ECE578: Intelligent Robotics I

Homework# 1 on Thursday: 11/01/12

Rami Alshafi

Intelligent Robotics I

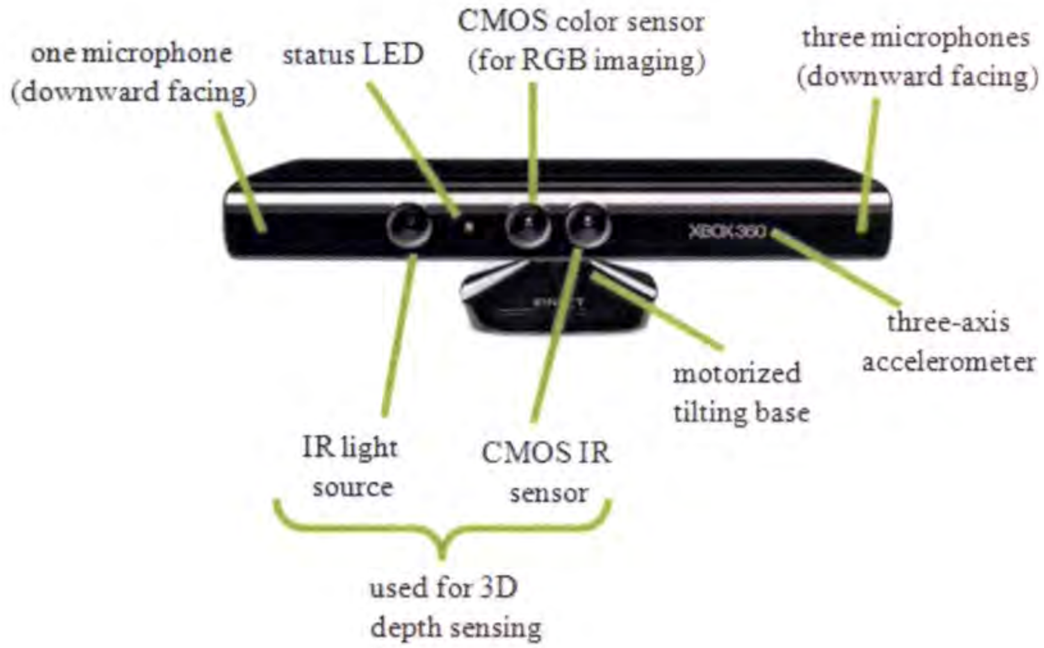
The history behind the Kinect's technology

The product known as "Microsoft Kinect" is a result of years in academic research and development in computer vision. A company called PrimeSense developed the hardware of the Kinect. Their main contribution was the infrared projection technique. PrimeSense has worked very closely with Microsoft and the Kinect was called "Project Natal" back then. Eventually, PrimeSense licensed the hardware design to Microsoft, but they still own the basic technology. Currently, PrimeSense is working with ASUS to make the Wavi Xtion, which is a similar product to the Kinect but it is designed for developers and not for end-consumers. It is smaller and lighter and has less hardware than the Kinect but it is more expensive than the Kinect and not as available as the Kinect. The Kinect; however, has much more value per dollar than Xtion. At any rate, various companies are integrating this technology into their own products such as smart televisions, personal computers and some other electronics' gadgets.

Microsoft released the Kinect to market on November 4, 2010 and became a major commercial success. Ten million units were sold in the first month, which makes it the fastest selling computer peripheral in history! Programmers and developers did not waste any time to hack the hardware and software of the Kinect. Adafruit, open source Hardware Company run by Limor Fried and Phillip Torrone, announced a bounty of $2,000 for whoever comes up with an open source driver for the Kinect. After Microsoft spokesperson reacted negatively to this action, Adafruit increased the bounty to $3,000! Only six days after releasing the Kinect, Hector Martin released the first open source Kinect driver and he won the $3000. Next, he joined Open Kinect community created by Josh Blake. After that, many developers have created multiple drivers for the Kinect including "OpenNI" from PrimeSense and Microsoft Software Development Kit. Those libraries made the Kinect accessible in a variety of environments including Processing.

Kinect's hardware

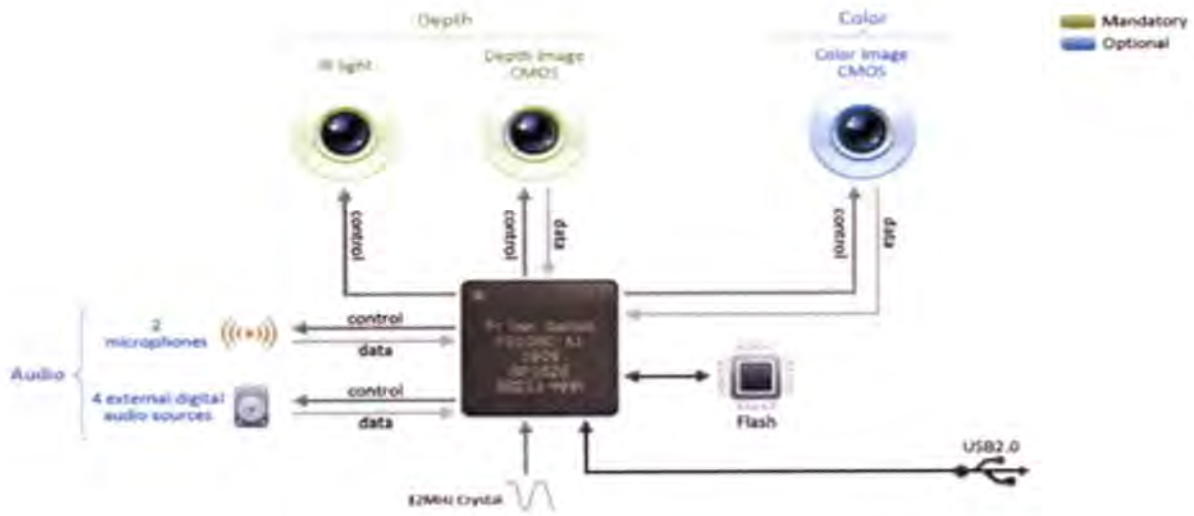
The hardware of the Kinect consist of IR projector, RGB Camera, CMOS Sensor, an array of four microphones, a 3-axis accelerometer, a gear motor and a fan.


one microphone (downward facing)
status LED
CMOS color sensor (for RGB imaging)
three microphones (downward facing)
three-axis accelerometer
motorized tilting base
IR light source
CMOS IR sensor
used for 3D depth sensing

The IR projector and CMOS Sensor are responsible for generating the depth image, which is the main selling point of the Kinect. The IR projector sparkles a grid of infrared dots over everything in front of it. Those IR light is not in the visible spectrum, which is the reason, why human cannot see it but it could be captured with night IR goggles. The IR light gets reflected by the objects around and comes back to the CMOS sensor and therefore the Kinect can construct a 3D depth image of what is in front of it.

The RGB digital camera does not have anything special. It is just like any other low quality web-cam available in the market. However, it is built into the Kinect at a fixed distance, which allows both depth image and color image to interact with each other and make up a good feature for certain applications and projects.
Moreover, the Kinect has an array of four microphones, which are distributed across the entire device. This unique distribution of microphone allows the Kinect not only to capture sound and recognize it but also allows it to locate where the sound is coming from. This is the answer to how the Kinect can receive multiple voice commands at once and still knows whose voice and said what! Unfortunately, drives

that can access the audio features of the Kinect are not as much as the drives for the depth image. So far, only Microsoft SDK has this capability.
The last two features of the Kinect are the motor and 3-axis accelerometer, which simply controls the neck of the Kinect within 30 degrees limit. Only a few libraries can control this motor and the reading of the accelerometer.



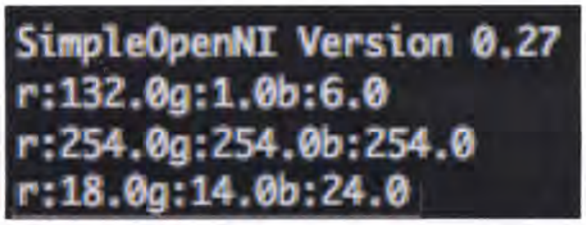
Analysis of the Kinect's images

There are two types of images that can be generated with the Kinect, which are conventional color image and a depth image. The depth image has many advantages over the color image and also has some limitations. The depth image is easier to process by a computer and it is more reliable and therefore gives us plenty of room to add heavy processing power in comparison with the conventional color picture. Another advantage for the depth image is that it is NOT sensitive to light at all, so that makes it reliable regardless of the light conditions, which is a major drawback of the color image. More importantly is its ability to construct an accurate three-dimensional measurement of the scanned area.

Color Image

I have written a simple program using the Processing Development Environment and the Simple OpenNI library made by PrimeSense in order to access the color and depth cameras and I noticed the following. I know that colors are generated from three main colors, which are Red, Green and Blue, so in my simple program I analyzed the colors back into its three components of red, green and blue. When I analyze what I see with my eyes as red, I notice the red component is higher than the green component and the blue component. It makes sense but the red component is not high enough even though the object was completely red. I expected the Red component would be full but instead it was around half.
Red=132/255, Green=1/225 and Blue=6/255

Next, I examined the white color and it had large portions of every color component. Red=254/255, Green=254/255 and Blue=254/255. Next, I examined the black color and it had only a little bit of every component. Red=18/255, Green=14/255 and Blue=24/255.



```
SimpleOpenNI Version 0.27
r:132.0g:1.0b:6.0
r:254.0g:254.0b:254.0
r:18.0g:14.0b:24.0
```

This observation brings me to the conclusion that the relative levels of the color components and their ration against each other determine the color in the image, and the brightness are determined but the total sum of all the color components. The higher the number, the brighter the picture is. I did a few more exploration and I noticed that the Green color under dark lighting has more Red component than Green, which I found very surprising and not expected at all. I research this phenomenon quickly in Google and found out it is called "Red-Shifting" which is another reason why depth image is better than color image.
Here is the code

```
import SimpleOpenNI.*; // improt the OpenNi library.
SimpleOpenNIkinect; /* declear object names "kinect"
to hold Kinect's data. This object is not instantiated
yet! to be done in the setup function.*/

void setup()
{
  size(640*2, 480); /* declaring the size of the app
  (images from the kinect are 640 pixels wide by 480 tall)
  I need twice the width to dispaly both pictures.*/

kinect = new SimpleOpenNI(this); // instantiating the SimpleOpenNI instance.

kinect.enableDepth(); // enabling the depth image from the depth camera
kinect.enableRGB(); // enabling the colored image from the RGB camera

  /* here both "enableDepth" and "enable RGB" functions will call the SimpleOpenNI
  library and it will gather the data from the Kinect and
  then it will be stored in the object we delared earlier and named it "kinect"*/

}  //end of the setup function which runs one at execution.
void draw() // this is the draw loop which will be run constantly as long as this app
is running
{
kinect.update();  // calling the update function in order to update our object
"kinect" so the library gets fresh data
PImagedepthImage = kinect.depthImage();

PImagergbImage = kinect.rgbImage();

  /* these 2 funtions are not important for this app but there are here so I
 can see what is going on in the background for any further analysis such as
 checking out each pixle and alter them or passing these info to other image
 processing libraries that do not know anything about kinect but more functional
  in terms of image processing*/

 image(kinect.depthImage(),0,0); /* this line gets the fresh stored in "kinect"
 and passes it to the image function to de displaced. The (0,0) is the location of the
picture*/
 image(kinect.rgbImage(),640,0); /* this line gets the fresh stored in "kinect"
 and passes it to the image function to de displaced. The (640,0) is the location of
the picture*/
}
void mousePressed(){// this function willbe called whenever the mpouse is pressed
 color c = get(mouseX, mouseY);
println("r:"+red(c)+"   g:"+green(c)+"  b:"+blue(c));
}
```

Depth Image

Another reason why depth image is better than color image when it comes to processing and extracting information from the image is that it is more difficult with a color image to determine the shape of an object that has the same color. Also, it is difficult for a color image processing to determine when an object ends and when another object starts. In order to extract this kind of information from a color image, I would need to process every single pixel in the entire image and take a lot of pictures of the same thing and re-process the whole pixels of these many pictures! In the other hand, all the color components in a depth image have the same exact value. They either go up together or down together, which eliminate the color complexity and leaves only the brightness. In fact, the brightness has a linear proportional relationship with the distance, which is the secret that makes Kinect capable of capturing an accurate 3D scanning of the environment regardless of the color or light condition.

Depth image limitations

Even though the depth image is so great, it still has many limitations, which are important to know in order to design our applications around these limitations and possibly turn these disadvantages into advantages for our application.

Range

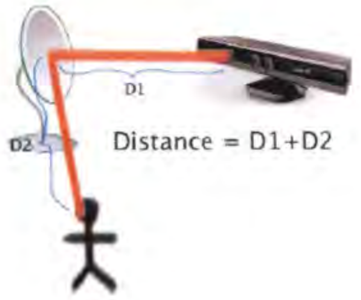
The first obvious limitation is the range of the Kinect. The minimum range is 20 inches or 508mm and the maximum range is 25 feet or 7.62m. Anything closer to the Kinect than 20 inches will be treated like if it were very far away, which is not the case in the physical world. The same is true for things that are far away from the Kinect than 25 feet. Since the depth image measures the brightness, which is linearly related to the distance, the farther points will be darker and has lower value and the closer points will be brighter and has higher value. As a result, things closer than the minimum range and things farther than the maximum range all will have a value of Zero and marked as the darkest black. More information on this limitation will be explained in the Measure Tape Application.

Noise at edges

We know that the depth image is constructed based on the IR dots reflected back to the CMOS IR sensor of the Kinect. The problem with edges is that they reflect some of the IR dots away from the Kinect and the dots will never find its way back to the Kinect CMOS IR sensor. As a result, there will be missing information, so the Kinect treats these missing pieces as if they were very far away and gives it a value of 0 and marks it as the darkest black. In order to recover this missing information, an appropriate angle of reflection needs to be found in multiple scans. This angle can be found in two ways. Rotating the scanned area or thing is the first method and moving the Kinect around the scanned area or thing is the second method. Since you will have multiple scans of the same thing, you will need a software that understands what you are doing and merge these multiple scans into one picture.

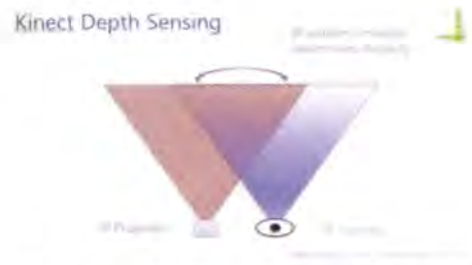
Reflection & distortion

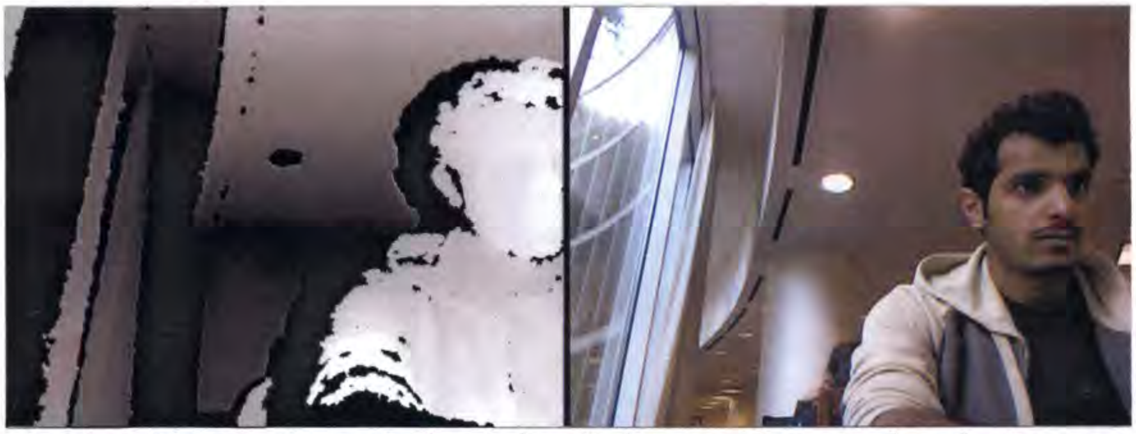
Reflective materials like mirrors could reflect the IR dots to other objects and back to IR sensor. As a result, the distance the Kinect sees is not the distance between the Kinect and the mirror alone but it is the distance between the Kinect, Mirror and the reflected object in the mirror.
Some artist like Kyle McDonald has takes advantage of this disadvantage and set up a mirror structure so the Kinect can see 360 degrees view of an object in a single scan!



Depth shadows and Misalignment

IR dots array cannot see objects behind the scanned objects because it does not penetrate objects like X-ray because it has a long wavelength and therefore low frequency, which is another reason IR light is not harmful in anyway. In fact, we are subjected to it from the sun. Anyways, the CMOS IR sensor is in different location than the IR projector so not all the IR dots will be reflected back to the CMOS IR sensor. The depth image will be the area overlapped between what the CMOS IR sensor sees and the area to which the IR projector can aim. As a result, there will be an area that can be seen by the CMOS IR sensor but unfortunately the IR dots never got there so the Kinect has no information what is there and then the Kinect will do what it does best, mark it as the darkest black and gives it a value of Zero and pretends like it is very far away.

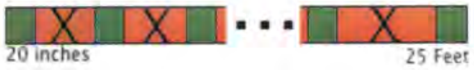


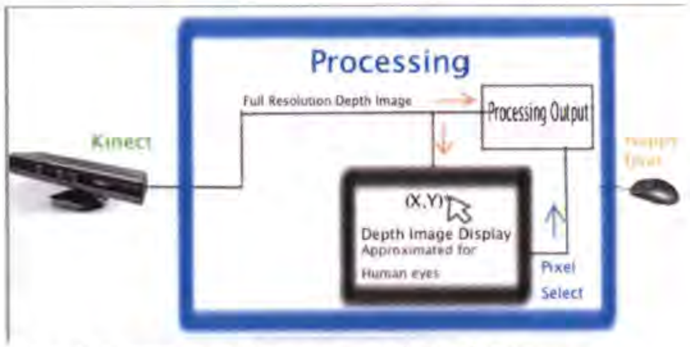


Another thing worth mentioning is the misalignment between the color image and the depth image. Both cameras are built into the Kinect and separated by a known distance. This physical separation of the cameras causes misalignment between the color and depth images. This should make sense as the same distance separates our eyes and if you would close one eye and then close the other eye and compare the two visions, you will notice the area covered by each eye and there is an overlap but our miracle brains does the processing necessary to clear the picture for us.

Measure tape application

One way for me to test the accuracy of the Kinect's measurements is to write a quick application where I can measure things, and I wanted an actual measurement and not brightness. Initially, this task seemed simple because I knew Kinect's range and the brightness values. In theory, Zero brightness value would equal to 25 feet or greater and 255 brightness value would equal to 20 inches or less. I wrote a quick application that converts brightness values to millimeters and inches and then put an object at a premeasured distance away from the Kinect and then measured this distance by the Kinect,and it was wrong! Next I moved the object to 20 inches and measured the distance by the Kinect and it was correct! Next, I kept incrementing the distance of the object away from the Kinect inch by inch and measure the distance by the Kinect and then I realized there are points the Kinect is accurate and points where the Kinect is wrong. What interesting was that those correct and wrong points had a fixed structure, so I knew it was a resolution issue. After research, I found out the Kinect has higher resolution than 255. Kinect has a resolution equal to $2^{11} = 1048$. The resolution was limited to 255 by the graphics development environment that I am using, which is Processing. I did not want to switch to another graphic development because it would require a new learning curve for me. I researched other graphic development environments to see if it was worth the extra effort in learning a new tool and it turns out that most of the graphic development environments had the same limitation and I did not want to spend a lot of money on a powerful image processing software. Most of the graphic development environments assume the human eyes cannot notice any difference in resolution and 255 was plenty for Processing. The cheap solution was to bypass the display and access Kinect's data before it is displayed in the software. Fortunately, Processing is an open source and I found a good discussion explaining exactly how to access this information before it was displayed to utilize the full resolution provided by the Kinect. I was relieved that there was a function in the SimpleOpenNI library that I can call and it does the heavy work for me. However, this function returns the depth values in a single one-dimensional long array full of integers, which is confusing because a picture would be 2 dimensional. So I needed to convert this single dimensional array to double dimensional array so it can simple make sense in my mind because I will be clicking on the displayed 2D picture with a mouse and wanted to know where that particular pixel is stored in the array. The function returns contains 640 X 480 integers arranged in a single linear array where the first integer corresponds to the top left pixel in the depth image. Each following value corresponds to the next pixel across each row until the last value finally corresponds to the last pixel in the bottom right. Knowing this relationship, it could be represented mathematically as follow





$$\boxed{\text{Mouse Position} = (\text{Y coordinate} \times 640) + \text{X coordinate}}$$

1st pixel/ 1st row		Last pixel/ 1st row	1st pixel/ 2nd row		Last pixel/ 2nd row	...	1st pixel/ Last row		Last pixel/ Last row
0	...	639	640	...	1279	...	306560	...	307199

(0X640)+0	(0X640)+1	(0X640)+2	...	(0X640)+639
(1X640)+0	(1X640)+1	(1X640)+2	...	(1X640)+639
(2X640)+0	(2X640)+1	(2X640)+2	...	(2X640)+639
.	.	.	.	.
.	.	.	.	.
.	.	.	.	.
(479X640)+0	(479X640)+1	(470X640)+2	...	(479X640)+639

Here is the code

```
import SimpleOpenNI.*;
SimpleOpenNIkinect;
void setup()
{
 size(640,480);
kinect = new SimpleOpenNI(this);
kinect.enableDepth();
}
void draw()
{
kinect.update();
PImagedepthImage = kinect.depthImage();
 image(depthImage,0,0);
}
void mousePressed()
{
 int[] depthValues = kinect.depthMap();
 int clickPosition = mouseX + (mouseY * 640);
 int millimeters = depthValues[clickPosition];
 float inches = millimeters / 25.4;
println("mm:" + millimeters + "  in:" + inches);
}
```

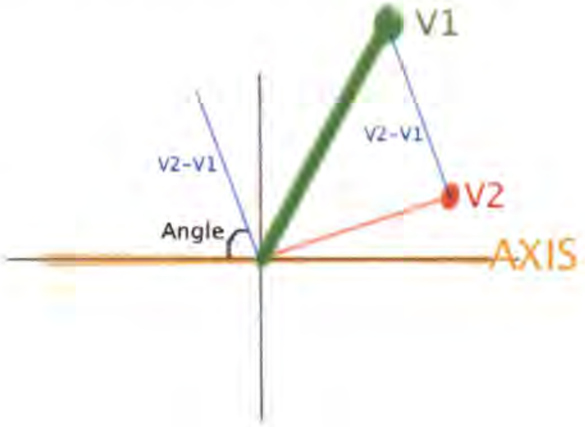
Homework

I have been working on my "Blind Vision Project" for some time and I have been using ultrasonic sensors as my inputs. Meanwhile, I have been reading about the Kinect and I knew I wanted to use the Kinect instead of the ultrasonic sensors.However, I have never worked with the Kinect before but I was motivated to learn all about it, so the goal of my homework is to explore the Kinect and understand its capabilities and investigate the possible applications in the blind community.

Avoiding obstacles is the most important aspect in guiding the blind but I have already achieved that with my ultrasonic sensors, and I wanted to do something more interesting than simply avoiding obstacles. From my previous reading, I knew the Kinect can recognize human bodies and skeleton, so I wanted to see if I can interpret some kind of buddy language in a way the Kinect will understand, and then do something about it. In order to make the project reasonable, I just wanted to understand a hand wave like "HI" or "Good Bye".Therefore, I wanted to build a simple robot arm that reproduces the position of my real arm as detected by the Kinect. I will send the joints data from Processing to the robot over a serial connection. The brain of my robot will be an Arduino microcontroller. Processing does the interactive graphical applications, and the Arduino listens to the commands from Processing and controls the robot's motors to execute them.

First milestone

So far, I know the Kinect can capture body skeleton but I did not know how. Thus, I made this my first milestone. Fortunately, the library I am using, which is OpenNI, has functions for accessing the user's joint data such as shoulder, elbow or hip. This is really great because without it this milestone would have been impossible for me to accomplish in a timely manner. So, my real task here is to know how to use these functions. There are two man angles that I am looking for, which are the shoulder angle and the elbow angle. The shoulder angle is between the upper arm and the torso and the elbow angle is between the upper arm and the lower arm. Also, I need to pay attention to the axis that I am considering for these angles, so the axis for the shoulder is the torso and the axis for the elbow is the upper arm. When the functions locate the joints of the user, they return three-dimensional vector representing the location of that detected joint. To make my assignment easier and more realistic, I wanted to work on a single plane only. So my hand will be waving to that Kinect for "hi" or "bye" in a motion in the X-Y plane alone. Motions in the Z-axis will be ignored. The best way to do this is convert the returned 3D vectors into 2D vectors for all of the joints that I want to detect. Even though I need only two angles, I still need to detect four joints and they are the ends of each limb. Those four joints are right hand, right elbow, right shoulder and right hip. Once I detected those joints and converted their vector to 2D, I then subtract the right hand 2D vector from the right elbow 2D vector, which generates a new single vector representing the lower arm. Then, I do the same on right elbow and right shoulder to generate a new vector representing the upper arm, which serves as the orientation axis for the elbow angle. Next, I subtract the right shoulder from the right hip to generate a new vector representing the torso, which serves as the orientation axis for the shoulder angle.



Please take a look at the code in the next page where I included a bunch of comments to help explain to you what is going on.

```
import SimpleOpenNI.*; // improt the OpenNi library.
SimpleOpenNIkinect;
/* declear object names "kinect"to hold Kinect's data.
This object is not instantiated yet! to be done in the setup function.*/
void setup() {
 size(640, 480);
// declaring the size of the app (images from the kinect are 640 pixels wide by 480
tall)
//I need the depth image alone.
kinect = new SimpleOpenNI(this);// instantiating the SimpleOpenNI instance.
kinect.enableDepth();// enabling the depth image from the depth camera
kinect.enableUser(SimpleOpenNI.SKEL_PROFILE_ALL);
// enable user tracking in order to access skeleton data
kinect.setMirror(true);
//mirror the depth image data so it makes sense when I look at the image. my left
hand will be on my left
/* here both "enableDepth" and "enableUser(SimpleOpenNI.SKEL_PROFILE_ALL)"
functions will call the SimpleOpenNI library and it will gather the data from the
Kinect and
then it will be stored in the object we delared earlier and named it "kinect"*/

}//end of the setup function which runs once at execution.
void draw() {// this is the draw loop which will be run constantly as long as this app
is running
// this has got to be here regardless if you were to draw something or not. In this
case we want to draw
kinect.update();
// calling the update function in order to update our object "kinect" so the library
gets fresh data
PImage depth = kinect.depthImage();
/* this funtion are not important for this app but there are here so I
can see what is going on in ther background for any further analysis such as
checking out each pixle and alter them or passing these info to other image
processing libraries that do not know anything about kinect but more functional
in terms of image processing */
 image(depth, 0, 0);
// image function display the depth picture. The (0,0) is the location of the picture
IntVectoruserList = new IntVector();
/*create an integer vector "userList" to store the list of users. This is a special type
of variable
provided by OpenNI specifically to store lists of integers representing user IDs*/
kinect.getUsers(userList);
/*write the list of detected users into the vector"userList"
```

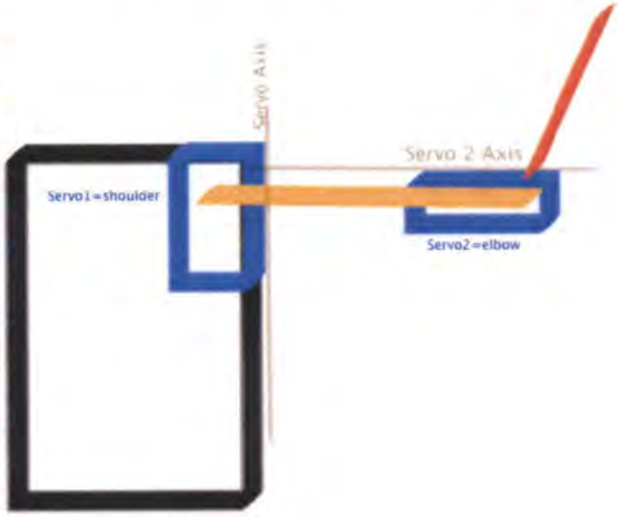
```
the "getUsers" function will put all of the users' IDs that OpenNI knows about
into the Vector "userList" we created in previous line. There is NO return value form
this function*/
 if (userList.size() > 0 ) { // check the "userList" vector to see if there is any user
detected
   int userId = userList.get(0); // if true, get the first user ID from "userList to
userId"
   if (kinect.isTrackingSkeleton(userId)) {
     /*check if calibration successful. If true, access positions of 4 joints below
     This function returns true only if the calibration process has been completed
     and joint data is available for the user. In the 1st many executions iterations of
"void draw()",
    this if statment will retun false since the calipration process is not finished yet */
PVectorrightHand = new PVector();
kinect.getJointPositionSkeleton(userId, SimpleOpenNI.SKEL_RIGHT_HAND,
rightHand);
PVectorrightElbow = new PVector();// position of right elbow
kinect.getJointPositionSkeleton(userId, SimpleOpenNI.SKEL_RIGHT_ELBOW,
rightElbow);
PVectorrightShoulder = new PVector(); // position of right shoulder
kinect.getJointPositionSkeleton(userId, SimpleOpenNI.SKEL_RIGHT_SHOULDER,
rightShoulder);
PVectorrightHip = new PVector(); // position of right hip
kinect.getJointPositionSkeleton(userId, SimpleOpenNI.SKEL_RIGHT_HIP, rightHip);
    /* The above 8 lines of code has 2 main lines.
    1- PVector Joint Name = new PVector();
    2- kinect.getJointPositionSkeleton(userId, SimpleOpenNI.SKEL_RIGHT_HAND,
Joint Name);
    The use of the 1st line is to create a vector and use it to store the position of the
joint
    the 2nd line passes the variable to "getJointPositionSkeleton" along with user ID
and
    a constant representing the joint we want to access*/
    /*
=================================================================
========================================
    the goal of the code below is to completely ignore the third dimension by
creating 2D vectors and not 3D
=================================================================
========================================*/
PVector rightHand2D = new PVector(rightHand.x, rightHand.y); // 2D vector for
right hand
PVector rightElbow2D = new PVector(rightElbow.x, rightElbow.y); // 2D vector for
right elbow
PVector rightShoulder2D = new PVector(rightShoulder.x, rightShoulder.y); // 2D
vector for right shoulder
PVector rightHip2D = new PVector(rightHip.x, rightHip.y); // 2D vector for right hip
PVectortorsoOrientation = PVector.sub(rightShoulder2D, rightHip2D); // identify
torse as orientation axis. To be used later for sholder angle
PVectorupperArmOrientation = PVector.sub(rightElbow2D, rightShoulder2D);
//identify upper arm as orientation axis. To be used later for elbow angle
    float shoulderAngle = angleOf(rightElbow2D, rightShoulder2D,
torsoOrientation); // Call angleOf function to calculate shoulder angle from torse
axis
    float elbowAngle = angleOf(rightHand2D, rightElbow2D, upperArmOrientation);
// Call angleOf function to calculate elbow angle from upper arm axis
    // code below to round results off and display them on screen
    fill(255, 0, 0);
    scale(3);
    text("shoulder: " + int(shoulderAngle) + "\n" + "elbow: " + int(elbowAngle), 20,
20);
   }
  }
}

float angleOf(PVector one, PVector two, PVector axis) { // take 3 arguments passed:
2 vectors representing the ends of a limb and oreintation axis
PVector limb = PVector.sub(two, one); // subtract 2 vectors representing the ends
of limb
  return degrees(PVector.angleBetween(limb, axis)); // calclate angle between limb
and oreintation axis. Next, convert from radians to Degrees. Finally, return value
}
// Below are the skeleton-tracking callbacks "onNewUser","onStartPose" and
"onEndCalibration"
void onNewUser(int userId) {
  /*this callback function takes 1 integer argument repesenting the detected user ID.
In case of multiple users, user IDs are important*/
println("start pose detection"); // prints out a message in the output so we know
where we are in the calibration process.
  // having "println" function helps us to get feedback, which offers help with
debugging when needed
kinect.startPoseDetection("Psi", userId);// this line checkes if the user is in the right
pose. In OpenNI, this pose known as "Psi"
  //when the new user is found in the "Psi" pose, the "onStartPose" callback fnction
will be called
  // at this point, user data should be available in the "void draw" loop for the first
time
}
void onEndCalibration(int userId, boolean successful) { // This function takes 2
arguments
/*the 1st argument is the user ID to keep track of whom we are calibrating
the 2nd argument is "successful" which is a boolean variable which checks if
```

```
user calibration is successful or not*/
 if(successful) {// if the calibration was successful, perform the following
println(" User calibrated !!!"); // prints out a message in the output so we know
where we are in the calibration process.
   // having "println" function helps us to get feedback, which offers help with
debugging when needed
kinect.startTrackingSkeleton(userId);// this function tracks the skeleton data and
finally skeleton data will be available in "void draw"
 }// at this point, this functon will not be called again unless a new user appear in
the screen or the current user need to be re-calibrated for some reason.
 else {// if the calibration falied, perform the following
println(" Failed to calibrate User !!!"); // prints out a message in the output so we
know where we are in the calibration process.
kinect.startPoseDetection("Psi", userId); // re-start the calibration process again for
this user for this pose
 }/* note that calibration process could be a loop and we do not get out of it unless
we met one of the following
 1- calibation is successful (the user could move around and make sure to have the
right pose)
 2- this entire sketch is terminated */
}
void onStartPose(String pose, int userId) {
 /*OpenNI triggers "onStartPose" as soon as it thinks the user is in the right pose for
calibration
 This callback funtion takes two arrguments. 1st is the nameof the pose. In theory,
we could have different poses than
 the "Psi" pose. This could be helpful if the user could not stand for example or could
not left thier arm for medical
 condition. 2nd argument is the use ID*/
println("Started pose for user");// prints out a message in the output so we know
where we are in the calibration process.
 // having "println" function helps us to get feedback, which offers help with
debugging when needed
kinect.stopPoseDetection(userId);
 /* Here we tell OpenNI to stop pose detection for the current user. Otherwise,
OpenNI will continue checking to see if
 the user has performed the calibration pose again and again and everytime it finds
this calibration pose it will call
 "onStartPose" function again and again and this will be very confusing since we are
already tacking the user successfully*/
kinect.requestCalibrationSkeleton(userId, true);
 /*Now It is a good opporonity to ask OpenNI to tack all of the joints of the user
because the user is in the right pose
 even though we may not need to tack all of the joits in most of the application, this
step is still important because
```

people have different bodyies such as longer arms, wider waste or shorter torso and this could fail calibration.
to avoid calibration faliur, OpenNI should NOW measure all of the limbs, arms, legs and width and adjust all of its assumptions
about the shape of the person to fit the current user's actual body type since user in right pose.
Next, OpenN calls the final callback function "onEndCalibration"*/
}

Second milestone

Second milestone is to create a robotic arm that can copy my arm. The arm will consist of two servos. One of them represent the shoulder and it can move 180 degrees and the other one represent the elbow, which also can move 180 degrees. Those servos are RC servos and not continuous rotation servos. The reason I prefer the RC servos than continuous servo is because you can control the angle of the RC servo where you control the speed in the continuous servo. This is important since I will be sending angles to those servos and not speed. I found the cheapest servos form HobbyKing for $1.99 plus shipping.Those servos are not very powerful so I need to keep the weight on the shoulder's servo as light as possible. I used an ice cream popsicle to represent the upper arm.
The brain of this simple robotic arm would be the Arduino micro-controller. Once the Arduino receives the joints angles, it should drive the servos. One more thing to mention about the servos that they require 5V, which can be provided by Arduino through the USB cable, so there will be no need for power supply or batteries.



Third milestone

Third milestone is establishing communication between Processing and Arduino

Programing the Arduino

The Arduino will establish a serial connection to the laptop where I am running Processing. Serial is a simple protocol that allows Processing to communicate with

Arduino, and it is bi-directional so Arduino can communicate with Processing as well. The Arduino will be listening for the two angles that will be sent from Processing but since the interface is serial, the angles will be sent one by one. This is a disadvantage for the serial protocol but since it is simple, cheap and easy to use, I need to program Arduino to go around this limitation. I will be sending the elbow angle first and then the shoulder angle and then the elbow again and then the shoulder again and so forth. So the Arduino should be able to keep track of which value is coming in when so that it can send the right values to the right servos.
In order to accomplish this task, I need to set up an array that holds both servo values. Then read in the next value from the serial connection, switch off which spot in the array we store it in. Finally, update the position of both servos based on the values in this array, which will hold the most recent angles for each joint.
Once I am done writing the software, I hook up the Arduino board to the USB and download the sketch in it. Please take a look at the code where I made plenty of comments to make things more clear

```
#include <Servo.h> // include the servo library
// the servo library makes it easy to control servos
Servo shoulder; // declare servo variable for shoulder
Servo elbow;  // declair servo variable for elbow
// setting up the servo array for positions
// its there to keep track of alternating servo positions
int nextServo = 0; // alternates servos 0 or 1
int servoAngles[] = {0,0}; // array to store angles for servos.
// initilizing the array to 0 will cause both servos to start at 90 degrees
// sothe arm starts at a right angle to prevent a violent jerk
void setup() {
shoulder.attach(9);// shoulder servo connected to pin 9
elbow.attach(10);// elbow servo connected to pin 10
Serial.begin(9600);// set up the baud rate to 9600 for serial communication
 // the baud rate could be any other value as long as it is kept constant
 //between the devices that are communicating with each other.
 //9600 is a standard value so I am using it.
 // it is fast enough and it wont cause any delay
}
void loop() {
 if(Serial.available()){ // Listen to any serial communication
 // If true, read serial communication and store in servoAngle
   int servoAngle = Serial.read();
  /* Next, store the value into the array. the variable "nextServo"
  is the index of this array to tell us exactly where we should
  store the read value in the array */
servoAngles[nextServo] = servoAngle;
nextServo++;// increment the nextServovaliable
  if(nextServo> 1) {
    /* we only have 2 servos so we need nextServo to alternate between
    0 and 1 only. 0 for shoulder and 1 for elbow. So, next servo need
    to be reset if it is greater than 1*/
nextServo = 0;
   }
  // Next, we need to send this information to servos.
  // The Write function will take care of that
shoulder.write(servoAngles[0]); // the 0 index for the shoulder
elbow.write(servoAngles[1]); // the 1 index for the elbow
  }
}
```

Testing the Arduino serial communication

There is a couple of option to test the serial communication of Arduino. There is the serial monitor, which is built in the Arduino environment. It is a simple interface and allows us to type values in it. Just make sure to set the baud rate as the one in the downloaded sketch, which is 9600 in this case. The Arduino is expecting the receive angles from 0 to 180 degrees. However, this interface takes in ASCII values. It is a system for encoding individual character as number for use in computers. It assigns each key on the Keyboard its own unique number from 0 to 128, including the number keys and control keys such as Enter and Space. Unfortunately, the numbers keys do NOT correspond to the numbers themselves. So, when I enter the number 0in the serial monitor, it will be converted to its ASCII equivalent, which is the number 48 and send it to Arduino. Arduino thinks its 48 degrees and moves the servo 48 degrees, which is wrong. As a result, this option of testing Arduino serial communication for this particular application is a hassle. The other option is to use Processing to test the serial communication. I will ultimately, use Processing to send information to Arduino so it makes sense to use Processing to test Arduino's serial communication. To do so, I need to write a simple software in Processing to establish the serial communication between the two. First, I need to initialize a serial connection, and then send numbers to Arduino based on keys that I type. I want to test only three points, which are 0, 90 and 180 degrees. To do that, I need to create three variables each represent an angle. These three values will demonstrate the complete range of servos and calibrate them in case I need to re-construct the arm. The code is below, please take a look.

```
import processing.serial.*; // include the serial library
Serial port; // create a serial object and call it "port"
void setup() {
  // I need to initialize the serial object but I need to
  // figure out which serial device I want to connect to
println(Serial.list());// call the Serial.list function to list all devices
 String portName = Serial.list()[6];// in my cmputer, Arduino is connected to port 6
  /* this could be different from computer to another. Run the sketch once and
  look at the output of Processing and find out where your Arduino is connected*/
  port = new Serial(this, portName, 9600); //initialize Serial object "port"
  /* call the Serial constructor and pass the portName we just figured out and the
 buad rate*/
}
```

```
void draw(){} // we must have the darwfuntion in Processing eventhough we are
NOT drawing anything
void keyPressed(){ // this function is executed when a key is pressed
 if (key == 'b') { // if the b key is presssed
port.write(0); // send the value 0
 }
 if (key == 'n'){ // if the n key is pressed
port.write(90);// send the value 90
 }
 if(key == 'm'){ // if the m key is pressed
port.write(180); // send the value 180
   /* by enclosing each character in a single quotes, Processing referes to the
number that
   represents the ASCII code for the given letter*/
 }
}
```

Kinematics is the study of motions, and Forward Kinematics here represent the relationship between the position of a joint and the angles of the other joints along the same limb. For example, the position of my right hand is determined by the changes I make to the angles of my shoulder and elbow. The advantage for applying Forward Kinematics is the ease of code implementation comparing with Inverse Kinematics.

Final milestone

Fourth milestone is to incorporate all of the pieces together and integrate everything. The final result should be that the little robot copies me. The Kinect sees me through the depth image and gather all of the information it needs and passes them to Processing where the information will be extracted from the depth image. Next, Processing will be sending the information to Arduino through serial and we already have the Arduino code in place and we tested everything individually "unit testing" which are the following.

1- Arduino controlling the Servos
2- Processing can talk to Arduino over Serial
3- Processing understand the depth image and can extract the information we need such as the angles of the shoulder and the elbow.

The other thing left is to add the Processing-Arduino serial communication code to the long code we have for tracking the skeleton and we should be done, in theory at least. The skeleton code already calculates the angles in the form we want in numbers between 0 and 180 which represent the degrees.
The first addition to the skeleton code is at the very top where we need to import the serial library, just like we did in testing the communication between Arduino and Processing earlier. In fact, most of the addition should be familiar to you at this point.

```
Import processing.serial.*;
```

```
Serial port;
```
Next, we need to find the serial port to which Arduino is connected and we do that in the "setup" function. IAs I explained earlier, it is important to have a matching baud rate which in tis case is 9600.
```
Println(Serial.list());
String portName = Serial.list()[0]; // change 0 to the appropriate serial port number
port = new Serial(this, portName, 9600);
```
The main change will be added in the "draw" function. Once we display the angles to the user, then we send the data to Arduino. It is important to remember to send the angles in an alternating manner:1st the shoulder, then the elbow, rinse and repeat. We also want to send both angles every time consecutively. To do that, we need to construct an array of bytes representing both angle values in order and then sending that whole array over the serial port all at once.
```
byte out[] = new byte[2];
out[0] = byte(shoulderAngle);
out[1] = byte(elbowAngle);
port.write(out);
```
note we call the byte function of every angle before storing them in the array because this way they will be rounded to the closest integer

that's it! For the demo please see the video on YouTube at this link
http://www.youtube.com/watch?v=HuhREgp6qmc&feature=g-upl

Bibliography

Borenstein, G. (2012). *Making Things See.* Sebastopol, CA, USA: O'Reilly Media, Inc.
iFixit. (2012, Jan 30). *Microsoft Kinect Teardown.* Retrieved from http://www.ifixit.com/Teardown/Microsoft+Kinect+Teardown/4066/1
Inverse Kinematics - Improved Methods. (n.d.). Retrieved from http://freespace.virgin.net/hugo.elias/models/m_ik2.htm
Make Magazine. (n.d.). *Kinect Hacking: Weather Map.* Retrieved from Make Projects : http://makeprojects.com/Project/Kinect+Hacking%3A+Weather+Map/1837/1#.UHudY2l26pU
McDonals, K. (n.d.). *Kyle McDonals.* Retrieved from Selected Work: http://kylemcdonald.net/
Shiffman, D. (n.d.). *Getting Started with Kinect and Processing.* Retrieved from http://www.shiffman.net/p5/kinect/
Szeliski, R. *Computer Vision: Algorithms and Applications.* Springer.
Wikipedia. (n.d.). Retrieved from http://en.wikipedia.org/wiki/Forward_kinematic_animation