University Of Southern Queensland
Faculty Of Engineering & Surveying

Machine Vision Based Traffic Monitoring With Real Time Tracking

A Dissertation submitted by

P. L. Menetrier

in fulfilment of the requirements of

ENG4112 Research Project

towards the degree of

Bachelor of Electrical & Electronic Engineering

Submitted: October, 2008

University of Southern Queensland

Faculty of Engineering and Surveying

ENG4111/2 *Research Project*

Limitations of Use

The Council of the University of Southern Queensland, its Faculty of Engineering and Surveying, and the staff of the University of Southern Queensland, do not accept any responsibility for the truth, accuracy or completeness of material contained within or associated with this dissertation.

Persons using all or any part of this material do so at their own risk, and not at the risk of the Council of the University of Southern Queensland, its Faculty of Engineering and Surveying or the staff of the University of Southern Queensland.

This dissertation reports an educational exercise and has no purpose or validity beyond this exercise. The sole purpose of the course pair entitled "Research Project" is to contribute to the overall education within the student's chosen degree program. This document, the associated hardware, software, drawings, and other material set out in the associated appendices should not be used for any other purpose: if they are so used, it is entirely at the risk of the user.

Prof F Bullen

Dean

Faculty of Engineering and Surveying

Contents

List of Figures **vii**

List of Tables **ix**

Chapter 1 Abstract **1**

Chapter 2 Certification **2**

Chapter 3 Introduction **3**

Chapter 4 Literature Review **5**

4.1 Background Subtraction 5

4.2 Optical Flow 6

4.3 3D Model Tracking 7

4.4 Region Based Tracking 8

4.5 Active Contour Based Tracking 8

4.6 Feature Based Tracking 10

Chapter 5 The Implementation **12**

5.1 Language Selection 12

5.2 The Video Source 13

5.3 Video Decompression and Graphics 14

5.4 DirectShow Details 15

Chapter 6 Background Modeling **18**

6.1 Traffic Footage Analysis For Background Subtraction 18

6.2 Initial Implementation of Background Subtraction 19

Chapter 7 Striping and Low Level Object Segmentation **21**

7.1 Introduction 21

7.2 The Striping Process 21

7.3 Low Level Object Segmentation Using Strips 24

Chapter 8 Object Tracking **29**

8.1 An Overview Of The Object Tracking Engine 29

8.2 Addition and Removal of Objects From the Object List 30

8.3 Object Position Sampling 31

8.4 Movement Filtering and Position Prediction 32

8.5 Mean Shift Tracking 34

8.6 Resizing Bounding Boxes 34

Chapter 9 Late Additions 38

9.1 Late Project Additions 38

9.2 Oriented Bounding Boxes 38

9.3 Drawing Oriented Bounding Boxes 39

9.4 Determining Bounding Box Orientation 39

9.5 Applying Mean Shift to Oriented Bounding Boxes 40

9.6 Resizing Oriented Bounding Boxes 42

9.7 Vehicle Counting And Speed Estimation 43

Chapter 10 Conclusions 45

Bibliography 48

Appendix A Project Specification 50

Appendix B Source Code 51

B.1 BackgroundSubtraction.h 51

B.2 ObjectTracker.h 52

B.3 Striper.h 57

B.4 VideoCapture.h 62

B.5 VisionX.h 63

B.6 VisionX.cpp 66

B.7 VideoCapture.cpp 87

B.8 Stiper.cpp 96

B.9 SelectionRectangle.cpp 117

B.10 ObjectTracker.cpp 120

B.11 ImageProcessing.cpp 197

B.12 BackgroundSubtraction.cpp 205

List of Figures

5.1 The FinePix S6500 Camera Used for Video Capture 14

5.2 A Series of filters chained together with GraphEdit to decode and render a video file 15

6.1 Noise generated in background subtraction routines due to camera movement and sporadic auto zooming problems. The screen shot should be blank except for the moving vehicle. 19

6.2 Result of Background Subtraction on a typical frame from traffic footage. The moving vehicles are clearly highlighted which generally makes it easier to implement object segmentation and identification. 20

7.1 Possible Object Structure. Used to store the information about each strip 22

7.2 The running sum algorithm used to calculate where strips are. 26

7.3 An early implementation of the striping algorithm where each strip is represented as a different color. 27

7.4 A better striping algorithm where close horizontal strips are joined together 27

7.5 The 'ObjectFound' Structure which contains details of a low level object 28

8.1 Bounding boxes around objects of interest 31

8.2 The 'AddObject(..)' Function 36

8.3 The structure of one element of the MSamples Array 37

9.1 Vector addition to achieve directional update 41

9.2 Tracking Line Proximity Calculations 44

9.3 Speed Estimation 44

10.1 Two vehicles recognized as one due to a previous occlusion 47

List of Tables

Chapter 1

Abstract

The aim of this project is to create software on a Windows platform that uses pre recorded video of traffic scenes to count and give speed estimation of vehicles using real time tracking techniques. The software was developed in C using Microsoft Visual Studio 2008 and utilizes Microsoft DirectShow and the Windows GDI+ graphics library.

The system utilizes background modeling and subtraction to distinguish foreground objects of interest such as moving vehicles from the static background. Object segmentation is then performed to group low level objects in the background subtracted image into blobs. Once the blobs pass certain movement criteria they are passed up to higher level tracking routines. The higher level tracking routines utilize mean shift tracking to follow the object of interest through the scene. Oriented bounding boxes are employed to provide a tighter fit around tracked vehicles and to provide visual feedback of the tracking process.

The program achieved the goals of providing real time tracking, counting and speed estimation but not without numerous glitches. The software provides a bare minimum system for this type of application and needs the implementation of additional techniques such as shadow suppression, path learning and occlusion reasoning.

Chapter 2

Certification

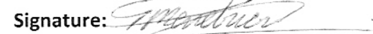
CANDIDATES CERTIFICATION

I certify that the ideas, designs and experimental work, results, analysis and conclusions set out in this dissertation are entirely my own efforts, except where otherwise indicated and acknowledged.

I further certify that the work is original and has not been previously submitted for assessment in any other course or institution, except where specifically stated.

Name: Paul Louis Menetrier

Student Number: 0050040066

Signature: 

Date: 28/10/08.

Chapter 3

Introduction

Machine Vision is a field which provides great utility to a growing number of industries. Computer Vision systems can be found in many roles that were historically occupied by people. They have major advantages over human inspectors because they never get tired or bored and in many situations can be programmed to produce very consistent or accurate results. The food industry employs Machine Vision systems for grading of produce, quality control and detection of foreign bodies. These vision systems are not only utilized in low tech industry but also in areas such as Semiconductor production which has become heavily dependent on Computer Vision and would not be able to maintain current production outputs without it.

Most of the Vision systems employed today are tailored for specific conditions and a specific task. In many inspection applications the lighting and other optical properties are carefully controlled to allow the inspection systems to work at their maximum potential. Vision systems that must be able to operate under different conditions where there maybe unknowns or complex environments are much more difficult to produce. Traffic surveillance falls into this category as there are many potential backgrounds and vehicles to be tracked as well as other objects which may interfere such as pedestrians.

Even though there a workable commercial traffic surveillance systems in use they do have problems especially under congested traffic conditions. However, these systems do have comparable accuracy to loop detectors which are traditionally used for traffic

monitoring and can be considered more reliable and able to yield more information about traffic flow. Advantages of automatic video surveillance are the ability to gauge speed, determine traffic congestion and provide incident detection.

There is still a lot of room for the improvement of the algorithms employed in automatic traffic surveillance systems and the focus of this project will be to explore current systems and to implement new or existing techniques to create software to count and give speed estimation of vehicles in a road scene using real time tracking.

Chapter 4

Literature Review

4.1 Background Subtraction

Background subtraction is a very commonly employed technique that can be quite powerful and is used in many attempts at machine vision based traffic surveillance. It is most useful when applied in a situation with a very static background. Such a situation exists for a traffic surveillance camera as it would usually be stationary, attached to permanent fixings. The vehicles and pedestrians in the scene can be considered to be the only moving objects and are commonly called foreground objects.

There are numerous approaches to tracking objects in a scene but most involve significant computational effort to firstly detect and then track the object along its path. Background subtraction provides a means to reduce this computation by limiting the area that needs to be processed in each frame. A tracking system that employs background subtraction will utilize a system to determine what the background is and there are numerous approached to doing so. Michalopoulos (1991) employs a background model in the Autoscope system. In this system the background model is created by a finite impulse response filter of the form:

$$BE(j,k) = clI(j,k) + (1 - cl)BE(j,k-1) \tag{4.1}$$

Where BE is the background estimation pixel array, j is the element number and k and k-1 denote the current and previous backgrounds. The constant cl controls how quickly the background will update. Equation 4.1 allows the background to change slowly but still maintain an accurate representation of the background. Areas of the image that contain a vehicle being tracked are not updated until the object moves on to another area of the image. This can be accomplished with the use of a simple bit mask.

4.2 Optical Flow

Optical flow is a analysis technique that tries to recover useful information about the movement in a video stream. As objects move in an image or the background moves itself the pixel intensities that represent the image move across the screen and create an array of moving pixels or optical flow. From the movement of these pixels object segmentation, velocity field estimation and other properties can be derived. Discontinuities in optical flow can be used to isolate regions of the image corresponding to different objects which provides a means of segmentation, a fundamental step in vision systems. (Davies 2005) gives a good overview of the basic ideas behind optical flow. Davies points out that optical flow might seem like a very intuitive way of extracting information about the movement in an image but like all other techniques in this field it has drawbacks that must be addressed.

To detect the movement of objects in an image a simple technique of image differencing can be employed. This is just simply the result of subtracting a reference image with a subsequent image in the next instant of time. The difference between the two should highlight movements. The problem with this approach is that the optical flow may not reveal enough meaningful information about what motion has occurred. Take for example the situation of a moving object of constant intensity as described in (Davies 2005). The difference image generated will indicate movement in front and behind the object but because of its constant density no information about movement in the central region of the object is present.

Other problems arise in determining velocities of the optical flow. Horn & Schunck

(1981) states that the optical flow cannot be determined at a point independently of any others. A mathematical derivation of this fact is given by (Horn & Schunck 1981) and he shows that the velocity normal to the surface gradient can be calculated but the component parallel to a surface edge can not be resolved without other constraints. Horn & Schunck (1981) suggests using a relaxation labelling technique to robustly solve the problem and this has generally been accepted as the solution to it. Overall it does seem that this solution is overkill considering the complexity of the problem. The same results should be possible with simpler methods that have more emphasis on clever algorithms which act on realistic discrete image data rather than have their roots in idealistic mathematical solutions that will never translate easily to a discrete faulty environment. Horn & Schunck (1981) also makes the point that this technique will have problems with occluded edges as their may be sharp differences in optical flow rather than a smooth flow across an edge which this technique thrives on.

Once an optical flow field has been generated indicating the flow of pixels across the image information on the motion with it can be extracted. If an object is moving in one particular direction than there should be an isolated region where the flow field is pointing in one direction at a uniform rate. If there is an isolated concentration of flow consisting of two or more directions then there maybe rotation occurring. Software can also be programmed to detect the camera moving towards a scene. In this situation all stationary points in the image will appear to be emanating from a focus of expansion or FOE.

Optical flow can yield useful information about the movement of objects in a scene, but as the many complex papers on this subject suggest it is not often easy to implement and has many inaccuracies. Optical flow is a system which is best used in conjunction with other techniques to create a workable system.

4.3 3D Model Tracking

3D model based tracking uses wire frame models for vehicle recognition. The main point of research is to identify the models that enable recognition on the largest number of

vehicles. Relying on geometric models to identify vehicles may be the weakest approach as vehicle designs vary so much and the very difficulty of edge and line detection places serious problems for the practical implementation of these systems.

Koller, Daniilidis & Nagel (1993) implements a 3D model based tracking system. In this system straight line edge segments extracted from the image are compared to a 2d model outline made by projecting a 3d model onto the image plane. An interesting addition to this system in the addition of an illumination model so that lighting conditions such as shadows can be understood. The 3D geometric models are parameterized by 12 length parameters this enables different vehicles to be identified from the same basic 3d model.

4.4 Region Based Tracking

Region based trackers try to identify a region containing a vehicle or other trackable object and then track this region by comparing it to the original segmentation. The initial segmentation is usually done by background subtraction which will highlight any foreground objects. The foreground objects will look like blobs after the background subtraction and must then be segregated into separate objects.A dynamic background model is usually used to allow for small variations in the background image caused by changes in lighting etc.

4.5 Active Contour Based Tracking

This tracking technique is based on finding the contours of each trackable object which are sometimes called snakes. To initially create the contours background subtraction is usually used. Edge detectors can then be employed on the subtracted image to find the rough outer edges of each trackable object. Bezier splines can then be used to smooth the outer contours further and create an outline that is easier to track. Snake or active contours are also commonly used. These lines are active in that they per sue important features such as edges. There is an advantage in using contour based tracking in that

it saves on computation. The major drawback to this system is finding the original contour of a trackable object when partially occluded.

Kass, Witkin & Terzopoulos. (1988) provides a good overview of the use of active contours or snakes. These features use the notion of image forces to push the snake towards features of interest. The idea here is that the snake will attach itself to an outer boundary when placed near it and will continue to track to this feature in subsequent frames thus provided object tracking. Mathematically the snake tries to minimize its image energy. If the position of the snake is represented parametrically by v(s)=(x(s),y(s)) The energy of the snake can be expressed as:

$$\begin{aligned} E_{snake} &= \int_0^1 E_{snake}(v(s))ds, \\ &= \int_0^1 E_{int}(v(s)) + E_{image}(v(s)) + E_{con}(v(s))ds, \end{aligned}$$

where E_{int} is the internal energy of the snake, E_{image} is the total energy from the image and E_{con} are any external constraint forces put on the snake.

Kass et al. (1988) suggests a method for describing the energy of edges. If the energy of an edge is set by equation 4.2 then the snake will be attracted to sections of the image with large image gradients.

$$E_{edge} = -|\nabla I(x,y)|^2 \tag{4.2}$$

Kass et al. (1988) gives an example of a snake wrapped around an image of a pear that is using this type of intensity gradient operator. The snake is pulled away from the pear and when released snaps back around it. The image is idealistic with a simple background and no objects close by to complicate the matter so it does suggest it works but doesn't give a good impression of its real functionality.

Koller, Weber & Malik (1994) implements a contour based tracking system. Initially after background subtraction a thresholding to the image gradient and the time derivative of the image is made. Initially a convex polygon is used to enclose a detected

object. Koller et al. (1994) makes the point that a polygon will be no good for tracking as the points defining the polygon will change greatly during repeated attempts at tracking the object. This will make it difficult for any algorithm to track its movement in a consistent way. The solution to this problem is to use splines or snakes. Koller et al. (1994) uses splines with 12 control points to enclose the object and replace the convex polygon approximation.

4.6 Feature Based Tracking

Feature based tracking relies on an approach of tracking much smaller features in an image such as corners and lines. This has the advantage that even under partial occlusion objects can still be tracked as some of the markers which identify the object will be visible.

In order to segment the objects Beymer, McLauchlan, Coifman & Malik (1997) uses a common motion constraint so that features moving in a similar motion are grouped together. This could become difficult in built up traffic conditions and it must be able to respond to very small changes in acceleration and lane drift. In order to overcome this problem of segmenting close vehicles (Beymer et al. 1997) integrates the tracking data of many frames. Only features that are tracked from an entry region to an exit point are allowed to be grouped. This gives the program more information to segment out close vehicles but takes more time to do so. In the case of free flowing traffic with constant spacing the grouping algorithms employed by (Beymer et al. 1997) would fail so he adds a spatial/proximity cue to aid in the grouping process under these conditions.

Corners are the major sub feature for tracking in (Beymer et al. 1997). A 2x2 matrix mask is used and when the numerical rank of the matrix is 2 a corner is detected. A 9x9 feature of the area where the corner has been detected is extracted for feature tracking. The tracking feature then tracks the features from the user defined entry region to the exit region using Kalman filtering where vehicle acceleration is incorporated into the system noise dynamics. Beymer et al. (1997) also implements an interesting addition to the Kalman filter. In this scheme the distance between the Kalman filter prediction

and the measured value for the feature are compared and anything above a threshold distance in marked as a bad tracking feature. This helps to filter out badly identified features so the tracking can be more accurate.

Chapter 5

The Implementation

5.1 Language Selection

There are many languages that the vision system could be implemented in and the following discussion outlines the rationale behind the choices I have made. One of the more popular languages on the Windows platform is Visual Basic. This language has evolved from being mainly an interpreted language to a professional compiled language with optimizations that allow it to almost match the more mainstream languages such as C. The major drawback I had with Visual Basic is my unfamiliarity with it. I did delve into the language to assess its potential for this project and my assessment of it generally is not as rosy as many others have made. The main selling point of Visual Basic is that is derived from traditional BASIC and the language constructs are supposedly easier to use and understand than a conventional language. After seeing them in practice I believe that the loop structures, flow control etc. are not any easier than a language such as 'C' and in my opinion are just as difficult to read through and maintain as the 'C' equivalents. The other major selling point for Visual Basic is the graphical component to the design where 'forms' can be visually assembled and integrated with code. I would argue that the traditional Visual Studio resource compiler for C does largely the same thing and if anybody knows how to use it well they can use the same graphical techniques with similar amounts of effort. One of the major drawbacks using Visual Basic is that interfacing with low level routines in graphical

libraries such as DirectShow and DirectX becomes more difficult. Because I want easy access to these functions and for the other listed reasons I decided not to use Visual Basic.

Java was also considered a serious option for the project. I have had some experience and training with the Java language and was very impressed with the recent advances in speed where in some cases it can outperform C which is often considered one of the faster languages to choose from. Whilst trialling the language I had significant doubts about the libraries used to decode movie files and grab samples. The libraries supplied did not support many formats and more specifically did not support the MJPEG formats used by the camera I was to use for the project. Even though I had a liking to the Java language I was forced to abandon this option due to its poor support and lack of examples for decoding Movie streams.

I decided to go with a safe option that I was familiar with, that being C/C++ using Win32. The majority of the code I wrote just uses plain 'C' but I liked the idea of using the class features of C++ to group like functions and data together. Using C/C++ I have direct access to all the low level functions that I will need with minimal fuss. There is also the added advantage that the majority of the Windows SDK's for DirectShow/DirectX have functions specified and described with C calling conventions and numerous examples supplied on the Microsoft Developer Network in these languages.

5.2 The Video Source

In a fully operational video surveillance system a camera would be mounted in a strategic location on a roadside or elevated position with a good view over the region of interest, a strip of highway or intersection. The aim of this project is not to implement a fully operational system but rather to develop software techniques which can be utilized in such a system. It is therefore more convenient to gather footage at a chosen place and time and then store that data in a convenient video format on a computer ready for use by the software when needed. The video data is first captured onto the internal memory of the camera and then later downloaded to computer using a standard USB

cable.

The video footage was gathered with the camera mounted on a tripod usually from overpasses with highways running beneath. The camera chosen for the task was a Fuji FinePix 6500 shown in Fig. 5.1. This camera was chosen for the project for no other reason other than the author already owned one before the commencement of the project. The camera supports two resolutions; 320x240 and 640x480. The resolution chosen for the project was 640x480. Ideally it would be more desirable to operate at full high definition but the hardware was not available. At 640x480 the colour depth is 24-bit. The colour data actually takes up 32-bits per pixel with 8 Bits red, 8 Bits green, 8 bits blue and an unused 8 bit alpha channel. This is also the format the software operates with to represent data on screen. The captured video data is stored by the camera in Motion JPEG AVI format and needs decompressing by a codec before it can be utilized by the software.

Figure 5.1: The FinePix S6500 Camera Used for Video Capture

5.3 Video Decompression and Graphics

The video stream as mentioned previously is compressed in MJPEG avi form. In order to get access to the raw video data a codec must be employed to decompress the

footage. It is not the subject of this project to concentrate too much on such matters so a pre existing programming library for windows called Direct Show was employed. DirectShow was bundled with DirectX in the past but is now distributed as part of the windows Software Development Kit. DirectShow allows a programmer to easily compress/decompress, display and also implement custom operations on compressed video and audio files without having to worry about too much detail.

DirectShow comes bundled with a program called GraphEdit. This software allows a developer to visualize and test how he/she intends to process the video/audio stream. GraphEdit uses a filter system structure for the flow of data. This refers to a system where the processing is broken up into a number of stages where each stage is called a filter and has a specific operation to perform on the data stream. Each filter has input pins and output pins. The input pins are connected to the output pins of the previous filter and in this way the filters are cascaded in series with the data flowing though each one. As the data passes through each filter it is transformed in some way depending on the function of the filter. An example of this is Fig. 5.2 which is a screen shot taken from the GraphEdit software showing a series of filters intended to decompress and display an avi file.

Figure 5.2: A Series of filters chained together with GraphEdit to decode and render a video file

5.4 DirectShow Details

The following section goes into more depth with code to show how DirectShow was set up and used to decompress the video stream. The main code for DirectShow setup and operation is encapsulated in the 'VideoCapture' class declared in the VideoCapture.cpp source file and listed in the appendices.

The first operation to be performed when setting up DirectShow is to create the 'FilterGraph' which is the main class that allows filters to be added and removed from it and controls data flow through them. This is done in the class constructor for the 'VideoCapture' class. In order to create this class the Component Object Model system within windows needs to be initialized with the Win32 call 'CoInitialize(NULL);.' Once this function has succeeded then the instance of the filter graph can be created with the call.

```
h = CoCreateInstance(CLSID_FilterGraph,
NULL,CLSCTX_INPROC_SERVER, IID_IGraphBuilder,
(void **)&pGraph);
```

Listing 5.1: Win32 Function to create instance of COM object - in this case the filter graph class'

This creates a COM - component object model class with the class ID $CLSID_FilterGraph$ and stores a pointer to this class in the variable pGraph. Filters to decompress the video file will then be added to the class pointed to by pGraph by using its interfacing functions. The rest of the constructor function initializes variables and then exits.

The next function to be called is the CreateGraph() function of the VideoCapture class. This function actually creates and adds the filters required for the decompression of the video stream to the filter graph class. The filters to be added to the filter graph class are: a file source filter, a MJPEG decompressor filter, a sample grabber filter and a null renderer filter. The file source filter actually reads a source file video stream and hands it on to the next filter the MJPEG filter. This filter decompresses the video data and hands the new data onto the sample grabber. The sample grabber allows the data to pass through itself unchanged and onto the null renderer which does not actually render the footage but makes a termination point for the data. The sample grabber filter does not modify the data as it passes through it is included to provide functions to allow video frames to be grabbed from the filter graph and passed on to the main program for traffic surveillance monitoring.

The code to create and add the filters to the filter graph is virtually identical for each filter. To create each filter the function call is made as in Fig. 5.1 with the appropriate

```
pGraph->AddFilter(pAVIFileSourceF,(LPCWSTR)"AVI Source file");
(void **)&pGraph);
```

Listing 5.2: Interface function to add filter to filter graph

```
Filter->EnumPins(&pAVIFileSourcePins)
```

Listing 5.3: Interface function to obtain the input and output pins of a filter

class ID for the filter required. Adding the filter to the graph is then accomplished by calling the function AddFilter as in Fig. 5.2 bearing in mind that the parameters passed are to add the AVI source file filter.

Once the filters have been added to the filter graph the input and output pins need to be obtained from each filter. This can be done by calling the interface function given in Fig 5.3. This function returns a standard enumeration listing the pins of the filter in this case the variable pAVIFileSourcePins passed to the function. The other pins are then obtained as listed in the source function VideCapture::CreateGraph().

Once the input and output pins have been established the input of one filter can be connected to the output of the preceding filter. This is done in the CreateGraph() function by using the code in Fig. 5.4. The two variables passed to the function are the output and input pins to be connected.

```
pGraph->Connect(pAVIFileSourceOutputPin,
pMJPEGDecompressorInputPin);
```

Listing 5.4: Interface function to connect input and output pins

Chapter 6

Background Modeling

6.1 Traffic Footage Analysis For Background Subtraction

Footage was shot with a FinePix S6500 camera mounted on a tripod but had inherent problems for the vision system as it would sporadically zoom in and out to a very small degree. This occurred roughly every 7 seconds and the zooming effect was only slight but enough to generate significant noise in background subtraction routines. To test the footage it was run through a simple routine which subtracted the current frame from the next to develop a negative. The continuous generation of negatives was represented as a continuous video and a screen shot is given in fig. 6.1. This image shows the result of the small sporadic zooming effect that was unable to be removed from the camera. The screen shot should be blank except for the moving vehicle. Noise was also generated by slight vibration of the camera due to wind and vibrations through the ground which created similar problems as in fig. 6.1. Vibrations from passing traffic was especially bad when the camera was mounted on a small overpass or bridge due to the play built into the structure.

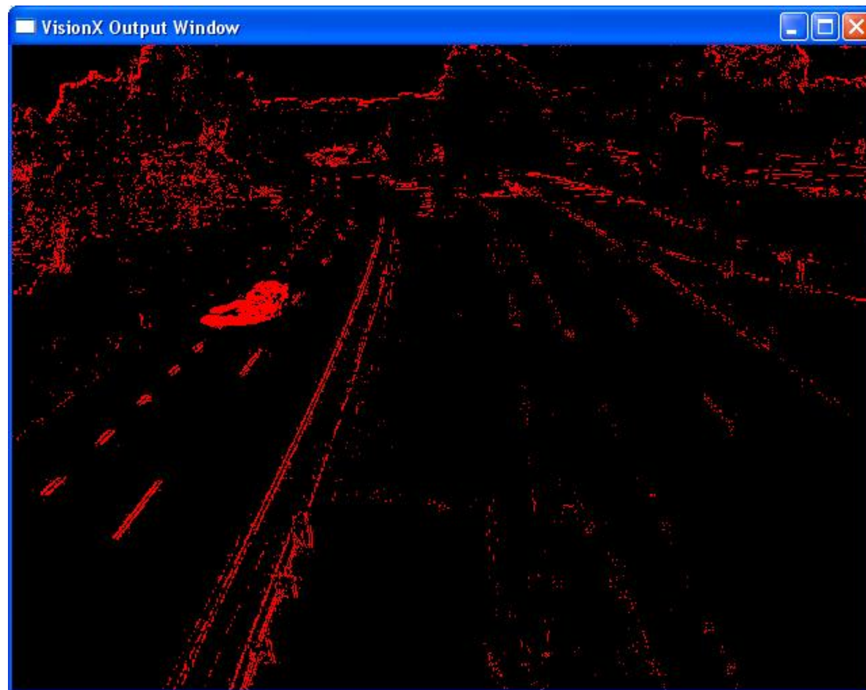


Figure 6.1: Noise generated in background subtraction routines due to camera movement and sporadic auto zooming problems. The screen shot should be blank except for the moving vehicle.

6.2 Initial Implementation of Background Subtraction

The utility of background subtraction was explored and a background modeling routine is incorporated into the software. Utilizing a system described by equation 4.1 the background can be dynamically modeled which allows for small slow changes in the background image. The system was observed to work quite well with moving traffic and clearly highlights the foreground objects in the image. Fig. 6.2 shows the typical results. In this image any pixel larger than a threshold value after background subtraction is colored white.

It can also be noted from the image that the background subtraction routine has not only highlighted the moving vehicles but also their shadows which in this particular footage are fairly large. The vehicle shadows lead to the vehicle being identified in regions of the image that it is not actually in and can impinge on another close vehicles area leading them becoming seen as one vehicle. Shadow removal will be addressed in a later section.

Figure 6.2: Result of Background Subtraction on a typical frame from traffic footage. The moving vehicles are clearly highlighted which generally makes it easier to implement object segmentation and identification.

Chapter 7

Striping and Low Level Object Segmentation

7.1 Introduction

The following chapter is a description of the algorithms applied to the raw background subtracted image. The objective at this stage of the program is to identify regions where the vehicles actually are. This might seem obvious when one looks at the vehicles of an example background subtraction such as in Fig. 6.2, but the program has to specifically define where the bounding box of each vehicle is for further processing at a later stage. This is still a low level process and does not have to be perfect. Situations where close vehicles are represented as one object are acceptable and the expectation is that the vehicles will be distinguished at higher levels of the program. The technique implemented to do this is referred to as 'Striping' and is described hereafter.

7.2 The Striping Process

The Striping process focusses on processing the image row by row and the idea is to allow just one pass of the image for the processing to be completed. The image should contain mostly blank space and the objective is to find where the concentration of white

pixels are on each row and store this information in an array as a series of strips. The starting and ending coordinates of each strip are stored for further processing such as object segmentation later.

The strips found on each row are stored in an array of objects named 'PossibleObject' which is defined in Striper.h listed in the appendices. The object definition is repeated in Fig. 7.1.

```
struct PossibleObject
{
unsigned int xstart;
unsigned int xend;
unsigned int y;
int MemberNo;
}
```

Figure 7.1: Possible Object Structure. Used to store the information about each strip

The 'xstart' member contains the x coordinate of the start of the strip and 'xend' contains the end coordinate of the strip. The 'y' variable is the y coordinate of the strip within the image, where the x and y origin is at the top left hand corner. The 'MemberNo' variable contains the object to which it belongs, this variable will be filled with a valid object number when further processing is done.

A dynamically allocated array of 'PossibleObjects' exists in the 'Stripper' class which also encapsulates all the functions operating on strips. This array holds a preset number of 'PossibleObject' structures for each row which limits the number of strips that can be found on each row. This array is filled with entries when the 'DetectPossibleObjects(...)' function of the 'Striper' class in called.

In order to calculate where the white pixels are the 'DeetectPossibleObjects(...)' function creates a simple running sum along each row. A fictitious window of defined size is placed at the start of the row and extends a defined size along the row. The sum of the pixels is taken within this window. For each white pixel the sum increases by

one. If the sum rises above a given threshold a possible object is detected on this row and a strip is created. To continue processing the row the window is moved across one pixel and the sum of the pixels within the new window is taken again. If the sum is still above the threshold the strip continues along the row. when the sum drops below the threshold the strip is considered to have ended and the the appropriate start and end coordinates are entered into an array.

In order to speed up computation the DetectPossibleObjects(..) function begins by adding the pixels on the current row up to the width of the predefined window size. If a pixel is white the pixel count is incremented by 1 and if 0 it is ignored. When the window needs to be moved on 1 pixel instead of readding all the pixels within the selected window only the next pixel after the current window and the first pixel of the window need be considered. If the next pixel to be included in the window is a 1 then increment the running count and if it is zero decrement one from the count. Also if the first pixel of the window is a 1 decrement the count as a white pixel is leaving the window. The logic of this is expressed in the code snippet of Fig. 7.2.

Amongst this code which is listed within Striper::DetectPossibleObjects(...) there will also be the logic required to identify and record the origin and end of strips this has been omitted from the code for brevity.

The results of an early implementation of the striping algorithm can be seen visually in Fig. 7.3. In Fig. 7.3 there are many instances where there are two strips close together where it may seem more logical to have just one covering the area of both. This is caused by the algorithm having a running sum drop lower than a threshold which terminates a strip and then having the running sum very quickly rise again above the threshold within a few pixels. The algorithm was modified so that the strip only terminates if the threshold does not rise again within 5 pixels or so. The result of this is that there are many more instances where there is just a single strip per row per object as shown in Fig 7.4. This quickens the algorithm even though this is not an issue at this stage and also makes things tidier.

7.3 Low Level Object Segmentation Using Strips

The main idea behind the striping process is to identify and record the areas of the screen where the background subtraction routines have revealed the presence of foreground objects. Once the background subtraction has been applied to the graphics buffer the result is a majority black screen with white pixels denoting background differences above a given threshold. Vehicles or pedestrians will then appear as white blobs. Once the striping routine has been run on the graphics buffer the result is an array filled with data on every strip detected on each row of the image. This data for each strip consists of a starting pixel an end pixel index and a y coordinate as well as a member object ID. The member object ID is used to identify which low level object the strip belongs to.

In order to form objects the strips are grouped together by association. The general requirement for two strips to be associated is to have one of the strips overlapping the other by any margin and that they exist one above the other or with only a small vertical gap between them. The main algorithm to find a low level object begins with a single strip which is passed to the function. An object ID is assigned to this strip and other strips associated with this one are actively searched for on the next row. As new strips are found they have their member object ID's updated and their index added to another array containing characteristics of this object. This is so that all the strips referring to a particular object can be referenced from this array when needed. A more accurate description of the 'ObjectFound' structure is given in Fig 7.5.

Fig 7.5 is the structure that stores the strip indexes of a low level object and also the bounding box coordinates of this object once the appropriate functions have been called to calculate each piece of data. The strip indexes are stored in the array 'StripIndexes' where each element contains an index into another array which in turn contains all the strips found in the current image. The exact layout and implementation can be seen in the 'Striper.cpp' source file listed in the appendices.

The algorithm that fills the structure is listed fully in 'striper.cpp' in the function 'int Striper::FindObjectFromStrip(...)' which can be found in the appendices.

The main advantage to using a system as described above is the simplicity and speed with which objects can be located and segmented. This object segmentation is described as low level because it does have difficulty discriminating objects when they are close together. The main reason for this is that the strips from either object become too close together and by association method used the strips from one object become joined to another and both objects become classified as one. The main use of the system is to quickly identify regions of interest where objects may lie. The actual accurate tracking and segmentation of objects is done by a higher level and is covered in subsequent chapters.

```
BYTE* pBuffer;  //Video buffer array holding a frame from the video stream
int RunningSum = 0;
int i=0;



for (int y=0; y<NumberOfRows; y++)
{

RunningSum = 0;

  for (i=0; i<SampleLength; i++) //Add up the pixels equal to one window
  {
    if (pBuffer[i] == 255) //If white pixel increment running sum
    RunningSum++;
  }

 for (int x=i; x<VideoXLength; x++)
 {
   if (pBuffer[i] == 255) RunningSum++;
   else RunningSum--;

   if (pBuffer[i-SampleLengh] == 255) RunningSum--;
 }

}
```

Figure 7.2: The running sum algorithm used to calculate where strips are.

Figure 7.3: An early implementation of the striping algorithm where each strip is represented as a different color.

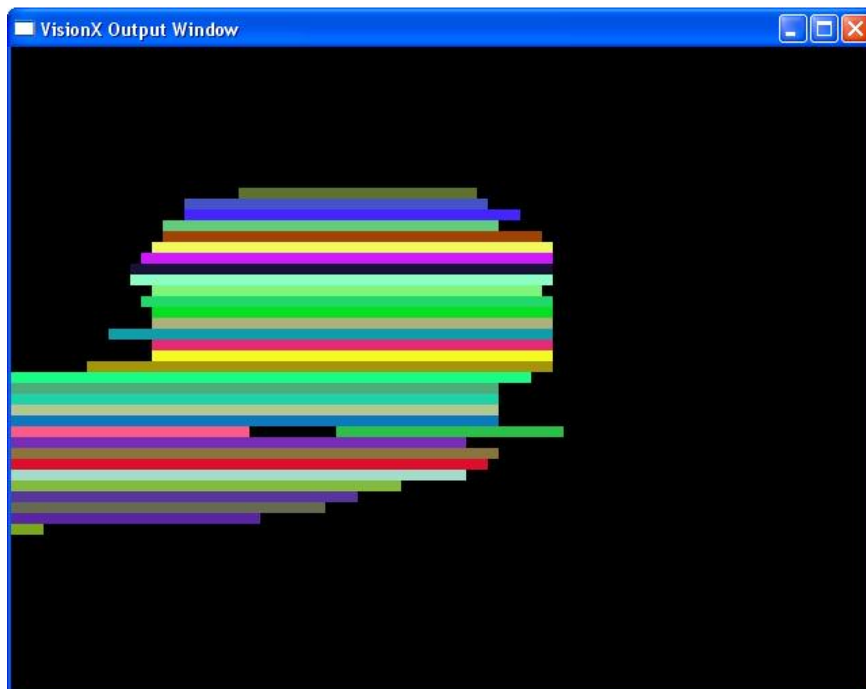


Figure 7.4: A better striping algorithm where close horizontal strips are joined together

```
struct ObjectFound  //When object found - its bounding box is stored withi
{
unsigned int x1;  //x1 and y1 are top left hand corner of bounding box
unsigned int y1;  //x2 and y2 are therefore bottom right corner of box
unsigned int x2;
unsigned int y2;

int StripIndexes[500];
static const int MaxStripsPerObject = 499;


int NumberOfStrips;    //The total number of strips that makes up the obj

};
```

Figure 7.5: The 'ObjectFound' Structure which contains details of a low level object

Chapter 8

Object Tracking

8.1 An Overview Of The Object Tracking Engine

The main program code responsible for the logistics of object tracking is contained in the class 'ObjectTracker' which is defined in the source header file 'ObjectTracker.h' listed in the appendices. The main functions of the code are: To maintain a list of the currently tracked objects, to maintain their internal information such as bounding box coordinates and identification data and keep these updated, and also to determine which objects no longer need to be tracked and can therefore be removed from the list. The only section of code that stands out from these roles is the motion filter which smooths and predicts the paths of tracked objects with the aid of raw samples from its previous positions.

At the heart of the object tracking class is a list of currently tracked objects. This list is stored as an array of structures with 1 structure for each object. The structure used to hold the object information is the 'ObjectFound' structure defined in chapter 7. There are other arrays that complement this structure and hold additional information relating to each object such as the number of times the object is seen and additional tracking information. The additional information arrays are of the same size as the object list array and the information at each index corresponds to the object defined at that same index in the object list array. The following sections in this chapter deal

with the specifics of how the previously mentioned data is maintained.

8.2 Addition and Removal of Objects From the Object List

At the conclusion of the low level operations performed by the 'Striping' class there should exist an array of ObjectFound structures filled with the information of all the objects found by that class within it. This information is then passed on to the 'Object-Tracker' class for higher level processing and some will find their way into the Object list array in the Object Tracker class and be formally recognized as tracked objects.

The first operation to be performed by the Object Tracking class is to identify which of the objects passed to it that are not currently tracked and may need to be tracked. It does this by comparing the regions of the objects passed to it by the Striper class to its own internal object list array and considers those objects with regions that do not overlap any existing tracked objects as new objects of interest. The Object Tracker class uses a function called FindSimilarObjects(...) to search its own internal object list and retrieve any objects overlapping the object passed to the function. The basic criteria for two bounding boxes to overlap are: That lines from either box intersect, or that any of the points from one bounding box lie within the other box.

On determining that a particular potential object is not similar to any existing objects within the ObjectTracker class it can then be added to the internal object list array using the AddObject(...) function. To keep track of all the objects that have been allocated an array of bytes called 'PObjectAllocated[]' is used. When a particular index in the object list array is filled with valid data of a trackable object the corresponding index in PObjectAllocated[] is set to 1 and when an object is to be removed it is set to 0. The main code for the AddObject(...) function is given in Fig]reffig:AddObject. The logic of the function is to loop through the PObjectPresent[] array until a free slot is found and then allocate it and copy the object data into the structure for subsequent tracking.

Figure 8.1: Bounding boxes around objects of interest

Removal of objects from the object list is done when they are no longer seen in the image for a set number of consecutive frame grabs. Each time an object is not recognised in a consecutive frame of the image stream a value is incremented at its index in an array of bytes called 'DeadTimer[]'. When this variable is incremented past the threshold the object becomes marked for removal from the active object list. The object can now be easily removed by setting the corresponding index in the 'PObjectPresent[]' array to 0 and zeroing its internal values to zero so that memory space can be used by the next available object.

8.3 Object Position Sampling

When an object from the Striper class is recognized in the ObjectTracker class in a new location its position must be updated. The ObjectTracker class also remembers the previous positions of each object it is tracking. This can be used to analyze the motion of the object or determine if it is moving at all and for how long. For each Object in the object list array there is another array of structures called 'MSamples' derived from a type Movement structure which holds this sampling information. This structure is

outline in Fig 8.3. There are entries to store the X,Y,Width and Height of the bounding box of each tracked object at each time sample. A circular buffer is implemented when manipulating each sample array so that 400 samples max are stored. Assuming a frame rate of 25 frames a second that will store 16 seconds of previous movement history for each object.

8.4 Movement Filtering and Position Prediction

The object position samples that are received are subject to a noise parameter that gives them some uncertainty; therefore, a means is needed to filter noise samples and smooth the signal so that better estimates of velocity and acceleration can be made. The motion filtering system revolves around sample averaging. To obtain estimates of the velocity and acceleration the last 5 or so samples of each variable are averaged to obtain a smoothed value. Provided the frame rate is high enough this system can provide noise filtered predictions for velocity and acceleration. It is not ideal to filter the position estimates but the filtered value of the difference between the predicted position values and the sampled position values is useful for position error correction. The averaging filter can be designed so that it can update itself with each new value rather than having to calculate the entire average in a similar way to the Kalman filter and is evaluated below.

The average of the last samples of a scalar quanity can be written as:

$$x_k = \frac{1}{k-a} \sum_{n=a}^{n=k} V_n \tag{8.1}$$

Where k is the current sample number and a is the first sample. The average with 1 new value added and the last old value discarded becomes:

$$x_{k+1} = \frac{1}{k-a} \sum_{n=a+1}^{n=k+1} V_n \tag{8.2}$$

To find an update equation we need the difference between V_k and V_{k+1}. This is:

$$
\begin{aligned}
x_{k+1} - x_k &= \frac{1}{k-a}\sum_{n=a+1}^{n=k+1} V_n - \frac{1}{k-a}\sum_{n=a}^{n=k} V_n \\
x_{k+1} &= x_k + \frac{v_{k+1} - v_a}{(k-a)}
\end{aligned}
$$

This now allows for the calculation of the running average but the major problem of needing to store all the samples remains. This problem is solved by replacing the last sample term v_a with the current average x_k and creating a pseudo average that will be smoother than the original average function. The running average then transforms into:

$$
\begin{aligned}
x_{k+1} &= x_k + \frac{v_{k+1} - x_k}{k-a} \\
x_{k+1} &= \frac{(RunningSum) + v_{k+1} - (x_k)}{NumberOfSamples}
\end{aligned}
$$

Generally speaking the number of samples term used in the running average should be small enough to update rapidly with changes in the variable but yet large enough to filter out some noise occurring in the sampling. Due to the nature of the running average it does have some lag, i.e. when the variable changes the running average lags behind these changes as the running sum changes much more slowly than the variable itself. To compensate for this the filtered value of the rate of change of the variable is needed to assist in predicting the actual value of the variable. As a general rule in order to make good predictions using this system the highest order of the derivative that should be sampled and filtered should be at least equal to the highest order of motion in the variable itself. In other words if modeling position and velocity as in the VisionX system and the acceleration only ever changes slowly than the velocity and acceleration would need to be sampled and filtered to make good predictions. Since the highest order of motion or acceleration only changes slowly the running average of acceleration should tail very closely to the real value and this enables smooth prediction of motion.

8.5 Mean Shift Tracking

At the heart of the tracking engine is a simple mean shift algorithm. This algorithm allows for fairly robust tracking of objects in the background subtracted image once they have been initially detected by lower level graphics routines. The algorithm can be thought of as a centre of mass calculation where the bounding rectangle defines the limits of the object and each pixel within it is an element of 'mass' of the object with a defined weighting. As the pixels drift in a given direction the centre of mass of the object should also shift in the same direction. The basic mean shift system moves the last centre x and y coordinates by a small amount each time it is run. The algorithm is expressed mathematically in equation 8.3 and 8.4 for 'n' tracking pixels.

$$CentreX_{NEW} = CentreX_{OLD} + \frac{\sum_{i=0}^{n} x_i P(x_i)}{n} \tag{8.3}$$

$$CentreY_{NEW} = CentreY_{OLD} + \frac{\sum_{i=0}^{n} y_i P(y_i)}{n} \tag{8.4}$$

The function 'P' in equation 8.3 and 8.4 is the probability that the referenced pixel is likely to be part of the tracked object. Various systems can be used for the implementation of this probability function such as using a kernel. In the VisionX system this probability function is ignored and more emphasis is placed on getting the boundaries of the bounding box in the right position so that the mean shift works adequately without the added complexity of the probability function.

8.6 Resizing Bounding Boxes

As objects move across the screen there may be changes in orientation and distance to the viewer, therefore, the bounding boxes will need to be constantly scaled. The initial method employed to achieve this goal used a standard deviation measurement of the distribution of the pixels within the bounding box in the x and y directions. Once the standard deviation/variance of the pixels is found in a given direction the length

in that direction is set to between 4 and 5 times this value. As the bounding box fills up with pixels when the object becomes larger the standard deviation of the pixels increases and this in turn will lead to increases in width and height with this algorithm. This algorithm worked fairly well when the bounding boxes are axis oriented but other methods were used when oriented bounding boxes were introduced and are detailed in later chapters. Equation 8.5 is the mathematical representation of this algorithm for the x direction. Each x coordinate is relative to the centre of the current object the algorithm is operating on and there are 'N' tracking pixels. The same operation is performed in the y direction, this allows scaling in width and height. The drawback to this function its heavy computational burden. This is due to the squaring operation being performed on each and every pixel within the bounding box. Later additions to VisionX utilize functions that rely mainly on summation which helps to eliminate the processing burden of the squaring operations.

$$\sigma_x = \sqrt{\frac{1}{N}\sum_{i=0}^{N} x_i^2} \tag{8.5}$$

```
void ObjectTracker::AddObject(ObjectFound* ObjectToAdd)
{


  //Find a free slot in the object array
  for (int i=0; i<MaxTrackableObjects; i++)
  {
    if (PObjectsPresent[i] == 0) //Have we found a free slot
        {
    //yes we have
        PObjectsPresent[i] = 1; //Mark as occupied

        PreliminaryObjectsFound++; //Record how many objects have been fou

        //Copy the new object data into the free slot
    memcpy((void*)&PreliminaryObjects[i],(void*)ObjectToAdd,
            sizeof(ObjectFound));



        return;
        }


  }

//If here no space found so report error
}
```

Figure 8.2: The 'AddObject(..)' Function

```
struct Movement //Records the movements of an object
{
int XSamples[400];
int YSamples[400];
int Width[400];
int Height[400];

BYTE ValidEnries[400]; //if 1 this entry is a valid sample 0 if not.
int SampleIndex; //Current Sample Index.
When it gets to 399 resets to zero. Thus a circular buffer is implementec

};
```

Figure 8.3: The structure of one element of the MSamples Array

Chapter 9

Late Additions

9.1 Late Project Additions

The following chapters detail late project additions which were not intended to be incorporated into the initial design.

9.2 Oriented Bounding Boxes

The object tracking engine was initially designed to use axis oriented bounding boxes to highlight the area of the screen occupied by a vehicle. The bounding boxes were therefore composed of vertical lines. During the later stages of the project and after reviewing other similar systems such as that outlined in (Veeraraghavan, Masoud & Papanikolopoulos 2003) it was decided to assess the performance of directionally oriented bounding boxes. These boxes are similar to the axes oriented bounding boxes but are defined by an additional parameter which is an angle defining the degree of rotation of the box around its own centre. The angle parameter should orientate the bounding box along an axis parallel to the direction of motion of the tracked object. The bounding boxes should then have a better fit around the contour of the vehicle.

9.3 Drawing Oriented Bounding Boxes

Drawing the bounding boxes on screen for visual confirmation of tracking is a fairly straightforward process. As the bounding box is defined by 4 points all that is needed is to draw lines between these points. During the coding of these routines there was some difficulty in getting the standard line drawing function from the Windows graphics library GDI+ to draw lines in an off screen buffer so I decided to implement a custom line drawing function.

The code for the line drawing function was copied from the Bresenham algorithm given by (Lampton 1993) pp. 189-190.

9.4 Determining Bounding Box Orientation

The CalcAngles(..) function of the ObjectTracker class calculates the directional heading or angle of each tracked object whether validated or not. The CalcAngles(..) function performs this operation on each iteration of the main engine loop which can be found in the ObjectTracker function UpdateObjects(...). The logic behind determining the angle of motion is essentially very simple as the information needed is already available in the position samples of each object. These samples as mentioned in previous chapters are stored in the MSamples[] array of the ObjectTracker class.

The initial approach taken to determine the angle of the objects motion was to use the last point sampled from the objects position and the current location coordinates and determine the angle between them. This approach proved to be ineffective as quite often the difference between samples was such that an accurate value for a gradient could not be determined. This becomes obvious when we consider a situation where the difference in position is only 2 or 3 pixels on the x or y axis. The number of possible angles with this number of pixels narrows down usually to 3 options: The object is determined to be going up or down, left or right or at a 45 Degree angle. This lack of angular resolution makes it difficult to determine the objects true movement.

The first method considered to address the lack of angular resolution was to use more

samples from the position sample list describing the motion of the object. The question then becomes how far does one go back in time to determine an accurate direction of motion? Given this lingering problem another solution was developed and is outlined below.

The method implemented relies on vector addition. The current angle the object is moving in is described by a vector of magnitude 1. Its magnitude within the code itself is 1 but its actual size is only important in relation to a second vector which describes the change in direction. This second vector is the angle determined by using the previous sample position and the current position. Even though the angular resolution is low it can still be used if it is considered to be a rate of change of angle. To determine the new direction of motion the new rate of change vector is added onto the existing vector of magnitude 1. The new vector must have a smaller magnitude compared to the current angle vector. When the two are added the current angle vector will change slightly towards the new direction. This process is illustrated in Fig 9.1. As vehicle directions only change slowly and smoothly this method is suitable. When the algorithm is continuously repeated and the smaller rate of change vectors are added onto the existing vector at each time instant the true direction of motion will eventually be revealed. It is important to note that the initial angle vector must be set roughly in the right direction otherwise there may be some delay before the direction is corrected. This problem was addressed by giving the change in direction vectors more weighting at the start of the tracking process so the correct direction is set quickly.

9.5 Applying Mean Shift to Oriented Bounding Boxes

The mean shift algorithm as it was applied to the axis oriented bounding boxes will not be as effective if applied directly to an oriented box as the algorithm is unable to determine which pixels are within the area of the box. If the pixels within the bounding box are determined then the mean shift algorithm can be applied the same as before with the centre of the oriented bounding box the origin of the coordinate system. A technique to determine these pixels was developed and is based around 3D graphics techniques or more precisely the Bresenham algorithm and polygon filling routines.

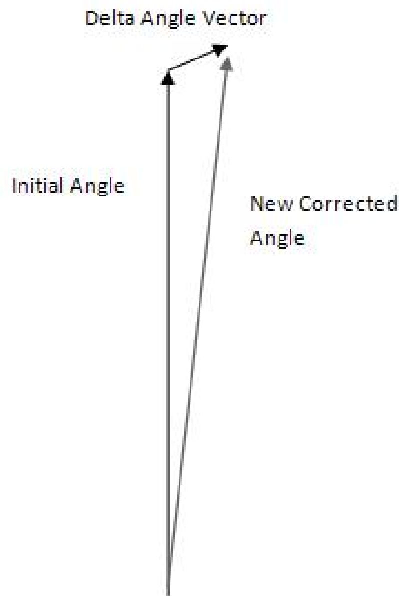


Figure 9.1: Vector addition to achieve directional update

The inspiration for this technique comes from chapter 8 of (Lampton 1993) - 'Polygon-Fill Graphics.' The polygon fill routines implemented are significantly different in terms of the way they are coded but the overall logic is the same. The main code for this algorithm exists in the AdvancedMeanShift(..) function of the ObjectTracker class which is listed in full in the appendices.

The main logic of the function is to determine every point along the edges of the bounding box and then join the points on the same y coordinate with a horizontal line which is fast to draw. The points along the edges of the box can be found by using the Bresenham line drawing algorithm. The Bresenham algorithm will not draw points on the screen but store the points in an array for later analysis. A special function was written to record all the unique vertical points along a line in this manner. This function is a member of the ObjectTracker class and is called ReturnVerticalPoints(..). Apart from the obvious coordinates passed to the function that define the start and end point of the line there are also two array arguments passed as pointers to the function called XPoints and YPoints. These arrays store each point of the line. Each element of the two arrays corresponds to a y screen coordinate. The 0th element is therefore the first row on screen. If the line has a point on this row the YPoints[0] element will be

set to 1 otherwise 0. The corresponding element in the XPoints array will be set to the X coordinate of that point on the row. Consecutive points are stored in this manner until one of the ends of the line is reached.

The AdvancedMeanShift function uses the ReturnVerticalPoints function to fill 4 pairs of XPoint and YPoint arrays with points. Each XPoint and YPoint array holds the points for 1 of the 4 lines consisting of the bounding box perimeter. After the arrays have been filled with points the highest y coordinate and the lowest y coordinate of the bounding box can then be found. This will indicate the rows where the object exists and only these rows will need lines drawn on them. The AdvancedMeanShift function then loops through these rows and looks in the respective elements of the XPoints and YPoints arrays to find the starting and ending point of the next horizontal line within the perimeter of the bounding box. Once the start point and end point have been found the points along this line can be processed as part of any algorithm that needs them, in this case the mean shift. The coordinates of each pixel relative to the centre point of the object can easily be determined and used in the mean shift.

9.6 Resizing Oriented Bounding Boxes

The previously developed algorithm for resizing axis oriented bounding boxes will not apply well on oriented bounding boxes. Another approach was tried and it also utilizes similar techniques to the AdvancedMeanShift to determine the pixels contained within the bounding box. This new technique revolves around analyzing the ratio of trackable pixels to blank space within the bounding box and tries to maximize the number of trackable pixels up to a certain limit wherever possible.

The resizing is implemented in the function ScaleObject(..) and it is also a member of the ObjectTracker class. This function utilizes repeated calls to the function ReturnPixelRatio which returns as its name implies the ratio of the trackable pixels to background pixels within the bounding box of an object. The main algorithm then increases and decrease the width and height. When doing so it determines the new coordinates of the bounding box and the new ratio and compares it to the old. If there

is an increase in the ratio then this scaling operation is in the right direction as it is including more trackable pixels. The algorithm seeks a balance of approximately 30 blank pixels to 70 trackable pixels.

9.7 Vehicle Counting And Speed Estimation

The main objectives of the software are to be able to give estimates of a vehicles speed and also to count the number of vehicles passing a point in the image. Both of these operations are closely related and use similar algorithms to achieve the desired results. The VisionX system uses tracking lines placed by the user at regions where speed estimation or counting is to occur. These lines are used to define a region where the program will look to see if a vehicle has crossed. When a vehicle does cross one of these lines it is recorded in a boolean variable within the ObjectFound structure of that object. To estimate speed two tracking lines need to be set so that the time taken to travel between the two can be determined.

On each iteration of the main program loop a function called CountVehicles(..) a member of the ObjectTracker class is invoked to update the counter and speed estimation data. The role of this function is to determine when a vehicle comes close enough to the counter and speed lines to be considered as crossing it. In order to determine when a vehicle is close enough and likely to cross the line a simple cartesian distance algorithm is used as given in figure 9.1 where x1,y1 and x2,y2 are the coordinates of the centre of the vehicle and one endpoint of the tracking line. The main algorithm calculates the distance from each of the end points of the counter line to the centre of all of the currently tracked objects. When the sum of the distances from the end points to the current object is very close to the length of the counter line the object is considered to have crossed the line and the time of crossing is recorded to the nearest millisecond. The process is illustrated in figure 9.2. When a vehicle crosses both a counter line and speed line a speed estimate can be derived by dividing the physical distance between the lines by the difference of the times of crossing of the two lines. The physical distance between the lines needs to be measured in the real world by the operator and then specified in the software. Figure 9.3 shows a screen grab of speed

estimation in practice. The number at the top left of the bounding boxes is the object identification number and the number to the right of that is the speed estimate. The car at the bottom of the screen shot is me in my car doing a set speed of 80km/h.

$$Distance = \sqrt{(y_2 - y_1)^2 + (x_2 - x_1)^2} \tag{9.1}$$

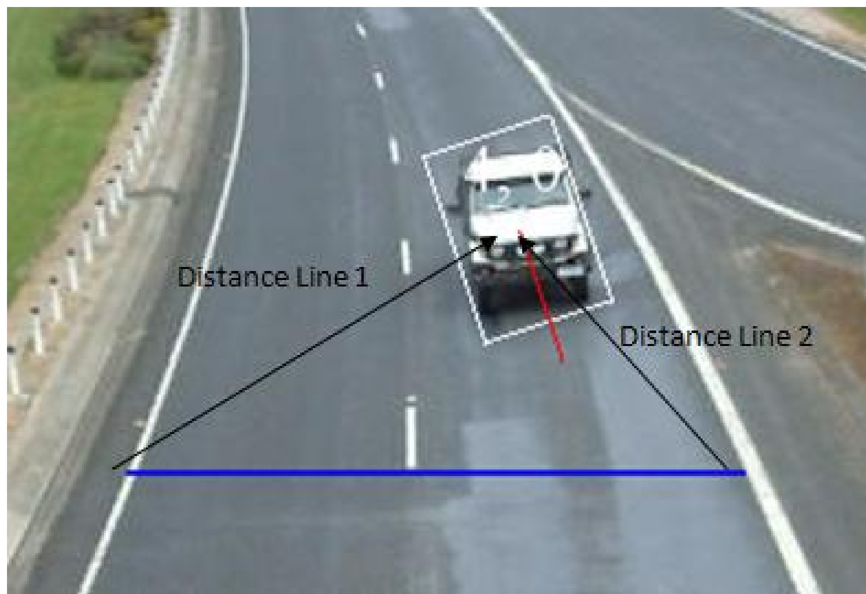


Figure 9.2: Tracking Line Proximity Calculations

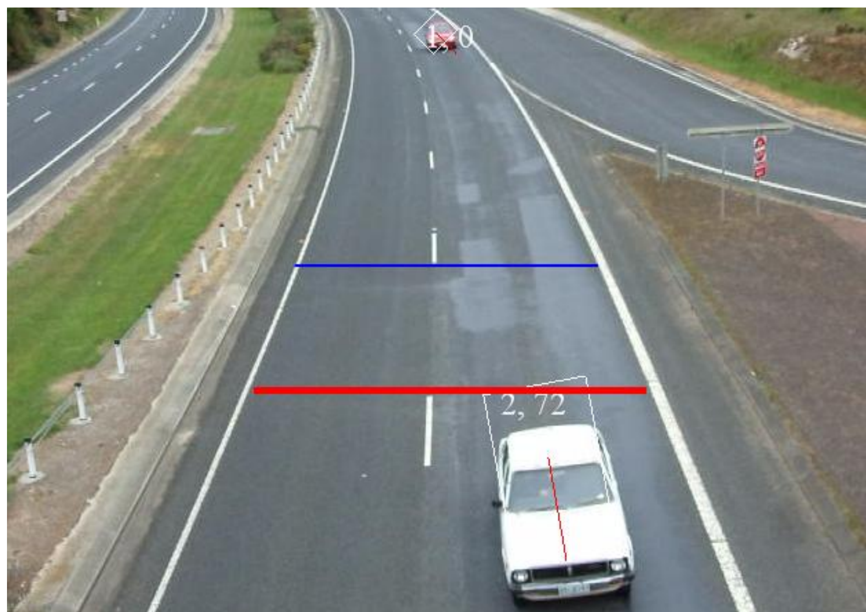


Figure 9.3: Speed Estimation

Chapter 10

Conclusions

The speed estimation and counting abilities of the program were tested in 6 different camera locations and angles. The speed estimation is not accurate and as its name implies is only really useful to obtain a rough estimate of the speeds of vehicles past a point. The accuracy is of the order of +-13km/h depending on how the camera is setup and the amount of perspective in the scene. This may still be useful for traffic control in major cities as this information could be sent back from the roadside to a central processing facility to analyze traffic congestion around a city. The major problem with the speed estimation is change in size of vehicles as they head through high perspective, i.e. they go through large changes in depth or the z dimension in the scene. The tracking software may not be able to consistently place the bounding box accurately around the vehicle as these changes occur, this results in the centre of the bounding box not representing the true centre and throwing off the speed estimation. The challenge is to bring the speed estimation tracking lines close enough together to prevent changes in perspective from affecting the result but yet far enough apart to give an accurate reading. To be practical as an accurate tool the speed estimation system needs a more robust tracking environment than that provided by the VisionX system. Vehicle counting is relatively accurate with accuracies of above 90 per cent easily attainable. The counting feature of the program is the most useful and accurate but still has problems.

The problems of both the speed estimation and counting boil down to glitches in the

tracking algorithms. There are a variety of reasons for these glitches the two major ones are object occlusion and shadows. A shadow suppression system was intended to be included in the VisionX system derived from the work of others but due to time constraints could not be added. With the addition of the shadow suppression the improvement in tracking accuracy and counting would improve substantially and any commercial system should implement one.

The problems presented by object occlusion are formidable and may require the implementation of artificial intelligence such as occlusion reasoning and belief networks to solve them adequately. One of the biggest problems occurs when two vehicles enter a scene but appear as one due to occlusion, they then separate into two vehicles as in figure 10.1. The problems in the image came about because of occlusions at the top of the image as the vehicles entered the scene. The VisionX system has been updated to separate vehicles when they get as far as indicated in figure 10.1 but the screen shot serves to illustrate the problem. The VisionX system does not implement any occlusion reasoning and does experience problems when the vehicle footage passed to it has many occlusions.

There were plans to develop mechanism to deal with object occlusion but due to time constraints were not implemented. One approach intended to be applied was to learn the paths of the vehicles along the road and use this information to assist in determining there likely location even when occlusions occur. I would suggest this to be a useful area of work if the project was to be continued.

Overall I believe the program does achieve the project goals of vehicle counting and speed estimation using real time tracking but there remain many areas for improvement. The program as it is would not be a fully practical system but with some additions such as shadow suppression and path learning it would definitely be useable as a tool and provide useful information on traffic parameters.

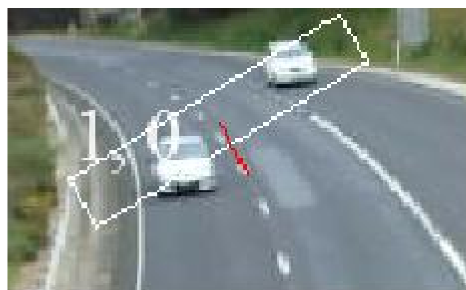


Figure 10.1: Two vehicles recognized as one due to a previous occlusion

Bibliography

Beymer, D., McLauchlan, P., Coifman, B. & Malik, J. (1997), 'A real time computer vision system for measuring traffic parameters', *IEEE Conference on Computer Vision and Pattern Analysis* pp. 495–501.

Davies, E. (2005), *Machine Vision Theory Algorithms Practicalities*, third edn, Elsevier, chapter 18, pp. 505–517.

Horn, B. K. & Schunck, B. G. (1981), 'Determining optical flow', *Aritificial Intelligence* **17**, 185–203.

Kass, M., Witkin, A. & Terzopoulos., D. (1988), 'Snakes: Active contour models', *International Journal of Computer Vision* pp. 321–331.

Koller, D., Daniilidis, K. & Nagel, H. (1993), 'Model based object tracking in monocular image sequences of road traffic scenes', *International Journal of Computer Vision* **10**, 257–281.

Koller, D., Weber, J. & Malik, J. (1994), 'Robust multiple car tracking with occlusion reasoning', *European Conference on Computer Vision* .

Lampton, C. (1993), *Flights of Fantasy Programming 3-D Video Games in C++*, Waite Group Press.

Michalopoulos, P. G. (1991), 'Vehicle detection video through image processing: The autoscope system', *IEEE Transactions on Vehicular Technology* pp. 21–29.

Rabie, T., Abdulhai, B., Shalaby, A. & El-Rabbany, A. (2005), 'Mobile active-vision traffic surveillance system for urban networks', *Computer-Aided Civil and Infrastructure Engineering* **20**, 231–241.

Veeraraghavan, H., Masoud, O. & Papanikolopoulos, N. (2003), 'Computer vision algorithms for intersection monitoring', *IEEE Transactions on Intelligent Transportation Systems* **4**.

Appendix A

Project Specification

ENG4111/4112 Research Project
PROJECT SPECIFICATION

FOR: **Paul Menetrier**

TOPIC: **Machine Vision Based Traffic Monitoring with Real Time Tracking**

SUPERVISOR: John Billingsley

SPONSORHSIP: None

PROJECT AIM: The aim of this project is to produce software on a Windows platform to perform various road traffic monitoring functions in real time when given a colour video stream. The said functions should comprise the following: vehicle counting and speed estimation with the aid of user calibration.

PROGRAMME: **(Issue A, 20th April 2008)**

1. Research basic Machine Vision techniques to gain grounding in the field
2. Write the main body of the program in C/C++ with Microsoft Visual Studio and Microsoft DirectShow. This code will create the main look and feel of the program and provide functions to control the video stream and other basic video operations.
3. Research existing techniques used in real time traffic monitoring
4. Implement Edge detection routines, background subtraction and Optical Flow and choose the best approach or a combination of each for the real time tracking of the vehicles
5. Develop tracking algorithms to monitor vehicle location in real time and count vehicles passing a particular location in the image
6. Implement speed estimation from the real time tracking data

Appendix B

Source Code

B.1 BackgroundSubtraction.h

```
class BackgroundSubtraction
{

public:

BOOL IsBaseImageInitialised;

BYTE* pBaseImage; /*The background image */
BYTE* pSDerivative; /* Background subtraction with second
    derviative type approach*/
BYTE* pMask;      //Each byte corresponds to a pixel in the
    pBaseImage a 1 == don't update pixel 0 == update pixel
                                   //in next background
    subtraction routine

BYTE* pResult;    /*The result of subtracting an image from
    base image with DoSubtraction() function */

int BufferXLength;
int BufferYLength;
int Threshold;  /* The difference between pixels considered
    big enough to be noticed */

int FramesBetweenBackgroundUpdates; //Number of frames to skip
     before updating background. Initialised in Constructor

int FramesProcessedSinceUpdate; /*Number of frames processed
    since background was updated */

BackgroundSubtraction(int BufferXLength,int BufferYLength,int
    Threshold); /*Buffer XLength & YLength are the length and
height of the buffers being used with 32-bit colour pixels */

void InitBaseImage(BYTE* pBaseData);
```

```
void DoSubtraction(BYTE* pNewImageData);
void DrawImage(BYTE* pDest); /*Draw any pixels of difference
    onto buffer */

void DrawDerivativePixels(BYTE* pDest); //Draws only the
    changes in the background subtraction image from frame to
    frame into this buffer

void DrawMaskRegions(BYTE* pBuffer); //Draws the regions where
     the mask bits are currently preventing

    background update


~BackgroundSubtraction();



};
```

B.2 ObjectTracker.h

```
#include "striper.h"
#include <gdiplus.h>

using namespace Gdiplus;



struct Movement //Records the movements of an object
{
static const int MaxSamples = 800;
int CentreXSamples[MaxSamples];
int CentreYSamples[MaxSamples];
int Width[MaxSamples];
int Height[MaxSamples];
double AngleSamples[MaxSamples];

int SamplesTaken;


BYTE ValidEnries[MaxSamples]; //if 1 this entry is a valid
    sample 0 if not.
int SampleIndex; //Current Sample Index.  When it gets to 399
    resets to zero.  Thus a circular buffer is implemented

//When an object is removed from the preliminary objects list
    the Valid entries structure must be reset to 0 therefore
//if another object is added and it occupies the sample memory
     it doesn't inherit any data left behind
};


struct Tracks //Stores all info relating to the tracks on the
    screen
{
const static int MaxTracks = 10;  //The maximum number of
    individual tracks allowed at one time
```

```
int XSamples[Movement::MaxSamples];
int YSamples[Movement::MaxSamples];
int SamplesInTrack;

};


struct HistData //Holds the important data used when histogram
    tracking an object in 1 structure
{
int X1;
int Y1;            //First histogram box within bounding box
   of object
int Width1;
int Height1;

int X2;
int Y2;            //First histogram box within bounding box
   of object
int Width2;
int Height2;

int X3;
int Y3;            //First histogram box within bounding box
   of object
int Width3;
int Height3;

int X4;
int Y4;            //First histogram box within bounding box
   of object
int Width4;
int Height4;


int Red1[256];
int Green1[256];   //The histograms used for tracking the
   object
int Blue1[256];

int Red2[256];
int Green2[256];   //The histograms used for tracking the
   object
int Blue2[256];

int Red3[256];
int Green3[256];   //The histograms used for tracking the
   object
int Blue3[256];

int Red4[256];
int Green4[256];   //The histograms used for tracking the
   object
int Blue4[256];

};


//Main class containing all object tracking code
class ObjectTracker
{
```

```
private:

int MaxTrackableObjects;
int VideoXLength;
int VideoYLength;

public:

BYTE* pSubBackground;
BYTE* pCurrentFrame;
BYTE* pBackground;      //Pointers to various important buffers

int* TimesSeen;                 //The number of times a particular
    area of the screen has been highlighted as an object found
    by the striper

int* DeadTimer;              //Counts the number of times the
    object has not been seen. If above a certain level it will
    be removed from the tracker

Movement* MSamples;  //Saves the X,Y location of top left hand
    corner of object and records the movements of the object
Tracks* TracksData;

ObjectFound* PreliminaryObjects;
BYTE* PObjectsPresent; //If an entry is equal to 1 it means
    the corresponding entry in the PreliminaryObjects array is
    occupied

int* TimeAlive;                 //For every
    PreliminaryObjectsFound entry describes how long this entry
    has been stored for

    if any potential objects are there for too long without
    moving they are removed from tracking

BYTE* Validated;  //contains a 1 if the corresponding
    PreliminaryObject has been verified


int PreliminaryObjectsFound;


void AddObject(ObjectFound* ObjectToAdd);       //Add this new
    object to the PreliminaryObjects arrray
void RemoveObject(int ObjectIndex,BYTE* pBackground,BYTE*
    pCurrentFrame); //Removes an object from the tracking
    classes arrays


int FindSimilarObject1(ObjectFound* PObject); //Compare this
    object to others in the PreliminaryObjects array and find
    similar one

    based on bounding window size and the proximity of the
    corners of the bounding box

void UpdateObjects(int NumberOfUpdateObjects,ObjectFound*
    Objects);
```

```
ObjectTracker(int MaxTrackableObjects,int VideoXLength, int
    VideoYLength,BYTE* pBackground,BYTE* pSubBackground,BYTE*
    pCurrentFrame);
void SetBuffers(BYTE* pBackground,BYTE* pSubBackground,BYTE*
    pCurrentFrame);

void AddSample(int ObjectIndex,double Angle,int CentreX,int
    CentreY,int Width, int Height); //Add a position sample to
    this objects movement list

void UpdateObjectMovements(int ObjectIndex);

void UpdateEstimate(TrackingData* TData,double NewValue); //
    Updates the estimate in a given TrackingData Structure


int  IsObjectStill(int ObjectIndex,int Validated); //If object
     is not moving returns roughly how long it has been still
    for

void DrawBoundingSquares(BYTE* pBuffer);

void DrawObjectNumbers(Graphics* pGraphics); //Draws the
    object numbers in the top right hand corner

void CreateMask(BYTE* pMask); //Creates a mask with 1 where
    any object is positioned so the background in those areas
    is not updated

    by the background model in BackgroundSubtraction class



//General algorithm to draw lines in the given video buffer
void ObjectTracker::DrawLine(BYTE* pBuffer,int X1,int Y1,int
    X2,int Y2,BYTE Red,BYTE Green,BYTE Blue);


//
    **********************************************************************
//Histogram tracking section - Al Deprecated!!!!!
//
    **********************************************************************


void CreateHistogram(int ObjectIndex,BYTE* pSubBackground,BYTE
    * pCurrentFrame,int* Red,int* Green,int* Blue,int X,int Y,
    int Width, int Height);


//Locate the new obect position using colour histograms and
    with the aid of the objects previous frame velocity
//store the new bounding box position in the X,Y,Width.Height
    variables
void LocateNewObjectPosition(int ObjectIndex,BYTE*
    pSubBackround,BYTE* pCurrentFrame);


//See if moving right bounding box line towards the center
    makes histograms look more like original
//+1 if yes -1 if not
```

```
int HistogramDeltaLeft(int ObjectIndex,BYTE* pSubBackground,
   BYTE* pCurrentFrame,HistData* Data,HistData* DelData);
int HistogramDeltaRight(int ObjectIndex,BYTE* pSubBackground,
   BYTE* pCurrentFrame,HistData* Data,HistData* DelData);
int HistogramDeltaUp(int ObjectIndex,BYTE* pSubBackground,BYTE
   * pCurrentFrame,HistData* Data,HistData* DelData);
int HistogramDeltaDown(int ObjectIndex,BYTE* pSubBackground,
   BYTE* pCurrentFrame,HistData* Data,HistData* DelData);

int CalculateImprovement(int  ObjectIndex,HistData* Data,
   HistData* DelData);
void UpdateObjectHistograms(int ObjectIndex);
void AddDelData(HistData* Data,HistData* DelData);
//
   *******************************************************************


//
   *******************************************************************
// 2D Orientation Tracking Routines.  These functions allows
   the tracking boxes to rotate when tracking objects
//
   *******************************************************************

void CalcTranslatedCoordinates(); //Goes through all the
   objects and rotates all the X,Y,Width and Height
                                                                    //
   parameters around the axis of the bounding box by the Angle
    parameter defined
                                                                    //
   within each ObjectFound structure.

void CalcTranslatedCoordinates(int ObjectIndex); //Same as
   above but for single object


void CalcAngles(); //Calculates the current orientation angles
    of all currently tracked objects

//Determines the angle from point X1,Y1 to X2,Y2 where 0
   Degreen means X2,Y2 is directly above X1,Y1
double AngleBetweenPoints(double X1,double Y1,double X2,double
    Y2);


//A Better mean shift that selects pixels for the mean shift
   from within a bounding box that
//maybe rotated to a given angle
void AdvancedMeanShift(int ObjectIndex,BYTE* pSubBackground);


int ReturnVerticalPoints(int X1,int Y1,int X2,int Y2,int*
   XPoints,int* YPoints);


double ReturnPixelRatio(int ObjectIndex,BYTE* pSubBackground);

//This function makes sure the bounding box is big enough or
   small
//enough to fit around the object
void ScaleObject(int ObjectIndex);
```

```
void DrawObjectTracks(BYTE* pBuffer); //Draws all the object
    tracks

//
    *************************************************************************



//
    *************************************************************************
//Velocity estimation routines
//
    *************************************************************************

//The two defining screen coordinate points of the counter
    line
static const int CounterX1 = 180,CounterY1=310,CounterX2=465,
    CounterY2=310;
static const int SpeedX1 = 210,SpeedY1 = 220,SpeedX2=430,
    SpeedY2=220;
static const int CounterToSpeedLineDistance = 9.2; //9.2m in
    between the two
int VehiclesCounted;

void DrawCounter(Graphics* GObject); //Draws the counter

void CountVehicles(); //Checks if any vehicles have crossed
    near or over the line


//
    *************************************************************************


FontFamily* pFontFamily;
Font* pFont;
PointF* pPointF;
SolidBrush* pSolidBrush;
Pen* pPen1;
Pen* pPen2;




~ObjectTracker();


};
```

B.3 Striper.h

```
#include <windows.h>


/* The following structure holds the positions of the possible
```

```
    objects on each pixel row of the image */
/* It is used in the Striper class
                                                                          */
struct PossibleObject
{
int xstart; /* Start of the detected object on the current
   pixel row */
int xend;   /* End of detected object on the current pixel row
    */
int y;
int MemberNo;  //Memebership number. Indicates what object it
   belongs to. -1 == processed but not part of group
               //0 is unprocessed and ungrouped and a positive
    integer means that it is grouped into a group of
                              //that number
};

struct TrackingData //Data used to track something
{
double Sum;
double CurrentEstimate;
double SamplesAcquired; //Records the number of samples
   required.  Once the number of samples aquired equals the
   ideal

   amount this value no longer increases.  This should be
   around 5 samples

 };


struct ObjectFound  //When object found - its bounding box is
   stored within this structure
{
int X;  //x1 and y1 are top left hand corner of bounding box
   with origin in top left hand corner of screen
int Y;
int Width;
int Height;

double Angle; //The orientation angle of the bounding box

int CentreX; //The screen coordinates of the centre of mass in
    the bounding box also the centre of the bounding box
int CentreY;


double TX1,TX2,TX3,TX4; //The coordinates of the bounding box
   rotated by 'Angle' to fit the object better
double TY1,TY2,TY3,TY4; //These coordinates are updated by the
    void CalcTranslatedCoordinates() in the ObjectTracker
   class.


BOOL IsObjectCounted1; //Has the object been counted by
   counter 1
BOOL IsSpeedLineCrossed;

int CounterTimeStamp; //Time when object passes over counter
   line
```

```
int SpeedTimeStamp; //Time when object passes over the speed
    line

double Speed; //The speed estimate in kph


double PhysicalX;
double PhysicalY;
double PhysicalWidth;
double PhysicalHeight;


//
    ************************************************************************
//Histogram tracking Depracated!!!!!!!
int Red1[256];
int Green1[256];    //The histograms used for tracking the
    object
int Blue1[256];

int Red2[256];
int Green2[256];    //The histograms used for tracking the
    object
int Blue2[256];

int Red3[256];
int Green3[256];    //The histograms used for tracking the
    object
int Blue3[256];

int Red4[256];
int Green4[256];    //The histograms used for tracking the
    object
int Blue4[256];

int TotalTrackingPixels; //Total pixels listed in all the
    individual histograms used for tracking
int TotalPixelsFound;
//
    ************************************************************************


int StripIndexes[500];     //Contains the indexes into Strip
    array PossibleObjects[] that indicates what strips

    are thought to be part of the object.
static const int MaxStripsPerObject = 499; //Must be the same
    or less than the size of StripIndex array.


int NumberOfStrips;     //The total number of strips that makes
     up the object if 0 not a valid object
int IsValidObject;      //Set to 0 if Invalid and 1 if valid
    !!!!Changed ******!!!!!!*******!!!!!!


TrackingData LLPositionError;    //The average error between
    the samples and the predictions
TrackingData LLVelocity;     //the average velocity of the
    samples
TrackingData LLAcceleration;    //the average acceleration of
    the samples
```

```
TrackingData RLPositionError;
TrackingData RLVelocity;
TrackingData RLAcceleration;

TrackingData TLPositionError;
TrackingData TLVelocity;
TrackingData TLAcceleration;

TrackingData BLPositionError;
TrackingData BLVelocity;
TrackingData BLAcceleration;

};




class Striper
{
public:

int VideoXLength;
int VideoYLength;
int XThreshold; /* Number of pixels in a given horizontal
    length before possible object is detected */
int SampleLength;
int MaxXObjects; //Maximum number of objects that can be
    detected in one row scan
int MaxTotalObjects;
int ObjectsDetected; //Records number of objects detected
    after DetectObjects(...) function runs

PossibleObject* PossibleObjects; // array holding detected
    strips
                                                                  //
     array size = maxxobjects*VideoYLength

ObjectFound* ObjectsFound; //Contains the bounding boxes of
    detected objects
                                                        //It is
    filled by DetectObjects() function. Its size is
    MaxTotalObjects

int* PossibleObjectsFound; //Each integer holds the number of
    possible objects on each row.


/* Class constructor. VideoLength variables are the length of
    the video frame in pixels.
    XThreshold is how many difference pixels in one
    SampleLength allowed before object is considered
    within that Sample Length. MaxXObjects Determines the
    maximum number of objects that can be
    detected in one row

           */

Striper(int VideoXLength,int VideoYLength,int XThreshold,int
    SampleLength,int MaxXObjects,int MaxTotalObjects);
```

```
void DetectPossibleObjects(BYTE* pBuffer);

//void StripTidy(); //Called after strip have been gotten with
    DetectPossibleObjects(...) If
                                        //two strips are on same row
    and only 3 pixels apart this function will merge them
                                    //function may be extended to
   fill in gaps in strips vertically etc...


//After finding a strip not allocated to an object this
   function is called to find any other
//strips on succeeding rows that can be considered part of the
    same object
//The function returns 0 if successful and -1 if this strip
   had not enough other strips associated with it
//or for some other reason was deemed not to be part of an
   object
int FindObjectFromStrip(int ObjectNumber,int StripBaseIndex,
   int StripIndex);


//Once objects have been detected by DetectObjects(..)
   function this function is then called
//to fill in the bounding box details of each object in its
   ObjectFound structure.
void GetBoundingSquares(BYTE* pBackground,BYTE* pCurrentFrame)
   ;

//Draw bounding squares of detected objects into the given
   buffer
void DrawBoundingSquares(BYTE* pBuffer);

void DetectObjects(BYTE* pBuffer);

//The spectral filter determines whether an object is actually
    a new trackable object or caused by vibration
//of the camera.  If the camera vibrates the reasoning is that
    the objects detected due to this noise
//will not have significant changes in their spectral or
   colour map over the area of the detected object
//Returns -1 if no change and +1 if a significant change
int ObjectSpectralFilter(int ObjectIndex,BYTE* pBackground,
   BYTE* pCurrentFrame);

BOOL IsInsideOtherObject(int ObjectIndex); //Checks to see
   whether this object is inside another

   if so mark as invalid

//Eliminates objects that are inside the bounding box of other
    objects
void EliminateInternalObjects();

~Striper();



};
```

B.4 VideoCapture.h

```
#include <dshow.h>
#include <qedit.h>



class VideoCapture
{

public:

AM_MEDIA_TYPE* MEDIA_TYPE; //Filled with values by the
    CreateGraph function

IGraphBuilder* pGraph;



ISampleGrabber* pGrabber; //Sample grabber that grabs samples
    of the video image stream for processing
IAMStreamConfig* pStreamConfig; //Pointer to interface for
    determining and setting video config.

IEnumPins* pGrabberPins;
IEnumPins* pRendererPins;
IEnumPins* pAVIFileSourcePins;
IEnumPins* pMJPEGDecompressorPins;

IPin* pAVIFileSourceOutputPin;
IPin* pMJPEGDecompressorInputPin;
IPin* pMJPEGDecompressorOutputPin;
IPin* pGrabberInputPin;
IPin* pGrabberOutputPin;
IPin* pRendererInputPin;

//Filters for MJPEG decompression
IBaseFilter* pAVIFileSourceF;
IBaseFilter* pMJPEGDecompressorF;
IBaseFilter* pGrabberF;



int CurrentModeXLength;  //Length and height of current video
    mode
int CurrentModeYLength;

IMediaControl* MC; //Used to control the graph e.g to control
    flow of data through the graph
IMediaSeeking* MS; //Need this to reset the graph once it is
    finished playing

BYTE* pVideoBuffer; //Initialised when a graphics mode is
    activated, holds the current
                                        //still image grabbed from
     video stream
BYTE* pTempBuffer;  //Used for image processing. Same size as
    pVideoBuffer
```

```
long BufferSize;    //Holds the buffersize required to store
    still image from video stream


BOOL CreateGraph();  //Creates the main video data flow in
    DirectShow
VideoCapture();

~VideoCapture();

//returns the IAMStreamConfig* variable of this class
IAMStreamConfig* GetStreamConfigInterface();


//Grabs a sample of the video stream and store it in the
    buffer pVideoBuffer
BOOL GetVideoSample(BYTE* pSource);

//Grabs a sample and store is the supplied buffer instead of
    storing in the local class
//buffer pVideoBuffer.  This function allows the video stream
    to continuously grab samples
//without interfering with any still captures grabbed by the
    user that are currently stored in
//pVideoBuffer
BOOL GetVideoSampleA(BYTE* pSuppliedBuffer);

};
```

B.5 VisionX.h

```
#include <windows.h>
#include <windowsx.h>
#include <mmsystem.h>
#include <stdio.h>

#include <D3D9.h>
#include <gdiplus.h>

#include "resource.h"

#include "VideoCapture.h"
#include "GSpace.h"
#include "LineDetector.h"
#include "OpticalFlow.h"
#include "BackgroundSubtraction.h"
#include "ObjectTracker.h"

using namespace Gdiplus;


struct GlobalData     //Structure to hold useful app data
{
HWND hWndOutputWindow;  //Output window handle
HWND hWndVideoWindow;  //Handle to window that display sequence
     of captured frames from frame grabber
HINSTANCE hInstance;
HWND hWndControlWindow;  //Control Window Handle
VideoCapture* VCapture;  //Pointer to video capture object
```

```
    initialised in WinMain
Graphics* GOutput; //GDI+ Graphics object associated with the
    output window
Graphics* GVideo; //GDI+ Graphics object associated with the
    video window


Bitmap* BVideo; //Holds the current frame to output to video
    window.
Graphics* GBVideo; //Graphics object associated with the
    BVideo Bitmap

Pen* StandardWhitePen; //Standard white pen for drawing lines
    etc

BOOL GraphicsInitialised;


BYTE* pVideoWindowBitmap; //Memory allocated in WinMain for
    the BVideo Bitmap GDI+ Object
BYTE* pTempBuffer;  //Holds enough room for a temporary full
    size image used for Video Window Frames
BYTE* pSampleBuffer; //Holds enough room for one sample image
    from the Video stream


Bitmap* BOutput; //GDI+ Bitmap holds frame to draw to output
    window
BYTE* pOutputWindowBuffer;


ULONG_PTR           GdiPlusToken;           //Used for GDI+ Init
     and destroy


BOOL IsVideoStreamPlaying; /*Set to true if the stream is
    active*/


//Direct 3D stuff not used yet
IDirect3D9* pDirect3D9;        //Main Direct 3D Object
IDirect3DDevice9* p3DDevice;  //Direct 3D Device Interface
    object

~GlobalData();


};





//Variables relating to the control window
struct ControlDialog
{
public:

HWND hGrabImageButton;
HWND hStopStreamButton;
HWND hPlayStreamButton;
```

```
GlobalData* pGData;


BOOL InitControls();

//Function must be called for structure functions to work
    properly
void SetGlobalData(GlobalData* pData); //Passes pointer to
    GlobalData Structure to access program data

};






//This class controls the selection rectangle in the Output
    Window
class SelectionRectangle
{

public:

int ScreenWidth;
int ScreenHeight;


int X; //current Starting x pixel of rectangle in sample frame
int Y; //current Starting y pixel
int CurrentWidth; //CurrentWidth as left mouse is held down
    mouse and moved
int CurrentHeight; //CurrentHeight as left mouse is held down
    mouse and moved

BOOL IsSelectionWindowActive; //determines whether selection
    window active or not

BOOL HasSelectionBeenMade; //Indicates Whether a selection
    rectangle has been used
//if so the OutputWindow will contain the magnified output of
    the selection data


BYTE* pBuffer; //Holds data selected by the selection
    rectangle
Bitmap* pDisplayData; //Used to write pixels to the screen.
    Initialised when GrabData is called
BYTE* pDummyBuffer; //Pixel data for pDisplay Data Bitmap
    Object. GDI+ needs this array even though
                                        //I thought it was
    supposed to allocate its' own

BYTE* pEnhancedData;  //For Testing 4x4 Alpha Blending
    Enhancement algorithm


//Draw the selection window at coordinates x and y in output
    window using graphics object
void DrawSelectionRectangle(Graphics* G);
```

```
//Initialises variables. Call this function before using class
//SWidth is the ScreenWidth and SHeight is the Screen Height
void Init(int SWidth,int SHeight);

void GrabData(BYTE* Data); //Gets the selected data from the
    framebuffer


void DrawData(BYTE* pSourceBuffer,int DestWidth,BYTE*
    pDestBuffer,int PixelMagnification); /*DrawData draws
the selected data onto a full screen buffer so the pixels
    appear enlarged
Pixel magnification to be used 8 is good for 640x480
DestStride is the width of the Destination buffer in bytes */


//Draws an enlarged pixel in a screen/Bitmap buffer
//Stride is the width of the buffer in bytes
//x and y are big pixel coordinates not small pixel
    coordinates
void DrawBigPixel(int x,int y,BYTE Red,BYTE Green, BYTE Blue,
    BYTE* pDestBuffer,int Stride,int Size);

//void GrabCustomData(); //Used for testing 4x4 Averaging
    Algorithm

~SelectionRectangle();


};



void Image4x4Blender(unsigned int* pDestImage,unsigned int*
    pSourceImage,Rect &SourceImageSize);
```

B.6 VisionX.cpp

```
#define WIN32_EXTRA_LEAN




#include "VisionX.h"





static void CALLBACK TimerFunc(UINT wID,UINT wUser, DWORD
    dwUser, DWORD dw1,DWORD dw2);
BOOL WinInit(HINSTANCE hInstance);
LRESULT CALLBACK OutputWindowProc(HWND hWnd,UINT uMsg, WPARAM
    wParam,LPARAM lParam);
LRESULT CALLBACK VideoWindowProc(HWND hWnd,UINT uMsg, WPARAM
    wParam,LPARAM lParam);
BOOL CALLBACK ControlDialogProc(HWND hwndDlg,UINT uMsg,WPARAM
    wParam,LPARAM lParam);
```

```
GlobalData GData;
ControlDialog CDialog;
SelectionRectangle SRectangle;



//Function to set client area of window to x pixels by y
    pixels
void SetWindowSize(HWND hWnd,int x,int y);



//Function to Initialise Direct3D in main window - not used at
     this stage
BOOL InitDirect3D();

//Function to render current graphics frame to output window
    not video window which displays the video
//stream
void Render();









GlobalData::~GlobalData()
{

if (StandardWhitePen != NULL) delete StandardWhitePen;

if (BOutput!=NULL) delete BOutput; //Delete GDI+ Graphics and
    Bitmap objects
if (GOutput!=NULL) delete GOutput;


if (BVideo!=NULL) delete BVideo; //Delete GDI+ Graphics and
    Bitmap objects
if (GBVideo!=NULL) delete GBVideo;

if (pVideoWindowBitmap !=NULL) delete pVideoWindowBitmap;
if (GVideo!=NULL) delete GVideo;

if (pTempBuffer!=NULL) delete pTempBuffer;
//if (pSampleBuffer!=NULL) free((void*)pSampleBuffer); //Had
    problems with this. This array mey be freed up previously.
if (pOutputWindowBuffer!=NULL) delete pOutputWindowBuffer;


//Release the 3D Device
if (p3DDevice != NULL) p3DDevice->Release();


//Release Direct3D9 Object last of Direct3D Objects
```

```
if (pDirect3D9!=NULL) this->pDirect3D9->Release();


//Shutdown GDI+
GdiplusShutdown(GdiPlusToken);

}




int WINAPI WinMain(HINSTANCE hInstance,HINSTANCE hPrevInstance
    ,LPSTR lpCmdLine,int nShowCmd)
{

MSG Msg;

GdiplusStartupInput GdiPlusStartupInput; //Used for GDI+
    Initialisiation

GData.GraphicsInitialised = FALSE;
GData.IsVideoStreamPlaying = FALSE;

GData.hInstance=hInstance;
GData.VCapture = new VideoCapture();


//Give a pointer to the global data structure to ControlDialog
    Structure
CDialog.SetGlobalData(&GData);



// Initialize GDI+.
GdiplusStartup(&GData.GdiPlusToken, &GdiPlusStartupInput, NULL
    );




//Initialise main windows and if can not be done return error
if (!WinInit(hInstance))
{
MessageBox(NULL,"Error Initialising Main Windows","Fatal Error
    ",MB_OK);
return(false);
}



GData.VCapture->CreateGraph();

SetWindowSize(GData.hWndOutputWindow,GData.VCapture->
    CurrentModeXLength,
                              GData.VCapture->CurrentModeYLength);
SetWindowSize(GData.hWndVideoWindow,GData.VCapture->
```

```
    CurrentModeXLength,
                                GData.VCapture->CurrentModeYLength);


GData.StandardWhitePen = new Pen(Color(255,255,255));


//Create the graphics object for GDI+ associated with output
    window
GData.GOutput = NULL;
GData.BOutput = NULL;
GData.GVideo = NULL;
GData.GBVideo = NULL;
GData.BVideo = NULL;
GData.pVideoWindowBitmap = NULL;

GData.GOutput = new Graphics(GData.hWndOutputWindow,false);
GData.GVideo = new Graphics(GData.hWndVideoWindow,false);

if (GData.GOutput == NULL || GData.GVideo == NULL)
{
MessageBox(NULL,"Error Initialising GDI+ Graphics Objects","
    Fatal Error!",MB_OK);
return(false);
}



//Allocate memory for the GData.pVideoWindowBitmap array
GData.pVideoWindowBitmap = new BYTE[GData.VCapture->
    CurrentModeXLength*GData.VCapture->CurrentModeYLength*4];

GData.pTempBuffer = new BYTE[GData.VCapture->
    CurrentModeXLength*GData.VCapture->CurrentModeYLength*4];

GData.pSampleBuffer = new BYTE[GData.VCapture->
    CurrentModeXLength*GData.VCapture->CurrentModeYLength*4];

GData.pOutputWindowBuffer = NULL;
GData.pOutputWindowBuffer = new BYTE[GData.VCapture->
    CurrentModeXLength*GData.VCapture->CurrentModeYLength*4];

GData.BOutput = new Bitmap( GData.VCapture->CurrentModeXLength
    ,
                                                    GData.
    VCapture->CurrentModeYLength,
                                                    GData.
    VCapture->CurrentModeXLength*4,
                                                    PixelFormat32bppRGB
    ,
                                                    GData.
    pOutputWindowBuffer);


GData.BVideo=new Bitmap( GData.VCapture->CurrentModeXLength,
                                                    GData.
    VCapture->CurrentModeYLength,
                                                    GData.
    VCapture->CurrentModeXLength*4,
                                                    PixelFormat32bppRGB
```

```
,
                                                        GData.
    pVideoWindowBitmap);

GData.GBVideo = new Graphics((Image*)&GData.BVideo);


if (GData.pVideoWindowBitmap == NULL || GData.BVideo == NULL)
{
MessageBox(NULL,"Error Creating Video Window GDI+ Objects","
    Fatal Error",MB_OK);
return(false);
}

//Initialise the Selection Rectangle class used in the Control
    Window
SRectangle.Init(GData.VCapture->CurrentModeXLength,GData.
    VCapture->CurrentModeYLength);

//Set the Timer to send WM_TIMER messages to main window
//SetTimer(GData.hWndVideoWindow,1,25,NULL);



//TIMECAPS  TimerInfo;
//char Message[128];

//timeGetDevCaps(&TimerInfo, sizeof(TimerInfo));
//sprintf_s(Message,128,"Minimum resolution  is %dms, Max
    resolution is %dms",TimerInfo.wPeriodMin,TimerInfo.
    wPeriodMax);
//MessageBox(NULL,Message,"",MB_OK);


GData.GraphicsInitialised = TRUE;



timeBeginPeriod(30); //Set High resolution timer to 30ms

timeSetEvent(30,0,TimerFunc,0,(UINT)TIME_PERIODIC);








while(1)
{
  if (PeekMessage(&Msg,NULL,0,0,PM_NOREMOVE))
  {
    if (!GetMessage(&Msg,NULL,0,0))
        {

        return(Msg.wParam);
        }
        TranslateMessage(&Msg);
```

```
            DispatchMessage(&Msg);

    }
    else
    {
    WaitMessage();
    }

}
}




BOOL WinInit(HINSTANCE hInstance)
{

WNDCLASSEX VisionXOutputWindowClass;
WNDCLASSEX VideoWindowClass;
WNDCLASSEX ControlWindowClass;
HWND OutputWindowHandle;
HWND ControlWindowHandle;

//Array for storing error messages for display
char ErrorMsg[250];


VisionXOutputWindowClass.cbSize=sizeof(WNDCLASSEX);
VisionXOutputWindowClass.style=CS_DBLCLKS;
VisionXOutputWindowClass.lpfnWndProc=OutputWindowProc;
VisionXOutputWindowClass.cbClsExtra=0;
VisionXOutputWindowClass.cbWndExtra=0;
VisionXOutputWindowClass.hInstance=hInstance;
VisionXOutputWindowClass.hIcon=NULL;
VisionXOutputWindowClass.hCursor=LoadCursor(NULL,IDC_ARROW);
//VisionXOutputWindowClass.hbrBackground=NULL;
VisionXOutputWindowClass.hbrBackground=(HBRUSH)GetStockObject(
    BLACK_BRUSH);
VisionXOutputWindowClass.lpszMenuName=NULL;
VisionXOutputWindowClass.lpszClassName="
    VisionX_Output_Window_Class";
VisionXOutputWindowClass.hIconSm=NULL;

if (!RegisterClassEx(&VisionXOutputWindowClass))
{
memset(ErrorMsg,0,sizeof(ErrorMsg));
sprintf_s(ErrorMsg,sizeof(ErrorMsg),"Error Registering Output 
    Window Class - Error Code %d",GetLastError());
MessageBox(NULL,ErrorMsg,"Fatal Error",MB_OK);
return FALSE;
}


GData.hWndOutputWindow = CreateWindowEx(WS_EX_APPWINDOW,
"VisionX_Output_Window_Class",
"VisionX Output Window",
WS_BORDER | WS_CAPTION | WS_POPUP | WS_VISIBLE |
    WS_MAXIMIZEBOX | WS_MINIMIZEBOX | WS_SYSMENU,
0,
0,
```

```
GetSystemMetrics(SM_CXSCREEN)/2,
GetSystemMetrics(SM_CYSCREEN)*2/3,
NULL,
NULL,
hInstance,
NULL);

if (GData.hWndOutputWindow == NULL)
{
MessageBox(NULL,"Error Creating Output Window","Fatal Error",
    MB_OK);
return false;
}




VideoWindowClass.cbSize=sizeof(WNDCLASSEX);
VideoWindowClass.style=CS_DBLCLKS;
VideoWindowClass.lpfnWndProc=VideoWindowProc;
VideoWindowClass.cbClsExtra=0;
VideoWindowClass.cbWndExtra=0;
VideoWindowClass.hInstance=hInstance;
VideoWindowClass.hIcon=NULL;
VideoWindowClass.hCursor=LoadCursor(NULL,IDC_ARROW);
//VideoWindowClass.hbrBackground=NULL;
VideoWindowClass.hbrBackground=(HBRUSH)GetStockObject(
    BLACK_BRUSH);
VideoWindowClass.lpszMenuName=NULL;
VideoWindowClass.lpszClassName="VisionX_Video_Window_Class";
VideoWindowClass.hIconSm=NULL;

if (!RegisterClassEx(&VideoWindowClass))
{
memset(ErrorMsg,0,sizeof(ErrorMsg));
sprintf_s(ErrorMsg,sizeof(ErrorMsg),"Error Registering Video 
    Window Class - Error Code %d",GetLastError());
MessageBox(NULL,ErrorMsg,"Fatal Error",MB_OK);
return FALSE;
}


GData.hWndVideoWindow = CreateWindowEx(WS_EX_APPWINDOW,
"VisionX_Video_Window_Class",
"VisionX Video Window",
WS_BORDER | WS_CAPTION | WS_POPUP | WS_VISIBLE |
    WS_MAXIMIZEBOX | WS_MINIMIZEBOX | WS_SYSMENU,
0,
0,
GetSystemMetrics(SM_CXSCREEN)/2,
GetSystemMetrics(SM_CYSCREEN)*2/3,
NULL,
NULL,
hInstance,
NULL);

if (GData.hWndVideoWindow == NULL)
{
MessageBox(NULL,"Error Creating Output Window","Fatal Error",
```

```
    MB_OK);
return false;
}



//Create the Control Dialog Box

GData.hWndControlWindow = CreateDialog(hInstance,
    MAKEINTRESOURCE(IDD_CONTROLDIALOG),
NULL, ControlDialogProc);

ShowWindow(GData.hWndControlWindow,SW_SHOW);

if (GData.hWndControlWindow==NULL)
{
MessageBox(NULL,"Error Creating Control Dialog Window","Fatal
    Error",MB_OK);
PostQuitMessage(0);
return(false);
}

//if (InitDirect3D() == false) return(false);


//Initialise the Controls in the Control Window
return(CDialog.InitControls());


}




LRESULT CALLBACK OutputWindowProc(HWND hWnd,UINT message,
    WPARAM wParam,LPARAM lParam)
{
HDC hDC;
PAINTSTRUCT PS;
BitmapData BData;
Bitmap* pTempBitmap;

static GSpace* GradientSpace = NULL;
static LineDetector* LDetector = NULL;

static BOOL HasGradientSpaceCreated=FALSE;
```

```
static BOOL HasThresholdingBeenDone=FALSE;

switch(message)
{
  case WM_MOUSEMOVE:

    if (SRectangle.IsSelectionWindowActive==TRUE)
        {
        SRectangle.CurrentWidth=LOWORD(lParam)-SRectangle.X;
        SRectangle.CurrentHeight=HIWORD(lParam)-SRectangle.Y;
        PostMessage(hWnd,WM_PAINT,0,0);
        }

  break;

  case WM_LBUTTONDOWN:

  SRectangle.X=LOWORD(lParam);
  SRectangle.Y=HIWORD(lParam);
  SRectangle.IsSelectionWindowActive=TRUE;

  break;

  case WM_LBUTTONUP:
  SRectangle.IsSelectionWindowActive=FALSE;

  SRectangle.GrabData(GData.pOutputWindowBuffer);
  SRectangle.HasSelectionBeenMade=TRUE;


  //For Testing can be removed
  //pTempBitmap = new Bitmap(640/8,480/8,640/2,
   PixelFormat32bppRGB, SRectangle.pBuffer);
  //GData.GOutput->DrawImage((Image*)pTempBitmap, RectF
   (0,0,640,480));

  PostMessage(hWnd,WM_PAINT,0,0);


  break;


  case WM_PAINT:

  if (GData.GraphicsInitialised == FALSE) break;

  hDC = BeginPaint(hWnd,&PS);
  Render();

  if (SRectangle.IsSelectionWindowActive==TRUE)
  {
  SRectangle.DrawSelectionRectangle(GData.GOutput);

  }

  ValidateRect( hWnd, NULL );
  EndPaint(hWnd,&PS);
  return(1);
```

```
case WM_SIZE:
break;
case WM_MOVE:
break;
case WM_KEYDOWN:
        switch(wParam)
        {

    //if user presses '0' display the gradient bitmap
        case 0x30:
            if (SRectangle.HasSelectionBeenMade==TRUE)
                {
                //Check that user has magnified window and
 apply gradient function to that 'zoomed' data

      GradientSpace = new GSpace(SRectangle.CurrentWidth,
 SRectangle.CurrentHeight);

                GradientSpace->CreateGradientSpace(SRectangle.
 pBuffer);

                SRectangle.pDisplayData->LockBits(&Rect(0,0,
 SRectangle.pDisplayData->GetWidth(),SRectangle.pDisplayData
 ->GetHeight()),
           ImageLockModeWrite,
                PixelFormat32bppRGB,&BData);

      SRectangle.DrawData(GradientSpace->pGBuffer,GData.
 VCapture->CurrentModeXLength*4,SRectangle.pDummyBuffer,8);
      SRectangle.pDisplayData->UnlockBits(&BData);

      GData.GOutput->DrawImage((Image*)SRectangle.
 pDisplayData,0,0);

      HasGradientSpaceCreated = TRUE;

                }  //End of if

    return(1);

        case 0x31:


            if (HasGradientSpaceCreated == TRUE &&
 GradientSpace!=NULL)
                {


      //Create the thresholded data
                GradientSpace->DoThresholding();
                SRectangle.pDisplayData->LockBits(&Rect(0,0,
 SRectangle.pDisplayData->GetWidth(),SRectangle.pDisplayData
```

```
->GetHeight()),
                ImageLockModeWrite,
                PixelFormat32bppRGB,&BData);

      SRectangle.DrawData(GradientSpace->pEBuffer,GData.
VCapture->CurrentModeXLength*4,SRectangle.pDummyBuffer,8);
      SRectangle.pDisplayData->UnlockBits(&BData);

      GData.GOutput->DrawImage((Image*)SRectangle.
pDisplayData,0,0);

                HasThresholdingBeenDone = TRUE;

                }


         return(1);

         case 0x32:

          if (HasGradientSpaceCreated == TRUE &&
HasThresholdingBeenDone == TRUE)
          {
     LDetector = new LineDetector(GradientSpace->
SourceBufferXLength,GradientSpace->SourceBufferYLength);

     LDetector->DetectLines(GradientSpace->pEBuffer);

          /*Lock the buffer and then draw to it*/
     SRectangle.pDisplayData->LockBits(&Rect(0,0,SRectangle.
pDisplayData->GetWidth(),SRectangle.pDisplayData->GetHeight
()),
          ImageLockModeWrite,
          PixelFormat32bppRGB,&BData);

          /*Draw the big pixels in pDummyBuffer from
SRectangle class */
      SRectangle.DrawData(GradientSpace->pEBuffer,GData.
VCapture->CurrentModeXLength*4,SRectangle.pDummyBuffer,8);
      SRectangle.pDisplayData->UnlockBits(&BData);

      GData.GOutput->DrawImage((Image*)SRectangle.
pDisplayData,0,0);

          }


         return(1);



            case VK_RETURN:

                          //If user has magnified region and
pressed enter run the blending algorithm
                          //On magnified region to hopefully
enhance the image
                          if (SRectangle.HasSelectionBeenMade==
TRUE)
```

```
                        {

            Image4x4Blender((unsigned int*)SRectangle.
   pEnhancedData,
                                (unsigned int*)SRectangle.
   pBuffer,Rect(0,0,SRectangle.CurrentWidth,
                                SRectangle.CurrentHeight));

                        //SRectangle.GrabCustomData();

            //May not be a good idea to do this, this way -
   garbage collection may not pick up
                        //could get memory leak
                        Bitmap* TestBitmap;

                        TestBitmap = new Bitmap(SRectangle.
   CurrentWidth*2,SRectangle.CurrentHeight*2,
                        SRectangle.CurrentWidth*4*2,
   PixelFormat32bppRGB,
                        SRectangle.pEnhancedData);

                        GData.GOutput->DrawImage((Image*)
   TestBitmap,0,0,640,480);



                        }


                return(1);


            case VK_ESCAPE:

                if (SRectangle.HasSelectionBeenMade==TRUE)
                {
                SRectangle.HasSelectionBeenMade=FALSE;
                PostMessage(hWnd,WM_PAINT,0,0);
                return(1);

                }

                if (SRectangle.HasSelectionBeenMade==FALSE)
                {
                PostMessage(hWnd,WM_CLOSE,0,0);
                }

                return(0);

        break;
            }
  break;

  case WM_DESTROY:
    PostQuitMessage(0);
  break;


}
return(DefWindowProc(hWnd,message,wParam,lParam));
```

```
}




LRESULT CALLBACK VideoWindowProc(HWND hWnd,UINT message,
    WPARAM wParam,LPARAM lParam)
{
HDC hDC;
PAINTSTRUCT PS;


static Rect SampleArea(0,0,640,480);
static BitmapData BData;
static OpticalFlow OFlow(640,10,480,10,25);
static BackgroundSubtraction BSubtraction(640,480,25);
static Striper ObjectDetector(640,480,5,10,20,500);
static ObjectTracker OTracker(100,640,480,BSubtraction.
    pBaseImage,BSubtraction.pResult,GData.pSampleBuffer);

static double TotalRenderingTime = 0;
static double MessagesProcessed = 0;
static int StartTime;
static int StopTime;
static char Message[128];

LONGLONG CurrentMediaPos;
LONGLONG StopPosition;


switch(message)
{
  case WM_TIMER:

   StartTime = timeGetTime();

  BData.Width = 640;
  BData.Height = 480,
  BData.Stride = 4*BData.Width;
  BData.PixelFormat = PixelFormat32bppRGB;
  BData.Scan0 = (VOID*)GData.pSampleBuffer;
  BData.Reserved = NULL;




  //Grab a sample from the video stream
  GData.VCapture->GetVideoSampleA((BYTE*)GData.pSampleBuffer);

  //Set the new background to the previously grabbed image
  if (BSubtraction.IsBaseImageInitialised == TRUE)
  {
  BSubtraction.InitBaseImage(GData.pSampleBuffer);
  }
```

```
/*Check if the base image has been initialised */
  if (BSubtraction.IsBaseImageInitialised==FALSE && GData.
 IsVideoStreamPlaying == TRUE)
     {
  /*Not Initialised so Initialise now */


        if (GData.VCapture->MS->GetPositions(&
 CurrentMediaPos,&StopPosition)!=S_OK)
        {
        MessageBox(NULL,"Error Seeking Video Stream VisionX.
 cpp VideoWindowProc()","Error",MB_OK);
        PostQuitMessage(0);
        }
        if (CurrentMediaPos>0)
        {
        BSubtraction.InitBaseImage(GData.pSampleBuffer);
        }

     }


if (BSubtraction.IsBaseImageInitialised == TRUE)
{

BSubtraction.DoSubtraction(GData.pSampleBuffer);
BSubtraction.FramesProcessedSinceUpdate++;

//Clear out the temp buffer
memset((void*)GData.pTempBuffer,0,GData.VCapture->
 CurrentModeXLength*GData.VCapture->CurrentModeYLength*4);

BSubtraction.DrawImage(GData.pTempBuffer);
}

ObjectDetector.DetectPossibleObjects(GData.pTempBuffer);
ObjectDetector.DetectObjects(GData.pTempBuffer);
ObjectDetector.GetBoundingSquares(BSubtraction.pBaseImage,
 GData.pSampleBuffer);
OTracker.UpdateObjects(ObjectDetector.ObjectsDetected,
 ObjectDetector.ObjectsFound);
//OTracker.DrawBoundingSquares(GData.pTempBuffer);
OTracker.CreateMask(BSubtraction.pMask);
//ObjectDetector.DrawBoundingSquares(GData.pTempBuffer);
//BSubtraction.DrawMaskRegions(GData.pTempBuffer);


memcpy((void*)GData.pTempBuffer,(void*)GData.pSampleBuffer
 ,640*480*4);

/*Update Output Bitmap with new data */
if (GData.BVideo->LockBits(&SampleArea,ImageLockModeWrite|
 ImageLockModeUserInputBuf,PixelFormat32bppRGB,&BData)!=Ok)
{
```

```
MessageBox(NULL,"Error Locking Video Stream Bitmap in 
  VideoWindowProc","Fatal Error",MB_OK);
PostQuitMessage(0);
return(FALSE);
}

GData.BVideo->UnlockBits(&BData);


GData.GVideo->DrawImage((Image*)GData.BVideo,0,0);

 StopTime = timeGetTime();

TotalRenderingTime+=(StopTime-StartTime);
MessagesProcessed++;

 break;


case WM_MOUSEMOVE:
break;

case WM_LBUTTONDOWN:
break;

case WM_PAINT:

  hDC = BeginPaint(hWnd,&PS);

  ValidateRect( hWnd, NULL );
  EndPaint(hWnd,&PS);
  return(1);

case WM_SIZE:

break;

case WM_MOVE:

break;

case WM_KEYDOWN:

        switch(wParam)
        {

        case VK_SPACE:

    sprintf_s(Message,128,"Average Rendering time  = %e ms",
 TotalRenderingTime/MessagesProcessed);
    MessageBox(NULL,Message,"",MB_OK);

                break;

          case VK_ESCAPE:
                break;

        }


case WM_DESTROY:
  PostQuitMessage(0);
```

```
   break;

}

return(DefWindowProc(hWnd, message ,wParam, lParam));
}





BOOL CALLBACK ControlDialogProc(HWND hwndDlg,UINT uMsg,WPARAM
    wParam,LPARAM lParam)
{
HRESULT hr;


switch(uMsg)
{


  case WM_COMMAND:



    if (HIWORD(wParam) == BN_CLICKED)
    {

           //User wants to grab a frame to use for processing
           if (lParam == (LPARAM)CDialog.hGrabImageButton)
           {

      memcpy(GData.pOutputWindowBuffer,GData.pTempBuffer,GData.
    VCapture->CurrentModeXLength*GData.VCapture->
    CurrentModeYLength*4);

      //Display the image in the output window
           Render();


           }


    //Check if user wants to play video
    if (lParam == (LPARAM)CDialog.hPlayStreamButton)
    {
    LONGLONG CurrentPosition;
    LONGLONG StopPosition;

    GData.VCapture->MS->GetPositions(&CurrentPosition,&
    StopPosition);

      //Check if stream stopped playing and needs a reset
      if (CurrentPosition >= StopPosition)
           {
      //Yes we need reset
           CurrentPosition = 0;
```

```
        GData.VCapture->MS->SetPositions(&CurrentPosition,
    AM_SEEKING_AbsolutePositioning,
        &StopPosition, AM_SEEKING_AbsolutePositioning);

        }


    GData.VCapture->MC->Run();

    GData.IsVideoStreamPlaying = TRUE;

    }


    //Check if user wants to stop video
    if (lParam == (LPARAM)CDialog.hStopStreamButton)
    {
    GData.VCapture->MC->Pause();
    GData.IsVideoStreamPlaying = FALSE;
    }


    } //End of if (HIWORD(wParam) == BN_CLICKED)

   break;



}


return(false);
}







void ControlDialog::SetGlobalData(GlobalData* pData)
{
pGData=pData;
}






BOOL ControlDialog::InitControls()
{

hPlayStreamButton = GetDlgItem(GData.hWndControlWindow,
    IDC_PLAYSTREAMBUTTON);
 if (hPlayStreamButton==NULL)
```

```
{
MessageBox(NULL,"Failed To Obtain 'Play Stream' Button Control
    Handle (VisionX.cpp)",NULL,MB_OK);
PostQuitMessage(0);
return(false);
}

hStopStreamButton = GetDlgItem(GData.hWndControlWindow,
    IDC_STOPSTREAMBUTTON);
if (hStopStreamButton==NULL)
{
MessageBox(NULL,"Failed To Obtain 'Stop Stream' Button Control
    Handle (VisionX.cpp)",NULL,MB_OK);
PostQuitMessage(0);
return(false);
}


hGrabImageButton = GetDlgItem(GData.hWndControlWindow,
    IDC_GRABIMAGEBUTTON);
if (hGrabImageButton==NULL)
{
MessageBox(NULL,"Failed To Obtain 'Grab Image Button' Control
    Handle (VisionX.cpp)",NULL,MB_OK);
PostQuitMessage(0);
return(false);
}

return(true);
}
```

```
BOOL InitDirect3D()   //Not used at this stage
{
D3DPRESENT_PARAMETERS D3DParameters;

   if (NULL == ( GData.pDirect3D9 = Direct3DCreate9(
    D3D_SDK_VERSION ) ) )
   {
   MessageBox(NULL,"Failed to Initialise Direct3D 9","Fatal
    Error",MB_OK);
   return (false);
   }
```

```
//zero the D3D Parameter Structure
memset(&D3DParameters,0,sizeof(D3DParameters));


D3DParameters.Windowed = TRUE;
D3DParameters.SwapEffect = D3DSWAPEFFECT_DISCARD;
//D3DParameters.BackBufferFormat = D3DFMT_UNKNOWN;
D3DParameters.BackBufferFormat = D3DFMT_R8G8B8;

D3DParameters.hDeviceWindow = GData.hWndOutputWindow;


if( FAILED( GData.pDirect3D9->CreateDevice( D3DADAPTER_DEFAULT
    , D3DDEVTYPE_HAL, GData.hWndOutputWindow,
      D3DCREATE_SOFTWARE_VERTEXPROCESSING,
      &D3DParameters, &GData.p3DDevice) ) )
{
MessageBox(NULL,"Error Creating Direct 3D Device Interface","
    Fatal Error",MB_OK);
return(false);
}






return(true);
}





//Renders the output to the output window
void Render()
{
BitmapData BData;



//Check if user wants a magnified window or not
if (SRectangle.HasSelectionBeenMade==FALSE)
{
//If in here user just wants the normal frame drawn in output
    window


GData.GOutput->DrawImage((Image*)GData.BOutput,0,0);


}
else
{
//If in here user selected to have a magnified window


SRectangle.pDisplayData->LockBits(&Rect(0,0,SRectangle.
    pDisplayData->GetWidth(),SRectangle.pDisplayData->GetHeight
    ()),
ImageLockModeWrite,
PixelFormat32bppRGB,&BData);
```

```
SRectangle.DrawData(SRectangle.pBuffer,GData.VCapture->
    CurrentModeXLength*4,(BYTE*)BData.Scan0,8);

SRectangle.pDisplayData->UnlockBits(&BData);

GData.GOutput->DrawImage((Image*)SRectangle.pDisplayData,0,0);

}


}


void SetWindowSize(HWND hWnd, int x, int y)
{
int nTitleHeight  = GetSystemMetrics(SM_CYCAPTION);
int nBorderWidth  = GetSystemMetrics(SM_CXBORDER);
int nBorderHeight = GetSystemMetrics(SM_CYBORDER);


SetWindowPos(hWnd, NULL, 0, 0, x + 2*nBorderWidth,
y + nTitleHeight + 2*nBorderHeight,
SWP_NOMOVE | SWP_NOOWNERZORDER);

}




static void CALLBACK TimerFunc(UINT wID,UINT wUser, DWORD
    dwUser, DWORD dw1,DWORD dw2)
{

static Rect SampleArea(0,0,640,480);
static BitmapData BData;

static OpticalFlow OFlow(640,10,480,10,25);
static BackgroundSubtraction BSubtraction(640,480,25);
static Striper ObjectDetector(640,480,5,10,20,500);
static ObjectTracker OTracker(100,640,480,BSubtraction.
    pBaseImage,BSubtraction.pResult,GData.pSampleBuffer);


static double TotalRenderingTime = 0;
static double MessagesProcessed = 0;
static int StartTime;
static int StopTime;
static char Message[128];

static LONGLONG CurrentMediaPos;
static LONGLONG StopPosition;


StartTime = timeGetTime();
```

```
BData.Width = 640;
BData.Height = 480;
BData.Stride = 4*BData.Width;
BData.PixelFormat = PixelFormat32bppRGB;
BData.Scan0 = (VOID*)GData.pSampleBuffer;
BData.Reserved = NULL;




//Grab a sample from the video stream
GData.VCapture->GetVideoSampleA((BYTE*)GData.pSampleBuffer);


//Set the new background to the previously grabbed image
if (BSubtraction.IsBaseImageInitialised == TRUE)
{
BSubtraction.InitBaseImage(GData.pSampleBuffer);
}


/*Check if the base image has been initialised */
  if (BSubtraction.IsBaseImageInitialised==FALSE && GData.
 IsVideoStreamPlaying == TRUE)
       {
  /*Not Initialised so Initialise now */


         if (GData.VCapture->MS->GetPositions(&
 CurrentMediaPos,&StopPosition)!=S_OK)
         {
         MessageBox(NULL,"Error Seeking Video Stream VisionX.
 cpp VideoWindowProc()","Error",MB_OK);
         PostQuitMessage(0);
         }
         if (CurrentMediaPos>0)
         {
         BSubtraction.InitBaseImage(GData.pSampleBuffer);
         }

       }


if (BSubtraction.IsBaseImageInitialised == TRUE)
{

BSubtraction.DoSubtraction(GData.pSampleBuffer);
BSubtraction.FramesProcessedSinceUpdate++;

//Clear out the temp buffer
memset((void*)GData.pTempBuffer,0,GData.VCapture->
 CurrentModeXLength*GData.VCapture->CurrentModeYLength*4);

BSubtraction.DrawImage(GData.pTempBuffer);
```

```
    }

    ObjectDetector.DetectPossibleObjects(GData.pTempBuffer);
    ObjectDetector.DetectObjects(GData.pTempBuffer);
    ObjectDetector.GetBoundingSquares(BSubtraction.pBaseImage,
     GData.pSampleBuffer);
    ObjectDetector.EliminateInternalObjects();
    OTracker.UpdateObjects(ObjectDetector.ObjectsDetected,
     ObjectDetector.ObjectsFound);
    OTracker.DrawObjectTracks(GData.pTempBuffer);
    OTracker.DrawBoundingSquares(GData.pSampleBuffer);
    OTracker.CreateMask(BSubtraction.pMask);
    //ObjectDetector.DrawBoundingSquares(GData.pSampleBuffer);
    //BSubtraction.DrawMaskRegions(GData.pTempBuffer);

    /* Update Output Bitmap with new data */
    if (GData.BVideo->LockBits(&SampleArea,ImageLockModeWrite|
     ImageLockModeUserInputBuf,PixelFormat32bppRGB,&BData)!=Ok)
    {
    MessageBox(NULL,"Error Locking Video Stream Bitmap in
     VideoWindowProc","Fatal Error",MB_OK);
    PostQuitMessage(0);
    return;
    }

    GData.BVideo->UnlockBits(&BData);



    GData.GVideo->DrawImage((Image*)GData.BVideo,0,0);
    //OTracker.DrawObjectNumbers(GData.GVideo); //Draw the
     object numbers of each validated object
    OTracker.DrawObjectNumbers(GData.GVideo); //Draw the object
     numbers of each validated object

  OTracker.DrawCounter(GData.GVideo);

  StopTime = timeGetTime();

TotalRenderingTime+=(StopTime-StartTime);
MessagesProcessed++;


}
```

B.7 VideoCapture.cpp

```
#include <stdio.h>

#include <windows.h>
#include <oleauto.h>
#include "VideoCapture.h"
```

```
BOOL VideoCapture::CreateGraph()
{
HRESULT hr;


//Create the AVIFileSource Filter
hr = CoCreateInstance(CLSID_AsyncReader, NULL,
    CLSCTX_INPROC_SERVER,
     IID_IBaseFilter, (void**)&pAVIFileSourceF);
if (hr!=S_OK) //Check for error Adding Capture Device to
    DirectShow FilterGraph
{
MessageBox(NULL,"Error Creating Asynch File Source for filter 
    graph","Fatal Error",MB_OK);
PostQuitMessage(0);
return(false);
}

//Add the AVI source file to the graph
hr = pGraph->AddFilter(pAVIFileSourceF,(LPCWSTR)"AVI Source 
    file");

if (hr!=S_OK)
{
MessageBox(NULL,"Error Adding AVI File Source to Graph","Fatal
     Error",MB_OK);
PostQuitMessage(0);
return(false);
}

//Setup the file source
IFileSourceFilter* pSourceFilter;

hr = pAVIFileSourceF->QueryInterface(IID_IFileSourceFilter, (
    void **)&pSourceFilter);
if (hr!=S_OK)
{
MessageBox(NULL,"Error Getting Source Filter Interface","Fatal
     Error",MB_OK);
PostQuitMessage(0);
return(false);
}

hr=pSourceFilter->Load(L"C:\\Test.avi",NULL);
if (hr!=S_OK)
{
MessageBox(NULL,"Error Loading source AVI file","Fatal Error",
    MB_OK);
PostQuitMessage(0);
return(false);
}




//Create the MJPEG Decompressor Filter
```

```
hr = CoCreateInstance(CLSID_MjpegDec, NULL,
    CLSCTX_INPROC_SERVER,
     IID_IBaseFilter, (void**)&pMJPEGDecompressorF);
if (hr!=S_OK) //Check for error Adding Capture Device to
    DirectShow FilterGraph
{
MessageBox(NULL,"Error Creating MJPEG Decompressor for filter
    graph","Fatal Error",MB_OK);
PostQuitMessage(0);
return(false);
}

//Add the MJPEG Decompressor to the graph
hr = pGraph->AddFilter(pMJPEGDecompressorF,(LPCWSTR)"MJPEG
    Decompressor");
if (hr!=S_OK)
{
MessageBox(NULL,"Error Adding MJPEG filter to Graph","Fatal
    Error",MB_OK);
PostQuitMessage(0);
return(false);
}



// Create the Sample Grabber.
hr = CoCreateInstance(CLSID_SampleGrabber, NULL,
    CLSCTX_INPROC_SERVER,
     IID_IBaseFilter, (void**)&pGrabberF);
if (hr!=S_OK) //Check for error Adding Capture Device to
    DirectShow FilterGraph
{
MessageBox(NULL,"Error Creating Sample Grabber","Fatal Error",
    MB_OK);
PostQuitMessage(0);
return(false);
}

//Add the sample grabber to Graph
hr = pGraph->AddFilter(pGrabberF,(LPCWSTR)"Sample Grabber");
if (hr!=S_OK)
{
MessageBox(NULL,"Error Adding Sample Grabber to DirectShow
    Graph","Fatal Error",MB_OK);
PostQuitMessage(0);
return(false);
}


//Get the interface ISampleGrabber
hr = pGrabberF->QueryInterface(IID_ISampleGrabber,(void**)&
    pGrabber);
if (hr!=S_OK)
{
MessageBox(NULL,"Error Obtaining SampleGrabber Interface from
    Graph","Fatal Error",MB_OK);
```

```
PostQuitMessage(0);
return(false);
}



//Set the expected data input stream type for the grabber
AM_MEDIA_TYPE mt;
memset((void*)&mt,0, sizeof(AM_MEDIA_TYPE));
mt.majortype = MEDIATYPE_Video;
mt.subtype = MEDIASUBTYPE_RGB32;
hr = pGrabber->SetMediaType(&mt);



//Create the Interface for the Video Mixing Renderer 9
IBaseFilter* pRendererF;

hr = CoCreateInstance(CLSID_NullRenderer, NULL,
    CLSCTX_INPROC_SERVER,
     IID_IBaseFilter, (void**)&pRendererF);
if (hr!=S_OK) //Check for error Adding Capture Device to
    DirectShow FilterGraph
{
MessageBox(NULL,"Error Creating NULL Renderer 9","Fatal Error"
    ,MB_OK);
PostQuitMessage(0);
return(false);
}

//Add the Null Renderer to Graph
hr = pGraph->AddFilter(pRendererF,(LPCWSTR)"Video Renderer");
if (hr!=S_OK) //Check for error Adding Capture Device to
    DirectShow FilterGraph
{
MessageBox(NULL,"Error Adding Video Renderer to DirectShow
    Graph","Fatal Error",MB_OK);
PostQuitMessage(0);
return(false);
}

//Obtain pin for AVI source file
if (pAVIFileSourceF->EnumPins(&pAVIFileSourcePins)!=S_OK)  //
    Obtain Sample Grabber Pins
  MessageBox(NULL,"Unable to Enumerate pins on on AVI File
    Source",NULL,MB_OK);

if (pAVIFileSourcePins->Next(1,&pAVIFileSourceOutputPin,NULL)
    !=S_OK)
        MessageBox(NULL,"Failed to get Source Output Pin!!",
    NULL,MB_OK);

//Obtain pins for MJPEG decompressor
if (pMJPEGDecompressorF->EnumPins(&pMJPEGDecompressorPins)!=
    S_OK)  //Obtain Sample Grabber Pins
  MessageBox(NULL,"Could not enumerate pins for MJPEG
    Decompressor",NULL,MB_OK);
```

```
if (pMJPEGDecompressorPins->Next(1,&pMJPEGDecompressorInputPin
    ,NULL)==S_FALSE)
        MessageBox(NULL,"Failed to get MJPEG Decompressor 
    Input Pin!!!",NULL,MB_OK);
if (pMJPEGDecompressorPins->Next(1,&
    pMJPEGDecompressorOutputPin,NULL)!=S_OK)
     MessageBox(NULL,"Failed to get MJPEG Decompressor Output 
    Pin",NULL,MB_OK);



pGrabberF->EnumPins(&pGrabberPins);   //Obtain Sample Grabber
    Pins
if (pGrabberPins->Next(1,&pGrabberInputPin,NULL)!=S_OK)
    MessageBox(NULL,"Could not get Sample Grabber Input Pin","
    Error",MB_OK);
if (pGrabberPins->Next(1,&pGrabberOutputPin,NULL)!=S_OK)
    MessageBox(NULL,"Could not get Sample Grabber Output Pin","
    Error",MB_OK);

pRendererF->EnumPins(&pRendererPins); //Obtain pins for
    Renderer should only be 1
if (pRendererPins->Next(1,&pRendererInputPin,NULL)!=S_OK) //
    Obtain the input pin
     MessageBox(NULL,"Could not get Renderer input pin","Error"
    ,MB_OK);




hr = pGraph->Connect(pAVIFileSourceOutputPin,
    pMJPEGDecompressorInputPin);
if (hr!=VFW_S_PARTIAL_RENDER && hr!=S_OK)
{
MessageBox(NULL,"!!! Error Connecting DirectShow Source File 
    Output Pin to MJPEG Decompr","Fatal Error",MB_OK);
PostQuitMessage(0);
return(false);
}

hr = pGraph->Connect(pMJPEGDecompressorOutputPin,
    pGrabberInputPin);
if (hr!=S_OK)
{
MessageBox(NULL,"!!! Error Connecting MJPEG Decompressor to 
    Sample Grabber","Fatal Error",MB_OK);
PostQuitMessage(0);
return(false);
}


hr = pGraph->Connect(pGrabberOutputPin,pRendererInputPin);
```

```
if (hr!=S_OK)
{
MessageBox(NULL,"Error Connecting Sample Grabber output pin to
    Renderer input pin","Fatal Error",MB_OK);
PostQuitMessage(0);
return(false);
}



//Grab the IMediaControl Inteface for controlling flow of data
     through graph
pGraph->QueryInterface(IID_IMediaControl,(void**)&MC);

//Grab the IMediaSeeking Interface from the Filter Graph Which
     allows accurate control
//of the video stream by enabling seeking etc..
pGraph->QueryInterface(IID_IMediaSeeking,(void**)&MS);

//Tell the sample grabber to Grab samples of the frames
    passing through filter
if FAILED(pGrabber->SetBufferSamples(true))
{
MessageBox(NULL,"Error Programming Sample Grabber","Fatal
    Error",MB_OK);
PostQuitMessage(0);
return(false);
}


//Get the connected media type information and store within
    class
MEDIA_TYPE = new AM_MEDIA_TYPE();
pGrabber->GetConnectedMediaType(MEDIA_TYPE);

if (MEDIA_TYPE->subtype!=MEDIASUBTYPE_RGB32)
{
MessageBox(NULL,"Video Input is not streaming at 32 bits per
    pixel","Error",MB_OK);
PostQuitMessage(0);
return(false);
}

if (MEDIA_TYPE->formattype!=FORMAT_VideoInfo)
{
        MessageBox(NULL,"Error Wrong MEDI_TYPE Format Block
    supplied","Error",MB_OK);
        PostQuitMessage(0);
        return(false);
}


//Obtain the current video mode width and height. Should be
    640x480
this->CurrentModeXLength=((VIDEOINFOHEADER*)MEDIA_TYPE->
    pbFormat)->bmiHeader.biWidth;
this->CurrentModeYLength=((VIDEOINFOHEADER*)MEDIA_TYPE->
```

```
    pbFormat)->bmiHeader.biHeight;

//Allocate memory for the buffer that will store grabbed
    frames
//Use Width*Height* 3Bytes per pixel
this->BufferSize = this->CurrentModeXLength*this->
    CurrentModeYLength*4;
this->pVideoBuffer = new BYTE[BufferSize];
this->pTempBuffer = new BYTE[BufferSize];


return(true);
}







//Initialise Direct Show and render Web Cam in Window
VideoCapture::VideoCapture()
{

pGraph=NULL;

MEDIA_TYPE=NULL;
pGrabber=NULL;
pStreamConfig=NULL;

pGrabberPins=NULL;
pRendererPins=NULL;
pAVIFileSourcePins=NULL;
pMJPEGDecompressorPins=NULL;

pAVIFileSourceOutputPin=NULL;
pMJPEGDecompressorInputPin=NULL;
pMJPEGDecompressorOutputPin=NULL;
pGrabberInputPin=NULL;
pGrabberOutputPin=NULL;
pRendererInputPin=NULL;


pAVIFileSourceF=NULL;
pMJPEGDecompressorF=NULL;
pGrabberF=NULL;

MC=NULL;
pVideoBuffer=NULL;
pTempBuffer=NULL;


//Initialise the COM system
HRESULT hr = CoInitialize(NULL);
if (FAILED(hr))  //Check for errors
{
MessageBox(NULL,"Error Initialising COM system","Fatal Error",
    MB_OK);
PostQuitMessage(0);
return;
```

```
}


//Create Instance of the FilterGraph class which controls all
    DirectShow filters
hr = CoCreateInstance(CLSID_FilterGraph, NULL,
    CLSCTX_INPROC_SERVER, IID_IGraphBuilder, (void **)&pGraph)
    ;
if (hr!=S_OK) //Check for error creating graph
{
MessageBox(NULL,"Error Creating FilterGraph","Fatal Error",
    MB_OK);
PostQuitMessage(0);
return;
}


}




VideoCapture::~VideoCapture()
{
//Release all COM stuff
if (MEDIA_TYPE!=NULL) delete MEDIA_TYPE;

if (pGrabber!=NULL) pGrabber->Release();


if (pGrabberInputPin!=NULL) pGrabberInputPin->Release();
if (pGrabberPins!=NULL) pGrabberPins->Release();
if (pRendererPins!=NULL) pRendererPins->Release();
if (pRendererInputPin!=NULL) pRendererInputPin->Release();

if (pAVIFileSourceF!=NULL) pAVIFileSourceF->Release();
if (pMJPEGDecompressorF!=NULL) pMJPEGDecompressorF->Release();
if (pGrabberF!=NULL) pGrabberF->Release();
if (pGrabber!=NULL) pGrabber->Release();
if (pGraph!=NULL) pGraph->Release();

if (MEDIA_TYPE!=NULL) free(MEDIA_TYPE);
if (pVideoBuffer!=NULL) free(pVideoBuffer);
if (pTempBuffer!=NULL) free(pTempBuffer);

//Uninitialise COM
CoUninitialize();
}




IAMStreamConfig* VideoCapture::GetStreamConfigInterface()
```

```
{
return(this->pStreamConfig);
}



//Obtains a sample of the video stream and stores in the
    Buffer supplied
BOOL VideoCapture::GetVideoSampleA(BYTE* pSuppliedBuffer)
{
int Size;
int Position1;
int Position2;

int* pSourceBuffer;
int* pDestBuffer;

pSourceBuffer = (int*)pTempBuffer;
pDestBuffer = (int*)pSuppliedBuffer;

   if ((pGrabber->GetCurrentBuffer(&BufferSize,(long*)
    pTempBuffer))!=S_OK)
   {

   //MessageBox(NULL,"Could not Grab Sample from image stream
    ","Fatal Error",MB_OK);

   //PostQuitMessage(0);
   //return(false);
   }

Position1 = CurrentModeXLength*CurrentModeYLength-1-
    CurrentModeXLength;
Position2 = 0;

int Increment = 2*CurrentModeXLength;

   for (int i=0; i<this->CurrentModeYLength; i++)
   {
      //Copy a row from reversed buffer to corrected buffer
      for (int j=0; j<this->CurrentModeXLength; j++)
            {
      pDestBuffer[Position1]=pSourceBuffer[Position2];
      Position1++;
            Position2++;
            }

   Position1=Position1-Increment;
   }


return(true);

}

BOOL VideoCapture::GetVideoSample(BYTE* pSource)
{
int Size;
int Position1;
int Position2;
```

```
int* pSourceBuffer;
int* pDestBuffer;

pSourceBuffer = (int*)pSource;
pDestBuffer = (int*)pVideoBuffer;

/*Just copy the current bitmap from pTempBuffer in GData as
    this will contain all the data
including overlays onto the current frame bitmap */

memcpy((void*)pDestBuffer,(void*)pSourceBuffer,this->
    CurrentModeXLength*this->CurrentModeYLength*4);

//  if ((pGrabber->GetCurrentBuffer(&BufferSize,(long*)
    pTempBuffer))!=S_OK)
//  {
//
//  MessageBox(NULL,"Could not Grab Sample from image stream
    ","Fatal Error",MB_OK);
//
//  PostQuitMessage(0);
//  return(false);
//  }
//
//Position1 = CurrentModeXLength*CurrentModeYLength-1-
    CurrentModeXLength;
//Position2 = 0;
//
//int Increment = 2*CurrentModeXLength;
//
//  for (int i=0; i<this->CurrentModeYLength; i++)
//  {
//     //Copy a row from reversed buffer to corrected buffer
//     for (int j=0; j<this->CurrentModeXLength; j++)
//        {
//     pDestBuffer[Position1]=pSourceBuffer[Position2];
//     Position1++;
//        Position2++;
//        }
//
//  Position1=Position1-Increment;
//  }


return(true);

}
```

B.8 Stiper.cpp

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

```
#include "Striper.h"

Striper::Striper(int VideoXLength, int VideoYLength, int
    XThreshold,int SampleLength,int MaxXObjects,int
    MaxTotalObjects)
{

this->VideoXLength = VideoXLength;
this->VideoYLength = VideoYLength;
this->XThreshold = XThreshold;
this->SampleLength = SampleLength;
this->MaxXObjects = MaxXObjects;
this->MaxTotalObjects = MaxTotalObjects;

PossibleObjects = new PossibleObject[MaxXObjects*VideoYLength
    ];
ObjectsFound = new ObjectFound[MaxTotalObjects];
this->PossibleObjectsFound = new int[VideoYLength];

}



//pBuffer is sent just so this function can draw onto the
    screen
void Striper::DetectObjects(BYTE *pBuffer)
{

int POBaseIndex = 0; //Base Index for PossibleObjects Array.
    This is a two dimensional array
                                              //so this index
    always points to the start of the current row

int BaseBufferIndex = 0;

ObjectsDetected = 0;

//Clean out Object Array of any previously calculated data
memset((void*) ObjectsFound,0,sizeof(ObjectFound)*
    MaxTotalObjects);



  for (int y=0; y<VideoYLength-2; y++)
  {

    //Loop through all the strips found on this row
    for (int l=0; l<PossibleObjectsFound[y]; l++)
        {

          //If the strip is not allocated to an object attempt
    to allocate it
          //A member of -1 means not part of a group
      if (PossibleObjects[POBaseIndex+l].MemberNo == -1)
          {

      PossibleObjects[POBaseIndex+l].MemberNo =
   FindObjectFromStrip(ObjectsDetected,POBaseIndex,l);
```

```
            if (PossibleObjects[POBaseIndex+l].MemberNo > -1)
   ObjectsDetected++;


         }

      //If we have found the maximum number of objects
   terminate the loop
      if (ObjectsDetected>=MaxTotalObjects)
            break;

       }

    BaseBufferIndex+=4*VideoXLength; //Point to next row
        POBaseIndex+=MaxXObjects; //Point to next row in 2D
   array

    //If we have found the maximum number of objects,
   terminate the loop.
    if (ObjectsDetected>=MaxTotalObjects)
            break;
  }



int y = 0;
int length = 0;
int BaseIndex = 0;
int Index = 0;

BYTE ColourA = 0;
BYTE ColourB = 0;
BYTE ColourC = 0;


  for (int o=0; o<ObjectsDetected; o++)
  {

  ColourA = rand()/128;
  ColourB = rand()/128;
  ColourC = rand()/128;

    for (int i=0; i<this->ObjectsFound[o].NumberOfStrips; i++)
        {

    y = PossibleObjects[ObjectsFound[o].StripIndexes[i]].y;
    length = PossibleObjects[ObjectsFound[o].StripIndexes[i]].
   xend -
                      PossibleObjects[ObjectsFound[o].
   StripIndexes[i]].xstart + 1;
    BaseIndex = y*VideoXLength*4;
    Index = PossibleObjects[ObjectsFound[o].StripIndexes[i]].
   xstart;


      //Draw the detected strip
      for (int s=0; s<length; s++)
          {
```

```
pBuffer[BaseIndex+(Index+s)*4+0] = ColourA;
        pBuffer[BaseIndex+(Index+s)*4+1] = ColourB;
        pBuffer[BaseIndex+(Index+s)*4+2] = ColourC;

        }

      }

  }

}



//After finding a strip not allocated to an object this
   function is called to find any other
//strips on succeeding rows that can be considered part of the
    same object
//The function returns 0 if successful and -1 if this strip
   had not enough other strips associated with it
//or for some other reason was deemed not to be part of an
   object
int Striper::FindObjectFromStrip(int ObjectNumber,int
   StripBaseIndex,int StripIndex)
{
BOOL ContinueLooping = TRUE;
int POBaseIndex1;
int POIndex1; //Index to current strip being processed
int POBaseIndex2;
int POIndex2; //Index to strips below current strip
int NextPOBaseIndex; //When a new strip is found it becomes
   the next strip of focus and other strips
int NextPOIndex;     //are searched for that lie under this
   new strip.  This new strip index is stored in these
                                          //variables and is
   transferred to POBaseIndex1 and POIndex1 on the next loop
   iteration.

int StripsFound = 1; //Number of strips constituting this
   current object
int MissingRows = 0; //Counts how many rows there are without
   finding any strips beneath the current one
                                          //if it gets to large
    the program gives up and therefore the current strip is
   not part
                                     //of a valid object.

int StripsFoundOnRow = 0;

int CurrentStripStartX;
int CurrentStripEndX;
int CurrentStripRow;
int NextCurrentStripRow;
int NextStripStartX;
int NextStripEndX;

int Temp;
```

```
int StripsInRow1; //Records the number of strips in the Row of
     the currently processed strip.
int StripsInRow2; //Records number of strips in the row that
    is being searched

POBaseIndex1 = StripBaseIndex; //Index to the row of 2D array
    PossibleObjects which contains the first
                                                            //
    strip for the object

POBaseIndex2 = StripBaseIndex+MaxXObjects; //Point to row
    below current strips row to find other connected strips
POIndex1 = StripIndex;
POIndex2 = 0;


//Add the first Strip to the Strip list in the object array
ObjectsFound[ObjectNumber].StripIndexes[StripsFound-1] =
    POBaseIndex1 + POIndex1;
ObjectsFound[ObjectNumber].NumberOfStrips=1;

//Record inside the strip itself what group it belongs to
PossibleObjects[POBaseIndex1+POIndex1].MemberNo = ObjectNumber
    ;

CurrentStripStartX = PossibleObjects[StripBaseIndex+StripIndex
    ].xstart;
CurrentStripEndX = PossibleObjects[StripBaseIndex+StripIndex].
    xend;
CurrentStripRow = PossibleObjects[StripBaseIndex+StripIndex].y
    ;
NextCurrentStripRow = CurrentStripRow;



//Loop through remaining rows trying to add new strips to the
    object
for (int y=PossibleObjects[StripBaseIndex+StripIndex].y; y<
    VideoYLength; y++)
{


StripsInRow1 = PossibleObjectsFound[CurrentStripRow];
StripsInRow2 = PossibleObjectsFound[y+1];
StripsFoundOnRow = 0; //Reset the row strip counter



  //search the strips of the row beneath for any that lie
    beneath the current strip
  for (int l=0; l<StripsInRow2; l++)
  {


        if ( ((CurrentStripStartX>=PossibleObjects[
    POBaseIndex2+POIndex2].xstart &&
                 CurrentStripStartX<=PossibleObjects[
    POBaseIndex2+POIndex2].xend) ||
                 (PossibleObjects[POBaseIndex2+POIndex2].
```

```
xstart>=CurrentStripStartX &&
               PossibleObjects[POBaseIndex2+POIndex2].
xstart<=CurrentStripEndX)) &&
               (PossibleObjects[POBaseIndex2+POIndex2].
MemberNo < 0))
     {
 StripsFoundOnRow++;
 MissingRows = 0;

       //if first new strip this strip will become the
current strip on next loop iteration
       if (StripsFoundOnRow == 1)
       {
   NextPOBaseIndex = POBaseIndex2;
       NextPOIndex = POIndex2;

       NextStripStartX = PossibleObjects[POBaseIndex2+
POIndex2].xstart;
       NextStripEndX = PossibleObjects[POBaseIndex2+
POIndex2].xend;

       if (CurrentStripStartX > NextStripStartX+5 &&
NextStripStartX+5<639)
               NextStripStartX = CurrentStripStartX - 2;
//Only allow 5 pixels behind last strip

        if (CurrentStripStartX+5 < NextStripStartX)
               NextStripStartX = CurrentStripStartX + 2;
//Only allow 3 pixels in front of last strip

       NextCurrentStripRow = PossibleObjects[POBaseIndex2+
POIndex2].y;
       }
       else if (StripsFoundOnRow>1) //Increase or decrease
the next strip length depending on whether we find any more
 strips
       {
     if (NextStripStartX>PossibleObjects[POBaseIndex2+
POIndex2].xstart)
                   NextStripStartX = PossibleObjects[
POBaseIndex2+POIndex2].xstart;

    if (CurrentStripStartX > NextStripStartX+5 &&
NextStripStartX+5<639)
               NextStripStartX = CurrentStripStartX - 2;
//Only allow 5 pixels behind last strip

        if (CurrentStripStartX+5 < NextStripStartX)
               NextStripStartX = CurrentStripStartX + 2;
//Only allow 3 pixels in front of last strip


             if (NextStripEndX<PossibleObjects[POBaseIndex2
+POIndex2].xend)
                NextStripEndX = PossibleObjects[
POBaseIndex2+POIndex2].xend;
       }
```

```
        //Record inside the strip itself what group it belongs
   to
   PossibleObjects[POBaseIndex2+POIndex2].MemberNo =
  ObjectNumber;


          if (ObjectsFound[ObjectNumber].NumberOfStrips<
  ObjectsFound[ObjectNumber].MaxStripsPerObject)
          {
          //Record the Indexes of this strip in the current
  object structure
     ObjectsFound[ObjectNumber].StripIndexes[ObjectsFound[
  ObjectNumber].NumberOfStrips] = POBaseIndex2 + POIndex2;

          ObjectsFound[ObjectNumber].NumberOfStrips++;
          }
          else
          {
          //MessageBox(NULL,"Too many strips to store in given
   Strip Array","Error",MB_OK);
          }




        }


        //Check if the remaining strips are too far over to
  the left. If so terminate loop and start on next row
   if (PossibleObjects[POBaseIndex2+POIndex2].xstart>
  CurrentStripEndX)
   {
   //l=StripsInRow2; //Terninate Loop
        //break;
        }

  POIndex2++; //Point to next strip

  } //End of for loop


StripsFound = StripsFound + StripsFoundOnRow;



if (StripsFoundOnRow == 0)
{
MissingRows++;
NextStripStartX = CurrentStripStartX;
NextStripEndX = CurrentStripEndX;
NextCurrentStripRow = CurrentStripRow;
}



//If there are too many missing rows, i.e. rows where there
   are no strips directly beneath the current one
```

```
//then terminate the search and report if the current object
    has enough rows or not
if (MissingRows > 2 || (y+1)>=(VideoYLength-1) )
{

  //If less than 8 strips found not a valid object
  if (StripsFound < 8)
  {
  //Remove all object associations in the PossibleObjects
    array

     for (int i=0; i<StripsFound; i++)
          {
      PossibleObjects[this->ObjectsFound[ObjectNumber].
    StripIndexes[i]].MemberNo = -1;

          }

  ObjectsFound[ObjectNumber].NumberOfStrips = 0;

  //delete any entries within the current object structure and
     return (-1)
  // **** could make this more efficient at a later date ****
  memset((void*)&ObjectsFound[ObjectNumber],0,sizeof(
    ObjectFound));

  return(-1);
  }
  else //Else more than 3 strips found and it is a valid
    object so just return the object number
  {
  ObjectsFound[ObjectNumber].IsValidObject = 0;
  return(ObjectNumber);
  }

}


POBaseIndex1 = NextPOBaseIndex;
POIndex1 = NextPOIndex;
POBaseIndex2 = POBaseIndex2+MaxXObjects; //Prepare to search
    the next row down
POIndex2 = 0;

CurrentStripStartX = NextStripStartX;
CurrentStripEndX = NextStripEndX;
CurrentStripRow = NextCurrentStripRow;

} //End of while loop


//should not reach this point so return no object found
return(-1);
}




/*Detects where blocks of foreground pixels are in each row
    and records how long they are.
  This is then passed onto higher level processing to
```

```
    determine the boundary around the object */

void Striper::DetectPossibleObjects(BYTE* pBuffer)
{
unsigned int BufferPos;
unsigned int RunningSum;

BufferPos = 0;

int y=0;
int x=0;
unsigned int ObjectsFoundOnRow=0;
unsigned int POBaseIndex = 0;

int ColourA; //Used for drawing the strips once found
int ColourB;
int ColourC;

int Temp;

 ColourA = rand()/128;
 ColourB = rand()/128;
 ColourC = rand()/128;

BOOL IsPossibleObjectFound=FALSE; //Set to true when pixel
    density is sufficient to suggest existence of object



//Clear the arrays we will use
memset((void*)this->PossibleObjects,0,sizeof(PossibleObjects)*
    MaxXObjects*VideoYLength);
memset((void*)this->PossibleObjectsFound,0,sizeof(UINT)*
    VideoYLength);


  for (y=0; y<VideoYLength; y++)
  {
  RunningSum = 0 ;
  ObjectsFoundOnRow = 0;
  IsPossibleObjectFound = FALSE;


    for (x=0; x<this->SampleLength; x++)
        {
     //Add the pixels up to the first sample size
       if (pBuffer[BufferPos]==255) RunningSum = RunningSum +
    1;

        BufferPos+=4;
        }


    for (x=x; x<VideoXLength; x++)
    {

           //First check to see if the Running Sum is above or
    equal to the threshold
       //If it is then we have a possible object
           if (RunningSum >= XThreshold &&
```

```
IsPossibleObjectFound == FALSE
                && (ObjectsFoundOnRow<MaxXObjects) )
        {

        //First check if this strip is opening less than 5
pixels from the end of previous strip
        //If this is the case just reopen the previous one
and don't create a new strip entry
    if (ObjectsFoundOnRow>0 && ((x-SampleLength)<=
PossibleObjects[POBaseIndex+ObjectsFoundOnRow-1].xend+5) )
        {
        IsPossibleObjectFound = TRUE; //Reopen old strip
   Temp = y*VideoXLength*4; //Point to start of row

          //Fill in the colour up to current posititinon
          for (int k=PossibleObjects[POBaseIndex+
ObjectsFoundOnRow-1].xend;
                        k<=x; k++)
               {
      //pBuffer[Temp+k*4] = ColourA;
          //pBuffer[Temp+k*4+1] = ColourB;
               //pBuffer[Temp+k*4+2] = ColourC;

               }

        }
        else
        {
    //Start a new strip with new colour
    ColourA = rand()/128;
    ColourB = rand()/128;
    ColourC = rand()/128;

      IsPossibleObjectFound = TRUE;
      //Now record the start pixel of this block
          PossibleObjects[POBaseIndex+ObjectsFoundOnRow].
xstart = x-SampleLength;
      PossibleObjects[POBaseIndex+ObjectsFoundOnRow].y = y;
          ObjectsFoundOnRow++;

          PossibleObjectsFound[y] = ObjectsFoundOnRow; //
record the number of objects found so far

      //Colour in the initial detected pixels
            for (int c=0; c<SampleLength; c++)
                 {
      //   pBuffer[BufferPos-(SampleLength*4)+c*4] = ColourA;
//         pBuffer[BufferPos-(SampleLength*4)+c*4+1] =
ColourB;
//                pBuffer[BufferPos-(SampleLength*4)+c*4+2] =
ColourC;
                 }
        }

        }

    //Has the pixel density fallen away after 5 or so pixels
 - therefore we have gone passed potential object
    if (RunningSum <= XThreshold && IsPossibleObjectFound ==
```

```
TRUE && x > (PossibleObjects[POBaseIndex+ObjectsFoundOnRow
-1].xstart+SampleLength+SampleLength) )
        {

      this->PossibleObjects[POBaseIndex+ObjectsFoundOnRow
-1].xend = x;
      PossibleObjects[POBaseIndex+ObjectsFoundOnRow-1].
MemberNo = -1;
          IsPossibleObjectFound = FALSE;

        }

        //Cover eventuality of coming to end of row.  End
any currently processed block of pixels in this row
        if ( (x == (VideoXLength-1)) &&
IsPossibleObjectFound == TRUE )
        {
    PossibleObjects[POBaseIndex+ObjectsFoundOnRow-1].xend =
x;
        PossibleObjects[POBaseIndex+ObjectsFoundOnRow-1].
MemberNo = -1;
        IsPossibleObjectFound = FALSE;

        }

  //Draw some markers into the buffer for display on the
screen to visualise the function working
      if (IsPossibleObjectFound == TRUE)
      {
//  pBuffer[BufferPos] = ColourA;
//      pBuffer[BufferPos+1] = ColourB;
      //pBuffer[BufferPos+2] = ColourC;
      }

  //Continue to calculate running sum
  if ( (pBuffer[BufferPos-SampleLength*4] == 255) && (
RunningSum > 0) )
              RunningSum = RunningSum - 1;

      if ( (pBuffer[BufferPos+4] == 255) && (RunningSum<
SampleLength))
              RunningSum = RunningSum + 1;
      else if ( (pBuffer[BufferPos+4] == 0) && (RunningSum
>0))
              RunningSum = RunningSum - 1;

  BufferPos+=4;

      } //End of x loop

  POBaseIndex+=MaxXObjects; //Point to next row in 2D array
  } //End of y loop


} //end of function
```

```
//void Striper::StripTidy()
//{

  //Loop through all rows
//   for (int y=0; y<VideoYLength; y++)
//   {

     //loop through each strip found so far on every row
     //for (int x=0; x<PossibleObjectsFound[y]-1; x++)
         //{

            //If two strips are close together merge them by
   giving them the same start and end
//        if ( (PossibleObjects[x].xend <= PossibleObjects[x+1].
   xstart) &&
//                    ( ((PossibleObjects[x+1].xstart -
   PossibleObjects[x].xend) < 4)) )
//         {
       //PossibleObjects[x+1].xstart = PossibleObjects[x].
   xstart;
            //PossibleObjects[x].xend = PossibleObjects[x+1].
   xend;

            //}

         //}

  //}



//}





//Once objects have been detected by DetectObjects(..)
   function this function is then called
//to fill in the bounding box details of each object in its
   ObjectFound structure.
void Striper::GetBoundingSquares(BYTE* pBackground,BYTE*
   pCurrentFrame)
{
int MinX = -1;
int MinY = -1;

int MaxX = -1; //X2 and Y2 lower right hand corner
int MaxY = -1;
int MaskIndex = 0;



  for (int i=0; i<this->ObjectsDetected; i++)
  {
  MinX = -1;
  MinY = -1;
  MaxX = -1;
  MaxY = -1;
```

```
 //Loop through all the strips of this object
 for (int l=0; l<this->ObjectsFound[i].NumberOfStrips; l++)
     {

         //Handle the MinX Calculation
         //If not assigned yet just assign the first value
         if (MinX == -1) MinX = PossibleObjects[ObjectsFound[
i].StripIndexes[l]].xstart;
         else if (PossibleObjects[this->ObjectsFound[i].
StripIndexes[l]].xstart<MinX)
                 MinX = MinX - (MinX-PossibleObjects[this->
ObjectsFound[i].StripIndexes[l]].xstart)/2;

         //Handle the MaxX Calculation
         if (MaxX == -1) MaxX = PossibleObjects[ObjectsFound[
i].StripIndexes[l]].xend;
         else if (PossibleObjects[this->ObjectsFound[i].
StripIndexes[l]].xend>MaxX)
                 MaxX = MaxX + (PossibleObjects[this->
ObjectsFound[i].StripIndexes[l]].xend-MaxX)/2;

    //Handle the MinY calculation
         //If not initialised just grab the first value -
this is guaranteed to be the top most strip
         //so therefore the final MinY value
         PossibleObjects[ObjectsFound[i].StripIndexes[l+1]].y
=PossibleObjects[ObjectsFound[i].StripIndexes[l+1]].y;
         PossibleObjects[ObjectsFound[i].StripIndexes[l-1]].y
=PossibleObjects[ObjectsFound[i].StripIndexes[l-1]].y;
         if (MinY == -1) MinY = PossibleObjects[ObjectsFound[
i].StripIndexes[l]].y-1;


    //Handle the MaxY calculation
    //this will increase as more indexes are found so just
assign to the current y value
        MaxY = PossibleObjects[this->ObjectsFound[i].
StripIndexes[l]].y+1;

     }


//Assign data to the appropriate object structure
ObjectsFound[i].X = MinX;
ObjectsFound[i].Y = MinY;
ObjectsFound[i].Width = (MaxX-MinX);
ObjectsFound[i].Height = (MaxY-MinY);

ObjectsFound[i].Angle = 0; //A rectangle oriented upwards
 parallel to screen y-axis
ObjectsFound[i].CentreX = ObjectsFound[i].X+ObjectsFound[i].
 Width/2;
ObjectsFound[i].CentreY = ObjectsFound[i].Y+ObjectsFound[i].
 Height/2;


  //report error if function doesn't work properly
```

```
    if ( MaxX < 0 || MinX < 0  || MaxY < 0 || MinY < 0 )
                MessageBox(NULL,"Calculating the Bounding box 
   failed","Error",MB_OK);



  }


}








void Striper::DrawBoundingSquares(BYTE* pBuffer)
{
int BufferPos=0;
int ColourA = 0;
int ColourB = 0;
int ColourC = 0;

  //Loop through all the objects found and draw each one
  for (int o=0; o<this->ObjectsDetected; o++)
  {

    if (this->ObjectsFound[o].IsValidObject == -1) continue;
   //Don't draw the invalid regions which may

   be due to noise


  ColourA = rand()/128;
  ColourB = rand()/128;
  ColourC = rand()/128;

  if (ObjectsFound[o].NumberOfStrips < 3)
          MessageBox(NULL,"Trying to draw box around object 
   with no strips","",MB_OK);

    //Draw the top horizontal line
    BufferPos = (ObjectsFound[o].Y*this->VideoXLength+
   ObjectsFound[o].X)*4;

    for (int i=0; i<this->ObjectsFound[o].Width; i++)
        {
    pBuffer[BufferPos]    = ColourA;
        pBuffer[BufferPos+1] = ColourB;
        pBuffer[BufferPos+2] = ColourC;

        BufferPos+=4; //Point to next byte
        }

    //Draw the right side vertical line
    for (int i=0; i<this->ObjectsFound[o].Height; i++)
```

```
{
pBuffer[BufferPos]     = ColourA;
pBuffer[BufferPos+1] = ColourB;
pBuffer[BufferPos+2] = ColourC;

BufferPos+=VideoXLength*4; //Point to next byte in
next row
}

//Draw the bottom line
for (int i=0; i<this->ObjectsFound[o].Width; i++)
{
pBuffer[BufferPos]     = ColourA;
pBuffer[BufferPos+1] = ColourB;
pBuffer[BufferPos+2] = ColourC;

BufferPos-=4; //Point to previous byte or head
leftwards
}

//Draw the left vertical line
for (int i=0; i<this->ObjectsFound[o].Height; i++)
{
pBuffer[BufferPos]     = ColourA;
pBuffer[BufferPos+1] = ColourB;
pBuffer[BufferPos+2] = ColourC;


BufferPos-=4*VideoXLength; //Point to previous byte on
higher row or head upwards
}

//Now draw the strips of the object in the same colour

int y=0;
int BaseIndex = 0;
int Index = 0;
int length = 0;

for (int i=0; i<this->ObjectsFound[o].NumberOfStrips; i++)
{

y = PossibleObjects[ObjectsFound[o].StripIndexes[i]].y;
length = PossibleObjects[ObjectsFound[o].StripIndexes[i]].
xend -
                PossibleObjects[ObjectsFound[o].
StripIndexes[i]].xstart + 1;
BaseIndex = y*VideoXLength*4;
Index = PossibleObjects[ObjectsFound[o].StripIndexes[i]].
xstart;


//Draw the detected strip
for (int s=0; s<length; s++)
{

//pBuffer[BaseIndex+(Index+s)*4+0] = ColourA;
//pBuffer[BaseIndex+(Index+s)*4+1] = ColourB;
```

```
            //pBuffer[BaseIndex+(Index+s)*4+2] = ColourC;
            }

        }


    }


}







//The spectral filter determines whether an object is actually
    a new trackable object or caused by vibration
//of the camera.  If the camera vibrates the reasoning is that
    the objects detected due to this noise
//will not have significant changes in their spectral or
   colour map over the area of the detected object
//Returns -1 if no change and +1 if a significant change. This
    function should also immediately update the
//background when an invalid object is detected
int Striper::ObjectSpectralFilter(int ObjectIndex,BYTE*
   pBackground,BYTE* pCurrentFrame)
{

int BSpectrumRed[256];
int BSpectrumGreen[256];
int BSpectrumBlue[256];

int CSpectrumRed[256];
int CSpectrumGreen[256];
int CSpectrumBlue[256];


int InitialPixelCountBlue = 0;
int InitialPixelCountGreen = 0;
int InitialPixelCountRed = 0;


int BufferPos=0;

memset((void*)&BSpectrumRed,0,sizeof(BSpectrumRed));
memset((void*)&BSpectrumGreen,0,sizeof(BSpectrumGreen));
memset((void*)&BSpectrumBlue,0,sizeof(BSpectrumBlue));

memset((void*)&CSpectrumRed,0,sizeof(BSpectrumRed));
memset((void*)&CSpectrumGreen,0,sizeof(BSpectrumGreen));
memset((void*)&CSpectrumBlue,0,sizeof(BSpectrumBlue));


//Create the colour histograms for the background and current
   image frame
BufferPos = ((ObjectsFound[ObjectIndex].Y*this->VideoXLength)+
   ObjectsFound[ObjectIndex].X)*4;
```

```
  for (int y=0; y<this->ObjectsFound[ObjectIndex].Height; y++)
  {

    for (int x=0; x<this->ObjectsFound[ObjectIndex].Width; x
   ++)
          {

          BSpectrumBlue[pBackground[BufferPos]]++;
          BSpectrumGreen[pBackground[BufferPos+1]]++;
          BSpectrumRed[pBackground[BufferPos+2]]++;

    CSpectrumBlue[pCurrentFrame[BufferPos]]++;
          CSpectrumGreen[pCurrentFrame[BufferPos+1]]++;
          CSpectrumRed[pCurrentFrame[BufferPos+2]]++;

    BufferPos+=4; //Point to next pixel data
          }

  //Point to the next row
  BufferPos+=(VideoXLength-ObjectsFound[ObjectIndex].Width)*4;

  }

  for (int i=0; i<256; i++)
  {
  InitialPixelCountBlue+=BSpectrumBlue[i];
  InitialPixelCountGreen+=BSpectrumGreen[i];
  InitialPixelCountRed+=BSpectrumRed[i];
  }


//Compare the colour histograms by doing a relatively simple
    subtraction
int StartingPos=0;


  for (int i=0; i<256; i++)
  {

    if (BSpectrumBlue[i]>0)
          {

      //if ((i-10)<0)
            //{
      //StartingPos = 0;
            //}
            //else
                  StartingPos = i;

      for (int l=StartingPos; l<(StartingPos+1); l++)
            {
              if (l>255) break;

              if (CSpectrumBlue[l]>0)
                    {
```

```
            if (CSpectrumBlue[l]>=BSpectrumBlue[i])
                    {
                    CSpectrumBlue[l] = CSpectrumBlue[l]-
    BSpectrumBlue[i];
                    BSpectrumBlue[i] = 0;
                    break; //break from loop as we have
    subtracted everything we can
                    }
                    else
                    {
            BSpectrumBlue[i] = BSpectrumBlue[i]-CSpectrumBlue[l
    ];
                    CSpectrumBlue[l] = 0;
                    }

                  }

            }

          } //end of if (BSpectrumBlue[i]>0)
  }

//Do the processing for green histogram

  for (int i=0; i<256; i++)
  {

    if (BSpectrumGreen[i]>0)
          {

      //if ((i-10)<0)
            //{
      //StartingPos = 0;
            //}
            //else
                  StartingPos = i;

      for (int l=StartingPos; l<(StartingPos+1); l++)
            {
      if (l>255) break;

              if (CSpectrumGreen[l]>0)
                  {
            if (CSpectrumGreen[l]>=BSpectrumGreen[i])
                    {
                    CSpectrumGreen[l] = CSpectrumGreen[l]-
    BSpectrumGreen[i];
                    BSpectrumGreen[i] = 0;
                    break; //break from loop as we have
    subtracted everything we can
                    }
                    else
                    {
            BSpectrumGreen[i] = BSpectrumGreen[i]-CSpectrumGreen
    [l];
                    CSpectrumGreen[l] = 0;
                    }
```

```
                }
            }
        } //end of if (BSpectrumGreen[i]>0)
  }


//Do the processing for Red histogram
  for (int i=0; i<256; i++)
  {
     if (BSpectrumRed[i]>0)
        {
      //if ((i-10)<0)
          //{
      //StartingPos = 0;
          //}
          //else
              StartingPos = i;

      for (int l=StartingPos; l<(StartingPos+1); l++)
          {
             if (l>255) break;

             if (CSpectrumRed[l]>0)
                {
          if (CSpectrumRed[l]>=BSpectrumRed[i])
                  {
                  CSpectrumRed[l] = CSpectrumRed[l]-
  BSpectrumRed[i];
                  BSpectrumRed[i] = 0;
                  break; //break from loop as we have
  subtracted everything we can
                  }
                  else
                  {
          BSpectrumRed[i] = BSpectrumRed[i]-CSpectrumRed[l];
                  CSpectrumRed[l] = 0;
                  }

              }

          }

        } //end of if (BSpectrumRed[i]>0)
  }


//Now work out how many pixels are in the modified histograms
   as a percentage of the initial pixel count
int NewPixelCountBlue = 0;
int NewPixelCountGreen = 0;
int NewPixelCountRed = 0;
double PercentageChangeBlue = 0;
double PercentageChangeGreen = 0;
```

```
double PercentageChangeRed = 0;

//InitialPixelCount = ObjectsFound[ObjectIndex].Width*
    ObjectsFound[ObjectIndex].Height;

  for (int i=0; i<256; i++)
  {
  NewPixelCountBlue+=BSpectrumBlue[i];
  NewPixelCountGreen+=BSpectrumGreen[i];
  NewPixelCountRed+=BSpectrumRed[i];
  }


PercentageChangeBlue = ((double)NewPixelCountBlue)/((double)
   InitialPixelCountBlue);
PercentageChangeGreen = ((double)NewPixelCountGreen)/((double)
   InitialPixelCountGreen);
PercentageChangeRed = ((double)NewPixelCountRed)/((double)
   InitialPixelCountRed);


if (PercentageChangeBlue  > 0.3 ||
        PercentageChangeGreen > 0.3 ||
        PercentageChangeRed   > 0.3 )
{

ObjectsFound[ObjectIndex].IsValidObject = 1;
return(1); //Report a change in the overall spectrum in the
   given objects box
}

//If we get here then there is no significant sprectral change
    and the object was probably just noise.

ObjectsFound[ObjectIndex].IsValidObject = -1; //Mark as not a
   valid object

//update the background region which the false object occupied
BufferPos = 0;

//Point to start of object area
BufferPos = (ObjectsFound[ObjectIndex].Y*VideoXLength+
   ObjectsFound[ObjectIndex].X)*4;

  for (int y=0; y<ObjectsFound[ObjectIndex].Height; y++)
  {
    for (int x=0; x<ObjectsFound[ObjectIndex].Width; x++)
        {
    //pBackground[BufferPos] = pCurrentFrame[BufferPos];
        //pBackground[BufferPos+1] = pCurrentFrame[BufferPos
   +1];
        //pBackground[BufferPos+2] = pCurrentFrame[BufferPos
   +2];

        BufferPos+=4;

        }

  //Point to the next row of object area
  BufferPos+=(VideoXLength-ObjectsFound[ObjectIndex].Width)*4;
```

```
   }



return(-1);
}




BOOL Striper::IsInsideOtherObject(int ObjectIndex)
{
int CornersWithin = 0;

int X1 = ObjectsFound[ObjectIndex].X;
int X2 = ObjectsFound[ObjectIndex].X+ObjectsFound[ObjectIndex
    ].Width;
int Y1 = ObjectsFound[ObjectIndex].Y;
int Y2 = ObjectsFound[ObjectIndex].Y+ObjectsFound[ObjectIndex
    ].Height;

     for (int o=0; o<this->ObjectsDetected; o++)
          {
     CornersWithin = 0;

        if (o == ObjectIndex) continue; //Don't compare the same
      object

        if ( (X1 >= ObjectsFound[o].X && X1 <= ObjectsFound[o].X
    +ObjectsFound[o].Width) &&
                    (Y1 >= ObjectsFound[o].Y && Y1 <=
    ObjectsFound[o].Y+ObjectsFound[o].Height)) CornersWithin++;

             if ( (X2 >= ObjectsFound[o].X && X2 <= ObjectsFound[
    o].X+ObjectsFound[o].Width) &&
                    (Y1 >= ObjectsFound[o].Y && Y1 <=
    ObjectsFound[o].Y+ObjectsFound[o].Height)) CornersWithin++;

             if ( (X1 >= ObjectsFound[o].X && X1 <= ObjectsFound[
    o].X+ObjectsFound[o].Width) &&
                    (Y2 >= ObjectsFound[o].Y && Y2 <=
    ObjectsFound[o].Y+ObjectsFound[o].Height)) CornersWithin++;

             if ( (X2 >= ObjectsFound[o].X && X2 <= ObjectsFound[
    o].X+ObjectsFound[o].Width) &&
                    (Y2 >= ObjectsFound[o].Y && Y2 <=
    ObjectsFound[o].Y+ObjectsFound[o].Height)) CornersWithin++;

          if (CornersWithin > 1) return(TRUE); //This object does
      lie within another

          }

return(FALSE); //did not find this object within any other
}
```

```
Striper::~Striper()
{
delete [] PossibleObjects;
delete [] PossibleObjectsFound;
delete [] ObjectsFound;


}




//Eliminate objects that exist inside other objects
void Striper::EliminateInternalObjects()
{

  for (int o=0; o<ObjectsDetected; o++)
  {
     if (this->IsInsideOtherObject(o) == TRUE) ObjectsFound[o].
   IsValidObject = 0; //Mark as invalid
        else ObjectsFound[o].IsValidObject = 1; //Mark as
   valid as not inside other object
  }

}
```

B.9 SelectionRectangle.cpp

```
#include "VisionX.h"



void SelectionRectangle::DrawSelectionRectangle(Graphics* G)
{
Pen PenObject(Color(255,255,255),1);
G->DrawLine(&PenObject,X,Y,this->X+CurrentWidth,Y);
G->DrawLine(&PenObject,X+CurrentWidth,Y,X+CurrentWidth,Y+
    CurrentHeight);
G->DrawLine(&PenObject,X+CurrentWidth,Y+CurrentHeight,this->X,
    Y+CurrentHeight);
G->DrawLine(&PenObject,X,Y+CurrentHeight,X,Y);
}
void SelectionRectangle::Init(int SWidth,int SHeight)
```

```
{

this->ScreenHeight = SHeight;
this->ScreenWidth = SWidth;

IsSelectionWindowActive=FALSE;
HasSelectionBeenMade=FALSE;

pEnhancedData = NULL;

pEnhancedData = new BYTE[640*480*4];

pDisplayData = NULL;
pDummyBuffer = NULL;
pBuffer = NULL;
pBuffer = new BYTE[640*480*4];

}


SelectionRectangle::~SelectionRectangle()
{
if (pDisplayData!=NULL) delete pDisplayData;
if (pBuffer!=NULL) delete pBuffer;
if (pDummyBuffer!=NULL) delete pDummyBuffer;

if (pEnhancedData!=NULL) delete pEnhancedData;
}





void SelectionRectangle::GrabData(BYTE* Data)
{
int SourceBufferPos;
int DestBufferPos;
int* pDestBuffer;
int* pSourceBuffer;


//Bigger pixels to be 8 times larger than original
CurrentHeight = this->ScreenHeight/8;
CurrentWidth = this->ScreenWidth/8;
//if (CurrentHeight*BigPixelSize>Data->GetHeight())
    CurrentHeight=Data->GetHeight()/4;
//if (CurrentWidth*BigPixelSize>Data->GetWidth()) CurrentWidth
    =Data->GetWidth()/4;


//Allocate the pDisplayData Buffer that will hold the Data to
    output to screen
if (pDisplayData==NULL)
{
//Allocate Blank Bitmap
pDummyBuffer = new BYTE[this->ScreenHeight*this->ScreenWidth
    *4];
memset((void*)pDummyBuffer,0,this->ScreenHeight*this->
```

```
    ScreenWidth*4);

pDisplayData = new Bitmap(this->ScreenWidth, this->ScreenHeight
    , this->ScreenWidth*4, PixelFormat32bppRGB, pDummyBuffer);
}

pDestBuffer = (int*)pBuffer;
pSourceBuffer = (int*)Data;
DestBufferPos = -1;
SourceBufferPos = Y*(this->ScreenWidth)+X-1;

//Grab the data from the source Bitmap

for (int l=0; l<CurrentHeight; l++)
{
  for (int j=0; j<CurrentWidth; j++)
  {
  DestBufferPos++;
  SourceBufferPos++;
  pDestBuffer[DestBufferPos]=pSourceBuffer[SourceBufferPos];
  }
  SourceBufferPos=SourceBufferPos+(this->ScreenWidth)-(
   CurrentWidth);
}

}

//Draws the selected data into a full size screen buffer to
    enlarge the pixels
//DestStride is the width of the Destination buffer in bytes
void SelectionRectangle::DrawData(BYTE* pSourceBuffer, int
    DestStride, BYTE* pDestBuffer, int PixelMagnification)
{
//Draw the pixels on the output window by drawing rectangles 1
     for each pixel
int SourceBufferPos = 0;

  for (int y=0; y<this->CurrentHeight; y++)
  {
    for (int x=0; x<this->CurrentWidth; x++)
         {
     DrawBigPixel(x, y, pSourceBuffer[SourceBufferPos+2],
    pSourceBuffer[SourceBufferPos+1], pSourceBuffer[
    SourceBufferPos+0],
                 pDestBuffer, DestStride, PixelMagnification);

         SourceBufferPos+=4;
         }

  }

}


//Draws an enlarged pixel in a screen/Bitmap buffer
//Stride is the width of the buffer in bytes
//x and y are big pixel coordinates not small pixel
```

```
    coordinates
void SelectionRectangle::DrawBigPixel(int x,int y,BYTE Red,
    BYTE Green, BYTE Blue,BYTE* pDestBuffer,int Stride,int Size
    )
{

int DestBufferPos;

//Calculate starting position for pixel
DestBufferPos = y*Stride*Size+(x*4*Size);

for (int i=0; i<Size; i++)
{
   for (int j=0; j<Size; j++)
   {
   pDestBuffer[DestBufferPos] = Blue;
   pDestBuffer[DestBufferPos+1] = Green;
   pDestBuffer[DestBufferPos+2] = Red;
   DestBufferPos+=4;
   }

DestBufferPos=DestBufferPos+Stride-Size*4;
}

}
```

B.10 ObjectTracker.cpp

```
#include "ObjectTracker.h"
#include <stdio.h>
#include <math.h>



ObjectTracker::ObjectTracker(int MaxTrackableObjects,int
    VideoXLength,int VideoYLength,BYTE* pBackground,BYTE*
    pSubBackground,BYTE* pCurrentFrame)
{
this->VideoXLength = VideoXLength;
this->VideoYLength = VideoYLength;

this->pBackground = pBackground;
this->pSubBackground = pSubBackground;
this->pCurrentFrame = pCurrentFrame;

this->MaxTrackableObjects = MaxTrackableObjects;

this->PreliminaryObjectsFound = 0;

this->VehiclesCounted = 0;

this->DeadTimer = new int[MaxTrackableObjects];
this->TimesSeen = new int[MaxTrackableObjects];
this->PObjectsPresent = new BYTE[MaxTrackableObjects];
this->PreliminaryObjects = new ObjectFound[MaxTrackableObjects
    ];
this->Validated = new BYTE[MaxTrackableObjects];
this->TimeAlive = new int[MaxTrackableObjects];
this->MSamples = new Movement[MaxTrackableObjects];
this->TracksData = new Tracks[Tracks::MaxTracks];
```

```
memset((void*)TracksData,0,Tracks::MaxTracks*sizeof(Tracks));
memset((void*)MSamples,0,MaxTrackableObjects*sizeof(Movement))
    ;
memset((void*)TimeAlive,0,MaxTrackableObjects*4);
memset((void*)Validated,0,MaxTrackableObjects);
memset((void*)TimesSeen,0,MaxTrackableObjects*4);
memset((void*)PObjectsPresent,0,MaxTrackableObjects);

memset((void*)DeadTimer,0,MaxTrackableObjects*4);



pFontFamily = new FontFamily(L"Times New Roman");
pFont = new Font(pFontFamily, 24, FontStyleRegular, UnitPixel)
    ;
pPointF = new PointF(30.0f, 10.0f);
pSolidBrush = new SolidBrush(Color(255, 255, 255, 255));


pPen1 = new Pen(Color(255,0,0),5); //Counter line
pPen2 = new Pen(Color(0,0,255),2); //Speed line brush






}


void ObjectTracker::SetBuffers(BYTE* pBackground,BYTE*
    pSubBackground,BYTE* pCurrentFrame)
{
this->pBackground = pBackground;
this->pSubBackground = pSubBackground;
this->pCurrentFrame = pCurrentFrame;
}









void ObjectTracker::UpdateObjects(int NumberOfUpdateObjects,
    ObjectFound* Objects)
{
int Result;
int Temp;



    //Loop through the objects found by the striper class
    for (int i=0; i<NumberOfUpdateObjects; i++)
```

```
{
  if (Objects[i].IsValidObject == 0) continue; //If not a
valid object don't operate on it

  //Compare this object to pre existing objects in this
class and try to find a match
      Result = this->FindSimilarObject1(&Objects[i]);

    if (Result == -1) //We need to add object
        {
    this->AddObject(&Objects[i]);

    }
        else if (Result>=0) //Found a pre-existing object
so update its details
        {

              if (Validated[Result] == 0) //Only update if
not validated as a proper object yet
              {
            TimesSeen[Result]++;
            DeadTimer[Result] = 0;

             if (TimesSeen[Result] > 5 && IsObjectStill(
Result,0) == -1 ) Validated[Result] = 1; //If seen enough
times from stripper class then a valid object so track
properly

        PreliminaryObjects[Result].CentreX = Objects[i].
CentreX;
            PreliminaryObjects[Result].CentreY = Objects[i].
CentreY;
            PreliminaryObjects[Result].Width = Objects[i].
Width;
            PreliminaryObjects[Result].Height = Objects[i].
Height;



              }


        }

}

CalcAngles();


//Update all the currently tracked objects
for (int i=0; i<this->MaxTrackableObjects; i++)
{
  if ( PObjectsPresent[i] != 1 ) continue; //No entry
present so exit

TimeAlive[i]++;

AddSample(i,PreliminaryObjects[i].Angle,PreliminaryObjects[
```

```
    i].CentreX, PreliminaryObjects[i].CentreY, PreliminaryObjects
    [i].Width, PreliminaryObjects[i].Height); //Record the
    position of the object at this point in time

    //See if object is not moving and remove regardless of
    whether it was validated
    Temp = IsObjectStill(i,1);
                    if  (Temp != -1 && TimeAlive[i]>20) this->
    RemoveObject(i, pBackground, pCurrentFrame);

     if (DeadTimer[i]>5) //Remove any objects that have not
    been seen for a certain amount of time
           {
           this->RemoveObject(i, pBackground, pCurrentFrame);
           }

     if (Validated[i] == 0) continue; //If not a Validated
    object don't track properly just use the stripper update

  //UpdateObjectMovements(i); //Create the filtered versions
    of the X,Y, Width & Height and velocity + acceleration data


  AdvancedMeanShift(i, pSubBackground);
  CalcTranslatedCoordinates(i); //Calculate the rotated
    coordinates
  ScaleObject(i);
  //LocateNewObjectPosition(i, pSubBackground, pCurrentFrame);
    //Locate the object using the most up to date movement info
  CountVehicles(); //Update the counter if any new vechicles
    cross the line

                    if ((double(PreliminaryObjects[i].
    TotalTrackingPixels)/double(PreliminaryObjects[i].
    TotalPixelsFound)) < 0.3) DeadTimer[i]++;
                    else DeadTimer[i] = 0;



    }


}









//Compare this object to others in the PreliminaryObjects
    array and find similar one
//based on bounding window size and and the proximity of the
    corners of the bounding box
int  ObjectTracker::FindSimilarObject1(ObjectFound* PObject)
{

int ClosestObjectIndex = -1; //When function successful will
    hold the
```

```
int ClosestRadius = 50; //The shortest distance from the
    center of the passed object to an object in the
    PreliminaryObjects array

int CurrentRadius;

BOOL ClosestRadiusSet = FALSE;
double CurrentX = PObject->CentreX;
double CurrentY = PObject->CentreY;
double TargetX;
double TargetY;


  //Loop through objects in the PreliminaryObjects Array
        for (int i=0; i<this->MaxTrackableObjects; i++)
        {
        if (this->PObjectsPresent[i] == 0) continue; //No
    object in this slot so continue on

        TargetX = PreliminaryObjects[i].CentreX;
        TargetY = PreliminaryObjects[i].CentreY;

     CurrentRadius = sqrt( (TargetX-CurrentX)*(TargetX-CurrentX
    )+(TargetY-CurrentY)*(TargetY-CurrentY) );
     CurrentRadius -= (PreliminaryObjects[i].Width+
    PreliminaryObjects[i].Height)/2;
     CurrentRadius -= (PObject->Width+PObject->Height)/2;

            if (ClosestRadiusSet == FALSE)
            {
       ClosestRadius = CurrentRadius;
            ClosestObjectIndex = i;
            ClosestRadiusSet = TRUE;
            }

       if (CurrentRadius<ClosestRadius)
            {
       ClosestRadius = CurrentRadius;
            ClosestObjectIndex = i;
            }

        }



  if (ClosestRadius<0) return(ClosestObjectIndex);
  else return(-1); //No object close enough so just return
    none found
}




void ObjectTracker::AddObject(ObjectFound* ObjectToAdd)
{
int TempX;
int YempY;
int TempWidth;
```

```
int TempHeight;
ObjectFound* O;

  //Find a free slot in the object array
  for (int i=0; i<this->MaxTrackableObjects; i++)
  {
        if (this->PObjectsPresent[i] == 0) //Have we found a
   free slot
                {
        //yes we have
                this->PObjectsPresent[i] = 1; //Mark as
   occupied

                PreliminaryObjectsFound++;


                //Copy the new object data into the free slot
      memcpy((void*)&PreliminaryObjects[i],(void*)ObjectToAdd
   ,sizeof(ObjectFound));


           //Clear the ObjectFound structure so all the data
   is zerod - most importantly the tracking data structures
           memset((void*)&PreliminaryObjects[i],0,sizeof(
   ObjectFound));

           //Restore the original bounding box coordinates
           PreliminaryObjects[i].X = ObjectToAdd->X;
        //    PreliminaryObjects[i].PhysicalX = (int)
   ObjectToAdd->X;
           PreliminaryObjects[i].Y = ObjectToAdd->Y;
          // PreliminaryObjects[i].PhysicalY = (int)
   ObjectToAdd->Y;
           PreliminaryObjects[i].Width = ObjectToAdd->Width;
          // PreliminaryObjects[i].PhysicalWidth = (int)
   ObjectToAdd->Width;
      PreliminaryObjects[i].Height = ObjectToAdd->Height;
          //         PreliminaryObjects[i].PhysicalHeight = (int
   )ObjectToAdd->Height;


    PreliminaryObjects[i].PhysicalX = ObjectToAdd->X;
        PreliminaryObjects[i].PhysicalY = ObjectToAdd->Y;
        PreliminaryObjects[i].PhysicalWidth = ObjectToAdd->
   Width;
        PreliminaryObjects[i].PhysicalHeight = ObjectToAdd->
   Height;

        PreliminaryObjects[i].CentreX = (ObjectToAdd->X+
   ObjectToAdd->X+ObjectToAdd->Width )/2;
        PreliminaryObjects[i].CentreY = (ObjectToAdd->Y+
   ObjectToAdd->Y+ObjectToAdd->Height)/2;

    PreliminaryObjects[i].IsObjectCounted1 = FALSE; //Object
   is not counted yet
    PreliminaryObjects[i].IsSpeedLineCrossed = FALSE;
        PreliminaryObjects[i].Speed = 0;

   //UpdateObjectMovements(i,ObjectToAdd); //Create the
   filtered versions of the X,Y, Width & Height coordinates
```

```
    //Create the histograms for the 4 sub windows
        O = &PreliminaryObjects[i];

            //CreateHistogram(i,pSubBackground,pCurrentFrame,O->
    Red1,O->Green1,O->Blue1,O->X,O->Y,O->Width,O->Height);
        /*CreateHistogram(i,pSubBackground,pCurrentFrame,O->Red2,
    O->Green2,O->Blue2,O->X+O->Width/2,O->Y,O->Width/2,O->
    Height/2);
        CreateHistogram(i,pSubBackground,pCurrentFrame,O->Red3,O
    ->Green3,O->Blue3,O->X,O->Y+O->Height/2,O->Width/2,O->
    Height/2);
        CreateHistogram(i,pSubBackground,pCurrentFrame,O->Red4,O
    ->Green4,O->Blue4,O->X+O->Width/2,O->Y+O->Height/2,O->Width
    /2,O->Height/2);*/

        if (PreliminaryObjects[i].PhysicalY<0)
    PreliminaryObjects[i].PhysicalY = 0;
        if (PreliminaryObjects[i].PhysicalY>478)
    PreliminaryObjects[i].PhysicalY = 478;
        if (PreliminaryObjects[i].PhysicalY+PreliminaryObjects
    [i].PhysicalHeight>479)
                        PreliminaryObjects[i].PhysicalHeight =
     479 - PreliminaryObjects[i].PhysicalY;

            return;
                }

    }
//If we get here there was no more room and this should be
    reported as an error
//MessageBox(NULL,"Error No more room in Preliminary Objects
    Array","Error",MB_OK);

}


void ObjectTracker::RemoveObject(int ObjectIndex,BYTE*
    pBackground,BYTE* pCurrentFrame)
{
int BufferPos=0;
PreliminaryObjectsFound--;

//Update the background where the current detected object is
    to remove its presence from background subtraction routines
BufferPos = (PreliminaryObjects[ObjectIndex].Y*VideoXLength+
    PreliminaryObjects[ObjectIndex].X)*4;

 //    for (int y=0; y<PreliminaryObjects[ObjectIndex].Height;
    y++)
        //{
 //          for (int x=0; x<PreliminaryObjects[ObjectIndex].
    Width; x++)
        //      {
 //          pBackground[BufferPos] = pCurrentFrame[BufferPos];
        //          pBackground[BufferPos+1] = pCurrentFrame[
    BufferPos+1];
        //          pBackground[BufferPos+2] = pCurrentFrame[
```

```
    BufferPos+2];

          //          BufferPos+=4;
          //          }

 //    BufferPos+=(VideoXLength−PreliminaryObjects[ObjectIndex
    ].Width)*4;
          //}

memset((void*)&PreliminaryObjects[ObjectIndex],0,sizeof(
    ObjectFound));

PreliminaryObjects[ObjectIndex].IsObjectCounted1 = FALSE; //
    Reset the counter flag
PreliminaryObjects[ObjectIndex].IsSpeedLineCrossed = FALSE;

memset((void*)this->MSamples[ObjectIndex].ValidEnries,0,400);
    //Clean up the movement history for next object that comes
    along
MSamples[ObjectIndex].SampleIndex = 0;
MSamples[ObjectIndex].SamplesTaken = 0;

//Remove other important data so another objects does not
    inherit it
this->PObjectsPresent[ObjectIndex] = 0; //Remove the object
    presence status
this->TimeAlive[ObjectIndex] = 0;
this->TimesSeen[ObjectIndex] = 0;
this->Validated[ObjectIndex] = 0;
this->DeadTimer[ObjectIndex] = 0;

}



void ObjectTracker::AddSample(int ObjectIndex,double Angle,int
     CentreX,int CentreY,int Width, int Height) //Add a
    position sample to this objects movement list
{
int CurrentIndex = 0;

//Note: The current Sample index points to the next free slot
    to place a sample not the last sample written!!!


//Store this sample and if we are at the end of buffer reset
    index to 0
CurrentIndex = MSamples[ObjectIndex].SampleIndex;

MSamples[ObjectIndex].AngleSamples[CurrentIndex] = Angle;
MSamples[ObjectIndex].CentreXSamples[CurrentIndex] = CentreX;
MSamples[ObjectIndex].CentreYSamples[CurrentIndex] = CentreY;
MSamples[ObjectIndex].Width[CurrentIndex] = Width;
MSamples[ObjectIndex].Height[CurrentIndex] = Height;

MSamples[ObjectIndex].SamplesTaken++;

MSamples[ObjectIndex].ValidEnries[CurrentIndex] = 1;

if (MSamples[ObjectIndex].SampleIndex == Movement::MaxSamples
    −1)
```

```
            MSamples[ObjectIndex].SampleIndex = 0;
   else
        MSamples[ObjectIndex].SampleIndex++;
}




//If object is not moving returns roughly how long it has been
      still for
//else returns -1 for moving
int  ObjectTracker::IsObjectStill(int ObjectIndex,int
    Validated)
{
int ItemsSearched = 0;
int CurrentIndex = this->MSamples[ObjectIndex].SampleIndex;
int CurrentX;
int CurrentY;
int MovementFound = 0; //Records number of times significant
    movement found

int LastXMovement = 0; //0 == undefined, 1 == to the left, 2==
      to the right
int LastYMovement = 0; //0 == undefined, 1 == to the top, 2 ==
      to the bottom
int DeltaXShift = 0; //Number of X direction changes
int DeltaYShift = 0; //Number of Y direction changes

   if (CurrentIndex != 0)
   {
            CurrentX = MSamples[ObjectIndex].CentreXSamples[
    CurrentIndex-1];
            CurrentY = MSamples[ObjectIndex].CentreYSamples[
    CurrentIndex-1];
   }
   else
   {
            CurrentX = MSamples[ObjectIndex].CentreXSamples[
    Movement::MaxSamples-1];
            CurrentY = MSamples[ObjectIndex].CentreYSamples[
    Movement::MaxSamples-1];
   }

//Search 30 samples back for any sign of movement
     while (ItemsSearched<20)
          {
          if (CurrentIndex == 0)
               {
                          CurrentIndex = Movement::MaxSamples-1;
               }
               else CurrentIndex--;

               if (this->MSamples[ObjectIndex].ValidEnries[
    CurrentIndex] != 1) return(ItemsSearched); //No more valid
    entries to read
```

```
      if ( (CurrentX − MSamples[ObjectIndex].CentreXSamples[
CurrentIndex]) > 1 )
                {
        if (LastXMovement == 1)
                  {
                  DeltaXShift++;
                    if (DeltaXShift>2 && Validated == 0)
return(ItemsSearched); //Changed direction from left to
right
                  }
                  LastXMovement = 2;
                }

                if ( (CurrentX − MSamples[ObjectIndex].
CentreXSamples[CurrentIndex]) < −1)
                {

          if (LastXMovement == 2)
                   {
                   DeltaXShift++;
                            if (DeltaXShift>2 && Validated ==
0)   return(ItemsSearched);
                   }
                   LastXMovement = 1;
                }

                if (CurrentY − MSamples[ObjectIndex].
CentreYSamples[CurrentIndex] > 1)
                {

                        if (LastYMovement == 1)
                        {
                        DeltaYShift++;
                                  if (DeltaYShift>2 && Validated
 == 0) return(ItemsSearched);
                        }
                        LastYMovement = 2;
                }
                if (CurrentY − MSamples[ObjectIndex].
CentreYSamples[CurrentIndex] < −1 )
                {

                  if (LastYMovement == 2)
                  {
                  DeltaYShift++;

                    if (DeltaYShift > 2 && Validated ==  0)
return(ItemsSearched);
                  }
                  LastYMovement = 1;
                }

                if (LastXMovement > 0 || LastYMovement > 0)
     {
                MovementFound++;

                }

        CurrentX = MSamples[ObjectIndex].CentreXSamples[
```

```
CurrentIndex];
        CurrentY = MSamples[ObjectIndex].CentreYSamples[
CurrentIndex];

     if (MovementFound>2 && Validated == 1) return(-1);
          if (MovementFound>4 && Validated == 0) return(-1);

  ItemsSearched++;

       }

return(ItemsSearched);
}

void ObjectTracker::DrawObjectNumbers(Graphics* pGraphics)
{

static wchar_t Number[20];
static size_t BufferSize = 20;

int Speed;

ObjectFound* O;


  for (int o=0; o<this->MaxTrackableObjects; o++)
  {
     if (this->Validated[o] == 0) continue;

  O = &PreliminaryObjects[o];

  //Draw the object number in the top right hand corner of the
     bounding box
  pPointF->X = PreliminaryObjects[o].CentreX -
   PreliminaryObjects[o].Width/2;
  pPointF->Y = PreliminaryObjects[o].CentreY -
   PreliminaryObjects[o].Height/2;

   /*if (pointF.X>639) pointF.X = 639;
       if (pointF.X<0) pointF.X = 0;

       if (pointF.Y>479) pointF.Y = 479;
       if (pointF.Y<0) pointF.Y = 0;*/


   Speed = O->Speed;

   //if (PreliminaryObjects[o].IsObjectCounted1 == TRUE &&
  PreliminaryObjects[o].IsSpeedLineCrossed == TRUE)
               swprintf_s(Number, BufferSize, L"%d, %d  ", o,
  Speed);
               //swprintf_s(Number, BufferSize, L"Test");
       //else
          //   swprintf_s(Number, BufferSize, L"%d", o);
```

```
  pGraphics->DrawString(Number, -1, pFont, *pPointF,
   pSolidBrush);

  }

}



//Determines the angle from point X1,Y1 to X2,Y2 where 0
    Degreen means X2,Y2 is directly above X1,Y1
double ObjectTracker::AngleBetweenPoints(double X1,double Y1,
    double X2,double Y2)
{
double Angle;




  //Work out what quadrant where the velocity vector is
   pointing
        if (X2>X1 && Y2<=Y1)
                {
                //Vector is pointing in top right quadrant
                Angle = atan( (abs(Y1-Y2) / abs(X2-X1))  );
            Angle = (3*3.141593/2)+Angle; //Get the angle with
   0 Degrees a vertical vector pointing up
                }

        if (X2<=X1 && Y2<Y1)
                {
        //Vector pointing into top left quadrant
                 if (X2 == X1) Angle = 0;
                 else
                 {
                 Angle = atan( (abs(Y1-Y2) / abs(X2-X1))  );
                 Angle = (3.141593/2)-Angle;
                 }

                }
                if (Y2>=Y1 && X2<X1)
                {
        //Vector is pointing in the bottom left quadrant
                Angle = atan( (abs(Y1-Y2) / abs(X2-X1))  );
                Angle = (3.141593/2)+Angle;
                }
        if (X2>=X1 && Y2>Y1)
                {
        //Vector is pointing in the bottom right quadrant
          if (X2 == X1) Angle = 3.141593;
                 else
                 {
                 Angle = atan( (abs(Y1-Y2) / abs(X2-X1))  );
                 Angle = (3*3.141593/2)-Angle;
                 }
```

```
                    }

return(Angle);
}




void ObjectTracker::CalcAngles()
{

ObjectFound* O;
double LastCentreX;
double LastCentreY;
double Angle=0;
int CurrentIndex;

     for (int o=0; o<MaxTrackableObjects; o++)
          {
     if (this->PObjectsPresent[o] == 0) continue;    //Only
    process if object stored in this element

    O = &PreliminaryObjects[o];

          Angle = O->Angle; //Adopt the current angle if no new
    angle can be computed

     //Obtain the last two CentreX and CentreY values to
    determine the velocity vector
     CurrentIndex = this->MSamples[o].SampleIndex;

        if (CurrentIndex != 0)
        {
             LastCentreX = MSamples[o].CentreXSamples[
    CurrentIndex-1];
             LastCentreY = MSamples[o].CentreYSamples[
    CurrentIndex-1];
        }
        else
        {
                       LastCentreX = MSamples[o].CentreXSamples[
    Movement::MaxSamples-1];
                       LastCentreY = MSamples[o].CentreYSamples[
    Movement::MaxSamples-1];
        }

     if (LastCentreX == O->CentreX && LastCentreY == O->CentreY
    ) continue; //Don't do any more calcs as object hasn't
    moved

          Angle = AngleBetweenPoints(LastCentreX,LastCentreY,O->
    CentreX,O->CentreY);


     //Add the new vector to the current Angle vector to
    determine a new direction.
          //The new vector should have a smaller weighting it
    has only a small effect on the
          //change in direction each time this function is
```

```
    called

          //First calculate the new X and Y coordinates of the
    vector sum result
          //assume that the Angle vector has magnitude of 1 and
    the new direction vector
          //has a magnitude of 0.1

          double CurrentY;
          double CurrentX;

        if (this->MSamples[o].SamplesTaken<10)
            {
            CurrentY = 10*cos(O->Angle) + 10*cos(Angle);
            CurrentX = -10*sin(O->Angle) - 10*sin(Angle);
            }
            else
            {

            CurrentY = 10*cos(O->Angle) + 3*cos(Angle);
            CurrentX = -10*sin(O->Angle) - 3*sin(Angle);
            }


          //if (this->MSamples[o].SamplesTaken<10) O->Angle =
    Angle;
          O->Angle = AngleBetweenPoints(30,30,30+CurrentX,30-
    CurrentY);


          }



}








void ObjectTracker::CalcTranslatedCoordinates(int ObjectIndex)
{
int RX1,RY1,RX2,RY2,RX3,RY3,RX4,RY4;
ObjectFound* O;




  O = &PreliminaryObjects[ObjectIndex];


  //Find the coordinates of the corners of the bounding box
    relative to its centre
  RX1 = -O->Width/2;
  RY1 = -O->Height/2;

  RX2 = O->Width/2;
```

```
  RY2 = -O->Height/2;

  RX3 = -O->Width/2;
  RY3 = +O->Height/2;

  RX4 = O->Width/2;
  RY4 = +O->Height/2;


  O->TX1 = RX1*cos(O->Angle) - RY1*sin(O->Angle);
  O->TY1 = RY1*cos(O->Angle) + RX1*sin(O->Angle);

  O->TX2 = RX2*cos(O->Angle) - RY2*sin(O->Angle);
  O->TY2 = RY2*cos(O->Angle) + RX2*sin(O->Angle);

  O->TX3 = RX3*cos(O->Angle) - RY3*sin(O->Angle);
  O->TY3 = RY3*cos(O->Angle) + RX3*sin(O->Angle);

  O->TX4 = RX4*cos(O->Angle) - RY4*sin(O->Angle);
  O->TY4 = RY4*cos(O->Angle) + RX4*sin(O->Angle);



  O->TX1 = O->TX1+O->CentreX;
  O->TY1 = O->CentreY-O->TY1;

  O->TX2 = O->TX2+O->CentreX;
  O->TY2 = O->CentreY-O->TY2;

  O->TX3 = O->TX3+O->CentreX;
  O->TY3 = O->CentreY-O->TY3;

  O->TX4 = O->TX4+O->CentreX;
  O->TY4 = O->CentreY-O->TY4;


  double TXArray[10];
  double TYArray[10];
  double TXFinal[5];
  double TYFinal[5];

  TXArray[1] = O->TX1;
  TXArray[2] = O->TX2;
  TXArray[4] = O->TX3;   //Reverse TX3 and 4 to make algorithm
    work
  TXArray[3] = O->TX4;

  TYArray[1] = O->TY1;
  TYArray[2] = O->TY2;
  TYArray[4] = O->TY3;
  TYArray[3] = O->TY4;
TXFinal[1] = TXArray[1];
TXFinal[2] = TXArray[2];
TXFinal[3] = TXArray[3];
TXFinal[4] = TXArray[4];


TYFinal[1] = TYArray[1];
TYFinal[2] = TYArray[2];
TYFinal[3] = TYArray[3];
TYFinal[4] = TYArray[4];
```

```
int XDiff;
int YDiff;

int PreviousIndex = 4;
int NextIndex = 2;       //Used to implement a circular buffer

int TArrayIndex = 0;

double Gradient;

// for (int i=1; i<5; i++)
// {

//   //Check if this vertex point is outside the display area
//   if (TXArray[i] > VideoXLength || TXArray[i] < 0 ||
      //       TYArray[i] > VideoYLength || TYArray[i] < 0 )
      //{
//   //It is, so adjust the coordinate so that it is brought
  inside the display area

//   //Find a pixel at right angles to this one that is
  inside the display area
      //  if (i == 1)
      //  {
      //  PreviousIndex = 4;
      //  NextIndex = 2;
      //  }
      //  else if (i == 4)
      //  {
//      PreviousIndex = 3;
      //  NextIndex = 1;
      //  }
      //  else
      //  {
//      PreviousIndex = i-1;
      //  NextIndex = i+1;
      //  }
      //
//       //Check if the previous pixel is within the display
  area and if it is use it for further calcs
//       if (TXArray[PreviousIndex] < VideoXLength && TXArray
  [PreviousIndex] >= 0 &&
      //             TYArray[PreviousIndex] < VideoYLength
  && TYArray[PreviousIndex] >= 0)
      //       TArrayIndex = PreviousIndex;

//       //else try the next pixel
      //       else if (TXArray[NextIndex] < VideoXLength &&
  TXArray[NextIndex] > 0 &&
      //             TYArray[NextIndex] < VideoYLength &&
  TYArray[NextIndex] > 0)
      //       {
      //       TArrayIndex = NextIndex;
      //       }
```

```
        //        else TArrayIndex = -1;

        //
//        if (TArrayIndex != -1)
        //        {
//        //Find the distance from the vertical and horizontal
    boundary
        //          if (TXArray[i] >= TXArray[TArrayIndex])
        //                  XDiff = VideoXLength-TXArray[
   TArrayIndex];
        //          else
        //              XDiff = TXArray[TArrayIndex];

        //          if (TYArray[i] >= TYArray[TArrayIndex])
        //                  YDiff = VideoYLength-TYArray[
   TArrayIndex];
        //          else
        //              YDiff = TYArray[TArrayIndex];

        //
        //    //Find the gradient between this pixel and pixel
    outside boundary
        //     Gradient = abs( double(TYArray[i]-TYArray[
   TArrayIndex])/(TXArray[i]-TXArray[TArrayIndex]) );

//          if (Gradient*XDiff <= YDiff)
        //        {
//              if (TXArray[i] > TXArray[TArrayIndex])
        //                {
        //                TXFinal[i] = VideoXLength-1;
        //
        //                  if (TYArray[TArrayIndex]>=TYArray[i
   ])
        //                  TYFinal[i] = TYArray[TArrayIndex] -
    Gradient*XDiff;
        //                  else TYFinal[i] = TYArray[
   TArrayIndex] + Gradient*XDiff;
        //                }
        //                else
        //                {
//              TXFinal[i] = 0;
        //
        //                        if (TYArray[TArrayIndex]>=
   TYArray[i])
        //                  TYFinal[i] = TYArray[TArrayIndex] -
    Gradient*XDiff;
        //                  else TYFinal[i] = TYArray[
   TArrayIndex] + Gradient*XDiff;

        //                }

        //        }
//
        //        else
        //        {
//
//              if (TYArray[i] >= TYArray[TArrayIndex])
```

```
        //                  {
//              TYFinal[i] = VideoYLength-1;

//                  if (TXArray[i] >= TXArray[TArrayIndex])
        //                      TXFinal[i] = TXArray[i] + (1/
   Gradient)*YDiff;
        //                      else
        //                        TXFinal[i] = TXArray[i] - (1/
   Gradient)*YDiff;

        //                  }

        //                  else
        //                  {
//              TYFinal[i] = 0;

        //                      if (Gradient != 0)
        //                      {
        //                  if (TXArray[i] >= TXArray[TArrayIndex
   ])
        //                          TXFinal[i] = TXArray[i] + (1/
   Gradient)*YDiff;
        //                          else
        //                            TXFinal[i] = TXArray[i] - (1/
   Gradient)*YDiff;

        //                      }
        //                      else
        //                      {
//                  if (TXArray[i] > VideoXLength) TXArray[i] =
   VideoXLength;
        //                              if (TXArray[i] < 0) TXArray[i]
    = 0;
        //                      }

        //                  }
//

        //              }
        //
        //          }
//          //Find the gradient from this pixel to the pixel
   outside the boundary
//
        //
        //          //If all else fails just set to the
   coordinates to the boundary
        //          else
        //          {
//
        //              //if (TXArray[i]>VideoXLength) TXArray[i] =
   VideoXLength;
        //              //if (TXArray[i]<0) TXArray[i] = 0;
        //              //if (TYArray[i]>VideoYLength) TYArray[i] =
   VideoYLength;
        //              //if (TYArray[i]<0) TYArray[i] = 0;
        //          }

//  }
```

```
//
// } //End of for loop looping through vertices of current
   object bounding box



// O->TX1 = TXFinal[1];
// O->TX2 = TXFinal[2];
// O->TX4 = TXFinal[3];
// O->TX3 = TXFinal[4];
//
// O->TY1 = TYFinal[1];
// O->TY2 = TYFinal[2];
// O->TY4 = TYFinal[3];
// O->TY3 = TYFinal[4];

 if (O->TX1>639) O->TX1 = 639;
         if (O->TX1<0) O->TX1 = 0;

         if (O->TX2>639) O->TX2 = 639;
         if (O->TX2<0) O->TX2 = 0;

         if (O->TX3>639) O->TX3 = 639;
         if (O->TX3<0) O->TX3 = 0;

         if (O->TX4>639) O->TX4 = 639;
         if (O->TX4<0) O->TX4 = 0;

         if (O->TY1>479) O->TY1 = 479;
         if (O->TY1<0) O->TY1 = 0;

         if (O->TY2>479) O->TY2 = 479;
         if (O->TY2<0) O->TY2 = 0;

         if (O->TY3>479) O->TY3 = 479;
         if (O->TY3<0) O->TY3 = 0;

         if (O->TY4>479) O->TY4 = 479;
         if (O->TY4<0) O->TY4 = 0;



}




void ObjectTracker::CalcTranslatedCoordinates()
{
```

```
int RX1,RY1,RX2,RY2,RX3,RY3,RX4,RY4;
ObjectFound* O;

  for (int o=0; o<this->MaxTrackableObjects; o++)
  {
  if (this->PObjectsPresent[o] == 0) continue;

  O = &PreliminaryObjects[o];


  //Find the coordinates of the corners of the bounding box
    relative to its centre
  RX1 = -O->Width/2;
  RY1 = -O->Height/2;

  RX2 = O->Width/2;
  RY2 = -O->Height/2;

  RX3 = -O->Width/2;
  RY3 = +O->Height/2;

  RX4 = O->Width/2;
  RY4 = +O->Height/2;


  O->TX1 = RX1*cos(O->Angle) - RY1*sin(O->Angle);
  O->TY1 = RY1*cos(O->Angle) + RX1*sin(O->Angle);

  O->TX2 = RX2*cos(O->Angle) - RY2*sin(O->Angle);
  O->TY2 = RY2*cos(O->Angle) + RX2*sin(O->Angle);

  O->TX3 = RX3*cos(O->Angle) - RY3*sin(O->Angle);
  O->TY3 = RY3*cos(O->Angle) + RX3*sin(O->Angle);

  O->TX4 = RX4*cos(O->Angle) - RY4*sin(O->Angle);
  O->TY4 = RY4*cos(O->Angle) + RX4*sin(O->Angle);



  O->TX1 = O->TX1+O->CentreX;
  O->TY1 = O->CentreY-O->TY1;

  O->TX2 = O->TX2+O->CentreX;
  O->TY2 = O->CentreY-O->TY2;

  O->TX3 = O->TX3+O->CentreX;
  O->TY3 = O->CentreY-O->TY3;

  O->TX4 = O->TX4+O->CentreX;
  O->TY4 = O->CentreY-O->TY4;




  double TXArray[10];
  double TYArray[10];
  double TXFinal[5];
  double TYFinal[5];
```

```
  TXArray[1] = O->TX1;
  TXArray[2] = O->TX2;
  TXArray[4] = O->TX3;   //Reverse TX3 and 4 to make algorithm
    work
  TXArray[3] = O->TX4;

  TYArray[1] = O->TY1;
  TYArray[2] = O->TY2;
  TYArray[4] = O->TY3;
  TYArray[3] = O->TY4;

TXFinal[1] = TXArray[1];
TXFinal[2] = TXArray[2];
TXFinal[3] = TXArray[3];
TXFinal[4] = TXArray[4];

TYFinal[1] = TYArray[1];
TYFinal[2] = TYArray[2];
TYFinal[3] = TYArray[3];
TYFinal[4] = TYArray[4];


  int XDiff;
  int YDiff;

  int PreviousIndex = 4;
  int NextIndex = 2;      //Used to implement a circular buffer

  int TArrayIndex = 0;

  double Gradient;

 // for (int i=1; i<5; i++)
 // {

 //   //Check if this vertex point is outside the display area
 //   if (TXArray[i] > VideoXLength || TXArray[i] < 0 ||
        //       TYArray[i] > VideoYLength || TYArray[i] < 0 )
        //{
 //   //It is, so adjust the coordinate so that it is brought
    inside the display area

 //   //Find a pixel at right angles to this one that is
    inside the display area
        //  if (i == 1)
        //  {
        //  PreviousIndex = 4;
        //  NextIndex = 2;
        //  }
        //  else if (i == 4)
        //  {
 //     PreviousIndex = 3;
        //  NextIndex = 1;
        //  }
        //  else
        //  {
 //     PreviousIndex = i-1;
```

```
        //   NextIndex = i+1;
        //   }
        //
//          //Check if the previous pixel is within the display
   area and if it is use it for further calcs
//          if (TXArray[PreviousIndex] < VideoXLength && TXArray
   [PreviousIndex] >= 0 &&
        //                    TYArray[PreviousIndex] < VideoYLength
   && TYArray[PreviousIndex] >= 0)
        //          TArrayIndex = PreviousIndex;

//          //else try the next pixel
        //          else if (TXArray[NextIndex] < VideoXLength &&
   TXArray[NextIndex] > 0 &&
        //                    TYArray[NextIndex] < VideoYLength &&
   TYArray[NextIndex] > 0)
        //          {
        //          TArrayIndex = NextIndex;
        //          }
        //          else TArrayIndex = -1;


        //
//          if (TArrayIndex != -1)
        //          {
//          //Find the distance from the vertical and horizontal
    boundary
        //             if (TXArray[i] >= TXArray[TArrayIndex])
        //                      XDiff = VideoXLength-TXArray[
   TArrayIndex];
        //             else
        //                  XDiff = TXArray[TArrayIndex];

        //             if (TYArray[i] >= TYArray[TArrayIndex])
        //                      YDiff = VideoYLength-TYArray[
   TArrayIndex];
        //             else
        //                  YDiff = TYArray[TArrayIndex];

        //
        //      //Find the gradient between this pixel and pixel
    outside boundary
        //      Gradient = abs( double(TYArray[i]-TYArray[
   TArrayIndex])/(TXArray[i]-TXArray[TArrayIndex]) );

//             if (Gradient*XDiff <= YDiff)
        //             {
//                 if (TXArray[i] > TXArray[TArrayIndex])
        //                       {
        //                       TXFinal[i] = VideoXLength-1;
        //
        //                          if (TYArray[TArrayIndex]>=TYArray[i
   ])
        //                          TYFinal[i] = TYArray[TArrayIndex] -
    Gradient*XDiff;
        //                          else TYFinal[i] = TYArray[
   TArrayIndex] + Gradient*XDiff;
        //                       }
```

```
        //                  else
        //                  {
//                TXFinal[i] = 0;
        //
        //                          if (TYArray[TArrayIndex]>=
  TYArray[i])
        //                    TYFinal[i] = TYArray[TArrayIndex] -
   Gradient*XDiff;
        //                    else TYFinal[i] = TYArray[
  TArrayIndex] + Gradient*XDiff;

        //                  }

        //          }
//
        //          else
        //          {
//
//              if (TYArray[i] >= TYArray[TArrayIndex])
        //                 {
//              TYFinal[i] = VideoYLength-1;

//                if (TXArray[i] >= TXArray[TArrayIndex])
        //                     TXFinal[i] = TXArray[i] + (1/
  Gradient)*YDiff;
        //                   else
        //                     TXFinal[i] = TXArray[i] - (1/
  Gradient)*YDiff;

        //                 }

        //                 else
        //                 {
//              TYFinal[i] = 0;

        //                   if (Gradient != 0)
        //                   {
        //                 if (TXArray[i] >= TXArray[TArrayIndex
  ])
        //                       TXFinal[i] = TXArray[i] + (1/
  Gradient)*YDiff;
        //                     else
        //                       TXFinal[i] = TXArray[i] - (1/
  Gradient)*YDiff;

        //                   }
        //                   else
        //                   {
//                  if (TXArray[i] > VideoXLength) TXArray[i] =
  VideoXLength;
        //                          if (TXArray[i] < 0) TXArray[i]
   = 0;
        //                   }

        //                 }
//

        //          }
```

```
        //
        //       }
//        //Find the gradient from this pixel to the pixel
  outside the boundary
//
        //
        //       //If all else fails just set to the
  coordinates to the boundary
        //       else
        //       {
//
        //         //if (TXArray[i]>VideoXLength) TXArray[i] =
  VideoXLength;
        //         //if (TXArray[i]<0) TXArray[i] = 0;
        //         //if (TYArray[i]>VideoYLength) TYArray[i] =
  VideoYLength;
        //         //if (TYArray[i]<0) TYArray[i] = 0;
        //       }

//   }

//

//   } //End of for loop looping through vertices of current
  object bounding box



 /*O->TX1 = TXFinal[1];
 O->TX2 = TXFinal[2];
 O->TX4 = TXFinal[3];
 O->TX3 = TXFinal[4];

 O->TY1 = TYFinal[1];
 O->TY2 = TYFinal[2];
 O->TY4 = TYFinal[3];
 O->TY3 = TYFinal[4];*/

  if (O->TX1>639) O->TX1 = 639;
        if (O->TX1<0) O->TX1 = 0;

        if (O->TX2>639) O->TX2 = 639;
        if (O->TX2<0) O->TX2 = 0;

        if (O->TX3>639) O->TX3 = 639;
        if (O->TX3<0) O->TX3 = 0;

        if (O->TX4>639) O->TX4 = 639;
        if (O->TX4<0) O->TX4 = 0;

        if (O->TY1>479) O->TY1 = 479;
        if (O->TY1<0) O->TY1 = 0;

        if (O->TY2>479) O->TY2 = 479;
        if (O->TY2<0) O->TY2 = 0;

        if (O->TY3>479) O->TY3 = 479;
        if (O->TY3<0) O->TY3 = 0;
```

```
        if (O->TY4>479) O->TY4 = 479;
        if (O->TY4<0) O->TY4 = 0;

        if (O->CentreX>VideoXLength-1) O->CentreX =
   VideoXLength-1;
        if (O->CentreX<0) O->CentreX = 0;
        if (O->CentreY>VideoYLength-1) O->CentreY =
   VideoYLength-1;
        if (O->CentreY<0) O->CentreY = 0;


  } //End of for loop looping through all objects




} //End of function




void ObjectTracker::DrawBoundingSquares(BYTE* pBuffer)
{
int BufferPos=0;
int ColourA = 0;
int ColourB = 0;
int ColourC = 0;

int X1,Y1,X2,Y2,X3,Y3,X4,Y4;

ObjectFound* O;

int VectorLength;
int VectorEndX = 0;
int VectorEndY = 0;

CalcTranslatedCoordinates(); //Calculate the rotated
    coordinates

  //Loop through all the objects found and draw each one
  for (int o=0; o<this->MaxTrackableObjects; o++)
  {

  O = &PreliminaryObjects[o];

    //if (this->PObjectsPresent[o] == 0) continue;
    if (this->Validated[o] == 0) continue;    //Don't draw the
   invalidated regions



    if (O->Width <= O->Height) VectorLength = O->Width;
    else VectorLength = O->Height;

    VectorEndX = 0*cos(O->Angle) - VectorLength*sin(O->Angle);
         VectorEndY = VectorLength*cos(O->Angle) + 0*sin(O->
   Angle);
```

```
     if (TimeAlive[o] < 25)
         {
    ColourA = 255;
    ColourB = 0;
    ColourC = 0;
         }
         else
         {
        ColourA = 255;
    ColourB = 255;
    ColourC = 255;
         }

  //Draw the lines linking the corners of the bounding box
  DrawLine(pBuffer,O->TX1,O->TY1,O->TX2,O->TY2,ColourA,ColourB
    ,ColourC);
  DrawLine(pBuffer,O->TX1,O->TY1,O->TX3,O->TY3,ColourA,ColourB
    ,ColourC);
  DrawLine(pBuffer,O->TX2,O->TY2,O->TX4,O->TY4,ColourA,ColourB
    ,ColourC);
  DrawLine(pBuffer,O->TX3,O->TY3,O->TX4,O->TY4,ColourA,ColourB
    ,ColourC);



  VectorEndX = O->CentreX+VectorEndX;
  VectorEndY = O->CentreY-VectorEndY;

    if (VectorEndX>639) VectorEndX = 639;
        if (VectorEndX<0) VectorEndX = 0;

        if (VectorEndY>479) VectorEndY = 479;
        if (VectorEndY<0) VectorEndY = 0;

  DrawLine(pBuffer,O->CentreX,O->CentreY,VectorEndX,VectorEndY
    ,255,0,0);



  } //End of o loop


}




 //Creates a mask with 1 where any object is positioned so the
     background in those areas is not updated
//by the background model in BackgroundSubtraction class
void ObjectTracker::CreateMask(BYTE* pMask)
{
int MaskIndex = 0;
int Y = 0;
int X = 0;
int Width;
int Height;
int BufferIndex = 0;
```

```
memset((void*)pMask,0,VideoXLength*VideoYLength);

        for (int i=0; i<this->MaxTrackableObjects; i++)
        {

            if (this->Validated[i] != 1) continue; //Not a
    validated object so don't create mask for it

          MaskIndex = PreliminaryObjects[i].PhysicalY*
    VideoXLength+PreliminaryObjects[i].PhysicalX;
          //BufferIndex = ((PreliminaryObjects[i].PhysicalY*
    VideoXLength)+PreliminaryObjects[i].PhysicalX)*4;
                  Width = PreliminaryObjects[i].PhysicalWidth;
                  Height = PreliminaryObjects[i].PhysicalHeight
    ;

                  for (int y=0; y<Height;     y++)
                  {

                         for (int x=0; x<Width;     x++)
                      {
                //                               if (pSubBackground
    [BufferIndex] == 255) pMask[MaskIndex] = 1;
                                                 pMask[MaskIndex] = 1;

                                                  MaskIndex++;
                                                  BufferIndex+=4;
                                                 }

                          MaskIndex+=VideoXLength-Width;
                          //BufferIndex+=(VideoXLength-Width)
    *4;
                          }

            } //End of i loop


}





void ObjectTracker::UpdateObjectMovements(int ObjectIndex)
{

//int Sample1Index;
//int Sample2Index;
//int Velocity1;
//int Velocity2;
//BOOL Sample1Valid = FALSE;
//BOOL Sample2Valid = FALSE;
//double Error;
//
//ObjectFound* ForceObject;
//
//ForceObject = &PreliminaryObjects[ObjectIndex];
//
```

```
//Movement* Samples = &MSamples[ObjectIndex];
//
//
//int CurrentIndex = this->MSamples[ObjectIndex].SampleIndex;
//
// //First try to read back 2 samples from previous positions
   and record how many we could actually read.
////If this object is young it may not have any samples or
   only 1 which means variables like the acceleration can not
   be updated yet
//
////Try and read the last sample stored
//  if (CurrentIndex == 0) //Point to last sample stored
//          CurrentIndex = 399;
//  else CurrentIndex--;
//
//      if (MSamples[ObjectIndex].ValidEnries[CurrentIndex] ==
   1) //Can we read this value? Is it valid?
//        {
//   //Yes its' valid
//   Sample1Valid = TRUE;
//   Sample1Index = CurrentIndex;
//        }
//
////Try and read the 2nd last sample stored
//  if (CurrentIndex == 0) //Point to last sample stored using
   circular buffer index
//          CurrentIndex = 399;
//  else CurrentIndex--;
//
//  if (MSamples[ObjectIndex].ValidEnries[CurrentIndex] == 1)
   //Can we read this value? Is it valid?
//        {
//   //Yes its' valid
//   Sample2Valid = TRUE;
//   Sample2Index = CurrentIndex;
//        }
//
//  if (Sample2Valid == TRUE && Sample1Valid == FALSE) //
   Detect a strange condition that should not happen
//  {
//  MessageBox(NULL,"Sample2 found but not sample 1","Error",
   MB_OK);
//  }
//
//
//
//  if (PreliminaryObjects[ObjectIndex].LLPositionError.
   SamplesAcquired == 0)
//    {
//    PreliminaryObjects[ObjectIndex].PhysicalX = ForceObject->
   X;
//    PreliminaryObjects[ObjectIndex].PhysicalY = ForceObject->
   Y;
//    PreliminaryObjects[ObjectIndex].PhysicalWidth =
   ForceObject->Width;
```

```
//    PreliminaryObjects[ObjectIndex].PhysicalHeight =
    ForceObject->Height;
//    }
//
//
//
//
////Calculate the average positional error between predictions
     and samples
//Error = ForceObject->X;
//Error-=PreliminaryObjects[ObjectIndex].PhysicalX;
//UpdateEstimate(&PreliminaryObjects[ObjectIndex].
    LLPositionError,Error);
//
//
//Error = ForceObject->X+ForceObject->Width;
//Error-=(PreliminaryObjects[ObjectIndex].PhysicalX+
    PreliminaryObjects[ObjectIndex].PhysicalWidth);
//UpdateEstimate(&PreliminaryObjects[ObjectIndex].
    RLPositionError,Error);
//
//
//Error = ForceObject->Y;
//Error-=PreliminaryObjects[ObjectIndex].PhysicalY;
//UpdateEstimate(&PreliminaryObjects[ObjectIndex].
    TLPositionError,Error);
//
//
//Error = (ForceObject->Y+ForceObject->Height);
//Error-=(PreliminaryObjects[ObjectIndex].PhysicalY+
    PreliminaryObjects[ObjectIndex].PhysicalHeight);
//UpdateEstimate(&PreliminaryObjects[ObjectIndex].
    BLPositionError,Error);
//
//
//
////See if we have at least 1 sample so we can estimate
    velocity
//      if (Sample1Valid == TRUE)
//        {
//      UpdateEstimate(&PreliminaryObjects[ObjectIndex].
    LLVelocity,ForceObject->X-Samples->XSamples[Sample1Index]);
//      UpdateEstimate(&PreliminaryObjects[ObjectIndex].
    RLVelocity,(ForceObject->X+ForceObject->Width)-(Samples->
    XSamples[Sample1Index]+Samples->Width[Sample1Index]));
//          UpdateEstimate(&PreliminaryObjects[ObjectIndex].
    TLVelocity,ForceObject->Y-Samples->YSamples[Sample1Index]);
//      UpdateEstimate(&PreliminaryObjects[ObjectIndex].
    BLVelocity,(ForceObject->Y+ForceObject->Height)-(Samples->
    YSamples[Sample1Index]+Samples->Height[Sample1Index]));
//          }
//
//    if (Sample2Valid == TRUE)
//    {
//    //Update left line acceleration
//    Velocity1 = (ForceObject->X-Samples->XSamples[
```

```
    Sample1Index]);
//      Velocity2 = (Samples->XSamples[Sample1Index]-Samples->
    XSamples[Sample2Index]);
//      UpdateEstimate(&PreliminaryObjects[ObjectIndex].
    LLAcceleration,(Velocity1-Velocity2));
//
//      //Update right line acceleration
//      Velocity1 = (ForceObject->X+ForceObject->Width)-(Samples
    ->XSamples[Sample1Index]+Samples->Width[Sample1Index]);
//      Velocity2 = (Samples->XSamples[Sample1Index]+Samples->
    Width[Sample1Index]) - (Samples->XSamples[Sample2Index]+
    Samples->Width[Sample2Index]) ;
//      UpdateEstimate(&PreliminaryObjects[ObjectIndex].
    RLAcceleration,(Velocity1-Velocity2));
//
//    //Update top line acceleration
//    Velocity1 = ForceObject->Y-Samples->YSamples[Sample1Index
    ];
//    Velocity2 = Samples->YSamples[Sample1Index] - Samples->
    YSamples[Sample2Index];
//    UpdateEstimate(&PreliminaryObjects[ObjectIndex].
    TLAcceleration,(Velocity1-Velocity2));
//
//    //Update bottom line acceleration
//    Velocity1 = (ForceObject->Y+ForceObject->Height)-(Samples
    ->YSamples[Sample1Index]+Samples->Height[Sample1Index]);
//    Velocity2 = (Samples->YSamples[Sample1Index]+Samples->
    Height[Sample1Index]) - (Samples->YSamples[Sample2Index]+
    Samples->Height[Sample2Index]);
//    UpdateEstimate(&PreliminaryObjects[ObjectIndex].
    BLAcceleration,(Velocity1-Velocity2));
//
//
//
//      }
//
//
//
//
//PreliminaryObjects[ObjectIndex].PhysicalX =
    PreliminaryObjects[ObjectIndex].X;
//PreliminaryObjects[ObjectIndex].PhysicalY =
    PreliminaryObjects[ObjectIndex].Y;
//PreliminaryObjects[ObjectIndex].PhysicalWidth =
    PreliminaryObjects[ObjectIndex].Width;
//PreliminaryObjects[ObjectIndex].PhysicalHeight =
    PreliminaryObjects[ObjectIndex].Height;
//
////PreliminaryObjects[ObjectIndex].LLVelocity.CurrentEstimate
    +=PreliminaryObjects[ObjectIndex].LLAcceleration.
    CurrentEstimate;
////PreliminaryObjects[ObjectIndex].LLVelocity.Sum+=
    PreliminaryObjects[ObjectIndex].LLPositionError.
    CurrentEstimate/40;
////PreliminaryObjects[ObjectIndex].PhysicalX =
    PreliminaryObjects[ObjectIndex].PhysicalX +
```

```
    PreliminaryObjects[ObjectIndex].LLVelocity.CurrentEstimate
    +
////

    PreliminaryObjects[ObjectIndex].LLPositionError.
    CurrentEstimate/5;
////
////
////
////
////PreliminaryObjects[ObjectIndex].TLVelocity.CurrentEstimate
    +=PreliminaryObjects[ObjectIndex].TLAcceleration.
    CurrentEstimate;
////PreliminaryObjects[ObjectIndex].TLVelocity.Sum+=
    PreliminaryObjects[ObjectIndex].TLPositionError.
    CurrentEstimate/40;
////PreliminaryObjects[ObjectIndex].PhysicalY =
    PreliminaryObjects[ObjectIndex].PhysicalY +
    PreliminaryObjects[ObjectIndex].TLVelocity.CurrentEstimate
    +
////

    PreliminaryObjects[ObjectIndex].TLPositionError.
    CurrentEstimate/5;
////
////
////
////
////PreliminaryObjects[ObjectIndex].RLVelocity.CurrentEstimate
    +=PreliminaryObjects[ObjectIndex].RLAcceleration.
    CurrentEstimate;
////PreliminaryObjects[ObjectIndex].RLVelocity.Sum+=
    PreliminaryObjects[ObjectIndex].RLPositionError.
    CurrentEstimate/40;
////PreliminaryObjects[ObjectIndex].PhysicalWidth =
    PreliminaryObjects[ObjectIndex].PhysicalWidth + (
    PreliminaryObjects[ObjectIndex].RLVelocity.CurrentEstimate-
    PreliminaryObjects[ObjectIndex].LLVelocity.CurrentEstimate)
     +
////

                        PreliminaryObjects[ObjectIndex].
    RLPositionError.CurrentEstimate/5;
////
////
////
////PreliminaryObjects[ObjectIndex].BLVelocity.CurrentEstimate
    +=PreliminaryObjects[ObjectIndex].BLAcceleration.
    CurrentEstimate;
////PreliminaryObjects[ObjectIndex].BLVelocity.Sum+=
    PreliminaryObjects[ObjectIndex].BLPositionError.
    CurrentEstimate/40;
////PreliminaryObjects[ObjectIndex].PhysicalHeight =
    PreliminaryObjects[ObjectIndex].PhysicalHeight + (
    PreliminaryObjects[ObjectIndex].BLVelocity.CurrentEstimate-
    PreliminaryObjects[ObjectIndex].TLVelocity.CurrentEstimate)
```

```
    +
////

                PreliminaryObjects[ObjectIndex].BLPositionError.
    CurrentEstimate/5;
////
//
//
//if (PreliminaryObjects[ObjectIndex].PhysicalX<0)
    PreliminaryObjects[ObjectIndex].PhysicalX = 0;
//
//if (PreliminaryObjects[ObjectIndex].PhysicalX>639)
    PreliminaryObjects[ObjectIndex].PhysicalX = 638;
//if (PreliminaryObjects[ObjectIndex].PhysicalX+
    PreliminaryObjects[ObjectIndex].PhysicalWidth>638)
//      PreliminaryObjects[ObjectIndex].PhysicalWidth = 639 -
    PreliminaryObjects[ObjectIndex].PhysicalX;
//
//if (PreliminaryObjects[ObjectIndex].PhysicalY<0)
    PreliminaryObjects[ObjectIndex].PhysicalY = 0;
//if (PreliminaryObjects[ObjectIndex].PhysicalY>478)
    PreliminaryObjects[ObjectIndex].PhysicalY = 478;
//if (PreliminaryObjects[ObjectIndex].PhysicalY+
    PreliminaryObjects[ObjectIndex].PhysicalHeight>479)
//   PreliminaryObjects[ObjectIndex].PhysicalHeight = 479 -
    PreliminaryObjects[ObjectIndex].PhysicalY;
//
//if (PreliminaryObjects[ObjectIndex].PhysicalHeight<0)
    PreliminaryObjects[ObjectIndex].PhysicalHeight = 1;
//
//

}



void ObjectTracker::UpdateEstimate(TrackingData* TData,double
    NewValue)
{

if (TData->SamplesAcquired==7)
{
TData->Sum = TData->Sum - TData->CurrentEstimate;
}

TData->Sum = TData->Sum + NewValue;

if (TData->SamplesAcquired<7) TData->SamplesAcquired++;   //
    Maximum sample number is 5 samples for calculations

TData->CurrentEstimate = TData->Sum/(TData->SamplesAcquired);

}
```

```
void ObjectTracker::CreateHistogram(int ObjectIndex,BYTE*
    pSubBackground,BYTE* pCurrentFrame,int* Red,int* Green,int*
     Blue,int X,int Y,int Width, int Height)
{
int BufferIndex;


BufferIndex = (Y*VideoXLength+X)*4;

//Clear the histogram arrays
memset((void*)Red,0,256*4);
memset((void*)Green,0,256*4);
memset((void*)Blue,0,256*4);

   for (int y = 0; y<Height; y++)
   {
           for (int x = 0; x<Width; x++)
                {
                         if (pSubBackground[BufferIndex] == 255)
     //Is this highlighted as different by background
    subtraction
                              {
                              //Yes so add to histogram values
                             Blue[pCurrentFrame[BufferIndex]]++;
                             Green[pCurrentFrame[BufferIndex
    +1]]++;
                             Red[pCurrentFrame[BufferIndex+2]]++;
                             }
                      BufferIndex+=4;
                      }
   BufferIndex+=(VideoXLength-Width)*4;
   }


}





//store the new bounding box position in the X,Y,Width and
    Height variables
void ObjectTracker::LocateNewObjectPosition(int ObjectIndex,
    BYTE* pSubBackground,BYTE* pCurrentFrame)
{
ObjectFound* O;

O = &PreliminaryObjects[ObjectIndex];

int XPos;
```

```
int YPos;
int Width;
int Height;

int BufferIndex;

int X1;
int Y1;
int X2;
int Y2;

int CurrentX = 0;
int CurrentY = 0;

int XSum = 0;
int XSquareSum = 0;
int YSquareSum = 0;
int YSum = 0;
int PixelCount = 0;

double StandardDevX = 0; //Standard deviation of pixels along
    the X
double StandardDevY = 0; //standard deviation of pixels along
    the Y
double Temp;

int SmallestXShift;  //The smallest shift of the boundary of
    the object
int SmallestYShift;


int RelCenterX; //The centre of the bounding box relative to
    X1,Y1 as the origin
int RelCenterY;

int CenterX;
int CenterY; //The center of the current object position
    before it gets shifted to new location

int NewCenterX = 0; //The new calculated center of the
    bounding box
int NewCenterY = 0;

int DesiredShift; //The amount to expand each side of the
    bounding box to calculate the new




XPos = O->CentreX-O->Width/2;
YPos = O->CentreY-O->Height/2;
Width = PreliminaryObjects[ObjectIndex].Width;
Height = PreliminaryObjects[ObjectIndex].Height;

CenterX = O->CentreX;
CenterY = O->CentreY;


//Use a centre of mass approach to work out where the object
    is moving from the pSubBackground Background subtracted
    image array

//Expand the window by 15 pixels on all sides to include
    slight changes in the objects position in the centre of
```

```
    mass calc
X1 = XPos;
Y1 = YPos;
X2 = XPos+Width;
Y2 = YPos+Height;

SmallestXShift = 15;
SmallestYShift = 15;

if (X1>=15)
{
}
else
{
SmallestXShift = X1;
}

if (Y1>=15)
{
}
else
{
SmallestYShift = Y1;
}

if (X2<=624)
{
}
else
{
    if (SmallestXShift > (639-X2))
        SmallestXShift = (639-X2);
}

if (Y2<=464)
{
}
else
{
     if (SmallestYShift > (479-Y2))
             SmallestYShift = (479-Y2);
}

//X1 -=  SmallestXShift;
//X2 += SmallestXShift;
//Y1 -=  SmallestYShift;
//Y2 += SmallestYShift;

RelCenterX = CenterX - X1;
RelCenterY = CenterY - Y1;

BufferIndex = (Y1*VideoXLength+X1)*4;


for (int y=0; y<(Y2-Y1); y++)
{
CurrentY = (y-RelCenterY);
```

```
    for (int x =0; x<(X2-X1); x++)
    {
    CurrentX = x - RelCenterX;

        if (pSubBackground[BufferIndex] == 255) //If this is a
    foreground pixel include in calculations
            {
        PixelCount++;
            XSum+=CurrentX;
            XSquareSum+=CurrentX*CurrentX;
            YSum+=CurrentY;
            YSquareSum+=CurrentY*CurrentY;
            }

    BufferIndex+=4; //Point to next pixel
    }

BufferIndex+=(VideoXLength-(X2-X1))*4;
}


//if (PixelCount/(O->Width*O->Height)>0.15)
//{
if (PixelCount==0) PixelCount = 1;

NewCenterX = CenterX + XSum/PixelCount;
NewCenterY = CenterY + YSum/PixelCount;

StandardDevX = sqrt(XSquareSum/(double)PixelCount);
StandardDevY = sqrt(YSquareSum/(double)PixelCount);

//The width and height should be 3 standard deviations of
    their respective standard deviation variables
if (O->Width > StandardDevX*5)  O->Width-=3;
else O->Width+=3;

if (O->Height > StandardDevY*5) O->Height-=3;
else O->Height+=3;


PreliminaryObjects[ObjectIndex].X = NewCenterX - O->Width/2;
PreliminaryObjects[ObjectIndex].Y = NewCenterY - O->Height/2;

O->CentreX = NewCenterX;
O->CentreY = NewCenterY;

O->TotalTrackingPixels = PixelCount;


//}
//else
//{
//Just leave the coordinates alone as there are not enough
    valid pixels to calculate new center position
//}
```

```
    if (O->X<0) O->X = 0;

        if (O->X>639) O->X = 639;

        if (O->X+O->Width > 639)
                O->Width = 639 - O->X;

        if (O->Y<0) O->Y = 0;

        if (O->Y>479) O->Y = 479;

        if (O->Y+O->Height > 479)
                O->Height = 479 - O->Y;

        O->CentreX = O->X+O->Width/2;
        O->CentreY = O->Y+O->Height/2;

}




void ObjectTracker::AddDelData(HistData* Data,HistData*
   DelData)
{
        for (int i=0; i<256; i++)
        {
    Data->Blue1[i]+=DelData->Blue1[i];
        Data->Green1[i]+=DelData->Green1[i];
        Data->Red1[i]+=DelData->Red1[i];

        Data->Blue2[i]+=DelData->Blue2[i];
        Data->Green2[i]+=DelData->Green2[i];
        Data->Red2[i]+=DelData->Red2[i];

        Data->Blue3[i]+=DelData->Blue3[i];
        Data->Green3[i]+=DelData->Green3[i];
        Data->Red3[i]+=DelData->Red3[i];

    Data->Blue4[i]+=DelData->Blue4[i];
        Data->Green4[i]+=DelData->Green4[i];
        Data->Red4[i]+=DelData->Red4[i];
        }

}

//Returns a positive value if histograms are better matched to
    originals when moved to the left and the size of the
   number
//is the number of pixels that are a good match.
//If there is no improvement it returns a negative value with
   the size of this value indicating the number of pixels
//that are a bad match
//Note that the caller of this function should make sure that
   the bounding box can move two pixels to the left
int ObjectTracker::HistogramDeltaLeft(int ObjectIndex,BYTE*
   pSubBackground,BYTE* pCurrentFrame,HistData* Data,HistData*
```

```
    DelData)
{
int BufferIndex;
int DeadPixelCount = 0;
int Result=0;



memset((void*)DelData->Red1,0,256*4);
memset((void*)DelData->Green1,0,256*4);
memset((void*)DelData->Blue1,0,256*4);

memset((void*)DelData->Red2,0,256*4);
memset((void*)DelData->Green2,0,256*4);
memset((void*)DelData->Blue2,0,256*4);

memset((void*)DelData->Red3,0,256*4);
memset((void*)DelData->Green3,0,256*4);
memset((void*)DelData->Blue3,0,256*4);

memset((void*)DelData->Red4,0,256*4);
memset((void*)DelData->Green4,0,256*4);
memset((void*)DelData->Blue4,0,256*4);


//If moving to the left 2 pixels than the current sub bounding
     box will lose 2 vertical rows from the right and add 2
   vertical rows
//of pixels from the left


//Sub Bounding box 1
for (int x=0; x<2; x++) //Loop through the first two columns
   of pixels to add
{
BufferIndex = (Data->Y1*VideoXLength+Data->X1-(x+1))*4; //
   Point to the right strip

    for (int y=0; y<Data->Height1; y++) //Add this row of
   pixels to the DelData->ta histogram
         {

                 //if ( Data->Blue1[pCurrentFrame[BufferIndex]]
   > 0 && Data->Green1[pCurrentFrame[BufferIndex+1]] > 0 &&
             //  Data->Red1[pCurrentFrame[BufferIndex+2]] > 0)
                 //{
                 DelData->Blue1[pCurrentFrame[BufferIndex]]++;
                 DelData->Green1[pCurrentFrame[BufferIndex
   +1]]++;
                 DelData->Red1[pCurrentFrame[BufferIndex+2]]++;
                 //}
                 if (pSubBackground[BufferIndex] == 0)   //If
   this pixel part of background it is a dead pixel which we
   are trying to minimize
                 {
         DeadPixelCount++;
                 }
```

```
        BufferIndex+=VideoXLength*4;
        }

}

for (int x=0; x<2; x++) //Loop through two columns on the
    right and subtract these
{
BufferIndex = (Data->Y1*VideoXLength+Data->X1+Data->Width1-x)
    *4; //Point to the right strip

     for (int y=0; y<Data->Height1; y++) //Subtract this row of
     pixels to the DelData->ta histogram
        {

                //if ( Data->Blue1[pCurrentFrame[BufferIndex]]
    > 0 && Data->Green1[pCurrentFrame[BufferIndex+1]] > 0 &&
            //  Data->Red1[pCurrentFrame[BufferIndex+2]] > 0)
                //{
                DelData->Blue1[pCurrentFrame[BufferIndex]]--;
                DelData->Green1[pCurrentFrame[BufferIndex
    +1]]--;
                DelData->Red1[pCurrentFrame[BufferIndex+2]]--;
                //}
                if (pSubBackground[BufferIndex] == 0)   //If
    this pixel part of background it is a dead pixel which we
    are trying to minimize
                {
        DeadPixelCount--;
                }

        BufferIndex+=VideoXLength*4;
        }

}


//Sub Bounding box 2
for (int x=0; x<2; x++) //Loop through the first two columns
    of pixels to add
{
BufferIndex = (Data->Y2*VideoXLength+Data->X2-(x+1))*4; //
    Point to the right strip

     for (int y=0; y<Data->Height2; y++) //Add this row of
    pixels to the DelData->ta histogram
        {

                //if ( Data->Blue2[pCurrentFrame[BufferIndex]]
    > 0 && Data->Green2[pCurrentFrame[BufferIndex+1]] > 0 &&
            //  Data->Red2[pCurrentFrame[BufferIndex+2]] > 0)
                //{
                DelData->Blue2[pCurrentFrame[BufferIndex]]++;
            DelData->Green2[pCurrentFrame[BufferIndex+1]]++;
                DelData->Red2[pCurrentFrame[BufferIndex+2]]++;
                //}
                if (pSubBackground[BufferIndex] == 0)   //If
```

```
    this pixel part of background it is a dead pixel which we
    are trying to minimize
                {
          DeadPixelCount++;
                }

          BufferIndex+=VideoXLength*4;
          }

}

for (int x=0; x<2; x++) //Loop through two columns on the
    right and subtract these
{
BufferIndex = (Data->Y2*VideoXLength+Data->X2+Data->Width2-x)
    *4; //Point to the right strip

     for (int y=0; y<Data->Height2; y++) //Subtract this row of
     pixels to the DelData->ta histogram
          {

                //if ( Data->Blue2[pCurrentFrame[BufferIndex]]
     > 0 && Data->Green2[pCurrentFrame[BufferIndex+1]] > 0 &&
             //  Data->Red2[pCurrentFrame[BufferIndex+2]] > 0)
                //{
                DelData->Blue2[pCurrentFrame[BufferIndex]]--;
                DelData->Green2[pCurrentFrame[BufferIndex
    +1]]--;
                DelData->Red2[pCurrentFrame[BufferIndex+2]]--;
                //}
                if (pSubBackground[BufferIndex] == 0)   //If
    this pixel part of background it is a dead pixel which we
    are trying to minimize
                {
          DeadPixelCount--;
                }

          BufferIndex+=VideoXLength*4;
          }

}


//Sub Bounding box 3
for (int x=0; x<2; x++) //Loop through the first two columns
    of pixels to add
{
BufferIndex = (Data->Y3*VideoXLength+Data->X3-(x+1))*4; //
    Point to the right strip

     for (int y=0; y<Data->Height2; y++) //Add this row of
    pixels to the DelData->ta histogram
          {

                //if ( Data->Blue3[pCurrentFrame[BufferIndex]]
     > 0 && Data->Green3[pCurrentFrame[BufferIndex+1]] > 0 &&
               //Data->Red3[pCurrentFrame[BufferIndex+2]] > 0)
```

```
                    //{
                    DelData->Blue3[pCurrentFrame[BufferIndex]]++;
                    DelData->Green3[pCurrentFrame[BufferIndex
    +1]]++;
                    DelData->Red3[pCurrentFrame[BufferIndex+2]]++;
                    //}
                    if (pSubBackground[BufferIndex] == 0)   //If
    this pixel part of background it is a dead pixel which we
    are trying to minimize
                    {
          DeadPixelCount++;
                    }

          BufferIndex+=VideoXLength*4;
          }

}

for (int x=0; x<2; x++) //Loop through two columns on the
    right and subtract these
{
BufferIndex = (Data->Y3*VideoXLength+Data->X3+Data->Width3-x)
    *4; //Point to the right strip

     for (int y=0; y<Data->Height2; y++) //Subtract this row of
     pixels to the DelData->ta histogram
          {

                    //if ( Data->Blue3[pCurrentFrame[BufferIndex]]
     > 0 && Data->Green3[pCurrentFrame[BufferIndex+1]] > 0 &&
               //   Data->Red3[pCurrentFrame[BufferIndex+2]] > 0)
                    //{
                    DelData->Blue3[pCurrentFrame[BufferIndex]]--;
                    DelData->Green3[pCurrentFrame[BufferIndex
    +1]]--;
                    DelData->Red3[pCurrentFrame[BufferIndex+2]]--;
                    //}
                    if (pSubBackground[BufferIndex] == 0)   //If
    this pixel part of background it is a dead pixel which we
    are trying to minimize
                    {
          DeadPixelCount--;
                    }

          BufferIndex+=VideoXLength*4;
          }

}



//Sub Bounding box 4
for (int x=0; x<2; x++) //Loop through the first two columns
    of pixels to add
{
BufferIndex = (Data->Y4*VideoXLength+Data->X4-(x+1))*4; //
```

```
    Point to the right strip
     for (int y=0; y<Data->Height2; y++) //Add this row of
    pixels to the DelData->ta histogram
          {
                    //if ( Data->Blue4[pCurrentFrame[BufferIndex]]
     > 0 && Data->Green4[pCurrentFrame[BufferIndex+1]] > 0 &&
               //   Data->Red4[pCurrentFrame[BufferIndex+2]] > 0)
                    //{
                    DelData->Blue4[pCurrentFrame[BufferIndex]]++;
                    DelData->Green4[pCurrentFrame[BufferIndex
    +1]]++;
                    DelData->Red4[pCurrentFrame[BufferIndex+2]]++;
                    //}
                    if (pSubBackground[BufferIndex] == 0)   //If
    this pixel part of background it is a dead pixel which we
    are trying to minimize
                    {
          DeadPixelCount++;
                    }

          BufferIndex+=VideoXLength*4;
          }

}

for (int x=0; x<2; x++) //Loop through two columns on the
    right and subtract these
{
BufferIndex = (Data->Y4*VideoXLength+Data->X4+Data->Width4-x)
    *4; //Point to the right strip
     for (int y=0; y<Data->Height2; y++) //Subtract this row of
     pixels to the DelData histogram
          {
                    //if ( Data->Blue4[pCurrentFrame[BufferIndex]]
     > 0 && Data->Green4[pCurrentFrame[BufferIndex+1]] > 0 &&
               //   Data->Red4[pCurrentFrame[BufferIndex+2]] > 0)
                    //{
                    DelData->Blue4[pCurrentFrame[BufferIndex]]--;
                    DelData->Green4[pCurrentFrame[BufferIndex
    +1]]--;
                    DelData->Red4[pCurrentFrame[BufferIndex+2]]--;
                    //}
                    if (pSubBackground[BufferIndex] == 0)   //If
    this pixel part of background it is a dead pixel which we
    are trying to minimize
                    {
          DeadPixelCount--;
                    }

          BufferIndex+=VideoXLength*4;
          }

}
```

```
//return(CalculateImprovement(ObjectIndex,Data,DelData));
return(CalculateImprovement(ObjectIndex,Data,DelData)+(
    DeadPixelCount*-6));

}






int ObjectTracker::HistogramDeltaRight(int ObjectIndex,BYTE*
    pSubBackground,BYTE* pCurrentFrame,HistData* Data,HistData*
     DelData)
{

int BufferIndex;
ObjectFound* O;
int DeadPixelCount = 0;
int Result=0;

O = &PreliminaryObjects[ObjectIndex];

memset((void*)DelData->Red1,0,256*4);
memset((void*)DelData->Green1,0,256*4);
memset((void*)DelData->Blue1,0,256*4);

memset((void*)DelData->Red2,0,256*4);
memset((void*)DelData->Green2,0,256*4);
memset((void*)DelData->Blue2,0,256*4);

memset((void*)DelData->Red3,0,256*4);
memset((void*)DelData->Green3,0,256*4);
memset((void*)DelData->Blue3,0,256*4);

memset((void*)DelData->Red4,0,256*4);
memset((void*)DelData->Green4,0,256*4);
memset((void*)DelData->Blue4,0,256*4);



//Sub Bounding box 1
for (int x=0; x<2; x++) //Loop through the first two columns
    of pixels to add
{
BufferIndex = (Data->Y1*VideoXLength+Data->X1+x)*4; //Point to
     the right strip

     for (int y=0; y<Data->Height1; y++) //Add this row of
    pixels to the DelData histogram
          {

                    if ( O->Blue1[pCurrentFrame[BufferIndex]] > 0
   && O->Green1[pCurrentFrame[BufferIndex+1]] > 0 &&
                 O->Red1[pCurrentFrame[BufferIndex+2]] > 0)
```

```
                {
                DelData->Blue1[pCurrentFrame[BufferIndex]]--;
                DelData->Green1[pCurrentFrame[BufferIndex
    +1]]--;
                DelData->Red1[pCurrentFrame[BufferIndex+2]]--;
                }
                if (pSubBackground[BufferIndex] == 0)   //If
    this pixel part of background it is a dead pixel which we
    are trying to minimize
                {
        DeadPixelCount--;
                }

        BufferIndex+=VideoXLength*4;
        }

}

for (int x=0; x<2; x++) //Loop through two columns on the
    right and subtract these
{
BufferIndex = (Data->Y1*VideoXLength+Data->X1+Data->Width1+(x
    +1))*4; //Point to the right strip

     for (int y=0; y<Data->Height1; y++) //Subtract this row of
     pixels to the DelData->ta histogram
        {

                if ( O->Blue1[pCurrentFrame[BufferIndex]] > 0
    && O->Green1[pCurrentFrame[BufferIndex+1]] > 0 &&
             O->Red1[pCurrentFrame[BufferIndex+2]] > 0)
                {
                DelData->Blue1[pCurrentFrame[BufferIndex]]++;
                DelData->Green1[pCurrentFrame[BufferIndex
    +1]]++;
                DelData->Red1[pCurrentFrame[BufferIndex+2]]++;
                }
                if (pSubBackground[BufferIndex] == 0)   //If
    this pixel part of background it is a dead pixel which we
    are trying to minimize
                {
        DeadPixelCount++;
                }

        BufferIndex+=VideoXLength*4;
        }

}


//Sub Bounding box 2
for (int x=0; x<2; x++) //Loop through the first two columns
    of pixels to add
{
BufferIndex = (Data->Y2*VideoXLength+Data->X2+x)*4; //Point to
     the right strip
```

```
    for (int y=0; y<Data->Height2; y++) //Add this row of
    pixels to the DelData->ta histogram
          {

                if ( O->Blue2[pCurrentFrame[BufferIndex]] > 0
   && O->Green2[pCurrentFrame[BufferIndex+1]] > 0 &&
               O->Red2[pCurrentFrame[BufferIndex+2]] > 0)
                {
                DelData->Blue2[pCurrentFrame[BufferIndex]]--;
                DelData->Green2[pCurrentFrame[BufferIndex
   +1]]--;
                DelData->Red2[pCurrentFrame[BufferIndex+2]]--;
                }
                if (pSubBackground[BufferIndex] == 0)  //If
    this pixel part of background it is a dead pixel which we
    are trying to minimize
                {
          DeadPixelCount--;
                }

          BufferIndex+=VideoXLength*4;
          }

}

for (int x=0; x<2; x++) //Loop through two columns on the
    right and subtract these
{
BufferIndex = (Data->Y2*VideoXLength+Data->X2+Data->Width2+(x
   +1))*4; //Point to the right strip

    for (int y=0; y<Data->Height2; y++) //Subtract this row of
    pixels to the DelData->ta histogram
          {

                if ( O->Blue2[pCurrentFrame[BufferIndex]] > 0
   && O->Green2[pCurrentFrame[BufferIndex+1]] > 0 &&
               O->Red2[pCurrentFrame[BufferIndex+2]] > 0)
                {
                DelData->Blue2[pCurrentFrame[BufferIndex]]++;
                DelData->Green2[pCurrentFrame[BufferIndex
   +1]]++;
                DelData->Red2[pCurrentFrame[BufferIndex+2]]++;
                }
                if (pSubBackground[BufferIndex] == 0)  //If
    this pixel part of background it is a dead pixel which we
    are trying to minimize
                {
          DeadPixelCount++;
                }

          BufferIndex+=VideoXLength*4;
          }

}
```

```
//Sub Bounding box 3
for (int x=0; x<2; x++) //Loop through the first two columns
    of pixels to add
{
BufferIndex = (Data->Y3*VideoXLength+Data->X3+x)*4; //Point to
     the right strip

     for (int y=0; y<Data->Height2; y++) //Add this row of
    pixels to the DelData->ta histogram
          {

                  if ( O->Blue3[pCurrentFrame[BufferIndex]] > 0
   && O->Green3[pCurrentFrame[BufferIndex+1]] > 0 &&
                O->Red3[pCurrentFrame[BufferIndex+2]] > 0)
                  {
                  DelData->Blue3[pCurrentFrame[BufferIndex]]--;
                  DelData->Green3[pCurrentFrame[BufferIndex
   +1]]--;
                  DelData->Red3[pCurrentFrame[BufferIndex+2]]--;
                  }
                  if (pSubBackground[BufferIndex] == 0)   //If
   this pixel part of background it is a dead pixel which we
   are trying to minimize
                  {
          DeadPixelCount--;
                  }

          BufferIndex+=VideoXLength*4;
          }

}

for (int x=0; x<2; x++) //Loop through two columns on the
    right and subtract these
{
BufferIndex = (Data->Y3*VideoXLength+Data->X3+Data->Width3+(x
   +1))*4; //Point to the right strip

     for (int y=0; y<Data->Height2; y++) //Subtract this row of
     pixels to the DelData->ta histogram
          {

                  if ( O->Blue3[pCurrentFrame[BufferIndex]] > 0
   && O->Green3[pCurrentFrame[BufferIndex+1]] > 0 &&
                O->Red3[pCurrentFrame[BufferIndex+2]] > 0)
                  {
                  DelData->Blue3[pCurrentFrame[BufferIndex]]++;
                  DelData->Green3[pCurrentFrame[BufferIndex
   +1]]++;
                  DelData->Red3[pCurrentFrame[BufferIndex+2]]++;
                  }
                  if (pSubBackground[BufferIndex] == 0)   //If
   this pixel part of background it is a dead pixel which we
   are trying to minimize
                  {
          DeadPixelCount++;
                  }
```

```
                BufferIndex+=VideoXLength*4;
                }

}




//Sub Bounding box 4
for (int x=0; x<2; x++) //Loop through the first two columns
    of pixels to add
{
BufferIndex = (Data->Y4*VideoXLength+Data->X4+x)*4; //Point to
     the right strip

     for (int y=0; y<Data->Height2; y++) //Add this row of
    pixels to the DelData->ta histogram
          {

                    if ( O->Blue4[pCurrentFrame[BufferIndex]] > 0
   && O->Green4[pCurrentFrame[BufferIndex+1]] > 0 &&
                O->Red4[pCurrentFrame[BufferIndex+2]] > 0)
                    {
                    DelData->Blue4[pCurrentFrame[BufferIndex]]--;
                    DelData->Green4[pCurrentFrame[BufferIndex
   +1]]--;
                    DelData->Red4[pCurrentFrame[BufferIndex+2]]--;
                    }
                    if (pSubBackground[BufferIndex] == 0)   //If
    this pixel part of background it is a dead pixel which we
    are trying to minimize
                    {
          DeadPixelCount--;
                    }

          BufferIndex+=VideoXLength*4;
          }

}

for (int x=0; x<2; x++) //Loop through two columns on the
    right and subtract these
{
BufferIndex = (Data->Y4*VideoXLength+Data->X4+Data->Width4+(x
    +1))*4; //Point to the right strip

     for (int y=0; y<Data->Height2; y++) //Subtract this row of
     pixels to the DelData histogram
          {

                    if ( O->Blue4[pCurrentFrame[BufferIndex]] > 0
   && O->Green4[pCurrentFrame[BufferIndex+1]] > 0 &&
                O->Red4[pCurrentFrame[BufferIndex+2]] > 0)
                    {
                    DelData->Blue4[pCurrentFrame[BufferIndex]]++;
                    DelData->Green4[pCurrentFrame[BufferIndex
   +1]]++;
```

```
                DelData->Red4[pCurrentFrame[BufferIndex+2]]++;
                }
                if (pSubBackground[BufferIndex] == 0)  //If
    this pixel part of background it is a dead pixel which we
    are trying to minimize
                {
        DeadPixelCount++;
                }

        BufferIndex+=VideoXLength*4;
        }

}


return(CalculateImprovement(ObjectIndex,Data,DelData)+(
    DeadPixelCount*-6));
//return(CalculateImprovement(ObjectIndex,Data,DelData));

}


int ObjectTracker::HistogramDeltaUp(int ObjectIndex,BYTE*
    pSubBackground,BYTE* pCurrentFrame,HistData* Data,HistData*
     DelData)
{

int BufferIndex;
int DeadPixelCount=0;

int Result=0;



memset((void*)DelData->Red1,0,256*4);
memset((void*)DelData->Green1,0,256*4);
memset((void*)DelData->Blue1,0,256*4);

memset((void*)DelData->Red2,0,256*4);
memset((void*)DelData->Green2,0,256*4);
memset((void*)DelData->Blue2,0,256*4);

memset((void*)DelData->Red3,0,256*4);
memset((void*)DelData->Green3,0,256*4);
memset((void*)DelData->Blue3,0,256*4);

memset((void*)DelData->Red4,0,256*4);
memset((void*)DelData->Green4,0,256*4);
memset((void*)DelData->Blue4,0,256*4);


//Sub Bounding box 1
```

```
for (int y=0; y<2; y++) //Loop through the first two columns
    of pixels to add
{
BufferIndex = ( (Data->Y1-(y+1))*VideoXLength+Data->X1)*4; //
    Point to the right strip

     for (int x=0; x<Data->Width1; x++) //Add this row of
    pixels to the DelData histogram
          {

                   if ( Data->Blue1[pCurrentFrame[BufferIndex]] >
     0 && Data->Green1[pCurrentFrame[BufferIndex+1]] > 0 &&
                  Data->Red1[pCurrentFrame[BufferIndex+2]] > 0)
                   {
                   DelData->Blue1[pCurrentFrame[BufferIndex]]++;
                   DelData->Green1[pCurrentFrame[BufferIndex
    +1]]++;
                   DelData->Red1[pCurrentFrame[BufferIndex+2]]++;
                   }
                   if (pSubBackground[BufferIndex] == 0)   //If
    this pixel part of background it is a dead pixel which we
    are trying to minimize
                   {
          DeadPixelCount++;
                   }

          BufferIndex+=4;
          }

}

for (int y=0; y<2; y++) //Loop through two columns on the
    right and subtract these
{
BufferIndex = ( (Data->Y1+Data->Height1-y)*VideoXLength+Data->
    X1)*4; //Point to the right strip

     for (int x=0; x<Data->Width1; x++) //Subtract this row of
    pixels to the DelData->ta histogram
          {

                   if ( Data->Blue1[pCurrentFrame[BufferIndex]] >
     0 && Data->Green1[pCurrentFrame[BufferIndex+1]] > 0 &&
                  Data->Red1[pCurrentFrame[BufferIndex+2]] > 0)
                   {
                   DelData->Blue1[pCurrentFrame[BufferIndex]]--;
                   DelData->Green1[pCurrentFrame[BufferIndex
    +1]]--;
                   DelData->Red1[pCurrentFrame[BufferIndex+2]]--;
                   }
                   if (pSubBackground[BufferIndex] == 0)   //If
    this pixel part of background it is a dead pixel which we
    are trying to minimize
                   {
          DeadPixelCount--;
                   }
```

```
          BufferIndex+=4;
          }

}


//Sub Bounding box 2
for (int y=0; y<2; y++) //Loop through the first two columns
    of pixels to add
{
BufferIndex = ( (Data->Y2-(y+1))*VideoXLength+Data->X2)*4; //
    Point to the right strip

     for (int x=0; x<Data->Width1; x++) //Add this row of
    pixels to the DelData histogram
          {

                    if ( Data->Blue2[pCurrentFrame[BufferIndex]] >
     0 && Data->Green2[pCurrentFrame[BufferIndex+1]] > 0 &&
                 Data->Red2[pCurrentFrame[BufferIndex+2]] > 0)
                    {
                    DelData->Blue2[pCurrentFrame[BufferIndex]]++;
                    DelData->Green2[pCurrentFrame[BufferIndex
    +1]]++;
                    DelData->Red2[pCurrentFrame[BufferIndex+2]]++;
                    }
                    if (pSubBackground[BufferIndex] == 0)   //If
    this pixel part of background it is a dead pixel which we
    are trying to minimize
                    {
          DeadPixelCount++;
                    }

          BufferIndex+=4;
          }

}

for (int y=0; y<2; y++) //Loop through two columns on the
    right and subtract these
{
BufferIndex = ( (Data->Y2+Data->Height2-y)*VideoXLength+Data->
    X2)*4; //Point to the right strip

     for (int x=0; x<Data->Width1; x++) //Subtract this row of
    pixels to the DelData histogram
          {

                    if ( Data->Blue2[pCurrentFrame[BufferIndex]] >
     0 && Data->Green2[pCurrentFrame[BufferIndex+1]] > 0 &&
                 Data->Red2[pCurrentFrame[BufferIndex+2]] > 0)
                    {
                    DelData->Blue2[pCurrentFrame[BufferIndex]]--;
                    DelData->Green2[pCurrentFrame[BufferIndex
    +1]]--;
                    DelData->Red2[pCurrentFrame[BufferIndex+2]]--;
                    }
```

```
                if (pSubBackground[BufferIndex] == 0)   //If
    this pixel part of background it is a dead pixel which we
    are trying to minimize
                {
        DeadPixelCount--;
                }

        BufferIndex+=4;
        }

}




//Sub Bounding box 3
for (int y=0; y<2; y++) //Loop through the first two columns
    of pixels to add
{
BufferIndex = ( (Data->Y3-(y+1))*VideoXLength+Data->X3)*4; //
    Point to the right strip

     for (int x=0; x<Data->Width3; x++) //Add this row of
    pixels to the DelData histogram
        {

                if ( Data->Blue3[pCurrentFrame[BufferIndex]] >
     0 && Data->Green3[pCurrentFrame[BufferIndex+1]] > 0 &&
              Data->Red3[pCurrentFrame[BufferIndex+2]] > 0)
                {
                DelData->Blue3[pCurrentFrame[BufferIndex]]++;
                DelData->Green3[pCurrentFrame[BufferIndex
    +1]]++;
                DelData->Red3[pCurrentFrame[BufferIndex+2]]++;
                }
                if (pSubBackground[BufferIndex] == 0)   //If
    this pixel part of background it is a dead pixel which we
    are trying to minimize
                {
        DeadPixelCount++;
                }

        BufferIndex+=4;
        }

}

for (int y=0; y<2; y++) //Loop through two columns on the
    right and subtract these
{
BufferIndex = ( (Data->Y3+Data->Height3-y)*VideoXLength+Data->
    X3)*4; //Point to the right strip

     for (int x=0; x<Data->Width3; x++) //Subtract this row of
    pixels to the DelData histogram
        {
```

```
                if ( Data->Blue3[pCurrentFrame[BufferIndex]] >
    0 && Data->Green3[pCurrentFrame[BufferIndex+1]] > 0 &&
              Data->Red3[pCurrentFrame[BufferIndex+2]] > 0)
                {
                DelData->Blue3[pCurrentFrame[BufferIndex]]--;
                DelData->Green3[pCurrentFrame[BufferIndex
    +1]]--;
                DelData->Red3[pCurrentFrame[BufferIndex+2]]--;
                }
                if (pSubBackground[BufferIndex] == 0)   //If
    this pixel part of background it is a dead pixel which we
    are trying to minimize
                {
        DeadPixelCount--;
                }

        BufferIndex+=4;
        }

}


//Sub Bounding box 4
for (int y=0; y<2; y++) //Loop through the first two columns
    of pixels to add
{
BufferIndex = ( (Data->Y4-(y+1))*VideoXLength+Data->X4)*4; //
    Point to the right strip

    for (int x=0; x<Data->Width4; x++) //Add this row of
    pixels to the DelData histogram
        {

                if ( Data->Blue4[pCurrentFrame[BufferIndex]] >
    0 && Data->Green4[pCurrentFrame[BufferIndex+1]] > 0 &&
              Data->Red4[pCurrentFrame[BufferIndex+2]] > 0)
                {
                DelData->Blue4[pCurrentFrame[BufferIndex]]++;
                DelData->Green4[pCurrentFrame[BufferIndex
    +1]]++;
                DelData->Red4[pCurrentFrame[BufferIndex+2]]++;
                }
                if (pSubBackground[BufferIndex] == 0)   //If
    this pixel part of background it is a dead pixel which we
    are trying to minimize
                {
        DeadPixelCount++;
                }

        BufferIndex+=4;
        }

}

for (int y=0; y<2; y++) //Loop through two columns on the
    right and subtract these
```

```
{
BufferIndex = ( (Data->Y4+Data->Height4-y)*VideoXLength+Data->
    X4)*4; //Point to the right strip

     for (int x=0; x<Data->Width4; x++) //Subtract this row of
    pixels to the DelData histogram
         {

                 if ( Data->Blue4[pCurrentFrame[BufferIndex] ]
    > 0 && Data->Green4[pCurrentFrame[BufferIndex+1]] > 0 &&
               Data->Red4[pCurrentFrame[BufferIndex+2]] > 0)
                 {
                 DelData->Blue4[pCurrentFrame[BufferIndex]]--;
                 DelData->Green4[pCurrentFrame[BufferIndex
    +1]]--;
                 DelData->Red4[pCurrentFrame[BufferIndex+2]]--;
                 }
                 if (pSubBackground[BufferIndex] == 0)   //If
    this pixel part of background it is a dead pixel which we
    are trying to minimize
                 {
         DeadPixelCount--;
                 }

         BufferIndex+=4;
         }

}



return(CalculateImprovement(ObjectIndex,Data,DelData)+(
    DeadPixelCount*-6));
//return(CalculateImprovement(ObjectIndex,Data,DelData));

}








int ObjectTracker::HistogramDeltaDown(int ObjectIndex,BYTE*
    pSubBackground,BYTE* pCurrentFrame,HistData* Data,HistData*
     DelData)
{

int BufferIndex;
ObjectFound* O;

int Result=0;

O = &PreliminaryObjects[ObjectIndex];
```

```
int DeadPixelCount = 0;

memset((void*)DelData->Red1,0,256*4);
memset((void*)DelData->Green1,0,256*4);
memset((void*)DelData->Blue1,0,256*4);

memset((void*)DelData->Red2,0,256*4);
memset((void*)DelData->Green2,0,256*4);
memset((void*)DelData->Blue2,0,256*4);

memset((void*)DelData->Red3,0,256*4);
memset((void*)DelData->Green3,0,256*4);
memset((void*)DelData->Blue3,0,256*4);

memset((void*)DelData->Red4,0,256*4);
memset((void*)DelData->Green4,0,256*4);
memset((void*)DelData->Blue4,0,256*4);


//Sub Bounding box 1
for (int y=0; y<2; y++) //Loop through the first two columns
    of pixels to add
{
BufferIndex = ( (Data->Y1+y)*VideoXLength+Data->X1)*4; //Point
     to the right strip

     for (int x=0; x<Data->Width1; x++) //Add this row of
    pixels to the DelData histogram
          {

                    if ( O->Blue1[pCurrentFrame[BufferIndex]] > 0
   && O->Green1[pCurrentFrame[BufferIndex+1]] > 0 &&
               O->Red1[pCurrentFrame[BufferIndex+2]] > 0)
                    {
                    DelData->Blue1[pCurrentFrame[BufferIndex]]--;
                    DelData->Green1[pCurrentFrame[BufferIndex
   +1]]--;
                    DelData->Red1[pCurrentFrame[BufferIndex+2]]--;
                    }
                    if (pSubBackground[BufferIndex] == 0)   //If
    this pixel part of background it is a dead pixel which we
    are trying to minimize
                    {
          DeadPixelCount--;
                    }

          BufferIndex+=4;
          }

}

for (int y=0; y<2; y++) //Loop through two columns on the
    right and subtract these
{
BufferIndex = ( (Data->Y1+Data->Height1+(y+1))*VideoXLength+
   Data->X1)*4; //Point to the right strip

     for (int x=0; x<Data->Width1; x++) //Subtract this row of
```

```
    pixels to the DelData->ta histogram
          {

                if ( O->Blue1[pCurrentFrame[BufferIndex]] > 0
   && O->Green1[pCurrentFrame[BufferIndex+1]] > 0 &&
               O->Red1[pCurrentFrame[BufferIndex+2]] > 0)
                {
                        DelData->Blue1[pCurrentFrame[
    BufferIndex]]++;
                        DelData->Green1[pCurrentFrame[
    BufferIndex+1]]++;
                        DelData->Red1[pCurrentFrame[
    BufferIndex+2]]++;
                }
                if (pSubBackground[BufferIndex] == 0)  //If
    this pixel part of background it is a dead pixel which we
    are trying to minimize
                {
          DeadPixelCount++;
                }

          BufferIndex+=4;
          }

}


//Sub Bounding box 2
for (int y=0; y<2; y++) //Loop through the first two columns
    of pixels to add
{
BufferIndex = ( (Data->Y2+y)*VideoXLength+Data->X2)*4; //Point
     to the right strip

     for (int x=0; x<Data->Width1; x++) //Add this row of
    pixels to the DelData histogram
          {

                if ( O->Blue2[pCurrentFrame[BufferIndex]] > 0
   && O->Green2[pCurrentFrame[BufferIndex+1]] > 0 &&
               O->Red2[pCurrentFrame[BufferIndex+2]] > 0)
                {
                        DelData->Blue2[pCurrentFrame[
    BufferIndex]]--;
                        DelData->Green2[pCurrentFrame[
    BufferIndex+1]]--;
                        DelData->Red2[pCurrentFrame[
    BufferIndex+2]]--;
                }
                if (pSubBackground[BufferIndex] == 0)  //If
    this pixel part of background it is a dead pixel which we
    are trying to minimize
                {
          DeadPixelCount--;
                }

          BufferIndex+=4;
```

```
        }

}

for (int y=0; y<2; y++) //Loop through two columns on the
    right and subtract these
{
BufferIndex = ( (Data->Y2+Data->Height2+(y+1))*VideoXLength+
    Data->X2)*4; //Point to the right strip

     for (int x=0; x<Data->Width1; x++) //Subtract this row of
    pixels to the DelData histogram
        {

                if ( O->Blue2[pCurrentFrame[BufferIndex]] > 0
    && O->Green2[pCurrentFrame[BufferIndex+1]] > 0 &&
              O->Red2[pCurrentFrame[BufferIndex+2]] > 0)
                {
                DelData->Blue2[pCurrentFrame[BufferIndex]]++;
                DelData->Green2[pCurrentFrame[BufferIndex
    +1]]++;
                DelData->Red2[pCurrentFrame[BufferIndex+2]]++;
                }
                if (pSubBackground[BufferIndex] == 0)   //If
    this pixel part of background it is a dead pixel which we
    are trying to minimize
                {
        DeadPixelCount++;
                }

        BufferIndex+=4;
        }

}




//Sub Bounding box 3
for (int y=0; y<2; y++) //Loop through the first two columns
    of pixels to add
{
BufferIndex = ( (Data->Y3+y)*VideoXLength+Data->X3)*4; //Point
     to the right strip

     for (int x=0; x<Data->Width3; x++) //Add this row of
    pixels to the DelData histogram
        {

                if ( O->Blue3[pCurrentFrame[BufferIndex]] > 0
    && O->Green3[pCurrentFrame[BufferIndex+1]] > 0 &&
               O->Red3[pCurrentFrame[BufferIndex+2]] > 0)
                {
                        DelData->Blue3[pCurrentFrame[
    BufferIndex]]--;
                        DelData->Green3[pCurrentFrame[
    BufferIndex+1]]--;
```

```
                        DelData->Red3[pCurrentFrame[
   BufferIndex+2]]--;
                }
                if (pSubBackground[BufferIndex] == 0)   //If
   this pixel part of background it is a dead pixel which we
   are trying to minimize
                {
        DeadPixelCount--;
                }

        BufferIndex+=4;
        }

}

for (int y=0; y<2; y++) //Loop through two columns on the
    right and subtract these
{
BufferIndex = ( (Data->Y3+Data->Height3+(y+1))*VideoXLength+
   Data->X3)*4; //Point to the right strip

    for (int x=0; x<Data->Width3; x++) //Subtract this row of
   pixels to the DelData histogram
        {

                if ( O->Blue3[pCurrentFrame[BufferIndex]] > 0
   && O->Green3[pCurrentFrame[BufferIndex+1]] > 0 &&
              O->Red3[pCurrentFrame[BufferIndex+2]] > 0)
                {
                        DelData->Blue3[pCurrentFrame[
   BufferIndex]]++;
                        DelData->Green3[pCurrentFrame[
   BufferIndex+1]]++;
                        DelData->Red3[pCurrentFrame[
   BufferIndex+2]]++;
                }
                if (pSubBackground[BufferIndex] == 0)   //If
   this pixel part of background it is a dead pixel which we
   are trying to minimize
                {
        DeadPixelCount++;
                }

        BufferIndex+=4;
        }

}


//Sub Bounding box 4
for (int y=0; y<2; y++) //Loop through the first two columns
    of pixels to add
{
BufferIndex = ( (Data->Y4+y)*VideoXLength+Data->X4)*4; //Point
     to the right strip

    for (int x=0; x<Data->Width4; x++) //Add this row of
```

```
    pixels to the DelData histogram
          {

                  if ( O->Blue4[pCurrentFrame[BufferIndex]] > 0
    && O->Green4[pCurrentFrame[BufferIndex+1]] > 0 &&
                 O->Red4[pCurrentFrame[BufferIndex+2]] > 0)
                  {
                          DelData->Blue4[pCurrentFrame[
    BufferIndex]]--;
                          DelData->Green4[pCurrentFrame[
    BufferIndex+1]]--;
                          DelData->Red4[pCurrentFrame[
    BufferIndex+2]]--;
                  }
                  if (pSubBackground[BufferIndex] == 0)   //If
    this pixel part of background it is a dead pixel which we
    are trying to minimize
                  {
          DeadPixelCount--;
                  }

          BufferIndex+=4;
          }

}

for (int y=0; y<2; y++) //Loop through two columns on the
    right and subtract these
{
BufferIndex = ( (Data->Y4+Data->Height4+(y+1))*VideoXLength+
    Data->X4)*4; //Point to the right strip

     for (int x=0; x<Data->Width4; x++) //Subtract this row of
    pixels to the DelData histogram
          {

                  if ( O->Blue4[pCurrentFrame[BufferIndex]] > 0
    && O->Green4[pCurrentFrame[BufferIndex+1]] > 0 &&
                 O->Red4[pCurrentFrame[BufferIndex+2]] > 0)
                  {
                  DelData->Blue4[pCurrentFrame[BufferIndex]]++;
                  DelData->Green4[pCurrentFrame[BufferIndex
    +1]]++;
                  DelData->Red4[pCurrentFrame[BufferIndex+2]]++;
                  }
                  if (pSubBackground[BufferIndex] == 0)   //If
    this pixel part of background it is a dead pixel which we
    are trying to minimize
                  {
          DeadPixelCount++;
                  }

          BufferIndex+=4;
          }

}
```

```
return(CalculateImprovement(ObjectIndex,Data,DelData)+(
    DeadPixelCount*-6));

}

int ObjectTracker::CalculateImprovement(int  ObjectIndex,
    HistData* Data,HistData* DelData)
{
int Diff1;
int Diff2;
int Result=0;

int Sum1;
int Sum2;
int Sum3;

//See if the delta histograms yield an improvement overall in
    the closeness to the original histogram from the previous
    frame
  for (int i=0; i<256-2; i++)
  {
  Sum1 = PreliminaryObjects[ObjectIndex].Red1[i] +
    PreliminaryObjects[ObjectIndex].Red1[i+1] +
    PreliminaryObjects[ObjectIndex].Red1[i]+2;
  Sum2 = Data->Red1[i] + Data->Red1[i+1] + Data->Red1[i+2];
  Sum3 = DelData->Red1[i] + DelData->Red1[i+1] + DelData->Red1
    [i+2];

  //Diff1 = PreliminaryObjects[ObjectIndex].Red1[i] - Data->
    Red1[i];   //If we don't have enough of a spectrum -
  Diff1 = Sum1 - Sum2;
      if (Diff1>0) Result += Sum3;       //Can we get more?


  //Diff1 = PreliminaryObjects[ObjectIndex].Blue1[i] - Data->
    Blue1[i];
  //Diff2 = PreliminaryObjects[ObjectIndex].Blue1[i] - (Data->
    Blue1[i]+DelData->Blue1[i]);
   Sum1 = PreliminaryObjects[ObjectIndex].Blue1[i] +
    PreliminaryObjects[ObjectIndex].Blue1[i+1] +
    PreliminaryObjects[ObjectIndex].Blue1[i+2];
   Sum2 = Data->Blue1[i] + Data->Blue1[i+1] + Data->Blue1[i
    +2];
   Sum3 = DelData->Blue1[i] + DelData->Blue1[i+1] + DelData->
    Blue1[i+1];
          //if (Diff1>0) Result += DelData->Blue1[i];   //Can
    we get more?
```

```
          if (Diff1>0) Result += Sum3;   //Can we get more?


   //Diff1 = PreliminaryObjects[ObjectIndex].Green1[i] - Data->
    Green1[i];
   //Diff2 = PreliminaryObjects[ObjectIndex].Green1[i] - (Data
    ->Green1[i]+DelData->Green1[i]);
          Sum1 = PreliminaryObjects[ObjectIndex].Green1[i] +
    PreliminaryObjects[ObjectIndex].Green1[i+1] +
    PreliminaryObjects[ObjectIndex].Green1[i+2];
          Sum2 = Data->Green1[i] + Data->Green1[i+1] + Data->
    Green1[i+2];
          Sum3 = DelData->Green1[i] + DelData->Green1[i+1] +
    DelData->Green1[i+2];
                 //if (Diff1>0) Result += DelData->Green1[i];
        //Can we get more?
          if (Diff1>0) Result += Sum3;   //Can we get more?

   //Diff1 = PreliminaryObjects[ObjectIndex].Red2[i] - Data->
    Red2[i];
   ////Diff2 = PreliminaryObjects[ObjectIndex].Red2[i] - (Data
    ->Red2[i]+DelData->Red2[i]);
                 //if (Diff1>0) Result += DelData->Red2[i];
          //Can we get more?

   //

   //Diff1 = PreliminaryObjects[ObjectIndex].Blue2[i] - Data->
    Blue2[i];
   ////Diff2 = PreliminaryObjects[ObjectIndex].Blue2[i] - (Data
    ->Blue2[i]+DelData->Blue2[i]);
                 //if (Diff1>0) Result += DelData->Blue2[i];
         //Can we get more?

   //
   //Diff1 = PreliminaryObjects[ObjectIndex].Green2[i] - Data->
    Green2[i];
   ////Diff2 = PreliminaryObjects[ObjectIndex].Green2[i] - (
    Data->Green2[i]+DelData->Green2[i]);
                 //if (Diff1>0) Result += DelData->Green2[i];
        //Can we get more?

   //

   //Diff1 = PreliminaryObjects[ObjectIndex].Red3[i] - Data->
    Red3[i];
   ////Diff2 = PreliminaryObjects[ObjectIndex].Red3[i] - (Data
    ->Red3[i]+DelData->Red3[i]);
                 //if (Diff1>0) Result += DelData->Red3[i];
          //Can we get more?
   //

   //Diff1 = PreliminaryObjects[ObjectIndex].Blue3[i] - Data->
    Blue3[i];
   ////Diff2 = PreliminaryObjects[ObjectIndex].Blue3[i] - (Data
    ->Blue3[i]+DelData->Blue3[i]);
```

```
                    //if (Diff1>0) Result += DelData->Blue3[i];
          //Can we get more?

  //

  //Diff1 = PreliminaryObjects[ObjectIndex].Green3[i] - Data->
    Green3[i];
  ////Diff2 = PreliminaryObjects[ObjectIndex].Green3[i] - (
    Data->Green3[i]+DelData->Green3[i]);
                    //if (Diff1>0) Result += DelData->Green3[i];
         //Can we get more?

  //

  //Diff1 = PreliminaryObjects[ObjectIndex].Red4[i] - Data->
    Red4[i];
  ////Diff2 = PreliminaryObjects[ObjectIndex].Red4[i] - (Data
   ->Red4[i]+DelData->Red4[i]);
                    //if (Diff1>0) Result += DelData->Red4[i];
           //Can we get more?
  //

  //Diff1 = PreliminaryObjects[ObjectIndex].Blue4[i] - Data->
    Blue4[i];
  ////Diff2 = PreliminaryObjects[ObjectIndex].Blue4[i] - (Data
   ->Blue4[i]+DelData->Blue4[i]);
                    //if (Diff1>0) Result += DelData->Blue4[i];
          //Can we get more?
  //

  //Diff1 = PreliminaryObjects[ObjectIndex].Green4[i] - Data->
    Green4[i];
  ////Diff2 = PreliminaryObjects[ObjectIndex].Green4[i] - (
    Data->Green4[i]+DelData->Green4[i]);
                    //if (Diff1>0) Result += DelData->Green4[i];
         //Can we get more?
  //



  }
return(Result);
}





void ObjectTracker::UpdateObjectHistograms(int ObjectIndex)
{
HistData NewData;

ObjectFound* O;

int XPos;
int YPos;
```

```
int Width;
int Height;

XPos = PreliminaryObjects[ObjectIndex].PhysicalX;
YPos = PreliminaryObjects[ObjectIndex].PhysicalY;
Width = PreliminaryObjects[ObjectIndex].PhysicalWidth;
Height = PreliminaryObjects[ObjectIndex].PhysicalHeight;

//Calculate the bounding box coordinates of each sub window
NewData.X1 = XPos;
NewData.Y1 = YPos;
NewData.Width1 = Width;      //Bounding box 1 on the top left
    corner
NewData.Height1 = Height;

NewData.X2 = XPos+Width/2;
NewData.Y2 = YPos;
NewData.Width2 = Width/2;
NewData.Height2 = Height/2;

NewData.X3 = XPos;
NewData.Y3 = YPos+Height/2;
NewData.Width3 = Width/2;
NewData.Height3 = Height/2;

NewData.X4 = XPos+Width/2;
NewData.Y4 = YPos+Height/2;
NewData.Width4 = Width/2;
NewData.Height4 = Height/2;

//Find the new Histogram data
CreateHistogram(ObjectIndex,pSubBackground,pCurrentFrame,
    NewData.Red1,NewData.Green1,NewData.Blue1,NewData.X1,
    NewData.Y1,NewData.Width1,NewData.Height1);
//CreateHistogram(ObjectIndex,pSubBackground,pCurrentFrame,
    NewData.Red2,NewData.Green2,NewData.Blue2,NewData.X2,
    NewData.Y2,NewData.Width2,NewData.Height2);
//CreateHistogram(ObjectIndex,pSubBackground,pCurrentFrame,
    NewData.Red3,NewData.Green3,NewData.Blue3,NewData.X3,
    NewData.Y3,NewData.Width3,NewData.Height3);
//CreateHistogram(ObjectIndex,pSubBackground,pCurrentFrame,
    NewData.Red4,NewData.Green4,NewData.Blue4,NewData.X4,
    NewData.Y4,NewData.Width4,NewData.Height4);


//An object is found in a new location but its histogram will
    change slightly as it moves just as the background model
    allows changes in the background
//we should also allow for changes in the object histogram

O = &PreliminaryObjects[ObjectIndex];

O->TotalTrackingPixels = 0;

//Update the old histogram but don't give it all the new data.
     change it slowly.
    for (int i=0; i<256; i++)
    {
```

```
    O->Blue1[i] = O->Blue1[i]*0.7 + 0.3*NewData.Blue1[i];
    O->Green1[i] = O->Green1[i]*0.7 + 0.3*NewData.Green1[i];
    O->Red1[i] = O->Red1[i]*0.7 + 0.3*NewData.Red1[i];

    O->Blue2[i] = O->Blue2[i]*0.7 + 0.3*NewData.Blue2[i];
    O->Green2[i] = O->Green2[i]*0.7 + 0.3*NewData.Green2[i];
    O->Red2[i] = O->Red2[i]*0.7 + 0.3*NewData.Red2[i];

    O->Blue3[i] = O->Blue3[i]*0.7 + 0.3*NewData.Blue3[i];
    O->Green3[i] = O->Green3[i]*0.7 + 0.3*NewData.Green3[i];
    O->Red3[i] = O->Red3[i]*0.7 + 0.3*NewData.Red3[i];

    O->Blue4[i] = O->Blue4[i]*0.7 + 0.3*NewData.Blue4[i];
    O->Green4[i] = O->Green4[i]*0.7 + 0.3*NewData.Green4[i];
    O->Red4[i] = O->Red4[i]*0.7 + 0.3*NewData.Red4[i];

    /* O->Blue1[i] = NewData.Blue1[i];
    O->Green1[i] = NewData.Green1[i];
    O->Red1[i] = NewData.Red1[i];

    O->Blue2[i] = NewData.Blue2[i];
    O->Green2[i] = NewData.Green2[i];
    O->Red2[i] = NewData.Red2[i];

    O->Blue3[i] = NewData.Blue3[i];
    O->Green3[i] = NewData.Green3[i];
    O->Red3[i] = NewData.Red3[i];

    O->Blue4[i] = NewData.Blue4[i];
    O->Green4[i] = NewData.Green4[i];
    O->Red4[i] = NewData.Red4[i];
*/

    O->TotalTrackingPixels+= O->Blue1[i];
    O->TotalTrackingPixels+= O->Blue2[i];
    O->TotalTrackingPixels+= O->Blue3[i];
    O->TotalTrackingPixels+= O->Blue4[i];


    }




}








//A Bresenham algorithm I copied from an old game programming
    book
void ObjectTracker::DrawLine(BYTE* pBuffer,int x1,int y1,int
    x2,int y2,BYTE Red,BYTE Green,BYTE Blue) //General
    algorithm to draw lines in the given video buffer
{
int y_unit,x_unit;
```

```
int offset=(y1*this->VideoXLength+x1)*4; //*4 for 4 bytes per
    pixel
int ydiff = y2-y1;

  if (ydiff<0)
  {
  ydiff=-ydiff;
  y_unit=-VideoXLength*4;
  }
  else y_unit = VideoXLength*4;

  int xdiff = x2-x1;

  if (xdiff<0)
  {
  xdiff=-xdiff;
  x_unit=-4;
  }
  else x_unit=4;

  int error_term = 0;

  if (xdiff>ydiff)
  {
  int length=xdiff+1;

      for (int i=0; i<length; i++)
          {
      pBuffer[offset]=Blue;
          pBuffer[offset+1]=Green;
          pBuffer[offset+2]=Red;
      offset+=x_unit;

          error_term+=ydiff;

            if (error_term>xdiff)
                {
        error_term-=xdiff;
                offset+=y_unit;
                }

          }
  }
  else
  {
  int length=ydiff+1;

      for (int i=0; i<length; i++)
          {
      pBuffer[offset]=Blue;
          pBuffer[offset+1]=Green;
          pBuffer[offset+2]=Red;

      offset+=y_unit;

          error_term+=xdiff;

            if (error_term>0)
                {
        error_term-=ydiff;
                offset+=x_unit;
```

```
                    }
            }
    }


}



//Returns the the ratio of foreground pixels to background
    pixels inside an object
//This is used to scale the width and height of an object
double ObjectTracker::ReturnPixelRatio(int ObjectIndex,BYTE*
    pSubBackground)
{
ObjectFound* O;

O = &PreliminaryObjects[ObjectIndex];

static int XPoints1[500];  //Data for line 1
static int YPoints1[500];

static int XPoints2[500];  //Line 2 etc...
static int YPoints2[500];

static int XPoints3[500];
static int YPoints3[500];

static int XPoints4[500];
static int YPoints4[500];

static int* XPointsArrays[4];
static int* YPointsArrays[4];

static int NPoints[4];


int CentreX = O->CentreX; //The current centre point of the
    selected object
int CentreY = O->CentreY;

int RelX;
int RelY;

double PixelsFound = 0; //Number of white pixels found in
    pSubBackground and considered
                                            //part of the current
     object and included in mean shift

double PixelsCounted = 0; //Total pixels searched

int XSum = 0; //Sum of all the X-Coordinates of all pixels
    found in mean shift
int YSum = 0; //Same as above but for all Y Coordinates


int LowestY;  //The lowest and highest points in this objects
    bounding box
int HighestY;
```

```
XPointsArrays[1] = XPoints1;
YPointsArrays[1] = YPoints1;

XPointsArrays[2] = XPoints2;
YPointsArrays[2] = YPoints2;

XPointsArrays[3] = XPoints3;
YPointsArrays[3] = YPoints3;

XPointsArrays[4] = XPoints4;
YPointsArrays[4] = YPoints4;


LowestY = O->TY1;
HighestY = O->TY1;

if (O->TY2 > LowestY) LowestY = O->TY2;
if (O->TY3 > LowestY) LowestY = O->TY3;
if (O->TY4 > LowestY) LowestY = O->TY4;

if (O->TY2 < HighestY) HighestY = O->TY2;
if (O->TY3 < HighestY) HighestY = O->TY3;
if (O->TY4 < HighestY) HighestY = O->TY4;

int RowsToDraw = LowestY - HighestY;

for (int i=HighestY; i<LowestY+1; i++)
{

YPointsArrays[1][i] = 0;
YPointsArrays[2][i] = 0;
YPointsArrays[3][i] = 0;
YPointsArrays[4][i] = 0;

}


NPoints[1] = ReturnVerticalPoints(O->TX1,O->TY1,O->TX2,O->TY2,
   XPoints1,YPoints1);
NPoints[2] = ReturnVerticalPoints(O->TX1,O->TY1,O->TX3,O->TY3,
   XPoints2,YPoints2);
NPoints[3] = ReturnVerticalPoints(O->TX3,O->TY3,O->TX4,O->TY4,
   XPoints3,YPoints3);
NPoints[4] = ReturnVerticalPoints(O->TX4,O->TY4,O->TX2,O->TY2,
   XPoints4,YPoints4);


int FirstX; //First x point found
BOOL FoundFirstX;
BOOL FoundSecondX;
int SecondX; //Second x point found
int BufferIndex = HighestY*VideoXLength*4;
int X_Inc = 1;
int Length = 0;
int XPos = 0;

for (int y=HighestY; y<LowestY+1; y++)
{
FoundFirstX = FALSE;
```

```
FoundSecondX = FALSE;

  for (int l=1; l<4+1; l++)
  {
    if (YPointsArrays[l][y] == 1 && FoundFirstX == FALSE)
        {
    FoundFirstX = TRUE;
        FirstX = XPointsArrays[l][y];
        }
        else if (FoundFirstX == TRUE && YPointsArrays[l][y] ==
    1 && XPointsArrays[l][y] != FirstX)
        {
    FoundSecondX = TRUE;
    SecondX = XPointsArrays[l][y];
        }

  }

    if (FoundSecondX == TRUE)
        {
    //Grab the points between the x coordinates found
          if (FirstX<SecondX) X_Inc = 1;
          else X_Inc = -1;

      Length = abs(FirstX-SecondX)+1;
      XPos = FirstX;

      RelX = XPos-CentreX;
          RelY = CentreY-y;

      for (int x=0; x<Length+1; x++)
          {

            //If a white pixel it is likely part of this
   object so include in mean shift
            if (pSubBackground[BufferIndex+XPos*4] == 255)
                {
                PixelsFound++;
                }

      PixelsCounted++;

      XPos+=X_Inc;
          RelX+=X_Inc;
          }

        }
        else
        {

    //Just grab the 1 pixel

    RelX = XPos-CentreX;
        RelY = CentreY-y;

                if (FoundFirstX == TRUE)
                {
                XPos = FirstX;

          if (pSubBackground[BufferIndex+XPos*4] == 255)
                {
```

```
                PixelsFound++;
                }
        PixelsCounted++;
                }

        }
        if (FoundFirstX == FALSE)
        {
        //MessageBox(NULL,"Error in ObjectTracker::
    AdvancedMeanShift(...) Not Enough Points Found","Error",
    MB_OK);
        }
BufferIndex+=VideoXLength*4; //Point to next row to grab
    pixels from
}

if (PixelsFound == 0) PixelsFound = 1;
if (PixelsCounted == 0) PixelsCounted = 1;

return(PixelsFound*1.2/PixelsCounted);
}




void ObjectTracker::ScaleObject(int ObjectIndex)
{
ObjectFound* O = &PreliminaryObjects[ObjectIndex];
double ExistingPixelRatio = 0;
double NewPixelRatio = 0;

ExistingPixelRatio = ReturnPixelRatio(ObjectIndex,
    pSubBackground);

//Increase the Height and see if this includes more foreground
     pixels compared to background
O->Height += 3;

CalcTranslatedCoordinates(ObjectIndex); //Calculate new
    bounding box coordinates

NewPixelRatio = ReturnPixelRatio(ObjectIndex,pSubBackground);


  //If no improvement then return coordinates back to normal
  if (NewPixelRatio*1.05 < ExistingPixelRatio)
  {
  O->Height-=3;
  CalcTranslatedCoordinates(ObjectIndex);



//  ExistingPixelRatio = ReturnPixelRatio(ObjectIndex,
```

```
    pSubBackground);

//
////Increase the Height and see if this includes more
    foreground pixels compared to background
      if (O->Height>=20)
      {

      O->Height -= 3;

      CalcTranslatedCoordinates(ObjectIndex); //Calculate new
    bounding box coordinates

      NewPixelRatio = ReturnPixelRatio(ObjectIndex,
    pSubBackground);

        //If no improvement then return coordinates back to
    normal
        if (NewPixelRatio < ExistingPixelRatio )
        {
        O->Height+=3;
        CalcTranslatedCoordinates(ObjectIndex);
        }

      }

  }



//Try to scale the object along the width

ExistingPixelRatio = ReturnPixelRatio(ObjectIndex,
    pSubBackground);

//Increase the Width and see if this includes more foreground
    pixels compared to background
O->Width += 3;

CalcTranslatedCoordinates(ObjectIndex); //Calculate new
    bounding box coordinates

NewPixelRatio = ReturnPixelRatio(ObjectIndex,pSubBackground);


  //If no improvement then return coordinates back to normal
  if (NewPixelRatio*1.05 < ExistingPixelRatio)
  {
  O->Width-=3;
  CalcTranslatedCoordinates(ObjectIndex);


////Decrease the Width and see if this includes more
    foreground pixels compared to background
      //if (O->Width>=10)
      //{

      //O->Width -= 3;

      //CalcTranslatedCoordinates(ObjectIndex); //Calculate new
    bounding box coordinates
```

```
    //NewPixelRatio = ReturnPixelRatio(ObjectIndex,
   pSubBackground);

    // //If no improvement then return coordinates back to
   normal
    //  if (NewPixelRatio < ExistingPixelRatio )
    //  {
    //  O->Width+=3;
    //  CalcTranslatedCoordinates(ObjectIndex);
    //  }

    //}

  }




}





//A Better mean shift that selects pixels for the mean shift
   from within a bounding box that
//maybe rotated to a given angle
void ObjectTracker::AdvancedMeanShift(int ObjectIndex,BYTE*
   pSubBackground)
{
ObjectFound* O;

O = &PreliminaryObjects[ObjectIndex];

static int XPoints1[500];  //Data for line 1
static int YPoints1[500];

static int XPoints2[500];  //Line 2 etc...
static int YPoints2[500];

static int XPoints3[500];
static int YPoints3[500];

static int XPoints4[500];
static int YPoints4[500];

static int* XPointsArrays[4];
static int* YPointsArrays[4];

static int NPoints[4];


int CentreX = O->CentreX; //The current centre point of the
   selected object
int CentreY = O->CentreY;

int RelX;
int RelY;

int PixelsFound = 0; //Number of white pixels found in
```

```
    pSubBackground and considered
                                        //part of the current
     object and included in mean shift
int TotalPixelsFound = 0;

int XSum = 0; //Sum of all the X-Coordinates of all pixels
    found in mean shift
int YSum = 0; //Same as above but for all Y Coordinates

int LowestY;  //The lowest and highest points in this objects
    bounding box
int HighestY;

XPointsArrays[1] = XPoints1;
YPointsArrays[1] = YPoints1;

XPointsArrays[2] = XPoints2;
YPointsArrays[2] = YPoints2;

XPointsArrays[3] = XPoints3;
YPointsArrays[3] = YPoints3;

XPointsArrays[4] = XPoints4;
YPointsArrays[4] = YPoints4;


LowestY = O->TY1;
HighestY = O->TY1;

if (O->TY2 > LowestY) LowestY = O->TY2;
if (O->TY3 > LowestY) LowestY = O->TY3;
if (O->TY4 > LowestY) LowestY = O->TY4;

if (O->TY2 < HighestY) HighestY = O->TY2;
if (O->TY3 < HighestY) HighestY = O->TY3;
if (O->TY4 < HighestY) HighestY = O->TY4;

int RowsToDraw = LowestY - HighestY;

for (int i=HighestY; i<LowestY+1; i++)
{

YPointsArrays[1][i] = 0;
YPointsArrays[2][i] = 0;
YPointsArrays[3][i] = 0;
YPointsArrays[4][i] = 0;

}


NPoints[1] = ReturnVerticalPoints(O->TX1,O->TY1,O->TX2,O->TY2,
    XPoints1,YPoints1);
NPoints[2] = ReturnVerticalPoints(O->TX1,O->TY1,O->TX3,O->TY3,
    XPoints2,YPoints2);
NPoints[3] = ReturnVerticalPoints(O->TX3,O->TY3,O->TX4,O->TY4,
    XPoints3,YPoints3);
NPoints[4] = ReturnVerticalPoints(O->TX4,O->TY4,O->TX2,O->TY2,
    XPoints4,YPoints4);
```

```
int FirstX; //First x point found
BOOL FoundFirstX;
BOOL FoundSecondX;
int SecondX; //Second x point found
int BufferIndex = HighestY*VideoXLength*4;
int X_Inc = 1;
int Length = 0;
int XPos = 0;

for (int y=HighestY; y<LowestY+1; y++)
{
FoundFirstX = FALSE;
FoundSecondX = FALSE;

  for (int l=1; l<4+1; l++)
  {
    if (YPointsArrays[l][y] == 1 && FoundFirstX == FALSE)
        {
    FoundFirstX = TRUE;
        FirstX = XPointsArrays[l][y];
        }
        else if (FoundFirstX == TRUE && YPointsArrays[l][y] ==
    1 && XPointsArrays[l][y] != FirstX)
        {
    FoundSecondX = TRUE;
    SecondX = XPointsArrays[l][y];
        }

  }

    if (FoundSecondX == TRUE)
        {
    //Grab the points between the x coordinates found
          if (FirstX<SecondX) X_Inc = 1;
          else X_Inc = -1;

      Length = abs(FirstX-SecondX)+1;
      XPos = FirstX;

      RelX = XPos-CentreX;
          RelY = CentreY-y;

      for (int x=0; x<Length+1; x++)
          {

            //If a white pixel it is likely part of this
  object so include in mean shift
            if (pSubBackground[BufferIndex+(XPos*4)] == 255)
                {
       XSum+=RelX;
            YSum+=-RelY;
                PixelsFound++;

                }

      TotalPixelsFound++;

      XPos+=X_Inc;
```

```
            RelX+=X_Inc;
            }

          }
          else
          {

     //Just grab the 1 pixel

     RelX = XPos-CentreX;
          RelY = CentreY-y;

                    if (FoundFirstX == TRUE)
                    {
                    XPos = FirstX;

            if (pSubBackground[BufferIndex+(XPos*4)] == 255)
                  {
            XSum+=RelX;
                 YSum+=RelY;
                  PixelsFound++;
                  }

           TotalPixelsFound++;
                    }

          }


BufferIndex+=VideoXLength*4; //Point to next row to grab
    pixels from
}

if (PixelsFound == 0) PixelsFound = 1;
if (TotalPixelsFound == 0) PixelsFound = 1;
  //if (PixelsFound > 10)
  //{
  O->CentreX = O->CentreX + (XSum/PixelsFound)*1;
  O->CentreY = O->CentreY + (YSum/PixelsFound)*1;
  //}

  if (O->CentreX>VideoXLength-1) O->CentreX = VideoXLength-1;
  if (O->CentreX<0) O->CentreX = 0;

  if (O->CentreY>VideoYLength-1) O->CentreY = VideoYLength-1;
  if (O->CentreY<0) O->CentreY = 0;

O->TotalTrackingPixels = PixelsFound;
O->TotalPixelsFound = TotalPixelsFound;
}




//Returns each vertical point on the line given by the X and Y
     coordinates and stores the vertical points
//in XPoints and YPoints and also the function returns the
```

```
    number of vertical points found.
//Each point in the XPoints arrays corresponds to a row on the
     screen and the value of that element indicates
//the x-coordinate where the line is on that row.
//Each element of the YPoints array also corresponds to a row
    on the screen and is set to 1 if the line is on that row or
     not
int ObjectTracker::ReturnVerticalPoints(int X1,int Y1,int X2,
    int Y2,int* XPoints,int* YPoints)
{

int y_unit,x_unit;
int X = X1;
int Y = Y1;
int PointsRecorded = 0;

int ydiff = Y2-Y1;

   if (ydiff<0)
   {
   ydiff=-ydiff;
   y_unit = -1;
   }
   else y_unit = 1;

   int xdiff = X2-X1;

   if (xdiff<0)
   {
   xdiff=-xdiff;
   x_unit = -1;
   }
   else x_unit = 1;

   int error_term = 0;

   if (xdiff>ydiff)
   {
   int length=xdiff+1;

         for (int i=0; i<length; i++)
               {
        X+=x_unit;

               error_term+=ydiff;

                  if (error_term>xdiff)
                        {
           error_term-=xdiff;

                        XPoints[Y] = X;
                        YPoints[Y] = 1;

                        PointsRecorded++;
                        Y+=y_unit;
                        }

               }
   }
   else
   {
```

```
  int length=ydiff+1;

      for (int i=0; i<length; i++)
          {


          XPoints[Y] = X;
          YPoints[Y] = 1;
          PointsRecorded++;

          Y+=y_unit;

          error_term+=xdiff;

             if (error_term>0)
                 {
        error_term-=ydiff;
                 X+=x_unit;
                 }
          }

  }

return(PointsRecorded);
}








void ObjectTracker::DrawObjectTracks(BYTE* pBuffer)
{
ObjectFound* O;
int StartingSampleIndex = 0;
int BufferIndex = 0;
int CurrentIndex = 0;

     for (int i=0; i<this->MaxTrackableObjects; i++)
         {
         O = &PreliminaryObjects[i];

       if (this->Validated[i] == 0) continue; //If not
   validated don't draw the tracks for this object

     StartingSampleIndex = MSamples[i].SampleIndex; //Remember
   when to stop
     CurrentIndex = MSamples[i].SampleIndex;


             for (int l=0; l<Movement::MaxSamples; l++)
             {

                     if (l >= MSamples[i].SamplesTaken) break;
   //Drawn all samples exit loop

                         if (CurrentIndex == 0) CurrentIndex =
   Movement::MaxSamples-1;
                         else CurrentIndex--;
```

```
                BufferIndex = (MSamples[i].CentreYSamples[
    CurrentIndex]*VideoXLength+MSamples[i].CentreXSamples[
    CurrentIndex])*4;

                pBuffer[BufferIndex] = 255;
                pBuffer[BufferIndex+1] = 200;
                pBuffer[BufferIndex+2] = 255;

            if (CurrentIndex == StartingSampleIndex) break; //
    back at starting position so exit

                }


        }


}






void ObjectTracker::DrawCounter(Graphics* GObject)
{

static wchar_t Number[10];
static size_t BufferSize = 10;

GObject->DrawLine(pPen1,CounterX1,CounterY1,CounterX2,
    CounterY2);
GObject->DrawLine(pPen2,SpeedX1,SpeedY1,SpeedX2,SpeedY2);

pPointF->X = CounterX1+5;
pPointF->Y = CounterY1+5;


//swprintf_s(Number,BufferSize,L"Count = %d",VehiclesCounted);

//GObject->DrawString(Number, -1, &font, pointF, &solidBrush);


}



void ObjectTracker::CountVehicles()
{
double Counter1Distance;
double Counter2Distance;
double Speed1Distance;
double Speed2Distance;
double CounterLength;
double SpeedLength;
```

```
ObjectFound* O;

CounterLength = sqrt( double((CounterX1-CounterX2)*(CounterX1-
    CounterX2) + (CounterY1-CounterY2)*(CounterY1-CounterY2)) )
    ;
SpeedLength = sqrt( double((SpeedX1-SpeedX2)*(SpeedX1-SpeedX2)
     + (SpeedY1-SpeedY2)*(SpeedY1-SpeedY2)) );

     for (int i=0; i<MaxTrackableObjects; i++)
          {

             if (Validated[i] == 0) continue; //Don't look at
    objects not validated

    O = &PreliminaryObjects[i];

     //Calculate distances from ends of counter line to the
    object
     Counter1Distance = sqrt( double((CounterX1-O->CentreX)*(
    CounterX1-O->CentreX) + (CounterY1-O->CentreY)*(CounterY1-O
    ->CentreY)) );
          Counter2Distance = sqrt( double((CounterX2-O->CentreX)
    *(CounterX2-O->CentreX) + (CounterY2-O->CentreY)*(CounterY2
    -O->CentreY)) );

     Speed1Distance = sqrt( double((SpeedX1-O->CentreX)*(
    SpeedX1-O->CentreX) + (SpeedY1-O->CentreY)*(SpeedY1-O->
    CentreY)) );
     Speed2Distance = sqrt( double((SpeedX2-O->CentreX)*(
    SpeedX2-O->CentreX) + (SpeedY2-O->CentreY)*(SpeedY2-O->
    CentreY)) );

        if (Counter1Distance<0.9*CounterLength &&
    Counter2Distance<0.9*CounterLength &&
                      abs(Counter1Distance+Counter2Distance -
    CounterLength) < 3 && O->IsObjectCounted1 == FALSE)
             {
       //Vehicle is crossing the line and needs to be counted
             O->IsObjectCounted1 = TRUE;
       O->CounterTimeStamp = timeGetTime();
             VehiclesCounted++;
             }

             if (Speed1Distance<0.9*SpeedLength && Speed2Distance
    <0.9*SpeedLength &&
                      abs(Speed1Distance+Speed2Distance -
    SpeedLength) < 3 && O->IsSpeedLineCrossed == FALSE)
             {
       //Vehicle is crossing the line and needs to be counted
             O->IsSpeedLineCrossed = TRUE;
       O->SpeedTimeStamp = timeGetTime();


             }

           if (O->IsObjectCounted1 == TRUE && O->
    IsSpeedLineCrossed == TRUE)
```

```
            {
        //Make speed estimate

        O->Speed = (CounterToSpeedLineDistance/double(abs(O->
    SpeedTimeStamp-O->CounterTimeStamp))) * 1000;

            O->Speed = (O->Speed*3.6);
            }

        }


}




ObjectTracker::~ObjectTracker()
{
delete TracksData;
delete MSamples;
delete TimesSeen;
delete PObjectsPresent;
delete PreliminaryObjects;

}
```

B.11 ImageProcessing.cpp

```
#include "VisionX.h"




//Increases the resolution of an image by a factor of 4 by
    using pixel averaging
void Image4x4Blender(unsigned int* pDestImage,unsigned int*
    pSourceImage,Rect &SourceImageSize)
{
int SourceBufferPos;
int DestBufferPos;

unsigned int SourceRed;
unsigned int SourceBlue;
unsigned int SourceGreen;

unsigned int DestRed2;
unsigned int DestGreen2;
unsigned int DestBlue2;

unsigned int DestRed;
unsigned int DestBlue;
unsigned int DestGreen;

unsigned int DestRed3;
unsigned int DestBlue3;
unsigned int DestGreen3;
```

```
unsigned long PixelCount;

unsigned int SourceImageWidth;
unsigned int SourceImageHeight;

unsigned int DestBufferStride;

double SourceContribution;
double DestContribution;

SourceContribution = 0.6;
DestContribution = (1-SourceContribution)/3;


SourceImageWidth=SourceImageSize.Width;
SourceImageHeight=SourceImageSize.Height;

DestBufferStride = SourceImageWidth*2;

DestBufferPos=0;
SourceBufferPos=0;


for (int y=0; y<SourceImageHeight; y++)
{

  for (int x=0; x<SourceImageWidth; x++)
  {
  DestRed=0;
  DestBlue=0;
  DestGreen=0;
  PixelCount=0;

  //Calculate top left sub-pixel color of source pixel

    //if (x>0) //Get Color from pixel to the left of this one
   if possible
        //{

        //DestRed   = DestRed   + (pSourceImage[
   SourceBufferPos-1]&0x00FF0000)>>16;
        //DestGreen = DestGreen + (pSourceImage[
   SourceBufferPos-1]&0x0000FF00)>>8;
        //DestBlue  = DestBlue  + (pSourceImage[
   SourceBufferPos-1]&0x000000FF)>>0;


        //PixelCount++;
        //}
    //if (y>0) //Get Color from pixel above this one
        //{

        //DestRed   = DestRed   + (pSourceImage[
   SourceBufferPos-SourceImageWidth]&0x00FF0000)>>16;
        //DestGreen = DestGreen + (pSourceImage[
   SourceBufferPos-SourceImageWidth]&0x0000FF00)>>8;
        //DestBlue  = DestBlue  + (pSourceImage[
```

```
SourceBufferPos-SourceImageWidth]&0x000000FF)>>0;

      //PixelCount++;
      //}
 if (x>0 && y>0) //Get Color from pixel left of this one
and above
      {

      DestRed    = (pSourceImage[SourceBufferPos-
SourceImageWidth-1]&0x00FF0000)>>16;
      DestGreen = (pSourceImage[SourceBufferPos-
SourceImageWidth-1]&0x0000FF00)>>8;
      DestBlue   = (pSourceImage[SourceBufferPos-
SourceImageWidth-1]&0x000000FF)>>0;

 DestRed2 =  (pSourceImage[SourceBufferPos-SourceImageWidth
]&0x00FF0000)>>16;
 DestGreen2 = (pSourceImage[SourceBufferPos-
SourceImageWidth]&0x0000FF00)>>8;
      DestBlue2   = (pSourceImage[SourceBufferPos-
SourceImageWidth]&0x000000FF)>>0;

 DestRed3 =  (pSourceImage[SourceBufferPos-1]&0x00FF0000)
>>16;
 DestGreen3 = (pSourceImage[SourceBufferPos-1]&0x0000FF00)
>>8;
      DestBlue3   = (pSourceImage[SourceBufferPos-1]&0
x000000FF)>>0;

      PixelCount=2;
      }


      if (PixelCount == 0) //If pixel on edge of screen in
awkward position
      {
 PixelCount = 1;

 DestRed    = (pSourceImage[SourceBufferPos]&0x00FF0000)
>>16;
      DestGreen = (pSourceImage[SourceBufferPos]&0x0000FF00)
>>8;
      DestBlue   = (pSourceImage[SourceBufferPos]&0x000000FF)
>>0;

      }


 SourceRed    = (pSourceImage[SourceBufferPos]&0x00FF0000)
>>16;
      SourceGreen = (pSourceImage[SourceBufferPos]&0
x0000FF00)>>8;
      SourceBlue   = (pSourceImage[SourceBufferPos]&0
x000000FF)>>0;


      if (PixelCount == 2)
      {
      DestRed    = SourceRed*SourceContribution + DestRed *
DestContribution + DestRed2*DestContribution + DestRed3*
DestContribution;
      DestGreen = SourceGreen*SourceContribution + DestGreen
```

```
*DestContribution + DestGreen2*DestContribution +
DestGreen3*DestContribution;
     DestBlue  = SourceBlue*SourceContribution + DestBlue*
DestContribution + DestBlue2*DestContribution + DestBlue3*
DestContribution;

     pDestImage[DestBufferPos] = ((DestRed<<16)+(DestGreen
<<8) + (DestBlue<<0));
     }


//Calculate bottom left sub-pixel color of source pixel
DestRed=0;
DestBlue=0;
DestGreen=0;
PixelCount=0;

 //if (x>0) //Get Color from pixel to the left of this one
 if possible
 //{

     //DestRed   = DestRed   + (pSourceImage[
 SourceBufferPos-1]&0x00FF0000)>>16;
     //DestGreen = DestGreen + (pSourceImage[
 SourceBufferPos-1]&0x0000FF00)>>8;
     //DestBlue  = DestBlue  + (pSourceImage[
 SourceBufferPos-1]&0x000000FF)>>0;

     //PixelCount++;
 //}
 if (x>0 && y<(SourceImageHeight-1)) //Get color from pixel
 to left and down one
 {

     DestRed   = (pSourceImage[SourceBufferPos+
SourceImageWidth-1]&0x00FF0000)>>16;
     DestGreen = (pSourceImage[SourceBufferPos+
SourceImageWidth-1]&0x0000FF00)>>8;
     DestBlue  = (pSourceImage[SourceBufferPos+
SourceImageWidth-1]&0x000000FF)>>0;

     DestRed2   = (pSourceImage[SourceBufferPos+
SourceImageWidth]&0x00FF0000)>>16;
     DestGreen2 = (pSourceImage[SourceBufferPos+
SourceImageWidth]&0x0000FF00)>>8;
     DestBlue2  = (pSourceImage[SourceBufferPos+
SourceImageWidth]&0x000000FF)>>0;

     DestRed3   = (pSourceImage[SourceBufferPos-1]&0
x00FF0000)>>16;
     DestGreen3 = (pSourceImage[SourceBufferPos-1]&0
x0000FF00)>>8;
     DestBlue3  = (pSourceImage[SourceBufferPos-1]&0
x000000FF)>>0;

     PixelCount=2;
 }
 //if (y<(SourceImageHeight-1)) //Get Color from pixel
```

```
below the current one
 //{
      //DestRed   = DestRed   + (pSourceImage[
SourceBufferPos+SourceImageWidth]&0x00FF0000)>>16;
      //DestGreen = DestGreen + (pSourceImage[
SourceBufferPos+SourceImageWidth]&0x0000FF00)>>8;
      //DestBlue  = DestBlue  + (pSourceImage[
SourceBufferPos+SourceImageWidth]&0x000000FF)>>0;

      //PixelCount++;
 //}
 if (PixelCount == 0) //If pixel on edge of screen in
awkward position
      {

      DestRed   = DestRed   + (pSourceImage[SourceBufferPos
]&0x00FF0000)>>16;
      DestGreen = DestGreen + (pSourceImage[SourceBufferPos
]&0x0000FF00)>>8;
      DestBlue  = DestBlue  + (pSourceImage[SourceBufferPos
]&0x000000FF)>>0;

      PixelCount = 1;
      }
 //Calculate the average value of these pixels
 //DestRed  = DestRed / PixelCount;
      //DestGreen = DestGreen / PixelCount;
      //DestBlue = DestBlue / PixelCount;


 SourceRed   = (pSourceImage[SourceBufferPos]&0x00FF0000)
>>16;
      SourceGreen = (pSourceImage[SourceBufferPos]&0
x0000FF00)>>8;
      SourceBlue  = (pSourceImage[SourceBufferPos]&0
x000000FF)>>0;


      if (PixelCount == 2)
      {

      DestRed   = SourceRed*SourceContribution + DestRed *
DestContribution + DestRed2*DestContribution + DestRed3*
DestContribution;
      DestGreen = SourceGreen*SourceContribution + DestGreen
*DestContribution + DestGreen2*DestContribution +
DestGreen3*DestContribution;
      DestBlue  = SourceBlue*SourceContribution + DestBlue*
DestContribution + DestBlue2*DestContribution + DestBlue3*
DestContribution;

 pDestImage[DestBufferPos+DestBufferStride] = (DestRed<<16)
+(DestGreen<<8) + (DestBlue<<0);

      }


//Calculate the top right sub-pixel color of source pixel
DestRed=0;
```

```
DestBlue=0;
DestGreen=0;
PixelCount=0;

  //if (y>0) //Grab the pixel above the current source pixel
  //{

      //DestRed   = DestRed   + (pSourceImage[
 SourceBufferPos-SourceImageWidth]&0x00FF0000)>>16;
      //DestGreen = DestGreen + (pSourceImage[
 SourceBufferPos-SourceImageWidth]&0x0000FF00)>>8;
      //DestBlue  = DestBlue  + (pSourceImage[
 SourceBufferPos-SourceImageWidth]&0x000000FF)>>0;

      //PixelCount++;
 // }
  if (y>0 && x<(SourceImageWidth-1)) //Grab pixel above and
 to the right
  {

      DestRed   = (pSourceImage[SourceBufferPos-
 SourceImageWidth+1]&0x00FF0000)>>16;
      DestGreen = (pSourceImage[SourceBufferPos-
 SourceImageWidth+1]&0x0000FF00)>>8;
      DestBlue  = (pSourceImage[SourceBufferPos-
 SourceImageWidth+1]&0x000000FF)>>0;

      DestRed2   = (pSourceImage[SourceBufferPos-
 SourceImageWidth]&0x00FF0000)>>16;
      DestGreen2 = (pSourceImage[SourceBufferPos-
 SourceImageWidth]&0x0000FF00)>>8;
      DestBlue2  = (pSourceImage[SourceBufferPos-
 SourceImageWidth]&0x000000FF)>>0;

      DestRed3   = (pSourceImage[SourceBufferPos+1]&0
 x00FF0000)>>16;
      DestGreen3 = (pSourceImage[SourceBufferPos+1]&0
 x0000FF00)>>8;
      DestBlue3  = (pSourceImage[SourceBufferPos+1]&0
 x000000FF)>>0;


      PixelCount=2;
  }
 // if (x<(SourceImageWidth-1)) //Grab pixel to the right
 // {

      //DestRed   = DestRed   + (pSourceImage[
 SourceBufferPos+1]&0x00FF0000)>>16;
      //DestGreen = DestGreen + (pSourceImage[
 SourceBufferPos+1]&0x0000FF00)>>8;
      //DestBlue  = DestBlue  + (pSourceImage[
 SourceBufferPos+1]&0x000000FF)>>0;

  //PixelCount++;
  //}

      if (PixelCount == 0) //If pixel on edge of screen in
 awkward position
```

```
      {
 PixelCount = 1;

      DestRed   = DestRed   + (pSourceImage[SourceBufferPos
]&0x00FF0000)>>16;
      DestGreen = DestGreen + (pSourceImage[SourceBufferPos
]&0x0000FF00)>>8;
      DestBlue  = DestBlue  + (pSourceImage[SourceBufferPos
]&0x000000FF)>>0;

      }

 //Calculate the average value of these pixels
// DestRed  = DestRed / PixelCount;
// DestGreen = DestGreen / PixelCount;
// DestBlue = DestBlue / PixelCount;

 SourceRed   = (pSourceImage[SourceBufferPos]&0x00FF0000)
>>16;
      SourceGreen = (pSourceImage[SourceBufferPos]&0
x0000FF00)>>8;
      SourceBlue  = (pSourceImage[SourceBufferPos]&0
x000000FF)>>0;


      if (PixelCount == 2)
      {
      //DestRed = SourceRed*0.5 + DestRed * 0.5;
      //DestGreen = SourceGreen*0.5 + DestGreen * 0.5;
      //DestBlue = SourceBlue*0.5 + DestBlue * 0.5;
      DestRed   = SourceRed*SourceContribution + DestRed *
DestContribution + DestRed2*DestContribution + DestRed3*
DestContribution;
      DestGreen = SourceGreen*SourceContribution + DestGreen
*DestContribution + DestGreen2*DestContribution +
DestGreen3*DestContribution;
      DestBlue  = SourceBlue*SourceContribution + DestBlue*
DestContribution + DestBlue2*DestContribution + DestBlue3*
DestContribution;

      pDestImage[DestBufferPos+1] = (DestRed<<16)+(DestGreen
<<8) + (DestBlue<<0);

      }




//Calculate the bottom right sub-pixel color of source pixel
DestRed=0;
DestBlue=0;
DestGreen=0;
PixelCount=0;


  //if (x<(SourceImageWidth-1)) //Grab pixel to the right if
  possible
  //{

  //DestRed   = DestRed   + (pSourceImage[SourceBufferPos
+1]&0x00FF0000)>>16;
      //DestGreen = DestGreen + (pSourceImage[
```

```
    SourceBufferPos+1]&0x0000FF00)>>8;
        //DestBlue  = DestBlue  + (pSourceImage[
    SourceBufferPos+1]&0x000000FF)>>0;

     //PixelCount++;
     //}
     if ( (x<SourceImageWidth-1) && (y<(SourceImageHeight-1)) )
     //Grab the pixel to right and below
     {

        DestRed   = (pSourceImage[SourceBufferPos+
    SourceImageWidth+1]&0x00FF0000)>>16;
        DestGreen = (pSourceImage[SourceBufferPos+
    SourceImageWidth+1]&0x0000FF00)>>8;
        DestBlue  = (pSourceImage[SourceBufferPos+
    SourceImageWidth+1]&0x000000FF)>>0;

        DestRed2   = (pSourceImage[SourceBufferPos+1]&0
    x00FF0000)>>16;
        DestGreen2 = (pSourceImage[SourceBufferPos+1]&0
    x0000FF00)>>8;
        DestBlue2  = (pSourceImage[SourceBufferPos+1]&0
    x000000FF)>>0;

        DestRed3   = (pSourceImage[SourceBufferPos+
    SourceImageWidth]&0x00FF0000)>>16;
        DestGreen3 = (pSourceImage[SourceBufferPos+
    SourceImageWidth]&0x0000FF00)>>8;
        DestBlue3  = (pSourceImage[SourceBufferPos+
    SourceImageWidth]&0x000000FF)>>0;


     PixelCount=2;
     }
//     if (y<(SourceImageHeight-1)) //Grab the pixel colour
    below this one
//     {

        //DestRed   = DestRed   + (pSourceImage[
    SourceBufferPos+SourceImageWidth]&0x00FF0000)>>16;
        //DestGreen = DestGreen + (pSourceImage[
    SourceBufferPos+SourceImageWidth]&0x0000FF00)>>8;
        //DestBlue  = DestBlue  + (pSourceImage[
    SourceBufferPos+SourceImageWidth]&0x000000FF)>>0;

        //PixelCount++;
     //}

        if (PixelCount == 0) //If pixel on edge of screen in
    awkward position
        {
     PixelCount = 1;


        DestRed   = DestRed   + (pSourceImage[SourceBufferPos
    ]&0x00FF0000)>>16;
        DestGreen = DestGreen + (pSourceImage[SourceBufferPos
    ]&0x0000FF00)>>8;
        DestBlue  = DestBlue  + (pSourceImage[SourceBufferPos
    ]&0x000000FF)>>0;
```

```
        }

    //Calculate the average value of these pixels
//  DestRed  = DestRed / PixelCount;
// DestGreen = DestGreen / PixelCount;
//  DestBlue = DestBlue / PixelCount;

     SourceRed    = (pSourceImage[SourceBufferPos]&0x00FF0000)
   >>16;
        SourceGreen = (pSourceImage[SourceBufferPos]&0
   x0000FF00)>>8;
        SourceBlue  = (pSourceImage[SourceBufferPos]&0
   x000000FF)>>0;

     if (PixelCount == 2)
        {

        //DestRed = SourceRed*0.5 + DestRed * 0.5;
        //DestGreen = SourceGreen*0.5 + DestGreen * 0.5;
        //DestBlue = SourceBlue*0.5 + DestBlue * 0.5;
     DestRed   = SourceRed*SourceContribution + DestRed *
   DestContribution + DestRed2*DestContribution + DestRed3*
   DestContribution;
        DestGreen = SourceGreen*SourceContribution + DestGreen
   *DestContribution + DestGreen2*DestContribution +
   DestGreen3*DestContribution;
        DestBlue  = SourceBlue*SourceContribution + DestBlue*
   DestContribution + DestBlue2*DestContribution + DestBlue3*
   DestContribution;

        pDestImage[DestBufferPos+DestBufferStride+1] = (
   DestRed<<16)+(DestGreen<<8) + (DestBlue<<0);

        }



    DestBufferPos=DestBufferPos+2; //Point to next group of
     subpixels
    SourceBufferPos++; //Point to next pixel in source image

    } //End of for (int x=0; x>SourceImageWidth; x++)

DestBufferPos=DestBufferPos+DestBufferStride;

} //End of for (int y=0; y<SourceImageHeight; y++)



} //End of function
```

B.12 BackgroundSubtraction.cpp

```
#include <windows.h>

#include "BackgroundSubtraction.h"
```

```
BackgroundSubtraction::BackgroundSubtraction(int BufferXLength
    ,int BufferYLength,int Threshold)
{
IsBaseImageInitialised = FALSE;

this->FramesBetweenBackgroundUpdates = 4;

FramesProcessedSinceUpdate = 0;

this->BufferXLength=BufferXLength;
this->BufferYLength=BufferYLength;
this->Threshold=Threshold;

this->pBaseImage = NULL;
this->pMask = NULL;
this->pResult = NULL;
this->pSDerivative = NULL;

this->pBaseImage = new BYTE[BufferXLength*BufferYLength*4];
this->pResult = new BYTE[BufferXLength*BufferYLength*4];
this->pMask = new BYTE[BufferXLength*BufferYLength];
this->pSDerivative = new BYTE[BufferXLength*BufferYLength*4];

memset((void*)this->pSDerivative,0,BufferXLength*BufferYLength
    *4);
memset((void*)pMask,0,BufferXLength*BufferYLength);

}



void BackgroundSubtraction::InitBaseImage(BYTE* pBaseData)
{
int MaskPos = 0;
int BufferPos = 0;

int OldRed = 0;
int OldBlue = 0;
int OldGreen = 0;
int Temp = 0;

   if (this->FramesProcessedSinceUpdate>this->
    FramesBetweenBackgroundUpdates ||
            this->IsBaseImageInitialised == FALSE)
   {
   FramesProcessedSinceUpdate = 0;

     /* If the first frame copy the data directly to the base
    image buffer */
     if (this->IsBaseImageInitialised==FALSE)
     {
     memcpy(this->pBaseImage,pBaseData,this->BufferXLength*this
    ->BufferYLength*4);
     IsBaseImageInitialised = TRUE;
     }
     else
     {
     /* The background will be updated according to the
```

```
following formula*/

        for (int y=0; y<this->BufferYLength; y++)
        {
      for (int x=0; x<this->BufferXLength; x++)
              {

              OldBlue = pBaseImage[BufferPos];
              OldGreen = pBaseImage[BufferPos+1];
              OldRed = pBaseImage[BufferPos+2];

                      if (this->pMask[MaskPos] == 1)
                      {

          pBaseImage[BufferPos]    = (0.80*pBaseImage[
BufferPos])+(0.2*pBaseData[BufferPos]);
                      pBaseImage[BufferPos+1] = (0.80*
pBaseImage[BufferPos+1])+(0.2*pBaseData[BufferPos+1]);
                      pBaseImage[BufferPos+2] = (0.80*
pBaseImage[BufferPos+2])+(0.2*pBaseData[BufferPos+2]);

                      //BufferPos+=4;
                      //MaskPos++;

                      //continue; //If not required to
update this pixel then don't
                      }
                      else
                      {

                      pBaseImage[BufferPos]    = (0.7*
pBaseImage[BufferPos])+(0.3*pBaseData[BufferPos]);
                      pBaseImage[BufferPos+1] = (0.7*
pBaseImage[BufferPos+1])+(0.3*pBaseData[BufferPos+1]);
                      pBaseImage[BufferPos+2] = (0.7*
pBaseImage[BufferPos+2])+(0.3*pBaseData[BufferPos+2]);

          //pBaseImage[BufferPos] = pBaseData[BufferPos];
              //pBaseImage[BufferPos+1] = pBaseData[
BufferPos+1];
              //pBaseImage[BufferPos+2] = pBaseData[
BufferPos+2];
                      }

                Temp = OldBlue - pBaseImage[BufferPos];
        if ( Temp >= 0)
                {
                   if (Temp>5)
                      pBaseImage[BufferPos] = OldBlue - 3;

                }
                else
                {
                   if (Temp<-5)
                                 pBaseImage[BufferPos] =
OldBlue + 3;

                }
```

```
                Temp = OldGreen - pBaseImage[BufferPos+1];
            if ( Temp >= 0)
                {
                  if (Temp>5)
                      pBaseImage[BufferPos+1] = OldGreen -
   4;

                }
                else
                {
                  if (Temp<-5)
                              pBaseImage[BufferPos+1] =
   OldGreen + 4;

                }

                Temp = OldRed - pBaseImage[BufferPos+2];
            if ( Temp >= 0)
                {
                  if (Temp>5)
                      pBaseImage[BufferPos+2] = OldRed - 4;

                }
                else
                {
                  if (Temp<-5)
                              pBaseImage[BufferPos+2] =
   OldRed + 4;

                }




              BufferPos+=4;
              MaskPos++;
              }

            }

       }

   }
   if (this->FramesProcessedSinceUpdate<this->
    FramesBetweenBackgroundUpdates)
   {
   this->FramesProcessedSinceUpdate++;
   }

}


void BackgroundSubtraction::DoSubtraction(BYTE* pNewImageData)
{
int BufferPos = 0;
int MaskPos = 0;
```

```
int ValueBefore;
int ValueAfter;

  for (int y=0; y<this->BufferYLength; y++)
  {
    for (int x=0; x<this->BufferXLength; x++)
        {

                if (pResult[BufferPos]>Threshold || pResult[
   BufferPos+1]>Threshold || pResult[BufferPos+2]>Threshold)
                {
            ValueBefore = 1;

                }
                else ValueBefore = 0;



                if (pNewImageData[BufferPos]>pBaseImage[
   BufferPos])
                        this->pResult[BufferPos]=pNewImageData
   [BufferPos]-pBaseImage[BufferPos];
                else
                        this->pResult[BufferPos]=pBaseImage[
   BufferPos]-pNewImageData[BufferPos];

                if (pNewImageData[BufferPos+1]>pBaseImage[
   BufferPos+1])
                        this->pResult[BufferPos+1]=
   pNewImageData[BufferPos+1]-pBaseImage[BufferPos+1];
                else
                        this->pResult[BufferPos+1]=pBaseImage[
   BufferPos+1]-pNewImageData[BufferPos+1];

                if (pNewImageData[BufferPos+2]>pBaseImage[
   BufferPos+2])
                        this->pResult[BufferPos+2]=
   pNewImageData[BufferPos+2]-pBaseImage[BufferPos+2];
                else
                        this->pResult[BufferPos+2]=pBaseImage[
   BufferPos+2]-pNewImageData[BufferPos+2];


        if (pResult[BufferPos]>Threshold || pResult[BufferPos
   +1]>Threshold || pResult[BufferPos+2]>Threshold)
                {
        //    ValueAfter = 1;
        pResult[BufferPos] = 255;
                pResult[BufferPos+1] = 255;
                pResult[BufferPos+2] = 255;

                }
                else
                {
                //ValueAfter = 0;
                pResult[BufferPos] = 0;
                pResult[BufferPos+1] = 0;
                pResult[BufferPos+2] = 0;
                }
```

```
        /*if  (ValueBefore == 0 && ValueAfter == 1)
            {
              pSDerivative[BufferPos] = 0;
                  pSDerivative[BufferPos+1] = 0;
                  pSDerivative[BufferPos+2] = 255;
            }
            else if (ValueBefore == 1 && ValueAfter == 0)
            {
                 pSDerivative[BufferPos] = 255;
                 pSDerivative[BufferPos+1] = 0;
                 pSDerivative[BufferPos+2] = 0;

            }
            else if ((ValueBefore ==1 && ValueAfter == 1) || (
    ValueBefore == 0 && ValueAfter == 0))
            {
         pSDerivative[BufferPos] = 0;
                 pSDerivative[BufferPos+1] = 0;
                 pSDerivative[BufferPos+2] = 0;
            }*/


      pResult[BufferPos+3] = 0;

          BufferPos+=4; /*Point to next pixel */
          MaskPos++;
          }

   }

}

/* Draws any pixels with Difference above a threshold into the
     destination buffer */
void BackgroundSubtraction::DrawImage(BYTE* pDest)
{
int BufferPos=0;


   for (int y=0; y<BufferYLength; y++)
   {
      for (int x=0; x<BufferXLength; x++)
          {
       // if (pResult[BufferPos]>Threshold || pResult[BufferPos
    +1]>Threshold || pResult[BufferPos+2]>Threshold)
            //{

            pDest[BufferPos]=pResult[BufferPos];
            pDest[BufferPos+1]=pResult[BufferPos+1];
            pDest[BufferPos+2]=pResult[BufferPos+2]; /*Draw
    white pixels for significant difference */

            //}

      BufferPos+=4;
          }
```

```
  }
}

void BackgroundSubtraction::DrawDerivativePixels(BYTE* pDest)
{
int BufferPos=0;

  for (int y=0; y<BufferYLength; y++)
  {
    for (int x=0; x<BufferXLength; x++)
        {
      if (pSDerivative[BufferPos]==255)
           {

           pDest[BufferPos]=255;
           pDest[BufferPos+1]=0;
           pDest[BufferPos+2]=0; /*Draw white pixels for
   significant difference */

           }
           else if (pSDerivative[BufferPos+2] == 255)
           {
           pDest[BufferPos]=0;
           pDest[BufferPos+1]=0;
           pDest[BufferPos+2]=255;

           }
           else
           {
      pDest[BufferPos]=0;
           pDest[BufferPos+1]=0;
           pDest[BufferPos+2]=0;
           }

    BufferPos+=4;
        }
  }

}


void BackgroundSubtraction::DrawMaskRegions(BYTE* pBuffer)
{

int BufferPos = 0;
int MaskPos = 0;

  for (int y=0; y<BufferYLength; y++)
  {
    for (int x=0; x<BufferXLength; x++)
        {
      if (pMask[MaskPos] == 1)
           {
      pBuffer[BufferPos] = 255;
           pBuffer[BufferPos+1] = 0;
           pBuffer[BufferPos+2] = 0;
```

```
            }

      BufferPos+=4;
            MaskPos++;
            }

   }

}


BackgroundSubtraction::~BackgroundSubtraction()
{
   if (pBaseImage != NULL) delete pBaseImage;
   if (pResult != NULL) delete pResult;
   if (pMask != NULL) delete pMask;
   if (pSDerivative!=NULL) delete pSDerivative;
}
```