3-2005

Real-Time Mapping Using Stereoscopic Vision Optimization

Kevin M. Biggs

REAL-TIME MAPPING USING STEREOSCOPIC VISION OPTIMIZATION

THESIS

Kevin M. Biggs
Captain, USAF

AFIT/GCS/ENG/05-03

DEPARTMENT OF THE AIR FORCE

AIR UNIVERSITY

AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

AFIT/GCS/ENG/05-03

REAL-TIME MAPPING USING STEREOSCOPIC VISION OPTIMIZATION

THESIS

Presented to the Faculty

Department of Electrical and Computer Engineering

Graduate School of Engineering and Management

Air Force Institute of Technology

Air University

Air Education and Training Command

In Partial Fulfillment of the Requirements for the

Degree of Master of Science

Kevin M. Biggs, BA

Captain, USAF

March 2005

AFIT/GCS/ENG/05-03

REAL-TIME MAPPING USING STEREOSCOPIC VISION OPTIMIZATION

THESIS

Kevin M. Biggs, BA
Captain, USAF

Approved:

/signed/	3 Mar 2005
Dr. Gilbert L. Peterson (Chairman)	Date
/signed/	4 Mar 2005
Dr. Henry B. Potoczny (Member)	Date
/signed/	7 Mar 2005
Dr. John F. Raquet (Member)	Date

Acknowledgements

Several selfless people made key contributions to help make this research come to fruition.

First, I would like to thank my *wonderful family* for their immense support over the past 18 months. Each of them sacrificially adapted to allow me the time necessary for my studies. Kevin Owens, my friend, wingman, and classmate, provided invaluable coding assistance, lucid problem solving, and a robust map viewer. I would also like to recognize my laboratory neighbors Ryan Hanson and Brian Smith for their impeccable LaTeX support.

Most importantly, I would like to thank God for sustaining and guiding me throughout this rigorous experience. *–Ephesians 3:14-21*

Kevin M. Biggs

Table of Contents

Page

List of Figures

List of Tables

List of Abbreviations

AFIT/GCS/ENG/05-03

Abstract

This research focuses efficient methods of generating 2D maps from stereo vision in real-time. Instead of attempting to locate edges between objects, we make the assumption that the representative surfaces of objects in a view provide enough information to generate a map while taking less time to locate during processing. Since all real-time vision processing endeavors are extremely computationally intensive, numerous optimization techniques are applied to allow for a real-time application: horizontal spike smoothing for post-disparity noise, masks to focus on close-proximity objects, melding for object synthesis, and rectangular fitting for object extraction under a planar assumption. Additionally, traditional image transformation mechanisms such as rotation, translation, and scaling are integrated. Results from our research are an encouraging 10Hz with no vision post processing and accuracy up to 11 feet. Finally, vision mapping results are compared to simultaneously collected sonar data in three unique experimental settings.

REAL-TIME MAPPING USING STEREOSCOPIC VISION OPTIMIZATION

I. Introduction

This thesis presents the results of a study into optimization methods of stereoscopic vision and mapping. Vision is a very flexible sense. Human beings rely mostly on vision when interacting in the world. We identify objects (depth and motion) mainly by just *seeing* them. Such natural sensing would be very desirable for a robot to possess. This project aims at studying robotic-based visual collection techniques to efficiently process and store a 2D map of a traversed environment.

Vision as a source for map building is superior to other kinds of sensing devices like laser scanners and ultrasonic sensors for a couple of key reasons. First, vision systems can collect environmental data passively (i.e. no detectable signal emission). This stealthy characteristic bodes well in situations where low observability is a high priority. Second, it collects a very dense set of data relatively fast. However, along with vision's rapid data collection rate comes the problem of how to handle large volumes of data quickly and efficiently; therefore, effective and robust algorithms have to be employed for vision processing and real-time map reconstruction.

To expedite and simplify processing, instead of focusing on attempting to locate edges between objects, we make the assumption that the representative surfaces of objects in a view provide enough information to generate a map while taking less time to locate during processing. We call this the *planar assumption*.

1.1 Motivation

This research is motivated by the new threats that currently face the United States. Historically, the United States has faced enemies that were typically conventional with respect to their organization. These conventional armies were well-defined by geographical boundaries, uniforms and a clear leadership hierarchy. However, after 45 years (1945-1990) of global competition between the United States and the Soviet Union, known as the Cold War, the United States has increasingly faced a burgeoning and somewhat unfamiliar foe–*terrorism*. These terrorists are intentionally loose-knit and often in deep hiding in subterranean environments such as caves or bunkers. Their ad-hoc nature makes combatting them intrinsically difficult. Although periodic terrorist acts have surfaced against U.S. assets throughout the globe, our country was not engaged in a full-scale war on terror until 9-11, the Pearl Harbor of the 21st Century.

Recent large-scale terrorist attacks on United States' soil has elevated them from a mere threat to a top priority enemy. The attacks on the World Trade Centers and Pentagon on September 11, 2002 killed 3,025 U.S. citizens and other nationals. [32] President George W. Bush responded nine days later by addressing a Joint Session of Congress and the American people. The focus of his address was to identify those responsible for the 9-11 attacks and to lay an aggressive plan to combat terrorism by hunting those directly responsible in addition to those who harbor them. In an excerpt from his address, he stated that:

> "On September the 11th, enemies of freedom committed an act of war against our country. Americans have known wars, but for the past 136 years, they have been wars on foreign soil, except for one Sunday in 1941. Americans have known the casualties of war, but not at the center of a great city on a peaceful morning. Americans have known surprise attacks, but never before on thousands of civilians." President Bush went on to say, "Our war on terror begins with al-Qaeda, but does not end there. It will not end until every terrorist group of global reach has been found, stopped and defeated." [30]

He backed his words and took action by directing military forces to Afghanistan to search for al-Qaeda terrorists and the Taliban leadership that harbored them in late September, 2001. Once there, U.S. forces found that our foes had gone into hiding–deep into Afghanistan's tortuous Karez caves.

1.2 Current Techniques to Combat "Buried" Enemies

The U.S. military has a special weapon for penetrating hardened underground targets called the Guided Bomb Unit-28 or GBU-28. The GBU-28 is a 5,000-pound laser-guided conventional munition that uses a 4,400-pound penetrating warhead. The bombs are modified Army artillery tubes, weigh 4,637 pounds, and contain 630 pounds of high explosives. They are fitted with GBU-27 LGB kits, 14.5 inches in diameter and almost 19 feet long. The operator illuminates a target with a laser designator and then the munition guides to a spot of laser energy reflected from the target. [26]

However before we can deploy this weapon against a target, the first, time-consuming step is to find these buried hideouts. This can be done from the ground or air. Some common ground techiniques include seismic, earth resistance tomography, ground-penetrating radar, microgravity, and thermal imagers. Other airborne techniques include electromagnetics, satellite imagery, and radar. However, experts still believe that the most valuable cave-finding means in Afghanistan will be people, civilians or former mujahedin, with knowledge of the caves used for hiding from Soviet troops during the 1980s and subsequent tunnel construction. [36]

Once an underground structure is found, then the second step is data collection on that structure. Forces must physically traverse it, akin to the Tunnel Rats in Vietnam, to find if it is inhabited and possibly map its structure for intelligence purposes. As mentioned earlier, this is often very dangerous and time consuming.

To reliably gather information about such hard to reach places while minimizing loss of life, autonomous systems–such as robots–are an attractive solution.

1.3 Proposed Solution to Combat "Buried" Enemies

Our research aims to provide a robotic platform to map subterranean environments using passive sensing in order to save lives and expedite the search process of these dangerous hideouts. Since the platform is autonomous, needing no human control, theoretically, hundreds of these could be scattered on top of a tunnel complex and later be aggregated together to provide a synthesized, master map without a single lost life.

It is important to note that the term passive refers to the method in which the environmental data is collected. Unlike more traditional methods that emit a signal and wait for the return to calculate distance based on the delay (such as sonar, laser, and radar), vision data is collected without emitting any signal. Its distance calculation is done mathematically from image pairs and known camera geometry. Furthermore, with regard to passivity, the size and noise level of the mobile robotic platform is not addressed in this paper. Ideally, the mechanical robotic platform should be small, quiet, and inexpensive.

Once a subterranean complex is stereoscopically mapped, the intelligence data could be relayed to military planners to help make quicker, better-informed decisions concerning guerilla-network warfare. Knowing that terrorists often are in deep hiding, this knowledge of internal structure is a vitally important link to swiftly combat terrorism.

1.4 Approach

Our research approach to efficiently map an environment is based upon a planar assumption where the environment is described in terms of orthogonal planes to reduce complexity. Such an approach simplifies the domain by merging

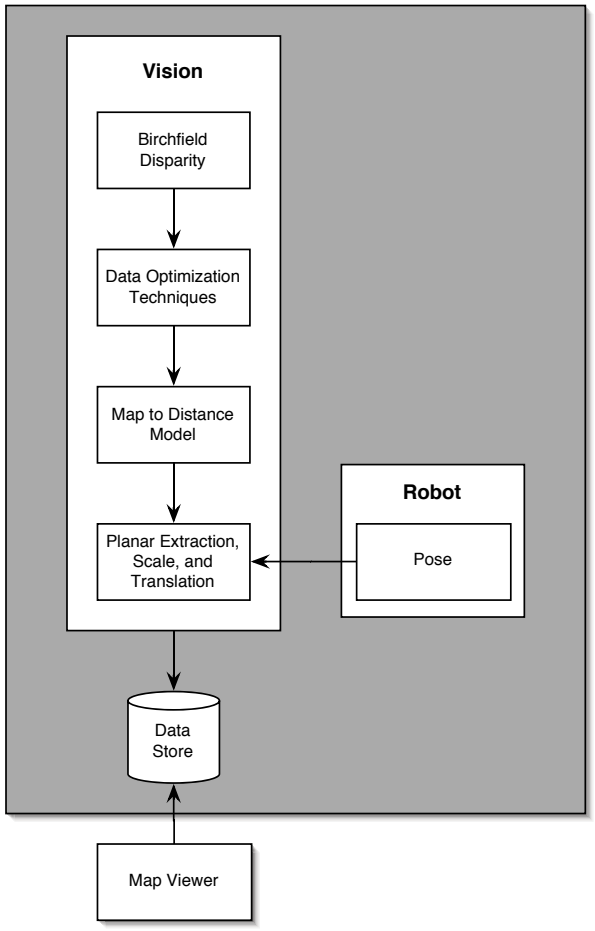


Figure 1.1: System dataflow

object data points into the closest plane depth. After which, an object can be represented by a few Cartesian coordinates, thus speeding performance capabilities.

Figure 1.1 shows the outline of the algorithm and data flow developed in this thesis. The area boxed in gray represents the scope of our research and how we handle our planar assumption. The map viewer is a peer-generated visualization tool. The first vision sub-task is calculating disparity via OpenCV's Birchfield correspondence algorithm [5]. Next, our data optimization techniques (horizontal spike smooth, masking and melding, stack-based flood fill piggyback) are applied to the Birchfield output and are subsequently mapped to a distance model. The final vision processing sub-task involves extracting object planes based

upon our planar assumption, scaling those planes based on depth, and translating the vision's local coordinates based upon the robot's global pose. After these sub-tasks are complete, the mapping information is saved to file and the process is iteratively executed again. Mapping information is constantly appended to the file until the program is terminated. Once the program is terminated the saved data can be viewed, without computation, in the map viewer.

1.5 Thesis Outline

In Chapter II, background information related to stereo vision is discussed along with a stereo geometry overview, calibration and rectification procedures, the stereo correspondence problem and some solutions. Implementation details and methodologies are explained in Chapter III. Key areas include data manipulation techniques, planar extraction, disparity mapping to distance model, data scaling, and map output. Chapter IV introduces experimental mapping results and analysis from different environments. Finally, conclusions and suggestions for future work and extensions are presented in Chapter V.

II. Background Theory and Related Work

2.1 Introduction

The goal of robot mapping is to build a representative map of the environment it has sensed. Traditional robot sensing mechanisms, such as sonar and laser range finders, are easily detectable because of a purposely emitted signal that is used to calculate range based on propagation times. Since only passive sensing is used in our research, it is important to understand its complex underlying principles. In the following sections we discuss foundational vision-related topics. We present three fundamental components of stereo geometry in Section 2.2. In Section 2.3, we detail camera calibration and image rectification. Next, we introduce the stereo correspondence problem and its ramifications in Section 2.4. In Section 2.5 we compare stereo correlation techniques. Finally, in Sections 2.6 and 2.7 we will bring the reader up-to-speed on progress in our field of research and in Section 2.8 we lay out our primary research contributions.

2.2 Stereo Geometry Basics

Understanding stereo geometry is critical to the interpretation, processing, and reconstruction of stereoscopically collected environmental data. The following sections lay the framework of key axioms that hold for stereo geometry.

2.2.1 The Fundamental Matrix. The *fundamental matrix* (F) describes the relationship between a pair of 2D pictures of a 3D scene. This is accomplished in terms of a rank deficient linear transform of the image points in homogeneous 2D coordinates. When the internal camera parameters are excluded, it is described by the *essential matrix* (E). When normalization of the internal camera parameters is included in the matrix, it is called the fundamental matrix. As will be discussed in Section 2.3.1, our camera rig uses to the latter matrix.

2.2.1.1 Projective Geometry. Projecting a point X in world coordinates onto an image plane of a camera with a center of projection C is done by calculating the point of intersection of the line connecting C and X with the image plane. Assuming the camera is a pinhole camera (focal length of 1), center C to be at $(0,0,0)^T$ and viewing direction along the z-axis, the projection is straightforward. As can be verified by simple triangulation, the coordinates of the projected point are $(\frac{x_1}{x_3}, \frac{x_2}{x_3})^T$ and the projection can be expressed in homogeneous coordinates by [21]:

$$X = \begin{pmatrix} X_1 \\ X_2 \\ X_3 \\ 1 \end{pmatrix} \longmapsto \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{pmatrix} X_1 \\ X_2 \\ X_3 \\ 1 \end{pmatrix} \tag{2.1}$$

The matrix in this equation is the *camera projection matrix* (P). The camera projection matrix can have different entries accounting for non-normalized camera geometries denoted by K or a camera in general position ($M = [R|T]$[1]). This will be written $P = KM$. A projection is then expressed as $x = PX$.

2.2.2 Epipolar Geometry. Epipolar geometry is the geometric relation that links two different view of one scene together. It is used to restrict the search space for corresponding points when matching feature points. The epipolar relationship is basically a geometric reasoning, that shows, that points in one image plane relate to lines in the other one and vice versa.

The basic layout of the problem is shown in Figure 2.1. From now on it is assumed that point P^1 in the image plane of camera C^1 is given, and the position of X and P^2 are unknown. It is intended to find a a point P^2 in the image plane

[1] R is the rotation, T the translation of the camera from world origin, transformation M moves the camera to the world origin with viewing direction along the z-axis

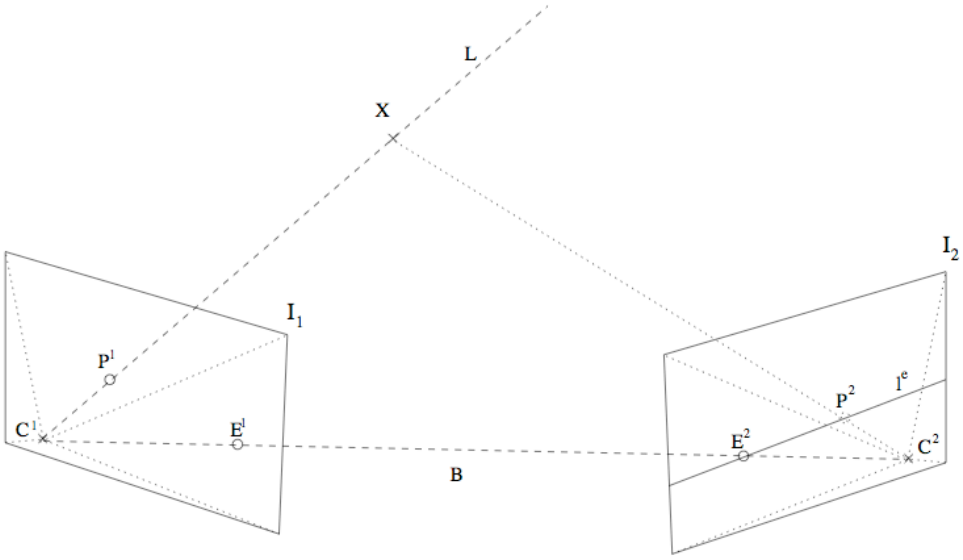


Figure 2.1: Point correspondence geometry

of camera C^2 such that it is the projection of the same point X onto I_2 as is P^1 on I_1. Since X could lie anywhere on line L, the search space is restricted to the projection of L onto I_2. The projection of line $L(l^e)$ is called the epipolar line for P^1. Because all reprojected lines L pass through the center of projection C^1, all epipolar lines intersect in one point, E^2. This point is called the epipole and is the projection of camera center C^1 onto image plane I_2. The line B joining the two camera centers is called the baseline. All epipolar planes contain the baseline and all points on the baseline project to the epipoles, and their 3D position cannot be reconstructed, since their triangulation degenerates. Since the positions of C^1 and C^2 are interchangeable, line $(P^1 - E^1)$ is an epipolar line for P^2 as well. Moreover, because P^2 could lie anywhere on the epipolar line l^e, line $(P^1 - E^1)$ is an epipolar line for all image points on l^e. Line $(P^1 - E^1)$ and l^e are corresponding epipolar lines, because they span the epipolar plane. All 3D points in this plane project to these two lines in both images, respectively. [21]

2.2.3 Depth Reconstruction. In order to reconstruct depth with regard to a scene, concepts such as triangulation and disparity need to be understood.

2.2.3.1 Triangulation. Active sensing mechanisms calculate depth based on signal propagation times of the medium. In contrast, the intrinsic passive nature of stereo vision forces its depth calculations to be calculated geometrically. Triangulation is the key principle in depth reconstruction from stereo vision. Since stereo vision does not emit any propagation signal, it needs three points to perform triangulation–a single point in the physical world together with its pictures seen from two different positions in space. [21]

Given these three rectified points (Section 2.3.2), distance can be calculated using the reconstruction function shown in equation 2.2, where z is the distance of physical point M to the cameras, and d is the so-called disparity. In order to precisely define this formula, we must examine how disparity is calculated in Section 2.2.3.2.

$$z = f(d) \tag{2.2}$$

However, before we can begin to find and extract the proper respective points, we have to account for camera imperfections. The calibration process (Section 2.3.1) can rectify such acquisition imperfections by determining the camera's internal parameters[2] and the external parameters[3]. During calibration, the fundamental matrix (F) (Section 2.2.1) is created and used to rectify (or optically correct) acquired images.

2.2.3.2 Disparity. Given a pixel m_1 of coordinates (u_1, v_1) in the first retinal plane and its corresponding pixel m_2 of coordinates (u_2, v_2) in the second retinal plane, disparity is defined as $v_2 - v_1$, or the amount of pixel shift between retinal planes. This definition assumes parallel retinal planes, which is the case for most stereo rig configurations (Figure 2.2). A disparity of 0 implies that the 3D

[2] lens focal length and center of projection

[3] translation of the center of projection and rotation of the optical axis between an image pair

point M is at infinity. If we bring point M toward the optical center C_1 along the infinite half-line $< m_1, C_1 >$, the disparity will increase from 0 to ∞.

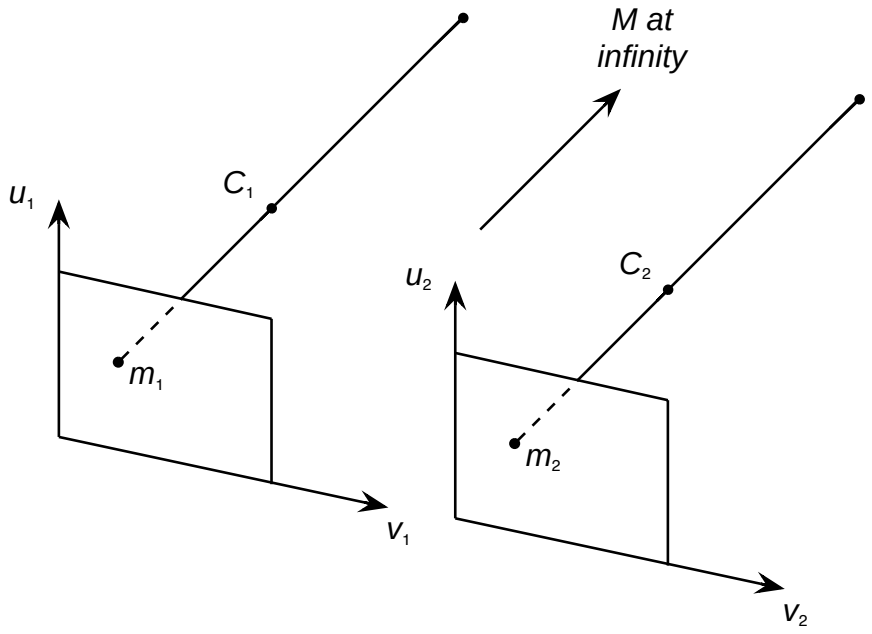


Figure 2.2: Simple disparity definition

There is a simple relationship between disparity and distance of the 3D point M, with distance measured from the two optical centers, d_{12} (Figure 2.3) [14]:

$$d = v_2 - v_1 = \frac{d_{12} f}{z} \tag{2.3}$$

This relation is readily obtained by noticing that the triangles $m_1 c_1 C_1$, $m_1 m M$, and $m_2 c_2 C_2$ are similar. Therefore, a fronto-parallel plane is a locus of points with constant disparity. It is also easy to prove that the horizontal coordinate x of M is given by:

$$x = \frac{d_{12}}{2d}(v_1 + v_2) \tag{2.4}$$

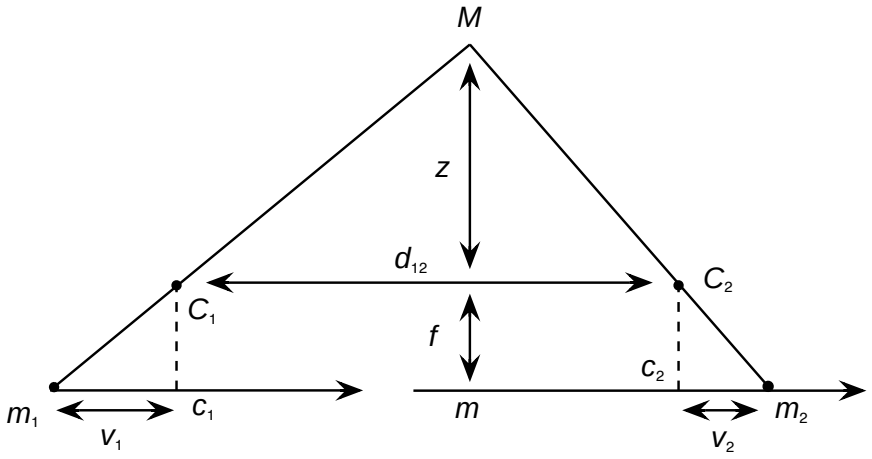


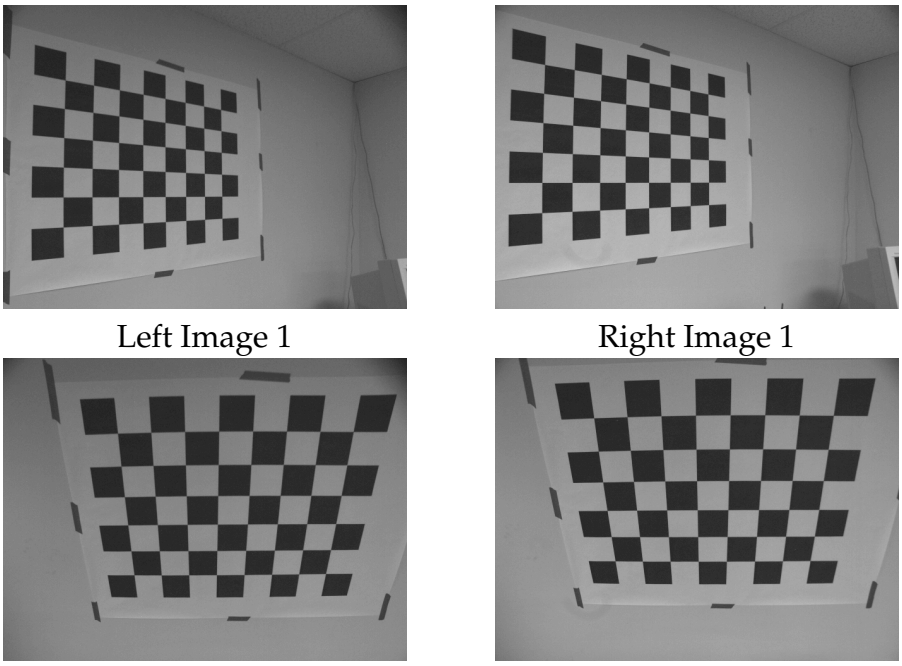
Figure 2.3: Depth and disparity relation

Equations 2.3 and 2.4 are at the basis of the techniques that aim at recovering surface properties of the objects being looked at from a disparity measurements.

2.3 Image Capture

2.3.1 Camera Calibration. Before a stereo rig can reliably collect and process images, the rig must be calibrated. Camera calibration is the task of determining the internal and external parameters of a camera system. It is a necessary step in 3D computer vision in order to extract reliable metric information from 2D images [38]. This practice is quite normal since stereo camera setups differ from an ideal setup in which the cameras are perfect pinhole imagers and are aligned precisely parallel. The divergence from ideal causes problems in the quality of the stereo match since epipolar lines (discussed in Section 2.2.2) are not horizontal. Additionally, if the camera calibration is unknown, one does not know how to interpret the stereo disparities in terms of range or distance to an object.

Common 3D computer vision camera calibration techniques [19], [16], [35], [13], [37] have built upon the early work of the photogrammetry community [9], [12]. These techniques are known as three-dimensional reference object-based



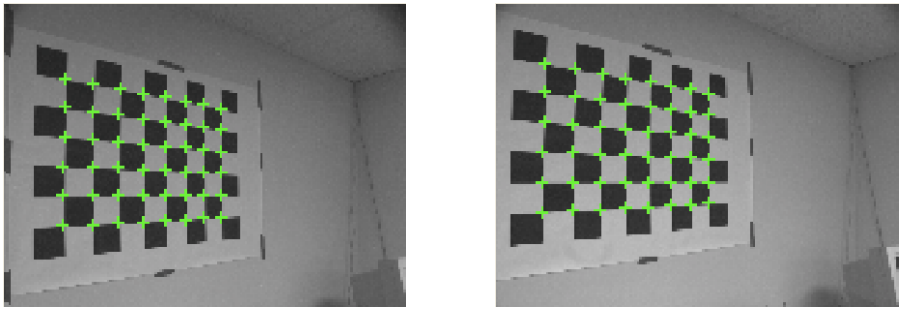
Left Image 1 Right Image 1

Left Image 2 Right Image 2

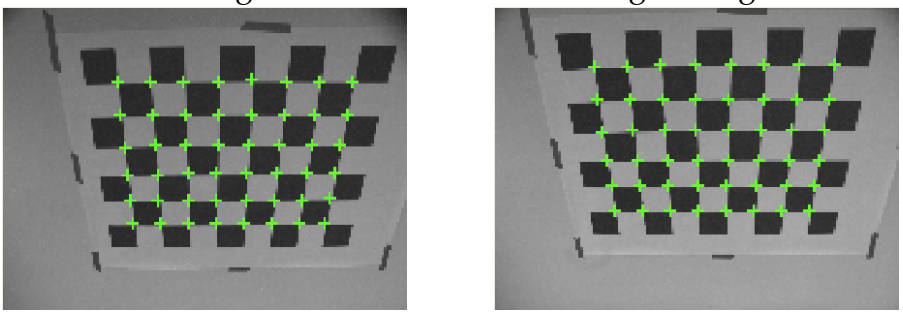
Figure 2.4: Calibration images from two different poses

calibration, where camera calibration is performed by observing an object whose geometry in 3D space is known with very good precision. The calibration object is usually captured by the stereo camera in a number of different poses (Figure 2.4) to provide an ample variety of calibration samples (generally, five to ten samples are sufficient for accurate calibration) [19]. This procedure finds model features[4] in the images, and then calculates a best-fit calibration for the stereo camera (Figure 2.5). From this, a mathematical model of the cameras internal (lens focal length and the center axis) and external parameters (distance between lenses) is created and saved to accurately interpret captured image pairs. This model is important for later reconstruction of three-dimensional objects, providing the d necessary for depth calculation in Equation 2.2.

[4]Points where two white and two black squares meet



Left Image 1 Right Image 1



Left Image 2 Right Image 2

Figure 2.5: Model features denoted by green crosshairs

2.3.2 Rectification. After successful calibration, stereo rigs can accurately capture and represent image pairs. However, before the images can be further processed, one must accomplish an alignment process called rectification. This process corrects captured images based on known camera characteristics (internal and external parameters) using known mathematics encapsulated in fundamental matrix (F).

Once these intrinsic camera anomalies are corrected via F, the captured images are accurately aligned representations of the environment that will meet epipolar constraints. This seemingly insignificant step has large search ramifications when we discuss the stereo correspondence problem in section 2.4.

Figure 2.6 illustrates an original image capture pair above its rectified counterparts. Notice the distortion differences in upper portions of the images.

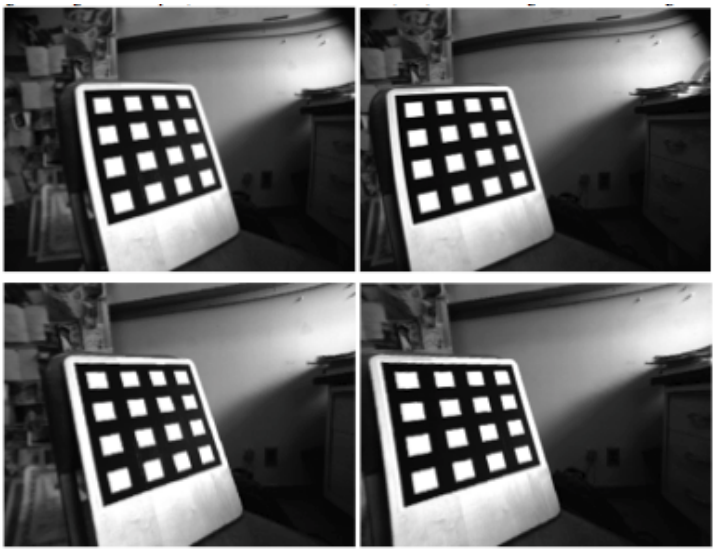


Figure 2.6: Original stereo pair (top) and rectified pair (bottom)

2.4 Stereo Correspondence Ambiguity

After camera attributes and geometrical relationships are known, the most challenging part of stereo vision processing needs to be addressed–*the stereo correspondence problem*. The stereo correspondence problem stems from the difficulty of matching tokens of one image to the same token in another image perspective. The stereo correspondence problem can be decomposed into the following steps:

1. Feature extraction
2. Feature matching

Regardless of correspondence technique, once a point match is found, we can submit our three points to the depth reconstruction stage (Section 2.2.3) for processing.

2.4.1 Pinhole Model and the Correspondence Problem. A fundamental problem to be solved in stereo vision is known as the *correspondence problem* and is shown in Figure 2.7 [14]. Two pinhole cameras form thee images m_1 and m_2 of a physical point M. As shown in this figure, we have chosen three coordinate

systems, one in each retinal plane ((u_1, v_1) and (u_2, v_2)) and one in 3D space (x, y, z), which is sometimes called the *world reference frame*. The relation between the camera reference frame and the world reference frame is given by a translation and a rotation. A point x = $[xyz]^T$ in the world reference frame is expressed with respect to the camera reference frame through a translation given by vector t = $[x_0 y_0 z_0]^T$. The distance between the two optical centers C_1 and C_2 is sometimes called the *baseline* and is used for depth reconstruction.

The most fundamental (and computationally expensive) problem to be solved in stereo vision is something known as the *correspondence problem*. For example, given a token in image 1, what is the corresponding token in image 2? Since there are, in general, many possibilities for the choice of the corresponding token in image 2, the stereo correspondence problem is said to be ambiguous, and therefore it raises a number of questions: Which tokens, which features, and which constraints can be used to reduce this ambiguity? We will discuss the tokens, the features, and the constraints that are good candidates in Section 2.5.

What we observe are two images formed in the retinal planes $\mathfrak{R}_1$ and $\mathfrak{R}_2$. Given these two images, we want to solve two problems:

1. For a point m_1 in plane 1, decide which point m_2 it corresponds to. *Correspond* means that they are the images of the same physical point M. This is known as the *correspondence problem*.
2. Given m_1 and m_2, compute the 3D coordinates of M in the world reference frame. This is the *reconstruction problem*.

2.4.2 Computational Expense. Once we have a calibrated camera we can take advantage of the epipolar constraint to determine depth or disparity information quickly by reducing vertical scanning.

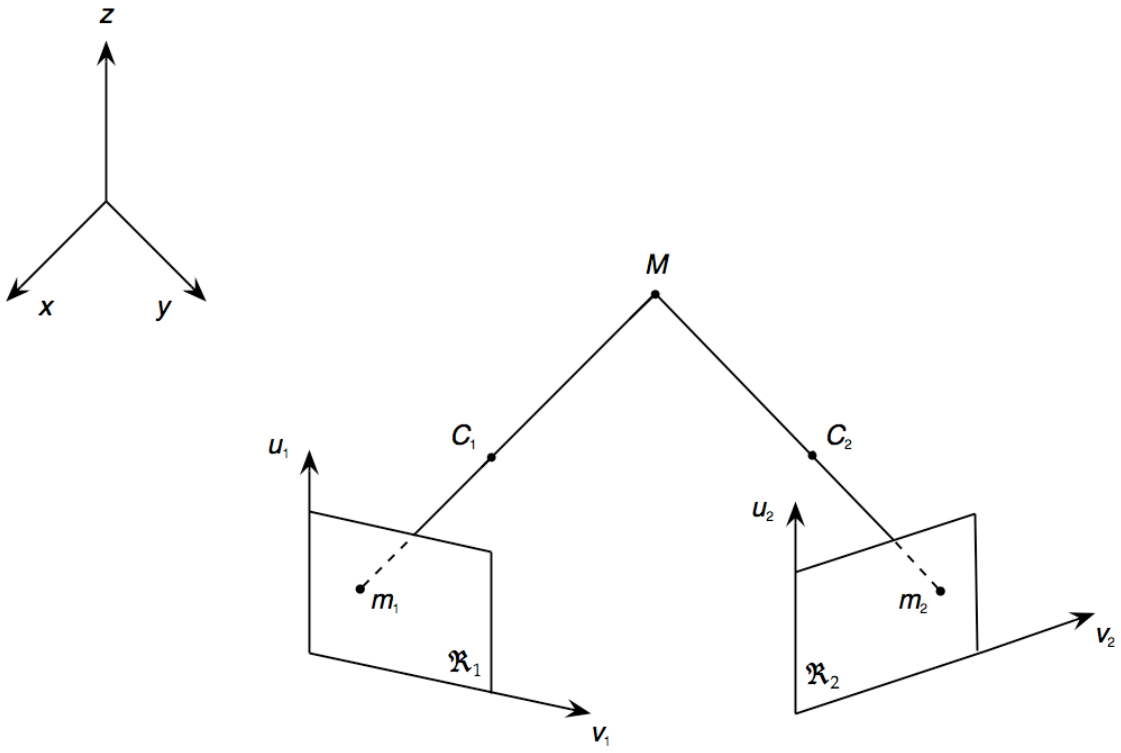


Figure 2.7: Steroscopic camera pinhole geometry

Correlating pixels between pictures is the most computationally expensive part of stereo processing because of the density of the data. There is a trade-off between the quality of the disparity map and the processing time. When processing time goes down, generally the quality of your depth information does as well and vice versa. In Section 2.5 we will examine two of the most common stereo correlation algorithms along with a recent new approach.

2.5 Correlation Techniques

2.5.1 Feature Extraction. To further reduce the ambiguity of correspondence, tokens (or relatively unique features) must be reliably extracted from the images. Token candidates often include general pixel characterization, the edge pixel, and image regions.

In general pixel characterization pixels m_1 and m_2 match and their features must also match. Many of the original techniques for computing stereo correspondences are based on the idea of correlating the left and right intensity images of a stereo pair. This works well when the reflectance functions of objects are close to Lambertian [5], but otherwise it is not. [14]

The next simplest token is the edge pixel. Line segments and curves can have both geometric and intensity-based features attached to them such as length, orientation, curvature, and average contrast across them. Except for the occluded case, edges are a very reliable source of information to guide the stereo matching process [14]. Edges are commonly detected with operators such such as Sobel and Canny.

Lastly, image regions can be used as tokens to be matched [14]. Depending on how these regions are extracted, their shapes and the intensity-based feature attached to them may or may not be invariant by perspective transformation.

2.5.2 Feature Matching. Once an established token has been agreed upon for extraction it needs to be located and matched between images (a.k.a. *the correspondence problem*). Approaches to the correspondence problem can be broadly classified into two categories: intensity-based matching techniques and feature-based matching techniques. In the first category, the matching process is applied directly to the intensity profiles of the two images, while in the second, features are first extracted from the images and the matching process is applied to the features.

2.5.2.1 Intensity-based. Matching these regions works by using a fixed, rectangular window around the pixels of interest in the first image. The window is correlated with a second window, which is moved over all epipolar

[5] A completely matte surface where the reflected light is the same in all directions

possible positions in the second image. The possible positions are defined by the minimal allowed distance between the camera and the object, which gives the maximum disparity. The position where correlation has the highest value determines the pixel in the second image that corresponds to the pixel of interest. Bigger correlation windows increase the reliability by averaging over a bigger area, thus reducing the effect of noise. However, if the correlation window overlaps a depth discontinuity, then a part of the window will affect the result arbitrarily. Generally, the choice of the correlation window size is a tradeoff between increasing reliability in areas with constant depth and decreasing errors in areas where depth changes [3]. Although intensity-based matching is easy to implement and provides dense disparity maps, it is computationally heavy, meaning that the amount of information to evaluate is infeasible to do in real-time applications.

2.5.2.2 Feature-based. Instead of using the fixed windows seen in the intensity-based technique, feature-based matching uses numerical and systemic properties of features available from feature descriptors. Examples of feature descriptors are edges, corners, line segments, curve segments, circles, ellipses, and regions (blobs or polygons). It uses a measure of the distance between feature descriptors and narrows the possible matches by geometric (epipolar constraint) and analytic (uniqueness and continuity constraint) analysis. Feature-based matching is suitable when a priori information is reliably known about the environment and/or target object. It generates a sparse set of points and is insensitive to illumination changes and highlights. [3]

2.5.3 Birchfield Algorithm. To expedite processing, in lieu of intensity- or feature-based matching we use the Birchfield disparity algorithm. Birchfield and Tomasi [5] presented this novel pixel-by-pixel matching algorithm for detecting depth discontinuities from a stereo pair of images. Their approach inverts the tra-

ditional role of a stereo algorithm because, to compute disparity more accurately, they compute a rough disparity map in order to get crisp discontinuities.

Instead of deriving a maximum a posteriori (MAP) cost function from a Bayesian formulation, they propose a simple cost function justified solely on empirical evidence. The cost of a match sequence is defined by a constant penalty for each occlusion (k_{occ}), a constant reward for each match (k_r), and a sum of the dissimilarities between the matched pixels:

$$\gamma(M) = N_{occ}k_{occ} - N_m k_r + \sum_{i=1}^{N_m} d(x_i, y_i), \tag{2.5}$$

where $d(x_1, y_i)$ is the dissimilarity between pixels x_1 and y_i and N_{occ} and N_m are the number of occlusions[6] and matches, respectively, in M.

Technically, k_{occ} is interpreted as the amount of evidence (in terms of mismatched pixel intensities) that is necessary to declare a change in disparity, while k_r is interpreted as the maximum amount of pixel dissimilarity that is generally expected between two matching pixels. The term $d(x_1, y_i)$ measures how unlikely it is that the intensities $I_L(x_i)$ and $I_R(y_i)$ are images of the same scene point. Instead of calculating this dissimilarity by taking the difference between $I_L(x_i)$ and $I_R(y_i)$, as is often done, they use a computationally efficient, linearly interpolated intensity functions surrounding two pixels to measure their dissimilarity, in a method that is probably insensitive to sampling.

Like several previous algorithms [4], [10], [15], and [20], their algorithm uses a form of dynamic programming to match epipolar scanlines[7] independently, detecting occlusions and depth discontinuities simultaneously with a disparity map.

[6]Unmatched pixels are occluded, and a subsequence of adjacent occluded pixels that is bordered by two non-occluded pixels

[7]Corresponding image pair rows from the beginning column position to the end

Then a postprocessing step propagates information between scanlines to refine the disparity map and the depth discontinuities using pixels that are quantized into one of three disjoint categories: *slightly reliable*, *moderately reliable*, or *highly reliable*. One can think of moderately reliable pixels as being aggressive, changing the values of their neighbors, while slightly reliable pixels are defensive, resisting change. A moderately reliable pixel propagates along its column, changing the disparities of the pixels it encounters, until it reaches either intensity variation or a slightly reliable region with a lower disparity. Regions with a higher disparity are overrun regardless of their reliability, because reliability is not a good indication that the disparities are correct when the background has little intensity variation. The only distinction between moderately and highly reliable pixels is that the former are not allowed to overrun their neighbors if the change in disparity is just one pixel.

Throughout the Birchfield process, they use neither windows nor preprocessing of the intensities, thus matching the individual pixels on one image with the pixels in the other image. Their approach contains three important novelties. First, the image sampling problem is overcome by using a measure of pixel dissimilarity that is insensitive to sampling. Secondly, the algorithm handles large untextured regions which present a challenge to many existing stereo algorithms. Finally, unlikely search nodes are pruned to reduce dramatically the running time of dynamic programming. Their algorithm gives the ability to process a stereo image at 1.5 microseconds per pixel per disparity. This makes it a very good candidate for real-time implementation.

2.6 The Robotic Mapping Problem

The problem of robotic mapping is that of acquiring a spatial model of a robot's environment. Maps are commonly used for robot navigation [6, 23], path/motion planning and reference. To acquire a map, a robot must possess

sensors that enable it to perceive the outside world. Sensors commonly used for this task include cameras, range finders (sonar, laser, and infrared), radar, tactile sensors and GPS. Key challenges in robotic mapping include noise, high dimensionality, the correspondence/data association problem, dynamic environments, and robot exploration.

2.7 Vision Mapping

Existing approaches to build 3D maps of the environment that use laser range finder sensors have evolved towards obtaining more and more compact models made up of planar surfaces [24, 25, 33]. In such approaches, the problem of simultaneously computing the map and robot poses is formulated in terms of maximizing a log-likelihood function and then using a gradient descent algorithm to obtain, at least, a local maximum. [34] [2] respectively, developed manual and autonomous platforms using active sensor methods that determine ego-motion to map subterranean voids.

In other approaches, range finders augment cameras. For instance, in [11] a range finder is combined with eight CCD cameras to compute a mesh model of the environment from a large number of overlapped 3D images. In [1, 17], a range scanner is combined with a single camera through fusing the results of 2D and 3D matching and planar segmentation processes with a best next view for modelling the environment.

In our research, we focus on a third group of approaches that only use stereo cameras as 3D sensors. Figure 2.8 illustrates implementations that use either strategically placed objects (a,b) or project a unique pattern over the scene [22] in (c) to artificially give texture, thus increasing the local discrimiation at each pixel and facilitating matches. In mobile platforms such as [29], 3D information is condensed into 2D maps that are represented by a 2D occupancy grid, because its 3D generalization [28] is neither computationally feasible nor scalable. Their

system autonomously explores *relatively textured* environments while building occupancy grid maps using a triclops stereo vision module at a scaled 2.5 Hz at 320 x 240 resolution. Some of the latest work in vision mapping by [31] generates high quality 2D maps through a fast ego-motion/action-estimation algorithm at a rate of 15-20 Hz. However, this computationally expensive implementation would be a poor choice for a real-time implementation.

(a) (b) (c)

Figure 2.8: Artificially textured scenes

2.8 Contribution of this Study to the Literature

Our goal is to further mobile vision mapping research [28, 29, 31] by implementing a stereo vision system, using the planar assumption, that excels in low-texture environments and builds 2D occupancy grid map information in *real-time*.

III. Methodology

3.1 Introduction

A large portion of our research involved the efficient integration of several hardware and software components (Figure 1.1). More specifically, we synthesized robot and camera hardware with commercial off-the-shelf software, open source software, and newly developed software with optimization techniques to develop an end-to-end real-time mapping solution. The overarching goal of our solution is to obtain *quick object detection* and to *accurately represent them using concise reconstruction information*. The early part of this chapter describes the key specifications of the multiple hardware and software components that were fused into our single implementation. The latter explains our software architecture model, its concepts and inner workings.

3.2 Hardware

3.2.1 Robot. Our mobile robot in Figure 3.1, *Bilbo*, is a Pioneer P2-AT8 designed and manufactured by ActivMedia Robotics, which is the all terrain AT robot in their Pioneer series of robots. *Bilbo* is equipped with a 1.6 GHz Pentium Mobile PC and 1GB of Random Access Memory (RAM) running the Windows XP Professional operating system as its onboard processing. *Bilbo* supports wireless communication to a host computer via a Orinoco 802.11b PC card.

3.2.2 Stereo Camera. Although *Bilbo* has a built-in sonar ring, an externally fixed mount stereo camera will act as his only means of environmental perception. The Videre Design STH-MDCS-C stereo camera is a compact, low-power digital stereo head with an IEEE 1394 (Firewire) digital interface. It consists of two 1.3 megapixel, progressive scan CMOS imagers mounted in a rigid body, and a 1394 peripheral module, joined in an integral unit. The imagers are 1/2" format, with a

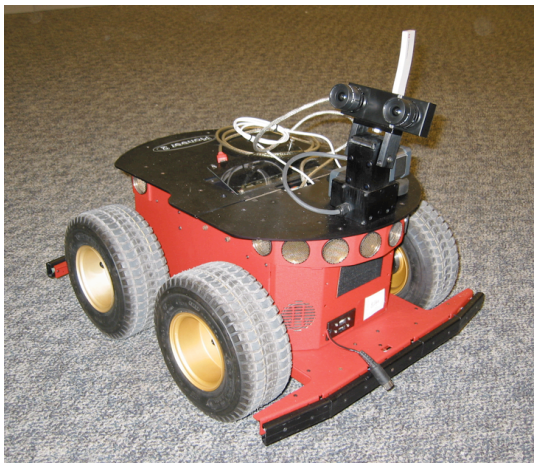


Figure 3.1: *Bilbo* the robot

resolution of 1280 horizontal by 1024 vertical pixels and have excellent dynamic range, sensitivity, anti-blooming, and noise characteristics. Our image capture has a resolution of 320 x 240 pixels. The camera has a 6mm focal length, a 90mm baseline and is mounted 18.5" above ground level and 8.5" offset from *Bilbo's* epicenter as depicted in Figure 3.1.

3.3 Off-the-Shelf Software

3.3.1 SVS. Small Vision System (SVS) by SRI International provides an interface to the STH-MDCS-C stereo camera head. SVS includes calibration software tools, stereo processing libraries, driver and capture application software, interface to MatLab, and application and display software. We only use SVS for calibration, initialization, capture, and rectification. We substitute an open source stereo processing module in lieu of SVS's stereo processing libraries to achieve better disparity results.

3.3.2 ARIA. ActivMedia Robotics' Interface for Applications (ARIA) is a powerful, object-oriented client-side interface for ActivMedia mobile robot servers written entirely in the object-oriented paradigm. Client applications developed with ARIA dynamically control your robot server's velocity, heading, relative

heading, and many other navigation settings, as well as manage the detailed odometry and operating conditions reported back to your clients from the robot server. *Bilbo's* pose (x, y, Θ) information comes from ARIA and is used to translate stereo vision's local coordinate scheme to a global one.

3.4 Open Source Software

3.4.1 OpenCV. As mentioned above, a portion of our vision processing is done from an open source, specifically Intel's Open Source Computer Vision Library (OpenCV) [18]. OpenCV is a library of computer code intended for use, incorporation and modification by researchers, commercial developers, government and camera vendors. The library is mainly aimed at real-time computer vision. Intel's overarching philosophy on maintaining such a library is to aid commercial users of computer vision in human-computer interface, robotics, monitoring, biometrics and security by providing a free and open infrastructure where the distributed efforts of the vision community can be consolidated and performance optimized. We chose to integrate the stereo correspondence module of OpenCV into our work because of its efficiency and overall robustness.

3.4.1.1 Stereo Correspondence Algorithm. We use the Birchfield algorithm as implemented in OpenCV [5] to perform stereo correspondence matching because of its real-time depth discontinuity information capabilities. Like several previous algorithms, it uses a form of dynamic programming to match epipolar scanlines independently, detecting occlusions and depth discontinuities simultaneously with a disparity map. Unlike other stereo processing algorithms that use windows or intensities, the Birchfield algorithm matches on an individual pixel basis. Information is then propagated between scanlines to refine the disparity map and the depth continuities. From this novel architecture emerge its strengths:

- It overcomes the image sampling problem by using a measure of pixel dissimilarity that is insensitive to sampling
- It handles large untextured regions which present a challenge to many existing stereo algorithms
- Dramatic reduction of dynamic programming running time

The structure of our correspondence call using OpenCV is described in table 3.1. Where, left and right images must be a rectified grayscale 8-bit image, the only supported disparity algorithm is Birchfield, and maximum disparity is carefully chosen since larger values greatly affect performance.

Table 3.1: *cvFindStereoCorrespondence* signature decomposition

Argument	**Semantics**
L	Left image of a stereo pair
R	Right image of a stereo pair
A	Algorithm used to find disparity
R_{depth}	Resulting depth image
D	Maximum possible disparity
O	Constant occlusion penalty
M_{reward}	Constant match reward
H	Highly reliable region
M	Moderately reliable region
S	Slightly reliable region

3.5 Approach Description

The critical point in our research lies in properly interpreting disparity results from the Birchfield algorithm in very low texture or highly repetitive textured areas as to extract accurate object points. When the Birchfield algorithm processes these

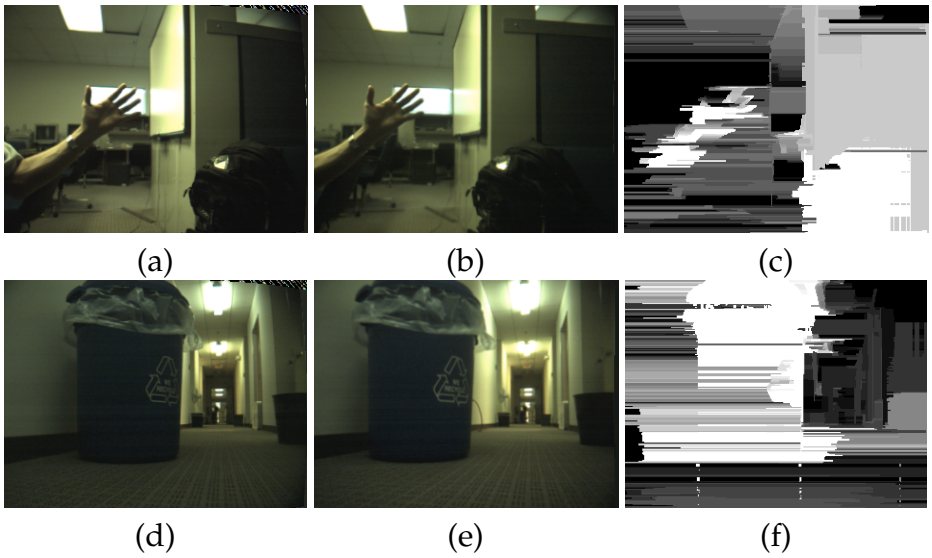


Figure 3.2: Samples of Birchfield-generated disparity data, where (c) is the disparity from the (a) and (b) stereo pair, and (f) is the disparity from the (d) and (e) stereo pair. Lighter intensities indicate closer objects.

types of problematic environments there are pixel mismatches that lead to noisy data[1] (Figure 3.2). Analysis of the disparity samples yields two obvious problems:

1. Errant horizontal lines or spikes occur with high frequently
2. Contiguous objects tend to become disjoint or broken apart

Our software framework addresses each of these problems separately through efficient mechanisms that remove spurious data while leaving accurate data in tact. After which, we extract data points for reconstruction. Our optimization and data manipulation approach can be decomposed into eight sequential steps:

1. Horizontal Spike Smoothing
2. Melding and Masking
3. Object Identification and Labeling

[1]It is important to note that the relatively small amount of noise the Birchfield algorithm generates is overshadowed by it strengths of speed and overall good low texture matching ability as compared to other matching algorithms.

4. Rectangle fitting
5. Disparity Intensity to Distance Mapping
6. 3D-to-2D Scaling
7. Translation and Rotation
8. Interior Point Calculation

The overarching goal of our data transformations involved implementing efficient mechanisms aimed at sifting though dense data and extracting important environmental data based upon a close-proximity and planar assumptions. Close-proximity refers to the idea that objects closer to the robot pose more of an immediate obstruction than those in the distance and, therefore, should be handled first. The planar assumption is our data simplification technique that sees the world a series of planes to expedite map reconstruction and alleviate a significant portion of vision computational expenses.

3.6 Optimization Techniques and Data Manipulation

3.6.1 Horizontal Spike Smoothing (HSS). Our first step involves eliminating errant horizontal lines. HSS is an optimization technique we developed to smooth correspondence match anomalies. When the Birchfield correspondence algorithm mismatches problematic pixels, whether it be due to low texture or other ambiguity, it causes a phenomenon we have termed horizontal spiking (Figure 3.2 rightmost pictures) to occur. It is very important to remove these misrepresentative spikes before the rectangle fitting step (Section 3.6.4). Furthermore, we have made the assumption that horizontal spikes of small pixel height are insignificant enough to disregard while maintaining the integrity of the reconstruction, since we aim to map an area's major components.

To optimally detect and correct horizontal spikes in array, A, we have developed a fast, efficient algorithm (Figure A.1) to remove spikes of a user-defined

value, s. The spike value indicates the vertical thickness that the user deems insignificant. Our algorithm will take the value s and remove all horizontal spikes with value s or less and replace them with the empty value of 0. The algorithm uses a vertical scan of each column in the width of the image. This is much more efficient than a horizontal scan because it eliminates cell revisiting (Figure 3.3). With begin (b) and end (e) markers identifying the upper and lower bounds of contiguous like-intensity cells, count can be computed by taking e - b + 1. Count, in turn, can be compared to s and appropriately retained or erased. This procedure is computationally efficient since we traverse the matrix optimally while removing horizontal spikes of s value or less in O($x * y$) time, where x and y are the height and width of the matrix, respectively.

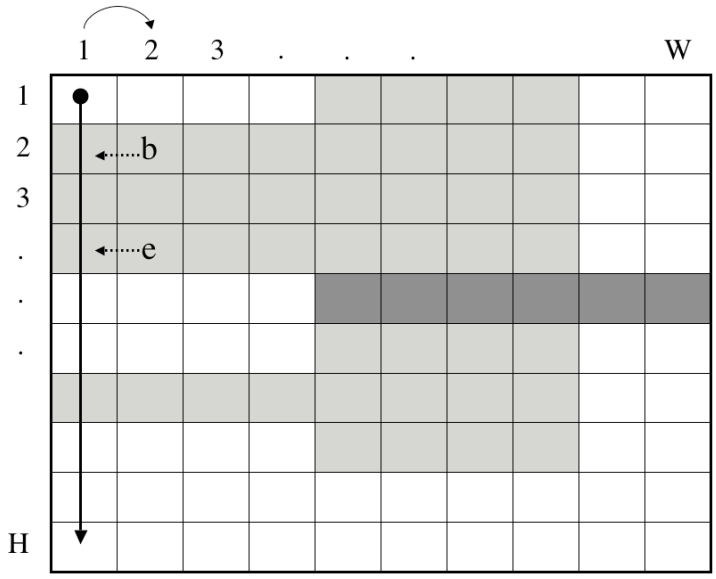


Figure 3.3: Horizontal Spike Smoothing concept

Figure 3.4 shows an original image reference, the Birchfield-generated disparity image, a representation of the removed horizontal spike data, and the final spike-removed disparity image.

The vertical spikes are left intact because they have no bearing on the final map reconstruction since we are taking vertical slices and reconstructing them in

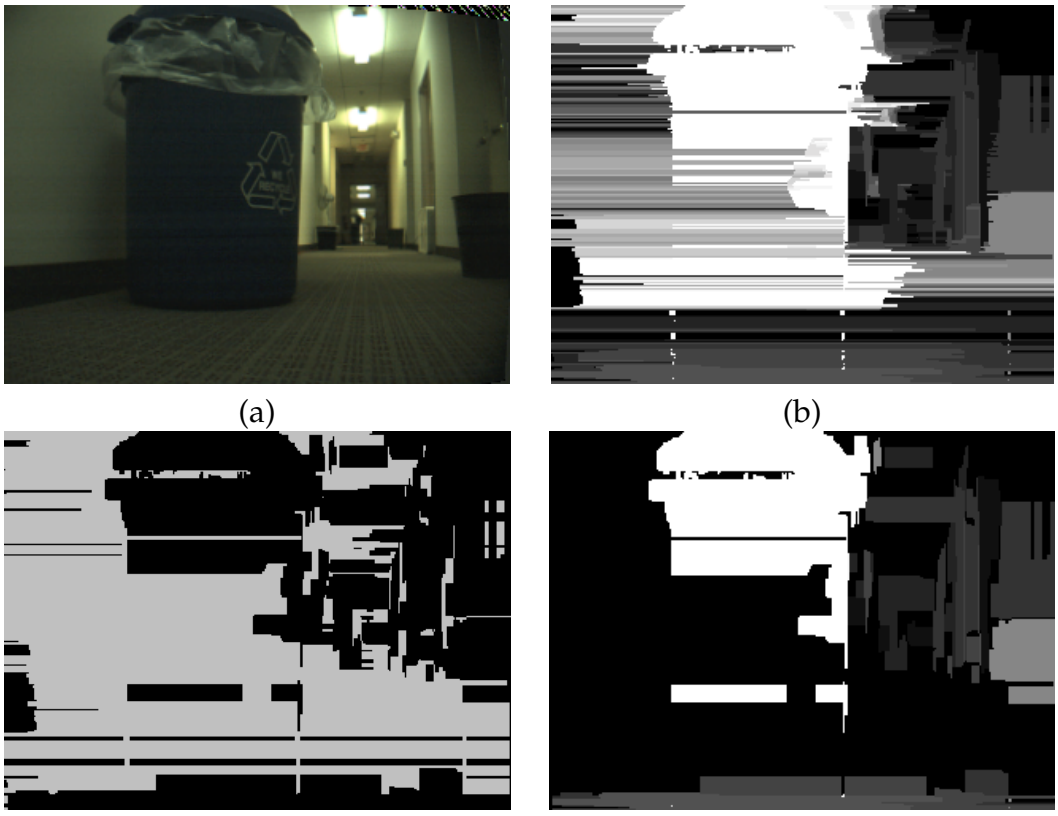


Figure 3.4: Horizontal spike removal progression: Original camera reference image (a), is processed through the Birchfield algorithm to generate the disparity image (b). Next, the HSS algorithm removes horizontal spikes from the disparity image highlighted in gray in (c), to produce the final product (d).

a 2D environment. If this implementation is expanded to 3D map reconstruction, vertical spike removal would help represent the reconstructed environment more accurately. Additionally, it is important to note that based upon our tests vertical spiking is relatively rare with regard to the stereo correspondence mismatching and, therefore, poses minimal problems.

3.6.2 Melding and Masking. After applying the HSS algorithm we have a better representation of the original scene but still have some remaining side effects from horizontal spiking. Chiefly, contiguous objects have become disjoint or broken apart because of match anomaly propagation. It is desirable to keep

original objects as close to their original representation to more accurately reconstruct the object. To rectify object disjointness we implement an algorithm (Figure A.2) that will meld like objects together based on some scalable pixel proximity value, p and like-intensity variance, v.

Our melding concept (Figure 3.5) involves horizontally traversing the array, from row_{p+1} to $row_{HEIGHT-p}$, while examining above and below the current pixel by an amount of p pixels. If we find v above *and* below the current pixel within p pixels, the current pixel (and all others between upper and lower bound) are assigned the largest intensity value found above or below, thus, melding it to the object. Care must be taken not to use a p value to large or you will risk melting two unrelated objects together. Through testing, we found that five to 10 pixels yielded best melding results for the environments we evaluated.

Although unrelated, we integrate masking into the melding algorithm for performance reasons–specifically to minimize array traversal. Masking is unrelated to pixel mismatch anomalies, as well. We chose to mask more distant information in an attempt to simplify the image to speed processing and focus in on closer proximity items that pose more of a threat to the robot in terms of obstruction or other danger.

Figure 3.6 shows a post-HSS image, a representation of the reduced pixels from melding and masking, and a the result. In 30ms we meld and mask the HSS-produced image's 76,800 pixels (320 x 240). In this example, 7,869 pixels or 10.2 percent of the pixels were modified–either removed by filtering or added from melding.

3.6.3 Object Identification and Labeling. After the HSS and masking/melding steps, we are left with a relatively good representation of the environment and, thus, can proceed to object identification and labeling. Being able to delineate

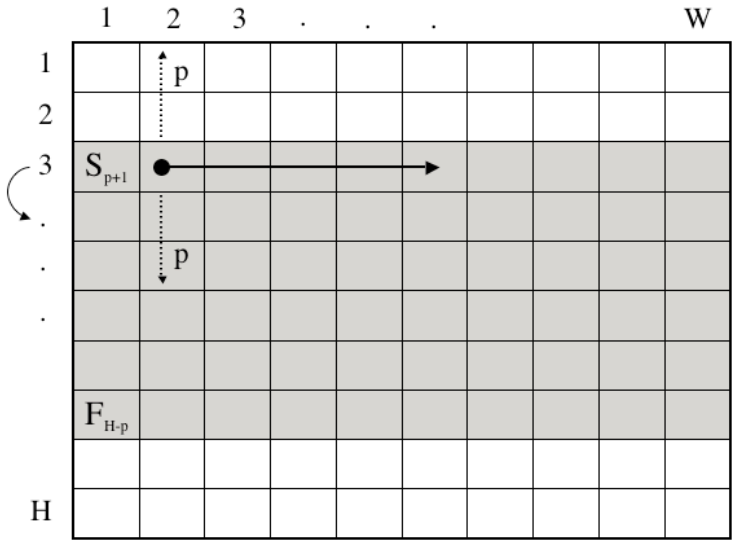


Figure 3.5: Melding concept

between objects is fundamental to extracting them successfully. Each object inherently has two descriptors:

1. A disparity intensity that is a function of its depth.
2. Dimensional characteristics that help define its planar shape.

The purpose of object identification and labeling is the latter. In order to find and uniquely mark (or fill) each object in an image we examined existing flood fill techniques.

We turned to a more complex flood fill algorithm (stack-based flood fill) that manages memory more efficiently. Stack-based flood fill is a quick, robust way to fill an object given an internal cell to the object. Figure 3.7 shows the steps involved in a stack-based flood fill:

a. Given an inner seed
b. Fill the scan line
c. Find rightmost pixel above and below and push on stack

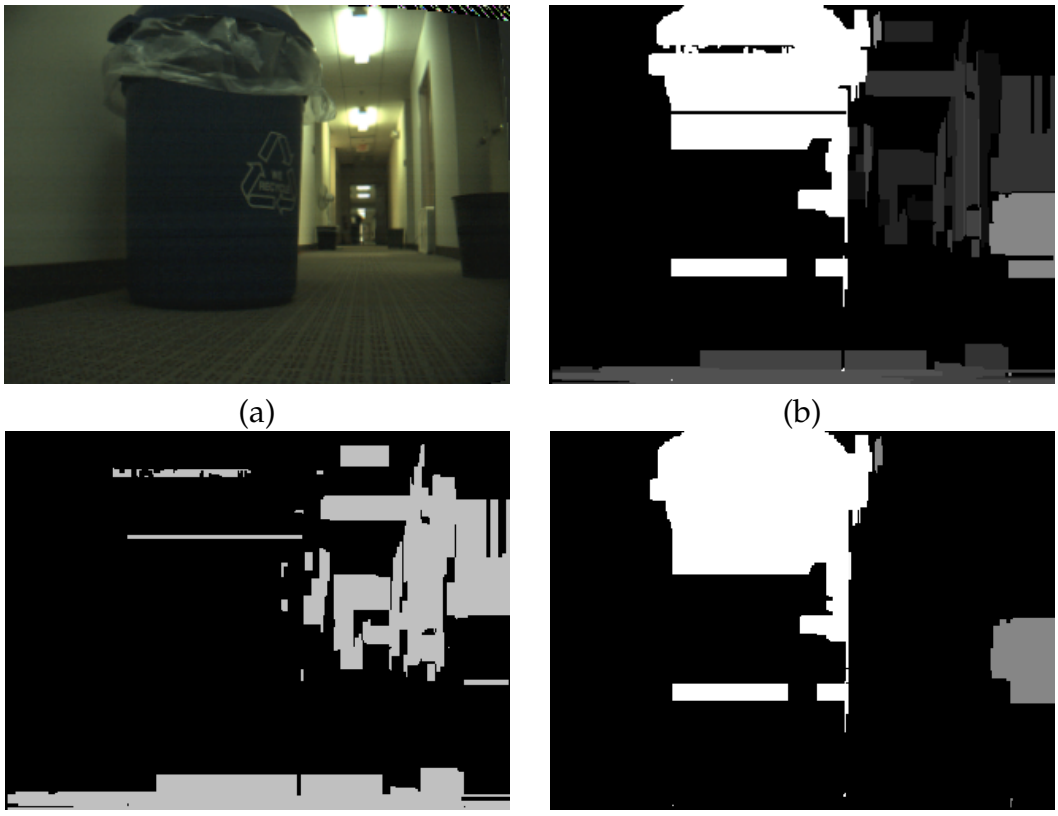


Figure 3.6: Melding/Masking progression: Original camera reference image (a). HSS product (b) is processed through the melding and masking algorithm modifying pixels in (c). The final product is in (d).

The reason each object has to be uniquely filled serves two purposes. First, it allows one to algorithmically iterate over the image and gain information about each object. We specifically track the number of objects and each object's maximum and minimum Cartesian x- and y-coordinates for planar extraction (Section 3.6.4). Secondly, unique filling provides an extendable framework where other methods of object extraction could be done. For example, one could use a convex hull algorithm in to more precisely extract the shape of the object. Without unique object identification, such an algorithm cannot perform optimally.

As mentioned earlier, our image array A has dimensions of 320 x 240. In order to be optimally efficient, we represent each cell in A as a byte. This optimization

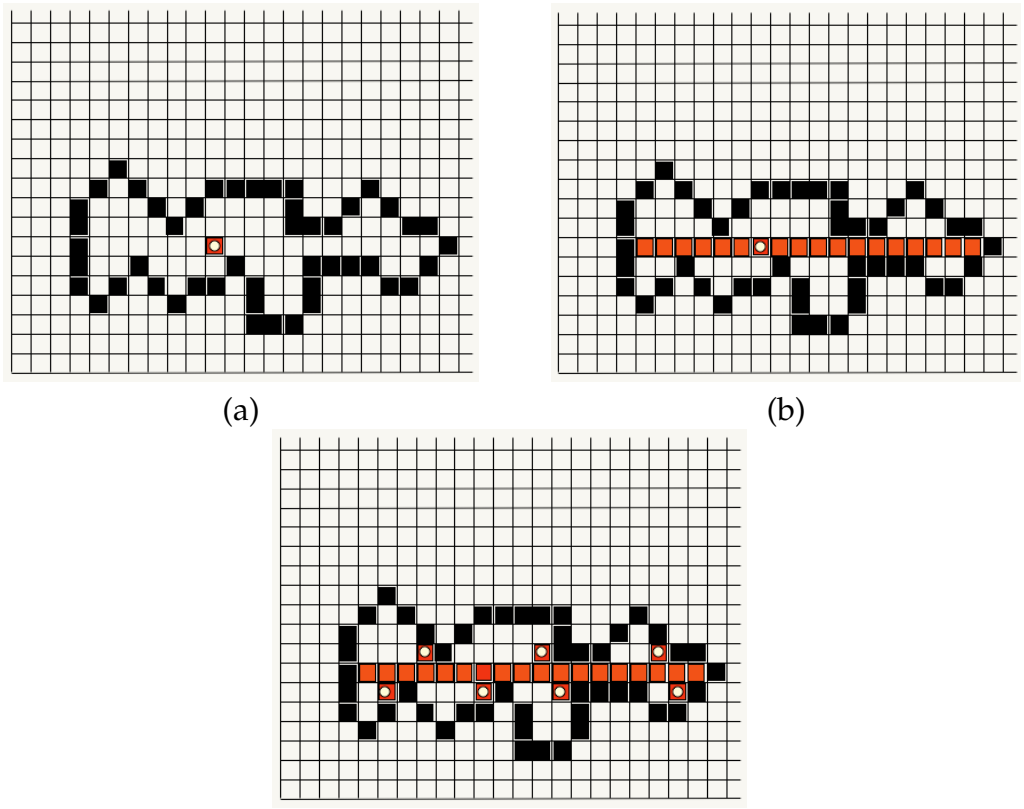


Figure 3.7: Stack-based flood fill algorithm

strategy minimizes memory storage requirements and affords us an ample amount of unique values (2^8). The unique values are separated into three semantically delineated partitions as seen in Table 3.2. The disadvantage of using only a byte of storage per pixel is the inability to track very large numbers of objects, which is not pertinent to our implementation.

Table 3.2: Byte values in image array A

Value	**Semantics**	**Total**
0	Empty	1
1-20	Disparity Intensity	20
21-255	Object Identifier	235

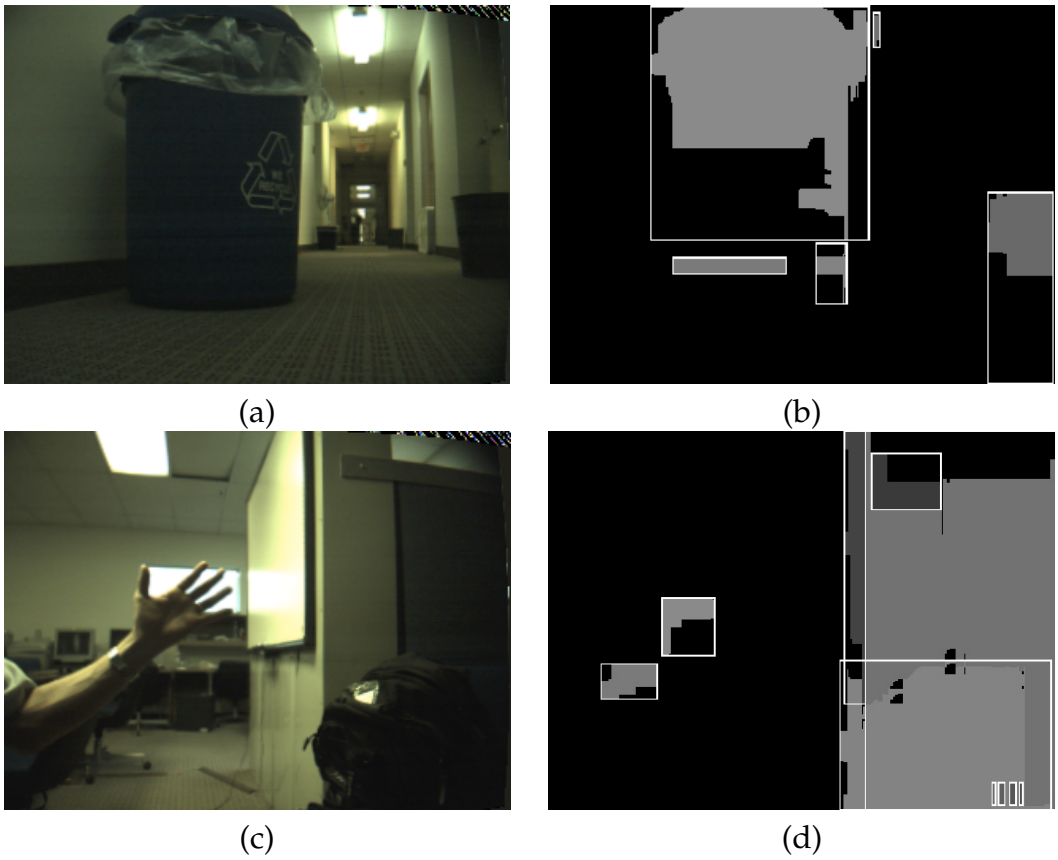


Figure 3.8: Rectangle fitting. Reference image 1 in (a) and rectangle fitted counterpart (b). Reference image 2 in (c) and rectangle fitted counterpart (d).

3.6.4 Rectangle Fitting. Using our planar assumption, the process of conversion from three dimensional data to two dimensional objects is sped by representing objects as rectangular planes for reconstruction purposes. Representing an object by two coordinates is computationally cheap with regard to processing and storage. From piggy-backing smartly on the stack-based flood fill algorithm, we collected the minimum and maximum Cartesian x- and y-coordinates with very little computational overhead.

From these four coordinates we can quickly and easily reconstruct the object. One way this can be accomplished is by connecting the coordinates $(minX, maxY)$, $(minX, minY)$, $(maxX, minY)$, and $(maxX, maxY)$ counterclockwise. Examples of actual rectangle fitting is shown in Figure 3.8.

(a) (b)

Figure 3.9: Horizontal Spike Smoothed (HSS) image (a) with 10 boxes compared to non-HSS image (b) with 30+ boxes.

Performing Horizontal Spike Smoothing is critical with regard to rectangle fitting, since using horizontal spike data would erroneously grow the object to largely misrepresentative proportions. Additionally, the number of objects decreases drastically if HSS is applied before rectangle fitting. In Figure 3.9 both of these principles can be clearly seen.

Although very efficient, rectangle fitting does have some rather obvious limitations. First, since all physical objects are not rectangular, using rectangle fitting would tend to represent objects larger than they really appear (sometimes grossly) and would not be recommended for applications where object granularity is vital. Second, since planes are extracted at certain intervals from a perpendicular perspective, objects lose some depth characteristics. Our research has recognized and accepted these limitations in order to expedite processing capabilities.

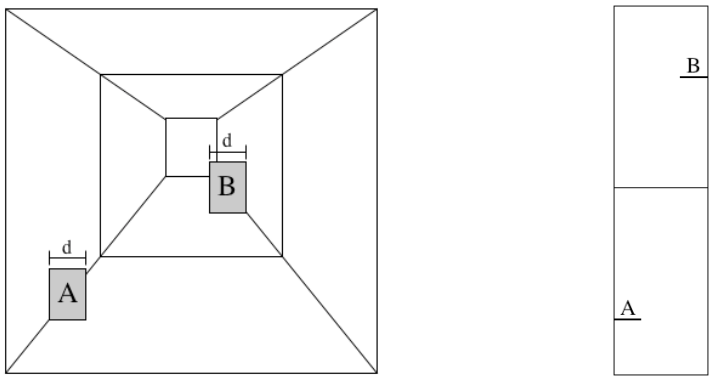
3.6.5 Disparity Intensity to Distance Mapping. As previously discussed in Section 2.3, given disparity d, baseline b, and corresponding pixels, m_1 and m_2 one can calculate the precise distance to that pixel. However, at this point we lack one vital piece of information–corresponding pixels. The Birchfield algorithm attempted to find all corresponding pixels but output noisy disparity data that cannot be effectively back-translated to the original image pair given real-time

Table 3.3: Disparity to distance mapping model

Birchfield Disparity Value	Normalized Byte Value	Distance in Feet
1	12	10.6
2	25	10.2
3	38	9.8
4	51	9.4
5	63	9
6	76	8.6
7	89	8.2
8	102	7.8
9	114	7.4
10	127	7
11	140	6.6
12	153	6.2
13	165	5.8
14	178	5.4
15	191	5
16	204	4.6
17	216	4.2
18	229	3.8
19	242	3.4
20	255	3

processing requirements. Trying to find what pixels the disparity image refers to takes us back to the correspondence problem previously discussed in Section 2.4.1.

Alternatively, we can get good estimated distance information if we act on other pieces of a priori information. That is, since disparity values are related to distance, we built a depth model that can be used to map distance information onto disparity intensities with good accuracy. To generate this mapping, we executed the Birchfield algorithm against an incrementally-increasing distanced object to gather empirical data in a controlled environment to construct a mapping from disparity to distance for each disparity level. Our final distance mapping model is depicted in Table 3.3 and is, generally, accurate within +/- 6" within six feet.



(a) (b)

Figure 3.10: Non-scaled 3D-to-2D rotation. (a) shows a proper 3D perspective, whereas (b) shows an erroneous rotated bird's eye view.

Once this data is mapped to the disparity data, we need to scale each object appropriately if it is to be accurately modeled in a 2D environment.

3.6.6 3D-to-2D Scaling. Scaling our 3D perspective to a 2D perspective is vital to proper proportionality as seen in Figure 3.10(a). In this figure, we are looking down a hallway and we see two objects, A and B. In reality, object A is closer and object B is more distant. However, as Figure 3.10(b) illustrates, without scaling both objects would appear to be the same size if the scene were rotated such that we were looking down at hallway because the difference between the maximum and minimum x-coordinates, d, remains the same as in the 3D perspective. Since our mapping output is displayed in two dimensional grid we properly transform the data with object reconstruction accuracy in mind.

The scaling computation is accomplished using known camera geometry in Figure 3.11 along with our disparity to distance mapping using formulae 3.1 through 3.6, where f is the focal length in millimeters, WIDTH is the image in pixels, and z is the left and right lens overlap at distance, d. Once we know the ratio of millimeters per pixel, we can convert our known pixel range to an appropriate scale. From this, we get accurate object representation in the 3D-to-2D transformation and can proceed to the translation and rotation calculation.

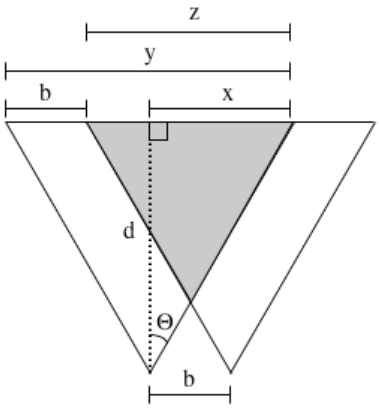


Figure 3.11: Stereo camera geometry

$$HFOV = 2 \times \arctan(\frac{3.84}{f}) \tag{3.1}$$

$$\Theta = \frac{HFOV}{2} \tag{3.2}$$

$$x = d \times \tan\Theta \tag{3.3}$$

$$y = 2x \tag{3.4}$$

$$z = y - b \tag{3.5}$$

$$mmPerPixel = \frac{z}{WIDTH} \tag{3.6}$$

3.6.7 Translation and Rotation. At this stage of processing we have good environmental reconstruction data on each object given a static robot. However, we are collecting data while *Bilbo* is moving, so we need to interpret our local coordinates, in light of his movements, into global coordinates where *Bilbo* exists.

Initially, we rotate the camera coordinates to take into account any turning *Bilbo* may have accomplished. Next, we translate the the camera coordinates. Translation, in effect, is shifting the camera's center coordinate to where it rests atop *Bilbo's* center coordinate.

3.6.8 Interior Point Calculation. After translation and rotation we quickly connect the two (x, y) coordinates that describe our planar object via the Bresenham algorithm (Figure A.3) [8]. Bresenham is an efficient algorithm used to render a line with pixels. The long dimension is incremented for each pixel, and the fractional slope is accumulated.

Consider a line with initial point (x_1, y_1) and terminal point (x_2, y_2) in device space[2]. If $\triangle x = x_2 - x_1$ and $\triangle y = y_2 - y_1$, we define the driving axis, DA, to be the x-axis if $|\triangle x| \geq |\triangle y|$, and the y-axis if $|\triangle y| > |\triangle x|$. The DA is used as the axis of control for the algorithm and is the axis of maximum movement. Within the main loop of the algorithm, the coordinate corresponding to the DA is incremented by one unit. The coordinate corresponding to the other axis (usually denoted the passive axis or PA) is only incremented as needed.

3.7 Post-Process Data Format

The final step in our software architecture is that of formatting the information for post-process viewing. Dense video streams of raw data have, in essence, been systematically reduced into a very concise, descriptive format for visualization. In contrast to the average picture containing 76,800 bytes, we have sustained object integrity at a real-time speed while saving 90% of the memory requirements. On average, our extracted data from a single image pair (including robot pose and translation) can be represented by a mere 800 bytes and is formatted as specified in Table 3.4. At the end of line two there can be an optional (shown in brackets) arbitrary amount of appended object points indicating a longer line segment. To visualize a sample of the final product, Appendix B contains information from a short run that mapped a 10′ segment.

[2]Device space is usually defined to be the $m \times n$ array of pixels that represents the area of the screen. A coordinate system is imposed on this space by labeling the lower-left-hand corner of the array as $(0, 0)$, with each pixel having unit length and width.

Table 3.4: Data format for occupancy grid viewer. Cartesian coordinates x and y are appended to robot and object positions, theta is heading, and probability (included for future work) indicates the certainty of a detection.

Line 1	robot.x	robot.y	robot.θ	object #	distance (mm)
Line 2	object.x	object.y	probability;	[object.x object.y probability;]	
Line 3	Blank				

3.8 System Description Conclusion

In this chapter we have detailed a system composed of robot and camera hardware, commercial off-the-shelf software (SVS, ARIA), open source software (OpenCV's Birchfield correspondence algorithm), newly developed optimization techniques (HSS, melding and masking, stack-based flood fill object identification, disparity intensity to distance mapping, and rectangle fitting), and common mapping data manipulation techniques (3D-to-2D scaling, translation and rotation, and Bresenham interior point calculation) acting in concert to produce concise, real-time mapping data given a dense vision data stream.

This architecture meets our research goal requirements of *quick object detection* and to *accurate representation using concise reconstruction information* resulting in an efficient, object representative vision mapping platform by providing optimization techniques and a planar assumption. In the next chapter we will test our architecture in different environments to determine its overall accuracy.

IV. Experimental Results

In order to determine the distance and size accuracy of our planar assumption in addition to overall system effectiveness, we conducted a series of experiments in several different environments varying in size, texture, and lighting. Below, each experimental domain is reconstructed listing all variable settings along with the resulting map and detailed analysis. To determine vision mapping accuracy, we visually compare, with the help of x and y grid lines, vision data with real probabilistic sonar data from the same mapping run.

4.1 Constant Settings

Most of our variable settings changed along with each new environment to maximize mapping potential. For example, in a narrow spaces we increased masking, whereas in an open area we had to account for an increased horizon. However, one setting that remained constant throughout was the signature for the *cvFindStereoCorrespondence* algorithm (Section 3.4.1.1). It was left unaltered because it yielded the best results under all conditions. Table 4.1 shows the actual parameter values we used. In this chapter, we will refer to this constant setting as the signature, S_{CONST}.

4.2 Corridor Experiment

4.2.1 Purpose and Environment. Our first experiment was conducted to determine the effectiveness of mapping a confined area such as a cave or corridor environment. We chose a narrow hallway in Bldg 194 to approximate such an environment. The particulars of the hallway include a medium colored patterned carpet, white low textured walls, dark baseboards, relatively constant fluorescent lighting, and small amounts of static objects, such as trash cans and recycling

Table 4.1: OpenCV correspondence constant signature, S_{CONST}

Argument	Semantics	Value
L	Left image of a stereo pair	*leftImage*
R	Right image of a stereo pair	*rightImage*
A	Algorithm used to find disparity	*CV_DISPARITY_BIRCHFIELD*
R_{depth}	Resulting depth image	*depthImage*
D	Maximum possible disparity	20
O	Constant occlusion penalty	5
M_{reward}	Constant match reward	12
H	Highly reliable region	15
M	Moderately reliable region	25
S	Slightly reliable region	15

bins. The actual length and width dimensions of the hallway was 66' and 5', respectively.

4.2.2 Settings. Settings for our corridor experiment are reflected in Table 4.2. Noteworthy settings include the pose and mask. The pose was changed to capture the environment from an orthogonal perspective and the mask was increased to help close proximity reliability.

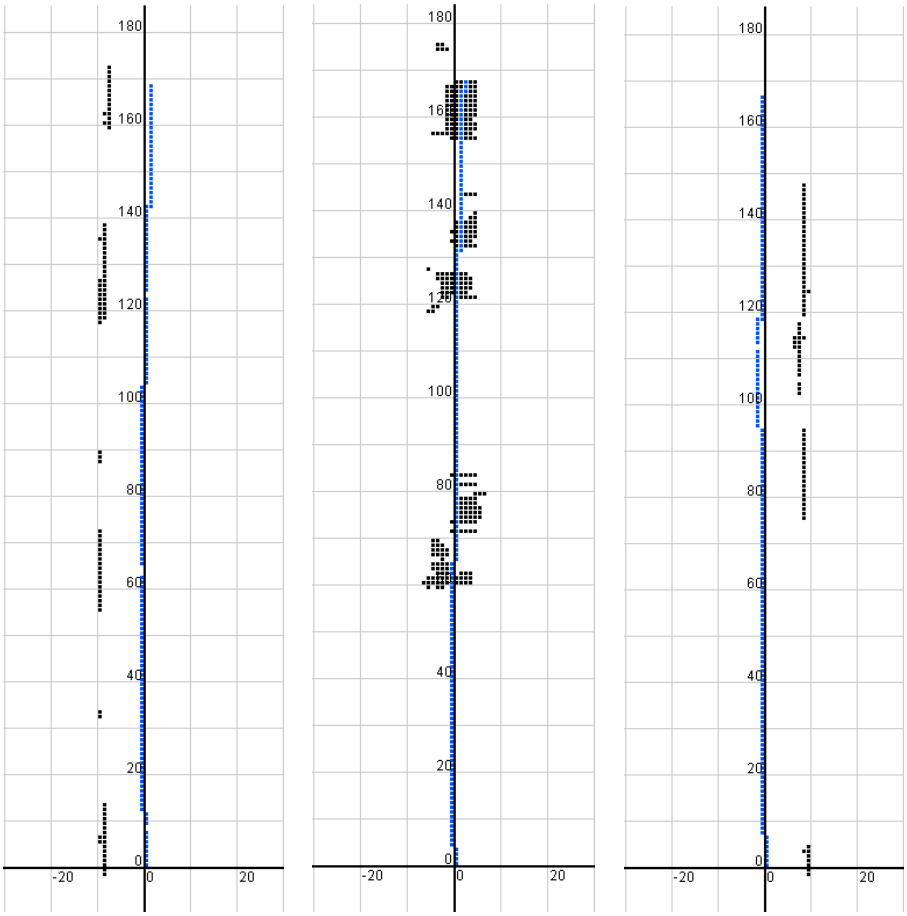
Table 4.2: Experiment 1 settings

Variable	Setting	Semantics
cvFindStereoCorrespondence	S_{CONST}	Correspondence algorithm signature
Pose	270, 0, 90	Camera orientation relative to robot
HSS	10	Spike removal size in vertical pixels
Mask	17	Mask threshold
Meld	5	Meld proximity in vertical pixels
Variance	1	Allowed difference in pixel intensity
Camera Tilt	Level	Vertical angle of camera

4.2.3 Results and Analysis. We conducted three separate runs that had the same start and end point where we changed the camera orientation for each run. Changing the orientation simulated having three cameras, one facing left,

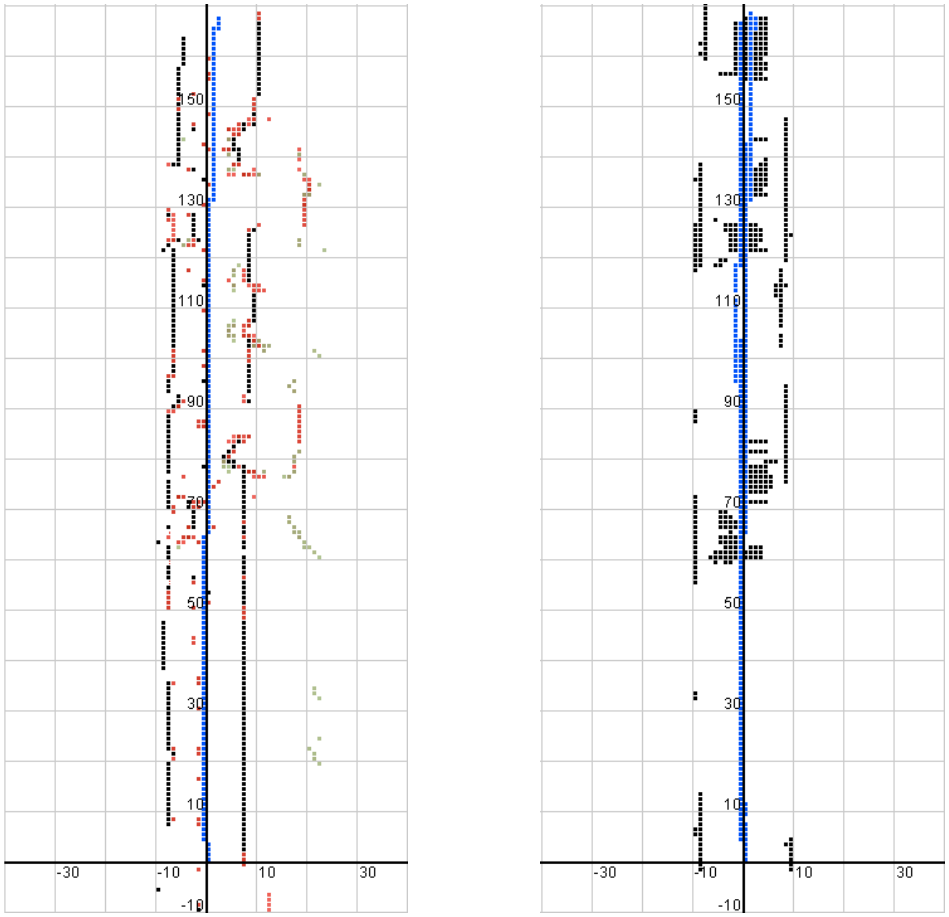
one forward, and one right, for denser environmental data collection akin to that of a robot with a sonar ring. The series of runs show a top down 2D view of the reconstructed maps of the left wall in 4.1(a), hallway obstructions in 4.1(b), and the right wall in 4.1(c). The blue points are the robots true path and the black points are detected objects.

Although the information might seem sparse to the reader at first glance, considering the low-texture solid-colored wall surface, the Birchfield disparity algorithm generated very good data exclusively from vision. Compared to sonar data on the same hallway, the vision processing detected and reconstructed the walls and obstacles with good precision (Figure 4.2) using only planes. Notice at approximately (-8,65), (8,80), and (-5,120) that obstacles were detected in the hallway for both sonar and vision. The vision detected objects appear larger because of perspective skewing. In other words, we are unable to accurately reconstruct a perspective view that is not normal (or orthogonal) to view angle. A unique pattern was affixed on the wall from (10,0) to (10,5) to determine how well the vision processing would perform given a pattern to help with horizontal scanline matching. From (-10,0) to (-10, 15) we see the result of a wall partition that has more texture than the normal white walls. Overall, the vision information in this environment shows to less reliable than sonar but acceptable considering our planar assumption, which loses some amount of object reconstruction granularity, and intrinsic sensor noise associated with vision. Lastly, doorways in 4.2(a) at centerpoints (-8,130) and (8,87) shows properly as open space in the sonar model, pose problems for the vision system where those spaces appear to be impassible. This impassible phenomenon caused by our planar assumption was later replicated in Section 4.4.3 where we discuss its cause in more detail.



(a) (b) (c)

Figure 4.1: Camera segments with pose of 270 in (a), 0 in (b), and 90 in (c). (Traditional x and y axes. Units in hundreds of mm.)



(a) (b)

Figure 4.2: Sonar data in (a) compared to unioned camera data from Figure 4.1 in (b). (Traditional x and y axes. Units in hundreds of mm.)

4.3 Low-Light Experiment

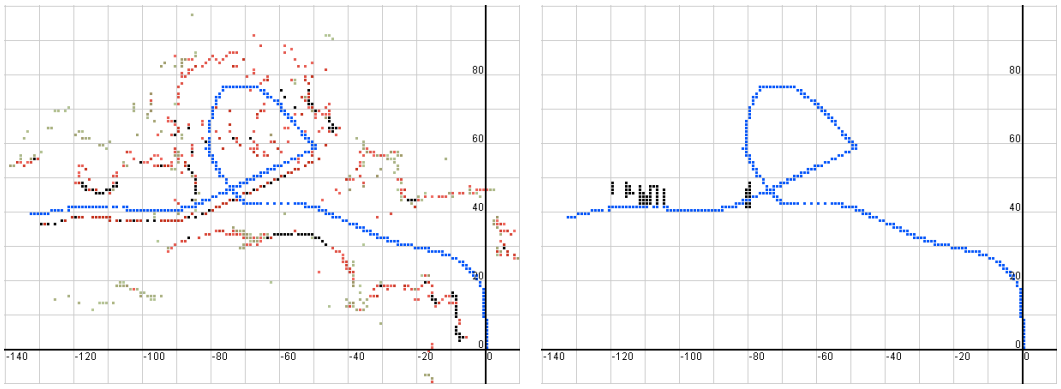
4.3.1 Environment. Our second experiment was conducted in the lobby of AFIT Bldg 642 to test a sparsely lit environment. The lobby contains many dark colored couches, chairs, and other obstacles. Flooring was predominately tile where the robot traversed while there was dark carpet under the furniture. The experiment objective was to determine the capabilities of our system given very little illumination.

4.3.2 Settings. Settings for the low-light experiment are reflected in Table 4.3. The mask was reduced to allow for more data capture while other settings are considered normal.

Table 4.3: Experiment 2 settings

Variable	**Setting**	**Semantics**
cvFindStereoCorrespondence	S_{CONST}	Correspondence algorithm signature
Pose	0	Camera orientation relative to robot
HSS	5	Spike removal size in vertical pixels
Mask	10	Mask threshold
Meld	5	Meld proximity in vertical pixels
Variance	1	Allowed difference in pixel intensity
Camera Tilt	Level	Vertical angle of camera

4.3.3 Results and Analysis. We did multiple mapping runs in this low-light environment. Each run displayed poor results due to lack of sufficient light and proper object differentiation. When objects are as indistinguishable as this, there is no way to match points, regardless of object representation, on a scanline because all the points look very similar, if not identical. We show the mapping limitation of a vision sensor given very low light, in Figure 4.3. The only visually detected items were a corner of the wall at (-80,45) and an elevator recess in the hall at (-110,45) which were contrasted with some glare off of the tile floors down the hall. At best, low-light vision reconstruction results are unreliable and, therefore,



(a) (b)

Figure 4.3: Sonar data in (a) compared to camera data in (b) in a very low light environment. (Traditional x and y axes. Units in hundreds of mm.)

should be mapped via other means. See our future extensions in 5.2.6 for a passive sensor extension recommendation.

4.4 Open Area Experiment

4.4.1 Purpose and Environment. Our last experiment was conducted to determine the effectiveness of mapping an open area. We chose a large classroom (Room 322) in AFIT Building 641. The room contained a medium colored patterned carpet, white low textured walls, dark baseboards, relatively constant fluorescent lighting, and a false wall made with tables lining nearly the entire room (Figure 4.5). The actual length and width dimensions of the room was 110' and 45', respectively.

4.4.2 Settings. Settings for our open area experiment are reflected in Table 4.4. Camera tilt and pose were the main variable modification. The camera was elevated to the specified tilt to better compensate for the below-the-horizon noise and the pose was adjusted to optimize results based upon a counter clockwise path traversal.

Figure 4.4: Open area test environment

Table 4.4: Experiment 3 settings

Variable	**Setting**	**Semantics**
cvFindStereoCorrespondence	S_{CONST}	Correspondence algorithm signature
Pose	90	Camera orientation relative to robot
HSS	10	Spike removal size in vertical pixels
Mask	9	Mask threshold
Meld	5	Meld proximity in vertical pixels
Variance	1	Allowed difference in pixel intensity
Camera Tilt	20°	Vertical angle of camera

4.4.3 Results and Analysis. We thoroughly tested this mapping environment because of repeated false readings close to the robot, although nothing was actually obstructing it. Our analysis indicated that the problem resulted from the information gathered when the horizon line was significant. In other words, when the robot looked across a large open area it would see the carpet, which is below the horizon line, and would classify it as a close-proximity obstacle causing false positives. This phenomenon is an example of the biggest limitation introduced by a pure planar assumption where there is no perspective. However, we examined several possible compensational solutions:

- Image alteration

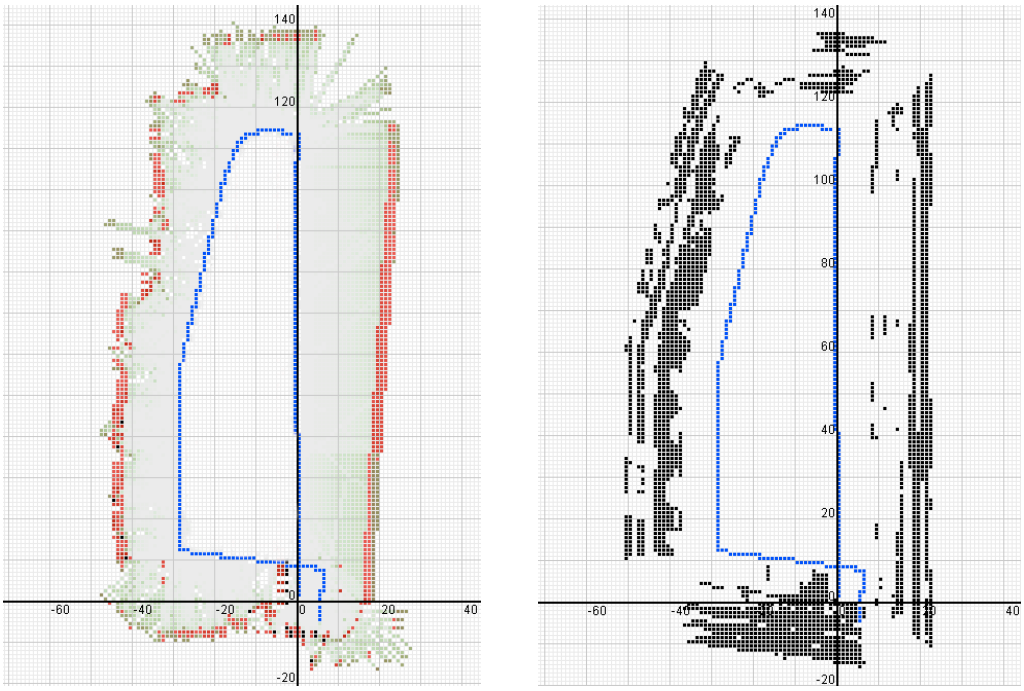
- Increase HSS *s* value
- Tilting the camera upward

We chose to tilt the camera upward sufficiently to avoid the below-the-horizon data, because it posed the least amount of undesirable side effects. The resulting 20° tilt can be seen in the original picture of *Bilbo* in Figure 3.1. In contrast to the camera tilt, altering the image involved changing the source code and losing image resolution, and increasing the HSS *s* value was not feasible because the *s* value would have to be so significant that we might lose good data.

Results from our room testing are shown in Figure 4.5. The overall shape of the large area came out well. The thicker looking walls comes from the space between the artificial wall barrier and actual wall (to include a portion of the ceiling because of the camera tilt). Notice at centerpoints (9,17), (9,41), (9,67), (11,67), (8,100), and (9,109) that we encountered small pockets of false positives. The rest of the mapping performed well showing the first barrier, which matches sonar readings, and subsequent layer information that is undetectable with *Bilbo's* sonar ring. The approximate 4' distance behind the initial barrier is accurate within our depth variance specification.

4.5 Conclusion of Results

Our experiments have given good insight into the stereo vision mapping capabilities of our system in three unique predominantly low-texture settings. Common across all experiments, we see detection and distance precision is proportional to the amount of texture in the environment. In the very-low textured corridor, experiment results were intermittently good (45% detection rate), whereas, in the more textured open area experiment results were markedly better (90% detection rate). The detection rate was calculated using the ratio of grid cells that indicate a wall detection to that of the probabilistic sonar data.



(a) (b)

Figure 4.5: Sonar data in (a) compared to camera in (b). (Traditional x and y axes. Units in hundreds of mm.)

As for mapping a very low-lit environment, we have determined that a pure stereo vision sensing system is intrinsically incapable of generating reliable maps because of insufficient texture contrast. This limitation could be easily overcome by augmenting the vision senors with either an illumination enhancing mechanism or an active sensor.

We have shown that given a moderately lit environment, our planar assumption has exhibited good accuracy for reconstructing objects using sonar as a baseline. It particularly excels in narrow settings (such as a corridor or hallway) where there is no horizon and introduces false positives in large spanning areas if compensational actions are not explicitly taken. Aside from good accuracy (as compared to reliable probabilistic sonar data) using only planes for reconstruction, we maintained a very fast 10Hz processing rate.

V. Conclusion and Extensions

5.1 Introduction

While many active methods exist for robots to map their environment [7,27,33,34], real-time passive sensor mapping is certainly not as prevalent due to computational expense. Our planar assumption, along with vision optimization techniques, have enabled us to build reliable 2D maps in real-time given a well-lit, semi-textured, and non-spanning environment. If these assumptions are met, we can efficiently apply our data manipulation methods to generate 2D planar reconstruction information at a speed of 10Hz. Given the rapidly increasing processor speeds, one can safely assume that more computations could be done in the near future to add some of the recommended extensions below while maintaining (or even increasing) the cycles per second. Each recommended extension provides a mechanism to help remove brittleness and/or increase reliability and information dimensionality[1].

5.2 Future Extensions

5.2.1 Probabilistic Overlay. Our first recommended future extension is to add a probabilistic overlay to our 2D occupancy grid. Doing so would help reject spurious returns while reinforcing good ones. Maps, would, therefore, not only show that an object was detected in the cell, but would add a probability figure to indicate a certainty strength based on reinforcement. This would probably be the easiest of the extensions to implement.

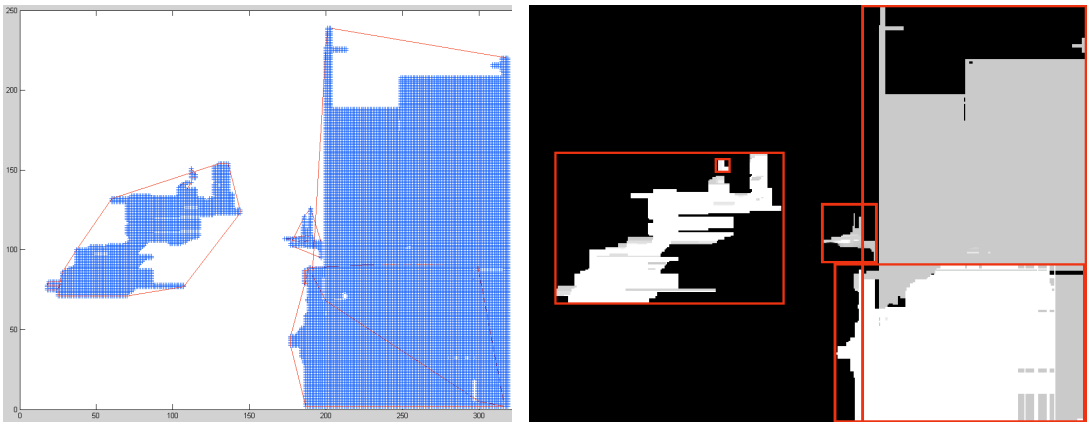
5.2.2 Edge Detection. Integrating an edge detection mechanism, such as Sobel or Canny, with the planar assumption would give richer reconstruction

[1]This concept refers to the richness of information with regard to point-to-point relationships and what can be interpreted from them

knowledge. Once a perspective can be determined, ideally toward a known vanishing point, borders and objects can be reconstructed with greater accuracy because more information is known about the environment. This extension would remove the below-the-horizon brittleness currently exhibited in our level-camera configuration and eliminate non-orthogonal skewing (Section 4.2.3).

5.2.3 Panning/Multiple Cameras. As with a sonar ring or other aggregated sensor, multiple readings are meshed into a probabilistic model to help determine the certainty. An example of this type of complimentary coordination can be seen in a sonar ring that is approaching an object, and as it passes the object other sonars are increasing or decreasing the probability associated with that object. Our research used a single stereo camera solution that required multiple runs of the same environment to fully collect the reconstruction information that was necessary to accurately reproduce and environment. A multiple camera or a panning camera extension would help collect a more dense data set from the environment, thus, eliminating the need to perform multiple passes.

5.2.4 Convex Hull. In Section 3.6.4 we proposed a rectangle fitting approach to describe an object in 2D space by four Cartesian coordinates. This approach was selected to expedite the object processing. Computational time aside, a more accurate approach might be to use a series of smaller rectangular fittings, as what is commonly done to determine the area under a curve, or a convex hull perimeter trace (like qhull in MatLab or C++). Convex hulls safely approximate an object's size while minimizing the worst-case-scenario area represented by the single rectangular fitting area. We briefly experimented with convex hulls (Figure 5.1) around objects but subsequently abandoned the technique in an effort to speed processing. Notice how the objects are more accurately represented with convex hull tracing. The unique object labeling in Section 3.6.3 takes care of a large portion of the algorithmic convex hull implementation. To get the full benefit of



(a) (b)

Figure 5.1: Convex hull traced objects in (a) compared to rectangle fitted objects in (b)

the convex hull function, one would need to increase the dimensionality of the maps since, from a 2D perspective, the maximum and minimum Cartesian x- and y-coordinates would be the same as that found in the rectangular fitting.

5.2.5 Increased Dimensional Mapping. Increased dimensional mapping is a substantial extension of our work. Extracting the data for, computing, and displaying 2.5D or 3D maps would require major code modification and would require efficient mechanisms on a competent processor to achieve real-time mapping. Furthermore, expanding the dimensionality would eliminate the below-the-horizon phenomenon we encountered when extracting 2D planes removing system brittleness.

5.2.6 Night Vision. As seen in our research low-light conditions virtually paralyze the vision system because there is insufficient contrast between objects leading to the inability to generate reliable disparity readings. A major extension to our work would be the addition of night vision capabilities so that the well-lit assumption could be removed. Furthermore, navigating in a subterranean cave or bunker complex is sure to have areas of low-lighting.

5.2.7 Obstacle Avoidance. With the real-time collection comes a greater ability to immediately recognize a dynamic object in a scene. An obstacle avoidance extension would serve good utility in all regards, especially in the role as a hostile environment mapper. The robot would be quite useless if it were to run into the first obstacle it comes in contact with and stay obstructed. Designing a passive robot platform that could autonomously navigate a dynamic environment, while mapping, would be a major feat.

5.2.8 Localization. All large-scale robust mapping solutions must contain some type of localization to account for navigation error that propagate over time. Localizing via vision in real-time is a hearty challenge that is mentioned last purposefully because it is, perhaps, the most difficult extension because of already limited processing abilities tied to vision processing.

5.3 Conclusion

We have successfully shown that reliable maps can be generated under a variety of textures using a planar assumption in real-time. However, the most reliable maps are generated where there ample light, rich texture, orthogonal perspective, and a minimal horizon. Without some sort of optimization it is infeasible to quickly process the dense amount of vision stream data necessary for a real-time application. Our planar assumption combined with optimization techniques not only reduced data requirements by 90% but also processed all necessary vision mapping tasks into concise map data at an impressive rate of 10Hz as compared to the closest related work [29] in the field.

Figure A.1: HORIZONTAL SPIKE SMOOTHING

Require: $Array_{HEIGHTxWIDTH}$, A
Require: Horizontal spike value s
  for every column, c, in A **do**
    for every row, r, in A **do**
      $i \leftarrow r * WIDTH + c$
      if HEIGHT $\geq s$ **then**
        if A[i] $\neq 0$ and $count = 0$ **then**
          $color \leftarrow A[i]$
          $begin \leftarrow i$
          increment $count$
        else if $A[i] = color$ **then**
          increment $count$
        else if $count \neq 0$ **then**
          if $count \leq s$ **then**
            for k cells between $begin$ to $A[i\text{-}1]$ **do**
              $A[k] \leftarrow$ empty
            end for
          end if
        else if $A[i]$ = empty **then**
          $count \leftarrow 0$
          $color \leftarrow$ reset
          $begin \leftarrow$ reset
        else
          $count \leftarrow 1$
          $color \leftarrow A[i]$
          $start \leftarrow i$
        end if
      end if
    end for
  end for

Figure A.2: MELDING

Require: $Array_{HEIGHTxWIDTH}$, A
Require: iIndex = inner index of a significant object
Require: ϕ = objects found thus far
  for Each disparity level, δ **do**
    $intensity \leftarrow A$[iIndex]
    push iIndex on stack, S
    while stack not empty **do**
      $left \leftarrow$ top S object
      while $A[left] \neq$a known border **do**
        $A[left] \leftarrow \phi$
        decrement $left$
      end while
      save smallest Cartesian x value
      $right \leftarrow$ top S object
      while $A[right] \neq$ a known border **do**
        $A[right] \leftarrow \phi$
        increment $right$
      end while
      save largest Cartesian x value
      pop S
      find all upper left-most seed cells and push on S
      find all lower left-most seed cells and push on S
    end while
  end for

Figure A.3: BRESENHAM LINE DRAWING

Require: $\triangle x = x_2 - x_1$
Require: $\triangle y = y_2 - y_1$
Require: $M = \frac{\triangle y}{\triangle x}$
Require: $j = y_1$
Require: $\varepsilon = m - 1$
  for $i = x_1$ to $x_2 - 1$ **do**
    illuminate (i, j)
    if $\varepsilon \geq 0$ **then**
      $j+ = 1$
      $\varepsilon- = 1.0$
    end if
    $i+ = 1$
    $\varepsilon+ = m$
  end for

Appendix B. Plane Data Sample

```
0 1 0 -1 -1

0 2 0 1 1133
9 0 1.0000;9 -1 1.0000;9 -2 1.0000;
0 3 0 -1 -1

0 4 359 -1 -1

0 5 359 -1 -1

0 6 359 -1 -1

0 7 359 -1 -1

0 8 359 -1 -1

0 9 359 1 1133
9 14 1.0000;9 13 1.0000;9 12 1.0000;9 11 1.0000;9 10 1.0000;9 9 1.0000;9 8 1.0000;9 7 1.0000;9 6 1.0000;9 5 1.0000;
-1 10 359 1 1133
8 15 1.0000;8 14 1.0000;8 13 1.0000;8 12 1.0000;8 11 1.0000;8 10 1.0000;8 9 1.0000;8 8 1.0000;8 7 1.0000;8 6 1.0000;
-1 11 359 1 1133
8 15 1.0000;8 14 1.0000;8 13 1.0000;8 12 1.0000;8 11 1.0000;8 10 1.0000;8 9 1.0000;8 8 1.0000;8 7 1.0000;
-1 11 359 2 1194
8 9 1.0000;8 8 1.0000;8 7 1.0000;
-1 11 359 3 1194
8 15 1.0000;
-1 12 359 1 1133
8 15 1.0000;8 14 1.0000;8 13 1.0000;8 12 1.0000;8 11 1.0000;8 10 1.0000;8 9 1.0000;8 8 1.0000;
-1 13 359 1 1133
8 16 1.0000;8 15 1.0000;8 14 1.0000;8 13 1.0000;8 12 1.0000;8 11 1.0000;8 10 1.0000;8 9 1.0000;
-1 13 359 2 1194
8 16 1.0000;
0 14 6 1 1133
9 16 1.0000;9 15 1.0000;9 14 1.0000;9 13 1.0000;9 12 1.0000;9 11 1.0000;9 10 1.0000;9 9 1.0000;
0 14 6 2 1194
9 15 1.0000;9 14 1.0000;
0 15 11 1 1133
9 15 1.0000;9 14 1.0000;9 13 1.0000;8 12 1.0000;8 11 1.0000;8 10 1.0000;8 9 1.0000;
0 15 11 2 1133
9 13 1.0000;
0 16 12 1 1133
9 13 1.0000;
0 16 12 2 1133
10 18 1.0000;10 17 1.0000;9 16 1.0000;9 15 1.0000;9 14 1.0000;9 13 1.0000;8 12 1.0000;8 11 1.0000;8 10 1.0000;
0 16 12 3 1133
9 13 1.0000;8 12 1.0000;
0 16 12 4 1194
9 13 1.0000;8 12 1.0000;
0 16 12 5 1194
9 13 1.0000;
0 16 12 6 1194
9 15 1.0000;9 14 1.0000;9 13 1.0000;
0 16 12 7 1194
9 13 1.0000;
0 16 12 8 1194
9 16 1.0000;9 15 1.0000;
0 16 12 9 1255
10 13 1.0000;
0 16 12 10 1255
10 14 1.0000;10 13 1.0000;
0 16 12 11 1316
9 12 1.0000;
0 17 21 1 1133
8 13 1.0000;8 12 1.0000;
0 17 21 2 1133
9 16 1.0000;9 15 1.0000;8 14 1.0000;8 13 1.0000;8 12 1.0000;7 11 1.0000;7 10 1.0000;
0 17 21 3 1133
9 16 1.0000;9 15 1.0000;8 14 1.0000;8 13 1.0000;
```

```
0 17 21 4 1133
8 14 1.0000;8 13 1.0000;

0 17 21 5 1194
8 14 1.0000;

0 17 21 6 1194
9 15 1.0000;9 14 1.0000;8 13 1.0000;8 12 1.0000;

0 17 21 7 1194
8 14 1.0000;8 13 1.0000;

0 17 21 8 1316
9 12 1.0000;

1 17 28 1 1133
8 12 1.0000;8 11 1.0000;

1 17 28 2 1133
10 14 1.0000;9 13 1.0000;9 12 1.0000;8 11 1.0000;8 10 1.0000;7 9 1.0000;

1 17 28 3 1133
9 13 1.0000;8 12 1.0000;

1 17 28 4 1194
9 14 1.0000;9 13 1.0000;

1 17 28 5 1194
9 14 1.0000;9 13 1.0000;8 12 1.0000;8 11 1.0000;

1 17 28 6 1255
10 13 1.0000;10 12 1.0000;

1 18 28 1 1133
8 12 1.0000;

1 18 28 2 1133
7 11 1.0000;7 10 1.0000;

1 18 28 3 1133
8 13 1.0000;

1 18 28 4 1133
11 17 1.0000;10 16 1.0000;10 15 1.0000;9 14 1.0000;8 13 1.0000;8 12 1.0000;7 11 1.0000;

1 18 28 5 1194
9 15 1.0000;9 14 1.0000;

1 18 28 6 1194
9 15 1.0000;8 14 1.0000;8 13 1.0000;

1 18 28 7 1255
10 14 1.0000;9 13 1.0000;9 12 1.0000;

1 19 31 1 1133
7 12 1.0000;

1 19 31 2 1133
7 12 1.0000;

1 19 31 3 1133
10 17 1.0000;9 16 1.0000;9 15 1.0000;8 14 1.0000;8 13 1.0000;7 12 1.0000;

1 19 31 4 1133
8 13 1.0000;

1 19 31 5 1194
9 15 1.0000;8 14 1.0000;8 13 1.0000;

1 19 31 6 1194
9 15 1.0000;9 14 1.0000;

1 19 31 7 1194
9 15 1.0000;8 14 1.0000;8 13 1.0000;7 12 1.0000;

1 19 31 8 1255
10 14 1.0000;9 13 1.0000;

1 19 31 9 1316
9 13 1.0000;

2 19 35 1 1133
11 16 1.0000;10 15 1.0000;10 14 1.0000;9 13 1.0000;9 12 1.0000;8 11 1.0000;

2 19 35 2 1133
9 13 1.0000;8 12 1.0000;

2 19 35 3 1194
11 15 1.0000;10 14 1.0000;9 13 1.0000;8 12 1.0000;

2 19 35 4 1255
10 13 1.0000;10 12 1.0000;

2 20 40 1 1133
11 16 1.0000;10 15 1.0000;9 14 1.0000;8 13 1.0000;7 12 1.0000;

2 20 40 2 1133
9 14 1.0000;

2 20 40 3 1194
10 16 1.0000;9 15 1.0000;9 14 1.0000;

2 20 40 4 1194
10 16 1.0000;9 15 1.0000;9 14 1.0000;

2 20 40 5 1255
```

```
10 13 1.0000;

3 20 47 1 1133
11 15 1.0000;10 14 1.0000;9 13 1.0000;8 12 1.0000;7 11 1.0000;

3 20 47 2 1194
11 15 1.0000;10 14 1.0000;9 13 1.0000;

3 20 47 3 1194
10 14 1.0000;

3 20 47 4 1194
11 15 1.0000;10 14 1.0000;9 13 1.0000;

3 20 47 5 1194
11 15 1.0000;

3 20 47 6 1255
10 13 1.0000;

3 20 47 7 1255
10 13 1.0000;

3 21 50 1 1133
11 16 1.0000;10 15 1.0000;9 14 1.0000;8 14 1.0000;7 13 1.0000;6 12 1.0000;

3 21 50 2 1194
11 16 1.0000;10 15 1.0000;9 14 1.0000;

3 21 50 3 1194
9 14 1.0000;

3 21 50 4 1255
9 13 1.0000;

3 21 50 5 1255
9 13 1.0000;

4 21 54 1 1133
12 16 1.0000;11 15 1.0000;10 14 1.0000;9 14 1.0000;8 13 1.0000;7 12 1.0000;

4 21 54 2 1194
12 15 1.0000;11 15 1.0000;10 14 1.0000;9 14 1.0000;

4 21 54 3 1194
10 14 1.0000;9 14 1.0000;

4 22 61 1 1133
11 16 1.0000;10 15 1.0000;9 15 1.0000;8 14 1.0000;7 14 1.0000;6 13 1.0000;

4 22 61 2 1133
9 15 1.0000;8 14 1.0000;

4 22 61 3 1133
9 15 1.0000;8 14 1.0000;

4 22 61 4 1194
9 15 1.0000;

4 22 61 5 1194
9 15 1.0000;8 14 1.0000;

5 22 63 1 1133
13 16 1.0000;12 16 1.0000;11 15 1.0000;10 15 1.0000;9 14 1.0000;8 14 1.0000;7 13 1.0000;6 13 1.0000;

5 22 63 2 1194
10 14 1.0000;

5 22 63 3 1194
10 14 1.0000;

6 22 76 1 1133
12 14 1.0000;11 14 1.0000;10 14 1.0000;9 13 1.0000;8 13 1.0000;7 13 1.0000;6 13 1.0000;

6 22 76 2 1133
4 12 1.0000;

6 22 76 3 1255
7 12 1.0000;6 12 1.0000;

6 22 76 4 1377
4 10 1.0000;3 10 1.0000;

7 22 88 1 1133
11 13 1.0000;10 13 1.0000;9 13 1.0000;8 13 1.0000;7 13 1.0000;6 13 1.0000;5 13 1.0000;

7 22 88 2 1133
4 13 1.0000;3 13 1.0000;

7 22 88 3 1133
9 13 1.0000;

8 22 89 1 1133
11 13 1.0000;10 13 1.0000;9 13 1.0000;8 13 1.0000;7 13 1.0000;6 13 1.0000;

8 22 89 2 1133
10 13 1.0000;

8 22 89 3 1377
2 11 1.0000;

9 22 89 1 1133
11 13 1.0000;10 13 1.0000;9 13 1.0000;8 13 1.0000;7 13 1.0000;6 13 1.0000;

9 22 89 2 1133
9 13 1.0000;
```

```
9 22 89 3 1438
11 10 1.0000;10 10 1.0000;

10 22 86 1 1133
12 13 1.0000;11 13 1.0000;10 13 1.0000;9 13 1.0000;8 13 1.0000;7 13 1.0000;

10 22 86 2 1133
12 13 1.0000;11 13 1.0000;

11 22 79 1 1133
17 14 1.0000;16 14 1.0000;15 14 1.0000;14 14 1.0000;13 13 1.0000;12 13 1.0000;11 13 1.0000;10 13 1.0000;

11 22 79 2 1255
12 12 1.0000;11 12 1.0000;10 11 1.0000;9 11 1.0000;

11 22 79 3 1316
9 11 1.0000;

11 22 79 4 1316
8 11 1.0000;7 11 1.0000;

11 22 79 5 1316
11 12 1.0000;10 12 1.0000;9 11 1.0000;8 11 1.0000;7 11 1.0000;

11 22 79 6 1377
11 11 1.0000;10 11 1.0000;9 10 1.0000;8 10 1.0000;7 10 1.0000;

12 22 79 1 1133
18 14 1.0000;17 14 1.0000;16 14 1.0000;15 14 1.0000;14 13 1.0000;13 13 1.0000;12 13 1.0000;11 13 1.0000;

12 22 79 2 1133
13 13 1.0000;12 13 1.0000;

12 22 79 3 1194
15 13 1.0000;

12 22 79 4 1194
15 13 1.0000;14 13 1.0000;

12 22 79 5 1194
18 14 1.0000;17 14 1.0000;16 14 1.0000;15 13 1.0000;14 13 1.0000;13 13 1.0000;12 13 1.0000;

12 23 79 1 1133
12 14 1.0000;11 14 1.0000;

12 23 79 2 1133
18 15 1.0000;17 15 1.0000;16 14 1.0000;15 14 1.0000;14 14 1.0000;

12 23 79 3 1194
18 15 1.0000;17 15 1.0000;16 15 1.0000;15 14 1.0000;14 14 1.0000;13 14 1.0000;12 14 1.0000;

12 23 79 4 1194
14 14 1.0000;

12 23 79 5 1194
14 14 1.0000;

12 23 79 6 1194
14 14 1.0000;

12 23 79 7 1194
14 14 1.0000;13 14 1.0000;

12 23 79 8 1194
13 14 1.0000;

12 23 79 9 1194
13 14 1.0000;

12 23 79 10 1194
16 15 1.0000;

12 23 79 11 1194
15 14 1.0000;

12 23 79 12 1194
15 14 1.0000;14 14 1.0000;

12 23 79 13 1194
14 14 1.0000;

12 23 79 14 1194
14 14 1.0000;

12 23 79 15 1194
16 15 1.0000;15 15 1.0000;14 14 1.0000;13 14 1.0000;

12 23 79 16 1194
16 15 1.0000;15 14 1.0000;

12 23 79 17 1194
18 15 1.0000;17 15 1.0000;16 15 1.0000;15 14 1.0000;14 14 1.0000;13 14 1.0000;12 14 1.0000;

13 23 79 1 1133
13 14 1.0000;12 13 1.0000;11 13 1.0000;

13 23 79 2 1133
19 15 1.0000;18 15 1.0000;17 15 1.0000;16 14 1.0000;15 14 1.0000;14 14 1.0000;

13 23 79 3 1194
18 15 1.0000;17 15 1.0000;16 15 1.0000;15 14 1.0000;14 14 1.0000;13 14 1.0000;

14 23 79 1 1133
13 14 1.0000;12 13 1.0000;
```

```
14 23 79 2 1133
19 15 1.0000;18 15 1.0000;17 14 1.0000;16 14 1.0000;15 14 1.0000;

14 23 79 3 1194
19 15 1.0000;18 15 1.0000;

14 23 79 4 1194
13 14 1.0000;12 13 1.0000;

14 23 79 5 1194
13 14 1.0000;

14 23 79 6 1255
18 14 1.0000;17 14 1.0000;16 14 1.0000;15 13 1.0000;14 13 1.0000;13 13 1.0000;

14 23 79 7 1255
12 12 1.0000;

14 23 79 8 1316
11 12 1.0000;10 12 1.0000;

15 23 79 1 1133
20 15 1.0000;19 15 1.0000;18 14 1.0000;17 14 1.0000;16 14 1.0000;

15 23 79 2 1194
19 15 1.0000;

15 23 79 3 1316
12 12 1.0000;11 12 1.0000;

15 23 79 4 1316
19 14 1.0000;18 14 1.0000;17 13 1.0000;16 13 1.0000;15 13 1.0000;14 13 1.0000;13 12 1.0000;12 12 1.0000;

15 23 79 5 1377
19 13 1.0000;18 13 1.0000;17 12 1.0000;16 12 1.0000;15 12 1.0000;14 12 1.0000;13 11 1.0000;12 11 1.0000;11 11 1.0000;

15 23 79 6 1438
19 12 1.0000;18 12 1.0000;17 11 1.0000;16 11 1.0000;15 11 1.0000;14 11 1.0000;13 10 1.0000;12 10 1.0000;

15 23 79 7 1438
12 10 1.0000;11 10 1.0000;

15 23 79 8 1438
19 12 1.0000;18 12 1.0000;17 11 1.0000;16 11 1.0000;15 11 1.0000;14 11 1.0000;13 10 1.0000;12 10 1.0000;11 10 1.0000;

16 23 79 1 1133
20 15 1.0000;19 15 1.0000;18 14 1.0000;17 14 1.0000;16 14 1.0000;

16 23 79 2 1194
19 14 1.0000;

17 23 79 1 1133
22 15 1.0000;21 15 1.0000;20 15 1.0000;19 14 1.0000;18 14 1.0000;17 14 1.0000;

17 24 79 1 1133
23 16 1.0000;22 16 1.0000;21 16 1.0000;20 15 1.0000;19 15 1.0000;18 15 1.0000;

18 24 79 1 1133
24 16 1.0000;23 16 1.0000;22 16 1.0000;21 15 1.0000;20 15 1.0000;19 15 1.0000;18 15 1.0000;

19 24 79 1 1133
24 16 1.0000;23 16 1.0000;22 15 1.0000;21 15 1.0000;20 15 1.0000;

19 24 79 2 1194
20 15 1.0000;

20 24 79 1 1133
25 16 1.0000;24 16 1.0000;23 16 1.0000;22 15 1.0000;21 15 1.0000;20 15 1.0000;

20 24 79 2 1194
21 15 1.0000;20 15 1.0000;

20 24 79 3 1377
26 14 1.0000;25 14 1.0000;24 14 1.0000;23 13 1.0000;22 13 1.0000;21 13 1.0000;20 13 1.0000;

20 24 79 4 1377
26 14 1.0000;25 14 1.0000;24 14 1.0000;

20 24 79 5 1438
22 12 1.0000;21 12 1.0000;20 12 1.0000;19 12 1.0000;

21 24 79 1 1133
26 16 1.0000;25 16 1.0000;24 16 1.0000;23 15 1.0000;22 15 1.0000;21 15 1.0000;20 15 1.0000;

21 24 79 2 1255
21 14 1.0000;

21 24 79 3 1316
20 14 1.0000;

21 24 79 4 1316
21 14 1.0000;

21 24 79 5 1377
27 14 1.0000;26 14 1.0000;25 14 1.0000;24 14 1.0000;23 13 1.0000;22 13 1.0000;21 13 1.0000;20 13 1.0000;

22 24 79 1 1133
28 16 1.0000;27 16 1.0000;26 16 1.0000;25 15 1.0000;24 15 1.0000;23 15 1.0000;

22 24 79 2 1194
23 15 1.0000;22 15 1.0000;21 14 1.0000;20 14 1.0000;

22 24 79 3 1255
24 14 1.0000;23 14 1.0000;22 14 1.0000;21 14 1.0000;

22 24 79 4 1316
```

```
20 13 1.0000;

22 24 79 5 1377
27 14 1.0000;26 14 1.0000;25 13 1.0000;24 13 1.0000;23 13 1.0000;22 13 1.0000;21 12 1.0000;20 12 1.0000;

22 24 79 6 1438
20 11 1.0000;

22 25 79 1 1133
27 17 1.0000;26 17 1.0000;25 17 1.0000;24 16 1.0000;23 16 1.0000;22 16 1.0000;

22 25 79 2 1194
23 16 1.0000;22 16 1.0000;21 15 1.0000;20 15 1.0000;

22 25 79 3 1255
27 16 1.0000;

22 25 79 4 1377
26 15 1.0000;

22 25 79 5 1377
19 13 1.0000;

22 25 79 6 1438
26 14 1.0000;25 14 1.0000;24 13 1.0000;23 13 1.0000;22 13 1.0000;21 13 1.0000;20 12 1.0000;19 12 1.0000;

23 25 79 1 1133
28 17 1.0000;27 17 1.0000;26 17 1.0000;25 16 1.0000;24 16 1.0000;23 16 1.0000;

23 25 79 2 1316
27 16 1.0000;

23 25 79 3 1438
26 13 1.0000;25 13 1.0000;24 13 1.0000;23 12 1.0000;22 12 1.0000;21 12 1.0000;20 12 1.0000;

24 25 79 1 1133
29 17 1.0000;28 17 1.0000;27 17 1.0000;26 16 1.0000;25 16 1.0000;24 16 1.0000;

24 25 79 2 1194
25 16 1.0000;24 16 1.0000;

24 25 79 3 1316
27 15 1.0000;

24 25 79 4 1377
27 14 1.0000;

25 25 79 1 1133
30 17 1.0000;29 17 1.0000;28 17 1.0000;27 16 1.0000;26 16 1.0000;25 16 1.0000;

25 25 79 2 1316
27 15 1.0000;

26 25 79 1 1133
30 17 1.0000;29 17 1.0000;28 16 1.0000;27 16 1.0000;

26 25 79 2 1194
28 16 1.0000;27 16 1.0000;

26 25 79 3 1316
28 15 1.0000;27 15 1.0000;

27 25 79 1 1133
33 17 1.0000;32 17 1.0000;31 17 1.0000;30 16 1.0000;29 16 1.0000;28 16 1.0000;27 16 1.0000;

27 25 79 2 1316
28 15 1.0000;

28 25 81 1 1133
32 17 1.0000;31 17 1.0000;30 17 1.0000;29 16 1.0000;28 16 1.0000;27 16 1.0000;

28 25 81 2 1316
30 15 1.0000;

28 25 81 3 1377
28 14 1.0000;

28 26 83 1 1133
32 17 1.0000;31 17 1.0000;30 17 1.0000;29 17 1.0000;28 17 1.0000;

28 26 83 2 1133
28 17 1.0000;27 17 1.0000;

28 26 83 3 1194
29 17 1.0000;28 17 1.0000;

28 26 83 4 1255
28 16 1.0000;27 16 1.0000;

28 26 83 5 1377
27 15 1.0000;

29 26 84 1 1133
33 17 1.0000;32 17 1.0000;31 17 1.0000;30 17 1.0000;29 17 1.0000;28 17 1.0000;

29 26 84 2 1194
34 17 1.0000;33 17 1.0000;32 17 1.0000;

29 26 84 3 1194
30 17 1.0000;29 17 1.0000;

29 26 84 4 1316
28 16 1.0000;

29 26 84 5 1377
28 15 1.0000;
```

```
29 26 84 6 1438
34 14 1.0000;

29 26 84 7 1438
29 14 1.0000;28 14 1.0000;27 14 1.0000;

30 26 85 1 1133
35 17 1.0000;34 17 1.0000;33 17 1.0000;32 17 1.0000;31 17 1.0000;30 17 1.0000;29 17 1.0000;

30 26 85 2 1194
34 17 1.0000;33 17 1.0000;32 17 1.0000;31 17 1.0000;

30 26 85 3 1438
35 14 1.0000;34 14 1.0000;33 14 1.0000;32 14 1.0000;31 14 1.0000;30 14 1.0000;29 14 1.0000;28 14 1.0000;

31 26 86 1 1133
34 17 1.0000;33 17 1.0000;32 17 1.0000;31 17 1.0000;30 17 1.0000;

31 26 86 2 1194
34 17 1.0000;

31 26 86 3 1255
32 16 1.0000;

31 26 86 4 1438
35 14 1.0000;34 14 1.0000;33 14 1.0000;32 14 1.0000;31 14 1.0000;30 14 1.0000;29 14 1.0000;

32 26 87 1 1133
31 17 1.0000;

32 26 87 2 1133
35 17 1.0000;34 17 1.0000;33 17 1.0000;32 17 1.0000;31 17 1.0000;30 17 1.0000;

32 26 87 3 1133
35 17 1.0000;

32 26 87 4 1255
32 16 1.0000;

32 26 87 5 1377
34 15 1.0000;

32 26 87 6 1438
35 14 1.0000;34 14 1.0000;33 14 1.0000;32 14 1.0000;31 14 1.0000;30 14 1.0000;29 14 1.0000;

33 26 86 1 1133
37 17 1.0000;36 17 1.0000;35 17 1.0000;34 17 1.0000;33 17 1.0000;32 17 1.0000;

33 26 86 2 1377
36 15 1.0000;

34 26 85 1 1133
37 17 1.0000;36 17 1.0000;35 17 1.0000;34 17 1.0000;33 17 1.0000;

34 26 85 2 1377
36 15 1.0000;

35 26 84 1 1133
39 17 1.0000;38 17 1.0000;37 17 1.0000;36 17 1.0000;35 17 1.0000;34 17 1.0000;

35 26 84 2 1377
36 15 1.0000;

36 26 84 1 1133
40 17 1.0000;39 17 1.0000;38 17 1.0000;37 17 1.0000;36 17 1.0000;35 17 1.0000;

36 26 84 2 1377
36 15 1.0000;

36 26 84 3 1377
36 15 1.0000;35 15 1.0000;

37 26 85 1 1133
41 17 1.0000;40 17 1.0000;39 17 1.0000;38 17 1.0000;37 17 1.0000;36 17 1.0000;

37 26 85 2 1133
36 17 1.0000;

37 26 85 3 1194
42 17 1.0000;41 17 1.0000;

37 26 85 4 1377
35 15 1.0000;

37 26 85 5 1438
36 14 1.0000;

38 26 88 1 1133
41 17 1.0000;40 17 1.0000;39 17 1.0000;38 17 1.0000;37 17 1.0000;36 17 1.0000;

38 26 88 2 1438
36 14 1.0000;

39 26 90 1 1133
42 17 1.0000;41 17 1.0000;40 17 1.0000;39 17 1.0000;38 17 1.0000;

39 26 90 2 1194
41 17 1.0000;40 17 1.0000;

39 26 90 3 1255
42 16 1.0000;41 16 1.0000;

39 26 90 4 1438
37 14 1.0000;
```

```
40 26 91 1 1133
43 17 1.0000;42 17 1.0000;41 17 1.0000;40 17 1.0000;39 17 1.0000;38 17 1.0000;

40 26 91 2 1194
43 17 1.0000;42 17 1.0000;41 17 1.0000;

40 26 91 3 1438
43 14 1.0000;42 14 1.0000;41 14 1.0000;40 14 1.0000;39 14 1.0000;38 14 1.0000;37 14 1.0000;

40 26 91 4 1438
40 14 1.0000;39 14 1.0000;38 14 1.0000;37 14 1.0000;

41 26 92 1 1133
42 17 1.0000;41 17 1.0000;40 17 1.0000;

41 26 92 2 1438
44 14 1.0000;

42 26 91 1 1133
45 17 1.0000;44 17 1.0000;43 17 1.0000;42 17 1.0000;41 17 1.0000;

42 26 91 2 1194
41 17 1.0000;40 17 1.0000;

42 26 91 3 1438
44 14 1.0000;

43 26 91 1 1133
45 17 1.0000;44 17 1.0000;43 17 1.0000;42 17 1.0000;

43 26 91 2 1133
43 17 1.0000;42 17 1.0000;

43 26 91 3 1194
43 17 1.0000;42 17 1.0000;

43 26 91 4 1438
44 14 1.0000;

44 26 91 1 1133
47 17 1.0000;46 17 1.0000;45 17 1.0000;44 17 1.0000;43 17 1.0000;42 17 1.0000;

44 26 91 2 1438
45 14 1.0000;44 14 1.0000;

45 26 89 1 1133
47 17 1.0000;46 17 1.0000;45 17 1.0000;44 17 1.0000;43 17 1.0000;

45 26 89 2 1377
45 15 1.0000;44 15 1.0000;

45 26 89 3 1438
44 14 1.0000;

46 26 89 1 1133
48 17 1.0000;47 17 1.0000;46 17 1.0000;45 17 1.0000;44 17 1.0000;

46 26 89 2 1438
45 14 1.0000;

47 26 89 1 1133
49 17 1.0000;48 17 1.0000;47 17 1.0000;46 17 1.0000;

47 26 89 2 1316
46 16 1.0000;45 16 1.0000;

47 26 89 3 1377
46 15 1.0000;45 15 1.0000;44 15 1.0000;

48 26 89 1 1133
50 17 1.0000;49 17 1.0000;48 17 1.0000;47 17 1.0000;46 17 1.0000;

48 26 89 2 1194
46 17 1.0000;

48 26 89 3 1377
47 15 1.0000;46 15 1.0000;45 15 1.0000;

48 26 89 4 1438
45 14 1.0000;

49 26 89 1 1133
52 17 1.0000;51 17 1.0000;50 17 1.0000;49 17 1.0000;48 17 1.0000;

49 26 89 2 1194
49 17 1.0000;48 17 1.0000;

50 26 89 1 1133
51 17 1.0000;50 17 1.0000;49 17 1.0000;48 17 1.0000;

51 26 89 1 1133
54 17 1.0000;53 17 1.0000;52 17 1.0000;51 17 1.0000;50 17 1.0000;

51 26 89 2 1194
54 17 1.0000;53 17 1.0000;

51 26 89 3 1255
54 16 1.0000;
```

Appendix C. Source Code

C.1 Aria Camera Test

```
////////////////////////////////////////////////////////////////////////////
// ACT: Aria Camera Test
// Capt Kevin M. Biggs and Capt Kevin L. Owens
// Program Entry Point
// 31 Jan 05
////////////////////////////////////////////////////////////////////////////

#pragma warning( disable : 4786 )

#include <fstream> #include <iostream> #include <ctime> #include <cstdlib> #include <string>

#include "RmSettings.h" #include "RmSonarMap.h" #include "RmBayesCertaintyGrid.h" #include
"RmSonarMapper.h" #include "RmPioneerController.h" #include "RmServer.h" #include "RmExceptions.h"
#include "include/CameraMapper.h"

bool mapFromRobot( RmSettings &settings, std::ofstream &cameraStream, std::string cameraStreamName,
    std::ofstream  &sonarStream, std::string sonarStreamName, RmSonarMap &grid, bool remote, bool wander );
std::string newLogFiles( std::ofstream  &cameraStream, std::string cameraStreamName,
    std::ofstream  &sonarStream, std::string sonarStreamName, bool reset = false );

// For scaling to grid
RmSettings g_settings;

const std::string DataPath( "../output/" );

/**
 * Demonstrates all the code necessary to connect to and drive a Pioneer robot,
 * either directly or in simulation, and to install a custom action event handler.
 */
void main( int argc, char* argv[] ) {
    std::string cameraStreamName( DataPath + "default" );
    std::ofstream cameraStream;

    // Sonar stuff so can record sonar data at same time as camera data
    std::string sonarStreamName( DataPath + "default" );
    std::ofstream sonarStream;
    RmBayesCertaintyGrid map( &g_settings );
    ////

    const bool remote = true;  //keyboard = false
    const bool wander = false;
    try {
        mapFromRobot( g_settings, cameraStream, cameraStreamName, sonarStream, sonarStreamName,
            map, remote, wander );
    }
    catch( RmExceptions::Exception e ) {
        std::cerr << e;
    }
}

bool mapFromRobot( RmSettings &settings, std::ofstream  &cameraStream, std::string
cameraStreamName,
    std::ofstream  &sonarStream, std::string sonarStreamName, RmSonarMap &grid,
        bool remote, bool wander )
{
    settings.Localize = false; // just get sonar data; localization causes delays

    std::vector<RmActionHandler*> actionHandlers;
    RmSonarMapper sonarMapper( settings, sonarStream, &grid );
    actionHandlers.push_back( &sonarMapper );
    CameraMapper cameraMapper( settings, cameraStream );
    actionHandlers.push_back( &cameraMapper );

    RmPioneerController *robot = NULL;
    ArActionKeydrive *keydriveAction = NULL;

    int totalFrames = 0;

    if ( remote )
    {
        RmServer server( 2000 );
        std::string cmdString;
        bool quit = false;
        while ( (robot == NULL || robot->isRunning()) && !quit )
        {
            // Get remote control command
            cmdString = server.getClientString();
            std::cout << cmdString << "\n";

            switch( cmdString.c_str()[0] )
```

```
            {
                case 'f':
                    if ( keydriveAction ) keydriveAction->up();
                    break;
                case 'b':
                    if ( keydriveAction ) keydriveAction->down();
                    break;
                case 'l':
                    if ( keydriveAction ) keydriveAction->left();
                    break;
                case 'r':
                    if ( keydriveAction ) keydriveAction->right();
                    break;
                case 's':
                    if ( keydriveAction ) keydriveAction->space();
                    break;
                case 'w':
                    if ( robot ) {
                        wander = !wander;
                        robot->setDriveMode(
                            wander ? RmPioneerController::WANDER : RmPioneerController::KEYDRIVE );
                    }
                    break;
                case 'o':
                    if ( robot == NULL ) {
                        server.sendClientString(
                            newLogFiles( cameraStream, cameraStreamName, sonarStream, sonarStreamName ) );
                        robot = new RmPioneerController( wander, !wander, &actionHandlers );
                        keydriveAction = robot->arKeydriveAction();
                    }
                    else {
                        sonarStream.close();
                        cameraStream.close();

                        delete robot;
                        robot = NULL;
                        keydriveAction = NULL;
                    }
                    break;
                case 'd':
                    sonarStream.close();
                    cameraStream.close();
                    server.sendClientString( newLogFiles( cameraStream, cmdString.substr(1),
                        sonarStream, cmdString.substr(1), true ) );
                    break;
                case 'q':
                    sonarStream.close();
                    cameraStream.close();
                    quit = true;
                    if ( robot ) delete robot;
                    robot = NULL;
                    keydriveAction = NULL;
                    break;
            }
        }
    }
    else
    {
        newLogFiles( cameraStream, cameraStreamName, sonarStream, sonarStreamName );
        robot = new RmPioneerController( wander, false, &actionHandlers );
    }
    if ( robot ) delete robot;

    return totalFrames;
}

std::string newLogFiles( std::ofstream  &cameraStream, std::string cameraStreamName,
    std::ofstream  &sonarStream, std::string sonarStreamName, bool reset )
{
    static const char *dataFail = "Unable to open sonar file '%s' or camera file '%s' for write.";
    static const char *dataOkay = "Sonar file '%s' and camera file '%s' opened for write.";
    static int segment = 0;

    if ( reset ) segment = 0;

    char cameraName[255];
    sprintf( cameraName, "%s%s%d.cd", DataPath.c_str(), cameraStreamName.c_str(), ++segment );
    cameraStream.open( cameraName );

    char sonarName[255];
    sprintf( sonarName, "%s%s%d.sd", DataPath.c_str(), sonarStreamName.c_str(), segment );
    sonarStream.open( sonarName );

    char buff[255];
    sprintf( buff, sonarStream.is_open() && cameraStream.is_open() ?
```

```
        dataOkay : dataFail, sonarName, cameraName );
    return std::string( buff );
}
```

C.2 Vision Processing Library

```
////////////////////////////////////////////////////////////////////////////
// Camera.cpp
// Capt Kevin M. Biggs
// Thesis Code
// 31 Jan 05
// Compiled as a .lib for ACT above
////////////////////////////////////////////////////////////////////////////
//  Includes
////////////////////////////////////////////////////////////////////////////
#include <stdio.h> #include <string> #include <iostream> using std::string; #include <stack>
// for pow
#include <math.h>
#include <iostream> #include <fstream>  //output to file using namespace std;
#include <time.h> //for calculating speed
// Image aquisition
#include "C:\Program Files\SRI International\svs31e\src\svsclass.h"
// Utilities
//#include "SegmentImage.h"
#include "..\include\Utilities.h" #include "..\include\Camera.h"
// Intel's OpenCV Libraries
#include "C:\Program Files\OpenCV\cvaux\include\cvaux.h" #include "C:\Program
Files\OpenCV\otherlibs\highgui\highgui.h"
// Vector
//#include <vector>
int g_cameraInitialized = false;

////////////////////////////////////////////////////////////////////////////
//  Function Prototypes
////////////////////////////////////////////////////////////////////////////
bool init(); bool cameraSetup(); svsStereoImage * getImage( bool, bool, char*, char* );
svsStereoImage * getImageFromCamera(); void outputParams( bool, char*, svsStereoImage* ); bool
isDisimilar( int, int, int, int ); int saveArray( unsigned char*, char*, int ); void applyMask(
unsigned char*, unsigned char*, int, int, bool ); void spikeSmooth( unsigned char[] ); int
maxValueIn( unsigned char* ); int minValueIn( unsigned char* ); int compareArrays( unsigned char*,
unsigned char*, unsigned char[] ); void copyArray( unsigned char*, unsigned char* ); bool
noROInear( unsigned char[], int, int, int, int* ); void initArray( unsigned char*, char ); int
findInnerIndex( unsigned char*, unsigned char );; void setBorders( unsigned char*, int ); Plane
stackFloodFill( unsigned char*, int, unsigned char ); void getDisparityLevel( unsigned char*, char
); void variableHSS ( unsigned char*, char );
// Image size constants
#define HEIGHT 240 #define WIDTH 320
svsStereoProcess *st; svsVideoImages *cam;  // Acquisition object
unsigned char _array[HEIGHT*WIDTH]; unsigned char _final[HEIGHT*WIDTH];
// The plane structure contains data about each plane, for map reconstruction
//struct Plane
//{
//  int disparity; // Should be 1-20 (This comes from OpenCv's Birchfield correspondence algorithm)
//  int distance;  // In millimeters (1 inch = 25.4 millimeters)
//  int minX, maxX;
//  int minY, maxY;
//};

//-----------------------------------------------------------------------------------------------------------
// Main
//-----------------------------------------------------------------------------------------------------------
std::vector< Plane > getPlanes( bool quick, bool hw, bool saveToFile, char *saveImageFilename,
                                char *readImageFilename, bool HSS, int spike, bool filter,
                                int mask, int proximity )
{
    int t1;
    int t2;
    std::vector< Plane > planeVector;
    if( !g_cameraInitialized )
```

```
{
    cerr << "Bad ACT, trying to get an image when camera isn't initialized." << endl;
}
// Get an image from hw or file
svsStereoImage *image = new svsStereoImage();
image = getImage( hw, saveToFile, saveImageFilename, readImageFilename );
// We have an image to process
if( image != NULL )
{
    // Create buffer for streamlined output naming convention
    string buffer(readImageFilename);
    // Acquire source color source images
    IplImage* srcLeft = cvLoadImage( (buffer + "-C.bmp").c_str(), 1 );
    IplImage* srcRight = cvLoadImage( (buffer + "-Q.bmp").c_str(), 1 );
    //IplImage* srcLeft = cvLoadImage( "left.jpg" , 1 );
    //IplImage* srcRight = cvLoadImage( "right.jpg", 1 );
    // Convert into proper OpenCV formatting
    IplImage* leftImage = cvCreateImage(cvGetSize(srcLeft), IPL_DEPTH_8U, 1);
    IplImage* rightImage = cvCreateImage(cvGetSize(srcRight), IPL_DEPTH_8U, 1);
    IplImage* depthImage = cvCreateImage(cvGetSize(srcRight), IPL_DEPTH_8U, 1);
    IplImage* cannyL = cvCreateImage(cvGetSize(srcRight), IPL_DEPTH_8U, 1);
    IplImage* cannyR = cvCreateImage(cvGetSize(srcRight), IPL_DEPTH_8U, 1);
    // Convert to grayscale (possibly eliminate this step by reading in B/W images)
    cvCvtColor(srcLeft, leftImage, CV_BGR2GRAY);
    cvCvtColor(srcRight, rightImage, CV_BGR2GRAY);
    /*-----------------------------------------------------------------------------------------------------------
    // Stereo Correspondence
    //-----------------------------------------------------------------------------------------------------------
    // These are the signature values used by birchfield in (his) research
    // Taken from "Depth Discontinuities by Pixel-to-Pixel Stereo" Stanford University Technical
    //    Report STAN-CS-TR-96-1537, July 1996
    // Signature definitions
        // leftImage - left image of stereo pair, rectified grayscale 8-bit image
        // rightImage - right image of stereo pair, rectified grayscale 8-bit image
        // mode - algorithm used to find a disparity (now only CV_DISPARITY_BIRCHFIELD is supported)
        // depthImage - destination depth image, grayscale 8-bit image that codes the scaled disparity, so that the
            // zero disparity (corresponding to the points that are very far from the camera) maps to 0, maximum disparit
            // maps to 255
        // maxDisparity - maximum possible disparity.  the closer the objects to the cameras, the larger value
            // should be specified here.  too big values slow down the processing significantly
        // param1, param2, param3, param4, param5
            // parameters for algorithm
            // param1 - constant occlusion penalty
            // param2 - constant match reward
            // param3 - defines a highly reliable region (set of contiguous pixels whose reliability is at least param3)
            // param4 - defines a moderately reliable region
            // param5 - defines a slightly reliable region
            // if any parameter is omitted the standard Birchfield values are used: (50, 15, 3, 6, 8, 15)
    */
    if( quick )
    {
        cvFindStereoCorrespondence( leftImage, rightImage, CV_DISPARITY_BIRCHFIELD,
            depthImage, 20, 5, 12, 15, 25, 15 );
    }
    else
    {
        t1 = clock(); // Start timing
        cvFindStereoCorrespondence( leftImage, rightImage, CV_DISPARITY_BIRCHFIELD,
            depthImage, 20, 5, 12, 15, 25, 15 );
        t2 = clock(); // End timing
        printf("%.4lf seconds of correspondence processing using OpenCV.",
            (t2-t1)/(double)CLOCKS_PER_SEC); cout << endl;
    }
    // Reference the image data (this is the grayscale value that should be between 0 and 255)
    unsigned char* array1 = (unsigned char*)depthImage->imageData;
    unsigned char* array2 = (unsigned char*)calloc(sizeof(unsigned char),HEIGHT*WIDTH);
    unsigned char* array3 = (unsigned char*)calloc(sizeof(unsigned char),HEIGHT*WIDTH);
    // Initialize the arrays to second argument
    initArray( array2, 0 );
    initArray( array3, 0 );
    // These values are extracted for later use in proper image scaling
    int max = maxValueIn( array1 );
```

```
int min = minValueIn( array1 );

if( !quick )
{
    cout << "max value = " << max << " min value = " << min << endl;
    saveArray( array1, "OriginalDisparityMap.pnm", max );
}

// Data values below mask value will be nullified is within +/- vertical pixel proximity
// Normal range of values are 0 - 255; however max is 20 from open cv (matlab scales to max of 255)

int adjustedPixels = -1;

if( !quick )
{
    cout << "Mask = " << mask << endl;
    cout << "Proximity = " << proximity << endl;
}

// Do this copy so we can further process data while retaining orignal data
copyArray( array1, array2 );

// Variable Horizontal Spike Smooth (vHSS)

    if( quick )
    {
        variableHSS( array2, spike );
    }

    // vHSS vs Original comparison data
    else if( !quick && HSS )
    {
        t1 = clock(); //Start timing
        variableHSS( array2, spike );
        t2 = clock(); // End timing
        printf("%.4lf seconds of variableHSS processing.", (t2-t1)/(double)CLOCKS_PER_SEC); cout << endl;

        saveArray( array2, "vHSS.pnm", max );
        adjustedPixels = compareArrays( array1, array2, array3 );
        saveArray( array3, "OrigninalVsVHSS.pnm", max );
        cout << "adjusted pixels = " << adjustedPixels << " or "
            << (float)adjustedPixels/(HEIGHT*WIDTH) * 100 << "%." << endl;
    }

// Smart mask

    if( quick )
    {
        applyMask( array2, array1, mask, proximity, true );
    }

    else if( !quick && filter )
    {
        t1 = clock(); //Start timing
        applyMask( array2, array1, mask, proximity, true );
        t2 = clock(); // End timing
        printf("%.4lf seconds of smartMask filter processing.", (t2-t1)/(double)CLOCKS_PER_SEC); cout << endl;

        saveArray( array1, "smartMask.pnm", max );
        adjustedPixels = compareArrays( array1, array2, array3 );
        saveArray( array3, "vHSSvsSmartMask.pnm", max );
        cout << "adjusted pixels = " << adjustedPixels << " or "
            << (float)adjustedPixels/(HEIGHT*WIDTH) * 100 << "%." << endl;
    }

// Set column borders to 0 to ensure proper boundary detection for Stack Flood Fill method
setBorders( array1, 0 );

// Copy to global variable
copyArray( array1, _array );

// The upper bound of the "char" space is used for an initializer since
// 0 = empty
// 1-20 represent values that still need to be processed
// 21-255 represent planes (234 max) that have been visited and is used as a marker
    // Total planes can be calculated by computing 255 - objectsFound.
unsigned char objectsFound = 255;

// This is the pertinent number of disparity levels we need to explore based on variable mask value
// Example if the max value was 20 and the mask value was 15, 6 layers would be iteratively
// examined (20, 19, 18, 17, 16, and 15).  All others are ignored.
char disparityLevels = max - mask + 1;

// For each of the disparity levels from max to min
for( int i = 0; i < disparityLevels; ++i )
{
    int level = max - i;

    // Get specific disparity level
    getDisparityLevel( array1, level );
```

```
            int innerIndex = findInnerIndex( array1, objectsFound ); //63635 backpack //42020 hand
            while( innerIndex != -1 && objectsFound != 20 )
            {
                if( !quick ) cout << "Initial inner index found= " << innerIndex << endl;
                // Stack flood fill from innerIndex coordinate
                //      Global "_final" array will be modified with object found values
                planeVector.push_back( stackFloodFill( array1, innerIndex, objectsFound-- ) );
                innerIndex = findInnerIndex( array1, objectsFound );
            }
            // Copy original matrix to array1 for layer recovery
            copyArray( _array, array1 );
        }
        if( !quick )
        {
            copyArray( _final, array2 );
        }
        //cout << "planeVector size is: " << static_cast <unsigned int>(planeVector.size()) << endl;
        std::vector< Plane >::const_iterator iterator = planeVector.begin();
        //get and save aria pose information (shouldn't change inside call, only the planes it finds)
        //static char buff[35]; //18
        while( iterator != planeVector.end() )
        {
            if( !quick )
            {
                // This code draw a box around the edges of each object
                /* minX, minY   ---------   maxX, minY
                        |                       |
                        |                       |
                        |                       |
                        |                       |
                    minX, maxY  ---------   maxX, maxY */

                // Top line
                for( int i = iterator->minY*WIDTH + iterator->minX; i < iterator->minY*WIDTH + iterator->maxX; ++i )
                {
                    array2[i] = 30;
                }
                // Bottom line
                for( i = iterator->maxY*WIDTH + iterator->minX; i < iterator->maxY*WIDTH+1 + iterator->maxX; ++i )
                {
                    array2[i] = 30;
                }
                // Left line
                for( i = iterator->minY*WIDTH + iterator->minX; i < iterator->maxY*WIDTH + iterator->minX; i+=320 )
                {
                    array2[i] = 30;
                }
                // Right line
                for( i = iterator->minY*WIDTH + iterator->maxX; i < iterator->maxY*WIDTH + iterator->maxX; i +=320 )
                {
                    array2[i] = 30;
                }
            }
            ++iterator;
        }
        if( !quick )
        {
            saveArray(array2, "boxedFinal.pnm", 30);
            cout << "Total disjoint objects found is: " << 255 - objectsFound << endl;
            saveArray( _final, "Final.pnm", 30 );
            //saveArray( _final, "Matlab.pnm", 255 );
            // For matlab convex hull processing; must use 255 to get correct object identification
        }
        free(array2);
        free(array3);
        // Return vector structure
        return planeVector;
}
else
{
    cout << "main(): Error getting an image.  Exiting..." << endl;
    Utilities::PauseForUserAcknowledgement();
```

```
        return planeVector;  //will be empty
    }
}//end main
//-----------------------------------------------------------------------------------------------------
// Functions
//-----------------------------------------------------------------------------------------------------
// Start initialization process
bool init() {
    if( g_cameraInitialized )
    {
        cerr << "Calling initialized when already initialized." << endl;
    }
    //  If camera setup fails
    if( !cameraSetup() )
    {
        cerr << "init(): Cannot initialize camera." << endl;
        Utilities::PauseForUserAcknowledgement();
        return false;
    }
    // Stereo processor
    st = new svsStereoProcess();
    g_cameraInitialized = true;
    return true; // All good
}
//-----------------------------------------------------------------------------------------------------
bool cameraSetup() {
    cam = getVideoObject();
    bool good = cam->Open();
    if( !good )
    {
        cerr << "cameraSetup(): Cannot Open Framegrabber Connection" << endl;
        return false;
    }
    // Calibration filename
    char * calibrationFile = "3711b.ini";
    good = cam->ReadParams( calibrationFile );
    if(!good)
    {
        string buffer(calibrationFile);
        cerr << "cameraSetup(): Cannot Read Calibration File " + buffer << endl;
        return false;
    }
    good = cam->SetSize( WIDTH, HEIGHT );
    if(!good)
    {
        cerr << "cameraSetup(): Cannot Set Size to " << WIDTH << "x" << HEIGHT << endl;
        return false;
    }
    // Signature is decimation (sub-sampling), binning (averaging of pixels)
    // Decimation values can be 1, 2, or 4 pixels
    // Binning values can be 1 or 2
    // Zach has this set to (0, 2)?
    cam->SetSample( 1, 2 );
    // Signature is left color and right color
    good = cam->SetColor( true, true );
    if( !good )
    {
        cerr << "cameraSetup(): Cannot Set Camera to Color Acquistion" << endl;
        return false;
    }
    good = cam->Start();
    if( !good )
    {
        cerr << "cameraSetup(): Cannot Start Image Acquistion" << endl;
        return false;
    }
    // Avoid the ghosting at the beginning of the acquistion.
    // This allows for proper camera warm up.
    svsStereoImage *flushImage;
    flushImage = new svsStereoImage();
    for( int q = 0; q < 40; q++ )
    {
        flushImage = cam->GetImage( 500 );
```

```
    }
    return true;
}
//-----------------------------------------------------------------------------------------------------
svsStereoImage * getImageFromCamera() {
    // Will try to get a camera image for up to 500 milliseconds before it returns
    // If successful it returns an svsStereoImage object containing the image; otherwise,
    // it returns NULL
    cam->SetRect( true );
    svsStereoImage *hold = cam->GetImage( 500 );  // Store acquisition object into a stereo image
    return ( hold );
}
//-----------------------------------------------------------------------------------------------------
svsStereoImage * getImage( bool hw, bool saveToFile, char *saveImageFilename, char
    *readImageFilename)
{
    // Create new svsStereoImage object
    svsStereoImage *image = new svsStereoImage();
    // Try to get image from camera
    if( hw )
    {
        // Check to see if initialized
        if ( !g_cameraInitialized )
        {
            return NULL;
        }
        // Camera initialized, continue
        else
        {
            image = getImageFromCamera();
            if( image == NULL )
            {
                cerr << "getImage(): No image acquired." << endl;
                return NULL;
            }
            // Save image (5 different files) to file
                // Saves left and right in black and white (basename-L.bmp and basename-R.bmp)
                // Saves left and right in color (basename-C.bmp and basename-Q.bmp)
                // Also, saves a parameter file (basename.ini)
            if( saveToFile )
            {
                image->SaveToFile( saveImageFilename );
            }
        }
    }
    // Read base image from file
    else
    {
        // Read an existing image from the specified file name
        bool good = image->ReadFromFile( readImageFilename );
        if( !good )
        {
            cerr << "getImage(): Cannot read from the specified filename: "
                << readImageFilename << endl;
            Utilities::PauseForUserAcknowledgement();
            return NULL;
        }
    }
    return image;
}
//-----------------------------------------------------------------------------------------------------
// Is disimilar
//      Returns if the current pixel within the threshold of either the upper or lower adjacent pixel
//      NOTE:  A helper to the spike smooth method
//-----------------------------------------------------------------------------------------------------
bool isDisimilar( int current, int upper, int lower, int threshold ) {
    return ( !((current >= upper-threshold && current <= upper+threshold) ||
                (current >= lower-threshold && current <= lower+threshold)) );
}
//-----------------------------------------------------------------------------------------------------
// Save array
//      Saves an unsigned character array in GIMP format to given filename with max threshold
//-----------------------------------------------------------------------------------------------------
int saveArray( unsigned char* array, char* filename, int max ) {
    // Create an ofstream object named myFile based upon fileName string in signature
    ofstream myFile( filename );
    // Ensure file opened properly
```

```
    if( !myFile )
    {
        cout << "Error opening output file." << endl;
    }
    // File opened ok, continue
    else
    {
        // Header file formatting for easy GIMP processing
        myFile << "P2\n" << "# CREATOR: The GIMP's PNM Filter Version 1.0\n" <<
            WIDTH << " " << HEIGHT << "\n" << max << endl;
        // Output data to file
        for( int i = 0; i < HEIGHT * WIDTH; ++i )
        {
            myFile << (int)array[i] << " ";
        }
        // Close file
        myFile.close();
    }
    return max;
}
//-------------------------------------------------------------------------------------------------
// Mask pixels
//      SMART MASK
//      Pixel values below mask will be erased or set to zero if not between similar values above and
//      below proximity pixels of current position (background noise).  Otherwise, pixel gets the max
//      value found in vertical proximity in the up and down directions.  This acts as a melding
//      function attaching two like "blobs" (accomplished in noROInear method).
//      STRAIGHT MASK
//      Masks all pixel values less than mask argument regardless of what might be around it.
//      NOTE: Last Boolean argument determines whether or not a smart mask is applied.
//-------------------------------------------------------------------------------------------------
void applyMask( unsigned char array[], unsigned char array2[], int mask, int proximity, bool smart
) {
    // Smart mask will merge open areas of vertical proximity if values greater than mask are on both sides
    if( smart )
    {
        int fillValue;
        for( int i = WIDTH; i < HEIGHT * WIDTH - WIDTH; ++i )
        {
            if( array[i] < mask )
            {
                if( noROInear( array, i, proximity, mask, &fillValue ) )
                {
                    array2[i] = 0;  // Mask value by setting to 0
                }
                else
                {
                    array2[i] = fillValue;
                }
            }
            else
            {
                array2[i] = array[i];
            }
        }
    }
    // No smart mask
    else
    {
        for( int i = 0; i < HEIGHT * WIDTH; ++i )
        {
            if( array[i] < mask )
            {
                array2[i] = 0;
            }
            else
            {
                array2[i] = array[i];
            }
        }
    }
}
//-------------------------------------------------------------------------------------------------
// No Region of Interest Near
//      Determines if given pixel is beyond the specified vertical proximity from pixel with value
//      greater than mask both upward and downward.  Returns true if pixel is a "loner."
//-------------------------------------------------------------------------------------------------
```

```
bool noROInear(unsigned char array[], int index, int proximity, int mask, int *value ) {
    int searchUp, searchDown;
    bool up = true;  //ROI upward?
    bool down = true;//ROI downward?
    int upColor = 0;
    int downColor = 0;
    int currentRow = index / WIDTH;  //used for boundary testing
    // START - Ensure matrix boundaries not breached
    searchUp = currentRow < proximity ? currentRow : proximity;
    if( currentRow > HEIGHT - proximity )
    {
        searchDown = HEIGHT - currentRow;
    }
    else
    {
        searchDown = proximity;
    }
    // END - Ensure matrix boundaries not breached
    for( int i = 1; i < searchUp; ++i )
    {
        if( array[index - i*WIDTH] > mask )
        {
            up = false;
            // Save max value found
            if( array[index - i*WIDTH] > upColor )
            {
                upColor = array[index - i*WIDTH];
            }
        }
    }
    for( i = 1; i < searchDown; ++i )
    {
        if( array[index + i*WIDTH] > mask )
        {
            down = false;
            // Save max value found
            if( array[index + i*WIDTH] > downColor )
            {
                downColor = array[index + i*WIDTH];
            }
        }
    }
    // If both are false, then noROINear is true, so update color
    if( !(up && down) )
    {
        *value = max( upColor, downColor );
    }
    return up || down;
}
//-----------------------------------------------------------------------------------------------
// Max value in
//      Returns the maximum value in the argument array
//-----------------------------------------------------------------------------------------------
int maxValueIn( unsigned char* array ) {
    int max = -1;  // Sentinel value
    for( int i = 0; i < HEIGHT*WIDTH; ++i )
    {
        int value = (int)array[i];
        if( value > max )
        {
            max = value;
        }
    }
    return max;
}
//-----------------------------------------------------------------------------------------------
// Max value in
//      Returns the maximum value in the argument array
//-----------------------------------------------------------------------------------------------
int minValueIn( unsigned char* array ) {
    int min = 256;  // Sentinel value
    for( int i = 0; i < HEIGHT*WIDTH; ++i )
    {
        int value = (int)array[i];
        if( value < min && value > 0 )
```

```
        {
            min = value;
        }
    }
    return min;
}
//-----------------------------------------------------------------------------------------------------
// Compare arrays
//      Returns the differce of the two input arrays (one and two) in result
//-----------------------------------------------------------------------------------------------------
int compareArrays( unsigned char* one, unsigned char* two, unsigned char result[] ) {
    int total = -1; // Sentinel value
    for( int i = 0; i < HEIGHT*WIDTH; ++i )
    {
        // Same values
        if( one[i] == two[i] ){
            result[i] = 0;
        }
        // Different values
        else
        {
            result[i] = 255;
            total++;
        }
    }
    return total;
}
//-----------------------------------------------------------------------------------------------------
// Copy array
//      Simply copies the origin array to destination array
//-----------------------------------------------------------------------------------------------------
void copyArray( unsigned char* origin, unsigned char* destination ) {
    for( int i = 0; i < HEIGHT*WIDTH; ++i )
    {
        destination[i] = origin[i];;
    }
}
//-----------------------------------------------------------------------------------------------------
//  Init(ialize) array does just that.
//-----------------------------------------------------------------------------------------------------
void initArray( unsigned char* array, char initializer ) {
    for( int i = 0; i < HEIGHT*WIDTH; ++i )
    {
        array[i] = initializer;
    }
}
//-----------------------------------------------------------------------------------------------------
// Find inner index
//      Returns an internal pixel of an object.
//      NOTE:  This is used by stack flood fill method as a seed.
//-----------------------------------------------------------------------------------------------------
int findInnerIndex( unsigned char* array, unsigned char objects ) {
    // For every row except first and last (buffer to look up and down)
    // 1 to 238
    for( int i = 1; i < HEIGHT - 2; ++i )
    {
        // For every column in each row except first and last (buffer to look left and right)
        // 1 to 318
        for( int j = 1; j < WIDTH - 2; ++j )
        {
            int current = i*WIDTH + j;
            char currentValue = array[current];
            // Object has yet to be traversed (in other words, cell is empty or already visited
            if( currentValue != 0 && currentValue < objects )
            {
                // Also, is this object's size at least 3 by 3?
                if( array[current-1] == currentValue &&     // left
                    array[current+1] == currentValue &&     // right
                    array[current-320] == currentValue &&   // up
                    array[current+320] == currentValue )    // down
                {
                return current;
                }
            }
        }
    }
    return -1;  // None found, return sentinel value
}
```

```
//-------------------------------------------------------------------------------------------------
// Set Borders
//      Sets the outermost pixels of the matrix to the value argument.  Useful for
//      boundary detection of matrix.
//-------------------------------------------------------------------------------------------------
void setBorders( unsigned char* array, int value ) {
    // Set top and bottom rows
    for( int i = 0; i < HEIGHT; i+=HEIGHT-1 )
    {
        for( int j = 0; j < WIDTH; ++j )
        {
            array[i*WIDTH+j] = value;
        }
    }
    // For each row
    for( i = 0; i < HEIGHT; ++i )
    {
        // Set first column
        array[i*WIDTH] = value;
        // Set last column
        array[i*WIDTH+WIDTH-1] = value;
    }
}
//-------------------------------------------------------------------------------------------------
// Stack flood fill(find min and max Cartesian X and Y coordinates)
//      Efficiently fills contiguous pixels in argument array with a unique value, given an inner index
//      and threshold(s) to bypass (objectsFound).  Does scan line fill based on seed, then
//      finds all upper and lower left points (can have more that one for each) until object
//      --dilineated by borders--is fully filled.
//      NOTE:  This data is used to extract a rectangular plan which will define the object to a map
//      LESSONS LEARNED:  A simple Four Fill algorithm was implemented prior to the stack
//      implementation.  However, it was overwhelmed by a large object (approx. 100 x 100 pixels)
//      that led to excessive recursions and caused a stack overflow (bad mojo).
//      NOTE: Piggy-backed on this funtion to find each planes min and max Cartesian X and Y coordinates
//      for recatangle tracing.
//-------------------------------------------------------------------------------------------------
Plane stackFloodFill( unsigned char* array, int innerIndex, unsigned char objectsFound ) {
    typedef stack<int> Stack;
    Stack st;
    // Min/max Cartesian coordinates are saved for rectangle reconstruction
    int minX = WIDTH;
    int maxX = -1;
    int minY = HEIGHT;
    int maxY = -1;
    // Save intensity value
    int intensity = array[innerIndex];
    // Prime the process
    st.push(innerIndex);
    // While we have a seed pixel to process
    while( !st.empty() )
    {
        // Go left from seed and mark (This should only be done once for original seed)
        int left = st.top(); // Origin point
        // We want to stop at a border (either background or other object)
        while( array[left] != 0 )
        {
            array[left] = objectsFound;  // Mark as traversed
            _final[left] = objectsFound;
            left--;
        }
        // Save smallest Cartesian X value
        if( left % WIDTH < minX )
        {
            minX = left % WIDTH;
        }
        // Go right from seed and mark
        int right = st.top(); // Origin point
        while( array[right] != 0 )
        {
            array[right] = objectsFound;  // Mark as traversed for this method's progress
            _final[right] = objectsFound;  // Also mark final product
            right++;
        }
```

```
    // Save largest Cartesian X value
    if( right % WIDTH > maxX )
    {
        maxX = right % WIDTH;
    }
    // Pop top item from stack
    st.pop();
    // Find all upper left-most cells
    for( int i = left; i < right; ++i )
    {
        // Look above for an open unexplored cell
        int up = array[i-WIDTH];
        if( up != 0 && up != objectsFound )
        {
            // Need to explore left to find left edge of row
            int upSeed = i-WIDTH;
            // Save smallest Caretsian Y value
            if( upSeed / WIDTH < minY )
            {
                minY = upSeed / WIDTH;
            }
            while( array[upSeed-1] != 0 )
            {
                upSeed--;  // Keep looking left
            }
            // Save smallest Cartesian X value
            if( upSeed % WIDTH < minX )
            {
                minX = upSeed % WIDTH;
            }
            // Push onto stack
            st.push(upSeed);
        }
    }
    // Find all lower left-most cells
    for( i = left; i < right; ++i )
    {
        // Look above for an open unexplored cell
        int down = array[i+WIDTH];
        if( down != 0 && down != objectsFound )
        {
            // Need to explore left to find left edge of row
            int downSeed = i+WIDTH;
            // Save largest Caretsian Y value
            if( downSeed / WIDTH > maxY )
            {
                maxY = downSeed / WIDTH;
            }
            while( array[downSeed-1] != 0 )
            {
                downSeed--;  // Keep looking left
            }
            // Save smallest Cartesian X value
            if( downSeed % WIDTH < minX )
            {
                minX = downSeed % WIDTH;
            }
            // Push onto stack
            // Efficiency TODO: only push unique seeds (same for the upSeed above)
            st.push(downSeed);
        }
    }
}// End while stack not empty
// Distance model based on disparity intensity
double x = 0;  // Initalize
switch(intensity)
{
    // Disparity to feet model
    case 1:  x = 10.6; break;
    case 2:  x = 10.2; break;
    case 3:  x = 10; break;
    case 4:  x = 9.6; break;
    case 5:  x = 9.2; break;
```

```
        case 6:  x = 8.8; break;
        case 7:  x = 8.4; break;
        case 8:  x = 7.8; break;
        case 9:  x = 7.4; break;
        case 10:  x = 7; break;
        case 11:  x = 6.6; break;
        case 12:  x = 6.2; break;
        case 13:  x = 5.8; break;
        case 14:  x = 5.4; break;
        case 15:  x = 5; break;
        case 16:  x = 4.6; break;
        case 17:  x = 4.2; break;
        case 18:  x = 3.8; break;
        case 19:  x = 3.4; break;
        case 20:  x = 3; break;
        default: cout << "Switch statement value out of valid scope in StackFloodFill" << endl;
    }

    // Convert x value (representing feet) to millimeters.  1 inch = 25.4 mm
    int distanceInMillimeters = static_cast <int>(x * 12 * 25.4);  // Static cast = (int)

    // Construct a plane object
    Plane myPlane;
    myPlane.minX = minX + 1;
    myPlane.maxX = maxX - 1;
    myPlane.minY = minY;
    myPlane.maxY = maxY;
    myPlane.disparity = intensity;
    myPlane.distance = distanceInMillimeters;

    return myPlane;
}
//-----------------------------------------------------------------------------------------------
// Get disparity level
//      Returns an array of the specified disparity level only, all other values are zereod
//      NOTE: an indirect helper for stack flood fill, in that the array is sent to this method,
//      processed then saved into a final array in discrete disparity layers
//      This helps in the deliniation of overlapping objects of different disparity
//      EFFICIENCY TODO:  Instead of passing in an array and wiping all the disparity
//      levels that differ from argument level, along with the argument array, pass in an
//      empty array and fill with matching disparity level.  This should minimize "writes."
//-----------------------------------------------------------------------------------------------
void getDisparityLevel( unsigned char* array, char level ) {
    for( int i = 0; i < HEIGHT*WIDTH; ++i )
    {
        if( array[i] != level )
        {
            array[i] = 0;
        }
    }
}
//-----------------------------------------------------------------------------------------------
//  Variable Horizontal Spike Smooth (HSS)
//      Eliminates horizontal spikes less than or equal to argument "spike" in array.
//      Scans vertically through the matrix of pixel values to determine size of horizontal
//      spikes.  This is done by maintaining start points of a color band and subsequently
//      looking for a different/transitionary color.  Once this transition is detected, we
//      examine the size of the spike and compare it to the threshold given in argument
//      "spike" and take the appropriate action (either erase or leave alone).
//      NOTE: Vertical traversal with saved start point saves time by visiting cells more
//      efficiently as compared to the horizonal scans of method spike smooth--regardless of
//      the "spike" value the same number of cells are visited.
//
//      TODO: Spikes on bottom of image aren't erasing for the hand, backpack, wall pic
//-----------------------------------------------------------------------------------------------
void variableHSS ( unsigned char* array, char spike ) {
    int count = 0;      // Holds contiguous cell count
    int color = -1;     // Holds color of previous start cell
    int start = -1;     // Holds start location of first non-black cell

    // For every column, c
    for( int c = 0; c < WIDTH; ++c )
    {
        // For every row, r, in c
        for( int r = 0; r < HEIGHT; ++r )
        {
            // For efficient access to index multiple times below
            int index = r*WIDTH+c;

            // Check to see how close we are to the bottom row of the image

            // We are ok, there are enough rows below us to constitute a significant spike
```

```
            //if( HEIGHT-1-r > spike )  // this line leaves in horizontal spikes near the
            //  bottom of the image
            if( HEIGHT-r >= spike )
            {
                // First occurance of a color preceded by black
                if( array[index] != 0 && count == 0 )
                {
                    color = array[index];
                    start = index;
                    count++;
                }
                // Subsequent occurance of the last detected color
                else if( array[index] == color )
                {
                    count++;
                }
                // Either encountered a black value or a value other than the last detected color
                else
                {
                    // If we have accumulated a count value
                    if( count != 0 )
                    {
                        // Count is not bigger than spike, erase
                        if( count <= spike )
                        {
                            // Erase column of cells from start cell to one above current cell
                            for( int k = start; k < index; k+=WIDTH )
                            {
                                //For debugging
                                if( start == -1 )
                                {
                                    cerr << "Trying to erase an invalid start index." << endl;
                                }
                                else
                                {
                                    array[k] = 0;
                                }
                            }
                        }
                        // If detected color was black
                        if( array[index] == 0 )
                        {
                            count = 0;
                            color = -1;
                            start = -1;
                        }
                        // If detected color was something other than black
                        else
                        {
                            count = 1;
                            color = array[index];
                            start = index;
                        }
                    }
                }
            }
            // Not enough space at bottom and no previous history of color, erase pixel
            else if( HEIGHT-1-r <= spike && count == 0 )
            {
                array[index] = 0;
            }
        }
    }
}
```

Bibliography

1. Allen, P., I. Stamos, A. Gueorguiev, E. Gold, and P. Blaer. "AVENUE: Automated site modeling in urban environments". *In Proceedings of 3DIM 2001: International Conference on 3D Digital Imaging and Modeling*, 2001.
2. Baker, C., A. Morris, D. Ferguson, S. Thayer, C. Whittaker, Z. Omohundro, C. Reverte, W. Whittaker, D. Hahnel, and S. Thrun. "A campaign in autonomous mine mapping". 2:2004–2009, 2004.
3. Ballard, Dana and Christopher Brown. *Computer Vision*. Prentice Hall, 1982.
4. Belhumeur, P. and D. Mumford. "A Bayesian treatment of the stereo correspondence problem using half-occluded regions". *In CVPR*, 506–512. 1992.
5. Birchfield, Stan and Carlo Tomasi. "Depth Discontinuities by Pixel-to-Pixel Stereo". *Proc. of the 1998 IEEE International Conference on Computer Vision*. 1998.
6. Borenstein, J. and B. Everett. *Navigating Mobile Robots: Systems and Techniques*. A. K. Peters, Ltd., Wellesley, MA, 1996.
7. Borenstein, J. and Y. Koren. "The vector field histogram - fast obstacle avoidance for mobile robots". 7(3):278–288, June 1991.
8. Bresenham, J. "Algorithm for computer control of a digital plotter". *IBM Systems Journal*, 4(1):23–30, 1965.
9. Brown, D.C. *Proc. Sixth International Conf. Computer Vision*, 855–866.
10. Cox, I., S. Hingorani, S Rao, and B. Maggs. "A maximum likelihood stereo algorithm". *Comp. Vision and Image Understanding*, 542–567. 1996.
11. El-Hakim, S., P. Boulanger, F. Blais, and J.-A. Beraldin. "Sensor-based creation of indoor virtual environment models". *In Proceedings of VSMM'97: IEEE International Conference on Virtual Systems and Multimedia*. 1997.
12. Faig, W. "Calibration of Close-Range Photogrammetry Systems: Mathematical Formulation". 41(12), 1975.
13. Faugeras, O and G Toscani. "The Calibration Problem for Stereo". *Proc. IEEE Conf. Computer Vision and Pattern Recognition*, 15–20. June 1986.
14. Faugeras, Oliver. *Three-Dimensional Computer Vision*. The MIT Press, 1993.
15. Geiger, D., B. Ladendorf, and A. Yuille. "Occlusions and binocular stereo". *International Journal of Computer Vision*, 221–226. 1995.
16. Gennery, D. "Stero-Camera Calibration". *Proc. 10th Image Understanding Workshop*, 101–108. 1979.

17. Gueorguiev, A., P. Allen, E. Gold, and P. Blaer. "Design, architecture, and control of a mobile site modeling robot". *In Proceedings of ICRA'00: IEEE International Conference on Robotics and Automation*, 2003.

18. Intel. "Open Source Computer Vision Library". *Intel Research*, 2004. URL `www.intel.com/research/mrl/research/opencv/index.htm`.

19. International, SRI. *Small Vision System*, 2004.

20. Intille, S. and A. Bobick. "Disparity-space images and large occlusion stereo". *In Proc. of the 3rd European Conf. on Comp. Vision*, 179–186. 1994.

21. Ivo, Ihrke. *Digital elevataion mapping using stereoscopic vision*. Master's thesis, Royal Institute of Technology, 2001.

22. Kang, S., J. Webb, C. Zitnick, and T. Kanade. *An Active Multibaseline Stereo System with Real-Time Image Acquisition*. Technical Report CMU-CS-1994-167, Carnegie Mellon University, 1994.

23. Kortenkamp, D., R. Bonasso, and R. Murphy. *AI-based Mobile Robots: Case studies of successful robot systems*. Technical report, MIT, Cambridge, MA, 1998.

24. Lui, Y., R. Emery, D. Chakrabarti, W. Bugard, and S. Thrun. "Using EM to learn 3D models with mobile robots". *In Proceedings of the International Conference on Machine Learning (ICML)*. 2001.

25. Martin, C. and S. Thrun. "Real-time acquisition of compact volumetric maps with mobile robots". 2002. URL `citeseer.ist.psu.edu/540668.html`.

26. Military.com. "Guided Bomb Unit-28". 2005. URL `http://www.military.com`.

27. Moravec, H. "Sensor fusion in certainty grids for mobile robots", 1988.

28. Moravec, H. *Robot spatial perception by stereoscopic vision and 3D grids*. Technical report, The Robotics Institute, CMU, 1996.

29. Murray, Don and James J. Little. "Using Real-Time Stereo Vision for Mobile Robot Navigation". *Autonomous Robots*, 8(2):161–171, 2000. URL `citeseer.ist.psu.edu/murray00using.html`.

30. President George W. Bush. "Address to a Joint Session of Congress and the American People". *United States Capitol*, 2001. URL `http://www.whitehouse.gov/news/releases/2001/09/20010920-8.html`.

31. Saez, J.M. and F. Escolano. "A global 3D map-building approach using stereo vision". 2:1197–1202, 2004.

32. U.S. Department of State, Office of the Historian. "Significant Terrorist Incidents, 1961-2003: A Brief Chronology". *Historical Backgound*, 2003. URL `http://www.state.gov.r/pa/ho/pubs/fs/5902.htm`.

33. Thrun, S., W. Burgard, and D. Fox. "A real-time algorithm for mobile robot mapping with applications to multi-robot and 3D mapping". *IEEE International Conference on Robotics and Automation*. 2000.

34. Thrun, S., D. Hahnel, D. Ferguson, M. Montemerlo, R. Triebel, W. Burgard, C. Baker, Z. Omohundro, S. Thayer, and W. Whittaker. "A system for volumetric robotic mapping of abandoned mines". 3:4270–4275, 14-19 September 2003.

35. Tsai, R.Y. "A Versatile Camera Calibration Technique for High-Accuracy 3D Vision Metrology Using Off-the-Shelf TV Cameras and Lenses". 3(4):323–334, August 1987.

36. Vergano, Dan. "Using technology to find al-Qaeda's caves". *USA Today*, 2001.

37. Weng, J, J. Cohen, and M. Herniou. "Camera Calibration with Distortion Models and Accuracy Elevation". 14(10):965–980, October 1992.

38. Zhang, Zhengyou. "A Flexible New Technique for Camera Calibration". 22(11), November 2000.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 074-0188

1. REPORT DATE *(DD-MM-YYYY)*	**2. REPORT TYPE**	**3. DATES COVERED** *(From – To)*
21-03-2005	Master's Thesis	Jan 2004 – Mar 2005

4. TITLE AND SUBTITLE

Real-Time Mapping Using Stereoscopic Vision Optimization

5a. CONTRACT NUMBER

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)

Biggs, Kevin M., Captain, USAF

5d. PROJECT NUMBER

5e. TASK NUMBER

5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAMES(S) AND ADDRESS(S)
Air Force Institute of Technology
Graduate School of Engineering and Management (AFIT/EN)
2950 Hobson Way
WPAFB OH 45433-7765

8. PERFORMING ORGANIZATION REPORT NUMBER

AFIT/GCS/ENG/05-03

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Dr. Mikel M. Miller AFRL/SNRP
2241 Avionics Circle
WPAFB OH 45433-7333
(937) 255-6127 ext. 4274

10. SPONSOR/MONITOR'S ACRONYM(S)

11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

13. SUPPLEMENTARY NOTES

14. ABSTRACT

This research focuses efficient methods of generating 2D maps from stereo vision in real-time. Instead of attempting to locate edges between objects, we make the assumption that the representative surfaces of objects in a view provide enough information to generate a map while taking less time to locate during processing. Since all real-time vision processing endeavors are extremely computationally intensive, numerous optimization techniques are applied to allow for a real-time application: horizontal spike smoothing for post-disparity noise, masks to focus on close-proximity objects, melding for object synthesis, and rectangular fitting for object extraction under a planar assumption. Additionally, traditional image transformation mechanisms such as rotation, translation, and scaling are integrated. Results from our research are an encouraging 10Hz with no vision post processing and accuracy up to 11 feet. Finally, vision mapping results are compared to simultaneously collected sonar data in three unique experimental settings.

15. SUBJECT TERMS
Vision mapping, stereo vision, passive mapping, robot mapping, real-time mapping, planar assumption

16. SECURITY CLASSIFICATION OF:			**17. LIMITATION OF ABSTRACT**	**18. NUMBER OF PAGES**	**19a. NAME OF RESPONSIBLE PERSON**
REPORT	**ABSTRACT**	**c. THIS PAGE**			Gilbert L. Peterson, Dr., USAF (ENG)
U	U	U	UU	99	**19b. TELEPHONE NUMBER** *(Include area code)* (937) 255-6565, ext 4281

Standard Form 298 (Rev: 8-98)
Prescribed by ANSI Std. Z39-18