I, Donald W. Rosselot, hereby submit this work as part of the requirements for the degree of:
Master of Science
in:
Electrical Engineering
It is entitled:
Processing real-time stereo video in disparity space for obstacle mapping

This work and its defense approved by:
Chair: Dr. Arthur Helmicki
Dr. Ernest Hall
Dr. C. Y. Han
Processing real-time stereo video in disparity space for obstacle mapping

A thesis submitted to the
Division of Graduate Studies and Advanced Research
Of the University of Cincinnati

In partial fulfillment of the
Requirements for the degree of

MASTER OF SCIENCE

In the Department of Electrical & Computer Engineering
And Computer Science
Of the College of Engineering

2005

By
Donald Rosselot

B.S., Electrical Engineering, University of Cincinnati, 1990

Thesis Advisor and Committee Chair: Arthur Helmicki, Ph.D.
Committee: Dr. Ernest Hall and Dr. C. Y. Han
Abstract

This thesis addresses the problem of the robust and rapid detection and accurate mapping of obstacles in 3-space to a 2D map using live real-time stereo vision. The significance of this work is the novel method of the algorithm, which processes information extremely fast and solely in disparity space and creates a real-time obstacle map. The accuracy and robustness are further improved by floor noise cancellation which features a calibration algorithm that allows expeditious setting of the calibration parameters.

The core of this approach is:

- Removing floor-ground background and other unwanted “noise”.
- Separating the image into depth planes.
- Aggregating column information into smaller usable sets and summing those sets by depth to produce obstacle information at each depth.
- Using that information to produce a “live” occupancy grid of objects.

Based on this approach, an algorithm is presented with Matlab code that processes still disparity maps to produce obstacle size and location information in a simple format. In addition, an algorithm is also presented in C++ that creates a calibrated obstacle map in real-time using a commercially available stereo camera system.
1 Introduction and problem statement

Recent advances in computer stereo vision algorithms that produce quality disparity images and the availability of low cost high speed synchronized stereo camera systems have simplified many of tasks associated with robot navigation and obstacle avoidance using stereo vision. Leveraging these benefits, this thesis describes a method for detecting obstacles in the field of view currently being implemented on the UC Bearcat Cub Robot. The general problem of designing a machine for real time navigation and obstacle avoidance in an arbitrary environment is ongoing. This problem is often described as a problem in artificial intelligence, fuzzy logic, sensor fusion, intelligent control or collaborative systems. When dealing with numerous, noisy, conflicting, incomplete, and uncertain information from multiple streams, designing robotic computer systems and algorithms for autonomous navigation and obstacle avoidance is non-trivial.

The more specific problem of using stereo vision to detect obstacles in the field of vision of an Autonomous Ground Vehicle (AGV or robot) requires considerable processing efficiencies to handle live video at real-time rates. Although commercial camera systems can deliver “pretty good” disparity maps at fifteen frames per second or better, processing and performing useful algorithms while maintaining high information velocities requires resourceful algorithms and careful programming techniques.

Reliable obstacle detection and mapping to 2D or 3D space is a first step in a multitude of problems in computer vision such as AGV navigation, object recognition, and object tracking. This paper discusses algorithms and techniques to process the disparity maps to extract information on obstacle size and location and map them to a 2D grid for AGV navigation. An extension to 3D is straightforward, but not required in this application, although an approach is
mentioned in the “Future Work” section of this document. The algorithm developed here will be termed the X-H map algorithm because it performs a histogram on sets of columns and maps them across the X dimension. The histogram information is then transformed into depth to give the Z dimension for the 2D X-Z dimension obstacle map. We are left with a “floor plan” of obstacles. Height information of objects is lost in the 3D to 2D mapping. Figure 1.1 includes an actual obstacle map (to scale) produced by the program and describes the coordinate system and Field of View (FOV) for the Cub Robot. Figure 1.2 is a simplified 3D view of the Cub Robot with two co-ordinate systems defined. The coordinate system associated with the Bumblebee camera, and a translated system associated with the tilt sensor, details to be discussed later. Figure 1.3 demonstrates the 3D to 2D transformation. The image coordinates (Figure 1.4) are defined as X and Y, the horizontal and vertical dimensions respectively.

![Figure 1.1 Coordinate system and FOV of Cub](image-url)
**Figure 1.2 Cub camera and tilt sensor co-ordinate frames**

**Figure 1.3 Coordinate systems: 3D to 2D Mapping**
This algorithm will answer the simple but important questions such as “How large is the object in front of the AGV?”, “How far in front of the AGV is the object?”, and “At what angle is the object, is it directly in front of the AGV or off to one side”. This algorithm cancels information from the floor, which enhances the accuracy, and it also compensates for tilt (roll and pitch) which is required here so that the floor is not mistaken for objects and objects can be clearly identified in a tilting situation. Matlab sample code, disparity maps and images are available at: http://www.ececs.uc.edu/~rosseldw/RobotStereoData.zip.

The basic hardware and software requirements to produce these results were the Point Grey Research “Bumblebee” digital stereo camera system (including the camera software library), Microsoft Visual Studio 2003, the Intel IPP image processing library, the Intel OpenCV computer vision library, Matlab software, and a Pentium 4 computer.
2 Literature Review

2.1 Stereo Correspondence

The basis for real-time obstacle avoidance using stereo vision is robust and fast stereo correspondence algorithms. Although many areas of computer vision research are very active, one area that is extremely active is finding a robust solution to the stereo correspondence problem. This is the problem of: given a point in one image; find the corresponding point in the other image. If we know the correspondence of each point in each image, their relative displacement (known as disparity) between images can be used to create a disparity map. The amazing thing is how easy it is to determine the 3D structure of the scene once the disparity is known. The depth is inversely proportional to the disparity and a dense disparity map can be immediately inverted and scaled to create a point cloud in 3D space.

Finding a perfect disparity map is impossible in the general case. Many points that appear in one image do not appear in the other for various reasons such as occlusions, reflections, or poor texture. This is not to imply that nothing can be done, but certainly makes the problem more interesting. It will never be possible to create accurate information to fill in areas where there is none, but accurately determining and using what information is there is amazingly elusive. As the algorithms become more sophisticated and computational power increases, optimal solutions seem now more than ever within grasp.

Many researches have attacked the problem of stereo correspondence and the literature is overwhelming. There are so many papers on algorithms that several papers have been written just to summarize and compare them. A classic is by Dhond and Aggarwal [Dhond 1989] that compared feature based and area based algorithms and concluded that much was desired in the
way of improvement. History has proved them correct with a vengeance as the number and sophistication of algorithms has exploded since then. More recent attempts to categorize and compare have been Scharstein and Szeliski [Scharstein 2002], who did performance analysis in a very systematic and rigorous comparison. They maintain a stereo vision research website at: http://www.middlebury.edu/stereo.

This site contains reference images, various stereo algorithms in c code, along with an evaluation program to automatically create statistics on arbitrary algorithms. Designers are encouraged to test their stereo algorithms with the included tools for standard benchmarking.

Brown [Brown 2003] has published a comprehensive overview of the various stereo correspondence methods. This paper differs significantly from [Scharstein 2002] in that the emphasis is on describing the various methods in more depth (with references and examples) rather than on benchmarking. The section comparing real-time methods with a table to analyze several implementations is interesting and relevant.

Stereo matching approaches are often broadly categorized into “area based” / “feature based” [Dhond 1989] or “Local Methods” / “Global Methods” as a method to differentiate scales at which the algorithms focus. As is often acknowledged, there are too many methods to be covered or even accurately categorized. Below follows what I believe to be the most successful and relevant to this research. These are the Local Method Correlation Based approach, and the Global Method Dynamic Programming.

A popular example of a Local Area based technique is Block Matching. Block Matching is the only method family at this time fast enough for real-time vision work. Block matching is, as its name implies, operations on a small region or block. The idea is to search and compare regions in an image and track various maximum scores, or minimum errors using methods of correlation such as Normalized Cross-Correlation (NCC), Sum of Squared Differences (SSD), or
*Sum of Absolute Differences* (SAD). The stereo correlation method that the Point Grey Bumblebee uses is the SAD. New implementations of the Intel IPP [Intel 2003] software library implement both the normalized SSD, and the NCC. These correlation methods execute in an extremely fast mode (SIMD Single Instruction Multiple Data) and may be used in part by both the SRI [Konolige 2003] and Point Grey Research [PGR 2003] stereo systems as a basis to compute disparity maps in real-time. Although the general method is known (SAD), a lot is done by Point Grey that is proprietary to increase the speed and improve the performance.

A good example of the Block Matching correlation technique is Faugeras, Hotz, et al. from INRIA [Faugeras 1993]. This is a real-time implementation (on specialized hardware) that produces dense disparity maps and features a trinocular algorithm. The trinocular algorithm uses the first image as a reference image, and disparity maps are compared against the other two images to determine which produces the best score. It is also a hierarchical algorithm, using a window of fixed size and matched at several levels of resolution, which has proved to be a popular and effective method. For a more recent and robust approach of the block-based hierarchically multi-resolution image pyramid method, see [Falkenhagen 1997]. Below is a summary [Brown 2003] of various block matching correlation methods.
### Block Correlation Techniques

**Figure 2-1** Block Correlation Techniques

[Brown 2003]

<table>
<thead>
<tr>
<th>MATCH METRIC</th>
<th>DEFINITION</th>
</tr>
</thead>
<tbody>
<tr>
<td>Normalized Cross-Correlation (NCC)</td>
<td>$\sum_{x,y} \left( I_1(u,v) - \bar{I}_1 \right) \left( I_2(u + d, v) - \bar{I}_2 \right)$</td>
</tr>
<tr>
<td></td>
<td>$\sqrt{\sum_{x,y} \left( I_1(u,v) - \bar{I}_1 \right)^2 \cdot \left( I_2(u + d, v) - \bar{I}_2 \right)^2}$</td>
</tr>
<tr>
<td>Sum of Squared Differences (SSD)</td>
<td>$\sum_{x,y} \left( I_1(u,v) - I_2(u + d, v) \right)^2$</td>
</tr>
<tr>
<td>Normalized SSD</td>
<td>$\sum_{x,y} \left( \frac{I_1(u,v) - \bar{I}<em>1}{\sqrt{\sum</em>{x,y} (I_1(u,v) - \bar{I}_1)^2}} - \frac{I_2(u + d, v) - \bar{I}<em>2}{\sqrt{\sum</em>{x,y} (I_2(u + d, v) - \bar{I}_2)^2}} \right)^2$</td>
</tr>
<tr>
<td>Sum of Absolute Differences (SAD)</td>
<td>$\sum_{x,y} \left</td>
</tr>
<tr>
<td>Rank</td>
<td>$\sum_{x,y} \left( I_1(u,v) - I_2(u + d, v) \right)$</td>
</tr>
<tr>
<td></td>
<td>$I_1(u,v) = \sum_{m,n} I_4(m,n) &lt; I_4(u,v)$</td>
</tr>
<tr>
<td>Census</td>
<td>$\sum_{x,y} \text{HAMMING}(I_1(u,v), I_2(u + d, v))$</td>
</tr>
<tr>
<td></td>
<td>$I_1(u,v) = \text{BITSTRING}_{m,n}(I_4(m,n) &lt; I_4(u,v))$</td>
</tr>
</tbody>
</table>

2.2 **Obstacle Detection using Stereo Vision**

The general problem of detecting obstacles using images and computers is quite complex and the solutions are embryonic and under tremendous development. “Obstacle illusions” are common in human vision, so it is best to approach the “state of the art” in computer vision with an appreciation of the shortcomings. Obstacle detection with stereo vision is a passive technology. It does not require an active source such as radar, laser, or sonar as do other technologies. The best solutions in the future may combine several methodologies with stereo vision such as color, texture, shape, and optical flow to approach truly robust obstacle detection using computer vision. It is becoming quite common for stereo vision to be combined with active source sensors to incorporate the strengths of two or more systems.

Many methods exist to detect objects using images, such as depth-from-focus [Nourbakhsh 1996], contour detection with a downward pointing camera [Fasola 2004], and optical flow [Coombs 1995]. Virtually all of these methods suffer from serious constraints, such
as the being extremely computationally expensive, unreliable, assume the ground-floor is monochromatic, or require advance knowledge about the surroundings or obstacle. The recent development of affordable stereo vision camera systems that can provide good quality disparity maps in real-time has opened up a multitude of research opportunities to provide more robust solutions.

A straight-forward but computationally expensive approach using stereo vision is to convert each disparity value in the map image to a 3D point in space and then map and process that information to detect obstacles [Moravec 1996] and [Singh 1999]. A 3D grid map is easy to look at and determine 3D structure, but quite a difficult problem to process the information to create meaningful results with techniques such as clustering and connect points in 3D to produce lines and shapes.

An interesting approach [Williamson 1998] to detect obstacles combines methodologies in image warping (referred to as “Affine Reconstruction in [Faugeras 2001] and [Hartley 2003]) to the stereo correspondence problem to more accurately determine “horizontal” and “vertical” surfaces for obstacle detection. This method blurs the line between the stereo correspondence problem and the obstacle detection problem. First, the stereo correspondence in calculated in the usual way. Then the image is warped to the ground plane and calculated again. When the image is warped to the ground plane, horizontal surfaces are more accurately mapped using stereo correspondence. By comparing the maps of warped and un-warped images, surfaces can be identified as horizontal or vertical. This method uses standard connected components methods of image processing to find and identify clusters of pixels and the size, centroid, etc are calculated for these clusters to be used to identify objects.

An extremely quick and simple method [Murray 2000] simple takes the maximum disparity of each column in the disparity image and maps it to a 2D grid. This method will
produce a mapping of the closest object in 2D, but will not provide a more detailed view of the space. For example, a tall cabinet a few feet behind a box of equal width would be totally excluded. This may be adequate in many situations, but for path planning through a course of various obstacles, it may not be ideal.

A very impressive solution by [Greenspan 2004] uses the Hough Transform methods to track an object in 3D space, even with translations and rotations. See Figure 2.2 below. The authors coined the term “Bounded Hough Transform” because they bound the rate of change in position and pose between frames. A serious drawback to this method for practical obstacle avoidance is that the initial surface model geometry and position must be input.

![Figure 2.2 Object tracking](Greenspan 2004)

Identifying and correct filtering of data from the ground or floor is paramount in obstacle detection. Obviously, lines or textures on the floor should not be mistaken for obstacles. The importance and use of the Ground Plane in obstacle detection was suggested early by [Ferrari 1990], who used orthogonal regression to determine properties of the ground plane using points in 3D. [Burschka 2002] modeled large 3D scene structures such as the floor and walls directly from the disparity image. By assuming and classifying portions of the image as “floor”, “foreground”, “background”, “left wall”, etc and choosing points on these assumed planes, planes were fit to the structures by using the RANSAC [Fischler 1981] method and the geometry of images as found in [Faugeras 2001].
Once an object is identified, the most basic requirement in obstacle detection is mapping the object to a 2D or 3D space. [Elfes 1989] proposed an “Occupancy grid” (sometimes called evidence grids) which is way of representing spatial information probabilistic framework. Each cell holds a probability number that represents the certainty that the cell is occupied. This method can be used when combining multiple sensors such as laser, IR, and vision to create a more robust mapping of obstacles that implicitly handles uncertainty.

The term “Occupancy grid” has sometimes been used to designate any mapping of objects from 3D space onto a 2D grid. That is the approached used in this project. Future work may include fusing sensor data from the Laser Scanner system and the vision system in a probabilistic schema in a manner similar to [Martin 1996] for the UC Cub AGV.
3  Methods

3.1  Hardware and Software for Stereo Video Acquisition

3.1.1  The Bumblebee stereo camera system

The sensors included on the Cub for stereo video acquisition required for obstacle avoidance is the Bumblebee stereo vision system by Point Grey Research (PGR). Also included for stereo obstacle avoidance is a SICK LADAR scanner. For line following, two color video cameras are used in conjunction with an ISCAN video processing system that feeds line position information to the Cub control computer (Dell Latitude D800 laptop) via a standard serial port. Refer to Figures 3.1 and 3.2.

The Bumblebee is a packaged system that includes two pre-calibrated digital progressive scan Sony ICX084 CCD cameras with a baseline (the distance between cameras) of 12cm, and a C/C++ Software Development Kit [PGR 2003], and a 400 Mbps IEEE-1394 Firewire interface for high speed communication of the digital video signal. Several camera lens sizes and HFOV (Horizontal Field Of View) are available, but our system has a 6mm lens and 100º HFOV and the Black and White CCD. Gain and shutter control can be set to automatic or adjustable manually. The calibration information is preload into the camera allowing the computer
software to retrieve it for XYZ coordinate calculations and image correction. The Bumblebee is advertised to acquire images at 30 frames per second. That is somewhat misleading because it does not include the time it takes their software to calculate the disparity map on a local computer. Our implementation operates at about 12 frames per second for the full processing of one frame (see benchmark section below). The images are not compressed or encoded as this would not only have a detrimental effect on processing speed (requiring encoding at the Bumblebee and decoding at the computer), but a lossy compression would affect the quality of the disparity maps. As might be inferred from the “triclopsGetImage()” function below, the image frame can be grabbed with this function and processed directly with the Intel IPP or OpenCV library.

The Point Grey SDK includes several functions for communicating with the camera, preprocessing of the images, creating disparity maps and 3D point cloud, and saving/retrieving images or image sequences for offline processing. The function sequence from the PGR SDK to communicate with the Bumblebee and to create disparity maps for this obstacle avoidance algorithm can be summed up in the function `GrabDisparityImage(imgDispMapSrc)` which is represented by:

```
Begin Loop

digiclopsGrabImage(digiclops);

digiclopsExtractTriclopsInput( digiclops, STEREO_IMAGE, &inputData );

triclopsPreprocess( triclops, &inputData );

triclopsStereo( triclops );

triclopsGetImage(triclops, TriImg_DISPARITY, TriCam_REFERENCE, &depthImg );

“process obstacle avoidance algorithm using disparity map(depthImg)” (detailed later)

End Loop
```
To understand this sequence, the PGR SDK uses the concept of a “context” to simplify storing information about the stereo images, calibration and timing in one structure. The digiclops context contains the raw packed information from the computer.

\texttt{digiclopsGrabImage(digiclops)} grabs the image from the camera and store all information in the \texttt{digiclops} context.

\texttt{digiclopsExtractTriclopsInput( digiclops, STEREO\_IMAGE, &inputData )} extracts the data from the context and put it into the \texttt{inputData} data structure.

\texttt{triclopsPreprocess( triclops, &inputData )} unpacks and rectifies the image and may do smoothing, and edge detection if requested. Note the \texttt{inputData} is put into another “context”, the \texttt{triclops} context. This context is used for unpacked data and is used for higher level function. The rectification stage is required to remove lens distortion. The Bumblebee uses a 6mm fixed focal length lenses, which creates images with a severe “fisheye” effect. The rectification (or image warping) is done by this function by using the calibration parameters already established and supplied with the camera (and included with the context). Figure 3.3 below is un-rectified and Figure 3.4 below is rectified. These figures were taken with the Bumblebee inside the robotics lab at the University of Cincinnati. A garbage can is in the foreground.
Figure 3.3 Raw un-rectified stereo image

Figure 3.4 Rectified right channel of the stereo image
Note how the floor lines no longer appear curved in the rectified image. You may also notice that image 4.3 seems to be in color, whereas image 4.4 is not. Actually, the top image contains two grey scale images (the stereo pair) taken for all practical purpose at “exactly” the same instant (synchronization is 125µs maximum deviation). The images are stored in RGB format using only the Red and Green image planes. PGR also manufactures a Digiclops system, which has three cameras instead of two for more robust stereo processing. The Digiclops uses all three channels of the RGB. The importance that the images are taken at the same moment cannot be overlooked. Since the creation of a disparity map requires mapping pixels from one image to the other and if the AGV or an object is moving, many or all pixels will not have a corresponding in the other image pair. Creating a quality disparity map is quite difficult as previously discussed and poor synchronization may well render the map useless at best.

triclopsStereo( triclops )

This function performs stereo processing on the image. The argument triclops is the “context” which is the structure that holds all input images and output disparity map information for this function. triclops contains the input image, the output image, camera calibration parameters, and information required to perform the processing (that are set elsewhere in the program), such as the minimum and maximum disparity values to search while performing the stereo correspondence.

triclopsGetImage(triclops, TriImg_DISPARITY, TriCam_REFERENCE, &depthImg );

This function retrieves the disparity map image from the triclops context and places it into depthImg for processing by the main algorithm.
Further documentation on these and other PGR functions are often (but not always) found in the PGR SDK manual. Other sources of information for the PGR are found by examining the header files “triclops.h and digiclops.h, and in the examples that come with the SDK.

3.2 Software

3.2.1 Commercial software libraries

To put the current Cub AGV processing needs into perspective, a 640x480x8bit image acquired at 15 frames per second produces 4.6 Mbytes of data a second. The C++ real-time code written for this thesis uses the software library Intel Performance Primitives (IPP) [Intel 2003] which is designed to make full use of the Intel Pentium architecture at the hardware level. The Pentium features the Single Instruction Multiple Data (SIMD) to enable this large data throughput. The IPP library was instrumental in simplify and speeding computer vision and matrix operation tasks in this project. Specific examples will be given throughout this document.

The Intel OpenCV library [Intel 2005] is a higher level computer vision library and is designed to work with or without the IPP layer, which will allow it to work much faster.

3.3 Main Algorithm Background: Disparity Map Processing

3.3.1 The disparity (depth) map image

A disparity map or “depth map” image is an efficient method for storing the depth of each pixel in an image. Each pixel in the map corresponds to the same pixel in an image, but the grey level corresponds to the depth at that point rather than the gray-shade or color.
An experiment to demonstrate disparity is demonstrated in Figure 3.5 below. In this figure a stereo pair with a baseline of about two feet has been overlaid using Adobe Photoshop. Note that objects nearer have greater separation (this is the disparity), and objects very far away line up very close (they will have less disparity). There is large separation between corresponding objects in foreground, but points converge as distance increases. In fact, this is how stereo vision algorithms work. The computer attempts to match every pixel in an image with every pixel in the other using a correspondence algorithm. This will always be imperfect due to occlusions, poor texture (think of snow blindness), etc but works amazingly well quite often, as does the human visual system.

Figure 3.5 Superimposed stereo image pair
Disparity Map Construction can be summarized as follows:

- Find every corresponding point between the images. See Figure 3.6 below
- Assign a value 0 to 255 at each point based on the “disparity” or separation.
- White = 255 = close, Black = 0 = far

A sparse matching of points is demonstrated in the stereo pair shown in Figure 3.6 below. Although several points have been matched accurately, many have not. These results were produce using a normalized cross correlation method written in Matlab and not optimized in any way. Examination of these images can give some idea of the difficulties associated with point matching, such as items (pixels) that occur in one image and not the other.

Figure 3.6 Matching points in an image pair
Matching as many points as possible, the Bumblebee can produced dense (as opposed to sparse) disparity maps at 15 fps. Figure 3.7 is a single image from a stereo pair of a garbage can in the middle of the floor inside the robotics lab. Figure 3.8 shows a histogram equalized disparity map produced from Figure 3.7 using the PGR function triclopsSetDisparityMapping(). This function creates a disparity map that is “balanced” visually for viewing and not to be used for calculations because of the distortion of the depth values. With this 256 bit gray level disparity map image, objects that are lighter are close, and darker object are farther away. Pixels that cannot be matched are given the value of 255 for identification or exclusion in further calculations. Creating quality depth maps at 15 fps is not trivial, and the Bumblebee enables research focus on other tasks. In Figure 3.8, note the lines on the floor going from the foreground to the background, and how they darken as their depth increases. Data from the floor can be problematic when processing the maps and will be discussed below.

3.3.2 Point Grey disparity maps

Close reading of the Point Grey Research Stereo Vision Manual [PGR 2003] included with the Bumblebee Stereo Camera system will reveal the finer points of their disparity maps.
The PGR system will produce disparity maps in either an 8 bit or 16 bit format in the portable grey map (.pgm) file format. The 8 bit image is an unsigned integer 8 bit format with 256 levels grayscale and can be viewed with MatLab or the freely available IrfanView viewer program (www.irfanview.com). Matlab 6.5 or above is required to read .pgm files.

Figure 3.9 Raw 8 bit disparity map

Figure 3.10 Histogram of raw 8 bit map.
The 16 bit depth images are in fixed point format, the high byte being the integer portion, and the low byte being the decimal portion. In reference to their 8 bit format, the actual image produced for extracting depth is quite different and shown in Figure 3.9, and its histogram is show in Figure 3.10. Note that all of the information is stored in the first 26 levels as may be inferred from the histogram in Figure 3.10. Because virtually all of the pixels values occur in this image from 0 to 26, it is very dark. In general, there may be more valid pixel values (gray levels) under different circumstances such as nearby objects, and the maximum disparity value can be set in the PGR software. This value should never be of a larger value than is required for the object depth that you are trying to detect. For example, is a disparity value of 26 corresponds to a depth of three feet, and three feet and farther is the depths of interest, this value should be set to 26. Higher values will only decrease the efficiency of the PGR system because it has to search the images over a larger distance when attempting to match corresponding points. Level 240 to 255 is reserved for invalid bits, and 0 is used for points at infinity. The white that you see in Figure 3.9 is mapped to invalid pixels, pixels that could not be mapped for various reasons. These values are ignored in our analysis. Zero values also need not be processed.

The PGR system will allow setting the system to sub-pixel interpolation mode. This mode will interpolate to a 16 bit fixed point image format to produce a more accurate result on the disparity map when matching pixels are found. This may be useful for example when trying to produce accurate 3D object reconstructions and rendering of a nearby object. For example, see Figures 3.11 and 3.12 below, which were taken off the PGR website in the Product Showcase section. This type of problem is best approached with the highest resolution possible to produce realistic results. Note that the object has been rotated and then rendered in 3D space. Missing
sections in the resultant image are caused by the failure of the PGR system to create “perfect” disparity maps, because of regions of poor texture for example. This example not only demonstrates the power to the PGR system, but also its shortcomings and how much is yet to be done to produce even better results.

When producing a disparity map in PGR 16 bit format, an attempt at direct display creates the archetypical and bizarre results as shown in Figure 3.14. Figure 3.14 was produced from the image in Figure 3.13. These 16 bit maps can be converted back to 8 bit by stripping off the fractional portion (and hence reduce the accuracy of the disparity map). This is demonstrated with Matlab code below and the results seen in Figure 3.15 below, which has been reduced to 8 bit and histogram equalized. Note that the garbage can is clearly visible in light gray, and the box, which is in the background and is a darker shade of gray.

```matlab
A = imread ('disparity16.pgm');
data = bitand(A,255) ;       % Strip off Fractional portion
A = uint8(data);             % Get back to 8 bit format
figure, imshow(A)            % Show the image before histogram
J = histeq(A);               % Perform histogram equalization
figure, imshow(J)            % Show the image after histogram equalization
```
3.4 Main Algorithm: Extracting Object Position from Disparity Maps

3.4.1 Preliminaries: Disparity processing algorithm

The information contained in the disparity maps may be interesting to look at, but it is useless for AGV obstacle avoidance until it is processed to obtain object position and depth information. Without loss of generality, consider a disparity map as 480 pixels high, 640 pixels wide and 256 (0-255) levels in gray-level depth. Referring to Figure 3.16 below, visualize a depth map as 256 separate images, each depth image will have information about the image at that depth. For example, if the garbage can is 10 feet from the camera, it might have several pixels at a disparity level 15. If it were at 14 feet it might have several pixels at a disparity value
of 11, (or found on plane 11). Recall that larger disparity values correspond to closer objects. Depending on depth resolution and object size and orientation, objects will generally appear at two or more image depth planes.

Figure 3.16 Image plane levels

Figure 3.17 Level 3 and Level 1.

Figure 3.18 The profile function
Assume an image of a cone as in Figure 3.17. Since level 3 (level, plane and disparity value are used interchangeably here) is nearer, the front surface of the cone may produce pixels as in Figure 3.17, since the plane is just cutting the front edge of the cone. Level 1 is cutting the center of the cone and may produce pixels similar to Figure 3.18. By summing and plotting the columns, curves are obtained similar to those shown below the figures in Figure 3.17. By finding the peak, the area, and the center of area under these curves, the size and position of the object can be determined with good precision. Finding the peak is trivial. Referencing Figure 3.19, the area and center of area are found as follows:

\[ A = \int y \, dx \]

First find the area. \[ M_{yy} = \int y_1 x \, dx \]
Second, find the first moment of the area \[ M_{xx} = \int x_1 y \, dy \]

Finally, the coordinates for the center of area are found using the equations:

\[ \bar{x} = \frac{M_{yy}}{A} = \frac{1}{A} \int y_1 x_1 \, dx \]

\[ \bar{y} = \frac{M_{xx}}{A} = \frac{1}{A} \int x_1 y_1 \, dx \]

Rather than processing disparity maps, a 3-D point cloud could be created from the pixels by using the formula \( Z = f \cdot B / d \), where \( Z \) is Depth, \( f \) is focal length, \( B \) is the baseline of the stereo camera, and \( d \) is the disparity value from the disparity map (depth map). However, the point
cloud data would have to be processed resulting in several extra steps. To calculate actual X and Y position using the u and v coordinate from the disparity map, the formulas are \( X = \frac{uZ}{f} \), and \( Y = \frac{vZ}{f} \). The formula \( Z = \frac{fB}{d} \) is used in this algorithm to determine the depth of each plane.

3.4.2 Specifics: Disparity processing algorithm in Matlab

First the algorithm will be presented in Matlab, which is used instead of pseudo-code because it is nearly as simple and can actually be run to demonstrate the algorithm. Some details of the implementation differ when implemented in C++ and will be discussed in the next section.

In practice, to find objects at each depth by summing rows and columns by the process of breaking an image out into 255 image planes would prove inefficient computationally. A trivial approach would require \( 480 \times 640 \times 255 = 78,336,000 \) summations for each frame, and the current implementation presented in the following section runs at 12 frames a second with stereo (2) images. To achieve that throughput would thus require \( 78,336,000 \times 12 \times 2 = 1,880,064,000 \) summations each second. The task-at-hand is to take these concepts and produce an algorithm that is fast and efficient. Referring to the code below, taking one column of data from any 480x640 image “A”, will produce a 480x1 array. Performing a histogram count on this data in the full range of pixel values (0-255) results in a 256x1 array. Each of these arrays is put into a column to produce an array “X”, 256x640.

```matlab
%% ** Column analysis, No cancellation
A = imread ('depth.pgm'); % Load the image
[m,n] = size(A); % Get the size of the image
X=zeros(256,n); % Create an array to hold results
for i = 1:n % For every column.
    dataCol=A(:,i); % Get the column of data:
    count=histc(dataCol, 0:255); % get the hist. count of each number
    X(:,i) = count; % and put it in a column of X
end % X is the output, a 256x640 matrix
figure, plot(X(23,:))
```
The X array contains information about the number of occurrences of each disparity value across the X axis (the width of the image). Plotting each pixel across this axis, would produce 256 plots. The data could be manipulated to reduce the resolution in order to decimate the calculation burden, but as it turns out this is not necessary. Note that in Figure 3.10 above, the information is contained in only the first 26 disparity values which would require only 26 plots.

Direct analysis of these 26 data sets is great in theory, but in practice the noise produced by the data from the floor is so great, real objects cannot generally be distinguished. Refer to Figure 3.19 below. Note that the data in these figures were scaled for better viewing by multiplying it by 6, so that X and Y data could be included on the same plot. These plots were produced by processing the disparity map from Figure 3.9 above. Note that peaks in Figure 3.19 along the Y axis is the height of objects found, with the Y axis direction reversed (Y axis positive is down). Therefore, the spike near the top is the signal from the floor, and the signal between the 250 and 300, is our object data. Looking along the X axis (X axis positive is right), it is nearly impossible to distinguish where the object might be. There are several peaks that correspond to the lines in the floor (Figure 13) that are giving extraneous peaks.

A strategy to remove the floor data is to use the information from the Y direction and determine the height of the objects producing the signals. By removing the signal from the floor, amazingly clear signals produced by the actual objects-of-interest can be seen. To cancel the floor data background noise at (for example) level 22:

```matlab
% Background Cancellation at level 22
k1=23;  % add 1 for level 22. Matlab starts at 1, pix starts at 0!
for i = 420:450  % Remove the floor data
    pixKill = find(A(i,:) == k1);  % Find background pixels in row to cancel
    A(i,pixKill) = 0;               % Cancel the background
end
```
Figure 3.20 is an example of this process. It can be seen that the floor data (yellow dotted line) has been drastically attenuated. The signal centered at 325 on the X axis (red solid line) that appears poorly attenuated is actually an initial signal from an object. At disparity value 19 (Figure 3.20), it is extremely clear that there is an object at this location. Note once again the attenuated signal is shown with dotted yellow lines.

![Figure 3.20 Floor signal](image1)

![Figure 3.21 Floor Signal Cancellation](image2)

![Figure 3.22 Obstacle Data](image3)

![Figure 3.23 Tilt Compensation](image4)

A close look at the code snippet above demonstrating cancellation reveals the numbers 420:450. This is a method to cancel the data at this pixel value in those rows. This is a simplistic and specific example for demonstrating the concept, and this actually works well on a calibrated system on level ground, with a fixed camera focal length. But a more complicated
algorithm must be used in practice to allow for roll, pitch, and changes in camera focal length.

Focal length on the Bumblebee will change with a change in resolution, for example from 480x640 to 280x320. Methods to deal with severely “bumpy” ground and other complications will be discussed below in the “Future Work” section. The cancellation must be done by image column or by rotation of the image in practice. This is computationally more expensive, but it is a price well worth paying. Imagine trying to walk with your inner ear unable to determine your head orientation. By using a tilt sensor and cancellation by columns, floor signal recognition and cancellation is possible. Referring to Figure 3.23, a 12º tilt would require a cancellation in the blacked out areas, which is done relatively efficiently while the columns are being processed. A Matlab code snippet is shown below that demonstrates this method with the features mentioned above. This is the code that was used to calculate and draw Figures 4.19 through 4.21 based on real disparity image data. This code (ready to run) and sample images are found at http://www.ececs.uc.edu/~rosseldw/RobotStereoData.zip

```matlab
%% ** Column analysis with cancellation
FloorIncrement=10; % This may change with forward/back tilt (pitch)
Spread=30; % Spread=Floor noise width
TiltFactor=0; % Changes with tilt, varies with roll, number of % pix to move base up/down per 5 pix over.
X=zeros(256,n); % Create the Matrix to hold data
for i = 1:n % n is the width of the Pic
    dataCol=A(:,i); % Get each column of data: note we are using % data,m x n row/column matrix ie (640x480, 16 % levels
    Base=220; % Good for DispGarbageCanRealDis.pgm This will % change with forward/back tilt, and focal % length, must be set inside this loop with % real-time data
    for killVal = 2:25 % Done inside each Column to compensate tilt.
        killZone = (dataCol(Base:Base+Spread))'; % Transp. to a row vec
        killPix = find(killZone == killVal); % Find value to cancel
        dataCol(Base + killPix)=0; % Line up killZone to % dataCol and % cancel floor pixels
        Base = Base + FloorIncrement;
        if(Base >= m - Spread) % Stay within bounds
            break
        end
    end
```
if((TiltFactor ~= 0) && (mod(i,5) == 0)) % We have moved 5
  % pixels, compensate
  % base with tilt factor
  Base = Base + TiltFactor; % Adjust Base for tilt
end end

count=histc(dataCol, 0:255); % get the count of each number
  % (0-255) in each column of the pic
  X(:,i) = count; % and put it in a row of X, each row
  % represents a depth, and the value
  % across the row are the number of
  % times a pixel was found at that
  % depth.

3.4.3 Specifics: Disparity processing algorithm with C# and C++

The core implementation of this algorithm is written in C++ for speed and compatibility with the Intel libraries, but the GUI (Graphical User Interface) was written in C#. C# was chosen because it is the language standard of the UC Cub software system. The program features a semi-automatic floor cancellation calibration and it can run “live” (plugged into the bumblebee camera) or “still” (using a previously saved disparity image, useful for development offline). The C++ core is wrapped as a DLL (Dynamic Link Library) and is called as a separate thread from the GUI. The Calibration Box appears light because it is disabled during run mode.
The implementation in C++ relies on the same basic principle as the Matlab implementation, which is to remove the floor background and sum the columns at each depth to detect obstacles, but it is somewhat different, and much faster.

The main function logic will be presented first and then each function will be discussed in some detail. The main loop is:

```
Begin Loop
    GrabDisparityImage (imgDispMapSrc)
   ippiRotateCenter_8u_C1R(imgDispMapSrc , imgDispMapRot, rotAngle, …)
    RemoveFloorBkGnd(imgDispMapRot, imgDispMap, FloorIncrement, Spread, Base)
    RemoveTopThird(imgDispMap)
    CreateAggregatedX-HistoMap(imgDispMap, XhMap)
    CreateOccupancyMap(triclops, XhMap)
    Roll compensation input from sensor
    Pitch compensation input from sensor

```
This loop continually grabs stereo image pairs from the bumblebee video stream and produces a disparity image, processes those images and produces a “live” calibrated occupancy map in real time. Objects that are moving can be seen to move across the map, and as the robot moves forward, objects flow from top to bottom of map as you would expect.

GrabDisparityImage(imgDispMapSrc) Grabs a stereo pair from the video stream and output a disparity image imgDispMapSrc. This function was discussed in detail in section 4.4.3 above.

ippiRotateCenter_8u_C1R(imgDispMapSrc , imgDispMapRot, rotAngle, …) This function performs roll compensation based on input from the tilt sensor. It simply rotates the image if required to offset any rotation about the roll axis of the AGV. Roll compensation is required to keep the floor-ground information from the cameras in the same relative position as determined at calibration. Referring to Figure 3.25 below (reproduced from above), roll is defined as $\alpha$, pitch (forward/back tilt) is defined as $\gamma$. Although the process of removing floor background is covered in detail just below, the pitch compensation input is also data received from the tilt sensor and deserves mention here. A rotation about $\gamma$ (pitch) will cause the image to translate in the Y dimension (appear to move up and down). Hence the position of the floor will translate in Y and must be cancelled. The variable “base” is the position in the image where floor cancellation begins and must be calibrated to offset pitch using information from the tilt sensor.
Figure 3.25 Cub camera and tilt sensor co-ordinate frames

RemoveFloorBkGnd(imgDispMapSrc, imgDispMap, FloorIncrement, Spread, Base) Removes the floor information from the disparity map. This function was made extremely efficient by the Intel IPP function ippiLUT_8u_C1R(…), which will replace a single or set of values in an image in a single function call. For floor cancellation, this function merely iterates through the 20 or so relevant depths and replaces the floor data with zeros so that the floor adds nothing to the column sum. Compare this 20 iterations with a direct search through a 40x640 section of image (they typical size need for floor cancellation) times 20 would require 512,000 iterations. The floor Spread typically encompasses forty adjacent rows on each depth plane and moves up by

FloorIncrement amount, typically nine rows of pixels with each iteration in increasing depth in the disparity map. These numbers are set during the calibration phase (described below).
numbers forty and nine are dependent on the properties of the camera, such as focal length which changes only with change in resolution (which is never in our application).

The image is not actually decomposed into “depth planes” to remove floor background. Figure 3.26 below is actual data from a disparity map image. Each hexadecimal number represents the disparity value at that pixel location in the disparity map. Assuming that the floor data appeared in rows five and six at depth “07”, replacing “07” with “00” effectively removes the floor data at that depth. The next depth plane would be “08” and would be removed in another set of rows. Removing the floor data simply requires replacing the appropriate numbers in the appropriate rows with zeros to cancel their contribution when the numbers are binned by the histogram operation. *FloorIncrement* is also the amount the floor peak advances in units of pixels per increase in depth during the calibration stage.

<table>
<thead>
<tr>
<th>Original image as seen in memory (hexadecimal)</th>
</tr>
</thead>
<tbody>
<tr>
<td>06 05 08 08 08 07 08 07 08 08 08 08 08 08 08 08</td>
</tr>
<tr>
<td>06 05 08 08 08 08 06 06 08 08 08 08 08 08 08 08</td>
</tr>
<tr>
<td>08 0a 08 08 08 08 08 08 08 08 08 08 08 08 08 08</td>
</tr>
<tr>
<td>07 0a 09 0a 09 09 09 07 07 07 08 08 08 08 08 08</td>
</tr>
<tr>
<td>fc 09 09 fc 09 09 07 07 07 06 06 07 07 07 08 08</td>
</tr>
<tr>
<td>09 09 09 09 fc 09 07 07 07 06 06 07 07 07 08 08</td>
</tr>
<tr>
<td>00 00 00 00 00 00 00 00 00 00 00 00 05 07 07 08 08</td>
</tr>
<tr>
<td>00 00 00 00 00 00 00 00 00 00 00 00 00 05 06 07 07 07</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ROI Rows 5 and 6 (level “7” removed)</th>
</tr>
</thead>
<tbody>
<tr>
<td>06 05 08 08 08 07 08 07 08 08 08 08 08 08 08 08</td>
</tr>
<tr>
<td>06 05 08 08 08 08 06 06 08 08 08 08 08 08 08 08</td>
</tr>
<tr>
<td>08 0a 08 08 08 08 08 08 08 08 08 08 08 08 08 08</td>
</tr>
<tr>
<td>07 0a 09 0a 09 09 09 07 07 07 08 08 08 08 08 08</td>
</tr>
<tr>
<td>fc 09 09 fc 09 09 00 00 00 06 06 00 00 00 08 08</td>
</tr>
<tr>
<td>09 09 09 09 fc 09 00 00 00 06 06 00 00 00 00 08 00</td>
</tr>
<tr>
<td>00 00 00 00 00 00 00 00 00 00 00 00 00 05 07 07 08 08</td>
</tr>
<tr>
<td>00 00 00 00 00 00 00 00 00 00 00 00 00 00 05 06 07 07 07</td>
</tr>
</tbody>
</table>

**Figure 3.26 Removing Floor Background**

Figure 3.27 below is one depth plane (depth level 20) from a disparity map. Figure 3.28 is the same depth plane with the floor information removed. *Base* is the position (in units of pixels)
in the disparity map to begin canceling the floor data. It is easily identifiable as the first peak in
the histogram set of histograms created while iterating through the depth planes during the
calibration phase. If the AGV pitches forward or aft, the tilt sensor information is sent to Base to
maintain the relative disparity image position determined at calibration. The argument
\texttt{imgDispMapSrc} is the input image, and \texttt{imgDispMap} is the output image.

![Figure 3.27 Level 20, data intact](image)
RemoveTopThird(imgDispMap) Removes the top third of the image information by replacing the data with zeros. This function is made extremely efficient by the IPP function ippiSet_8u_C1R(…), which will set a region of an image to a specified value in one operation. Removing the top third can make the algorithm more robust in cluttered indoor environments when objects above a certain height can be neglected, such as on the current application of an AGV robot.

CreateAggregatedX-HistoMap(imgDispMap, XhMap) uses the IPP function ippiHistogramEven_8u_C1R(...) to sum columns at each depth in one operation. This is the heart of the algorithm and is extremely efficient, but not very intuitive and requires a bit of explanation.

First and foremost is the speed requirement issue. For the current occupancy grid of 20x20, this function merely calls ippiHistogramEven_8u_C1R twenty times to aggregate 640 columns.
down to 20 and sum all twenty depths to produce the matrix required for the occupancy grid. This function contains only nine lines of code, five of which are definitions, and 20 iterations. Compare this for example, with looping through all 256 depths and 640 columns or 256x640 = 163,840 iterations. Since this loop needs to run 15 times a second, it can ill afford to be inefficient.

The input to this function is the disparity map image, and the output is an X-H map matrix defined below. Consider a 640x480 image divided into 20 vertical columns by aggregating the information from sets of 32 columns (640/32 = 20) to produce something as represented by the topmost graphic in Figure 3.29 below. This 20x480 section of image contains values in the range of 0-255, each value representing a depth. Since we are only interested in values 4 thru 24 in our application (which scales to depths of about 1 meter to 5.3 meters), we will perform a histogram on this 32x480 section in the reduced range of 0-32 (to encompass 4-24) instead of the full range of 0-255. The histogram just bins the numbers, giving sums for the number of pixels that occur at that depth. If the histogram from each column is put into a column of an X-H map a result similar to the matrix of numbers in Figure 3.29 below will be obtained. Each column of the original disparity image map image is now transformed into an X-H map matrix of 20 by 32 numbers. Figure 3.29 below only contains 16 columns rather than 20 merely so the graphic fit on one page.
Figure 3.29 Creating the X-H Map

32x480 pixel ROI that is “histogramed”

640 columns aggregated to 20

Z

Image matrix coordinates

X

Z = f*B/d
X = uZ/f
Dia = 6*c
The X-Z coordinates of a number (for example, 6 at depth 8 as circled in Figure 3.29) can be calculated by the equations $Z = f \times B/d$, and $X = uZ/f$ and plotted as in the grey obstacle map in figure 3.28. The diameter of each obstacle is calculated by $D = q \times c$, where $q$ is the value at that point in the X-H Map and $c$ is a constant to give a good appearance to the obstacle map. Since $q$ (6 in our example) is the number of pixels found at that depth and in that 32x480 area (Region of Interest or ROI) in the disparity map, it is a pretty good indication of the area or size of the obstacle. Since height information is collapsed in this process, a high and low obstacle at the same depth will combine to produce the larger combined area.

CreateOccupancyMap(triclops, XhMap)

This function has the bumblebee triclops context and XhMap as inputs. The context is required because this function call the PGR library function triclopsRCD8ToXYZ(...), to calculate the calibrated X (horizontal dimension) and Z (distance dimension) position of an objects. We are not concerned with the height of an object at this time, so Y is not used.

Recall the formula $Z = f \times B/d$, where $Z$ is Depth, $f$ is focal length, $B$ is the baseline of the stereo camera, and $d$ is the disparity value from the disparity map (depth map) and the formulas $X = uZ/f$, and $Y = vZ/f$ to calculate X and Y position using the u and v coordinate from the disparity map. Since the triclops context contains the camera calibration information such as baseline and focal length, this function can easily and accurately calculate the X and Z dimensions of each point in the X-H Map to create the Occupancy map.

The logic for this function in pseudo code is:

For each column of the X-H map

For each row of the X-H map
triclopsRCD8ToXYZ(…) % Get the X-Z co-ord for the row/col

Scale a circle to size of object

Draw the circle using calibrated X Z found above

End for row

End for column

This function produces a scaled occupancy grid map for objects as seen in Figure 3.30 below. The increase in distance between mapped points in both the increasing X and increasing Z direction in this map is a direct result of the formulas \( Z = f^*B/d \) and \( X = uZ/f \). The sizes of the circles are proportional to the area (number of pixel at that depth in that column set) of the object detected. Two shade of grey are used to assist in distinguishing between smaller and larger objects.

![Figure 3.30 Occupancy grid map to scale](image)

**3.5 Calibration Algorithm**

The calibration program used to set the variables to remove floor-ground background requires that the floor have texture enough to produce disparity data. A good example is seen in
Figure 3.31 which was used to produce the (histogram equalized) disparity map in Figure 3.32 below. Figure 3.31 and 3.32 are reproduced from Figures 3.7 and 3.8 above. There is plenty of pixel area as seen in the checkerboard-like lines in the floor in the histogram equalized disparity map (Figure 3.32). Just about any level surface will suffice for the calibration as long as it is not too cluttered with obstacles and is not monochromatic. If a monochromatic surface is all that is available during calibration, texture can be added by laying newspapers on the floor. The calibration only need to be performed when the stereo camera is installed on the AGV, or adjusted relative to the AGV.

Using live video from the bumblebee, the calibration routine produces a succession of real-time histograms of that represent the floor data as detailed on the vertical axis in the image sequence below. The vertical axis is sum of the row at that depth, and the horizontal is the sum of the columns at that depth. As the sequence progresses from 24 (nearest in depth) to 9 (the farthest in depth) the peak moves down the vertical axis at a predictable rate. This sequence keeps repeating (until program exit) to allow the setting of the Base, and Spread variables in real-time to match the cancellation area to the peak. Figure 3.33 below is a histogram of an image
after the floor data has been removed and is similar to the live histograms used in the calibration program, which also shows the histogram with the floor data removed. Figure 3.33 below is not taken from the image use to produce the 9 to 24 histogram sequence, and hence has a slightly different histogram signature.
Depth Z=8.333333
Disparity Val=18
Image File=DispGarbageCanRealDis.pgm
X Data is Multiplied by a factor of 7, Y by 4

Depth Z=8.823529
Disparity Val=17
Image File=DispGarbageCanRealDis.pgm
X Data is Multiplied by a factor of 7, Y by 4

Depth Z=9.375000
Disparity Val=16
Image File=DispGarbageCanRealDis.pgm
X Data is Multiplied by a factor of 7, Y by 4

Depth Z=10.000000
Disparity Val=15
Image File=DispGarbageCanRealDis.pgm
X Data is Multiplied by a factor of 7, Y by 4

Depth Z=10.714286
Disparity Val=14
Image File=DispGarbageCanRealDis.pgm
X Data is Multiplied by a factor of 7, Y by 4

Depth Z=11.538462
Disparity Val=13
Image File=DispGarbageCanRealDis.pgm
X Data is Multiplied by a factor of 7, Y by 4

Depth Z=12.500000
Disparity Val=12
Image File=DispGarbageCanRealDis.pgm
X Data is Multiplied by a factor of 7, Y by 4

Depth Z=13.636364
Disparity Val=11
Image File=DispGarbageCanRealDis.pgm
X Data is Multiplied by a factor of 7, Y by 4
Sequence 9-24

Figure 3.33 Histogram with floor
4 Results, limitations and benchmarks

4.1 Results

The goal of this project was to produce a practical solution for detecting obstacles in real-time using stereo vision for use on an AGV. This goal has been accomplished with a unique solution, and the algorithm may be applicable to a variety of other problems in computer vision, and as a first step in wide class of other problems.

Two movie clips are included to demonstrate the results Figures 4.1 and 4.4 below. Click on the movie figures to run the video clips. These sequences were taken in cluttered environments with varying floor background. No effort was made to idealize or optimize the data collection. Figure 4.1 is an AVI movie file of Justin Gaylor, (a University of Cincinnati computer science student) walking across the robotics lab from right to left and then left to right. There are two photos, Figure 4.2 and 4.3 taken during the sequence. You will note a yellow dot move from right to left and then left to right. During the first few seconds of each movie the calibration was being verified, so there is about a 5 second delay before the actual start.

Figure 4.4 is an AVI movie file of the robot moving around a jack stand and a garbage can. A photo of this scene can be seen in Figure 4.5. The robot navigation program is not yet complete so the robot was moved manually to demonstrate the obstacle mapping algorithm in real time. In Figure 4.4 the robot moves toward the jack stand, and then veers off to the left toward the wall, and then to the right in the clear. You can clearly see all of the “blobs” in the map moving toward the bottom of the frame, which is toward the robot. That is, all objects appear to move relative to the robot because the robot is moving. Specifically you may notice blob in the center of the frame moving toward the robot and appear to veer off to the right when
very near the bottom of the frame as the robot is turned to the left to avoid the jack stand. The wall (larger blob) appears in front as the AGV turns toward it to avoid the jack, and then a blob appear on the right, which is the large garbage can as the robot move past.

---

**Figure 4.1 Movie of live obstacle sequence: Justin Walking**

**Figure 4.2 Justin starting walk**

**Figure 4.3 Justin at center**
Figure 4.4 A jack stand and garbage can AVI movie. Robot moves through obstacles.

Figure 4.5 Photo of jack stand and garbage can taken from robot before movie sequence
4.2 Benchmarks

Speed

The speed of the X-H Map algorithm was benchmarked for speed in the following five configurations. All measurements were taken with a Dell Latitude D800 Pentium-M processor, 1.8 GHz unless otherwise noted. 1. With a single frame of a disparity image without the acquisition time of the bumblebee and NO graphical display of the obstacle graphics. 2. With a single frame of a disparity image without the acquisition time of the bumblebee and WITH graphical display of the obstacle graphics. 3. With Bumblebee live video acquisition and disparity map creation only. This establishes a baseline for the Bumblebee camera system with no contribution by the algorithm. 4. With live acquisition of the disparity image video including all camera and video data communication delays and without graphical display of the obstacle graphics. 5. With live acquisition of the disparity image video including all camera and video data communication delays and WITH graphical display of the obstacle graphics. All measurements were taken by running 100 iterations and using the _ftime function found in all version of the Microsoft C run-time libraries.

1. Single frame, NO graphics. This is a useful measurement because it gives the exact time of the algorithm itself without acquisition delays. The speed of the X-H Map algorithm is quite fast considering all of the processing that must be done on the disparity map image. On a full size PC Pentium 4 running at 2.8 GHz it took 1.25 milliseconds to process a single frame. The time on a Dell Latitude D800 Pentium-M processor, 1.8 GHz is 2.51 milliseconds. This would be the most typical use of the algorithm. During autonomous
navigation, there is no need to display the graphics. The obstacle position would simply be sent to the navigation routine. The CPU usage was 52 percent on the PC during the execution of the algorithm.

2. Single frame, AND graphics. This took 6.1 ms on the laptop and 6.4 ms on the PC. This is surely attributable to the more advanced graphics card in the laptop. The CPU usage was 35 percent on the PC during the execution of the algorithm.

3. With Bumblebee live video acquisition/ disparity map creation only. 79.5 ms per frame or 12.6 frames/second. Note that CPU usage is 100% in this mode.

4. With live acquisition, NO graphics. This could be the configuration used during AGV navigation, as no human visualization of the data is required. 80.9 ms per iteration, or 12.4 frames per second.

5. With live acquisition, AND graphics. This configuration would be helpful for setup and visualization of the navigation algorithms. 85.6 ms per frame or 11.7 frames/second. Since adding the graphical display only costs about ½ frame/sec, the will be the mode that is actually used on the AGV. With a 10 ms delay added, CPU usage is 86% and runs at 11.46 frame/sec

**Spatial distance and geometry**

Figure 4.6 below is the scaled obstacle map image produced by the program to which dimensions have been overlaid. The diverging geometry and dimensions are a physical realization of the equations \( Z = \frac{f*B}{d} \), \( X = \frac{uZ}{f} \), and \( Y = \frac{vZ}{f} \). The discrete nature of the map is a consequence discretization of the input parameters. Since the horizontal dimension 640 pixel columns of the original image was been aggregated to 20 columns, there are 20 diverging
columns. The 20 rows are a result of the 20 disparity values that were used (integer values 4 to 24). The accuracy of this design can be inferred from the figure. The localization accuracy is about 6 inches for nearby object (3.62 feet) and degrades to 4.3 feet at 18 feet away. The nearest object that can be detected is 3.62 feet or 1.103 meters. This can be calculated by using the largest disparity value in this design (d=24) and the Point Grey provided focal length and baseline f=221.58 meters B=0.119562 meters; the closes object we can detect is \( Z = 221.58 \times \frac{0.119562}{24} = 1.103 \) meters. By choosing a larger maximum disparity value of 64 for example, the minimum distance would be \( Z = 0.414 \) meters or about 1.35 feet with an unfavorable increase of processing time. Since the camera is about 3.5 feet above the ground and the Field of view is conical, little can be gained in terms of detecting low objects by choosing a larger maximum disparity unless the camera is lowered.

4.3 Limitations

The algorithm presented here does not try to identify objects. For example, it will not produce results such as “There is a red Easter basket four feet ahead, and at 27 degrees, but
instead will furnish “There is an object of area one square foot at four feet ahead and at 27 degrees”.

The results are dependent on the quality of the disparity map which is dependent on the constitution of the images. For example, a white box on a white floor would produce little information in the disparity map (and hence this detection algorithm) because of the lack of contrast and texture. To put this in perspective, humans may well trip over such an object on the floor because it “blends in”. The uncertainty due to this effect produces an equivalent uncertainty in obstacle detection. The AGV with a (any) stereo vision algorithm, may also trip over (run into) such an object in a similar situation.

Stereo vision must have enough light to produce images of sufficient quality to perform stereo processing. It will not work in the dark. Although stereo vision is considered a passive sensing technology and typically used with ambient light, it can be used with a light source. This project does not address the possibility of using light amplification devices to work in extremely low light environments.

This algorithm was not designed to provide information on the height (size and position in the vertical dimension) of an object, but rather to produce a 2D obstacle map for AGV navigation. Providing height information for each obstacle is straightforward, and becomes a problem in 3D map generation discussed in “Future Work”.
5 CONCLUSIONS AND FUTURE WORK

5.1 Conclusions

The overall obstacle mapping method for the University of Cincinnati AGV robot was discussed. A robust algorithm was presented for the processing of disparity maps in real time to create the obstacle map. The algorithm features roll and tilt compensation and cancellation for floor noise that is fast, and effective. This method may have advantages over many other existing methods in several applications, and may be useful as a first step in other problems in computer vision.

5.2 Future Work

The technologies for stereo obstacle detection are still in their infancy. Although the results presented here are effective, much can and will be done to improve the area of research in obstacle detection using stereo video. Given that the set of future work is therefore large indeed, this section contains items that I consider “the low hanging fruit”. That is, items that have rather large returns in performance for a given amount of effort.

5.2.1 Connected component detection to improve detection robustness

A simple and obvious immediate extension of this technique is to use connected component detection on the information in the occupancy grid map. Consider for example Figure 5.1 below. Note the red circle near the bottom of the map. This circle contains a cluster of three adjacent “objects” which are really one object whose information in contained at three separate depths.
Using connected component detection in obstacle map space has several advantages over using the same technique in 3D point cloud space or even disparity space for obstacle detection. Searching in a 3D point cloud or disparity space could require orders of magnitude increase in the computation load over similar a similar technique in the obstacle map space.

A non-optimized direct search is made over this entire occupancy grid to find adjacent objects would require searching over $20 \times 20 = 400$ points and is restricted to two dimensions. If a similar search was made in 3D space or disparity space to search for adjacent pixels, the search would require $640 \times 480 \times 20 = 15,360,000$ points, a factor of 3840. It is granted that various methodologies exist to optimize the search, but this example suggests the advantages of connected component detection in occupancy space. It is worth mentioning that connected component detection is implemented quite efficiently in the “Connected component detection via contour functions” in the DrawContours example in the OpenCV documentation.
5.2.2 Shape fitting to identify basic obstacle types

Often a search is made for a known obstacle. It may be helpful to identify these objects by matching their basic shapes. Shape fitting could dramatically improve the obstacle detection ability in circumstances where an object has good contrast, but poor texture. A dramatic example would be a large black sheet of plywood standing vertically in a white room. The large black sheet would produce little disparity information except around the edges due to the lack of color/texture variation. But by knowing that the plywood sheet is one of the objects you are looking and finding a geometric fit to this obstacle would greatly increase the likelihood that the object was detected. The general case of contour matching was addressed by M. Hu [Hu62]. Hu demonstrated that a set of seven features derived from the second and third moments of an arbitrary closed contour are invariant to translation, rotation and scale changes. Various functions in the OpenCV library such as “MatchShapes” (which make use of Hu’s results) could greatly simplify this procedure. A very simple demonstration of this method can be found below that demonstrate the power of the OpenCV library. Using an OpenCV example program (one hundred lines of c code, most of which are overhead) that finds contours and fits ellipses I quickly was able to produce “elliptical” contour of “case with handle” as seen in Figure 5.2 below. The program basically uses just two powerful and fast functions cvFindContours() and cvFitEllipse(). In practice, a search for shapes could be made for any set of items of interest, such as triangles or more complex shapes. Depending on the application, the search for the shapes may not need to be over the entire image, but only near the areas where an object was detected using the main algorithm.
Figure 5.2 Image of Case with handle

Figure 5.3 Contour fitting: Case with handle
5.2.3 Improved floor detection and cancellation

The algorithm presented here makes the assumption that that floor is “flat” to a relative degree. If the floor has significant 3D gradients, problems may arise. For example bumps may appear as small obstacles, and a steep ramp may appear as a long low obstacle. A simple method to detect a ramp is to look at the progression of the peaks in the histogram, as detailed in the image sequence show 9-24 above. Since the camera is calibrated with a flat and non-sloping ground plane, the progression of the peaks will have a known value. If the distance between floor peaks is closer, the ground plane is sloping upward, and if the distance between peaks is larger, the ground plane slopes downward.

Since floor detection is important in many obstacle detection algorithms, more complicated methods have been used, such as trying to fit a plane to a set of 3D points in the a bounded region assumed to contain the ground plane by choosing several sets of random points and choosing the best fit for a plane [Burschka 2002]. However, a method that I would prefer is to use the OpenCV function FitLine3D() which fits a 3D line to a set of points in 3D space. By using the PGR library function such as triclopsRCD8ToXYZ(), which converts the Row, Column, Disparity value to X,Y,Z in the region of the floor. By breaking up the floor into small interlaced triangles, points in the triangle could be fit to a line, and all lines connected to fit a ground mesh that corresponds to the true ground gradient.

5.2.4 Rapid obstacle location and detection in 3D

This thesis presented a method to create a 2D map of obstacles that exist in 3D space using a disparity map. A straightforward extension to 3D space using this method would be to create an Y-Z map as we did the X-Z map, and use a Fourier Transform as is used in
Computer Aided Tomography (CAT) on each depth plane to find the shape of each object in 3D. But this is a rather round-about method, since these shapes already exist on each plane of the map (see Figure 5.4). Why convolute the data only to de-convolute it? The strength of this method to rapidly produce 2D occupancy grids from disparity maps is weakened when generalizing it 3D.

A more efficient method to locate objects in 3D using a disparity map may be to reduce the resolution by image pyramiding and using connected components on each plane of the disparity map. Connected components could then be used on the X-Z map produced the algorithm presented here to link areas that are separated by one depth plane to greatly reduce the search space between planes in disparity space. This may prove faster and/or simpler than the direct application of a three dimensional "connected components" algorithm.

A standard three dimensional "connected components" algorithm [Griener 1994] perhaps could be optimized for a disparity map based on the following constraints. A disparity map is a two dimensional representation of 3D space. Since a camera cannot “see through” objects, each pixel maps to simply one depth. A disparity map is therefore not really “full 3D” and not 2D. It may be thought of as “2.5D”. Areas of connected pixels on individual planes of the map are unique to a single depth. Figure 5.4 below is depth plane 18 (dark gray) and depth plane 19 (light gray). Pixels from the floor and the garbage can are seen in both planes. Any single 3D object (such as the garbage can) can exist in two or more planes which are adjacent planes, but the pixels can not overlap. If pixels are adjacent in the disparity map, and also in adjacent planes, it may well be the same object. For many applications, it could be classified as the same “object” and located in 3-space at that level of computation. For example, a red box on top of a black box for practical purposes would be the same object in the problem of AGV navigation.
Other methods such as color, texture, contour, area or 3D transformation mapping could be used in that Region Of Interest (ROI) for an exact identification of the object in 3D.

Figure 5.4 Two planes from disparity map

Figure 5.5 Histogram equalized disparity map
References


http://www.intel.com/research/mrl/research/opencv


http://www.ptgrey.com


