Brad Pitney

12/11/2013

ECE 578

Perkowski

Homework #3: Kinect-based Person Tracking with Simulated Countess Quanta Robot

# Introduction

For this project, I used Microsoft Kinect hardware along with Microsoft’s ‘Kinect for Windows SDK v1.8’ to implement person tracking for a simulated Countess Quanta robot. The sample program ‘SkeletonBasics-WPF’ included within the ‘Kinect for Windows Developer Toolkit v1.8’ was extended to support this simulation. Skeleton positioning data was used to control the servo 10 neck rotation of the simulated robot, so that it would turn to face the tracked person. A simulated robot was used in place of the actual Countess Quanta robot, due to issues encountered while trying to install and utilize the Kinect hardware under the Ubuntu 12.04 operating system, which hosts the existing Countess Quanta motion control software.

# Purpose

The intent of this project was to gain experience with using the Kinect system and add basic Kinect support to the Countess Quanta robot that could be extended in the future. Person tracking was selected because it complements the existing voice synthesis and music playing functionality that has already been developed for the Countess Quanta robot. For instance, the person tracking feature could be used during interaction with a human by having the robot turn to face the person before speaking. Person tracking can also be used as a starting point to develop other functionality, such as gesture recognition, since the same underlying skeleton tracking methods in the SDK could be used.

# Design Process

I had originally intended to implement this person tracking feature on my Ubuntu 12.04 virtual machine, to allow for control the Countess Quanta robot hardware through the existing software, which had been developed under Ubuntu. However, I found the process of establishing communication with the Kinect hardware to be far more complex and time consuming under Ubuntu than it had been under Windows 7. This is due in part to having to integrate separate hardware drivers, communication libraries, and skeleton tracking software from different developers, while ensuring that each component is of a version that is compatible with the operating system and with each other component. For example, many online guides to running Kinect under Ubuntu 12.04 indicate that you should install the latest version of OpenNI, which is a primary SDK used in communicating with the Kinect. However, after much research and experimentation, I found that support for the Kinect hardware under Linux has been removed in the 2.x version of OpenNI (latest version is 2.2.0.33, at the time of writing). These issues, combined with my own lack of expertise in Ubuntu, eventually caused me to abandon this path and transition to simulated robot development under Windows 7.

In Windows 7, I downloaded and installed both the ‘Kinect for Windows SDK v1.8’ and the ‘Kinect for Windows Developer Toolkit v1.8’, from Microsoft’s website. I then ran the included ‘Developer Toolkit Browser v1.8.0’ software, which displays a list of many sample programs for working with different aspects of the Kinect hardware. From here, I was able to run the ‘Skeleton Basics-WPF’ sample program, which immediately recognized my Kinect hardware and began displaying skeleton outlines of anyone within the Kinect’s visual range. From the Toolkit Browser, I was able to download the ‘Skeleton Basics-WPF’ source code and begin modifying it to simulate the intended behavior of the Countess Quanta robot.

To simulate the robot, I added a new ‘RobotDisplay’ form to the ‘Skeleton Basics-WPF’ project and added code to automatically display this form on startup, so that both the existing skeleton display and the simulated robot display would be visible. In the RobotDisplay class, I wrote some code to display a circle at the top of the form to represent a simplified top-down view of the Countess Quanta robot. This circle also represents the location of the Kinect sensor oriented towards the bottom of the form, since the Kinect will eventually be mounted to the front of the actual Countess Quanta robot. To represent the target person in front of the simulated robot, I added a second circle to the form, below the first circle. I added a red line from the robot to the target, to indicate the current facing of the robot’s head.

After reviewing the existing sample code and performing some tests with the Kinect, I found that the Kinect SDK provides three-dimensional Cartesian positioning data (in meters) for each skeleton joint. Each SkeletonJoint object contains an X coordinate reflecting the horizontal position of the joint, with the very center of the Kinect image being zero, the right side of the image being positive values, and the left side being negative. Similarly, the Y coordinate represents placement on the vertical axis. The Z coordinate represents the joint’s distance in the direction away from the front of the sensor, with the Kinect sensor being the zero point.

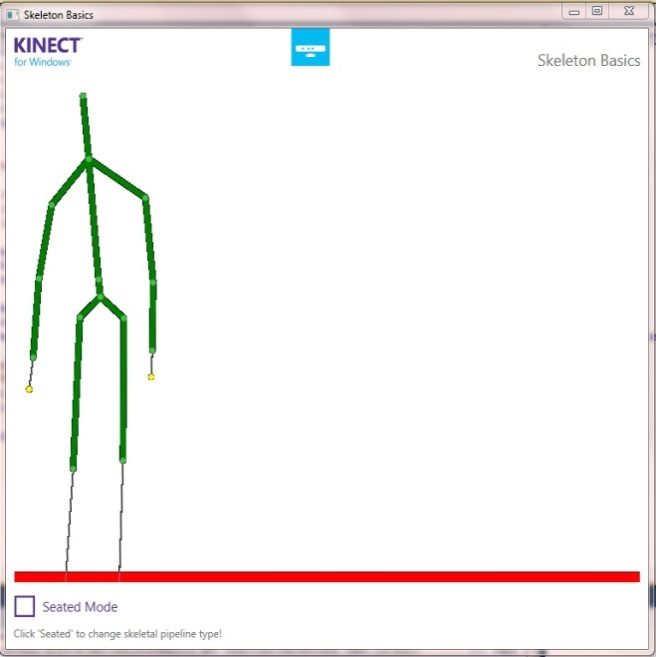
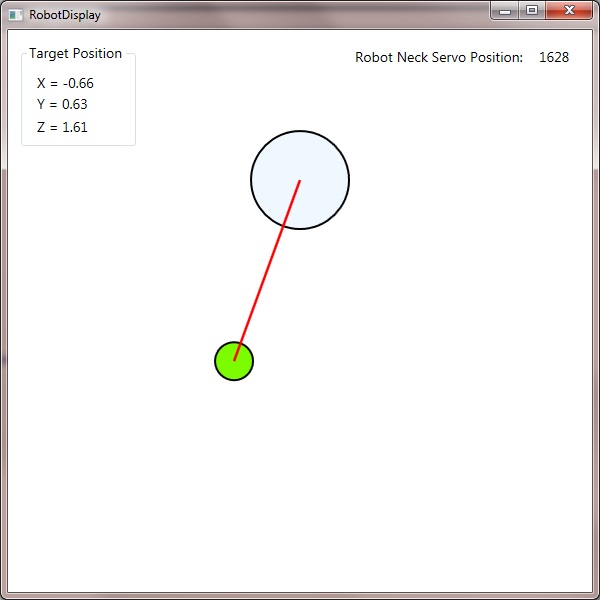
Since it makes the most sense for the robot to look at the target’s face, only the location of the skeleton’s ‘Head’ joint needs to be tracked. To modify the target’s location, I added an UpdateTargetPosition method to the RobotDisplay class, and called this method from the existing DrawBone method, which processes the skeleton joint data in the sample application. When the DrawBone method processes a ‘Head’ joint, it calls UpdateTargetPosition and passes the position data to this method. UpdateTargetPosition then displays the latest XYZ coordinates of the joint on the RobotDisplay form. To update the target’s top-down position relative to the robot, the X and Z coordinates are scaled and oriented to match the display on the form.

To determine the required position of the neck rotation servo for the robot to correctly face the target, I first experimented with the robot to collect range data on this servo. I found that the allowed physical range of this servo extended from position value 1350 at the farthest left rotation, to position value 1750 at the farthest right. At value 1550, the robot appears to be facing straight forward.

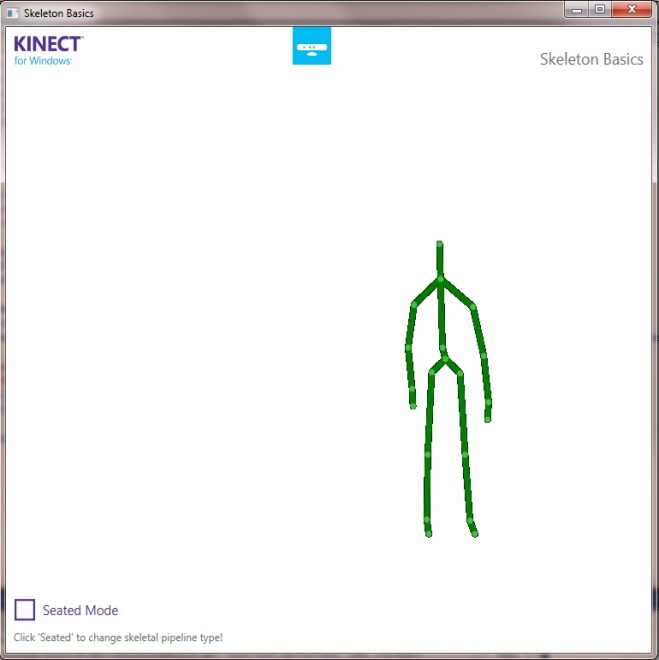
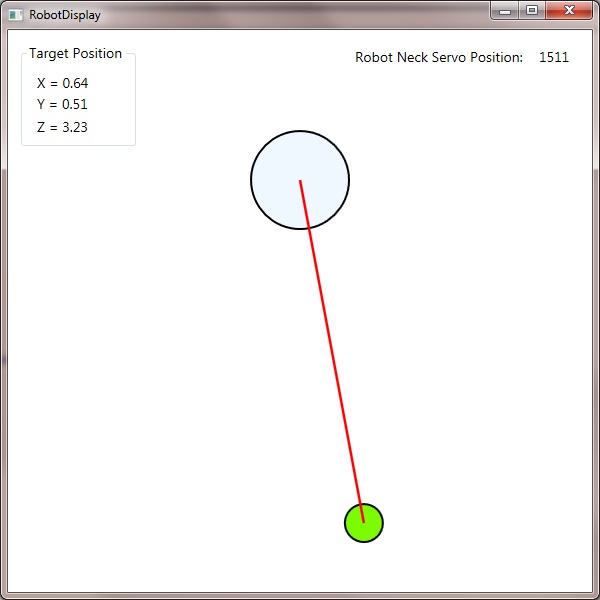
To map the Kinect data to these servo values, I added code to calculate the rotation angle off of center of the simulated robot’s facing direction, using arctangent and the targets X and Z Kinect location. I then scaled this by an approximated ‘servo increments per radian’ value and added the 1550 servo position offset. For the ‘servo increments per radian’, I didn’t know the exact value, so I estimated a value of 200, which kept the neck rotation within the 1350-1750 limits while moving within the Kinect area. This value will need to be revised once this logic is applied to the actual robot hardware. I displayed the final servo position at the top of the RobotDisplay form, for now. The expectation is that, if a motion control path existed (i.e. by either porting this Kinect functionality to Ubuntu, or moving the robot motion code to Windows), this value could be sent to the robot hardware and would result in the correct physical motion, with some minor calibrations.

# Results

The figures below are screenshots from the application, taken while the Kinect was tracking a person standing in different locations. The window on the left is the ‘Skeleton Basics’ sample application, which displays the skeleton data of a viewed person. The window on the right is the new RobotDisplay form, which shows the top-down view of the simulated robot (large blue circle) and the target person (small green circle). Note that the skeleton display is actually mirrored compared to what the Kinect actually sees, which matches the top-down view shown in the RobotDisplay form. Since the simulated robot is oriented towards the bottom of the form, it has to turn its head to the right to face the target. The displayed neck servo position of 1628 reflects this head rotation.

The figures below show the target in another location. In this case, the target has stepped back, away from the robot, and has stepped to the right of the robot (i.e. to the robot’s left). To track the target, the robot turns its head slightly left, to servo position 1511.

# Future Development

During development of the person tracking feature, many ideas arose for how this, and other Kinect functionality, could be extended in the future, to improve the human interaction capabilities of the Countess Quanta robot. Listed below are a few of these ideas:

* Integrate the person tracking feature with the Countess Quanta motion control software, to allow the robot to rotate its head and track the target.
* Add support for vertical tracking, allowing the robot to tilt its head up and down to look at the target’s face.
* Add support for multiple targets, such as periodically turning to face each detected target.
* Add support for recognizing and responding to gestures from the target.
* Integrate existing robot eye motion and base wheel motion to give the robot additional options in how it faces the target. For instance, the robot might face the target by turning its body, rotating its head, moving its eyes, or any combination of the three.
* Incorporate servo velocity control to make motions appear more natural. This could include modifications to reflect some emotional state, such as turning quickly to face the target if the robot is alert or turning slowly after some delay if the robot is tired.
* Use the target facing to display attentiveness of the robot. For instance, the robot might watch the target closely if the robot is focused, or it might periodically look around the room if the robot is distracted.

# Conclusion

Microsoft Kinect hardware was used with Microsoft’s ‘Kinect for Windows SDK v1.8’ to implement person tracking in a simulated Countess Quanta robot. The ‘SkeletonBasics-WPF’ sample program from the ‘Kinect for Windows Developer Toolkit v1.8’ was extended to support the robot simulation display. Spatial positioning data from the Kinect was sent from the existing sample program to update the location of the target person in the simulation. The target’s location was then used to calculate the approximate neck servo position that would be required for the physical robot to face the target.

# Source Code

The code below was added to the existing ‘SkeletonBasics-WPF’ project, which is sample code included in ‘Kinect for Windows Developer Toolkit v1.8’. Code was added to ‘MainWindow.xaml.cs’ to create and update the new RobotDisplay form. The ‘RobotDisplay.xaml.cs’ contain the new RobotDisplay class, which implements the simulated robot and person tracking.

## Modifications to MainWindow.xaml.cs

private void WindowLoaded(object sender, RoutedEventArgs e)

{

[Existing code omitted]

robotDisplayWindow = new RobotDisplay();

robotDisplayWindow.Show();

}

private void DrawBone(Skeleton skeleton, DrawingContext drawingContext, JointType jointType0, JointType jointType1)

{

[Existing code omitted]

// If the Head joint is updated, update the target position on the RobotDisplay form.

if (jointType0 == JointType.Head)

{

robotDisplayWindow.UpdateTargetPosition(joint0.Position);

}

}

## RobotDisplay.xaml.cs

// Brad Pitney

// ECE 578

// Fall 2013

// The RobotDisplay class implements a person tracking with a simulated Countess Quanta

// robot. Skeleton joint data from the Kinect hardware is used to update the location of

// the target person. This is used to calculate the new neck servo position that would

// cause the head to turn and face the target, if this were value were set on the physical

// robot. A top-down representation of the robot and target are displayed on the form.

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Windows;

using System.Windows.Controls;

using System.Windows.Data;

using System.Windows.Documents;

using System.Windows.Input;

using System.Windows.Media;

using System.Windows.Media.Imaging;

using System.Windows.Shapes;

using Microsoft.Kinect;

namespace Microsoft.Samples.Kinect.SkeletonBasics

{

/// <summary>

/// Interaction logic for RobotDisplay.xaml

/// </summary>

public partial class RobotDisplay : Window

{

public RobotDisplay()

{

InitializeComponent();

}

// Radius of the circle representing the Countess Quanta robot.

const double robotRadius = 50;

// Distance from the top of the form to the robot circle.

const double robotFromTop = 100;

// Radius of the circle representing the target person.

const double targetRadius = 20;

// Position of the robot neck servo (servo 10) at which the robot faces straight forward.

const double robotNeckServoCenterPostion = 1550;

// Number of servo position increments per radian of rotation for the robot's neck servo.

const double servoIncrementsPerRadian = 200;

Ellipse target;

Line targetLine;

// This method initializes the circles representing the robot and target, as well

// as the line indicating the direction the robot is facing.

private void WindowLoaded(object sender, RoutedEventArgs e)

{

const double defaultTargetDistance = 200;

// Circle representing the robot.

Ellipse robot = new Ellipse();

robot.Fill = Brushes.AliceBlue;

robot.StrokeThickness = 2;

robot.Stroke = Brushes.Black;

robot.Width = robotRadius \* 2;

robot.Height = robotRadius \* 2;

Canvas.SetTop(robot, robotFromTop);

Canvas.SetLeft(robot, RobotDisplayCanvas.ActualWidth / 2 - robotRadius);

RobotDisplayCanvas.Children.Add(robot);

// Circle representing the target person.

target = new Ellipse();

target.Fill = Brushes.LawnGreen;

target.StrokeThickness = 2;

target.Stroke = Brushes.Black;

target.Width = targetRadius \* 2;

target.Height = targetRadius \* 2;

Canvas.SetTop(target, robotFromTop + defaultTargetDistance);

Canvas.SetLeft(target, RobotDisplayCanvas.ActualWidth / 2 - targetRadius);

RobotDisplayCanvas.Children.Add(target);

// Line representing the facing direction of the robot.

targetLine = new Line();

targetLine.Stroke = System.Windows.Media.Brushes.Red;

targetLine.StrokeThickness = 2.5;

targetLine.X1 = RobotDisplayCanvas.ActualWidth / 2;

targetLine.Y1 = robotFromTop + robotRadius;

targetLine.X2 = RobotDisplayCanvas.ActualWidth / 2;

targetLine.Y2 = robotFromTop + defaultTargetDistance + targetRadius;

RobotDisplayCanvas.Children.Add(targetLine);

}

// The UpdateTargetPosition method is called from the DrawBone method in the sample

// 'Skeleton Basics' program. It updates the position of the target and calculates

// the required position of the neck servo to track the target on the actual robot.

public void UpdateTargetPosition(SkeletonPoint targetPosition)

{

// Update coordinates of the target joint.

targetXLabel.Content = "X = " + targetPosition.X.ToString("F2");

targetYLabel.Content = "Y = " + targetPosition.Y.ToString("F2");

targetZLabel.Content = "Z = " + targetPosition.Z.ToString("F2");

// Calculate the new target position, based on the Kinect data.

double displayScale = 100;

double targetCenterTop = robotFromTop + robotRadius + targetRadius + targetPosition.Z \* displayScale;

double targetCenterLeft = RobotDisplayCanvas.ActualWidth / 2 + targetPosition.X \* displayScale;

// Update the location of the target circle.

Canvas.SetTop(target, targetCenterTop - targetRadius);

Canvas.SetLeft(target, targetCenterLeft - targetRadius);

targetLine.X2 = targetCenterLeft;

targetLine.Y2 = targetCenterTop;

// Update the position of the Countess Quanta neck servo, to track the target.

double robotNeckServoPosition = robotNeckServoCenterPostion - Math.Atan(targetPosition.X / targetPosition.Z) \* servoIncrementsPerRadian;

neckServoLabel.Content = robotNeckServoPosition.ToString("F0");

}

}

}