# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>LIST OF FIGURES</td>
<td>ii</td>
</tr>
<tr>
<td>LIST OF TABLES</td>
<td>ii</td>
</tr>
<tr>
<td>SUMMARY</td>
<td>iii</td>
</tr>
<tr>
<td>1.0 INTRODUCTION</td>
<td>1</td>
</tr>
<tr>
<td>1.1 Sign Language</td>
<td>1</td>
</tr>
<tr>
<td>1.2 Concept</td>
<td>1</td>
</tr>
<tr>
<td>1.3 Implementation</td>
<td>3</td>
</tr>
<tr>
<td>2.0 DATA ACQUISITION</td>
<td>5</td>
</tr>
<tr>
<td>3.0 DATA ANALYSIS</td>
<td>7</td>
</tr>
<tr>
<td>3.1 Point-Pattern Matching Optimization Algorithm</td>
<td>7</td>
</tr>
<tr>
<td>3.2 Range of Movement Test</td>
<td>9</td>
</tr>
<tr>
<td>4.0 USER INTERFACE</td>
<td>11</td>
</tr>
<tr>
<td>5.0 FUTURE IMPROVEMENTS</td>
<td>13</td>
</tr>
<tr>
<td>6.0 CONCLUSION</td>
<td>14</td>
</tr>
<tr>
<td>REFERENCES</td>
<td>15</td>
</tr>
<tr>
<td>APPENDIX A: LIST OF REQUIRED DRIVERS</td>
<td></td>
</tr>
<tr>
<td>APPENDIX B: MAIN PROGRAM C++ PROGRAM</td>
<td></td>
</tr>
<tr>
<td>APPENDIX C: POINT-PATTERN MATCHING PROBLEM</td>
<td></td>
</tr>
<tr>
<td>APPENDIX D: DATA ANALYSIS ALGORITHM MATLAB PROGRAM</td>
<td></td>
</tr>
<tr>
<td>APPENDIX E: USER INTERFACE C++ PROGRAM</td>
<td></td>
</tr>
<tr>
<td>APPENDIX F: PROGRESS REPORT 1</td>
<td></td>
</tr>
<tr>
<td>APPENDIX G: PROGRESS REPORT 2</td>
<td></td>
</tr>
</tbody>
</table>
LIST OF FIGURES

FIGURE 1  System Flow Chart ................................................................. 2  
FIGURE 2  Screen Shot of a User Calibration ............................................. 5  
FIGURE 3  3D Plot Showing Recorded Test Data and the Data from the Matching  
Sign for the Word “Hungry” ............................................................... 10  
FIGURE 4  Screen Shot of the ASL Translator User Interface .................... 11  
FIGURE 5  OpenNI Function Calls ............................................................ 12  

LIST OF TABLES

TABLE 1  Data Analysis Algorithm .......................................................... 7  

SUMMARY

The American Sign Language (ASL) Translator uses 3D video processing which was designed to help bridge the communication gap between the hearing impaired and the hearing able. American Sign Language is used all around the world and is the first language for many people; the ASL Translator can be used as a tool for these people as well as for those who want to learn sign language.

The ASL Translator uses the low cost XBOX Kinect as hardware and utilizes the open source software OpenNI to gather 3D data of the joints of the user. After the data is acquired for a recorded gesture, it runs through our data analysis algorithm to find a match from our 3D pre-recorded library of ASL signs. This data analysis algorithm includes 3D point-pattern matching which provides excellent accuracy. Once the gesture is matched to a library data set, the software outputs the word or phrase onto the graphic user interface. The graphic user interface provides feedback to the user and is designed using OpenGL.

The system proved to work well but improvements such as facial recognition could be made to improve comprehension of such a complex language. The Kinect is also very new and hardware and software updates will allow for a more accurate system in the future.
1.0 INTRODUCTION

1.1 Sign Language

The Canadian Association of the Deaf estimates there are nearly 300,000 deaf and upwards of 2.8 million hard of hearing Canadians [1]. This number suggests 1 in 10 Canadians are unable to communicate using spoken languages. The primary solution to this problem is for individuals to study and use a sign language. Sign languages are considered a ‘manual language’ as opposed to a spoken language using gestures and hand formations to convey thoughts. Like spoken languages, sign language uses its own grammar and rules and is not simply a translation of English into gestures. Two sign languages are predominately used in Canada, Quebec Sign Language based primarily in Quebec, and American Sign Language (ASL) which is not only the most common in Canada, but also North America.

The deaf community is a unique culture who constantly find themselves battling the misconception that being deaf or hearing impaired is a defect that must be corrected whether by medical procedures or technological devices. The definition of ‘culture’ states it is the development of the intellect through training or education [2], and we can see the deaf culture has grown into just this. Development of sign language as the first language of those who are deaf, deaf schools such as Gallaudet University, and the continual improvement in accessibility into mainstream culture indicates the deaf culture is strong and growing rapidly.

Despite this strength in the deaf culture, American Sign Language’s use amongst the hearing able is limited. Communication between the hearing able and hearing impaired or deaf commonly creates a barrier when one of the two parties is unable to communicate via sign language. Overcoming this barrier from the viewpoint of those who are unable to currently sign ASL forms the basis of our project, the American Sign Language Translator Using 3D Video Processing.

1.2 Concept

Our proposed solution to overcome this barrier was to develop a system that would allow a user to sign using American Sign Language gestures in front of a device capturing their movement and subsequently translating any recognized gesture into its prospective word or phrase.

Initial design flow is illustrated in Figure 1. Our design is segmented into three major sections; data acquisition; data analysis; and the graphic user interface.
Figure 1: System Flow Chart
The data acquisition phase begins once a user performs a gesture. Our intent is to track their motion and capture an individual movement. As movement is detected we determined the beginning and end of this motion and pass this data into our data analysis phase.

Data analysis utilizes recognition algorithms that compare current movement data against a predefined library. If a gesture is not found we return to the beginning of our flow chart, and require the user to re-perform the gesture. If the gesture is found, the translation is displayed on the user interface. The user interface provides feedback to the user illustrating through text output the translated word or phrase.

The intended application of the ASL Translator was to allow those that use American Sign Language as their first language to communicate with those who are unable to sign. A device such as this would allow them to seamlessly communicate despite the language barrier.

1.3 Implementation

Implementation of our design began with choosing a method by which we would capture motion. Real time motion capture devices are typically categorized into two types, active marker and markerless. Active marker motion capture devices are commonly used in large productions such as television, video games, and the movie industry. This variant can be identified by the use of external markers fixed onto a user and then tracked via software image processing. The markers act as reference points for the software identifying particular joints and extremities while in motion.

Markerless devices do not require users to wear any special attire. Optical sensors are typically used such as infrared to map out the depth of an environment. Infrared systems using advanced algorithms are able to extrapolate when a person has entered the scene and then identify joints and extremities.

The motion capture device we utilize in our project is the Kinect from Microsoft. The Kinect is a peripheral for the XBOX 360 also produced by Microsoft and incorporates an infrared sensor, motor, microphone and RGB video camera, all in a fairly inexpensive package at $159.99 [3]. Microsoft’s strategy with the Kinect is to tout the device as a motion capture peripheral allowing users to interact physically with games abandoning the constraints of a controller.

Interfacing the Kinect which uses USB 2.0 to a personal computer requires drivers produced by Primesense [4]. Primesense is the company behind the Kinect’s hardware and has released ‘middleware’ support drivers allowing users to access the Kinect’s internal hardware [4]. Appendix A indicates the particular drivers we used to access the rgb and infrared cameras as well as their revision status.
To interact with the Primesense drivers we used the OpenNI application programming interface (API). OpenNI released by Primesense is a collection of libraries and functions for use with natural interaction devices inclusive of the Kinect [5]. These libraries allow us to seamlessly interact with the Primesense drivers from within a Visual C++ environment.

Development of our software uses OpenNI which uses the Kinect’s infrared camera to create a three dimensional map of the environment before it. This program is able to find a user once they enter the environment using the OpenNI libraries and map a primitive skeleton connecting main joints and features as illustrated in Figure 4, a screenshot of a user being tracked. These main features, the hands, elbows, head, torso become associated with particular variables and will be further discussed in 3.0 Data Acquisition.

As movement data is accumulated, the next stage in our implementation is to analyze this data against prerecorded data from the American Sign Language library we created. Creation of our library is discussed in section 2.0 Data Acquisition. To search our library for a matching gesture we employ our data analysis algorithm as described in section 2.0. The strength of this algorithm is the point-pattern matching which attempts to match the recorded sign against each sign from our library using 3D optimization techniques.

Upon detection of a recorded gesture, the software then correlates the found gesture through a lookup table to its matching phrase or word. We then output this via text onto our graphic user interface, as discussed in section 3.0 Data Analysis.
2.0 DATA ACQUISITION

To collect data for processing, a user generator and a depth generator need to be set up using the OpenNI libraries. The depth generator is responsible for creating a depth map of the scene in front of the camera. This depth data is then processed by the user generator function to calculate what depth pixels belong to the user. This user detection is completed by watching for a group of pixels that move, as a solid object that is not a person will not move in the scene (and as such won’t be identified as a user). After user identification is complete the user is required to calibrate using the “psi pose” as shown in the screen capture of a calibration shown in Figure 2. This calibration allows for more accurate user tracking by the user generator as a user’s proportions are calculated. These calculated proportions allow the user generator to better track a user as they go through complex movements. Once this calibration is complete the user generator will begin tracking the user’s skeleton and this data can be accessed by the associated OpenNI functions. Our software allows for multiple users to be tracked provided only one is initialized at once. This convention allows background users to be picked up and ignored without interfering with the current user in the foreground who may be signing.

Figure 2: Screen Shot of a User Calibration
Our program collects all of the required skeleton body points and saves them into a data structure. Each data point has 4 values associated with it, an X, Y, and Z component as well as a confidence value. The confidence value indicates whether the user tracker has successfully tracked that body part. During this saving process several important steps are taken, first each data point is normalized to allow different sized users to use the program without error. The second step that is taken is to change the point of origin for all of the points. The head is used as the point of origin as the hands are usually in the same position in relation to the head regardless of user positioning in the scene.

After these two important steps, data for each single movement is then gathered. The beginning and end of a movement is detected by a pause from the user. For instance, to begin a sign the user is standing still and the movement is recorded once the user starts moving and the data is gathered until the user stops moving. This means that there must be pauses between different signed words and phrases.

It is required that the users most currently recorded data be converted to a fixed number of data points so that it has the same data points as the signs in the library. This conversion to a set number of points is done using a 3D interpolation function. This data is now in the correct format of a fixed number of data points in X, Y and Z and can be sent to be analyzed by the point-pattern matching algorithm.
### DATA ANALYSIS

The recorded, normalized data is passed to the data analysis algorithm in the form of a fixed number of data points of X, Y and Z. For the purposes of this project, we chose to use 100 data points. The data analysis algorithm is comprised of the steps in Table 1.

<table>
<thead>
<tr>
<th></th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>The data is compared to each sign in the library using a 3D point-pattern matching optimization algorithm</td>
</tr>
<tr>
<td>2</td>
<td>A value signifying the error of the recorded data is returned as well as the minimum and maximum X, Y and Z values of the recorded data and from the data in each sign from the library</td>
</tr>
<tr>
<td>3</td>
<td>The main program then checks the returned data for the library signs with the 2 lowest errors from the point pattern matching optimization.</td>
</tr>
<tr>
<td>4</td>
<td>If the minimum and maximum 3D values from the recorded data match those from the library sign with the lowest error, then this word is a match.</td>
</tr>
<tr>
<td>5</td>
<td>If the minimum and maximum 3D values from the recorded data do not match the minimum and maximum 3D values within a certain threshold from the library sign with the lowest error, then the second lowest error is checked as long as it is below a preset threshold.</td>
</tr>
<tr>
<td>6</td>
<td>If the minimum and maximum 3D values from the recorded data match those from the library sign with the second lowest error, then this word is a match.</td>
</tr>
<tr>
<td>7</td>
<td>If the minimum and maximum 3D values from the recorded data do not match the minimum and maximum 3D values within a certain threshold from the library sign with the second lowest error, then there is no match for the recorded data.</td>
</tr>
</tbody>
</table>

**Table 1: Data Analysis Algorithm**

### 3.1 Point-Pattern Matching Optimization Algorithm

To begin the point-pattern matching optimization algorithm, it is necessary that each sign in the library has equal number of data points as the recorded data. In our case the data acquisition software already deals with this. Point-pattern matching is what takes care of the fact that a person may never make the
same movement the same each time. Since different people may have different interpretations of signed words, we found it necessary to use an optimization technique like the 2D point-pattern matching as studied in section 9.2 of Practical Optimization: Algorithms and Engineering Applications [6]. The method in this book describes how a set of 2D data points from a database can be optimized to fit a set of 2D data points by applying scaled rotation plus a translation. Using the help of Dr. Wu-Sheng Lu and the document [7] he prepared to help solve this 3D point-pattern matching problem, we developed the point-pattern matching algorithm for 3D data. For 3D point-pattern matching, including a rotational translation is technically difficult and is not practical [7].

For each comparison of the recorded 3D data $Q = \{q_1, q_2, \ldots, q_n\}$ with the 3D data from a sign from the library $P = \{p_1, p_2, \ldots, p_n\}$ where $p_i = [p_{xi}, p_{yi}, p_{zi}]^T$, $q_i = [q_{xi}, q_{yi}, q_{zi}]^T$ and $i=1,2,3\ldots n$. In our case, the fixed number of data points, $n$, is 100. $P$ is transformed so that it approximates $Q$, this approximated pattern is called $\tilde{P}$ where $\tilde{P} = \{\tilde{p}_1, \tilde{p}_2, \ldots, \tilde{p}_n\}$ and $\tilde{p} = [\tilde{p}_{xi}, \tilde{p}_{yi}, \tilde{p}_{zi}]^T$. The approximation pattern $\tilde{P}$ is generated by applying a scaling matrix $S = \begin{bmatrix} s_1 & 0 & 0 \\ 0 & s_2 & 0 \\ 0 & 0 & s_3 \end{bmatrix}$ and a translation vector $r = [r_1, r_2, r_3]^T$. We find the approximated pattern $\tilde{p}_i = \eta p_i + r$. These parameters are minimized to approximate $P$ as close as possible and this is done by minimizing the following function $f(x) = \sum_{i=1}^{n} \|\tilde{p}_i - q_i\|^2$ with respect to $x = [s_1, s_2, s_3, r_1, r_2, r_3]^T$. The solution to the optimization problem can be seen in Appendix C.

Since Matlab has many powerful mathematical functions, we used it to model this point-pattern matching problem and implemented it in our main C++ program in the form of an executable Matlab function. This method has a small delay but proved to be sufficient for the requirements of our design. The Matlab code that was developed for the ASL Translator can be found in Appendix D.

Each call to this point-pattern matching algorithm takes the most recently recorded data and performs the point-pattern matching optimization with respect to each data set in the library of signs and phrases. The steps in Table 1 are then followed to successfully match the recorded data to a sign or phrase from the library.

Figure 3 shows a 3D plot of recorded test data and the successfully matched data from the library for the signed word “hungry”. Although the patterns look very similar, they are not an exactly the same but the point-pattern matching algorithm detects these two as a match.
3.2 Range of Movement Test

After the point-pattern matching algorithm is run the minimums and maximums from each of the top two matches are calculated and compared with the minimums and maximums from the sign that the user just made. If the minimums and maximums from the first sign match (within a preset threshold) it is accepted as being the best match for that sign and the result is printed to the screen. If the minimums and maximums of the library sign with the lowest error do not match then the second lowest error is checked and if the error is adequate (below a preset threshold) then it has its minimums and maximums checked. If it does not pass this range of movement test it is then decided that this is not a sign from the library and it is a transition between signs or a sign that is not yet in our library.
**Figure 3:** 3D Plot Showing Recorded Test Data and the Data from the Matching Sign for the Word “Hungry”
4.0 USER INTERFACE

The design of our graphic user interface is meant to aid both the user generating the sign and the person they are trying to communicate with. As Figure 4 indicates we create three viewports using the open source graphics library, OpenGL to illustrate our infrared depth map on the left, our video feedback on the right and our text output and system status on the bottom.

Figure 4: Screen Shot of the ASL Translator User Interface

The left screen is generated using the depthmapgenerator() function to initialize our infrared camera. Similarly we use the imagemapgenerator() function to initialize our RGB camera for our video feedback onto the right screen. Using the member function GetMetaData() we load the information from our respective cameras into g_depthMD, and g_imageMD. Each of these variables is then able to be accessed via the Data() function to extract raw camera data, as indicated in Figure 5.
Each set of camera data from the infrared sensor and the RGB camera is then mapped via texture map process courtesy of OpenGL. Appendix E illustrates our gl_video_output() function which shows the syntax used.

The bottom viewport builds a text output word by word, or phrase by phrase as gestures are recognized. On this screen some helpful instructions are included to aid the user in using the program. Currently a ‘Begin Sign’ command is issued when a user is free to make movement. Once the user motion has been detected and the data analysis process begins, processing time is indicated using a red progress bar. This feedback allows the recognition algorithms to calculate as well as indicates to the user they are free to move to the next sign’s start position.
5.0 FUTURE IMPROVEMENTS

Although the American Sign Language Translator is a proof of concept, there are many improvements that could be made to make it better. Since there are limitations in the accuracy of the Primesense drivers, we could not get the accuracy we desired. Although the accuracy was within 10 cm, this did not allow us to include the joints in the hands (i.e. fingers). Since finger signing is a very important part of American Sign Language, this would be a substantial improvement on our current project. If Primesense does release new drivers with more accurate data acquisition, finger joints could be implemented using the data analyzing algorithms and the open ended library. Optimally the algorithms would need to be adjusted and tested thoroughly to account for the extra functionality. Specifically, the point-pattern matching optimization may need to be designed to account for a 3D rotation of a certain amount instead of only using scalar and translational approximations. Also, the data captured and the library may need to include joint rotational angles in addition to the 3D data to account for the additional complexity of the system. This can be easily implemented using OpenNi.

American Sign Language also includes facial gestures (i.e. raised eyebrows) that could also be incorporated into the American Sign Language Translator using imaging techniques. Adding facial gestures would allow the American Sign Language Translator to be much more robust.

An American Sign Language linguist would also be needed to work on sentence concatenation and correct signing of words. There are also different dialects in different regions where American Sign Language is used and this would need to be considered while recording the signs.
6.0 CONCLUSION

Sign Language is used all around the world by the hearing impaired with the most prevalent sign language in North America being American Sign Language. There is often a barrier for those who use American Sign Language as a first language and those who don’t know it and we have tried to bridge this gap in communication by designing the American Sign Language Translator. The American Sign Language Translator captures gestures in 3D data coordinates using the low cost Microsoft Kinect hardware and utilizing OpenNI libraries to develop our software. Our program uses point-pattern matching optimization algorithms to detect signs very accurately and display them onto a computer screen in a user friendly manner.

The ASL Translator was designed to be a proof of concept and the results of the final product are very accurate. A library consisting of 15 signs and phrases was recorded and when the ASL 3D Translator was tested, the desired sign was almost always detected by the software. There are, however, improvements that can be made easily using the core concepts derived in this project.
CITED REFERENCES


OTHER REFERENCES

APPENDIX A: LIST OF REQUIRED DRIVERS

Primesense Drivers

Nite-Bin-Win32-v1.3.0.18.exe
SensorKinect-Win32-5.0.0.exe

OpenNI Libraries

OpenNI-Bin-Win32-v1.0.0.25.exe
APPENDIX B: MAIN PROGRAM C++ PROGRAM
The book [6] provides detailed information on the derivation of the 2D point-pattern matching problem and implementation findings. Using the help of Dr. Wu-Sheng Lu and the document [7] he prepared to help solve this 3D point-pattern matching problem, an overview of the mathematical solution is shown below. This solution is what was used in the ASL Translator and was implemented using Matlab. The code for the implementation can be found in Appendix C.

For each comparison of the recorded 3D data, the recorded data is

\[ Q = \{q_1, q_2, \ldots, q_n\} \] where \( q_i = [q_{xi}, q_{yi}, q_{zi}]^T \)

with the 3D data from a sign from the library being

\[ P = \{p_1, p_2, \ldots, p_n\} \] where \( p_i = [p_{xi}, p_{yi}, p_{zi}]^T \)

for \( i=1,2,3,\ldots,n \). Where \( n \) is the fixed number of data points. P is transformed so that it approximates Q, this approximated pattern is called

\[ \tilde{P} = \{\tilde{p}_1, \tilde{p}_2, \ldots, \tilde{p}_n\} \] where \( \tilde{p}_i = [\tilde{p}_{xi}, \tilde{p}_{yi}, \tilde{p}_{zi}]^T \)

The approximation pattern \( \tilde{P} \) is generated by applying a scaling matrix

\[
S = \begin{bmatrix}
s_1 & 0 & 0 \\
0 & s_2 & 0 \\
0 & 0 & s_3 \\
\end{bmatrix}
\]

and a translation vector

\[ r = [r_1, r_2, r_3]^T \]

We find the approximated pattern

\[ \tilde{p}_i = Sp_i + r \quad \text{for} \quad i=1,2,\ldots,n \]

These parameters are minimized to approximate P as close as possible and this is done by minimizing the following function

\[
f(\lambda) = \sum_{i=1}^{n} \|\tilde{p}_i - q_i\|^2
\]
with respect to the parameter vector

\[ x = [s_1, s_2, s_3, r_1, r_2, r_3]^T \]

The solution to the objective function is

\[ f(x) = \sum_{i=1}^{n} \| \hat{p}_i - q_i \|^2 = \sum_{i=1}^{n} \| Sp_i + r - q_i \|^2 \]

\[ = \sum_{i=1}^{n} \| (P_i I_3)x - q_i \|^2 = x^T H x - 2x^T b + \kappa \]

where

\[ P_i = \begin{bmatrix} p_{xi} & 0 & 0 \\ 0 & p_{yi} & 0 \\ 0 & 0 & p_{zi} \end{bmatrix}, I_3 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \]

and \( H = \begin{bmatrix} \sum_{i=1}^{n} P_i^2 & \sum_{i=1}^{n} P_i \\ \sum_{i=1}^{n} P_i & nI_3 \end{bmatrix}, b = \begin{bmatrix} \sum_{i=1}^{n} P_i q_i \\ \sum_{i=1}^{n} q_i \end{bmatrix} \) and \( \kappa = \sum_{i=1}^{n} \| q_i \|^2 \)

where the Hessian (H) is positive definite and therefore the objective function \( f(x) \) is globally strictly convex and therefore has a unique global minimize \( x^* \).

We minimize \( f(x) \) to find the design parameter minimizer to be

\[ x^* = H^{-1} b \]

and using the minimizer in the objective function, we obtain the “error” value used in our analysis algorithm.
APPENDIX D: DATA ANALYSIS ALGORITHM MATLAB PROGRAM

```matlab
function [H] = getData(p_filename)
%This function returns the Hessian of the passed csv library file

numDataPoints = 100; %number of data points
sum_pi_squared = zeros(3,3);
sum_pi = zeros(3,3);
P = zeros(3,numFrames);
I = numFrames * eye(3,3); %I3 is a 3x3 identity matrix

P = csvread(p_filename);
P = P';

for i=1:numDataPoints
    d = [P(1,i),P(2,i),P(3,i)];
    sum_pi_squared = sum_pi_squared + diag(d)^2;
    sum_pi = sum_pi + diag(d);
end

H = [sum_pi_squared,sum_pi;sum_pi,I];
```

```matlab
function [error, min_q, max_q] = matchPattern(p_filename, q_filename)
%This function solves the point-pattern matching problem for the
%csv files input to the function

H = getData(p_filename); %get hessian from library data
q = csvread(q_filename); %recorded test data
p = csvread(p_filename); %library data

%delete the 100th row
q(100,:) = [];
```
p(100,:) = [];

% find min and max of test data to be used to find matched word
max_q = max(q);
min_q = min(q);

% transpose matricies
p = p';
q = q';

%H is the predetermined hessian 6x6 matrix
%q is the data recorded in 3x100

numDataPoints = 99; % number of data points of data files

sum_pi_qi = zeros(3,1);
sum_qi = zeros(3,1);
k = 0;

for i=1:numDataPoints

% sum individual p*q elements of entire matrix
sum_pi_qi(1,1) = sum_pi_qi(1,1) + p(1,i)*q(1,i);
sum_pi_qi(2,1) = sum_pi_qi(2,1) + p(2,i)*q(2,i);
sum_pi_qi(3,1) = sum_pi_qi(3,1) + p(3,i)*q(3,i);

% sum individual q elements of entire matrix
sum_qi(1,1) = sum_qi(1,1) + q(1,i);
sum_qi(2,1) = sum_qi(2,1) + q(2,i);
sum_qi(3,1) = sum_qi(3,1) + q(3,i);

k = k + norm([q(1,i);q(2,i);q(3,i)])^2;
end
b = [sum_pi_qi(1,1);sum_pi_qi(2,1);sum_pi_qi(3,1);sum_qi(1,1);sum_qi(2,1);sum_qi(3,1)];

x = inv(H)*b;

error = x'*H*x-2*x*b+k;

function FindMatch()
% This function utilizes the matchPattern.m function to test the recorded
% data against each data set from the library
	numRecordedGestures = 20; % Number of gestures in the library

H = zeros(numRecordedGestures,8); % H=[p_filename,error,min_x,min_y,min_z,max_x,max_y,max_z] x files

q_filename = 'Test\Test.txt';

% Go through the library of signs and compare each one to the recorded data
for i=1:numRecordedGestures
    str = strcat('Test\', int2str(i), '.txt');
    [H(i),minQ(i,:),maxQ(i,:)] = matchPattern(str,q_filename);
end

[H_sort,H_index] = sort(H); % Sort the results

H_data_out(:,1) = H_index(:,1);
H_data_out(:,2) = H_sort(:,1);

% Concatenate results to be put into a csv file
DataOut = horzcat(H_data_out, minQ, maxQ);

format shortE;
% write to a csv file to be read by main program
csvwrite('Test\BestMatchoutput.txt', DataOut);
APPENDIX E: USER INTERFACE C++ PROGRAM

void gl_video_output()
{
    XnStatus rc = XN_STATUS_OK;
    // Read a new frame
    rc = g_Context.WaitAnyUpdateAll();
    if (rc != XN_STATUS_OK)
    {
        printf("Read failed: %s\n", xnGetStatusString(rc));
        return;
    }
    float g_pDepthHist[MAX_DEPTH];

    rc = g_Context.FindExistingNode(XN_NODE_TYPE_DEPTH, g_DepthGenerator);
    rc = g_Context.FindExistingNode(XN_NODE_TYPE_IMAGE, g_ImageGenerator);
    g_DepthGenerator.GetMetaData(g_depthMD);
    g_ImageGenerator.GetMetaData(g_imageMD);
    const XnDepthPixel* pDepth = g_depthMD.Data();
    const XnUInt8* pImage = g_imageMD.Data();
    unsigned int nImageScale = GL_WIN_SIZE_X / g_depthMD.FullXRes();
    // Texture map init
    g_nTexMapX = (((unsigned short)(g_depthMD.FullXRes() - 1) / 512) + 1) * 512;
    g_nTexMapY = (((unsigned short)(g_depthMD.FullYRes() - 1) / 512) + 1) * 512;
    g_pTexMap = (XnRGB24Pixel*)malloc(g_nTexMapX * g_nTexMapY * sizeof(XnRGB24Pixel));

    // Texture map init
    xnOSMemSet(g_pTexMap, 0, g_nTexMapX * g_nTexMapY * sizeof(XnRGB24Pixel));
    const XnRGB24Pixel* pImageRow = g_imageMD.RGB24Data();
    XnRGB24Pixel* pTexRow = g_pTexMap + g_imageMD.YOffset() * g_nTexMapY;
    for (XnUInt y = 0; y < g_imageMD.YRes(); ++y)
    {
        const XnRGB24Pixel* pImage = pImageRow;
        XnRGB24Pixel* pTex = pTexRow + g_imageMD.XOffset();
        for (XnUInt x = 0; x < g_imageMD.XRes(); ++x, ++pImage, ++pTex)
        {
            *pTex = *pImage;
        }
        pImageRow += g_imageMD.XRes();
    }
}
pTexRow += g_nTexMapX;
}

// Create the OpenGL texture map
glTexParameteri(GL_TEXTURE_2D, GL_GENERATE_MIPMAP_SGIS, GL_TRUE);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR_MIPMAP_LINEAR);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, g_nTexMapX, g_nTexMapY, 0, GL_RGB, GL_UNSIGNED_BYTE, g_pTexMap);

// Display the OpenGL texture map
setColor4f(1.1,1.1,1.1);
begin(GL_QUADS);
int nXRes = g_imageMD.FullXRes();
int nYRes = g_imageMD.FullYRes();
//map texture output to correct window size/viewport
// bottom left
TexCoord2f(0, 0);
Vertex2f(0, GL_WIN_SIZE_Y);
// bottom right
TexCoord2f((float)nXRes/(float)g_nTexMapX, 0);
Vertex2f(GL_WIN_SIZE_X, GL_WIN_SIZE_Y);
// upper right
TexCoord2f((float)nXRes/(float)g_nTexMapX, (float)nYRes/(float)g_nTexMapY);
Vertex2f(GL_WIN_SIZE_X, 0);
// upper left
TexCoord2f(0, (float)nYRes/(float)g_nTexMapY);
Vertex2f(0,0);
free(g_pTexMap);
disable(GL_TEXTURE_2D);
disable(GL_BLEND);
end();
}
APPENDIX F: PROGRESS REPORT 1

Communication barriers between the hard of hearing or deaf, and hearing able exist in countless situations. Solutions such as hearing aids and sign language are more and more prevalent in today’s world. The popularity of American Sign Language in particular rivals many spoken languages in North America, however like spoken languages a mutual understanding between the two parties is needed in order to fluently communicate.

Our 499 Design Project is the American Sign Language Translator Using 3D Video Processing. Using the Microsoft Kinect we will visually interpret the gestures and motions that make up a select group of the American Sign Language words and output text via on screen application. This product would allow an individual using sign language to communicate with any one or a group of people whom are not fluent in sign language.

Development will see the utilization of the Kinect’s three dimensional video processing in conjunction with our own software. We will be using the Open NI and NITE software development tools to aid in our design. Our goal is use this software to track a user’s movements then process and convert them on the fly to their respective meanings.

Timetable:

<table>
<thead>
<tr>
<th>Date</th>
<th>Tasks &amp; Goals</th>
</tr>
</thead>
</table>
| Jan 21 – Jan 28 | - **Progress Report #1**  
- Research OpenNI & NITE open source code  
- Setup Google Code for project control  
- Have skeletal tracking and coordinate output complete |
| Jan 28 – Feb 4   | - Recording individual word data points separated by pauses  
- Complete website |
| Feb 4 – Feb 11   | - Begin recognition algorithms  
- Recognize at least one word |
| Feb 18 – Feb 25  | - **Midterm Review/ Progress Report #2**  
- Begin logging word/phrase data points |
| Feb 25 – Mar 4   | - Complete recognition of 20 words/phrases  
- Begin integration into graphic user interface |
| Mar 4 – Mar 11   | - Conclude integration into graphic user interface  
- Complete testing and debugging of project |
| Mar 11 – Mar 18  | - Complete final report & poster presentation |
| Mar 18 – Mar 25  | - **Presentation** |

Progress:

Currently our team has purchased a Microsoft Kinect. Driver interface from the PC to the device has been completed.

Using the NITE software libraries we are able to access the Kinect’s peripherals to draw a skeletal representation of a user in an OpenGL environment.
APPENDIX G: PROGRESS REPORT 2

1.0 Summary

Communication barriers between the hard of hearing or deaf, and hearing able exist in countless situations. Solutions such as hearing aids and sign language are more and more prevalent in today’s world. The popularity of American Sign Language in particular rivals many spoken languages in North America, however like spoken languages a mutual understanding between the two parties is needed in order to fluently communicate.

Our 499 Design Project is the American Sign Language Translator Using 3D Video Processing. Using the Microsoft Kinect we will visually interpret the gestures and motions that make up a select group of the American Sign Language words and output text via on screen application. This product would allow an individual using sign language to communicate with any one or a group of people whom are not fluent in sign language.

Development will see the utilization of the Kinect’s three dimensional video processing in conjunction with our own software. We will be using the Open NI and NITE software development tools to aid in our design. Our goal is use this software to track a user’s movements then process and convert them on the fly to their respective meanings. Figure 1.0 illustrates our basic design from user input to output graphic user interface.

The biggest problems we faced were:

- Getting familiar with the NITE libraries and integrating the NITE functions into our program.
- Creating a library of pre-recorded movements, gestures and phrases as determined by American Sign Language.
- Filtering our pre-recorded data in comparison to our current user positional data to define when a particular movement has been committed by a user.
- Creating a normalization algorithm to ensure users of all dimensions can be analyzed with a single library of data.

![Figure 1.0](image-url)
2.0 Components

2.1 Data Output

To use the data output by the Kinect we first had to learn the NITE and Open NI libraries. There are many components to these libraries and they are very complex. NITE has sample programs that we went through to help us learn the libraries and functions in the libraries. Using NITE, the data is collected and grouped by joints using the XnSkeletonJointPostion data type (Appendix A). The data type includes positional data for the X, Y and Z coordinates, all measured in millimetres with respect to a predetermined origin, in our case the head.

We had a problem with keeping the data collected organized so we created a custom class upperbody_t (Appendix B) that gathered only the upper torso joint positions relevant to our project. When the program starts, and a new body is found, the necessary joint data becomes populated within the upperbody_t class. This data needed to be stored in an efficient way so we decided to write it to a CSV (Comma-Separated Values) file. From the time a movement starts, each frame (each time data is collected) the upperbody_t joint data is written to the CSV file until the movement finishes. This is generally about seven to ten frames.

The CSV file created acts as the basis for which our filtering needs be applied to.

2.2 Filtering:

The second major component of our project is taking our current positional data and comprising a filtering technique to compare the XYZ coordinates to our pre-recorded values within the CSV file. As described earlier our CSV file holds single movement vectors comprised of XYZ coordinates over a span of seven to ten frames.

Speed of our sorting and analyzing is a high priority. To ensure optimal speed we utilize a bubble sort algorithm to organize our CSV libraries using the pre-recorded total combined range of the XYZ vectors in comparison to the total combined range of our current movement. This allows us to quickly sort our CSV files before ever opening one to read data.
Filtering our data within the CSV file is encapsulated within the `findSignName()` function (Appendix C). We use this function to read in our pre-recorded sign data and isolate what particular movement a user has just committed.

We’re able to find this information using three steps:

1. Check relevant body parts
2. Calculate positional range
3. Calculate hit/miss ratio

The first body part check is found in the preliminary data of each CSV file. By setting flags for `checkHead`, `checkNeck`, `checkLeftShoulder` etc, we can quickly decide which body part data is relevant during a certain movement. There would be no need to analyze data in regards to foot position if our hand waving CSV file was currently being checked.

Once we know which body parts to check we proceed with reading our CSV’s positional data for each given frame. Using nested loops we then compare current position data for each relevant body part with respect to our recorded data. Using the `XNSkeletonJointSubtract` function we are able to pass our recorded and current positions and have the difference returned. We compare this difference with a sensitivity value we have predefined. This operation takes place at each vertex for which we have positional data. Upon comparison if we are within our sensitivity range we increment a hit count, where for every miss we increment a miss count.

The final step is to assess our hit/miss ratio and determine if the current CSV file’s data is in fact the movement the user has just committed. A predefined hit/miss ratio value of 90% is our current threshold.

### 2.3 Normalization

The problem of normalization made itself evident during testing of our sign recognition. Kylee who is 5’4” served as our recorder of functions. During testing her movements were quickly recognized with a high hit ratio. However during testing of others who were taller, recognition of movements illustrated a high miss ratio. Trouble shooting found this error resulted from the difference in length of limbs and overall dimensions of the user.

In order to resolve this issue a normalization function will need to be written to allow users of all sizes to be down or up-sized according to a particular sizing ratio. This ratio will result in all user dimensions averaging out allowing for use with a single pre-recorded library as opposed to creating libraries for different sized users.
Development of this function is currently in work.

### 3.0 Timetable

<table>
<thead>
<tr>
<th>Date</th>
<th>Tasks &amp; Goals</th>
</tr>
</thead>
</table>
| Feb 21 – Feb 25 | - Get website finished  
                  | - Begin GUI using Visual C++  
                  | - Begin adding signs to library of recorded signs                  |
| Feb 25 – Mar 4 | - Begin poster, presentation and report  
                  | - Create final library of recorded signs  
                  | - Start integration with GUI and main program                      |
| Mar 4 – Mar 11 | - Conclude integration into graphic user interface                |
| Mar 11 – Mar 18 | - Complete testing and debugging of project                       |
| Mar 18 – Mar 25 | - Complete final report & poster presentation                    |
|              | - Presentation                                                    |
4.0 Appendices

4.1 Appendix A, XnSkeletonJoint Position Data Type

typedef struct XnSkeletonJointPosition
{
    /** The actual position in real world coordinations */
    XnVector3D position;

    /** The confidence in the position */
    XnConfidence fConfidence;
} XnSkeletonJointPosition;

4.2 Upperbody_t Class

class upperbody_t{
    public:
        XnSkeletonJointPosition getJoint(XnSkeletonJoint joint){
            return Data[((int)joint - 1)];
        }
        void setJoint(XnSkeletonJoint joint,
                      XnSkeletonJointPosition jointData){ Data[((int)joint - 1)] = jointData; }

        upperbody_t operator - (upperbody_t);
        void useTorsoAsOrigin();
        void calculateAngle(XnSkeletonJoint joint);
        upperbody_t();
        ~upperbody_t();

        jointAngle_t getJointAngle(XnSkeletonJoint joint){
            return JointAngle[((int)joint - 1)];
        }

    private:
        XnSkeletonJointPosition Data[16];
        jointAngle_t JointAngle[16];
};