ROBOTIC VISION COMMAND RECOGNITION SYSTEM (RCS) REPORT 2

Group 6
Michael Cohen
Rebekkah Zimmermann

May 11, 2006

Abstract

The Robotic Vision Command Recognition System (RCS) controls the motion of a robot with finger gestures. The commander simply moves his or her index finer toward the camera to direct the robot to move forward or away from the camera to direct the robot to move back. Two Logitech Fusion Cameras, positioned adjacent to each other, stream left and right video feeds of the commander. Intel Corporations OpenCV toolkit allows for C++ coding and therefore fast real-time performance. The theoretical backbone of the RCS is *Calibrated Stereo Reconstruction*, a process in which 3D information is calculated from two 2D images of an object. In addition, Principle Component analysis (PCA) is utilized to recognize the commander's index finger.

1 Goals

1.1 OpenCV Setup

In order to optimize speed and efficiency of all algorithms, Intel's OpenCV must be the *RCS* image processing platform. OpenCV, a collection of useful C++ classes and functions, processes images significantly faster than Matlab. Unfortunately, OpenCV takes many hours to setup and has a steeper learning curve than Matlab. Becoming familiar with OpenCV programming style is the first essential step of developing any real-time image processing system.

1.2 Two-Camera Vision System

Configuring OpenCV for two USB cameras streaming video simultaneously in a state-machine loop provides two images available for processing in each frame. The left and right video sampling must be synchronized automatically by DirectX filters, and the *stereocallback()* function must provide the programmer with pointers to the two frames.

When two cameras are adjacent to each other, but separated by some large distance, the disparity results in scene dissimilarities between the stereo images. In other words, the left camera will not view some objects that the right camera views, and vice versa. To solve this problem, the left camera must be inverted, thereby minimizing the disparity. The Logitech Fusion camera driver can automatically invert the video about the horizontal and/or vertical axes.

1.3 Finger Recognition

Recognizing the tip and base of the commander's index finger is necessary to reconstruct a 3D vector pointing in the direction of the finger. Detected features in the left frame and the right frame that are both recognized as fingers can be set as a point correspondence.

1.4 Automatic Point Correspondence

Tracking features from the left frame and finding the corresponding feature in the right frame is a difficult, but necessary task in order to reconstruct a 3D model of a given scene. The point correspondence algorithm described below is robust, accurate, and fully automated.

1.5 Calibration

Calibration is also a necessary task in order to reconstruct a 3D model of a given scene using *stereo reconstruction*. Calibration results in the position and orientation of the cameras with respect to the world coordinate frame. The calibration algorithm described below utilizes edge detection to minimize error associated with selecting pixel locations with a mouse.

1.6 Stereo Reconstruction

Using the intrinsic and extrinsic camera parameters from calibration, and the point correspondences, stereo reconstruction generates 3D coordinates from two 2D images. Once a stereo reconstruction algorithm works well for a simple scene, then the algorithm will be used to reconstruct a portion of a human hand.

1.7 System Integration

The Commander's finger-tip will move in the direction that the robot should move. Stereo reconstruction of the index region of the Commander's hand will provide a vector pointing in the direction of the index finger. The resulting vector will be interpreted to move the robot in a given direction.

1.8 Robot and Command Center

A simple lego robot is needed with a wide suspension to reduce error associated with independent motors driving each wheel. A command center, containing both mounted cameras, a black background, and space for the commanders, must be constructed.

2 Algorithms

2.1 Good Features To Track

2.1.1 Theory

The OpenCV function cvGoodFeaturesToTrack(), named after the fundamental computer vision paper, detects features using either Harris Corner detection or Eigenvalue analysis. [1] The algorithms utilized by this function include cvHarris(), cvCornerMinEigenVal, and others.

There are several ways to find features. The first is through a spatial intensity profile. By searching for a high standard deviation in the spatial intensity profile one can distinguish features based on texturedness and cornerness. Secondly, the features can be better selected by also calculating the presence of zero crossings of the Laplacian of the image's intensity and by finding the locations of the images corners. The downside to these algorithms is that feature tracking accuracy can be diminished if the image has a depth discontinuity or if the boundary of a reflection highlight is found on a glossy surface. If a feature that was previously a good feature becomes occluded, or dissimilarity is noted, then the feature is removed from the list of features and is either just neglected or is replaced by another feature.

In order to track a feature, two models of image motion are required. Given two sequential image frames I and J, then J(A$\mathbf{x}$+$\mathbf{d}$) = I($\mathbf{x}$),where $\mathbf{d}$ is the displacement matrix, and A = I + D, where I = identity matrix, and D is the affine deformation matrix.

The translational model of image motion tracks the feature across the two images when inner-frame changes are small. In the case of pure translation, the deformation matrix D = 0 such that J($\mathbf{x}$+$\mathbf{d}$) = I($\mathbf{x}$). Then the task of feature tracking is reduced to finding the displacement matrix $\mathbf{d}$.

If inner-frame changes are not small, the affine model of image motion should be used to reduce. The affine model distinguishes the similarities and dissimilarities between features in a current image frame and the next frame for a large number of cumulative changes.

Good features can be found using eigenvalue analysis. Two small eigenvalues mean a constant intensity profile within the window, one large and one small eigenvalue mean an unidirectional texture pattern, and two large eigenvalues mean a reliable pattern to track has been found. Each one of these features is tracked by placing a window around the feature in the first image and then scanning for it in the next image. Once the eigenvalues change indicating the intensity profile is bad, the feature is then dropped.

2.1.2 Application

This project utilizes cvGoodFeaturesToTrack() to find features in the left and right camera frames, and to detect edges for calibration. The following function definition shows how cvGoodFeaturesToTrack() is customized for different applications:

void cvGoodFeaturesToTrack(const IplImage image, IplImage* eig_image, IplImage* temp_image, CvPoint2D32f* corners, int* corner_count, double quality_level, double min_distance, const IplImage* mask=NULL, int block_size=3, int use_harris=0, double k=0.04)*

Where

- **image** is the source single-channel (grayscale) image of type IplImage.
- **eig_image** is a temporary image used by cvCornerMinEigenVal().
- **temp_image** is a temporary image used by cvCornerMinEigenVal().
- **corners** is an output parameter: a list of corners detected in image.
- **corner_count** is an output parameter: the number of detected corners.
- **quality_level** specifies the minimal accepted quality of image corners.
- **min_distance** specifies the minimum possible distance between corners.
- **mask** is NULL by default and unimportant in this particular application.
- **block_size** is used by cvCornerMinEigenVal or cvCornerHarris.
- **use_harris** selects cvCornerHarris(), if nonzero, to be used instead of cvCornerMinEigenVal().
- **k** is a parameter of used by cvCornerHarris, and is used only if use_harris $\neq 0$.

To find point correspondences between the left and right camera frames, the cvGoodFeaturesToTrack() is set to detect edges using cvCornerMinEigenVal(). Thus, specular highlights, depth discontinuities, and other bad features will not be selected, providing better features that can be tracked over many frames. During camera calibration, however, the calibration cube provides very distinct edges and cvCornerHarris() yields better results.

To summarize, cvGoodFeaturesToTrack() utilizes either cvCornerMinEigenVal() or cvCornerHarris() to detect edges. If cvCornerMinEigenVal() is used, fewer features will be returned because the Eigenvalue analysis adds an additional criterion for feature selection. When using cvCornerHarris(), any major intensity change will return a feature.

2.2 Principle Component Analysis (PCA)

2.2.1 Theory

Principle Component Analysis can be divided into an initial stage, where training images of an object of interest are processed, and a real-time stage, where new images are matched with training images. [3] The initial stage is comprised of the following steps:

1. Acquire initial training set of images.
2. Calculate the k most important eigenvectors of the training set. These eigenvectors become the basis vectors defining the training image space.

3. Project each training image onto the basis vectors to determine the training coefficients. Each image can be represented by the basis vectors and a set of training coefficients.

Once the training coefficients and basis vectors are known, for each new image:

1. Project the image onto the basis vectors to determine the test coefficients.
2. Determine the euclidean distance between the test coefficients and the training coefficients of each training image. The closest distance corresponds to the recognized object.

In order to calculate the basis vectors corresponding to the set of $MNxN$ training images $\{T_1...T_M\}$, each image in the set can be reshaped into a column vector, stacking each row on top of the next. This new set of M N^2x1 column vectors $\{\tau_1...\tau_M\}$ can be thought of has points in M^2 dimensional space. The mean training image is defined as

$$\Psi_i = \frac{1}{M}\sum_{i=0}^{M}\tau_i \tag{1}$$

Then each training image differs from its mean by $\Phi_i = \tau_i - \Psi_i$. The covariance matrix relating the set $\{\Phi_1...\Phi_M\}$ is defined

$$\begin{aligned}\mathbf{C} &= \frac{1}{M}\sum_{i=0}^{M}\Phi_i\Phi_i^T \\ &= AA^T\end{aligned} \tag{2}$$

where $A = \{\Phi_1...\Phi_M\}$. Rather than finding the eigenvectors of the N^2xN^2 covariance matrix $\mathbf{C} = AA^T$, the eigenvectors and eigenvalues of the MxM matrix $L = A^TA$ can be calculated with minimal computational complexity, assuming that $M < N$. The eigenvectors are sorted based on corresponding eigenvalue. Thus, the first several eigenvectors contain the most important information.

2.2.2 Application

The training images must be obtained using Matlab, then reshaped into column vectors. A image grabbing tool is written to capture an image of the commander, then select the center-point of a $30x30$ feature window. The program then automatically crops this region of the captured image for use as a training image. For example, the original image may be of the mid-section of the commander's body, including the arm and hand. A point on the tip of the index finger would be selected, thereby generating a $30x30$ image centered around the finger tip. This process is repeated taking pictures of the finger tip from different angles.

The resulting set of M column vectors $\{\tau_1...\tau_M\}$ undergoes eigenvector analysis, as described in 2.2.1. Matt's Matlab Tutorial provides code to construct the set $\{\tau_1...\tau_M\}$ and perform the eigenvector analysis. The resulting eigenvector and training coefficients are ported into the RCS project.

2.3 Sum of Squared Differences (SSD)

2.3.1 Theory

SSD analysis is used to minimize the error between two functions (or images). Given a left image $I_L(x_1, y_1)$ and a right image $I_R(x_2, y_2)$, two rectangular regions are defined by $R_L = [x_{1a}...x_{1b}, y_{1a}...y_{1b}]$ and $R_R = [x_{2a}...x_{2b}, y_{2a}...y_{2b}]$. Then the error between the image intensities within the regions R_L and R_R is defined as $(I_L(R_L) - I_R(R_R)$. Then the SSD can be defined as [2]

$$\sum \left[I_L(R_L) - I_R(R_R)\right]^2 \tag{3}$$

2.3.2 Application

Stereo color images I_L and I_R are converted to grayscale (single-channel) images G_L and G_R. cvGoodFeaturesToTrack() returns a list of corners in each grayscale image. For each corner in G_L:

- A rectangular window is created around the given corner.
- G_R is searched for any corners that could be matches.
- For each potential match, a rectangular window is created and SSD analysis is performed on the two windows according to (3), returning a result.
- The minimum result corresponds to the best match.

This process is repeated until all corners in G_L have a match in G_R.

This algorithm is an improvement upon the previous SSD algorithm, in which corner detection was performed on G_L and a large region of G_R was searched for each corner. The benefit of performing corner detection on both frames is that only several corners in G_R must be searched, rather than a several-thousand-pixel region.

2.4 Camera Calibration

2.4.1 Theory

Given a 3D point in the world coordinate frame $[X^w\ Y^w\ Z^w]^T$ and a 3D point in the camera coordinate frame $[X^c\ Y^c\ Z^c]^T$, the transformation between coordinate systems can be defined as [2]

$$\begin{aligned} X^c &= R_{11} * X^w + R_{12} * Y^w + R_{13} * Z^w + T_x \\ Y^c &= R_{21} * X^w + R_{22} * Y^w + R_{23} * Z^w + T_y \\ Z^c &= R_{31} * X^w + R_{32} * Y^w + R_{33} * Z^w + T_z \end{aligned} \tag{4}$$

where R is the rotation matrix of the camera with respect to the world coordinate system and T is the translation vector of the camera with respect to the world coordinate system. The 2D pixel coordinates are defined as (x,y), where x and y can be found from X^c and Y^c. A vector $\mathbf{v}$ is created such that $\mathbf{v} = [R_{21}\ R_{22}\ R_{23}\ T_y\ \alpha R_{11}\ \alpha R_{12}\ \alpha R_{13}\ \alpha T_x\ ]^T$, where $\alpha = f_x/f_y$, and f_x and f_y are intrinsic camera parameters. Using $\mathbf{v}$ and a new matrix $\mathbf{A}$ the equation

$$Av = 0 \tag{5}$$

can be solved, where

$$A = \begin{bmatrix} x_1 * X_1^w & x_1 * Y_1^w & x_1 * Z_1^w & x_1 & -y_1 * X_1^w & -y_1 * Y_1^w & -y_1 * Z_1^w & -y_1 \\ x_2 * X_2^w & x_2 * Y_2^w & x_2 * Z_2^w & x_2 & -y_2 * X_2^w & -y_2 * Y_2^w & -y_2 * Z_2^w & -y_2 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ x_N * X_N^w & x_N * Y_N^w & x_N * Z_N^w & x_N & -y_N * X_N^w & -y_N * Y_N^w & -y_N * Z_N^w & -y_N \end{bmatrix}$$

The matrix A is composed by matching pixel points (x_k,y_k) with world points $[X^w\ Y^w\ Z^w]^T$. Singular value decomposition (SVD) can be used to solve for the intrinsic and extrinsic camera parameters. The camera parameters of interest are R, T, O_x, O_y, f_x, and f_y.

2.4.2 Application

Camera calibration is used to determine the initial position and orientation of the left camera with respect to the world coordinate system. The results of calibration will be necessary in order to perform stereo reconstruction.

Normally, the user has to select each pixel point by hand and pre-program the corresponding world points. A feature has been added so that cvGoodFeaturesToTrack() selects corners on the calibration cube and each corner can be selected with the mouse. As each corner is selected, the image is updated with a green circle around that particular corner. This eliminates the error associated with selecting exact pixel locations by hand, and expedites the calibration process.

2.5 Stereo Reconstruction

2.5.1 Theory

Stereo reconstruction by triangulation requires the intrinsic and extrinsic camera parameters found during *calibration*. (4) Let p_l and p_r be pixel points in the left and right image planes pil and pir, where π_l and π_r are parallel to each other and face the scene. Let O_l and O_r be the centers of projection for the left and right cameras, respectively. Given two scalar coefficients a and b, then $l = ap_l$ is defined as a ray extending from O_l through p_l and into the scene. Similarly, given ${}^lR_r$, the orthonormal rotation matrix, then $r = T + b^lR_rp_r$ is the ray extending fromO_rr through p_r, with respect to the left camera frame. [2] The 3D point corresponding the feature defined by p_l and p_r is the point P defined by the intersection of the rays r and l. The rays l and r, however, do not necessarily intersect, so a vector w is defined such that w is orthogonal to both l and r. Then the point P is the midpoint of the segment parallel to w that joins l and r. The equation

$$ap_l - bR^Tp_r + (cp_lx^lR_rp_r) = T \tag{6}$$

must be solved for $a_0, b_0, andc_0$, which define P, the 3D reconstructed point.

2.5.2 Application

The left and right cameras are separated by a known translation, and known rotation (identity matrix). Hence, once the extrinsic matrix of the left camera with respect to world is known, the right extrinsic matrix with respect to world is known as well. Once the left camera is calibrated, the extrinsic matrices for the left and right cameras are used for stereo reconstruction.

3 Mechanical Design

3.1 Lego Robot

A simple robot was constructed utilizing the Lego Mindstorm kit. The robot features a wide suspension, treds, and two motors. These three features allow the robot to travel in a straight line with minimal drift.

3.2 Vision Command Center

The Vision Command Center essentially consists of the stereo cameras mounted on a stationary platform. Wire is used to mount the cameras so that the position and orientation is adjustable. A box is surrounding the cameras to filter some of the ambient light from the fluorescent lights that often illuminate the room. The commander will issue commands from the Command Center, and the robot will move independently. See 5.4 for an image of the Command Center.

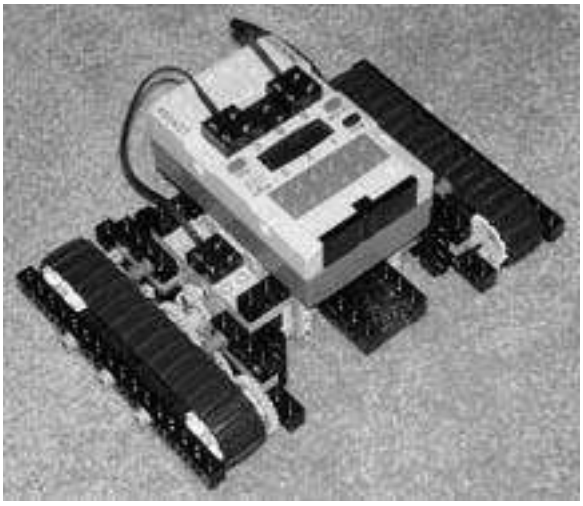

Figure 1: The Lego Robot.

4 Code Implementation

4.1 Principle Components Analysis

4.1.1 Convert an IplImage to an integer matrix

For a given test image region of an IplImage, the function ipl2intPCA() converts this region to an integer matrix for computational simplicity.

void ipl2intPCA(IplImage window, int x1, int y1, int x2, int y2, int matrix[PCA_SIZE][PCA_SIZE])*

This function traverses through the IplImage window within the region defined by a box with lower left and upper right corners $[x_1, y_1], [x_2, y_2]$, respectively. Each element in window is casted from char to int and stored in the integer matrix.

4.1.2 Eliminate Repeat Points

Before reconstructing a given pixel correspondence, it is necessary to ensure that there are no other points within close vicinity. Ideally, there should only be one detected feature in each $5x5$ pixel region, but often multiple features are detected in a given region. The function isRepeatPoint() eliminates any such repeat points.

bool isRepeatPoint(CvPoint edges[MAX_FEATURES], CvPoint point)

This function is based upon a similar function in (4.5.2). The code simply searches each feature in edges[] and computes the distance from point. If more than one feature is within a threshold distance of point, then there are repeat point(s). Thus, the function returns false and point will not be used.

4.1.3 Initialize basis vectors and training coefficient matrix from file

The basis vectors and training coefficients are the only two components needed to perform PCA. The function fillBasisVecs() initializes both the basisVecs[][] and trainingCoef[][] matrices, utilizing the fstream.h library for file input.

void fillBasisVecs(void)

The self-explanatory code iterates through the files, reading each float number and storing it sequentially.

4.1.4 Find fingers in left and right frames

Given any feature in the left image and feature in the right image, a point correspondence can be established if both features are finger tips. PCA is used to check every possible combination of left and right image features. If a given pair of features is sufficiently close to the finger tip training image, then isFingerPCA returns the number of training image that is the best match.

int isFingerPCA(IplImage gray1, IplImage* gray2, int lindex, int rindex)*

The arguments gray1 and gray2 are pointers to the left and right grayscale frames. The two indices lindex and rindex correspond to a detected feature in the left frame and a detected feature in the right frame, respectively. The function ipl2intPCA() is used to create two $30x30$ test images centered around lindex and rindex. These test images are converted into column vectors and are projected on the basis vectors (which are calculated prior to runtime). The resulting test coefficients are compared to the training coefficients to find the minimum distance. If smallest distance corresponds to the best match.

4.2 Stereo Reconstruction Code

PCA initially determines the corresponding point pair that is a finger tip. This point is then reconstructed to determine the 3D location of the finger tip. Using this information, a vector in the direction of the finger tip can be created and interpreted to move the robot in a given direction.

void stereoReconstruction(double Ox, double Oy, double fx, double fy, CvMat l_ExtMat_w, int tipMatch)*

This function takes the calibration parameters Ox, Oy, fx, fy, and l_ExtMat_w. The parameter tipMatch is the index of the corresponding point determined using PCA to be the finger tip. This function uses the CvMat matrix format because there are many useful functions such as cvInvert() and cvMulTranspose() that allow for optimized linear algebra computations. Using the calibration parameters, stereoReconstruction() converts the left and right pixel points into the 2D camera coordinate system. The vector q[] is generated, as explained in (2.5.1), and the matrices **A** and **b** are generated to find the solution to the equation (). The resulting wP[] solution is the 3D coordinates of the finger tip.

4.3 Stereo Callback Function

The two Logitech Fusion cameras capture left and right image frames, process these frames, and move onto the next frame. This function is responsible for processing each set of frames. The processing includes edge detection to find all possible features in both images, PCA analysis to find which features are finger tips, and stereo reconstruction to find the 3D position of the finger tip.

void stereocallback(IplImage image1, IplImage* image2)*

The constructor of class CorrespondingPts uses the cvcamSetProperty() function to synchronize and stream both left and right camera frames and pass pointers to the two frames to the stereocallback() function. Thus, image1 and image2 are pointers to the left and right color image frames. cvGoodFeaturesToTrack() requires grayscale (single-channel) images, so image1 and image2 are converted to gray1 and gray2 using

the cvCvtColor() function. Then the corners of gray1 and gray2 are detected using cvGoodFeaturesToTrack(). After removing any corners near the borders using the removeBorders() function, the outer-most for-loop selects the first corner of the left frame. A vertical search region is defined around each corner in the left image. Any corners within this search region in the right image undergo PCA analysis to determine if the corner is a finger tip. If both the left corner and the right corner are finger tips, then a match is found. The match is then checked by isRepeatPoint() to determine if there are any other matches within close proximity. (see 4.1.2) The variable FingerIndex is set by the isFingerPCA() function and corresponds to the best matching training image. (see 4.1.4) Depending on whether the feature is a finger tip, a thumb tip, or some point on the palm, the match will be displayed on the image using a certain color using cvCircle(). The match is then reconstructed using stereoReconstruction() and the direction[] vector is created to point in the direction of the finger tip, with respect to some point on the z-axis of the camera coordinate frame.

4.4 Sum of Squared Differences Code

4.4.1 Converting an IplImage to an integer Matrix

void ipl2int(IplImage window, int x1,int y1, int x2, int y2, int matrix[2 * PCA_SIZE + 1][2 * PCA_SIZE + 1])*

The IplImage is converted to an integer matrix in the function ipl2int(), which takes 6 parameters. The first is an IplImage pointer to a grayscale image (either left or right). The second through fifth parameters define the region of interest of the IplImage and the sixth is an empty integer matrix. The IplImage is converted to an integer to allow for the data to be squared and subtracted. IplImage data is stored as characters, which prevents numeric computation directly. The conversion is done by casting the character value in IplImage-¿imageData to an integer and storing it in the integer matrix.

4.4.2 Subtraction of the Region of Interest

*void intMatrixSubtract(int matrix1[2 * PCA_SIZE + 1][2 * PCA_SIZE + 1], int matrix2[2 * PCA_SIZE + 1][2 * PCA_SIZE + 1], int matrixDiff[2 * PCA_SIZE + 1][2 * PCA_SIZE + 1])*

The function takes in three parameter: the two matrices to be subtracted, and a matrix to store the difference. The maximum height and width are calculated to define the far side of the region of interest. Using these parameters a nested for loop is implemented which traverses the features of the right image, comparing them to the region of interest in the left image. Within these loops is where the subtraction takes place. This function is used by ssd2().

4.4.3 Squaring the Integer Matrix

*void matrixSquare(int matrix[2 * PCA_SIZE + 1][2 * PCA_SIZE + 1])*

Squaring the matrix is done in the function matrixSquare(), which takes one parameter, the integer matrix. The integer matrix is squared and then stored in the same matrix, so the original data is lost. This function is used by ssd2().

4.4.4 Summing the Integer Matrix

*int matrixSum(int matrix[2 * PCA_SIZE + 1][2 * PCA_SIZE + 1])*

The integer matrix is summed using the function matrixSum(). This function takes one parameter, the integer matrix from matrixSquare(). The function uses a nested for loop to sum every element in the matrix. This number is then returned to the calling function.

4.4.5 Sum of Squared Differences (SSD)

*long int ssd2(int matrix1[2 * PCA_SIZE + 1][2 * PCA_SIZE + 1], int matrix2[2 * PCA_SIZE + 1][2 * PCA_SIZE + 1])*

The two integer matrices passed to ssd2() correspond to feature windows in the left and right images, respectively. The function utilizes intMatrixSubtract(), matrixSquare(), and matrixSum() to perform the SSD calculation as presented in 2.3.1.

4.4.6 Remove Corners near Borders of Image

void removeBorders(CvPoint points1[MAX_FEATURES], CvPoint points2[MAX_FEATURES], CvSize size)

The two CvPoint arrays are the corners detected in the left and right camera frames, and the CvSize argument is the size of the frames. The function removes any corners in points1 or points2 that is within 1.5*PCA_SIZE of the image borders, where PCA_SIZE is the dimension of the square window created around each corner for SSD analysis.

4.5 Calibration Code

4.5.1 Mouse Callback Function

void mouse_callback (int event, int x, int y, int flags, void param)*

The constructor of class CorrespondingPts uses the cvcamSetProperty() function to return the (x,y) pixel coordinates of the point on the image clicked by the mouse to the mouse_callback() function. However, the y coordinate must be inverted because the coordinate system of the mouse_callback() function and of the image are not the same. Each point selected is stored in a static global array. Every time mouse_callback() is called, the variable ptsGrabbed is incremented to keep track of how many calibration points have been selected.

4.5.2 Eliminate Repeated Points

void eliminatePoints(CvPoint edges[NUM_CALIBRATION_PTS])

The edge detector often finds several copies of a given edge, and the copies must be eliminated. For each edge, this function checks to see if there is another edge within 10 pixels in the x and y directions, and eliminates any points meeting this criterion.

4.5.3 Refresh the Screen with new Calibration Points

void refresh(IplImage frame, CvPoint edges[NUM_CALIBRATION_PTS])*

As the user selects calibration points on the image, the points selected are highlighted with a green circle. Every edge in the image is marked with a blue circle, so green circles show the contrast between those edges selected to be calibration points and those edges that are not used. The function destroys the previous window, creates a new window, and displays the list of selected calibration points.

4.5.4 Find Nearest Point to the Point given by Mouse Callback

void findNearestPoint(int Xp, int Yp, CvPoint edges[NUM_CALIBRATION_PTS], int &index)

Given a pixel point from the mouse_callback() function, the closest edge must be found and the

point replaced with the edge coordinates. Thus, when the user clicks on a pixel point in the image, the nearest edge (found with edge detection) is stored.

4.5.5 Calibrate the Camera

void calibrate(IplImage imgl, double &Ox, double &Oy, double &fx, double &fy, double &Tz, double &Tx, double &Ty, double &zDistance,double pcRotw[3][3], double pwRotc[3][3])*

A set of 3D points on the calibration cube, with respect to the world coordinate frame, are initialized. These 3D points correspond to a pattern of pixel points (edges) that will be selected by the user. Next matrix A is filled, according to equation 5 described in 2.4.1, using the cvmSet() function. cvSVD() is used to find the matrix **v**, as defined in 2.4.1. Matrix **v** is decomposed into its 8 elements $[R_{21}\ R_{22}\ R_{23}\ T_y\ \alpha R_{11}\ \alpha R_{12}\ \alpha R_{13}\ \alpha T_x\ ]^T$ and the camera parameters are extracted.

4.5.6 Calibration Class Constructor

The constructor uses cvCaptureFromCAM() to capture a single frame from the camera. The function cvSetMouseCallback() defines which callback function should be called for each mouse click. The function cvGoodFeaturesToTrack detects edges in the image and eliminatePoints() destroys any repeated points. Then the main loop iterates until ptsGrabbed = NUM_POINTS_GRABBED, upon which the function calibrate() is called.

4.6 OpenCV Borrowed Code

Many of the function calls mentioned above come from the OpenCV libraries. These libraries include highgui.lib, cv.lib, cvcam.lib, and cxcore.lib. The usage of each function is explained throughout the report as each function is mentioned.

5 RESULTS

5.1 Point Correspondence

The accuracy of SSD for point correspondence, as described in 2.3.1, is inconsistent at best. Small lighting changes effect the pixel intensities inconsistently over certain regions of the image. SSD analysis relies directly on the pixel intensities surrounding the features, rather than on the features themselves.

PCA presents a far more accurate method of determining point correspondence for the RCS. Rather than comparing pixel intensities with a region surrounding a given feature, PCA compares the most important pixel intensities of a region surrounding a given feature to a training image, as described in 2.2.1. PCA has replaced SSD analysis in the point correspondence code, but the old SSD code remains in the report because of its prior importance to the RCS project. The two figures below demonstrate that when using PCA for point correspondence, only one match is found and it is a correct match (finger-tip). When using SSD for point correspondence, however, many matches are found, some of them being incorrect.

Matches are less consistent using SSD because there are more corners appearing in the left frame than in the right frame. The corners detected in the left frame are only printed if there is a matching corner in the right frame, so there must be several false-positive corners detected. If a corner in the left frame is matched to only one corner in the right frame, and the right frame corner has already been printed, it may be printed again. This bug is easily resolved using PCA.

Secondly, the program seems to use more memory as the program continues to run. Several optimizations have been made, such as declaring large variables as static globals, but with little success. After 5 minutes of operation, the program consistently crashes due to insufficient memory. The solution may be to declare all variables, even counters and small local variables as static globals.

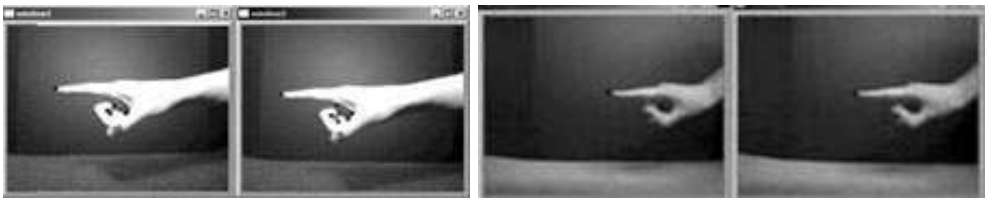

Figure 2: This figure illustrates the difference between SSD-based point correspondence and PCA-based point correspondence. The left image shows SSD finding incorrect matches, and the right image shows PCA finding only one correct match on the finger-tip.

5.2 Calibration

Utilizing edge detection to select the corners of the calibration cube practically eliminated error associated with mouse-selection of calibration points. The user is able to click any pixel location near a given edge and the program will store the exact pixel location of that edge. Thus, one of the largest sources of error associated with camera calibration has been eliminated. Figure shows the calibration image.

As the camera moves away from the calibration cube, two problems arise. First, fewer and fewer calibration points can be recognized by the corner detector. When selecting a set pattern of calibration points, many corners may be missing, which introduces tremendous error. To solve this problem, the distance between the camera and cube must be limited to some known value. At the maximum distance, a pattern of calibration points (and their corresponding world points) can be selected. As the camera moves closer to the calibration cube, these points will remain. The second problem background features may become more apparent, as the camera moves further from the calibration cube, and could be selected as corners instead of calibration points. One solution to this problem is to use a large black background to prevent other features from being selected. The results of camera calibration using are shown in below.

${}^cX_w$	${}^cY_w$	${}^cZ_w$
-0.732759	-0.0109125	0.680401
0.106035	-0.9915530	0.074695
-0.673925	-0.1268960	-0.727820
${}^cTx_w$	${}^cTy_w$	${}^cTz_w$
0.432352	1.38777	-10.447

Table 1: Calibration Results

Figure 3: Automatic edge detection selects the corners of the calibration cube. Any circled edge can be selected as a calibration point.

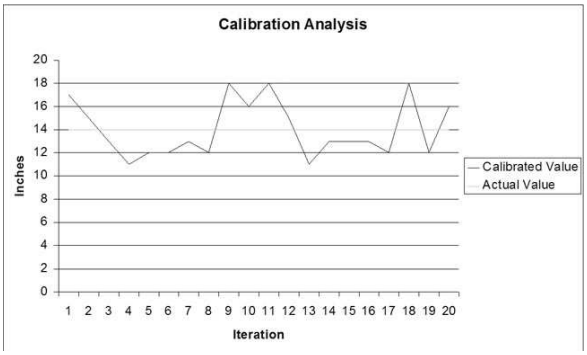

Figure 4: This figure shows the difference between the calibrated and actual distance between the camera and cube. The trial is repeated at different distances, and the results show that the further the distance, the greater the error.

5.3 Stereo Reconstruction

The reconstructed 3D finger-tip points were accurate enough to determine relative position of the commander's finger. Although the stereo algorithm, as described in 4.2, is theoretically correct, error was inevitably introduced.

The first potential source of error is the point correspondence code, being driven by PCA. In order to eliminate this source of error, the number of features detected by cvGoodFeaturesToTrack() was reduced to one. Thus, only one feature was found in the left and right frames. The black background behind the commander made the protruding index finger an excellent feature to track. If the left and right features are not both finger-tips, then PCA will elliminate both features, and they will not be reconstructed. Thus, point correspondence is not a source of error.

Calibration generally returned accurate results, but often the y-component of the translation vectors were incorrect. Small inaccuracies in calibration often create larger errors in stereo. Further, stereo seemed to be less accurate in the ${}^{W}Y$ direction, but fairly accurate in the ${}^{W}X$ and ${}^{W}Z$ directions.

Accuracy in the ${}^{W}X$ and ${}^{W}Z$ directions, allowed for change-in-depth to be inferred from frame to frame. Thus, as the commander moves his or her index finger closer to or further from the cameras, a change in reconstructed depth translates to a change in robot position.

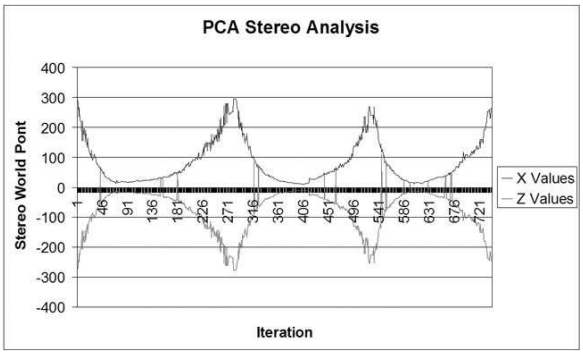

Figure 5: This figure illustrates the reconstructed 3D world points over several thousand frames as the commander moves her finger. As the finger moves further away fromt the cameras, the reconstructed depth grows, and as the finger moves closer to the cameras, the reconstructed depth approaches zero.

5.4 Robot and Command Center

The two cameras are removed from the Logitech rubber mounts, and remounted on custom wire holders. The cameras will remain stationary and facing the Commander, while the robot responds to the commands nearby. The advantage to stationary cameras is that the Commander can remain in one position, and the robot does not have to find the user to recognize commands. This allows for easier alignment of the cameras, and will reduce error in the *stereo reconstruction* algorithm.

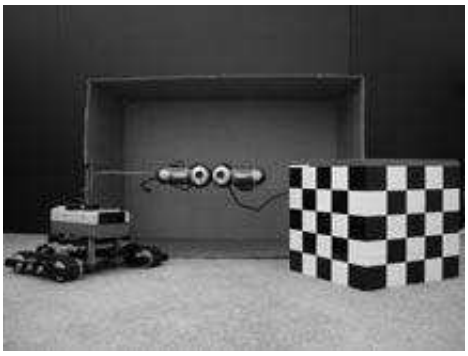

Figure 6: The Command Center including the robot, cameras and calibration cube.

6 OpenCV Setup Instructions

Processing video data from two cameras simultaneously can be accomplished most efficiently by utilizing the cvcam project, included with OpenCV. The cvcam stereocallback() function allows automated synchronization between the stereo video streams. Further, utilizing a callback function allows more efficient real-time video streaming than the cvCaptureFromCAM() function. The following steps should be taken to setup (in Visual C++ 6.0) the cvcam project to stream and process video data from two cameras simultaneously:

1. Download and install OpenCV
2. Download and install DirectX 9.0 SDK Update - (Summer 2003) found at msdn.microsoft.com
3. Setup Visual C++ 6.0

(a) create an empty Win32 Console Application.

(b) Create a main.cpp file that will contain your main() function code.

(c) Include the header files cv.h cvcam.h and highgui.h for OpenCV.

4. Setup Project Settings

   (a) Go to Project/Setting/General to 'Use MFC in a Shared DLL'.

   (b) Go to Project/Setting/All Configuration/C/C++/Preprocessor/Additional Include Directories and add OpencvInstallFolder/cv/include, OpencvInstallFolder/cxcore/include, and OpencvInstallFolder/ot

   (c) Go to Project/Setting/Link/Input/Additional library path and add OpencvInstallFolder/lib.

   (d) Go to Project/Setting/Link/Input/Object/library modules and add cv.lib highgui.lib cxcore.lib

   (e) Go to Project/Setting/Link/General/Object/library modules and add cv.lib highgui.lib cxcore.lib.

5. Setup .dll files for Video Stream Windows

   (a) Copy all .dll files from OpencvInstallFolder/bin to VC++ProjectFolder/Debug
   Note: this is necessary to ensure the video output windows have all necessary library files

   (b) If the CVd.dll and HighGUId.dll not found, do a search for these two files and copy both of them to VC++ProjectFolder/Debug

6. Insert the cvcam project into your workspace. If you run into a stream.h or streams.h error after compilation, move on to step 7, otherwise skip to step 9.

7. Build the directshow baseclasses to get strmbase.lib and strmbasd.lib.

   (a) Open DirectXSDKInstallFolder/samples/Multimedia/DirectShow/BaseClasses/baseclasses.dsw.

   (b) Go to Build/Batch Build, check both debug and release, and say Rebuild All.

   (c) The libraries can be found at
   DirectXSDKInstallFolder/samples/Multimedia/DirectShow/BaseClasses/Debug
   DirectXSDKInstallFolder/samples/Multimedia/DirectShow/BaseClasses/Release

8. Setup Visual C++ 6.0 once you have built the directshow baseclasses.

   (a) Copy strmbase.lib and strmbasd.lib to DirectXSDKInstallFolder/lib.

   (b) In Visual C++ 6.0, go to Tools/Options/Directories/Include files and add the following lines to the Include Search path:
   DirectXSDKInstallFolder/include and
   DirectXSDKInstallFolder/samples/Multimedia/DirectShow/BaseClasses

   (c) In Visual C++ 6.0, goto Tools/Options/Directories/Library files and add the paths DirectXSDKInstallFolde to the libraries search path. *Note: in parts b and c, move the added DirectX paths to the top of the lists.*

9. Insert cv, cxcore, highgui, and any other projects needed into your workspace.

   (a) See online OpenCV documentation to learn about different functions in each project:
   http://www.itu.dk/stud/projects_f2004/handtracking/OpenCV/docs/

   (b) Any other OpenCV projects (other than cv, cxcore, highgui, and cvcam) that you insert must be included in the Project Settings (see Step 4).

10. Setup Project Dependencies in Visual C++ 6.0

    (a) Go to Project/Dependencies, select your console project from menu.

    (b) Check all OpenCV projects that you have inserted (cv, cxcore, highgui, etc)

11. Edit main.cpp to setup cameras and a stereocallback function as shown in Appendix

References

[1] Jianbo Shi and Carlo Tomasi. *Good Features to Track.* IEEE Conference on Computer Vision and Pattern Recognition, Seattle, WA, 1994.

[2] Emanuele Trucco and Alessandro Verri. *Introductory Techniques for 3-D Computer Vision.* Prentice Hall, Upper Saddle River, NJ, 1998.

[3] Matthew Turk and Alex Pentland. *Eigenfaces for Recognition.* Journal of Cognitive Neuroscience, Boston, MA, 1991.

7 Implications of Robotics and Computer Vision in Society

Although investments and interest in robotics and computer vision had declined in the past, current trends seem to suggest that future investments in robotics and computer vision will increase and robotics will become more widespread as technology continues to evolve and improve. In the past, interest in robotics had fallen away due to a number of institutional, economic, and technological barriers, including lack of national funding, lack of communication between researchers and developers, and lack of reliable hardware and software. As time has passed, these barriers are slowly being overcome as computers became faster, more reliable, cheaper, and able to process large amounts of data in very little time (The Future and Intelligent Machines: Charting the Path, 1997).

Robots have fascinated mankind for thousands of years, all throughout history. The first written mention of intelligent machines comes from Homer's Iliad, where the god of fire and metalwork, Hephaestus, builds twenty tripod robots to do his bidding. About eight hundred years later, Hero of Alexandria created mechanical animals that ran on air, water, or steam. In the sixteenth century, clockwork machines were refined by craftsmen in Western Europe and from this, were refined further to be used to create the animated dolls found today in both Europe and Japan (Graefe, 2003). The first robots, as we know them today, were introduced in 1961 in the United States of America and were used in industrial manufacturing. Robots today are considered to have some kind of computational intelligence and, as technology gets more powerful, the possibilities become endless. To be considered a robot, a machine must be able to do two things: obtain and process information from its surroundings and react or do something physical in or to its surroundings. Since 1961, robotics has spread into fields other than industry, including the military, medicine, space exploration, service, entertainment, and home use (Graefe, 2003). Today, the most widespread type of robot is the industrial robot. More than one million industrial robots are used in factories around the world. Although not as artistic as personal or service robots, industrial robots play an important role in creating goods for our everyday needs. Industrial robots aid in the production of goods, even though they are not the most intelligent type of machine available (Wilson, 1994). Industrial robots generally have no intelligence and very little sensory abilities. This means that, in order for them to function, they must work in very strict conditions with an expert supervising them, and must stop at emergency, whether or not it directly involves them. Most industrial robots are designed to fulfill a specific task and only under certain conditions (Graefe, 2003). Due to the dip in industrial robotic growth, the robotic industry is looking to recreate industrial robots. Recent trends in industrial technology are to make things compact, synergetic, intelligent, and environmentally friendly. The United States alone has over forty thousand robots working in factories all across the country, which is a number far behind that of other countries, such as Japan and Europe whose industrial robots number in the hundred thousands (Ejiri, 2001). If the U.S. invested more in industrial robots, by simply doubling the number of robots available to work, it could double productivity, leading to possibly delivering one trillion more dollars into the U.S.'s economy (Thro, 1993). Although these robots are the most commonplace, robotics has expanded into other areas of life as well. These

areas include the workplace, the military, medicine, space exploration, entertainment, and even the home. Recently, robotics has been expanding past the common industrial robots used to produce goods in factories to personal robots that can be found in the home doing mundane tasks and helping the elderly, service robots geared at helping or replacing workers in the workplace, and to medical robots which assist doctors and nurses in hospitals. Robotics is expected to expand further in both service and home use, as service and personal robots become more refined and become higher in demand. As of now, robots created to serve as servants or butlers only exist as prototypes in laboratories, but may one day become as common as the home personal computer (Graefe, 2003). Service robots are very popular in Japan, which invests largely in research and development in robotics, closely followed by the United States and the European Union. Robotics may also one day become popular in aiding with the elderly both in their homes and in nursing homes, performing tasks such as monitoring an elderly person's vital statistics or simply keeping the elderly company. Many hospitals already use robots to deliver equipment, carry charts, to aid in surgeries, and to dispense medicines (Kanellos, 2004). Research has also begun in using robots in maintenance work, both on Earth to survey architecture and in space, for repairs on the Hubble telescope and other satellites.

Robots found in the home are commonly called "personal robots." These robots are still evolving, but were mainly created to serve and help people in their homes. There are two types of personal robots: the type that is meant to comfort and entertain people and the type that is made to do chores around the house. The first type of robot deals generally with emotions and does not do anything useful in the household. This type of robot is the kind that would be designed as a robotic pet or companion. An example of this type of robot is Sony's AIBO which resembles and acts like a robotic dog. The original AIBO sold for two thousand five hundred dollars, and was sold out within twenty minutes of being released. Within weeks, orders for one hundred thirty-five thousand AIBO dogs were placed in the U.S., Europe, and Japan. Robotic pets are simpler to implement than robotic servants, because deficiencies in its functioning and specifications would be permissible to its owner, since most people do not expect their cute robotic pet to be able to complete many tasks. Also, aiding the pet could actually make its owner appreciate it more through interacting with it. On the other hand, the second type of robot, the personal robotic servant or assistant, needs to function perfectly every time to please its owner. This kind of personal robot is the type that will eventually evolve into a robotic butler or maid. The humanoid versions of these robots are still being developed in laboratories, due to the fact that any of the mistakes they make will probably not be tolerated well by their owners and robots are not yet equipped to apologize or show any type of regretful emotion in response to these mistakes (Graefe, 2003). Many smaller, less intelligent robots can be found in the home executing daily household tasks such as vacuuming (Asami, 1994). The most popular type of household robot is known as Roomba, a small robotic vacuum. A Roomba sells for about two hundred dollars each and is designed to vacuum the house. In the first two years it was marketed, Roomba's sales were reported to be over five hundred

thousand, leading to it successor, the Discovery. Demand for robots such as Roomba increases every year, since over twenty-five percent of households claim that they are too busy to do everyday chores such as cleaning and vacuuming. Demands such as these will also one day fuel the market for robotic butlers or maids, as well as other automatic robotics systems such as cars that drive themselves (Ichbiah, 2005). Many companies are putting in time and research into developing personal robots, including Fujitsu, NEC, Omron, Sanyo, Sony, and Honda (Graefe, 2003).

Robots are also actively used in the entertainment industry, as well as for entertainment. The Japanese public has endeared the world to robots through use of movies and television shows, due to their love of robots, while in American culture, movies such as Robots!, AI, I, Robot, Bicentennial Man, and the Terminator leave mixed feelings about robots. In the movie making industry, robots are being used in conjunction with computer graphics to create life like interpretations of movie characters, such as the shark in Jaws, the dinosaurs in Jurassic Park, and even the ape in King Kong to entertain audiences. These robots are known as animatronics. Robots have also become popular as toys, which can be seen through toys such as Furbies and LEGO Mindstorms. Robotic pets are another form of entertainment, which were mentioned previously (Ichbiah, 2005).

Another avenue for robotics in the home to take is assisting the elderly and chronically ill. As the number of elderly people increases each year, the nurses to aid them and the young people to keep them company will become harder and harder to find. To combat this, "smart" houses and robotic assistants can be used to keep the elderly company and to assist them with their everyday needs, allowing them to possibly live independently (The Future and Intelligent Machines: Charting the Path, 1997). In the past few years, robots similar to those that would be working with the elderly and chronically ill have been used in hospitals, museums, and stores to help keep buildings clean, deliver needed objects, educate, and entertain patrons. These robots may one day function as a cognitive prosthesis to help the elderly remember tasks, such as when to take medicine dosages. These robots will also be capable of safeguarding the patient, so they can alert someone if the patient needs help. Service robots that assist the elderly will also be capable of collecting data on the patient's daily activities, allowing the doctor to remotely speak with the patient, and social interaction with the patient. Some designs are currently available, but they require a lot of work on the user's part to run the robot (Roy).

The next recent trend in robotics is the robot termed "service robot," which is popular in Japan. Service robots interact with or replace people in the workplace. Generally, these robots replace workers in dangerous situations, such as those with high temperatures, vacuums, underwater activity, or radioactivity. These robots also aid in firefighting, police and military operations, architecture inspections and maintenance, publicity (Asami, 1994), and collecting and disposing of hazardous wastes in hospitals. Robots have been used to explore many hazardous landscapes, such as the underwater explorations

of the Titanic, the close quarters of the Egyptian pyramids, and the foreign surfaces of other planets (Ichbiah, 2005). In amusement parks, service robots act as robotic characters that interact with park visitors to entertain them. Service robots can also be used in the military, to assist soldiers and run reconnaissance missions to lessen fatalities (The Future and Intelligent Machines: Charting the Path, 1997). Another example of robots in the military is the unmanned aircraft, which is currently being developed to save pilots from the danger of flying. Over four hundred billion dollars goes into robotics military research each year, researching how to make equipment for soldiers refined, smart, and light weight (Ichbiah, 2005). Another big sector in robotics is space exploration. Robotics and computer vision have been used to develop planetary rovers and probes, which roam a selected planet's surface in conjunction with humans on Earth. A popular example of these rovers are the Mars rovers, Spirit and Opportunity, which went up into space in early 2004 and continue to roam Mars' surface despite many problems (Ichbiah, 2005). Another controversial topic in robotics is using robots to repair the Hubble Space Telescope. Although initially thought to be a given due to the failures of the past space shuttle launches, research has shown that a robotic mission to save Hubble might not be enough to save it. The robot that would be repairing Hubble would need to be very autonomous due to communication lag and would have to have very sensitive arms that would be able to change fragile parts in confined conditions. Also, repair by the robot would be much slower than if a human astronaut were to repair it. Costs of this project are in the billions, which may inhibit using robotics to repair Hubble (Ashley, 2004). Similar to space exploration, robotics has also been used

The United States is largely regarded as the world's most productive and technologically advanced society. Naturally, the U.S. wants to be part of the leading edge in research and development in the field of robotics. Since the creation of the Robotic and Intelligent Machines Cooperative Council (RIMCC), industrial robot sales almost tripled between 1991 and 1997, which was attributed to both drops in overall price and increases in machine capability (The Future and Intelligent Machines: Charting the Path, 1997). U.S. investment continued to increase through 1999 but then dropped in 2000. Since 2001, U.S. investment in robotics research and development is on the rise once again, although slower due to much research and development funds going into homeland security.

Although the United States has put much funding into robotics, the leading country in robotics is currently Japan. Their main focus is in the implementation of service robots and the components needed to implement them. These components include mobility, manipulation, sensing and artificial intelligence. Japan's program was implemented in 1993, with the national project called the "Research and Development of Micromachines." Currently, there are no statistics on the number of Japan's service robots, but based on statistics of its industrial robot deliveries, Japan's robot population increases every year, when billions of robots are purchased (Asami, 1994). Most industrial robots are found in the automotive and electronics industries, where assembly robots are ideal for the

repetitive motion and multiple shifts (Wilson, 1994). Service robots will also become more popular as shortages in the simple and aging specializing labor markets decrease, and as human workers are replaced by robots in hazardous working environments. Unfortunately, service robots are held back by insufficient equipment and high cost. These problems will hopefully be alleviated in the future, as technology progresses. In addition to the applications listed above, service robots can also be found in civil engineering and in medical care.

Therefore, as we have seen, robotics can be found and used in many aspects of life. Whether the robots are the personal robots found in homes or service robots who replace and aid workers in the workplace, robots have many useful applications which can save humans time and effort and can even safeguard them from hazardous jobs. As technology becomes cheaper and robots become more available, useful robots, such as Roomba, and entertainments robots, such as Sony's AIBO, will become more in demand. This fact is supported by the hundreds of thousands of buyers these two robots have attracted since they became available to the general public. Robotics and computer vision have many applications, allowing them to expand further in space exploration, medicine, industry, home use, entertainment, and in the workplace. For these very reasons, robotics seems to have a bright future, expanding in the areas of life it already affects and even possibly moving onto new fields.

Investment in robotics research and development grows every year and is lead by Japan, who is closely followed by the United States and European Union. These investments may one day lead to a better, safer world thanks to robots. In conclusion, for the reasons and facts presented above, robotics and computer vision applications are alive and thriving, despite the disinterest and declines in funding in the past. Current trends point to increases in robot use and funding, as technology continues to evolve and improve, making robots more accessible to the general public.

8 Code

8.1 CorrespondingPts.cpp

```
#include <iostream>
#include <fstream>
using namespace std;

#include "CorrespondingPts.h"
#include "Calibration.h"


/*------------------------------- Variables --------------------------------*/


//PCA Variables
static CvPoint leftPoint;
static CvPoint rightPoint;
static int leftTestImage[PCA_SIZE][PCA_SIZE];
static int rightTestImage[PCA_SIZE][PCA_SIZE];
static float leftColVec[PCA_SIZE * PCA_SIZE], leftCoef[PCA_SIZE * PCA_SIZE];
static float rightColVec[PCA_SIZE * PCA_SIZE], rightCoef[PCA_SIZE * PCA_SIZE];
static float basisVecs[NUM_BASIS_VECS][PCA_SIZE * PCA_SIZE];
static double leftDist[NUM_IMAGES], rightDist[NUM_IMAGES];
static double minDist;
static double trainCoef[NUM_BASIS_VECS][NUM_IMAGES];
static int matches = 0, minIndex = 0, FingerIndex = 0, tipCount = 0;
static float tipVec[3], baseVec[3], direction[3];
static int tipMatch[MAX_FEATURES];
static int lastPoint = 5000;

//Good Features to Track Variables
static CvPoint image1_points[MAX_FEATURES], image2_points[MAX_FEATURES];
static CvPoint leftFingerPoints[MAX_FEATURES], rightFingerPoints[MAX_FEATURES];
static CvPoint2D32f image1_features[MAX_FEATURES],
image2_features[MAX_FEATURES];

//Stereo Variables
static double leftPixelPoint[2][NUM_FINGER_PTS];
static double rightPixelPoint[2][NUM_FINGER_PTS];
static double leftCameraPoint[3][NUM_FINGER_PTS];
static double rightCameraPoint[3][NUM_FINGER_PTS];
static double Ox, Oy, fx, fy, Tz, Tx, Ty;
static double leftTransVec[3];
static double rightTransVec[3];
static double l_ExtMat_r[4][4];
static double q[3];
static double qNormalized[3];
static CvMat* affineMotion;
static CvMat* worldPoint;
static CvPoint3D32f wP[NUM_FINGER_PTS];
static CvMat* lExtMatw;
```

```
/*----------------------------- PCA Functions -----------------------------*/


//read basis vector info from file and fill basisVecs[][]
void fillBasisVecs(void)
{
	int i, j;
	ifstream readFile1, readFile2;
	readFile1.open("basisvecs.txt");

	for(i = 0; i < PCA_SIZE * PCA_SIZE; i++)
	{
		for(j = 0; j < NUM_BASIS_VECS; j++)
		{
			readFile1 >> basisVecs[j][i];
		}
	}

	readFile1.close();

	readFile2.open("trainingcoef.txt");

	for(i = 0; i < NUM_BASIS_VECS; i++)
	{
		for(j = 0; j < NUM_BASIS_VECS; j++)
		{
			readFile2 >> trainCoef[j][i];
		}
	}

	readFile2.close();
}


//converts an IplImage to an int matrix
void ipl2intPCA(IplImage* window, int x1, int y1, int x2, int y2,
                                int matrix[PCA_SIZE][PCA_SIZE])
{
	int iindex = 0;
	int jindex = 0;
	int i = 0;
	int j = 0;

	for(iindex = y1; iindex < y2; iindex++)
	{
		for(jindex = (x1 + (window->widthStep) * iindex);
		jindex < (x2 + (window->widthStep) * iindex); jindex++)
		{
			matrix[i][j] = (int)(window->imageData[jindex]);
			j++;
```

```
                }

                i++;
                j = 0;
        }
}


//set any point within WINDOW_SIZE of the image border to NULL (-1)
void removeBorders(CvPoint points1[MAX_FEATURES],
                                CvPoint points2[MAX_FEATURES], CvSize size)
{
        for(int i = 0; i < MAX_FEATURES; i++)
        {
                if(points1[i].x < (BORDERSCALE * WINDOW_SIZE) ||
                        points1[i].x > size.width - (BORDERSCALE * WINDOW_SIZE) ||
                        points1[i].y < (BORDERSCALE * WINDOW_SIZE) ||
                        points1[i].y > size.height - (BORDERSCALE * WINDOW_SIZE))
                {
                        points1[i].x = -1;
                        points1[i].y = -1;
                }
                if(points2[i].x < (BORDERSCALE * WINDOW_SIZE) ||
                        points2[i].x > size.width - (BORDERSCALE * WINDOW_SIZE) ||
                        points2[i].y < (BORDERSCALE * WINDOW_SIZE) ||
                        points2[i].y > size.height - (BORDERSCALE * WINDOW_SIZE))
                {
                        points2[i].x = -1;
                        points2[i].y = -1;
                }
        }
}


//eliminate repeated points
bool isRepeatPoint(CvPoint edges[MAX_FEATURES], CvPoint point)
{
        int i = 0, numRepeats = 0;
        bool repeat = false;

        for(i = 0; i < MAX_FEATURES; i++)
        {
                if(abs(edges[i].x - point.x) < 10 && abs(edges[i].y - point.y) < 10)
                        numRepeats++;
        }

        if(numRepeats > 1)
                repeat = true;

        return repeat;
}
```

```
//check to see if left feature and right feature (match) are both a
finger using PCA
int isFingerPCA(IplImage* gray1, IplImage* gray2, int lindex, int rindex)
{
	int i = 0, j = 0, k = 0;
	leftPoint = image1_points[lindex];
	rightPoint = image2_points[rindex];

	//create test images
	ipl2intPCA(gray1, leftPoint.x - ((PCA_SIZE / 2) - 1),
		leftPoint.y - ((PCA_SIZE / 2) - 1), leftPoint.x + (PCA_SIZE / 2),
		leftPoint.y + (PCA_SIZE / 2), leftTestImage);
	ipl2intPCA(gray2, rightPoint.x - ((PCA_SIZE / 2) - 1),
		rightPoint.y - ((PCA_SIZE / 2) - 1), rightPoint.x + (PCA_SIZE / 2),
		rightPoint.y + (PCA_SIZE / 2), rightTestImage);

	//reshape test images into column vectors
	for(i = 0; i < PCA_SIZE; i++)
	{
		for(j = 0; j < PCA_SIZE; j++)
		{
			leftColVec[k] = (float)leftTestImage[i][j];
			rightColVec[k] = (float)rightTestImage[i][j];
			k++;
		}
	}

	//take dot product with basis vectors to get test coef
	for(k = 0; k < NUM_BASIS_VECS; k++)		//for each basis vector
	{
		leftCoef[k] = 0.0;
		rightCoef[k] = 0.0;

		for(i = 0; i < PCA_SIZE * PCA_SIZE; i++)
		{
			leftCoef[k] += leftColVec[i] * basisVecs[k][i];
			rightCoef[k] += rightColVec[i] * basisVecs[k][i];
		}
	}

	//calculate distance between test coef and training coef
	minDist = PCA_THRESH;
	minIndex = -1;

	for(i = 0; i < NUM_IMAGES; i++)
	{
		leftDist[i] = 0.0;
		rightDist[i] = 0.0;

		for(k = 0; k < NUM_BASIS_VECS; k++)
		{
```

```
                        leftDist[i] += pow(leftCoef[k] - trainCoef[i][k], 2);
                        rightDist[i] += pow(rightCoef[k] - trainCoef[i][k], 2);
                }

                leftDist[i] = pow(leftDist[i], 0.5);
                rightDist[i] = pow(rightDist[i], 0.5);

                if(leftDist[i] < minDist && rightDist[i] < minDist)
                {
                        minDist = (leftDist[i] + rightDist[i]) / 2.0;
                        minIndex = i;
                }
        }

        return minIndex;
}


/*---------------------------- Stereo Functions----------------------------*/


//stereo initializations
void stereoInit()
{
        int i = 0;

        //stereo variable initialization
        for(i = 0; i < 3; i++)
        {
                leftTransVec[i] = 0;
                rightTransVec[i] = 0;
                qNormalized[i] = 0;
                q[i] = 0;
        }

        rightTransVec[0] = 1.2; //measured by hand w.r.t. left camera frame

        for(i = 0; i < 4; i++)
        {
                for(int j = 0; j < 4; j++)
                {
                        l_ExtMat_r[i][j] = 0;

                        if(i == j)
                        {
                                l_ExtMat_r[i][j] = 1;

                        }
                }
        }

        for(i = 0; i < NUM_FINGER_PTS; i++)
```

```
        {
                leftPixelPoint[0][i] = -1000;
                leftPixelPoint[1][i] = -1000;
                rightPixelPoint[0][i] = -1000;
                rightPixelPoint[1][i] = -1000;
                leftCameraPoint[0][i] = -1000;
                leftCameraPoint[1][i] = -1000;
                leftCameraPoint[2][i] = 1;
                rightCameraPoint[0][i] = -1000;
                rightCameraPoint[1][i] = -1000;
                rightCameraPoint[2][i] = 1;
                wP[i].x = -1000;
                wP[i].y = -1000;
                wP[i].z = -1000;
        }
}


//stereo reconstruction
void stereoReconstruction(double Ox, double Oy, double fx, double fy,
CvMat* l_ExtMat_w)
{
        int i = 0;
        double sx = 1 / fx;
        double sy = 1 / fy;

        //setup blank matrix of size 2N by 12
   CvMat* A;
        A = cvCreateMat(3, 3, CV_32F);
        CvMat* b;
        b = cvCreateMat(3, 1, CV_32F);
        CvMat* lP;
        lP = cvCreateMat(3, 1, CV_32F);
        CvMat* w_ExtMat_l;
        w_ExtMat_l = cvCreateMat(3, 3, CV_32F);
        CvMat* temp;
        temp = cvCreateMat(3, 3, CV_32F);
        CvMat* temp2;
        temp2 = cvCreateMat(3, 3, CV_32F);

        rightTransVec[0] = 1.2; //measured by hand w.r.t. left camera frame
        rightTransVec[1] = 0;
        rightTransVec[2] = 0;

        //find right w.r.t left parameters
        l_ExtMat_r[0][3] = rightTransVec[0];
        l_ExtMat_r[1][3] = rightTransVec[1];
        l_ExtMat_r[2][3] = rightTransVec[2];

        for(i = 0; i < matches; i++ )
        {
                if(leftPixelPoint[0][i] > -1000 && rightPixelPoint[0][i] > -1000)
```

```
        {
                leftCameraPoint[0][i] = leftPixelPoint[0][i] - Ox * (-1 * sx);
                leftCameraPoint[1][i] = leftPixelPoint[1][i] - Oy * (-1 * sy);
                rightCameraPoint[0][i] = rightPixelPoint[0][i] - Ox * (-1 * sx);
                rightCameraPoint[1][i] = rightPixelPoint[1][i] - Oy * (-1 * sy);
        }
}

for(i = 0; i < matches; i++)
{
        if(leftCameraPoint[0][i] > -1000 && rightCameraPoint[0][i] > -1000)
        {
                q[0] = leftCameraPoint[1][i] * rightCameraPoint[2][i] -
                        leftCameraPoint[2][i] * rightCameraPoint[1][i];
                q[1] = -1 * (leftCameraPoint[0][i] * rightCameraPoint[2][i] -
                        leftCameraPoint[2][i] * rightCameraPoint[0][i]);
                q[2] = leftCameraPoint[0][i] * rightCameraPoint[1][i] -
                        leftCameraPoint[1][i] * rightCameraPoint[0][i];
                qNormalized[0] = q[0] / pow((pow(q[0], 2) + pow(q[1], 2) +
                   pow(q[2], 2)), 0.5);
                qNormalized[1] = q[1] / pow((pow(q[0], 2) + pow(q[1], 2) +
                   pow(q[2], 2)), 0.5);
                qNormalized[2] = q[2] / pow((pow(q[0], 2) + pow(q[1], 2) +
                   pow(q[2], 2)), 0.5);

                //A = [pl_c(:,i), - (lRr*pr_c(:,i)), q];
                cvmSet(A, 0, 0, (double)leftCameraPoint[0][i]);
                cvmSet(A, 1, 0, (double)leftCameraPoint[1][i]);
                cvmSet(A, 2, 0, (double)leftCameraPoint[2][i]);

                //R is assumed to be identity
                cvmSet(A, 0, 1, (double)(-1 * rightCameraPoint[0][i]));
                cvmSet(A, 1, 1, (double)(-1 * rightCameraPoint[1][i]));
                cvmSet(A, 2, 1, (double)(-1 * rightCameraPoint[2][i]));

                cvmSet(A, 0, 2, (double)qNormalized[0]);
                cvmSet(A, 1, 2, (double)qNormalized[1]);
                cvmSet(A, 2, 2, (double)qNormalized[2]);

                cvmSet(b, 0, 0, (double)l_ExtMat_r[0][3]);
                cvmSet(b, 1, 0, (double)l_ExtMat_r[1][3]);
                cvmSet(b, 2, 0, (double)l_ExtMat_r[2][3]);

                cvMulTransposed(A, temp, 1);   //find solution to the system
                cvInvert(temp, temp, CV_LU);
                cvTranspose(A, temp2);
                cvGEMM(temp2, temp, 1, NULL, 1, temp, 0);
                cvGEMM(temp, b, 1, NULL, 1, affineMotion, 0);

                cvmSet(lP, 0, 0, (cvmGet(affineMotion, 0, 0) * leftCameraPoint[0][i] +
                        (cvmGet(affineMotion, 2, 0) / 2) * cvmGet(A, 0, 2)));
                cvmSet(lP, 1, 0, (cvmGet(affineMotion, 0, 0) * leftCameraPoint[1][i] +
```

```
                                (cvmGet(affineMotion, 2, 0) / 2) * cvmGet(A, 1, 2)));
                        cvmSet(lP, 2, 0, (cvmGet(affineMotion, 0, 0) * leftCameraPoint[2][i] +
                                (cvmGet(affineMotion, 2, 0) / 2) * cvmGet(A, 2, 2)));

                        cvInvert(l_ExtMat_w, w_ExtMat_l, CV_LU);

                        cvGEMM(w_ExtMat_l, lP, 1, NULL, 1, worldPoint, 0);

                        wP[i].x = cvmGet(worldPoint, 0, 0);
                        wP[i].y = cvmGet(worldPoint, 1, 0);
                        wP[i].z = cvmGet(worldPoint, 2, 0);
                }
        }

        cvReleaseMat(&A);
        cvReleaseMat(&b);
        cvReleaseMat(&temp);
        cvReleaseMat(&temp2);
        cvReleaseMat(&w_ExtMat_l);

}


/*------------------------- Callback Functions -------------------------*/


//stereo callback function
void stereocallback(IplImage* image1, IplImage* image2)
{
        int i, j, N1 = MAX_FEATURES, N2 = MAX_FEATURES, yMax, yMin, xMax;

        stereoInit();   //initilaize stereo matrices

        IplImage* im1 = image1;
        IplImage* im2 = image2;

        CvSize imgsize;
        imgsize.width = im1->width;
        imgsize.height = im1->height;

        IplImage* eigImg = cvCreateImage(imgsize, IPL_DEPTH_32F, 1);
        IplImage* tempImg = cvCreateImage(imgsize, IPL_DEPTH_32F, 1);

        //change images to grayscale
        IplImage* gray1 = cvCreateImage(imgsize, 8, 1);
        IplImage* gray2 = cvCreateImage(imgsize, 8, 1);
        cvCvtColor(im1, gray1, CV_BGR2GRAY);
        cvCvtColor(im2, gray2, CV_BGR2GRAY);
        image1 = gray1;
        image2 = gray2;

        assert(gray1);
```

```
assert(gray2);

for(i = 0; i < MAX_FEATURES; i++)
{
        image1_points[i].x = -1;
        image1_points[i].y = -1;
        image2_points[i].x = -1;
        image2_points[i].y = -1;
        image1_features[i].x = -1;
        image1_features[i].y = -1;
        image2_features[i].x = -1;
        image2_features[i].y = -1;
        leftFingerPoints[i].x = -1;
        leftFingerPoints[i].y = -1;
        rightFingerPoints[i].x = -1;
        rightFingerPoints[i].y = -1;
        tipMatch[i] = -300;
}

cvGoodFeaturesToTrack(gray1, eigImg, tempImg, image1_features,
        &N1, 0.01, 0.01, NULL, 3, 0, 0.04);
cvGoodFeaturesToTrack(gray2, eigImg, tempImg, image2_features,
        &N2, 0.01, 0.01, NULL, 3, 0, 0.04);

for(i = 0; i < MAX_FEATURES; i++)
{
        //copies all image1 features to dPoints
        image1_points[i] = cvPointFrom32f(image1_features[i]);
        //copies all image2 features to dPoints
        image2_points[i] = cvPointFrom32f(image2_features[i]);
}

//filter out points near borders of image (set to -1)
removeBorders(image1_points, image2_points, imgsize);

//use SSD and PCA functions to find corresponding points
CvScalar indexColor = CV_RGB(255, 0, 0);
CvScalar elseColor = CV_RGB(0, 255, 0);

matches = 0;
tipCount = 0;

for(i = 0; i < MAX_FEATURES; i++)
{
if( image1_points[i].x != -1 && image1_points[i].y != -1)
{
        for(j = 0; j < MAX_FEATURES; j++)
        {
                //region defined by y-[yMin,yMax] and x-[0,xMax]
                yMax = image1_points[i].y + YSEARCH_SIZE;
                yMin = yMax - 2 * YSEARCH_SIZE;
                xMax = image1_points[i].x + 20;
```

```
                        if( image2_points[j].y < yMax && image2_points[j].y > yMin &&
                                image2_points[j].x != -1 )
                        {
                                FingerIndex = isFingerPCA(gray1, gray2, i, j);

                                if(FingerIndex == 1 || FingerIndex == 2 ||
                                        FingerIndex == 3 || FingerIndex == 4 ||
                                        FingerIndex == 5 || FingerIndex == 6 ||
                                        FingerIndex == 7 || FingerIndex == 8)
                                {
                                if( !isRepeatPoint(leftFingerPoints, image1_points[i]) &&
                                        !isRepeatPoint(rightFingerPoints, image2_points[j]))
                                {
                                cvCircle(im1, image1_points[i], 3, indexColor, 1);
                                cvCircle(im2, image2_points[j], 3, indexColor, 1);

                                leftPixelPoint[0][matches] = image1_points[i].x;
                                leftPixelPoint[1][matches] = image1_points[i].y;
                                rightPixelPoint[0][matches] = image2_points[j].x;
                                rightPixelPoint[1][matches] = image2_points[j].y;

                                leftFingerPoints[matches] = image1_points[i];
                                rightFingerPoints[matches] = image2_points[j];

                                tipMatch[tipCount] = matches;
                                tipCount++;
                                matches++;
                                }
                                }
                                else
                                {
                                if( !isRepeatPoint(leftFingerPoints, image1_points[i]) &&
                                        !isRepeatPoint(rightFingerPoints, image2_points[j]))
                                {
                                        cvCircle(im1, image1_points[i], 3, elseColor, 1);
                                        cvCircle(im2, image2_points[j], 3, elseColor, 1);

                                        leftPixelPoint[0][matches] = image1_points[i].x;
                                        leftPixelPoint[1][matches] = image1_points[i].y;
                                        rightPixelPoint[0][matches] = image2_points[j].x;
                                        rightPixelPoint[1][matches] = image2_points[j].y;

                                        leftFingerPoints[matches] = image1_points[i];
                                        rightFingerPoints[matches] = image2_points[j];
                                }
                                }
                        }
                }
        }
        }
```

```
	if(tipCount != -1)
	{
		stereoReconstruction(163.047, 117.303, 257.962, 224.782, lExtMatw);

		if(wP[0].x > -MATCH_THRESH && wP[0].x < MATCH_THRESH)
		{
			tipVec[0] = wP[0].x;
			tipVec[2] = wP[0].z;

			baseVec[0] = wP[0].x + 17;  //24*cos(45) ~= 17
			baseVec[2] = wP[0].z + 17;

			direction[0] = tipVec[0] - baseVec[0];
			direction[2] = tipVec[2] - baseVec[2];

			direction[0] /= pow(pow(direction[0],2)+pow(direction[2],2),0.5);
			direction[2] /= pow(pow(direction[0],2)+pow(direction[2],2),0.5);

			cout<<direction[0]<<" "<<" "<<direction[2];
			cout<<endl;

			cout<<wP[0].x<<" "<<wP[0].y<<" "<<wP[0].z;
			cout<<endl;

			if(lastPoint < 50 && wP[0].x > 50)
			{
				system("nqc -Susb -d reverse2.nqc -run");

			}
			else if(lastPoint > 50 && wP[0].x < 50)
			{
				system("nqc -Susb -d forward2.nqc -run");

			}

			lastPoint = wP[0].x;

		}
	}
}


/*----------------- Constructor and Destructor Functions ------------------*/


//constructor
CorrespondingPts::CorrespondingPts(void)
{
	int ncams = cvcamGetCamerasCount( );
	if(ncams < 2)
	{
		std::cout << "2 Cameras are needed.";
```

```
                exit(0);
        }

        cvNamedWindow("window1", CV_WINDOW_AUTOSIZE);
        cvNamedWindow("window2", CV_WINDOW_AUTOSIZE);
        HWND wnd1 = (HWND)cvGetWindowHandle("window1");
        HWND wnd2 = (HWND)cvGetWindowHandle("window2");
        cvcamSetProperty(0, CVCAM_PROP_ENABLE, CVCAMTRUE);
        cvcamSetProperty(0, CVCAM_PROP_RENDER, CVCAMTRUE);
        //cvcamGetProperty(0, CVCAM_CAMERAPROPS, NULL);
        cvcamSetProperty(0, CVCAM_STEREO_CALLBACK, stereocallback);
        cvcamSetProperty(0, CVCAM_PROP_WINDOW, &wnd1);

        cvcamSetProperty(1, CVCAM_PROP_ENABLE, CVCAMTRUE);
        cvcamSetProperty(1, CVCAM_PROP_RENDER, CVCAMTRUE);
        //cvcamGetProperty(1, CVCAM_CAMERAPROPS, NULL);
        cvcamSetProperty(1, CVCAM_STEREO_CALLBACK, stereocallback);
        cvcamSetProperty(1, CVCAM_PROP_WINDOW, &wnd2);

        cvcamInit();
        cvMoveWindow("window1", 80, 450);
        cvMoveWindow("window2", 430, 450); //510, 176);

        affineMotion = cvCreateMat(3, 1, CV_32F);
        worldPoint = cvCreateMat(3, 1, CV_32F);

        cvmSet(affineMotion, 0, 0, 0);
        cvmSet(affineMotion, 1, 0, 0);
        cvmSet(affineMotion, 2, 0, 0);
        cvmSet(worldPoint, 0, 0, 0);
        cvmSet(worldPoint, 1, 0, 0);
        cvmSet(worldPoint, 2, 0, 0);

        lExtMatw = cvCreateMat(3, 3, CV_32F);

        cvmSet(lExtMatw, 0, 0, 0.736662);//temp2.cRotw[0][0]);
        cvmSet(lExtMatw, 0, 1, -0.0154981);//temp2.cRotw[0][1]);
        cvmSet(lExtMatw, 0, 2, -0.676261);//temp2.cRotw[0][2]);
        cvmSet(lExtMatw, 1, 0, -0.0115726);//temp2.cRotw[1][0]);
        cvmSet(lExtMatw, 1, 1, 0.999877);//temp2.cRotw[1][1]);
        cvmSet(lExtMatw, 1, 2, -0.0122758);//temp2.cRotw[1][2]);
        cvmSet(lExtMatw, 2, 0, -0.676162);//temp2.cRotw[2][0]);
        cvmSet(lExtMatw, 2, 1, -0.00243494);//temp2.cRotw[2][2]);
        cvmSet(lExtMatw, 2, 2, -0.73656);//temp2.cRotw[2][2]);

        //fill basisVecs[7][PCA_SIZE*PCA_SIZE] from file
        fillBasisVecs();

}


//destructor
```

```
CorrespondingPts::~CorrespondingPts(void)
{
	int key = 0;

	while(key != 27)
	{
		if(key == 'p')
			PauseStream();
		else if( key == 'r')
			ResumeStream();

		key = cvWaitKey(100);
	}

	system("nqc -Susb -d stop.nqc -run");

	EndStream();

}


/*---------------- Camera Stream Control Functions ----------------*/


//start camera stream
void CorrespondingPts::StartStream(void)
{
	cvcamStart();

}


//pause camera stream
void CorrespondingPts::PauseStream(void)
{
	cvcamPause();

}


//resume camera stream
void CorrespondingPts::ResumeStream(void)
{
	cvcamResume();

}


//end camera stream
void CorrespondingPts::EndStream(void)
{
	cvcamStop();
```

```
        cvcamExit();

}
```

8.2 CorrespondingPts.h

```
#ifndef _CORRESPONDING_PTS_H_
#define _CORRESPONDING_PTS_H_

#define BORDERSCALE 3.0
#define YSEARCH_SIZE 30
#define NUM_FINGER_PTS 10
#define MAX_FEATURES 1
#define WINDOW_SIZE 8
#define PCA_SIZE 30
#define NUM_IMAGES 8
#define NUM_BASIS_VECS 7
#define MATCH_THRESH 400
#define PCA_THRESH 1750

#include "cvcam.h"
#include "cv.h"
#include "highgui.h"
#include "windows.h"

class CorrespondingPts
{
  public:
  CorrespondingPts(void);
    ~CorrespondingPts(void);
    void StartStream(void);
    void PauseStream(void);
    void ResumeStream(void);
    void EndStream(void);

};

#endif
```

8.3 Calibration.cpp

```
#include<iostream>
using namespace std;

#include "cvcam.h"
#include "cv.h"
#include "highgui.h"
#include "windows.h"

#include "Calibration.h"
#include "CorrespondingPts.h"
```

```
/*------------------------- Variables ----------------------------------*/


static int ptsGrabbed;
static bool click = FALSE;
static int Xp[NUM_CALIBRATION_PTS];
static int Yp[NUM_CALIBRATION_PTS];

CvScalar clickColor = CV_RGB(0, 255, 0);        //color of circle edge markers
CvScalar circleColor = CV_RGB(0, 0, 255);       //color of circle edge markers

CvCapture* capture;


/*--------------------- Callback Functions -----------------------------*/


//mouse callback function for camera calibration
void mouse_callback (int event, int x, int y, int flags, void* param)
{
       if (event == CV_EVENT_LBUTTONDOWN)
       {
               Xp[ptsGrabbed] = x;
               Yp[ptsGrabbed] = 240 - y; //must invert y-coord 240
               ptsGrabbed++;
               click = TRUE;

       }
}


/*---------------------------- Calibration Functions --------------------*/


//eliminate repeated points
bool isRepeatPointCalibration(CvPoint edges[NUM_CALIBRATION_PTS], CvPoint point)
{
       int i = 0, numRepeats = 0;
       bool repeat = false;

       for(i = 0; i < NUM_CALIBRATION_PTS; i++)
       {
               if(abs(edges[i].x - point.x) < 5 && abs(edges[i].y - point.y) < 5)
                       numRepeats++;
       }

       if(numRepeats > 1)
               repeat = true;

       return repeat;
```

```
}


//refreshes screen
void refresh(IplImage* frame, CvPoint edges[NUM_CALIBRATION_PTS])
{
	int i = 0;
	CvPoint newPoint;
	cvDestroyWindow("window");

	cvNamedWindow("window", 0);
- Show quoted text -
	cvResizeWindow("window", 640, 480);

	capture = cvCaptureFromCAM(CV_CAP_ANY);
	cvSetCaptureProperty(capture, CV_CAP_PROP_FRAME_WIDTH, 640);
	cvSetCaptureProperty(capture, CV_CAP_PROP_FRAME_HEIGHT, 480);
	cvSetMouseCallback("window", mouse_callback);

	for(i = 0; i < NUM_CALIBRATION_PTS; i++)
	{
		if(edges[i].x != -1)
		{
			cvCircle(frame, edges[i], 4, circleColor, 1);

		}
	}

	for(i = 0; i < ptsGrabbed + 1; i++)
	{
		newPoint.x = Xp[i];
		newPoint.y = Yp[i];
		cvCircle(frame, newPoint, 5, clickColor, 1);
	}

	//show image with circles and text
	cvShowImage("window", frame);

}


//find points found by edge detection nearest to point clicked
void findNearestPoint(int Xp, int Yp, CvPoint
edges[NUM_CALIBRATION_PTS], int &index)
{
	index = 0;

	int dx, dy, d, i;
	int dmatch = 1000;

	for(i = 0; i < NUM_CALIBRATION_PTS; i++)
	{
```

```
                dx = abs(edges[i].x - Xp);
                dy = abs(edges[i].y - Yp);
                d = sqrt(dx * dx + dy * dy);

                if(d < dmatch)
                {
                        dmatch = d;
                        index = i;
                }
        }
}


//calibrate the cameras
void calibrate(IplImage* imgl, double &Ox, double &Oy, double &fx, double &fy,
                         double &Tz, double &Tx, double &Ty, double &zDistance,
                         double pcRotw[3][3], double pwRotc[3][3])
{
        int i = 0;
        int j = 0;
        int flags = 0;

        double Dunit = 0;
        double Dinch = 0;
        double sigma = 1;  //sigma value from the book, can be + or - 1
        double r[3][3];

        //setup blank matrix of size 2N by 12
        CvMat* A;
        A = cvCreateMat(2 * NUM_CALIBRATION_PTS, 12, CV_32F);

        //setup SVD matrices
        CvMat* W;
        W = cvCreateMat(2 * NUM_CALIBRATION_PTS, 12, CV_32F);
        CvMat* V;
        V = cvCreateMat(12, 12, CV_32F);

        double q[12];

        //x coords of the N points
        int X[NUM_POINTS_GRABBED] = {1, 1, 1, 1, 2, 3, 4, 0, 0, 0, 0, 0, 0, 0};
        //y coords of the N points
        int Y[NUM_POINTS_GRABBED] = {1, 2, 3, 4, 4, 4, 4, 1, 2, 3, 4, 4, 4, 4};
        //z coords of the N points
        int Z[NUM_POINTS_GRABBED] = {0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 2, 3, 4};

        //setting up the matrix A
        for(i = 0; i < NUM_POINTS_GRABBED; i++)
        {
                cvmSet(A, 2 * i + 1, 0, (double)0);
                cvmSet(A, 2 * i + 1, 1, (double)0);
                cvmSet(A, 2 * i + 1, 2, (double)0);
```

```
        cvmSet(A, 2 * i + 1, 3, (double)0);
        cvmSet(A, 2 * i + 1, 4, (double)X[i]);
        cvmSet(A, 2 * i + 1, 5, (double)Y[i]);
        cvmSet(A, 2 * i + 1, 6, (double)Z[i]);
        cvmSet(A, 2 * i + 1, 7, (double)1);
        cvmSet(A, 2 * i + 1, 8, (double)(-Yp[i] * X[i]));
        cvmSet(A, 2 * i + 1, 9, (double)(-Yp[i] * Y[i]));
        cvmSet(A, 2 * i + 1, 10, (double)(-Yp[i] * Z[i]));
        cvmSet(A, 2 * i + 1, 11, (double)-Yp[i]);
        cvmSet(A, 2 * i, 0, (double)X[i]);
        cvmSet(A, 2 * i, 1, (double)Y[i]);
        cvmSet(A, 2 * i, 2, (double)Z[i]);
        cvmSet(A, 2 * i, 3, (double)1);
        cvmSet(A, 2 * i, 4, (double)0);
        cvmSet(A, 2 * i, 5, (double)0);
        cvmSet(A, 2 * i, 6, (double)0);
        cvmSet(A, 2 * i, 7, (double)0);
        cvmSet(A, 2 * i, 8, (double)(-Xp[i] * X[i]));
        cvmSet(A, 2 * i, 9, (double)(-Xp[i] * Y[i]));
        cvmSet(A, 2 * i, 10, (double)(-Xp[i] * Z[i]));
        cvmSet(A, 2 * i, 11, (double)-Xp[i]);

}

//solve with SVD
cvSVD(A, W, NULL, V, CV_SVD_MODIFY_A);

//releases unused matrices, A and W
cvReleaseMat(&A);
cvReleaseMat(&W);

for(i = 0; i < 12; i++)
{
        q[i] = cvmGet(V, i, 11);  //12 by 1 solution vector m11,m12,...,m33,m34

}

//release V
cvReleaseMat(&V);

//setup blank matrix of size 3 by 4
double M[3][4];

for(i = 0; i < 4; i++)
{
        M[0][i] = q[i];      //m11 m12 m13 m14
        M[1][i] = q[i + 4];  //m21 m22 m23 m24
        M[2][i] = q[i + 8];  //m31 m32 m33 m34

}

// scale factor
```

```
double scale = pow((pow(M[2][0], 2) + pow(M[2][1], 2) +
   pow(M[2][2], 2)), 0.5);

for(i = 0; i < 3; i++)
{
        for(j = 0; j < 4; j++)
        {
                M[i][j] = M[i][j] / scale;  //dividing M by the scale factor

        }

}

//distance D in world "units"
Dunit = abs(M[2][3]);
//distance D in inchs - each world "unit" is 1.25 inches
Dinch = abs(1.25 * Dunit);

zDistance = Dinch;

//the following are equations from the book to recover the extrinsic parameters
double q1[3] = {M[0][0], M[0][1], M[0][2]};
double q2[3] = {M[1][0], M[1][1], M[1][2]};
double q3[3] = {M[2][0], M[2][1], M[2][2]};
double q4[3] = {M[0][3], M[1][3], M[2][3]};

double q1total = 0, q2total = 0;

for(i = 0; i < 3; i++ )
{
        Ox += q1[i] * q3[i];
        Oy += q2[i] * q3[i];
        q1total += q1[i] * q1[i];
        q2total += q2[i] * q2[i];
}

fx = pow((q1total - pow(Ox, 2)), 0.5);
fy = pow((q2total - pow(Oy, 2)), 0.5);

r[0][0] = sigma * (Ox * M[2][0] - M[0][0]) / fx;
r[0][1] = sigma * (Ox * M[2][1] - M[0][1]) / fx;
r[0][2] = sigma * (Ox * M[2][2] - M[0][2]) / fx;
r[1][0] = sigma * (Oy * M[2][0] - M[1][0]) / fy;
r[1][1] = sigma * (Oy * M[2][1] - M[1][1]) / fy;
r[1][2] = sigma * (Oy * M[2][2] - M[1][2]) / fy;
r[2][0] = sigma * M[2][0];
r[2][1] = sigma * M[2][1];
r[2][2] = sigma * M[2][2];

//xaxis of world w.r.t. the camera
double xaxis[3] = {r[0][0], r[1][0], r[2][0]};
//yaxis of world w.r.t. the camera
```

```
        double yaxis[3] = {r[0][1], r[1][1], r[2][1]};
        //zaxis of world w.r.t. the camera
        double zaxis[3] = {r[0][2], r[1][2], r[2][2]};

        //z translation of world w.r.t. the camera
        Tz = sigma * M[2][3];
        //x translation of world w.r.t. the camera
        Tx = sigma * (Ox * Tz - M[0][3]) / fx;
        //y translation of world w.r.t. the camera
        Ty = sigma * (Oy * Tz - M[1][3]) / fy;

        //rotation matrix of world w.r.t. the camera
        pcRotw[0][0] = xaxis[0];
        pcRotw[0][1] = yaxis[0];
        pcRotw[0][2] = zaxis[0];
        pcRotw[1][0] = xaxis[1];
        pcRotw[1][1] = yaxis[1];
        pcRotw[1][2] = zaxis[1];
        pcRotw[2][0] = xaxis[2];
        pcRotw[2][1] = yaxis[2];
        pcRotw[2][2] = zaxis[2];

        //rotation matrix of camera w.r.t. the world
        pwRotc[0][0] = xaxis[0];
        pwRotc[0][1] = xaxis[1];
        pwRotc[0][2] = xaxis[2];
        pwRotc[1][0] = yaxis[0];
        pwRotc[1][1] = yaxis[1];
        pwRotc[1][2] = yaxis[2];
        pwRotc[2][0] = zaxis[0];
        pwRotc[2][1] = zaxis[1];
        pwRotc[2][2] = zaxis[2];

        //output results
        cout << "Rotation matrix of the world with respect to the camera:" << endl;
        cout << pcRotw[0][0] << " " << pcRotw[0][1] << " " << pcRotw[0][2] << endl;
        cout << pcRotw[1][0] << " " << pcRotw[1][1] << " " << pcRotw[1][2] << endl;
        cout << pcRotw[2][0] << " " << pcRotw[2][1] << " " << pcRotw[2][2] << endl;
        cout << "Rotation matrix of the camera with respect to the world:" << endl;
        cout << pwRotc[0][0] << " " << pwRotc[0][1] << " " << pwRotc[0][2] << endl;
        cout << pwRotc[1][0] << " " << pwRotc[1][1] << " " << pwRotc[1][2] << endl;
        cout << pwRotc[2][0] << " " << pwRotc[2][1] << " " << pwRotc[2][2] << endl;
        cout << "x translation: " << Tx << endl;
        cout << "y translation: " << Ty << endl;
        cout << "z translation: " << Tz << endl;
        cout << "Camera distance from object: " << Dinch << endl;
        cin >> i;

}


/*---------------- Constructor and Destructor Functions ----------------*/
```

```
//constructor grabs a frame and gets 14 points from mouse
Calibration::Calibration(void)
{
	int key = 0, i, index, NUM_PTS = NUM_CALIBRATION_PTS;

	//initialize ptsGrabbed counter
	ptsGrabbed = 0;
	Ox = 0;
	Oy = 0;
	fx = 0;
	fy = 0;
	Tx = 0;
	Ty = 0;
	Tz = 0;
	zDistance = 0;

	//grab a frame using cvCaputureFromCAM()
	cvNamedWindow("window", 0);
	cvResizeWindow("window", 640, 480);

	capture = cvCaptureFromCAM(CV_CAP_ANY);
	//cvcamGetProperty(0, CVCAM_CAMERAPROPS, NULL);
	cvSetCaptureProperty(capture, CV_CAP_PROP_FRAME_WIDTH, 640);
	cvSetCaptureProperty(capture, CV_CAP_PROP_FRAME_HEIGHT, 480);
	cvSetMouseCallback("window", mouse_callback);

	frame = cvQueryFrame(capture);
	cvSaveImage("calibrationFrame.jpg", frame);

	//size of frame
	CvSize imgsize;
	imgsize.width = frame->width;
	imgsize.height = frame->height;

	//change image to grayscale
	IplImage* gray = cvCreateImage(imgsize, 8, 1);
	cvCvtColor(frame, gray, CV_BGR2GRAY);
	assert(gray);

	//initialize edge detection variables
	IplImage* eigImg = cvCreateImage(imgsize, IPL_DEPTH_32F, 1);
	IplImage* tempImg = cvCreateImage(imgsize, IPL_DEPTH_32F, 1);
	CvPoint2D32f calibration_edges[NUM_CALIBRATION_PTS];
	CvPoint edges[NUM_CALIBRATION_PTS];

	for(i = 0; i < NUM_CALIBRATION_PTS; i++)
	{
		calibration_edges[i].x = -1;
		calibration_edges[i].y = -1;
		edges[i].x = -1;
```

```
                edges[i].y = -1;

        }

        cvGoodFeaturesToTrack(gray, eigImg, tempImg, calibration_edges, &NUM_PTS,
                0.01, 0.01, NULL, 3, 1, 0.1);

        for(i = 0; i < NUM_CALIBRATION_PTS; i++)
        {
                edges[i] = cvPointFrom32f(calibration_edges[i]);
        }

        for(i=0; i<NUM_CALIBRATION_PTS; i++)
        {
                if(edges[i].x != -1 && isRepeatPointCalibration(edges, edges[i]) == false)
                {
                        cvCircle(frame, edges[i], 4, circleColor, 1);
                }
                else
                {
                        edges[i].x = -1;
                        edges[i].y = -1;
                }
        }

        //show image with circles and text
        cvShowImage("window", frame);

        while(key != 27)
        {
                key = cvWaitKey(100);

                if(click == TRUE)
                {
                        findNearestPoint(Xp[ptsGrabbed - 1], Yp[ptsGrabbed - 1], edges, index);
                        Xp[ptsGrabbed - 1] = edges[index].x;
                        Yp[ptsGrabbed - 1] = edges[index].y;
                        refresh(frame, edges);
                        click = FALSE;
                }

                if(ptsGrabbed == NUM_POINTS_GRABBED)
                {
                        calibrate(frame, Ox, Oy, fx, fy, Tz, Tx, Ty, zDistance, cRotw, wRotc);
                        key = 27;
                }

        }

}
```

```
//destructor
Calibration::~Calibration(void)
{
	cvReleaseCapture(&capture);
	cvDestroyWindow("window");

}
```

8.4 Calibration.h

```
#ifndef _CALIBRATION_H_
#define _CALIBRATION_H_

#define NUM_CALIBRATION_PTS 80
#define NUM_POINTS_GRABBED 14
#define YMAX 430//240

class Calibration
{
public:
    Calibration(void);
    ~Calibration(void);

IplImage* frame;

double Ox, Oy, fx, fy, Tz, Tx, Ty, zDistance;
double cRotw[3][3], wRotc[3][3];

};

#endif
```

8.5 Picture.cpp

```
#include<iostream>
using namespace std;

#include "cvcam.h"
#include "cv.h"
#include "highgui.h"
#include "windows.h"

#include "Calibration.h"
#include "CorrespondingPts.h"
#include "Picture.h"


/*------------------------------ Variables --------------------------------*/


static bool click = FALSE;
static int Xp;
```

```
static int Yp;

CvCapture* capturePic;


/*------------------------ Picture Functions--------------------------------*/


//crops image
void cropImage(IplImage* srcframe, IplImage* dstframe)
{
       int isrc = 0, jsrc = 0, idst = 0, jdst = 0;
       int iindexsrc = 0, jindexsrc = 0, iindexdst = 0, jindexdst = 0;
       CvPoint centerpt;

       centerpt.x = Xp;
       centerpt.y = Yp;
       iindexsrc = centerpt.y - ((PCA_SIZE / 2) - 1);

       for(iindexdst = 0; iindexdst < PCA_SIZE; iindexdst++)
       {
              jindexsrc = centerpt.x - ((PCA_SIZE / 2) - 1) +
                 srcframe->widthStep * iindexsrc;

              for(jindexdst = ((dstframe->widthStep) * iindexdst);
              jindexdst < (PCA_SIZE + (dstframe->widthStep) * iindexdst); jindexdst++)
              {
                     dstframe->imageData[jindexdst] = srcframe->imageData[jindexsrc];
                     jindexsrc++;
              }

              iindexsrc++;
       }
}


//refreshes screen
void refresh(IplImage* frame)
{
       cvDestroyWindow("window");

       cvNamedWindow("window", 0);
       cvResizeWindow("window", PCA_SIZE, PCA_SIZE);

       cvShowImage("window", frame);
}


/*-------------------------- Callback Functions----------------------------*/


//mouse callback function for picture taking
```

```
void mouse_callback_pic(int event, int x, int y, int flags, void* param)
{
	if (event == CV_EVENT_LBUTTONDOWN)
	{
		Xp = x;
		Yp = 240 - y; //must invert y-coord 240

		click = TRUE;
	}

}


/*------------------- Constructor and Destructor Functions--------------------*/


//constructor grabs a frame
Picture::Picture(void)
{
	int key = 0;

	CvSize imgsizesrc;
	imgsizesrc.width = 320;
	imgsizesrc.height = 240;

	CvSize imgsizedst;
	imgsizedst.width = PCA_SIZE;
	imgsizedst.height = PCA_SIZE;

 //displacement image (in pixels)
	IplImage* finalframe = cvCreateImage(imgsizedst, 8,1);
	cvSetZero(finalframe);  //sets all elements in dx to zero

	//grab a frame using cvCaputureFromCAM()
	cvNamedWindow("window", 0);

	capturePic = cvCaptureFromCAM(CV_CAP_ANY);
	//cvcamGetProperty(0, CVCAM_CAMERAPROPS, NULL);
	cvSetCaptureProperty(capturePic, CV_CAP_PROP_FRAME_WIDTH, CV_WINDOW_AUTOSIZE);
	cvSetCaptureProperty(capturePic, CV_CAP_PROP_FRAME_HEIGHT, CV_WINDOW_AUTOSIZE);
	cvSetMouseCallback("window", mouse_callback_pic);

	frame = cvQueryFrame(capturePic);

	//change image to grayscale
	IplImage* gray = cvCreateImage(imgsizesrc, 8, 1);
	cvCvtColor(frame, gray, CV_BGR2GRAY);
	assert(gray);

	cvShowImage("window", frame);

	while(key != 27 && click == FALSE)
```

```
	{
		key = cvWaitKey(100);

	}

	cropImage(gray, finalframe);
	cvFlip(finalframe, finalframe, 0);
	cvSaveImage("trainingImage.jpg", finalframe);

	cvDestroyWindow("window");

	cvNamedWindow("window", 0);
	cvResizeWindow("window", PCA_SIZE, PCA_SIZE);

	cvShowImage("window", finalframe);

}


//destructor
Picture::~Picture(void)
{
	cvReleaseCapture(&capturePic);
	cvDestroyWindow("window");
}
```

8.6 Picture.h

```
#ifndef _PICTURE_H_
#define _PICTURE_H_

class Picture
{
public:
Picture(void);
~Picture(void);

IplImage* frame;

};

#endif
```