained multi-scale classification using on test image 1.

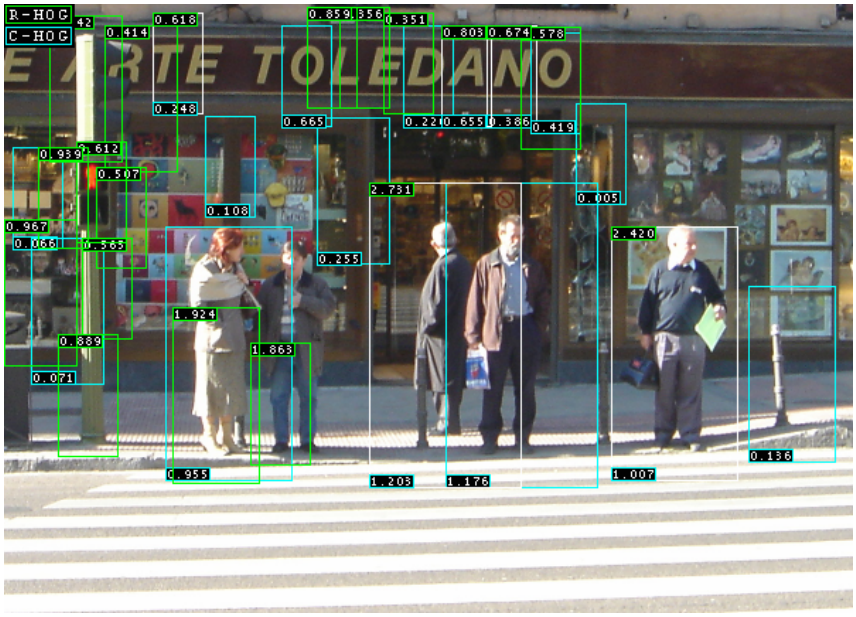

Figure A.2: Results of INRIA trained multi-scale classification using on test image 2.

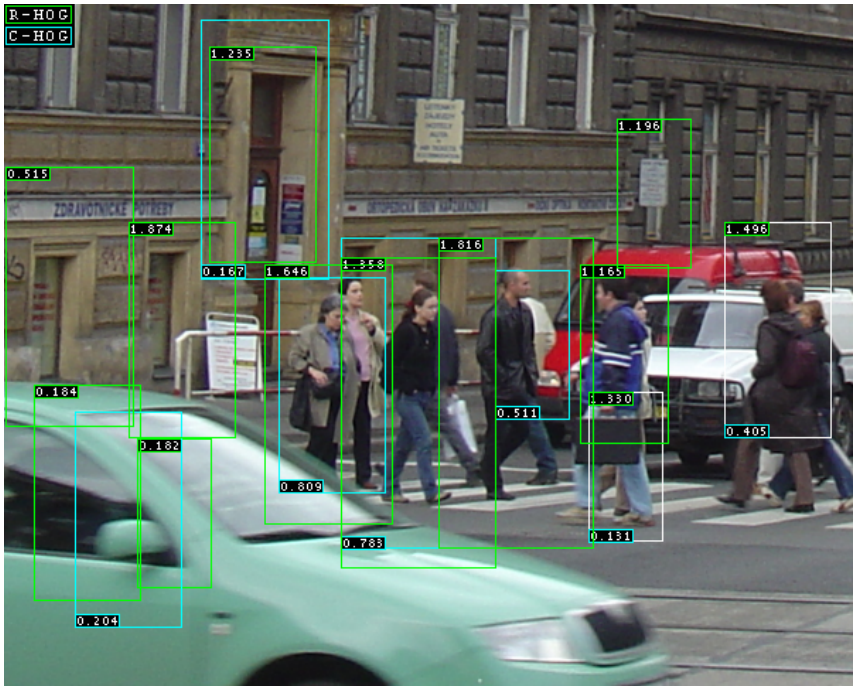

Figure A.3: Results of INRIA trained multi-scale classification using on test image 3.

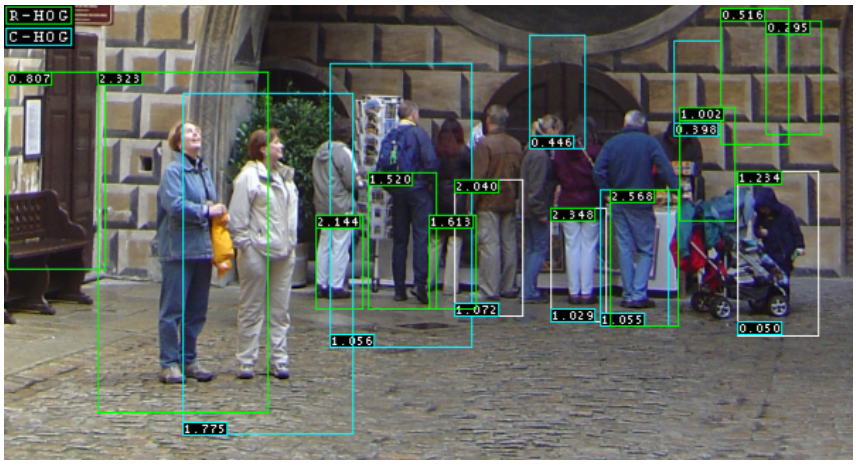

Figure A.4: Results of INRIA trained multi-scale classification using on test image 4.

Figure A.5: Results of INRIA trained multi-scale classification using on test image 5.

A.2 Poser Database Classifier Results

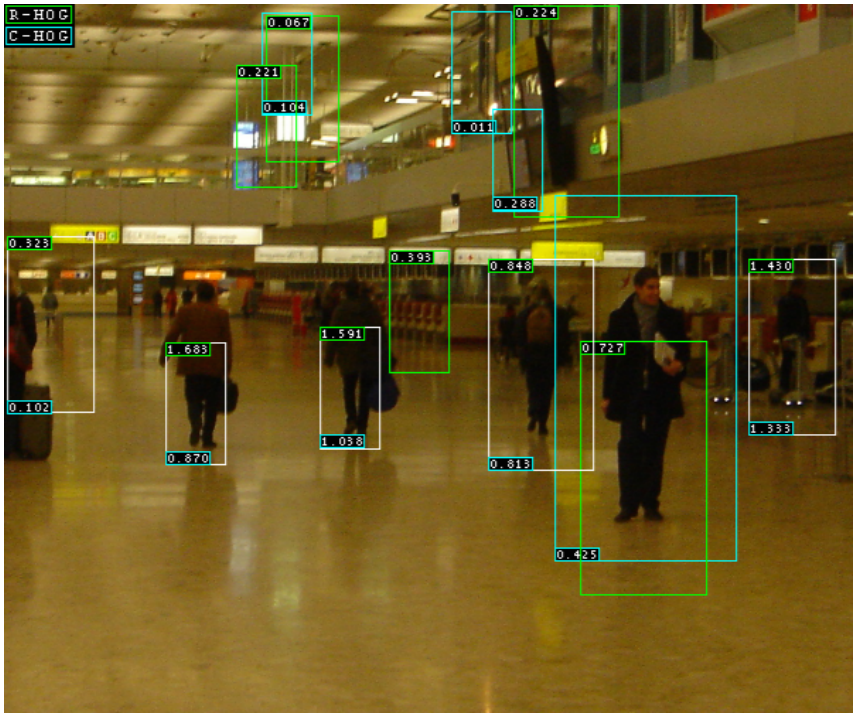

Figure A.6: Results of Poser trained multi-scale classification using on test image 1.

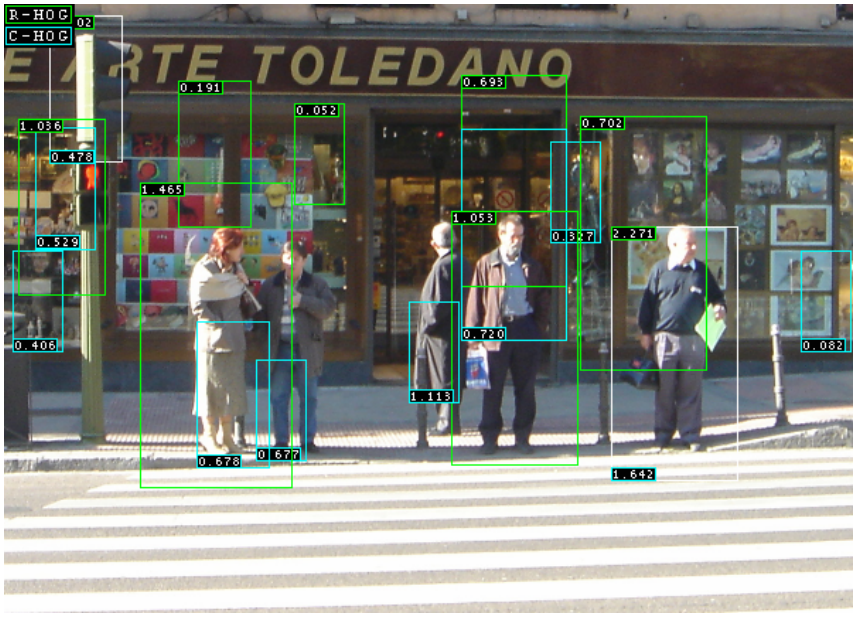

Figure A.7: Results of Poser trained multi-scale classification using on test image 2.

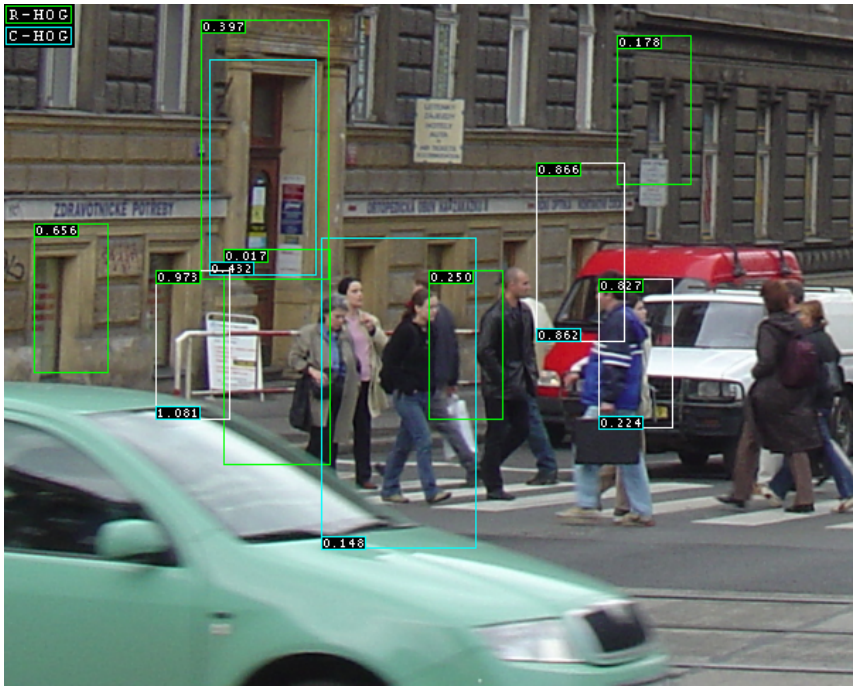

Figure A.8: Results of Poser trained multi-scale classification using on test image 3.

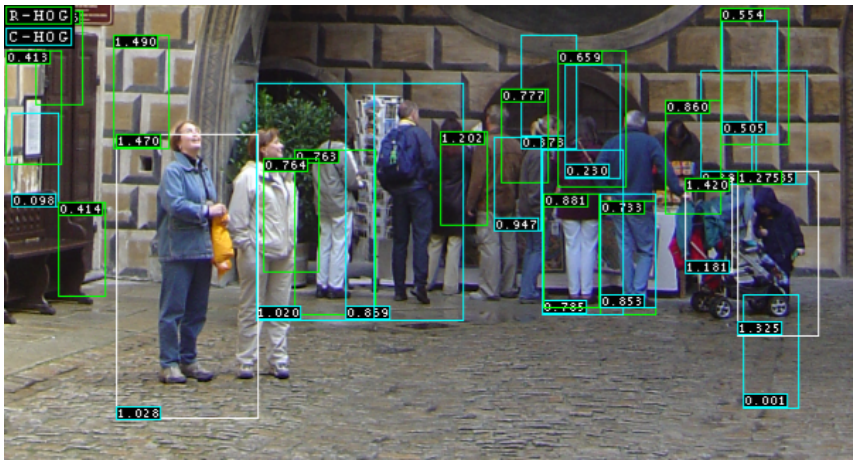

Figure A.9: Results of Poser trained multi-scale classification using on test image 4.

Figure A.10: Results of Poser trained multi-scale classification using on test image 5.

Appendix B

SIFT Program Source

The following pages list the complete source code for the SIFT implementation. The program is run in the following way:

```
SIFT inputfile outputfolder [options]
```

Where `inputfile` is a PNG or (binary) PGM image, `outputfolder` is the destination folder for all the output files generated by the program. All output files are named using the original file name, plus an appropriate suffix for the type of file, *i.e.* an example output file name might be `inputfile_desc.txt` for the plain text descriptor data file.

The options can be any of the following:

`-c [float]` Specifies a minimum contrast threshold to use for rejecting extrema. Any extrema in the difference-of-Gaussian function with values less than the minimum contrast value are rejected. The default value for this parameter is 0.02.

`-r [float]` Specifies a principle curvature ratio threshold to use for rejecting extrema. Any extrema point that has a principle curvature ratio of equal greater than this value is rejected. The default value for this parameter is 10.

`-o [int]` Specifies the number of octaves to use for analysing the image. The default value for this parameter is 4.

`-i [int]` Specifies the number of intervals to use for analysing the image. The default value for this parameter is 3.

`-f` Enables FFT based convolution for creating the Gaussian scale-space pyramid for this source image. The default is not to use FFT based convolution.

`-p` Save image Gaussian scale-space pyramid and difference-of-Gaussian pyramids created from the source image to file. The default setting is not to save the pyramids.

`-p` Save image Gaussian scale-space pyramid and difference-of-Gaussian pyramids created from the source image to file. The default setting is not to save these pyramids.

`-s` Enables subsampling of image gradients when creating the SIFT descriptors. This is where samples at positions with fractional parts are sampled using bilinear interpolation based on the surrounding image gradient data. The default is not to interpolate local gradient data when creating descriptors.

Appendix C

Test and Train Program Source

The `Train` and `Test` programs are modified versions of those contributed by M. Pawan Kumar for training a linear support vector machine under SVM Light ready for classification.

The main areas of change are within the `HOG.cpp` and `CHOG.cpp` files in their respective descriptor block creation code. There are also some additions such as a `vect2` type to simplify the C-HOG descriptor calculation.

The other change is that instead of compiling to a single file, the make file is configured to create two separate binaries: one for C-HOG and one for R-HOG descriptors.

C.1 Train Program

The following pages are the main source code files for computing descriptors to be used with the SVM Light `svm_learn` program for training. The programs are run in the following way:

```
train_chog pos_examples.txt neg_examples.txt mode
train_rhog pos_examples.txt neg_examples.txt mode
```

Where `pos_examples.txt` is a file containing the set of normalised (96x160) positive example files, and `neg_examples.txt` is a file containing a set of negative images to sample random patches from to create negative examples. The `mode` parameter is set to 0 for all runs in the experiments for this dissertation.

C.2 Test Program

The following pages are the main source code files for computing descriptors to be used with the SVM Light `svm_classify` program to assess performance. The program is run in the following way:

```
test_chog pos_examples.txt neg_examples.txt testoutput.txt
test_rhog pos_examples.txt neg_examples.txt testoutput.txt
```

Where `pos_examples.txt` is a file containing the set of normalised positive example files, and `neg_examples.txt` is a file containing a set of negative images to sample random patches from to create negative examples, and `testoutput.txt` is the name for the output file.

The normalised positive example images are usually 96x160 in resolution which are made up of a centred 64x128 sample area for the detection window a plus 16 pixel margin around it, but the program supports positive examples with margins of any number greater than 0 and will center the 64x128 sample window in the example image when creating the feature vectors.

C.3 Shared Source Code

The following pages contain source files that are shared between both programs and contain all the code for actually generating the HOG descriptors and feature vector as well as some more low-level memory management and utility functions.

Appendix D

ScaleClassify Program Source

The following pages contain source files for the `ScaleClassify` program. The program can be run in the following way:

```
ScaleClassify (-chog/-rhog) svm_weight.txt files.txt outputpath [options]
```

Where either `-chog` or `-rhog` must be specified for C-HOG or R-HOG classification respectively; `svm_weight.txt` is the file containing the appropriately sized weight vector; `files.txt` contains the set of files to classify; ouputpath is the destination folder for the output files. The options can be any of the following:

`-t [float]`	Threshold value for classification (default: 0).
`-w [int]`	The minimum width for levels in the scale pyramid (default: 96).
`-h [int]`	The minimum height for levels in the scale pyramid (default: 160).
`-i [float]`	Initial scale for scale pyramid (default: 1).
`-z [float]`	Scale separation between levels in the scale pyramid (default: 1.2).
`-s [int]`	HOG window search stride (multiples of 4 are fastest, default: 8).
`-m [int]`	The maximum number of levels for the pyramid (default: no maximum).
`-fnms`	Filter positive classifications using non-max suppression (default filter type).
`-fmax`	Filter by just considering largest response as the only positive.
`-fnone`	Don't apply a filter to positive classifications.
`-d`	Only save metadata and don't save an output image with bounding boxes drawn for positive classifications.

Appendix E

RenderMD Program Source

The following pages contain the source code for the `RenderMD` program. The program can be run in the following way:

```
RenderMD files.txt outputdir -t[r/b/g/y] name1 [-t[r/b/g/y] name2 ...] [options]
```

Where `files.txt` is the set of files to process, `outputdir` is the output directory to place the rendered images, and: `-tr name`, `-tg name`, `-tb name` and `-ty name` are the tag names used for drawing classification boxes in red, blue, green, or yellow respectively. Additional options may be any of the following:

`-fnone`	Refilter classifications using no filter (*i.e.* display positives + rejections).
`-fnms`	Refilter classifications using non-max suppression.
`-fmax`	Refilter classifications choosing only maximum response.
`-n [float]`	Specifies the minimum nearest neighbour overlap ratio threshold. This is the minimum percentage overlap between two boxes that must occur before they are considered neighbours. The value must be in the range [0,1] (default: 0.5).
`-b [float]`	Use this additional threshold bias for results (default: don't use additional threshold).
`-g [image]`	Draw the specified PPM image at the top left of the all output images. This is useful for drawing a key defining the what each classification box colour represents. The default is to not display an image.

The metadata is loaded based on the name given to the specified tag. When after loading an image for processing, the tag name is appended to the directory that the image is within to find the metadata associated with the image. For instance if a tag was specified as `-tr rhog` and the

full path name of the image being processed was `C:\images\image.ppm` then for the metadata for the `rhog` tag would be loaded from `C:\images\rhog\image_metadata.txt`.

Appendix F

PoseLib Source code

The following pages contain the source code for the main classes and methods from the set of PoseLib scripts. The method used to generate the pose database using these scripts is discussed in detail in chapter 4.

Appendix G

Superimpose Program Source

The following pages contain the source code for the `superimpose` program used to create the normalised versions of the poser training images by superimposing them on random background regions. The program is run as follows:

```
superimpose figures.txt backgrounds.txt [options]
```

Where the additional program options can be the following:

`options=file.txt`	Load options from the specified file.
`subregion=x,y,w,h`	Use this subregion area from the source images, where `x` and `y` is the position of the top left of the subregion, with `w` and `h` specifying the width and height of the subregion. The default is to use the original image dimensions.
`resample=w,h`	Resample the subregion to this size.
`innermargin=m`	Use this value for the inner margin (see section 4.2.5).

Bibliography

Adnin, M. M. & Smith, W. (2005), 'Mayang's free texture library',
Website: `http://www.mayang.com/textures/`.

Dalal, N. & Triggs, B. (2005), Histograms of oriented gradients for human detection, *in* C. Schmid, S. Soatto & C. Tomasi, eds, 'International Conference on Computer Vision & Pattern Recognition', Vol. 2, INRIA Rhône-Alpes, ZIRST-655, av. de l'Europe, Montbonnot-38334, pp. 886–893.
*http://lear.inrialpes.fr/pubs/2005/DT05

El-Maraghi, T. F. (2004), 'MatLab SIFT tutorial', University of Toronto
website: `http://www.cs.toronto.edu/~jepson/csc2503/`.

Frigo, M. & Johnson, S. G. (2005), 'The design and implementation of FFTW3', *Proceedings of the IEEE* **93**(2), 216–231. special issue on "Program Generation, Optimization, and Platform Adaptation".

Gustavsson, C., Hui, A. & Turitzin, M. (2004), 'Improving SIFT features / finding planes in hallways', website: `http://robots.stanford.edu/cs223b04/project9.html`.

Hearn, D. & Baker, M. P. (2004), *Computer Graphics: C Version with OpenGL*, Pearson Prentice Hall.

Iijima, T. (1959), Basic theory of pattern observation, *in* 'Papers of Technical Group on Automata and Automatic Control', IECE, Japan. (in Japanese, cited in Weickert et al. (1999)).

Joachims, T. (1999), *Making large-Scale SVM Learning Practical. Advances in Kernel Methods - Support Vector Learning, B. Schlkopf and C. Burges and A. Smola (ed.)*, MIT-Press, chapter 11.

Ke, Y. & Sukthankar, R. (2003), PCA-SIFT: A more distinctive representation for local image descriptors, Technical report IRP-TR-03-15, Intel.

Koenderink, J. J. (1984), 'The structure of images', *Biological Cybernetics* **50**, 363–370.

Lindeberg, T. (1994), 'Scale-space theory: A basic tool for analysing structures at different scales', *Journal of Applied Statistics* **21**(2), 224–270.

Lowe, D. G. (1999), Object recognition from local scale-invariant features, *in* 'International Conference on Computer Vision', Corfu, Greece, pp. 1150–1157.

Lowe, D. G. (2004), 'Distinctive image features from scale-invariant keypoints', *International Journal of Computer Vision* **60**(2), 91–110.

PNG Development Group (2005), 'Libpng', Website: `http://www.libpng.org/pub/png/libpng.html`.

Sigal, L., Bhatia, S., Roth, S., Black, M. & Isard, M. (2004), 'Tracking loose-limbed paople', *Proc. IEEE Conf. on Computer Vision and Pattern Recognition* .

Weickert, J. A., Ishikawa, S. & Imiya, A. (1999), 'Linear scale-space has first been proposed in Japan', *Journal of Mathematical Imaging and Vision* **10**(3), 237–252.

Witkin, A. P. (1983), Scale–space filtering, *in* 'International Joint Conference on Artificial Intelligence', Karlsruhe, Germany, pp. 1019–1022.