EXPLORATION OF PHYSICAL COMPUTING: DIGITAL INTERFACES USING MICROSOFT KINECT AND PEGGY 2 LED MATRIX

JEFF FRANKLIN
FALL 2012

ABSTRACT:

Physical computing, or more specifically social immersive media, is an offshoot of New Media that incorporates interactive physical systems (software/hardware systems) to sense and respond to the analog world. This serves as a creative framework for understanding our own relationship to the digital world. In this project, computer vision is incorporated as a medium between both input and output through the use of two cameras, a typical webcam and a Microsoft Kinect. For the first part of the project, a webcam streams visual data into a computer program which interpolates the video data into a 25x25 virtual dot matrix. This interpolation is transmitted via USB port to a Peggy 2 LED display in real time. In the second portion of the project, a Microsoft Kinect for Windows camera streams visual data into a Greenfoot Java environment. This virtual environment tracks the user's skeleton and allows the user to play musical tones based on the movement of the user's right hand. Together these two interfaces are intended to provide an introduction to physical computing and social immersive media.

A BRIEF HISTORY OF PHYSICAL COMPUTING:

Physical computing (microcontroller embedded systems) have been around since the 1950s, but the physical computing of today, as associated with artists and designers, was first popularized by Ars Electronica in 1979. Ars Electronica is a new media think tank, an R&D facility, a museum, an art and music festival, and an awards show in Linz, Austria. Today, physical computing is gaining popularity, especially in Australia, South America, and Europe where many colleges now commonly have new media and interactive telecommunications programs. Scott Snibbe, an interactive media artist here in America, has had an extremely successful career in physical computing and related fields and stands as a leading figure. Mr. Snibbe recently coined the term 'social immersive media' to more accurately describe the direction in which physical computing is heading. Two others, Casey Reas and Ben Fry of MIT, created the Processing computer language. All three of these gentlemen have put in time with MIT's Media Lab - America's premier design and technology development facility. The Interactive Telecommunications Graduate Program (ITP) at NYU also focuses on physical computing and new media design and implementation. The growth of microcontrollers like Arduino has been brought on in part by the concept of rapid prototyping. Today's microcontrollers and microcontroller systems can go from brainstorm to implementation faster than ever and more companies, like IDEO and Teague, are catching on to the cost-effectiveness and creativity brought forth by this burgeoning field. Today, there are numerous software environments like Processing such as PureData, Max/MSP, VVVV, and Arduino. Even Ableton Live can be considered to have similar capabilities to do similar tasks.

Physical computing offers new ways to market products, create branding, invite interaction, and gauge consumer and user feedback through its intrinsic psychological value. Physical computing offers novel ways of human-computer interaction and allows for greater social participation. As the technology advances, increased accessibility allows for increased innovation for more participants than ever before. While physical computing is a broad term, its applications in industries across the board grows increasingly

larger every year. This puts the field at the forefront of technology and stands as a flagship of new media with great potential into the future.

INTRODUCTION TO PROJECT:

This project arose from the combined brainstorming of Dr. Michael Haungs and I. Based loosely on the audio/visual performance art of music festivals, this project seeks not only to entertain in the same capacity as performance art but also to provide new equipment and content for the benefit of the Liberal Arts & Engineering Studies (LAES) program at Cal Poly. This project also aims to offer an introduction to physical computing through the steps involved in interfacing software and hardware as applicable to this project. I will attempt to expound upon the relative ease of use and advanced capabilities of the software environments and how their features prove ideal for the expansion of expressive technologies in LAES.

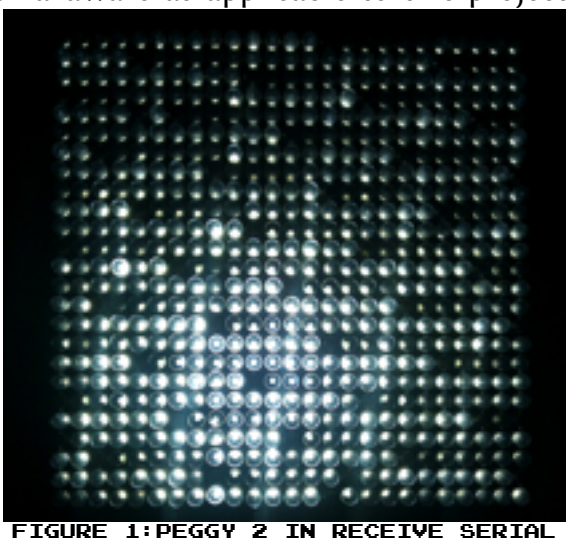


FIGURE 1:PEGGY 2 IN RECEIVE SERIAL MODE

I will be attempting to keep this project as OS-generic as possible. I did my senior project on a computer running Windows 7 but I also understand that most people attempting anything similar will be working on a Mac. There still may be certain limitations on what a Mac can do with a Kinect for Windows but it is a constantly evolving process still and new updates so the software features are subject to change. For this reason I am including version numbers of software for ease in comparing what I did to what is currently available.

A list of the components involved in the two projects follows:

PROJECT 1: PEGGY 2 LIVE VIDEO STREAM		
HARDWARE:		
Peggy 2 LED Display Kit	An LED matrix using 10mm white LEDs on a 25x25 grid. (Hackable)	http://shop.evilmadscientist.com/productsmenu/tinykitlist/75-peggy2
Generic webcam	I used the built-in camera on my laptop although any webcam will work.	N/A
SOFTWARE:		
Arduino 1.0.1	The integrated development environment (IDE) for the Arduino microcontroller.	http://www.arduino.cc
Processing 2.0b6	The Processing IDE. Created out of MIT's Media Lab, this programming language is based on Java and offers a simplified coding structure specifically for Audio/Visual (A/V) and human interaction projects.	http://www.processing.org

PROJECT 2: KINECT & GREENFOOT MUSIC GAME		
HARDWARE:		
Microsoft Kinect for Windows (K4W)	The Microsoft Kinect for Windows is an update to the Xbox Kinect and must be treated differently. It has increased functionality including near-mode and face-tracking but decreased compatibility. It was designed for Windows only but open-source projects are increasing K4W's functionality to Mac as well.	http://www.microsoft.com/en-us/kinectforwindows/
SOFTWARE:		
Greenfoot 2.2.1	Greenfoot is a graphical Java environment developed by Oracle and the University of Kent. It offers simplified functionality aimed toward game creation and interactivity. The API is explained in Appendix A.	http://www.greenfoot.org
CONNECTIVITY PROGRAMS:		
OpenNI 1.5.4 stable 32-bit binary	OpenNI provides a framework for working with the NITE middleware.	http://www.openni.org/Downloads/OpenNIModules.aspx
NITE 1.5.2 stable 32-bit middleware binary	NITE is a middleware program that grants the user access to the special functionality of the Kinect camera such as the IR sensor, depth map, and point cloud.	http://www.openni.org/Downloads/OpenNIModules.aspx
SensorKinect093 5.1.2 32-bit hardware binary	SensorKinect is a driver that will allow your computer to "see" the Kinect camera properly. Avin's SensorKinect driver is a forked version of OpenNI driver source code made to work with Kinect for Windows. It should work for Mac OS X.	https://github.com/avin2/SensorKinect/downloads
Kinect Server for Greenfoot	This is an additional program that Greenfoot uses. It runs in the background. Without it, Greenfoot will not be able to access the Kinect camera, even if everything else is installed properly.	http://www.greenfoot.org/doc/kinect

"PEGGY 2" LED DISPLAY MATRIX

The Peggy 2 is a printed circuit board (PCB) with 625 multiplexed LEDs controlled by two LED drivers, two 74HC154 demultiplexers, 25 MOSFET transistors, and 25 620 ohm resistors, along with other components. Multiplexing allows the LEDs on the board to be expanded beyond the limited pin availability of the ATMega328 microcontroller and also allows each LED to be individually addressable. In essence, multiplexing connects the anodes of LEDs to the columns and the cathodes of the LEDs to the rows, creating a unique connection for every LED. I recommend Wikipedia's article on multiplexing if you would like to learn more. On the Peggy 2, these columns and rows of LEDs are controlled by the LED driver chip and the 74HC154 demultiplexer respectively. The LEDs are also dimmable through 16 shades of gray with pulse width modulation (PWM). In Peggy's case of PWM, the MOSFET transistors essentially send 'on' and 'off' signals into the matrix. The ratio of time between these signals is known as a duty cycle. If 'on' is a constant signal, then the duty cycle is 100% and the LED is white. If there is equal time between on and off signals, then the duty cycle is 50% and the LED is medium gray. The longer the time between 'on' pulses, the dimmer the LED.

The Atmel ATMega328 microcontroller on the Peggy 2 (and the Arduino Uno, as a reference) has a certain computer architecture (organization of memory and CPU) known as AVR. All of the 'instructions' for how AVR works are contained in a small program burned into the flash memory of the microcontroller known as a bootloader. This bootloader allows a user the ability to upload a sketch through the Arduino IDE to the microcontroller without any additional hardware other than the included USB FTDI TTL-232 cable which converts USB into serial. The bootloader on the ATMega328 can be erased and a custom bootloader can be created, should you decide to develop in another software environment and would still like to test your program on the board or are just feeling adventurous. To recap, the reason Peggy 2 can be programmed inside the Arduino IDE is because the Arduino and Peggy 2 share the same microcontroller with the same basic bootloader program. To learn more about bootloaders and what is in the chip visit http://arduino.cc/en/Guide/Environment?from=Tutorial.Bootloader

There is also an (as of yet) unused prototyping breadboard area on the board (see Figure 2). The holes are hooked up to important pins on the microcontroller and offer a space to 'break away' from the board to create an installation elsewhere, while still operating with similar functionality to the Peggy 2 board itself. It should be noted though that the jumpers installed at P1 and P2 are a special modification known as a 'serial hack' which allows the serial port to remain open on the board. In a normal installation, these jumpers would be installed at P3 and P4 on the board. These jumpers can be desoldered and moved, but a modified Peggy 2 library is available for the 'serial hack' at http://code.google.com/p/peggy/ under Peggy2Serial.41.zip. This repository also contains other examples of code for the Peggy 2 as well. For more information on the Peggy 2 or to view the instructions online (they are also in the Peggy 2 box) visit: http://s3.amazonaws.com/evilmadscience/KitInstrux/peggy2.3.pdf

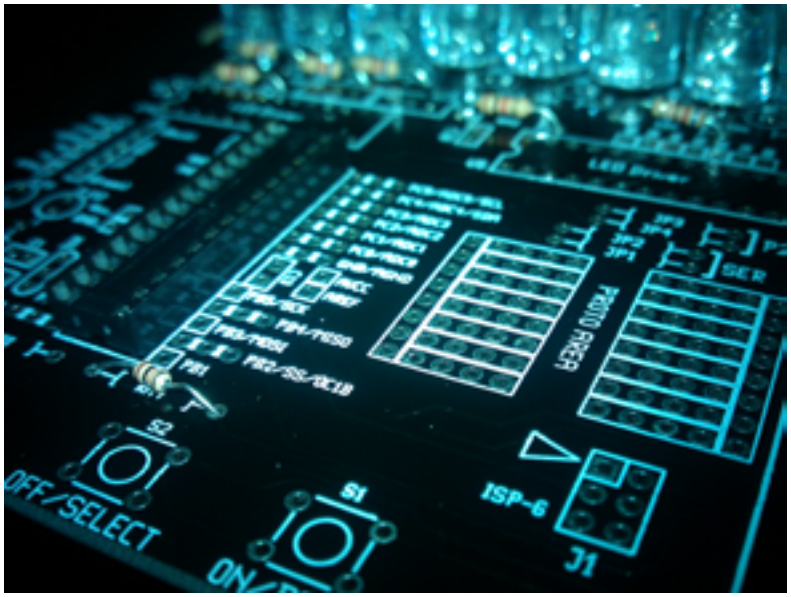


FIGURE 2: PROTOTYPING AREA ON PEGGY 2

ARDUINO & PROCESSING:

From the Arduino website (arduino.cc): "Arduino is an open-source electronics prototyping platform based on flexible, easy-to-use hardware and software. It's intended for artists, designers, hobbyists, and any-

one interested in creating interactive objects or environments." The Arduino Uno runs on an Atmel ATmega328 microcontroller with 32KB of Flash memory, and a 16 MHz clock speed. The bootloader written to this microcontroller allows it to be reprogrammed in the Arduino Integrated Development Environment (IDE). This gives the user access to Java level functions, allowing for advanced coding of the board. Libraries need to be imported however for this code to work:

```
#include ‹avr/io.h›
#include ‹avr/interrupt.h›
#include ‹avr/sleep.h›
#include ‹avr/pgmspace.h›
#include ‹stdio.h›
```

AVR refers to in-system code for the microcontroller. It is more basic than Arduino and is necessary to access the proper features on the ATmega328. To learn more about the Arduino API visit the Reference Page at http://arduino.cc/en/Reference/HomePage.

The Arduino IDE looks more or less identical to the Processing IDE. This is because Arduino is based on the Processing IDE. The Arduino IDE is, relative to Processing, more limited in functionality because the Arduino IDE focuses more on controlling the flow of analog-to-digital and digital-to-analog signals that guide sensors and hardware. Processing, on the other hand, is not only capable of controlling an Arduino, but also developing audio/visual and digital interaction with the user through the computer. Processing is also Java based and offers high level functions that would take tens of lines of code if not more to represent in pure Java. For example,

```
PImage peggyImage = new PImage(25,25);
```

creates an image with 25 by 25 cells. Unspecified, each cell equates to a single pixel, even this code can be customized with relative ease though, and PImage can come to represent, as in my case, the 25x25 LED matrix imitating 10mm LEDs. Additionally, writing something such as:

```
Serial peggyPort;
```

in Processing creates a variable that has direct access to the appropriate serial input/output line. It should be stated though that as in Arduino, these features as well as many others rely on the inclusion of libraries within your code. For our purposes,

```
import processing.video.*;
import processing.serial.*;
```

will accomplish this task. These two lines of code import the Processing video library and the Processing serial library. Libraries provide the definitions for high level functionality such as PImage and Serial.

The preceding are just a couple of the numerous examples of how Processing makes audio/visual coding more accessible and more efficient. To learn more about the Processing API visit the Reference Page at http://processing.org/reference/.

As a final note, the code written in Arduino could have been written into Processing as well. The advantage of writing a separate code in Arduino is Arduino's direct access to the chip, whereas in Processing the same access ports would have to be explicitly declared.

MICROSOFT KINECT FOR WINDOWS

The Microsoft Kinect for Windows debuted in early 2012 and offered increased functionality over the

First Generation Xbox Kinect. The Kinect for Windows offers face tracking, near-mode, and increased depth sensitivity over its predecessor. Both cameras still have IR cameras, depth sensors, VGA cameras, microphones, and a motorized base, but the Microsoft Kinect SDK (the official programming environment for the Kinect) only offers programming of the K4W. Since the official Microsoft Kinect SDK will not run on Mac OS X, the OpenNI framework, open-source as it is, did not initially support the Kinect for Windows. While it is capable of supporting the K4W now, it still lacks the control over the functionality of the K4W's additional functionality over the Xbox Kinect. This makes the K4W and the Xbox Kinect equal in the eyes of OpenNi. If a Windows computer and the Microsoft Kinect SDK were used to access the Kinect, more advanced features could be unlocked and used in our projects, although I doubt this route will ever be pursued. In the end, OpenNI will update itself to support the K4W better and to allow access to all of its functionality - it's only a matter of time.

GREENFOOT:

Greenfoot is a simplified and graphical Java environment designed for game creation and human-computer interaction. It focuses on Java classes by creating graphical objects representative of that class. Within each class, notes are written to help the user begin coding by themselves. To access a class, simply double click one of the object boxes on the right side of the environment.

Within each object/class there are a number of constructors and methods where the user inputs their desired Greenfoot API commands. By writing short, high level codes within these constructors and methods, the user is able to essentially create simple actions for objects, such as "eat leaves" or "play note." An extended Greenfoot API is in Appendix A for further reference.

CONNECTIVITY PROGRAMS:

Important Note: Check out http://www.greenfoot.org/doc/kinect for more information about this setup on Windows/Linux/Mac OS X. You will need to install certain C++ libraries or the full Visual Studio on Windows. If you are on OS X, you will need to make sure Xcode and MacPorts are both installed before installing any of the other software. Also, only unstable versions of OpenNI and NITE are available for Mac OS X. These should still work fine. The SensorKinect driver is special download. The original OpenNI SensorKinect driver does not support K4W so this special driver is needed in order to complete this set up successfully.

OpenNI is an open source framework and software that allows a user access to a Kinect camera (Xbox or K4W) that is hooked up to a computer via USB. OpenNI consists of the OpenNI framework, NITE middleware, and the SensorKinect driver. The framework itself allows the computer to access the NITE middleware. The NITE middleware accesses the various special features of the Kinect camera such as the IR sensor, the depth image, and the point cloud (see Figure 3). The SensorKinect driver is simply a driver - a program that allows the computer to understand the hardware being connected to it. To summarize, the SensorKinect driver tells the computer how to recognize the Kinect. The NITE middleware takes in the raw Kinect data and converts it into useful data, and the OpenNI framework allows for other programs to talk to the middleware.

The Kinect Server for Greenfoot operates like a library or a network drive. It lets Greenfoot connect to the camera in real-time. This Kinect Server is Greenfoot specific, however. If you work with Kinect in Processing, there is no "Kinect Server" to activate.

INSTRUCTIONS THROUGH THE PROCESS:

SEE PAGE 12.

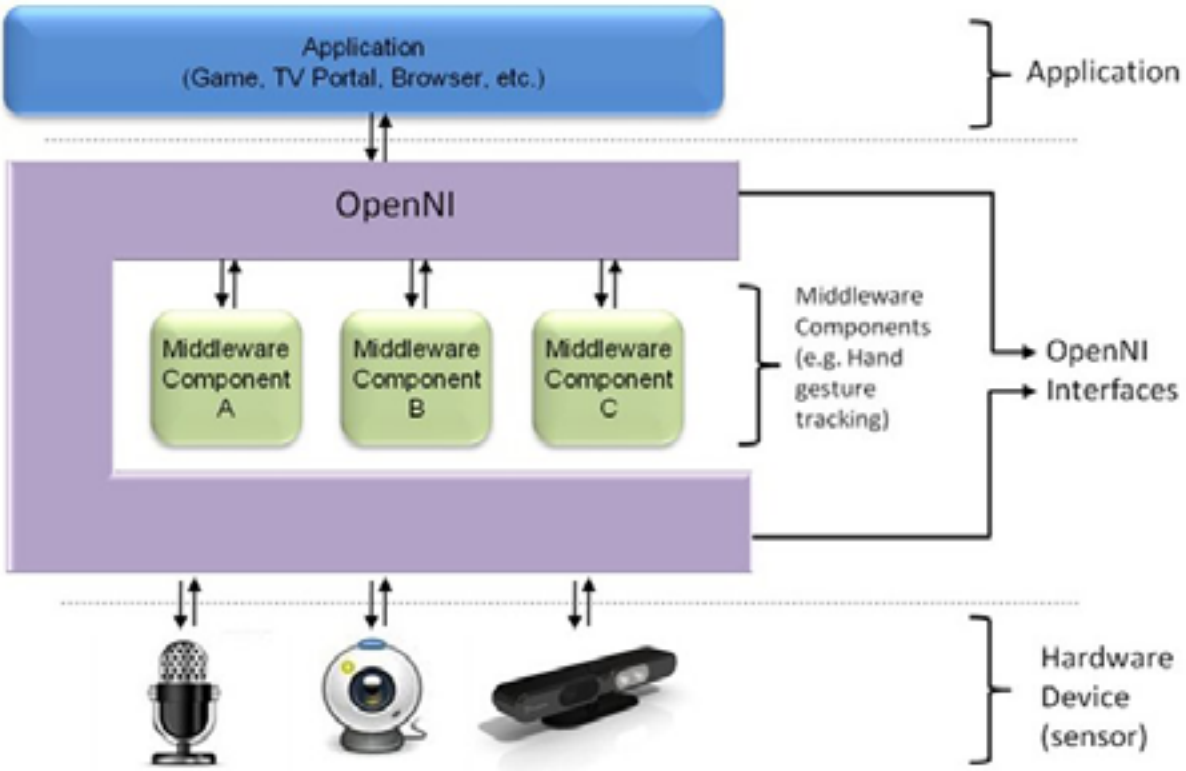


FIGURE 3: KINECT INTERFACE WITH OPENNI AND NITE

FINAL RESULT/THOUGHTS:

There are many things that went wrong with the project and numerous delays caused me to limit the scope of the project more than I would have liked. Difficulty with installation alone accounted for a couple weeks of work. Beyond creating my own LED matrix, I would have liked to incorporate the Greenfoot and Processing environments. With more time I'm sure all the data for either program could have been done in one. I would have also liked to explore the face-tracking feature of the Kinect. I would have also created more documentation for the process. Most references never made a distinction between either K4W or Xbox Kinect, yet the installations require different files. I would have also liked to explore firsthand the possibility of using a K4W on an Apple system. Since the LAES lab is Apple-centric any peripheral device should work with Apple computers. As of writing this, the K4W is supported, unstably, on Apple computers using Avin's SensorKinect093 as mentioned prior, Since OpenNI is supported in Apple, I imagine a driver will soon emerge that is capable of coupling the two. The Peggy 2 board is open-source and is perfect for prototyping and hacking.

Ultimately, this project was intended to develop my own knowledge and experience with physical computing, natural interaction, and social immersive media. To this end, the project was a success. After so much trial and error, a successful build, both in hardware and in software, is a welcome experience. I hope that these peripherals will offer future LAES students an opportunity to explore the growing world of physical computing and to pique their interest in such subjects in the same way as me. I believe that LAES has a bright future ahead of it and while I am sad to be leaving, behind the program is exciting energy, I am proud to leave my mark and to open the doors on what I believe is LAES's greatest strength.

ALTERNATIVES/NOTES/PROJECT IDEAS:

There are a number of ideas that I would have liked to incorporate but did not have the time. Hopefully these ideas could be used as inspiration for future students:

- The Greenfoot project could be expanded to incorporate other body parts. In this way the blue button hit areas could play several different tones based on which body part was in range of the hit area.
- Try to add an array of different Actors to act as buttons.
- Work on timing and precision for the hit areas.
- Add Z-index to check for a press inside Greenfoot.
- Use the Kinect's Infrared camera to output video to Peggy 2 in pitch black.
- Explore the program openCV (a cousin to openNI) to explore face tracking and computer vision (CV) with the Kinect.
- Use Kinect with the programs Synapse, Quartz Composer, Max for Live, or Max/MSP/Jitter.
- The program Unity is a 3D world renderer used for virtual interaction. It has been used in a 'Second Life' capacity for some time to help artists and designers create virtual interfaces for their art installations. It is also free of charge!

FINAL FILES & VIDEOS:

https://github.com/ultraJeff/seniorProject/ - Contains all the files for this project.
http://www.youtube.com/watch?v=Odf-gjsKCJE - "Peggy 2 Video Feed"
http://www.youtube.com/watch?v=aFImLMweq_A - "Kinect Greenfoot Interaction"

BIBLIOGRAPHY/FURTHER READING LINKS:

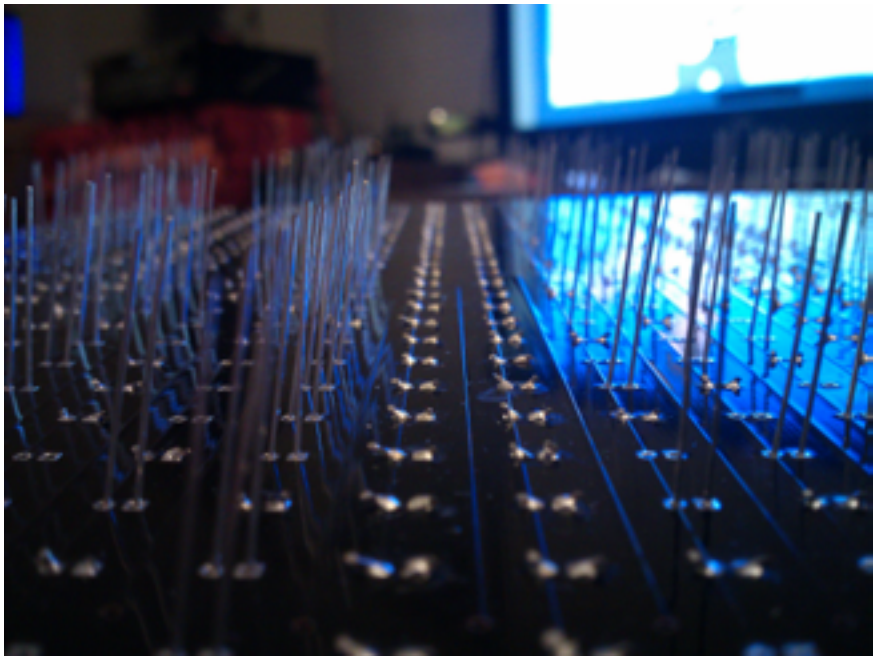
- WWW.PROCESSING.ORG - PROCESSING MAIN SITE. AN INVALUABLE RESOURCE
- WWW.ARDUINO.CC - ARDUINO MAIN SITE. ALSO INVALUABLE!
- PLAYGROUND.ARDUINO.CC - THE ARDUINO PLAYGROUND
- WWW.GREENFOOT.ORG - GREENFOOT'S MAIN SITE
- WWW.GREENFOOT.ORG/DOC/KINECT - INSTRUCTIONS FOR KINECT & GREENFOOT
- WWW.OPENNI.ORG - OPENNI AND NITE MAIN SITE
- WWW.GITHUB.COM/AVIN2/SENSORKINECT - MODIFIED SENSORKINECT DRIVER FOR K4W
- EN.WIKIPEDIA.ORG/WIKI/NEW_MEDIA - TO LEARN MORE ABOUT NEW MEDIA
- EN.WIKIPEDIA.ORG/WIKI/PHYSICAL_COMPUTING - TO LEARN MORE ABOUT PHYSICAL COMPUTING
- OPENCV.WILLOWGARAGE.COM/WIKI/ - TO LEARN ABOUT OPENCV
- S3.AMAZONAWS.COM/EVILMADSCIENTIST/SOURCE/P23SCHEM.PDF - THE PEGGY 2 SCHEMATIC
- S3.AMAZONAWS.COM/EVILMADSCIENCE/KITINSTRUX/PEGGY2.3.PDF - PEGGY 2 INSTRUCTIONS
- PLANETCLEGG.COM/PROJECTS/QC-PEGGY.HTML - QUARTZ COMPOSER WITH PEGGY 2 TO GET A LIVE VIDEO FEED
- SYNAPSEKINECT.TUMBLR.COM/POST/6307739137/ABLETON-LIVE - USING SYNAPSE WITH KINECT AND ABLETON
- UNITY3D.COM/UNITY/DOWNLOAD/ - TO LEARN MORE ABOUT UNITY 3D

PEGGY 2 BUILD:

CURRENT PROCESS/CONSTRUCTION LOG):

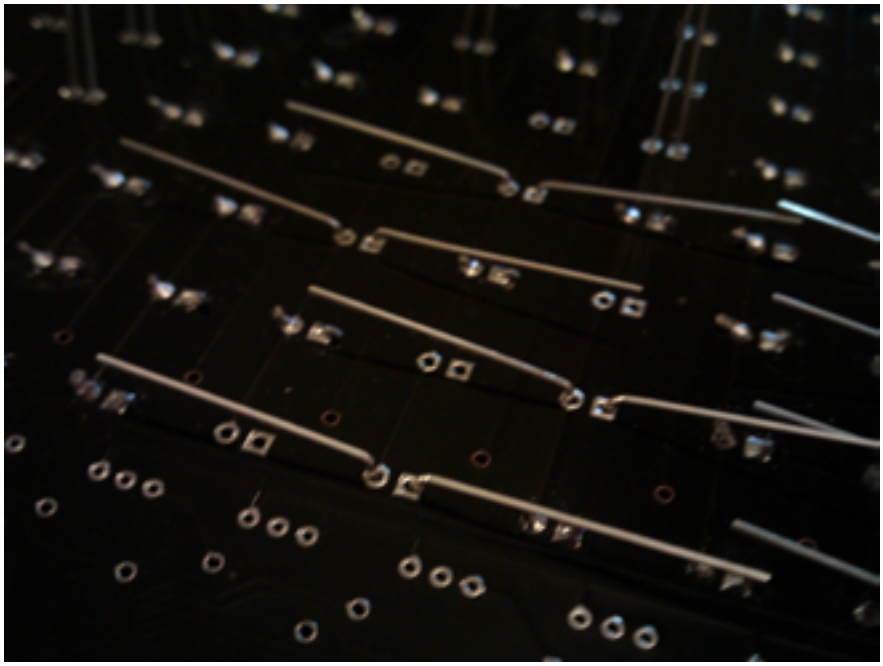
The Peggy 2 came as an unsoldered kit. These photos are meant as documentation to the process:

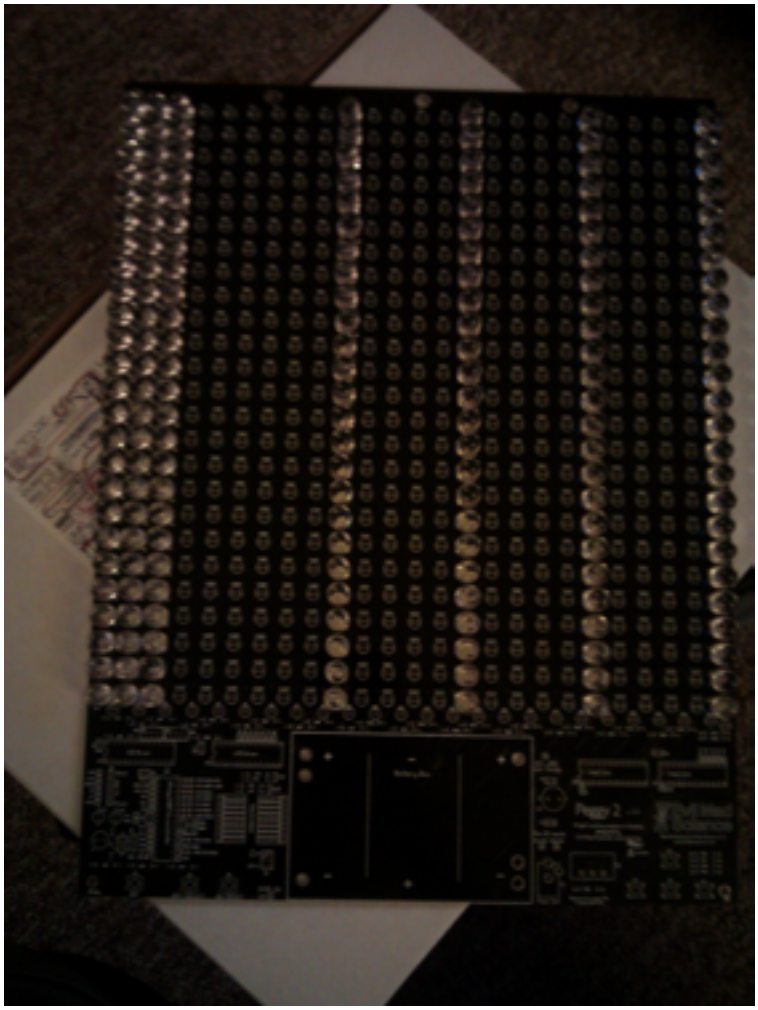
THE PEGGY 2 CAME AS AN UN-SOLDERED KIT. A GIANT BAG OF LEDS AND PARTS WITH A NICE SET OF INSTRUCTIONS (STILL IN THE BOX!)



THE LEDS HAD TO BE PLACED IN SEPARATE BATCHES OR ELSE THE WIRES WOULD CROSS WHEN THE LEADS WERE FOLDED FLAT.

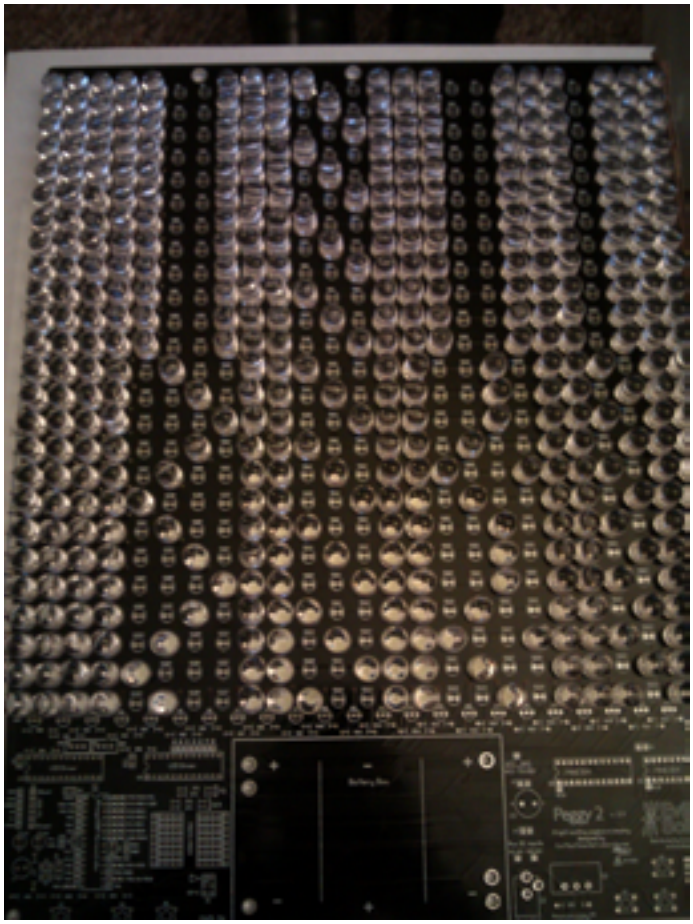
HERE IS A PICTURE OF THE LED LEADS FOLDED DOWN. THEY ARE OFFSET SO THAT THEY ARE EASY TO SOLDER WHILE STAY-ING IN PLACE.





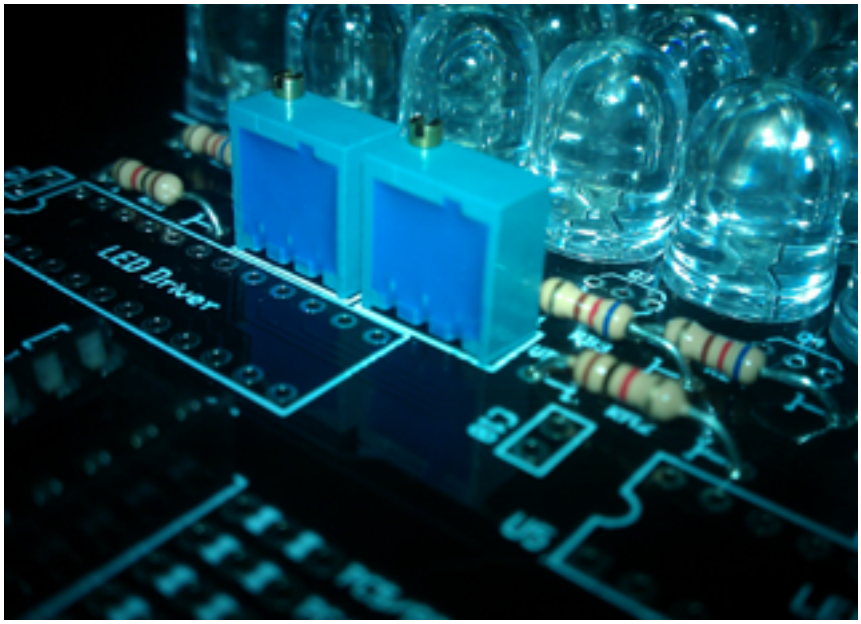
THE LED BOARD IS COMING TOGETHER!

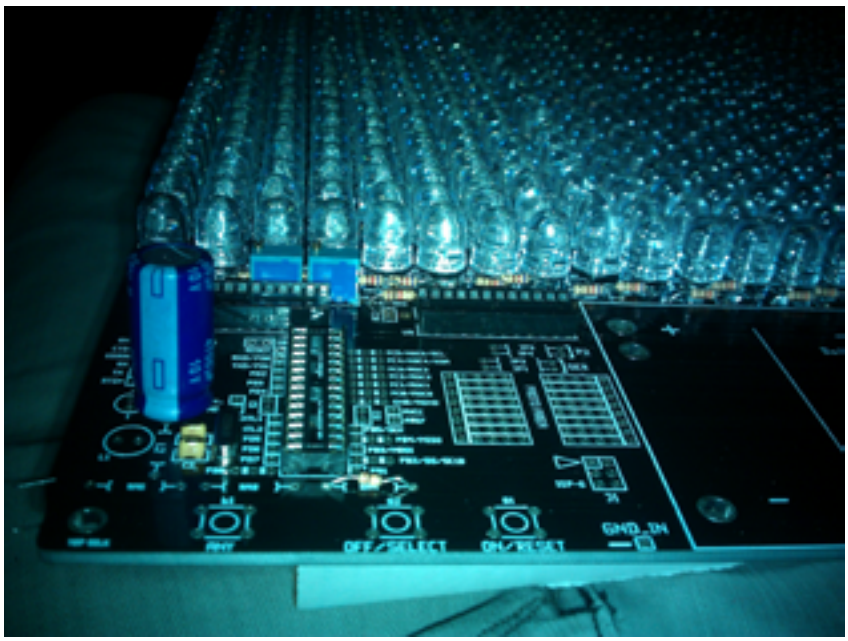
JUST A LITTLE WHILE LATER AND THE BOARD IS ALMOST COMPLETE.



TIME TO ADD 25 620-OHM RESISTORS TO CONTROL THE FLOW OF POWER TO THE COLUMNS OF LEDS.

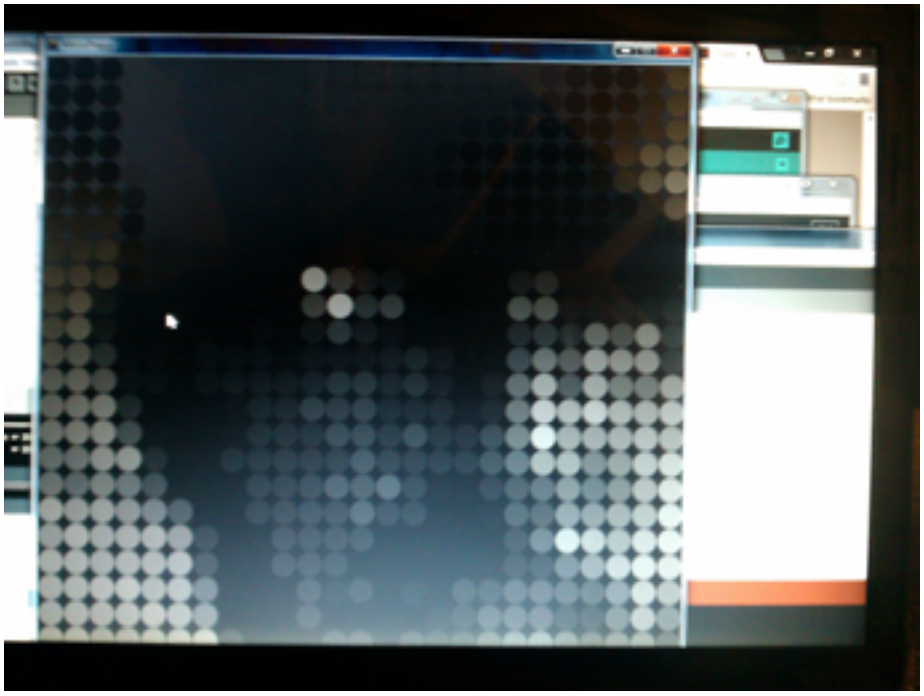
TWO LARGE POTEN-
TIOMETERS CONTROL
THE BRIGHTNESS
OF THE LEFT AND
RIGHT SIDE OF THE
DISPLAY. LOOK FOR
THESE!

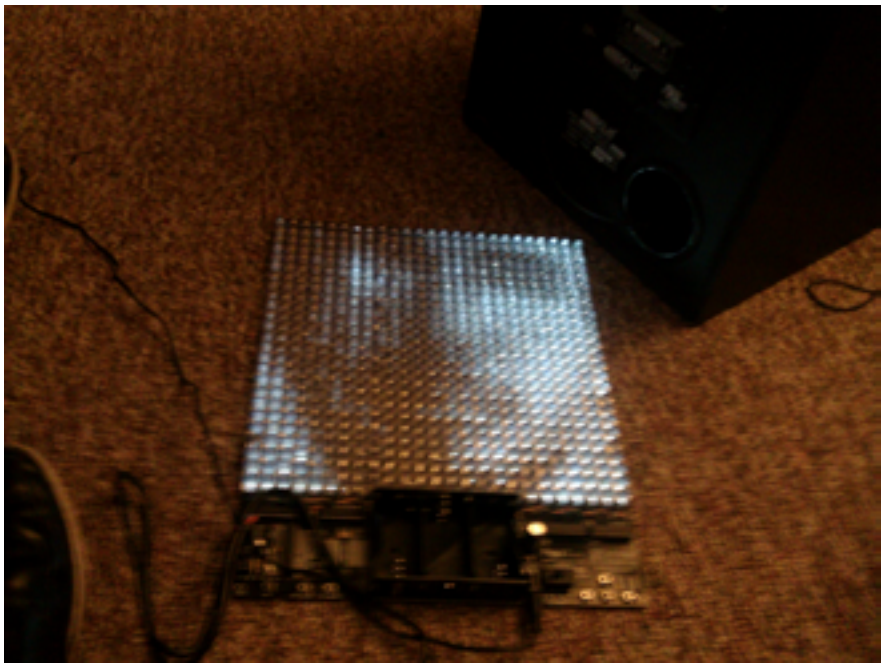




THE 'CHIP CLIPS'
ARE SOLDERED INTO
PLACE AS WELL AS
A 4400 MF CAPACI-
TOR TO KEEP POWER
CONSISTENT AS THE
SCREEN CONSTANTLY
UDPATES.

THIS IS THE VIDEO MIRROR FOR PEGGY CODE BEING RAN IN PROCESSING. THE CODE IS IN AP-PENDIX B.





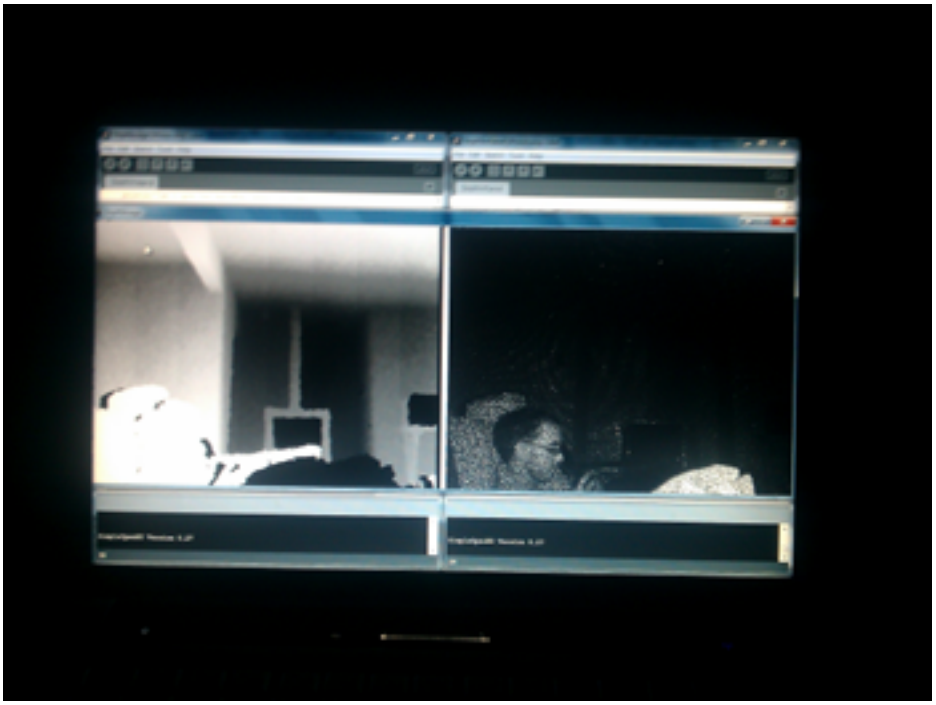
THIS IS THE FI-NAL RESULT ON THE PEGGY 2 DISPLAY. WHATEVER MOTION OF MY HEAD TURN-ING IS CAPTURED IN THE COMPUTER AND THEN EXPORTED TO THE PEGGY 2 OVER THE ACTIVATED 'SE-RIAL HACK' SERIAL LINE. THE CODE FOR THIS SECTION (THE ARDUINO) IS ALSO AVIALABLE UNDER RECEIVE SERIAL IN APPENDIX B

KINECT EXPLORATION & GREENFOOT BUILD:

CURRENT PROCESS/CONSTRUCTION LOG):

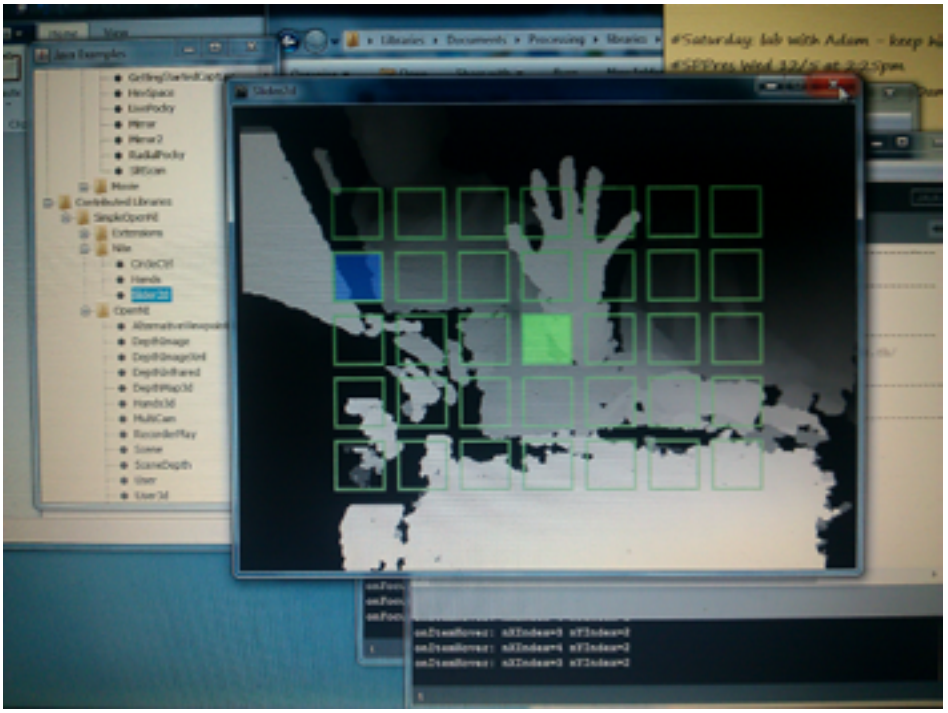
The Kinect for Windows offered a bevy of challenges when it came time to program it. This log is meant to document the functionality I discovered as well as the process of making my Kinect game in Greenfoot.

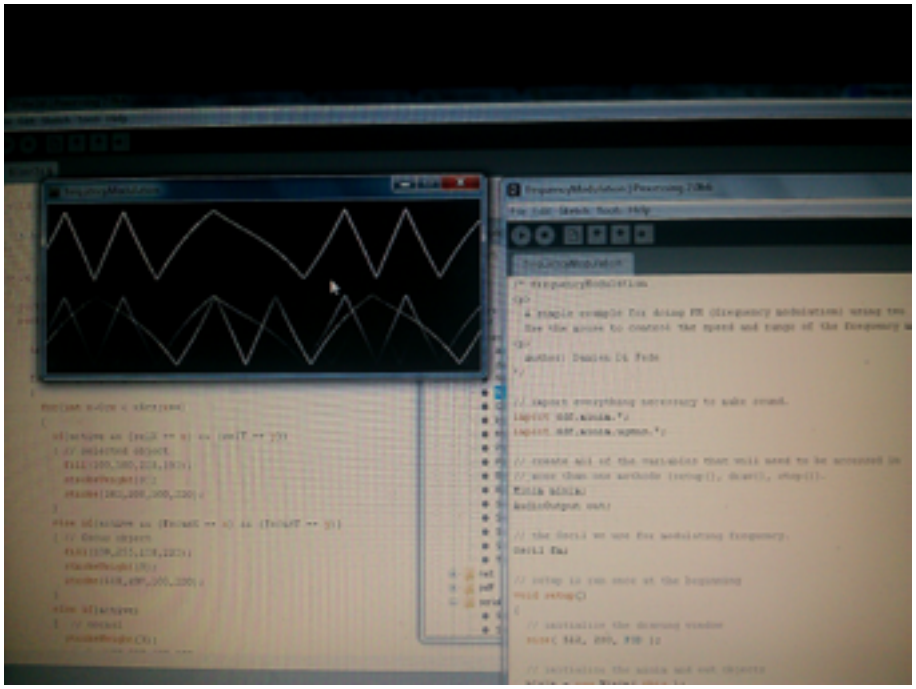
THE KINECT IR CAMERA IS VERY NOTICEABLE WHEN PHOTOGRAPHED. IT IS USUALLY MORE RED AND LESS NOTICEABLE.



I USED PRE-MADE PROCESSING SKETCHES AS I EXPLORED THE POSSIBILITIES OF WHAT THE KINECT CAMERA CAN DO. IN THIS PHOTO, THE DEPTH SENSOR AND THE IR CAMERA ARE DISPLAYED NEXT TO EACH OTHER.

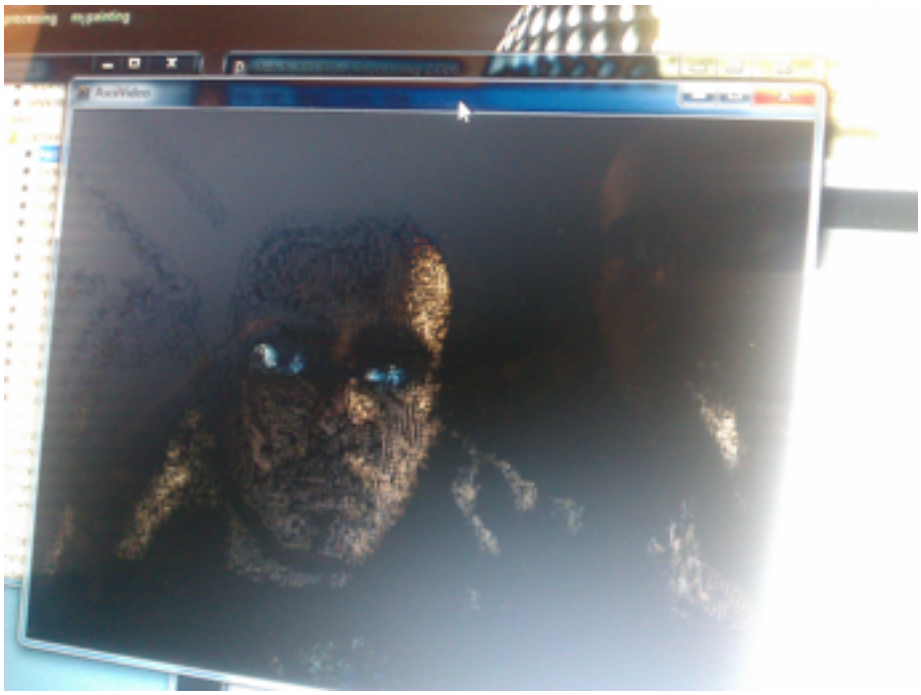
IN THIS PROCESSING SKETCH, THE USER'S HAND IS DETECTED IN RELATION TO SQUARES ON THE SCREEN. WHEN THE USER MOVES THEIR HAND FORWARD THROUGH THE Z-DEPTH (PRESS), THE SQUARE TURNS BLUE AND STAYS ACTIVATED.

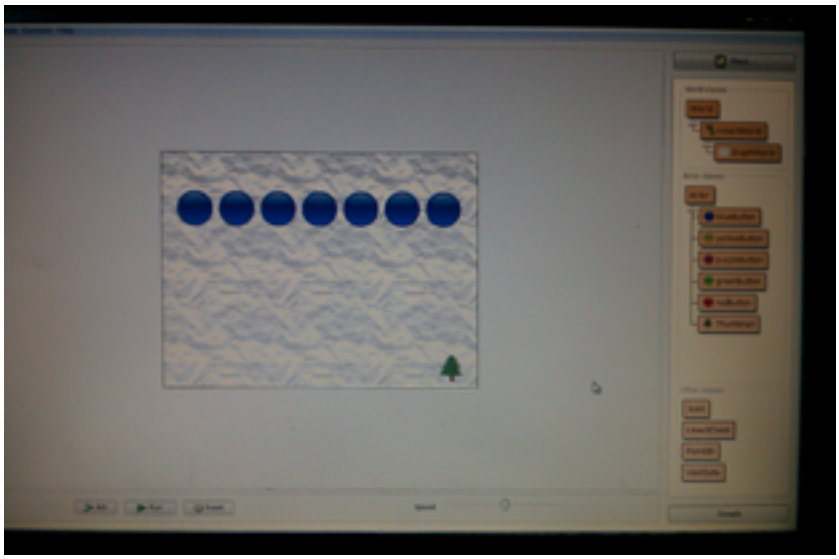




IN THIS PROCESSING SKETCH, AN OSCILLOSCOPE IS CREATED FROM SCRATCH AND DISPLAYED IN A WINDOW. AS THE USER MOVES THE MOUSE ACROSS THE SCREEN, THE FREQUENCY IS MODULATED AND A SOUND TONE IS WARPED.

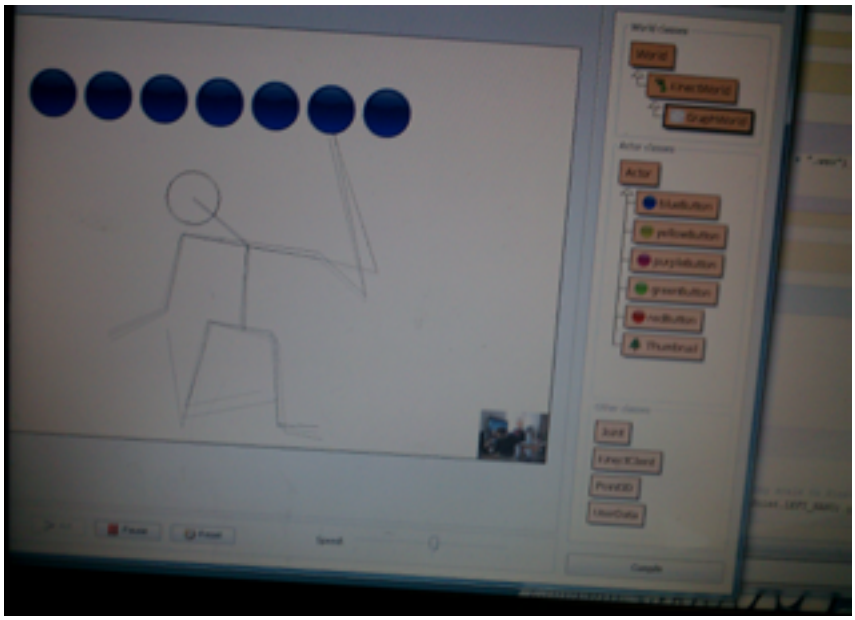
IN THIS PROCESSING SKETCH, THE WEBCAM TAKES IN VIDEO DATA WHICH IS THEN INTERPOLATED INTO ASCII CHARACTERS TO REPRESENT MY FACE.

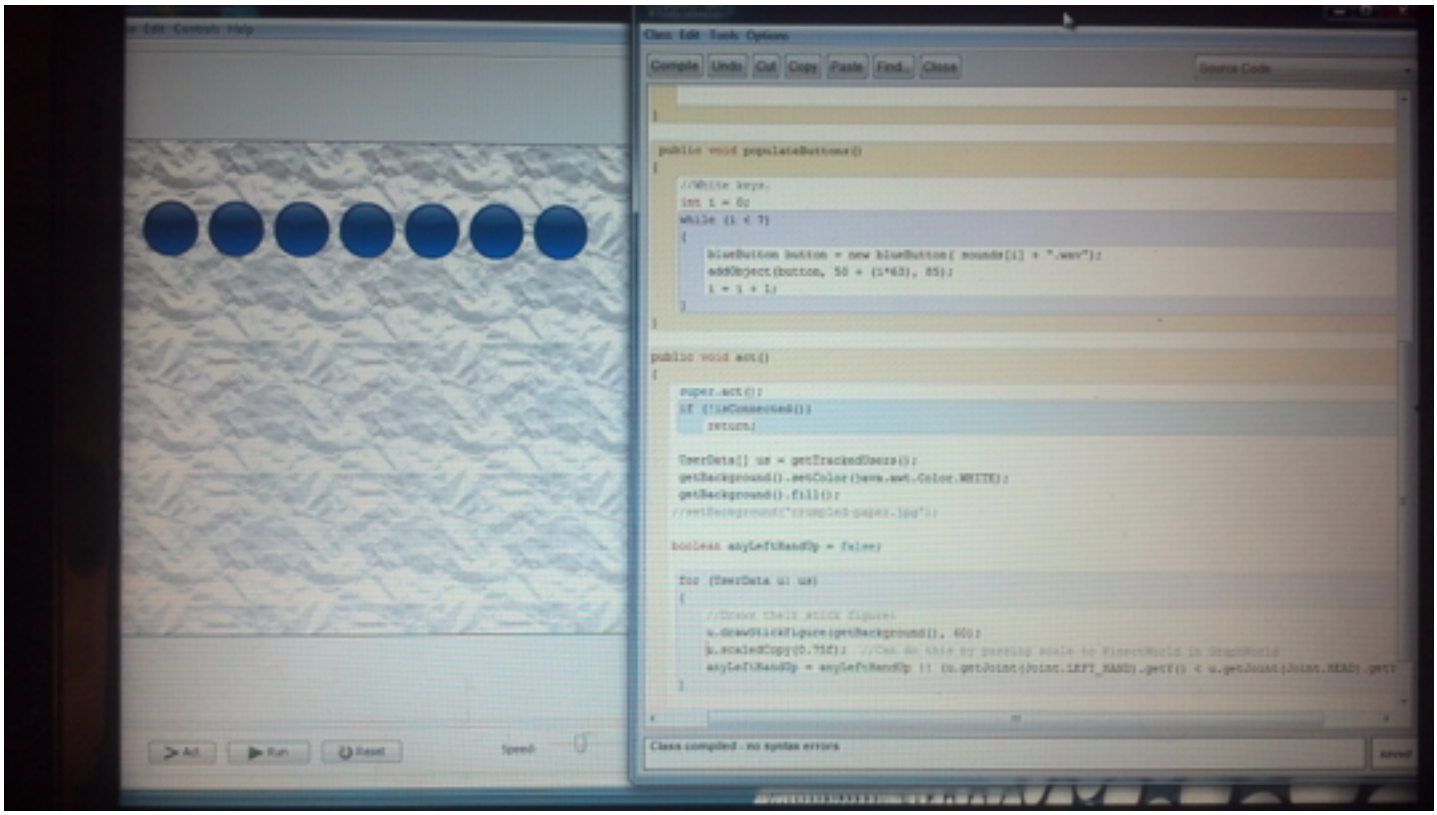




NOW INTO GREENFOOT! THIS IS THE DISPLAY OF THE KINECT MUSIC PROGRAM IN GREENFOOT. ALL OF THE CLASSES ARE LISTED TO THE RIGHT, THE GRAPHICAL WORK AREA IS ON THE LEFT.

THE GRAPHICAL WORK AREA IS NOW COMPILED. THE BACKGROUND IS ERASED AS THE SKELETON UPDATES (UNINTENTIONAL.) IN THE BOTTOM RIGHT CORNER OF THE WORK AREA, THE VGA CAMERA CAPTURES THE REAL IMAGE.





THE WORK AREA AND AN EXPANDED CLASS IN ONE. DOUBLE CLICKING ONE OF THE CLASSES IN THE CLASS LIST ON THE RIGHT EXPANDS TO THIS SCREEN WHERE THE USER CAN BEGIN TO CODE EACH CLASS/OBJECT FOR FUNCTIONALITY.

APPENDIX A:

Kinect
&
Greenfoot

1.Greenfoot API
2. Graphworld Code
3. Blue Button code

The Greenfoot API consists of six classes:

World	World methods are available to the world.	Greenfoot	Used to communicate with the Greenfoot environment itself.	GreenfootImage	For image presentation and manipulation.
Actor	Actor methods are available to all actor subclasses.	MouseInfo	Provide information about the last mouse event.	GreenfootSound	For controlling sound playback.

Class World	
`World(int worldWidth, int worldHeight, int cellSize)`	Construct a new world.
`void act()`	Act method for the world. Called once per act round.
`void addObject(Actor object, int x, int y)`	Add an Actor to the world.
`GreenfootImage getBackground()`	Return the world's background image.
`int getCellSize()`	Return the size of a cell (in pixels).
`Color getColorAt(int x, int y)`	Return the color at the center of the cell.
`int getHeight()`	Return the height of the world (in number of cells).
`List getObjects(Class cls)`	Get all the objects in the world.
`List getObjectsAt(int x, int y, Class cls)`	Return all objects at a given cell.
`int getWidth()`	Return the width of the world (in number of cells).
`int numberOfObjects()`	Get the number of actors currently in the world.
`void removeObject(Actor object)`	Remove an object from the world.
`void removeObjects(Collection objects)`	Remove a list of objects from the world.
`void repaint()`	Repaint the world.
`void setActOrder(Class... classes)`	Set the act order of objects in the world.
`void setBackground(GreenfootImage image)`	Set a background image for the world.
`void setBackground(String filename)`	Set a background image for the world from an image file.
`void setPaintOrder(Class... classes)`	Set the paint order of objects in the world.
`void started()`	Called by the Greenfoot system when execution has started.
`void stopped()`	Called by the Greenfoot system when execution has stopped.

Greenfoot API version 2.0.2 is supported since Greenfoot version 2.0.1.

Class Actor	
Actor()	Construct an Actor.
void **act**()	The act method is called by the Greenfoot framework to give objects a chance to perform some action.
protected void **addedToWorld**(World world)	This method is called by the Greenfoot system when the object has been inserted into the world.
GreenfootImage **getImage**()	Returns the image used to represent this Actor.
protected List **getIntersectingObjects**(Class cls)	Return all the objects that intersect this object.
protected List **getNeighbours**(int distance, boolean diagonal, Class cls)	Return the neighbours to this object within a given distance.
protected List **getObjectsAtOffset**(int dx, int dy, Class cls)	Return all objects that intersect the given location (relative to this object's location).
protected List **getObjectsInRange**(int r, Class cls)	Return all objects within range 'r' around this object.
protected Actor **getOneIntersectingObject**(Class cls)	Return an object that intersects this object.
protected Actor **getOneObjectAtOffset**(int dx, int dy, Class cls)	Return one object that is located at the specified cell (relative to this objects location).
int **getRotation**()	Return the current rotation of the object.
World **getWorld**()	Return the world that this object lives in.
int **getX**()	Return the x-coordinate of the object's current location.
int **getY**()	Return the y-coordinate of the object's current location.
protected boolean **intersects**(Actor other)	Check whether this object intersects another given object.
void **setImage**(GreenfootImage image)	Set the image for this object to the specified image.
void **setImage**(String filename)	Set an image for this object from an image file.
void **setLocation**(int x, int y)	Assign a new location for this object.
void **setRotation**(int rotation)	Set the rotation of the object.

Class GreenfootImage	
GreenfootImage(GreenfootImage image)	Create a GreenfootImage from another GreenfootImage.
GreenfootImage(int width, int height)	Create an empty (transparent) image with the specified size.
GreenfootImage(String filename)	Create an image from an image file.
GreenfootImage(String string, int size, Color foreground, Color background)	Create an image with the given string drawn as text using the font size, foreground color and background color.
void **clear**()	Clear the image.

(continued next page)

Class GreenfootImage	
`void` **`drawImage`**`(GreenfootImage image, int x, int y)`	Draws the given Image onto this image.
`void` **`drawLine`**`(int x1, int y1, int x2, int y2)`	Draw a line, using the current drawing color, between the points (x1, y1) and (x2, y2).
`void` **`drawOval`**`(int x, int y, int width, int height)`	Draw an oval bounded by the specified rectangle with the current drawing color.
`void` **`drawPolygon`**`(int[] xPoints, int[] yPoints, int nPoints)`	Draws a closed polygon defined by arrays of x and y coordinates.
`void` **`drawRect`**`(int x, int y, int width, int height)`	Draw the outline of the specified rectangle.
`void` **`drawString`**`(String string, int x, int y)`	Draw the text given by the specified string, using the current font and color.
`void` **`fill`**`()`	Fill the entire image with the current drawing color.
`void` **`fillOval`**`(int x, int y, int width, int height)`	Fill an oval bounded by the specified rectangle with the current drawing color.
`void` **`fillPolygon`**`(int[] xPoints, int[] yPoints, int nPoints)`	Fill a closed polygon defined by arrays of x and y coordinates.
`void` **`fillRect`**`(int x, int y, int width, int height)`	Fill the specified rectangle.
`BufferedImage` **`getAwtImage`**`()`	Returns the BufferedImage that backs this GreenfootImage.
`Color` **`getColor`**`()`	Return the current drawing color.
`Color` **`getColorAt`**`(int x, int y)`	Return the color at the given pixel.
`Font` **`getFont`**`()`	Get the current font.
`int` **`getHeight`**`()`	Return the height of the image.
`int` **`getTransparency()`**	Return the transparency of the image (range 0 to 255).
`int` **`getWidth`**`()`	Return the width of the image.
`void` **`mirrorHorizontally`**`()`	Mirror the image horizontally (flip around the x-axis).
`void` **`mirrorVertically`**`()`	Mirror the image vertically (flip around the y-axis).
`void` **`rotate`**`(int degrees)`	Rotates this image around the center.
`void` **`scale`**`(int width, int height)`	Scales this image to a new size.
`void` **`setColor`**`(Color color)`	Set the current drawing color.
`void` **`setColorAt`**`(int x, int y, Color color)`	Sets the color at the given pixel to the given color.
`void` **`setFont`**`(Font f)`	Set the current font.
`void` **`setTransparency`**`(int t)`	Set the transparency of the image (range 0 to 255).
`String` **`toString`**`()`	Return a string representation of this image.

Class Greenfoot	
Greenfoot()	Constructor.
static void **delay**(int time)	Delay execution by a number of time steps. The size of one time step is defined by the speed slider.
static String **getKey**()	Get the most recently pressed key since the last time this method was called.
static MouseInfo **getMouseInfo**()	Return an object with information about the mouse state.
static int **getRandomNumber**(int limit)	Return a random number between 0 (inclusive) and limit (exclusive).
static boolean **isKeyDown**(String keyName)	Check whether a given key is currently pressed down.
static boolean **mouseClicked**(Object obj)	True if the mouse has been clicked on the given object.
static boolean **mouseDragEnded**(Object obj)	True if a mouse drag has ended.
static boolean **mouseDragged**(Object obj)	True if the mouse has been dragged on the given object.
static boolean **mouseMoved**(Object obj)	True if the mouse has been moved on the given object.
static boolean **mousePressed**(Object obj)	True the mouse has been pressed on the given object.
static void **playSound**(String soundFile)	Play sound from a file.
static void **setSpeed**(int speed)	Set the speed of the simulation execution.
static void **start**()	Run (or resume) the simulation.
static void **stop**()	Stop the simulation.

Class MouseInfo	
Actor **getActor**()	Return the actor (if any) that the current mouse behaviour is related to.
int **getButton**()	The number of the pressed or clicked button (if any).
int **getClickCount**()	The number of mouse clicks of this mouse event.
int **getX**()	The current x position of the mouse cursor.
int **getY**()	The current y position of the mouse cursor.
String **toString**()	Return a string representation of this mouse event info.

Class GreenfootSound	
GreenfootSound(String filename)	Create a new sound from the given file.
boolean **isPlaying**()	True if the sound is currently playing.
void **pause**()	Pauses the current sound if it is currently playing.
void **play**()	Start playing this sound.
void **playLoop**()	Play this sound repeatedly in a loop.
void **stop**()	Stop playing this sound if it is currently playing.
String **toString**()	Returns a string representation of this sound containing the name of the file and whether it is currently playing or not.

The remaining API table I created from premade classes within Greenfoot to showcase the additional functionality when used with a Kinect.

Class Point3D	
Point3D(float x, float y, float z)	Creates a simple three-dimensional point class
float getX()	Gets the X coord of Point3D
float getY()	Gets the Y coord of Point3D
float getZ()	Gets the Z coord of Point3D
Point3D midpoint(Point3D o)	Gets a new point that is halfway between this point and the given parameter
Point3D scaledCopy (float scale)	Gets a copy of this point with x, y, and z scaled by the given factor
String toString()	Outputs the value of Point3D to the screen.

Class KinectWorld	
KinectWorld()	Constructs a KinectWorld that is 640 by 480 pixels, with a full-sized thumbnail (RGB) image available, but no depth information.
KinectWorld(double scale, boolean depth)	Constructs a KinectWorld that is scaled and may have depth available.
Kinectworld(int thumbnailWidth, int thumbnail-Height, double scale, boolean depth)	Constructor for objects of class KinectWorld.
void act()	The world's act method: updates information from the Kinect.
protected void disconnect()	Disconnects from the KinectServer.
UserData[] getAllUsers()	Gets all users currently in front of the Kinect.
GreenfootImage getCombinedUserImage	Gets an image featuring a cut-out of all the users in the scene.
short getDepthAt(int x, int y)	Gets the depth value at a given location in the world.
short getMaxDepth()	Gets the maximum depth value.
GreenfootImage getThumbnail()	Gets the current thumbnail (VGA image)
GreenfootImage getThumbnailUnscaled()	Gets the actual thumbnail, without any scaling applied.
UserData getTrackedUser()	Gets the first user who is being tracked.
UserData[] getTrackedUsers()	Gets only those users who are currently being tracked
UserData getUser	Gets the UserData for a user with the specified identifier.
boolean isConnected()	Checks whether we are currently connected properly to a functioning KinectServer.

Class KinectClient	
KinectClient()	Creates a KinectClient that attempts to connect to a server on the local machine, and that uses a full-size image from the Kinect feed (640x480) and does not receive depth information.
KinectClient(int thumbnailWidth, int thumbnail-Height, boolean depth)	Creates a KinectClient that attempts to connect to a server on the local machine.
KinectClient(int thumbnailWidth, int thumbnail-Height, boolean depth, java.lang.String host)	Creates a KinectClient that attempts to connect to the specified server ("127.0.0.1" by default).
void disconnect()	Disconnects from the server.
GreenfootImage getCombinedUserImage()	Gets an image of all the users the sensor currently sees.
short getMaxDepth()	Gets the maximum depth value (if depth is turned on, 0 if there is a problem).
short[] getRawDepth()	Gets the depth array (or null if not available).
GreenfootImage getThumbnail()	Gets a picture of what the sensor currently sees.
UserData getUser(int userId)	Gets the user data for a particular user id.
UserData[] getUsers()	Gets the user data for all users detected by the sensor (who may or may not have their skeletons currently tracked).
boolean isConnected()	Indicates whether the client is currently connected to the server.
void update()	Attempts to read the latest update from the server.

Class UserData	
UserData(int id)	For internal use only
void drawStickFigure(greenfoot.GreenfootImage img, int headradius)	Draws the user's joint data as a stick figure onto the given image.
Joint[] getAllJoints()	Gets the position of all the joints for the user.
int getHighestJoint()	Gets the index of the joint that is vertically highest.
int getId()	Gets the identifer of this user.
GreenfootImage getImage()	Gets the current user outline image.
int getImageX()	The left of the image returned by getImage().
int getImageY()	The top of the image returned by getImage().
Joint getJoint(int index)	Gets the joint associated with a given index.
int getNearestJoint()	Gets the index of the joint that is nearest to the sensor.
boolean isCalibrating()	Returns true if the user is currently calibrating.
boolean isTracking()	Returns true if the user is being tracked.

Class UserData	
UserData scaledCopy(float scale)	Makes a scaled copy of this UserData where the image and all the screen positions for joints are scaled by the given factor.
void setImage(greenfoot.GreenfootImage img, int x, int y)	For internal use only.
void setJoint(int j, Joint joint)	For internal use only.
void setState(int state)	For internal use only.
boolean startedCalibrating()	Returns true if isCalibrating() currently returns true, but would have returned false before the last update from the server.
boolean startedTracking()	Returns true if isTracking() currently returns true, but would have returned false before the last update from the server.

t

Class Joint	
Joint(int joint, float confidence, Point3D positionWorld, Point3D positionScreen)	For internal use only.
Comparator‹Joint› compareX()	Compare the X value of two joints.
Comparator‹Joint› compareY()	Compare the Y value of two joints.
Comparator‹Joint› compareZ()	Compare the Z value of two joints.
float getConfidence()	Gets the confidence value associated with a joint (a number between 0 and 1, where 0 is least confidence and 1 is most confidence).
int getJointIndex()	Gets the joint index for this joint
Point3D getPhysicalPosition()	Gets the physical position of the joint in the world as a 3D point.
Point3D getScreenPosition()	Gets the position of this joint on screen as a 3D point.
int getX()	Gets the X coordinate of this joint on the screen.
int getY()	Gets the Y coordinate of this joint on the screen.
static int maxJointBy(Joint[] joints, java.util.Comparator‹Joint› cmp)	Gets the maximum joint according to the given comparator.
static int minJointBy(Joint[] joints, java.util.Comparator‹Joint› cmp)	Gets the minimum joint according to the given comparator.
Joint scaledCopy(float scale)	Gets a scaled copy of the Joint where all the screen positions are scaled by the given factor.
String toString()	Outputs to a string.
FIELDS USED IN CLASS JOINT (ALL static int):	
ELBOW	FOOT
HAND	HEAD
HIP	KNEE

Class Joint	
LEFT	LEFT_ELBOW
LEFT_FOOT	LEFT_HAND
LEFT_HIP	LEFT_KNEE
LEFT_SHOULDER	NECK
NUM_JOINTS	RIGHT
RIGHT_ELBOW	RIGHT_FOOT
RIGHT_HAND	RIGHT_HIP
RIGHT_KNEE	RIGHT_SHOULDER
SHOULDER	TORSO

```
import greenfoot.*;  // (World, Actor, GreenfootImage, Greenfoot and MouseInfo)


/**
 *GraphWorld::
 * Creates the visual environment with Greenfoot and populates with buttons and a thumbnail
 * The background issue seems like an easy fix for anyone daring enough to go forward.
 *
 * @author Jeff Franklin
 * @version 12/3/12
 */
public class GraphWorld extends KinectWorld
{
   private static final int THUMBNAIL_WIDTH = 80;
   private static final int THUMBNAIL_HEIGHT = 60;
   public String[] sounds = {“2c”, “2d”, “2e”, “2f”, “2g”, “2a”, “2b”, “3c”, “3d”, “3e”, “3f”, “3g”, “3a”, “3b”};

   public GraphWorld()
   {
      super(THUMBNAIL_WIDTH, THUMBNAIL_HEIGHT, 1.0, false);

      final int width = getWidth();
      final int height = getHeight();

      //addObject(new blueButton(“2c.wav”),width/2,height/3); Can add a unique soundFile to each button if de-
sired
      populateButtons();
      addObject(new Thumbnail(), width - THUMBNAIL_WIDTH/2, height - THUMBNAIL_HEIGHT/2);


   }

    public void populateButtons()
   {
      //White keys.
      int i = 0;
      while (i < 7)
      {
         blueButton button = new blueButton( sounds[i] + “.wav”);
         addObject(button, 50 + (i*63), 85);
         i = i + 1;
      }
   }

   public void act()
   {
      super.act();
      if (!isConnected())
         return;
```

```
        UserData[] us = getTrackedUsers();
        getBackground().setColor(java.awt.Color.WHITE);
        getBackground().fill();
       //setBackground("crumpled-paper.jpg");

       boolean anyLeftHandUp = false;

        for (UserData u: us)
        {
           //Draws their stick figure:
           u.drawStickFigure(getBackground(), 60);
           //u.scaledCopy(0.75f);      //For making the skeleton fit the screen better?
           anyLeftHandUp = anyLeftHandUp || (u.getJoint(Joint.LEFT_HAND).getY() < u.getJoint(Joint.HEAD).
getY());
        }

    }
}
```

```
import greenfoot.*;  // (World, Actor, GreenfootImage, Greenfoot and MouseInfo)

/**
 * blueButton::
 * a musical hit area for your right hand when using the Kinect.
 *
 * @author Jeff Franklin
 * @version 12/3/12
 */
public class blueButton extends Actor
{
    private String sound;
    private int framesNearTarget;


    /**
     * Create a new button with a soundFile variable
     */
    public blueButton(String soundFile)
    {
        sound = soundFile;
    }

    /**
     * Button Kinect Interaction
     */
    public void act()
    {
        mousePlay();
        proxPlay();
    }

    /**
    * Right Hand Proximity Clip Play
    */
    public void proxPlay()
    {
        GraphWorld world = (GraphWorld)getWorld();
        UserData[] users = world.getTrackedUsers();
        for (UserData user : users)
        {
            Joint rightHand = user.getJoint(Joint.RIGHT_HAND);
            int xdist = rightHand.getX() - this.getX();
            int ydist = rightHand.getY() - this.getY();
            int straightDist = (int)Math.sqrt(xdist*xdist+ydist*ydist);


        if (straightDist < 20)
        {
```

```
            framesNearTarget += 1;
              if (framesNearTarget > 10)
                 {
                       play();
                       framesNearTarget = 0;
                 }
        }
              else
              {
                 // Their hand is not near:
                 framesNearTarget = 0;
              }
         }
    }

    /**
     * Clicking the clip plays it
     */
    public void mousePlay()
    {
       if(Greenfoot.mousePressed(this))
       {
          play();
       }
    }

    /**
     * Play a sound
     */
    public void play()
    {
       Greenfoot.playSound(sound);

    }
}
```

APPENDIX B:

Peggy 2
Live Video

1.Video Mirror for Peggy (Processing program)

2. Receive Serial (Arduino program)

```
/**
 * Video Mirror for Peggy
 * Based on the processing sketch “Mirror” by Daniel Shiffman.
 * Modified by Windell H Oskay to adapt video to Peggy 2.0
*
 *
 * Also incorporates code from Jay Clegg, http://planetclegg.com/projects/
 *
 *
 * To use this file, please see http://www.evilmadscientist.com/article.php/peggy2twi
 *
 * You will need to edit the line below with your actual serial port name listed; search for “CHANGE_HERE”
 *
 */


import processing.video.*;
import processing.serial.*;

Serial peggyPort;
PImage peggyImage = new PImage(25,25);
byte [] peggyHeader = new byte[]
   { (byte)0xde, (byte)0xad, (byte)0xbe,(byte)0xef,1,0 };
byte [] peggyFrame = new byte[13*25];



// Size of each cell in the grid
int cellSize = 9;
int cellSize2 = 34;  // was 20

// Number of columns and rows in our system
int cols, rows;
int ColLo, ColHi;
// Variable for capture device
Capture video;

int xDisplay,yDisplay;
int xs, ys;
float brightTot;
int pixelCt;
color c2;

int OutputPoint = 0;

//int GrayArray[625];
int[] GrayArray = new int[625];
```

```
int j;
byte k;

int DataSent = 0;


// NEEDS TO HAVE YOUR ACTUAL SERIAL PORT LISTED!!!

void setup() {
   String portName = Serial.list()[1];
   peggyPort = new Serial(this, portName, 115200);    // CHANGE_HERE

   smooth();
   noStroke();
//  size(640, 480, P3D);
  size(cellSize2*25, cellSize2*25, JAVA2D);
  //set up columns and rows
  cols = 25; //width / cellSize;
  ColLo = 4; //Was 4
  ColHi = 29; // Was 29
  rows = 25; //height / cellSize;
  colorMode(RGB, 255, 255, 255, 100);
  rectMode(CENTER);

  // Uses the default video input, see the reference if this causes an error
  String[] cameras = Capture.list();
  video = new Capture(this, cameras[0]);
//    video = new Capture(this, 320, 240, 15);  //Last number is frames per second
    video.start();
  background(0);

j = 0;
k = 0;

while (j < 625){


   GrayArray[j] = 4;//k;

k++;

if (k > 15)
  k = 0;

j++;
}
```

```
}



// render a PImage to the Peggy by transmitting it serially.
// If it is not already sized to 25x25, this method will
// create a downsized version to send...
void renderToPeggy(PImage srcImg)
{
  int idx = 0;

  PImage destImg = srcImg;

  // iterate over the image, pull out pixels and
  // build an array to serialize to the peggy
  for (int y =0; y < 25; y++)
  {
    byte val = 0;
    for (int x=0; x < 25; x++)
    {
      color c = destImg.get(x,y);
      int br = ((int)brightness(c))>>4;
      if (x % 2 ==0)
        val = (byte)br;
      else
      {
        val = (byte) ((br<<4)|val);
        peggyFrame[idx++]= val;
      }
    }
    peggyFrame[idx++]= val;  // write that one last leftover half-byte
  }

  peggyPort.write(peggyHeader);
  peggyPort.write(peggyFrame);
}




void draw() {
  if (video.available()) {
    video.read();
    video.loadPixels();

    background(0, 0, 0);
```

```
//int MaxSoFar = 0;
int thisByte = 0;
int e,k;
int br2;

int idx = 0;

PImage img2 = createImage(25, 25, ARGB);



  // Begin loop for columns

  k = 0;
  for (int i = ColLo; i < ColHi;i++) {
   // Begin loop for rows
   for (int j = 0; j < rows;j++) {

     // Where are we, pixel-wise?
     int x = i * cellSize;
     int y = j * cellSize;

     int loc = (video.width - x - 1) + y*video.width; // Reversing x to mirror the image

     // Each rect is colored white with a size determined by brightness
     color c = video.pixels[loc];

     pixelCt = 0;
     brightTot = 0;

     for (int xs = x; xs < (x + cellSize); xs++) {
      for (int ys = y; ys < (y + cellSize); ys++) {

       pixelCt++;
       loc = (video.width - xs - 1) + ys*video.width;
       c2 = video.pixels[loc];
       brightTot += brightness(c2);

     }
     }

     brightTot /= pixelCt;

     xDisplay = (i-ColLo)*cellSize2 + cellSize2/2;
     yDisplay = j*cellSize2 + cellSize2/2;



// Linear brightness:
```

```
        br2 = int(brightTot / 8);

         idx = (j)*cols + (i-ColLo);
        GrayArray[idx] = (int) br2;  //inverted image


       br2 = br2*8;

       fill(br2,br2,br2);         // 8-level with true averaging
         ellipse(xDisplay+1, yDisplay+1, cellSize2-1, cellSize2-1);

      img2.pixels[idx] = br2;
  k++;
  }
    }

    renderToPeggy(img2);

  }  // End if video available
} // end main loop
```

```
/*
*
* Copyright 2009 Windell H. Oskay.  All rights reserved.
*
*   This program is free software: you can redistribute it and/or modify
*   it under the terms of the GNU General Public License as published by
*   the Free Software Foundation, either version 3 of the License, or
*   (at your option) any later version.
*
*   This program is distributed in the hope that it will be useful,
*   but WITHOUT ANY WARRANTY; without even the implied warranty of
*   MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
*   GNU General Public License for more details.
*
*   You should have received a copy of the GNU General Public License
*   along with this program.  If not, see <http://www.gnu.org/licenses/>.
*/

/*


Credit: This program was written by Jay Clegg and released by him under
the GPL.  I've copied it to Arduino format.

Please see http://planetclegg.com/projects/QC-Peggy.html


*/
#include <avr/io.h>
#include <avr/interrupt.h>
#include <avr/sleep.h>
#include <avr/pgmspace.h>
#include <stdio.h>

////////////////////////////////////////////////////////////////////////////////////////////
// FPS must be high enough to not have obvious flicker, low enough that serial loop has
// time to process one byte per pass.
// 75-78 seems to be about the absolute max for me (with this code),
// but compiler differences might make this maximum value larger or smaller.
// any lower than 60 and flicker becomes apparent.
// note: further code optimization might allow this number to
// be a bit higher, but only up to a point...
// it *must* result in a value for OCR0A  in the range of 1-255

#define FPS 70


// 25 rows * 13 bytes per row == 325
#define DISP_BUFFER_SIZE 325
```

```
#define MAX_BRIGHTNESS 15

////////////////////////////////////////////////////////////////////////////////////////////

void displayInit(void)
{

	// set outputs to 0
	PORTC = 0;
	// leave serial pins alone
	PORTD &=  (1<<1) | (1<<0);

	// need to set output for SPI clock, MOSI, SS and latch.  Eventhough SS is not connected,
	// it must apparently be set as output for hardware SPI to work.
	DDRB =  (1<<DDB5) | (1<<DDB3) | (1<<DDB2) | (1<<DDB1);
	// SCL/SDA pins set as output
	DDRC =  (1<<DDC5) | (1<<DDC4);
	// set portd pins as output, but leave RX/TX alone
	DDRD |= (1<<DDD7) | (1<<DDD6) | (1<<DDD5) |(1<<DDD4) | (1<<DDD3) | (1<<DDD2);

	// enable hardware SPI, set as master and clock rate of fck/2
	SPCR = (1<<SPE) | (1<<MSTR);
	SPSR = (1<<SPI2X);

	// setup the interrupt.

	TCCR0A = (1<<WGM01); // clear timer on compare match

	TCCR0B = (1<<CS01); // timer uses main system clock with 1/8 prescale

	OCR0A  = (F_CPU >> 3) / 25 / 15 / FPS; // Frames per second * 15 passes for brightness * 25 rows
	TIMSK0 = (1<<OCIE0A);      // call interrupt on output compare match

	// set to row 0, all pixels off
	//setCurrentRow(0,0,0,0,0);
}


void setCurrentRow(uint8_t row, uint8_t spi1, uint8_t spi2, uint8_t spi3, uint8_t spi4)
{
	uint8_t portD;
	uint8_t portC;

	// precalculate the port values to set the row.
	if (row < 15)
	{
		row ++;
		portC = ((row & 3) << 4);
```

```
            portD = (row & (~3));
        }
        else
        {
            row = (row - 14) << 4;
            portC = ((row & 3) << 4);
            portD = (row & (~3));
        }

        // set all rows to off
        PORTD = 0;
        PORTC = 0;

        // set row values.  Wait for xmit to finish.
        // Note: wasted cycles here, not sure what I could do with them,
        // but it seems a shame to waste 'em.
        SPDR = spi1;
        while (!(SPSR & (1<<SPIF)))  { }
        SPDR = spi2;
        while (!(SPSR & (1<<SPIF)))  { }
        SPDR = spi3;
        while (!(SPSR & (1<<SPIF)))  { }
        SPDR = spi4;
        while (!(SPSR & (1<<SPIF)))  { }

        // Now that the 74HC154 pins are split to two different ports,
        // we want to flip them as quickly as possible.  This is why the
        // port values are pre-calculated



        PORTB |= (1<<1);

        PORTD = portD;
        PORTC = portC;


        PORTB &= ~((1<<1));
}



uint8_t frameBuffer[DISP_BUFFER_SIZE];
uint8_t * rowPtr;
uint8_t currentRow=26;
uint8_t currentBrightness=20;
uint8_t currentBrightnessShifted=20;
```

```
SIGNAL(TIMER0_COMPA_vect)
{


        // there are 15 passes through this interrupt for each row per frame.
        // ( 15 * 25) = 375 times per frame.
        // during those 15 passes, a led can be on or off.
        // if it is off the entire time, the perceived brightness is 0/15
        // if it is on the entire time, the perceived brightness is 15/15
        // giving a total of 16 average brightness levels from fully on to fully off.
        // currentBrightness is a comparison variable, used to determine if a certain
        // pixel is on or off during one of those 15 cycles.   currentBrightnessShifted
        // is the same value left shifted 4 bits:  This is just an optimization for
        // comparing the high-order bytes.

        currentBrightnessShifted+=16; // equal to currentBrightness << 4

        if (++currentBrightness >= MAX_BRIGHTNESS)
        {
                currentBrightnessShifted=0;
                currentBrightness=0;
                if (++currentRow > 24)
                {
                        currentRow =0;
                        rowPtr = frameBuffer;
                }
                else
                {
                        rowPtr += 13;
                }
        }

        // rather than shifting in a loop I manually unrolled this operation
        // because I couldnt seem to coax gcc to do the unrolling it for me.
        // (if much more time is taken up in this interrupt, the serial service routine
        // will start to miss bytes)
        // This code could be optimized considerably further...

        uint8_t * ptr = rowPtr;
        uint8_t p, a,b,c,d;

        a=b=c=d=0;

        // pixel order is, from left to right on the display:
        //  low order bits, followed by high order bits
        p = *ptr++;
        if ((p & 0x0f) > currentBrightness)             a|=1;
```

```
        if ((p & 0xf0) > currentBrightnessShifted)     a|=2;
        p = *ptr++;
        if ((p & 0x0f) > currentBrightness)            a|=4;
        if ((p & 0xf0) > currentBrightnessShifted)     a|=8;
        p = *ptr++;
        if ((p & 0x0f) > currentBrightness)            a|=16;
        if ((p & 0xf0) > currentBrightnessShifted)     a|=32;
        p = *ptr++;
        if ((p & 0x0f) > currentBrightness)            a|=64;
        if ((p & 0xf0) > currentBrightnessShifted)     a|=128;
        p = *ptr++;
        if ((p & 0x0f) > currentBrightness)            b|=1;
        if ((p & 0xf0) > currentBrightnessShifted)     b|=2;
        p = *ptr++;
        if ((p & 0x0f) > currentBrightness)            b|=4;
        if ((p & 0xf0) > currentBrightnessShifted)     b|=8;
        p = *ptr++;
        if ((p & 0x0f) > currentBrightness)            b|=16;
        if ((p & 0xf0) > currentBrightnessShifted)     b|=32;
        p = *ptr++;
        if ((p & 0x0f) > currentBrightness)            b|=64;
        if ((p & 0xf0) > currentBrightnessShifted)     b|=128;
        p = *ptr++;
        if ((p & 0x0f) > currentBrightness)            c|=1;
        if ((p & 0xf0) > currentBrightnessShifted)     c|=2;
        p = *ptr++;
        if ((p & 0x0f) > currentBrightness)            c|=4;
        if ((p & 0xf0) > currentBrightnessShifted)     c|=8;
        p = *ptr++;
        if ((p & 0x0f) > currentBrightness)            c|=16;
        if ((p & 0xf0) > currentBrightnessShifted)     c|=32;
        p = *ptr++;
        if ((p & 0x0f) > currentBrightness)            c|=64;
        if ((p & 0xf0) > currentBrightnessShifted)     c|=128;
        p = *ptr++;
        if ((p & 0x0f) > currentBrightness)            d|=1;
        //if ((p & 0xf0) > currentBrightnessShifted)  d|=2;

        setCurrentRow(currentRow, d,c,b,a);
}

////////////////////////////////////////////////////////////////////////////////////////////
// Serial IO routines
////////////////////////////////////////////////////////////////////////////////////////////


// must be 1 or zero. U2X gets set to this.

#define USART_DOUBLESPEED   1
```

```
#define _USART_MULT  (USART_DOUBLESPEED ? 8L : 16L )

#define CALC_UBBR(baudRate, xtalFreq) (  (xtalFreq / (baudRate * _USART_MULT))  - 1 )


void uartInit(unsigned int ubbrValue)

{

        // set baud rate
        UBRR0H = (unsigned char)(ubbrValue>>8);

   UBRR0L = (unsigned char)ubbrValue;

        // Enable 2x speed

        if (USART_DOUBLESPEED )

                UCSR0A = (1<<U2X0);



        // Async. mode, 8N1

   UCSR0C = /* (1<<URSEL0)| */(0<<UMSEL00)|(0<<UPM00)|(0<<USBS0)|(3<<UCSZ00)|(0<<UCPOL0);



        UCSR0B = (1<<RXEN0)|(1<<TXEN0)|(0<<RXCIE0)|(0<<UDRIE0);


        //sei();

}


// send a byte thru the USART

void uartTx(char data)

{

        // wait for port to get free

        while (!(UCSR0A & (1<<UDRE0))) { }

   UDR0 = data;
```

```
}



// get one byte from usart (will wait until a byte is in buffer)

uint8_t uartRxGetByte(void)

{

   while (!(UCSR0A & (1<<RXC0))) { } // wait for char

   return UDR0;

}



// check to see if byte is ready to be read with uartRx

uint8_t uartRxHasChar(void)

{
        return  (UCSR0A & (1<<RXC0)) ? 1 : 0;
}


////////////////////////////////////////////////////////////////////////////////////////////
// MAIN LOOP: service the serial port and stuff bytes into the framebuffer
////////////////////////////////////////////////////////////////////////////////////////////

void serviceSerial(void)
{
        uint8_t *ptr = frameBuffer;
        int state = 0;
        int counter = 0;
        while (1)
        {
                uint8_t c = uartRxGetByte();

                // very simple state machine to look for 6 byte start of frame
                // marker and copy bytes that follow into buffer
                if (state < 6)
                {

                        // must wait for 0xdeadbeef to start frame.
```

```
                        // note, I look for two more bytes after that, but
                        // they are reserved for future use.

                        if (state == 0 && c == 0xde) state++;
                        else if (state ==1 && c == 0xad) state++;
                        else if (state ==2 && c == 0xbe) state++;
                        else if (state ==3 && c == 0xef) state++;
                        else if (state ==4 && c == 0x01) state++;
                        else if (state ==5)  // dont care what 6th byte is
                        {
                                state++;
                                counter = 0;
                                ptr = frameBuffer;
                        }
                        else state = 0; // error: reset to look for start of frame
                }
                else
                {
                        // inside of a frame, so save each byte to buffer
                        *ptr++ = c;
                        counter++;
                        if (counter >= DISP_BUFFER_SIZE)
                        {
                                // buffer filled, so reset everything to wait for next frame
                                //counter = 0;
                                //ptr = frameBuffer;
                                state = 0;
                        }
                }
        }
}

void setup()                 // run once, when the sketch starts
{
        // Enable pullups for buttons
        PORTB |= (1<<0);
        PORTC |= (1<<3) | (1<<2) | (1<<1) | (1<<0);

        uartInit(CALC_UBBR(115200, F_CPU));

        displayInit();

        sei( );

        // clear display and set to test pattern
        // pattern should look just like the “gray test pattern”
        //  from EMS
        int i = 0;
        uint8_t v = 0;
```

```
        uint8_t *ptr = frameBuffer;

        for (i =0; i < DISP_BUFFER_SIZE; i++)
        {
                v = (v+2) % 16;
           // set to 0 for blank startup display
                // low order bits on the left, high order bits on the right
                        *ptr++ = v + ((v+1)<<4);
                //          *ptr++=0;
        }

        serviceSerial();  // never returns


}


void loop()                 // run over and over again
{



}
```