1. Year and Semester : 2013 SRING

2. Course Number : CS-492

3. Course Title : Kinect Programming

4. Work Number : LK-02

5. Work Name : Kinect Laboratory Report

6. Work Version : Version 1

7. Long Date : Sunday, 10, Frebruary, 2013

8. Author(s) Name(s) : Jake Waffle

Kinect Project 2

This project was probably as simple as it gets. It teaches the reader how to setup the Kinect to read the depth and rgb values in front of it and also how to draw what the Kinect sees onto a window.

So now I'll go through step-by-step what I did to complete the project:

1. The first thing I did was copy the code from our Kinect book (Borenstein p54).
2. Then I analyzed it after first running it.
3. The setup () function's purpose is to setup a window that can be drawn onto, instantiate the Kinect object given to use by our SimpleOpenNI library and finally tell that Kinect object to record depth and rgb images. (This function is executed once at the start of the program.)
4. The draw () function's purpose is to update the Kinect object so that it has the most current depth and rgb images and to draw those images onto our window. (This function is executed continuously for as long as the program is running.)

When going to alter the program, I couldn't figure out much to do. But I did change the dimension of the program's window and made it so that the depth and rgb images are drawn vertically instead of horizontally. To do this in the code, you can change the size() function's arguments to 640, 480*2 (this changes the dimensions of the screen.) Then you can alter the position at which the depth and rgb images are drawn. I just changed the second image() function to "image(Kinect.rgbImage(), 0,480)."

Kinect Project 3

This project was also pretty simple. It involved the same things from Project 2. But it also included a mousePressed function that will print the rgb values of the pixel the mouse clicks on. I assume that the mousePressed function is triggered upon a mouse click and that it implicitly passes the mouse's position as arguments.

So now I'll go through step-by-step what I did to complete the project:

1. First I copied the code from our Kinect book (Borenstein p62).
2. Then I analyzed it after running it.
3. All of the code from Project 2 still exists in this project.
4. The new addition is the mousePressed() function.
5. The mousePressed() function uses the get function with the mouse's position as an argument to grab the pixel color at the mouse's position.
6. From here, there is the choice of red(), green(), blue() and alpha() functions to grab the rgba values and the brightness() function to grab the depth value of the depth image.
7. And we just simply printed those values after retrieving them.

The brightness and alpha functions were what I added to the existing code; they seemed like they could be important in later projects. Later I'd like to merge the depth and rgb images together by altering the alpha values of the front image to make it transparent. But then I'd have to know how to loop through all of the pixels and alter the rgba value of a specific pixel.

Kinect Project 4

This project was about getting the depth length in a recognizable unit of measure. Grabbing the depth at a particular pixel location is a lot like how it was done in Project 3. But instead of getting the color of a pixel in the window with the get function, we grab the depth value of a pixel from the Kinect's depth map (the Kinect's depthMap function will return an array of all of the depth values.)

So now I'll go through step-by-step what I did to complete the project:

1. First I copied the code from our Kinect book (Borenstein p70).
2. Then I analyzed it after running it.
3. There are two major differences I see in the code: First is that the rgb image isn't being enabled in the setup() function or drawn in the draw() function. The other difference is that the mousePressed() function is grabbing the depth value from the depth map instead of basing it on an rgb value.
4. The last project I learned of a brightness() function that would return a depth value based on the rgb value of the depth image (being grabbed from the depth image that is visible to us in our window.)
5. But for this project, we're grabbing a more accurate depth value from the Kinect object's depth map with the Kinect.depthMap() function (instead of the depth image on the window.)
6. Since the depth map is held within an array with a single dimension and the pixel coordinates we have for the mouse are for two dimensions, we must convert the pixel coordinates to its position within a single dimension. This can be done by multiplying the

mouseY value by the width of the screen and then adding the mouseX value. And after that is done, we can say that we want to grab the depth value at the position that we just calculated within the depth map, like so “int clickedDepth = depthValues[mouseX+ (mouseY*640).”

7. Then after obtaining the depth at the mouse’s position, the code in the book converts the depth to inches (from millimeters) and prints the result.

The depth values in the depth map will all use millimeters as their unit of measure by default. So I just included the millimeter-based depth in the print statement. And we can totally convert millimeters to any other unit of measure if we wanted to (divide by 25.4 to get inches.) I can’t think of anything interesting to do with this project other than print the different units of measure for the depth at the selected pixel.

Kinect Project 5

This project essentially is all about searching for the pixel with the smallest (non-zero) depth value. Then after searching through all of the pixels on the screen, an ellipse is drawn onto the position of the pixel with the smallest depth (non-zero.)

So now I'll go through step-by-step what I did to complete the project:

1. First I copied the code from our Kinect book (Borenstein p78).
2. Then I analyzed it after running it.
3. The code doesn't seem too complex, the only really involved part is within the draw () function.
4. First of all, the variables needed for the draw() function I notice are declared in the global scope (I think that's because having them declared every time during the loop would be unnecessary.)
5. Second, at the beginning of each draw() call, the closestValue variable is overwritten with a depth that is unlikely to be larger than (8000 or 8 meters.) This is so that the closestValue from last draw() call isn't still the same (the closest point would have to be closer than that value in order to update anything.)
6. So after updating the closestValue, the Kinect object is updated (we've seen this in every project thus far.)
7. Then the code uses a for loop with another embed for loop to loop through all possible pixel coordinates in the window (each for loop will iterate through the possible pixel positions for a single axis (the x-axis and y-axis.))

8. For each iteration of the embed for loop, first the depth value for the current position is taken from the depth map array (using the converted single-dimension position on the screen.) Then the depth value we just received is checked against the closestValue variable that holds our closest depth (so far.) And if that depth value is smaller than the closestValue variable, we'll overwrite the closestValue with that depth and also update the closestX and closestY variables with the current x and y positions.
9. Then after we've looped through the entire set of pixel locations on the window and have obtained the position of the closest pixel, we'll draw the depth image to the screen and also draw an ellipse at the position represented by the closestX and closestY variables.

When going to make changes to the project, I wanted to make sure that the position at the ellipse was in fact the closest position. So I added the mousePressed() function from the previous project and that allowed me to click on the ellipse to print a depth close that is close to the ellipse. And since this is the first project that uses drawing primitives, I changed the color of the ellipse and also its width and height arguments. I also found out about the rect() drawing primitive by typing in the names of different shapes (it replaced the ellipse with a rectangle.) I couldn't figure out any other primitives.

Kinect Project 6

This project seemed similar to that of Project 5. But instead of just keeping track of the closest point, it requires keeping track of the closest pixel for the previous and current loop. Then with these two points we draw a line between the two, instead of drawing a shape at the current closest point. And the depth image in this project isn't drawn each loop, so that allows us to see all of the previous lines that were drawn. There is also a way to save the window's picture by clicking the mouse in this project.

So now I'll go through step-by-step what I did to complete the project:

1. First I copied some code from our Kinect book (Borenstein p87).
2. Then I analyzed it after running it.
3. This piece of code will draw a line between the previous and current closest point in relation to the Kinect.
4. Since the code draws the depth image each loop, the previous lines get overwritten by the updated depth image.
5. Essentially, the code is the same as from Project 5, except there is an extra point being accounted for (previousX and previousY are declared by the closestX and closestY variables.) And a line is drawn between the two points instead of just ellipse at the current closest point.
6. Then I copied some more code from our Kinect book (Borenstein p91).
7. And analyzed it after running it.
8. There are five big differences I saw from the last bit of code

9. First the x-axis is mirrored for the points that are found to be the closest (this is so that images we draw (with our movements) are placed on the screen in the same perspective as when we drew them.)
10. Second the depth image is no longer being drawn each loop. Instead the window's pixels are colored as black at the setup sequence and when the image is saved (which will give us a blank screen to play with after saving the previous one.) This results with is a black window that will show all of the lines that were drawn after the screen was last wiped.
11. Third is that the transition between the previous and current closest point. The code doesn't actually draw a line from the previous to the current closest point. It instead will interpolate between the previous and current point (interpolation allows smooth transitions and gets rid of the immediate jumps between the previous and current closest point) and use that interpolated point instead of the current closest point. So the line goes from the previous closest point and the interpolated point between the previous and current closest point.
12. Fourth is that there is now a way to save the drawn image in the window by clicking the mouse. This is done with the mousePressed() function that we've used in previous projects and the save() function is used to save the contents of the window with a specific filename. Then after saving the image, the window's pixels all get changed to the color black.

Now for the changes I've made. First, I changed the argument for the lerp() function (that does the interpolation between the last and current closest point) so that the drawn lines transition faster between the previous and current closest points (it was the third argument that I

changed from 0.3f to 0.75f.) I also changed the line's color and width (done so by changing the arguments of the stroke() and strokeWeight() functions.)

Kinect Project 7

This project has two parts to it: one introduces the simple idea of placing an image at the pixel with the smallest depth value (within the boundary of course,) the other allows the user to control multiple images on the screen (separately though.)

So now I'll go through step-by-step what I did to complete the project:

1. First I copied some code from our Kinect book (Borenstein pp96-98).
2. Then I analyzed it after running it.
3. This project essentially is project 6, as it uses interpolation to smoothly track the closest object to the Kinect (within some boundaries) and the image is placed at a point between the last and current closest point (due to interpolation.)
4. A big difference between project 6 and 7 is that project 7 introduces images into our projects. The mousePressed() function is also used to toggle the imageMoving boolean (that determines if the image's position is affected by the closest object or not.)
5. The fact that this project tells us how to load, store and draw images onto the screen is valuable. I'm sure that many of us will use images when we do our actual projects later on in the class.
6. So then I copied the rest of the code for this project from the Kinect book (Borenstein pp100-102).
7. Then I analyzed it after running it.
8. This bit of code seemed to be doing the same as the bit before: except it has multiple images on the screen, a way to switch control between the images and the controlled image changes it scale with respect to the closet object's depth value.

9. To do this the code essentially has three different groups of global variables (one for each image.) The image variables are all loaded with images and drawn to the screen using these groups of variables.
10. The control of these images is done by altering the group of variables associated with the current image (which is determined by an integer called “currentImage.”) And to switch the control from one image to another, the mousePressed() function is used to cycle through the three images (by altering the currentImage variable.) This all wouldn't work if the draw() function wasn't setup to alter the image variables based off of the currentImage variable. (That's just done with a switch-case statement with currentImage as the argument and in each of the three cases the appropriate image variables get updated with the interpolated position and the closestValue for the image's scale.)

After analyzing the first part to this project, I made it so that the mousePressed() function would change the image that was being drawn at the closest object's position. And to do that I first had to add in two more variables in the global scope: to store the second image in and to store the image that would be drawn to the screen.

```
>PImage image2;
>PImage curImg;
```

Then I loaded the second image into the first of the two new PImage variables and linked the first image to the second of the two new PImage variables. (All of this being added to the setup()

```
>image2 = loadImage("image2.jpg");
>curImg = image1;
```

After that I changed the variable that was used for drawing the image to “curImg” (in the draw() function.)

```
>image(curImg, image1X, image1Y);
```

Finally I took out every reference to the imageMoving variable and rewrote the mousePressed() function. It essentially checks to see which image is being used currently and changes the image to the other one (since there are two images now.)

```
>if(curImg == image1)
>{
>    curImg = image2;
>}
>else
>{
>    curImg = image1;
>}
```

When altering the second part of Project 7, I gave the images initial values for their positions and scales (I'm surprised the book didn't do that, images don't pop-up until they have been chosen to be controlled.) I also changed the parameters for the map() function inside of the draw() function and switch-case statement (it is used to compute the scale by converting the closestValue variable to a number between zero and four, I think.)

```
>image1Scale = map(closestValue, 1525, 610, 0, 4);
```

That made it so that image1 was larger the closer it got to the Kinect (opposite of what it was beforehand).

References

-Borenstein, G (2012). Making Things See. Sebastopol,California: O'Reilly Media.