Computer Vision Detection of Negative Obstacles with the Microsoft Kinect

Luke Wang
Russell Vanderhout
Tim Shi

Project Sponsor:
Dr. Ian Mitchell

ENPH 459
Engineering Physics
The University of British Columbia

Project Number: 1220

April 2, 2012

Executive Summary

The objectives of this project were to use the Microsoft Kinect to develop an obstacle detection system for wheelchair users and output the obstacle data in a way useful for future projects using the system. This project was done as part of the CanWheel project, which is meant to improve the mobility of old adults using wheelchairs.

The Kinect is essentially a 3-D camera which can produce depth images, images in which each pixel contains distance data. The Kinect is attached to the back of the wheelchair and aimed towards the floor. Obstacles may then be found by using the depth data and locating parts of the depth image which do not belong to the floor. This system is designed for indoor use, as it requires that the ground is a flat surface. Also, since the depth camera transmits and receives infrared waves, sunlight causes a disturbance.

The purpose of this report is to provide technical details related to the obstacle detection system, explain the method used, describe the results, and make future recommendations. Background information includes details of pinhole camera analysis and the stereo camera parallax. The algorithm used to find obstacles consists of detecting planes to find the floor, then determining which pixels correspond to obstacles. Finally, all the detected information is all mapped onto an overhead view.

It is recommended that further optimization of computational resources is done with the application of multi-threading CPU, General Purpose GPU and FPGA. One may also modify the algorithm based on their specific use case to further optimize the software.

Contents

List of Figures

List of Tables

1 Introduction

1.1 Background

The Kinect is a 3-D camera developed by Microsoft for their Xbox 360 video game console. Since its release on Nov. 10, 2010, it has been widely used for research and development purposes. The Kinect contains an RGB camera, an infrared camera used to generate depth data, and an accelerometer. The Kinect also has an internal stepper motor used to adjust its tilt angle. The system developed in this project analyzes the depth data frame by frame, identifies planes within each depth image, and uses accelerometer data to find which planes correspond to the floor. Detailed algorithm and mathematical analysis will be discussed later.

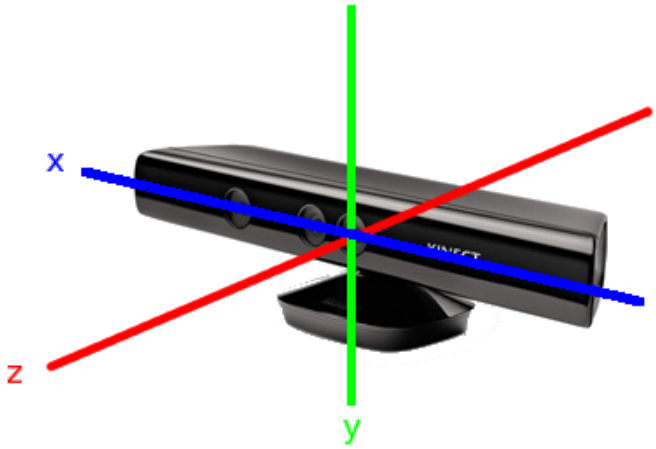

Figure 1: The Kinect
The Kinect axes are shown. From the camera view, the z-axis travels through the image, the y-axis travels up and down, and the x-axis travels left and right.

For this project, the Microsoft Kinect was used to develop a system for electrical wheelchair navigation, which involves identifying safe areas for the wheelchair to travel on. The Kinect is a cost-efficient 3-D camera, and the 3-D data it collects from the environment is used to find obstacles. This report explains the steps of the work process, as well as the algorithms used along with the necessary mathematical analysis. This project is a part of the UBC CanWheel project[6].

All code for this project was developed under Linux (Ubuntu 11.10) and Mac

OS X (10.6/10.7) with the open source integrated development environment Eclipse. Code was mainly written in C, and the compiler used is gcc 4.6.1. The driver we used for communicating with Kinect is Libfreenect. It is an open source driver developed by OpenKinect[5]. OpenCV[2], an open source library, was used to standardize the basic image data structure. It was also used to create a simple GUI to control the Kinect's tilt angle, algorithm procedure and results display.

1.2 Project Objectives

Initially, the project goal was to develop an algorithm that detects negative obstacles when the wheelchair is going backwards. However, the goal was not meant to be exclusive to negative obstacles, but was only defined in that way as there already exist many well-developed techniques for positive obstacle detection (i.e. Ultrasonic detection, laser beam detection). Negative obstacles are defined as a sudden drop in height, such as the edge of a cliff or a hole in the ground. Positive obstacles can be seen as an increase in height, such as objects lying on the floor. It was found that the algorithm developed allowed for simultaneous positive and negative obstacle detection. As the project sponsor requested, the final output of the system is an overhead view map with reference to the wheelchair, coloured to show safe, unsafe, and unknown areas.

Since our project will become part of the CanWheel project, our design of the program makes sure that the program can be easily modified to a function library. The CanWheel project is still under development, so we need to provide the flexibility to change and optimize the program.

1.3 Scope And Limitations

Our algorithm of floor detection does not convert the depth data to point cloud data, which saves computational resources. It can process 6.5 frames per second using a 2.2GHz Intel Core Duo CPU.

We can apply an algorithm directly to the depth data is due to the low noise level of the active stereo system. As a result, our algorithm is not suitable for a passive stereo camera. The Kinect's IR signal would be interrupted by strong sunlight, therefore, it is not suitable for outdoor usage. For indoor usage, it seems to be working well most of the time based on our experience.

2 Discussion

2.1 Theory

In this section we will discuss the two fundamental theories for the analysis and modelling of the Microsoft Kinect.

2.1.1 Pinhole Camera Model

A lens is spherical and it maps straight lines into curved lines on its focal plane. Therefore, without rectification, the lens camera will generate distorted images.

Figure 2: Image Rectification
An image before rectification (left) and after (right)[1].

An study on ROS shows that the output images of Kinect have extremely low distortion. This suggests that the Kinect has already rectified images[4]. Therefore we can directly apply our algorithms on the raw output from the Kinect and model the Kinect as a pinhole camera.

A pinhole camera has a small aperture and no lens. Light from a scene passes through a single point and projects an inverted image onto a plane on the other side of the pinhole, as shown in Figure 3.

The depth camera for the Kinect is similar; the depth image enters a rectangular CCD (charge-coupled device) sensor, so the view angles may be determined by physical measurements of the CCD and perpendicular distance from the CCD to the pinhole by the following equations:

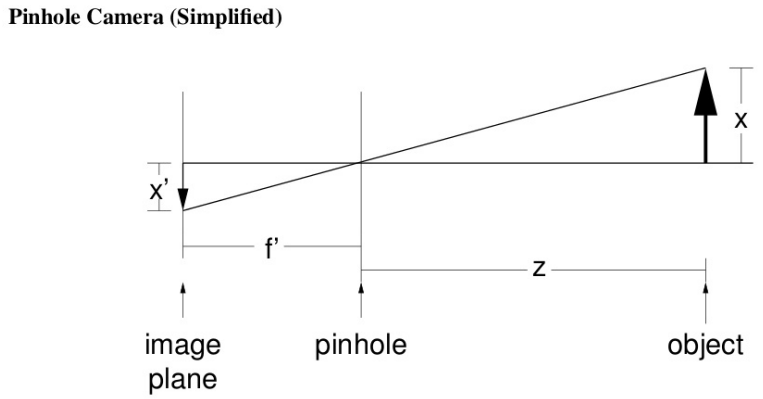

Figure 3: Pinhole Camera Diagram

Image points are mapped to the image plane by travelling along a straight line through the pinhole.

$$Horizontal\ view\ angle\ = 2 \times \arctan(\tfrac{Vmax}{2 \times f})$$
$$Vertical\ view\ angle\ = 2 \times \arctan(\tfrac{Hmax}{2 \times f})$$

$Vmax$ and $Hmax$ are the vertical and horizontal lengths of the CCD respectively and f is the distance to the pinhole. The horizontal and vertical view angles are approximately 58° and 45° respectively (see Appendix for Kinect data sheet).

2.1.2 Stereo Parallax and Distance

The Kinect depth sensor works by using an IR camera and an IR projector. It can be modeled as two pinhole cameras facing the same direction, with the pinholes separated by a distance of B, as shown in Figure 4. An object located at distance D away from the focal plane is detected at x by the left camera and at x' by the right camera. The sum of x and x' from center of their image plane is called the parallax ($P = x + x'$).

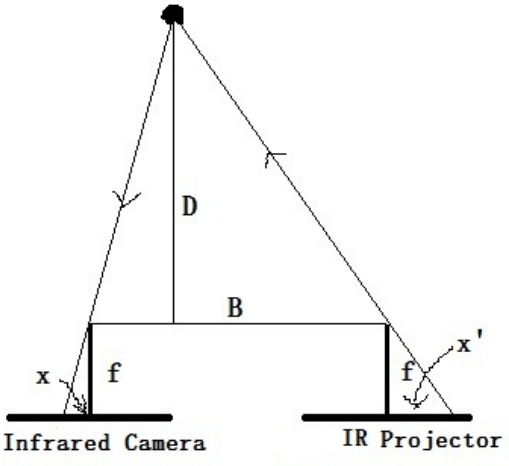

Figure 4: Stereo Parallax

By similar triangles, the distance may be found through $D = \frac{f \times B}{P}$. This is how the Kinect calculates the distance to an object. The object's Cartesian coordinates $[X, Y, Z]$ in the reference frame of the Kinect can be calculated by the following equations:

$$X = \frac{(x - x_{center}) * B}{P}$$

$$Y = \frac{(y - y_{center}) * B}{P}$$

$$Z = \frac{f * B}{P}$$

Developers have experimentally determined the following relationship between $DepthValue$ from Kinect and $Parallax$.

$$Parallax = (1090 - DepthValue) * \frac{1}{8}$$

2.2 Methods

Obstacles consist of anything which the wheelchair can not travel over, such as a wall, objects on the floor, or a hole in the ground. Obstacles can be classified as deviations from the plane defining the floor.

The method of finding obstacles consists of determining planes which make up large sections of the depth image. Anything that is not part of a plane is an obstacle. It then uses the accelerometer to determine which of these planes are close to level, since planes such as the wall are also obstacles. Finally, it checks each pixel to see if they are within one of the safe planes. This analysis is done on every frame of the depth video, so it is useful to avoid code with heavy calculations. Other methods considered consisted of using point cloud data, but such algorithms would likely be too slow for the purposes of this project.

2.3 Algorithm

2.3.1 Plane Analysis

The depth image is divided into several square blocks, referred to as ROIs (Regions of Interest). The method of finding planes in view of the depth camera assumes that at least one ROI is contained within the plane. For each ROI, it is determined whether or not the depth data strictly within the ROI corresponds to a plane, for those which are planes, the parameters defining the plane are determined. The reason ROIs are square is to have the same number of rows as columns in determining the plane parameters, which may or may not be important.

The ROI side length used is 70 pixels. It was not determined as to whether or not this is the optimal value, but it does result in a working system. Users may change this value if seen as necessary. Values which are too low would result in too many ROIs to analyze, which would slow down the performance, and ROIs may not have enough data to accurately determine the plane. ROIs which are too large are less likely to be contained within a plane. The ROI grid starts from the left, and is offset from the bottom by 3 pixels (the 3 bottom rows and 3 rightmost columns are outside of the IR projection range and do not contain any data). Not every pixel is placed into an ROI, since 70 is not a factor of either dimension (637 and 477); This results in some lost information. For someone wanting to minimize the number of unused pixels, Table 1 shows the five best values, in the range of 7 by 5 to 14 by 10 square ROIs.

Our chosen value of 70 results in 39249 unused pixels which is far from optimal in terms of pixel use. However, the system is able to detect planes, so it is likely that the number of used pixels is sufficient.

ROI side length (pixels)	# of ROIs	unused pixels
53	12 by 9	477
79	8 by 6	4281
52	12 by 9	11817
78	12 by 9	11817
68	9 by 7	12537

Table 1: ROI Sizes for Least Unused Pixels
This shows ROI sizes corresponding to the lowest number of unused pixels, for a reasonable number of ROIs (in the range of 7 by 5 to 14 by 10).

An ROI is considered to be a plane if its variance value is low enough. In this system, the variance value of an ROI is calculated by taking the 70 by 70 block of pixels, folding it in half horizontally and vertically and adding the overlapping depths to get a 35 by 35 matrix, as shown below. The variances of the entries in this resulting matrix is taken as the ROI variance.

$$\begin{bmatrix} d_{1,1} & d_{1,2} & \cdots & d_{1,70} \\ d_{2,1} & d_{2,2} & \cdots & d_{2,70} \\ \vdots & \vdots & \ddots & \vdots \\ d_{70,1} & d_{70,2} & \cdots & d_{70,70} \end{bmatrix} \rightarrow \begin{bmatrix} d_{1,1}+d_{1,70}+d_{70,1}+d_{70,70} & \cdots & d_{1,35}+d_{1,36}+d_{70,35}+d_{70,36} \\ d_{2,1}+d_{2,70}+d_{69,1}+d_{69,70} & \cdots & d_{2,35}+d_{2,36}+d_{69,35}+d_{69,36} \\ \vdots & \ddots & \vdots \\ d_{35,1}+d_{35,70}+d_{36,1}+d_{36,70} & \cdots & d_{35,35}+d_{35,36}+d_{36,35}+d_{36,36} \end{bmatrix}$$

Perfect planar data would produce a variance of zero. It is assumed that the maximum allowed variance is proportional to the ROI area. Different thresholds were tested by guessing, and a working scalar value which is used is 3.5, meaning that only ROIs with a variance of less than 70*70*3.5 are considered to be a plane.

The parameters defining a plane with respect to the Kinect are *Dx*, the change in depth between pixels from left to right, *Dy*, the depth change going up in a column, and the perpendicular distance or average value in the ROI. *Dx* is calculated as the average of the right half of the ROI minus the average of the left, divided by 35. This method may be thought of as pairing each pixel in the left half with a pixel 35 to the right, then dividing by 35 to find the average depth change over one pixel. This is not quite the same as the average depth change; average depth change would not be useful since middle columns would cancel out and any information they contain would be lost. A similar method is used to calculate *Dy*. The perpendicular distance was already calculated in determining the ROI variance, as described earlier.

2.3.2 Plane Determination and Seed Selection

Multiple planar ROIs will likely belong to the same plane, so ROIs must be compared in order to avoid defining the same plane more than once. This process consists of examining each ROI and checking to see if it is already in a defined plane. If it isn't, or if no planes have been defined yet, then a new plane is defined. Planes with multiple ROIs have their parameters based on the ROI with the lowest variation, referred to as the seed. FIGURE 7 shows an RGB image from the Kinect in which two seeds have been identified, as well as several non-planar ROIs. An ROI is considered to be part of an existing plane if its *Dx* and *Dy* values are within 5% of those belonging to the existing plane, and the depth offset is within 10.

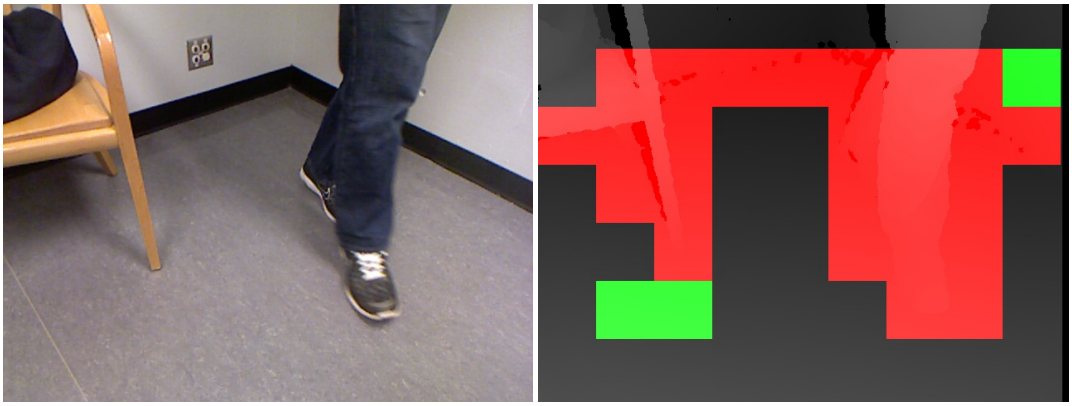

Figure 5: Seed Selection
An RGB image (left) and the corresponding depth image with some ROIs highlighted (right) are shown. Seed ROIs are highlighted in green. In this depth image, it has been determined that the floor consists of two slightly different planes, and a plane defining the wall has also been found. The red ROIs are those with higher variation than the threshold and are not considered as planes.

2.3.3 Angle Calculation

The algorithm can detect planes within the depth image and define them in the reference frame of the Kinect. In order to determine if these planes are close to being level (possible for the wheelchair to travel on), one must know the Kinect's orientation with respect to the horizon. The accelerometer is used to calculate $KinectTiltAngle$ and $KinectBalanceAngle$, which are the angles of the Kinect's x- and z-axes with respect to the horizon. These angles are most accurate when the wheelchair is stationary, with no accelerometer disturbance, but it is assumed that any disturbance due to the slow movement of the wheelchair (both acceleration

and noise) is negligible. For each plane, $PlaneAngleX$ and $PlaneAngleY$, the angles between the Kinect view normal and the horizontal and vertical components of the plane normal (with respect to the Kinect) are calculated.

$$KinectTiltAngle = \arctan(\frac{Dz}{Dy})$$

$$KinectBalanceAngle = \arctan(\frac{Dx}{Dy})$$

$$PlaneAngleY = \arctan(\frac{-Focal}{\frac{1090-Offset}{Dy}-y+240})$$

$$PlaneAngleX = \arctan(\frac{-Focal}{\frac{1090-Offset}{Dx}-x+320})$$

To determine if a plane is level, $PlaneAngleY + KinectTiltAngle$ must be 90°, plus or minus whatever deviation is allowed. We have allowed a deviation of 5°. This concept is illustrated in Figure 6. The relationship for a floor being level in the x-direction is $KinectBalanceAngle = PlaneAngleX$.

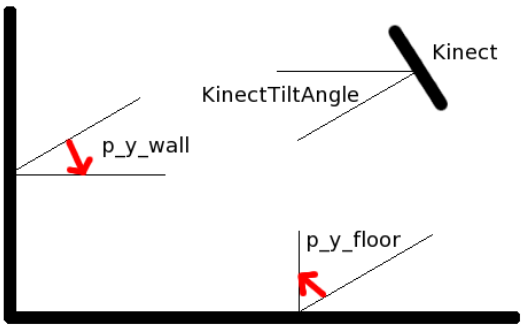

Figure 6: Plane Boundaries

In the above image, the vertical line represents a vertical wall, and the horizontal line represents a level floor. The Kinect is pointed towards the floor. In the case of the wall, $PlaneAngleY = -KinectTiltAngle$, so $PlaneAngleY + KinectTiltAngle$ is 0. For a level floor, $PlaneAngleY$ is $90° - KinectTiltAngle$, and $PlaneAngleY + KinectTiltAngle$ is exactly 90°.

2.3.4 Safety Probability

Once all the safe planes are known, each individual pixel is labelled with a probability of being safe. To explain how the probability is determined, consider a depth image with only one safe plane. Pixels within a distance of 10 parallax of this plane are 100% safe, and pixels outside of the plane by 30 or more are 0%

safe. Any depth in between corresponds to a linear relationship: as depth difference goes from 10 to 30, the probability goes from 100% to 0%. This relationship is shown in following figure.

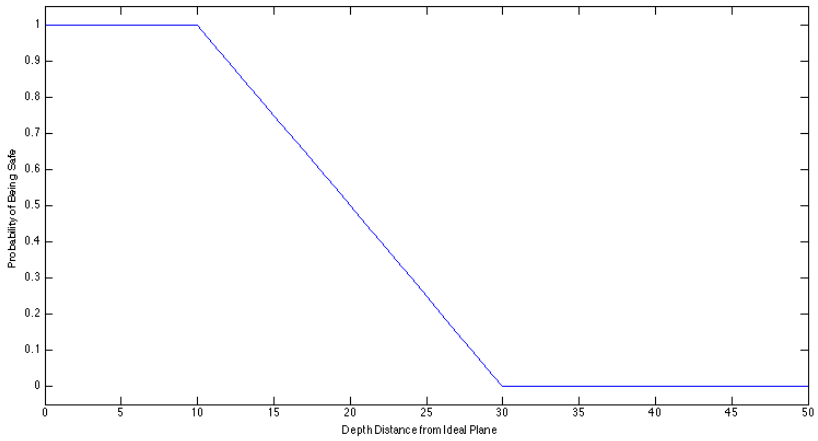

Figure 7: One Plane Safety Probability
Probability of a pixel being safe for a depth image containing a single plane.

In the case of there being more than one plane, the probability is estimated by the same distribution in terms of the closest plane. This is illustrated for two planes, in figure below.

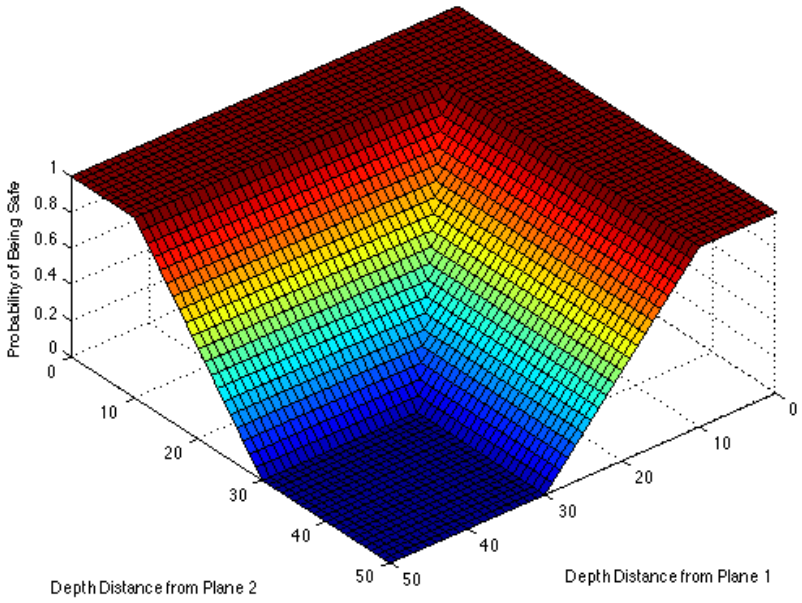

Figure 8: Two Planes Safety Probability
Probability of a pixel being safe for a depth image containing two planes.

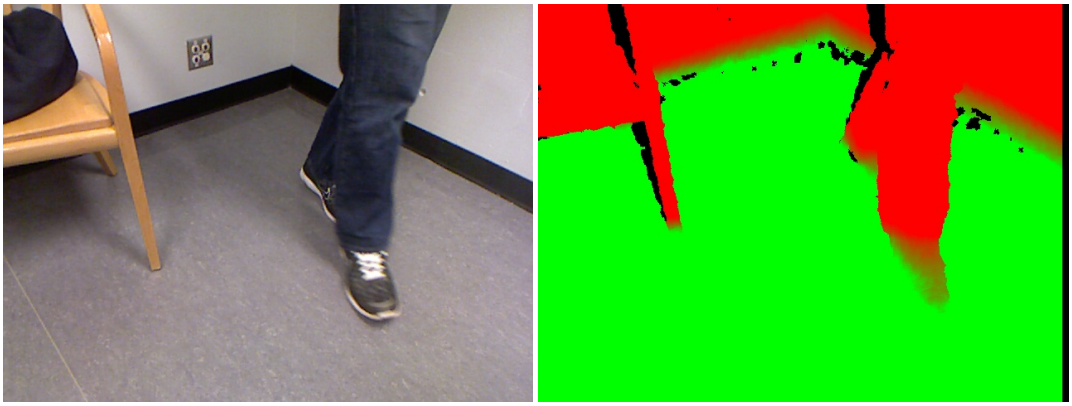

Figure 9: Safety Probability
A Kinect RGB image (left) is shown with the corresponding depth image (right) in which pixels have been coloured based on how safe they are. The colour goes from completely green to completely red as the probability of being safe goes from 100% to 0%. Pixels which are black represent unknown areas.

2.3.5 Overhead View Map

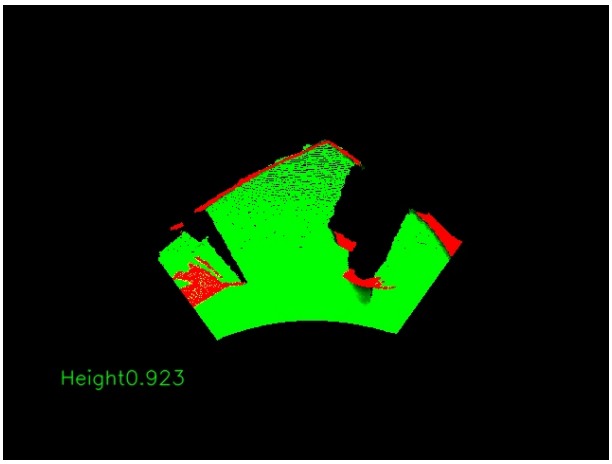

Figure 10: Overhead View Map

Based on the angle information of kinect and pixel parallax. The safety probability is mapped into the overhead view, as shown in the figure above. Again, green areas are safe, red areas are unsafe, and black areas are unknown.

2.4 Results

In order to quantify the results, tests were done to analyze certain scenarios. These scenarios include looking at a floor, looking at a wall, and looking at a wall-floor intersection. As expected, looking at the floor identifies the area is 100% safe; this is shown in Figure 11. Also as expected, looking at a wall is 0% safe, and looking at the wall-floor intersection results in identifying the wall portion as unsafe, and the floor portion as safe. This is shown in Figure 12.

As the sponsor required, the result of our program is summarized in a three channels floating point array. Which includes all the information about the overhead view map mentioned in section 2.3.5. The size of the array is 640×480, which corresponds to the pixel location on the depth map. The three channels store the probability score, the view angle and the radius distance. Other procedure parameters are also fully accessible. Please look at the source code for more information.

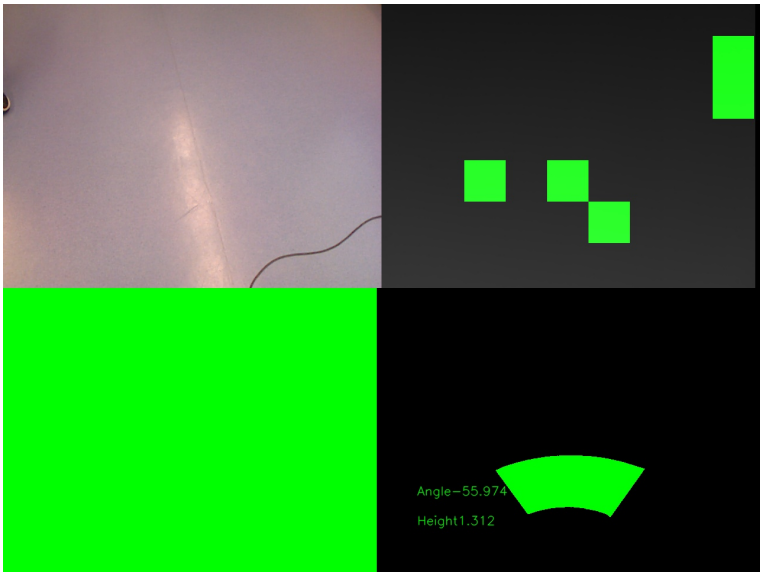

Figure 11: Floor Analysis
This shows an RGB image (top-left), seed locations (top-right), safe areas from depth camera view (bottom-left) and the overhead view (bottom-right). As expected, looking down at a floor with no obstacles identifies the entire area as 100% safe.

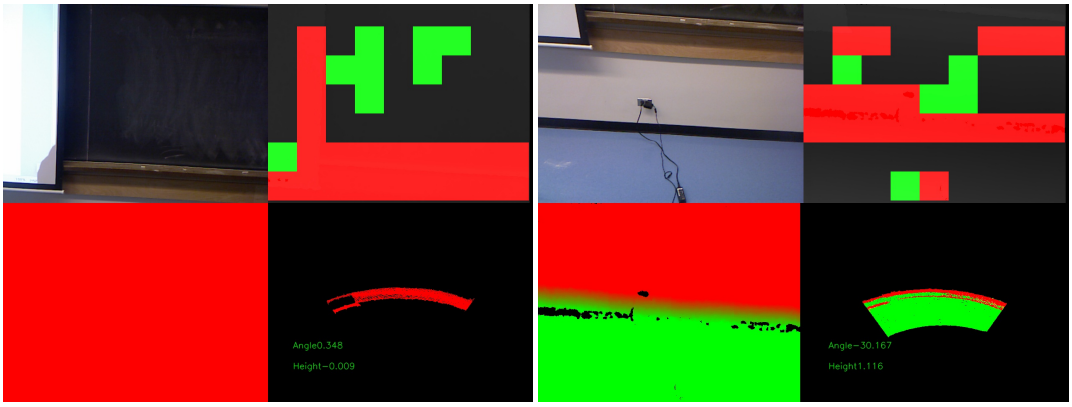

Figure 12: Wall and Corner Analysis

Similiar images to Figure 11 are shown here. If the Kinect looks straight at a wall (left), the area is 100% unsafe, as seen by the camera view being completely red. A picture containing the wall and floor is shown (right). In this case, roughly half the depth view is red with the other half green, as expected. Black areas are due to material which absorbs the IR, so no signal is returned.

3 Conclusions

The obstacles detection system is fast and light-weight, so it can perform real-time depth detection and analysis at 6.5 fps on a single thread 2.0 GHz CPU. The plane detection and classification algorithm identifies safe areas, and allows simultaneous detection of both positive and negative obstacles. This result has surpassed the initially intended goal of only performing negative obstacle detection.

4 Project Deliverables

4.1 List of Deliverables

The sponsor is expected to have the following:

#	Item	Description
1	Source code	Contains all the code created for the system developed
2	Readme file	Contains instruction for source code and developer's contact information

We will also return all the equipments provided by the sponsor.

4.2 Financial Summary

The following equipment was provided by the sponsor with no additional cost.

#	Description	Quantity	Vendor(s)	Cost	Purchased by:	To be funded by:
1	MS Kinect	1	Microsoft	$150 USD	Ian Mitchell	UBC CS Dept.
2	Li Battery	1	batteryspace	$130 USD	Ian Mitchell	UBC CS Dept.
3	Smart Charger	1	batteryspace	$30 USD	Ian Mitchell	UBC CS Dept.

4.3 Open Source

The software created will be released as open source under BSD License.

5 Recommendations

5.1 Further Algorithm Optimization

There are several ways to improve performance of the system. Due to the project time limit, we cannot implement them.

5.1.1 Multi-threading

Currently, our program is single-threading. In other words, the program handles image capture, algorithm computations and image mapping in series. By implementing multi-threading computation, these tasks can be processed simultaneously. The program can easily be modified for multi-threading as the three tasks above are already separate in the program.

5.1.2 GPGPU

As an image processing program, our program applies a function to each individual pixel. Conventional CPU will process the function for each pixel one at a time, while GPU is designed to process a large number of pixels at the same time. OpenCV provides functions based on general-purpose computing for graphics processing units (GPGPU).[3]

5.2 Algorithm Modification

Since the CanWheel project is still under development, our program is designed to provide the flexibility for further customization.

5.2.1 Fixed Parameters

Once the Kinect is mounted to the wheel chair, all the parameters based on the accelerometer reading only need to be computed once, at a time when the Kinect is stationary. This will provide more accurate information while increasing the processing speed.

5.2.2 Fuzzy Decision

The final output of this program is mapped to an overhead view map, producing a scaled image of the real environment. In order to generate this map, the actual position of each pixel is calculated. This is one of the heaviest computations in our program. If the mount angle and height are fixed, the relative distance of the object can easily be estimate based on its position in the image.

6 Appendices

6.1 Mathematical Proof for Plane Detection on Depth Image

Kinect's *DepthValue* outputs are given by:

$$DepthValue = 1090 - 8 \times parallax$$

and we know that *parallax* is calculated by

$$parallax = \frac{f \times B}{D}$$

At this point, it may be confusing since the Kinect's *DepthValue* is directly proportional to the parallax, while the parallax is inversely proportional to the distance of an object. Analysis based on directly plotting *DepthValue* outputs and producing a correct depth map seems illogical. We shall demonstrate that this mathematical relationship is valid. Let us consider the case when the Kinect is placed at a height H with *KinectTiltAngle* and *KinectBalanceAngle* both 0.

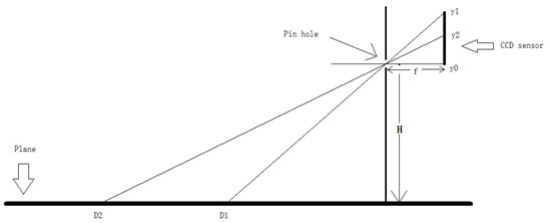

Figure 13: Side view of the Kinect mounted at a height H.

Plugging the *parallax* into the *DepthValue* equation we get:

$$DepthValue = 1090 - \frac{8 \times f \times B}{D} = 1090 - \frac{K1}{D}$$

Since the focal length f and baseline B are constant, we can replace the numerator with a constant $K1$. Let us go back and determine an expression for D. By similar triangles we have:

$$D = \frac{f \times H}{y - y0} = \frac{K2}{y - y0}$$

Since the Kinect is fixed at a height H, we replace the product of f and H with a constant $K2$. After plugging the expression for D back into the *DepthValue* equation we have:

$$D = 1090 - \frac{K1}{K2}y + \frac{K1}{K2}y0 = 1090 - K3 \times (y - y0)$$

where the constant $K3$ is the quotient of $K1$ and $K2$. From the equation derived, we can see that a plane's *DepthValue* detected is a linear function of the y-position of a pixel.

Now let's consider the case where a plane is at an angle to the vertical axis.

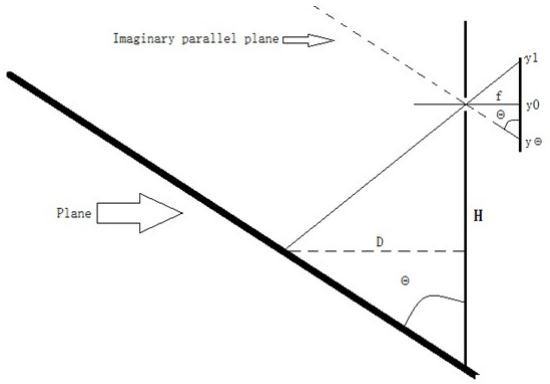

Figure 14: Side view of Kinect mount at a height H to a tilt plane.

By geometry, we have:

$$\frac{y1 - y\theta}{f} = \frac{H}{D}$$

Using

$$y\theta = y0 - cot(\theta) \times f$$

the expression for D becomes:

$$D = \frac{H \times f}{y1 - y\theta} = \frac{H \times f}{y1 - y0 + cot(\theta) \times f}$$

Finally, the expression for *DepthValue* becomes:

$$D = 1090 - K3 \times (y - y0 + cot(\theta) \times f)$$

As we can see, that $DepthValue$ for a plane at an angle θ has linear relationship with the y-position of a pixel.

In the previous cases, we only consider the center column. Now let's consider the relationship for any column. Consider a plane that is intersected with the focal plane and sensor plane., The intersection line is $A \times y + x = w0$. There are lines:

$$A \times y + x = wn$$

The lines are parallel to the intersection line of the measured plane and the sensor plane, as well as the focal plane. For each pixel on one line, the corresponding position on the measured plane also lines up and is parallel to the intersection line. Therefore, the depth values are the same.

The depth value of a point (xa, ya) on $A \times y + x = wn$ is the same as the depth value of the center column of this line $(x0,\ ya - (xa - x0)/A)$. This indicates that the depth change of a horizontal movement is equivalent to a scaled vertical movement. So a shift in the x direction also changes the depth linearly.

7 References

References

[1] Stereo engine technical description, October 2008.

[2] Willow Garage. Opencv 2.3.1, August 2011.

[3] Willow Garage. Opencv gpu, October 2011.

[4] Konolige. Kinect calibration/ technical, December 2011.

[5] OpenKinect. libfreenect, 2011.

[6] UBC. Canwheel, 2012.