Journal

# Learning Outcomes 2 (LO):

## LO: Adjust the Kinect sensor's angle to ensure the player is being tracked correctly

**Goal:**

Implement procedural adjustment of the Kinect sensor bar’s angle so that the player is kept within the field of view of the sensor.

**Problems:**

1. I had to research how to adjust the sensor’s angle and the limitations of the adjustments.
2. I needed to ensure adjustment would only happen if the player was in the correct area, to ensure accuracy.
3. The adjustment would need to happen more than once if the sensor got moved or in other unusual circumstances.
4. Gesture detection would need to be disabled during sensor adjustment.

**Solutions:**

I found the documentation for the Kinect sensor’s angle adjustment on msdn at <http://msdn.microsoft.com/en-us/library/microsoft.kinect.kinectsensor.elevationangle.aspx>. This informed me of the limits of the adjustments which could be made and also that there was a limit on the amount of consecutive adjustments which can be made to the angle.

To ensure the player was in the correct area for the adjustment to happen I employed a number of methods.

The first was to only allow the adjustment to happen while there is a skeleton in view and to perform the adjustment to suit the closest skeleton. The following is a simplified version of the code I used to track the closest skeleton.

if (this.ReadSkeletonFrame()) // This reads in skeleton data and returns true if successful.

{

Skeleton closestSkeleton = null;

foreach (Skeleton skeleton in this.skeletonData)

{

switch (skeleton.TrackingState)

{

case SkeletonTrackingState.Tracked: // Only check against fully tracked skeletons.

if (closestSkeleton == null || skeleton.Position.Z < closestSkeleton.Position.Z)

closestSkeleton = skeleton;

break;

default:

break;

}

}

// The code below goes here.

}

The second was to monitor the player’s depth value and not allow the angle adjustment to occur until the player was within a set range. This ensures that the player is far enough away that their entire body would be within the sensor’s field of view. Gesture detection is also disabled while the player is too close. The video below shows the user being informed to step back with their position in floor space displayed above.



The third was to readjust the angle if the player’s head or legs were out of view and all the previous criteria were met. This allows for correction from an inaccurate adjustment at the start of the game. For example; the player starts the game at an elevated point, the sensor adjusts to suit the elevated position and then the player steps down. The sensor would then readjust.

A simplified version of the code to implement the two of these is as below. The closest skeleton is the one calculated above. This calls the method to adjust the sensor on the first run through then sets the angle set flag to false if either the lower half or the upper half of the player’s body is not being tracked. This triggers a readjustment of the sensor angle on the next update. This code also prevents the gestures from updating while the sensor is adjusting.

if (closestSkeleton.Position.Z > this.minimumPlayerDistance)

{

if (this.kinectAngleSet)

{

//Update gesture data, etc.

this.kinectAngleSet = closestSkeleton.Joints[JointType.Head].TrackingState == JointTrackingState.Tracked && closestSkeleton.Joints[JointType.KneeLeft].TrackingState == JointTrackingState.Tracked;

}

else

{

this.AdjustSensorAngle(closestSkeleton);

}

}

The actual adjustment method is quite simple. If the player’s spine joint is being tracked I use basic trigonometry to calculate the angle between centre of the sensor’s field of view and the skeleton’s spine joint. The sensor is then adjusted by this angle. If the spine is not in view the head or a foot is used as a point of reference and the sensor is incrementally adjusted until the spine is in view.

private void AdjustSensorAngle(Skeleton skeleton)

{

if (skeleton.Joints[JointType.Spine].TrackingState == JointTrackingState.Tracked)

{

Vector2 jointMapping = Vector2.Normalize(new Vector2(skeleton.Joints[JointType.Spine].Position.Z, skeleton.Joints[JointType.Spine].Position.Y));

// Calculate the angle from the centre of the field of view to the skeleton’s spine

float angle = MathHelper.ToDegrees((float)Math.Asin(jointMapping.Y));

if (angle > this.thresholdAngleToBody || -angle > this.thresholdAngleToBody)

{

// The method TrySetElevetionAngle is just a wrapper for exception handling

// if the angle value is too high.

this.kinectSensor.TrySetElevationAngle(this.kinectSensor.ElevationAngle + (int)angle);

}

this.kinectAngleSet = true;

}

else if (skeleton.Joints[JointType.Head].TrackingState == JointTrackingState.Tracked)

{

this.kinectSensor.TrySetElevationAngle(this.kinectSensor.ElevationAngle - 4);

}

else if (skeleton.Joints[JointType.FootLeft].TrackingState == JointTrackingState.Tracked)

{

this.kinectSensor.TrySetElevationAngle(this.kinectSensor.ElevationAngle + 4);

}

}

## LO: Track a user's hand position smoothly, consistently and accurately

**Goal:**

Refine the tracking of the player’s hand position so the menu system is easier to navigate using Kinect gesture input.

**Problems:**

1. The on screen hand position had to be completely independent of the player’s body position.
2. I would need to research how to smooth the skeleton data coming from the Kinect sensor.
3. The changes would have to have a limited or no impact on the gestures which are already defined.
4. The hand tracking would need to allow people with varying arm lengths to reach all menu items.

**Solutions:**

I took a multi-pronged approach to this task. The first change I made was to increase the resolution of the depth-stream data which was being retrieving from the sensor. This allowed for greater precision when converting between 3D world space and 2D screen space and reduced jumpiness in the controls. This is done by setting the DepthImageFormat to a higher resolution value when enabling the depth stream.

this.kinectSensor.DepthStream.Enable(DepthImageFormat.Resolution640x480Fps30);

I then altered the algorithm I was using to position the hand on screen. I set this up so that it is based on the offset from the shoulder to the hand. This works by converting the shoulder’s position and the hand’s position into 2D space then calculating the offset from the shoulder to the hand. This value is then scaled and added to the centre position of the screen to get the on-screen hand position. This makes the on screen representation independent of the player’s overall body position. The following is the code to perform this task (with some defensive code removed). Tools.Convert is a method to convert a skeleton point into 2D space.

public Vector2 HandPosition

{

get

{

Vector2 handPosition = Tools.Convert(this.kinectSensor, this.gestureManager.HandPosition, this.coordinateMapper);

Vector2 shoulderPosition = Tools.Convert(this.kinectSensor, this.gestureManager.ShoulderPosition, this.coordinateMapper);

Vector2 scaling = this.screenDimensions \* 1.8f;

Vector2 relativeHandPosition = handPosition - shoulderPosition;

relativeHandPosition.X \*= scaling.X;

relativeHandPosition.Y \*= scaling.Y;

return (this.screenDimensions / 2.0f) + relativeHandPosition;

}

}

I then added smoothing parameters to the skeleton stream to reduce jitter in the controls and to make menu navigation smoother. I researched how to implement this and found the blog at <http://cm-bloggers.blogspot.ie/2011/07/kinect-sdk-smoothing-skeleton-data.html> to be the most useful by far. It gives a detailed breakdown of what each smoothing parameter does. I had to test this with a number of different parameter values so that I could keep the gesture detection quick and accurate while still getting the advantages of the data smoothing. To enable smoothing of the skeleton data the smoothing parameters must be passed to the skeleton stream when you are enabling it, as so:

TransformSmoothParameters smoothing = new TransformSmoothParameters();

smoothing.Smoothing = 0.6f;

smoothing.Correction = 0.2f;

smoothing.JitterRadius = 0.125f;

smoothing.Prediction = 0.5f;

smoothing.MaxDeviationRadius = 0.04f;

this.kinectSensor.SkeletonStream.Enable(smoothing);

## LO: Create a motor driven entity using multiple bodies and joints using the Farseer physics engine

**Goal:**

Implement a moveable game character which suits movement along platforms and uneven terrain using the Farseer physics engine.

**Problems:**

1. The character would have to move in a human-like manner.
2. The character would have to move over flat and curved surfaces with ease.
3. The character would have to be easily controllable.
4. I would have to research how to create joints using the Farseer physics engine.
5. The character would have to have modifiable/deformable collision bounds to accommodate both standing and crouching body states.

**Solutions:**

I found some information on creating a game character with reasonably realistic movement using Farseer at <http://amazingretardo.simiansoftwerks.com/2010/02/17/platformer-character-control-farseer-physics-engine/> and about three quarters of the way down [www.madgamedev.com/post/2010/09/09/Article-XNA-Farseer-Platform-Physics-Tutorial.aspx](http://www.madgamedev.com/post/2010/09/09/Article-XNA-Farseer-Platform-Physics-Tutorial.aspx). This solution eliminates solves a lot of the problems associated with moving a character in a 2D physics engine as the wheel can be used to simulate acceleration accurately and can also be used to reset some of the character’s abilities (explained further down).

I used this idea of a box connected to a motor driven circle as a base for my character. Basically a box is connected to a circle using a revolute (revolving) joint which turns it into a wheel. To prevent the box from just flopping around on the wheel it is kept upright using a fixed angle joint. To move the character all you then have to do is apply a motor force to the circle which causes it to spin, pushing the box forward.

My character needed to have a changeable height so I added a third body to this setup. The third body was a longer body which would span the full height of the character. The smaller box would only span the height of the crouching character. These two bodies would be joined using a weld joint. I could then use collision filtering to prevent collisions with the large body while the character is crouched and prevent collisions with the small body while the character is standing.

Original idea My Implementation

The bodies are created using the Farseer engine’s BodyFactory and the joints can be created using the JointFactory. The code to create the bodies and joints is below. (I have removed a lot of conversions and settings for simplicity)

// Full length body

this.fullBody = BodyFactory.CreateRectangle(physicsWorld, this.Width, this.Height \* 0.75f, density, this.fullBodyOffset);

this.fullBody.CollisionCategories = EntityConstants.StickManCategory;

// Half-length body

this.smallBody = BodyFactory.CreateRectangle(physicsWorld, this.Width, this.Height \* 0.25f, density, this.smallBodyOffset);

this.smallBody.IgnoreCollisionWith(this.fullBody);

this.smallBody.CollisionCategories = EntityConstants.StickManCategory;

// Wheel body

this.wheelBody = BodyFactory.CreateCircle(physicsWorld, this.Width / 2.0f, density, this.wheelBodyOffset);

this.wheelBody.IgnoreCollisionWith(this.smallBody);

this.wheelBody.IgnoreCollisionWith(this.fullBody);

// This is set high to prevent rolling when stationary and to give the greatest control over movement

this.wheelBody.Friction = float.MaxValue;.

this.wheelBody.CollisionCategories = EntityConstants.StickManCategory;

// Weld joint between two boxes.

this.bodyJoint = JointFactory.CreateWeldJoint(physicsWorld, this.fullBody, this.smallBody, this.smallBody.Position);

// Fixed angle joint to hold the boxes upright

this.angleUprightJoint = JointFactory.CreateFixedAngleJoint(physicsWorld, this.smallBody);

// Revolute joint to act as the entity’s motor

this.motorJoint = JointFactory.CreateRevoluteJoint(physicsWorld, this.smallBody, this.wheelBody, Vector2.Zero);

this.motorJoint.MotorEnabled = true; // Enables the use of the joint as a motor.

To achieve the collision size switching I used collision filtering. On a collision with the tall body, the collision is only processed if the character is not crouching. Inversely the collisions with the small body are disregarded unless the character is crouching.

The wheel can then be used to control the character’s ability to jump and collide with platforms. I refined the collision detection with the wheel to only reset the jump ability by checking if the point of the collision was below the wheel’s centre position. I used this page as a guide: <http://farseerphysics.codeplex.com/discussions/250916>. The code to do this is as follows:

// Adds the event handler for collisions to the wheel physics body

this.wheelBody.OnCollision += this.CollisionHandlerWheel;

// The return value of this event handler tells the physics engine

// whether to calculate the collision response or not.

// Multiple lines have been left out for simplicity.

private bool CollisionHandlerWheel(Fixture fixtureOne, Fixture fixtureTwo, Contact contact)

{

switch (fixtureTwo.CollisionCategories)

{

case EntityConstants.PlatformCategory:

collided = false;

// This checks if the platform is below the wheel if collisions

if (fixtureOne.Body.Position.Y < fixtureTwo.Body.Position.Y)

{

this.Land(); // Land resets the character’s state to running/standing.

collided = true;

}

break;

//…

}

}

## LO: Implement procedural population of verbal commands for use in speech detection

**Goal:**

Implement procedural population of the speech engine with the names of all selectable items in all menus so that I do not have to manually add commands as menus are added.

**Problems:**

1. The input manager would need to be fed with all possible commands.
2. Selectable items in the level select menu would have to retain their numerical value, for use in level selection, while displaying and responding to the verbally spoken version of the number.
3. The menu manager would have to be able to retrieve selectable item names from each menu.

**Solutions:**

In each menu, upon adding an item to the menu, it checks if the item is selectable and adds its name to a collection of selectable item names. This takes advantage of the polymorphic structure of my menu item hierarchy. The menu manager can then be called to amalgamate all of these collections into one single collection of selectable item names. The code for populating the menu’s selectable item collection is below.

public void AddItem(MenuItem menuItem)

{

if (typeof(MenuSelectableItem).IsAssignableFrom(menuItem.Type))

{

this.selectableItemNames.Add((menuItem as MenuSelectableItem).Name.ToUpperInvariant());

}

this.menuItems.Add(menuItem);

}

However this system breaks down somewhat when it comes to the level select menu. The problem is that the numerical values of the buttons are used in level selection but the numerical values cannot be used to build the grammar of the speech engine. For this I had to build a converter from a numerical value to its verbal counterpart (i.e. “1” goes to “one”). The converter only operates on numbers from 1 to 999 and works as follows;

I create lookup arrays of digits, teens and tens:

string hundred = "hundred";

string[] otherTens = { string.Empty, string.Empty, "twenty", "thirty", "fourty", "fiftey", "sixtey", "seventy", "eighty", "ninety" };

string[] teens = { "ten", "eleven", "twelve", "thirteen", "fourteen", "fifteen", "sixteen", "seventeen", "eighteen", "nineteen" };

string[] singles = { string.Empty, "one", "two", "three", "four", "five", "six", "seven", "eight", "nine" };

I then use division and modulus operations to count the number of hundreds, tens and digits and assign their verbal counterparts to the string numberAsWords:

string numberAsWords = string.Empty;

int leftover = number;

int hundredCount = leftover / 100;

leftover = leftover % 100;

if (hundredCount > 0)

{

numberAsWords += singles[hundredCount] + " " + hundred;

if (leftover > 0)

numberAsWords += " and ";

}

int tenCount = leftover / 10;

leftover = leftover % 10;

if (tenCount == 1)

{

numberAsWords += teens[leftover];

}

else if (tenCount != 0)

{

numberAsWords += otherTens[tenCount];

if (leftover > 0)

numberAsWords += " ";

}

if (leftover > 0 && tenCount != 1)

{

numberAsWords += singles[leftover];

}

return numberAsWords;

The level select menu then has to be treated slightly differently by the menu manager when amalgamating the selectable item names, to account for this. This is done as follows: (I have removed some error checks and other defensive code for simplicity).

public List<string> GetAllSelectableNames()

{

List<string> selectableNames = new List<string>();

foreach (MenuType menuType in this.menus.Keys)

{

switch (menuType)

{

case MenuType.LevelSelect:

foreach (string name in this.menus[menuType].SelectableItemNames)

{

// If name is “1” it is parsed to the integer 1, then converted to “one”.

selectableNames.Add(ConvertToWords.ConvertIntToWords(int.Parse(name)));

}

break;

default:

if (this.menus[menuType] != null)

{

selectableNames.AddRange(this.menus[menuType].SelectableItemNames);

}

break;

}

}

return selectableNames;

}

This collection is then passed to the input manager when trying to initialize the speech engine. The collection is then iterated through. Each element is used to create a grammar choice which the speech engine can interpret as a command. The original, manual, version of this code can be seen in the learning outcome; “Implement voice commands based on the Kinect's audio stream”.

// selectableNames is the list of selectable item names passed into this method.

this.speechEngine = new SpeechRecognitionEngine(recognizerInfo.Id);

Choices grammarChoices = new Choices();

foreach (string name in selectableNames)

{

grammarChoices.Add(new SemanticResultValue(name.ToLower(), name));

}

GrammarBuilder grammarBuilder = new GrammarBuilder();

grammarBuilder.Culture = recognizerInfo.Culture;

grammarBuilder.Append(grammarChoices);

Grammar grammar = new Grammar(grammarBuilder);

this.speechEngine.LoadGrammar(grammar);

## LO: Detect run and jump actions from Kinect input using gestures

**Goal:**

Implement action based input commands, using input from multiple gesture detectors. In plain English, using the gesture manager which I implemented earlier, monitor two gesture detectors tracking each of the player’s feet and interpret run and jump actions where appropriate.

**Problems:**

1. I would need to create a vertical swipe gesture detector, which would need to track feet and allow for switching off down gesture detection.
2. The DateTime which was used to track gesture times would not be accurate enough for differentiating between run and jump actions.
3. Converting the detected gestures from these detectors would need to be handled differently to the other gesture detectors.
4. The run action would need to be easy enough to do without heavy physical exertion.

**Solutions**

I created a vertical swipe gesture detector which I could toggle up and/or down gesture detection on and off. I created two instances of this gesture detector type in the gesture manager and set each of them to track one of the player’s ankles. I set them both to ignore downward swipe gestures.

On updating of the gesture manager, if a gesture detector which was tracking an ankle had detected a gesture a leg gesture processing method was called. This also updates a leg lift timer which is used by the process leg gesture method. This timer gives greater accuracy than DateTime.Now() or DateTime.UtcNow() as these are only accurate to within about 50ms. I had issues with run actions being detected as jump actions due to this inaccuracy initially.

DateTime.Now() is also horribly inefficient. See <http://www.eggheadcafe.com/tutorials/csharp/71b57428-6b59-4466-9762-ecb437ffac98/is-twitter-good-for-developers--and-datetimenow.aspx> or <http://jason-mitchell.com/xna/xna-and-c-calling-datetime-now-is-expensive/> if you are interested

if (this.lastLegLiftCounter < this.runTimeLimit)

{

this.lastLegLiftCounter += gameTime.ElapsedGameTime.TotalSeconds;

}

this.skeletonJoints = skeleton.Joints;

foreach (GestureDetector gestureDetector in this.gestureDetectors)

{

gestureDetector.Add(this.skeletonJoints[gestureDetector.JointToTrack].Position);

if (gestureDetector.GestureDetected != GestureType.None)

{

if (gestureDetector.JointToTrack == JointType.AnkleLeft || gestureDetector.JointToTrack == JointType.AnkleRight)

{

this.ProcessLegGesture(gestureDetector.JointToTrack);

}

}

}

This method kept track of which ankle was last lifted last and how long since the lift happened. If the current ankle being lifted was not the last ankle lifted then the method would check for a run or jump action. If both legs were lifted within a very short timeframe of each other, the action was taken to be a jump otherwise if they were both lifted within a longer timeframe the action was taken to be a run action. This window of detection for a run gesture would allow for detection of moderate march up to a fast sprint.

private void ProcessLegGesture(JointType jointTracked)

{

if ((this.lastLegLifted == JointType.AnkleLeft && jointTracked == JointType.AnkleRight) || (this.lastLegLifted == JointType.AnkleRight && jointTracked == JointType.AnkleLeft))

{

if (this.lastLegLiftCounter < this.jumpTimeLimit)

{

this.detectedGestures.Enqueue(GestureType.Jump);

}

else if (this.lastLegLiftCounter < this.runTimeLimit)

{

this.detectedGestures.Enqueue(GestureType.Run);

}

this.lastLegLiftCounter = 0.0;

}

this.lastLegLifted = jointTracked;

}

# Learning Outcomes 1 (LO):

## LO: Use the Kinect sensor as an input device

**Goal:**

Set up the use of a Kinect sensor correctly in the project.

**Problems:**

1. I had to research how this was done.
2. I needed to research how to initialize and dispose of the sensor’s resources correctly.
3. I needed to tailor the implementation to suit the requirements of a game’s update cycle.

**Solution:**

I used the guide found at <http://tobint.com/blog/kinect-for-windows-sdk/> and read through the source contained in the “XNA Basics” and “Kinect Explorer” samples, which are contained in the Kinect for Windows Developer Toolkit (part of the SDK), to get up and running. Most of this is aimed at use with Windows Presentation Foundation applications and the XNA version wasn’t suited to my needs either but they both contributed heavily to my final solution.

The following tries to initialize the sensor and enables the depth and skeleton data streams. This is done by checking if there are any Kinect sensors available, checking if the sensor is connected and ensuring that the device is not already in use by another process.

bool successful = false;

if (KinectSensor.KinectSensors.Count > 0)

{

this.kinectSensor = KinectSensor.KinectSensors[0];

if (this.kinectSensor.Status == KinectStatus.Connected)

{

this.kinectSensor.DepthStream.Enable(DepthImageFormat.Resolution320x240Fps30);

this.kinectSensor.SkeletonStream.Enable();

try

{

this.kinectSensor.Start();

successful = true;

}

catch (IOException)

{

// The device is in use by another process.

}

}

}

The following code gets the skeleton data from the skeleton data stream using polling. The solution here comes from the “XNA Basics” sample project in the Kinect for Windows Developer Toolkit.

using (this.skeletonFrame = this.kinectSensor.SkeletonStream.OpenNextFrame(0))

{

// Sometimes we get a null frame back if no data is ready

if (null == this.skeletonFrame)

{

return false;

}

// Reallocate if necessary

if (null == this.skeletonData || this.skeletonData.Length != this.skeletonFrame.SkeletonArrayLength)

{

this.skeletonData = new Skeleton[this.skeletonFrame.SkeletonArrayLength];

}

this.skeletonFrame.CopySkeletonDataTo(this.skeletonData);

return true;

}

## LO: Implement gesture tracking using the Kinect sensor

**Goal:**

Create a gesture detection system by tracking bone or joint positions over time.

**Problems:**

1. Figuring out how to create a gesture interpreter which can accept a variety of gesture types.
2. Converting the gestures detected into actionable game commands.

**Solutions:**

I used the project at <http://kinecttoolbox.codeplex.com/> as a guide for handling gesture tracking. This project was built for the use with Windows Presentation Foundation so I had to rewrite the gesture detector to suit XNA better.

Basically a gesture detector contains a list of gesture entries, each of which stores a position and a time. This abstract gesture detector is defined by the variables in this constructor:

public GestureDetector(JointType jointToTrack = JointType.HandRight, int maxRecordedPositions = 20, int millisecondsBetweenGestures = 0)

{

this.GestureDetected = GestureType.None;

this.gestureEntries = new List<GestureEntry>();

this.JointToTrack = jointToTrack;

this.maxRecordedPositions = maxRecordedPositions;

this.millisecondsBetweenGestures = millisecondsBetweenGestures;

}

It implements methods to add GestureEntry objects, reset the detector and flag a gesture as found. It leaves the method to look for a gesture as abstract so any inherited class must implement their own specific code to detect their specific gesture type. For instance the “push” gesture detector implements this as below. ScanPositions is a method which performs a search on the list of gesture entries based on the functions passed in.

protected override void LookForGesture()

{

if (this.ScanPositions(

(p1, p2) => Math.Abs(p2.Y - p1.Y) < this.pushMaximumHeight,

(p1, p2) => Math.Abs(p2.X - p1.X) < this.pushMaximumWidth,

(p1, p2) => p2.Z - p1.Z < 0.01f,

(p1, p2) => Math.Abs(p2.Z - p1.Z) > this.pushMinimumLength,

this.pushMinimumDuration,

this.pushMaximumDuration))

{

this.GestureFound(GestureType.Push);

return;

}

}

I then implemented a gesture manger class to monitor all of the active gestures which contains a collection of gesture detectors and a queue of detected gestures. This manages the updating of gesture detectors and allows for the input manager to query for any available gestures. I have created it this way to allow for the processing of multiple individual gestures into one unified action (e.g. running). The following is the update method of the gesture manager class. Skeleton is a parameter of the method.

this.skeletonJoints = skeleton.Joints;

foreach (GestureDetector gestureDetector in this.gestureDetectors)

{

if (this.skeletonJoints[gestureDetector.JointToTrack].TrackingState != JointTrackingState.NotTracked)

{

gestureDetector.Add(this.skeletonJoints[gestureDetector.JointToTrack].Position);

if (gestureDetector.GestureDetected != GestureType.None)

{

this.detectedGestures.Enqueue(gestureDetector.GestureDetected);

gestureDetector.Reset();

}

}

}

The input manager can then simply query the gesture manager for the next gesture detected and apply the correct game action.

## LO: Create an extendable NUI-friendly menu system

**Goal:**

Create a tile based menu system which will work with Kinect gesture input and voice commands.

**Problems:**

1. The menu system must be flexible enough to cater for multiple menu types and layouts.
2. Any menu text must be large, self-centring and easy to read.
3. If a menu item is selectable it must be selectable by position or by name and must share its name with the voice command system.
4. Any selectable item must be selectable using a hand gesture.

**Solutions:**

To tackle the issue of flexibility I utilized an inheritance hierarchy with its root at an abstract menu item class. This class only stores its position relative to its parent/owner. It also allows retrieval of its type. This will be explained later. It is defined as follows:

public abstract class MenuItem

{

public MenuItem(Vector2 relativePosition)

{

this.RelativePosition = relativePosition;

}

public Vector2 RelativePosition { get; protected set; }

public abstract Type Type { get; }

public abstract void Draw(Vector2 parentPosition);

}

I created a number of sub-classes which inherit from this. These include; A menu image class which draws a sprite centred on its relative position, a menu text class which renders text centred on its relative position and an abstract selectable menu item class which contains a bounding box, a name and a method to check if its bounding box has been hit. From the selectable menu item I created a button sub-class which stores a menu image for its background, a menu image for its icon and a menu text item to display its name.

I then created a generic menu class which stores a collection of menu items and a collection of selectable item names. As items are added to the menu, the list of selectable item names is generated. This is one of the places the Type property of the menu item class is used. This is done by taking advantage of polymorphism, as so:

public void AddItem(MenuItem menuItem)

{

if (typeof(MenuSelectableItem).IsAssignableFrom(menuItem.Type))

{

this.selectableItemNames.Add((menuItem as MenuSelectableItem).Name.ToUpperInvariant());

}

this.menuItems.Add(menuItem);

}

This class also implements methods to check for a selection based on a string input (for voice commands) and/or based on a position input. This is also done using the Type attribute and polymorphism. The following is the selection check based on the name passed in. (Some defensive code has been removed for simplicity.)

public string CheckForSelection(string name)

{

string itemFound = null;

foreach (MenuItem menuItem in this.menuItems)

{

if (typeof(MenuSelectableItem).IsAssignableFrom(menuItem.Type))

{

if (name.ToUpperInvariant() == (menuItem as MenuSelectableItem).Name.ToUpperInvariant())

{

itemFound = (menuItem as MenuSelectableItem).Name;

break;

}

}

}

return itemFound;

}

Although not implemented yet, this class allows for querying of the selectable item names contained within, which will allow for procedurally loaded grammar sets based on the contents of all the menus in the game.

I created a menu manager class to manage the active menus and to call events which change the game state. I created a menu factory class to abstract the creation of menus from the manager. The menus are stored in a dictionary with the menu type as the key and the menu as the value. This allows the manager to just change the active menu type and then update the corresponding menu stored at the active type’s location.

The menus are created in the manager as so:

this.ActiveMenu = MenuType.Main;

this.menus.Add(MenuType.Main, MenuFactory.CreateMainMenu(contentManager, spriteBatch, this.screenDimensions / 2.0f));

And updated as so, where selectionPosition and selectionName are parameters of the menu manager’s update method:

if (this.menus[this.ActiveMenu] != null)

{

string selectedItemName = null;

if (selectionPosition != Vector2.Zero)

{

selectedItemName = this.menus[this.ActiveMenu].CheckForSelection(selectionPosition);

}

else if (selectionName != null)

{

selectedItemName = this.menus[this.ActiveMenu].CheckForSelection(selectionName);

}

// State logic here based on selectedItemName

}

The manager sends the following events which are implemented by the game to control game state:

/// <summary>

/// An event triggered on the user selecting exit.

/// </summary>

public event Action<bool> OnQuitGameDetected;

/// <summary>

/// An event triggered on the user selecting to continue gameplay.

/// </summary>

public event Action<bool> OnResumeGameDetected;

/// <summary>

/// An event triggered on the user selecting to start a new level.

/// </summary>

public event Action<bool> OnBeginLevelDetected;

I implemented a push gesture, as mentioned in the implement gesture tracking LO, to act as a selection gesture while the game is in the menu state. I also created a class containing string constants which are used for naming selectable menu items and in the creation of grammar for voice commands. I intend to replace this with procedurally set voice commands, based on the contents of the menus.

Note: In the following video the red circle represents the users hand position, translated into screen coordinates. Also any actions which are happening without any on screen input are happening as a result of voice commands.



## LO: Configure Visual Studio to create an installer package which will download and install custom dependencies

**Goal:**

Figure out how to configure the Visual Studio “publish” functionality to include custom dependencies in the installer package and to download these dependencies.

**Problems:**

1. Research adding custom dependencies to a Visual Studio generated installation package.
2. Once I had the installer working, the installed game kept crashing when the Kinect sensor was plugged in on any computer except my development machine.

**Solutions:**

I did some research into adding custom dependencies and found this StackOverflow post (<http://stackoverflow.com/questions/1334436/adding-custom-prerequsites-to-visual-studio-setup-project>) which further linked me to a msdn article (<http://msdn.microsoft.com/en-us/library/ms165429.aspx>) on how to create custom “bootstrappers” for Visual Studio.

Basically, to add custom dependencies to an installation package in Visual Studio you first have to create custom bootstrapper files for the dependencies and then add these files to the directory that Visual Studio reads its additional dependencies list from.

The msdn article actually had some incorrect information in it. I found this out by reading the installation log of a failed reinstallation. It turned out that the XML below was the offender. The product and property tags were the wrong way around which was causing installer errors.

<InstallChecks>

<MsiProductCheck

Product="IsMsiInstalled"

Property="{XXXXXXX-XXXX-XXXX-XXXX-XXXXXXXXXXXX}"/>

</InstallChecks>

The XXXXX….XXXXX section is the GUID of the installed msi file. This is required to bypass the installation of components which are already installed. I tried to find the GUIDs for my dependencies but to no avail. I ended up finding this blog (<http://elmaskubilay.blogspot.ie/2012/06/find-guid-globally-unique-identifier-of.html>) on how to get the list of programs and their msi GUID values. I ran these steps on my own machine and then searched through the generated file for the correct GUIDs. This worked a treat.

The next step was to get the installer package to download the dependencies instead of including them in the installation package. To do this I had a look at some of the package and product files which were already there. It turned out that if you included a link to the download location in the package.xml file, referenced this in the product.xml file and changed the CopyAllPackageFiles value as below I could achieve this result.

Product.xml:

<PackageFiles CopyAllPackageFiles="IfNotHomeSite">

<PackageFile Name="KinectRuntime-v1.6-Setup.exe" HomeSite="KinectRedistHomeSite" />

</PackageFiles>

Package.xml:

<Strings>

<String Name="KinectRedistHomeSite">DownloadLocationHere</String>

<Strings>

The issue with the installed game crashing when the Kinect sensor was connected turned out to be due to a lack of support, in the redistributable, for the Xbox Kinect sensor. The Kinect for Windows runtime only supports the Kinect for Windows device. I found this out by writing a basic logger and outputting states of objects before and after their initialization. I implemented a simple check for this and defaulted to keyboard input if status of the device was not supported. The code to do this is in the following line in the learning outcome “Use the Kinect sensor as an input device”.

if (this.kinectSensor.Status == KinectStatus.Connected)

## LO: Implement voice commands based on the Kinect's audio stream

**Goal:**

Using the Kinect sensor and the Microsoft Speech platform to accept voice input and convert a set of speech commands into in-game commands.

**Problems:**

1. I need to research how audio is retrieved from the Kinect for Windows SDK.
2. I need to research how to use the Microsoft Speech platform to interpret sounds as words.
3. I need to get these two technologies working together.

**Solution:**

I found the following sources as a guide and created a solution that meets the project’s needs:

The Purple Book: (<http://channel9.msdn.com/coding4fun/kinect/The-Purple-Book-Using-Kinect-for-Windows-with-XNA>)

SpeechBasics-WPF tutorial: Contained in the Kinect developer toolkit browser which is installed as part of the Kinect for Windows SDK.

The following is the event which is fired when the speech engine may have detected a registered piece of grammar. SpeechConfidenceThreshold is a constant which defines the cut off level of confidence at which to ignore false positives.

private void SpeechRecognized(object sender, SpeechRecognizedEventArgs e)

{

if (e.Result.Confidence >= InputManager.SpeechConfidenceThreshold)

{

this.selectedWord = e.Result.Semantics.Value.ToString();

}

}

The following starts the speech engine and adds two words to the grammar recognized by the speech engine instance. The second last line sets the Kinect sensor’s audio stream as the input of the speech engine. The SelectableNames class contains string constants. It is also possible to build grammar sets from predefined files but this does not suit my needs, as I intend to load all of the menu control grammar dynamically.

RecognizerInfo recognizerInfo = this.GetKinectRecognizer();

this.speechEngine = new SpeechRecognitionEngine(recognizerInfo.Id);

Choices grammarChoices = new Choices();

grammarChoices.Add(new SemanticResultValue(SelectableNames.PlayButtonName.ToLower(), SelectableNames.PlayButtonName));

grammarChoices.Add(new SemanticResultValue(SelectableNames.PauseCommandName.ToLower(), SelectableNames.PauseCommandName));

GrammarBuilder grammarBuilder = new GrammarBuilder();

grammarBuilder.Culture = recognizerInfo.Culture;

grammarBuilder.Append(grammarChoices);

Grammar grammar = new Grammar(grammarBuilder);

this.speechEngine.LoadGrammar(grammar);

this.speechEngine.SpeechRecognized += this.SpeechRecognized;

this.speechEngine.SetInputToAudioStream(this.kinectSensor.AudioSource.Start(), new SpeechAudioFormatInfo(EncodingFormat.Pcm, 16000, 16, 1, 32000, 2, null));

this.speechEngine.RecognizeAsync(RecognizeMode.Multiple);

This code here instructs the speech engine to use the Kinect specific language pack for the culture set on the device (I have a default value but it is omitted here for the sake of simplicity).

RecognizerInfo kinectRecognizer = null;

foreach (RecognizerInfo recognizer in SpeechRecognitionEngine.InstalledRecognizers())

{

string value;

recognizer.AdditionalInfo.TryGetValue("Kinect", out value);

if ("True".Equals(value, StringComparison.OrdinalIgnoreCase) && CultureInfo.CurrentCulture.Name.Equals(recognizer.Culture.Name, StringComparison.OrdinalIgnoreCase))

{

kinectRecognizer = recognizer;

break;

}

}

# Log:

## 15/10/12

I attended a planning poker session with my project supervisor and the other students who are using Kinect in their project. This helped a lot with identifying oversights in my user scenarios and in narrowing down time estimates for the user scenarios.

## 18/10/12 & 19/10/12

I set up all of my stories and sub-tasks on Jira. I could not set up a sprint as I do not have administration privileges. I have emailed my supervisor in relation to this but will work around it, for now. I downloaded and installed all the required software for development with the Kinect and Xbox360.

## 20/10/12

I went about setting up my Microsoft account for developing for the Xbox360. I could not register my account due to a glitch in the system. I submitted a support ticket containing all the required details and the reproducible steps I took. I shall develop for Windows until this issue is resolved as I cannot deploy Xbox360 projects without this registration. This has and will cause severe delays, due to the loss of time today and the extra time it will take to port the game to the Xbox360 once this is resolved.



## 21/10/12

I created a Windows game to work on while I’m waiting on a response from the Xbox360 developer support team. I implemented the Farseer physics engine to update on-screen sprites, using keyboard input to apply forces. The engine turned out to be easier to implement than I expected. It contains far more helper functions and classes than the average Box2D port. This has helped recuperate some of the time lost yesterday.

Video available, if required.

I created the input manager to place a layer of abstraction between the game and the input device. This will allow me to debug while the Kinect is not present/available. This will also make the game easier to port to different platforms, should the opportunity arise. I have skeleton code in place for a number of input types but have only fully implemented keyboard input so far.

I researched what was contained in the Kinect SDK API and what functionality I would require from this. I read up on these features to get to grips with what I would be using. I also added Kinect device support to the project. This does not perform input yet but the device is supported and managed correctly.

## 23/10/12

Read up more on implementing polling to retrieve data from the Kinect sensor as this is better suited to games than event driven frame requests. I implemented this on the skeleton stream to track joint positions for any active skeleton in the field of view of the camera. The resources I used for reference here were the “Developer Toolkit Browser” and multiple articles from the msdn Channel 9 Coding4Fun site.

Some of the potential issues which I have identified from this are inaccurate readings for the positions of joints which are not fully visible and trying to identify which skeleton is the correct player skeleton.

## 25/10/2012

Today I started to implement state management logic for menu updating. I also started adding gesture tracking to the project. I used the project at <http://kinecttoolbox.codeplex.com/> as a guide for handling gesture tracking. This project was built for the use with Windows Presentation Foundation so I have had to port the gesture management to suit XNA better. I have only implemented a base gesture detector class so far. I intend to implement a gesture fully over the weekend.

## 26/10/2012

Today I implemented a gesture manager and basic swipe gestures. I used the project at <http://kinecttoolbox.codeplex.com/> as a guide again. I experienced a few minor issues porting the event based code to XNA. I implemented the gesture manager class to deal with the events and allow for simple frame based polling from my input manager.

## 27/10/2012

I changed the structure of the gesture detector and gesture manager again to get rid of the last of the event based logic. I edited my sprite class so each object now handles its own loading. I separated the file paths for images from the main game class into a static class of constants to isolate the hard typed strings from the main game logic.

I implemented displaying of the user's hand active position on screen and the querying of this position for menu selection.

I ran into a number of issues implementing this:

1. I had to separate the displayed hand position from that of the skeleton being tracked so it could be scaled across the screen, otherwise the user could only reach a small portion of the screen.
   1. The original skeletal hand position is then only used to trigger the gestures for selection.
   2. The separated screen hand position is then used when querying what position the user selected.
2. The position of the users hand needed to be offset by the shoulder position to centre it on the screen.
3. The position of the screen hand then needed to be scaled dynamically so that the user could reach all of the edges of the screen.

Video available, if required.

I researched speech recognition using the Microsoft Speech Platform SDK and implemented basic voice commands in the project. I used “The Purple Book” (<http://channel9.msdn.com/coding4fun/kinect/The-Purple-Book-Using-Kinect-for-Windows-with-XNA>) and the “SpeechBasics-WPF” tutorial, from the Kinect developer toolkit browser, as points of reference when implementing this. I have only added basic voice control so far so have not come across any major problems.

## 28/10/2012

I implemented most of the required menu items and basic menu logic. This is taking a lot longer than I originally estimated. This is mostly due to the additional time required to ensure that both voice and touch input are catered for.

## 29/10/2012

I implemented the rest of the core of the menu system, created a few menu icons and created the main menu. I got to test the menu system also. Everything implemented so far is in order.



I implemented and tested both gesture and voice control in the menu system and implemented voice control to pause the game. This has taken less time to complete than originally estimated. This is mostly due to the extra time I spent on implementing the menu system correctly over the last two days.

See video in learning outcome: Create an extendable NUI-friendly menu system.

## 30/10/2012

I implemented game pad and touch input management logic in the input manager. I also refined some of the gestures which I implemented and refined the implemented menus some more.

## 31/10/2012

I went about building a stand-alone installer for the game but it did include all of my dependencies so it failed. I researched how to create “bootstraper” packages to add custom dependencies to a Visual Studio project on msdn (<http://msdn.microsoft.com/en-us/library/ms165429.aspx>). I also had to research how to get the GUIDs of installed applications so I could set the installer to bypass elements if they are already installed. I found out how to do this using PowerShell here (<http://elmaskubilay.blogspot.ie/2012/06/find-guid-globally-unique-identifier-of.html>). I then created bootstrapper packages for both the Kinect runtime and the Microsoft Speech runtime. This required a few trials to get the settings correct.

Once I got the installer working, I ran into another problem. The installed game would run fine if the Kinect sensor was not connected but on running it with the sensor connected the game crashed on initialization with no exception thrown. This was not reproducible on my own machine. To pinpoint where the error was I wrote a basic logger and output the initialization steps which were being hit. I pinpointed the issue to where the Kinect audio source was being started. I then output all of the relevant Kinect properties, only to find the status flag as “device unsupported”. The problem was that the Kinect SDK allows you to use the Xbox360 sensor but the Kinect Runtime only allows the use of a Kinect for Windows sensor. To fix this issue I added logic to check if the device is supported then fall back on keyboard input if it is not.

## 2/11/12

I implemented a vertical swipe gesture. I added logic to the gesture manager class to track both feet, using the vertical swipe gesture detector and detect if the user is running or jumping. This is done by logging the leg which last rose (via a swipe up gesture on an ankle joint) and the time. Dependant on the time between leg rises the action is detected as a jump action, a run action or ignored.

I also implemented the crouch and stand gestures. I based these on the vertical swipe gesture detector as well. The head is tracked in this case instead.

I then tested and tweaked these gestures by moving an in game object and changing the values settings to suit.

## 3/11/12

I altered the vertical swipe gesture detector with the ability to switch on and off detection of up or down gestures. This is useful for detecting jumping and running as I only wanted to track the up swipe for these and also saves compute time as it decreases the number of gesture entry searches.

I also created the basis of the stick man class. I am implementing the physics for this using three bodies and the method described here (<http://amazingretardo.simiansoftwerks.com/2010/02/17/platformer-character-control-farseer-physics-engine/>) for movement of the character.

This should allow the player to move freely over any terrain and allow me to switch on/off collisions with the different parts of the player based on their pose (standing, crouching, etc.)

## 04/11/12

I fixed the menu system so it is easily navigable using the keyboard or game pad. I swapped all the DateTime.Now calls in the project with DateTime.UtcNow calls as they are far less expensive. If you want to know more about this read the following; <http://www.eggheadcafe.com/tutorials/csharp/71b57428-6b59-4466-9762-ecb437ffac98/is-twitter-good-for-developers--and-datetimenow.aspx>, or <http://jason-mitchell.com/xna/xna-and-c-calling-datetime-now-is-expensive/>.

I implemented the movement of the player’s in game representation using the method I described yesterday. This required some trial and error to get the motor and weld joints positioned correctly between the wheel and the other two physics bodies in the object.

## 07/11/12

I modified the player’s update and run logic to allow for acceleration and deceleration of the player. I added some basic collision based state management to the player and added logic to reset the player and their physic objects.

## 09/11/12

I decided on a final structure for the game’s level content file. I implemented deserialization of level description files into settings classes for various game entities and implemented loading of the point data which defines the floor of a level. I ended up using a process which I already knew for this. It is described in this blog post: <http://flax.ie/wp7-game-development-a-beginners-guide-xml-specified-content/>. All of the entity descriptions are loaded by an entity loader. The positions of all entities, floor points and platforms are read in using a level loader. This method is fine for the levels which I create but I will need to save and load to/from xml files for custom levels, if I add a level editor to the game. This is due to limitations of the redistributable package.

## 10/11/12

I added a level manager to the game which will handle creation, destruction and updating of all entities within a level. I refactored all of the physics and player management code into the level manager and added logic to set the player’s position from the data loaded in by the level loader. I also added logic to create an edge list to define the level’s floor from the point data loaded in. All the edges are removed from the physics world at the beginning of a level and the list is repopulated.

## 16/11/12

I added a level factory to abstract the creation and disposal of the game entities from the level. I added more entity types and added the corresponding settings files for these entities. I created place holder classes for the platforms and interactive level entities so I could implement the bones of the level factory. I also started to implement the interactive entity class.

## 17/11/12

I fleshed out the abstract interactive object base class so that it sets up its physics body from the description passed into it and tidied up the player class to match the other entities more closely. I implemented the platform class fully so that it will create a platform of any length, based on a description passed in. I also implemented the creation of all platforms from a settings file passed in in the level factory. The level factory also disposes of these platforms.

I also added a class to hold data required to render an edge (as the required data cannot be pulled from a Box2D edge). I added logic so that a list of these is created and disposed of along with the floor. I also added logic to render the floor from this list.

I refactored the entity settings loader and entity constants classes so that I could filter entity settings from their names more easily. I utilized this in the level factory for creation of interactive entities. The code for their creation is there; I just need to implement the entity sub classes before they can be added to the game. I also added placeholder logic for the creation of the level exit entity.

I also added collision filtering so that the player can jump up and down through the platforms and land on them again, as the player would expect. I also added the collision filtering for when the player is crouched but I can’t test this until I add the mine cart.

## 18/11/12

I added a manager for creating, loading, saving and clearing a settings file. This includes conditional compilation to allow for differences in different platforms. I used this <http://msdn.microsoft.com/en-us/library/ff604992.aspx> as a guide, though I had written very similar code before. I then refactored some of the settings and entity description classes to match the coding standard I am applying.

I also implemented the masking (both visually and functionally) of selectable items which are disabled. I also implemented changeable pages within the menus. I then implemented the level select menu and refined the gestures used to navigate the menu system.

This whole process took longer than initially expected due to the large amount of work required before implementing the menu itself.

Video available, if required.

## 20/11/12

I created a 2D camera class to manage the scrolling action of the game. This will also allow for easy application of parallax scrolling and other effects without any modification to any other objects, at a later date.

I created most of the basic art assets for the level entities. I then went about starting the level editor. I decided to write it in XNA. I added a separate game project to the solution and set its content references to the SticKart game’s content project. This allows me to reuse the same assets between the two projects without duplication. I also added references to the SticKart project and the Custom content project. This allows me to use the same classes as the actual game which means that adding the editor to the game later will be much simpler.

I experienced some issues with setting up the shared content project. I couldn’t find a solution so I deleted the project and started again. The second time I set it up it worked so there was probably an issue in the project file or configuration file somewhere.

## 21/11/12

I went through the project to perform a code clean and ensure everything was adhering to the coding standard.

I implemented the addition and removal of floor points, platforms and other level entities in the level editor level. I added the rendering of floor points.

I implemented most of the controls of the level editor. I am having a small issue with the positioning of the floor edges but I think it is just related to how I am calculating the angles of these.

## 22/11/12

I fixed the floor positioning bug in the level editor. It turned out that I had a number of issues. I had forgotten to set the last position to the position being inserted. I was also calculating the angle incorrectly. I also added cart and switch placement to the level editor.

Video available, if required.

## 23/11/12

I implemented saving and loading of the levels in the level editor using serialization and deserialization of XML files. The output files will be fine for custom levels once it is implemented in the game, however I will need to write a tool to process these into the correct format for my precompiled levels. The format change is only minor so this should be fairly simple.

## 24/11/12

I implemented custom serialization logic in the level editor level. It can now save to either the format required for the xml serializer or the format required by the content manager to load compiled levels. This allows the greatest flexibility in the editor. I used the following blog post as a guide and built out the solution to meet my requirements: <http://davisxna.wordpress.com/2011/07/27/saving-xml-documents/>.

I implemented logic to snap items being placed to a grid. This will allow easier and more accurate creation of levels.

I also added procedural population of the speech engine with grammar from the menu system. This required the creation of a system to convert integers to their verbal representations so that they could be registered with the speech engine correctly.

I refined the hand and skeleton tracking. This was done using a number of techniques. The first was to increase the resolution of the depth stream. This leads to more accurate conversions from 3D space to 2D space. I then changed the algorithm I was using to position the on screen hand. I changed this so that hand position is now a scaled representation of the actual hand’s position relative to the shoulder position (which is positioned at the centre of the screen). I then enabled and configured data smoothing on the Kinect’s skeleton stream. I used this blog to find out what the smoothing parameters represented: <http://cm-bloggers.blogspot.ie/2011/07/kinect-sdk-smoothing-skeleton-data.html>.

## 25/11/12

I implemented a fix for the bug in the player collision detection. I did this by modifying the player’s bodies so that the large body is longer and the only upper body that detects collisions during running, standing, falling or jumping. While crouching the larger body then does not detect collisions and the smaller body does.

I also added further collision filtering to the wheel body so that it will only change the player’s state to standing if it collides with a platform from above. I used this page as a guide: <http://farseerphysics.codeplex.com/discussions/250916>.

I implemented a heads up display which presents the user with their score, health and active power up. I added the power up class to the game and added creation logic to the level manager to create these. I added power up management to the player so that the acquired power up affects the player’s abilities and disables after the correct amount of time.

I added collision management to the player to handle collision with power ups, obstacles and bonuses.

Video available, if required.

## 26/11/12

I implemented the scrolling death class and attached the game’s camera to it. I added logic to kill the player on collision with the scrolling death entity and added logic to stop updating the level if the player is dead.

## 29/11/12

I implemented the dynamic adjustment of the scrolling death entity’s speed based on the player’s state. I also added player camera tracking on the y-axis.

## 01/12/12

I implemented the cart class. The speed of the cart adjusts dynamically based on the player’s speed and position. I added collision detection between the player and the cart. Upon collision with the cart a joint is created between the player and the cart so that the player is moved by the cart. When the player jumps the joint is destroyed. This allows the player to enter and exit the cart easily. I had to add stabilizers (which are not drawn) to the cart to prevent it tipping over.

Video available, if required.

## 02/12/12

I implemented the switch and exit classes. The switch activates a mine cart. To get this working correctly I had to add an anchor joint to the mine cart so that it stays stationary until activated by the switch.

The exit class is similar to the switch but it is monitored by the level manager to detect the level complete state. I also added logic to update the number of unlocked levels in the game settings file, continue to the next level and end the game. I also implemented logic to restart the current level upon player death.

## 03/12/12

I fixed a bug with beginning a new level without finishing the current level. This was due to end level not being called.

I also fixed the bug which allowed the player to stand while underneath a platform which they could not fit under while standing. I did this by testing for a fixture above the player’s current position and blocking the stand call if there was a fixture there. I used <http://farseerphysics.codeplex.com/discussions/267958> as a guide on how to find fixtures by position.

## 04/12/12

I added tracking of the player’s position on a plane horizontal to the Kinect sensor so that they could be informed when they are too close. I added a basic notification to tell the player to step back when they are too close. I also added a display of their floor position so they can tell where they should be standing.

## 06/12/12

I fixed a few minor bugs, including the player’s movement when jumping from the mine cart and a level loading bug after completing the game.

I also implemented adjustment of the Kinect sensor’s angle, based on the angle to the player’s spine joint. I did this by calculating the angle to the player’s spine then adjusting the sensor by that angle. If the player’s spine is not in the field of view their head or foot is used as a point of reference and the sensor is incrementally adjusted until the spine is in the field of view. I used <http://msdn.microsoft.com/en-us/library/microsoft.kinect.kinectsensor.elevationangle.aspx> as a reference for this.

## 08/12/12

I designed and created the first two levels. I tested game play on these levels and found a few bugs which I then fixed. I added basic instructions for navigating the menu and playing the game. I will flesh this out into a proper tutorial once I have implemented the notification system in sprint three.

I made some game play changes to make the game easier and made some level tweaks. I also added the bonuses and obstacles to the game. This was quite simple due to the work I had already put in on the editor and level loader earlier.

## 11/12/12

I added the monitoring of the colour stream data. I added a way of outputting the colour stream data to the screen and added ability to toggle monitoring and displaying of the colour stream data on and off.

Used the XNA Basics (Part of the Kinect developer toolkit) as a guide for this. I created a colour stream renderer which turns the stream data into a texture and displays it.