The goal of this project was to use the Microsoft Kinect to track a person and render a lightsaber in their hands. The project was accomplished by utilizing several frameworks and renders that were available to me. I will start off by explaining how I set up the Kinect SDK and the frameworks that I used. From there, I will explain how the tracking was handled to successfully put the lightsaber in the player’s hands. Lastly, I will go over all of the features that I added into the program to flesh it out.

1. Getting Started with the Kinect SDK

In order to use the Microsoft Kinect on a computer instead of an Xbox one has to first install drivers and download the Kinect SDK. Both are now available from Microsoft and can be found at http://kinectforwindows.org/. Channel 9 has a great guide to an overview of the Kinect hardware and code which can be found at http://channel9.msdn.com/coding4fun/kinect/Getting-started-with-the-Kinect-for-Windows-SDK-quickly-with-the-Kinect-Quickstarts.

After the Kinect has been set up, we can take a look at the basic functions that the Kinect can perform. The Kinect has three different streams that it can use for gathering information about what it can see. The first is the Color Video Stream which is exactly what it sounds like, just a standard RGB image. We use this stream to display the final result of our project. The second stream is the Depth Stream. The Kinect has two cameras that combine to create a depth map. The two cameras are necessary to capture depth in 3d. The depth map saves each pixel as a distance value which we can use to create a grayscale image representing
distance from the Kinect. The last stream is the Skeleton. The Kinect can form a skeleton

![Image of Depth Stream, Skeleton, and Color Video Stream](image)

31 fps

of the player with the locations of 20 joints. This will become the most important stream in
achieving the results we desire.

### 2. Placing the Lightsaber in the Player’s Hands

Before I begin, the framework for the lightsaber code that I used can be found here:
The first piece of code that is relevant here is the lightsaber render itself. I did almost no modifications to this (as I am not a graphic artist), but I was able to determine where the color was determined and changed the color of the saber from red to green because I would much rather be a Jedi.

Code Snippet:

```
// Combine the two images.
return back + base + bloom * float4(0, 1, 0, 1);
```

This is a line from the BloomCombine.fx class that combines the 3 pieces of the render and multiplies them by a color value such as 0,1,0,1 which gives a green value.

Now that we have a working render, all that’s left is to attach it to the player. This is where the skeleton stream comes in handy. Here is the method in RenderGame.cs:

Code Snippet:

```
void DrawSabre(Vector3 leftPoint, Vector3 middlePoint, Vector3 rightPoint)
{
    sabre.View = Matrix.Identity;
    sabre.Projection = Matrix.CreatePerspectiveFieldOfView(0.87f,
        GraphicsDevice.Viewport.AspectRatio, 0.1f, 1000.0f);

        Vector3.Normalize(middlePoint - rightPoint)) / 3.0f);

    Vector3 rotAxis = Vector3.Cross(Vector3.Up, axis);
    float rotcos = Vector3.Dot(axis, Vector3.Up);


    sabre.World = Matrix.CreateScale(0.01f, 2.0f, 0.01f) * rotate *
        Matrix.CreateTranslation(leftPoint * new Vector3(1, 1, -1f)) *
        Matrix.CreateTranslation(0, -0.08f, 0);

    sabre.Draw();
}
```

We are trying to identify three things in this code that we want to put into sabre.World. First we have to identify the scale which is hardcoded here as (0.01f, 2.0f, and 0.01f) (the values are in meters). We then multiply that by the rotation angle which we find from the axis of the 3 points we are using. We depend on the hand joint, elbow joint, and wrist joint of the hand that we want
to put the saber into. We find each of these joint’s vectors and average them together. We take this normalized axis to find the angle that we want to place the lightsaber. The last bit that we put into Sabre.World is the translation (where in the X,Y coordinate do we place the sabre?). We simply move it to the location of the hand and then shift it up a few centimeters so it isn’t on top of the hand.

3. Fleshing Out the Features

Now that the lightsaber is in the frame and attached to the player’s hand, we can start implementing other features for the program.

The first feature that I tackled was Dual Wielding. The easy part was tracking all of the left side joints in addition to the right side joints. This was the only change I made to MainWindow.cs. For each joint that they initialized in SkeletonFrameReady, I also initialized the corresponding left side joint:
From there, the vast majority of the code changes were in RenderGame.cs. Once we had the main window properly tracking the joints, we had to initialize them in the RenderGame class with getters and setters. Once done, I had to change the draw method to take into account two lightsabers if the dual wield constant was negative.

```csharp
if (P1IsActive && gestureRecognizer.start == 1)
{
    if (dualWeild1 > 0)
    {
        DrawSabre(P1RightHandPosition, P1RightWristPosition, P1RightElbowPosition);
    }
    else
    {
        DrawSabre(P1LeftHandPosition, P1LeftWristPosition, P1LeftElbowPosition);
        DrawSabre(P1RightHandPosition, P1RightWristPosition, P1RightElbowPosition);
    }
}
```

Of course, in order to know whether one is dual wielding, I had to implement a trigger that would determine when the player wants to swap from single to dual wield. Here is the code:

```csharp
void CheckDualWeild1()
{
    if (Vector3.Distance(P1LeftHandPosition, P1RightHandPosition) < 0.2 &&
    Math.Abs(P1LeftHandPosition.Z - P1RightHandPosition.Z) < 0.1)
    {
        hold1++;  
    }
    else if (hold1 > 0 && Vector3.Distance(P1LeftHandPosition, P1RightHandPosition) > 0.6)
    {
        hold1 = 0;
        dualWeild1 = dualWeild1 * -1;
    }
}
```
CheckDualWield1 checks to see if the hands’ vectors are within 0.2 meters of each other (and that the depth is no more than 0.1 meter to take into account when the player is sideways, though this fix isn’t perfect). If the hands are that close, then it increments the hold1 integer beyond 0. As soon as the hands move further than 0.6 meters apart, then hold1 is thrown back to 0 and the dualwield constant is flipped to negative and the second saber will appear in the players left hand. The reason a hold variable is needed is so that the dualwield variable won’t flip while the hands are close together at which point the next frame would flip it back and so on.

Of course, the Kinect is able to track up to two separate skeletons at the same time, so that means it is possible to have two players. It is simple enough to render a lightsaber in the second player’s hand. Just check to see if the second skeleton is active and then call DrawSabre. The dual wield constant and method was also duplicated for player 2.

In addition to looking at dual wield states, I wanted to make it so that the Kinect would recognize a certain gesture before turning on the lightsabers (to simulate turning on the weapon). I called this gesture “DiagonalUpAndRight”. Here is the equation for determining if the gesture was performed:

Code Snippet:

```csharp
// Diagonal Up and to the Right
if (ScanPositions((p1, p2) => p2.Y - p1.Y > 0.2f, //Start vs Finish Height (p1, p2) => p2.X - p1.X > -0.01f, //Progression to the right (p1, p2) => Math.Abs(p2.X - p1.X) > 0.4f, //length 250, 1500)) //Duration
{
    RaiseGestureDetected(SupportedGesture.DiagonalUpAndRight);
    return;
}
```

This code looks at 4 different inputs to determine if the user performed this gesture. First of all, p1 and p2 represent the points of the joint at the start time and end time respectively which are determined by the Duration input (the gesture can be between 250 and 1500 milliseconds long). It has a Height input which I said must be greater than 0.2 meters, so the joint that it is tracking must travel at least 0.2 meters up. There is also a X,Y progression input. I wanted the gesture
to travel right so I said it the direction had to be positive (x-values increasing). Lastly there is a length input, which I set to only recognize the gesture if it went more than 0.4 meters in the right. This would then indicate that the gesture was performed and return. If you look back at the DualWield snippet where I called DrawSabre, you can see that there are 2 conditions that must be met for the sabre to be drawn. One is that player 1 must be active and the other is that the gestureRecognizer’s variable Start must equal 1 (which it will if DiagonalUpAndRight is recognized).

The last feature I added was to play a sound if the user swings his/her lightsaber. The framework comes with a few sounds, but they play on a continuous loop. Here is the code that I modified:

Code Snippet:

```csharp
if (currentSoundIndex == -1 || soundInstances[currentSoundIndex].State != SoundState.Playing)
{
    currentSoundIndex = rnd.Next(4);
    gestureRecognizer.Add(P1RightHandPosition, gestureRecognizer.sawGesture);
    if (gestureRecognizer.sawGesture > index && gestureRecognizer.start == 1)
    {
        soundInstances[currentSoundIndex].Play();
        index++;
    }
}
```

The first important addition is the presence of another Gesture Recognizer object. I implemented another gesture called “Swing” in the GestureRecognizer class which is defined as follows:

Code Snippet:

```csharp
// Swing (large shift in any direction)
if (ScanPositions((p1, p2) => Math.Abs(p2.Y - p1.Y) > 0.1f, //Start vs Finish Height
    (p1, p2) => Math.Abs(p2.X - p1.X) > 0.1f, //Right or Left movement
    (p1, p2) => Math.Abs(p2.Z - p1.Z) > 0.1f, //length
    250, 1000)) //Duration
{
    RaiseGestureDetected(SupportedGesture.Swing);
    return;
}
```
Much like the DiagonalUpAndLeft gesture from before, this gesture just looks for any large movement and then returns, indicating that the user is swinging the lightsaber and a sound should be played. I had to create a new variable called index to keep the sounds in check though. Otherwise, sounds would play before the player had drawn their lightsaber for the first time. Once the player has drawn their lightsaber however and gestureRecognizer.start is incremented to 1, then the if condition will be met. From there, everytime gestureRecognizer.sawGesture is incremented because the user swung (or performed a DiagonalUpAndRight gesture, which would meet the criteria a swing gesture anyway) the if condition is again met and a sound will play again.

4. Next Steps

There were a few features that I had hoped to implement but for some reason or another were just not feasible to get done. Perhaps the most important to me was a way to change the color of the lightsaber per player. I wanted Player 1 to have a green lightsaber and for Player 2 to have a red one. Unfortunately, the render files cannot be modified beyond simply drawing them. The only solution was to then create multiple render files for the various colors that I wanted to have. The problem is that the way RenderGame.cs is set up doesn’t allow for multiple renders to be loaded. Maybe this is a limitation of the type of render they used, but I doubt it. In the end I am just not familiar enough with Xbox coding to get this done.

The other big thing was to play a particular sound effect when lightsabers connected. However, tracking this was also more difficult than initially anticipated. The joints are easy to access, but the lightsabers are rendered as cubes and it isn’t as easy to determine when they connect. In addition, with both players dualwielding, it can get out of hand quickly.

The last thing I tried was to recognize a gesture to turn the lightsabers off. Originally I had a gesture that was called DiagonalDownAndLeft, but this gesture was used often accidentally
when simply swinging the lightsaber around. I was unable to find an appropriate gesture to work with. The only viable option was voice activation, but I ran out of time to implement this.