Real Time Avateering with Skeleton and Face Tracking

Prof. Matthias Zwicker Eashaan Kumar

Department of Computer Science
Rm. 4471 A.V. Williams Building,
University of Maryland-College Park,
MD 20742

August, 2017

A. Abstract

This research project aims to create a system for remote presence for users in the form of a 3D virtual avatar. It allows them to establish eye contact with their avatar. Through the use of Kinect SDK and facial landmark detection techniques, the system does full body tracking and facial expression tracking. For a more immersive experience, it makes use of virtual reality through Oculus Rift. The project is implemented in a popular game engine called Unity3D that renders the avatar and talks to the Kinect sensor. Skeleton tracking and avatar mesh deformation is done directly by the Kinect. Facial landmark detection is done by an open source library called the Deformable Shape Tracking library. Blend weights are used to alter the avatar's facial expressions and successfully mirror the user.

Table of Contents

I. Introduction

Computer Vision

Computer Vision opens up various possibilities for computer scientists. It allows them to give computers better capability to extract, analyze and understand useful information from a sequence of images. Whether it is identifying shapes in images or recognizing faces in video streams, computer vision makes any form of image processing faster and more interesting. OpenCV is one of the most well known open-source libraries with great vision capabilities, which is available in various languages including C++, Java and Python.

Virtual Reality

Virtual Reality is the computer-generated simulation of a three-dimensional image or environment that can be interacted with in a seemingly real or physical way. The user wears a set of virtual reality goggles that render the scene and change the virtual world's camera's orientation based on the orientation of the user's head. Often times the users interact with the virtual world with hand controllers and sends their relative coordinates in space to the computer. For example, if the user bends down while wearing a VR headset and holding a controller, the camera in the virtual world will rotate to face downwards and translate some unit distance realistically. The user might extend the hand outwards, causing the virtual hand in the scene to extend outwards as well. Many virtual reality hand controllers have buttons and triggers that also allow users to grab, release, point and perform other gestures in the virtual world.

Game Engines

A game engine is a piece of software used for developing computer and video games. They are usually free to download. A game engine usually comes with built-in features such as physics and sound engines and a rendering system. With the rise of 3D games, many game engines like Unity3D and Unreal also include animation support that allow programmers to import animations from modeling software such as Blender. Games engines also allow easy access to different developing environments such as PC, mobile and VR. For this research, we are interested in VR development. Hence game engines are very useful to render 3D meshes in a VR environment.

II. Objective

We wanted to use the benefits of computer vision to accomplish our research project titled “Ultimate Vision.” In its simplest form, the title takes the form of just “Vision”; the “Ultimate” was applied after pre-established research goals were met and additional work was done. In other words, our research project is modular, with a basic part to start with and then two extensions that are more challenging and interesting from a research point of view.

In Summer 2017, Professor Zwicker in Department of Computer Science and I attempted to create a system for remote presence for users. Ultimate Vision will allow users to establish eye contact in the virtual world. This system also includes full body tracking and simple facial-landmark detection for facial expression tracking. Users will be able to see each other’s hand, leg, finger, neck, and torso movements and also see facial expressions. The “mirroring” effect is implemented on Oculus Rift VR headsets.

III. Previous Solutions

There are many implementations of face tracking and skeleton tracking available online. Before beginning the implementation, we researched various phD papers by students and professors around online and evaluated their implementations and their feasibility.

We came across a solution by students at Texas A&M University who implemented “the first realtime 3D eye gaze capture method that simultaneously captures the coordinated movement of 3D eye gaze, head poses and facial expression deformation using a single RGB camera” (“Realtime 3D Eye Gaze” 1). At first, this seemed like the perfect solution for us since it detected landmarks on the face and generated an accurate face mesh. However, it used an RGB camera instead of the Kinect which meant the RGB camera may or may not have the same depth of field view as the Kinect RGB camera or a worse resolution.

Another interesting paper published by a Stanford student used landmark detection to perform facial overlays of the user on a target. At run time, their implementation tracked “facial expressions of both source and target video using dense photometric consistency measure” (“Face2Face” 1). Then, reenactment was achieved by “fast and efficient deformation transfer between source and target.” This seemed like a potential idea we could implement in our research. However, it was a bit complicated and we were looking for something simpler and easy to integrate into Unity. Also, it did not work in VR.

A preliminary version of the paper by the same authors caught our attention because it worked with VR. Called FaceVR, it is a “method for gaze-aware facial reenactment in the Virtual Reality (VR) context” (Thies, Justus et al. 1). Their algorithm performs real-time facial motion capture of user and eye tracking from monocular videos. It also renders photorealistic rendering of the face and eye. This seemed like the perfect match for our research specifications. However, the VR headset they used was augmented. They had placed small cameras inside for eye tracking. We chose to avoid altering any equipment in our lab.

An approach by students at Carnegie Mellon relied heavily on geometry to calculate how open a specific facial part is. For example, a mouth can be approximated by a line when “fully closed”, an ellipse with a line in the middle when “relatively closed”, and an ellipse with no line when “open” (Tian, Ying-li, et al 10). Although we didn’t use their mathematical techniques for our blend weight calculation, it served as a great inspiration for the algorithm I have developed. To see how the algorithm works, look at section “Blend Weight Mathematics.”

IV. Underlying Technologies

Skeleton Tracking

The Kinect skeleton tracking system can be broken down into two stages: first computing a depth map and second inferring body position.

First, the depth map is constructed by the Kinect's **time-of-flight** camera. This camera "emits light signals and then measures how long it takes them to return" (Meisner 1). It is accurate to the speed of light: 1/10,000,000,000 of a second. Thus, the camera is able to differentiate between light reflected from objects in the surrounding environment.

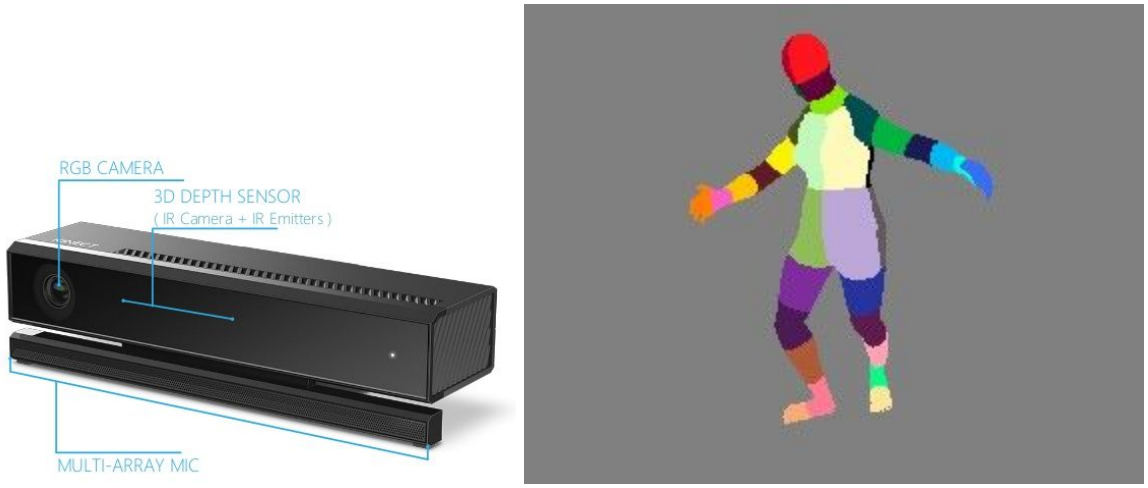

Figure 1a (left): Kinect's various sensors. Figure 1b (right): Distributions color coded by classifier.

Upon receiving the depth sensor information, the Kinect uses an object recognition approach that simplifies the complexity of body-part detection into a "simpler per-pixel classification problem" (Shotton, Jamie et al. 1). From a single input depth image, a per-pixel body part distribution is calculated as color-coded as shown above. Each distribution is labeled either as skeleton joint of interest or predictors for other joints. These form pairs of distributions (what the Kinect thinks is a body-part of interest) and the inferred body part. They are used as labeled data for classifiers that are trained on large database of such pairs.

Randomized decision forests, which are binary trees that take in a distribution and output the probability of the classification, are used to process each pixel at the rate of 5 arithmetic operations per 3 image pixels. Moreover, "each tree is trained on a different set of randomly synthesized images" (Shotton, Jamie et al. 4).

The final step is to use this per-pixel information to generate reliable proposals for 3D positions of skeleton joints. Using equation 7 (Shotton, Jamie et al. 1) various parameters such as number of pixels, pixel weight and pixel depth are used to calculate the x,y and z coordinates.

Facial Feature Tracking

DEST, a facial landmark tracking library used in this research depends on OpenCV's CascadeClassifier class. The CascadeClassifier class relies on the Haar Feature-based Cascades to perform facial detection operations. Below is the explanation of **Viola-Jones** face detection that is used for facial landmark tracking.

Computer scientists Paul Viola and Michael Jones published an algorithm using Haar feature-based classifiers in their 2001 paper "Rapid Object Detection using a Boosted Cascade of Simple Features." According to *Face Detection using Haar Cascades*[1] tutorial page provided by opencv.org, this involves a machine learning based approach "where a cascade function is trained from a lot of positive and negative images. It is then used to detect object in other images."

Positive images are images of faces and **Negative images** are images without faces. For face detection, the algorithm needs a lot of positive and negative images to train the classifier. After training, haar features are used to extract features. "Each feature is a single value obtained by subtracting the sum of pixels under white rectangle from sum of pixels under black rectangle."

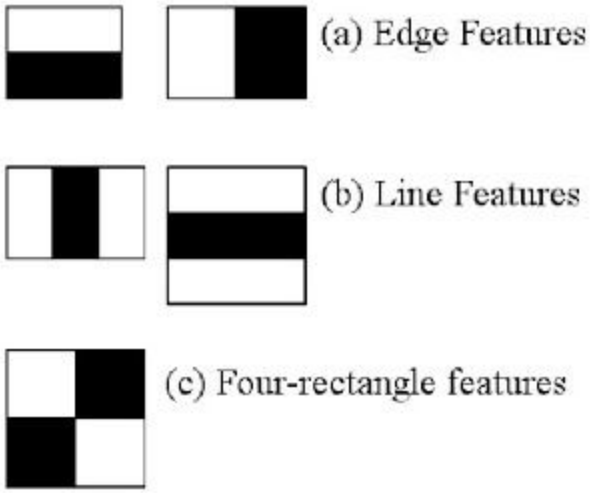

Figure 2: Example of Haar features used on images to find light/dark regions of interest on the face.

But sometimes the features that are calculated are irrelevant. For example, features used to detect eyes are darker on top and lighter on bottom to mimic how light interacts with the eye-region of the face. This would be useless if applied to cheeks or any other place. To select the best features out of hundreds of thousands of combinations, a technique called **Adaboost** is used.

[1] http://docs.opencv.org/trunk/d7/d8b/tutorial_py_face_detection.html

For this method, every feature is trained on positive and negative images and only the ones with minimum error rate are chosen for face feature tracking. But applying this new set of features over a 24x24 image is still very inefficient. For this, the concept of **Cascade of Classifiers** is used to group the features into different stages and then they are applied one-by-one. It is much faster since if a given window fails at a particular stage, all other stages can be discarded and hence avoid using the remaining features. A windows that passes all stages is a face region. This is extended by the DEST library to detect individual landmarks on the face.

Blendshapes

Blendshapes are simple linear model of facial expressions used for realistic facial animation. They are prevalent in Hollywood films and commercial animation software. Although they originated in industry, they became a subject of academic research relatively recently. It is important to examine how blendshapes work since they are heavily used to animate the avatar's face in this research.

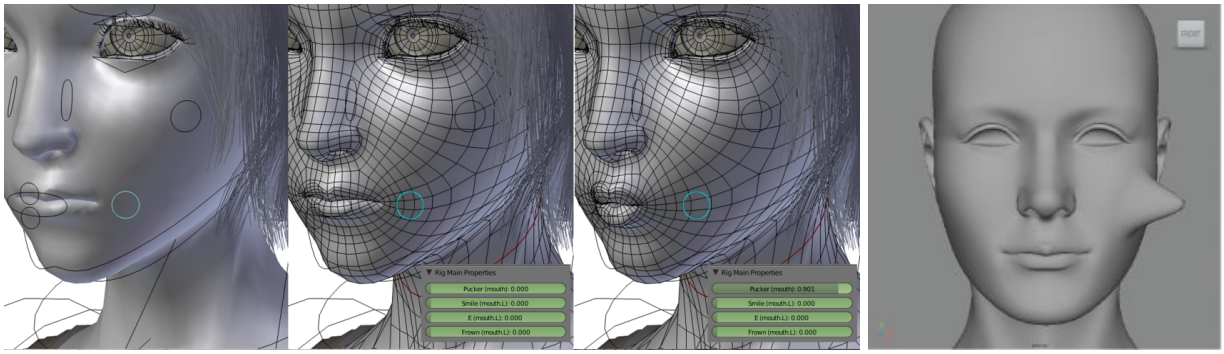

Figure 3a (left): Blendshapes used on a character mesh. Figure 3b (right): Example of arbitrary deformations that blendshapes *prevent*.

A blendshape model "generates a facial pose as a linear combination of a number of facial expressions" (Lewis, et al. 1). These expressions are known as **targets.** Then, the weights of the linear combination are varied within defined minimum and maximum values. Hence, they form a domain of possible facial states that the model mesh can take, the minimum being the initial state and the maximum being the target state (figure 3a).

This is advantageous for the animators since these weights have intuitive meaning as the strength or influence of various facial expressions. Also, it allows animators to stay on target since arbitrary deformations are not possible (figure 3b).

Unity3D

Overview

Unity3D is a cross-platform game engine developed by Unity Technologies that comes with a built in rendering engine, physics engine, sound engine and animation engine ("Unity -

Manual”). It is primarily used to develop video games, simulations for various gaming consoles, computers, and mobile platforms. It supports drag and drop functionality, scripting through the C# language and 2D and 3D graphics.

GameObjects

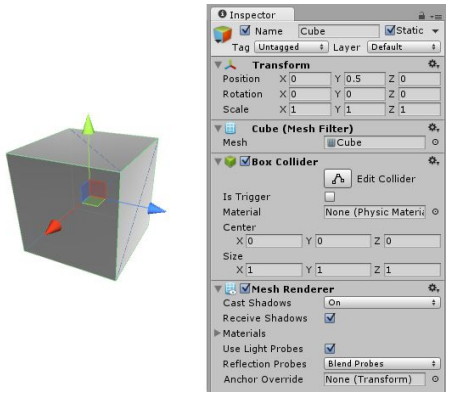

Figure 4: Different Components attached to a GameObject in Unity.

A GameObject is a C# class in Unity that describes any object that exists in the game world. Each GameObject has a Transform object that contains its position, rotation and scale data. GameObjects can be arranged in hierarchical structure with a parent-child relationship. For example, if a square was placed as child under a circle, and the circle was scaled by a factor of 2, the square will also be scaled by a factor of 2. For the purposes of this research, the hierarchical structure was used to pair various body parts of the avatar is a manner that allows us to easily rotate and animate the character model’s bones. For example, the hands, legs, and head were children of the body so that if the position of the body was changed, the limbs and head would stay with the body.

Components

Scripting in Unity is done through Components. A Component is a class or object that is attached to each GameObject in the scene. Each Component can be enabled or disabled which will enable or disable its effects on the GameObject accordingly. The scripts that were written for Kinect avateering and face detection were attached to the avatar model as separate components. One advantage of separating code in this manner is cleanliness and ease of use since any change made to one script will not affect the other.

Each Component has its own set of parameters that can be set through the Unity Editor in the Inspector Panel. When writing a script, such as the ones used for this research, public variables can be declared in the program that will appear as fields in the parameters list of that component. This allows the programmer to change and tweak the parameters without hardcoding in a value into the program.

Rendering and Unity Blendshapes

Figure 5: Skinned Mesh Renderer attached to our character model

Rendering in Unity3D is done through the “Mesh Renderer” Component. A Mesh Renderer was used to render the image of the user’s face captures by Kinect’s own camera. That image was then sent to the DEST library and OpenCV for facial tracking.

The avatar model contains the “Skinned Mesh Renderer”, which is a Mesh Renderer but also contains blendshapes. The Skinned Mesh Renderer renders the following body parts of our avatar model: head, body, hair, eyes, teeth, and tongue. However only the blendshapes on the head are used to animate the face.

Asset Store

The Asset Store is a global marketplace where developers can place their own implementations of projects and Unity modules for sale. Some products are free to download and free to use, while others are paid and some with particular licenses. Many types of products can be found there, including textures, images, UI, sound effects and even fully implemented games. We used the Asset Store to download code that made use of Kinect Windows API to seamlessly apply the correct transformations to the animation bones of the avatar.

OpenCV

OpenCV stands for Open Source Computer Vision Library which is released under a BSD license. Hence, it is free to use academically and commercially. It comes in many languages including C++, C, Python and Java. OpenCV supports Windows, Linux, Mac OS, iOS and Android. According to opencv.org, the library was “designed for computational efficiency and with a strong focus on real-time applications.” Because it is implemented in C/C++, it can take advantage of multi-core processing. Around the world it has been downloaded 14 million times and used in applications such as interactive art, mines inspection, stitching maps and advanced robotics.

There are two popular versions: OpenCV2 and OpenCV3. We used OpenCV3 to communicate with Unity to ensure the latest updates and patches. The library came as a zip file which was extracted and compiled using Cmake (“Cmake Useful Variables”). The generated binaries are required by the DEST library.

D.E.S.T.

Figure 6: Examples of DEST face landmark detection in action.

Overview

Deformable Shape Tracking (DEST) is a C++ library that provides high performance 2D shape tracking leveraging machine learning methods (Heindl, Christoph 1). It is built on top of OpenCV and uses the Eigen library to perform tracking and classifying operations. DEST features include:

- Framework for arbitrary shape transformations
- Efficient landmark alignment algorithm
- Pre-trained trackers for a quick start

DEST comes with a trained classifier and tracker included. According to the github repository owners, the pre-trained face trackers have been trained on a dataset of 3200 images.

Functionality

The release version is downloaded as a zip file and compiled and built using Cmake. DEST includes a wrapper for OpenCV based face detection in face_detector.cpp. The tracker and classifier must be loaded using Tracker and Classifier classes. To detect a single face, the **detectSingleFace(image, rect)** method can be called. Here, *image* is an instance of the Image class and *rect* of Rect class provided by this library. Image is the library’s version of cv::Mat and is used for calculations performed on images by this library. Finally, the tracker must be invoked by calling Tracker’s **predict()** method with returns a Shape object. DEST’s Shape class contains the landmark locations in columns (x,y) for the given image. The number of landmarks depends on the data used during training.

Command Line Tools

DEST also provides command line tools to test databases of images and trackers. Command line tools such as **dest_align** and **dest_track_video** can be used to run the tracking algorithm on an image or video. Commands such as **dest_train** and **dest_evaluate** allow us to train trackers on our own database and then evaluate the effectiveness and accuracy of that tracker.

V. Implementation

Ultimate Vision was implemented in three separate phases: installation, skeleton tracking, and extensions. The project was split in this manner because each phase required extensive research and time.

Installation

In the beginning, I attempted to setup the Kinect on Mac OS since that is my preferred operating system. To do so, I first had to install OpenNI for Mac OS to be able to communicate with the Kinect. But after running into various compatibility issues and finding no solutions on the Internet, I learned that Mac El Capitan requires NITE (another supporting library) and does not support Windows SDK for Kinect. As discussed earlier, the Kinect SDK was necessary for Unity to communicate with the Kinect. Furthermore, the SDK offered skeleton tracking code that was not available in C# in Unity.

Staying with Mac OS, I searched for possible solutions or a work around directly through Unity on the Asset Store. As mentioned earlier, the Asset Store contains many products and wrappers for problems that other programmers have encountered. An asset called *Dlib FaceLandmark Detector*[2] by the company Enox Software was available for costed $40. Another asset that came to my attention was *FaceTacker Example*[3] which was a free asset also provided by Enox Software. However, it required the purchase of *OpenCV for Unity*[4] asset which costs $95.

Unable to find a free solution that could get the Kinect to work in Unity on Mac OS, I decided to switch to Windows 10 and use *Kinect for Windows SDK*[5]. Windows was installed on my Macbook via Bootcamp (“How to Install Windows” 1) since there was no other computer available in the research lab.

[2] https://www.assetstore.unity3d.com/en/#!/content/64314
[3] https://www.assetstore.unity3d.com/en/#!/content/35284
[4] https://www.assetstore.unity3d.com/en/#!/content/21088
[5] https://developer.microsoft.com/en-us/windows/kinect

Skeleton Tracking

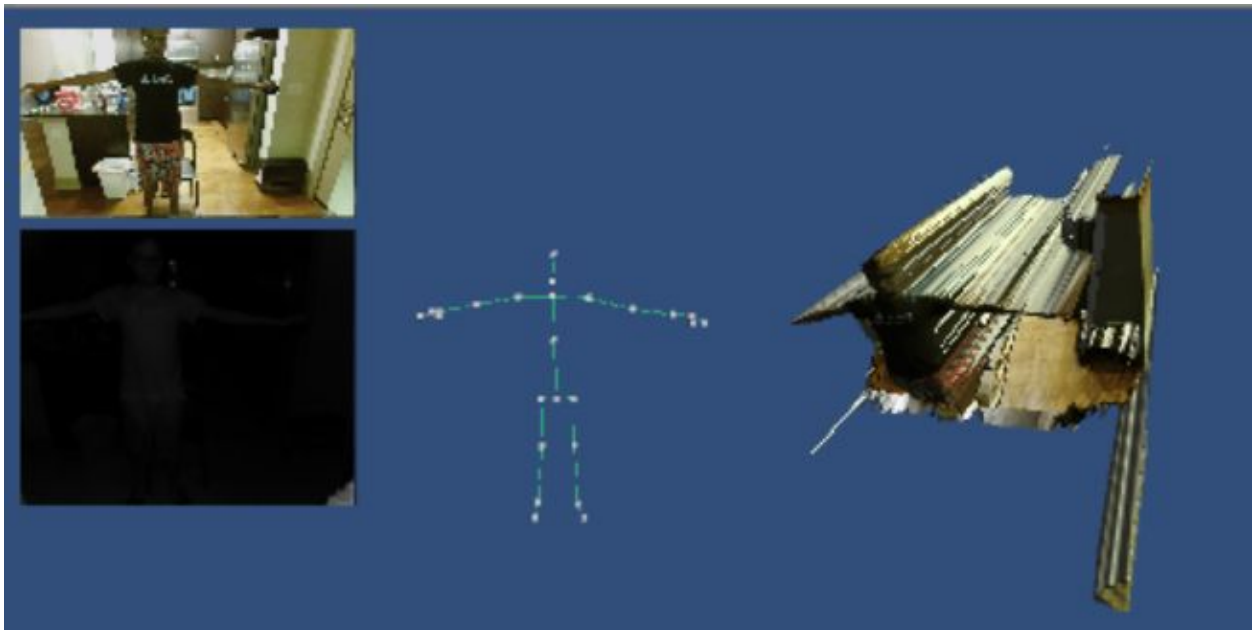

Figure 7: A screenshot of the Kinect SDK's sample Unity project. Upper left: my body in RGB. Lower left: my body through a depth filter. In the middle: Kinect's skeleton tracking; green lines are bones; white cubes are joints. On the right: Kinect's 3D point of view.

The SDK came with a sample Unity project that provided basic skeleton tracking code. However, it did not provide code to apply the skeleton tracking data to a 3D character model. I wrote a script in Unity that could apply the correct movements from the skeleton data (shown in green in Figure 7) to a 3D character model I downloaded from the Asset Store. This character model came with the asset *Male Character Pack*[6] which is now deprecated.

To do this, every joint in the character was mapped to every joint in the green skeleton. Then, the position and rotation of the character joints were updated and tested. The result was unexpected:

[6] https://www.assetstore.unity3d.com/en/#!/content/124

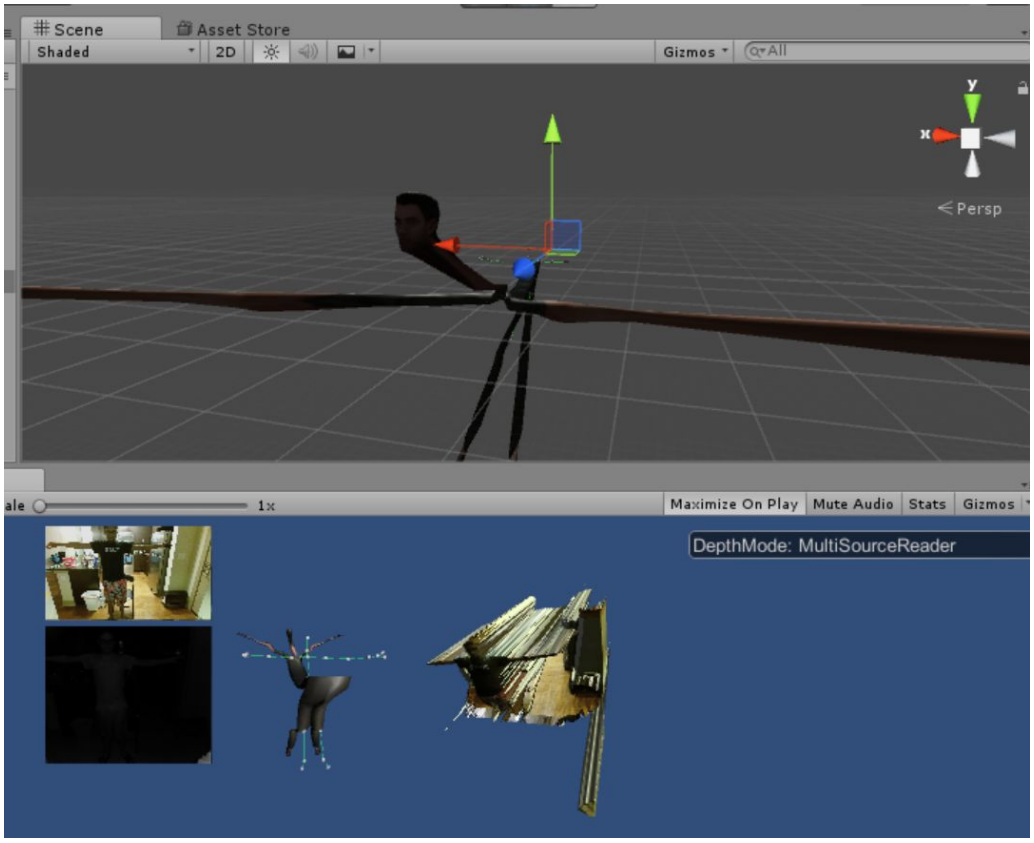

Figure 8: Character distortion due to my methodology of setting bone positions directly. Top: character from the front with clear limb distortions. Bottom: character from the back with a clear view of the torso lagging behind the rest of the body.

Obviously, my strategy did not work; manually setting the position and rotation of each joint distorts the character mesh in unexpected ways. The distortion can be seen by the stretching of the arms, legs and neck above. Due to the distortion, the texture also has been distorted. Also, some body parts, like the hips, seem to be way behind of the rest of the body. This is due to the incongruence between Unity character bone structures and Kinect's skeleton bone structure; there are bones in Unity's character models that do not exist in Kinect's skeleton tracking. One solution to this problem may have been to use a lerp function to estimate the hip position based on the positions of other body parts. But even so, the distortion of the other body parts would remain an issue.

The solution for this problem was found in the asset called *Kinect v2 Examples with MS-SDK*[7]. As the name suggests, this asset comes with scripts and demos that demonstrate the capabilities of Windows Kinect SDK. It comes with 12 demos with all the code available to use: AvatarsDemo, BackgroundRemovalDemo, ColliderDemo, FaceTrackingDemo, FittingRoomDemo, GesturesDemo, InteractionDemo, MovieSequenceDemo, MultiSceneDemo, OverlayDemo, PhysicsDemo.

[7] https://www.assetstore.unity3d.com/en/#!/content/18708

The AvatarsDemo contained a script called AvatarController that correctly overlaid the Kinect skeleton data onto any character model of our choice. It solved the problem by keeping track of every bone *and muscle* to estimate the rotation and position of each body part. My initial method of editing the bones directly was only half correct; the mathematics behind muscle movements involves keeping track of every muscle movement as angles in an array. In the **Update()** method, the **CheckMuscleLimits()** method clamps the bones within the designated range of angles to avoid the contortions seen in my attempts. Seeing the success that this script achieved, I decided to keep it and use it on my own character model. Hence, the skeleton tracking on an avatar is completed.

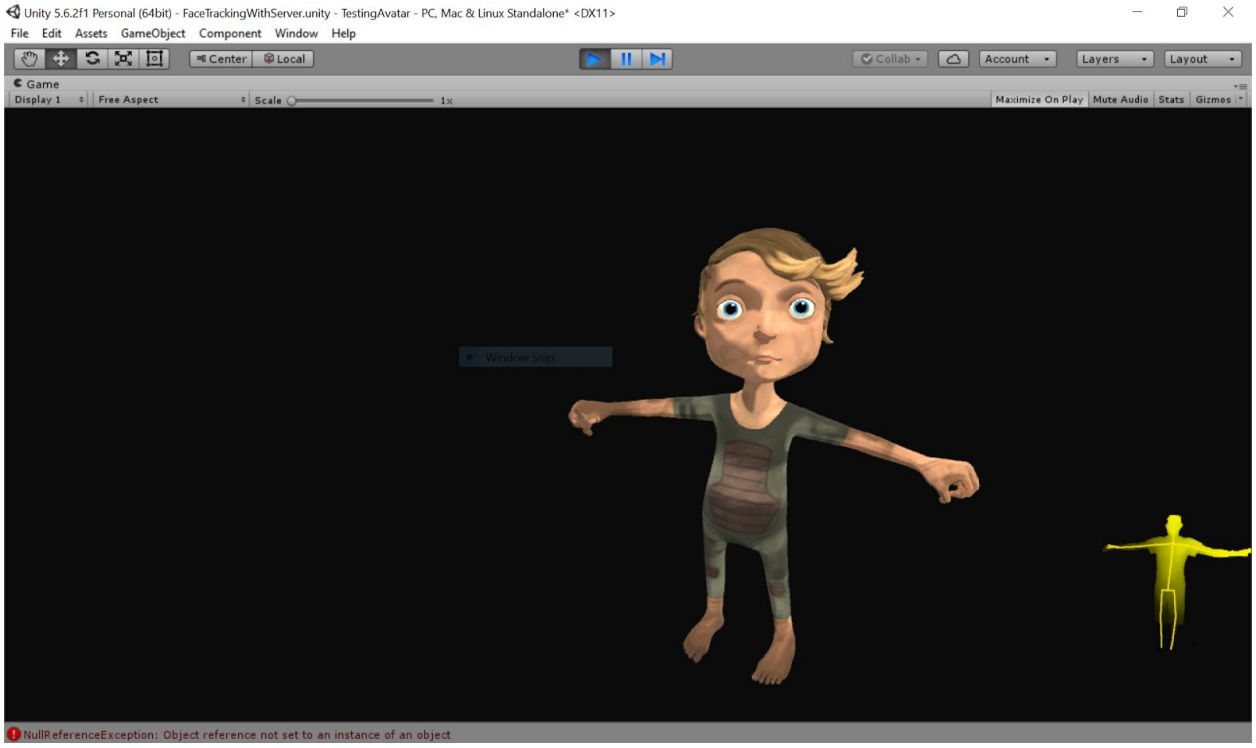

Figure 9: Skeleton detection working correctly.

Extensions

Now that the base requirement of the project was done, I began implementing facial tracking features. Kinect SDK's Unity sample project included a script called FacetrackingManager that performed basic facial landmark detection for eyes, eyebrows, and mouth. This Kinect face detector returned a float value from -1 to 1 which represents the percent of which a facial feature is open or closed. For example, To raise the upper lip:

```
float fAU0 = manager.GetAnimUnit(KinectInterop.FaceShapeAnimations.LipPucker);
```

fAU0 will now hold a value that represents how much to the raise the upper lip to simulate a lip pucker. The "manager" refers to an instance of FacetrackingManager. **KinectInterop** is a class provided by Kinect SDK that contains constants and other utility functions.

“AvatarFaceController” Script

After obtaining the percentage value, I used Unity’s SkinnedMeshRenderer to access the avatar’s facial blendshapes. For the particular avatar that is being used, values in the range from 0 - 200 blend the face mesh a reasonable amount as seen below.

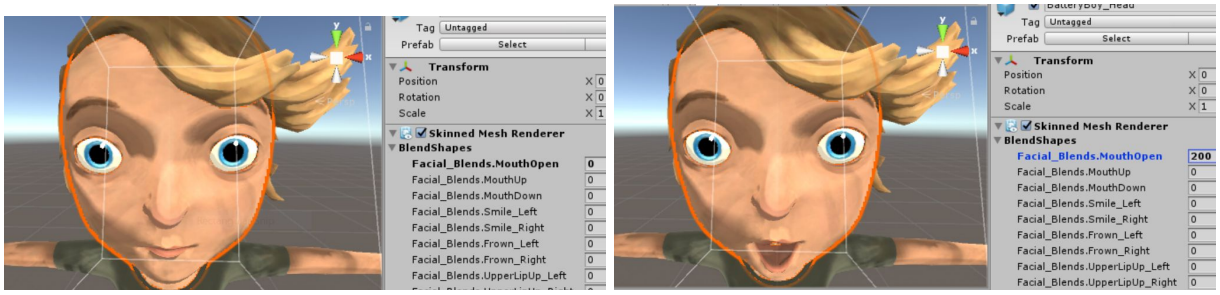

Figure 10a (left), 10b (right): Examples of blend shapes used to alter facial expressions on our character model. Here the mouth is being opened by entering a value between 0 (closed) and 200 (open).

Next, the blend weight can be set by **SetBlendShapeWeight()** method of the SkinMeshRenderer. To manage this system, I created a script called AvatarFaceController that holds a reference to this SkinMeshRenderer and sets the blend weights accordingly. The result can be seen in Figure 11.

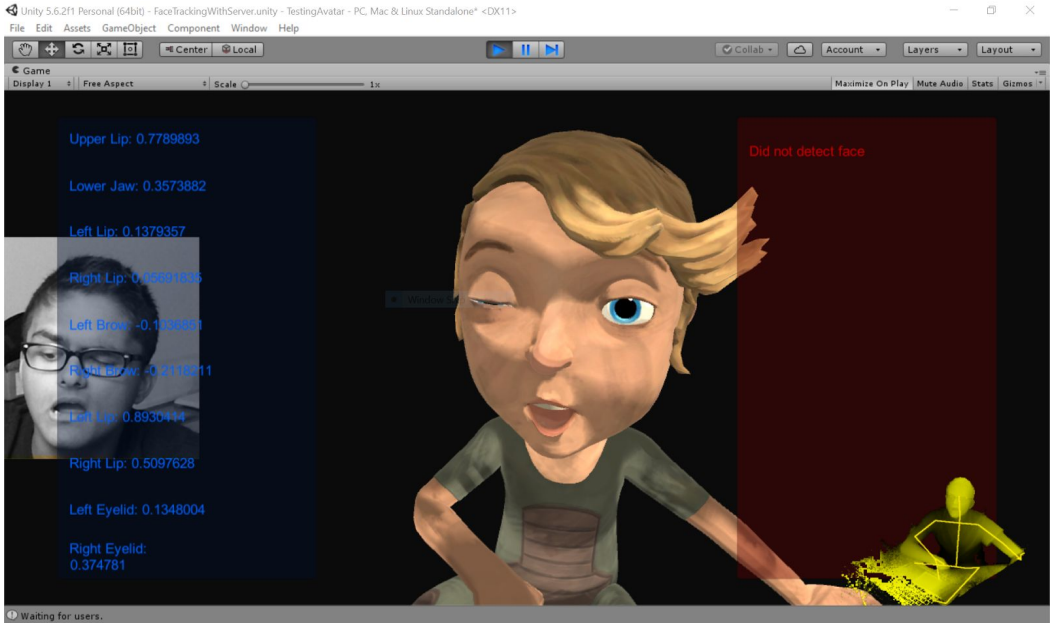

Figure 11: Rudimentary facial tracking provided by Kinect SDK. Blue box on the left shows facial expression values used for blend shapes calculations. Grayscale image on the left depicts my expression being mirrored by the avatar in the middle. Ignore red box on the right

However, the face tracking at this point was inconsistent and caused random jitters of the mouth and eyes to blink when my head was turned at an angle. This might have been due to poor

image processing capabilities provided by the Kinect SDK. Furthermore, the Kinect failed to track any facial features if my hand blocked my face from the Kinect's view. Since our plan also included the additional step of integrating VR, it is obvious that the big headset on the user's face would break this face tracking system.

We needed a more universal system that provided smooth facial landmark positions and was capable of detecting facial features when an obstacle obstructed the Kinect's view. After testing the tracking system with obstacles in front of my face, I reached the conclusion that the Kinect was able to track my face when it was partially or even completely obstructed but unable to detect my facial features. If the Kinect couldn't keep track of my face, this image would appear as black.

Figure 12: Same grayscale image that was present in Figure 11 on the left. This is evidence that Kinect tracks the user's head even if view is obstructed.

The FaceTrackingDemo included a script called **SetFaceTexture** that communicates with Kinect SDK's FacetrackingManager to obtain an image of the user's face. It renders the image onto a plane, as seen in Figure 11. To use that image for face landmark detection, I created a script called **DestManager**. DestManager obtains the image from the plane every frame and processes it grayscale image in a coroutine called **UpdateDest()**. After the grayscale image has been created, that image is sent off to DEST for processing. DEST requires the incoming image to be a floating-point array with grayscale values from 0 - 1.

DEST Methods

The most important aspect of this process is the way in which Unity and DEST communicate. DEST is a C++ library which means that it cannot be used in Unity directly. For that reason, I created a separate project using DEST and OpenCV in C++ titled **ultimate_vision**. Inside this project, there is a **main.cpp** program that contains the code for doing the image processing and tracking. There are 5 methods: **Init(), DetectFace(), SetLandmarksOfInterest(), GetXPos(), and GetYPos().**

```
Int Init(const char *fullPathCalssifier, const char *fullPathTracker)
```

Init takes in the full paths of the classifier and tracker as arguments. It loads the classifier and tracker and returns an error if either failed to load.

```
int DetectFace(const char *fullPathTempImg, double* a, int rows, int cols)
```

DetectFace() takes in the full path to a temporary image, a double array or pixels and dimensions of the image. The reason for the temporary image will be explained below. The double array is converted to an OpenCV compatible image called Mat with the following line:

```
cv::Mat imgCV = cv::Mat(rows, cols, CV_64F, (uchar*)a);
```

The constant **CV_64F** signifies that the image is 64 bit with an array of floating point numbers for pixels. To convert this image into a readable format for DEST:

```
imgCV.convertTo(imgCV, CV_64F, 255);
```

Here, **255** converts the pixel array from values 0-1 to 0-255. At this point, imgCV appears as a black image if rendered using cv::imshow(). After extensive research, I learned that initializing an image via 1D array does not create the three color channels correctly. To fix this issue, cv::imwrite() is used to save the current image to the temporary path that was provided as the first parameter. Then, it is reloaded from the same path by using cv::imread() and the constant CV_LOAD_IMAGE_GRAYSCALE (“OpenCV” 1). This was an important step since it creates all color channels appropriately and renders correctly as a grayscale image. At this point, if the image was not read or was empty, the method will exit with exit code -3.

```
void SetLandmarksOfInterest(int* points, int count)
```

This method updates the list of landmark indices that will be tracked by Unity. It takes an array of indices and copies each element into a local array. While drawing the landmarks in the window, each landmark that is present in this local array is highlighted red to set it apart from other landmarks.

```
int GetXPos(int landmarkIndex) and int GetYPos(int landmarkIndex)
```

These two methods take a landmark index as parameter and return the landmark’s x and y pixel coordinates. This method is used by Unity to obtain the coordinates of the edges of the mouth and nose to estimate the facial expression.

The next step is doing the face landmark detection for a single face by calling FaceDetector’s **detectSingleFace()** method. If no faces were detected, the method will exit with exit code -4. Assuming a face was detected, a Shape object is created by Tracker’s predict() method and is added onto the image to be rendered. The input image is rendered in the window titled “Dest input” and the landmarks are rendered in the window titled “Landmarks”:

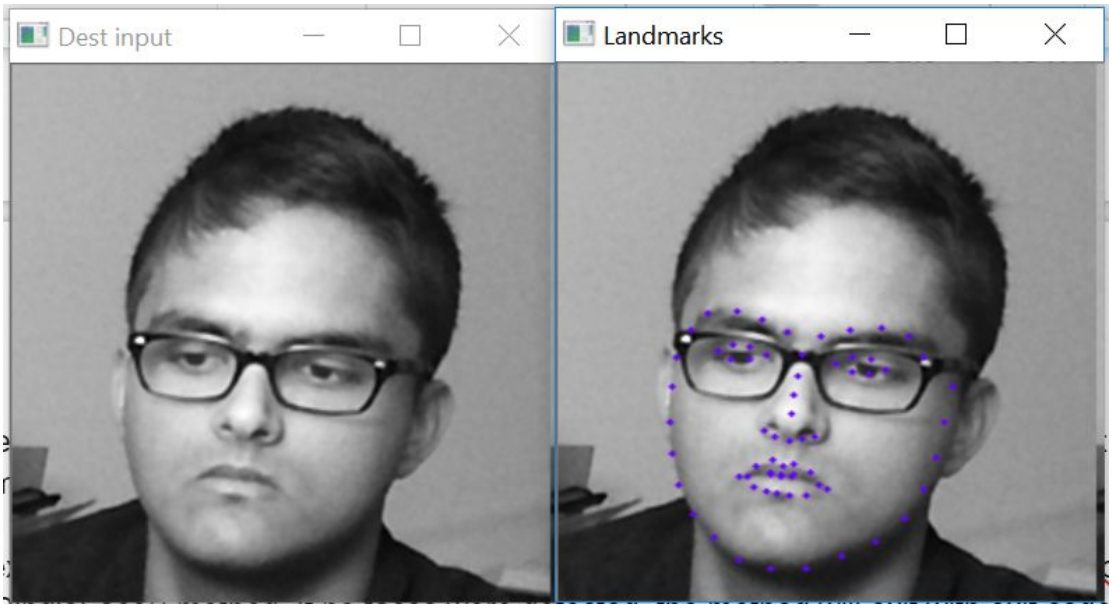

Figure 13a (left), 13b (right): Input image received by DEST and output image after landmark detection.

Finally, main.cpp is exported as a dll and imported into Unity using C#'s System.Runtime.InteropServices. In DestManager, Init() is called in Start() and DetectFace() is called after every frame is converted to grayscale. As mentioned before, this is done in a coroutine since DEST processing can be intensive and too much load on the main Unity thread causes a significant drop in frame rate.

Now that the landmark detection works from Unity, it can be used to relay landmark information back to Unity every frame. To do this, the indices of each landmark (purple dots in Figure 13b) must be known. An easy solution is to render the indices as text on the image itself rather than dots. DEST's drawShape() method does not render text, so I created a wrapper method called drawShapeMain() that takes in the same parameters as drawShape() with the addition of a second color. This color will be used to highlight the landmarks of interest.

The method **drawShapeMain()** calls an auxiliary method **drawShapeAux()** that iterates through every landmark available in the given Shape object and create a string from that index. If the local array, **landmarksOfinterst**, contains that index, that means Unity wants to that landmark to be highlighted. So that text is rendered in red color. Otherwise, it is rendered in purple color.

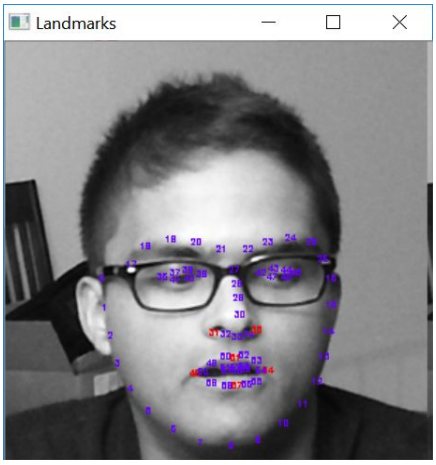

Figure 14: Individual landmarks represented by their array index provided by DEST. Red colored landmarks are tracked by Unity for blend weight calculations.

Unity uses the imported dll method SetLandmarksOfInterest() to pass in an array of 6 integers including:

- 48 = mouth edge left
- 54 = mouth edge right
- 51 = mouth edge up
- 57 = mouth edge down
- 31 = nose edge left
- 35 = nose edge right

The indices for eyes and eyebrows have been intentionally left out since we plan to train our own tracker that will only track the bottom half of the face.

Blend Weight Mathematics

DestManager sends AvatarFaceController the frame information to be used for blend weight calculations. The math behind calculating blend shapes is as follows:

We must first find the aspect ratio of the window. This is necessary so that we can scale distances and positions of landmarks based on the image resolution.

```
aspectRatio = 1 / frame.width
```

Next, we can use main.cpp's GetXPos() and GetYPos() to get their positions in 2D space and then find the distance between them. But different image resolutions will cause the image to scale up or down, causing the landmarks to be separated by varying number of pixels. The raw distance must be scaled by the aspect ratio in order to get an accurate measurement of the distance relative to window size.

```
dis = Vector2.Distance(landmark1Pos, landmark2Pos) * aspectRatio
```

The distance can be used to calculate a percentage value representing how opened or closed a facial expression is that corresponds to the two landmarks. Let's call this value **prcnt**. For prcnt to be 0, the distance between landmarks 1 and 2 must be zero. For prcnt to be 1, the distance between them must be at their maximum. One issue is that two landmarks will never have a distance of 0 since human facial anatomy doesn't let one point on the face cover the other. Hence prcnt will never reach 0. Also, there is no concrete way to tell how large the maximum distance can be which means there is no way to define when prcnt can take on the value of 1. This can be solved by establishing a domain.

A **domain** consists of a range of distances, **[min, max]**, that describe the lowest and highest distance values that have ever occurred between two landmarks since the program was started. Every frame, these min and max values are updated by comparing them to distance:

```
min = Mathf.Min(dis, min)
max = Mathf.Max(dis, max)
```

This means that every frame the domain holds the smallest and the largest possible value distance has taken on so far. Now, we can normalize prcnt between these two values

```
prcnt = (dis - min) / (max - min)
```

Here is a pictorial depiction of what is taking place. Lets imagine that our two landmarks are mouth edge left and mouth edge right.

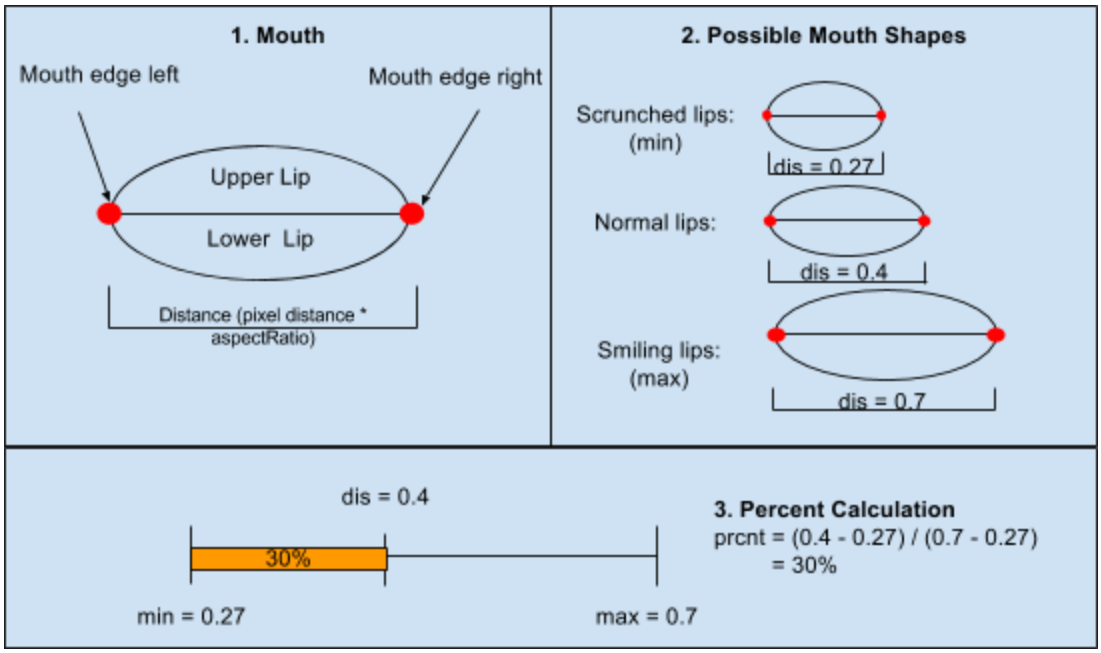

Figure 15a (top left): An elliptical representation of a mouth with upper and lower lip. Red dots represent the two landmarks. Figure 15b (top right): An example of all possible distances between the landmarks as lips come closer together and move farther apart; Inspired by Figure 4 in Carnegie Mellon's paper (Tian, Ying-li, et al 10). Figure 15c (bottom): Percent calculation of how much the user is smiling in this example.

Finally, the blend weight can be calculated by multiplying the percent value by some constant k that represents the largest value a blend weight can take on. The ‘k’ constant will be different for each facial feature on the model.

```
blendweight = prcnt * k
```

With this formula, the avatar model can be made to smile by using mouth edges left and right and made to open its mouth using edges up and down. The two nose edges can be used to scrunch the nose. Due to time constraints, I was only able to implement these facial features. But the landmark data can allow us to create countless more expressions.

VR

Implementing VR is as simple as downloading **Oculus Utility Package**[8] from Oculus Rift’s website and checking the “Virtual Reality Supported” in the Player Settings menu in Unity. The headset must be connected to the computer that Unity is running on.

[8] https://developer.oculus.com/downloads/unity/

VI. Results

Achievements

The end result is a full-body tracking system with VR integration. When users look through the headset, they see an avatar mirroring their movements. The avatar also follows their movement in 3D space. For example, if they move closer to the Kinect, the avatar moves closer to the game camera and vice versa.

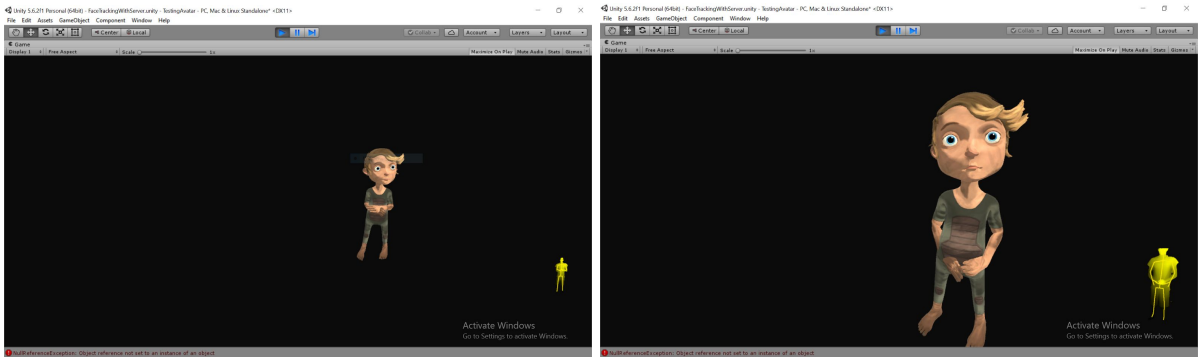

Figure 16a (left), 16b (right): Avatar moving in 3D space based on user's movements.

However, if the users get too close the Kinect is unable to detect some of their body parts such as head, legs or arms. This results in unexpected skeleton behaviour and causes distortions in the corresponding bones in the avatar. If this happens the user must step back from the Kinect so it can pick up their entire body.

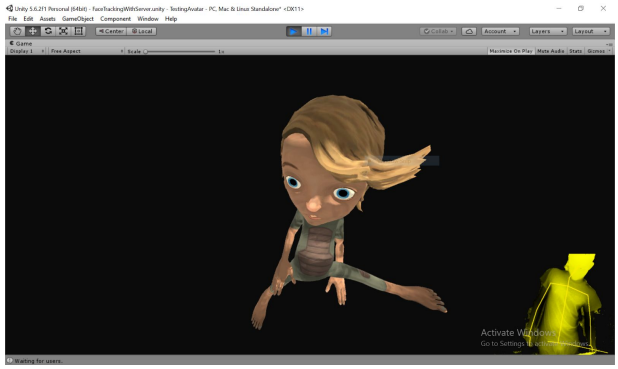

Figure 17: Leg distortion being caused by me standing too close to the Kinet. The yellow skeleton on the right shows my legs being obstructed.

Currently, the face tracking does not work with VR headsets because I was unable to get to the face-tracker training part of this research. The user must take off the headset to make the avatar mirror their facial movements.

Improvements

To improve this system, it is necessary to use DEST's command line tools to train a tracker on a database of faces with VR headsets. Then, the paths to the tracker and classifier have to be switched in DestManager to the new trained trackers and classifiers. The DEST methods in main.cpp will work the same way and thus the dll does not have to be re-compiled.

A temporary image is being used to store the input image received by DEST and then converted into grayscale when read from file. This is a slow and unnecessary step that could be optimized by sending in a RGB image rather than a single grayscale channel. Further research must be done into other image formats supported by DEST. If a suitable format is found, then Unity can send each frame as an array of integers where each integer contains RGB data. Upon receiving the data, DEST can initialize an image with the three channels and then convert it to grayscale to be used for processing. This will allow faster runtimes for the image detection and help maintain steady frame rates.

The approach for using blend shapes must also be considered. Blend shapes are suitable for animation purposes. But unlike animation, landmark detection provides exact positions of interest on a face. It may be treated as discrete vertex positions of the face of the avatar. A better approach might be to directly alter the vertices of the face model mesh based on individual landmarks. This will allow greater flexibility as there are more points on the face and hence more emotions can be created.

Based on Carnegie Mellon's paper (Tian, Ying-li, et al 10), a better way to calculated mouth shapes must be implemented. Rather than only tracking the edges, all landmarks on the lips could be used to generate an outline of the mouth. If the outline is just a line, that would mean the lips are tightly closed. If the outline forms an ellipse and there is dark line or contour inside the ellipse, then the lips are relatively closed. If there are two ellipses, one inside of the other, then the lips are open. The percent value for openness can be calculated by multiplying the area of the embedded ellipse by a constant to use as blendweight. This will generate more accurate mouth expressions in the avatar since all landmarks around the mouth are being taken into consideration.

VII. Conclusion

This project can be greatly improved. Throughout the development process, Prof. Zwicker and I faced multiple challenges including not having enough computers to work on and getting Kinect to work with Unity. I personally spent a significant amount of time researching and learning Cmake to compile OpenCV and DEST on Windows. The lack of a windows machine forced us to install Windows via Bootcamp which came with its own set of challenges.

It was worth pushing through those challenges to bring together this system that could one day prove beneficial to the computer vision industry. The applications of live skeleton tracking include the entertainment industry. Currently, movie makers and video game designers use complicated motion-capture systems that require a large room filled with cameras and a suit studded with sensors for the wearer. With the help of few Kinect sensors and face landmark detectors like DEST, the entertainment industry can significantly simplify motion capture by tracking the actors' skeletons and applying them to a 3D model.

Another application of this technology is a multiplayer-mirroring system where two people can establish eye contact in a virtual world. Long distance video calling will feel like in-person chatting as users are able to experience the other person's facial expressions and gestures. It will push the boundaries of what virtual reality headsets can do by adding a seemingly real element to the user experience.

Before that could happen, many Ultimate Vision needs to be improved. The avatar facial expressions need to be more realistic and its body movements less jittery. Better trackers must be trained on a diverse database of images so that facial landmark detection is more accurate and works when the face is obstructed from view. A better 3D avatar model is needed with more blend shapes so that more facial expressions can be formed. Ultimate Vision, when fully implemented along with networking multiple users, will drastically change how we interact with other online.

VIII. References

"CMake Useful Variables - KitwarePublic." N.p., n.d. Web. 24 Aug. 2017.

"Face2Face: Real-Time Face Capture and Reenactment of RGB Videos." N.p., n.d. Web. 24 Aug. 2017.

Heindl, Christoph. *Dest: :Panda_face: One Millisecond Deformable Shape Tracking Library (DEST)*. N.p., 2017. *GitHub*. Web. 24 Aug. 2017.

"How to Install Windows on Your Mac with Boot Camp." *Apple Support*. N.p., n.d. Web. 24 Aug. 2017.

Jeffrey Meisner. "Collaboration, Expertise Produce Enhanced Sensing in Xbox One." *The Official Microsoft Blog*. N.p., n.d. Web. 27 Aug. 2017.

Lewis, J.P., Ken Anjyo, Taehyun Rhee, Mengjie Zhang, Fred Pighin, and Zhigang Deng.*Practice and Theory of Blendshape Facial Models*. Eurographics, n.d. Web.

"OpenCV: Image File Reading and Writing." N.p., n.d. Web. 24 Aug. 2017.

"Realtime 3D Eye Gaze Animation Using a Single RGB Camera." N.p., n.d. Web. 24 Aug. 2017.

Shotton, Jamie et al. "Real-Time Human Pose Recognition in Parts from a Single Depth Image." *Microsoft Research* (2011): n. pag. *www.microsoft.com*. Web. 27 Aug. 2017.

Thies, Justus et al. "FaceVR: Real-Time Facial Reenactment and Eye Gaze Control in Virtual Reality." *arXiv:1610.03151 [cs]* (2016): n. pag. *arXiv.org*. Web. 24 Aug. 2017.

Tian, Ying-li, et al. "Multi-State Based Facial Feature Tracking and Detection." *Robotics Institute, Carnegie Mellon University*, Aug. 1999, pp. 1–30.

"Unity - Manual: Unity User Manual (2017.1)." N.p., n.d. Web. 24 Aug. 2017.