Integrating Microsoft Kinect with Arduino: Real-Time Skeleton Human Tracking on Android Puppet

I Putu Dody Lesmana
Department of Information Technology
Politeknik Negeri Jember, Jember 68101, Indonesia
dody@polije.ac.id

Beni Widiawan
Department of Information Technology
Politeknik Negeri Jember, Jember 68101, Indonesia
beniw@polije.ac.id

***Abstract*—This paper describes the use of capabilities of the Microsoft Kinect as skeleton human tracking. With the Kinect and Arduino will build a gesture-based puppeteering system that we will utilize to control a simple, servo-driven puppet. We use Processing and Simple-OpenNI to develop Kinect for Windows application to detect the user's joints and limbs movements in space by skeleton human tracking. In response to the user's skeleton movement, the user data will be mapped to servo angles and transmitted via serial and network communication to a remote receiver program, which will in turn send them to the Arduino board controlling the servos of the robotic puppet.**

Keywords—Kinect, Processing, Arduino, skeleton human tracking, Simple-OpenNI, NITE, serial communication, network

I. INTRODUCTION

Motion control computing is the discipline that processes, digitalizes, and detects the position and velocity of people and objects in order to interact with software systems [1]. Motion control computing has been establishing itself as one of the most relevant techniques for designing and implementing a natural user interface. Natural user interfaces are human-machine interfaces that enable the user to interact in a natural way with software systems [2]. Kinect embraces the natural user interfaces principle and provides a powerful multimodal interface to the user. We can interact with complex software applications or video games simply by using our voice and our natural gestures. Kinect can detect our body position, velocity of our movements, and our voice commands [3]-[5].

In this research, we are going to make use of one of the most amazing capabilities of the Kinect: skeleton tracking. This feature will allow to build a gesture-based puppeteering system that we will use to control a simple, servo-driven puppet [3]. Natural interaction applied to puppetry has an important outcome: the puppeteer (operator) and the puppet don't need to be tied by a physical connection anymore. In the second part, we will extend our project and implement network connectivity that will permit to control the puppet remotely from any point in the world, as shown in Figure 1.

This research will introduce to the ins and outs of skeleton tracking so we can detect the user's joints and limbs movements in space. The user data will be mapped to servo angles and transmitted via the Internet to a remote receiver program, which will in turn send them to the Arduino board controlling the servos of the robotic puppet. Then, we will also be learning how to drive servos from Arduino and how to communicate over networks with a few lines of code using Processing's Net library.

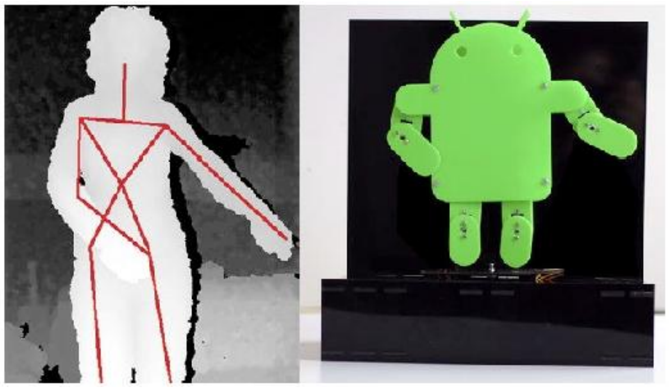

Fig. 1. Web puppet controlled by a remote user

II. DESIGN OF ANDROID PUPPET

A. Material

The parts list to build this project is shown in Table 1.

TABLE I. THE PARTS LIST

Part	Description
8 small servos	Hi Tec HS 55
1 big servo	Hi Tec HS-311
Strip board	94 x 53 mm copper
Extension lead cables	1 for each servo
23 breakaway headers-straight	-
1 green Perspex board	Colored acrylic 400 x 225 mm
1 black Perspex board	Colored acrylic 450 x 650 mm
Microcontroller	Arduino Uno
Wireless module	Arduino Xbee Shield

B. Servos

The movements of puppet will be based on the rotation of nine small servos. Servos are DC motors that include a feedback mechanism that allows to keep track of the position of the motor at every moment [3], [6]. The way we control a servo is by sending it pulses with a specific duration. Generally, the range of a servo goes from 1 millisecond for 0 degrees to 2 milliseconds for 180 degrees. There are servos that can rotate 360 degrees, and we can even hack servos to allow for continuous rotation. We need to refresh control signals every 20ms because the servo expects to get an update every 20ms, or 50 times per second [3]. Figure 2 shows the normal pattern for servo control.

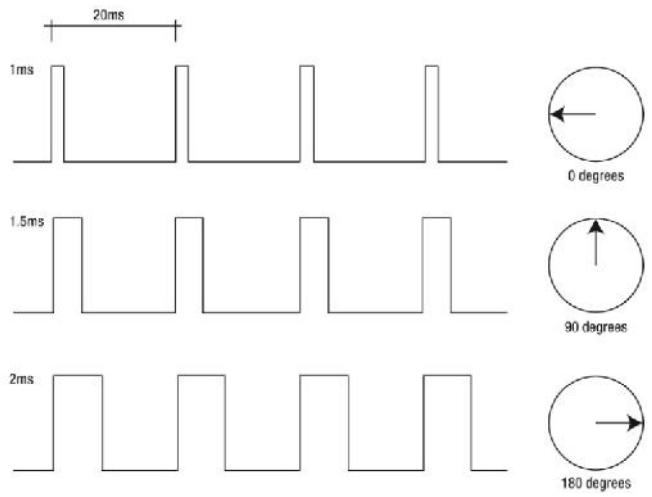

Fig. 2. Servo wave form

C. Building The Stage

Figure 3 shows the stage parts where the two small rectangular pieces have been added in order to support Arduino and give stability to the stage. The two round holes on the central element permit the servo cables to reach the Arduino board at the base. The rectangular hole is for the servo that rotates the robot. The stage consists of a hollow base to hide all the mechanisms plus a backdrop, as we can see in Figure 4.

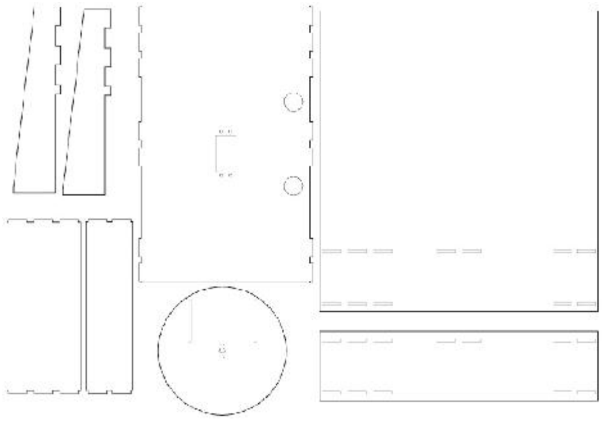

Fig. 3. Stage parts

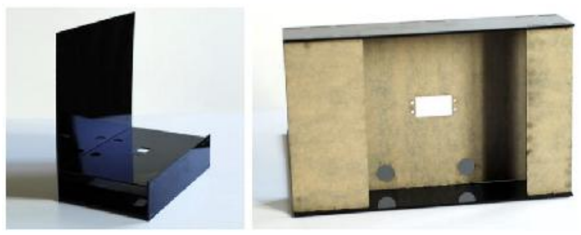

Fig. 4. Assembled stage

The robot is attached to two supports that are also made of black Perspex so they don't stand out from the background, as shown in Figure 5. These supports are connected to a disc, thereby allowing rotation. This disc will be screwed to a servo circular horn, and then attached to the rotation servo that we have previously fixed to the base.

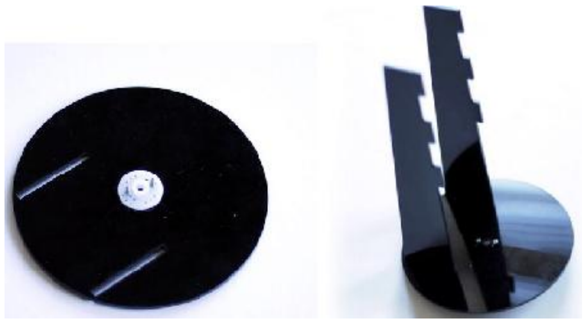

Fig. 5. Servo arm fixed to disc (left), and robot supports (right)

D. Building The Puppet

The puppet is made of laser-cut green Perspex and the design is shown in Figure 6. There is a front body that covers up all the mechanisms and a back body that connects all the servos and is fixed to the support. Both bodies are connected by four bolts passing through the holes provided for that purpose, which we can see in Figure 7. Before screwing and fixing all arms to the corresponding servos, we should set all servos to the middle position (1500) so later we can map them in an easy way. The complete arm will look like that in Figure 7. Repeat the same process with the other arm and legs. Once we have all these parts assembled, attach the arms and legs to the back body piece and fix the puppet onto the stage, as shown in Figure 8.

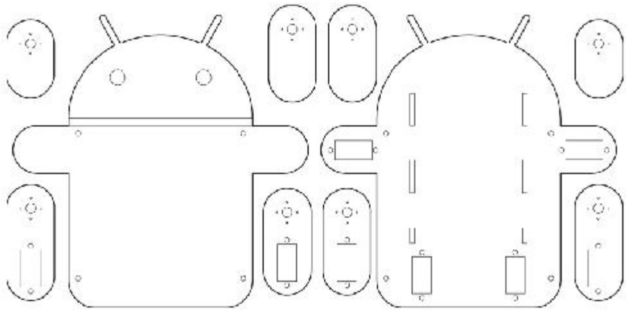

Fig. 6. Puppet pieces

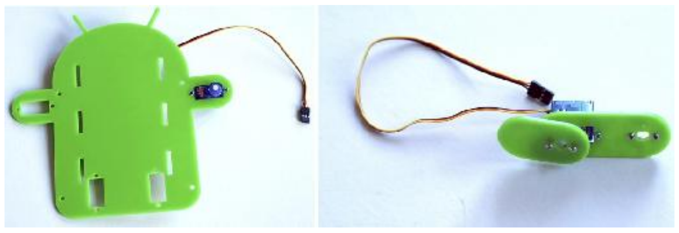

Fig. 7. Assembling servos to the back body and arm

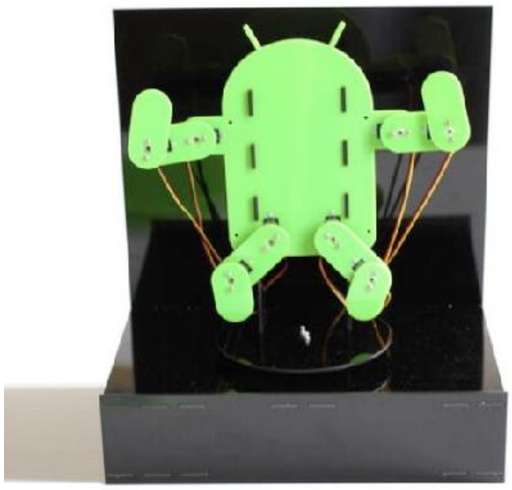

Fig. 8. The assembled puppet on the stage

E. *Building The Circuit*

The circuit that we are going to prepare is going to be plugged into the Arduino board and is designed to control nine servomotors. We first run all servos using just the power of USB connection, but we then add an external power source, as all servos working at the same time will exceed the power of the USB. We will use Arduino pins 3, 4, 5, 6, 7, 8, 9, 10, and 11, following the diagram in Figure 9.

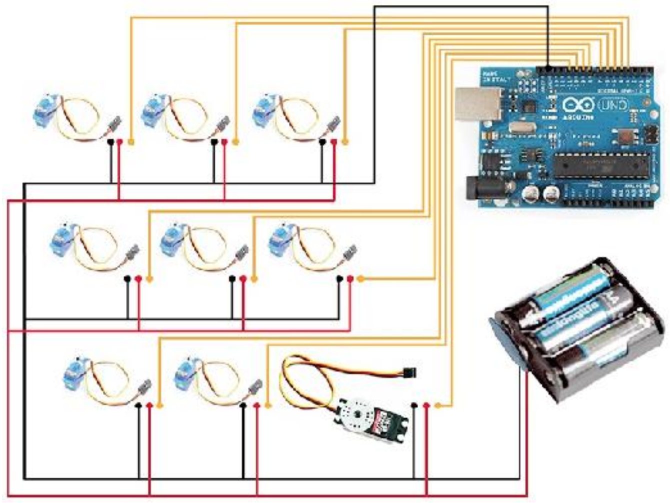

Fig. 9. Circuit diagram

III. IMPLEMENTATION

A. *Setting The Servos to the Starting Position*

Reconnect Arduino and the batteries and check that all servos move smoothly. The last thing we need to do is set all servos to a starting position, so we can fix all of the servo arms. We are going to send all shoulder and leg servos to 500 (0 degrees) or 2500 (180 degrees) depending if they are on the left side or right side, and the elbows to 1500 (90 degrees), allowing the movement in two directions. The last thing is to position the servo that will rotate the puppet to 1500 (90 degrees). Figure 10 shows all the angles.

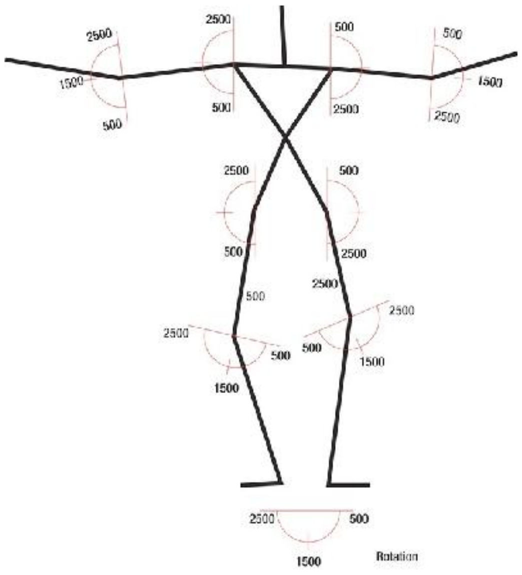

Fig. 10. Puppet servo angles

The code for setting the servos to the starting position is shown below.

```
int servo3Pin = 3; int servo4Pin = 4; int servo5Pin = 5; int servo6Pin = 6;
int servo7Pin = 7; int servo8Pin = 8; int servo9Pin = 9;
int servo10Pin = 10; int servo11Pin = 11; int servoPulse = 500;
int servoPulse2 = 1500; int servoPulse3 = 2500; int speedServo = 50;
unsigned long previousMillis = 0; long interval = 20;

void setup() {
  pinMode (servo3Pin, OUTPUT); pinMode (servo4Pin, OUTPUT);
  pinMode (servo5Pin, OUTPUT); pinMode (servo6Pin, OUTPUT);
  pinMode (servo7Pin, OUTPUT); pinMode (servo8Pin, OUTPUT);
  pinMode (servo9Pin, OUTPUT); pinMode (servo10Pin, OUTPUT);
  pinMode (servo11Pin, OUTPUT);
}

void loop() {
  unsigned long currentMillis = millis();
  if(currentMillis - previousMillis > interval) {
    previousMillis = currentMillis;
    updateServo(servo3Pin, servoPulse);//lef Shoulder
    updateServo(servo4Pin, servoPulse2);//left Elbow
    updateServo(servo5Pin, servoPulse);//left Hip
    updateServo(servo6Pin, servoPulse2);//left Knee
    updateServo(servo7Pin, servoPulse3); //right Shoulder
    updateServo(servo8Pin, servoPulse2);//rigt Elbow
    updateServo(servo9Pin, servoPulse3);// right Hip
    updateServo(servo10Pin, servoPulse2);//right Knee
    updateServo(servo11Pin, servoPulse2);//move it to the central position
  }
}

void updateServo (int pin, int pulse){
  digitalWrite(pin, HIGH);   delayMicroseconds(pulse);
  digitalWrite(pin, LOW);
}
```

B. *Skeleton Tracking On-Screen*

We will start off by tracking the skeleton on-screen, and then we will attach the virtual puppet to the physical one we have just built. Start by importing and initializing Simple-OpenNI.

```
import SimpleOpenNI.*;
SimpleOpenNI kinect;
```

Set the mirroring to On so it is easy to relate to on-screen image, and enable the depth image and the User. Pass the parameter SimpleOpenNI.SKEL_PROFILE_ALL to this function to enable all the joints. Set sketch size to the depth map dimensions.

```
public void setup() {
  kinect = new SimpleOpenNI(this);
  kinect.setMirror(true);  kinect.enableDepth();
  kinect.enableUser(SimpleOpenNI.SKEL_PROFILE_ALL);
  size(kinect.depthWidth(), kinect.depthHeight());
}
```

In the draw() loop, simply update the Kinect data, draw the depth map, and, if Kinect is tracking a skeleton, call the function drawSkeleton() to print skeleton on screen.

```
public void draw() {
  kinect.update(); image(kinect.depthImage(), 0, 0);
  if (kinect.isTrackingSkeleton(1)) { drawSkeleton(1); }
}
```

We took this drawSkeleton() function from one of the examples in Simple-OpenNI. It takes an integer as a parameter, which is the user ID. This function runs through all the steps to link the skeleton joints with lines, defining the skeleton that we will see on screen. The function makes use of the public method drawLimb() from the Simple-OpenNI class [7], [8].

```
void drawSkeleton(int userId) {
  pushStyle(); stroke(255,0,0); strokeWeight(3);
  kinect.drawLimb(userId, SimpleOpenNI.SKEL_HEAD,
    SimpleOpenNI.SKEL_NECK);
  kinect.drawLimb(userId, SimpleOpenNI.SKEL_NECK,
    SimpleOpenNI.SKEL_LEFT_SHOULDER);
  kinect.drawLimb(userId, SimpleOpenNI.SKEL_LEFT_SHOULDER,
    SimpleOpenNI.SKEL_LEFT_ELBOW);
  kinect.drawLimb(userId, SimpleOpenNI.SKEL_LEFT_ELBOW,
    SimpleOpenNI.SKEL_LEFT_HAND);
  kinect.drawLimb(userId, SimpleOpenNI.SKEL_NECK,
    SimpleOpenNI.SKEL_RIGHT_SHOULDER);
  kinect.drawLimb(userId, SimpleOpenNI.SKEL_RIGHT_SHOULDER,
    SimpleOpenNI.SKEL_RIGHT_ELBOW);
  kinect.drawLimb(userId, SimpleOpenNI.SKEL_RIGHT_ELBOW,
    SimpleOpenNI.SKEL_RIGHT_HAND);
  kinect.drawLimb(userId, SimpleOpenNI.SKEL_LEFT_SHOULDER,
    SimpleOpenNI.SKEL_TORSO);
  kinect.drawLimb(userId, SimpleOpenNI.SKEL_RIGHT_SHOULDER,
    SimpleOpenNI.SKEL_TORSO);
  kinect.drawLimb(userId, SimpleOpenNI.SKEL_TORSO,
    SimpleOpenNI.SKEL_LEFT_HIP);
  kinect.drawLimb(userId, SimpleOpenNI.SKEL_LEFT_HIP,
    SimpleOpenNI.SKEL_LEFT_KNEE);
  kinect.drawLimb(userId, SimpleOpenNI.SKEL_LEFT_KNEE,
    SimpleOpenNI.SKEL_LEFT_FOOT);
  kinect.drawLimb(userId, SimpleOpenNI.SKEL_TORSO,
    SimpleOpenNI.SKEL_RIGHT_HIP);
  kinect.drawLimb(userId, SimpleOpenNI.SKEL_RIGHT_HIP,
    SimpleOpenNI.SKEL_RIGHT_KNEE);
  kinect.drawLimb(userId, SimpleOpenNI.SKEL_RIGHT_KNEE,
    SimpleOpenNI.SKEL_RIGHT_FOOT);
  popStyle();
}
```

C. *Simple-OpenNI Events*

Use the Simple-OpenNI callback functions to trigger the pose detection and skeleton tracking capabilities. The function onNewUser() is called when a user is detected. We know the user is there, but we have not calibrated their skeleton yet, so we start the pose detection.

```
public void onNewUser(int userId) {
  println("onNewUser - userId: " + userId);
  if (kinect.isTrackingSkeleton(1)) return;
  println(" start pose detection");
  kinect.startPoseDetection("Psi", userId);
}
```

When the user is lost, we simply print it on the console.

```
public void onLostUser(int userId) {
  println("onLostUser - userId: " + userId);
}
```

When we detect a pose, we can stop detecting poses and request a skeleton calibration to NITE.

```
public void onStartPose(String pose, int userId) {
  println("onStartPose - userId: " + userId + ", pose: " + pose);
  println(" stop pose detection");
  kinect.stopPoseDetection(userId);
  kinect.requestCalibrationSkeleton(userId, true);
}
```

We also display messages when the pose is finished and when the calibration has started.

```
public void onEndPose(String pose, int userId) {
  println("onEndPose - userId: " + userId + ", pose: " + pose);
}
public void onStartCalibration(int userId) {
  println("onStartCalibration - userId: " + userId);
}
```

If the calibration is finished and it was successful, we start tracking the skeleton. If it wasn’t, we restart the pose detection routine.

```
public void onEndCalibration(int userId, boolean successfull) {
  println("onEndCalibration: " + userId + ", success: " + successfull);
  if (successfull) {
    println(" User calibrated !!!");
    kinect.startTrackingSkeleton(userId);
  } else {
    println(" Failed to calibrate user !!!");
    kinect.startPoseDetection("Psi", userId);
  }
}
```

Run this sketch and stand in front of Kinect in the start pose. After a few seconds, skeleton should appear on screen, and follow every movement until we disappear from screen (see Figure 11).

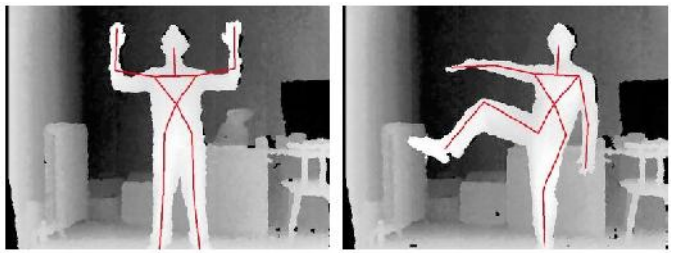

Fig. 11. Start position and calibrated user

D. *Angle Calculation*

Now we need to implement a routine that translates poses to the angles of eight servos that will ultimately drive the puppet. First, add some PVectors at the beginning of the sketch (outside of any function) to store the position of all joints.

```
// Left Arm Vectors
PVector lHand = new PVector();
PVector lElbow = new PVector();
PVector lShoulder = new PVector();
// Left Leg Vectors
PVector lFoot = new PVector();
PVector lKnee = new PVector();
PVector lHip = new PVector();
// Right Arm Vectors
PVector rHand = new PVector();
PVector rElbow = new PVector();
PVector rShoulder = new PVector();
// Right Leg Vectors
PVector rFoot = new PVector();
PVector rKnee = new PVector();
PVector rHip = new PVector();
```

We need a PVector array to contain the angles of joints.

```
float[] angles = new float[9];
```

We are interested in nine angles, one for each servo on puppet, namely angles [0] as rotation of the body, angles [1] as left elbow, angles [2] as left shoulder, angles [3] as left knee, angles [4] as left hip, angles [5] as right elbow, angles [6] as right shoulder, angles [7] as right knee, and angles [8] as right hip. We will be wrapping all these calculations into a new function called updateAngles() that we call from main draw() function only in the case we are tracking a skeleton. Add this function to the if statement at the end of the draw() function, and then print out the values of the angles array.

```
if (kinect.isTrackingSkeleton(1)) {
  updateAngles(); println(angles);
  drawSkeleton(1);
}
```

We need a function that calculates the angle described by the lines joining three points, so implement it next and call it the angle() function.

```
float angle(PVector a, PVector b, PVector c) {
  float angle01 = atan2(a.y - b.y, a.x - b.x);
  float angle02 = atan2(b.y - c.y, b.x - c.x);
  float ang = angle02 - angle01;
  return ang;
}
```

The function updateAngles() performs the calculation of all these angles and stores them into the previously defined angles[] array. The first step is storing each joint position in one PVector. The method getJointPosition() helps us with this process. It takes three parameters: the first one is the ID of the skeleton, the second one is the joint we want to extract the coordinates from, and the third one is the PVector that is set to those coordinates.

```
void updateAngles() {
  // Left Arm
  kinect.getJointPositionSkeleton(1, SimpleOpenNI.SKEL_LEFT_HAND,
  lHand);
  kinect.getJointPositionSkeleton(1,
    SimpleOpenNI.SKEL_LEFT_ELBOW,  lElbow);
  kinect.getJointPositionSkeleton(1,
    SimpleOpenNI.SKEL_LEFT_SHOULDER, lShoulder);
  // Left Leg
  kinect.getJointPositionSkeleton(1, SimpleOpenNI.SKEL_LEFT_FOOT,
    lFoot);
  kinect.getJointPositionSkeleton(1, SimpleOpenNI.SKEL_LEFT_KNEE,
    lKnee);
  kinect.getJointPositionSkeleton(1, SimpleOpenNI.SKEL_LEFT_HIP,
    lHip);
  // Right Arm
  kinect.getJointPositionSkeleton(1,
    SimpleOpenNI.SKEL_RIGHT_HAND, rHand);
  kinect.getJointPositionSkeleton(1,
    SimpleOpenNI.SKEL_RIGHT_ELBOW, rElbow);
  kinect.getJointPositionSkeleton(1,
    SimpleOpenNI.SKEL_RIGHT_SHOULDER, rShoulder);
  // Right Leg
  kinect.getJointPositionSkeleton(1,
    SimpleOpenNI.SKEL_RIGHT_FOOT, rFoot);
  kinect.getJointPositionSkeleton(1,
    SimpleOpenNI.SKEL_RIGHT_KNEE, rKnee);
  kinect.getJointPositionSkeleton(1, SimpleOpenNI.SKEL_RIGHT_HIP,
    rHip);
```

Now we have all joints nicely stored in the PVectors we defined for that purpose. These are three-dimensional vectors of real-world coordinates. We transform them to projective coordinates so we get the second angles, but first we need to extract body rotation angle (see Figure 12), which is the only rotation of the plane that we need.

```
angles[0] = atan2(PVector.sub(rShoulder, lShoulder).z,
PVector.sub(rShoulder, lShoulder).x);
```

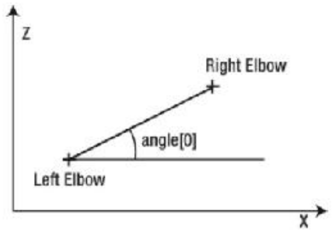

Fig. 12. Rotation of the body

After this angle has been calculated, we can transform all the joints to projective coordinates (on screen coordinates).

```
kinect.convertRealWorldToProjective(rFoot, rFoot);
kinect.convertRealWorldToProjective(rKnee, rKnee);
kinect.convertRealWorldToProjective(rHip, rHip);
kinect.convertRealWorldToProjective(lFoot, lFoot);
kinect.convertRealWorldToProjective(lKnee, lKnee);
kinect.convertRealWorldToProjective(lHip, lHip);
kinect.convertRealWorldToProjective(lHand, lHand);
kinect.convertRealWorldToProjective(lElbow, lElbow);
kinect.convertRealWorldToProjective(lShoulder, lShoulder);
kinect.convertRealWorldToProjective(rHand, rHand);
kinect.convertRealWorldToProjective(rElbow, rElbow);
kinect.convertRealWorldToProjective(rShoulder, rShoulder);
```

And finally, we use the angle function we implemented previously to compute all the necessary angles for the control of the puppet.

```
  // Left-Side Angles
  angles[1] = angle(lShoulder, lElbow, lHand);
  angles[2] = angle(rShoulder, lShoulder, lElbow);
  angles[3] = angle(lHip, lKnee, lFoot);
  angles[4] = angle(new PVector(lHip.x, 0), lHip, lKnee);
  // Right-Side Angles
  angles[5] = angle(rHand, rElbow, rShoulder);
  angles[6] = angle(rElbow, rShoulder, lShoulder );
  angles[7] = angle(rFoot, rKnee, rHip);
  angles[8] = angle(rKnee, rHip, new PVector(rHip.x, 0));
}
```

E. Communicating Over The Internet Using Applet

To communicate across networks, we are going to add this functionality to angle calculation sketch so we can send the angles over the network for another program to retrieve them from a remote location.

Server Applet: Sending the angles over a network

All helper functions and callbacks stay the same, but we add some lines to the setup() and draw() functions. We first import the Network library and declare a Server object.

```
import processing.net.*;
import SimpleOpenNI.*;
SimpleOpenNI kinect;
Server s; // Server Object
…
```

In the setup() function, initialize the server object, passing the port we are communicating through as a parameter.

```
public void setup() {
  …
  s = new Server(this, 12345); // Start a simple server on a port
}
```

And finally, at the end of the draw() loop and only in case we are tracking a skeleton, write a long string of characters to port. The string is composed of all the joint angles separated by white spaces, so we can split the string at the other end of the line.

```
public void draw() {
  …
  if (kinect.isTrackingSkeleton(1)) {
    …
    // Write the angles to the socket
    s.write(angles[0] + " " + angles[1] + " " + angles[2] + " "
      + angles[3] + " " + angles[4] + " " + angles[5] + " "
      + angles[6] + " " + angles[7] + " " + angles[8] + "\n");
  }
}
```

Client Applet: Controlling the puppet

The server applet is finished and ready to start sending data over the Internet. Now we need a client sketch to receive the angle data and send it to Arduino to get translated into servo positions that will become movements of puppet. We are going to implement a virtual puppet resembling the physical puppet (Figure 13). We will implement the serial communication at the same time.

After importing the Net and Serial libraries, create a boolean variable called serial that we set to false if we have not plugged the serial cable yet and want to try the sketch. We are going to be displaying text, so we also need to create and initialize a font.

```
import processing.net.*;
import processing.serial.*;
Serial myPort;
boolean serial = true;
PFont font;
```

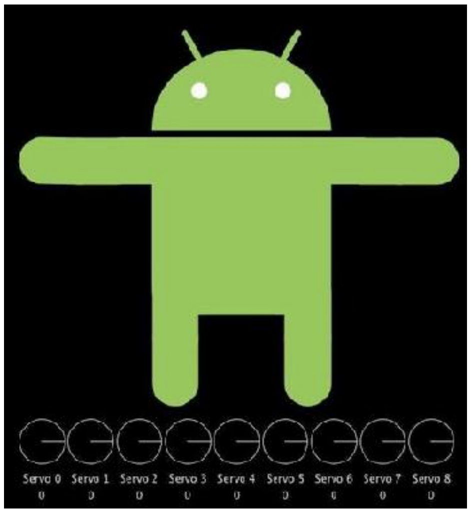

Fig. 13. Client applet

We need to declare a client object called c, declare a String to contain the message coming from the server applet, and a data[] array to store the angle values once we have extracted them from the incoming string. We are going to use a PShape to draw a version of physical puppet on screen so we can test all the movements virtually first.

```
Client c;
String input;
float data[] = new float[9];
PShape s;
```

In the setup() function, initialize the client object. Remember to replace the IP address with the external IP of the server computer if we are communicating over the Internet or the internal IP if we are communicating within the same local network.

```
void setup() {
  size(640, 700); background(255); stroke(0); frameRate(10);
  // Connect to the server's IP address and port
  c = new Client(this, "127.0.0.1", 12345);
  font = loadFont("SansSerif-14.vlw"); textFont(font);
  textAlign(CENTER);
  s = loadShape("Android.svg"); shapeMode(CENTER); smooth();
  if (serial) {
    String portName = Serial.list()[0]; // The first port on computer.
    myPort = new Serial(this, portName, 9600);
  }
}
```

Within the draw() loop, check for incoming data from the server; in a positive case, read is as a string. Then trim off the newline character and split the resulting string into as many floats as there are sub-strings separated by white spaces in incoming message. Store these floats (the angles of limbs sent from the server sketch) into the data[] array. Run a little validation test, making sure that all the values acquired fall within the acceptable range of -PI/2 to PI/2.

```
void draw() {
  background(0);
  // Receive data from server
  if (c.available() > 0) {
    input = c.readString();
    input = input.substring(0, input.indexOf("\n"));
    data = float(split(input, ' '));
```

```
        for (int i = 0 ; i < data.length; i++) {
          if(data[i] > PI/2) { data[i] = PI/2; }
          if(data[i] < -PI/2) { data[i] = PI/2; }
        }
      }
```

Once the data is nicely stored in an array, draw puppet on screen and add the limbs in the desired position. We first draw the shape on screen in a specific position, which we have figured out by testing. Then use the function drawLimb() to draw the robot's limbs using the angles in the data array as parameters.

```
shape(s, 300, 100, 400, 400);
drawLimb(150, 210, PI, data[2], data[1], 50);
drawLimb(477, 210, 0, -data[6], -data[5], 50);
drawLimb(228, 385, PI/2, data[4], data[3], 60);
drawLimb(405, 385, PI/2, -data[8], -data[7], 60);
```

Then draw an array of circles showing the incoming angles in an abstract way so we can debug inconsistencies and check that we are receiving reasonable values.

```
stroke(200); fill(200);
for (int i = 0; i < data.length; i++) {
  pushMatrix(); translate(50+i*65, height/1.2); noFill();
  ellipse(0, 0, 60, 60);
  text("Servo " + i + "\n" + round(degrees(data[i])), 0, 55);
  rotate(data[i]); line(0, 0, 30, 0); popMatrix();
}
```

And finally, if our boolean serial is true, use the sendSerialData() function to communicate the angles to Arduino.

```
if (serial)sendSerialData(); }
```

The drawLimb() function is designed to draw an arm or leg on screen starting at the position specified by the parameters x and y. The angle0 variable specifies the angle that we have to add to the servo angle to accurately display its movement on screen. angle1 and angle2 determine the servo angles of the first and second joint angles on the limb. Limbsize is used as the length of the limb.

```
void drawLimb(int x, int y, float angle0, float angle1, float angle2,
  float limbSize) {
  pushStyle(); strokeCap(ROUND); strokeWeight(62);
  stroke(134, 189, 66); pushMatrix(); translate(x, y); rotate(angle0);
  rotate(angle1); line(0, 0, limbSize, 0); translate(limbSize, 0);
  rotate(angle2); line(0, 0, limbSize, 0); popMatrix(); popStyle();
}
```

The function sendSerialData() works similarly, sending a triggering character to Arduino and then writing a series of integer data values to the serial port. These values are the angles mapped to a range of 0 to 250, expressed as integers.

```
void sendSerialData() {
  myPort.write('S');
  for (int i=0;i<data.length;i++) {
    int serialAngle = (int)map(data[i], -PI/2, PI/2, 0, 255);
    myPort.write(serialAngle);
  }
}
```

F. *Program Implementation*

If we run the server applet and then the client applet, we should be able now to control virtual puppet by dancing in front of Kinect, as shown in Figure 14.

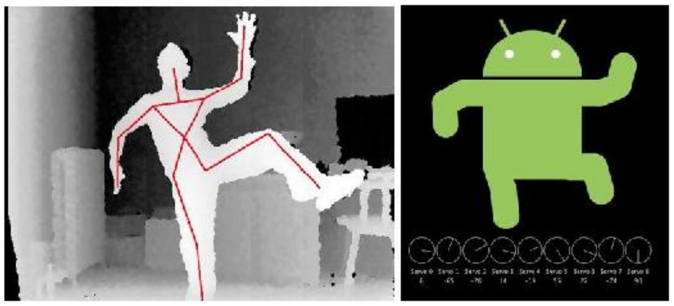

Fig. 14. Server and client applets communicating over a network

IV. CONCLUSION

In this research, we learned how to use NITE's skeleton tracking capabilities from Processing and how to use the data acquired to control a physical puppet. We were introduced to servos and how to control them from Arduino and also learned how to make use of network communication. The outcome of the research is a system with which we can control remote devices using natural interaction from home. In this example, we used it to control a puppet that literally copies our body gestures, but the concept can be endlessly extended. We could develop it to control more complicated devices, using other types of natural interaction algorithms and adapting it to your particular needs.

REFERENCES

[1] Tao, G., Archambault, P. S., & Levin, M. F. (2013, August). Evaluation of Kinect skeletal tracking in a virtual reality rehabilitation system for upper limb hemiparesis. In Virtual Rehabilitation (ICVR), 2013 International Conference on (pp. 164-165). IEEE.

[2] You, Y., Tang, T., & Wang, Y. (2014, August). When Arduino Meets Kinect: An Intelligent Ambient Home Entertainment Environment. In Intelligent Human-Machine Systems and Cybernetics (IHMSC), 2014 Sixth International Conference on (Vol. 2, pp. 150-153). IEEE.

[3] Parzych, M., Dabrowski, A., Cetnarowicz, D., Wroblewski, M., Szwed, H., & Staszewski, M. (2013, September). Investigation of gesture scenarios for servomotor control using Microsoft Kinect sensor. In Signal Processing: Algorithms, Architectures, Arrangements, and Applications (SPA), 2013 (pp. 112-117). IEEE.

[4] Fabian, J., Young, T., Peyton Jones, J. C., & Clayton, G. M. (2014). Integrating the microsoft kinect with simulink: Real-time object tracking example. Mechatronics, IEEE/ASME Transactions on, 19(1), 249-257.

[5] Catuhe, D. (2012). Programming with the KinectTM for Windows® Software Development Kit: Add gesture and posture recognition to your applications. " O'Reilly Media, Inc.".

[6] Giori, C. (2013). Kinect in Motion–Audio and Visual Tracking by Example. Packt Publishing Ltd.

[7] Banzi, M. (2011). Getting started with Arduino. " O'Reilly Media, Inc.".

[8] Ira/Xu Greenberg (Dianna/Kumar, Deepak). (2013). Processing: Creative Coding and Generative Art in Processing. Springer Verlag.