Tutorial 5

In this tutorial we're going to introduce rotations into the engine. I will discuss the implementation of a quaternion class, and how to use it. This is only a short tutorial as there are many example implementations of quaternions, but many of the tutorials on quaternions don't provide any examples on how to use them. Also, the unit tests that complement the code show how quaternions are used, and provide a reference for anybody else who wants to implement a quaternion class.

A rotation is a transformation that changes the orientation of a rigid body (the point of rotation is specified to be the body's centre, so that the rotation does not transform the location of the object). The orientation is a rotation that rotates the body away from its reference orientation (taken such that the body's axes coincide with the world axes.)

There are numerous ways to represent rotations. One of the simplest to understand is angle-axis rotation, where the axis of rotation is specified, along with the angle that the object should be rotated about that axis. Euler angles can be formed by creating a quaternion by concatenating together three successive rotations about the Euler axes.

The quaternion has been implemented as a class, but similar to the `Vector` class we introduced in an earlier tutorial, it would probably give better performance as a struct (to avoid stressing the garbage collector). Additionally, the code has been implemented to be simple, for example the quaternion operations use the vector cross product rather than explicitly expanding out the operations. We will investigate the effect on performance of making these changes in a later tutorial where we benchmark and profile the application.

The quaternion operations are as given in Physics for Game Developers (except for quaternion concatenation, which is as given on the sjbrown website, see http://www.sjbrown.co.uk/?article=quaternions).

Additionally, my implementation of the method to rotate a quaternion by another quaternion (concatenation) normalises the quaternion returned. This is done to ensure that the quaternion maintains a magnitude of 1. If we didn't do this, the quaternion would quickly become non-unit due to rounding errors. One example of where this would cause problems is in the `Angle` property, attempting to return the `Acos` of a value greater than 1 would result in `double.NaN` being returned.

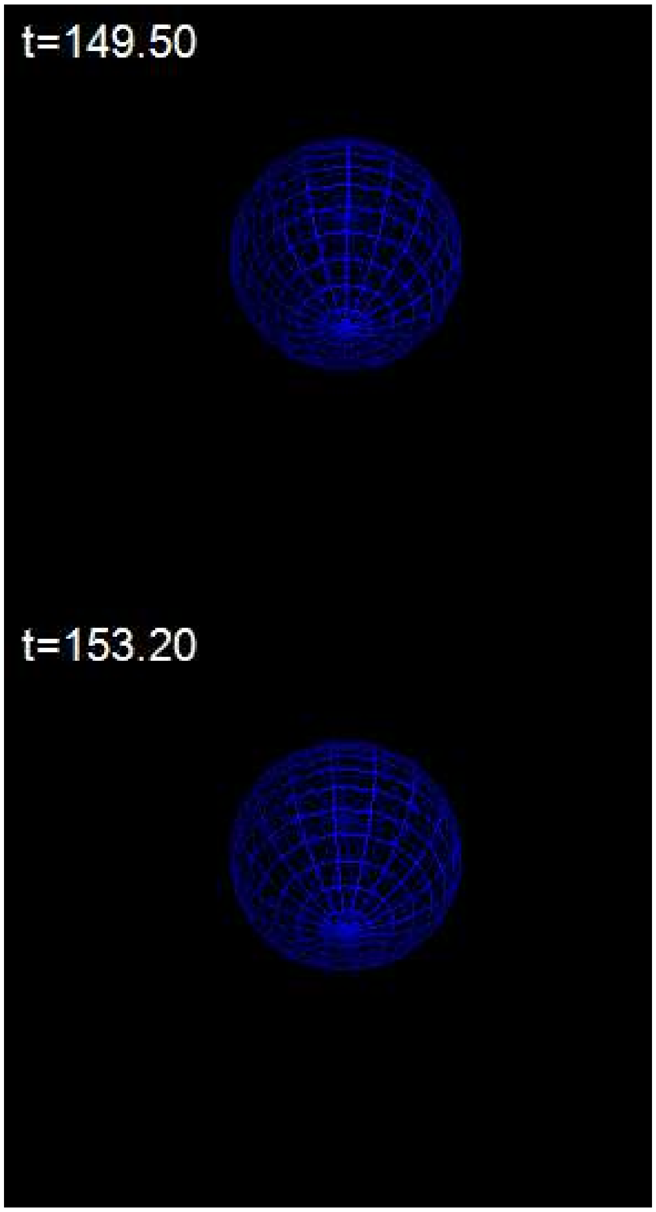
Using Quaternions

As an example of using quaternions to represent rotations, I've built an example of a tilted globe rotating about its axis. This is implemented in Scene5.cs (and the unit tests break these steps down further).

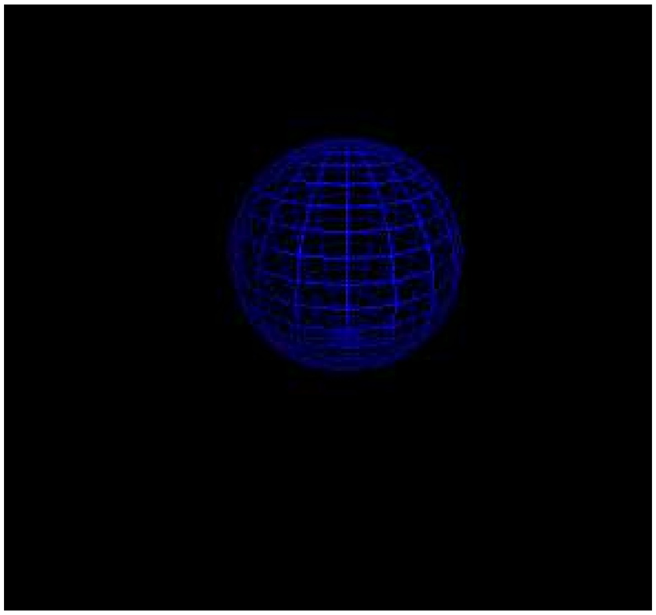
The `Sphere` has been changed so that it is drawn as wireframe rather than solid as a quick and easy way to see that it is rotating. The `gluSphere()` call draws a sphere with its poles aligned along the z axis.

We want to rotate the sphere about the line that passes through both of its poles, i.e. the sphere's local z axis. The `Quaternion.MakeFromAngleAxis()` method can be used to construct the quaternion with the angle to rotate by, and the z axis. To get the sphere to rotate, the `Orientation` property can be set to a new rotation on each call:

```
private double angleToRotate = 0.0;
private double angleIncrement = Math.PI / 500.0;
/// <summary>
/// Updates all of the objects in the scene with the given
/// time step.
/// </summary>
/// <param name="deltaTime">The time step, in seconds.</param>
public void Update(double deltaTime)
{
    eulerRigidBody.Orientation =
        Quaternion.MakeFromAngleAxis(angleToRotate,
        new Vector(0.0, 0.0, 1.0));
    angleToRotate += angleIncrement;
    foreach (EulerRigidBody rigidBody in this.rigidBodies)
    {
        rigidBody.Update(deltaTime);
    }
}
```



We now want to draw the sphere so that the north pole is displayed on *top*, i.e. so that the sphere's poles are drawn aligned with the world's y axis. We can rotate the globe to be in the desired orientation by rotating the globe by 90^o about the x axis:



```
            eulerRigidBody.Orientation =
                Quaternion.MakeFromAngleAxis(Math.PI / 2.0,
                    new Vector(1.0, 0.0, 0.0));
```

Now, we want to rotate the globe about its axis when it's in the new orientation. We can do this by first rotating the quaternion about its local z axis, and follow this by concatenating it with a quaternion representing a rotation about the world's x axis.

```
            eulerRigidBody.Orientation =
                Quaternion.MakeFromAngleAxis(angleToRotate,
                new Vector(0.0, 0.0, 1.0)).Rotate(
                    Quaternion.MakeFromAngleAxis(Math.PI / 2.0,
                        new Vector(1.0, 0.0, 0.0)));
```

This creates a quaternion representing the rotation about the local z axis, and then calls `Rotate()` on it, passing in a rotation about the world's x axis.

To represent a globe spinning about a tilted axis follows exactly the same method as shown above. The code is prevented slightly differently however. In the scene's constructor, we create a sphere tilted on its axis by setting an orientation formed by rotating first by approximately 85º about the x axis, followed by a rotation about the z axis of 11.25º.

```
            // Glu sphere draws a sphere with the poles aligned
            // along the z axis.
            // We want to display the sphere with the poles aligned
            // along the y axis, so we perform an initial 90 degree
            // rotation around the x axis.
            sphereOrientation = Quaternion.MakeFromAngleAxis(
                (15.0 / 16.0)
                * Math.PI / 2.0, new Vector(1.0, 0.0, 0.0));
            // We want to tilt the sphere by a rotation about the
            // world's z axis:
            sphereOrientation = sphereOrientation.Rotate(
                new Quaternion(
                Math.PI / 16.0, new Vector(0.0, 0.0, 1.0)));
```

We split the forming of the initial orientation into two separate statements purely for clarity, and it allows us to insert more comments to describe what we're doing.

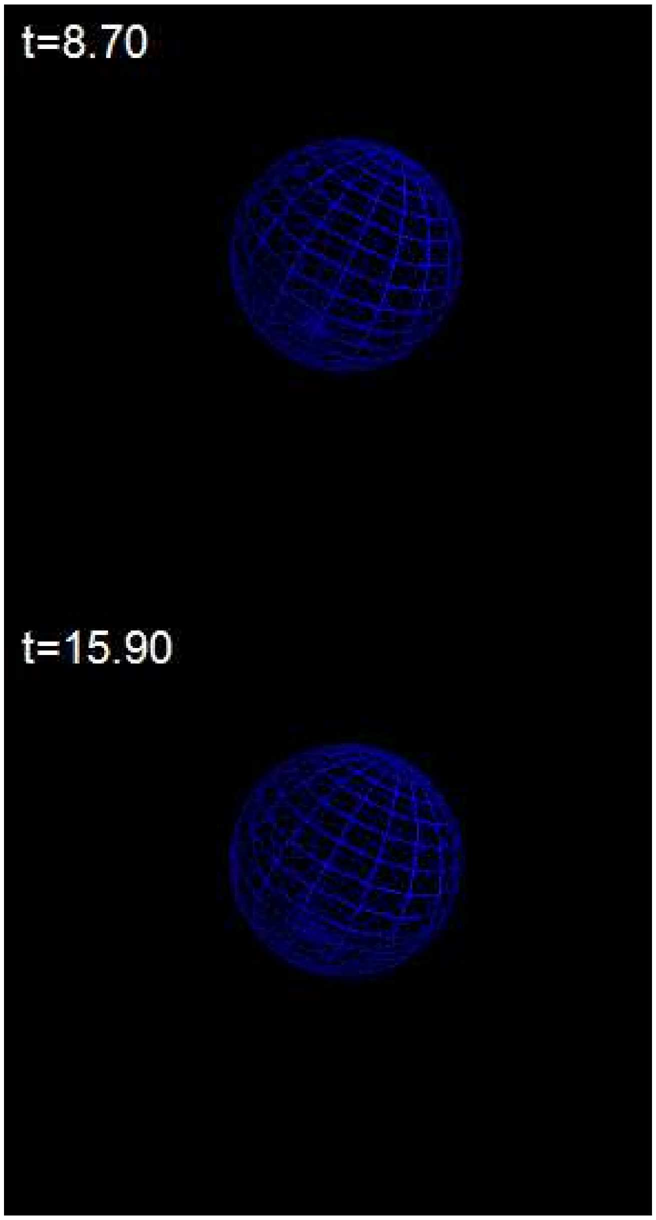
We create a quaternion representing the increment. This represents how the sphere's orientation is going to change on every update:

```
            // The incremental rotation is about the body's local
            // z axis (aligned with the sphere's poles)
            increment = new Quaternion(0.1 / 100.0,
                new Vector(0.0, 0.0, 1.0));
```

Now, to perform the rotation in each update we simply have:

```
                eulerRigidBody.Orientation =
                   increment.Rotate(eulerRigidBody.Orientation);
```

And that's it; we have a sphere rotating about an axis tilted from the y axis.


t=8.70
t=15.90