A Novel Video Game Peripheral for Detecting Fine Hand Motion and Providing Haptic Feedback

by

Samantha N. Powers

Submitted to the Department of Mechanical Engineering in partial fulfillment of the requirements for the degree of Bachelor of Science in Mechanical Engineering

and

Lauren K. Gust

Submitted to the Department of Mechanical Engineering in partial fulfillment of the requirements for the degree of Bachelor of Science in Engineering as Recommended by the Department of Mechanical Engineering

at the

Massachusetts Institute of Technology

June 2012

Signatures of Authors ...[illegible]...
Department of Mechanical Engineering
May 25, 2012

Certified by ...[illegible]...
David Wallace
Professor of Mechanical Engineering
Thesis Supervisor

Accepted by ...[illegible]...
John H. Lienhard V
Samuel C. Collins Professor of Mechanical Engineeering
Undergraduate Officer

A Novel Video Game Peripheral for Detecting Fine Hand Motion and Providing Haptic Feedback

by

Lauren K. Gust

Submitted to the Department of Mechanical Engineering
on May 25, 2012, in partial fulfillment of the
requirements for the degree of
Bachelor of Science in Engineering as Recommended by the Department of
Mechanical Engineering

and

Samantha N. Powers

Submitted to the Department of Mechanical Engineering
on May 25, 2012, in partial fulfillment of the
requirements for the degree of
Bachelor of Science in Mechanical Engineering

Abstract

This thesis documents the design and implementation of a game controller glove that employs optical tracking technology to detect movement of the hand and fingers. The vision algorithm captures an image from a webcam in real-time and determines the centroids of colored sections on a glove worn by the player; assigning a distinctive identifier for each section which is associated with a 3D model retrieved from a pre-existing library. A Vivitouch artificial muscle module is also mounted to the top of the glove to provide vibratory haptic feedback to the user. The system has been user tested and a number of potential use scenarios have been conceived for integration of the controller in various gaming applications.

Thesis Supervisor: David R. Wallace
Title: Professor

Acknowledgments

We would like to thank a number of people who have provided invaluable assistance in making this thesis possible: our advisor David Wallace for his advice and patience, Artificial Muscle for providing us with the Vivitouch components, Bradley Abruzzi for his NDA assistance, and the many friends and family who have kept us sane these four years.

Contents

List of Figures

Chapter 1

Introduction

In recent years, a number of game controllers have been developed which are based on player movements, for example: the Nintendo Wii, the PlayStation Move, and the Xbox Kinect [1, 2, 3]. These control schemes are offered as an alternative to the more traditional analog controller which presents an array of buttons. Both traditional and motion-based controllers have a number of issues inherent in their design, providing motivation for the development of a new controller capable of detecting fine movements of the hand and fingers and providing improved haptic feedback based on player actions.

The proposed controller will accomplish these aims by utilizing a colored glove which is placed on the player's hand (Figure 1-1).

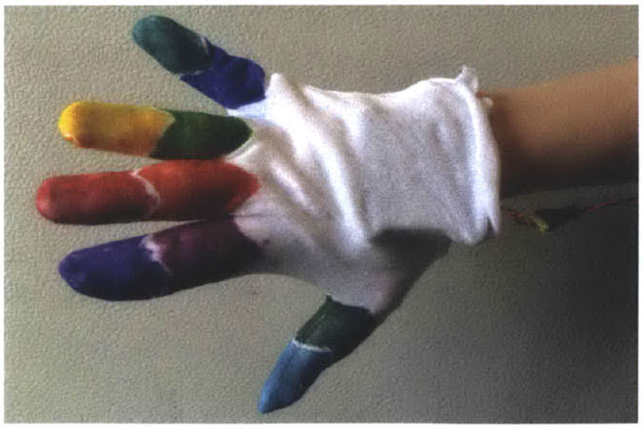


Figure 1-1: The controller prototype

When the player moves his gloved hand in front of a standard webcam, the image is captured and processed by a vision algorithm in real-time which determines the position of these colored patches and retrieves a three-dimensional model of the gesture from a library (Figure 1-2).

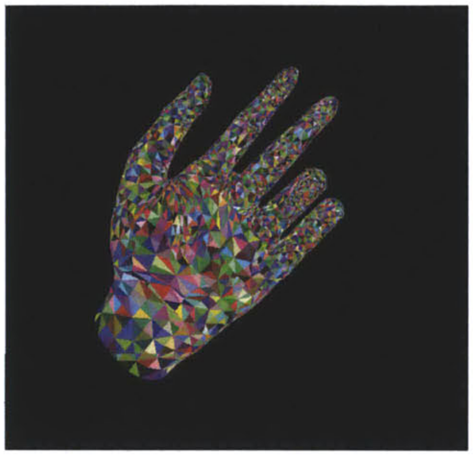


Figure 1-2: A sample image from the pose library

Haptic feedback is accomplished by placing a Vivitouch haptics component on the back of the glove. The Vivitouch acts like an artificial muscle and is able to vibrate at different frequencies with a fast response time, producing complex and realistic haptic responses to in-game actions [4].

1.1 Thesis Outline

Chapter two will focus on analog and motion-based game controllers and the technologies they utilize. Issues associated with these controllers will also be discussed.

Chapter three describes the design of the controller, the vision processing algorithm it utilizes to identify movements and gestures, and the implementation of a haptic module.

Chapter four describes how the product has been tested by means of 3D hand manipulation and how this controller may be integrated into possible gaming scenarios.

Chapter 2

Current controller schemes

2.1 Conventional controllers

Conventional analog game controllers consist of a handheld device with an array of buttons and joysticks which are used to control character actions in the game. Button actions are dependent on the game being played and are often customizable through game menus.

Analog controllers have been adapted to incorporate other body motions by translating the basic button pressing functionality into new form factors. The game Dance Dance Revolution incorporates analog controls into a pad which is placed on the floor [5]. The player then uses his feet to step on arrows in time to visual and audio cues in the game. The Rock Band game series developed by Harmonix integrates analog buttons into controllers designed to look like musical instruments [6].

2.2 Acceleration-based controllers

The Nintendo Wii console is the first commercial game controller to employ an accelerometer for control [7]. The Wii remote is able to sense three axes of acceleration using the ADXL330 accelerometer[8]. The system also uses a PixArt optical sensor to determine where the remote is pointing. The remote senses infrared light from a Sensor Bar and calculates the distance between the remote and the Sensor Bar using triangulation [9, 10]. The Wii remote still incorporates analog buttons for actions and selections and provides basic haptic feedback through the use of a rumble pack in the controller.

The PlayStation Move is a motion-sensing controller used with the Sony PlayStation [2]. A glowing orb at the top of the controller serves as an active marker that is tracked by the PlayStation Eye camera. The orbs uniform spherical shape and known size allow the system to dynamically track the controllers position and distance from the camera, resulting in precise three-dimensional motion tracking. The controllers internal sensors include two inertial sensors, a three-axis linear accelerometer and a three-axis gyroscopic sensor that are used to track rotation as well as overall motion. An internal magnetometer is employed to calibrate the controller against the Earths magnetic field to help correct against sensor drift [11]. Like the Wii remote, the Move incorporates analog buttons in addition to motion tracking technology. Additionally, the orb at the top of the Move controller glows using RGB light-emitting diodes (LEDs). These act as a three-dimensional pixel, providing visual feedback to the player as he manipulates the controller.

2.3 Full motion controllers

The Microsoft Xbox Kinect is the first full body motion sensing input device [3]. The Kinect sensor is connected to a base with a motorized pivot and is designed to be placed above or below the video display. The device includes 3D depth sensors, an RGB camera and a microphone for motion and sound detection [12]. The depth sensor employs an infrared laser projector and a monochrome CMOS senor to capture 3D video data. Using proprietary software, the device is able to track two active players for motion analysis with feature extraction of 20 joints per player [13].

2.4 Issues with motion-based controllers

Both the Wii remote and PlayStation Move suffer from the same drawback—namely, they are only able to track a single point through 3D space. While the Kinect is able to track full body motions, its resolution is not sufficient to detect finer movements; for example, the motion of individual fingers. Furthermore, the absence of any body-mounted peripherals in the Kinect system means that feedback given by the system is purely audio-visual without haptic feedback.

From a product design standpoint, current handheld controller models can be improved. The Wii remote (Figure 2-1) has a square profile with beveled edges which can dig into the palm of the hand over time and does not accommodate different grip styles.

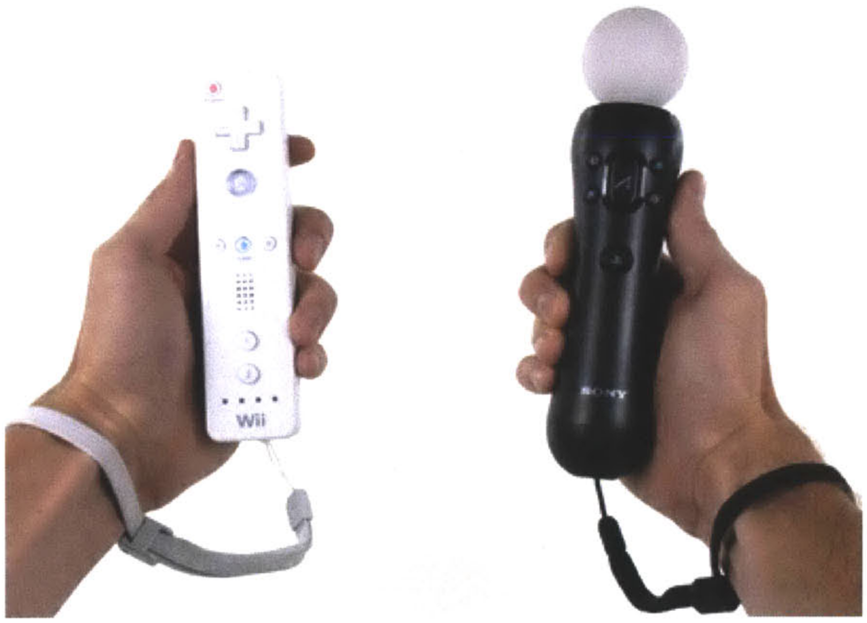


Figure 2-1: Comparison of Wii remote (left) and PlayStation Move (right) [14]

Buttons are placed in such a way that the entire hand must be moved to press the lower buttons. Inadvertent button presses are also an issue when the remote is held in the players hand. This design choice was likely made to allow the remote to be used in a sideways orientation that mimics the layout of older Nintendo game controllers, though overall results in a weaker design.

The PlayStation Move (Figure 2-1) has a rounded design that is better for accommodating different grip styles. The button configuration does not require significant grip adjustments, however buttons placed on the side of the controller are still prone to inadvertent selection due to grip. However, the Moves product form has caused many users to draw visual comparisons between it and objects like a lollipop or a massage vibrator [15].

Chapter 3

Designing a hand tracking system

3.1 Previous work

Researchers at the Massachusetts Institute of Technology Computer Science and Artificial Intelligence Laboratory (CSAIL) have developed a user-input device that captures unrestricted motion of a hand as well as the individual articulation of the fingers [16]. This system determines the movement of a hand wearing a colored glove by utilizing a single camera.

Their approach utilizes a database of hand poses, each of which is indexed by a rasterized image of the pose. Given a query image from the camera, the correct pose is estimated by searching the database of these index images. To determine the nearest neighbor to this image, the query image and a database image are compared by computing the divergence from the database to the query and from the query to the database. The average of these two distances is computed to obtain a symmetric distance.

While this method can produce the approximate hand pose, it cannot account for the distance of the gloved hand to the camera. To address this limitation, the work in this thesis adds 2D projection constraints in association with each database image for the centroids of each color patch. By transforming the projection constraints into the coordinate space of the original query image, the global hand position is obtained.

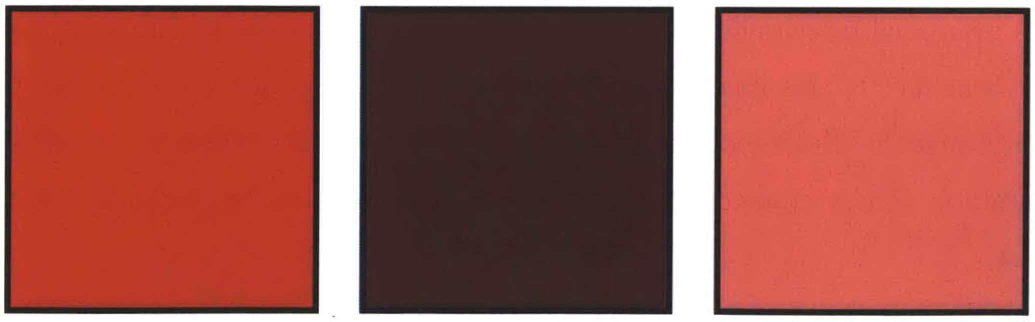
3.2 Algorithm design considerations

The purpose of the proposed game controller is to be used in games. This implies three things: it needs to be robust to a wide variety of situations and players, it needs to be readily incorporated into games by game developers, and it needs to be fast enough that other game processing can occur simultaneously.

3.2.1 Robustness

Robustness in vision systems is primarily threatened by changing lighting conditions, which causes colors to change greatly with both location and time of day.

Images are typically read from cameras in the RGB colorspace, as cameras are typically equipped with an array of three sensor types, one of which detects red, one green, and the last blue. This colorspace is, unfortunately, not very robust to changing light conditions. Figure 3-1 shows an example of this. Figure 3-1a shows a pure red, which has an RGB value of (255, 0, 0). Figure 3-1b shows a pure red with decreased lightness (as though it were less well-lit), which has an RGB value of (128, 0, 0). Finally, Figure 3-1c shows a pure red with a decreased saturation (same color, but less intensely colored), which has an RGB value of (255, 128, 128). RGB values change dramatically for the same color in various lighting conditions, which increases the amount of analysis necessary to match colors to a saved palette.



(a) RGB (225, 0, 0) (b) RGB (128, 0, 0) (c) RGB (255, 128, 128)

Figure 3-1: Example of RGB colorspace values: (a) baseline red, (b) darker red, (c) less saturated red

However, there are a variety of transformations that can be done to RGB colors to make them more suitable for image processing. The hand-tracking algorithm described in Section 3.1 uses the Chong colorspace, but this paper opts for the more commonly used HSV colorspace. In HSV the color wavelength, or "hue", is represented by one number, which typically ranges between 0 and 360, which is based on the color circle. For example, red has a hue of 0, blue 240, green 100, etc. The other two dimensions of color in HSV are the saturation and the value. Saturation represents how intense a color is, and value is related to how dark a color is. The color in Figure 3-1a has an HSV representation of (0, 100, 100); 3-1b (0, 100, 50); 3-1c (0, 50, 100). Important to note is the fact that all three have the same hue.

Furthermore, color-based vision systems also have the difficulty of needing to be robust to the colors of the environment. A user with a brightly colored shirt or a vibrant mural should ideally not impact the performance of the program. This algorithm's solution to this problem is described below.

3.2.2 Game industry compliant

To be feasibly used by game developers, the libraries developed need to be written in a language that may be easily incorporated into existing development structures. Games are generally developed using a game engine, which is essentially a library of premade functions that aid event scripting, collision physics, state transitions, character manipulation and animation, etc [17]. A large number of these engines are written in C++, or scripted in C derivatives (Lua). C++ is so widespread in game development that it has been called the industry standard [18]. Therefore it was decided to program the libraries for this gaming peripheral in C++.

3.2.3 Speed

C++ is so common in the gaming industry at least partially because of its speed. However, some games can be quite processor intensive, and it would be best if the use of the glove as a peripheral caused as minimal an impact on computation power as possible. Thus the image algorithm used is as minimalistic as possible, sacrificing the accuracy afforded by predictive algorithms for the speed of a more direct solution.

3.3 Algorithm design

Put simply, the algorithm first determines the centroids of the colors in the image. Second, it assigns a distinctive identifier to an image captured by the webcam based on these centroids. Finally, it compares this identifier to a library of existing images, each of which is associated with a 3D model.

3.3.1 Determining the centroids

During processing, the image captured by the webcam is iterated through twice. The pixels are processed starting in the top left and proceeding right and downwards. The key challenge being solved by this part of the algorithm is identifying and segmenting colors into clusters of pixels, called "Connected Components". The centroids of these blobs of color can then be found and processed.

During the first pass, the color at the current pixel is converted to the HSV colorspace. The hue is then compared to the list of colors known to be on the glove. If it is found to be a relevant color, the pixel above and the pixel to the left are both checked. The result of this check can be separated into three cases. The first is if neither the top pixel or the left pixel is the same color as the current pixel, in which case the current pixel is assigned a new number, its Connected Components number. The second case occurs when exactly one of those pixels is found to be the same color as the current pixel, in which case the current pixel is assigned the Connected Components number of that pixel. These two cases are shown for the first row of a sample image in Figure 3-2.

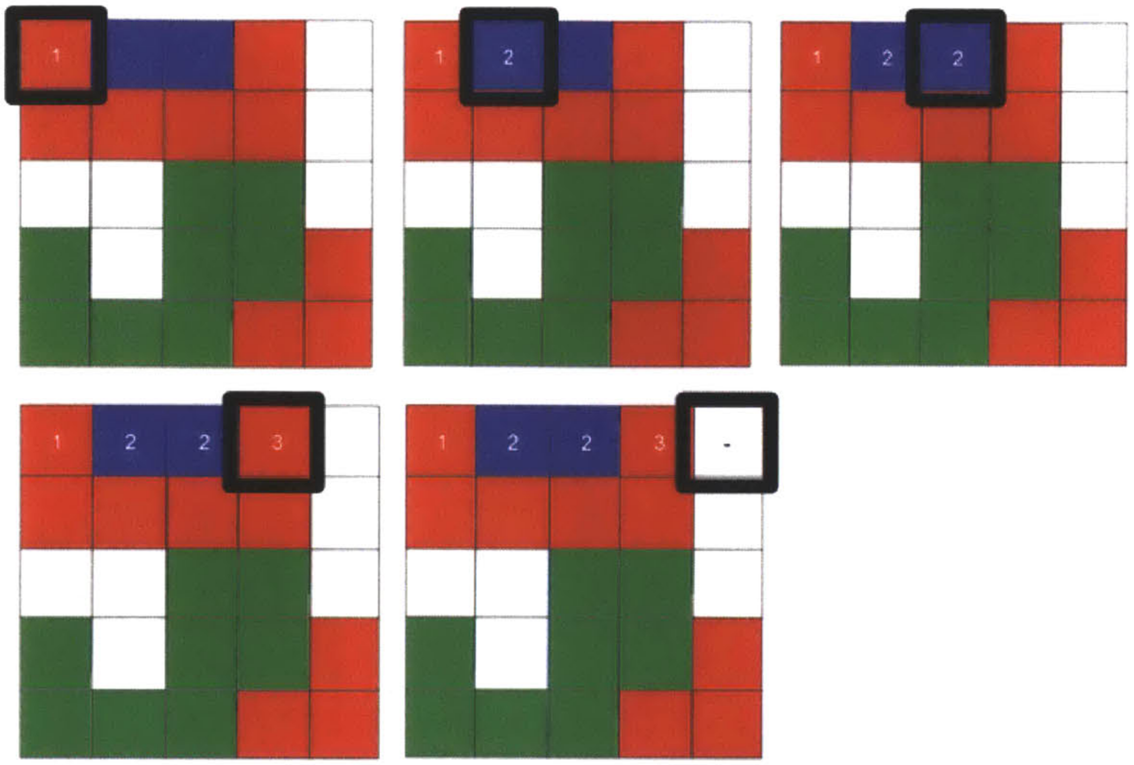


Figure 3-2: The pixels of the first row are assigned Connected Components numbers. The second 2 is assigned because the pixel to its left is the same color, and had already been assigned a number. The white block is not a color of interest, so it is not assigned a number at all

The third case occurs when both the pixels to the top and to the left are identified as being the same color as the current pixel, but have been assigned different numbers, as illustrated in Figure 3-3. In this case the two numbers are saved as being equivalent, and the current pixel is assigned the lower number.

The process is then continued, giving the result after the first pass as shown in Figure 3-4. Notice that each connected component has not necessarily been assigned the same number across all its pixels, yet.

During the second pass, the Connected Component number assigned to every pixel is checked against the equivalence map. If it has an equivalent number, the lower one is assigned to the pixel. The finished image is shown in Figure 3-5. Since we are interested in the centroids of the blobs, this is also computed in the second pass. This is accomplished by totaling the x positions and y positions of the pixels

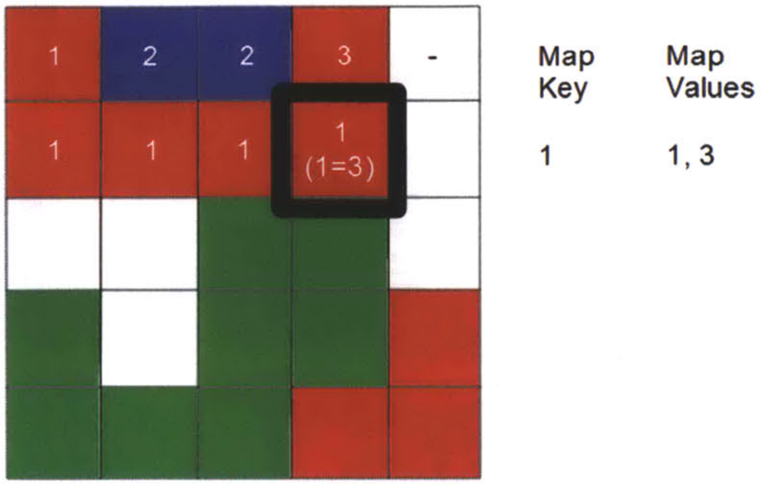


Figure 3-3: An illustration of the third case, where the pixels to the top and left of the current pixel are different colors. The two numbers (3 and 1) are declared to be describing the same connected component, and the lower number is chosen for the current pixel

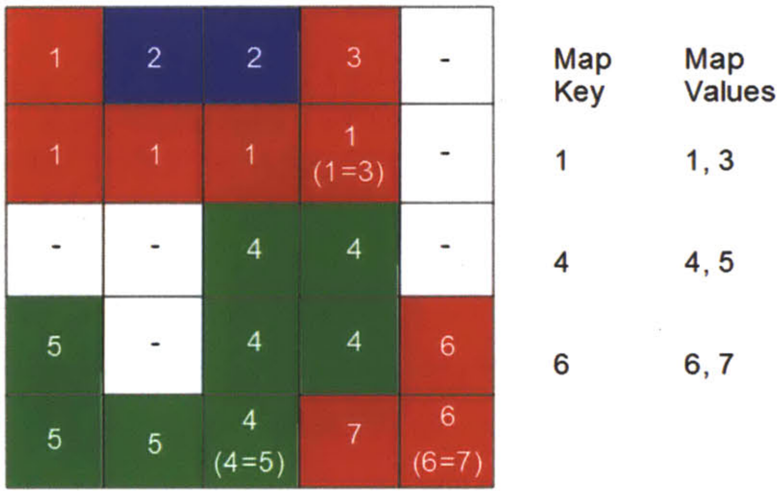


Figure 3-4: The completed first pass over our sample image. The numbers declared equivalent are shown to the right

belonging to each Connected Component, to be divided by the pixel count at the end.

This algorithm can be improved by having each connected component be a linked list which points to the start of the list of pixels it contains. When two connected components numbers are determined to be the same, the list belonging to the higher number would simply be appended to the list of the lower number. This would

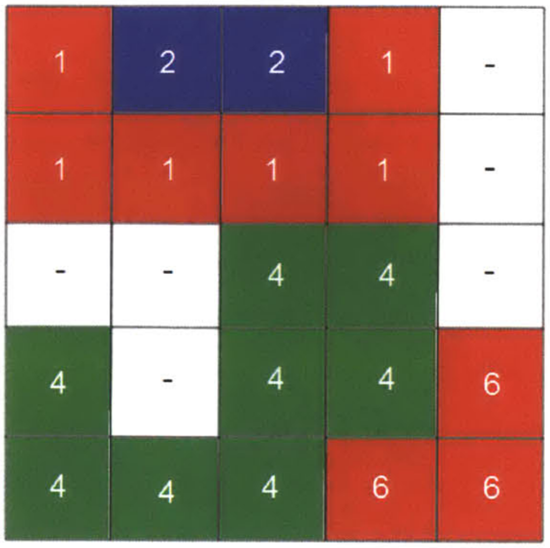


Figure 3-5: The finished connected components graph

remove the need for the second pass through the pixels for the purpose of completing the connected components algorithm. To determine the centroids of each connected component, its list of pixels would simply be iterated through.

3.3.2 Determining the identifier

This algorithm primarily concerns itself with the orientation of the hand, rather than the distance from the camera or its position in the webcam's field of view, which will be added in after the pose has been identified from the library. Therefore the relevant metric is the angle between components, which is invariant of the size of the hand or the location in the frame. This eliminates the need for a complex algorithm to center the hand and resize it in the frame, which can be unnecessarily susceptible to colors in the environment. This information is encoded in a string such as "0 1 354," which indicates that using color 0 as the origin, color 1 is at 354 degrees from the x axis. Using a color's centroid as the origin is what allows the algorithm to be independent of the hand's location in the frame.

3.3.3 Comparing to the library

Library entries are created by capturing an image from the camera and manually specifying a solid model that matches. This image is used to create an identification string, as defined above.

These identification strings can then be matched against images the web camera captures, to determine the position the hand is in. This matching is done by a basic distance metric; the squared differences in the angles are summed, and the library entry with the lowest value is selected. The code to compute these distances is given in Appendix B. Not every color is necessarily in every image; if a pair is missing from either entry that exists in the other, the angle difference is assumed to be 180 degrees, the maximum possible.

3.4 Implementation

Glove colors are defined using a GUI programmed in C++ that allows the developer to simply drag a box over a colored region in an image of the glove captured by the webcam. The average and standard deviation are calculated and saved for use by the processing code.

The camera capture and image processing for the algorithm described above were programmed in C++ with the aid of existing OpenCV libraries. The primary code used to turn an image into a characteristic string is given in Appendix A.

In addition to turning images into strings, the processing code continuously publishes the characteristic strings to a socket. This socket can then be listened to by any code to which the developer wishes to link the image processing. For testing purposes, a Java listener has been programmed. This Java listener has two modes: the first mode is used to develop the pose library. The second mode displays the hand position characteristic of the library entry closest to that seen by the webcam (Figure 3-6).

For the first mode, library entries are first specified and then linked to an image string (received via the socket). There is a preloaded selection of hand poses that can

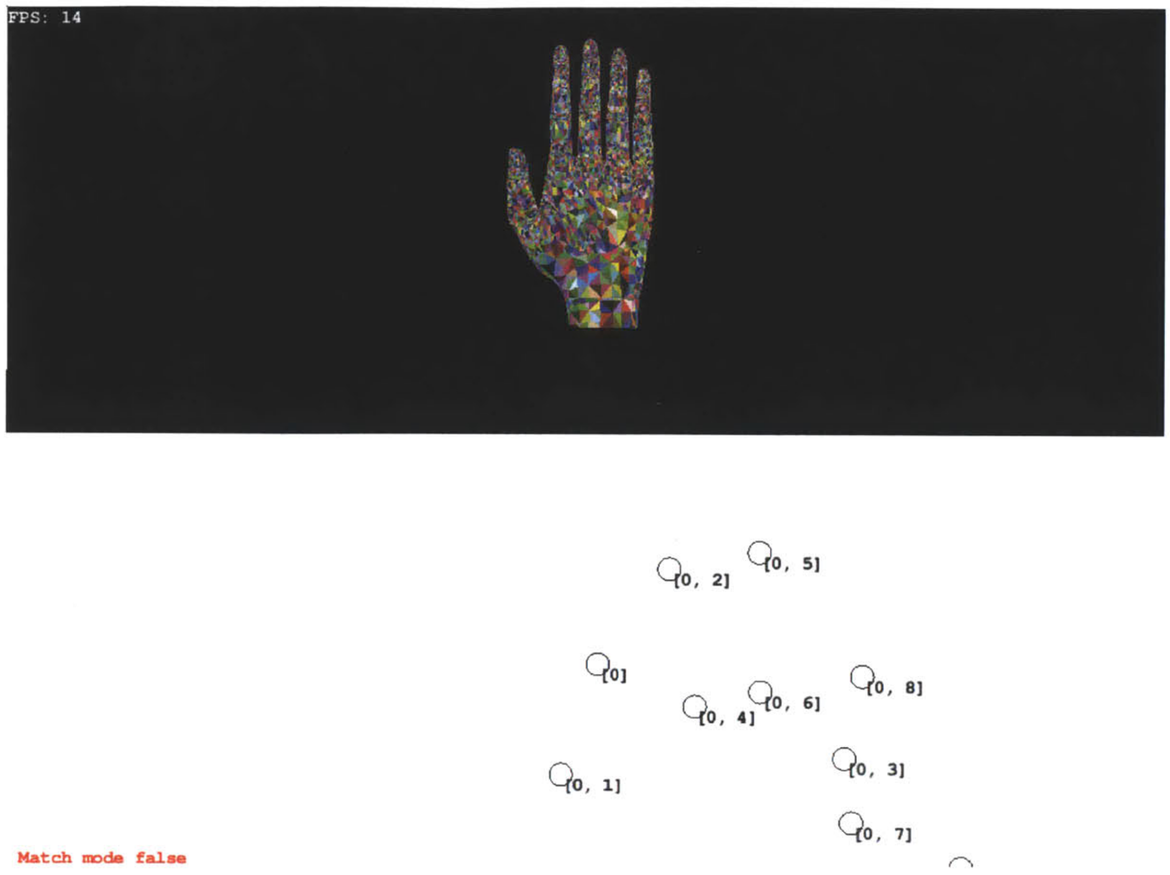


Figure 3-6: The GUI for the Java listener; the circles below the pose display represent the centroids of the colored regions

be flipped through with the arrow keys. These poses can be further rotated in any of the three dimensions. When the model position is specified, the developer puts the glove on, holds her hand in front of the camera, and presses the capture button. The combination of model, angle, and image string together define the library entry.

The second mode disables the user's ability to rotate and switch between poses, instead pulling up the model that matches the hand position currently in view of the camera. Since this model matching occurs in real-time, what has been achieved is a virtual hand mirroring that of the user.

3.5 Glove design

The key features of the glove are that it needs to have colors in locations that are important to track, it needs to be comfortable to wear for a variety of users, and it needs to be aesthetically pleasing so that users are willing to wear it.

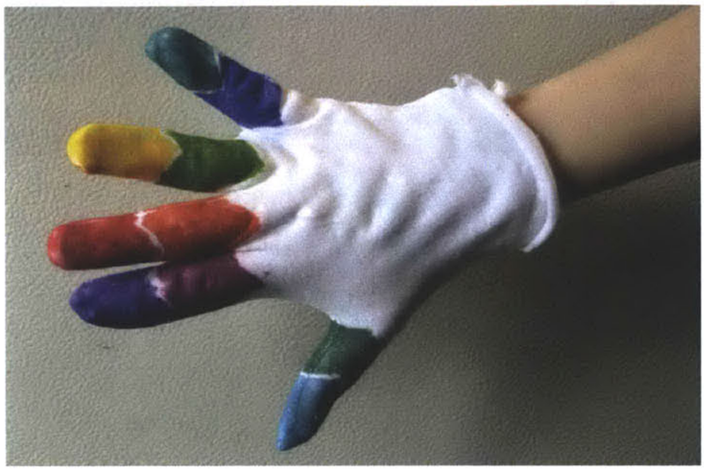


Figure 3-7: The glove prototype

A white glove (Figure 3-7) has been colored such that the colored regions correspond to the individual joints in the hand. These regions have been intentionally overdefined for robustness as they undergo algorithmic processing. The palm and back of the hand have been left white, however a product logo could be placed in these areas. A pocket has been sewn into the back of the glove such that the Vivitouch module rests on the top of the hand. While a cotton glove has been used for this prototype, future versions could use a thinner lycra or spandex material. The size of the user's hand and correct positioning of the colored sections is an acknowledged challenge in developing this product. This may be addressed by releasing different sized gloves where a stretchy material can accomodate variations in hand size.

3.6 Haptic feedback

To enhance the gaming experience, controllers are equipped to produce haptic feedback. Conventionally, this is achieved by connecting an eccentric weight to a small motor inside the controller. When the motor spins the weight at high speeds, the controller vibrates. The major drawback of this method is that it outputs only a single type of vibration across the entire controller, resulting in limited feedback that feels one-dimensional.

In 2011, Artificial Muscle Incorporated released the Vivitouch, a proprietary haptics component which produces a wide range of realistic feedback (Figure 3-8) [19].

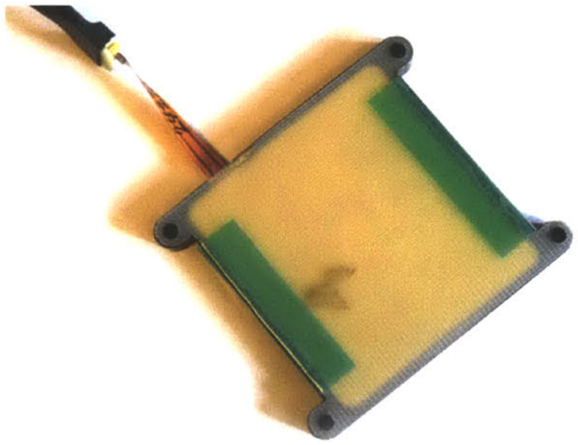


Figure 3-8: A Vivitouch haptic feedback component

Vivitouch is based on electroactive polymer (EAP) technology, which are polymers that respond to electrical stimulation with a change in physical, optical, or magnetic properties [20]. The ViviTouch actuator consists of a dielectric elastomer film which is sandwiched between two electrodes (Figure 3-9).

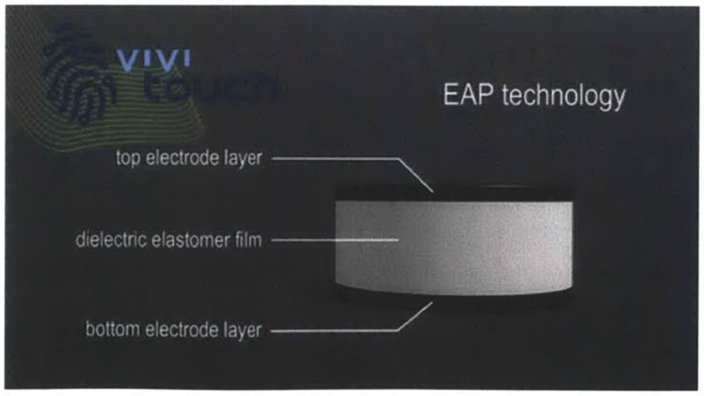


Figure 3-9: How EAP technology works [20]

When the polymer is subjected to a voltage potential, it flattens and expands, moving in a planar direction [19]. An inertial mass included in the module amplifies the vibration of the device and the actuator is controlled using waveforms created as audio files [22]. Consequently, the device is capable of recreating effects such as the beating of a heart, or the impact of an arrow in a target. As an integrated product, a dedicated haptic control channel should be used to drive the haptics.

Given the size of the device and the limited space in the glove, the Vivitouch actuator has been placed on the back of the hand so as not to interfere with the user's grip. In future prototypes, placement of a smaller haptic component in the palm of the hand should enhance feedback.

Chapter 4

Testing and applications

4.1 Testing the system

To test the controller, motion of the hand is visualized using a three-dimensional hand model created using Blender, an open source 3D content creation program [23]. The basic structure of the hand is modeled using a series of cubes, which are manipulated into the shape of the palm and index finger. The index finger is then copied and scaled to produce the remainder of the fingers. Using a proportional editing tool, the mesh is adjusted until a realistic hand shape is obtained. Finally, the mesh is smoothed to improve the surface appearance (see Figure 4-1).

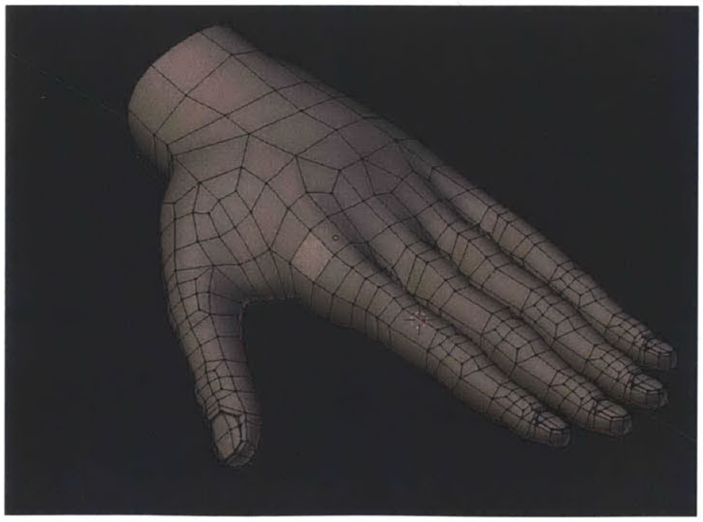


Figure 4-1: The initial hand model mesh

Since this model must be adjusted into different pose configurations, it is "rigged" by adding a number of constraints (called an armature) which is made up of a series of elements called bones (Figure 4-2a). The bone placements mimic the layout of the specific hand joints. The bones are then controlled via a custom armature (Figure 4-2b) which, when moved, proportionally rotates each joint.

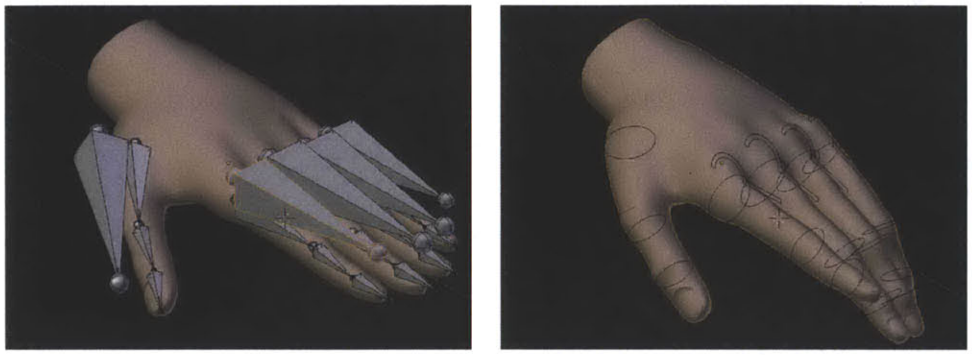

(a) Hand rigged with bones (b) Armature controls the bones

Figure 4-2: Bone and armature placement in the hand model

Adding constraints to each joint ensures that only natural motion is achieved. For example, the finger joints have been constrained to ensure that they cannot bend backwards or sideways. A technique called "weight painting" (Figure 4-3) is used to control the degree to which moving a given armature deforms nearby regions of the mesh. Each joint is painted using a heat map, where red indicates areas of 100% deformation and blue indicates areas of 0% deformation. Using the armatures, the model is posed in a number of configurations and the raw face information is exported for use in the test demonstration (Figure 4-4).

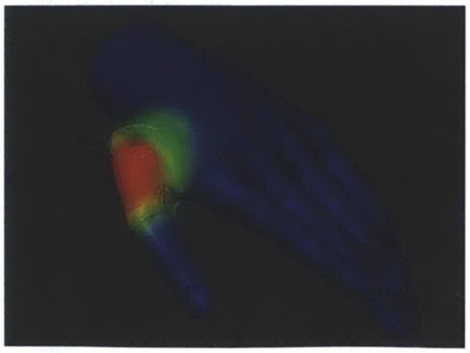


Figure 4-3: Weight painting is used to control deformation

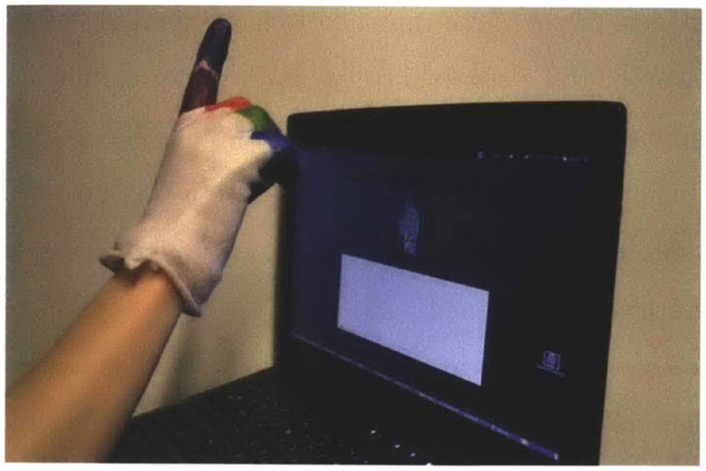
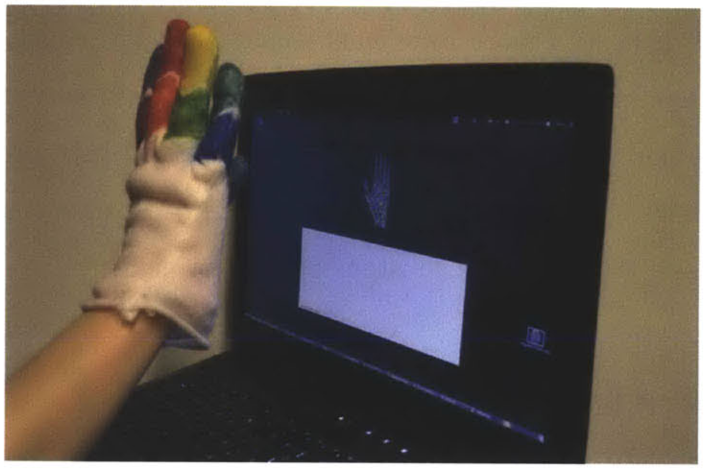



Figure 4-4: Testing pose recognition

4.2 Issues encountered

While testing the system as described above, several issues came up. The first issue is camera calibration. If the camera is continuously adjusting the white balance, the colors saved for the algorithm rapidly become inaccurate. This means that the product will need to disable such white balancing in end-users' cameras to be effective.

The second issue is the camera field of view. It needs to be trivial for the user to maintain their hand poses in the frame of their webcam. This can most easily be achieved by giving feedback about the location of the user's hand such they know when they exceed the limits of their setup.

The third and by far the biggest issue is noise in the environment. In testing the system, the colors seen in the environment often dominated the colors seen in the glove, and testing needed to be stopped to tweak the color definitions and thresholds. The system was sufficiently robust that the system would often still obtain the correct result, but it was insufficiently reliable to be put into a product as-is.

This problem could potentially be solved in a few different ways. The first would be to include a calibration step done by the user, where a snapshot of the background is taken and subtracted out of future images. The second would be to more tightly bound the color definitions, so that fewer colors from the environment are mistaken for glove colors. This solution has the problem of being highly light-dependent. The third would be to develop a more intelligent algorithm that subtracts out elements from the image that are not moving appreciably. It would keep track of the last known location of the glove, and simply modify this model based on the differences between successive images taken from the camera. This last option is likely to be the best, but would itself require fairly substantial development.

4.3 Applications

While gesture-based gaming can be used to create a more immersive experience, it is important to recognize that not all game play lends itself to gestural control. In some cases it may even detract from the user experience if complex or unnecessary gestures are required to complete actions that might be accomplished with the single press of a button on a conventional controller (i.e. selecting menu options). Potential applications for the controller in existing games will be presented to provide examples of how this technology can enhance the users experience when appropriately used.

4.3.1 Spellcasting

The first and perhaps most obvious application of this controller is for spellcasting mechanics in games. *Okami* is a Japanese action-adventure game which features a "celestial brush" which players use in combat, puzzles, and general game play [24]. This mechanic involves drawing patterns with an analog controller stick or the Wii remote to repair bridges, slash enemies, or create in-game effects like wind or wire. The adventure game *Myst V: End of Ages* features a similar mechanic where the player draws on an on-screen tablet to trigger events and move through the world [25].

A study comparing a pen and a mouse in editing graphic diagrams showed that users take approximately twice as long to draw a diagram while using a mouse [27]. From these data it may be inferred that a computer mouse or analog controller may not be the optimal control scheme for such mechanics. Instead, diagrams could be created by allowing the player to "fingerpaint" by tracing her finger into the air or on a surface. Additionally, the spellcasting mechanic may be enhanced by mapping spells to certain hand movements or gestures.

4.3.2 Simulation games

Simulation games attempt to replicate "real life" activities while still retaining elements of traditional goal-based gaming. In the *Trauma Center* game series, players assume the role of a surgeon in order to heal their patients from injuries or diseases using various surgical instruments and suturing. The games were originally designed for the Nintendo DS handheld gaming device and the Nintendo Wii, utilizing the DS' touch screen and the Wii remote motion capabilities, respectively, to simulate the use of surgical implements. A glove-based controller would improve the plausibility and immersiveness of the simulation by mapping "real" movements to their in-game counterparts.

Chapter 5

Conclusion

A successful works-like prototype of a glove-based gaming peripheral has been built and tested. It comprises both hardware and software components: a colored glove with a Vivitouch artificial muscle haptics module and a real-time vision processing algorithm, respectively.

Following manual color calibration of the glove with the webcam, gestures can be mapped to various orientations of a corresponding 3D hand model which are stored in a library. Testing demonstrates that when the user moves her hand into a given position, the vision algorithm recognizes the gesture and updates the display with the correct pose from the library. This can be readily extended to other behaviors, simply by linking a position to an action in a game, for instance, rather than just rendering a different model. Furthermore, the code developed produces the image characterization in the form of a string that can be accessed using any programming language.

The Vivitouch haptics component has been successfully programmed to provide vibratory feedback in an independent system, however limited driver support has prevented its full integration into the glove functionality. In addition to the future work outlined below, the next iteration of the design will strive to incorporate full haptic support.

5.1 Future work

Despite having a working model, there are number of improvements that must be made to this system before it can be a viable product. Orientation is not supported in the current implementation and the algorithm should be extended to localize the orientation of the hand within its reference frame.

For the glove, a looks-like prototype should be created to explore whether different designs or color schemes might be more aesthetically appealing to users. Ergonomics and fabric choice should also be considered to ensure that the glove can fit a range of hand sizes.

A prominent issue with the ergonomics of the works-like prototype is that the current Vivitouch module is too large to be mounted on the hand. Further collaboration with Vivitouch could allow smaller haptics modules to be sourced. This would provide a more comfortable user experience and future controllers may be outfitted with multiple small Vivitouch actuators.

Finally, extensive user testing must be completed to identify potential issues and improve the user experience. The limited testing performed (Section 4.2) has already revealed a number of issues with the current device which must be addressed. Testing the prototype with other webcam setups is particularly important to determine the system reliability and robustness.

Appendix A

ImageProcessor.cpp

```
#include "../include/ImageProcessor.h"
#include "../include/Image.h"
#include "stdio.h"
#include <tr1/unordered_map>
#include <tr1/unordered_set>
#include <vector>
#include <algorithm>
#include <iostream>
#include <sstream>
#include <string>

using namespace std::tr1;

typedef unordered_map<int, unordered_set<int> > imMap;

struct centroidInfo{
    int count, xTotal, yTotal, colorIndex, ccIndex;
};

ImageProcessor::ImageProcessor()
```

```
{
}

ImageProcessor::~ImageProcessor()
{
    //dtor
}

string ImageProcessor::process(Image* img, ColorPalette* colorPalette)
{
    /*
    First loop: determine what the "colorsample" for each pixel is,
    and do the first pass of connected components
    */
    //static int* dims = img->getDimensions()/scale;
    std::cout<<colorPalette->currIndex;
    static int dims[2] = {img->getDimensions()[0]/scale,
    img->getDimensions()[1]/scale};
    std::cout<<dims[0]<<"\n";
    int hue, colorIndex, ccIndex = 0, colorIndexUp, colorIndexLeft;
    Image::RgbPixelFloat temp, tempUp, tempLeft;
    bool newColor;
    imMap::iterator leftIter, upIter;
    int** cc = new int*[dims[0]];
    imMap equivalence;

    for(int x = 0; x<dims[0]; x++) {
        cc[x]=new int[dims[1]];
        for(int y = 0; y<dims[1]; y++){
        newColor = true;
        //Grab relevant hue
        temp = (*img)[x*scale][y*scale];
        hue = convertRGBToHSV((int)temp.r,(int)temp.g,(int)temp.b);
        //Compare to each entry in the ColorPalette
        //If a matching color is found, check the pixels above and
        to the left, and if they're the same, assign the pixel the
        number of the matching pixel. If they both match, add the
        pair to the actuallyTheSame array
        colorIndex = colorPalette->findSampleIndex(hue);

        if (colorIndex==-1) {
            cc[x][y] =-1;
            continue;
        }

        //Check above
        if (y>0) {
            tempUp=(*img)[x*scale][(y-1)*scale];
            colorIndexUp = colorPalette->findSampleIndex
(convertRGBToHSV((int)tempUp.r,(int)tempUp.g,(int)tempUp.b));
            if (colorIndex==colorIndexUp){
                newColor= false;
                cc[x][y]=cc[x][y-1];
            }
        }

        //Check left
        if (x>0) {
            tempLeft=(*img)[(x-1)*scale][y*scale];
            colorIndexLeft = colorPalette->findSampleIndex
(convertRGBToHSV((int)tempLeft.r,(int)tempLeft.g,(int)tempLeft.b));
            //If this color is the same as the one to the left,
            but not the same as the one above
            if ((colorIndex==colorIndexLeft)&&(newColor)){
                newColor = false;
                cc[x][y]=cc[x-1][y];
            }
            //else if this color is the same as both the one to the
            left and above
            else if (colorIndex==colorIndexLeft){
            //newColor is already false
                if (cc[x-1][y]==cc[x][y-1]) {
                    cc[x][y]=cc[x-1][y];
                    continue;
            }
            //Then colorIndex==colorIndexLeft and colorIndexUp, and
            so cc@colorIndexLeft is equiv cc@colorIndexUp, and this
            should be added to equiv array
            // check existing equivalence map, and if either member is a
            dictionary entry, add the other to the list (making sure it's
            not already there)
            //If neither is, add the smaller one as the key, and then BOTH
            as an entry. If both exist, take the smaller one and add the
            bigger one's stuff to it.
                    //while(!upIter->second.empty()) delete
                    upIter->second.back(), upIter->second.pop_back();
                   //TODO: make sure if "new" is being used, I use "delete".
                   Currently I'm assuming the memory will be deallocated
                   when the vector goes out of scope.
```

```
leftIter = getMapKey(cc[x-1][y],equivalence);
upIter = getMapKey(cc[x][y-1],equivalence);


if (leftIter!=equivalence.end() && upIter!=equivalence.end()){
    if (cc[x-1][y]<cc[x][y-1]) {
       //Add colorIndexUp's equivalence vector to
       colorIndexLeft's, and delete colorIndexUp's entry
        addValToMapVector
       (leftIter,upIter,equivalence, cc, x, (y-1), x, y);
    }
    else{
        //delete leftIter->second;
        addValToMapVector
       (upIter, leftIter, equivalence, cc, x-1, y, x, y);
    }
}
else if (leftIter!=equivalence.end()){
    addValToMapVector(leftIter,cc[x][y-1]);
    cc[x][y] = leftIter->first;
}
else if (upIter!=equivalence.end()){
    addValToMapVector(upIter,cc[x-1][y]);
    cc[x][y] = upIter->first;
}
else{
    if (cc[x-1][y]<cc[x][y-1]){
        addValToMapVector
       (equivalence, cc, (x-1), y, x, (y-1), x, y);
   }
   else{
```

```
                addValToMapVector
                (equivalence, cc, x, (y-1), (x-1), y, x, y);
            }
          }
        }
    }
    if (newColor){
        cc[x][y]=ccIndex;
        ccIndex++;
    }
}
}


/*
Second loop: do the second pass of connected components
and get centroids
*/
//Have a map from CC num to [total pixels of that, total x, total y]
//Have another map from Color Sample index to max num of pixels
unordered_map<int, centroidInfo> CCPixelMap, sampleMap;
for (int x = 0; x<dims[0]; x++){
    for (int y = 0; y<dims[1]; y++){
        if (cc[x][y]==-1) continue;
        imMap::iterator iter = getMapKey(cc[x][y],equivalence);
        if (iter!=equivalence.end()) cc[x][y]=iter->first;

        //Add the pixel to the data structure storing the sizes of
        the components
        if (CCPixelMap.find(cc[x][y])==CCPixelMap.end()){
            CCPixelMap[cc[x][y]].count = 1;
```

```
                CCPixelMap[cc[x][y]].xTotal = x*scale;
                CCPixelMap[cc[x][y]].yTotal = y*scale;
                temp = (*img)[x*scale][y*scale];
                colorIndex = colorPalette->findSampleIndex
(convertRGBToHSV((int)temp.r,(int)temp.g,(int)temp.b));
                CCPixelMap[cc[x][y]].colorIndex = colorIndex;
            }
            else {
                CCPixelMap[cc[x][y]].count+=1;
                CCPixelMap[cc[x][y]].xTotal+= x*scale;
                CCPixelMap[cc[x][y]].yTotal+= y*scale;
            }
        }
    }
    //Go through all the CCPixelMap entries,
    and find the largest of each color
    for (unordered_map<int, centroidInfo>::iterator iter =
CCPixelMap.begin(); iter!=CCPixelMap.end(); iter++){
    if (sampleMap[iter->second.colorIndex].count < iter->second.count){
            sampleMap[iter->second.colorIndex] = iter->second;
            sampleMap[iter->second.colorIndex].ccIndex = iter->first;
        }
    }

    std::stringstream convert;
    std::string output;
    convert<<"";
    double angle;
    int x0, y0, x1, y1;
    for (unordered_map<int, centroidInfo>::iterator
iter = sampleMap.begin(); iter!=sampleMap.end(); iter++){
        for (unordered_map<int, centroidInfo>::iterator iter2 = iter;
iter2!=sampleMap.end(); iter2++){
            x0 = iter->second.xTotal/iter->second.count;
            y0 = iter->second.yTotal/iter->second.count;
            x1 = iter2->second.xTotal/iter2->second.count;
            y1 = iter2->second.yTotal/iter2->second.count;
            angle = atan2((x1-x0), (y1-y0));
//I think I may have somehow switched x and y?
            std::cout<<iter->second.count<<" "<<iter->
second.colorIndex<<" "<<y0<<" "<<x0<<"\n";
            std::cout<<iter2->second.count<<" "<<iter2->
second.colorIndex<<" "<<y1<<" "<<x1<<"\n\n";
            if (iter->second.count<50 || iter2->second.count<50){
                convert<<iter->second.colorIndex<< " " << iter2->
second.colorIndex<<" "<<-1000.0<<"\n";
            }
            else{
                convert<<iter->second.colorIndex<< " " << iter2->
second.colorIndex<<" "<<angle<<"\n";
            }
        }
    }
    output = convert.str();

    //printEquivMap(equivalence);
    printCC(cc,dims[0],dims[1]);
    //std::cout<<output;
    std::cout<<"Next frame!\n";
    // Deallocate cc

    for (int i = 0; i < dims[0]; ++i){
        delete[] cc[i];
    }
    delete[] cc;
    return output;
}

//For one of top or left in equiv
void ImageProcessor::addValToMapVector(imMap::iterator iter, int val){
    iter->second.insert(val);
}

//For both top and left in equiv
void ImageProcessor::addValToMapVector(imMap::iterator& iterTo,
imMap::iterator& iterFrom, imMap& equivTemp,
int **ccTemp, int oldX, int oldY, int currX, int currY){
    unordered_set<int>::iterator setIter;
    for (setIter = iterFrom->second.begin();
setIter!=iterFrom->second.end(); setIter++){
        iterTo->second.insert(*setIter);
    }
    equivTemp.erase(ccTemp[oldX][oldY]);
    ccTemp[oldX][oldY] = iterTo->first;
    ccTemp[currX][currY] = iterTo->first;
}

//For neither top nor left in equiv
// Small: small value, Big: bigger value
```

```
void ImageProcessor::addValToMapVector
(imMap& equivTemp, int **ccTemp, int smallX, int smallY,
int bigX, int bigY, int currX, int currY){

    equivTemp[ccTemp[smallX][smallY]].insert(ccTemp[smallX][smallY]);
    equivTemp[ccTemp[smallX][smallY]].insert(ccTemp[bigX][bigY]);
    ccTemp[currX][currY] = ccTemp[smallX][smallY];
    ccTemp[bigX][bigY] = ccTemp[smallX][smallY];

}

imMap::iterator ImageProcessor::getMapKey
(int num, imMap& mapToCheck){
    imMap::iterator iter;
    unordered_set<int>::iterator returnIndex;

    for (iter = mapToCheck.begin(); iter!=mapToCheck.end(); iter++){
        returnIndex = iter->second.find(num);
        if (returnIndex != iter->second.end()) {
            return iter;
        }
    }
    return mapToCheck.end();
}

void ImageProcessor::printEquivMap(imMap& equivTemp){
    imMap:: iterator iterMap;
    unordered_set<int>::iterator iterVec;
    for (iterMap = equivTemp.begin();
iterMap!=equivTemp.end(); iterMap++){
        std::cout<<iterMap->first<<"\n";
        for (iterVec = iterMap->second.begin();
iterVec != iterMap->second.end(); iterVec++){
            std::cout<<"       "<<*iterVec<<"\n";

        }
    }
}

void ImageProcessor::printCC(int **ccTemp, int dimX, int dimY){
    for (int x = 0; x<dimX;x++){
        std::cout<<"\n";
        for (int y = 0; y<dimY; y++){
            std::cout<<" "<<ccTemp[x][y];
        }
    }
}

double ImageProcessor::convertRGBToHSV(int r, int g, int b)
{
    float alpha = .5*(2*r-g-b);
    float beta = (sqrt(3)/2)*(g-b);
    float hue = atan2(beta,alpha);
    float lightness = (r+g+b)/3;
    //std::cout<<lightness<<", ";
    if (lightness>lightnessThresh || lightness<minLightThresh)
return -1000;
    return hue*180/3.1415926535;
}
```

Appendix B

Library Matching

This method determines which library entry best matches the string taken from the webcam.

```
public LibraryEntry matchToLibrary(){

double minDistance = 10000000, distance;
LibraryEntry minLib = new LibraryEntry();
this.cam.lock.lock();
HashMap<TreeSet<Integer>, Double> currentMap =
(HashMap<TreeSet<Integer>, Double>)
this.cam.angleMap.clone();
this.cam.lock.unlock();
for (LibraryEntry lib:library){
//Iterate through the hashmap, and for every entry find
the angle distance distance = 0;
//Go through the library map and find the error for each entry
for (Map.Entry<TreeSet<Integer>,Double> entry: lib.match.entrySet()){
if (entry.getKey().size()==1) continue;
if (currentMap.containsKey(entry.getKey())){
```

```
distance+=Math.pow((currentMap.get(entry.getKey())-entry.getValue()),2);
}
else {
distance+=Math.pow(Math.PI,2);
}
}
distance/=lib.match.size();

//Go through the current map and find the error for each entry
double distance2 = 0;
for (Map.Entry<TreeSet<Integer>,Double> entry: currentMap.entrySet()){
if (entry.getKey().size()==1) continue;
if (lib.match.containsKey(entry.getKey())){
distance2+=Math.pow((lib.match.get(entry.getKey())-entry.getValue()),2);
}
else {
distance2+=Math.pow(Math.PI,2);
}
}
distance2/=currentMap.size();
distance+= distance2;
System.out.println("For pose: " + lib.id + " Error total is:" + distance+"\n");
if (distance<minDistance){
minDistance = distance;
minLib = lib;
}
}
return minLib;
}
```

Bibliography

[1] *Nintendo Wii.* Retrieved from http://www.nintendo.com/wii/.

[2] *PlayStation Move.* Retrieved from http://us.playstation.com/ps3/playstation-move/.

[3] *Xbox 360 Kinect.* Retrieved from http://www.xbox.com/en-US/kinect.

[4] *Vivitouch.* Retrieved from http://www.vivitouch.com/.

[5] *Dance Dance Revolution.* Retrieved from http://konami.com/ddr.

[6] Current Projects. *Harmonix.* Retrieved from http://www.harmonixmusic.com/projects.

[7] Company History. *Nintendo Corporate.* Retrieved from http://www.nintendo.com/corp/history.jsp.

[8] Analog Devices. (2006, May 5). *Analog Devices and Nintendo Collaboration Drives Video Game Innovation with IMEMS Motion Signal Processing Technology* [Press Release]. Retrieved from http://www.analog.com/en/press-release/May_09_2006_ADI_Nintendo_Collaboration/press.html

[9] Castaneda, K. (2006, May 13). Nintendo and PixArt Team Up. *Nintendo World Report.* Retrieved from http://www.nintendoworldreport.com/news/11557

[10] Ohta, K. (2006). *U.S. Patent No. 2007/0211027.* Washington, DC: U.S. Patent and Trademark Office.

[11] Sony Computer Entertainment, Inc. (2010, March 10). *PlayStation Move Motion Controller Delivers a Whole New Entertainment Experience to PlayStation 3: New PlayStation Move Sub-Controller, Enabling Intuitive Navigation, to Accompany the Release of the Motion Controller This Fall and 36 Developers and Publishers to Support PlayStation Move Platform* [Press Release]. Retrieved from `http://scei.co.jp/corporate/release/100311e.html`

[12] Barras, C. (2010, January 7). Microsoft's body-sensing, button-busting controller. *NewScientist.* Retrieved from `http://www.newscientist.com/article/mg20527426.800-microsofts-bodysensing-buttonbusting-controller.html`

[13] *Kinect Including Kinect: Adventures!.* Retrieved from `http://www.play.com/Games/Xbox360/4-/10296372/Project-Natal/Product.html#jump-tech`.

[14] PlayStation Move Teardown. *iFixit.* Retrieved from `http://www.ifixit.com/Teardown/PlayStation-Move-Teardown/3594/1#.T70iMdxYsjp`

[15] Bolton, M. (2010, September 5). PlayStation Move review: Design. *Tech Radar.* Retrieved from `http://www.techradar.com/reviews/gaming/gaming-accessories/playstation-move-713638/review?artc_pg=2`

[16] Wang, R., Popovic, J. (2009). Real-time hand-tracking with a color glove. *ACM Transaction on Graphics (SIGGRAPH 2009)*, 28(3). Retrieved from `http://people.csail.mit.edu/rywang/handtracking/s09-hand-tracking.pdf`

[17] Ward, J. (2008, April 29). What is a game engine? *Game Career Guide.* Retrieved from `http://www.gamecareerguide.com/features/529/what_is_a_game.php`

[18] Wilson, K. (2006, July 15). Why C++? *GameArchitect.net.* Retrieved from `http://gamearchitect.net/Articles/WhyC++.html`

[19] Feeling is Believing. *Vivitouch.* Retrieved from `http://www.vivitouch.com/eap.php`.

[20] Bar-Cohen, Y. (2002, March 14). *Electroactive Polymers as Artificial Muscles: Reality, Potential and Challenges* [PDF Slides]. Retrieved from `http://trs-new.jpl.nasa.gov/dspace/bitstream/2014/38252/1/04-0044.pdf`

[21] EAP Technology. (2011). *Joystiq.* Retrieved from `http://www.blogcdn.com/www.joystiq.com/media/2011/09/vivitouch.jpg`

[22] Vivitouch Haptic Feedback Evaluation Kit Operating Instructions. (2011, October). *Artificial Muscle Inc.*

[23] *Blender.* Retrieved from `http://www.blender.org/`

[24] *Okami.* Retrieved from `http://www.okami-game.com/`

[25] *Myst V: End of Ages.* Retrieved from `http://www.ubi.com/UK/Games/Info.aspx?pId=3450`

[26] *Trauma Center: Under the Knife 2.* Retrieved from `http://www.atlus.com/tcutk2/`

[27] Apte, A., Kimura, T. (1993, March). *A Comparison Study of the Pen and the Mouse in Editing Graphic Diagrams.* Paper presented at IEEE/CS Symposium on Visual Languages, Bergen, Norway.