Lab 10 - Animating Orientation

You may work in pairs on this assignment. To receive credit, push your code to your git repository by 11:59PM on April 24th. Be sure to send me an email with your partner's name and which repository the code is pushed to.

So far we've been using matrices to represent orientations. Remember that a 3x3 matrix is required to represent a 3D orientation, so a minimum of 9 values. Quaternions are another representation of orientation that use only 4 values. Quaternions also provide a nice mathematical means of interpolating between two quaternions along the shortest path, called spherical linear interpolation, or slerp. Euler angles are another way of representing orientation and are often the form exposed to animation artists. Conceptually, Euler angles are easier to grasp than quaternions, but suffer from gimbal lock, where you can lose the ability to rotate about an axis in certain configurations. In this lab, we'll explore the difference between animating these different orientation representations.

Part 1 - Interpolation of matrices

You just about NEVER want to directly interpolate transformation matrices. This part is intended to show you why. Each vector that makes up the matrix is interpolated, which can result in a skewed and scaled intermediate space. In the provided code, an array of euler angle rotations is used to represent our animation sequence. Feel free to add or change the values in the rotations array. In the animate method 'from' and 'to' variables are defined that represent the current 2 second chunk of animation. A variable 't' is a floating point value from 0 to 1 that represents the percentage of the way through the current 2 second chunk. Construct two matrices, one that represents the 'from' orientation and one that represents the 'to' orientation. Use the 't' variable to linearly interpolate between the from matrix and to matrix and store it in the matrixLerp variable defined as a private member variable in GLWidget. Setting matrixLerp will change the model matrix of one of three cubes rendered in the paintGL method. In the following parts, we'll animate the other two using different techniques.

Part 2 - Interpolation of Euler angles

In this step, you need to use 'from', 'to' and 't' to get an interpolated vec3 of Euler angles. Convert the resulting vec3 to a 4x4 matrix and store it in eulerLerp. eulerLerp is used in paintGL to modify the orientation of another cube.

Part 3 - Spherical linear interpolation of quaternions

Finally, you will convert the 'from' and 'to' variables to quaternions before performing any interpolation. glm provides a quaternion data structure called quat. You can construct a quaternion using Euler angles by passing the constructor a vec3. Do that with 'from' and 'to'.

Quaternions can be interpolated using spherical linear interpolation, also known as slerp. Spherical linear interpolation is similar to linear interpolation as it takes a value between 0 and 1 to combine two other values. In the case of quaternions, it returns a combined orientation along the shortest path between them. Use the glm::slerp function to combine your quaternions using the ‘t’ variable. The last step is to convert the resulting quaternion into a 4x4 matrix that can be used by our vertex shader. Use glm::toMat4 and store the result in the quatSlerp matrix to change the animation of the third cube.

Things to notice

Linearly interpolating transform matrices is a bad idea as the interpolated matrix will have strange scaling and skewing.

Animating Euler angles is the standard way to animate orientation. Conceptually, it’s easy to grasp and can be used to spin objects around multiple times. Gimbal lock can be a serious problem, though, so animators must watch out for those situations. You can notice the effects of gimbal lock if you rotate about Y 90 degrees. After doing so, notice how changing the X or Z rotation results in a rotation about the same axis.

Quaternions don’t inherently suffer from gimbal lock, but as Euler angles are commonly what orientation is input as you still must be aware that it can be a problem.

When animating between quaternions, notice how the shortest path is taken, whereas with Euler angles it can take a much longer path.

There are no quaternion operations implemented in GLSL, so if you want to use them in a vertex or fragment shader you’d need to implement them yourself (or find them online).

Recommended Reading

Quaternions and spatial rotations
Rotations - OpenGL Tutorial
glm quaternion API