Stereoscopic Imaging for Slow-Moving Autonomous Vehicle

Project Report

By:
Alex Norton

Advisor:
Dr. Huggins

July 24th, 2012

Abstract

This project utilizes two cameras and OpenCV (open source computer vision) to perform computer stereo vision for extracting 3D information from the images. This information can then be sent to an autonomous vehicle for navigation. The system operates using two modes: calibration mode and run mode. Calibration mode involves calculating the intrinsic and extrinsic parameters of each camera by using a chessboard with known geometry and easily detectable features. Run mode involves sending a signal to have the cameras acquire images, process the images using matrices computed via the calibration process, and then compute distances to objects in the field of view of the cameras.

Table of Contents

I. Introduction

The objective of Stereoscopic Imaging for Slow-Moving Autonomous Vehicle, SISAV, is to develop a system that can provide an accurate terrain map to be used for navigating an autonomous vehicle. The system uses two digital cameras and OpenCV to perform stereoscopic imaging to obtain distance information about objects in front of the cameras. The system has two modes of operation: calibrate and run. Calibration mode involves calculating the intrinsic and extrinsic parameters of each camera by using a chessboard with known geometry and easily detectable features. Run mode involves sending a signal to have the cameras acquire images, process the images using matrices computed via the calibration process, and then compute distances to objects in the field of view of the cameras. This information can be used by the control algorithm running an autonomous vehicle to determine the direction in which to move.

Disparity is a fundamental concept that underlies extracting 3D information from 2D projections onto camera imaging planes**.** The book *Learning OpenCV* has an excellent section on using pin hole cameras to use disparity measurements on pages 415-418 [1]. An ideal configuration is shown in Fig. 1 which is extracted from page 419 of *Learning OpenCV*. The ideal configuration is characterized by two identical pinhole cameras with coplanar image planes and parallel optic axis. The optical axis is the ray from the center of projection *O* through the principal point *c* and is also known as the principal ray. As shown in Fig. 1, the optical axes are a distance of T apart, which is the same as the distance between the cameras. The cameras are assumed to have equal focal lengths $f_l = f_r = f$. Another assumption made it that c_x^{left} and c_x^{right} have the same pixel coordinates in their respective left and right images after the cameras have been properly calibrated. The principal points c_x^{left} and c_x^{right} are where the principal rays from their respective cameras intersect the imaging plane of their respective camera. This intersection depends on the optical axis of the lens. The image plane is rarely aligned exactly with the lens and so the center of the imager is almost never exactly aligned with the principal point [1].

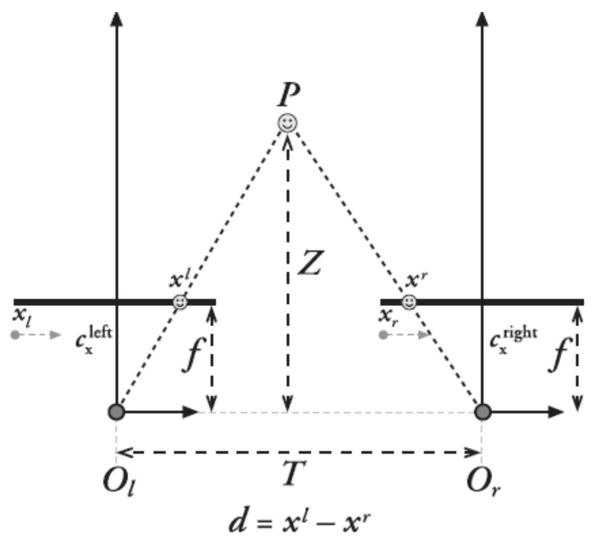


Figure 1: The above figure is figure 12-4 from *Learning OpenCV* [1]. With perfectly undistorted, aligned stereo rig and known correspondence, the depth Z can be found by similar triangles; the principal rays of the imagers begin at the centers of projection O_l and O_r and extend through the principal points of the two image planes at c_l and c_r [1, pp.416].

It is further assumed that the images of both cameras have been rectified, therefore making them row-aligned, and the pixels along a row of one camera's image are aligned with the corresponding row of pixels in the other camera's image. Such a camera arrangement is called frontal parallel, and is one of the results of properly calibrating the two cameras. Using cameras that are in a frontal parallel arrangement, it can be assumed that any point in the physical world in the left and right images, at point p_l in the left camera image and point p_r in the right camera image, will have the same vertical coordinates y_l and y_r, and will only differ in horizontal coordinates, x_l and x_r.

In this case, it can be shown that the depth to an object in the left and right images is inversely proportional to the disparity between the two images. Here, disparity is defined as $d = x_l - x_r$, where x_l and x_r are the horizontal coordinates of the points in the left and right images respectively. This situation is shown in Figure 1, where it can be seen that the depth *Z* can be easily derived by using similar triangles. Referring to the figure, this gives:

$$\frac{T-(x^l-x^r)}{Z-f}=\frac{T}{Z} \quad \Rightarrow \quad Z=\frac{fT}{x^l-x^r} \qquad \text{Eqn. 1}$$

II. System Description

The system consists of two digital cameras, a mount for the cameras, and a computer running OpenCV. The two cameras are attached to a stable platform that, in turn, will be attached to the vehicle. There are two modes of operation: calibration mode and run mode. During calibration mode, a calibration rig is used to obtain the extrinsic and intrinsic parameters of each camera which are used to correct for distortion in images captured during run mode [1, pp.378]. After the cameras are calibrated, the system enters run mode. Run mode involves sending a signal to have the cameras acquire images, process the images using matrices determined via the calibration process, and then compute distances to objects in the field of view of the cameras. This information can be used by the control algorithm running an autonomous vehicle to determine the direction in which to move.

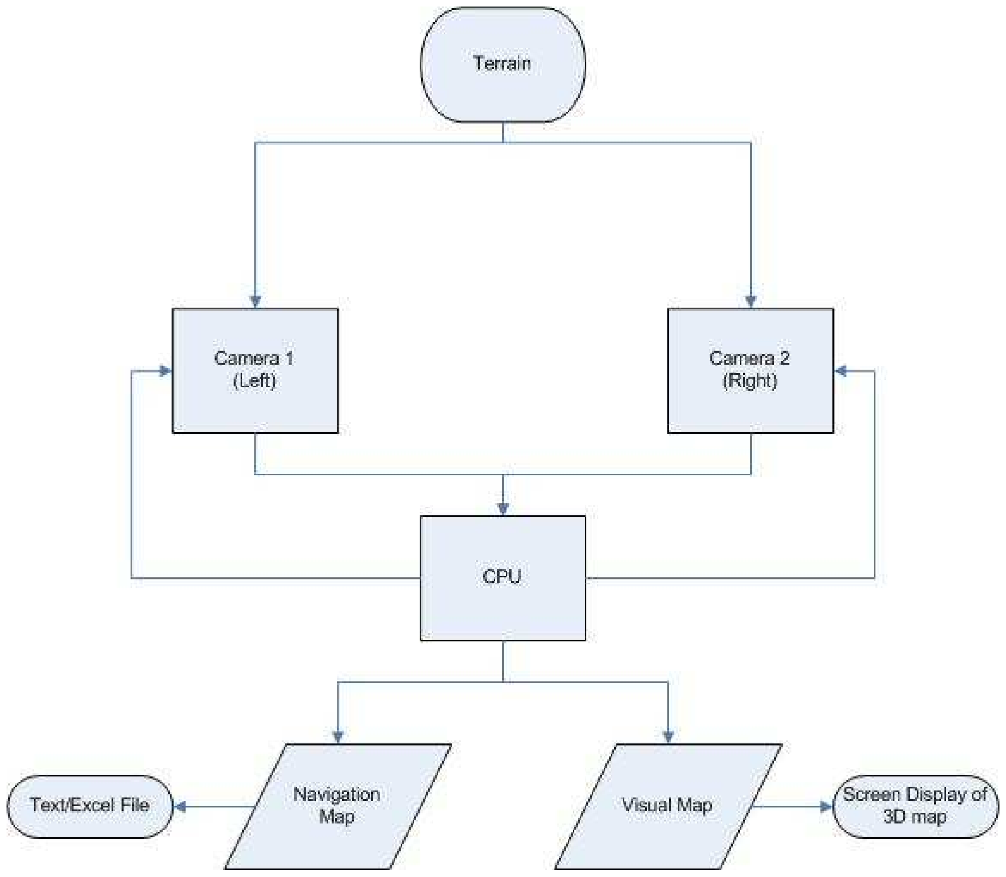


Figure 2: System Block Diagram

A. Subsystems

As shown in figure 2, the system is made up of 2 subsystems: the camera subsystem and the computer subsystem.

1. Camera Subsystem

The camera subsystem**,** shown in Fig. 3, consists of two digital cameras mounted on a stable platform. The cameras convert photons of light into binary data each time they receive signals to capture images. The data consists of 8-bit arrays, three from each camera, containing values from 0 to 255 for the RGB values of each pixel. The color information is then converted to grayscale using a built-in OpenCV function. A value of 0 corresponds to black and a value of 255 corresponds to white. These data are then sent to OpenCV to be processed further. The cameras function in an identical fashion in both the calibration mode and run mode.

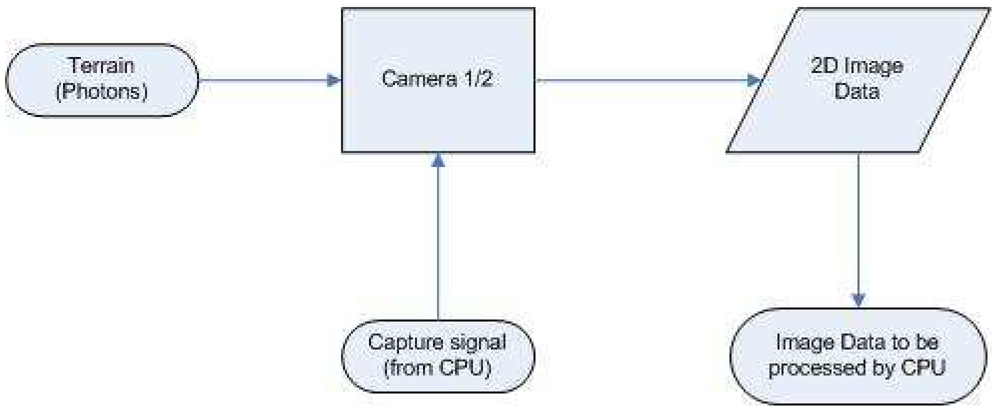


Figure 3: Camera Subsystem

2. Computer and Software Subsystems

The computer subsystem, shown in Fig. 4, runs the necessary software to capture and process the images from the cameras and generate the data for input to the navigation software of an autonomous vehicle. The computer vision software, running on the computer, simultaneously acquires images from the cameras which are then downloaded to the computer via a USB connection. The software then computes the correspondence between pixel groups in the set of images. Here, correspondence is the measure of the similarity between corresponding pixel groups in the set of images. The correspondence values are used to compute the disparity between the corresponding pixel groups, and finally computes distances based on the disparity map, in which the disparity is the offset in pixels between corresponding pixel groups in the left and right cameras. These distances can then be used by the vehicle control system for navigation

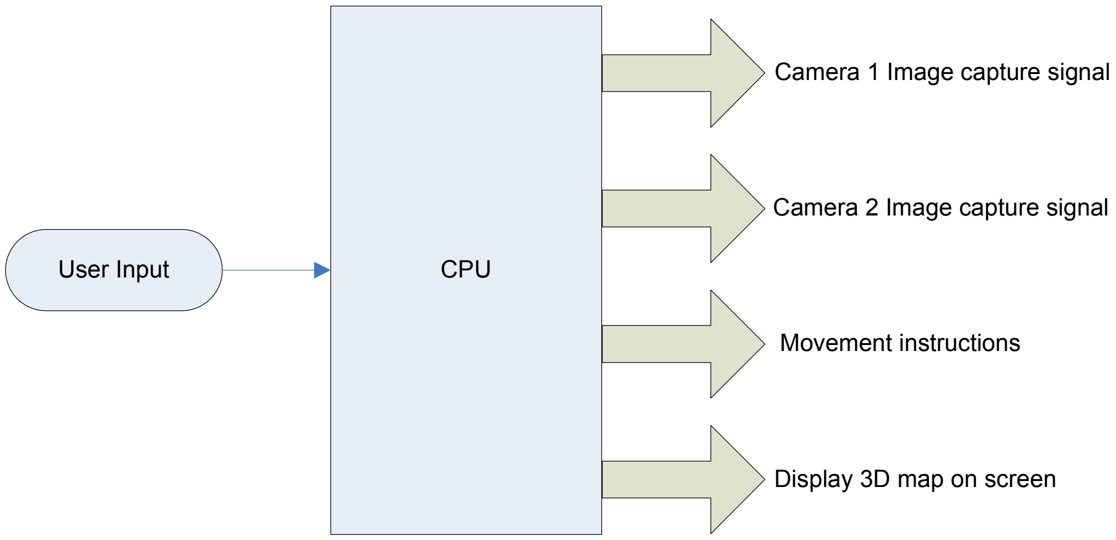


Figure 4: Computer Subsystem

B. Operational Modes
There are two modes of operation, calibration mode and run mode. These modes are described in more detail below.

1. Calibration Mode

Calibration mode is the initial mode of operation for the system. Its primary function is to correct for deviations of the actual camera system from an ideal pin hole camera system. The deviations are due to internal and external properties of the physical cameras. Once the system is powered on, the software waits for user input to enter calibration mode. In this mode, the software prompts a user to place a chessboard target in an arbitrary position and orientation(within limits) and then the system will acquire an image from each camera. This is repeated in additional positions and orientation as needed by the calibration software algorithm to be used. Once an appropriate number of images, normally 10, are acquired, the calibration determines the calibration matrices to be used to correct distortions in the images acquired in run mode. The flow chart for the Calibration mode software is shown in Fig. 5***.***

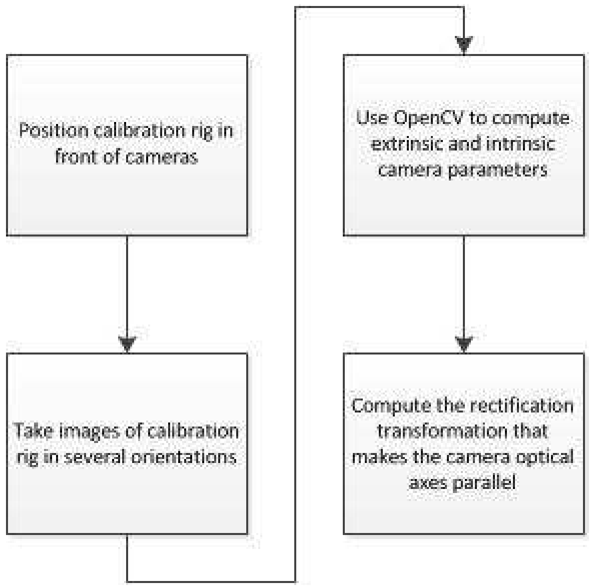

Position calibration rig in front of cameras
Take images of calibration rig in several orientations
Use OpenCV to compute extrinsic and intrinsic camera parameters
Compute the rectification transformation that makes the camera optical axes parallel

Figure 5: Calibration Mode Flow Chart

2. Run Mode

Run mode is entered upon successful calibration of the two cameras. In this mode, the user can set up the computer vision software to respond to commands from the user or from an automated prompt from the navigation control software. In both instances, the computer vision software acquires the images from the cameras. The software then uses OpenCV libraries to compute the correspondence between pixel groups in the set of images, uses the correspondence values to compute the disparity between the corresponding pixel groups, and finally computes distances based on the disparity map. These distances can then be used by the vehicle control system for navigation. Run mode is exited by closing the computer vision software. The flow chart for the run mode software is shown below in Fig. 6

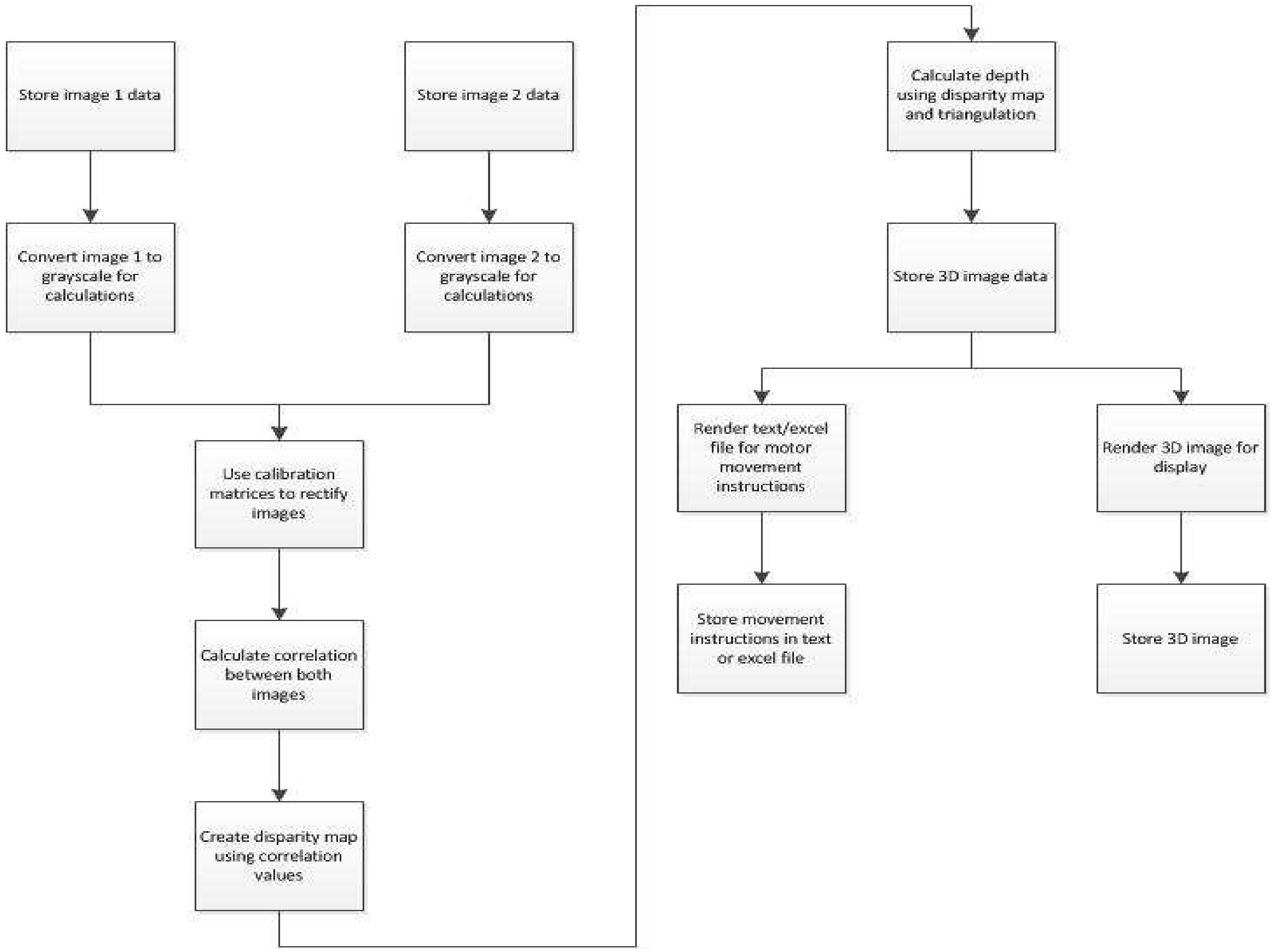

Store image 1 data
Store image 2 data
Convert image 1 to grayscale for calculations
Convert image 2 to grayscale for calculations
Use calibration matrices to rectify images
Calculate correlation between both images
Create disparity map using correlation values
Calculate depth using disparity map and triangulation
Store 3D image data
Render text/excel file for motor movement instructions
Render 3D image for display
Store movement instructions in text or excel file
Store 3D image

Figure 6: Run Mode Flow Chart

III. Subsystem Requirements

Each subsystem, as well as each mode of operation, will be expected to perform according to certain specifications. These specifications are described in the following sections.

A. Camera Subsystem Requirements

- The cameras shall have a field of view of at least 45 degrees
- The cameras shall have a depth of view from 1 meter to 10 meters
- The cameras shall be calibrated and focused after startup to ensure accurate image data
- In order to maximize speed for the system, the cameras shall output images at a resolution of 320x240
- The cameras shall be secured to a mount to ensure they do not move out of alignment during operation
- The cameras shall not have any face-tracking software built into them
- The cameras shall interface with the computer via USB connections
- The cameras shall be compatible with Windows 7.

B. Computer Subsystem Requirements

- The computer shall have at least two USB ports to interface with the two cameras
- In order to run Microsoft Visual Studio 2010, the computer shall have2 GB of RAM, 5 GB of memory, 32-bit or 64-bit Windows, a 1.6 GHz or faster processor, a 5400 RPM or higher hard disk drive, a DirectX 9 capable video card running at 1024 x 768 or higher-resolution display, and a DVD-ROM drive [2]

C. Computer Vision *S*oftware Requirements

- The software shall use OpenCV and be split into two modes of operation; calibration mode and run mode.

D. Calibration Mode Requirements

- Once the system is powered on, it shall wait for user input to enter calibration mode
- The calibration mode shall be compatible with a chessboard type calibration rig
- The calibration mode software shall compensate for internal and external distortion effects.
- The calibration shall compensate for internal and external distortions of the camera system so distance information calculated in run mode is accurate to 5% in specified operating range.

- The calibration mode software shall transmit appropriate parameters to the run mode software.

E. Run Mode Requirements

- Run mode shall be entered upon successful calibration of the two cameras
- The cameras shall receive signals from the navigation control software of an autonomous vehicle or a user to acquire a pair of images for processing.
- The software shall complete all image processing within 5 seconds of receiving the images from the cameras
- The run mode software shall determine distances to objects with 5% accuracy
- The run mode software shall complete an image acquisition and computation cycle in less than 5 seconds.
- The run mode software shall present distance information as a text file for use by navigation control software.
- The run mode software shall be capable of displaying the detected edges, disparity map, or distance information on the monitor based on user input.

IV. Results

This year, significant progress was made on the project in two areas. The OpenCV libraries and functions were successfully used to create and run code for both modes of operation for the project. This allowed significantly more progress to be made in terms of calibrating the cameras than in all other prior stereo vision projects attempted at Bradley. In addition to these major accomplishments, some progress was made in implementing the “run mode” portion of the project in that maps were generated in real time that had some correlation to disparity.

A. Calibration Mode Results

After the program sets up the webcams to capture sets of images, the user has to calibrate the camera by analyzing multiple images of a chessboard of known dimensions. The user must input the number of internal corners and dimension of the squares of the chessboard. The user also can choose the number of images for processing, which has 10 for this project. Though the chessboard can be placed in many orientations at various distances from the cameras, there are limits. For example, the images shown in Fig. 7 are what are displayed on the screen if the software fails to detect all the corners on the chessboard.

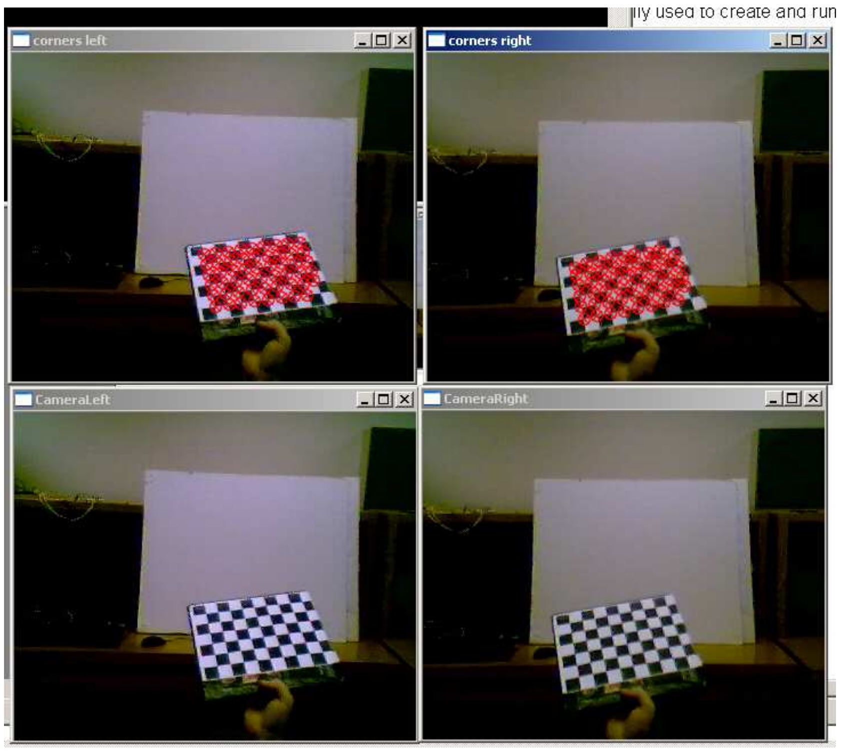
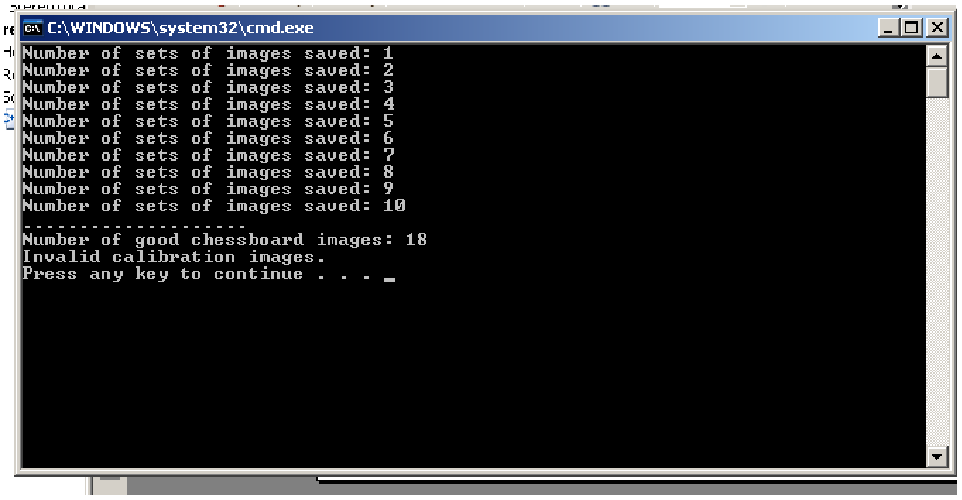


Figure 7: Display when chessboard corners are not found

Saving at least one set of chessboard images such as these will result in the command window in fig. 8 being displayed after saving the required number of sets of images.



```
Number of sets of images saved: 1
Number of sets of images saved: 2
Number of sets of images saved: 3
Number of sets of images saved: 4
Number of sets of images saved: 5
Number of sets of images saved: 6
Number of sets of images saved: 7
Number of sets of images saved: 8
Number of sets of images saved: 9
Number of sets of images saved: 10
....................
Number of good chessboard images: 18
Invalid calibration images.
Press any key to continue . . . _
```

Figure 8: Command window display when chessboard corners are not found

On the other hand, when the software is able to find all the chessboard corners in a pair of chessboard images, colored lines are displayed as shown in Fig. 9, indicating valid images for calibration.

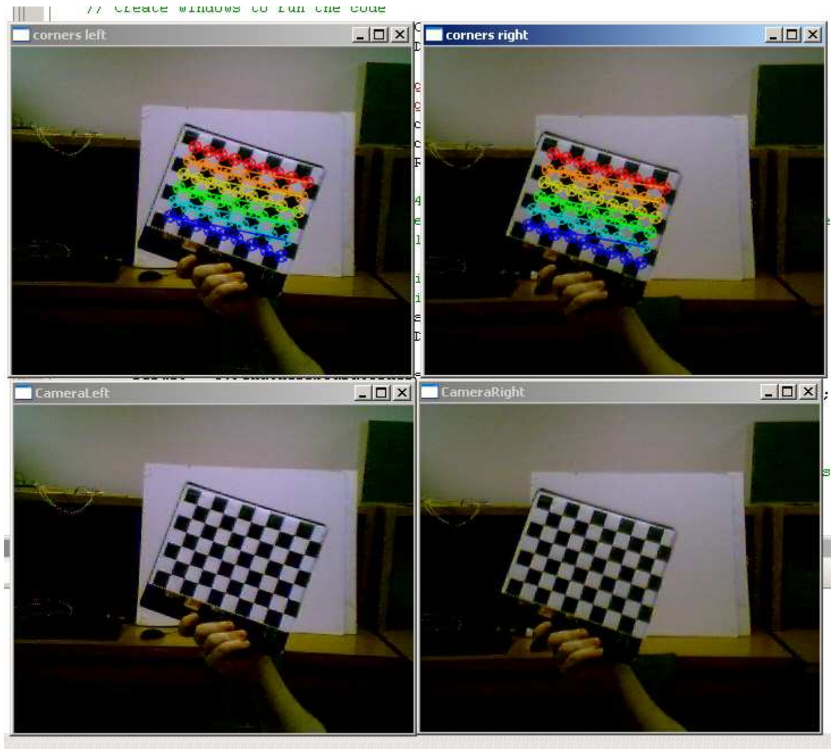
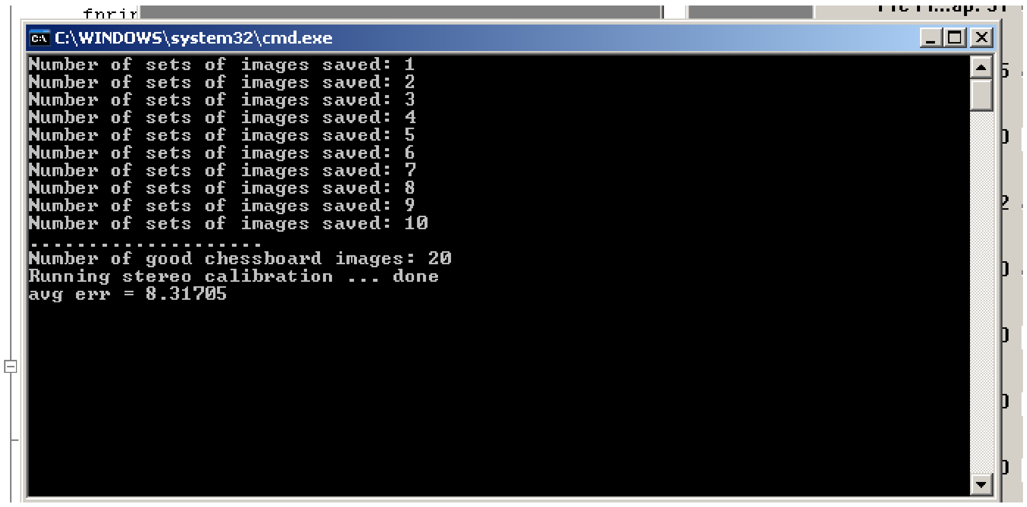


Figure 9: Display when chessboard corners are found

Once the required number of valid chessboard image pairs are saved, the command window in fig. 10 will be displayed.



```
Number of sets of images saved: 1
Number of sets of images saved: 2
Number of sets of images saved: 3
Number of sets of images saved: 4
Number of sets of images saved: 5
Number of sets of images saved: 6
Number of sets of images saved: 7
Number of sets of images saved: 8
Number of sets of images saved: 9
Number of sets of images saved: 10
....................
Number of good chessboard images: 20
Running stereo calibration ... done
avg err = 8.31705
```

Figure 10: Command window when chessboard corners are found in all images

The Software then proceeds to run the calibration mode software, which is listed in appendix C. Once the calibration is finished, the code will display any pair of rectified chessboard images as requested by the user. However, due to errors present in the system, the sets of images are not always rectified properly. This is shown in Fig. 11.

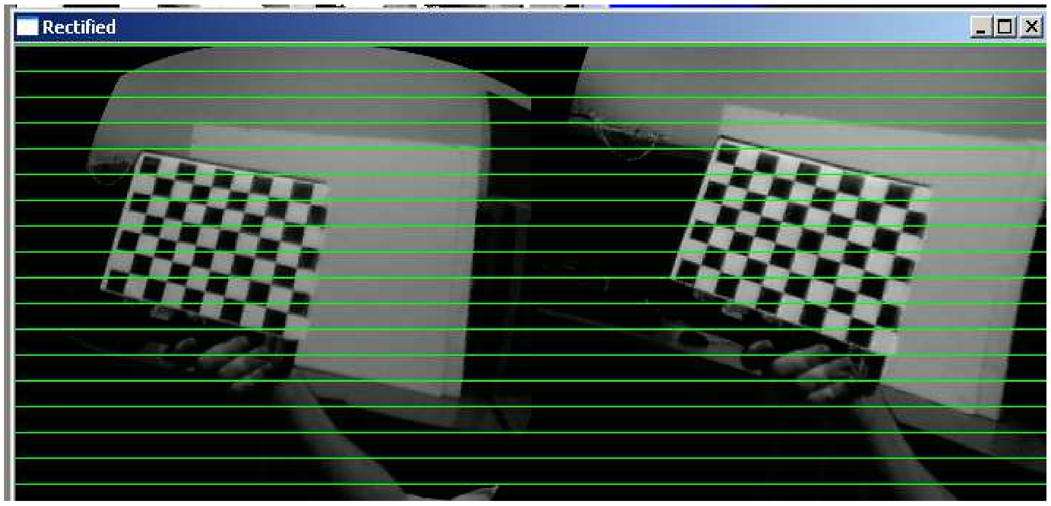


Figure 11: Output set of chessboard images from calibration mode

B. Run Mode Results

After the cameras have been calibrated, the program enters run mode, where the user can capture a pair of images by pressing the “enter” key twice. Once a set of images is captured, the software will use the calibration mode results to rectify the set of images so they appear as though captured by identical pinhole cameras. However, as stated before, due to errors present in the system, the sets of images are not always rectified properly, and possible reasons for this are given in Section C: Possible Errors on page 18. Below in fig. 12 is a set of images of me seen by the cameras.

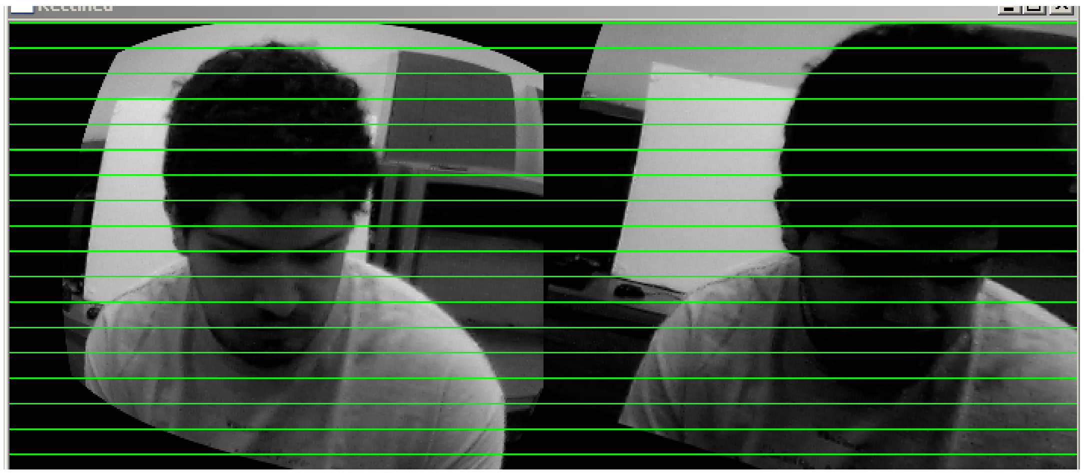
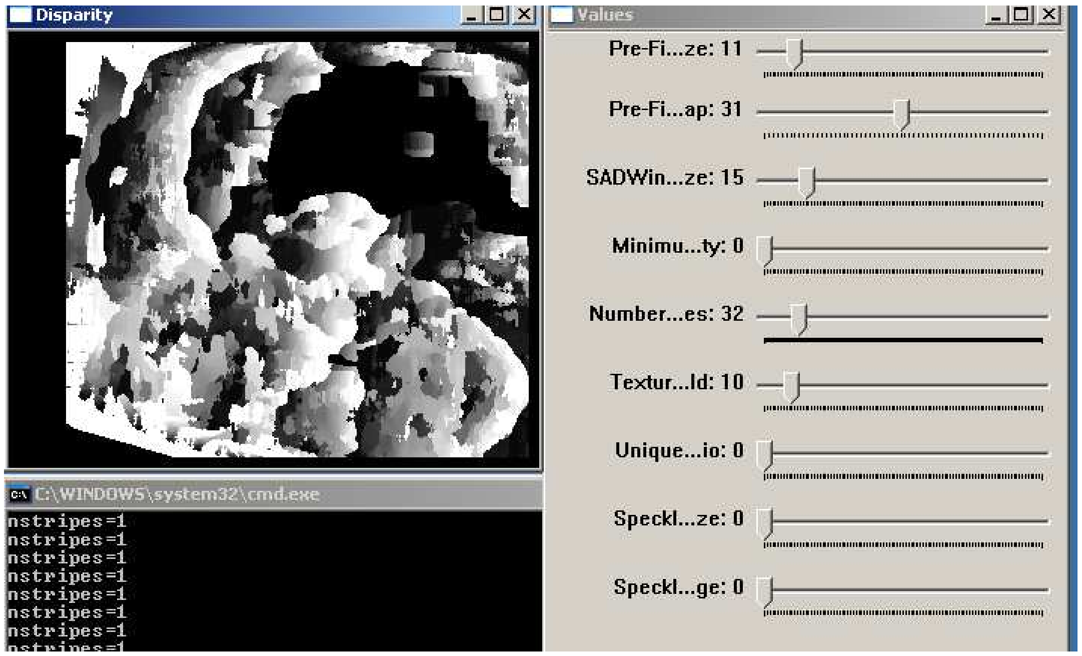


Figure 12: Output set of images obtained in run mode

Once the set of images has been rectified, the software uses the “cvFindStereoCorrespondenceBM” function to find the corresponding pixel groups and then compute the disparity between them. The functions uses a block matching algorithm [1, pp.444]. The disparity between corresponding pixel groups are stored in a matrix. This matrix is then displayed to show the disparity map obtained from the set of images, which is shown in Fig. 13 along with a window that is used to adjust the values that change the way the disparity map is computed. The parameters are: **preFilterSize**, the nxn size of the filters used to normalize the input images; **preFilterCap**, the number of those filters; **minDisparity**, the minimum disparity, usually 0; **SADWindowSize**, the linear size of the blocks compared by the algorithm, larger block size implies smoother, though less accurate disparity map while smaller block size gives more detailed disparity map, but there is higher chance for algorithm to find a wrong correspondence; **numberOfDisparities**, difference between the maximum disparity and minimum disparity; **textureThreshold**, sets the minimum threshold, under which areas with no texture are ignored; **uniquenessRatio**, which is used to filter out pixels if there are other close matches; **speckleWindowSize**, which is the maximum area of speckles to remove; and **speckleRange**, which is the acceptable range of disparity variation in each connected component internal data.

The values the bars are set to are the values that produce the best disparity map that could be determined, and were obtained through trial and error by changing the values on the slidebars and noting what changes were seen.



Pre-Fi...ze: 11
Pre-Fi...ap: 31
SADWin...ze: 15
Minimu...ty: 0
Number...es: 32
Textur...ld: 10
Unique...io: 0
Speckl...ze: 0
Speckl...ge: 0

```
nstripes=1
nstripes=1
nstripes=1
nstripes=1
nstripes=1
nstripes=1
nstripes=1
```

Figure 13: Disparity map computed from a set of images obtained in run mode

Once the disparity map is obtained, the distance to each pixel in the image can be computed by using the Eqn 2 [1, pp 417]

$$z = (B*F)/D \qquad \text{Eqn. 2}$$

This equation is essentially the same as equation 1, with the only differences being that the variables f and T are replaced by F and B, and $x^l - x^r$ is replaced by *D*. Here, *z* is the distance to a given pixel in meters, *B* is the distance between the two cameras in meters, *F* is the focal length of the cameras (in pixels), and *D* is the disparity, the horizontal difference in pixels between the corresponding pixel groups in left and right images. However, the disparity map obtained from sets of images shown in Fig. 13 is not the correct disparity map, and therefore any distance calculations performed will also be incorrect. If the map was showed the correct disparity, the young man visible in fig. 12 would also be easily visible in fig. 12, and the region would be lighter. Other terrain configurations were also investigated with similar errors. A correct disparity map is shown in Fig. 14 [3, pp. 4].

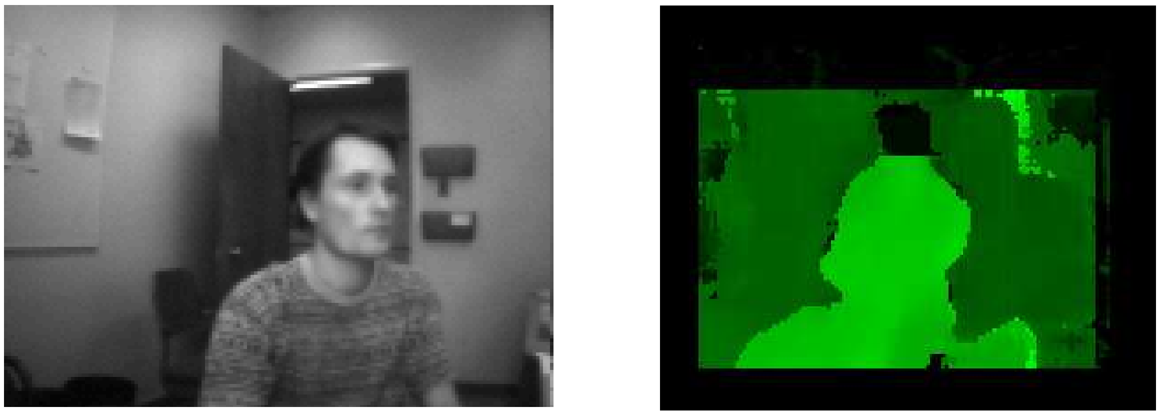


Figure 14: One images from a set of stereo images (left), the disparity map obtained from that set of images (right) [3, pp. 4]

Clearly, the person in the image on the left is easily visible in the image on the right, and is represented by higher values of disparity due to being closer to the cameras than the background behind him. Such results were what were desired for this project, but errors present made the system unable to achieve those results.

C. Possible Errors

Since OpenCV has been successfully used for stereoscopic imaging, it is clear that there must be some errors present in the methods or codes that were used in this project. One possible source of error is that the calibration results are incorrect. Theoretically, the calibration is supposed to convert the actual images into images that perfectly oriented pinhole cameras would have captured. One characteristic of the rectified images is that corresponding pixels in each set of images should lie on parallel horizontal lines. This makes the correspondence determination of pixel groups to be efficient. However, this was not the case for most calibration attempts. Points in one image do not line up with the corresponding points in the other image on the same line, and instead also have a vertical offset as well. The vertical offset in the rectified images could be due to subtle errors in the code implemented or parameters used by the function that were not set correctly. In any case, the disparity map function assumes that the images input have been rectified so that points in one image line up with the corresponding points in the other image. Failure to rectify the images will result in a wrong disparity map due to errors in the correspondence determination.

Another possible error is that the cameras could have internal flaws that cannot be corrected with sufficient accuracy by the calibration process. This could be a wide range of flaws, such as a flaw in the lenses that the calibration software fails to take into account, or possible a flaw in the image sensor within the webcams. Due to the age of the cameras, it is very possible that the internal electronics of the cameras are not performing as well as they should be, which could be causing images taken by the cameras to contain errors that cannot be corrected for. It is possible that the cameras being used simply cannot be calibrated.

D. Suggestions for Future Work

Although the project was not completely successful this year, significant progress was made, which means there is still some work that should be done in the future. In particular, there are two main areas that the next group should focus on. First, they should investigate the mathematics underlying the OpenCV functions. OpenCV is a very complex collection of many different image processing functions, with many of those functions using multiple parameters and having underlying mathematics that is also very complex. Therefore, in order to make better use of the functions within OpenCV's libraries, it is necessary to obtain a better understanding of the math that makes the functions work. Doing so will allow them to write better, more efficient code with those functions, and may also reveal what errors were made in this project,

The other area the next group should focus on is developing methods to find and correct for errors that occur as a result of incorrect calibrations and/or correspondence computations. This is something that would have been attempted for this project; however, there was not enough people working on the project, and also not enough time to accomplish it. As a result, the code used has no way to actively detect for and correct errors that result from incorrect calibrations or correspondence computations. If a future group if able to implement this into their system, it is very likely that they will be successful in completing the goals of the project, therefore error detection and correction should be one of the main focuses in the future.

V. Equipment List

- Two Logitech Quickcam Express webcams
- Dell Optiplex 755 computer
- Microsoft Visual Studio 2008
    - OpenCV 2.3

VI. Patents and Standards

Patents
Although there are many patents related to stereoscopic imaging and autonomous navigation, these are the ones most related to our project.

Table 2: Related Patents

Patent Number	**Brief Description**
6728582	System and method for determining the position of an object in three dimensions using a machine vision system with two cameras
6137893	Machine vision calibration targets and methods of determining their location and orientation in an image
7680323	Method and apparatus for three-dimensional object segmentation
5383013	Stereoscopic computer vision system
6392688	High accuracy stereo vision camera system
6807295	Stereoscopic imaging apparatus and method
6661449	Object detection device for autonomous vehicle

Standards
Applicable standards for our project are those related to the JPEG and PNG image formats, USB 2.0 and 3.0, and OpenCV versions 2.1 and 2.3.

The JPEG standard can be viewed at http://www.stanford.edu/class/ee398a/handouts/papers/Wallace%20-%20JPEG%20-%201992.pdf
The PNG standard can be viewed at http://www.libpng.org/pub/png/spec/iso/index-object.html#1Scope.
The USB 2.0 and 3.0 standards can be viewed at http://www.usb.org/developers/docs/.
Documentation for OpenCV 2.1 can be found at http://opencv.willowgarage.com/documentation/cpp/index.html
Documentation for OpenCV 2.3 can be found at http://opencv.itseez.com/.

References Cited

[1] Gary Bradski and Adrian Kaehler. “Learning OpenCV”: Internet: http://www.cse.iitk.ac.in/users/vision/dipakmj/papers/OReilly%20Learning%20OpenCV.pdf, 2008 [Sept. 20, 2011]

[2] Microsoft. “Visual Studio 2010 System Requirements”: Internet: http://www.microsoft.com/visualstudio/en-us/products/2010-editions/professional/overview, 2011 [Nov 28, 2011]

[3] Kurt Konolige. “Small Vision Systems: Hardware and Implementation”: Internet: https://willowgarage.com/sites/default/files/ISRR%201997%20-%20Small%20vision%20systems.pdf, 1997 [May 10, 20012]

References for Additional Information

Jean-Yves Bouguet. “Camera Calibration Toolbox for Matlab”: Internet: http://www.vision.caltech.edu/bouguetj/calib_doc/htmls/example.html, 2010 [Nov 13, 2011]

Martin Peris. “OpenCV: Stereo Camera Calibration”: Internet: http://blog.martinperis.com/2011/01/opencv-stereo-camera-calibration.html, 2011 [Oct 5, 2011]

Digital-Tutors. “Stereo 3D Disparity Maps”: Internet: http://www.digitaltutors.com/dtlabs/?p=645, 2010 [Nov 13, 2011]

Appendix A
OpenCV Code for Setting Up the Webcams

```
            //-------------------------------------------------------------------//
                        //Library declarations and webcams setup
             //-------------------------------------------------------------------//
#include "cv.h"
#include "cxmisc.h"
#include "highgui.h"
#include "cvaux.h"
#include <vector>
#include <string>
#include <algorithm>
#include <stdio.h>
#include <ctype.h>
using namespace std;

int main(void)
{
   CvCapture* capture = cvCaptureFromCAM( 0 );
   CvCapture* capture2 = cvCaptureFromCAM( 1 );
   int displayCorners = 0; //if 1, displays each chessboard image with corners found in them
   int showUndistorted = 0; //if 1, diplays each pair of chessboard images, rectified
   bool isVerticalStereo = false;//Cameras are horizontally aligned
   int numChessBoards = 10;  //number of chessboard images to be taken
   int nx = 9;  //number of chessboard corner along X axis
   int ny = 6;  //number of chessboard corner along Y axis
   int c = 0;
   int i, j, lr, nframes, n = nx*ny, N = 0;
   int Exit = 0;
   int focal = 500;
   double base = 0.0925;
   vector<CvPoint2D32f> temp(n);
   vector<uchar> active[2];
   int count = 0, result=0;
   const char* imageList = "list_10.txt"; //Filename can be changed depending on numChessboards
   char loc2[10];
   char loc3[10] = ".jpeg"; //filetype extension
   if ( !capture ) {
     fprintf( stderr, "ERROR: capture is NULL \n" );
     getchar();
     return -1;
   }
    if ( !capture2 ) {
     fprintf( stderr, "ERROR: capture2 is NULL \n" );
     getchar();
     return -1;
   }
   // Create windows to run the code
   cvNamedWindow( "CameraLeft", CV_WINDOW_AUTOSIZE );
   cvNamedWindow( "CameraRight", CV_WINDOW_AUTOSIZE );
   while ( c < numChessBoards ) {
            char locR1[30] = "imagelist/imageR";
            char locL1[30] = "imagelist/imageL";
            IplImage* imgR = cvQueryFrame( capture );
```

```
        IplImage* imgL = cvQueryFrame( capture2 );
        cvShowImage( "CameraRight", imgR );
        cvShowImage( "CameraLeft", imgL );
        //The next 10 lines were added 4-12-2012 to make calibration easier to do
        //They draw the chessboard corners on the images currently being taken by the
        //cameras. This allows you to know you'll have a good set of chessboard images
        //Before you actually save them
        IplImage* timgR = imgR; //temp image for right camera
        IplImage* timgL = imgL; //temp image for left camera
        result = cvFindChessboardCorners( timgR, cvSize(nx, ny),
        &temp[0], &count, CV_CALIB_CB_ADAPTIVE_THRESH | CV_CALIB_CB_NORMALIZE_IMAGE);
        cvDrawChessboardCorners( timgR, cvSize(nx, ny), &temp[0],count, result );
        result = cvFindChessboardCorners( timgL, cvSize(nx, ny),
        &temp[0], &count, CV_CALIB_CB_ADAPTIVE_THRESH | CV_CALIB_CB_NORMALIZE_IMAGE);
        cvDrawChessboardCorners( timgL, cvSize(nx, ny), &temp[0], count, result );
        cvShowImage( "corners right", timgR );
        cvShowImage( "corners left", timgL );
         //When the "enter" key is pressed, save the current set of chessboard images
        if ( cvWaitKey(10) == 13 ){
              c++;
              _itoa(c, loc2, 10);
              strcat(locR1, loc2);
              strcat(locR1, loc3);
              strcat(locL1, loc2);
              strcat(locL1, loc3);
              IplImage* imgR = cvQueryFrame( capture );
              IplImage* imgL = cvQueryFrame( capture2 );
              cvSaveImage(locR1, imgR); //saves right camera image to imageRc.jpeg
              cvSaveImage(locL1, imgL); //saves left camera image to imageLc.jpeg
              printf("Number of sets of images saved: %d", c);
              printf("\n");
        }
        if ( cvWaitKey(10) == 27){
              Exit = 1; //sets a flag to skip over the code
              break;
        }
     }
```

Appendix B
OpenCV Code for Calibration Mode

Note: The following code is still a part of the same main() function created in the code in appendix B. This code, and the code in the following appendix, are all a part of the same file, and also the same main() function.

```
//---------------------------------------------------------------------------------------------//
                                    // Run the calibration code
//---------------------------------------------------------------------------------------------//
   if(Exit == 0){
      int GoodChessBoardImages = 0;
      const int maxScale = 1;
      const float squareSize = 1.0;
      FILE* f = fopen(imageList, "rt");
      vector<string> imageNames[2];
      vector<CvPoint3D32f> objectPoints;
      vector<CvPoint2D32f> points[2];
      vector<int> npoints;
      CvSize imageSize = {0,0};
      // ARRAY AND VECTOR STORAGE:
      double M1[3][3], M2[3][3], D1[5], D2[5];
      double R[3][3], T[3], E[3][3], F[3][3];
      double Q[4][4];
      CvMat _M1 = cvMat(3, 3, CV_64F, M1 );
      CvMat _M2 = cvMat(3, 3, CV_64F, M2 );
      CvMat _D1 = cvMat(1, 5, CV_64F, D1 );
      CvMat _D2 = cvMat(1, 5, CV_64F, D2 );
      CvMat _R = cvMat(3, 3, CV_64F, R );
      CvMat _T = cvMat(3, 1, CV_64F, T );
      CvMat _E = cvMat(3, 3, CV_64F, E );
      CvMat _F = cvMat(3, 3, CV_64F, F );
      CvMat _Q = cvMat(4, 4, CV_64F, Q);
      CvMat part;
      if( displayCorners )
      cvNamedWindow( "corners", 1 );
      // READ IN THE LIST OF CHESSBOARDS:
      if( !f )
      {
      fprintf(stderr, "can not open file %s\n", imageList );
      return 0;
      }
      for(i=0;;i++)
      {
      char buf[1024];
      lr = i % 2;
      vector<CvPoint2D32f>& pts = points[lr];
      if( !fgets( buf, sizeof(buf)-3, f ))
      break;
      size_t len = strlen(buf);
      while( len > 0 && isspace(buf[len-1]))
      buf[--len] = '\0';
      if( buf[0] == '#')
      continue;
      IplImage* img = cvLoadImage( buf, 0 );
```

```
if( !img )
break;
imageSize = cvGetSize(img);
imageNames[lr].push_back(buf);
//FIND CHESSBOARDS AND CORNERS THEREIN:
for( int s = 1; s <= maxScale; s++ )
{
IplImage* timg = img;
if( s > 1 )
{
timg = cvCreateImage(cvSize(img->width*s,img->height*s),
img->depth, img->nChannels );
cvResize( img, timg, CV_INTER_CUBIC );
}
result = cvFindChessboardCorners( timg, cvSize(nx, ny),
&temp[0], &count,
CV_CALIB_CB_ADAPTIVE_THRESH |
CV_CALIB_CB_NORMALIZE_IMAGE);
if( timg != img )
cvReleaseImage( &timg );
if( result || s == maxScale )
for( j = 0; j < count; j++ )
{
temp[j].x /= s;
temp[j].y /= s;
}
if( result )
break;
}
if( displayCorners )
{
printf("%s\n", buf);
IplImage* cimg = cvCreateImage( imageSize, 8, 3 );
cvCvtColor( img, cimg, CV_GRAY2BGR );
cvDrawChessboardCorners( cimg, cvSize(nx, ny), &temp[0],
count, result );
cvShowImage( "corners", cimg );
cvReleaseImage( &cimg );
if( cvWaitKey(0) == 27 ) //Allow ESC to quit
exit(-1);
}
else
putchar('.');
N = pts.size();
pts.resize(N + n, cvPoint2D32f(0,0));
active[lr].push_back((uchar)result);
//assert( result != 0 );
if( result )
{
GoodChessBoardImages++;
//Calibration will suffer without subpixel interpolation
cvFindCornerSubPix( img, &temp[0], count,
cvSize(11, 11), cvSize(-1,-1),
cvTermCriteria(CV_TERMCRIT_ITER+CV_TERMCRIT_EPS,
30, 0.01) );
copy( temp.begin(), temp.end(), pts.begin() + N );
}
```

```
cvReleaseImage( &img );
}
fclose(f);
printf("\n");
// HARVEST CHESSBOARD 3D OBJECT POINT LIST:
nframes = active[0].size();//Number of good chessboads found
objectPoints.resize(nframes*n);
for( i = 0; i < ny; i++ )
for( j = 0; j < nx; j++ )
objectPoints[i*nx + j] =
cvPoint3D32f(i*squareSize, j*squareSize, 0);
for( i = 1; i < nframes; i++ )
copy( objectPoints.begin(), objectPoints.begin() + n,
objectPoints.begin() + i*n );
npoints.resize(nframes,n);
N = nframes*n;
CvMat _objectPoints = cvMat(1, N, CV_32FC3, &objectPoints[0] );
CvMat _imagePoints1 = cvMat(1, N, CV_32FC2, &points[0][0] );
CvMat _imagePoints2 = cvMat(1, N, CV_32FC2, &points[1][0] );
CvMat _npoints = cvMat(1, npoints.size(), CV_32S, &npoints[0] );
cvSetIdentity(&_M1);
cvSetIdentity(&_M2);
cvZero(&_D1);
cvZero(&_D2);
printf("Number of good chessboard images: %d", GoodChessBoardImages);
//Since the calibrate function does not work if it can't find all chessboard corners
//in each set of chessboard images, the following if statement was added. It ends after
//run mode ends.
if(GoodChessBoardImages == (2*numChessBoards)){
      // CALIBRATE THE STEREO CAMERAS
      printf("\nRunning stereo calibration ...");
      fflush(stdout);
      cvStereoCalibrate( &_objectPoints, &_imagePoints1,
      &_imagePoints2, &_npoints,
      &_M1, &_D1, &_M2, &_D2,
      imageSize, &_R, &_T, &_E, &_F,
      cvTermCriteria(CV_TERMCRIT_ITER+
      CV_TERMCRIT_EPS, 100, 1e-5),
      CV_CALIB_FIX_ASPECT_RATIO +
      CV_CALIB_ZERO_TANGENT_DIST +
      CV_CALIB_SAME_FOCAL_LENGTH );
      printf(" done\n");
      // Save the matrices to text files, added 2-28-2012
      cvSave("Rotation.txt",&_R);
      cvSave("Translation.txt",&_T);
      cvSave("Fundamental.txt",&_F);
      cvSave("Essential.txt",&_E);
      cvSave("Camera1.txt",&_M1);
      cvSave("Camera2.txt",&_M2);
      cvSave("Dist1.txt",&_D1);
      cvSave("Dist2.txt",&_D2);

      // CALIBRATION QUALITY CHECK
      // because the output fundamental matrix implicitly
      // includes all the output information,
      // we can check the quality of calibration using the
      // epipolar geometry constraint: m2^t*F*m1=0
```

```
vector<CvPoint3D32f> lines[2];
points[0].resize(N);
points[1].resize(N);
_imagePoints1 = cvMat(1, N, CV_32FC2, &points[0][0] );
_imagePoints2 = cvMat(1, N, CV_32FC2, &points[1][0] );
lines[0].resize(N);
lines[1].resize(N);
CvMat _L1 = cvMat(1, N, CV_32FC3, &lines[0][0]);
CvMat _L2 = cvMat(1, N, CV_32FC3, &lines[1][0]);
//Always work in undistorted space
cvUndistortPoints( &_imagePoints1, &_imagePoints1,
&_M1, &_D1, &_R, &_M1 );
cvUndistortPoints( &_imagePoints2, &_imagePoints2,
&_M2, &_D2, &_R, &_M2 );
cvComputeCorrespondEpilines( &_imagePoints1, 1, &_F, &_L1 );
cvComputeCorrespondEpilines( &_imagePoints2, 2, &_F, &_L2 );
double avgErr = 0;
for( i = 0; i < N; i++ )
{
double err = fabs(points[0][i].x*lines[1][i].x +
points[0][i].y*lines[1][i].y + lines[1][i].z)
+ fabs(points[1][i].x*lines[0][i].x +
points[1][i].y*lines[0][i].y + lines[0][i].z);
avgErr += err;
}
printf( "avg err = %g\n", avgErr/(nframes*n) );
//COMPUTE AND DISPLAY RECTIFICATION
CvMat* mx1 = cvCreateMat( imageSize.height,
imageSize.width, CV_32F );
CvMat* my1 = cvCreateMat( imageSize.height,
imageSize.width, CV_32F );
CvMat* mx2 = cvCreateMat( imageSize.height,
imageSize.width, CV_32F );
CvMat* my2 = cvCreateMat( imageSize.height,
                                     imageSize.width, CV_32F );
CvMat* img1r = cvCreateMat( imageSize.height,
imageSize.width, CV_8U );
CvMat* img2r = cvCreateMat( imageSize.height,
imageSize.width, CV_8U );
CvMat* disp = cvCreateMat( imageSize.height,
imageSize.width, CV_16S );
CvMat* vdisp = cvCreateMat( imageSize.height,
imageSize.width, CV_8U );
CvMat* depth = cvCreateMat( imageSize.height,
                                       imageSize.width, CV_32FC3 );
CvMat* pair;
double R1[3][3], R2[3][3], P1[3][4], P2[3][4];
CvMat _R1 = cvMat(3, 3, CV_64F, R1);
CvMat _R2 = cvMat(3, 3, CV_64F, R2);
// BOUGUET'S METHOD
CvMat _P1 = cvMat(3, 4, CV_64F, P1);
CvMat _P2 = cvMat(3, 4, CV_64F, P2);
cvStereoRectify( &_M1, &_M2, &_D1, &_D2, imageSize,
&_R, &_T,
&_R1, &_R2, &_P1, &_P2, &_Q, 0/*CV_CALIB_ZERO_DISPARITY*/ );
isVerticalStereo = fabs(P2[1][3]) > fabs(P2[0][3]);
//Precompute maps for cvRemap()
```

```
cvInitUndistortRectifyMap(&_M1,&_D1,&_R1,&_P1,mx1,my1);
cvInitUndistortRectifyMap(&_M2,&_D2,&_R2,&_P2,mx2,my2);
cvNamedWindow( "Rectified", 1 );
cvNamedWindow( "Disparity" );
cvNamedWindow( "Depth" );
// RECTIFY THE IMAGES AND FIND DISPARITY MAPS
pair = cvCreateMat( imageSize.height, imageSize.width*2,
CV_8UC3 );
//Setup for finding stereo correspondences
//The BMState variable consists of multiple values which affect the disparity map
//outputby the block matching stereo correspondance function
CvStereoBMState *BMState = cvCreateStereoBMState(/*CV_STEREO_BM_BASIC*/);
assert(BMState != 0);
BMState->preFilterSize=11;
BMState->preFilterCap=31;
BMState->minDisparity=0;
BMState->numberOfDisparities=32;
BMState->textureThreshold=10;
BMState->uniquenessRatio=0;
BMState->speckleWindowSize=0;
BMState->speckleRange=0;
if(showUndistorted){
for( i = 0; i < nframes; i++ ){
IplImage* img1=cvLoadImage(imageNames[0][i].c_str(),0);
IplImage* img2=cvLoadImage(imageNames[1][i].c_str(),0);
if( img1 && img2 ){
cvRemap( img1, img1r, mx1, my1 );
cvRemap( img2, img2r, mx2, my2 );
cvFindStereoCorrespondenceBM( img1r, img2r, disp, BMState); //calculate disparities
//cvNormalize( disp, vdisp, 0, 256, CV_MINMAX ); //normalize the disparities
cvShowImage( "Disparity", disp );
cvReprojectImageTo3D(disp, depth, &_Q);

cvShowImage( "Depth", depth );
cvGetCols( pair, &part, 0, imageSize.width );
cvCvtColor( img1r, &part, CV_GRAY2BGR );
cvGetCols( pair, &part, imageSize.width,
imageSize.width*2 );
cvCvtColor( img2r, &part, CV_GRAY2BGR );
for( j = 0; j < imageSize.height; j += 16 )
cvLine( pair, cvPoint(0,j),
cvPoint(imageSize.width*2,j),
CV_RGB(0,255,0));
cvShowImage( "Rectified", pair );
if(cvWaitKey() == 27)
break;
}
cvReleaseImage( &img1 );
cvReleaseImage( &img2 );
}
}
cvDestroyWindow( "corners" );
cvDestroyWindow( "corners right" );
      cvDestroyWindow( "corners left" );
```

Appendix C
OpenCV Code for Run Mode

```
//--------------------------------------------------------------------------------------------//
                                    // Run the run mode code
//--------------------------------------------------------------------------------------------//
      //Creates a window to change the BMState values while running the program
      cvNamedWindow( "Values", CV_GUI_EXPANDED );

      cvCreateTrackbar("Pre-Filter Size", "Values", &BMState->preFilterSize, 100, NULL);
      cvCreateTrackbar("Pre-Filter Cap", "Values", &BMState->preFilterCap, 63, NULL);
      cvCreateTrackbar("SADWindow Size", "Values", &BMState->SADWindowSize, 100, NULL);
      cvCreateTrackbar("Minimum Disparity", "Values", &BMState->minDisparity, 100, NULL);
      cvCreateTrackbar("Number of Disparities", "Values", &BMState->numberOfDisparities, 256,
      NULL);
      cvCreateTrackbar("Texture Threshold", "Values", &BMState->textureThreshold, 100, NULL);
      cvCreateTrackbar("Uniqueness Ratio", "Values", &BMState->uniquenessRatio, 100, NULL);
      cvCreateTrackbar("Speckle Window Size", "Values", &BMState->speckleWindowSize, 100,
      NULL);
      cvCreateTrackbar("Speckle Range", "Values", &BMState->speckleRange, 100, NULL);
      /*A few notes about the constraints of the values:
      preFilterSize must be odd and from 5 to 255,
      preFilterCap must 1 to 63, it can be odd or even,
      SADWindowSize must be odd and from 5 to 255,
      numberOfDisparities must be positive and evenly divisable by 16 (16, 32, 64, 128, etc)*/
      while(1){
            if ( cvWaitKey() == 13 ){ //captures images when "enter" key is pressed
                  IplImage* imgR = cvQueryFrame( capture );
                  IplImage* imgL = cvQueryFrame( capture2 );
                  //IplImage* imgR = cvLoadImage("TsukubaLeft.jpg"); //test image
                  //IplImage* imgL = cvLoadImage("TsukubaRight.jpg"); //test image
                  imageSize = cvGetSize(imgR);
                  //create matrices for rectified images
                  CvMat* imgRrect = cvCreateMat( imageSize.height,
                  imageSize.width, CV_8U );
                  CvMat* imgLrect = cvCreateMat( imageSize.height,
                  imageSize.width, CV_8U );
                  CvMat* disp = cvCreateMat( imageSize.height,
                  imageSize.width, CV_16S );
                  CvMat* vdisp = cvCreateMat( imageSize.height,
                  imageSize.width, CV_8U );
                  CvMat* depth = cvCreateMat( imageSize.height,
                  imageSize.width, CV_32FC3 );
                  //convert unrectified images to grayscale
                  IplImage* imgRgray = cvCreateImage( imageSize, 8, 1 );
                  IplImage* imgLgray = cvCreateImage( imageSize, 8, 1 );
                  cvCvtColor( imgR, imgRgray, CV_RGB2GRAY );
                  cvCvtColor( imgL, imgLgray, CV_RGB2GRAY );
                  cvShowImage( "CameraLeft", imgLgray );
                  cvShowImage( "CameraRight", imgRgray );
                  //use matrices from calibration to rectify the images
                  cvRemap( imgRgray, imgRrect, mx1, my1 );
                  cvRemap( imgLgray, imgLrect, mx2, my2 );
                  //compute disparity of rectified images
                  cvFindStereoCorrespondenceBM(imgLrect, imgRrect, disp, BMState);
                  cvNormalize( disp, vdisp, 0, 256, CV_MINMAX );
```

```
               cvShowImage( "Disparity", vdisp );
               //compute depth map from disparity map
               cvReprojectImageTo3D(vdisp, depth, &_Q);
               cvShowImage( "Depth", depth );
               //steps needed to combine the 2 rectified images into a single image
               cvGetCols( pair, &part, 0, imageSize.width );
               cvCvtColor( imgRrect, &part, CV_GRAY2BGR );
               cvGetCols( pair, &part, imageSize.width, imageSize.width*2 );
               cvCvtColor( imgLrect, &part, CV_GRAY2BGR );
               for( j = 0; j < imageSize.height; j += 16 )
               cvLine( pair, cvPoint(0,j),
               cvPoint(imageSize.width*2,j),
               CV_RGB(0,255,0));
               cvShowImage( "Rectified", pair );
         }
         if(cvWaitKey() == 27) //exit when ESC key is pressed
               break;
         }
      cvReleaseStereoBMState(&BMState);
      cvReleaseMat( &mx1 );
      cvReleaseMat( &my1 );
      cvReleaseMat( &mx2 );
      cvReleaseMat( &my2 );
      cvReleaseMat( &img1r );
      cvReleaseMat( &img2r );
      cvReleaseMat( &disp );
}
else
         printf("\nInvalid calibration images.\n");
   }
   else
         printf("\nProgram has been exited by user. Please run again.\n");
// Release the capture device housekeeping and close all windows
cvReleaseCapture( &capture );
cvReleaseCapture( &capture2 );
cvDestroyAllWindows();
return 0;
```