The XH-map algorithm: A method to process stereo video to produce a real-time obstacle map.

Donald Rosselot and Ernest L. Hall
Center for Robotics Research
Department of Mechanical, Industrial, and Nuclear Engineering
University of Cincinnati
Cincinnati, OH 45221-0072

ABSTRACT

This paper presents a novel, simple and fast algorithm to produce a "floor plan" obstacle map in real time using video. The XH-map algorithm is a transformation of stereo vision data in disparity map space into a two dimensional obstacle map space using a method that can be likened to a histogram reduction of image information. The classic floor-ground background noise problem is addressed with a simple one-time semi-automatic calibration method incorporated into the algorithm. This implementation of this algorithm utilizes the Intel Performance Primitives library and OpenCV libraries for extremely fast and efficient execution, creating a scaled obstacle map from a 480x640x256 stereo pair in 1.4 milliseconds. This algorithm has many applications in robotics and computer vision including enabling an "Intelligent Robot" robot to "see" for path planning and obstacle avoidance.

Keywords: Autonomous, robot, stereo vision, disparity map, XH-map algorithm, OpenCV, computer vision, video

1. INTRODUCTION

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

The core of the XH-map algorithm uses a disparity map and:

- Removes floor-ground background and other unwanted "noise".
- Separates the image into depth planes.
- Aggregates column information into smaller usable sets and sum those sets by depth to produce obstacle information at each depth.
- Uses that information to produce a "live" occupancy grid of objects.

The algorithm creates a calibrated obstacle map in real-time using a commercially available stereo camera system. Detecting obstacles using stereo vision in an application such as navigation for 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 such as obstacle mapping while maintaining high information velocities with a standard Pentium 4 computer requires resourceful algorithms and careful programming techniques.

Reliable obstacle detection and mapping of objects to 2D or 3D space is a first step in a multitude of problems in computer vision such as navigation and path planning, object recognition, and object tracking. The XH-map algorithm create a "floor plan" of obstacles. Height information of objects is purposefully lost in the 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 dynamic rolling and pitching 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. METHODS

2.1 The stereo video acquisition system

The stereo video acquisition system used is the Bumblebee™ stereo vision system by Point Grey Research (PGR) [1] and used with a Dell Latitude D800 Pentium-M processor 1.8 GHz laptop computer, which is roughly equivalent to a 2.4 GHz Pentium 4 desktop computer. 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 [2], and a 400 Mbps IEEE- 1394 Firewire interface for high speed communication of the digital video signal. The bumblebee camera used for this work has a 6mm lens and 100º HFOV and the Black and White CCD. The calibration information is factory preload into the camera allowing the computer software to retrieve it for XYZ coordinate calculations and image correction. The time synchronization is 125µs maximum deviation for each stereo image pair. Good time synchronization is required to produce quality disparity maps with moving objects or camera. The Bumblebee is advertised to acquire images at 30 frames per second. This rate does not include the time it takes their software to calculate the disparity map on a local computer. On our Dell laptop, their acquisition and disparity calculation operates at about 12 frames per second for the processing of one frame. Using the PGR function **`triclopsGetImage`**, image frames can be grabbed and placed directly into the Intel IPP or OpenCV **`IplImage`** structure for performing computer vision processing. 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 PGR SDK uses the concept of a "context" which is a structure for storing information about the stereo images, calibration and timing.

The loop function sequence from the PGR SDK to communicate with the Bumblebee and to continuously create disparity maps for this obstacle avoidance algorithm is the first five functions in the loop and is later discussed as GrabDisparityImage() and be summarized by:

```
Begin Loop
  digiclopsGrabImage(digiclops);
     Grab the image from the camera and store all information in the digiclops context
  digiclopsExtractTriclopsInput( digiclops, STEREO_IMAGE, &inputData );
     Extracts the data from the context and puts it into the inputData data structure.
  triclopsPreprocess( triclops, &inputData );
     Unpacks and rectifies the image and may do smoothing, and edge detection if requested.
  triclopsStereo( triclops );
     This function performs stereo processing on the image to produce the disparity map.
  triclopsGetImage(triclops, TriImg_DISPARITY, TriCam_REFERENCE, &depthImg );
     This function retrieves the disparity map image from the triclops context and places it into depthImg.
  "Process obstacle avoidance algorithm using disparity map(depthImg)"
     This is the XH-map Algorithm and is expanded and discussed in detail below.
End Loop
```

2.2 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 algorithm uses the software library Intel Performance Primitives (IPP) [3] 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. The Intel OpenCV library [4] 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. XH-map ALGORITHM: EXTRACTING OBJECT POSITION FROM DISPARITY MAPS

3.1 XH-map algorithm overview

Figure 3.1 is one image from a stereo pair taken inside the UC robotics lab with a garbage can in the center of the floor. Figure 3.2 is a disparity map from this stereo pair that has been grey scale balanced (Histogram equalized) for better viewing. All calculations on disparity maps in this algorithm are on the raw disparity maps (as opposed to equalized) unless otherwise noted, which is typical. Consider a disparity map image 480 pixels high, 640 pixels wide and 256 (0-255) levels in gray-level depth. Visualize this map as 256 separate images, one for each gray level or depth plane. The depth planes can be thought of a slices in depth of the three dimensional world. Each slice will have information about the world at that depth. Figure 3.3 is an example of a depth plane taken from the disparity map of figure 3.2. Two depth planes or "world slices" were chosen at the depth of the garbage can and the pixels can be clearly seen. Depending on depth resolution and object size and orientation, objects will generally appear at two or more image depth planes. By summing each column or row at each depth plane, a profile function at that depth can be obtained as sketched in Figure 3.2. A large peak in the vertical profile usually is the "signal" from the floor, and peaks in the horizontal profile are often objects at that depth.

Figure 3.1 Indoor image of garbage can.

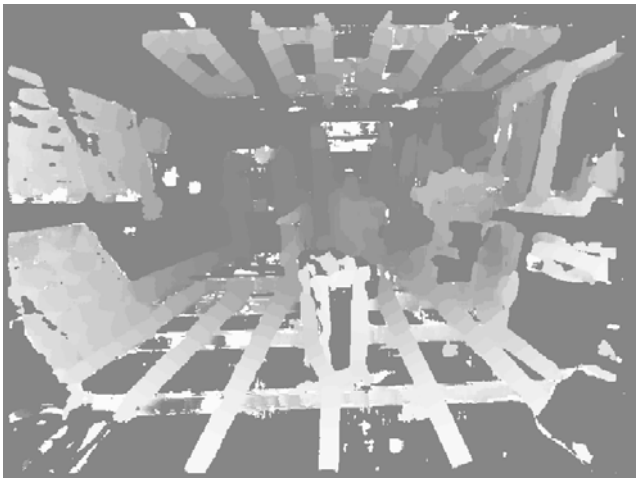

Figure 3.2 Histo equalized disparity map.

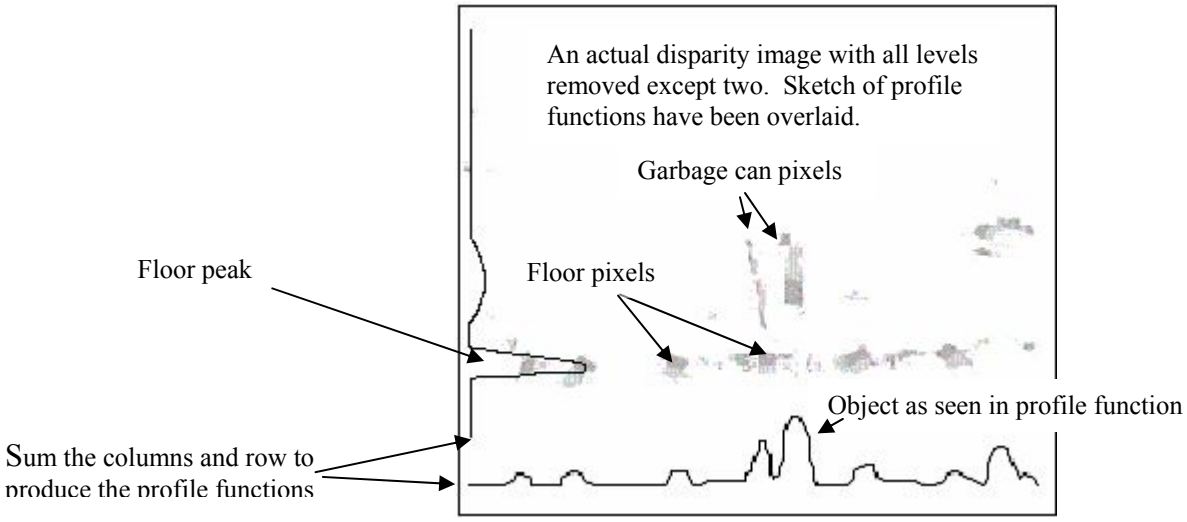

Figure 3.3 Actual depth plane from a disparity map

3.2 Floor Noise cancellation

Using the Matlab [5] Image processing library, Figures 3.4 through 3.6 below were created for the purpose of illustrating the concepts of Floor cancellation. By summing the rows (the Y axis) and the columns (the X axis) of a single depth plane (disparity value 22) Figure 3.4 was created from the (original) disparity image of Figure 3.2. It is difficult to determine where an object is by looking at the X axis. It appears that there may be 5 or more objects based on the signal peaks. These peaks correspond to the thin black lines on the floor as seen in Figure 3.1. Note that the Y axis direction is reversed (Y axis positive is down) as is common in image processing. The spike near the top along the Y axis is the signal from the floor. The strategy here is to remove the floor data by using information from the Y direction. During the calibration stage, the height of the floor is determined at each depth to enable removing that signal during runtime. By removing the signal at the floor, amazingly clear signals produced by the actual objects-of-interest can be seen.

Figure 3.5 (at disparity value 22) and Figure 3.6 (disparity value 19) are examples of this process. Note that larger disparity values correspond to closer distance. Figure 3.5 is at a level near the front surface of the garbage can and Figure 3.6 is in the plane of the garbage can. It can be seen that the floor data (yellow dotted line) has been drastically attenuated in Figures 3.5 and 3.6. The signal centered at 325 on the X axis (red solid line) that appears poorly attenuated is actually an initial signal from the nearest edge of the garbage can. At disparity value 19 (Figure 3.6), it is extremely clear that there is an object at this location. The signal before attenuation is shown with dotted yellow lines and is overlaid by the red attenuated signal.

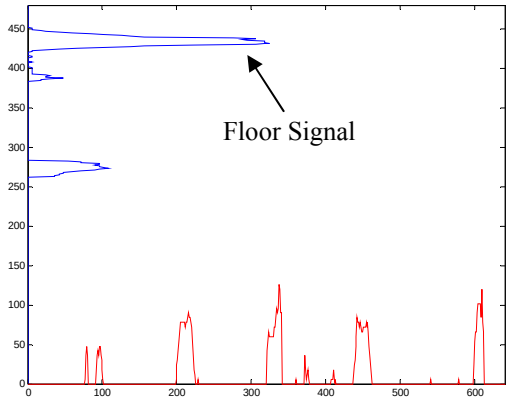

Figure 3.4 Floor signal

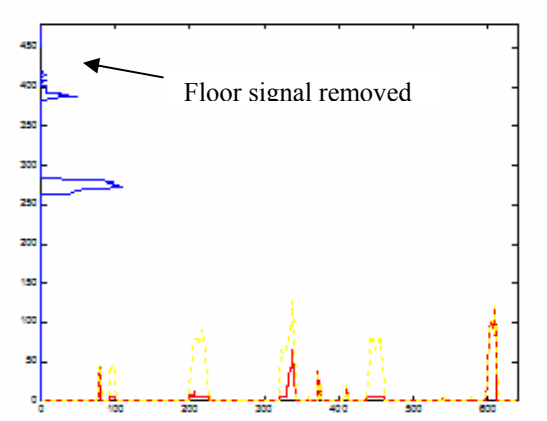

Figure 3.5 Floor Signal Cancellation

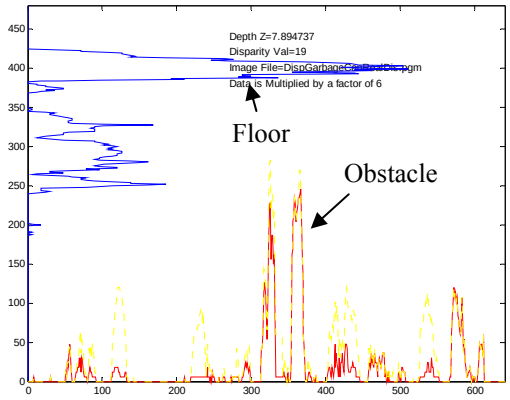

Figure 3.6 Profile function of obstacles and floor

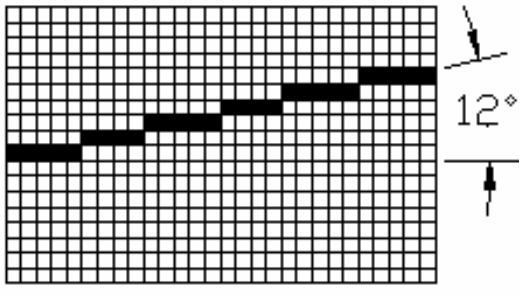

Figure 3.7 Tilt Compensation

Expansion on some of these key concepts can be found in [6] and [7]. Sample images and Matlab code to produce these figures are at http://www.ececs.uc.edu/~rosseldw/RobotStereoData.zip.

3.3 Implementation of the XH-map algorithm

In practice, to find objects at each depth by summing rows and columns by the process of dividing an image into 255 image planes would prove inefficient computationally. A trivial approach would require 480*640*255=78,336,000 summations for each frame, and the current implementation runs at 12 frames a second. To achieve that throughput would thus require 78,336,000*12=940,032,000 summations each second. The task-at-hand is to take these concepts and produce an algorithm that is fast and efficient.

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#. 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 implementation in relies on the basic principles discussed above but uses several innovations to obtain efficiency. 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)
End Loop
```

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. Each of these steps are now discussed in more depth.

1. GrabDisparityImage(imgDispMapSrc) Grabs a stereo pair from the video stream and outputs a disparity image imgDispMapSrc(). This function was discussed in detail above.
2. 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 robot. Roll compensation is required to keep the floor-ground information from the cameras in the same relative position as determined at calibration. For example assume that the floor information at that depth was level at calibration, but now has been rotated 12 degrees as sketched in Figure 3.7 above. By detecting the rotation with a tilt sensor and counter rotating the disparity image 12 degrees the floor data will once again be at level and can be removed. Referring to Figure 3.6 below, roll is defined as α, pitch (forward/back tilt) is defined as γ. 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 γ (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 accounted for. 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.

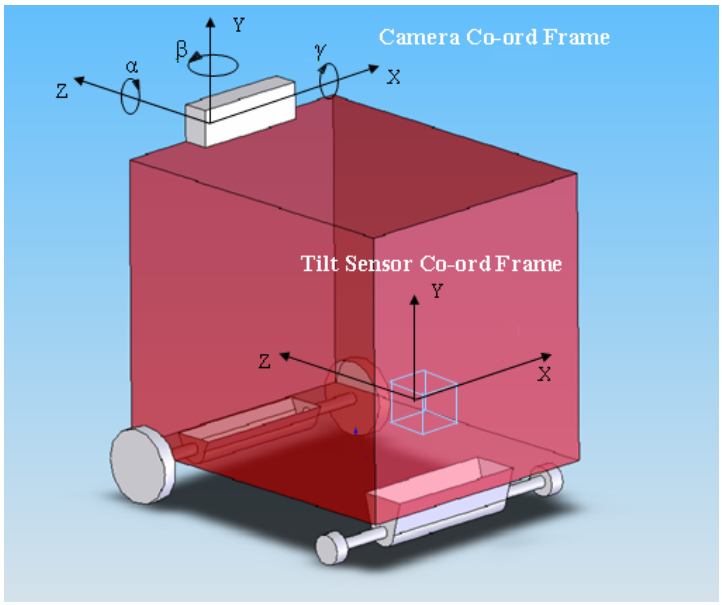

Figure 3.6. A coordinate system for tilt compensation

3. RemoveFloorBkGnd(imgDispMapSrc, imgDispMap, FloorIncrement, Spread, Base) Removes the floor information from the disparity map. 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. 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 one 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 pixel 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). The numbers forty and nine are dependent on the properties of the camera such as focal length and of image resolution, both which never change in our application. *FloorIncrement* is also the amount the floor peak advances in units of pixels per increase in depth during the calibration stage. Figure 3.9 below is one depth plane (depth level 20) from a disparity map. Figure 3.10 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 imgDispMapSrc is the input image, and imgDispMap is the output image.

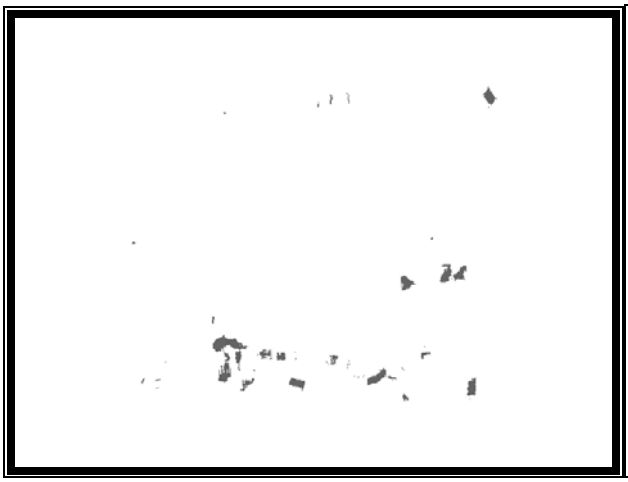

Figure 3.9 Level 20, data intact.

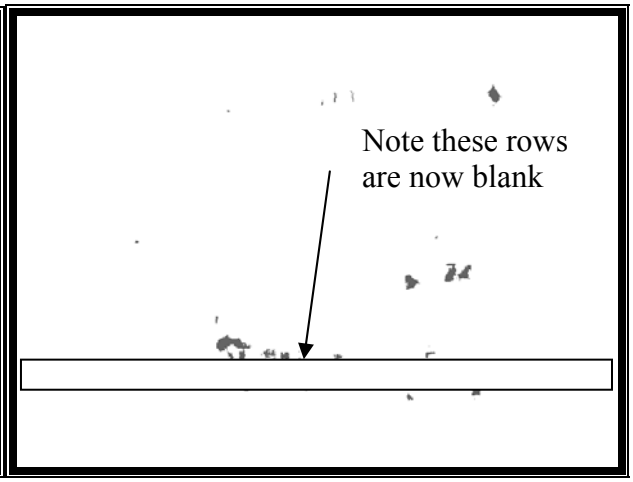

Figure 3.10 Floor cancellation. Level 20, data removed.

4. 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.
5. 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 (using a Region of Interest) 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 loops through 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 12 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 XH-map matrix as can be seen in Figure 3.11below. 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 3.11 below.

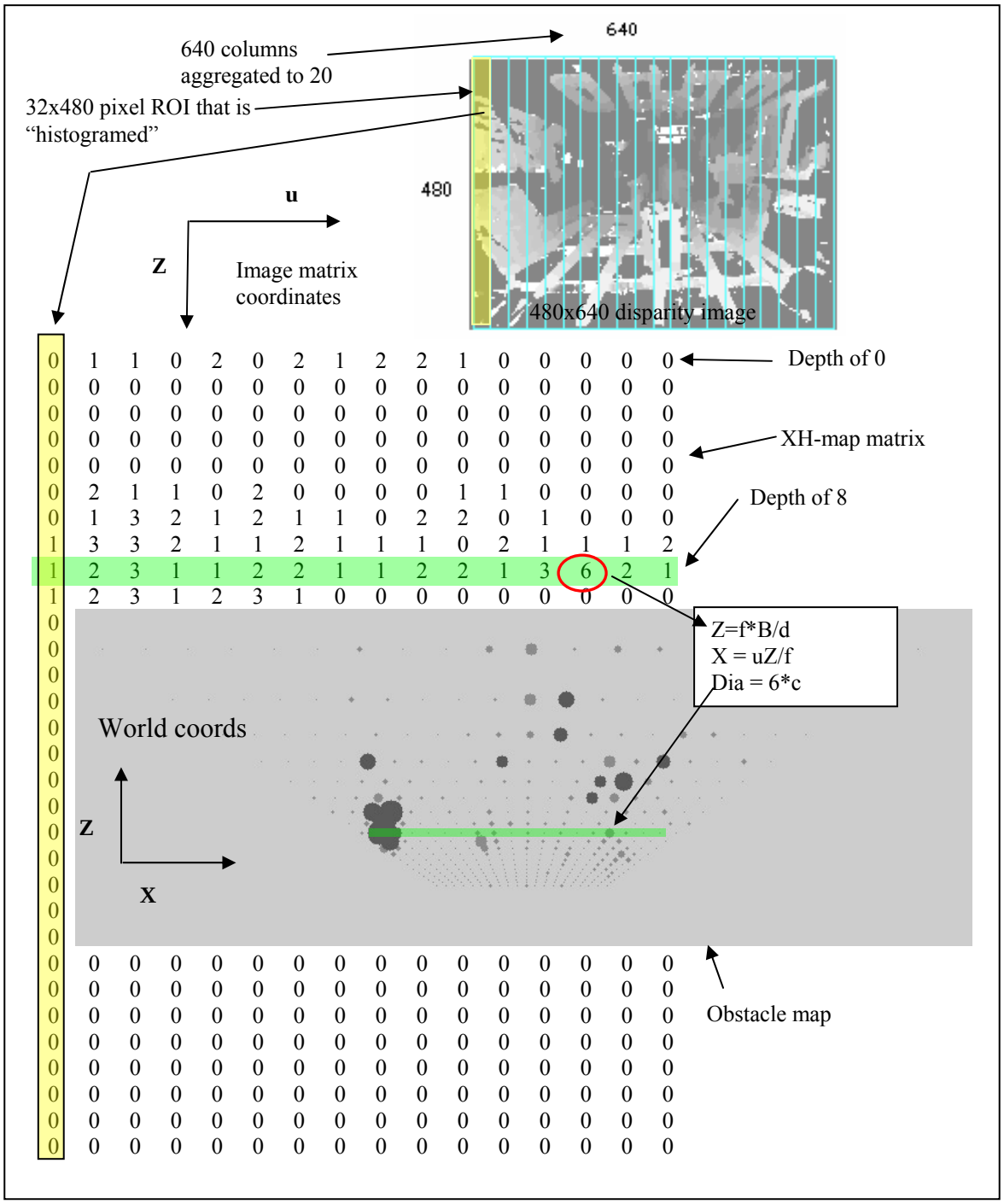

Figure 3.11 Creating the X-H Map

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 XH-map a matrix of numbers is obtained. Each column of the original disparity image map image is now transformed

into an XH-map matrix of 20 by 32 numbers. Figure 3.11 contains only 16 columns rather than 20 merely so the graphic fits on one page. The X-Z coordinates of a number (for example, number 6 at depth 8 as circled in Figure 3.11) can be calculated by the equations Z=f*B/d, and X = uZ/f and plotted as in the grey obstacle map in the Figure. The diameter of each obstacle is calculated by D=q*c, where q is the value at that point in the XH-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.

4.0 CALIBRATION ALGORITHM

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 in Figure 4.1 below. Although the algorithm iterates through all depth planes of interest, only four planes are shown below for brevity. 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 (nearer in depth) to 14 (the farther 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.

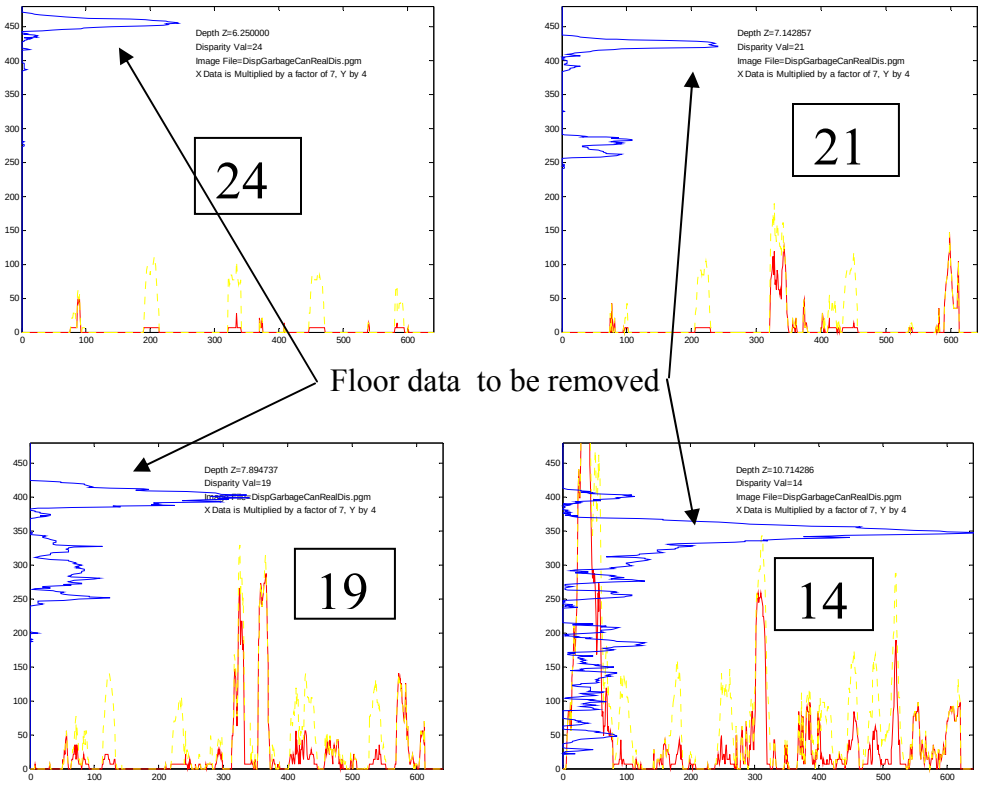

Figure 4.1 Sequence depicting floor noise

5. RESULTS AND DISCUSSION

The visual output of this algorithm is the scaled obstacle map seen in the dimensioned Figure 5.1 below. In actual use, the matrix of numbers representing the obstacle map can be used for a variety of purposes including path planning of a robot. Two AVI movie clips in a 5Meg zip file can be found at http://www.ececs.uc.edu/~rosseldw/XHmapDemos.zip to demonstrate the algorithm producing an obstacle map in real time with live video data. These sequences were taken in cluttered environments with varying floor background. No effort was made to idealize or optimize the data collection or results. The first is a movie file rendering an obstacle map created from video of a person walking across the robotics lab. The second is the real time map of obstacles relative motion as the robot moves down a cluttered hall. Details and further explanation are in the results section of [7].

5.1 Speed

The speed of the XH-map algorithm is remarkably fast, executing a single frame in 1.4 milliseconds. All measurements were taken by running 100 iterations and using the _ftime function found in the Microsoft C run-time libraries. This speed was benchmarked for speed in several configurations including. **1.** 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 and was 79.5 ms per frame or 12.6 frames/second. Note that CPU usage is 100% in this mode. **2.** With live acquisition of the disparity image video including all camera and video data communication delays was 80.9 ms per iteration, or 12.4 frames per second. With a 10 ms delay added, CPU usage is 86% and runs at 11.46 frame/sec with live acquisition.

5.2 Spatial distance and geometry

Figure 5.1 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=f*B/d, X = uZ/f, and Y = vZ/f. The discrete nature of the map is a consequence discretization of the input parameters, that is the horizontal dimension 640 pixel columns of the original image was been aggregated to 20 columns, so 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 somewhat 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 closest object we can detect is Z = 221.58 * 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 a slight increase of processing time.

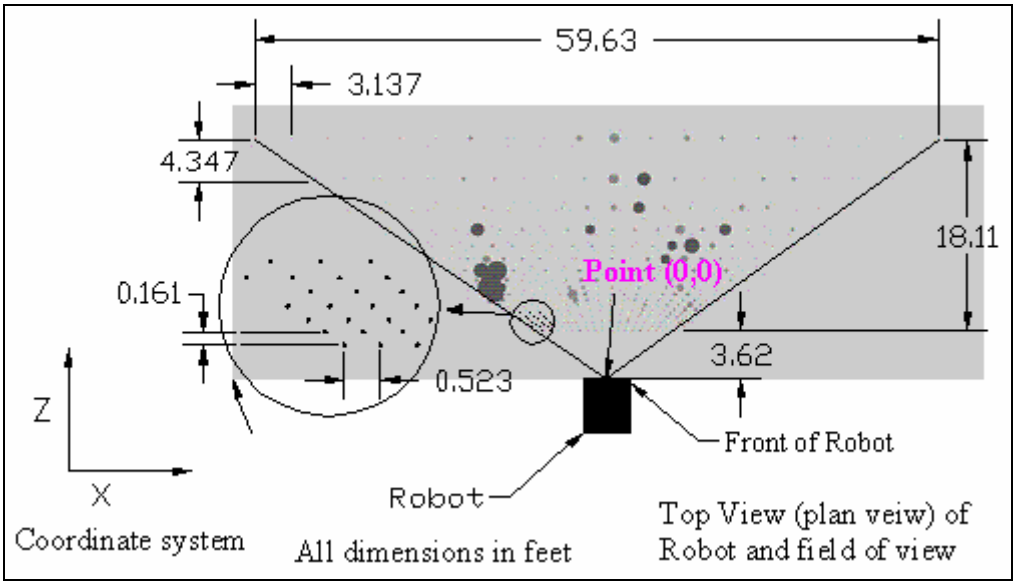

Figure 5.1 Dimensions of the field of view

5.3 Discussion

The XH-map algorithm may be applicable to a variety of other problems in computer vision, and as a first step in wide class of other problems. For example, in trying to identify an object in space the first step would be a quick search using the X-H May algorithm to determine the size of the object matches the expected value. A straight forward extension of this work would be to use connected components to combine adjacent objects to obtain a more realist obstacle footprint.

REFERENCES

1. Point Grey Research. http://www.ptgrey.com/
2. Point Grey Research, "Triclops StereoVision System Manual Version 3.1" 2003. http://www.ptgrey.com
3. Intel Corp, "Integrated Performance Primitives for Intel®Architecture Reference Manual", Volume 2: Image and Video Processing, Volume 3: Small Matrices, 2003.
4. Intel Inc., "The OpenCV Open Source Computer Vision Library", http://www.intel.com/research/mrl/research/opencv
5. Matlab and The MathWorks Inc, "Image Processing Toolbox", Natick, MA, 2001. www.mathworks.com
6. D. Rosselot, and E. Hall, "Processing real-time stereo video for an autonomous robot using disparity maps and sensor fusion", Proceedings of SPIE Volume: 5608, pp70-79, Oct 2004.
7. D. Rosselot, " Processing real-time stereo video in disparity space for obstacle mapping", MS Thesis, University of Cincinnati, 2005.