Interpolating My Sanity: A Story of Hope

When this assignment started, I saw that there were a number of conversion methods we needed to develop. Figuring they would be used by other functions later in the program, I got to implementing them first. First, **Euler2Rotation**.

Once I realized that rotation matrices aren't necessarily confined to world space transforms, I knew I just had to form and combine three rotation matrices corresponding to the three euler angles, so that's what I did. The code had methods for multiplying 4x4 matrices in 2D arrays, but I wanted to keep to 3x3 1D arrays, so I wrote a helper function to multiply them. Next was **Euler2Quaternion**.

I had no idea how to convert between the two off the top of my head. I'd seen people on Piazza suggesting it could be done easily by transforming the euler angles into a matrix and then converting that to a quaternion, but I wanted to find a more direct algorithm. There were a number of suggestions online, solutions hard coded to a particular sequence of euler angles with little explanation of how they were formed. I wanted something robust, a general algorithm. So I implemented the one from this scientific paper online https://www.ncbi.nlm.nih.gov/pmc/articles/PMC9648712/. I later had to change this to the easier euler 2 matrix 2 quaternion method, when I realized there were issues and ran short on time.

Quaternion2Rotation was the most straightforward, I just used the quaternion to matrix formula we'd covered in class. **Euler2Quaternion** just made a quaternion for each axis of rotation, then combined them through multiplication. These methods all seemingly worked when I first developed them, and outputted answers consistent with matlab and online converters, but I later had to change them after getting a weird glitch where characters would T-pose constantly, the culprit being that I didn't realize AMC used degrees instead of radians for euler angles. I had to add appropriate conversions once I realized that, and everything worked fine then. I then wrote helper LERP and SLERP methods for vectors and quaternions, respectively, as well as the DeCasteljau evaluation methods.

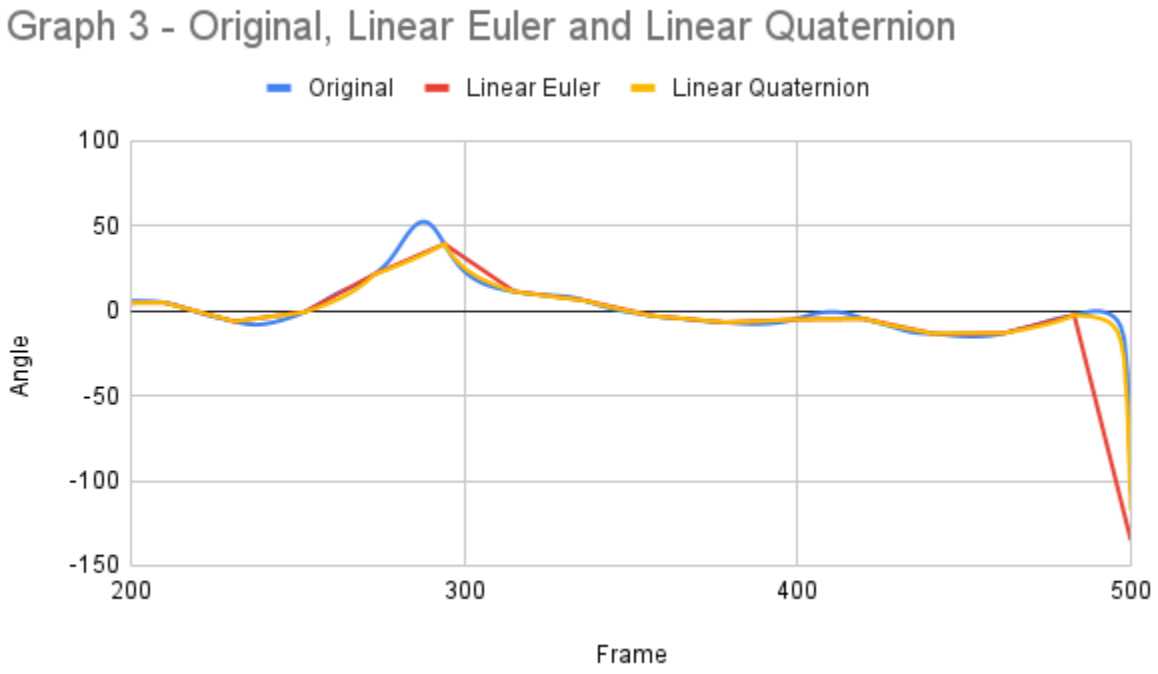
I then developed the remaining three interpolation methods. **BezierInterpolationEuler** was first. When I first began developing it, I didn't fully understand how we were supposed to use bezier curves for euler angles. It took a fair amount of trial and error to figure out how to calculate and keep a and b in memory in relatively efficient ways, especially with how many variables some of the formulas for control points needed, but I got through it. Debugging was assisted with the CompareMotion helper method that I wrote, which at the time, would print both interpolated and original angles / root positions to a csv spreadsheet, which I could then comb through with statistics to detect problems.

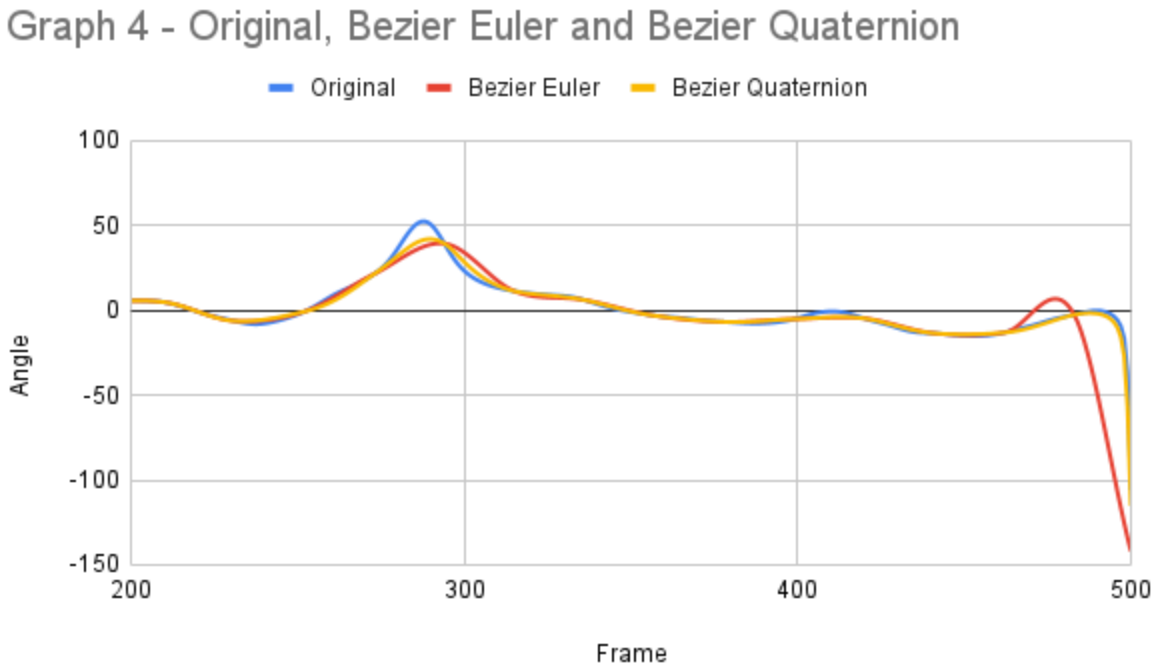
Once that was done I worked on **LinearInterpolationQuaternion**. I thought this one would be trivial to implement, but it ended up exposing both problems with units (degrees vs. radians as mentioned above) and with the SLERP method. I was aware SLERP could yield a suboptimal solution if the angle between quaternions was too large, but I wasn't aware that it was undefined at sin(theta) = 0, or that floating point precision errors could cause illegal inputs

to be fed into arccosine, or lead to nans being produced by it. Once all these errors were fixed, interpolation was straightforward, just transforming values from euler angles to quaternions, slerping, then transforming back into euler angles.

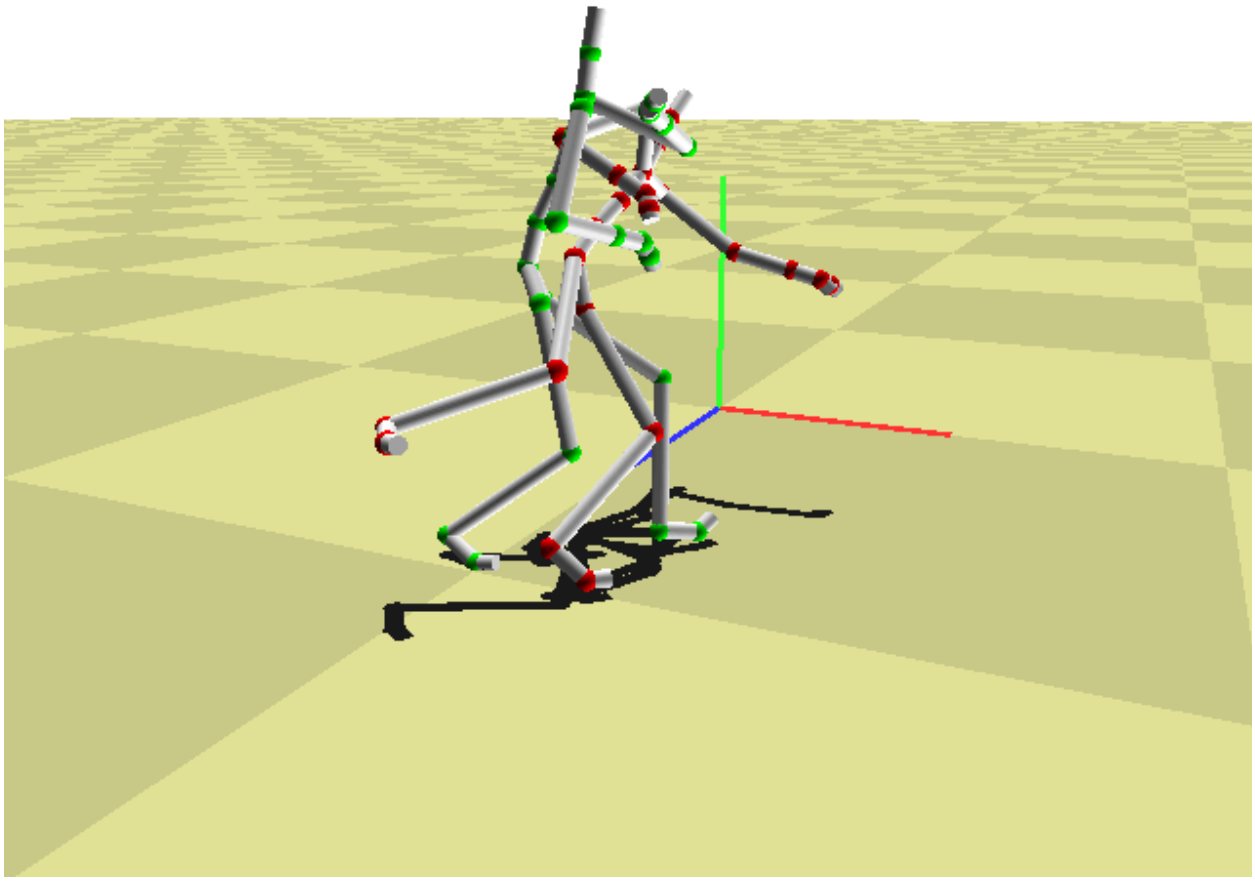
BezierInterpolationQuaternion ended up being the easiest to implement. I basically just copied the code from BezierInterpolationEuler, replaced angle vectors with quaternions, and called it a day.

I'll admit, it's hard to compare these methods from just implementing the code, drawing basic statistics, and watching videos, but I'll try. Euler angles beat quaternions in terms of speed. Interpolating euler angles was much faster, and it's no surprise, SLERP has to perform a number of trigonometric calculations for every interpolated angle. Linear methods are also faster than bezier, which is no surprise, considering the huge amount of calculations required for bezier curve interpolation. Bezier interpolation also requires much more information to be held in memory for formulas to be calculated with ease.

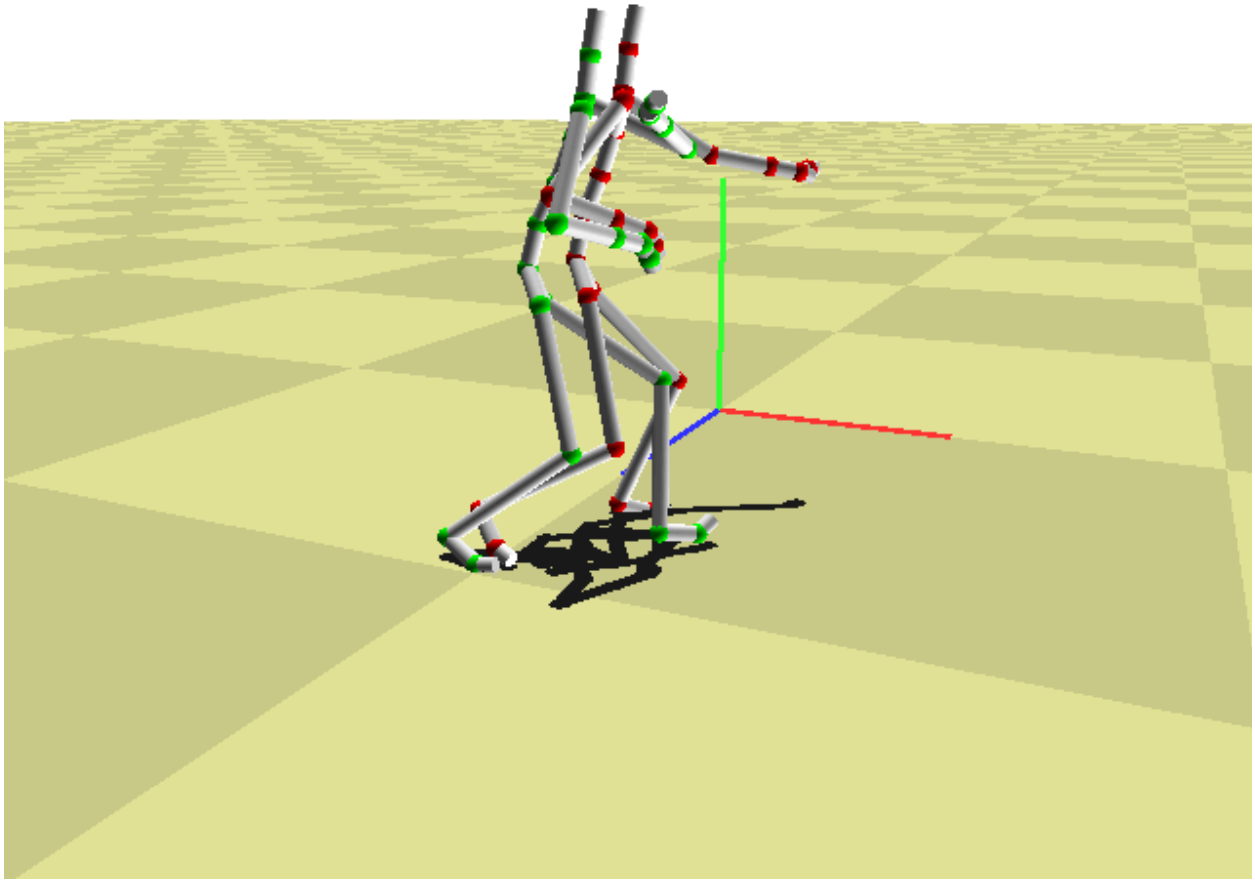




Where euler angles benefit with speed and efficiency, they suffer in accuracy. They're generally less accurate than quaternions. In particular, they're bad at interpolating when angle changes are sudden and steep, like a character jumping or turning suddenly. This is evident in both linear euler and bezier euler interpolation, reflected by graphs 3 and 4. Appearance becomes exceptionally worse as the gap between keyframes grows larger, too, sometimes making character movement feel unnatural and robotic, as shown below. Despite this, when angle changes are small and gradual, and gaps between keyframes are relatively small, euler angles can look just about as good as quaternions.

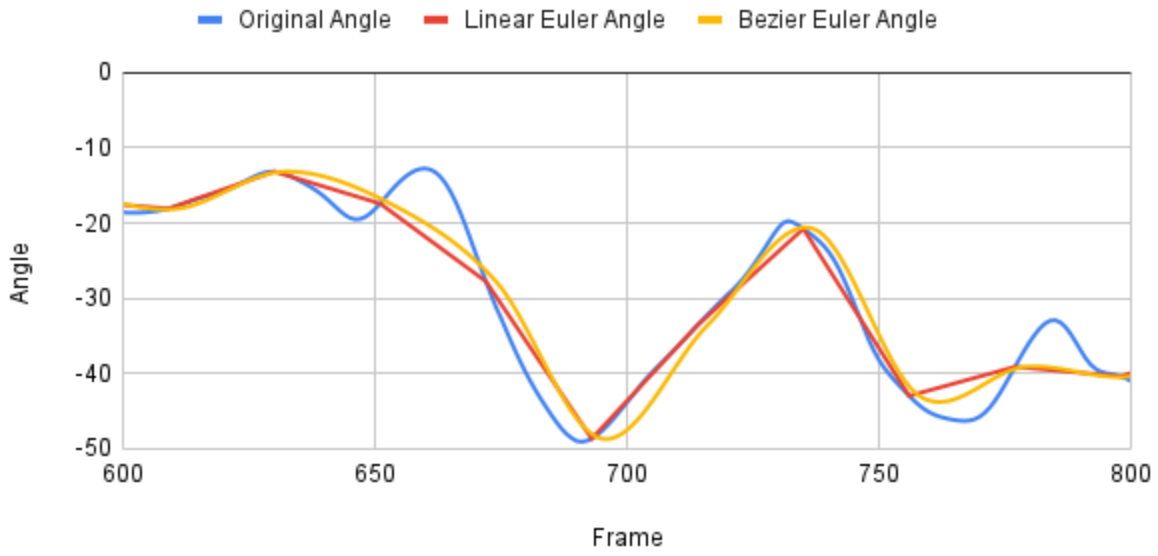


Linear euler interpolation at N = 100

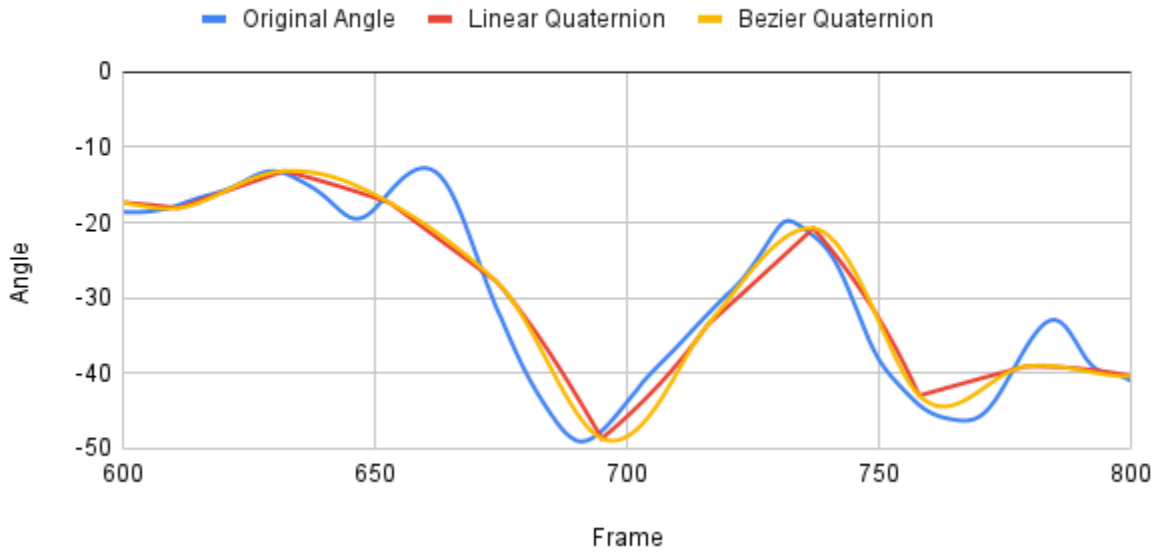


Linear quaternion interpolation at N = 100

Graph 1 - Original Angle, Linear Euler Angle and Bezier Euler Angle



Graph 2 - Original Angle, Linear Quaternion and Bezier Quaternion



Quaternion interpolation is better in nearly every way, save for performance. While it was never so slow as to render the system unwieldy, SLERP uses six trigonometric, linear algebraic, and seven arithmetic methods, as well as some housekeeping, vs. LERP's four arithmetic operations. SLERP also fails to work when quaternions are at a 0 or 180 degree angle, requiring

the assistance of LERP sometimes. Due to this and the problems mentioned above, it was harder to implement and use. Converting between euler angles and quaternions also takes extra computation time.

In terms of appearance, quaternions are, again, superior in nearly every circumstance. Quaternion interpolation curves are smoother, and avoid the sudden (undifferentiable) jolts and jerks that appear in euler angle interpolation, as evidenced by graphs 1 and 2, although these jerks are still present around keyframes for linear interpolation (which makes sense, that's the whole point of bezier interpolation). As shown in graphs 3 and 4, quaternion interpolation does a better job handling and approximating sudden angle shifts than euler angles, even with decent gaps between keyframes.

Both methods struggle around inflection points of the rotation, such as quickly moving forward then backwards. This will be a problem with all interpolation methods, as they can't really guess whether an angle is going up to down or down to up, so they move through the average. If keyframes are in the right spot this can be avoided, but they usually aren't, at least when these changes happen quickly.

I didn't write it with extra credit in mind, but my CompareMotion method automatically parses a motion and prints it to a new csv file, which I can then use for statistics. I don't know if that's creditworthy, but I thought it was a cool thing to implement!