Michael McQuade

Zhigang Deng

Computer Graphics

4 May 2021

Virtual Ball Player

As the final project for the COSC 6372 class, the objective was to replicate work presented in the article *Virtual Ball Player* by Jong-In Choi, Shin-Jin Kang, Chang-Hun Kim, and Jung Lee. The idea behind this paper is that it is very difficult work to generate an animation in which a character can juggle or manipulate virtual balls. That is because the balls have to be synchronized with the motion of the character and their movement needs to make sense in terms of how we perceive physics.

Juggling is a skill and not everyone can do it, but this research goes beyond juggling and extends to dribbling basketballs, and juggling soccer balls up and down with a character's feet. Previous to this work, research on motion analysis has not been applied to juggling or manipulating virtual balls, but has been applied to synchronizing music and character motion. That being said, the approaches in this paper are relatively simple to implement and understand after digesting. The greatest challenge I found was interpreting the paper, as it seemed much more difficult at first because of the new terminology.

The first step in order to create this animation was to first have an animation of an actor doing a juggling motion. Ideally, this would be in a format that could be used in some popular 3D modeling or animation software. At first, I looked for suitable material online and couldn't find something suitable. As I pondered how to solve this, my research led on to using the Oculus

Quest's body tracking technology to capture an animation and use that for replicating the research.

My first implementation of this was to use the Microsoft Mixed Reality Toolkit, which is a virtual reality SDK that helps you build and deploy VR applications to the Oculus Quest 2. The SDK integrates with Unity 3D, and gives you virtual hands, which are GameObjects in Unity. It is possible to bring these hands up in the simulator by using some keyboard shortcuts, and then move them around in the editor's play mode. This allows for testing your VR application before deploying it to the actual device. Using this, I was able to attach a transform and a script to those virtual hands, and capture the up and down movement of the hands in what I call the "recording" phase of the project.

This then allows me to move the simulated hands up and down and then have a reference of their movement in the script I attached. However, applying this animation onto a 3D model or character's hands is difficult and creates a relatively choppy animation because of the runtime calculations that have to be done in order to replicate this after each recording phase. In order to resolve this, it would be better if the recording of the hand animation could be stored into a standard animation style format and applied to a humanoid character.

In order to facilitate this, I utilized a tool called Glycon3D, which is a project being developed to capture motion from the Oculus Quest 2. With this, I no longer needed to utilize the MRTK solution that I had started with, and so I scrapped the original work and started to build off of this new base. Using the android developer tools you can sideload applications onto your Oculus Quest 2. In order to get started with Glycon3D, you need to follow this sideloading process. Once you have it installed, Glycon3D will be available on the Oculus under the "unknown sources" section of the installed apps.

Upon opening the application you'll be presented with an environment as seen below.

The character rig that is in the above picture moves along with your body. There are controls that allow you to begin and stop recording, as well tools for changing out the environment. For the purposes of this work, only the recording feature was used as I only need to record the hand motion.

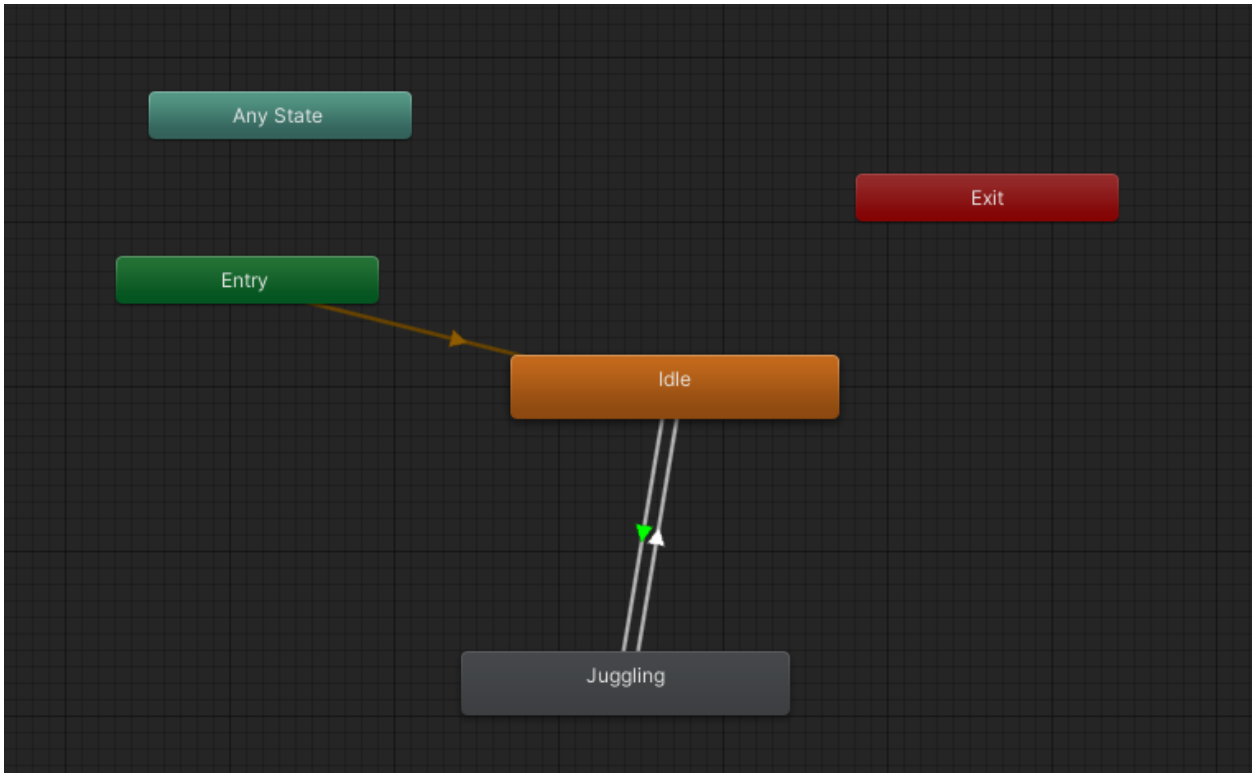
I started recording and made some basic juggling motions using my right hand, and after stopping there was output on the screen that an FBX file was created and stored locally on my Oculus Quest 2. Using the android developer tools to remotely connect to the Oculus Quest 2, I was able to securely copy the generated file back to my computer. The file was able to be opened up in Autodesk Maya 3D, and I was able to trim the animation down to just the parts where my hand was moving using the animation timeline tools in Maya. Afterwards, I could export this FBX file into Unity using the built into Send to Unity feature in Maya.

Now, I needed a 3D model to apply this to, so I found “Ethan” a 3D model provided by Unity in their standard assets pack. This standard asset pack has demo models, animations, and scripts for use in getting started with Unity 3D. The animation wasn’t by default marked as a humanoid animation and thus didn’t work with Ethan. After changing that, and making sure that the animation was for an “A” pose, the animation was able to be applied to Ethan. Now, the animation of me moving my hands that I recorded using my Oculus was playing on the 3D model from the standard asset pack. This was a big step up from the original approach using MRTK because the motion was much smoother.

Now, in the Unity code, it was time to implement the recording for this FBX file. The paper talks about breaking the animation down into “sections” where the section is how long it takes for the hand to move from its maximum value to its minimum value. That is, it can be summarized as an object that has an end point in space, a start point in space, a duration, and a direction. For direction, the data structure only needs to know whether it is an up or down movement of the hand. For the research here, I’ve only focused on reproducing one ball juggling.

Unity has a built in “animation controller” which handles changing between animation states. In order to facilitate the recording phase of the animation, I take advantage of this animation controller. First, the animation starts in an “idle” state where Ethan is simply moving slightly on the spot as is often seen in video games. This animation acts as the entry state in the controller and will be the state that Ethan is in whenever he is not performing any juggling. In order to switch to the animating state, animation controllers have the concept of a “trigger” which can be set or called from within any of the scripts that are a part of the runtime. Triggers can be thought of as a boolean value, except that if they are used as the condition to transition into another animation, then their “true” value is consumed. For example, say you have a trigger

called “Juggling”, if you have a condition on the current animation state to transition to the next state whenever “Juggling” is set, then the animation will transition to that next state, and the “Juggling” trigger will immediately be unset. This is very useful when you might have control of the animation flow before the action but not necessarily after the action.



Above, you can see the animation controller that is used for Ethan. The green arrow indicates that the Juggling animation should be a “solo” animation, that is, no other animations should play at the same time as the Juggling animation.The transition here uses a trigger as described above to transition anytime that trigger is set.

The actual trigger is set in a few places, which is when the “Start Recording” button is pressed, or when the “Start Playback” button is pressed. The trigger is also used to transition back to the idle phase, that way the entire animation process is handle programmatically. The exit transitions happen after the animation has finished one cycle. This is done by checking the normalized animation time property of the animation controller’s current layer, which holds a

double type number, where the integral part of the number is the number of loops the animation has gone through, and the floating point part of the number represents a percentage of the current animation loop that has been completed. Below, the start recording and start playback buttons can be seen.

The recording phase of this process generates a list of “Section” objects, which is the data structure explained above. This list of section objects is used in the playback phase to handle the animation of the ball. None of the ball animation is handled by any animation controller, instead the ball is manually moved to its peak position and end position based on the section data structures. Since the sections are categorized into “up” and “down” sections, an up section tells the ball its starting position by using the “end” position of the right hand. By that token, the down section's “end” position for Ethan’s right hand is the position at which the ball should be at when the hand reaches that point.

In order to facilitate this in the playback phase, linear interpolation of the ball’s position is considered with the duration of each hand movement, or each section. This process of linear

interpolation in animation is called inbetweening or tweening. Inbetweening is the process of generating intermediate animation frames given key frames. Since the maximum and minimum points are characteristically keyframes, this tweening process fits well as a technique to animate the ball here after all of the sections are determined.

Adding multiple balls to the current solution would not be a very difficult matter, as all that needs to change is deciding which section is a given section's "end" section. Right now, the end section is always the down section right after the up section. To implement this with two balls and an even number of sections, one could simply choose the second "down" section after the up section instead of the first.

In order to return back to the idle state, the trigger needs to be activated again. This happens at the end of the playback phase, when the animation controller attached to Ethan has normalized animation time of greater than 1, and every section has been handled. There is a flag set when every section has been played in case Unity continues slightly longer than it should with the animation. In this case, the animation will immediately stop.

In conclusion, animating virtual balls can become a relatively simple task. By combining motion capture of a juggling motion, regardless of if that juggling motion could be considered "skillful" or not, with the section based data structure proposed in the paper *Virtual Ball Player*. This animation technique makes it so a convincing juggling animation can be created even without a skilled actor. By relying on the techniques in this paper, and the motion capture technology readily available using software like Glycon3D and the Oculus Quest 2, even a non-juggler can create juggling animations.

Works Cited

Choi, Jong-In, et al. “Virtual Ball Player.” *The Visual Computer*, Springer Berlin Heidelberg, 5 May 2015, link.springer.com/article/10.1007/s00371-015-1116-9.

“Inbetweening.” *Wikipedia*, Wikimedia Foundation, 13 Mar. 2021, en.wikipedia.org/wiki/Inbetweening.

“Input Simulation Service.” *Mixed Reality Toolkit Documentation*, microsoft.github.io/MixedRealityToolkit-Unity/Documentation/InputSimulation/InputSimulationService.html.

“Instant Motion Capture.” *Glycon3D*, www.glycon3d.com/.

“Interpolation (Computer Graphics).” *Wikipedia*, Wikimedia Foundation, 26 Jan. 2021, en.wikipedia.org/wiki/Interpolation_(computer_graphics).