University of Southern Queensland
Faculty of Engineering & Surveying

Real Time Implementation of Obstacle Avoidance for an Autonomous Mobile Robot Using Monocular Computer Vision

A dissertation submitted by

Iain Brookshaw

in fulfilment of the requirements of

ENG4112, Research Project

towards the degree of

Bachelor of Mechatronic Engineering

Submitted: October, 2011

Abstract

For autonomous robotic motion, it is essential for the mobile machine to be able to judge its position relative to potential obstacles. This implies the ability to identify potential obstacles, study their approach and take appropriate action when necessary to avoid collision. In this age of cheap and available digital cameras and powerful computers, it is desirable to achieve this with a single digital camera as the sensor.

The digital camera has the advantages that it is cheap, easily installed, well understood, passive and physically small. When coupled with a small, powerful on-board computer it has the potential to create a effective obstacle avoidance system.

This investigation was attempt to create just such a system. A series of methods dealing with identifying, tracking and avoiding obstacles were be investigated, and the results of their implementation discussed. Alternative methods were be analysed and weighed. It was the initial intention to produce a program that could identify parts of the image as "obstacles", determine the approach of these obstacles and use this information to direct a small mobile, autonomous machine.

To achieve this a region based segmentation approach was used to identify the boundaries and extents of obstacles, while several Looming based methods were employed to judge object range and approach, specifically Looming through blur and Looming through area. Also considered were the ways and means of creating

frame to frame correlation of objects based on region geometry. Unfortunately, the final result was too inconsistent to enable true avoidance to be implemented.

The inconsistencies in the results were largely due to small errors in each section compounding as the program evolved. However, a wide range of tests on each of the component parts of the system illustrated that the concepts and methods selected are viable individually. While there were problems with consistency when the program is run, it was clear from the results that the individual components could be made to function if additional time was spent correcting errors.

The end conclusion was that, although the methods discussed were clearly viable, they require further experimentation and development before they can be fully implemented

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

Certification of Dissertation

I certify that the ideas, designs and experimental work, results, analyses and conclusions set out in this dissertation are entirely my own effort, 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.

Iain Brookshaw

0050086292

Signature

Date

Acknowledgments

This thesis was typeset using the LaTeX 2_ε typesetting program.

The author would like to thank Dr. Tobias Low for his clear guidance, Dr. Leigh Brookshaw, for his generous assistance, Mr. Erin Heaton for his longstanding support and finally, all those poor souls who showed great patience whilst the author excitedly showed off the latest results.

IAIN BROOKSHAW

University of Southern Queensland
October 2011

Contents

Abstract i

Acknowledgments v

List of Figures xiii

Chapter 1 Introduction 1

1.1 Computer Vision and Mobile Robots 3

1.2 Objectives 3

1.3 Background, Vision and Image Processing 4

1.3.1 Image Acquisition and Processing 4

1.3.2 Software and Libraries 5

1.3.3 Hardware Preconceptions 6

1.4 Chapter Summaries 7

Chapter 2 Image Segmentation and Obstacle Identification 9

2.1 Why Segment an Image? 11
2.2 Segmentation Methods 13
2.2.1 Thresholding 13
2.2.2 Edge Detection 14
2.2.3 Region Growing 18
2.2.4 Split and Merge Techniques 20
2.3 Single Pass Split Merge Segmentation 21
2.3.1 Merging Algorithm 22
2.3.2 Splitting Algorithm 24
2.4 Image Pre-Processing 31
2.4.1 Noise Removal and Image Smoothing 31
2.4.2 Finding a Tolerance 32
2.5 Potential Problems 34

Chapter 3 Distance and Approach 35

3.1 Methods of Distance Estimation 38
3.2 Looming 41
3.2.1 Area 42
3.2.2 Irradiance 44

3.2.3 Texture 45

3.2.4 Blur 46

3.3 Implementation 52

3.3.1 Blur Calculation 52

3.3.2 Looming Calculation 55

3.4 Avoidance 56

3.4.1 Approach Categorisation 56

3.4.2 Direction Decisions 57

Chapter 4 Tracking and Correlation 59

4.1 Frame to Frame Correlation 61

4.2 Point or Feature Tracking 62

4.3 Region Tracking 64

4.4 Chosen Algorithm 65

4.4.1 Centroid Calculation 66

4.4.2 Data Recording 69

4.4.3 Centroid Matching 70

Chapter 5 Results and Discussion 73

5.1 Ideal Test Images 75

5.2 Segmentation Verification 76

5.2.1 Effects of Pre-Processing Filtering 78

5.3 Centroid Verification 80

5.4 Tracking Testing 81

5.5 Blur Estimation 84

5.6 Looming Computation 93

5.6.1 Area Looming 93

5.6.2 Blur Looming 95

5.7 Discussion 96

5.7.1 Segmentation 96

5.7.2 Tracking 97

5.7.3 Looming 99

Chapter 6 Conclusions **103**

6.1 Segmentation 105

6.2 Tracking 105

6.3 Looming 106

6.4 Overview 106

6.5 Completion of Objectives 106

Chapter 7 Future Work **109**

7.1 Obstacle Detection 111

7.2 Tracking and Correspondence 112

7.3 Looming, Approach and Avoidance 113

7.4 Navigation 115

References **117**

Appendix A Original Specifications **121**

A.1 Research Specification 123

Appendix B Program Listings **125**

B.1 Final Programs 127

B.1.1 Main Driver Function 127

B.1.2 Segmentation Functions 131

B.1.3 Centroid Finding Functions 146

B.1.4 Tracking Function 153

B.1.5 Looming Function 157

B.2 Testing Algorithms 158

B.2.1 Segmentation Test Program 158

B.2.2 Tracking Test Program 160

B.2.3 Looming Test Program 165

List of Figures

2.1 Illustration of the pixels in an image. Counter clockwise from top: the original image, a close up illustrating the graduations of pixels and a extreme close up, illustrating the graduation of pixels in an object. 12

2.2 Illustration the single pass split and merge operation, showing the four pixel group and the previously labeled pixels. 20

2.3 The merging algorithm as implemented 23

2.4 The orientation of the pixels in the four pixel block. This orientation is also used in the final program 24

2.5 The original splitting algorithm, which was not implemented successfully 25

2.6 The various combinations possible with a four by four pixel block. Note that *viii* and *ix* are the same. 28

2.7 The final splitting algorithm, where splitting was based on patterns in order of precedence 31

3.1 illustration of the difference between range and depth. 37

3.2 Stereo vision, using one camera and the frame difference to find depth. 38

3.3 Graphical depiction of Looming using projected area. 41

3.4 The Gaussian curve, showing standard deviation (σ) and the true radius of blur 47

3.5 Plot of various step functions representing the region edge with varying degrees of ideal blur. $f(x)$ is the original step, with $b(x)$ being the camera's blurred edge and $b_a(x)$ and $b_b(x)$ being the re-blurred edges. 48

3.6 Plot of R_{max} values recovered in Note the symmetrical nature of the plot around the region boundary. 49

3.7 Illustration of finding the four points of a region for blur computation. The left hand image shows the case of the centroid being in the region. The right hand image is for the centroid being out of the region. 54

4.1 Illustration of the difficulties of tracking for non-orthogonal movement 63

4.2 the recursion algorithm as used by the tracking program 68

5.1 Original checkerboard test image. 600 by 600 pixels 75

5.2 Illustration of segmentation of a complex image. On the left, the original image, on the right the segmented output. n.b. this image was used as a test image in (Hu & de Haan 2006) (among others) and was repeated as a test image here as the author found its provenance rather amusing. In actuality any reasonably uncomplicated image would do. 78

5.3 Illustration of the differences between two segmented images. The top set is 2% smaller than the bottom set. Notice the region discrepancies, especially in the foreground. 79

5.4 The test image, showing the successful finding of the centroids. . . 81

5.5 The test more complex test image image, again showing the successful finding of the centroids. 81

5.6 Checkerboard test image distorted for tracking. 600 by 600 pixels 82

5.7 Distorted checkerboard showing tracked centroids. The grey line from the centroids in the right hand figure shows the computed position of the centroids in the previous (left hand) frame. 82

5.8 The results of tracking the segmented images in figure 5.3. The line from each centroid indicates the calculated position of that region in the previous frame. 85

5.9 The ideal blur test image. It is simply a two square checkerboard 400 by 200 pixels. Note the grey values used to colour the squares are not black and white. 85

5.10 The recovered blur radius for the ideal test image with initial radius of 1 and a re-blur σ values of 4 and 7. 89

5.11 The step functions for the blurred edges showing the initial recovered blur and the two re-blurred edges. Re-blur σ values of 4 and 7. 90

5.12 Maximum recovered blur difference for the ideal test image with initial radius of 1 and a re-blur σ values of 4 and 7. 91

5.13 Multi-coloured test image for general blur recovery tests. 92

5.14 The area Looming test images. Not the black area in the left image is 64% smaller than the right. 94

Chapter 1

Introduction

1.1 Computer Vision and Mobile Robots

Giving functioning robots the ability to "see" and understand their environment through complex anthropomorphic optical sensors is a long sought after goal in robotics. Such sensors have the distinct advantage of being passive (in that they do not give off any signal, they simply receive data) and are able to obtain much information about their environment from relatively simple sources. Currently, with the development of sophisticated, inexpensive web cams such systems are practicable for minimal cost.

In order to implement a system that can use such vision to avoid obstacles, much information must be recovered from the digital image. The information that a human obtains without conscious effort (relative size of objects, their position and their very extent) must be laboriously recovered from the image data. What follows is chiefly concerned with recovering information that could then be used to guide a hypothetical autonomous machine.

The critical information for the avoidance of obstacles is: the extent of objects in the image, their correspondence to objects in previous images and their position relative to the viewer. This is fairly obvious and the reader can do all of these things without effort, however it can be a laborious process to recover these things from the image flow. Thus the minimal form of any machine attempting to navigate via vision is a camera, a computer (of significant processing speed) and a mechanism for controlling the drive.

1.2 Objectives

The initial project objectives were to design a means for an autonomous mobile robot to avoid unknown obstacles. It was intended that this be accomplished in real time with the focus on an small scale autonomous platform.

The final programs were intended to separate an object from its surroundings, make an estimate of its approach and use this data to formulate a response for the machine. This was meant to be implemented with only one camera.

Time permitting, these were intended to be extended to include true navigation, as opposed to simple avoidance.

In the event, these were found to be overly simplistic objectives. Other unforeseen problems extended the project in some directions and curtailed it in others. For a detailed list of initial specifications see Appendix A.1.

1.3 Background, Vision and Image Processing

1.3.1 Image Acquisition and Processing

Machine vision has a great many meanings and applications. All, to some degree, revolve around interpreting the response of photo-sensitive electronics to changing light conditions. In this case a simple web-cam type digital camera was used to obtain a detailed picture of the immediate surroundings. It was intended at all stages of the project that the final results would be obtained from inexpensive cameras of this sort.

Once the image has been obtained from the camera, a variety of filters and algorithms were applied to obtain information, such as object location, position and approach. All of these filters involved running over the image (frequently pixel by pixel) and recording data.

A digital image, the output of the camera, is simply an array of numbers containing the values for brightness at that point, as interpreted by the camera. [1]

[1] The basics of a digital image can be obtained from any text on the subject. The information recounted here is an amalgamation of the background given in (Sonka, Hlavac & Boyle 1994),

Add enough elements to this matrix and an image of recognisable complexity is created. This image could be colour or gray-scale. In the former, the matrix is three dimensional with several values or channels (usually Red, Green, Blue and alpha or transparency) for each pixel or point in the image. It is the combination of these values that comprises the "colours" one sees. By contrast the latter arrangement has only one number per pixel, a integer of value 0 to 255 (in a colour image data is also stored in integers in this range). This means that there is only shadings of light to dark for each pixel (usually 255 is fully white and 0 is fully black). The reason for this range is that it enables one byte, 8 bits to be used for every pixel ($2^8 = 256$, hence 256 is the maximum integer that can be stored in an 8 byte space). For simplicity, single channel gray-scale images were used throughout this project. It was not felt that colour would give any noticeable advantage.

The key point from all this description, is that the image fundamentally remains an array of integer points from 0 to 255, and as such can be treated with numerical approaches like any other array of data.

1.3.2 Software and Libraries

To implement any algorithm in the computer vision field it is necessary to first select the programming environment. In this case the selection was governed by two key constraints, speed and ease of use. As the focus was real time applications, the program execution must be rapid in the extreme. This immediately ruled out scripting languages such as MATLAB (although MATLAB was used in very early stages as a familiar environment in which to test ideas, the code developed was very limited and is not included). Furthermore, the programing language needed to be simple to learn and debug, with a wide range of support and documentation (when the project began the author had familiarity with MATLAB only) as it was

(Jain, Kasturi & Schunck 1995), (Davies 1997) and (Bradski & Kaehler 2008) as well as other sources.

anticipated that no time could be spared for programing errors brought on by ignorance.

Considering these factors and soliciting advice from several parties, finally led to the selection of *C* as the programing language and the Open Computer Vision library as the basis for further programing. The existence of a open source computer vision library (recommended by Dr. Tobias Low) was critical in this decision.

All programs developed throughout this project use this library as a basis and rely on it for frame acquisition, image format and data storage and low level manipulation. The complete library can be obtained from `http://sourceforge.net/projects/opencvlibrary`.

All code was compiled on a Debian Linux machine (version 6.0.3) using the gcc compiler (version 4.4.5) linked with the OpenCV library (version 2.1).

1.3.3 Hardware Preconceptions

Mobile robots and autonomous machines are the objectives of many fields of study. While the applications are obvious, there are a great number of them and the methods used to develop solutions to one may not necessarily apply to the others. Throughout this project the focus has been on developing a vision system for a small scale autonomous device. The resultant designs may not function out of this context.

Although the title "Autonomous Mobile Robot..." could imply a machine of almost any size and application, it was always envisaged as a small platform of desktop size. Thus the robot's speed and operating environment were similarly modest in scope. it was imagined such a device would spend much of its time moving at a slow walking pace along smooth corridors and around rooms. In

addition it was assumed that the objects in view would all be rigid and static. Thus the initial vision for the end user was a small machine operating indoors in an environment where the only movement in the image was caused by the camera's motion.

The reasons for these limitations were simple. Although the focus called for "real time" initial research quickly showed that any method would involve a good deal of image manipulation. This was expected to slow computation considerably. Thus the speed of the machine (and hence the discrepancy between frame) should be reduced. As this implied a slow moving device, the other assumptions soon followed.

As it transpired the programs were never implemented on actual hardware (time being prohibitive and results limited). Nevertheless, the above considerations formed the background to all stages of the project

1.4 Chapter Summaries

- Chapter 2 covers the ways and means of separating objects from their surroundings. A number of processes are reviewed and the final method elucidated.
- Chapter 3 discusses ways of quantifying the approach of these objects. While a number of methods were reviewed, the Looming approach was finally decided upon.
- Chapter 4 describes how frame to frame correlation of objects was accomplished.
- Chapter 5 covers the results of implementation.
- Chapter 6 discussed the conclusions drawn from chapter 5.

- Chapter 7 discusses the future possibilities of the project and possibilities for fixing the remaining errors.

Chapter 2

Image Segmentation and Obstacle Identification

2.1 Why Segment an Image?

When instructing a computer to avoid obstacles, the first difficulty is defining what is meant by an “obstacle”. When a person examines a scene, they are capable of separating one object from another. Intuitively a person knows that this object is here, it is occluded by this object, which is obscured by this object and so on. Many people are probably wholly unaware of what they are doing when they make these distinctions. Electronic eyes have no innate ability to create these distinctions. The computer does not “see” anything, per say, instead the light is detected and transformed into a matrix of numbers, representing the colour and intensity of the light at that discrete point or pixel.

This matrix is the computer’s representation of an image. The light and dark patterns, which the human eye and mind would instantly recognise as a representation of three dimensional space, mean nothing to the computer (Jain et al. 1995)). This is a critically important point and one that must be held in constantly mind. While the image may represent familiar objects to the viewer, to the computer they are only numbers in an array. Furthermore, these numbers represent the *projection* of the three-dimensional space on a two-dimensional plane. Thus it follows that the computer is incapable of recognising one object from another, or even distinguishing between foreground and background without aid.

This would appear to be fairly obvious, yet it is easy to confuse the image as the eye sees it and the image as the computer interprets it. Segmentation is the first step in enabling the computer to differentiate between objects. More succinctly, segmentation is “to divide an image into parts that have a strong correlation to objects or areas of the real world” (Sonka et al. 1994) pg 112). A close examination of a digital image will reveal that a single object is not represented by pixels of a single colour (see figure 2.1). Instead, an object is represented by pixels of graduated intensity[1]. This represents the shading caused

[1]Many sources refer to intensity, grey-scale, illumination, etc as if they were distinguishable

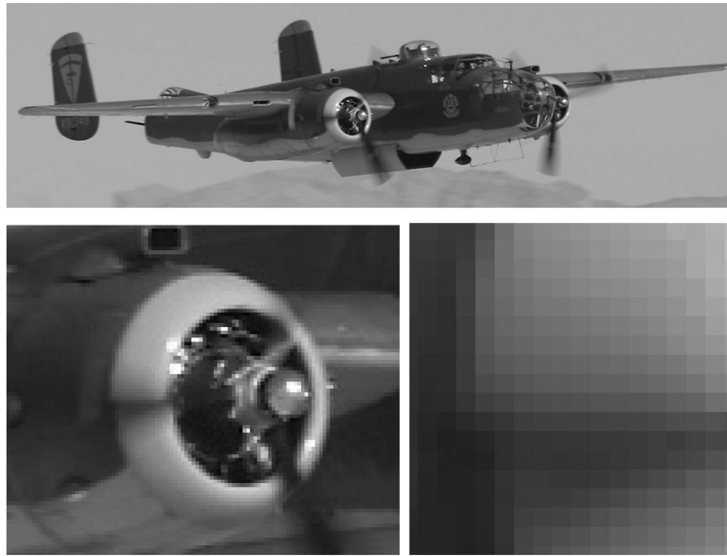

Figure 2.1: Illustration of the pixels in an image. Counter clockwise from top: the original image, a close up illustrating the graduations of pixels and a extreme close up, illustrating the graduation of pixels in an object.

by the reflection of light from the surface. Notice that while there may be a clear edge (see section 2.2.2) the pixel values within this edge (the object boundary) may all be different.

Segmentation, then is the means by which all pixels within an object are distinguished from the pixels in another object (Sonka et al. 1994). This can be done several ways, but its purpose is to enable subsequent routines to distinguish between one obstacle and another, assign values of range and depth (see section 3.1) and conduct avoidance and navigation calculations. This can only be done swiftly if each object in view is clearly distinguishable from all others.

quantities. In fact what they usually mean is the numerical grey value of the given pixel (Jain et al. 1995), for an image of 8-bit depth this is 0 to 255. The terms intensity, grey-scale, etc are in fact usually used interchangeably.

2.2 Segmentation Methods

There are a number of ways of splitting an image into it's component parts. All to some extent rely on the change in some numeric property at the object boundary, or similarity of connected pixels that represent an object (Jain et al. 1995). The various approaches most commonly described in the relevant literature include: Thresholding, Edge Detection and Split-Merge techniques . In this particular project, a species of the split merge technique was selected.

2.2.1 Thresholding

Although too simple for the purposes of this project, the Thresholding method provides a good illustration of many of the techniques used in more applicable methods.

Essentially, the Thresholding method is based separating an object from its background (Sonka et al. 1994). A threshold is selected based on analysis of the image properties (Jain et al. 1995) and each pixel compared to this value. The pixels in the output image are then assigned as 255 or 0 depending on which side of the threshold the input pixel falls.

Relying on a clear distinction between objects and background, this method is usually divides the image into two segments, background and foreground. A more complex version of this is band Thresholding, described in (Sonka et al. 1994). In this approach, bands of grey-scales are selected and the pixels sorted into these bands. In essence it is the same technique, but using several thresholds instead of one. This is a useful method for objects of known grey-levels (Sonka et al. 1994).

This is of course, far too simplistic approach for this application, relying as it does on clear demarcation between only two objects. As the objectives of the project involve unknown images of undetermined complexity, there would probably be

obscure edges and indeterminate outlines. Additionally, the assumption that there is two or a small limited number of arbitrary regions is clearly too crude for obstacle avoidance in an unknown environment. However, the Thresholding method illustrates the basic idea that like pixels are arranged into groups and that this likeness is judged by the numerical value of each pixel. Furthermore, the concept of a threshold is an important one. Most segmentation methods employ some means of establishing demarcation, some numerical value acquired from analysis of the entire image data that establishes a boundary.

2.2.2 Edge Detection

A much more sophisticated method than Thresholding, edge detection methods rely on the change in pixel values between objects. Requiring much less prior knowledge of the image (Sonka et al. 1994), edge detection is a potentially more powerful tool. However, in the context of obstacle avoidance it created a number of difficulties in application. Namely, the difficulty in creating closed outlines and problems related to objects that may not have edges (such as walls in close proximity). While the former may be overcome (see section 2.2.2), the latter poses a more complex problem. Although a useful tool, it was felt to possess more drawbacks and fewer advantages compared to the the chosen system (see section 2.3).

Edge detection functions on the assumption that between one object and the next there is a notable demarcation. If this distinction is sharp enough, an object edge has been located (Jain et al. 1995). These distinctions are important local changes and are widely used in image analysis (Jain et al. 1995), in some form all segmentation ultimately employs the idea that different objects are represented by dissimilar pixels. In consequence, a number of methods exist for distinguishing edges, judging the probability of that edge's actual existence (Sonka et al. 1994) and combining them into contours. However, as it was felt that the problems with

edge detection were inherent in the result, rather than the process, these were not investigated in great detail. Suffice it to say that most edge detection relies on the assumption that there is some clear demarcation between the group of pixels representing one object and the pixels representing another (Sonka et al. 1994). It is frequently assumed that the this intensity changes takes the form of a step, ramp, impulse or similar function (in one dimension) (Jain et al. 1995). The result of this process is a series of unrelated "dots", pixels that fit a criterion of an edge.

Once the edges have been established it is becomes necessary to produce contours, a list or mathematical curve that describes the edges (Jain et al. 1995). The validity of the edges must be assessed, their strength or weakness evaluated to remove weak edges (those with a low probability of existence, based on their surroundings (Sonka et al. 1994). Then the gaps in edges must be filled. Finally, one has to break the image into regions based on the results of this process (or use the contours themselves, see below). Each step in this process would require a unique pass of the image and all of these passes are simply those needed to create successful segmentation. It does not include the prepossessing steps. Such an involved process was felt to be too computationally intensive to employ.

It could be argued that once these contours have been established, one need only use the resultant edges to define obstacles, without further segmentation. This is partially true, however it causes problems not only with the method used to track and gauge distance [2] but causes problems with obstacles that are too large to see.

Consider a wall, it is clearly an obstacle, yet unless its end is in view or it is heavily textured, it produces no edge, only a line on the floor. This line would not noticeably expand or contract as the camera approached, it would simply move

[2]see section 3.2 it could become difficult to establish the motion of a contour owing the possibility of its being one dimensional. This problem could be overcome with the centroids approach described in chapter 4, but the other problems were felt too large to render it useful.

"up" and "down" the image frame. During the early stages of the project this was felt to be unsatisfactory from a Looming perspective (see section 3.2), owing to the fact that the line area was not expanding. Later when blur was selected this problem became less of an issue and the need to have two dimensional objects became less important. However, by that stage other segmentation methods were chosen.

Edge detection was investigated in the initial phases of the project and discarded when many of the methods outlined in sources returned only a disconnected series of dots. These "edges" could have to then be amalgamated into contours, that could then be used to create regions. This seemed a very time consuming and pain full process. Discouragement with uninspiring initial results involving the Canny Edge detection algorithm (Bradski & Kaehler 2008) reinforced this view. It was later discovered that there were far more effective means of employing edge detection methods (see section below).

Edge Tracing

Following the author's decision to use region based segmentation (see below) as the basis of obstacle detection, an alternative method, Edge Tracing was suggested. Although a potentially highly applicable and useful tool, this method was not implemented due to time constraints, the completion of segmentation sections and other, contrary advice.

While the simple edge detection methods researched in earlier phases of the project produced unsatisfactory results, edge tracing had the potential to produce accurate and computationally simple answers. The dissatisfaction with previous edge detection devices resulted from the disjointed output. As mentioned above, they mostly searched the image pixel by pixel, flagging all pixels that matched their definition of "edge" and blacking out the others. This produced a collection of disjointed dots that then had to be transformed into continuous contours

(Billingsley 26th May 2011).

By contrast, an tracing algorithm, searches until an "edge" is found, then follows the edge by searching the surrounding pixels for the next stage. The next pixel in the line is then found and the program moves forward until no surrounding pixels meet the edge criteria. Thus from this process a continuous line is found, with a definite start and conclusion for each edge. This method can be made swifter by employing various search patterns on the basis of spirals to remove the necessity of searching every pixel in the image (Billingsley 26th May 2011).

While far more efficacious than any other edge detection method, the edge tracing algorithm was not selected for final use in the finished program for a number of reasons. When identified to the author, writing for the final segmentation algorithm (see below) was already nearing completion. It was initially felt that additional expenditure of time could not be justified, especially as virtually all existing work would have to be rewritten and for a method that was bound to have unforeseen and time consuming side issues. Secondly, it was believed that the edge based approach suffered from a serious flaw. By definition only the outlines of objects are considered. While this requires far less computational power than other methods (only a tiny fraction of the image is under consideration (Billingsley 26th May 2011), it renders most of the image a blank map with no information about those pixels. If the machine was to approach a blank wall, an edge detection device could only extract information about the line where the floor meets the wall.

This lack of information was considered a serious problem If an edges are used, the only points in the image that are known are those edges. There is no information about the remainder of the image. From a avoidance perspective this is workable, but not satisfactory and leaves little room for expansion. While the machine could be instructed to avoid the line on the floor that represents the wall, it would be better if it were known more about it, such as how far the wall extends. A edge based approach may show the top of the wall, but the machine has no way of

knowing that the line representing the top is connected to the line representing the bottom. Thus a shallow "step" could be misinterpreted as an insurmountable obstacle. This and other similar objections lead to the belief that it would be far more useful if all the pixels representing the object could be labeled and the object's full extent known.

Having full knowledge of an object's extents could enable the final program to be expanded into other areas (eg: object recognition, identification etc.) and while such expansion was beyond the scope of this project, after consultation (Low 15/12/2010 to October 27/10/2011) it was agreed that Edges did not offer the best chances for future expansion and that a region based approach might be more efficacious.

2.2.3 Region Growing

Where edge detection can be used to outline a region for later segmentation, region growing functions in the opposite direction. The regions are defined directly, producing blocks of continuous colour that define the extent of an object. The resultant image can then be combined with edge detection methods if the edges are desired (Davies 1997). The methods for region growing are based around determining the similarity (or dissimilarity) of regions, on the assumption that all pixels that represent an object will exhibit similar characteristics.

The similarity of regions is assessed via the "homogeneity criterion". Simply, the definition of the permissible difference in some numerical property between similar regions if those regions are to be considered part of the same object. This is often based on the difference in grey-scale levels from one region to the next.

Merging Techniques

The merging technique is relatively self explanatory. Essentially, like regions are joined or merged as judged by some homogeneity criterion. This criterion is usually based on the grey-scale properties of the regions in question.

The simplest, but most expensive way of implementing this method (as described in (Sonka et al. 1994) is to consider every pixel as a separate region and try to merge a pixel with its neighbours. If the neighbours are “similar” then those pixels are marked by the same colour in the output image and the program moves on. Therefore for a image of n pixels by m pixels one begins with $n \times m$ separate regions and continues to merge adjacent regions into large segments until it is no longer possible to merge a new pixel without violating the predefined homogeneity criterion. Thus one grows regions much as one would grow a crystal in a jar and at about the same speed.

Needless to say, this is far to slow and simple a method to employ for a real time application. However, it does serve to illustrate the merging method.

Splitting Techniques

Virtually the exact algorithmic inverse of the pure merging method, region splitting takes the entire image as one region and successively breaks it into smaller regions. The original full image region is broken down repeatedly until it is impossible to segment further without violating the homogeneity criterion. When this point is reached it means that all the remaining regions satisfy the criterion and can be considered as a single object.

Interestingly, while the methodologies are clear inversions, the results are not identical (Sonka et al. 1994). Despite the complementary nature of the algorithm the same image segmented by these two methods will yield different results.

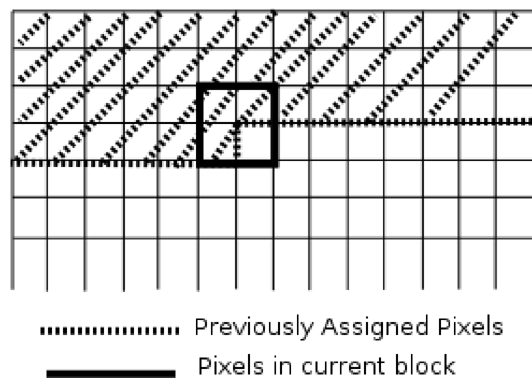

Figure 2.2: Illustration the single pass split and merge operation, showing the four pixel group and the previously labeled pixels.

Once again, it is apparent that this method, by itself is unsatisfactory for the same reasons as above.

2.2.4 Split and Merge Techniques

The most promising method of image segmentation arose from the split and merge branch of techniques. As mentioned above they are not direct inversions, however when used in conjunction they may produce an efficient and effective algorithm (Sonka et al. 1994).

It is possible to combine the split and merge algorithms to produce a more satisfactory result. The simplest way to do this is to subdivide the image into a grid of four large regions (Yang & Lee 1997), then recombine two or more of the four if they meet some homogeneity criterion (usually the mean grey-scale value). The resulting regions are then split into four again (if this is possible without violating the criterion) and the process of combination repeated with the new sub-regions. This process is repeated until it is no longer possible to split the sub-regions without violating the criterion used (Sonka et al. 1994).

This criterion is usually grey-scale. The average grey value or brightness of the

prospective region is measured and compared to a set tolerance (Davies 1997). Alternative approached include approximating the region to a planar surface (Yang & Lee 1997). If the mean squared error is below a certain tolerance the region is homogeneous.

Needless to say, this approach is almost as slow and computationally intensive as the previously mentioned split or merge methods. It is significant, however in illustrating the combination of the split and merge concepts. The idea of initial splitting and later re-merging can be adapted to give a much more rapid and successful method.

2.3 Single Pass Split Merge Segmentation

The basic split and merge technique, while easy to explain, somewhat computationally expensive and slow (Sonka et al. 1994). This can be remedied by a conceptual modification referred to as Single Pass Split and Merge. This algorithm is capable of segmenting the image in one pass, rather than the complex tree structure described above. Due to the real-time focus of the project, this was eventually the method selected.

The method works in two distinct stages. First a small section of the image is subjected to a splitting algorithm and labels assigned to the pixels accordingly. These split pixels are then compared to the regions in the main image to which they were previously assigned and the split groups merged to their larger counterparts as appropriate. The similarity or dissimilarity of pixels and regions is determined by a tolerance established by the image properties (Sonka et al. 1994).

This method enables one to segment the image in one pass. The small image section is a square of n pixels that is stepped through the image sequentially until it reaches the end. As it passes the pixels in the square are split, as though that square is an entire image, then the split groups are merged back into the main

image where possible and new regions begun where not. This way the entire segmentation process is performed as the square advances. Other merging or splitting methods imply multiple levels of processing and multiple passes through the image (Dep 1998). See figure 2.2 for the practical application of this process.

2.3.1 Merging Algorithm

Although merging is actually the second step in the process, it is the simplest and will of necessity be described first. From the above description of the single pass split merge algorithm, it is clear that the n pixel splitting block is far to small to be of any use in representing entire regions. To fully segment the image it was necessary to merge the split block with larger regions in the image.

To accomplish this, one uses the labels that the splitting algorithm assigned to the n pixels. This algorithm divided these n pixels into a maximum of n groups and a minimum of 1. Each group was compared to the region it was part of *before* the splitting algorithm was run. Notice in figure 2.2 that by because of the sequential nature of the process at least two of the pixels in the four pixel block have been assigned before. For the four pixel block pictured, up to two pixels are previously unassigned to any region (for most of the image three are previously assigned, but for edges the reduces to two or one). Thus one can compare the groups designated in the n pixel block to the larger regions and merge them as appropriate.

The merging is performed by finding the difference between the mean grey level of the large region and the mean grey level of the group in the block. If this difference is within a tolerance, the regions are merged. If an entire split group of pixels have never been assigned before, they become a new region in the main image. It is assumed that the new region would have been incorporated in another of the small groups if its boundary were not marked.

Merging Algorithm

1. Obtain a block of four pixels split into groups by the splitting algorithm (see section 2.3.2).

2. Compare each pixel in group n to its previous region and find the best match. Note: some pixels may not have been previously assigned (this is accounted for in the program).

3. Assign all pixels in group n to the region which produces the best match *if* that match is within tolerance. If not, then make that group a new region in the output image.

4. If a group is comprised wholly of pixels that have no previous assignment, make them a new region in the output image.

5. Repeat until all groups have been merged.

Figure 2.3: The merging algorithm as implemented

What this means is that most pixels are assessed several times while the algorithm still makes only one pass through the image. This ensures that the sub-groups in the n pixel block are merged with the most appropriate region. To ensure the best match, the sub-group mean is compared with the mean of the larger regions. The pixels in the sub-group that are already part of larger regions (and are labeled as such in the output image) from this the large region mean is known. If the pixels in the subgroup comprise components of several regions, the region mean which is closest to the sub-group mean is the best match for the sub-group. This means that pixels may be reassigned to several different regions as the program continues and better matches are found.

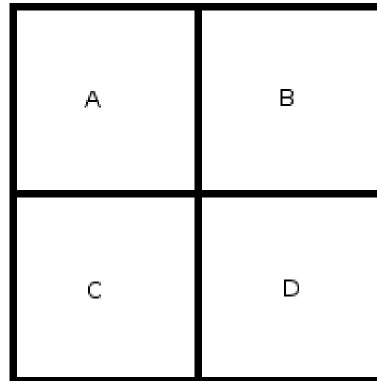

Figure 2.4: The orientation of the pixels in the four pixel block. This orientation is also used in the final program

2.3.2 Splitting Algorithm

In order to successfully employ merging, it was necessary to select a small section of the image to split, as discussed above. Having done this, the split regions could be merged back into the larger image, thus creating regions.

The section to be split is a square of $n = 4$ pixels, the top left hand pixel being the current location in the main image [3] (see figure 2.2). This four pixel block is now split according to the dissimilarity of the pixels. Now this seems fairly straightforward, simply see how dissimilar the pixels are and split accordingly. Unfortunately, a more rigorous definition is needed than "dissimilar", dissimilar to what or whom? It was exceedingly difficult to find a good criterion to distinguish between varying pixels, especially as the various sources studied were annoyingly vague on the subject. Several approaches were tried, but most sources remained exasperatingly silent on just *how* one is supposed to assess dissimilarity.

Initially, this was not judged to be of much importance. The main reference for this topic, (Sonka et al. 1994), did not even discuss how the splitting was

[3] There is nothing especially magical about four pixels, virtually any conglomeration of manageable size would do, (Sonka et al. 1994) list several examples. In this case four was chosen as the most workable number

Original Splitting Algorithm

1. Obtain the four pixel block
2. Check to see if range of grey-scale values are within tolerance. If they are, make all four pixels part of the same group and stop.
3. Find the grey-scale mean.
4. Group pixels below the mean together and group those above the mean together.
5. Continue to split the groups up based on the groups' mean until the groups' grey-scale range is less than tolerance.

Figure 2.5: The original splitting algorithm, which was not implemented successfully

to be accomplished. Clearly the working area was too small for the large scale splitting methods described in previous sections. Because of this gap, there grew the erroneous belief that any slack in the splitting would be picked up by the subsequent merging. Out of this grew the first splitting algorithm, discussed below (readers disinterested in a method that did not work and why it failed should skip to section 2.3.2).

Original Algorithm

In the original algorithm the dissimilarity was judged according to the distance of the various pixels from the mean (at this point it is worth reiterating that the numerical value attached to all pixels is the grey-scale value). When the block was split, only the four pixels are in consideration. The block is treated as though it was an image in isolation.

Once obtained, the grey-scale values of the four pixel block were examined and a mean grey-scale value and grey-scale range found. If the range is less than a tolerance, all four pixels are labeled as one group and the splitting algorithm is ended. If not, then the four pixels are divided based on the mean. All those falling below the mean comprise one group, while all those above the mean comprise the other group. This process was continued in the subgroups until the grey-scale range in the subgroups was less than the tolerance.

This appears a complex and difficult process. Why not just compare each pixel to each other pixel? There are after all, only six comparisons to make. Unfortunately this option, in addition to being difficult to code successfully, creates an additional problem. Suppose that one has determined that pixel A and B are similar, A and C may also be similar, yet C and B may be dissimilar. Which pixel to assign to a given group may end up being a matter of where the algorithm started first, thus creating different segmentation results for each order. This is clearly unsatisfactory.

Having thus split the four pixels into their component groups (a minimum of one and a maximum of four separate groups), one now had labels for each pixel in the group indicating it's relationship with the other pixels in the four pixel set. These labels were then utilised in the merge section of the algorithm.

This was a complicated and fairly arbitrary process. However it did function after a fashion, the algorithm ran without computational error and, when combined with the merging algorithm (see section 2.3.1), produced a segmented image that appeared to correspond to the objects in view. It was only after the introduction of centroid calculations and tracking, much later (see section 4.4) that it became obvious that something was very wrong with the original method.

Without getting ahead of sequence (readers interested in why centroids or tracking are of interest should turn to section 4.4), later steps in the project indicated that the image regions were wildly unstable and tended to split and join without

much warning. The reason why this was not picked up much earlier was that the edges of the regions were fairly stable, although they tended randomly fragment inside. Thus, while being difficult to spot through observation, a unacceptable randomness was being introduced into the segmentation process.

Long hours of painful and tedious investigation later, it became apparent that the problem was with the splitting algorithm. The original splitting algorithm, as described in figure 2.5, functioned on means. Splitting the four pixel block was accomplished by comparing the individual pixels to the mean of their prospective groups. Unfortunately, this method contained a very large logical flaw. If two pixels A and B were close together, close enough that they were obviously part of the same group, it was thought that they would be assured assignment to the same group. However if they lay on opposite sides of the mean, they would be irrevocably split into the upper and lower groups. This meant that an artificial barrier could conceivably be created in the middle of the four pixel block. This barrier may, or may not be re-merged in the merging stages. It was this unfortunate possibility that was introducing a random divide into the finished regions.

Ironically, this problem was considered (although its true importance was not full recognised) early in the project. However it was overshadowed by the need to determine what to do if A and B are a group and A and C are a group, but C and B are not (as discussed above). Essentially, it was thought that the initial method was the best compromise in dealing with a difficult situation. For some considerable time, no better answer could be formulated.

Final Algorithm

Once it had been recognised that the initial algorithm was unsatisfactory, it became necessary to produce an new method for splitting the four pixels. In the end, it became necessary to consider all possible combinations.

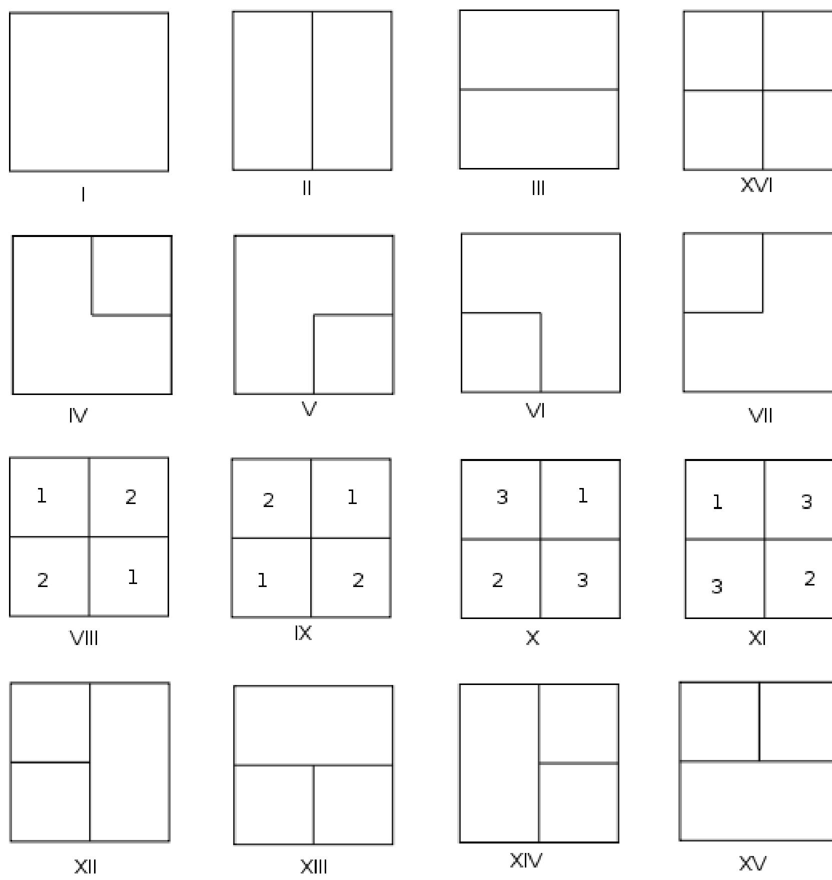

Figure 2.6: The various combinations possible with a four by four pixel block. Note that *viii* and *ix* are the same.

With four pixels, simple observation showed that there is a maximum of sixteen different patterns possible. Further investigation shows that there are actually fifteen for the purposes of this project (see figure 2.6). Thus, regardless of the pixel values they must form one of these patterns. Viewed from this perspective pixel similarity is simply a problem of finding which of the sixteen is the best match.

This may seem simple, but it is in fact an exacting and demanding process. One must be careful not to be too demanding in the criteria that decides the pattern match, or it is entirely possible that no match will be found (nothing will exactly fit the search if the criteria are defined too narrowly). Once again the problem of A, B vs A, C appears. The logic needs to be sufficiently rigorous to prohibit random edges from being formed, yet adequately flexible so a pattern is eventually made.

The best way to achieve this was found after much experimentation and revolves around comparison of differences. The crucial decider of homogeneity is the *difference* between each pixel. Once this is grasped, one can easily find the six difference combinations for the four pixel block. It is these differences, $A-B$, $A-C$, $A-D$, $C-D$, $B-C$ and $B-D$, that are used to form that basis of the decision making process. Once they have been computed (note that it is the absolute difference, negatives are not used), a set of comparisons are used. To prevent the logic from becoming to rigid, only the most desired patterns were explicitly defined.

Analysis of the fifteen combinations shown in figure 2.6 showed that a number would encourage the formation of new regions. If a single pixel was left in a solitary group, the chances were higher that it could form a new region when merged back into the main image. This is because it may not have been considered before and is therefore, by definition a new region (see section 2.3.1). This would be especially true if that lone pixel was the lower right (or D) pixel. As this pixel not part of any previous region in the main image, it would be guaranteed to

become a new region, at least temporarily. Thus all combinations that left this pixel in a group of one were undesirable, this included combinations *v, xi, xiii, xiv* and *xvi*. Therefore these were at the bottom of the list.

Considering the above in reverse, the most desirable combinations were those that maintained pixel D in as large a group as possible. As can be seen in figure 2.6, these sets were: *i, iv, vi* and *vii* followed by *ii, iii, viii, x* and *xii* through *xv*. The selection logic was arranged so that these best and the most distinctive candidates were looked for first, with explicit requirements. Following this several layers of less rigorous logic followed with the least desired combinations at the bottom, for groups that failed all other tests.

The initial tests were simple, using the six differences and the image tolerance (see section 2.3.1). If the largest difference was less than the tolerance, then all pixels were of a group, if the smallest difference was greater, they were all separate. Following this *ii, iii, xiii* and *ix* were also searched for explicitly, using exacting `if` statements that ensured the derided shapes existed. Then an `if` statement was constructed for each difference. If the considered difference was less than tolerance, then the number of shapes the block could be was restricted. For example if $A - B < T$ then all combinations that split A and B can be ignored. This list of probables can be reduced still further if once considers the combination that were rejected to reach this point. Often, there was only one possibility. In cases where there were more, they were selected by nested `if`, `else` statements that used other differences and guaranteed preference to preferred shapes. Thus an appropriate assignment was ensured, and all four pixel blocks assigned some split pattern.

This, when combined with the merging algorithm, produced much more stable results.

Final Splitting Algorithm

1. Obtain the four pixel block
2. Find the absolute differences of all the pixels.
3. Try and fit the desired patterns explicitly.
4. Try for the less desired patterns in order of precedence. Continue fitting patterns until one is selected.

Figure 2.7: The final splitting algorithm, where splitting was based on patterns in order of precedence

2.4 Image Pre-Processing

The output of the segmentation process described above was sufficient all one wished was to subdivide a still image. However, upon implementation it became aspirant that the process was rather too simple. Regions representing immobile objects tended to grow and shrink in an apparently random fashion. This was attributed to insufficient flexibility in the program relative to variable conditions in the image.

2.4.1 Noise Removal and Image Smoothing

The creation of regions was found to be highly susceptible to the image noise. As the initial camera was not of the highest quality (it was the stock machine attached to the author's laptop), steps had to be taken to alleviate the noise.

Some simple investigations[4] showed that the OpenCV library contained some

[4]It in necessary to confess that the research in this area was not as in depth as others. This problem was considered less important than others and less time was spent on it.

preexisting functions for image smoothing.

Image smoothing involves subjecting the image to a controlled blur, this smooths out the random high points in the image caused my noise (Bradski & Kaehler 2008). The difficulty with this, as might be imagined, is that important features can become obscured if the blur is severe enough. Initially, a standard Gaussian blur was used however it was found that the more complicated Bilateral filter would be able to perform a Gaussian blur on the interior of objects whilst preserving the edges, the effect of which is "typically to turn an image into what appears to be a watercolor painting of the same scene. This can be useful as an aid to segmenting the image" (Bradski & Kaehler 2008). This quote was taken at face value and a moderate Bilateral filter installed before the segmentation algorithm and run once over each frame.

2.4.2 Finding a Tolerance

All the segmentation methods described above required some form of homogeneity criterion or tolerance to work. This is yet another area upon which the literature was annoyingly glib. Because of this, several methods were tried to find a truly flexible and accurate tolerance. Indeed, for some time it was believed that one of the key problems with the regions stability was lack of a good tolerance value.

In original versions, the tolerance value was fixed to an arbitrary number at the start of the program. Clearly this was an unacceptable situation. Changing light patterns, movement, the intrusion of darker or lighter objects into the screen all had the effect of changing the relationship between various objects.

In order to alleviate this problem it was necessary to make the tolerance relative. That is, to recompute the tolerance for each frame. Thus, for an insignificant computational penalty, it was possible to ensure that the tolerance adapted for changing conditions.

While it is easy to comprehend the necessity for such a scheme, the literature provides very few explicit references regarding methodology. Most sources simply state the need for such a system, without describing the minutiae of execution. Initially, it was decided to find the maximum and minimum grey values in the given frame and set the tolerance to n% of the difference. This value n was found, after patient experimentation to be best set to 10%.

The disadvantage of this system is that it still contained an arbitrary quantity, the constant n. While the method was, in theory flexible, it wasn't much of an improvement over its predecessor, as a rule it was found that all "real" images contained a 0 and 255 pixel. So in effect the value of tolerance remained static.

The final modification was to base the tolerance on the standard deviation of the image. The standard deviation is the square root of sum of the average of the squares of the distance from the mean (Moore 1995). What that cumbersome descriptor means is that standard deviation is a measure of the spread of the image pixel values. In other words, how widely separate the numerical values of the pixels are. Thus an almost black image will have a very low standard deviation while a image of many sharp, vibrantly coloured objects will have a very high one.

However, the standard deviation, while tied to the image itself, evolves in precisely the opposite way to desired. If an image is largely blank, the tolerance should increase, to prevent every article of noise from becoming a region. If there are a multitude of shades, the tolerance should decrease, to capture each surface as a region. Thus the inversion of the standard deviation was finally used as a tolerance. As this was too small to be of practical value it was found that multiplication by 1000 would bring it up to a useful magnitude.

2.5 Potential Problems

Despite all the investigations and considerations mentioned, there remain a number of potential difficulties with the implemented method. The first of these arises when the merging process is considered. In order to accurately judge which groups to merge with what regions, a tally of the total number of pixels and the total grey-scale value of each region. This enables a mean to be computed and compared to the group mean to assess merging potential.

The difficulty that arises in how this information is stored. The program creates two vectors 256 entries long. The number of each entry is the output grey-scale value of the pixels filling it. This means that if pixel A is merged to a region of output value 45, then the 45th element in vector `RegionCount` is incremented by one and the 45th element in vector `RegionSum` is increased by the original value of A. In this way an average of all pixels of value n in the output image can be kept. However, this process does not take into account region congruity. By definition (Jain et al. 1995) pixels in a region must be both homogeneous *and* share a common edge. This approach tallies all the pixels in all the regions that are value n in the output, as though they were in one region.

Despite this problem, the practical results are still workable. Extensive though and consultation resulted in the conclusion that the problem outlined above was not significant (Low 15/12/2010 to October 27/10/2011). This is due to the fact that, while regions (as defined by `RegionCount` and `RegionSum`) may not be continuous, they are of similar grey value in the input image. This means that were they to be continuous, they would almost certainly be included in the same region anyway. This means that the region means will not be severely affected by this approach. Furthermore, any modification would result in a fundamental re-design of the segmentation algorithm. This was felt too drastic a remedy for such a problem.

Chapter 3

Distance and Approach

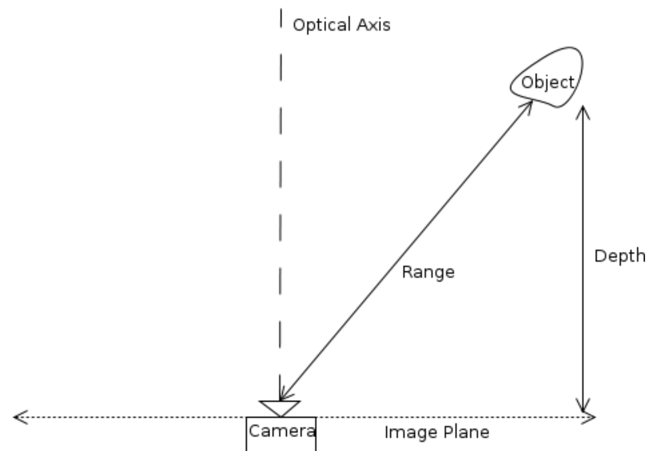

Figure 3.1: illustration of the difference between range and depth.

Once the image has been segmented, one might reasonably ask how can we find the location of these segmented objects, relative to the camera? Recall that the purpose of segmentation was to determine which pixels in the image belong to which object in reality. Thus there is a correspondence between the objects in the projected image and the objects before the viewer.

Before one sets about the task of establishing the distance between the segmented objects and the camera, it is necessary to bear certain restrictions in mind. First of these is the existence of only one camera. This is crucial as it implies that the objects cannot be found by direct triangulation. Secondly is the necessity for the chosen method to function uniformly and not to be confused by the changing environment, recall that the purpose of this exercise is to maneuver a mobile machine through a unknown obstacle field. Finally, the method must be computationally swift and inexpensive, as the focus is a autonomous real time implementation.

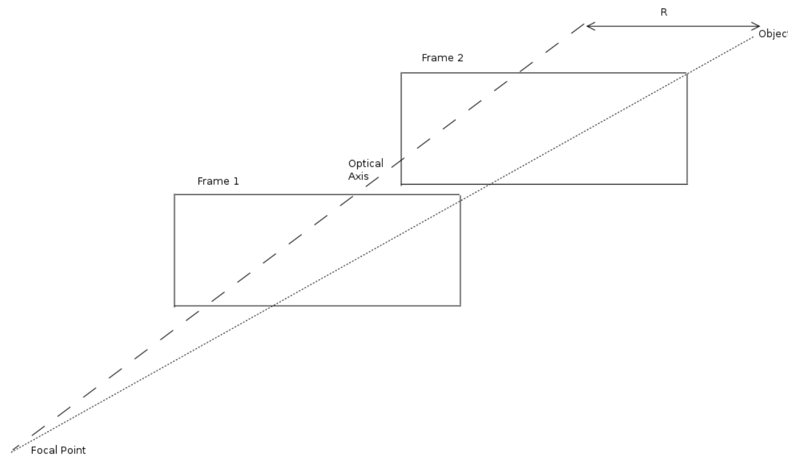

Figure 3.2: Stereo vision, using one camera and the frame difference to find depth.

3.1 Methods of Distance Estimation

The literature describes two broad methods of estimating the position and motion of an object relative to the camera. These two groups are, those based on stereo vision and those based on the evolution of the image over time (Jain et al. 1995), (Davies 1997). Obviously, true stereo vision is not an option (the focus is on monocular vision). This restriction led to the conclusion that the location of objects would have to be calculated based on analysis and information from a number of successive frames.

Before proceeding it is perhaps best to clarify some terminology. Depth in most sources refers to the distance from the object to the image plane. That is the length of the vector that stretches from the image plane to the object and is orthogonal to the image plane (see figure 3.1). Range, on the other hand, refers to the distance from the object to the focal point (Davies 1997). This is what most people mean when they discus "distance" in everyday conversation. These distinctions are important to keep in mind as some methods will relate to one or another. With these definitions clear, it becomes possible to conduct an investigation into the various methods of depth and range recovery.

As mentioned, there are two broad categories of distance estimation. Stereo appears the most frequently and although true stereo vision was not considered, it is possible to approximate it from the difference between two frames of a moving image. Classic stereo vision involves two cameras set some distance apart. Both record the same scene but from a different perspective. The discrepancy between the two provides the basis for calculations that describe depth and range (Jain et al. 1995). Such an arrangement would be familiar to most readers in the form of human eyes and binocular optical range finders.

While this arrangement is perhaps the most intuitive of distance judging techniques, the focus on monocular methods render it inappropriate in this case. Nevertheless it is possible to achieve the same effect from one camera and the image sequence. This is because, for a moving scene, each frame is slightly different than the last. Thus one may approximate the stereo arrangement by noting that as the scene moves, the camera is seeing the scene from a slightly different perspective (Davies 1997). This shift in perspective can be combined with some simple geometry and used to generate the distance from the camera to the point. Therefore there is now sufficient information to compute the range of any given point, if it has been tracked from frame to frame (see figure 3.2).

While potentially useful, this method suffers from one serious flaw. It cannot assess motion if that motion is directly long the optical axis (the line orthogonal to the image plane and passing through the focal point, the center of the image in effect). Examination of the equations used to compute range revealed that they fail when distance from the optical axis approaches zero (Davies 1997). This is because there ceases to be any discrepancy between the frames, the point of view becomes constant. Hence an object can approach at any speed, but if the line of approach is the optical axis, there will be no movement in the projected image plane. This means that objects approaching from directly ahead remain invisible. This clearly renders it an unsuitable method for this application.

Despite this disadvantage it was briefly considered as a viable option for when the

robot was turning. At early stages, it was assumed that the hypothetical robot would be able to execute a zero point turn (also it was assumed that it would be desirable to do this). As other methods considered (especially Looming, the selected method, see section 3.2) require some forward component of motion, stereo from motion was considered an ideal alternative. As the camera turned, all objects in frame would undergo motion from one frame to the next. There would be no blind spot as described above, because all image motion vectors would have been orthogonal to the optical axis.

This effect would enable the range to be calculated while turning, providing a good estimation of range for when the rotation had ceased and other methods could be resumed. Otherwise, it was thought that the machine would have to shuffle backwards and forwards to artificially create motion and so orient itself after each rotation.

In the event, it was found that the hardware envisaged would not necessarily need to perform this maneuver. Every turn could be accompanied by some forward motion component. Also, it was observed (Low 15/12/2010 to October 27/10/2011) that the movement from frame to frame as the machine turned may be insufficient to enable accurate computations to be made. Thus, for these reasons, Looming was recommended as a preferred alternative (Low 15/12/2010 to October 27/10/2011). Nevertheless, stereo from motion remains a potentially powerful tool, especially for rotational motion.

Other methods of distance estimation were reviewed and considered. However many dealt with multi camera platforms, or amalgamating cameras with other forms of sensors, both clearly inapplicable. Others, such as methods based on optical flow alone were discounted after advice indicated that they would be unsatisfactory (Low 15/12/2010 to October 27/10/2011).

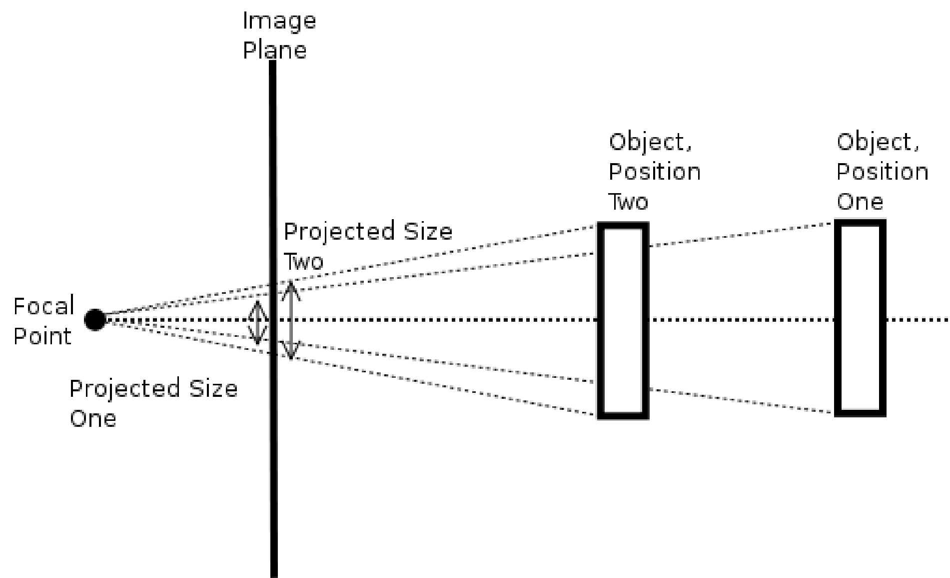

Figure 3.3: Graphical depiction of Looming using projected area.

3.2 Looming

A review of literature and certain advice, indicated that Looming is potentially the most useful method of range estimation. It possessed the potential for the most intuitive, simple results, it matched perfectly with the segmentation approach and was computationally simple. Furthermore it could be easily implemented with monocular vision, rendering it particularly suitable for the aims of the project.

Looming is a generic term given to a number of methods, all based around the fundamental premise that objects look larger when closer then when they are further away (Sahin & Gaudiano 1998). This is intuitive and based on the size of the projection on the image plane (see figure 3.3). More accurately, Looming is founded on the concept that as objects approach, certain measurable characteristics change with time. Identifiers such as physical range, projected area, image irradiance, image focus and so on may be tracked over time and used to compute a "Looming Value" (Raviv & Joarder 2000). This value, referred to as L, is related to the *range* from the camera to the object. With this value it is possible

to imply the range of an object without further calculation (Raviv 1995). The quantity L is defined by the following equation,

$$L = \frac{\frac{dR}{dt}}{R} \tag{3.1}$$

Where:

L is the Looming Value ($[time]^{-1}$)
R is the Range (m)

Thus it is evident that there is a relationship between the Looming value L, and the range R. Notice that L is proportional not only to the range, but also to the rate of change of range, the approach of the object. Thus if L can be calculated another way, one could use the value of L for any given point to imply the range of that point. Therefore, one defines L as:

$$L = \frac{\frac{dg}{dt}}{g} \tag{3.2}$$

Where g is some property of the image that may be calculated (Raviv 1995).

Recalling that Looming is based on the idea that aspects of an image region change as the real object approaches, it is possible to define L in terms of one of these changing aspects. The most obvious solution is to simply use the change in region area. Although this is possible, other aspects may be more useful. The other region properties include: irradiance, texture and blur (Raviv 1995). These four features occur repeatedly in the literature.

3.2.1 Area

Area is the most intuitive of Looming estimators. It is clear that as one approaches a rigid object, it appears larger. This is clearly illustrated in figure 3.3.

Notice that the projected size of the object is increasing as the object approaches. Mathematically, the value L can be expressed as:

$$L = \frac{\frac{dA}{dt}}{A} \tag{3.3}$$

While it is the simplest to explain and illustrate, the area method is flawed in several important ways. Firstly, the area of the regions as derived by the segmentation algorithm (see section 2.3) was found to fluctuate in practice. When the segmentation program was run, the area of the segments was not stable enough to produce consistent values of A (this is discussed further in section 5.2). Secondly, and more importantly, the area method is limited by the field of view (Sahin & Gaudiano 1998).

In order to accurately compute L from area or apparent size, one must have the region in question fully in the frame in at least one dimension. If an object approaches so that it begins to move out of the field of view, the total number of pixels in that region begins to decline. Even though the object is still Looming larger and approaching, the number of pixels that are used to represent it is decreasing as the region slides off the edge of the image. Thus the Looming value L would be decreasing even as the object is actually Looming larger.

This can be overcome to some degree by using length instead of area to measure L (as suggested in (Sahin & Gaudiano 1998)). This would involve using the length of the region in a given dimension to compute L. This length would be selected as the dimension that grows as the object approaches. While possible, this was considered too inelegant a solution. To implement it fully the computer would need to be able to determine which dimension was still fully in the screen. This implies tracking and remembering the growth of the object over its visual history and determining which dimension would be best to track. This was felt to be too complex an approach.

3.2.2 Irradiance

The illumination approach to Looming is yet another method of estimating the range through the use of image data. In this case the image data used is the temporal change in image irradiance (Raviv & Joarder 2000). Where "irradiance" is some measure of the reflected light off the object surface. The idea is that this will noticeably alter as the camera approaches the object, thus providing the Looming indicator. The literature indicates that there are three approaches to this method. All however, rely on the assumption that the surfaces in view are Lambertian surfaces.

A Lambertian surface is essentially a surface that appears equally bright from all viewing directions (Jain et al. 1995). This was considered a risky assumption, considering that many surfaces in the field of view may not be strictly Lambertian. This was a key reason in the decision to abandon this line of research.

The differing methods are based around the movement of the light source. In the first case, the light source is stationary, while the camera moves. This is perhaps the most apt approach to the problem for a mobile robot (assuming the machine is not carrying its own light source). However, the method described in (Raviv & Joarder 2000) makes no mention of the appearance of new light sources, as would certainly happen as the machine moves down a corridor. Perhaps more importantly this method requires the calculation of the angle θ. This is the angle between the surface normal and the optical axis. While possible, the computation of this angle was felt initially to be too complex a task in comparison with other methods.

The second method is based around the assumption that the light source is moving with the camera (Raviv & Joarder 2000). This could be easily achieved for a mobile robot, however once again a Lambertian surface is required. Also, in common with the first method, the angle θ is also needed.

These complexities ensured that Looming through irradiance was abandoned early as a potential method. Although they could probably be solved, other methods were considered easier to implement and more reliable in operation.

3.2.3 Texture

Looming through texture is yet another method of gauging object approach. Fundamentally similar to all other Looming methods, the texture approach takes as its changing variable the texture density. The idea is that as one approaches, the texture of an object becomes more intricate.

To gauge this, one defines a "textel" (sometimes given other, similar names) as a description of the texture pattern. The assumption is that any texture is comprised of repeating units. These "textels" in summation comprise the entire pattern (Pietikäinen 2000). As described in (Raviv 1995), the increase in "textels" indicates an increase in the intricacy of the pattern and thus the approach of the object. The general idea being that the more complexity one can see in an object, the closer it must be.

This method of Looming was discounted for two important reasons. One, it appeared unduly complex. The method of establishing descriptors for textures (as described in (Jain et al. 1995) and (Pietikäinen 2000)) was excessively intricate. More importantly, it was expected that many objects viewed as part of the obstacle navigation application would be planar surfaces without significant texture (walls, boxes etc) and thus the entire approach was deemed unsuitable for the project.

3.2.4 Blur

The final Looming method mentioned in the literature is Looming through radius of blur. If one considers a camera focused at a point, all points at a different distance from the camera than the focused point will exhibit some degree of defocus or blur. The edges of each object will be fuzzy in a way which is proportional to its distance from the camera. However, if the camera is focused at infinity, *all* points not at infinity will exhibit some degree of blur, proportional to their proximity to the camera. This means that objects in the far distance will appear almost clear, while objects in the foreground will be badly out of focus. The evolution of this optical blur magnitude over time can be used to compute the Looming value (Raviv 1995).

$$L = \frac{\frac{dr}{dt}}{r} \tag{3.4}$$

Equation 3.4 illustrates the relationship between the Looming value L and the radius of blur r (Raviv 1995). However, to employ this equation, it becomes necessary to somehow rigorously define blur. Some value which describes blur at a given point must be obtained from the image data.

It transpires that the value in question can be taken as the radius of the point spread function (PSF). This is the function which describes the blur circle and when convolved with the original image produces the blur (Subbarao 1987). One can say that r is some quantity that can be obtained from this function (Raviv & Joarder 2000). If this value of r can be recovered, it could be possible to compute L.

The challenge is thus the recovery of this radius. Many sources suggest that blur can be modeled acceptably by a two dimensional Gaussian function or a rotated one dimensional Gaussian (Subbarao 1987), (Luxen & Förstner 2002) . Using this model (see equation 3.5), one may use the value of σ (in statistics this is the

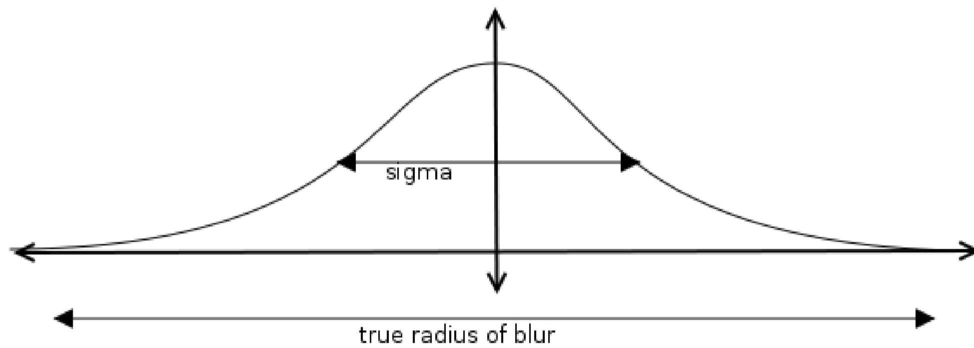

Figure 3.4: The Gaussian curve, showing standard deviation (σ) and the true radius of blur

standard deviation (De Veaux, Velleman & E. 2004) as r. Note that it is not r, but rather proportional to r, however it is always proportional by the same relationship and is thus an acceptable measure of r (Raviv 1995).

This relationship is demonstrated in figure 3.4. Notice that the true radius of blur is where the curve becomes practically insignificant.

$$g(n, \sigma) = \frac{1}{\sqrt{2\pi\sigma}} \exp\left(-\frac{n^2}{2\sigma^2}\right) \tag{3.5}$$

Where,

n is the pixel range around the current position.
σ is the standard deviation of the Gaussian distribution.

Notice in equation 3.5 there is only one variable σ. In this application it is assumed that the blur is symmetric. That is, it can be modeled by a rotated one dimensional Gaussian. This means that only one value, σ need be recovered. There exist models for two dimensional Gaussian blur, with two orthogonal values σ_1 and σ_2 (Luxen & Förstner 2002), however this was felt to add needless complexity when there was every reason to suppose that one σ would be suffi-

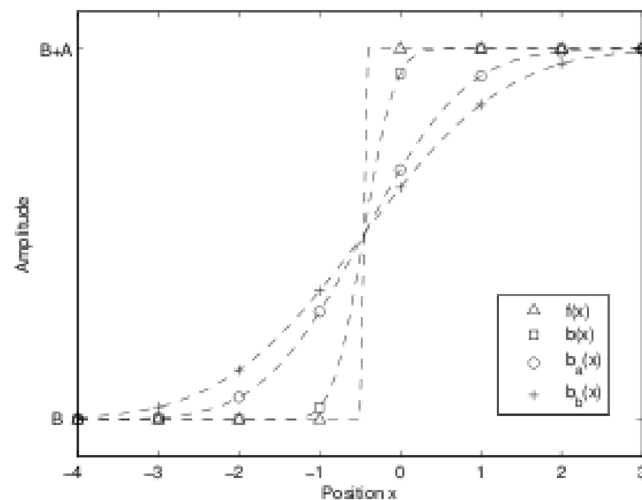

Figure 3.5: Plot of various step functions representing the region edge with varying degrees of ideal blur. $f(x)$ is the original step, with $b(x)$ being the camera's blurred edge and $b_a(x)$ and $b_b(x)$ being the re-blurred edges.

cient. Naturally, this assumes that the blur is symmetric. At this stage it will be considered to be so, this will have to be tested in implementation.

At this point the question became how to recover this value from the image data. Research has indicated that this can be done several ways. Firstly, there exist iterative methods (Hu & de Haan 2006). These were excluded from the investigation on the grounds of computational complexity. It is important to recall that reducing computational complexity is critical, given the real time project specification.

Further reading indicated the existence of other low cost blur estimation algorithms, especially one based on blurring the image several times with a known blur (see (Hu & de Haan 2006)).

In essence, the methodology eventually adopted relies on the difference between blurred images. The original image is taken and re-blurred several times. The differences between the images at each pixel location can be used to produce an estimation for blur radius.

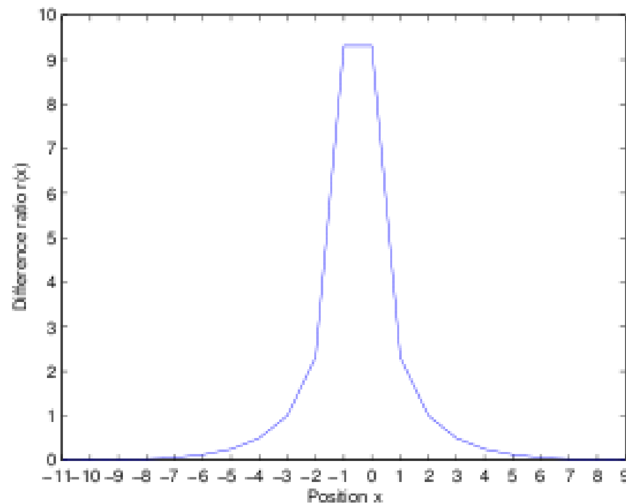

Figure 3.6: Plot of R_{max} values recovered in Note the symmetrical nature of the plot around the region boundary.

Consider the step function description of an ideal edge. Ideally, between one object and another, there is a sharp demarcation. This demarcation could be used in an edge detection algorithm to describe the object edge (see section 2.2.2). Ideally this is a step, as shown by the $f(x)$ line in figure 3.5, however in a real image, the edges of objects are seldom so crisp. There will be a certain amount of blur present. If the camera is de-focused, as discussed above, this blur will be guaranteed. This causes a "softening" of the edge step function, as illustrated by the $b(x)$ curve in figure 3.5. Notice that there is no longer a sharp drop between regions, but that the edge has been rounded. pixels of a medium value now lie between the two regions. This effect becomes more pronounced as the blur becomes more severe (lines $b_a(x)$ and $b_b(x)$ in figure 3.5)

It is important to note that the values illustrated in 3.5 represent the ideal blur edges. This is what would be produced if an ideal Gaussian blur was used. Mathematically these curves exhibit the perfect result. However, when implemented they were not so exact (see figure 5.11).

How can this property be used? Recall that it is assumed that these blurred edges

can be produced by performing a convolution on the original $f(x)$ step with a Gaussian function. The Gaussian (equation 3.5), can be expressed as:

$$g(n,\sigma) = \frac{1}{\sqrt{2\pi}\sigma} \exp\left(-\frac{n^2}{2\sigma^2}\right) \tag{3.6}$$

Where n is the position in the image and σ is the standard deviation. From this it is possible to show that [1] the blared step edge will be:

$$\begin{aligned} b(x) &= \sum_{n\in I} f(x-n)g(n,\sigma) \\ &= \begin{cases} \frac{A}{2}\left(1 + \sum_{n=-x}^{x} g(n,\sigma)\right) + B, & x \geq 0 \\ \frac{A}{2}\left(1 - \sum_{n=-x}^{x} g(n,\sigma)\right) + B, & x \leq 0 \end{cases} \end{aligned} \tag{3.7}$$

Where A and B are the constants that define the idealised step function shown in figure 3.5. The original Gaussian blur is now convolved with two new blurs. Using the property illustrated in equation 3.2.4, the new blurred edge $b_a x$ is illustrated in equation 3.2.4.

$$g(n,\sigma_1) * g(n,\sigma_2) = g(n, \sqrt{\sigma_1^2 - \sigma_2^2}) \tag{3.8}$$

$$b_a(x) = \begin{cases} \frac{A}{2}\left(1 + \sum_{n=-x}^{x} g(n, \sqrt{\sigma^2 - \sigma_a^2})\right) + B, & x \geq 0 \\ \frac{A}{2}\left(1 - \sum_{n=-x}^{x} g(n, \sqrt{\sigma^2 - \sigma_a^2})\right) + B, & x \leq 0 \end{cases} \tag{3.9}$$

[1] All calculations here reproduced from (Hu & de Haan 2006).

The same equation can be used to describe the result of the second re-blur, except σ_b is used instead of σ_a. Note that the two convolutions are separate operations, the initial blurred image is *not* convolved sequentially. Thus there are two re-blurred images that both started as the initial image.

The next step is to make the final operation independent of the constants A and B. To do this, the ratio of differences ($r(x)$) between the two blurred edges is computed:

$$r(x) = \frac{b(x) - b_a(x)}{b_a(x) - b_b(x)} \tag{3.10}$$

This equations for $b_a(x)$ and $b_b(x)$ (equation 3.2.4) can be substituted into equation 3.2.4. Using the properties of the edge location it is possible to say that:

$$r(x)_{max} = r(-1) = r(0) = \frac{\frac{1}{\sigma} - \frac{1}{\sqrt{\sigma^2+\sigma_a^2}}}{\frac{1}{\sqrt{\sigma^2+\sigma_a^2}} - \frac{1}{\sqrt{\sigma^2+\sigma_b^2}}} \tag{3.11}$$

This can be simplified further by the following assumption:

$$\sqrt{\sigma^2 + \sigma_a^2} \approx \sigma_a \tag{3.12}$$

Needles to say, the approximation in equation 3.2.4 is only valid if $\sigma_a >> \sigma$. In addition one also finds that $\sigma_b >> \sigma$ and $\sigma_a > \sigma_b$. If this is assumed, it is possible

to find an expression for the initial blur:

$$\sigma \approx \frac{\sigma_a \sigma_b}{(\sigma_a - \sigma_b) r(x)_{max} + \sigma_b} \tag{3.13}$$

By examining figure 3.6, it can be seen that the maximum value of $r(x)$ is always at the region boundary. Thus the value of $r(x)_{max}$ can be found by substituting values into equation 3.2.4 at that point. Using these properties, assumptions and equation 3.2.4, it is possible to recover an approximation of the original blur at the region boundary. (fuller working can be seen in (Hu & de Haan 2006)).

This method possess a number of great advantages over other Looming methods. Firstly, it is applicable at a point, thus computational costs could be reduced by computing the Looming value at one or at most several points in a region and applying that value to the whole region (as it is assumed that a region is a flat plane). Secondly, the above method of blur involves less assumptions than irradiance, less complexity than texture and is not subject to the field of view problems associated with area. For these reasons it was eventually selected as the looming identifier.

3.3 Implementation

3.3.1 Blur Calculation

Now that a method of gauging blur has been selected, one needs to consider how to implement this method in practice. As discussed, the blur recovery method described in (Hu & de Haan 2006) can be (in theory) applied a single point on the edge of a region. This is because the maximum blur ratio R_{max} should occur at an edge (see above).

Consider the regions approach described. It is possible to swiftly find the edge of a given region, by simply going from any point in the region to where the next pixel is no longer of the same region as the current. The maximum blur ration should be at this point. Thus the blur radius can be computed at any region edge [2].

To avoid errors, it would be inadvisable to simply take the result as computed at one point. Ideally the result should be the average of all blur values around the entire region edge. However to compute the blur at every pixel around the edge was felt to be too computationally expensive. Thus a compromise was selected, where the blur was computed at the four cardinal points of each region.

To fully describe this method it becomes necessary to move slightly ahead to section 4.4.1. For reasons which will be expounded later, it was found convenient to compute the region centroids. Thus a known handle for the region geometry existed and could be utilised in blur finding. To find the blur at the four cardinal points, one began at the centroid and advanced East, West, North and South to the region borders. At these points, the blur radius was computed. The average of these four comprised the blur estimation for the region.

This method only held true if the region was regular and not concave in shape. If the latter, the centroid was likely to lie outside the region. To account for this, a simple test was performed on each region before commencing blur calculations. The grey value of the pixel representing the centroid was checked against the grey value of the region. If they matched, the centroid was in the region and one may proceeded as described. If not, then a slight modification to the above method was needed.

If the centroid was found to be outside its region, the same process applied with the distinction that the algorithm would go one pixel beyond the region border,

[2] This would appear to lend weight to the Edge Tracing method of obstacle detection outlined in section 2.2.2, however regions were already selected as outlined.

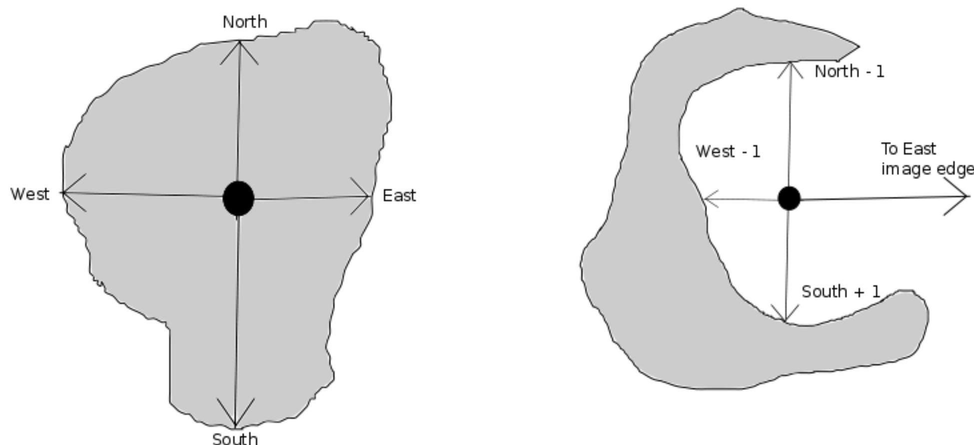

Figure 3.7: Illustration of finding the four points of a region for blur computation. The left hand image shows the case of the centroid being in the region. The right hand image is for the centroid being out of the region.

into the next region before computing the blur. However, this means that in at least one direction, the edge of the region will never be encountered. If the region is a crescent with the concave section facing east, the eastern region edge will always be to the west of the centroid. Thus going east from the centroid causes one to fetch up on the edge of the image. For this reason a fail-safe was included in the final algorithm to it from trying to find the blur at the edge of the image.

Thus, this method implies that the final blur result would be an average of at least two, widely separate points. While this is not as good an estimation as finding the blur at every point, it was considered a beginning. It was also hoped that the blur recovery would be accurate enough to make the four point approach sufficient. This was supported by the encouraging results obtained from (Hu & de Haan 2006).

3.3.2 Looming Calculation

Having found blur, it became necessary to create the means by which this blur value would be used to find L, the Looming value. This was done in amalgamation with the tracking algorithm (this algorithm will be described in detail in section 4.4, at this stage suffice it to say that a correlation is found between a region and its corresponding match in the previous frame). Once value for blur has been found it is recorded and combined with the value obtained for that region in the previous frame to obtain Looming as per equation 3.4.

While there is little to be said on the calculation of blur (equation 3.4 is fairly simple), it should be mentioned that there are several special conditions for L. Because of the realities of implementation there will exist times where there is no information about the previous blur. In this case, the Looming value was set to zero. This can cause confusion as there are also times when the Looming value is legitimately zero, for example when the region is not moving. However, it was considered unlikely that any region would ever exhibit exactly zero movement, therefore this duplication was considered acceptable.

The other special case is that of r, the current radius blur being zero. Mathematically, when this happens equation 3.4 goes to infinity. In actuality what this means is that the object in question is at infinity (the camera being focused at infinity, an object at that point produces zero blur). Therefore, in the program if the magnitude of the current blur is less than some very small number, the Looming value is assigned to some very large, arbitrary number (10^{10}).

The final point that should be mentioned is the significance of a negative L. As the derivative in equation 3.4 is implemented by dividing the blur difference by the time difference, the sign of the resultant L indicates whether the object in question is advancing or retreating. As the blur difference is computed by taking the current blur from the previous blur, an object of increasing blur (moving towards the camera) will have a positive value of L.

3.4 Avoidance

Once all the segmentation, tracking and Looming calculations were complete, it became necessary to use the information so gained to direct the machine around obstacles. Using the information gathered from these sections it should be possible to avoid obstacles. However, the final results (see section 5.5) were too inexact to permit a stable avoidance algorithm being implemented.

In addition, it was found that most sources ((Wang, Xu, Guzman, Jarvis, Goh & Chan 2001), (Sahin & Gaudiano 1998), etc.). Were irritatingly glib concerning the actual method used to avoid obstacles. Most of the sources devote the majority of their time to extracting the relevant information, not discussing exactly how it would be used. Thus very little information could be obtained on the best method of using the Looming data. In the event this was not critical, as the Looming results were too unstable to be employed anyway. However for the sake of completeness there follows a brief description of the planed avoidance method. This was never developed fully as it became clear from a relatively early stage that it would not be implemented.

3.4.1 Approach Categorisation

Recall that the purpose of the exercise is to avoid the various objects in view. This implies steering away from the objects as they approach to avoid collision. The Looming algorithm has provided an estimation of the object's approach and the segmented object's location in the original image gives some indication of the position in the plane (left and right). From this the computer may place an object in terms of two dimensional position and approach.

Steering would be accomplished by creating an imaginary line around the machine. This imaginary line would be based on Looming value, all objects that

exhibit an Looming value larger than this limit will be treated as impending collisions. This limiting Looming value would have to be considered on the basis of the minimum range desired.

3.4.2 Direction Decisions

Now that the approach of a object can be categorised, what needs to be done with this information? If an “avoidable” object has been detected, the motors need to be directed to avoid the object. It is obviously insufficient to simply steer randomly and hope that avoidance just happens.

To resolve this, the position of the object in the image plane is used. Chapter 4.4 discusses the computation of region centroids. The location of these centroids in the image plane would give some indication of the position of the object in space. Thus motor control can be directed proportionally to the location of the object left or right of the center plane. For example, if the object in view was 10% of the image width from the center of the image, the motors would turn the machine at a rate proportional to that 10%. Other factors that would need to be considered in deciding motor speed would be the Looming value of the object at that point. Using this method, proportional control could be achieved. It was simply assumed that all obstacles would be positive, in that holes in the ground would not occur (as in (Wang et al. 2001)).

Chapter 4

Tracking and Correlation

4.1 Frame to Frame Correlation

Upon reading the above sections the reader will note that most of the methodologies described (particularly looming and related algorithms) rely on information from a previous frame. This in turn implies some form of correlation from frame to frame. The path of an object must be known through a series of progressive frames to acquire this information. While human viewers of easily and intuitively make connections between objects in successive frames (based on similarities in shape colour etc.), the computer has no instinctive ability to do this. As in region segmentation (see chapter 2), one must be careful not to anthropomorphize the computer's operation.

To fully define the idea of frame correspondence, the concept of "optical flow" has been created. (Bradski & Kaehler 2008) describes optical flow as "the quantitative assessment of the two dimensional movement of pixels from frame to frame" and notes that values may be assigned to all pixels (dense optical flow) or only some (sparse optical flow). This can be achieved a number of ways, with differing emphasis. In this case, the real-time nature of the application necessitated that a computationally simple and rapid method be adopted. However, this had to be balanced with the critical importance of frame to frame correspondence for the Looming methods[1].

The first issue that needs to be addressed is sparse versus dense optical flow. Here the choice is relatively simple, assigning movement values to every pixel in the image would be a complex and time consuming process (Jain et al. 1995). As the image is segmented into distinct regions anyway, there is no need to assign values to every pixel as it was assumed that all pixels in a region move in tandem. Thus the best method would be to compute the movement of each region and assign that value to every pixel in the said region. This implies a sparse optical flow

[1] It is also possible to compute approach directly from optical flow. However, advice suggested that this would not be advisable (see above)

method.

4.2 Point or Feature Tracking

When creating an algorithm for computing optical flow and correspondence there is another fundamental choice that needs to be made. Will it be feature or region based? The former tracing method is based on distinct features or points in the image, while the latter attempts to track large objects as they evolve in time (Davies 1997).

In the feature based approach, one takes points that possess good track-able properties (as described in (Bradski & Kaehler 2008), these are usually corners of objects). One uses these properties to find similar pixels in the next frame, thus establishing continuity. As comparisons are generally based on grey-scale similarities, using these comparisons it is in principle possible to track every pixel in the image by finding the best local match for its numerical value in the next (or previous) frame. However this would be not only very time consuming, but in many real cases impossible (Neumann 1998). This is due to several problems with optical illusions.

Consider the following; pixels in the center of an object are all of a similar grey value. Thus as the objects move, only the pixels that represented the edges change in value. This means that the interior pixels cannot be tracked with any degree of reliability as they exhibit no significant change from frame to frame until they become the object's edge. One interior pixel could very well be any other (Jain et al. 1995).

This would seem to signify that only edges can be tracked successfully. This is partially true, however an examination of figure 4.1 shows that only edges perpendicular to the direction of motion can be successfully tracked.

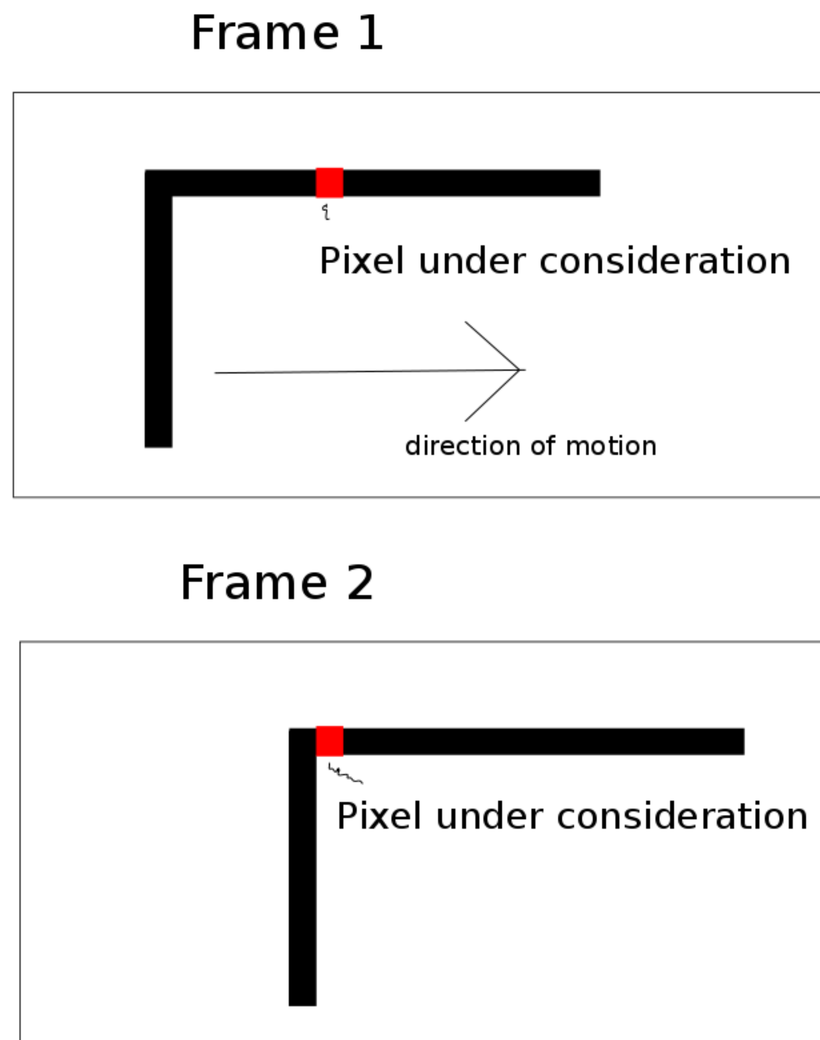

Figure 4.1: Illustration of the difficulties of tracking for non-orthogonal movement

Figure 4.1 shows clearly that as the top edge is parallel to the line of motion, the same problem applies. Any pixel in the top edge except the last two could be any other. This leads to the idea that only the corners of an object can be tracked (Bradski & Kaehler 2008). These unique pixels can be distinguished regardless of the motion direction. This is because as a point, they are always "orthogonal" to the direction of motion, no matter the direction the corners are always changing with time.

Thus the "features" that are tracked are the corners of the object. However, not all objects have traceable corners. A round ball for instance will probably not the produce sharp projections necessary to enable unambiguous tracking.

While the corner approach is fairly robust, inexpensive and can be performed on any image without much prior work, it has one important weakness when applied in this case. Feature tracking thus assumes that there is a track-able point in every object.

Consider a blank wall. This object is an untextured surface and as such probably has no features that are amenable to tracking. As discussed above a white pixel surrounded by other white pixels is impossible to track from frame to frame. In the case of a wall, it is probable that the edges of the wall are not in view (in fact, as the robot approaches closer and closer this becomes a certainty). Therefore it will be impossible to track large (and therefor important) regions from frame to frame, unless they have significant internal texture. The existence of such internal texture was considered too much of an assumption for such a crucial point.

4.3 Region Tracking

The second approach considered was region tracking. This method aims to find a correspondence between large regions, as distinct from individual pixels (Fuh & Maragos 1989). Where the feature based method assigns values of optical flow

to regions based on an individual pixel, the region tracking methods describe individual pixels through the movement of the corresponding regions.

The advantages of such a system are obvious in this context. When the image is already segmented (as in this case, see section 2.3), it is possible to use the existing regions to compute the motion of all pixels within them. To do this, it is assumed that all pixels in a given region move in the same fashion (Fuh, Maragos & Vincent 1993).

Furthermore, such a method is not constrained by the necessity of selecting "track-able" pixels. There is no necessity to select the corners or any other single point. The region's own geometry can be used to provide the unique tracking requirements and so divorce the tracking program from the necessity of considering individual pixels (Javed & Shah 2006). A consideration of the region's properties reveals at least three distinct quantities that, combined distinguish a region from its neighbours; the location of its centroid, the area of the region (or bounding box, (Javed & Shah 2006)) and the average grey-scale value of its component pixels.

4.4 Chosen Algorithm

Once the above problems had been carefully considered, it became clear that the region based approach was the most applicable. The existing segmentation of (hopefully) stable regions made it especially useful. To provide region correspondence between frames three properties were used: the location of the centroid, the area of the region and the average pixel value (Fuh et al. 1993). Other values that have been seen include, bounding box size (Javed & Shah 2006).

To implement this method, a critical assumption was made. All tracking algorithms and methods discovered during research were based on the assumption that a given feature, region or object moves very little from frame to frame. The

frame rate is such that one may presume an object's motion to be smooth, regular and minimal from frame to frame. This implies that the geometric location of an object's projection in one frame will be very close to its projection in another (Jain et al. 1995). While this seems obvious, it is important to mention because it relies on the frame rate being very fast relative to motion. As motion increases this assumption becomes increasingly inaccurate, thus reducing the accuracy and effectiveness of tracking (Fuh et al. 1993).

If one accepts this assumption, it is possible to employ a number of region identifiers to achieve correspondence. As mentioned, these are commonly the geometric center of the region and the average pixel grey-value. The geometric center, or centroid is the key indicator. This location is a physical spot on the image plane, but is not tied to any one pixel. Thus it is not susceptible to the aperture problem outlined above (or rather, as a unique point it is always a corner). If the frame rate is sufficiently rapid, one may assume that, in the next frame the closest centroid to the one under consideration in the previous frame is a match.

4.4.1 Centroid Calculation

In order to use the centroid of a region to provide correspondence, this property must first be calculated. The centroid of a shape refers the location of the geometric center. In this case, as the image region is a two dimensional object the centroid is the center of area and can be computed through use of equation 4.1.

$$y = \frac{\Sigma A \times y}{A} \tag{4.1}$$

Thus all that needs to be done to calculate the centroid is to find the position of every pixel in a region and the area of the region. This requires an algorithm capable of visiting every pixel in a region, preferably with the minimum of overhead (as it needed to perform this function for every region in the image) and recording

the location. The method selected for this was a simple recursion method. This is not the best mechanism, but was simplest to implement (Brookshaw 10/12/2010 to 27/10/2011).

Recursion

The recursion method chosen to count all pixels in the region is a crude method. While simple it is expensive in both time and memory, however it was relatively easy to implement.

In essence the algorithm runs through the image pixel by pixel. If the current pixel is part of the region in consideration it is recorded and its surrounding pixels checked. If these pixels are part of the region in consideration they are also recorded and *their* surrounds checked. This process continues until all pixels in a given region are visited[2].

A simple function "recursion"[3] is created and checks the surrounding pixels for any given point and flags pixels of the same region. If a surrounding pixel is of the same region, the function "recursion" is called again, to pursue this method with that pixel (the algorithm is given in figure 4.2). The implemented function used a "four-connectivity" pattern, checking the pixels North, South, East and West of the subject pixel. Although more complicated algorithms use all eight surrounding pixels, this approach was rejected as too complex to implement. This means that the function is called for every pixel that is found in the region. This is a very time consuming process.

To ensure that the recursion is limited to previously unchecked pixels in the one region, two comparisons are made with every new pixel. First, the new pixel is checked to see if it is of the same region currently being considered. Next its

[2]Remember that a regions is an area of pixels of the same grey value. Thus the boundaries of a region are simply where the colour changes to any other value

[3]See the centroid finding functions in Appendix B.1.3

Recursion Algorithm

1. Check current pixel
2. Record if criteria satisfied.
3. Check its surrounding pixels.
4. If they satisfy criteria, move to that pixel and return to 1.
5. Continue this process until all pixels accounted for.

Figure 4.2: the recursion algorithm as used by the tracking program

"flag" is checked to see if it has been tallied before. This flag is a counter assigned to each pixel (in reality a matrix of zeros the same size as the image). When a pixel is counted, the flag (corresponding entry in the matrix) set. If either the flag is set or the pixel is not part of the same region, then it is not recorded or included in the recursion (its neighbours are not checked either). This ensures that the algorithm (when all flags are checked), will stop and move onto the next region, not simply run round forever inside the same group.

What this means is that a given position in a region, the recursion algorithm will begin to expand until it has counted every pixel inside the region. However, it is evident that this is a very slow and laborious method. Despite this it was considered acceptable due to its ease of implementation. Given the time constraints of the project, such ease of implementation was considered more important than elegance in this case.

The real impact of this method is a slight, almost imperceptible delay in processing and the necessity of permitting the operating system access to a larger stack than usual. It was found that if this modification was not made, the amount of numbers needed to be held in memory grew unacceptably large for big regions. This lead to the program crashing with memory errors.

4.4.2 Data Recording

Once a centroid has been identified it needs must be stored. Remember that tracking will proceed by locating the best match for a given centroid in another image. Thus we will need several lists of centroids, one from the new image and another from the old. Each list will need to contain a table for each centroid, with data such as x, y location, average grey value, region label colour and so on. Given this requirement, the centroid location is stored in a linked list structure.

This list is a simple construct, its only difficulty being that we do not know in advance how many centroids we have (the exact number of regions in a given image is unknown). Thus we proceed as follows: Create a structure called "Centroid" this contains all the information pertinent to that region, x location, y location, grey sum, area, Looming value etc. The last entry in this list is a pointer to the next "Centroid" structure in the series. This creates a chain of centroids the last link of which points to nothing (House 1994).

This chain is then investigated and all centroids relating to regions of an area less than A are removed. This is to prevent tiny regions (on order of several pixels) from cluttering up the calculations and slowing the computational time. We are only interested in objects that are large enough to see and be a hazard, not tiny textures and grains on an object's surface. This raises the concern that important details may be lost. Thin wires for example, could represent an "obstacle" yet be ignored by this contingency. This was considered and thought to be too insignificant a probability to warrant the extra computational expense of tracking tiny regions. Of course, the minimum size tolerance could simply be reduced to zero to include all regions.

Once the list is cleaned to the essentials, we have a selection of centroids that we can track in the next frame.

4.4.3 Centroid Matching

To track, the program attempts to find correspondence between the various centroids from one frame to another. This can, as with most things, be done a number of ways. Most sources ((Fuh et al. 1993), (Hager & Belhumeur 1998), etc.) begin with creating a search area. This search area is the area around a given centroid that will be considered for correspondence. If a centroid from the other frame falls within this search area, it can be considered as a match for the current centroid.

The program defines the search area by creating a square of the same area as the region, centered on the region's centroid (this was considered the easiest shape to implement), recall that the centroid list contains the region's area. Thus it is a simple matter to define the square's side. This done one may search through the centroid list from the other frame for all centroids that fall within this area and compile a list of probables. This square approach was selected over the bounding box idea described in (Javed & Shah 2006) because it was considered easier to implement. Initially, it was believed that the bounding box would be much more difficult to code for no noticeable advantage.

Area alone however, is insufficient. There will probably be several centroids in the search area, therefore some method must be made to distinguish between them. The simplest method is just to choose the closest. However, this is still flawed. Analysis of the segmentation program showed that regions and objects will splinter into several regions (or re-merge) based on changes in the light, occlusion and perspective change. Thus what was a single region in the previous frame may be several regions in a new frame. Hence a algorithm that simply selects the closest is not enough, it must have sufficient flexibility to assign more then one region to another.

This is achieved by considering the difference in grey averages between regions. The centroid calculating algorithm was modified to include a grey sum entry and

the total grey value for a region was recorded. Thus when combined with region area it was possible to compute a grey mean. The probable regions are then assigned based on this grey mean. If the mean difference of the region under consideration and a probable region are within tolerance (a tolerance supplied and used by the segmentation algorithm for this frame), then a match is found. This method ensures that several matches are possible.

The flaw in this method is that several regions of similar colour could lie very close to each other in space. This objection is countered by considering if so close a match was possible then it would have been accounted for by the segmentation algorithm and the regions would be merged.

The other advantage of this method is that it takes into account the arrival and departure of regions as the image evolves. If a new object appears within frame, there will probably be no object (within the search area) that matches its grey mean value in the previous frame. Thus this new region will find no match when the grey mean is considered. This region will remain undefined until the next frame.

The final consideration is whether to consider the previous frame in relation to the next or the current frame in relation to the last, whether to go backwards or forwards. The final program operates from current to previous, finding a match for the current regions in the previous image. The reason for this is to ensure that new regions remain unmatched until the new frame (as above).

This is, in essence the method used to create correspondence of objects from frame to frame. The tests and trial results of this method may be found in sections 5.3 and 5.4.

Chapter 5

Results and Discussion

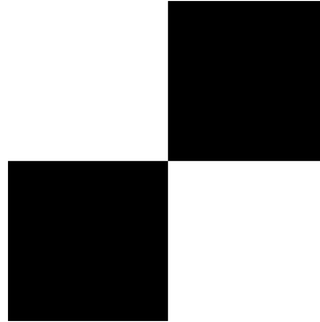

Figure 5.1: Original checkerboard test image. 600 by 600 pixels

The final program was long, complex and difficult to follow. While it certainly did not work as expected, it was difficult to see exactly what it *did* do. To resolve this each component part of the program was tested individually. The results showed that, while there were inaccuracies that prevented coherent function of the whole, each individual component could be shown to function correctly, within certain limits.

5.1 Ideal Test Images

No computer program in the author's experience has ever run perfectly when written. There is always some logical error that causes unexpected failure, unintended side effects or general mayhem. In many cases these errors are too subtle to be detected when the program is run in a real world environment. This was very much the case with the early stages of the project. Errors had a tendency to appear randomly, control of the image was difficult and some problems simply went unnoticed in the welter of real time image information.

To permit debugging, the programs involved were modified to operate on single, still images. Artificially perfect stills such as figure 5.1 were then fed in and the results examined. The results from these test images were simple enough that problems could be swiftly identified. Each major section of the main program

was removed from its place and tested individually. This permitted verification of the output from each major function.

The initial test image was a checkerboard of four squares (see figure 5.1). This painfully trivial image was selected as it was easy to produce four distinct regions. As the squares were absolutely black and white, there could be no ambiguity about the region's extent. This image and other similar pictures were then given to each of the main program functions and manipulated, modified and otherwise altered so that all the key aspects of the various functions could be tested.

Once the functions had passed the checkerboard test, they were given much more complex images. These more sophisticated images depicted "real life" type scenes and were intended to give a less rigorous "feel" for the workability of a given process. Unlike the idealised images there was no exact check for success on these images, rather a careful examination for any obvious ambiguities.

5.2 Segmentation Verification

Segmentation was easily tested using the checkerboard image. A guaranteed black and white image was fed in and an identical image returned. This may seem trivial, for segmentation is designed to produce such an image, therefore what is the point of giving it an already segmented picture? However the checkerboard test is doubly important for the segmentation function. If it cannot properly segment the image (i.e.: return *exactly* the same image it received), then the algorithm is randomly introducing variance into the image. This is critically important as such random segmentation means that the output image would be unstable. Regions would be appear and disappear randomly and tracking (and hence Looming and avoidance) would be similarly random. If the stability of the region algorithm is in question, the entire process becomes doubtful.

Thankfully, the tests run indicated that the output image was in every respect

identical to the input image. This was verified by the simple expedient of subtracting every pixel in the output from the input. All pixels in the resulting image were then verified to be exactly zero. This simple test was run many times without deviation. [1]

Thus it is known that the segmentation function does not introduce new regions randomly. However, we need to verify that more complex images are satisfactorily segmented. This could have been done by simply watching the real time output of the function. While this was done (many early and obvious problems were caught this way), it was felt to be too subjective an assessment. To create more scientific results, a complete image was introduced, segmented and the output compared with the input. This can be seen in figure 5.2. As these images illustrate, there is a fair correspondence between the two images, the edges of objects largely correspond to the edges of regions and the regions seem to represent the flat planes.

Given a slightly more complex image, it can be seen that segmentation is fairly accurate, with respect to region boundaries and object locations. However, these tests only covered the one image. What of the results for a sequence of images? This is the section where problems begin. When the segmentation algorithm is run for a sequence of images, it becomes apparent that there is some variation (this may be seen by simple observation of the feed from a motionless camera that has been processed by this function). Small variations in light and shade, minute movements in the scene, even the subtle effect of image noise all combine to make the regions in one frame different to the regions in the next.

This is difficult to demonstrate, however observe the images in figure 5.3. Note that the segmented regions are subtly different. All that has been done to this image is to increase its scale by 2% (with the top left hand corner in the same

[1] The test images are not replicated here (there is nothing to see). Interested parties with time on their hands may see the outline of the modifications needed to produce this test algorithm in appendix B.2.1

Figure 5.2: Illustration of segmentation of a complex image. On the left, the original image, on the right the segmented output. n.b. this image was used as a test image in (Hu & de Haan 2006) (among others) and was repeated as a test image here as the author found its provenance rather amusing. In actuality any reasonably uncomplicated image would do.

place). This implies that small changes in the image can create relatively large changes in the resultant segmented images. This has serious connotations for the tracking and Looming calculations.

Notice however, that (by and large) the background regions, where the original focus was less sharp, retain their original composition. The discrepancies are far less marked the further the original object was from the camera. Thus the greater the de-focus the more stable the segmentation. Conversely the regions closer to the observer are those that show the greatest discrepancy between segmented frames.

5.2.1 Effects of Pre-Processing Filtering

The pre-processing filtering with a Bilateral filter (as described in section 2.4.1, met with mixed success. Although, as advertised it smoothed the images and left

Figure 5.3: Illustration of the differences between two segmented images. The top set is 2% smaller than the bottom set. Notice the region discrepancies, especially in the foreground.

the edges intact, it was also inordinately slow.

The greatest effect of the noise removal section was a lengthy time delay. This delay (on order of several seconds) was longer than any other component part of the entire program and far too large for a real-time application.

Despite this, the need for some form of noise reduction was found to be very real, especially for poor quality cameras. The removal or reduction of the smoothing algorithm caused significantly visible increase in the instability of the segmentation routine. These same results were obtained when the numerical blur was substituted for an optical blur (a camera with a de-focused lens).

5.3 Centroid Verification

Like the segmentation function, the centroid functions were tested initially using the simple checkerboard shown in figure 5.6. The centroids of these squares were well known, their location can be simply calculated from the image size and the knowledge that the squares are of equal area.

Given these known positions and the simplicity of the image structure, it could be expected, with reasonable certainty that the centroid functions would return the centroid locations as the center of the squares. This they dutifully did (see figure 5.4).

Thus the ability of the centroid functions to perform rudimentary calculations was demonstrated. However, the real world will not return perfect squares, shapes will often have the centroid outside their dimensions. To test the functions on more complex images figure 5.5 was created. Again the centroids were found to be in expected positions.

Finally, having ascertained the effectiveness of the centroid function on test im-

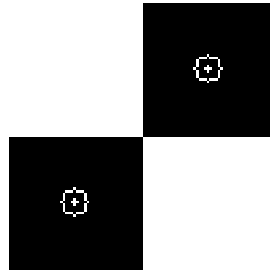

Figure 5.4: The test image, showing the successful finding of the centroids.

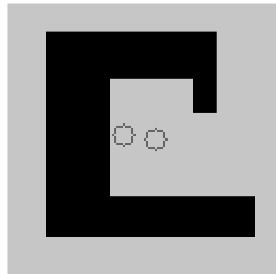

Figure 5.5: The test more complex test image image, again showing the successful finding of the centroids.

ages, it was incorporated into a test function with the segmentation function and given "real" images. These test images showed that centroids were being computed reliably for all regions. This held true for any test image no matter the number of times run. The accuracy of the placement was more difficult to ascertain, as "real" image regions tended to be very irregular. However visual inspections returned no anomalies and the x, y positions were the same for each image, regardless of the number of times the test was run.

5.4 Tracking Testing

Before the Looming algorithm could be implemented fully it was necessary to investigate the efficacy of the tracking algorithm. This was first accomplished through use of the idealised test image 5.6. A test algorithm was produced which ran the tracking function on only two images. The intention was that the two

Figure 5.6: Checkerboard test image distorted for tracking. 600 by 600 pixels

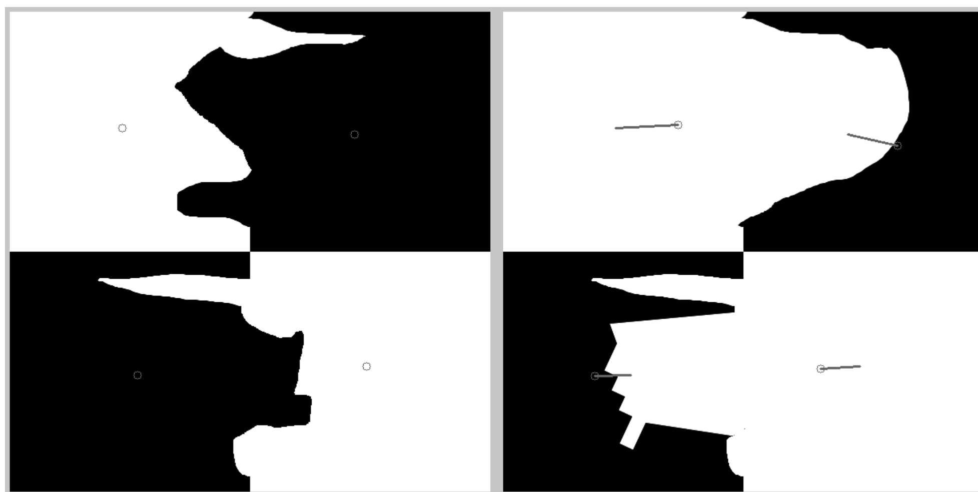

Figure 5.7: Distorted checkerboard showing tracked centroids. The grey line from the centroids in the right hand figure shows the computed position of the centroids in the previous (left hand) frame.

images would be slightly dissimilar, but not so different that tracking would be problematic. Recall that tracking assumes that there is only a small movement from frame to frame.

To this end, figure 5.1 was given to the test algorithm as both image one and two. The intention of this seemingly trivial exercise was to see if the tracking algorithm could successfully recognise that when no movement occurs, the centroids remain in the same location. This would test the fundamental concepts of the tracking program and ensure that no random element was present at so simple a level. This test worked successfully over a number of trials and indicated that tracking was at least possible by the method outlined.

The next step was to deform the ideal image, figure 5.1 so that the tracking algorithm could be tested in a more complex environment. To this end, figure 5.1 was modified to figure 5.6. This retained the black and white perfect regions, but modified their geometry so the centroid was forced to move. Once again, when run the results clearly showed that the centroids can be tracked via the methods used (see figure 5.7).

However, up to the present only perfect test images have been considered. It is crucial to know how the system will respond to the more complex environment it is likely to encounter when run. To determine this, the test images used in figure 5.3 were given to the tracking test function. As mentioned above, these two differ by 2%. This is analogous to a small shift of the camera closer to the object. To enable the tracking function to work, the segmented versions of these images (the right hand images in figure 5.3) were used. The results (shown in figure 5.8) are somewhat mixed. Some regions have tracked successfully and their corresponding regions in the previous frame are as expected. However these represent a small proportion. It was found that less than 50% of the regions had been tracked at all (roughly 20 to 55% for repeated tests with various images) and of these, one can see that some are not to the correct region.

When the full real time program was run, it was found that "tracked regions" (those regions for which any match was found in the previous frame) ran to about 60 to 70% of the total *if* the image was held stationary (no camera motion, or motion in the image, no changes in light intensity, etc). If the camera was moved, this fell to around 50%, as found in the static test frames.

Interestingly, if the camera is de-focused so the image exhibits a severe blur, the incidence of successful tracking rises to around 80 to 85% for a stationary camera and about 75% for a moving one. It was found that the magnitude of the blur was not overly significant and could be judged approximately by eye to find the optimum results. If the blur was such that objects in the fore ground were just distinguishable from objects in the background, then the results were noticeably improved.

Upon receipt of these discouraging results attempts were made to investigate the efficacy of the tracking algorithm itself. It was suspected that, like the segmentation algorithm, it was introducing randomness into the program. To verify this, the tracking test function was given exactly the same still image to segment and track. This returned a successful tracking rate of 100%, repeatedly and for all images tried. This indicated that the algorithm itself was functioning successfully. It also indicated that the camera, no matter how still it is held, does not return exactly the same image from frame to frame.

Despite this, the unsatisfactory results for moving images implied that the temporal evolution of the image was less well understood that desired. This almost certainly had effect on the results for Looming and avoidance (see below).

5.5 Blur Estimation

Having established that the segmentation, centroid and tracking functions work (at least in controlled situations), it became necessary to establish the efficacy

Figure 5.8: The results of tracking the segmented images in figure 5.3. The line from each centroid indicates the calculated position of that region in the previous frame.

Figure 5.9: The ideal blur test image. It is simply a two square checkerboard 400 by 200 pixels. Note the grey values used to colour the squares are not black and white.

of the blur recovery function. This function is of pivotal importance for the Looming calculations, without success here, the Looming (and hence avoidance and navigation) would be forced to rely on some other, less satisfactory method, such as area (see section 3.2.1).

The aim of the initial experiments was to reproduce the results claimed in (Hu & de Haan 2006). If successful in this endeavour, the program would then be expanded to operate on general regions as might be encountered in the real images. Once satisfactory results were obtained in a more complex field, it was felt that it would be safe to incorporate the blur algorithm into the real time program. The output could then be used to calculate Looming.

To determine the abilities of the blur function, the much used checkerboard (figure 5.6) was again employed. The Open CV library contains the useful function `cvSmooth`, which can be used to apply a Gaussian blur to an image (recall that it was assumed that the camera de-focus could be approximated by a Gaussian blur, see section 3.2.4). Using this function, a blur with a known radius (σ) was applied to the image. It was intended that the blur function recover this value. This was an attempt to re-create the tests performed in (Hu & de Haan 2006). This would both verify the data in (Hu & de Haan 2006) and prove that the blur function worked in test conditions.

While simple in theory, this proved very difficult to implement in practice. The mathematical justification given in (Hu & de Haan 2006) (and expounded in section 3.2.4) was quite simple and straightforward. However on no account could it be persuaded to work.

The method used was simple, the test image (initially the four square checkerboard of figure 5.6, later a simpler, two square version, figure 5.9) was blurred using `cvSmooth` and known blur radius. the resultant image was then re-blurred twice using two other different blur radii, to provide the two comparative images as described in section 3.2.4. This worked, insofar as could be determined,

perfectly. From this point on, however a coherent result was very difficult to obtain.

The test image was of known dimensions. In consequence, it was possible to simply go directly to the region boundary in the code, without having to search for it. As described in section 3.2.4, the point of maximum difference occurs at the region boundary. Once there the arithmetic portions of the code worked correctly, but the answer was never even close to the initial blur (it was frequently several orders of magnitude different). Worse still the answer was insensitive to changes in the initial blur. As can be appreciated, this was unsatisfactory. The change in blur in the real image would denote the approach or retreat of actual objects, thus the recovery algorithm must be able to compute values that reflect this movement.

The problem took considerable time to resolve. While it could be verified that the calculations were functioning correctly and that the initial premise was sound, the results were obstinately wrong. It was finally discovered that the problem lay in the `cvSmooth` function. The function operated by specifying a blur radius and kernel size. If no kernel size was specified one was automatically computed from the desired blur radius. As the method was based around blur radius, the kernel size was not initially considered and the program left to determine this on its own. Unfortunately, this resulted in a very small kernel. It was discovered that the kernel size had critical bearing on the result, after much experimentation it was found that it needed to be of dimensions roughly three times the size of the magnitude of blur. If this was not the case, the blur recovery was not stable or accurate.

When the results of the algorithm are plotted over a series of pixels both before and after the region boundary, the improvements of enlarged kernel size can clearly be seen. Figures 5.10, 5.11 and 5.12 illustrate the results for when the initial blur radius is 1 and the re-blur radii are 4 and 7, with kernel sizes of 13 and 25, respectively. The true edge for the test image (see figure 5.9) is located

at pixel 199. At this point we can see a σ value of 1.4 is recovered. This was considered a sound estimate. However, the true value of 1 is almost exactly recovered, at pixels 197 and 198. These are not the true region boundary, yet they return the best value.

However, notice that the results in figure 5.11 are nowhere near as regular as those illustrated in figure 3.5. The step functions have roughly the same number of points, but the grey values for the re-blurred image in figure 5.11 do not come to rest at the original grey values as in figure 3.5. Additionally, while the values for maximum radius illustrated in figure 5.12 form a similar shape to that in figure 3.6, the values for R_{max} in the former are not constant over more than one pixel, as in the latter case. This indicates that different values of blur will be found on both sides of the region line, as is demonstrated by figure 5.10. This is not a good value because the same blur function is used on both sides of the region divide. All this implies that there are errors in the original code. However, despite many attempts, these could not be uncovered.

Despite the less than satisfactory results from the static tests, it was decided to implement the more general blur finding method. This involved adapting the methods described above to find the blur at the north, south, east and west points in each region (see section 3.3). To do this, a function was written that moved a pointer from the centroid to the requisite region edge (but not counting the image edges). The blur was then calculated at each region edge point and an average obtained. The program was somewhat more complex than this, accounting for cases where the centroid was not in the image region. However, in general this was the approach taken.

Unfortunately, when implemented this was not fully successful. The answers were in general neither accurate nor consistent, yet they did display interesting traits under various conditions.

Initially, the program was tested on test images such as figure 5.9 and figure 5.4.

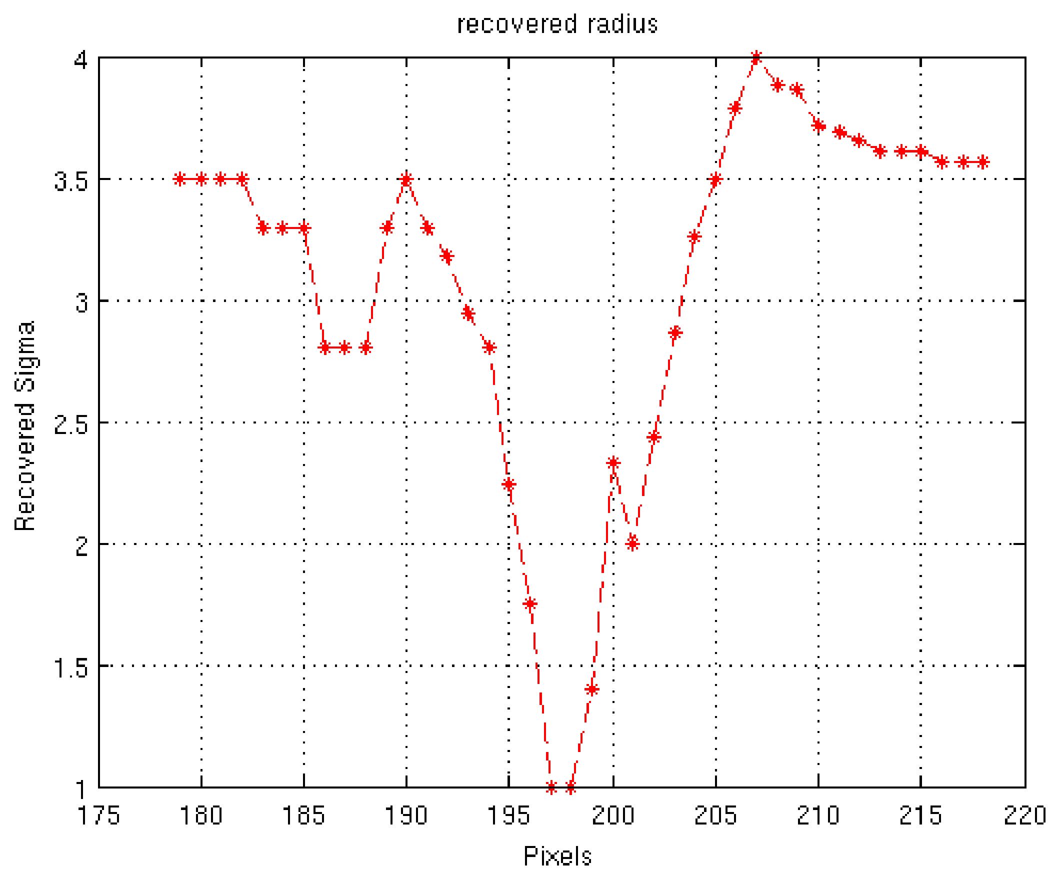

Figure 5.10: The recovered blur radius for the ideal test image with initial radius of 1 and a re-blur σ values of 4 and 7.

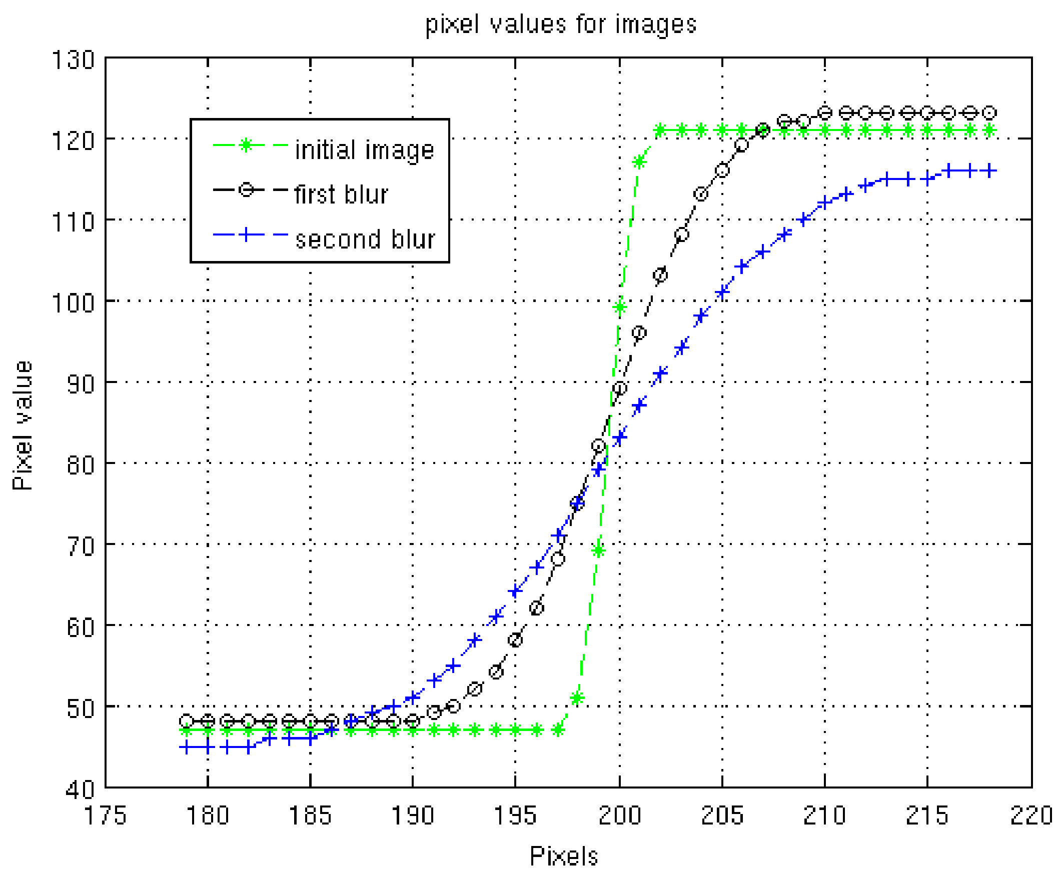

Figure 5.11: The step functions for the blurred edges showing the initial recovered blur and the two re-blurred edges. Re-blur σ values of 4 and 7.

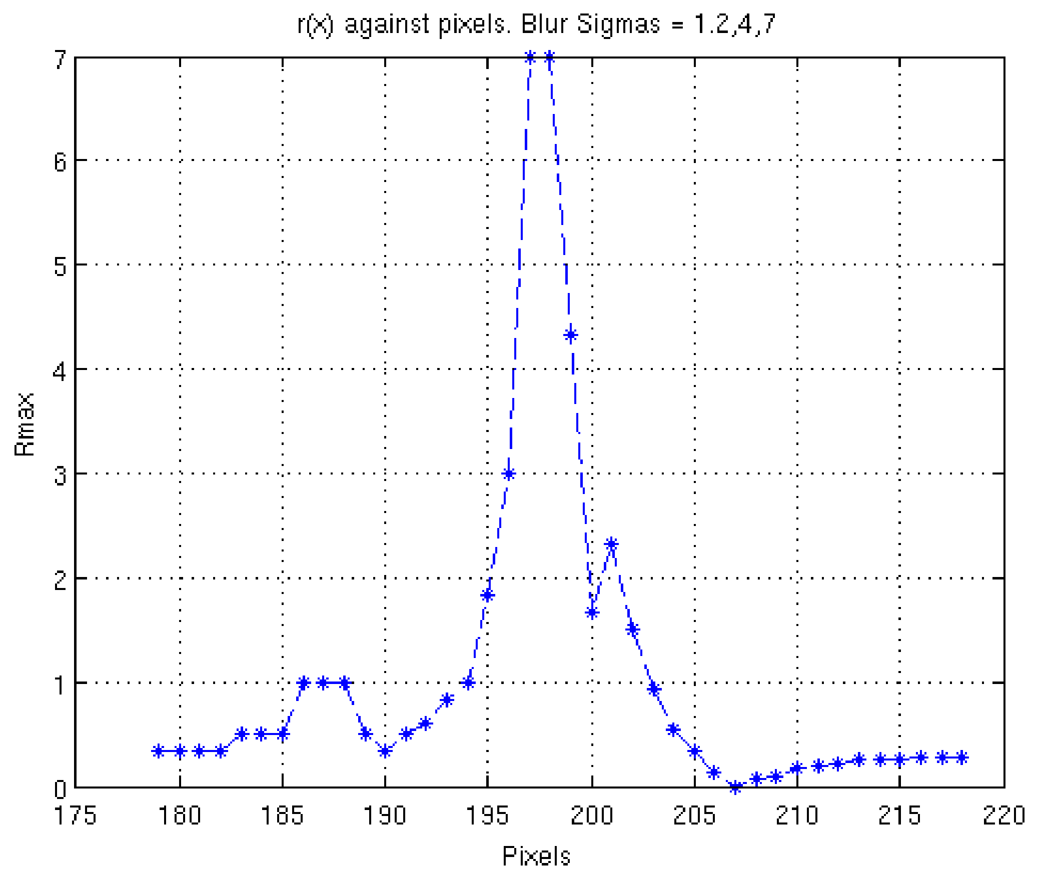

Figure 5.12: Maximum recovered blur difference for the ideal test image with initial radius of 1 and a re-blur σ values of 4 and 7.

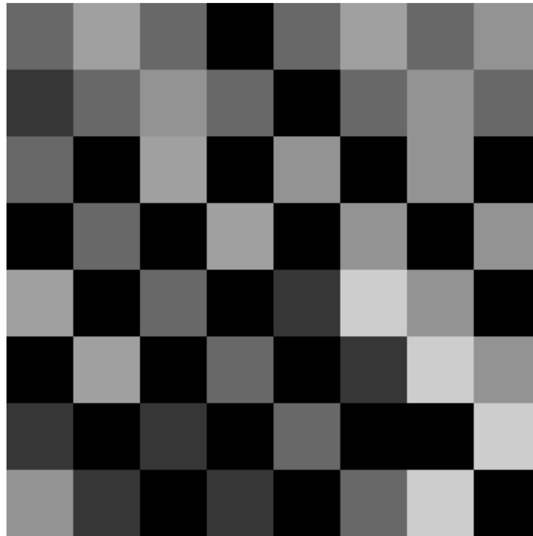

Figure 5.13: Multi-coloured test image for general blur recovery tests.

These verified that the program was functioning correctly, going to the region edges and correctly identifying the difference between a region and image border. This done, the test images were subjected to a known blur and the full algorithm run for recovery. The results on from the black and white test images were not especially accurate (although roughly in range, values of approximately 2.1 were returned for a blur of 1.5), but were consistent *within colours*. That is, all black squares returned the same value at all points and all white squares returned the same value. These values were never the same, yet it did illustrate internal consistency.

However, real images do not exhibit this clear cut contrast between regions. To simulate a more realistic image, while maintaining clear regions, a larger checker-board image was prepared and the regions coloured distinct but similar colours. This image is illustrated in figure 5.13.

When run, this image produced very interesting results. The recovered blur for the regions was much less consistent, even internally. The north, south, east, west points in each region no longer guaranteed the same result. In general each region returned the same value for all four points, but several cases could be seen that

did not. Additionally, the variance of returned values became much larger. To gain some insight into the overall result an “average of averages” was computed, being the average of all the region averages.

This value has no real application in a real world image, which would have a different value of blur for every region in any case. However, in a test image, the whole image was blurred by the same value. Thus the average of all recovered blurs should be the initial blur value

When found, the “average recovered blur” was much more accurate than any individual blur and that accuracy increased markedly after the initial blur value passed 2.0

The other important point that was noticed was that when run repeatedly, for the same image and blur, the program always returned the same answer. This was encouraging as it indicated that the errors were not a product of random behaviour in the program itself.

5.6 Looming Computation

In order to verify that the Looming method was functioning as advertised, several test routines were created. As this is a fairly key area for success in the project, several methods were tried under controlled conditions before the algorithm was incorporated into the real time program.

5.6.1 Area Looming

While area was considered an unattractive method to use in real time (see section 3.2.1), in constrained environments it possesses the unique advantage of being easily verifiable. The calculations could be easily checked if sufficiently simple

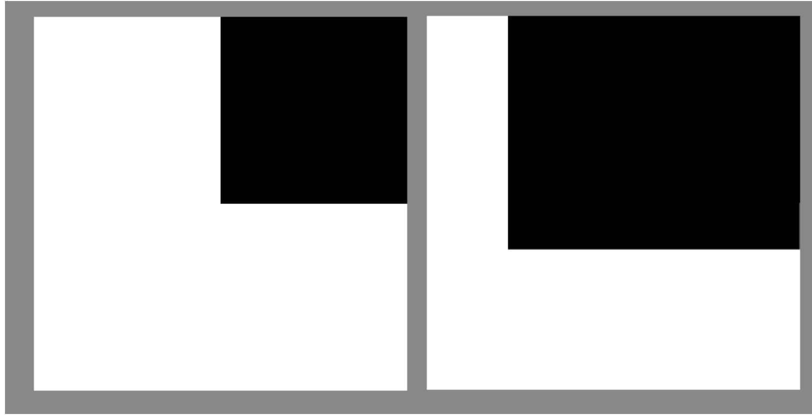

Figure 5.14: The area Looming test images. Not the black area in the left image is 64% smaller than the right.

test images were given. To this end, the test image illustrated in figure 5.14 was created. The black square was 200 by 200 pixels in a 400 by 400 image. This image was then modified by increasing the black square to 250 by 250 pixels, thus simulating a "approaching object". The known increase in size was to enable Looming to be computed by hand before hand. This was done and a value of $(22500/\Delta t)/62500$ found. A test algorithm was then created that used the Looming equations defined in section 3.2.

The hand result was compared to the calculated value of from the test algorithm and found to be identical. Now this seems fairly trivial. The Looming algorithm is in itself, fairly simple, just a simple equation barely worthy of the name algorithm. However to compute Looming, many stages of the program have to work in unison. The centroid functions must accurately compute the x, y positions, the mean grey-scale value and the area of the regions, the tracking function must assign a correct correlation between the moving regions and then assign the values of the previous region area to the current regions. An accurate value of Looming confirmed that these operations worked, at least under test conditions[2].

However, computing the Looming value for idealised test images is one thing,

[2]Interested readers may view the test algorithm in appendix B.2.3

estimating it for a sequence of real time moving images, is something else entirely. When the area based Looming was implemented in real time, it was found that there was substantial instability in the result.

This instability was, to some extent expected. Despite the efforts expended on the segmentation sections, there was still some degree of divergence from frame to frame (this is discussed in detail above). This divergence meant that the value of area for a given region fluctuates mightily from frame to frame. Although it does not look so serious to the naked eye, this fluctuation can create seriously erroneous values for Looming. This meant that the "closest" or most "distant" object as computed by the Looming function were rarely the same region from one frame to the next.

Although exact tests were difficult on the live feed, the main reasons for this erratic behaviour were thought to be due to region fluctuation and tracking errors. When tracking, the function works perfectly for test images. As mentioned in section above, perfectly segmented images that are clearly delineated can be tracked with ease. It is somewhat less successful in finding matches in a stream of random shapes. The slight (or sometimes not so slight) variations in the segmented image from frame to frame can be highly confusing. The result is that the less stable regions seldom have correct matches from one frame to the next. Thus, in addition to fluctuations in area, the region assigned "closest" or "furthest" in one frame may not have any satisfactory match in the next, or (worse) an incorrect match. This made the end result for the Looming value very erratic.

5.6.2 Blur Looming

When Looming through blur was implemented the results were initially as discouraging as previously. The "closest object" as decided by the blur-Looming algorithms flickered and shifted from frame to frame. The same lack of stability

that had plagued the previous method was evident in blur Looming.

However to recover blur necessitated a different camera (the original camera being unable to de-focus its image). This change highlighted some interesting points, especially once the image focus had be decreased. As mentioned previously, the original program had a section dedicated to "smoothing" the camera image with a Gaussian blur. As the new camera[3] had the ability to optically blur the image, this was unnecessary and removed. The results were compatible (70% successful tracking and higher when stationary and about 40 to 50% when moving smoothly), but the processing time was much faster as a result of the removal of the prepossessing stage.

However, the blur recovery was too unstable to be able to provide coherent values of L. Simply running the program and asking for the max value of L illustrated that there was little or no consistency, the same region rarely produced the same value of Looming twice running. This meant that the stability of the "closest region" was not noticeable improved over the area Looming calculations, regardless of the success of tracking.

5.7 Discussion

5.7.1 Segmentation

The results in this area indicate that nascent problems still exist with regards to stability. This is evidenced by the discrepancies in the segmented frames. However it is interesting to observe that these discrepancies are noticeably less marked in background the objects. This indicates that background objects (which

[3]This new camera was an IDS GmbH "ueye". Drivers and other technical information can be obtained from `www.ids-imaging.com`. The camera was equipped with a fish-eye lens, the image being cropped for quicker run time

are less clearly focused) suffer much less than foreground ones from whatever the destabilising effects are.

As the foreground objects are relatively small (the closer the object is, the more prone it is to break into separate regions) they could be more susceptible to subtle changes in lighting, noise and other image factors. Increasing the robustness of the tracking algorithm's ability to decide on split regions could alleviate this.

Another potential cause for the region instability could be the nature of the single pass split merge algorithm. This algorithm depends heavily on the overlap and precedence of previous splitting runs. Thus a potential source of instability is the differences in starting position. The differences in starting position could produce an altered evolution in the segmentation process, resulting in different regions. This could be one possible solution worth investigating.

The somewhat discouraging results for segmentation inevitably raise the question, should it be replaced by another approach? As mentioned in section 2.2.2, there exists other approaches, such as edge tracing. It would probably be worth implementing such a device for the sake of comparison.

5.7.2 Tracking

It is difficult to know if there are any serious problems with the tracking algorithm, or those that exist are the result of instability in the segmentation algorithm. The idealised test results are promising, the ideal regions can be tracked successfully. However when the real time algorithm is run there is some indication from visual observation that the tracking is less than perfect. This is supported by the return of about 50% of all regions tracked in the complex test images.

However it should be noted that the value of 50% success is only for two frames. The next pair of frames may also return a value of 50% but it is not necessarily

the *same* 50%. It is virtually certain that the percentage of regions tracked for more than two frames is actually much less than this.

It is possible that the means used to detect the regions in the previous frames are not rigorous enough. As mentioned in section 4.4.3 the first thing defined is a search area. Certain sources (Hager & Belhumeur 1998), employ a bounding box instead. Such an approach could improve the tracking algorithm by making the search area more responsive to region geometry. However, the response of 100% success for motionless images is encouraging that the problem is not that of the tracking algorithm.

The results obtained from a out of focus camera support this conclusion. There is a noticeable improvement in tracking success when the image exhibits a degree of de-focus. Thus there is an indication that the tracking algorithm is working and that further stabilisation would improve the result. In this case the stabilisation was provided by the blurry image, resulting in less shades, fewer and more consistent regions and better results generally. This implies that some form of prepossessing, either by optical blur or a more effective Gaussian blur in software is advisable.

The final obvious point with the tracking algorithm is its ability to distinguish regions that have split or merged from one frame to the next (this would not be tested by the motionless images that gave such good results). Currently it employs a crude method for noting such events. However if additional time were spent perfecting this, to account for multiple regions becoming one, one region splitting into several and violent changes in geometry, a more robust result could potentially be obtained. The simplest enhancement could be along the lines of seeing if the location of a centroid in the previous frame is the same region label (or a similar label) to the current region.

5.7.3 Looming

Blur

While the key source for blur recovery presented a simple idea, it was found difficult to implement in practice. As can be seen in the relevant graphs, the recovered values were not as smooth in practice as indicated by the results in that paper.

The most frustrating part of the problem was that the method described in (Hu & de Haan 2006) was easy to understand. There appeared to be nothing wrong with the mathematical theory. Using this as a base assumption, the test algorithm was inspected again. The trouble was found to stem from the `cvSmooth` function. When formulating the method (Hu & de Haan 2006) used an *analytical* Gaussian function to approximate blur. This is a true Gaussian, in that it actually goes to infinity in both directions. Of course, the magnitude as one departs from the mean shrinks asymptotically towards zero, yet it still exists. When this function is implemented *numerically* it is done via a convolution matrix of finite size. No matter the dimensions of the kernel used, at some point a truncation error occurs. It was believed to be this truncation error at the root of many difficulties.

Notice that the resulting step functions in figure 5.11 are not the exact mathematical step and convolved functions as are those illustrated in figure 3.5. Again, this is believed to be due to the numerical rendering of the functions depicted in figure 3.5.

Also notice (as mentioned above) that the recovered vale for R_{max} is not symmetrical around the region boundaries as illustrated by figure 3.6. Instead it comes to a point and does not give the correct result at this point, rather at the region boundary (the 199th pixel). This indicates that there is an error in the recovery of the blur. It also indicates that this error is subtle enough to permit the function to work partially.

These problems have implications for when the blur recovery algorithm is implemented on real regions. The results were neither consistent nor especially accurate. This is a serious problem for the Looming calculations. As the blur values are not as expected, it is unclear where randomness in entering the algorithms. As it cannot be assumed that the inaccuracies in the blur recovery are proportional to the real values there is no guarantee that they will not shift wildly with no relation to actuality as the region evolves.

The lack of consistency and accuracy in the general blur recovery was believed due to the effects illustrated in figure 5.10. This illustrates a fairly accurate result, but not necessarily exactly at the region boundary. Thus when run in general terms on an abstract region the resultant blur recovered, may not be the exact value used to blur the original image.

The results in figure 5.10 would seem to indicate that a more accurate value could be found if the program was to compute the blur values at several points before the region boundary was reached. Unfortunately, while this may be possible in a test image where the original blur is known, this is not possible in a image of unknown blur values as the program would have no way of knowing which of the blur values was the most accurate.

The "average of averages" value computed would seem to give a more accurate value. However, this is not a very good estimate of blur. To begin with it ignores the large variance between individual results. To be fully comprehensive a standard deviation or a variance value should be computed for comparison. Also, it is impractical in a real image to find the "average blur." Recall that the purpose of blur recovery is to find *different* blur values at various points and hence establish a Looming value. The average of averages value was simply an attempt to illustrate that the inaccuracies were such that a ball park figure could be arrived at if more points in the blurred region were considered. This it did, especially for larger values of blur. This implies that more than four points should be included in the calculation of blur for any one region.

These difficulties have, of course spilled over into the Looming through blur section. The erratic results for blur Looming imply that the errors noted above have great implications for the stability of the Looming functions. This is clearly another key area for further work. The stabilisation of the blur recovery is clearly necessary before this system could be implemented.

Area

It was not really a surprise that area Looming was not overly successful. The segmentation algorithm does not operate in a way that is conducive to reliable area Looming, even when it is successful. When the segmentation algorithm operates successfully, an object should break into smaller and smaller regions as it approaches the camera. This is because, as the object approaches, it is becoming more complex. Textures and shades that were previously indistinct are now fully apparent. This means that what was previously one large region has the potential to become several smaller ones as the object approaches. Such alterations should be able to be handled by the tracking function with its ability to assign two new regions to the same common ancestor. However, as can be seen, this did not work all the time.

The upshot is that as regions approach closer and closer to the camera, they fragment more and more, thus creating area *losses* just when one would expect area *gains*. Thus just as the object is approaching the position where it is the most danger to the moving machine, the ability to judge that hazard is decreasing. Recall that the tracking and segmentation works fairly well on objects that are distant. However distant objects are not an issue, it is close ones that are would be problem.

Chapter 6

Conclusions

From the observations above it can be seen that the approach taken to detect obstacles has some potential. Although they could not be made to successfully operate at the present time, the results indicate that the systems could be made to operate if the various problems could be resolved. Although the same could be said of any proposal, the author is confident that the various problems are in fact, easily solvable.

6.1 Segmentation

The region based approach is unstable and prone to frame to frame discontinuities. Efforts should be made to make the algorithm more robust and investigate plausible alternatives.

However, the region based approach is capable of rendering passable segmentation of still images. It can be illustrated that the current algorithm does not introduce any randomness into the segmented image.

From this one can conclude that the seeming instability of the segmentation approach is probably due to the delicacy of the method itself or the difference in the image from frame to frame. This indicates that the method has potential.

6.2 Tracking

While partially successful, this section could be improved. It is difficult to distinguish between inherent errors and problems due to the segmentation process, yet it is felt that the current tracking framework is solid enough to permit future experimentation along similar lines.

6.3 Looming

It is difficult to determine whether Looming has succeeded as a method, the problems with other sections render the results too fragmentary. However, if those difficulties could be resolved it is thought that Looming could become a viable method.

Both looming through blur and looming through area exhibit severe instabilities. However, it is believed that the problems with area are inherent, while the problems with the blur method could be resolved if a stable functioning output could be obtained for blur radius. Additional work should be invested in improving the blur recovery method.

6.4 Overview

In general, specific tests have indicated that the methods outlined are viable. Despite the real time results being too fragmentary to state categorically that the methods were successful, the test images strongly indicate that the concepts outlined are valid.

6.5 Completion of Objectives

In general the objectives outlined at the commencement of the project were met. The noticeable omissions involved the implementation and hardware integration. This was not considered a serious restriction as the information about the efficacy of the algorithms was garnered from other tests, which showed that the hardware implementation would have revealed little more than was already known.

What was noticeable when reading the objectives upon completion was the un-

foreseen problems that occurred. Nowhere in the objectives is there a mention of tracking or blur recovery, yet these problems consumed many pages of explanation and research and many weeks of work before solutions were found. When the original objectives were reviewed at the end it became clear that they were inadequate and did not fully describe all the steps of the problem.

This is to be expected. With the information to hand at the beginning of the project the objectives were not going to be comprehensive, however despite this the following points have been satisfied,

1. Review the literature to establish the best methodologies.
2. To design and implement a program capable of separating one object in a digital image from another.
3. Once segmentation is achieved, create some means of estimating the distance of obstacles from the camera, using the segmented image. Despite instability, a possible solution has been clearly identified.
4. The above points shall be implemented in real time, thus permitting truly autonomous mobility.
5. The above points shall be implemented using only one camera.

Chapter 7

Future Work

7.1 Obstacle Detection

The first and arguably most critical section is the identification and recognition of potential obstacles. as mentioned above, the segmentation method is clearly functional, yet has some nagging inadequacies. Specifically the differences between one frame and another when the image is moved slightly. The entire segmentation method appears unduly sensitive, which in turn makes it difficult to track. It is possible that this could be improved by a more responsive tolerance, better image pre-processing, using previous frames to help segmentation or perhaps employing a different method altogether.

The tolerance is perhaps the first item that should be addressed. As discussed in section 2.4.2, the tolerance is computed off the standard deviation. However it is scaled by an arbitrary number to ensure that the result is of the correct order of magnitude. Perhaps this is too crude an identifier for what is, after all a very crucial measurement. While the tolerance is tied to the original image it still contains arbitrary components. If these could be removed, then possibly the image output will improve.

Another possibility could be the improvement of the pre-processing. As mentioned in section 5.2.1, the image is smoothed before being given to the segmentation algorithm. This is done to remove the destabilising effects of noise from the image. However, experimentation has shown that the magnitude of the image smoothing strongly effects the stability of the image output. However, this smoothing algorithm is the slowest single element of the program and increasing the smoothing greatly decreases the speed of the algorithm [1]. Investigating different ways of removing image noise from the camera feed whilst still retaining consistent object edges and interiors would also be a good means of improving

[1] The OpenCV function `cvSmooth`, which is responsible for the smoothing, has caused serious trouble here and with blur recovery. A good direction for future work would be to modify or at least investigate closely the workings of this function and to adapt them better to the problem.

the region stability.

A method more likely to produce advantageous results would involve incorporating the previous image data into the current frame segmentation process. As has been demonstrated, the problem is not with the segmentation process itself, but rather with the consistency from frame to frame. However the previous segmented frame contains all the information needed to describe the segmented image. The only difference will be at the edges of regions as the camera moves.

What this means is that the information required to partially segment any given frame already exists, in the previous output image. If the segmentation process could be limited to the edges of the image regions the center of the regions could be made far more stable. Perhaps the tolerance could be weighted so that groups are more likely to be merged if those pixels were joined in previous frames. This would necessitate a great increase in complexity for the programs with the current process being the mere beginning of the process.

However, it may be advantageous to reject regions altogether and instead implement edge finding algorithms as mentioned in section 2.2.2. It is not known to the author at this time if such an approach would guarantee a more stable output, however an edge tracing algorithm could produce output that could be easily inserted in place of the segmentation algorithm. The edge segments could be tracked as the regions have been, by finding their centroid. Looming could also proceed as normal. These advantages may outweigh the difficulties discussed in section 2.2.2, but the advantages of edge tracing as opposed to regions should be easy to investigate in the current program structure.

7.2 Tracking and Correspondence

By and large the tracking section functioned reasonably well. Although it is true that a large number of regions in any image went "un-tracked", this was

considered due to the instability in the region segmentation process, rather than inherent flaws in the tracking algorithm. It is possible, however that tracking could be improved by restructuring the tolerance, enhancing the search area and considering the problems of occlusion and perspective in greater detail.

The tracking algorithm as covered in section 4.4 is relatively simple. The more detailed tracking methods described in (Fuh & Maragos 1989), etc employ bounding boxes in the tracking calculations. The addition of a bounding box would enable the tracking algorithm to distinguish more accurately between regions of different distribution but similar size. However, this would actually render the tracking algorithm less robust unless the frame to frame segmentation process could be stabilised.

If the segmentation process can not be stabilised, another possible solution would be to adapt the tracking algorithm to compensate for the frame to frame variations in regions. This would be the more difficult option (it would be far more satisfactory to improve the segmentation algorithm), as it would involve compensating for wide variations in region area.

Another focus in tracking could be to expand the algorithm to include data from more than two frames. The introduction of an analysis of the motion of various regions over time could go along way to improving the incidence of successful tracking. The present program only incorporates information from the previous and current frame. With a more detailed picture of the region movement it would be possible to assess the next frame based on probabilities, rather than only region similarity.

7.3 Looming, Approach and Avoidance

The greatest weakness in the Looming approach was the difficulty in extracting reliable results for blur. Clearly the first focus for future work should be improving

the blur recovery function until reliable results are constantly obtainable. At this stage it is not known were exactly the problem lies, however, it is expected that consistent results are possible.

In this process it may be advantageous to create or modify the blurring function `cvSmooth` provided as part of the open CV library. It is possible, though it must be acknowledged not very probable, that this is behaving unexpectedly and interfering with the result.

Once stable values for blur have been obtained, more rigorous tests would need to be implemented on the looming algorithm. It is at this point that hardware would become important. The program would need to be implemented on a wheeled platform and test footage of a simple landscape shot. The footage would then need to be analysed to see if the looming values were working, ie labeling the correct objects as “farthest” and “closest”. Following this more rigorous avoidance algorithms would need to be implemented.

Once looming could be verified as functioning correctly, the hardware implementation could be extended to full avoidance. This would imply more detailed investigation of avoidance algorithms and their implementation in the program structure. Their success could be simply gauged by the success of the hardware platform in avoiding obstacles.

Originally the intention was to simply steer the robot based on the proximity of the obstacles in view to the center of the image and distance from the camera. The machine would be instructed to seer away from the closest object and towards the furthest object at a speed proportional to that objects distance from the center and distance. This was a simplistic approach and never implemented (due to the instability in the output).

7.4 Navigation

Once the avoidance program is modified to the point of stable operation, it would probably become desirable to utilise the system to implement a Navigation algorithm.

Although the implementation of such an algorithm is described in the objectives (see Appendix A.1), time was too short to fully investigate the implications of navigation. Also, the finished avoidance algorithm was found to be too unstable to make such an investigation worthwhile. However, it could be possible once the current problems have been fixed to add a final function to the tail of the current structure that focuses on Navigation.

This would need to employ considerable information from previous frames. The evolution of Looming values would have to be analysed in considerably more detail than currently. Additionally, the evolution of region movement would also have to be considered in more detail. These requirements imply considerable more robustness in the tracking algorithm than currently exists.

This detailed tracking and recording could be implemented by expanding the structure described in `Centroid.h` (see appendix B.1.3). Tables could be incorporated into the centroid structure detailing the evolution of the region over time, with respect to both spatial location and looming value. These modifications could me made simply within the existing structure.

References

Billingsley, J. (26th May 2011), 'private correspondence'.

Bradski, G. & Kaehler, A. (2008), *Learning OpenCV, Computer Vision with the OpenCV Library*, O'Reily.

Brookshaw, L. (10/12/2010 to 27/10/2011), 'private correspondence'.

Davies, E. R. (1997), *Machine Vision*, Academic Press.

De Veaux, R., Velleman, P. F. & E., B. D. (2004), *Intro Stats*, Pearson Education.

Dep (1998), *Homogenious Region Merging Approach for Image Segmentation Preserving Semantic Object Contours*.

Fuh, C.-S. & Maragos, P. (1989), Region-based optical flow estimation, *in* 'Computer Vision and Pattern Recognition, 1989. Proceedings CVPR '89., IEEE Computer Society Conference on', pp. 130 –135.

Fuh, C.-S., Maragos, P. & Vincent, L. (1993), Visual motion correspondence by region based approaches, *in* 'Asian Conference on Computer Vision, November 1993, Osaka, Japan'.

Hager, G. D. & Belhumeur, P. N. (1998), Efficient region tracking with parametric models of geometry and illumination, *in* 'IEEE Transactions on Pattern Analysis and Machine Intelligence', Vol. 10.

House, R. (1994), *Beginning With C, and introduction to professional programing*, International Thompson Publishing.

Hu, H. & de Haan, G. (2006), Low cost robust blur estimator, *in* 'Image Processing, 2006 IEEE International Conference on', pp. 617 –620.

Jain, R., Kasturi, R. & Schunck, B. G. (1995), *Machine Vision*, McGraw-Hill.

Javed, O. & Shah, M. (2006), Tracking and object classification for automated surveillance, *in* A. Heyden, G. Sparr, M. Nielsen & P. Johansen, eds, 'Computer Vision ECCV 2002', Vol. 2353 of *Lecture Notes in Computer Science*, Springer Berlin / Heidelberg, pp. 439–443.
URL: *http://dx.doi.org/10.1007/3-540-47979-1_23*

Low, T. (15/12/2010 to October 27/10/2011), 'private correspondence'. Project Supervisor.

Luxen, M. & Förstner, W. (2002), Characterizing image quality: Blind estimation of the point spread function from a single image, *in* 'PCV02', p. A: 205.

Moore, D. S. (1995), *The Basic Practice of Statistics*, W.H. Freeman and Company.

Neumann, U.; You, S. (1998), Integration of region tracking and optical flow for image motion estimation, *in* 'Image Processing, 1998. ICIP 98. Proceedings. 1998 International Conference on', Vol. 3, pp. 658 – 662.

Pietikäinen, M. K., ed. (2000), *Texture Analysis in Machine Vision*, Vol. 40 of *Series in Machine Perception Artificial Intelligence*, World Scientific, Singapore.

Raviv, D. (1995), 'A quantitative approach to looming', *International Symposium on Computer Vision* .

Raviv, D. & Joarder, K. (2000), 'The visual looming navigation cue: A unified approach', *Computer Vision and Image Understanding* **79**.

Sahin, E. & Gaudiano, P. (1998), Mobile robot range sensing through visual looming, *in* 'Intelligent Control (ISIC), 1998. Held jointly with IEEE In-

ternational Symposium on Computational Intelligence in Robotics and Automation (CIRA), Intelligent Systems and Semiotics (ISAS), Proceedings', pp. 370 –375.

Sonka, M., Hlavac, V. & Boyle, R. (1994), *Image Processing, Analysis and Machine Vision*, Chapman and Hall.

Subbarao, M. (1987), Direct recovery of depth-map i: Differential methods, *in* 'Proceedings of the IEEE Computer Society workshop on Computer Vision', pp. 58–65.

Wang, H., Xu, J., Guzman, J. I., Jarvis, R. A., Goh, T. & Chan, C. W. (2001), Real time obstacle detection for agv navigation using multi-baseline stereo, *in* 'Experimental Robotics VII', Vol. 271 of *Lecture Notes in Control and Information Sciences*, Springer Berlin / Heidelberg, pp. 561–568.
URL: *http://dx.doi.org/*

Yang, H. S. & Lee, S. U. (1997), 'Split-and-merge segmentation employing thresholding technique', *International Conference on Image Processing* .

Appendix A

Original Specifications

A.1 Research Specification

Iain J. Brookshaw

Topic Obstacle Detection using Vision for Mobile Robots

Title Real Time Implementation of Obstacle Avoidance for an Autonomous Mobile Robot Using Monocular Computer Vision

Supervisor Dr. Tobias Low

Project Aim It is the author's intention to research, design and practically implement a methodology for real time obstacle avoidance. This will be attempted using an vision controlled autonomous platform. It is intended that the obstacles will be static, thus all motion will be provided by the movement of the camera. Time permitting an attempt will be made to expand the program to include navigation.

Programme

1. Review the literature to establish the best methodologies.
2. To design and implement a program capable of separating one object in a digital image from another (image segmentation). This is the first step in identifying potential obstacles.
3. Once segmentation is achieved, create some means of estimating the distance of obstacles from the camera, using the segmented image. This will enable the devise to establish the location of obstacles relative to itself.

4. Upon the successful implementation of the above, steps will be taken to control the robot's motors and enable it to avoid obstacles. Control will be based on information from the image as discussed above. This will necessitate communication between the camera, on-board computer and the motor controls.

5. Time permitting, expand the design to enable navigation between obstacles, as opposed to mere avoidance.

6. The above points shall be implemented in real time, thus permitting truly autonomous mobility.

7. The above points shall be implemented using only one camera.

8. Once the above is completed existing results will be analysed and compared to current practice.

Iain Brookshaw

Dr. Tobias Low (supervisor)

Iain Brookshaw

Appendix B

Program Listings

B.1 Final Programs

All code was compiled using the gcc compiler (version 4.4.5) on a Debian Linux machine (version 6.0.3) against the Open Computer Vision Library (version 2.1).

B.1.1 Main Driver Function

This is the main function for the program. All other functions are ultimately called from here.

Listing B.1: Main Driver Function.

```
// This program is Master Driver Function
// IB.
//
// This version will use segmentation, centroid finding and tracking.
// 26-7-11
//
// modified to iclude area looming 1/9/11. This works but is very
unstable.
// will probably need to include some form of stabilisation
including previous
// data.
//
// modified to include blur looming 13/10/11.
//========================================================================

#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include <cv.h>
#include <highgui.h>
#include "centroid.h"

int seg_function(IplImage *, IplImage *, int, int, CvSize, int T);

Centroid* cent_function(IplImage *SegImage, int minarea, int
MinRegionCount);

void track_general(IplImage* Seg2, Centroid* Seg1Cent, Centroid*
Seg2Cent,int Tol);

void general_blur(IplImage* IN, IplImage* Blur1, IplImage* Blur2,
Centroid* Current, float Sigma1, float Sigma2);

void loomf_blur(IplImage* Segmented, Centroid* CentList, float dt);

int main( ) {
  printf(" \n Starting main driver function\n");
  printf(" Iain Brookshaw\n    USQ \n 13/10/11\n");
```

```
// ============ INITIALISATION & SET UP ==============================
//define image variables and Centroid names
IplImage* IN;
IplImage* frame;
IplImage* frame1;
IplImage* frameOut;
IplImage* Blur1;
IplImage* Blur2;
Centroid* CurrentImg = NULL;
Centroid* PreviousImg = NULL;

//create tolerance variable as used in seg_function and track
function
int Tol = 0;

//get image.
//printf("please select desired camera...\n");
CvCapture* capture = cvCreateCameraCapture(0);

//this should ask for desired camera.

printf("begining...\n");
//CvCapture* capture = cvCaptureFromCAM(0);

//set up image windows.
cvNamedWindow( "Original", CV_WINDOW_AUTOSIZE );
cvNamedWindow( "Segmented", CV_WINDOW_AUTOSIZE );


//begin camera feed loop.
while(1){

  //------------------- GET IMAGE & Preprocessing
  ------------------------------

  IN = cvQueryFrame( capture );

  int xMax = IN -> width;
  int yMax = IN -> height;

  CvSize Size;
  Size.width  = xMax;
  Size.height = yMax;

  frame = cvCreateImage(Size,IPL_DEPTH_8U,1);
  frame1 = cvCreateImage(Size,IPL_DEPTH_8U,1);

  //make image grayscale.
  cvConvertImage( IN,
                  frame,
                  0
                  );


  // smooth to remove blur.
  cvSmooth(
          frame,
          frame1,
          CV_BILATERAL,
          1,//9,
          0,
          3,//15,
          3
          );
```

```
//====================== FUNCTION CALLS
=====================================

// ------------------- SEGMENTATION -----------------------
// Now call the segmentation function seg_function.c
frameOut = cvCreateImage(Size,IPL_DEPTH_8U,1);
int MinRegionCount = seg_function(frame1, frameOut, xMax,
yMax, Size,Tol);

// ------------------- CENTROID CALC' ---------------------
//now calculate the centroids of all regions larger than
400 pixels.
CurrentImg = cent_function(frameOut, 400, MinRegionCount );

// ------------------- TRACKING ---------------------------
//call the tracking function.
track_general(frameOut, PreviousImg, CurrentImg, Tol);


// ------------------- BLUR RECOVERY ----------------------
// re-blur the original input image to Blur1 and Blur2.
// define the re-blur values:
double Sigma1 = 7;
double Sigma2 = 10;

//create the blured comparison images.
Blur1 = cvCreateImage(Size,IPL_DEPTH_8U,1);
Blur2 = cvCreateImage(Size,IPL_DEPTH_8U,1);

cvSmooth(frame,
         Blur1,
         CV_GAUSSIAN,
         73,
         73,
         Sigma1,
         Sigma1
         );

//and again,

cvSmooth(frame,
         Blur2,
         CV_GAUSSIAN,
         101,
         101,
         Sigma2,
         Sigma2
         );

//now call the blur recovery function.
Centroid* Current = CurrentImg;

general_blur(frameOut, Blur1, Blur2, Current, Sigma1,
Sigma2);

// ------------------- LOOMING ----------------------------
//find looming through blur
float dt = 0.05;
```

```
    loomf_blur(frameOut, CurrentImg, dt);

    //show the region with the greatest looming value:
    Centroid* temp = CurrentImg;
    float maxL = 0;
    int maxX = -1;
    int maxY = -1;

    while(temp){
      if(temp->L > maxL){
        maxL = temp->L;
        maxX = (int)(temp->x / temp->area);
        maxY = (int)(temp->y / temp->area);
      }
      temp = temp->next;
    }
    CvPoint MAX_Loom;
    MAX_Loom.x = maxX;
    MAX_Loom.y = maxY;

     cvCircle(frameOut,            //image name
              MAX_Loom,                 //center,
              10,                  //radius,
              CV_RGB(0, 0, 100), //CvScalar color,
              8,                   //int thickness=1,
              8,                   //int lineType=8,
              0 );                 //int shift=0 );


    //now make the current centroid list the previous centroid list.
    PreviousImg = CurrentImg;

    // ================= DISPLAY AND CLEAN UP ==============

    //display image.
    cvShowImage( "Original", frame1);
    cvShowImage( "Segmented", frameOut );

    char c = cvWaitKey(33);// see chapter four
    //o'rilly book to see how to fix this frame rate.
    if( c == 27 ) break;

    if( frame != NULL )      cvReleaseImage( &frame );
    if( frame1 != NULL )     cvReleaseImage( &frame1 );
    if( frameOut != NULL ) cvReleaseImage( &frameOut );
    if( Blur1 != NULL ) cvReleaseImage( &Blur1 );
    if( Blur2 != NULL ) cvReleaseImage( &Blur2 );

  }//END OF IMAGE WHILE(1) LOOP.

  cvReleaseCapture( &capture );
  cvDestroyWindow( "Original" );
  cvDestroyWindow( "Segmented");

  //Remove the centroid list.
  deleteCentroids(CurrentImg);
  CurrentImg = NULL;
  deleteCentroids(PreviousImg);
  PreviousImg = NULL;
```

```
}//================== END OF PROGRAM ===========================
```

B.1.2 Segmentation Functions

These functions are called by the main function to segment the original input image. This set includes the merging and splitting sections and the mechanism for moving through the image.

Listing B.2: Segmentation functions.

```
// This function is designed to segment the camera's image into
distinct regions
// Inputs ---- IplImage* frame, the input image (grayscale single
channel).
//            IplImage* frameOut, the empty image (grayscale
single channel)
//                                   the segmented image will be
written to.
//            int xMax, the x size of the image.
//            int yMax, the y size of the image.
//            CvSize Size, the image size (duplication of above).
// There are no outputs, the output image frameOut is simply written
to as the
// function progresses.
//
// 27-3-11
// 29-6-11
// IB
// This function uses the function SplitFunc
// This function contains the splitting algorithm.
// Inputs ---- unsigned char* IN, a vector containing the four
values pixels
//                                  to be split into groups.
//            unsigned char* OUT, a vector (empty) that will
contain the
//                                   output group labels in the same
positions
//                                   of the input groups.
//            float T, the tolerance for segmentation
// There are no outputs
//====================================================================

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

#include <cv.h>
#include <highgui.h>
```

```
void SplitFunc(unsigned char* IN, unsigned char* OUT, float T);

int seg_function(IplImage *frame, IplImage *frameOut, int xMax,
int yMax, CvSize Size)
  {

    // ================== SEGMENT THE IMAGE -===================

    // Now set the counters RegionCount & RegionSum. These will be
    vectors of
    // length 256 (entries will be 0 ----> 255).

    int RegionCount[256];
    int RegionSum[256];

    // now fill these with 0's
    int i;

    for(i = 0; i < 256; i++){
      RegionCount[i] = 0;
      RegionSum[i] = 0;
    };

    // Initalise the x and y counters.

    int x;
    int y;

    // At this point we need to set the tolerance.

    // find the start of the image.
    unsigned char* ptr0 = (unsigned char*)(frame -> imageData);
    int tMax = 0;
    int tMin = 1000;


    //finding T using standard devation
    ptr0 = (unsigned char*)(frame -> imageData);

    int Sum = 0;
    int total = yMax*xMax;

    for(y=0; y<yMax; y++){
      for(x=0; x<xMax; x++){
        Sum = Sum + *ptr0;
        ptr0++;
      }
    }

    //reset ptr0
    ptr0 = (unsigned char*)(frame -> imageData);
    float Mean = (float)Sum/(float)total;
    float internal = 0;

    for(y=0; y<yMax; y++){
      for(x=0; x<xMax; x++){
        internal = internal + ((*ptr0 - Mean)*(*ptr0 - Mean));
        ptr0++;
      }
    }

    float s = sqrt (1.0/((float)total-1) * internal);
```

```
    float T = (1.0/s)*1000;
    //float T = 1000.0;

    // This 1000 is arbitrary at this time. The inverse is necessary
    to ensure that tol'␣becomes
␣␣␣␣//␣less␣selective␣as␣s␣decreaces.

␣␣␣␣unsigned␣char␣Split[4];

␣␣␣␣//␣==================␣Begin␣Loops␣===================
␣␣␣␣for(y=0;␣y<yMax-1;␣y++){

␣␣␣␣␣␣//␣get␣the␣pointer␣into␣the␣image␣matrix.␣This␣is␣the
␣␣␣␣␣␣pointer␣to
␣␣␣␣␣␣//␣the␣start␣of␣the␣current␣row!

␣␣␣␣␣␣//␣input␣image
␣␣␣␣␣␣unsigned␣char*␣ptr1=␣(unsigned␣char*)(frame->imageData␣+
␣␣␣␣␣␣y␣*␣frame->widthStep);
␣␣␣␣␣␣//␣output␣image
␣␣␣␣␣␣unsigned␣char*␣ptr2=␣(unsigned␣char*)(frameOut->imageData␣+
␣␣␣␣␣␣y␣*␣frameOut->widthStep);

␣␣␣␣␣␣for(x=0;␣x<xMax-1;␣x++){

␣␣␣␣␣␣␣␣//␣This␣is␣the␣begining␣of␣the␣x␣(col') counter.
        // IF x = y = 0 then we are at the start
        // of the image and a seperate section is needed to handle
        this.

        // If we are not at the start of the image, we can proceed.

        // ------- INITIATE THE PIX' BLOCK (SPLITTING)
        ALGORITHM. --------

        // get the 1*4 vector of the local block, given the current
        location
        // of the x,y counters.

        unsigned char B_in[4];

        B_in[0] = *ptr1;
        B_in[1] = *(ptr1+ 1);
        B_in[2] = *(ptr1+ frame->widthStep);
        B_in[3] = *(ptr1+ frame->widthStep +1);


        // for various reasons explained bellow, we also need
        the output
        // image for this location. This vector shall be called B_seg
        unsigned char B_seg[4];
        B_seg[0] = *ptr2;
        B_seg[1] = *(ptr2+ 1);
        B_seg[2] = *(ptr2+ frameOut->widthStep);
        B_seg[3] = *(ptr2+ frameOut->widthStep +1);

        //split the input vector B_in
        Split[0] = 0;
        Split[1] = 0;
```

```
Split[2] = 0;
Split[3] = 0;
SplitFunc(B_in, Split, T);

//We need values for GroupSum and Count
int Count[4];
Count[0] = 0;
Count[1] = 0;
Count[2] = 0;
Count[3] = 0;

int GroupSum[4];
GroupSum[0] = 0;
GroupSum[1] = 0;
GroupSum[2] = 0;
GroupSum[3] = 0;

//now fill these values. Note that the index into GroupSum
and Count are
// the group label values.

int n;
int m;
for(n=0; n<4; n++){
  for(m=0; m<4; m++){
    if(Split[m] == n+1){
      GroupSum[n] = GroupSum[n]+B_in[n];
      Count[n] = Count[n]++;
    }
  }
}
//printf("Point1\n");

unsigned char Iout[4];
Iout[0] = 0;
Iout[1] = 0;
Iout[2] = 0;
Iout[3] = 0;

// ==================================================================
// |              ASSIGN VALUES TO OUTPUT IMAGE                     |
// ==================================================================

// Make Contingency for X AND Y == 0 THIS IS THE FIRST PIXEL

if( x == 0 && y == 0){

  // initalise grayscale counters.
  int G1 = 256;
  int G2 = 256;
  int G3 = 256;
  int G4 = 256;

  // Run through the vectors and assign the
  // correct values to the output as one goes.

  for(i=0; i<4; i++){

    //first group
    if(Split[i] == 1){
```

```
        if(G1 == 256){
          G1 = B_in[i];
          RegionCount[G1]++;
          RegionSum[G1] = RegionSum[G1] + B_in[i];
        }
        Iout[i] = G1;
      }

      //second group
      if(Split[i] == 2){
        if(G2 == 256){
          G2 = B_in[i];
          RegionCount[G2]++;
          RegionSum[G2] = RegionSum[G2] + B_in[i];
        }
        Iout[i] = G2;
      }
      //third group
      if(Split[i] == 3){
        if(G3 == 256){
          G3 = B_in[i];
          RegionCount[G3]++;
          RegionSum[G3] = RegionSum[G3] + B_in[i];
        }
        Iout[i] = G3;
      }

      //fourth group
      if(Split[i] == 4){
        if(G4 == 256){
          G4 = B_in[i];
          RegionCount[G4]++;
          RegionSum[G4] = RegionSum[G4] + B_in[i];
        }
        Iout[i] = G4;
      }
    }

    // Now write Iout to output image. To do this use
    // pointer arithmetic developed above.
    // to do this use *ptr2 = Iout[X] REMEMBER: *ptr =
   X means put X in
    // LOCATION signified by the address ptr.
    //ptr2 is location in OUTPUT IMAGE frameOUT. Get this
    first!

    *ptr2 = Iout[0];
    *(ptr2 +1) = Iout[1];
    *(ptr2+ frameOut->widthStep) = Iout[2];
    *(ptr2+ frameOut->widthStep +1) = Iout[3];

  }

  // ======== NOW MERGE THE GROUPS ================

  if( x != 0 && y == 0){

    // B and D pixels have not yet been sequenced.
```

```
// The first step in assigning values is to assign
values to
// Iout based on the groups described in Split and the
existing
// regions.

// Once the groups defined by the splitting function
are known,
// one must discover which of the pixels in that group
makes the
// best match with the corresponding region in the
outer image.

// run through all the possible groups (there are a
maximum of 4)
int j = 0;
unsigned char ind[4];
float MeanDiff;
float Md[4];
Md[0] = 1000;
Md[1] = 1000;
Md[2] = 1000;
Md[3] = 1000;

unsigned char index[4];
index[0] = 5;
index[1] = 5;
index[2] = 5;
index[3] = 5;

float LocalMean = 0;
float RegionMean = 0;

//begin group counter (1->4)
for(j=0; j<4; j++){

  //begin pixel counter (1->4)
  for(i=0; i<4; i++){
    if(i == 1 || i == 3) continue;

    if(Split[i] == j+1){
      LocalMean = GroupSum[j]/Count[j];
      if(RegionCount[B_seg[i]] != 0){
        RegionMean =
        RegionSum[B_seg[i]]/RegionCount[B_seg[i]];
      }else{
        RegionMean = 0;
      }
      MeanDiff = fabsf(RegionMean-LocalMean);

      if(MeanDiff < Md[j]){
        Md[j] = MeanDiff;
        index[j] = i;
      }

    } //if(Split[i]==j)
  } //end of i
} //end of j
```

```
//------------------------------------------------------------
int Gray;
for(j=0;j<4;j++){

  Gray = 256;
  for(i=0;i<4;i++){

    if(Split[i] == j+1){

      if(index[j] == 5){ // This group is comprised only
      of pixels
        //not previously assigned to regions.
         if (Gray == 256){
          Gray = B_in[i];
        }
        Iout[i] = Gray;
      }

      else {
        if( Md[j] <= T){
          Iout[i] = B_seg[index[j]];

        } else if( T < Md[j]){
          if (Gray == 256){
            Gray = B_in[i];
          }
        Iout[i] = Gray;
        }
      }
    } //if split
  } //i
} //j
//============================================================

// Now assign this Iout vector to the output image (ptr2).
(*ptr2) = Iout[0];
(*(ptr2+ 1)) = Iout[1];
(*(ptr2+ frameOut->widthStep)) = Iout[2];
(*(ptr2+ frameOut->widthStep +1)) = Iout[3];

//Now assign the new RegionSum & RegionCount vectors.

//Reassign Pixel A.
RegionCount[B_seg[0]] = RegionCount[B_seg[0]] - 1;
RegionSum[B_seg[0]]    = RegionSum[B_seg[0]] - B_in[0];

RegionCount[Iout[0]] = RegionCount[Iout[0]]++;
RegionSum[Iout[0]]    = RegionSum[Iout[0]] + B_in[0];

//Assign Pixel B.
RegionCount[Iout[1]] = RegionCount[Iout[1]]++;
RegionSum[Iout[1]]    = RegionSum[Iout[1]] + B_in[1];

//Reassign Pixel C.
RegionCount[B_seg[2]] = RegionCount[B_seg[2]] - 1;
RegionSum[B_seg[2]]    = RegionSum[B_seg[2]] - B_in[2];

RegionCount[Iout[2]] = RegionCount[Iout[2]]++;
```

```
    RegionSum[Iout[2]]    = RegionSum[Iout[2]] + B_in[2];

    //Assign Pixel D.
    RegionCount[Iout[3]] = RegionCount[Iout[3]]++;
    RegionSum[Iout[3]]    = RegionSum[Iout[3]] + B_in[3];

}
//==========================================================================

if( x!=0 && y !=0){

    // D pixel has not yet been sequenced.

    // Once the groups defined by the splitting function
    are known,
    // one must discover which of the pixels in that group
    makes the
    // best match with the corresponding region in the
    outer image.

    // run through all the possible groups (there are a
    maximum of 4)

    int j = 0;
    unsigned char ind[4];
    float MeanDiff;
    float Md[4];
    Md[0] = 1000;
    Md[1] = 1000;
    Md[2] = 1000;
    Md[3] = 1000;
    unsigned char index[4];
    index[0] = 5;
    index[1] = 5;
    index[2] = 5;
    index[3] = 5;
    float LocalMean = 0;
    float RegionMean = 0;

    //begin group counter (1->4)
    for(j=0; j<4; j++){

        //begin pixel counter (1->4)
        for(i=0; i<4; i++){
            if( i == 3) continue;

            if(Split[i] == j+1){
                LocalMean = GroupSum[j]/Count[j];
                if(RegionCount[B_seg[i]] != 0){
                    RegionMean =
                    RegionSum[B_seg[i]]/RegionCount[B_seg[i]];
                }else{
                    RegionMean = 0;
                }
                MeanDiff = fabsf(RegionMean-LocalMean);

                if(MeanDiff < Md[j]){
                    Md[j] = MeanDiff;
                    index[j] = i;
                }
```

```
    } //if(Split[i]==j)
  } //end of i
} //end of j
//------------------------------------------------------------

int Gray;
for(j=0;j<4;j++){

  Gray = 256;
  for(i=0;i<4;i++){

    if(Split[i] == j+1){

      if(index[j] == 5){ // This group is comprised only
      of pixels
        //not previously assigned to regions.
         if (Gray == 256){
          Gray = B_in[i];
        }
        Iout[i] = Gray;
      }

      else {
        if( Md[j] <= T){
          Iout[i] = B_seg[index[j]];

        } else if( T < Md[j]){
          if (Gray == 256){
            Gray = B_in[i];
          }
        Iout[i] = Gray;
        }
      }
    } //if split
  } //i
} //j
//============================================================

// Now assign this Iout vector to the output image (ptr2).
(*ptr2) = Iout[0];
(*(ptr2+ 1)) = Iout[1];
(*(ptr2+ frame->widthStep)) = Iout[2];
(*(ptr2+ frame->widthStep +1)) = Iout[3];

//Now assign the new RegionSum & RegionCount vectors.

//Reassign Pixel A.
RegionCount[B_seg[0]] = RegionCount[B_seg[0]] - 1;
RegionSum[B_seg[0]]   = RegionSum[B_seg[0]] - B_in[0];

RegionCount[Iout[0]] = RegionCount[Iout[0]]++;
RegionSum[Iout[0]]   = RegionSum[Iout[0]] + B_in[0];

//Reassign Pixel B.
RegionCount[B_seg[1]] = RegionCount[B_seg[1]] - 1;
RegionSum[B_seg[1]]   = RegionSum[B_seg[1]] - B_in[1];

RegionCount[Iout[1]] = RegionCount[Iout[1]]++;
```

```
    RegionSum[Iout[1]]     = RegionSum[Iout[1]] + B_in[1];

    //Reassign Pixel C.
    RegionCount[B_seg[2]] = RegionCount[B_seg[2]] - 1;
    RegionSum[B_seg[2]]   = RegionSum[B_seg[2]] - B_in[2];

    RegionCount[Iout[2]] = RegionCount[Iout[2]]++;
    RegionSum[Iout[2]]   = RegionSum[Iout[2]] + B_in[2];

    //Assign Pixel D.
    RegionCount[Iout[3]] = RegionCount[Iout[3]]++;
    RegionSum[Iout[3]]   = RegionSum[Iout[3]] + B_in[3];

  }
  //==========================================================

  if(x == 0 && y != 0){

    // C and D pixels have not yet been sequenced.
    // Once the groups defined by the splitting function
    are known,
    // one must discover which of the pixels in that group
    makes the
    // best match with the corresponding region in the
    outer image.

    // run through all the possible groups (there are a
    maximum of 4)

    int j = 0;
    unsigned char ind[4];
    float MeanDiff;
    float Md[4];
    Md[0] = 1000;
    Md[1] = 1000;
    Md[2] = 1000;
    Md[3] = 1000;
    unsigned char index[4];
    index[0] = 5;
    index[1] = 5;
    index[2] = 5;
    index[3] = 5;
    float LocalMean = 0;
    float RegionMean =0;
    //begin group counter (1->4)
    for(j=0; j<4; j++){

      //begin pixel counter (1->4)
      for(i=0; i<4; i++){
        if(i == 2 || i == 3) continue;

        if(Split[i] == j+1){
          LocalMean = GroupSum[j]/Count[j];
          if(RegionCount[B_seg[i]] != 0){
            RegionMean =
            RegionSum[B_seg[i]]/RegionCount[B_seg[i]];
          }else{
            RegionMean = 0;
          }
```

```
        MeanDiff = fabsf(RegionMean-LocalMean);

        if(MeanDiff < Md[j]){
          Md[j] = MeanDiff;
          index[j] = i;
        }

      } //if(Split[i]==j)
    } //end of i
  } //end of j
  //------------------------------------------------------------

  int Gray;
  for(j=0;j<4;j++){

    Gray = 256;
    for(i=0;i<4;i++){

      if(Split[i] == j+1){

        if(index[j] == 5){ // This group is comprised only
        of pixels
          //not previously assigned to regions.
           if (Gray == 256){
            Gray = B_in[i];
          }
          Iout[i] = Gray;
        }

        else {
          if( Md[j] <= T){
            Iout[i] = B_seg[index[j]];

          } else if( T < Md[j]){
            if (Gray == 256){
              Gray = B_in[i];
            }
          Iout[i] = Gray;
          }
        }
      } //if split
    } //i
  } //j
  //============================================================

  // Now assign this Iout vector to the output image (ptr2).
  (*ptr2) = Iout[0];
  (*(ptr2+ 1)) = Iout[1];
  (*(ptr2+ frame->widthStep)) = Iout[2];
  (*(ptr2+ frame->widthStep +1)) = Iout[3];

  //Now assign the new RegionSum & RegionCount vectors.

  //Reassign Pixel A.
  RegionCount[B_seg[0]] = RegionCount[B_seg[0]] - 1;
  RegionSum[B_seg[0]]   = RegionSum[B_seg[0]] - B_in[0];

  RegionCount[Iout[0]] = RegionCount[Iout[0]]++;
```

```
            RegionSum[Iout[0]]     = RegionSum[Iout[0]] + B_in[0];

            //Reassign Pixel B.
            RegionCount[B_seg[1]] = RegionCount[B_seg[1]] - 1;
            RegionSum[B_seg[1]]   = RegionSum[B_seg[1]] - B_in[1];

            RegionCount[Iout[1]] = RegionCount[Iout[1]]++;
            RegionSum[Iout[1]]     = RegionSum[Iout[1]] + B_in[1];

            //Assign Pixel C.

            RegionCount[Iout[2]] = RegionCount[Iout[2]]++;
            RegionSum[Iout[2]]     = RegionSum[Iout[2]] + B_in[2];

            //Assign Pixel D.
            RegionCount[Iout[3]] = RegionCount[Iout[3]]++;
            RegionSum[Iout[3]]     = RegionSum[Iout[3]] + B_in[3];
          }
          // ----------------------------------------
          ptr2++;
          ptr1++;

        }//X LOOP

      }// Y LOOP

      //now that we know everything there is to know about regions, we
      //need to estimate the total number of regions. Experience
      shows that
      //a image of one region causes problems later with centroid
      finding
      //and recursion.

      //therefore find the minimum number of regions
      int MinRegionCount = 0;

      for(i=0;i<255;i++){
        if(RegionCount[i] != 0) MinRegionCount++;
      }
      return MinRegionCount;

    }//end of seg_function


//=======================================================================

//split function.

void SplitFunc(unsigned char* IN, unsigned char* OUT, float T){

    int Max = 0;  //This is max difference
    int Min = 1000;  //This is min difference

    //find all 6 differences

    int AB = abs(IN[0] -IN[1]);
    if(AB > Max) Max = AB;
    if(AB < Min) Min = AB;

    int AC = abs(IN[0] -IN[2]);
    if(AC > Max) Max = AC;
```

```
if(AC < Min) Min = AC;

int AD = abs(IN[0] -IN[3]);
if(AD > Max) Max = AD;
if(AD < Min) Min = AD;

int BC = abs(IN[1] -IN[2]);
if(BC > Max) Max = BC;
if(BC < Min) Min = BC;

int BD = abs(IN[1] -IN[3]);
if(BD > Max) Max = BD;
if(BD < Min) Min = BD;

int CD = abs(IN[2] -IN[3]);
if(CD > Max) Max = CD;
if(CD < Min) Min = CD;

// now find what is not possible.
//first check for I and XVI.

// combination I, all in the same group.
if(Max <=T){
  OUT[0] = OUT[1] = OUT[2] = OUT[3] = 1;
  // printf("1\n");
  return;
 }

//combination XVI, all in different groups.
if(Min > T){
  OUT[0] = 1;
  OUT[1] = 2;
  OUT[2] = 3;
  OUT[3] = 4;
  //printf("2\n");
  return;
}

//combination II, two horizontal groups.
if(AB<=T && CD<=T && AD>T && AC>T && BC>T && BD>T){
  OUT[0] = 1;
  OUT[1] = 1;
  OUT[2] = 2;
  OUT[3] = 2;
  return;
}

//combination III, two vertical groups.
if(AC<=T && BD<=T && AB>T && AD>T && BC>T && CD>T){
  OUT[0] = 1;
  OUT[1] = 2;
  OUT[2] = 1;
  OUT[3] = 2;
  return;
}

//combination VIII and IX, diagonals.
if(AD<=T && BC<=T && AC>T && AB>T && CD>T &&BD>T){
```

```
    OUT[0] = 1;
    OUT[1] = 2;
    OUT[2] = 2;
    OUT[3] = 1;
    return;
  }

  //3rd attempt. Successive <=.

  if(CD<=T){// can be IV,VII,XII
    if(AC<=T){//return IV
      OUT[0] = 1;
      OUT[1] = 2;
      OUT[2] = 1;
      OUT[3] = 1;
      return;
    }

    if(BD<=T){//return VII
      OUT[0] = 2;
      OUT[1] = 1;
      OUT[2] = 1;
      OUT[3] = 1;
      return;
    }
    else{//return XII
      OUT[0] = 1;
      OUT[1] = 2;
      OUT[2] = 3;
      OUT[3] = 3;
      return;
    }
  }
  //-------------------------------------------
  if(AC<=T){// can be IV,V,XIII
    if(CD<=T){//return IV
      OUT[0] = 1;
      OUT[1] = 2;
      OUT[2] = 1;
      OUT[3] = 1;
      return;
    }

    if(AB<=T){//return V
      OUT[0] = 1;
      OUT[1] = 1;
      OUT[2] = 1;
      OUT[3] = 2;
      return;
    }
    else{//return XIII
      OUT[0] = 1;
      OUT[1] = 2;
      OUT[2] = 1;
      OUT[3] = 3;
      return;
    }
  }
```

```
//------------------------------------------------
if(BD<=T){// can be VI, VII, XV
  if(AB<=T){//return VI
    OUT[0] = 1;
    OUT[1] = 1;
    OUT[2] = 2;
    OUT[3] = 1;
    return;
  }
  if(CD<=T){//return VII
    OUT[0] = 2;
    OUT[1] = 1;
    OUT[2] = 1;
    OUT[3] = 1;
    return;
  }
  else{//return XV
    OUT[0] = 1;
    OUT[1] = 2;
    OUT[2] = 3;
    OUT[3] = 2;
    return;
  }
}
//------------------------------------------------
if(AB<=T){//can be V, VI, XIV
  if(AC<=T){//return V
    OUT[0] = 1;
    OUT[1] = 1;
    OUT[2] = 1;
    OUT[3] = 2;
    return;
  }
  if(BD<=T){//return VI
    OUT[0] = 1;
    OUT[1] = 1;
    OUT[2] = 2;
    OUT[3] = 1;
    return;
  }
  else{//return XIV
    OUT[0] = 1;
    OUT[1] = 1;
    OUT[2] = 2;
    OUT[3] = 3;
    return;
  }

}
//------------------------------------------------
if(AD<=T){//can be X.
  OUT[0] = 3;
  OUT[1] = 1;
  OUT[2] = 2;
```

```
    OUT[3] = 3;
    return;
  }

  //--------------------------------------------

  if(BC<=T){ //can be XI.
    OUT[0] = 1;
    OUT[1] = 2;
    OUT[2] = 2;
    OUT[3] = 3;
    return;
  }

  //IF we are here then there has been a serious error!
  //Crash and inform.
   printf("split: Should never get here!\n");
   printf("%hhu %hhu %hhu %hhu\n", IN[0], IN[1], IN[2], IN[3]);
   printf("Tol' %f\n", T);
   exit(1);

}
```

B.1.3 Centroid Finding Functions

Main Function

This function is called by the main driver function when it becomes necessary to find all the centroids. It it turn uses the utility functions (see below).

Listing B.3: Centroid main function.

```
// The is the function version of the region centroid finding
program.
// returns an pointer to a linked list.
// IB
// Inputs -- IplImage* SegImage, the segmented image
//           int minarea, the minimum sized region to find a
centroid for
//           int MinRegionCount, the mimumum number of regions.
//
// Outputs - Centroid* Parent, the centroid linked list.
//
//=========================================================================

#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include <cv.h>
#include <highgui.h>
#include "centroid.h"
```

```
Centroid* cent_function(IplImage *SegImage, int minarea, int
MinRegionCount){
  Centroid *Parent;

  //get the image. nb, the program call will need the image name
  //as an argument.
  if(SegImage == NULL) {
    printf("No image loaded!\n");
    exit(1);
  }

  //Now find the centroids and place circles on them.
  //call the function 'getALLCentroids'
  // Centroid *Parent = NULL;
  if(MinRegionCount > 1){
    Parent = getALLCentroids(SegImage);
    if(Parent == NULL) printf("NULL PARENT!!!\n");
    Parent = cleanCentroids(Parent, minarea);
  } else {

    int width = SegImage->width;
    int height = SegImage->height;
    Parent = addCentroid( NULL);

    Parent->area = width*height;
    Parent->x =(width/2)*(Parent->area);
    Parent->y =(height/2)*(Parent->area);


    int i;
    unsigned char *ptr0;
    ptr0=SegImage->imageData;

    for(i=0; i<(width*height); i++){
      Parent->Gsum += *ptr0;
      ptr0++;
      i++;
    }
  }
  if(Parent == NULL) {
    printf("No centroid found this is a bit of an error!\n");
    exit(1);
  }

  //Now that all the centroids are found, draw a circle at each of
  the points.
  // create a cvPoint structure for the x and y coordinates.
  ///*
  CvPoint Cent;
  Centroid *ptr = Parent;

  while(ptr){
    if(ptr->area  > 0) {
      Cent.x = (int)(ptr->x/ptr->area);
      Cent.y = (int)(ptr->y/ptr->area);

      cvCircle(SegImage,          //image name
               Cent,              //center,
               5,                 //radius,
```

```
              CV_RGB(0, 0, 100), //CvScalar color,
              1,                 //int thickness=1,
              8,                 //int lineType=8,
              0 );               //int shift=0 );
    } else {
      printf("Empty Centroid found\n");
    }
    ptr = ptr->next;
  }
  //*/

  return Parent;
}
```

Utility Functions

These functions are designed for the centroid finding operations. They include that is necessary to find centroids, allocate memory for the linked list, create flag arrays, de-allocate memory, and clean up the lists.

Listing B.4: Centroid finding functions.

```
//This file defines all the centroid functions.
//including the flags functions and the calc', get and destroy
//functions.

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

#include <cv.h>
#include <highgui.h>
//nb <> means system headers "" is my headers (local)
#include "centroid.h"
//This defines the structure type 'Centroid'

int AllocateFlags(int n, int m);
int DeallocateFlags();
int SetFlag(int n, int m);
int isFlagSet(int n, int m);


static char* flags = NULL;
static int height = 0;
static int width  = 0;

// height and width can ONLY be seen by functions in THIS file. this
is the
// purpose of the 'static' cmd.

//==========================================================================
//create a function that creates a new structure for a new region.

Centroid *addCentroid( Centroid *parent){
  //This function needs a pointer to the 'Centroid' type struct'
```

```
  //that was the previous centroid structure.

  Centroid *ptr;
  Centroid *new;

  new = (Centroid*)malloc(sizeof(Centroid));
  if(new == NULL) return NULL; //malloc failed !

  //initialise new, now that we have the memory.

  new->colour = -1;
  new->area = 0;
  new->x = 0.0;
  new->y = 0.0;
  new->Gsum = 0;
  new->g_prev = -1;
  new->g_current = -1;
  new->next = NULL;
  new->L = -1;


  if(parent != NULL){
    ptr = parent;
    while(ptr->next) ptr = ptr->next; //loop through continuously
    until a null is found, you are now at the end of the list.

    ptr->next = new;
  }
  return new;
}
//==========================================================================

void deleteCentroids(Centroid *parent){
  Centroid *ptr;
  while(parent){
    ptr = parent->next;
    //ptr becomes the next centroid IF parent actually points to
    a centoid.
    free(parent);
    parent = ptr;
  }
}

//==========================================================================

Centroid *cleanCentroids(Centroid *start, int minarea){
  Centroid *parent;
  Centroid *current;

  if(!start) return NULL;

  while(start && start->area < minarea){
    current = start;
    start = start->next;
    free(current);
  }
  if(!start) return NULL;

  parent = start;
  current = start->next;

  while ( current ) {
    if( current->area < minarea ) {
```

```
      parent->next = current->next;
      free(current);
    } else {
      parent=current;
    }
    current = parent->next;
  }

  return start;
}

//note: The above will clear the linked list FROM WHERE *parent
is. If you give it a address that is half way down the list IT WILL
CLEAR FROM THERE.
//==================================================================

Centroid *getALLCentroids(IplImage *SegImage){
  Centroid *parent = NULL;
  unsigned char *grey = NULL;
  Centroid *new;
  int i, j;

  int width = SegImage->width;
  int height = SegImage->height;

  //AllocateFlags(SegImage->width,SegImage->height);
  AllocateFlags(width,height);

  //printf("Image width and height %d %d\n",width,height);
  for(j=0; j < height; j++){

    for(i=0; i < width; i++){

      if(isFlagSet(i,j)) continue; //continue is skip to end of
      interior loop.

      new = addCentroid(parent);
      if(new==NULL) return NULL;

      if(parent == NULL) parent = new;

      grey = (unsigned char *)(SegImage->imageData +
      SegImage->widthStep * j + i); //Assume Grey-scale

      //printf("(x,y,g) (%d,%d,%hhu)\n",i,j,*grey);

      calcCentroid(i,j,new,*grey,SegImage);

    }//i Loop
  }//j Loop
  //clean the flags.
  DeallocateFlags();

  return parent;
}//END of getALLCentroids

//==========================================================================

void calcCentroid(int i, int j, Centroid *current, unsigned char
grey, IplImage *SegImage){
  //grey is THIS segment colour in segmented image.

  if(i<0 || i >= SegImage->width)     return;
```

```
  if(j<0 || j >= SegImage->height)      return;
  if( isFlagSet(i,j) ) return;

 unsigned char *pixel = (unsigned char *)(SegImage->imageData +
 SegImage->widthStep * j + i); //Assume Grey-scale
  if( pixel == NULL ) {
    printf("pixel == NULL\n");
  }
  if( (*pixel) != grey) return; //the current location is not in
  same segment.

  SetFlag(i,j);

  current->colour = grey;

  current->x += (float)i;
  current->y += (float)j;
  current->area += 1;
  current->Gsum += *pixel;


  //recurse to the surrounding pixels. Note the if statements above
  will stop us from dropping of edge of image &c.

  calcCentroid(i+1,j, current, grey, SegImage);
  calcCentroid(i-1,j, current, grey, SegImage);
  calcCentroid(i,j+1, current, grey, SegImage);
  calcCentroid(i,j-1, current, grey, SegImage);

}//End of calcCentroid.

// ======================== FLAG FUNCTIONS ========================

int AllocateFlags(int w, int h){
  int i;
  char* ptr;
  flags = (char*)malloc(w*h*sizeof(char));
  // create a matrix of char's the correct size as defined by n & m;

  if(flags == NULL) return 1;
  // if that did not work return an error. it probably means you
  are out of
  // memory.

  for(i=0, ptr=flags; i < w*h; i++, ptr++){
    *ptr = 0;
  }
  //initalise this matrix. Note: i< not i<= as it is assumed that
  // this loop will increment i and ptr one last time after the
  penultimate
  // entry to do i = n*m.

  height = h;
  width =  w;
  return 0;
  // if it did work return no error.
}
//----------------------------------------------------------------------------


int DeallocateFlags(){
  // This function dealocates the variable flags and is designed to
```

```
  // prevent the memory from filling up with 'flags'.

  if(flags) free(flags);
  // if there is something in flags, free flags.
  height =0;
  width = 0;
  flags = NULL;
}
//---------------------------------------------------------------------------
int SetFlag(int w, int h){
  char *ptr;

  ptr = flags + h*width + w;
  *ptr = 1;

}
// NOTE!! width in the function above and bellow is defined
outside the
// functions at the top of the page. The allocation function (which
should be
// run first) gives it a value. It is then used with this value by
the functions
// above and bellow

//---------------------------------------------------------------------------
int isFlagSet(int w, int h){

  if(w<0 || w >= width) return 1;
  if(h<0 || h >= height) return 1;

  return (int) (*(flags + h*width + w));

  // This will return either a one or a zero
  // depending on weather the flag has been set at
  // this point or not. 1 == set, 0 == not set.
}
```

Data Type Definition

This section of code defines the `Centroid` data type. It is used in all functions requiring use of centroid type structures.

Listing B.5: "Centroid" data structure.

```
// Header for the centroid finding code.
// This defines the structure Centroid. It will have these type
variables in it.
// THIS MUST be inclued in the start of all files that use this
structure.
// eg #include "centroid.h" is required!

typedef struct _Centroid{
  int colour;
```

```
  int area;
  float x;
  float y;
  int  Gsum;
  float g_prev;
  float g_current;
  float L;
  struct _Centroid *next;
} Centroid;

Centroid  *getALLCentroids(IplImage *SegImage);
void       deleteCentroids(Centroid *);
Centroid *cleanCentroids(Centroid *, int);
Centroid  *addCentroid( Centroid *parent);
Centroid *cleanCentroids(Centroid *start, int minarea);
void calcCentroid(int i, int j, Centroid *current, unsigned char
grey, IplImage *SegImage);

//colour is the colour in segmented image of current region.
//area is area of region.
//x is the sum of all x locations.
//y is the sum of all y locations.
//Gsum is the sum of all gray values for that region in original
image.
//g_prev is the looming quantitiy for the previous image (as defined
in tracking).
//g_current is the looming quantity for the current image, as define
by blur.
// L is the looming value for this region.
```

B.1.4 Tracking Function

Having found the centroids the main program calls this function to establish the correspondence from frame to frame.

Listing B.6: Tracking function.

```
/*
This is the tracking function designed for use in the 'driver'
program.
This will not actualy return anything except the lines on the image.

It is intended that it be passed the centroid lists.

INPUTS:  IplImage* Seg2 -- this is the segmented image that the
tracked
                          lines will be drawn on. Note that it
                          is the
                          CURRENT image.
         Centroid* Seg1Cent -- This is the centroid list for the
         PREVIOUS
                               frame.
         Centroid* Seg2Cent -- This is the centroid list for
```

```
        the CURRENT
                                frame.
        int Tol  -- The tolerance used in the segmentation function.

IB
26/7/11
*******************************************************************************/

#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include <cv.h>
#include <highgui.h>
#include "centroid.h"

void track_general(IplImage* Seg2, Centroid* Seg1Cent, Centroid*
Seg2Cent,int Tol){


  //provide contingency for their being no centroid. (first frame
  of sequence).
  if(Seg2Cent == NULL) return;
  if(Seg1Cent == NULL) return;

  //now that we have the centroids for all three Segs, start with
  Seg one
  //and find the closest centroid to (Seg1Cent 1st point) in
  Seg2Cent.
  //the first step is to go to Seg2 and the first centroid on
  the list.
  //find the area of the region attached to that centroid.

  Centroid *NewImg = Seg2Cent;
  Centroid *OldImg = Seg1Cent;
  Centroid *temp;

  //These will be the coordinates in the PREVIOUS image that are
  //best match for current centroid in CURRENT image.
  int BestXMatch = -1;
  int BestYMatch = -1;

  //These are the coordinates of a centroid in a PREVIOUS image that
  //we are currently considering.
  int PrevX;
  int PrevY;
  float PrevG;

  //these are the cooridinates of the current centroid in the CURRENT
  //image
  int CurX;
  int CurY;

  //this is colour tolerance.
  Tol -= Tol*0.1;

  //get number of centroids in old image
  int N = 0;
  temp = Seg1Cent;

  while(temp){
    N++;
    temp=temp->next;
```

```
}
//The variable N is the number of centroids in the PREVIOUS image.

//
//                    CURRENT CENTROID LOOP
//

int count = 0;
while(NewImg){
    //reset the PREVIOUS image pointer to the start of the PREVIOUS
    //image centroid list.
    OldImg = Seg1Cent;

    //reset the final  match
    BestXMatch = -1;
    BestYMatch = -1;

    //Find the search area dimension based on size of CURRENT region
    int L = (int)(sqrt(NewImg->area)*0.5);

    //get the location of the CURRENT centroid
    CurX = (int)(NewImg->x / NewImg->area);
    CurY = (int)(NewImg->y / NewImg->area);


    //Now go through all the centroids in the PREVIOUS image and
    //look for centroids that match. The match will be decided by
    // 1) if the PREVIOUS centroid is in the CURRENT search area.
    // 2) if the PREVIOUS region's mean is within tol of the
    CURRENT mean.
    //
    // if these two conditions are satisfied, then a match is made
    and the
    // BestXMatch &c. variables can be set. Note that this makes
    it possible
    // to have several CURRENT centroids assigned to one PREVIOUS
    centroid.
    // this is intended.

    int GMdiff = 1000;
    int GM = 0;
    int PrevGM;
    int CurGM;

    while(OldImg){
        //find the PrevX and PrevY variables.

        PrevX = (int)(OldImg->x / OldImg->area);
        PrevY = (int)(OldImg->y / OldImg->area);
        CurGM = (int)(NewImg->Gsum / NewImg->area);

        if(CurX-L < PrevX && CurX+L > PrevX && CurY-L < PrevY &&
        CurY+L > PrevY){
            //we are in search area.
            //now check the colour match.
            PrevGM = (int)(OldImg->Gsum / OldImg->area);
```

```
        if(abs(PrevGM - CurGM) < GMdiff){
          GMdiff = abs(PrevGM - CurGM);
          BestXMatch = PrevX;
          BestYMatch = PrevY;
          PrevG = (OldImg->g_current);
          //printf("PrevG:%f\n", PrevG);

        }else{//end colour if
          //printf("Colour not match!\n");
        }
      }//end search area if

      OldImg = OldImg->next;
    }//end OldImg

    if(BestXMatch > 0 && BestYMatch > 0){

      CvPoint Old;
      CvPoint New;

      New.x = CurX;
      New.y = CurY;
      Old.x = BestXMatch;
      Old.y = BestYMatch;

      //printf("The previous x,y position: %d %d\n",
      BestXMatch,BestYMatch );
      //printf("previous area from track %d\n", PrevG);

      cvLine(
             Seg2,
             New,
             Old,
             CV_RGB(100, 100, 100),
             2,
             8,
             0
             );
      count++;
    }/*
       else{//end if.
      printf("NO MATCH FOUND \n");
      }*/
    // printf("End of CURRENT centroid\n\n");
    NewImg->g_prev = PrevG;
    NewImg = NewImg->next;

  }//END OF NewImg LOOP!

 //show how many regions tracked...
 //we are going to see how many regions in the new list have
 //succesfully tracked.
 int total = 0;
 Centroid* Temp = Seg2Cent;
 while(Temp){
   total++;
   Temp = Temp->next;
 }
```

```
  //the precentage number of tracked...
  float p = (float)(count)/(float)(total)*100;

  // printf("Total no of centroids: %d\n", total);
  // printf("Total number tracked: %d\n", count);
  printf("The percent no of regions tracked: %0.2f\n", p);

}//end of void trackf(....

// ==================== END OF FUNCTION ======================
```

B.1.5 Looming Function

Listing B.7: Looming function.

```
/* This function is designed to calculate looming through blur
radius.
   It is intended that this shall simply take the blur value from the
``current'' and  ``previous'' lists and do the blur calculation.
IB
12/10/11

INPUTS ---- IplImage* Segmented, the segmented image
            Centroid* CentList, the current frame centroids.
            float dt, the time step.

No outputs.

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


#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include <cv.h>
#include <highgui.h>
#include "centroid.h"

void loomf_blur(IplImage* Segmented, Centroid* CentList, float dt){

  //create a centroid counter.
  Centroid* temp;

    //go through the centroid list and find the Looming (L) values
    for all
  //regions in list.

   float L = -1;

   temp = CentList;
   while(temp){
     if(temp->g_prev >= 0 && temp->g_current > 0 ){
       L = ((temp->g_current - temp->g_prev) / dt) / temp->g_current;
     }

     if(temp->g_prev < 0){
```

```
        L = 0;
      }

      if(temp->g_current == 0){
        L = 1e10;
      }

      if(temp->g_current <0){
        L = 0;
      }

      // if L is '0' there is insufficient information to compute L (
      // either the current or previous blur is undefined).
      //this is a comprimise as '0' looming can occur
      legitimately. Howvever, no
      //other usable marker existed.
      //if L > 1e9, then there is 'infinite' looming, the object is
      upon you.

      //printf("L = %f\n", L);
      temp->L = L;
      temp = temp->next;
      L = -1;
    }
    //the looming value is now known.


}//end of loomf_blur
```

B.2 Testing Algorithms

B.2.1 Segmentation Test Program

This function is designed to test the segmentation function. The full results of this testing is discussed in section 5.2. To operate this function needs access to the test image and the segmentation function.

Listing B.8: Segmentation Test Program.

```
/* This is a test program for verifying the segmentation algorithm.
   This program will create a test image (two black and white
   squares)
   and subject them to segmentation. It will then go through the
   segmented
   image and attempt to find any pixels that are not 0 or 255. If
   it finds
   any it will report them.
   IB
```

```
  22/7/11
***************************************************************************/
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include <cv.h>
#include <highgui.h>

int seg_function(IplImage *, IplImage *, int, int, CvSize);

int main( ){

  IplImage * TestImage;
  IplImage * SegImage;

  //now create the size

  CvSize Size;
  Size.width = 200;
  Size.height = 200;

 TestImage = cvCreateImage(Size,IPL_DEPTH_8U,1);
 SegImage = cvCreateImage(Size,IPL_DEPTH_8U,1);

 //now fill Test image with the apropriate colours.

 unsigned char* PtrT;

 PtrT = (unsigned char*)(TestImage->imageData);

 int x = 0;
 int y = 0;

 for(y = 0; y < Size.height; y++){

   for(x = 0; x < Size.width; x++){
     PtrT = (unsigned char*)(TestImage->imageData + x +  y *
     TestImage->widthStep);

     if(x < (Size.width)/2){
       *PtrT = 0;
     }else{
       *PtrT = 255;
     }
   }//end for(x< ...
 }//edn for(y<...

 //we now have a half white and half black image. Give this image
 to the
 //segmentation function.

 int MinRNo = seg_function(TestImage, SegImage, 100,100, Size);

 //now go through this image and find all suspect pixels.

 x = 0;
 y = 0;
 int count = 0;
 unsigned char* PtrS = (unsigned char*)(SegImage->imageData);

 for(y = 0; y < Size.height; y++){
   for(x = 0; x< Size.width; x++){
     PtrS = (unsigned char*)(SegImage->imageData + x + (y *
     SegImage->widthStep));
```

```
      if(*PtrS != 0 || *PtrS != 255){
        count++;
        }
   }//x
 }//y

 printf("The number of suspect pixels is: %d\n", count);



 //==================DISPLAY==================
 cvNamedWindow("TestImage",CV_WINDOW_AUTOSIZE);
 cvNamedWindow("SegImage",CV_WINDOW_AUTOSIZE);

 cvShowImage( "TestImage", TestImage );
 cvShowImage( "SegImage", SegImage);
 cvWaitKey(0);

 cvReleaseImage( &TestImage );
 cvReleaseImage( &SegImage );

 cvDestroyWindow( "TestImage" );
 cvDestroyWindow( "SegImage" );



}//end of Main
```

B.2.2 Tracking Test Program

This program was designed to test the operation of the tracking function. The full results can be found in section 5.4

Listing B.9: Tracking Test Program.

```
/* This program is the tracking test program. It demonstrates that
the tracking
algorithm works correctly for test images. It takes as its input
2 still
images. It is recommended that these stills be modified versions
of the same
image or identical images. The program will save the tracked
images as
'Track1OUT.png' and 'Track2Out.png' in the current directory. All
inputs must
be single channel gray-scale

The executable file is 'TrackTest'
```

```
Iain Brookshaw
w0086292

28/9/11
************************************************************************

#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include <cv.h>
#include <highgui.h>
#include "centroid.h"

int seg_function(IplImage *, IplImage *, int, int, CvSize);
Centroid* cent_function(IplImage *, int , int );

int main(int argc, char** argv){
  //now to load three images

  //define and get the three images that we will be using.
  IplImage* frame1;
  IplImage* frame2;
  // IplImage* frame3;
  IplImage* Seg1;
  IplImage* Seg2;

  cvNamedWindow("Frame1",CV_WINDOW_AUTOSIZE);
  cvNamedWindow("Frame2",CV_WINDOW_AUTOSIZE);

  cvNamedWindow("Seg1",CV_WINDOW_AUTOSIZE);
  cvNamedWindow("Seg2",CV_WINDOW_AUTOSIZE);

  frame1 = cvLoadImage(argv[1],
                       CV_LOAD_IMAGE_GRAYSCALE
                       );
  frame2 = cvLoadImage(argv[2],
                       CV_LOAD_IMAGE_GRAYSCALE
                       );

  //now find the centroids of regions in frame1.

  CvSize Size;

  int xMax = frame1->width;
  int yMax = frame2->height;
  Size.width = xMax;
  Size.height = yMax;

  Seg1 = cvCreateImage(Size,IPL_DEPTH_8U,1);
  Seg2 = cvCreateImage(Size,IPL_DEPTH_8U,1);

  int RMCount1 = seg_function(frame1, Seg1, xMax, yMax, Size);
  int RMCount2 = seg_function(frame2, Seg2, xMax, yMax, Size);

  Centroid* Seg1Cent = cent_function(Seg1, 100, 2 );
  Centroid* Seg2Cent = cent_function(Seg2, 100, 2 );


   //now that we have the centroids for all three Segs, start with
   Seg one
  //and find the closest centroid to (Seg1Cent 1st point) in
  Seg2Cent.
  //the first step is to go to Seg2 and the first centroid on
```

```
the list.
//find the area of the region attached to that centroid.

Centroid *NewImg = Seg2Cent;
Centroid *OldImg = Seg1Cent;
Centroid *temp;

//These will be the coordinates in the PREVIOUS image that are
//best match for current centroid in CURRENT image.
int BestXMatch = -1;
int BestYMatch = -1;

//These are the coordinates of a centroid in a PREVIOUS image that
//we are currently considering.
int PrevX;
int PrevY;

//these are the cooridinates of the current centroid in the CURRENT
//image
int CurX;
int CurY;

//this is colour tolerance. SHOULD COME FROM SEG_FUNCTION.
int Tol= 10;

//get number of centroids in old image
int N = 0;
temp = Seg1Cent;

while(temp){
  N++;
  temp=temp->next;
}

//The variable N is the number of centroids in the PREVIOUS image.

//--------------------------------------------------------------------------
//                        CURRENT CENTROID LOOP
//--------------------------------------------------------------------------

int count = 0; // this is the number of successfully tracked
regions.

while(NewImg){

  //reset the PREVIOUS image pointer to the start of the PREVIOUS
  //image centroid list.
  OldImg = Seg1Cent;

  //reset the final  match
  BestXMatch = -1;
  BestYMatch = -1;

  //Find the search area dimension based on size of CURRENT region
  int L = (int)(sqrt(NewImg->area)*0.5);

  //get the location of the CURRENT centroid
  CurX = (int)(NewImg->x / NewImg->area);
  CurY = (int)(NewImg->y / NewImg->area);
```

```
//printf("CurX: %d\n", CurX);
//printf("CurY: %d\n", CurY);


//Now go through all the centroids in the PREVIOUS image and
//look for centroids that match. The match will be decided by
// 1) if the PREVIOUS centroid is in the CURRENT search area.
// 2) if the PREVIOUS region's mean is within tol of the
CURRENT mean.
//
// if these two conditions are satisfied, then a match is made
and the
// BestXMatch &c. variables can be set. Note that this makes
it possible
// to have several CURRENT centroids assigned to one PREVIOUS
centroid.
// this is intended.

int GMdiff = 1000;
int GM = 0;
int PrevGM;
int CurGM;

while(OldImg){
  //find the PrevX and PrevY variables.

  PrevX = (int)(OldImg->x / OldImg->area);
  PrevY = (int)(OldImg->y / OldImg->area);
  CurGM = (int)(NewImg->Gsum / NewImg->area);


  if(CurX-L < PrevX && CurX+L > PrevX && CurY-L < PrevY &&
  CurY+L > PrevY){
    //we are in search area.
    //now check the colour match.
    PrevGM = (int)(OldImg->Gsum / OldImg->area);
    /*
    printf("area Previous: %d\n", OldImg->area);
    printf("gray sum Previous: %d\n", OldImg->Gsum);
    printf("area Current %d\n", NewImg->area);
    printf("gray sum Currnet: %d\n", NewImg->Gsum);

    printf("Area match found\n");
    printf("Prev Gray Mean %d\n", PrevGM);
    printf("Current Gray Mean %d\n", CurGM);
    */
    if(abs(PrevGM - CurGM) < GMdiff){
      GMdiff = abs(PrevGM - CurGM);
      BestXMatch = PrevX;
      BestYMatch = PrevY;
      //printf("colour match found\n");
    }else{//end colour if
      //printf("Colour not match!\n");
    }
  }//end search area if

  OldImg = OldImg->next;
}//end OldImg

//printf("BestXMatch: %d\n",BestXMatch);
```

```
    //printf("BestYMatch: %d\n",BestYMatch);

    if(BestXMatch > 0 && BestYMatch > 0){

      CvPoint Old;
      CvPoint New;

      New.x = CurX;
      New.y = CurY;
      Old.x = BestXMatch;
      Old.y = BestYMatch;

      cvLine(
              Seg2,
              New,
              Old,
              CV_RGB(255, 255, 255),
              2,
              8,
              0
              );

      count++;

    }else{//end if.
      //printf("NO MATCH FOUND \n");
    }
    //printf("End of CURRENT centroid\n\n");
    NewImg = NewImg->next;

  }//END OF NewImg LOOP!



  //we are going to see how many regions in the new list have
  //succesfully tracked.
   int total = 0;
  Centroid* Temp = Seg2Cent;
  while(Temp){
    total++;
    Temp = Temp->next;
  }

  //the precentage number of tracked...
  float p = (float)(count)/(float)(total)*100;

  printf("Total no of centroids: %d\n", total);
  printf("Total number tracked: %d\n", count);
  printf("The percent no of regions tracked: %0.2f\n", p);

  //================ Save images! ===================================

  int s =cvSaveImage("Track1OUT.png", Seg1, 0);

  int s1= cvSaveImage("Track2OUT.png", Seg2, 0);

  //================ Display and Clean ==============================

  cvShowImage( "Frame1", frame1 );
  cvShowImage( "Frame2", frame2 );
  cvShowImage( "Seg1", Seg1 );
```

```
  cvShowImage( "Seg2", Seg2 );
  cvWaitKey(0);

  //clean up
  //Remove the centroid list.
  deleteCentroids(Seg1Cent);
  deleteCentroids(Seg2Cent);

  Seg1Cent = NULL;
  Seg2Cent = NULL;

  cvReleaseImage( &frame1 );
  cvReleaseImage( &frame2 );
  cvReleaseImage( &Seg1 );
  cvReleaseImage( &Seg2 );

  cvDestroyWindow( "Frame1" );
  cvDestroyWindow( "Frame2" );
  cvDestroyWindow( "Seg1" );
  cvDestroyWindow( "Seg2" );

}//END OF MAIN!!!
```

B.2.3 Looming Test Program

Listing B.10: Area Looming Test Program.

```
/* This is a test program designed to test All aspects on a still
image.
   This is primarily intended to test looming through area
*/

#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include <cv.h>
#include <highgui.h>
#include "centroid.h"

int seg_function(IplImage *, IplImage *, int, int, CvSize, int T);
Centroid* cent_function(IplImage *SegImage, int minarea, int
MinRegionCount);
void trackL(IplImage* Seg2, Centroid* Seg1Cent, Centroid* Seg2Cent,
int T);

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

  IplImage *Segmented;
  IplImage *Segmented2;
  IplImage *Input= cvLoadImage( argv[1], CV_LOAD_IMAGE_GRAYSCALE );
  IplImage *Input2 = cvLoadImage( argv[2], CV_LOAD_IMAGE_GRAYSCALE );

  cvNamedWindow("segmented1", CV_WINDOW_AUTOSIZE );
  cvNamedWindow("segmented2", CV_WINDOW_AUTOSIZE );
```

```
cvNamedWindow("original", CV_WINDOW_AUTOSIZE );
cvNamedWindow("original2", CV_WINDOW_AUTOSIZE );

// ---------------------------------------------------------------//
//                SEGMENT THE IMAGES.                              //
// ---------------------------------------------------------------//

//get image size
int xMax = Input -> width;
int yMax = Input -> height;

CvSize Size;
Size.width  = xMax;
Size.height = yMax;

//prepare the segmented images.
Segmented = cvCreateImage(Size,IPL_DEPTH_8U,1);
Segmented2 = cvCreateImage(Size,IPL_DEPTH_8U,1);

float Tol = 0.0;

int MinCount1 = seg_function(Input, Segmented, xMax, yMax,
Size, Tol);
int MinCount2 = seg_function(Input2, Segmented2, xMax, yMax,
Size, Tol);

printf("images segmented\n ");

//----------------------------------------------//
//          get the centroids                    //
//----------------------------------------------//

Centroid* CentList1;
Centroid* CentList2;

CentList1 = cent_function(Segmented, 200, 2);
printf("have the 1st centroid\n");

CentList2 = cent_function(Segmented2, 200, 2);
printf("have the 2nd centroid\n");

Centroid* temp;

// ------------------------------------------------- //
//                   TRACKING                         //
// ------------------------------------------------- //

trackL(Segmented2, CentList1, CentList2, Tol/2);

//----------------------------------------------//
//                  Looming                      //
//----------------------------------------------//

float L = -1;
float dt = 0.05;

temp = CentList2;
while(temp){
  L = ((temp->area - (int)temp->g_prev) / dt) / temp->area;
  printf("Looming value: %f\n", L);
```

```
        temp->L = L;
        temp = temp->next;
        L = -1;
    }
    // ---------------------------------------------------------------

    printf("Have all looming values, getting largest...\n");
    //let us find the region that has grown the largest.

    float maxL = -1e6;
    int maxLx = -1;
    int maxLy = -1;
    CvPoint Loc;

    temp = CentList2;
    while(temp){
        if(temp->L > maxL){

            maxL = temp->L;
            maxLx = (int)(temp->x / temp->area);
            maxLy = (int)(temp->y / temp->area);
        }
        temp = temp->next;
    }

    if(maxLx > 0 && maxLy > 0){
        printf("The centroid of the fastest growing region is: %d,
    %d\n", maxLx, maxLy);
        //draw a circle there.
        Loc.x = maxLx;
        Loc.y = maxLy;

        cvCircle(Segmented2,          //image name
                 Loc,                //center,
                 10,                 //radius,
                 CV_RGB(0, 0, 100),  //CvScalar color,
                 5,                  //int thickness=1,
                 8,                  //int lineType=8,
                 0 );                //int shift=0 );
    }else{
        printf("No max L value found!");
    }

    //---------------------------------------------

    cvShowImage( "Segmented", Segmented);
    cvShowImage( "Segmented2", Segmented2);
    cvShowImage( "original", Input);
    cvShowImage( "original2", Input2);

    int r = cvSaveImage("Out1.png", Segmented);


    cvWaitKey(0);
    if( Input != NULL )    cvReleaseImage( &Input );
    if( Segmented != NULL )    cvReleaseImage( &Segmented );

    cvDestroyWindow( "Segmented1" );
    cvDestroyWindow( "Segmented2" );
```

```
  cvDestroyWindow( "original" );
  cvDestroyWindow( "original2" );

  cvDestroyWindow( "Input");
}//end main
```