**Automated Surveillance using Background Subtraction and Shape Analysis**

Final Report for CS39440 Major Project

*Author*: Alexander Shaun O’Donnell ([alo14@aber.ac.uk](mailto:alo14@aber.ac.uk))

*Supervisor*: Bernie Tiddeman (bpt@aber.ac.uk)

16th April 2016

Version 1.0 (Draft)

This report is submitted as partial fulfilment of a BSc degree in  
Computer Science (Inc Integrated Industrial and Professional Training) (G401)

Department of Computer Science

Aberystwyth University

Aberystwyth

Ceredigion

SY23 3DB

Wales, UK

**Declaration of originality**

I confirm that:

* This submission is my own work, except where clearly indicated.
* I understand that there are severe penalties for Unacceptable Academic Practice, which can lead to loss of marks or even the withholding of a degree.
* I have read the regulations on Unacceptable Academic Practice from the University’s Academic Quality and Records Office (AQRO) and the relevant sections of the current Student Handbook of the Department of Computer Science.
* In submitting this work I understand and agree to abide by the University’s regulations governing these issues.

Name Alexander Shaun O’Donnell



Date 16/04/2017

**Consent to share this work**

By including my name below, I hereby agree to this dissertation being made available to other students and academic staff of the Aberystwyth Computer Science Department.

Name Alexander Shaun O’Donnell



Date 16/04/2017

**Acknowledgements**

I am grateful to my supervisor Bernie Tiddeman for his code suggestions/recommendations during development and checking sections of the report.

I’d like to thank my family for proof-reading my report. And the actors who appeared in the videos and images within the datasets used.

**Abstract**

Computer vision and image interpretation are evolving topics in the world of technology. Many institutions have invested in this field of research for security solutions, as technologies such as facial recognition, object placement and pedestrian detection can provide potential safety nets to the challenges they face. The rise of automation technologies is reducing human error, which is something a high-risk field such as security could do without. This project looks at pedestrian detection/tracking, the idea that a program could take up the role of monitoring live CCTV footage and keep a record of pedestrians and other movements. We can achieve this by using what is known as a classifier, an algorithm that can lead software to make categorical decisions. By providing examples of how pedestrians may appear in a video, the classifier can distinguish between what is and what isn’t a person. Assuming the cameras used in the videos are static, moving objects can be separated from a non-moving environment using background subtraction. Combining these tools, we can extract large moving objects from video footage and use the classifier to see whether the object is a person or not. Rather than having a police officer go through hours of CCTV footage, this program can view the video instead, recording any passenger or movement that occurred with a timestamp of when it happened.

**Contents**

1. Background, Analysis & Process 7

1.1. Background 7

1.1.1. Security and Automation 7

1.1.2. Detection through Vision 7

1.1.3. Pedestrian Detection 8

1.1.4. Motion Detection 10

1.2. Analysis 11

1.2.1. Objectives 11

1.2.2. Datasets and Frameworks 12

1.2.3. Motion Detection Choice 13

1.2.4. Feature Selection and Classification 13

1.2.5. Record Log 14

1.2.6. Computer Vision Issues 14

1.2.7. Security 15

1.3. Process 15

1.3.1. Methodology 15

1.3.2. Support tools 16

2. Design 17

2.1. Overall Architecture 17

2.2. Features List and feature design plans 18

2.2.1. Input threshold values and paths into a menu 18

2.2.2. Loading the training data 18

2.2.3. Outlining shape contours 19

2.2.4. Building the pedestrian models 20

2.2.5. Training the pedestrian finder 20

2.2.6. Running the input video 20

2.2.7. Applying BGS 21

2.2.8. Classifying shapes 21

2.2.9. Record findings 21

2.3. Pedestrian model pseudocode 22

2.3.1. Search for the head 23

2.3.2. Torso 23

2.3.3. Waist 23

2.3.4. Feet 24

2.3.5. Shoulders 24

2.3.6. Elbows 25

2.3.7. Hands 25

2.4. Final classes 26

2.4.1. Data Structures 27

2.4.2. Function Allocation 28

3. Implementation 28

3.1. BGS 28

3.1.1. Setup 28

3.1.2. KNN vs MOG 29

3.1.3. Reducing Noise 31

3.1.4. Problems 31

3.2. BlobDetector 33

3.2.1. Setup 33

3.2.2. Contours and Hulls 33

3.2.3. Shape Extraction 33

3.3. PeopleFinder 34

3.3.1. Setup 34

3.3.2. File handling 35

3.3.3. Create skeleton function 35

3.3.4. Head and torso detection 35

3.3.5. Waist and feet detection 36

3.3.6. Shoulder detection 36

3.3.7. Elbow detection 37

3.3.8. Hand detection 37

3.3.9. Dealing with bad shapes 38

3.4. RecordLog 38

3.4.1. Setup 38

4. Testing 39

4.1. Overall Approach to Testing 39

4.2. Unit Testing 39

4.3. System Testing 40

5. Critical Evaluation 41

6. Appendices 42

A. Third-Party Code and Libraries 42

B. Ethics Submission 43

C. Code Samples 44

Annotated Bibliography 45

# Background, Analysis & Process

## Background

### Security and Automation

As an interdisciplinary field, computer vision has provided many solutions to security. From facial recognition for matching perpetrators to crowd analysis for tackling gang culture and hooliganism, it synergises well with high risk environments. Originating in academic environments, computer vision made its way into the world of security as early as 2001. Police in Tampa, Florida used facial recognition software during the Super Bowl XXXV in January 2001 to detect known criminals and terrorists attempting to attend the event. The software, known as “FaceIt” was developed by Identix and managed to find 19 people with pending arrest warrants [1].

Computer Vision is an abstract topic, the idea of getting a system to understand what is being depicted in an image is particularly useful for security. Many institutions have researched this field as it can provide solutions to some of the challenges faced by the industry. For example, in 2015 the Ministry of Defence (MoD) cooperated with QinetiQ to develop middleware software architecture that would be used alongside sophisticated sensing equipment to perform detection and behaviour classification in a specified area [2].

A typical surveillance setup would involve a security guard or member of law enforcement observing several cameras. These cameras are usually static, where a single camera will maintain the same viewing angle for its duration. This results in institutions investing in many cameras to cover all angles of a premises, and can be straining on the employee tasked to observe them all. This is amplified in isolated or rural environments, where cameras may only observe a small number of pedestrians every hour leading to fatigue and thus an increase in human error. Previous studies have coined the term ‘boredom factor’, where an observer’s levels of concentration degrade over time from viewing the same monotonous feeds for long hour shifts [3].

Part of the job for police officers is to review pre-recorded CCTV footage recorded for potential evidence, with footage lengths ranging from minutes to days. In 2016 CCTV operators from the Suffolk Constabulary and the Metropolitan Police observed more than 1100 hours CCTV footage in the search for a missing person [4]. The hours spent on this particular case involved detection rather than recognition, as they do not include identifying the individuals found. The ability to automate this process would save time, resources and manpower that could be better spent in agencies such as these. It is vital that our law enforcement adapts and becomes more efficient at retrieving and storing information for the sake of the victims and our privacy. To automate the process of collecting timestamps of people appearing in videos, we can look no further than the field of pedestrian detection.

### Detection through Vision

Pedestrian detection is a sub topic of object detection, and has become a fundamental task in video surveillance. The ability to detect and track a human provides great support for reducing trespassers in restricted areas. Typical detection applications in computer vision rely on a classifying algorithm, and some form of machine learning with training/testing phases.

The training phase provides the program with examples of how the object it’s looking for may appear (a person in our case). These examples come in the form of a dataset, a collection of images representing the object to detect. Some datasets include what are known as negative examples, these images will usually contain a different object or no object altogether. This opens up the classifier to different categories so it can overcome common appearances that interfere with its decision making. For example, including images of cars or animals would help the program differentiate pedestrians from other objects relevant to the environment.

There are a wide range of methods for detecting and classifying objects, one of which is the Viola-Jones object detection framework. Although the framework was initially intended for face detection, there have been adaptations to suit pedestrian detection. Viola-Jones is also referred to as a ‘Haar Cascade Algorithm’ since the framework relies on Haar-like feature selection and cascade architecture for classification.

Haar-like feature works by looking at the difference in the sum of the pixel intensities in specific regions of the image. These rectangular regions are positioned relative to a boundary box, this window is scanned across the image to find intensity changes that appear within the regions. Since faces share common appearances such as changes in colour between the eyes and cheek, it is worth positioning the rectangles in the upper areas of the bounding box. In the Fig 1, the rectangle in the middle compares the difference in colour between the eyes/eye brows and the nose/cheeks. The rectangle on the right compares the changes in colour for either side of the nose bridge, assuming that these areas will have greater changes in intensities.



Fig 1. Rectangular regions used for detecting eyes during face detection. (Source: <http://docs.opencv.org/trunk/d7/d8b/tutorial_py_face_detection.html> )

By applying feature finding techniques to a training set, the program can get a better idea of how faces may appear, as well as a threshold for distinguishing faces from non-faces. There will always be a false positive risk with detection algorithms, for example perceived images without a face, that happen to have darker regions where the eyes would be could still possibly be classified as a face. Which is why many classifiers take in a threshold or a Gaussian of intensity values to differentiate faces from noisy images.

Viola-Jones relies on cascade classification, a combination of many classifiers in order of complexity. Each classifier is trained on the same sample, if one classifier rejects the image than the classification breaks out, reducing the processing time so it can search the next sample. This also enables the cascade to focus on more promising regions in the image that are more likely to contain the features, increasing the detection rates as the search space is reduced [5].

### Pedestrian Detection

Pedestrian detection can be achieved using Haar-like features, changes in pixel intensity in simple environments can be seen on people as well as faces. The rectangles can be rearranged within the search window, looking for intensity changes around the head, hands and feet. This assumes the appearance of the person stands out from the environment or that an appropriate threshold is used.

Local Haar features can be combined to build a pedestrian model. Recently, pedestrian detection was achieved by constructing square body shape models from gradient magnitudes and colour channels (Zhang et al, 2014) [6]. Their statistical model takes in images and computes an average edge map of the sample. The resulting edge map is divided into cells, which construct the square regions that make up the body. Their body model is divided into three sections, the head, torso and legs. This is useful as it reduces the search space for features, since it would be safe to assume the feet features always appear below the torso. The core of their project uses multi-channel descriptors for the cells for more information at a local level, and AdaBoost classification, similar to that of Viola-Jones.

Pedestrian detection has seen progress outside of Haar Cascade in recent decades, the increase in publicly available datasets and frameworks has made computer vision more accessible. 2005 saw Dalal & Triggs introduce the histogram of gradients (HOG) feature descriptor which became a landmark for human detection. The descriptor computes the gradients of the input image, and then dividing it into cells. For each cell, a histogram is made of the edge orientations, with the most common orientation representing the direction of the edge within that cell. Linking these orientations together forms a representation of the shape, and excels at handling rotations and different human poses [7].

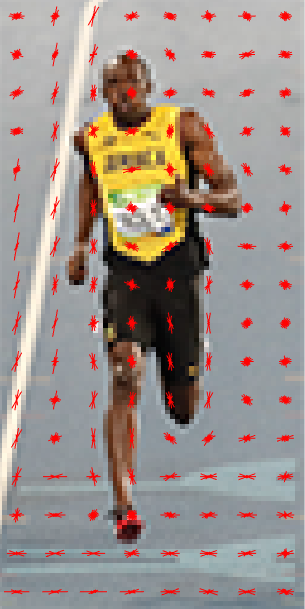


Fig 2. Visual representation of cell directions derived from the HOG feature descriptor. (Source: <http://www.learnopencv.com/histogram-of-oriented-gradients/> )

Detectors such as these begin to use more information from the given images to handle variation in clothing and lighting. These methods demonstrate how effective it is to reduce feature complexity by summarising them into simpler regions. Ultimately making pedestrian detection more adjusted to variations in a person’s appearance. Apart from deriving shapes from edges, colour and orientation, we can also derive shape from motion.

### Motion Detection

Points of interest in an image or video are usually moving, with CCTV cameras for example interesting frames usually involve people entering and leaving the perspective. It worth considering the options that motion detection algorithms provide regarding pedestrian detection. As Haar has shown, reducing the search space to regions of interest lead to more computationally efficient programs. Reducing the search space of our pedestrian detector to moving objects could increase its optimisation, as well as reduce the amount of background interference.

One method for detecting motion is background subtraction(BGS), which involves taking the parts of a video that aren’t moving and ignoring them. The moving/non-moving parts in the video are separated, the moving parts are extracted to what is known as the foreground mask. BGS requires a static camera, since it would fail to distinguish anything from a frame where everything is moving. Fortunately, many surveillance systems use on static cameras, and rely on using many cameras to cover key areas of the perimeter. A simple BGS formula would simply subtract the change in pixel intensities between the current frame and the previous frame.



Fig 3. BGS being run on the CAVIAR dataset [8].

You also need to consider lighting and animated background features. An environment that has a constant shift in lighting effects may return a poor foreground mask. A BGS camera setup of a road with street lamps would struggle with lighting changes as well as shadows emanating from objects at certain angles. Meanwhile a coastal area would have light reflecting of the waves further distorting the mask. Considering not all moving feature of an image are of interest, animated background areas may cause issues. Things such as waves, flags, vehicles and small animals would all appear on the foreground mask if no filtering process was present. Many of these issues can be filtered or reduced through parameters and noise cancelling.

To determine whether a pixel is counted as moving, more advanced methods rely on a ‘history’ parameter. This parameter is usually arbitrary, if something moves in that pixel, the history represents the number of time/frames that pass before the pixel is counted as static. Reducing the history value could tackle the lighting issue, depending how slow the change in lighting is. If the foreground mask is only registering quicker movements, than slower changes in lighting could be ignored.

For the non-interest animated background features, a threshold can be introduced. This threshold would define how much the pixel intensity needs to change between frames before the pixel can be seen as ‘moving’. Through thresholding the foreground mask, smaller movements within the picture such as shadows can be ignored. Modern BGS functions can adapt this threshold to deal with long term changes within the background, such as objects being placed in the scene and day/night cycles.

Another way of detecting motion is using optical flow. Similar to the visualisation in Fig 2, regions within an image are said to have a direction. An optical flow method works out the direction of travel for each cell of an image. Optical flow methods are useful as they can show the strength of the direction as well as it’s orientation, which gives information on how fast the motion is. Optical flow is more suited to dynamic environments, since it can quickly adapt if the camera moves.

One of the biggest issues in computer vision is tackling noise. Image noise is unwanted side effects that occur when visualising a computer vision algorithm, and has a wide range of forms and causes. Examples of noise include speckles, film grain and static, which all add extraneous information. The most common cause of noise is the use of low resolution cameras, a lower number of pixels means that noisy areas will have a greater effect on the overall image quality. It is important to consider noise reduction techniques as clearer images which produce more obvious features and shapes, making easier for algorithms to find them.

Erosion morphology can be used to reduce the amount of speckle noise in BGS. Assume it is being applied to binary foreground mask, where pixel greyscale values are either 0 (non-moving) or 255 (moving). Erosion uses a structuring element to determine which of the current pixel’s neighbours must be 255, for its value to still be 255 after processing. Erosion examines the neighbouring pixels of a selected pixel, if the current pixel has at least one neighbour with a value of 0, the pixel’s value becomes 0.

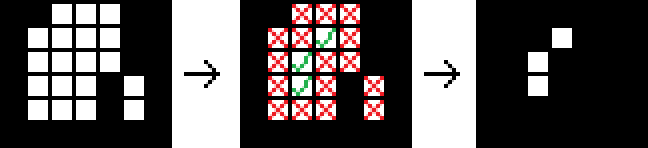


Fig 4. Applying a 3x3 structuring element across an image, a 255 pixel (white) must have the pixels adjacent to it be 255 to survive erosion.

## Analysis

### Objectives

The objectives for the project can be summarised as the following:

* Create a pedestrian detection system that can highlight pedestrians in images
* Run pedestrian detection on source videos
* Locate localised features within shapes to build an overall skeleton model
* Train the system on ground truth data
* Have a system that can differentiate between pedestrians and noise
* Record detected pedestrians in a log

The aim of this project was to automate the process of detecting and tracking pedestrians within a video. From loading the project, the user would be able to select a video to analyse, gaining live feedback of the programs interpretation of what is going on. An ideal project would have a menu or interface, where the user can input threshold values and see their effects in real time. To have pedestrian detection run on live footage would be beneficial, where it could track a livestream through screen capture functionality. However, this required extensive coding which may not be possible during the timeframe. Since the primary aim of this project is successful pedestrian detection, running it on pre-recorded videos may be suitable enough.

The program needed some live feedback, whatever the pedestrian detector is reading should be displayed alongside the source video. This would make the project more engaging if the user could at least visualise the readings while the videos are being played.

It needed some interpretation or knowledge of what it’s looking at, so it can distinguish pedestrians from noise. This would require the code to implement a basic training stage so it can construct its own idea of what a pedestrian looks like. It should return accurate cut outs of pedestrians that appear in the video, rather than printing out every single entire frame as this would show it has poor reliability.

The program must be able to somehow document the pedestrians/features it has witnessed throughout the video. Rather than having security officers go through hours and hours of footage looking for people, the program will have a full list of the pedestrians that appeared with timestamps. The record log will contain information such as the time in the video, cut outs of the pedestrian alongside the programs interpretation of their shape, and a verdict on whether the program believes it was a pedestrian or something else. This log wouldn’t need to be too complex, as long as it is clear which pedestrians appeared at what time.

The project was done in C++, my industrial year gave me experience with C, however there are computer vision related frameworks available in C++. An object-oriented language such as C++ provides better class structure and data management than C which relies heavily on structs.

### Datasets and Frameworks

With the amount of freely available datasets and open source frameworks, its ideal to use proven computer vision functions/datasets for this project. An ideal dataset would provide video files of static cameras observing realistic behaviours and environments. Better datasets would provide ground truth data for videos, so it would be possible to compare when certain pedestrians were found with when they actually appeared. Datasets providing ground truth images of pedestrians could be useful for training the classifier, giving it examples of how pedestrians may appear in the footage. Different datasets were used to ensure the pedestrian detection is robust by applying it to various environments.

The CAVIAR project provides video clips featuring actors in common scenarios of interest for CCTV cameras. These include clips of people entering and leaving shops, meandering outside shops, people running etc. These videos also contain shadows, various lighting effects and reflections, so it would be ideal for testing and developing the pedestrian detector. CAVIAR also provides ground truth files for each of the videos which will be useful for analysing the performance of the program [8].

CVLAB provides multi-camera sequences of pedestrian behaviours for developing people detection algorithms and frameworks. This dataset contains video files with different angles of a pavement area with little to no shadows or reflections. These videos are useful for testing and demonstrating the program in a cleaner environment with less lighting interference [9].

The Daimler pedestrian segmentation benchmark provides contour-labelled and ground truth outlines of pedestrian shapes taken with a stereo camera. The ground truth images provided in this dataset could be useful for training the classifier. It provides a variety of formats for pedestrians including source images, contour outlines and binary ground truth shapes [10].

OpenCV is an open source computer vision library that is optimised for C++ programming. OpenCV provides a wide range of implementations for feature selection methods such as Haar, as well as basic image processing/video reading functions that would be useful for the project.

The ‘Mat’ class is OpenCV’s basic image container used for reading and writing images. It is a matrix containing pixel intensities that come in various formats such as greyscale (0-255) and blue/red/green (BGR). Mat’s memory is also automatically allocated, even when passing on existing Mat objects. This class can be used in the program for reading frames from video footage so they can be processed later, as well as loading dataset images for training purposes [11].

OpenCV has a ‘VideoCapture’ class that outputs frames from a video into a Mat object. The VideoCapture class allows videos to be played in real time by iterating over the frames while displaying each one. This would be useful for getting live feedback from not only the source footage but any analysis the program can give.

### Motion Detection Choice

For pedestrian detection, the people that appear in the footage will be always moving in and out of the scene, so it makes sense to have a motion based system. Having a system that prioritises movement will help filter hours of inactive scenes that occur in security footage backlogs. This also means that the motion detection must be reliable, as a low sensitive motion detector may miss key moments that occur during the video. Having an appropriately tuned motion detector ensures that the pedestrian detection can be reliable and precise.

As discussed in 1.1.4., BGS excels at highlighting moving objects from static perspectives such as a CCTV camera. In a still environment such as an alleyway or corridor, pedestrians moving in the scene are highly noticeable, especially in BGS. While optical flow helps represent large areas of motion, it would be more sensitive than BGS in an environment such as these. Optical flow could be considered overkill for such static environments, and may even perform worse depending on the quality of the cameras used. Using BGS reduces the complexity of the project, as it outputs binary images. This makes processing and representing pedestrian shapes easier.

OpenCV provides several implementations of BGS, with some flexibility to the thresholds used. Allowing the user to input the threshold values allows them to tune the pedestrian detection to suit the video’s scene. Some environments require different levels of history, depending how fast the pedestrians move within the scene. The BGS view can be displayed alongside the source video files, so the user can see what areas of the image are being processed.

### Feature Selection and Classification

Although BGS provides an adequate solution to detecting moving areas of interest, it doesn’t interpret what it’s looking at like Haar/HOG does. It’s important for programs such as these to not only detect, but to draw conclusions from the data it’s given. By combining BGS with feature selection, the program can use basic classification to distinguish between people and noise.

OpenCV provides an implementation for the Haar cascade classifier, which was implemented to test how well it performed on one of the CAVIAR videos. Despite trying out various minimum/maximum ranges for the object sizes, the ‘detectMultiScale’ function had a slow performance along with many false positives. Instead of using the Haar algorithm, an alternative approach was taken involving localised feature detection combined with BGS. BGS can be used to extract shapes from the source footage that may appear as pedestrians, and by locating features within certain areas of the shape, a larger skeleton model can be derived.

This skeleton model attempts to locate human features such as the head, hands and feet within the shape. The regions that each feature appears in can be taught through the training phase, where the code fits the feature skeleton to examples of pedestrians in BGS. Ground truth binary images such as those in the Daimler dataset have a similar appearance to shapes within BGS, so they are useful for the training stage. Once training is finished, the code will know which areas of the shape the features should appear. When running the skeleton finder on the video shapes, features located outside these regions will be considered invalid. The classifier can then count the number of features in the model that are valid, models with a certain number of valid features will be considered pedestrians and recorded within the log.

### Record Log

Choices for storing the record log data came down to using a table within a HTML/Excel file or using a database. A database would be unnecessary for this project as it would bring redundant functionality. If this project was installed on a high security system, having a database with restricted access would be useful for protecting pedestrian logs from being edited by unauthorised personnel. Writing code to output information to Excel format may be difficult, whereas HTML table tags are an easier alternative and can also be viewed on a wider variety of formats.

### Computer Vision Issues

Computer vision systems often rely on assumptions, which can either be based on logic or predications. Examples of logic assumptions include the ways that the skeleton model looks for features. It assumes that the head will always appear towards the top of the shape with shoulders just below, and that feet would appear below the waist. A prediction assumption would be that the pedestrians are always walking in the videos, when they may stop to stretch or sit down. These assumptions could lead the system to struggle with particular pedestrian poses.

A big assumption is that pedestrian shapes will be isolated, when more complex environments will have groups of pedestrians passing by each other. Since BGS relies on binary images, it will have less information for breaking down occluding shapes. The system will need to handle different orientations for people, most environments will have people in different directions. Also need to consider how such a system can deal with lighting, shadows and reflections, as they could distort the appearance of the shape.

Not only will variations in orientations occur, there will be variations in pedestrian appearance such as their clothing, skin tone and size. Environments such as military bases and offices will have uniform clothing, public areas however would have greater variations in appearance. Also need to consider pedestrians with holding items, objects such as bags, rucksacks and equipment could potentially distort the binary shapes.

One way to judge the performance of the pedestrian detector is by looking at the rate of false positives and negatives. False negatives could be argued as a bigger issue, pedestrian detection that believes no one is there when a trespasser is present could be detrimental to a security system. Efforts to reduce false positives should be considered, if the detector is consistently finding pedestrians when they are not present, users may lose trust in the system.

BGS will often pick up traces of small areas of motion, it is important to consider techniques for filtering smaller sizes of motion as they most likely not be pedestrians (thus reducing the false positive rate). Most cameras will contain noise due to resolution and speckle effects, so it’s important to consider noise reduction implementations.

### Security

With a project related to surveillance, it’s important to consider the security aspects and ethics of the work being done. The chosen datasets allow the videos and images to be used for academic purposes, and the actors within the videos are aware that they’re being recorded. No video sources have been used where pedestrians are unaware that they are being filmed. Websites hosting 24 hours livestreaming were avoided for this reason. Any videos recorded by myself were done ensuring that the people within the videos gave consent to record them.

It’s also vital to ensure no copyrighted material has been downloaded and played back through the program. Websites such as YouTube were avoided as it can be difficult to verify the uploader of the video and ensure that they’ve taken the correct precautions with their footage. Videos involving areas of restricted access were avoided, having pedestrian logs of high risk areas is forbidden.

## Process

### Methodology

For the software development side of the project, taking an agile approach allows the project to evolve and adapt throughout the development process. With a topic such as computer vision, goals and tasks are likely to change as algorithms are tested on dataset videos. Issues may arise that may contradict the goals originally set. A sequential model such as waterfall could restrict the necessary changes needed to increase the efficiency of the program.

Feature Driven Development (FDD) is a form of agile software development, it is a model-driven iterative process where the goals are defined by the necessary features. FDD involves having idea of the overall model and having a general idea of what features will be needed. Having a features list provides clear concise goals, while allowing necessities discovered during implementation to be included. FDD also has more flexible working hours than other methodologies such as Scrum which rely on sprints. Since this is a one-man project, other methodologies such as Extreme Programming that rely on pair programming aren’t as applicable. This also means that parts of FDD must be modified, such as generalising the team roles.

### Support tools

There are many IDE’s to support C++ development, IDE’s are useful as they highlight common errors in code and simplify the compilation process. Visual Studio 2015 proved useful for the project, as integrating the OpenCV framework into the project was quick and easy. The structure of the code could be easily managed using the project managers and class view tabs.

The project was stored in a git repository via BitBucket, a web service for hosting source code for large scale projects. Backing up code using a web-hosting service is more reliable and efficient than exporting it to USB, especially for larger projects. Documentation could also be exported to git, which made it easier to work on it from multiple machines. A diary was kept throughout development to keep track of the progress made as well as any sources or useful websites.

# Design

## Overall Architecture

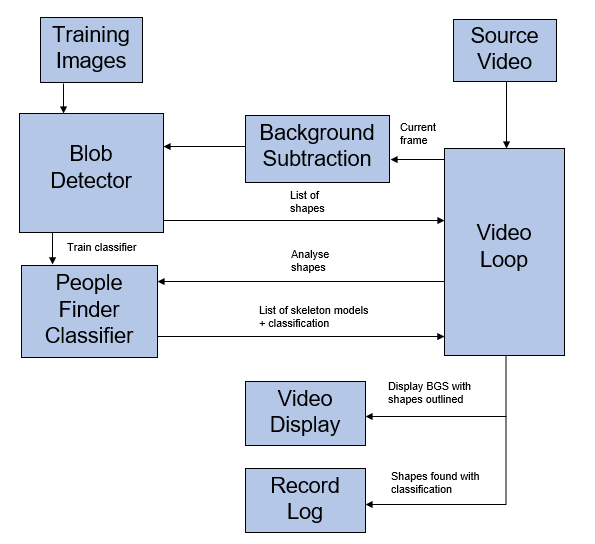


Fig 5. Flow of information with the overall model

As you can see from Fig 5, an overall model of the system was drafted displaying the relationships between the core functionality. The people finder assumes that it will be looking at outlines of binary image shapes. Before the program runs the video, the blob detector outlines contours for the images so the people finder can be trained on the same format. Once the people finder has finished training, the video loop begins playing the video while applying BGS. The video loop is needed to iterate through each frame in the source video. Background subtraction can then be used to find moving objects in the video, as it has knowledge of the previous frame in the loop. The blob detector will then draw contours around the shapes, and outline the larger shapes that could potentially be pedestrians. A list of shapes is returned to the video loop so they can be displayed and used in the record log. The people finder applies the skeleton building function on the shapes, and determines whether the shape is a pedestrian or not. The people finder’s interpretations of the shapes and verdicts are returned to the video loop so they can be recorded in the log. The video loop helps encapsulate information between the different parts of the program so they it can be summarised in the record log. Having each class create their own versions of the log could convolute it. BGS also needs to be integrated into the video loop, as keeping the BGS separate from the frames could interfere with the motion detection by losing its frame history.

## Features List and feature design plans

After developing an overall model of the program, a list of features was created to formalise the requirements to be completed. Each feature describes a necessary action that may be done multiple times throughout the program. Plans were also created for each feature which describes how they work. A summary of the features can be seen below:

* Input threshold values and paths into a menu
* Loading the training data
* Outlining shape contours
* Building the pedestrian models\*
* Training the pedestrian finder
* Running the input video
* Applying BGS
* Classifying shapes
* Record findings

The pedestrian model requires more detail through sub features, listed below:

* Searching for the head
* Torso
* Waist
* Feet
* Shoulders
* Elbows
* Hands

### Input threshold values and paths into a menu

When the program loads, it presents the user with a menu where they can enter the training folder path, video path and BGS thresholds. A user interface is unnecessary as long as the user is able to input values and have the program filter invalid responses.

### Loading the training data

A directory path is entered from the input menu of the program, this path leads to a folder containing the images that the people finder will be trained on. The code retrieves the paths for each file. Using OpenCV’s image handling, each image is loaded using the paths.



Fig 6. Binary images of pedestrians that can be used for training. From the Daimler dataset [10]

### Outlining shape contours

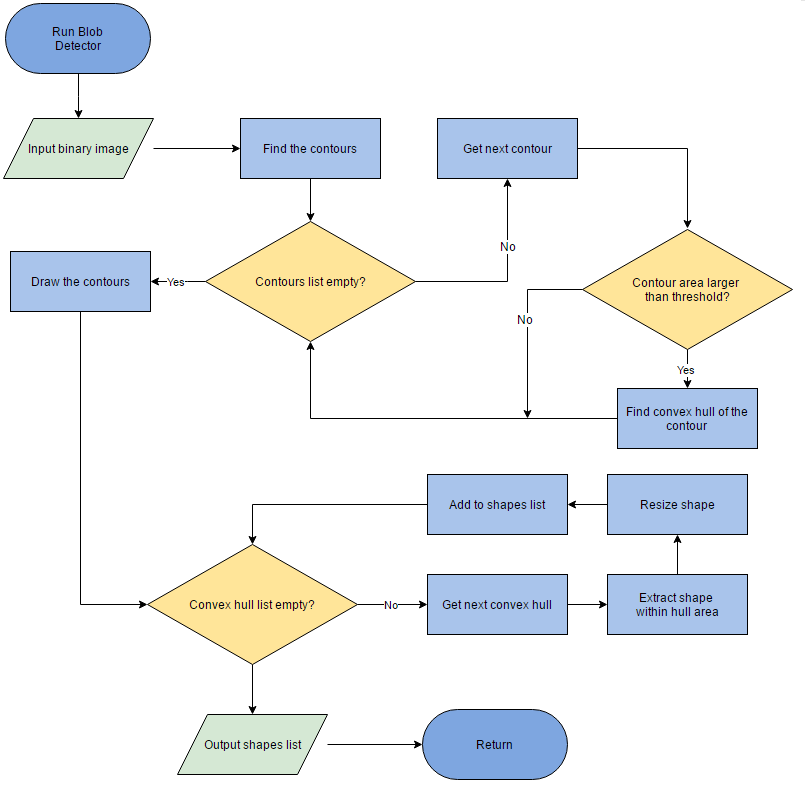


Fig 7. Flow chart for outlining shape contours.

Outlining shape contours is needed for training and testing. Input images for testing are resized and outlined so the people finder has a better idea of where the shape boundaries are. It is used during the video stage to extract the larger moving shapes within the overall frame. It assumes that the input images it receives are in binary format. Resizing shapes to the same size makes it easier for the people finder to interpret them.

### Building the pedestrian models

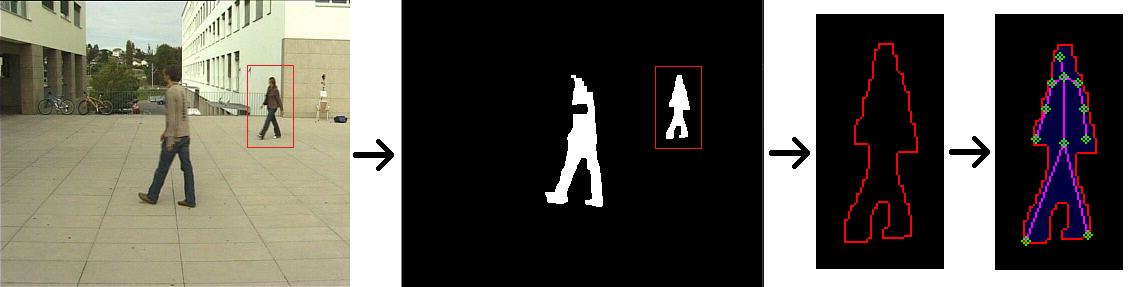


Fig 8. Summary of the detection process using CVLAB’s dataset [9]. Left to right: Source video, BGS, contour outline, pedestrian skeleton model

As with the shape contours, building pedestrian models will have similar functionality when being used for non-trained and trained examples. The main people finder class calls each of the local feature finding function to fit a pedestrian model to the shape. After locating the pedestrian features it plots them within the contour outline and connects the corresponding body parts. The sub features for this process are outlined in 2.3.

### Training the pedestrian finder

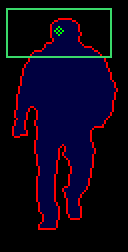


Fig 9. An example of the minimum and maximum boundary that valid head features fall within (green square) using the Daimler dataset [10].

As the people finder applies pedestrian models to the training images, it takes the minimum and maximum (x, y) positions for each feature. This creates rectangular regions which features from the video shapes must appear within to count as a valid feature. After the people finder is trained, it can differentiate between pedestrians and noise using these position ranges.

### Running the input video

With the help of OpenCV, the video is loaded and a frame is opened each loop iteration. These frames are displayed in windows, with the windows being updated with each new frame. As long as the performance is optimal the videos run smoothly on the screen. The BGS version of the frame will be visible alongside the source video as well as the outlined shapes.

### Applying BGS

For each new frame, OpenCV’s BGS function is applied. The binary image created using the function is filtered to reduce the noise. Optimal methods of filtering involved using a closing operation (dilation then erosion) followed by an opening operation (erosion then dilation) to reduce the speckle noise produced in the videos (see section **3.1.3.**). Dilation is the opposite of erosion in that it looks to expand areas with low pixel values within the kernel. These binary images are sent to the blob detector so the larger shapes can be extracted.

### Classifying shapes

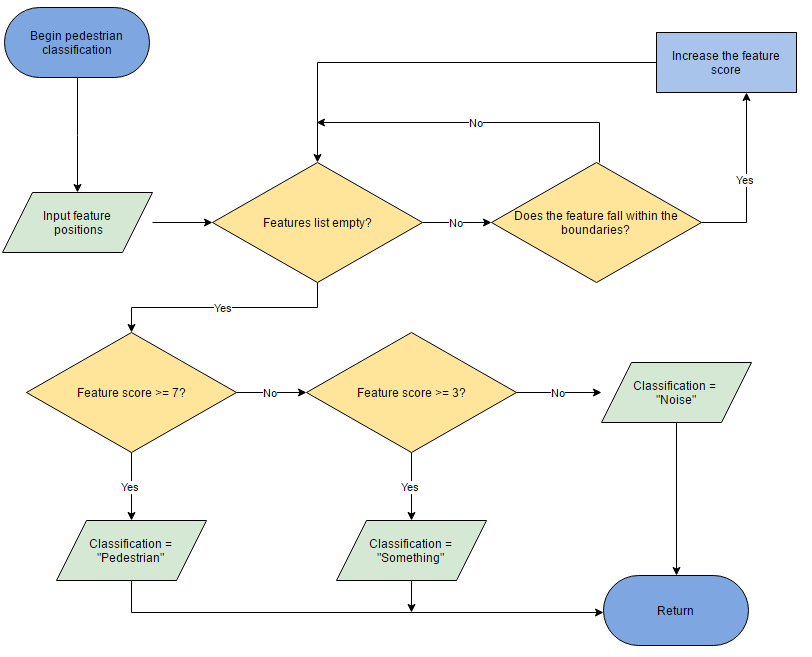


Fig 10. Flow chart of the classification process

The people finder classifies the skeleton model built within the current shape. The program uses a feature tally system to decide whether the shape is a pedestrian. The feature skeleton consists of ten (x, y) coordinates that represent the different parts of the pedestrian. Each feature has their own (x, y) boundaries that were set during the training phase. Using the head feature for example, if the head feature plotted in the current shape falls within the boundaries, the feature is considered valid and the score increases. This is done for every feature, the resulting score determines what the verdict for the shape will be.

### Record findings

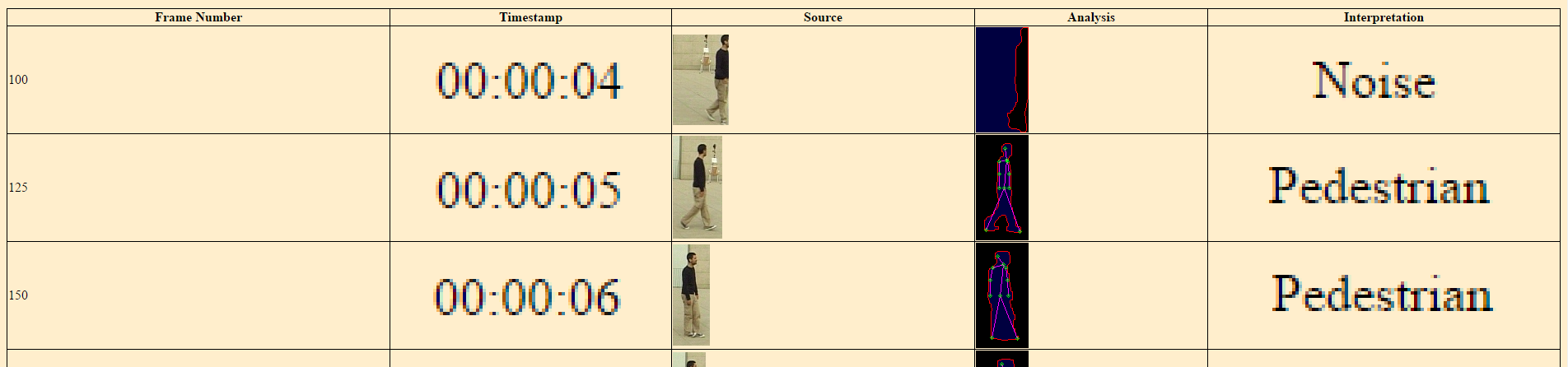


Fig 11. Example of the record log produced by the program that’s running a CVLAB video [9]

A new HTML file is created before running the video. The settings and directories used during runtime are saved, and the table is initialised. After classifying all the shapes found in the current frame, the log creates a new record for each one. The frame number and the timestamp indicate when the shape was found, along with the source of the shape and how the pedestrian detector interpreted it.

## Pedestrian model pseudocode

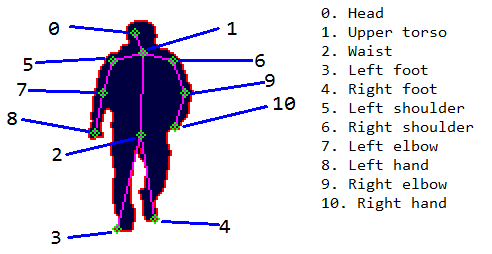


Fig 12. Summary of the local features that make up the skeleton

Various algorithms are used for finding different features of the human body, with some being more complex than others. Some feature nodes require knowledge from other nodes to find their optimal position. For example, the torso node being positioned relative to the head node and elbows relative to the shoulders. Thresholds are applied to certain features to make the positions more realistic. The original pedestrian model had no elbow nodes, which made estimating the hand positions more difficult and inaccurate. An alternative model considered implementing nodes for the knees, which would be use a similar algorithm to the elbows. Searching for more features increases the total classification score, resulting in a more detailed classification process. This would however increase the complexity of the people finder class, and many models that were produced showed little variation in knee positions. The feet positions reveal enough information about the legs.

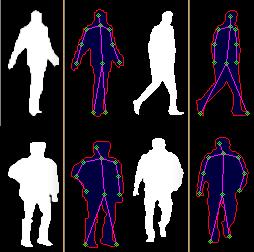
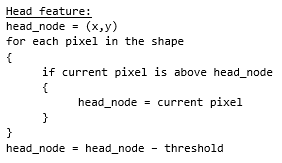
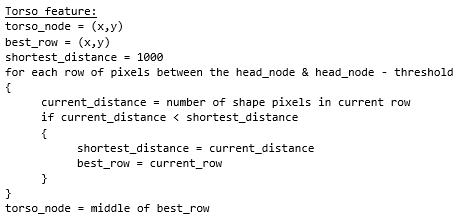


Fig 13. Pedestrian skeletons created on the Daimler dataset [10]

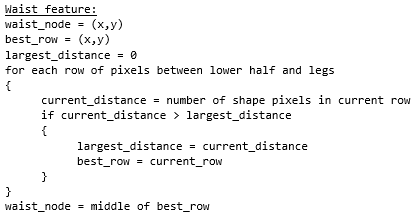
### Search for the head



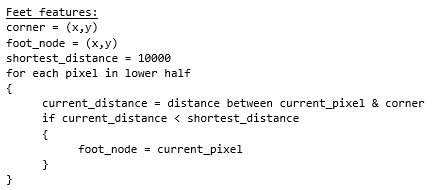
### Torso



### Waist

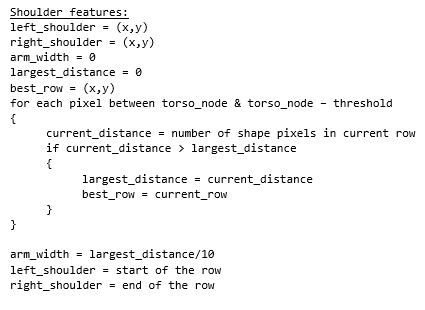


### Feet

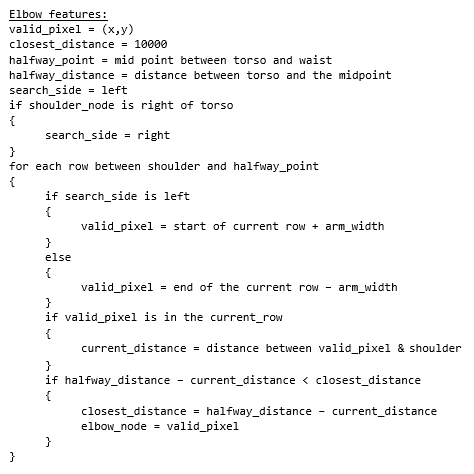


Euclidean distance is used to calculate the current distance. Where (x1, y1) are the corner’s coordinates and (x2, y2) are the current pixel’s location.

### Shoulders

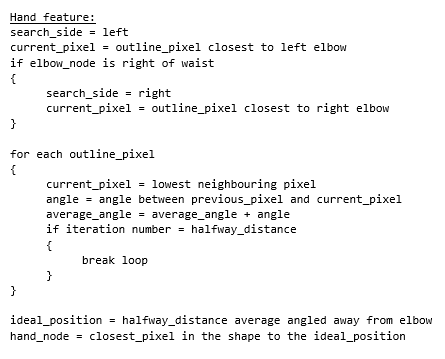


### Elbows



Euclidean distance is used again, where (x1, y1) is the valid pixel’s location and (x2, y2) is the shoulder node’s location.

### Hands



The following equation calculates the ideal position of the hand, where (x, y) is the elbow location, d is the halfway distance, and is the average angle.

## Final classes



Fig 14. UML diagram for version 1.0 of the system.

Regarding the initial features list, Functionality for features involving the training data and building/classifying pedestrian models is within the *PeopleFinder* class. Functionality for features involving applying BGS and running the video is within the *BGS* class. This class runs OpenCV’s BGS function alongside the noise reducing functions. The *BlobDetector* class handles shape outlining and extraction and is used on the frames extracted from BGS. *PeopleFinder* has the code for creating and classifying pedestrian models generated in the shapes. *RecordLog* handles the HTML log functionality. This includes creating the log and exporting images of the shapes and pedestrians detected to the log.

The main provides a menu for inputting the training folder and video paths. It also allows the user to enter the history and distance threshold parameters for the BGS. An alternative solution was to have the input options within their respective classes, however that would make the runtime more inconsistent. Rather than having the program pause each time the class object’s values are needed, the user inputs all necessary information at the beginning and then allows the system run without stopping.

### Data Structures

The core video loop side of the architecture is incorporated into the *BGS* class. Since the *BGS* is what generates the shapes, it is logical to make it accessible to the *BlobDetector* and *PeopleFinder*. The *run()* function in *BGS* uses a *VideoCapture* object to open the video via the *video\_path* string. *VideoCapture* is useful as it returns each frame as a *Mat* object which is used in most of the OpenCV functions. *VideoCapture* also gives information such as FPS, which is useful for tracking performance and can be used to apply pedestrian detection every second. All images are stored as *Mat* objects which use predefined ‘types’ that define how the pixel information is stored, for example an image with type CV\_8UC3 has three unsigned *chars* for each pixel that represent the colour channels. This increases the flexibility of the program as it handles different types of images such as full colour and binary. Having the images and shapes use the same data structure means less data conversions and simplified architecture.

As with many programs, *string* manipulation is often required. C++ uses *std::string* objects to handle text with class members such as *video\_path*. Unlike its predecessor language C, C++ automates the memory allocation process for strings whereas C relies on char arrays with null byte termination. This makes string manipulation more safer and easier to use, C++ also provides a class for altering *strings* called *stringstream*. *Stringstream* can read and write variables into strings, making manipulating paths and outputting information to the log easy. Another useful data structure is *Point*, which consists of two integers representing (x,y) positions. Since *PeopleFinder* is heavily reliant on pixel positions for finding features, using *Point* objects is more efficient and organised than using two *vector<int>* sequences for (x,y).

*Vector* is another data structure introduced with C++, it is a sequence container that can change in size after being declared. One of the downsides to C arrays was that they are inherently static, so it would struggle with varied input numbers. To deal with large input collections a large index size would need to be declared to prevent out of bound errors. Although *vector* still requires an index size before storing objects, *vector* can have an index variable representing the number of inputs, and then use that variable to declare the *vectors* size beforehand. *Vector* can be used with many data structures including *Point, Mat* and *vector*. This makes *vector* objects useful for handling different numbers of contours, shapes and training images.

*BGS* has *RecordLog* as a class member to ensure that it is freely accessible throughout the program. The *RecordLog* member keeps track of the number of records it has inputted into the log through *total\_records*, to ensure previous entries don’t get overwritten.

Pointers were used to refer to the location of variables through the project, most pointers in the projects were used to change variables passed through function parameters. Variables such as *Mat* objects were often referred to with pointers so images could be altered within separate functions such as *draw\_annotations()* in *BlobDetector*. Other functions would return various calculations at once that were needed elsewhere such as *calc\_halfway\_torso\_dist()* in *PeopleFinder*.

### Function Allocation

Only the *PeopleFinder* class requires the training directory, so the functionality resides within its *search\_dataset\_files()* and *load\_datasets\_files()* functions. Each body part feature was split into separate searching functions. Since certain body parts appear twice in single model, search functions such as feet, elbows and hands were developed to handle left and right detection. By using a single function to find both elbows, it reduces the complexity of the program. Functions that relied on repetitive calculations were refactored into smaller functions. For example, elbow detection and hand detection relies on finding the halfway point between the torso and waist, so a function was made for this calculation. The *is\_within\_bound()* function was used throughout the *PeopleFinder* class. Many detection functions needed to check whether the features they found fell within the boundaries of the image. The same function was also needed in *BlobDetector* to check whether the outlines of the extracted shapes are within the image dimensions.

Efforts were made to reduce the number of loops and statements per function to increase the readability of the code. Other functions were made purely to reorganise the *create\_skeleton()* function in *PeopleFinder*. Functions such as *highlight­\_pixels()* and *draw\_skeleton()* were originally part of *create\_skeleton()* but were refactored due to their nested loops.

In *BGS,* functions for handling noise were split into operation types *erode\_first()* and *dilate\_first()* rather than calling OpenCV’s *erode()* and *dilate()* four times in a single function. This helped during development as the order of the functions could be interchanged to see which order was more optimal. The *VideoCapture* object stores the timestamp in milliseconds,so the *convert\_milliseconds()* function in *RecordLog* is needed for the readability of the log.

# Implementation

## BGS

### Setup

Using the *BGS* constructor, the video and training paths can be passed through the menu input to the *PeopleFinder* object via the class member strings. *Run()* is the core function for the program, it initialises local instances of *BlobDetector* and *PeopleFinder* as well the *RecordLog rlog* class member. Six local *Mat* objects are used through the function, *frame* contains the current frame of the *VideoCapture* object. *Frame2* is the same frame but converted into greyscale using OpenCV’s *cvtColor()* (see 3.1.4.). Having the input Mat be the same as the output Mat could potentially corrupt the data so a new *Mat* object was needed. *fgMaskKNN* is the unfiltered output from OpenCV’s BGS function containing a noise filled binary image, *filteredMask* is the output *Mat* from the noise reduction function *filter­­\_noise()*. *Contourimg* and *contoursonly* are output images produced by the *BlobDetector*, *contoursonly* contains shape outlines whereas *contourimg* contains the outlines and convex hulls.

OpenCV was useful for presenting live feedback to the user, *imshow()* takes in a *Mat* object and displays it in a new window. As long as the images have the same dimensions, the same window can be used to smoothly iterate through the video frames.

The initial video loop applied feature finding algorithms in *PeopleFinder* every frame*,* the live feedback performance was poor with the FPS falling below 5. Applying pedestrian detection every second would suffice, as there is little information change between frames and frame by frame conclusions could possibly flood the record log. A solution was needed to apply pedestrian detection every second rather than every frame, using the *get()* for *VideoCapture* the frame number can be retrieved.

if (frame\_number-(fps\*iteration)==fps)

{

iteration++;

//run pedestrian detection

}

In the code above, *frame­\_number* is the current frame, *fps* is the number of frames played per second, and *iteration* is the number of times the statement has been true. Every time the *fps* is a factor of the *frame­\_number*, pedestrian detection is applied.

### KNN vs MOG

As of version 3.1.0, OpenCV provides two class extensions for BGS, *BackgroundSubtractorKNN* and *BackgroundSubtractorMOG2.* The KNN version uses K-nearest neighbours based segmentation for separating background/foreground pixels. The KNN BGS has two parameters, *history* and *dist2threshold*. The *history* represents the number of previous frames that affect the background model*. Dist2threshold* defines how different the pixel intensity needs to be to register as moving.The MOG version is Gaussian-mixed background/foreground segmentation. The Gaussian distribution is used to find the range in which a pixel is changing, if the changing range is larger than a threshold it is considered a ‘moving’ pixel. It has a *history* parameter similar to KNN, and a *varThreshold* *float.* *VarThreshold* is the threshold of the squared distance to see whether a sample is close to existing components (areas of movement). If pixel is not within the threshold distance to a component it is added to the foreground mask.

Both functions were initially implemented to see which one was more optimal for detecting larger moving shapes. The performance of *PeopleFinder* relies on *BGS* sending *BlobDetector* complete shapes that are not disjointed in anyway. This is because the *PeopleFinder* assumes that a single shape represents a single pedestrian, so gaps between shapes could result in two pedestrians being detected. It’s important for *BGS* to have an optimal sensitivity, a low shape detection rate could result in a greater false negative rate, while a high shape detection rate makes the system unreliable.

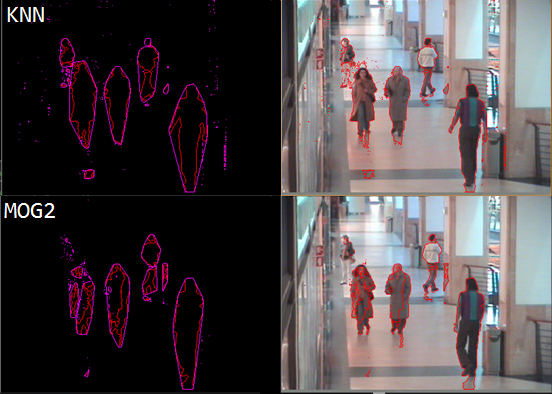


Fig 15. *BlobDetector* outlining shapes extracted from the BGS methods on CAVIAR [8].

When applying the unfiltered BGS functions to the CAVIAR videos, the KNN algorithm produce better quality shapes than the MOG2. As you can see from Fig 15, KNN produced more wholesome shapes, whereas the shapes found by MOG2 were fragmented and disjointed. Gaps in the moving pixels cause contours outlined by the *BlobDetector* to split, this results in shapes (highlighted in purple) dividing into smaller shapes. Furthermore, if a noise reduction process is applied to MOG2, the smaller shapes are removed altogether, leaving little information behind.

*BackgroundSubtractorKNN* is also capable of distinguishing shadows in the shapes. If *detectShadows* is set to true, moving pixels perceived as shadows are highlighted in grey in the resulting *BGS* frame. This capability is redundant for the system as the *BlobDetector* relies on shape outlines, and has no comprehension of colour. The shadow detection feature reduces the performance of the live feedback, and in CAVIAR/CVLAB videos it would misinterpret pedestrians when their clothing is similar to the background.

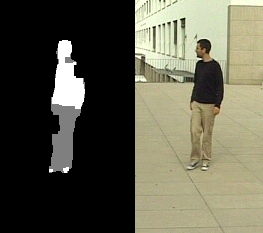


Fig 16. *BackgroundSubtractorKNN* with shadow detection enabled. Shadows(grey) visible on clothing.

### Reducing Noise

The aim of *BGS* is to send the least amount of clear and concise shapes for the *BlobDetector* to extract*.* Many of the noise issues produced in videos can be minimalised or filtered completely. Videos such as the ones from the CAVIAR dataset produced speckles and shadow noise in the resulting binary images due to the reflective surfaces. OpenCV provides functions for erosion and dilation, which were combined to form closing/opening operations. Not only can they be used to reduce noise, they can fill gaps that appear within shapes, potentially connecting shapes that became disjointed. The order in which the operations are called affects the nature of the noise reduction, as shown below.

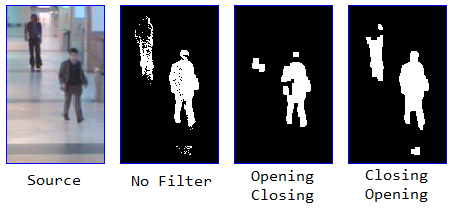


Fig 17. The results of the different operation orders on CAVIAR [8].

Fig 17 shows how the different operation orders can produce different BGS interpretations of the same frame. As the two actors walk down the corridors, their movements are reflected on the surfaces. In the unfiltered frame, the speckle patterns can be seen in the deteriorating shape for the actor on the left. Gaps are also emerging with the right actor, with traces of reflection movements being shown within BGS. Each isolated speckle is a redundant shape that gets analysed by the *BlobDetector* so it is important to reduce the shape load for the sake of performance*.* The opening then closing sequence calls the OpenCV *erode()* function followed by *dilate()* and then *dilate()* followed by *erode()*. Although this removes the reflection noise, it causes gaps within the shapes to be exaggerated. The left actor is almost completely eroded which could result in a potential false negative. Closing then opening is the opposite sequence of operations, and produces much better results. The shape of the right actor is clearer, and although the left actors shape isn’t perfect, it is still enough to be detected. The speckles appearing in the reflections amalgamate into a small shape, it is beneficial for *BlobDetector* to process and skip one small shape rather than many tiny speckle shapes.

### Problems

While the filtering process increases the quality of the resulting BGS binary image, there are limitations to what can be extracted. Many of the problems produced in *BGS* have knock on effects to the other classes and the way they perform, this is often due to interference with the shape.

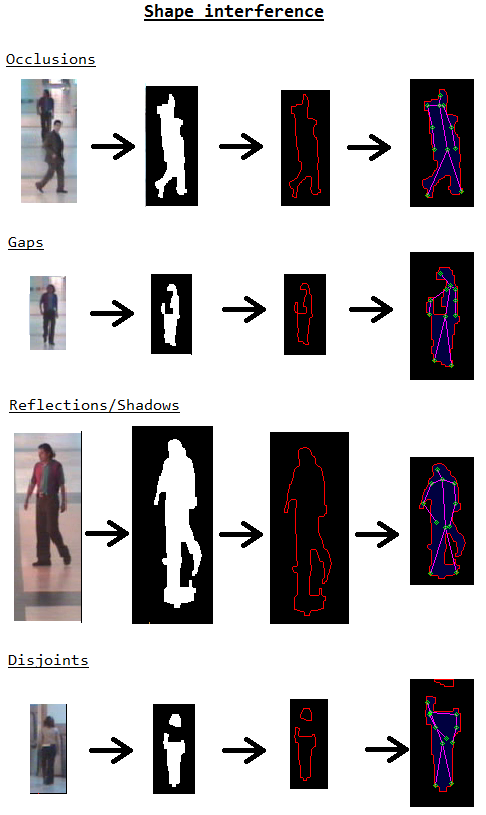


Fig 18. Different types of interferences that occur in *BGS*, and how they affect the *BlobDetector* outlines (right) and *PeopleFinder* models (far right).

When two or more pedestrians occlude each other, the shape blobs in *BGS* merge, resulting in multiple pedestrians being in a single shape. These multi-pedestrian shapes will most likely not bare the resemblance of a pedestrian, as Fig 18 shows. Gaps are a common occurrence with shapes, where parts of a shape blob become unclear. This is often cause by the person’s clothing blending into the background, or the person being too far away from the camera. Sometimes speckle noise will amalgamate and connect to a nearby pedestrian shape, causing the resulting outline shape to be disfigured. Shapes become disjointed and divide into smaller shapes when the size of the gaps worsen. This is again caused by camouflage clothing, or long distance pedestrians.

It was later discovered that converting the source image into greyscale slightly reduces the amount of reflection/shadow noise as well as camouflage interference. Start-up noise was another issue *BGS* faced, since the BGS algorithm has no frame history to work with when initialised, the first BGS frame is filled with noise. For this reason, the *RecordLog* in *BGS* skips frame zero.

## BlobDetector

### Setup

The *BlobDetector* class is used to outline larger shapes within an image and save them as their own *Mat* objects. This class is used in *BGS* as well as *PeopleFinder* because the convex hulls show which shapes are being analysed in the live feedback. *BlobDetector* has a class member called *src­\_shapes,* which is a *vector* of *Mat* objects. This class member is used to store the shape sources so they can be exported to the log. Since this class contains the code for extracting shape outlines, the same functionality can be used for extracting the corresponding source shapes.

### Contours and Hulls

*BlobDetector* relies on contours and convex hulls for outlining and extracting large shapes, which OpenCV can calculate. OpenCV’s *findContours()* retrieves the boundaries from a binary image as a 2D *vector* of *Points*. The function has an optional *hierarchy* parameter which contains information about image topology, but is only used for the drawing the annotations in *draw\_annotations()*. *FindContours()* also has alternative modes for contour retrieval and approximation. CV\_RETR\_EXTERNAL retrieval was used as it only retrieves the outer contours, and hierarchical computation is unnecessary. The function uses CV\_CHAIN\_APPROX\_SIMPLE approximation to compress the contours.

*ConvexHull()* finds the convex hull of a *Point vector* and outputs the hull as the same data structure. The *highlight\_contours()* function in *BlobDetector* uses *findContours()* to create contour outlines on the binary image from *BGS*. A 2D *vector* of *Points* is made for storing the hulls using the number of contours found. Every contour with an area more than 300 has a hull created using *convexHull()*. This allows the *BlobDetector* to skip over smaller shapes that would most likely not be pedestrians. *Draw\_annotations()* is then used to visualise the contours/hulls in their respective images.

### Shape Extraction

*Get\_large\_shapes()* requires several parameters for finding shapes within the overall image. Both the source image and the filtered foreground mask is needed so the *BlobDetector* can save the source images as its processing the shapes. The hulls *vector* and size are needed, as well as an *edge­\_space* integer. When each image is saved as its own *Mat* object, an empty pixel border is added to push the shape towards the centre. In the Daimler training examples, none of the ground truth images had pixels touching the borders of the image, so extracted shapes filling these areas would be considered invalid. The *edge\_space* parameter defines how large the empty areas on each side of the shape will be.

The function then finds the top left and bottom right *Points* in each hull so it can draw a rectangle around the hull’s area. An *is\_within\_bound()* check is used to ensure the *edge\_space* parameter doesn’t fall outside of the image boundaries. If the shape passes the check, then the *edge\_space­* is added to each side of the rectangle. These rectangles will then cover the entire shape, which is then resized and saved as its own *Mat* object in *bg\_shapes*. Before resizing the shape, the sources of each image are stored in the *src\_shapes* class member.

## PeopleFinder

### Setup

The *min\_range* and *max\_range* class members are *vectors* containing 10 *Points,* each *Point* representing minimum and maximum valid positions for each feature. These ranges are used to determine whether a feature found during testing falls within the validation boundaries. *Verdicts* is a *vector* of *strings* that contains the classification results for each of the shapes. The *bad\_skel\_flag* is a *bool* that is set to true whenever the *PeopleFinder* can’t process the shape.

*Demo(), train()* and *test()* all run the skeleton building functions for different purposes. *Demo()* iterates through the training directory and visualises how the skeleton models in the training data appear. *Train()* creates skeleton models for all of the training files while adjusting the *min­\_range* and *max\_range* class members accordingly. *Test()* uses the shapes passed through *BGS/BlobDetection* and creates skeletons within each one. From here the feature score is calculated and a classification for the shape is given.

When creating the pedestrian skeleton models, all source images are resized to 128x64, as the detection algorithms were more robust when working with a uniform size. Initial efforts to make feature detection work regardless of size were difficult to the seemingly infinite complexity of size. Pixels in the shape images are colour coded with the red pixels being the contour outlines, black pixels are considered outer shape and blue pixels are inner shape. This made storing pixel data relatively simple and helped keep detected features inside the shape.

OpenCV’s *floodFill()* function proved useful for colour coding pixels. Assuming the pixel in the centre of the 128x64 image is inside the shape, the entire shape can be filled. That way pixels inside the shape can be easily distinguished by colour. Problems can arise if the middle of the image is outside the shape, or if the shape outline intersects the centre pixel.

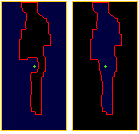


Fig 19. Example of gap interference causing the fill pixel (green) to be outside the shape.

When referencing a position in the *Mat, x* refers to the row number and *y* refers to the column number. *Mat* coordinates start from the top left hand corner.

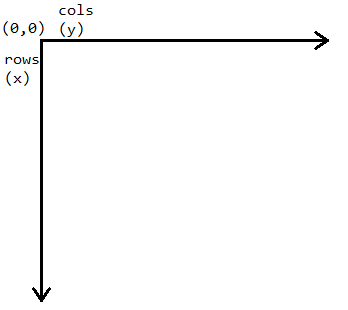


Fig 20. Summary of the scale used with *Mat* and *Point*.

### File handling

*PeopleFinder* uses the *WINAPI HANDLEs* from the Windows library to get the file names for the training images, using *FindFirstFile()* and *FindNextFile()* the paths for each image are stored in a *vector* of *strings*. The code initially searched for files by finding the current working directory and navigating the path, but this proved difficult to maintain in different working environments. Release/debug modes had different working directories, and the access levels to the training/video folders wasn’t always clear when running from a different computer. The final version of the *search\_dataset\_files()* was changed so the full path is required.

### Create skeleton function

*Create\_skeleton()* is the core function that prepares the shape for analysing and calls each feature detection function. *Shape\_pixels* and *Outline\_pixels are* *Point vectors* that contains the (x,y) positions of key pixels in the current *Mat* object. *Shape\_pixels* refers to each pixel that is inside the shape, whereas *outline\_pixels* refers to the pixels in the contour lines. *Nodes* is a *vector* of *Points* that contains the (x,y) positions of each feature detected. Certain feature detection functions iterate through the shape pixels to skip areas that most likely won’t contain that feature, the indexes for specific features are also stored as *ints* to speed up the iteration process. After the pixel colours have been assigned, *create\_skeleton()* loops through each pixel of the *Mat* adding (x,y) positions to the respective *Point vector.* Once the inner/outline shape pixels are established, the feature detection funtions are called. The pixels in each *vector* are stored row by row, which is useful when calculating distances between shape sides, but inefficient when it comes to examining neighbouring pixels.

### Head and torso detection

Head detection has the simplest algorithm out of the feature detectors. *Find\_head\_feature()* finds the highest point in the shape, and offsets it slightly by an arbitrary threshold. The loop assumes that if it has iterated passed the first few rows of the *shape\_pixels*, then the (x,y) positions for the head node have been found. This reduces the search time as the lower parts of the shape don’t need to be considered when searching for the head.

*Find\_torso\_feature()* uses *lower­­\_bound\_x*  to ensure that the torso feature is near the head. It then loops through each *shape­\_pixeI* ahead of the head feature and counts the number of pixels that appear on the same row. *Best\_fit\_node* is a *Point* object that locates the end pixel of the best row. The loop considers rows that appear just below the head position, the row with the least number of pixels will be chosen. The torso feature’s position is then declared as:

torsonode.x=best\_fit\_node.x+threshold;

torsonode.y=best\_fit\_node.y-(shortest\_dist/2);

### Waist and feet detection

*Find\_waist\_feature()* function has two integer boundaries, *upper\_bound\_x* and *lower\_bound\_x* that ensure the waist feature is in the lower half of the image*.* The upper boundary is 64 (halfway), some anomaly images had torso positions that were below this, so the function was changed so the upper boundary can be altered. Waist detection uses similar code to torso detection, except it searches for the row with the largest number of *shape\_pixels.* Waist detection excludes discontinuities cause by arms when searching for the largest distance, for example if the *y* position of the current pixel jumps from 10 to 12, the loop iterates to the next row.

*PeopleFinder* can use *find\_foot\_feature()* for both feet, the *Point corner* is the location that the foot should ideally be close towards. By using the bottom corners of the image, the foot features will be drawn towards them. The upper boundary ensures that the foot feature is below the waist. The resulting foot feature is the closest *shape\_pixel* to the *corner*. The implemented code for detecting feet is as follows:

while (shape\_pixels[i]!=Point(0, 0))

{

i++;

distx=(corner.x-shape\_pixels[i].x)\*(corner.x-shape\_pixels[i].x) disty=(corner.y-shape\_pixels[i].y)\*(corner.y-shape\_pixels[i].y);

current\_dist=sqrt(distx+disty);

if (current\_dist<shortest\_corner\_dist)

{

shortest\_corner\_dist=current\_dist;

best\_fit\_node=shape\_pixels[i];

}

}

### Shoulder detection

*Set\_shoulder\_positions()*  calculates both shoulder positions simultaneously, assuming the person’s shoulders are more or less the same height. The upper/lower boundaries ensure the shoulders are located near the upper torso. The function finds the row with the largest number of *shape\_pixels* with *best\_fit\_node* being the pixel at the end of the row. The shoulder positions are declared as:

\*arm\_width=largest\_dist/10;

if (\*arm\_width==0)

{

\*arm\_width=1;

}

\*left\_shoulder=Point(best\_fit\_node.x,best\_fit\_node.y-largest\_dist+\*arm\_width);

\*right\_shoulder=Point(best\_fit\_node.x,best\_fit\_node.y-\*arm\_width);

### Elbow detection

Once all features outside of elbows and hands have been found, *calc\_halfway\_dist()* uses Euclidean to find the midpoint between the torso and waist. *Find\_elbow\_feature()* can be used for both arms, the elbow it finds depends on whether the shoulder is left or right of the torso. ­*Shape\_pixels* that are *arm\_widths* length away from the side of the shape are considered as *valid\_pixels*. The Euclidean distance is calculated between the *valid­\_pixel* and shoulder position. The ideal elbow area is *halfway\_dists* away from the shoulder, the *valid­\_pixel* that is closest to this area becomes the elbow position.

While this algorithm is flexible for dealing with variations in shapes, it struggles when a pedestrian’s arm is above their head. It is safe to assume that pedestrians will rarely appear with their arms above their head when walking through a surveillance area.

### Hand detection

*Find\_hand\_feature()* relies on using the *outline\_pixels* adjacent to the current elbow feature, following the neighbouring *outline­\_pixels* and taking the average angle between each one. The function examines the neighbours of the current *outline\_pixel*, while prioritising southern neighbours it selects the optimum neighbour to move towards. It then calculates the angle between the optimal neighbour and the current pixel using the following code:

prev\_valid\_pixel=curr\_pixel;

curr\_pixel=neighbours[i];

angle=atan2(curr\_pixel.y-prev\_valid\_pixel.y,curr\_pixel.x-prev\_valid\_pixel.x);

average\_angle+=angle;

The number of iterations the function does is equal to half of the *halfway­\_dist* from the *calc\_halfway\_dist()* function. Once the loop is finished, the following code is used:

average\_angle=average\_angle/dist\_iteration;

best\_fit\_node=Point(elbow\_feature.x+halfway\_dist\*cos(average\_angle), elbow\_feature.y+halfway\_dist\*sin(average\_angle));

hand\_node=find\_closest\_pixel(shape\_pixels,best\_fit\_node,

elbow\_feature.x+halfway\_dist, j);

The *best\_fit\_node* is the goal node which the optimal hand position would be, however it sometimes falls outside of the shape, so the *find\_closest\_shape()* function is used to find the *shape\_pixel* that is closest to the *best\_fit\_node.*

This detection function encountered some problems in the original implementation. Initially the loop iterated through the *outline\_pixels* *vector* rowby row when finding pixel neighbours. This caused many of the hand positions to drift towards the centre of the image, because a nearby *outline\_pixel* may not necessarily be the next *outline­\_pixel* in the *vector*. So instead of iterating through each pixel in the *vector*, the function looks at nearby pixels in terms of (x, y).

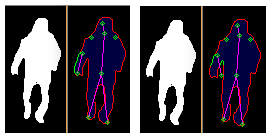


Fig 21. The current hand detector (left) compared with the old (right). Outline pixels used

highlighted in green.

### Dealing with bad shapes

Throughout the runtime, *PeopleFinder* encounters many ‘bad shapes’. These shapes are a result of heavy interference in *BGS* and can cause the *PeopleFinder* to break at various stages in pedestrian detection. It is important that the *PeopleFinder* can handle the errors as bad shapes are inevitable. *Is\_within\_bounds()* was useful for ensuring that the features remain within the dimensions of the image. Some of the feature detection functions would return null exceptions if a *best­\_fit\_node* was unable to be found, so the variable had to be set to the first pixel it could find beforehand.

*PeopleFinder* initially struggled to handle many of the errors produced during runtime due to the variation in input shapes (see 24/03/2017-26/03/2017 of diary log). Try-catch blocks were used for the functions where errors were occurring, which were useful for catching errors in *draw\_skeleton()*. The problems persisted in the feature detection functions despite try-catch blocks being implemented. C++ has no concept of a null pointer exception, which is what was occurring in most of the feature functions. The *bad\_skel\_flag* was added as a class member to *PeopleFinder*, it is a *bool* that is flagged as true whenever a ‘bad feature’ is detected. This means that the *creat\_skeleton()* process could break early, preventing more features being found on a ‘bad shape’. This flag was also useful for the training stage, as it could skip over images containing ‘bad shapes’.

## RecordLog

### Setup

*RecordLog* makes use of the *iomanip* and *fstream* libraries for creating and outputting data to a HTML file. It has an *ofstream* class member that refers to the file output, along with the total number of records in the log. *Init\_log()* creates the new file and adds the opening table tags. The function uses *get­\_date()* to convert the current date into a *string* which is used for the file name. In *new\_record()* the shapes/source shapes from *BGS and BlobDetector* are saved with different file names based on their type (source or model) and the record number*.* The paths to these images are hard coded into new table row tags in the HTML file. Having to store each image individually can be inefficient, but storing them all within the *record\_log* folder encapsulates them into the same place.

# Testing

## Overall Approach to Testing

Ensuring the robustness of the *PeopleFinder* was a key priority for testing the pedestrian detector. While the success of the project could be defined by its pedestrian finding performance, it is vital the *PeopleFinder* can handle a wide variety of shapes without crashing. The project can handle issues such as videos not being present and training directories being empty. The video player from *BGS* and *RecordLog* show the program highlighting shapes in real time, but the main objectives of the project stem from *PeopleFinder*. If the tests show the *PeopleFinder* class finding features while avoiding bad shapes, the localised features, training and noise differentiation objectives can be considered complete. How well it finds pedestrians also needs to be considered to examine the performance of the program.

## Unit Testing

Building pedestrian models is the core feature of the project design, the robustness of each algorithm is important for the whole class. Visual Studio provides a unit testing framework for C++. This framework is useful, as it is embedded into the IDE making it easy to navigate, and tests can be stored in a different project. A series of unit tests were created for the *PeopleFinder* functions. These tests aim to reassure the working behavior of the feature detection functions, with some tests checking for exceptional behavior with bad shapes. The *AutoSurvTests* class in the *UnitTests* projectcreates a *PeopleFinder* object, with *shape­\_pixels* and *outline­­\_pixels vectors* similar to the ones in *create\_skeleton().* These unit tests require two image files to be present in the *AutoSurvCV* folder, “test\_good.png” and “test\_bad.png”. The good test image is a contour outline of a shape, with the inner shape pixels already highlighted. Flood filling the shape beforehand allows the tests to call each feature detection function individually, rather than using *create\_skeleton()*. The bad test image is a contour outline of a messy shape that isn’t highlighted. The bad shape is used with the *create\_skeleton()* function to test its ability to break early when a call to a feature detector function is made on a bad shape.

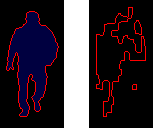


Fig 22. Example of the good shape (left) and bad shape (right) used in testing.

*LoadGoodImageForTestingTest* simply tests whether the test project can open the good image or not. It was useful to have this separate test as the project failing to reach the image files was a common issue. *LoadAndHandleBadImageTest* attempts to read the bad image file, and runs *create\_skeleton()* to see if it can detect and break out of analysing a bad shape. The test passes if the bad skeleton flag was raised. The other tests call each feature detection function individually on the good shape, the functions pass if the feature position is within the dimensions of the image.

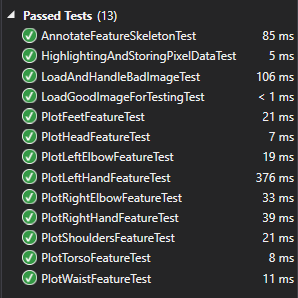


Fig 23. A list of the test methods passing.

Unit testing proved useful during the development process, if the system crashed during runtime it was difficult to identify the problematic function. Shapes could be saved during the runtime and put through the tests to see which function in *create\_skeleton()* was causing the issue.

## System Testing

Testing computer vision systems can be difficult, the results from a program such as the pedestrian detector can be interpreted in different ways. Whether the shapes extracted look like pedestrians is down to the user’s perspective. One way of testing the systems performance is to use ground truth data. Ground truth data consists of observations of the events that occurred in the video. CAVIAR provides ground truth data for each video, the data describes the locations and movements of each person. The performance of the pedestrian detector can be examined using the ground truth data. Information such as frame number, pedestrian id and size can be compared with the *RecordLog* output.

A series of tests were done to compare the number of pedestrians that appear each second in *RecordLog* and the ground truth data. Ground truth/*RecordLog* entries with the same frame numbers contain multiple pedestrians. If the pedestrian detector is finding similar numbers of people as the ground truth specifies, the system accuracy can be accepted. These tests can also highlight problematic scenarios, if the system fails to detect a pedestrian, the ground truth information may reveal why that was the case. These tests also show the impact that the *BGS* parameters have on the performance of system, as an alternative threshold could mean the difference between finding or missing a person.

In the following experiments, the pedestrian detector was executed using Daimler’s *left­\_groundtruth* images for training on CAVIAR’s “OneStopEnter2cor.mpg” video. The *BGS* history/threshold parameters for each test were changed to show how they affect the overall model.

Fig 24. High history, high threshold

Fig 25. Low history, high threshold

Fig 26. High history, low threshold

Fig 27. Low history, low threshold

These show the number of pedestrians *PeopleFinder* believed were in each frame versus the number of people that were actually in the frame via the ground truth data. The closer the *PeopleFinder*’s results are to the ground truth, the better the performance. As you can see in Fig 25, the lower history causes the *PeopleFinder* to miss a pedestrian between frames 400-650. Fig 26 shows the higher history with a lower threshold increased the accuracy of detection. The two pedestrians between frames 400-1000 were tracked relatively smoothly, and was optimal for detecting the pedestrian between frames 1400-1800. The *PeopleFinder* in Fig 26 over estimated the number of people in the scenes, especially towards the end of the video when there were more people present. This increases the false positive rate as it is finding more pedestrians than the ground truth states, as previously discussed, false positives are preferable over false negatives (see section **1.2.6.**). The settings in Fig 27 showed more uncertainty than the settings in Fig 26. This could be due to the lower history parameter causing shapes to degrade more quickly as people stood still, leading to fluctuations in total pedestrian counts.

These tests show that the pedestrian detector can detect pedestrians in the video footage. The peaks and troughs in the figures show that the system has brief uncertainty when pedestrians enter or leave the scene. The most difficult pedestrians to detect in “OpenStopEnter2Cor.mpg” were between 100-400 and 1200-1800. The pedestrians in these frames were at the far end of the corridor, so the smallest amount of noise would greatly interfere with the shape quality. Overall a lower threshold results in a lower false negative rate, as fewer pedestrians were undetected when the thresholds were lower. This also means that the *BGS* is more susceptible to movement/reflections with a lower threshold, increasing the chance of false detections. For this video, the lower history proves costly for the performance. Many of the pedestrians stand still at certain points in the image, causing their shapes to degrade and break down at lower history values. These tests show the importance of tuning the system variables to suit the environment, as reducing the threshold value made the difference in detecting certain pedestrians such as the one between frames 1400-1800 in Fig 26. There also other factors to consider besides pedestrian numbers, such as the shapes including the actual pedestrian, what the shapes are classified as, and the amount of interference being recorded.

# Critical Evaluation

## Project retrospective

Overall, the pedestrian detection system can detect and record people moving in and out of a scene in a video file. By combining BGS with the *PeopleFinder* algorithms, the program provides a relatively simple solution to a complex problem. Many computer vision systems require extensive training on mixed examples such as Viola-Jones [5], whereas this project requires no negative examples and can apply low-detail classification. Provided the pedestrians clothing isn’t identical to the background, the record log shows that the program can detect them regardless of appearance. The record logs also show the program can detect pedestrians with different orientations, and by summarising people as binary shapes it removes the risk of handheld objects interfering with detection. It is a flexible application as it can be applied to a wide range of videos, as the BGS variables can be tuned to suit the environment. There is confidence that this project could be used on a 10 hour length CCTV and build a summary of each pedestrian that was in the scene. This would save law enforcement time and resources for these procedures, and reduce the human error of missing a person in a particular frame. The project shows some levels of precision, as it can select interesting shapes while ignoring small areas of noise that appear in BGS. The live feedback is optimised, the release version runs at a consistent 25 FPS while applying pedestrian detection each second.

This biggest issue the system faces is interference caught in the BGS stage from occlusions and other factors. When pedestrians occlude each other, they morph into a single shape that the classifier writes of as noise. This also deceives the *RecordLog* as the occluded shape only counts as one pedestrian. The project was also developed without vehicles in mind, the *create\_skeleton()* function would most likely struggle with scenes containing scooters or cyclists. FEATURE BUILD LIMITATIONS

While the project is flexible with video inputs, it would take a performance hit on higher resolution videos as it resizes shapes to a uniform size (128x64). The dataset videos had dimensions no larger than 360x290, so large shapes from high resolution videos would suffer from information loss. This would have no effect on the source image saved in the *RecordLog*, but would affect the pedestrian model image as its size is severely reduced.

Struggles with. Occlusions due to blobby summary, while flexible, some videos where it could struggle. Feature building limitations (elbows). Look through Log and critique it.

## Code design and work process

## Additional features

Suit livestream applications. Replace timestamp with date/time.

Examiners expect to find in your dissertation a section addressing such questions as:

* Were the requirements correctly identified?
* Were the design decisions correct?
* Could a more suitable set of tools have been chosen?
* How well did the software meet the needs of those who were expecting to use it?
* How well were any other project aims achieved?
* If you were starting again, what would you do differently?

Other questions can be addressed as appropriate for a project.

Such material is regarded as an important part of the dissertation; it should demonstrate that you are capable not only of carrying out a piece of work but also of thinking critically about how you did it and how you might have done it better. This is seen as an important part of an honours degree.

There will be good things and room for improvement with any project. As you write this section, identify and discuss the parts of the work that went well and also consider ways in which the work could be improved.

In the latter stages of the module, we will discuss the evaluation. That will probably be around week 9, although that differs each year.

PRUNE WORDS FROM EACH SECTION, MOSTLY BACKGROUND .

DISCUSS HOW CODE COULD BE STRUCTURED BETTER, REFACTORED RECORDS INTO UNIQUE CLASS. PEDESTRIAN MODEL OBJECTS. More flexible vectors(index\_shapes\_found instead of 20). More features->more detailed classification-> enum more classification types. First time C++ was used in a while, more integral structure, exploration into class types.

Incorporate time delta into video player to accomadate lag.

Shape within shape detection to break up occluding shapes.

Design perspective: use case diagrams pointless-lack of inputs, next time incorporate some time structure and review SELF ASSESSMENT the log, would help with coding sessions. Further planning before implementation, was using “if you don’t need it, don’t implement it” mentality”, useful but often short sighted.

Ideal program would have threshold adjustment in real time to see changes, rather than restarting the program.

Few words on opencv/visual studio. (visual studio automatically sync settings between test project and main project, libraries etc.).

# Appendices

The appendices are for additional content that is useful to support the discussion in the report. It is material that is not necessarily needed in the body of the report, but its inclusion in the appendices makes it easy to access.

For example, if you have developed a Design Specification document as part of a plan-driven approach for the project, then it would be appropriate to include that document as an appendix. In the body of your report you would highlight the most interesting aspects of the design, referring your reader to the full specification for further detail.

If you have taken an agile approach to developing the project, then you may be less likely to have developed a full requirements specification. Perhaps you use stories to keep track of the functionality and the ’future conversations’. It might not be relevant to include all of those in the body of your report. Instead, you might include those in an appendix.

There is a balance to be struck between what is relevant to include in the body of your report and whether additional supporting evidence is appropriate in the appendices. Speak to your supervisor or the module coordinator if you have questions about this. GITHUB LINK DIARY LINK. Microsoft windows library <https://www.draw.io/> glyffy. OPENCV. EXCEL SPREADSHEETS

* 1. Third-Party Code and Libraries

If you have made use of any third party code or software libraries, i.e. any code that you have not designed and written yourself, then you must include this appendix.

As has been said in lectures, it is acceptable and likely that you will make use of third-party code and software libraries. If third party code or libraries are used, your work will build on that to produce notable new work. The key requirement is that we understand what is your original work and what work is based on that of other people.

Therefore, you need to clearly state what you have used and where the original material can be found. Also, if you have made any changes to the original versions, you must explain what you have changed.

As an example, you might include a definition such as:

**Apache POI library** – The project has been used to read and write Microsoft Excel files (XLS) as part of the interaction with the client’s existing system for processing data. Version 3.10-FINAL was used. The library is open source and it is available from the Apache Software Foundation **Error! Reference source not found.**. The library is released using the Apache License **Error! Reference source not found.**. This library was used without modification.

* 1. Ethics Submission

This appendix includes a copy of the ethics submission for the project. After you have completed your Ethics submission, you will receive a PDF with a summary of the comments. That document should be embedded in this report, either as images, an embedded PDF or as copied text. The content should also include the Ethics Application Number that you receive.

* 1. Code Samples

This is an example appendix. Include as many appendices as you need. The appendices do not count towards the overall word count for the report.

For some projects, it might be relevant to include some code extracts in an appendix. You are not expected to put all of your code here - the correct place for all of your code is in the technical submission that is made in addition to the Final Report. However, if there are some notable aspects of the code that you discuss, including that in an appendix might be useful to make it easier for your readers to access.

As a general guide, if you are discussing short extracts of code then you are advised to include such code in the body of the report. If there is a longer extract that is relevant, then you might include it as shown in the following section.

Only include code in the appendix if that code is discussed and referred to in the body of the report.

Random Number Generator

The Bayes Durham Shuffle ensures that the pseudo random numbers used in the simulation are further shuffled, ensuring minimal correlation between subsequent random outputs.

// Some example code here…

# Annotated Bibliography

This final section should list all relevant resources that you have consulted in researching your project. Each reference should also include a brief annotation.

1. John Perdikaris, “*Physical Security and Environmental Protection*”, January 2014, Chapter 5.3 Surveillance and Counter Surveillance, page 147.
2. UK Government, “*Plug and Play Autonomous Sensors*”, 2015. Available: <https://www.gov.uk/government/news/plug-and-play-autonomous-sensors>. [Accessed 06 February 2017].

A press release discussing the development of new concepts in modular autonomous sensing. Provides details on some of the technologies used.

1. Gavin J.D Smith, “*Behind the Screens: Examining Constructions of Deviance and Informal Practices among CCTV Control Room Operators in the UK*”, pages 388-389, 2004. Available: <http://surveillance-and-society.org/articles2(2)/screens.pdf>. [Accessed 18 April 2017].
2. BBC, "*Missing Corrie Mckeague: Suffolk Police ‘search lacks resources’”,* November 2016. Available: <http://www.bbc.co.uk/news/uk-england-suffolk-38052508>. [Accessed 18 April 2017]
3. Paul Viola & Michael J. Jones, “*Robust Real Time Face Detection”,* received September 2001, pages 138-139. Available: <http://www.vision.caltech.edu/html-files/EE148-2005-Spring/pprs/viola04ijcv.pdf>. [Accessed 19 April 2017]
4. Shanshan Zhang, Christian Bauckhage, Armin B. Cremers, “*Informed Haar-like Features Improve Pedestrian Detection*”, 2014 IEEE Conference on Computer Vision and Pattern Recognition, pages 949-950.
5. Navneet Dalal & Bill Triggs, “*Histograms of Oriented Gradients for Human Detection*”, 2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR’05), Chapters 3 & 6.4.
6. EC Funded CAVIAR project/IST 2001 37540, found at URL: [*http://homepages.inf.ed.ac.uk/rbf/CAVIAR/*](http://homepages.inf.ed.ac.uk/rbf/CAVIAR/). [Accessed 20 April 2017]

Dataset containing video footage for testing/demonstrating the project.

#### [9] Jérôme Berclaz, François Fleuret, Engin Türetken, Pascal Fua “Multiple Object Tracking using K-Shortest Paths Optimization” IEEE Transactions on Pattern Analysis and Machine Intelligence 2011

CVLab – EPFL. “*Multi-Camera pedestrian videos*”, available at: <http://cvlab.epfl.ch/data/pom>. [Accessed 20 April 2017]

[10] F. Flohr and D. M. Gavrila.   
*“PedCut: an iterative framework for pedestrian segmentation combining shape models and multiple data cues.”* Proc. of the British Machine Vision Conference, Bristol, UK, 2013.

Daimler dataset, available: <http://www.gavrila.net/Datasets/Daimler_Pedestrian_Benchmark_D/Daimler_Pedestrian_Segmentatio/daimler_pedestrian_segmentatio.html> [Accessed 20 April 2017]

[11] OpenCV, “*Open source computer vision and machine learning software library*”, available: <http://opencv.org/> [Accessed 24 April 2017]

ANNOTATE/DESCRIBE REFERENCES CAVIAR KEY FILES USED