AS2: Process Letter

Doing the quaternion homework was challenging. It has been a while since we covered the quaternion math, so I had to go back and reread a bunch of the book to understand what I was doing. However, since I'm currently in a robotics class I have had to deal with similar transformations, so it only took an hour or so to pick it up and understand it from reading in the book. I also thought it was convenient that the HW problems were both just rotating about one axis instead of multiple. That simplified the math a lot. It was also cool to remember that since we have all the added rules of matrices in VR, we get to the inverse of a 4x4 matrix by just transposing the rotation part of the matrix and negating the transformation part of the matrix. Except for the identity bit at the bottom.

It honestly took me a minute to understand where the equation was for how to transform from a 3D rotation matrix to a quaternion, but once I found it the math was simple to convert each rotation to a quaternion. The tricky part was doing the multiplication of the two quaternions because following which indices multiplied was confusing. I made a mistake while going through the process and caught the mistake because I knew that the end points of the quaternion all must square and add up to one ($a^2 + b^2 + c^2 + d^2 = 1$). Then I finished this part and moved onto starting the project in unity.

Getting the project setup was easy since I had done it for the first part before, the repetition made it nice. Then I got stuck. Like really stuck on getting the camera to reset. I tried so many different things and the canvas post was quite vague on what to do. I tried to look it up on google as well but I was searching for the wrong thing. I tried to extend the Tracked Pose Driver and was unable to override the methods I wanted to. I think after reading through the code I still couldn't completely understand how it was tracking the movements input, so I gave up on this approach and decided to try moving the Tracked Pose Driver to a parent object of the camera. This kind of worked. It let me reset the camera but then every input command was causing the camera to go the opposite direction from the parent object. Now that I think back on it its probably because this and the parent object were facing opposite directions, however I did not think of that at the time and it was another weird bug, so I decided to google more solutions again. This time I decided to search it up with different keywords and a YouTube video came up that described the process perfectly and it was simple to implement while leaving the Tracked Pose Driver attached to the camera. The difference was that it moved the camera by rotation and translation instead of just setting a new local position and rotation. By causing it to rotate or translate it was able to successfully move the camera back to start and the whole movement system moved with it. Another approach I tried was to implement my own Tracked Pose Driver, but I wanted to use all the code their file gave, unfortunately when I did this part of the code was unfixable because they had restricted access to some of the classes as private.

After this the rest of the assignment moved swiftly. Since I had received a lot of practice with the Tracked Pose Driver, I decided to move onto part 2.2.2 and disable tracking based on that driver. It takes in a parameter called tracking type. By changing this you can enable the driver to only track position or rotation, and then by changing it back it is possible to restart tracking that part again. So the simple implementation was to grab the camera, get the component which was the tracked pose driver and then to change the values of the tracking type. To get both to be disabled at once all that was necessary was to disable the tracked Pose Driver in general so that wasn't too much work either.

Next step was to change the ball sizes according to the retinal image, so they were always the same size. This was some tricky math and the hint in the book gave a good insight into what needed to be done to get the spheres to the right size. To completely understand this math, I turned to google and there was a website that gave an example of this with the numbers shown on the page and a visual example where I could modify how objects would appear in the retinal view. This was very helpful for understanding the math involved and following this it was easy to implement the math and it worked like a charm. It was cool to see when circling the spheres that they always matched up perfectly when making one cover the other.

The next step was slightly frustrating, it didn't take long but was frustrating, however I did learn something as well. When trying to make the blue balls disappear, they always disappear instantly along with the red ball even though I was using the "Time.DeltaTime", however, I had placed the counter inside a while loop so the code was running to fast and adding the same delta time over and over again and adding it all before the update method finished. I had to realize that the update method itself was my while loop and I needed to make conditions upon which the time would pass, how to count it, and when that would cause the blue balls to disappear. Once I realized this it was quick to implement.

Now I went back to position tracking and getting the mirror head to either mirror or match my movements. This process was slightly tedious because I tried to implement it in many ways and each of them had their own bugs to deal with. The first is I tried to track my movement in a similar way to moving the camera back to the origin. This worked for position if I did not rotate the camera, whenever I rotated the camera, it would cause some funky transitions. So, I moved on from this and tried to look it up on the internet. One example in stack overflow used the velocity vectors to move the objects in similar manners. When I tried to implement this, it caused some crazy spinning/circular movement of the mirror head. Eventually I ended up using the mirror as my basis and took the cameras position relative to the mirror and reversed that and gave that position to the mirror head. It worked like a charm and was simple to change the code between the mirror and matching modes because it was just flipping a couple negatives.

Next, I moved onto rotation, I knew that rotating the head would be confusing, but I took a similar approach to what I did above, by first trying to track how the camera moved from one stage to the next and then mimic that movement over to the mirror head. This came with a lot of complexities that arose from changing between methods of viewing. Because of that I decided to take the cameras rotation and just map that to the rotation of the mirror head. Since the mirror head was offset by 90 degrees, I also just needed to add a rotation of 90 degrees to the head at each transition. This worked like a charm and was easily converted to the other form of mirroring or matching by just flipping a negative and changing the 90-degree rotation.

Looking back this assignment taught me a lot, from tracking to sizing and perception. A couple of times while working on the project I started to feel motion sick, which was unpleasant. I think the coolest thing that I learned in this assignment was the retinal image sizing. I thought that math was fascinating in how it caused my brain to hurt when I watched the balls change. I also learned looking back that most of the knowledge I gained was caused by looking it up on google and then implementing it. Overall this was a very fun and informative assignment.