

# COMP3201 – Computer Graphics

## Module 3: Realism and Performance

### 3.7 Euler Angles, Gimbal Lock, and Quaternions

In this section we will examine three related but independent concepts. These are Euler angles, gimbal lock and quaternions.

#### 3.7.1 Euler Angles

Euler angles are angles of rotation about three mutually orthogonal axes (for example the three coordinate axes  $x,y,z$ ). These angles are used to orient a body in 3D space (equivalently, think about the roll, pitch, and yaw).

In many mathematical and physical applications great use has been made of Euler angles. Because of this success Euler angles came to be used in 3D graphics. However there are a few problems with using a naïve approach to Euler angles to describe rotations. The first of these problems is the ease with which one can end up in a condition known as gimbal lock. Another problem is that it is possible to achieve non-smooth interpolations of Euler angles in an animation application.

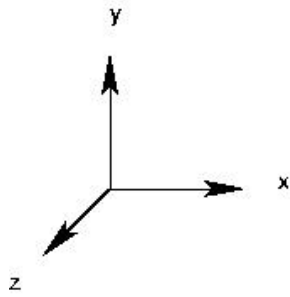
Euler angles are a complete description of 3D orientation and can be used successfully if care is taken. However, due to the ease with which problems occur when using Euler angles and the computation efficiency of quaternions, in many professional applications quaternions are the preferred method of specifying orientation.

#### 3.7.2 Gimbal Lock

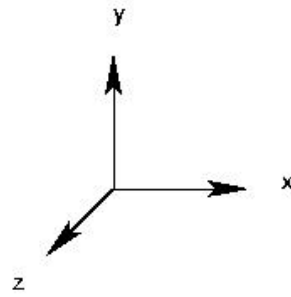
In terms of 3D graphics, gimbal lock is the phenomenon of rotating around 3 axes in the model coordinate system but only achieving a rotation about 2 axes in world coordinates. One way of achieving gimbal lock is explicitly demonstrated in Figure 1. Firstly an arbitrary roll around the x-axis occurs. Next a 90 degree roll around the y-axis is executed. Now the z-axis in the current local coordinates is aligned with the original x-axis, thus a rotation around the current z-axis is equivalent to a rotation around the original x-axis.

This may seem like a contrived example and that nobody would ever make such a mistake. However consider the fact that this mistake is so important that it actually has its own name!

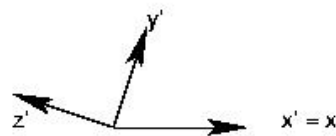
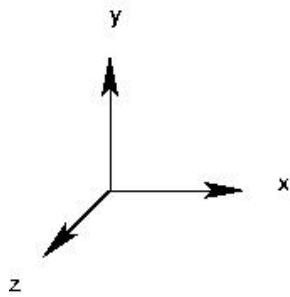
Eye coordinates



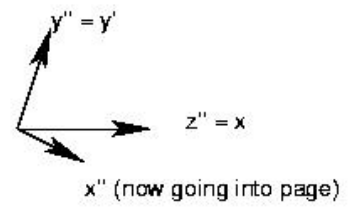
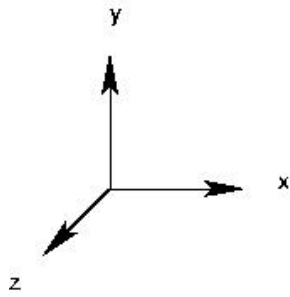
World coordinates



perform x-roll



perform 90 degree y-roll



perform z-roll

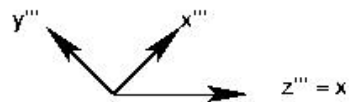
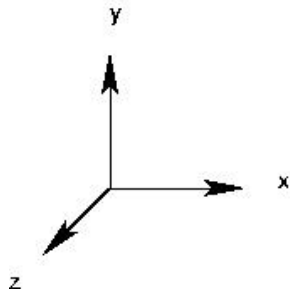


Figure 1: Gimbal lock

### 3.7.2.1 Mathematics of Gimbal Lock

Rotations around the x, y, and z, axes are mathematically represented by the following matrices.

$$R_x = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos(\theta_x) & -\sin(\theta_x) & 0 \\ 0 & \sin(\theta_x) & \cos(\theta_x) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$R_y = \begin{bmatrix} \cos(\theta_y) & 0 & \sin(\theta_y) & 0 \\ 0 & 1 & 0 & 0 \\ -\sin(\theta_y) & 0 & \cos(\theta_y) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$R_z = \begin{bmatrix} \cos(\theta_z) & -\sin(\theta_z) & 0 & 0 \\ \sin(\theta_z) & \cos(\theta_z) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

If we execute the sequence of rotations outlined in Figure 1, the entire transformation matrix is given by (using  $\theta_y = 90^\circ$ )

$$M = R_z R_y R_x = \begin{bmatrix} 0 & -\sin(\theta_z - \theta_x) & \cos(\theta_z - \theta_x) & 0 \\ 0 & \cos(\theta_z - \theta_x) & \sin(\theta_z - \theta_x) & 0 \\ -1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

From the above transformation matrix we can see that the effect of the rotation around the x-axis is exactly the same as the effect of the rotation around the z-axis but in the opposite direction. Thus what we may have thought were three independent rotations are in fact only two independent rotations.

### 3.7.3 Euler's Displacement Theorem

*The general displacement of a rigid body with one point fixed is a rotation about some axis.*

Consider now changing the orientation of an object. If we make an arbitrary number of rotations to achieve this change in orientation then Euler's displacement theorem tells us that we could equivalently find some axis that would take us from the initial orientation to the final orientation using only one rotation.

OpenGL supports rotation around an arbitrary axis as the `glRotatef` command takes an angle and an axis. Unfortunately there are a few problems with this approach. The first is that although Euler's displacement theorem tells us of the existence of this "one axis" that we can rotate about, it doesn't tell us how to calculate that axis.

The second problem that arises is when finer animation is required. When animating frames for a movie output, finer animation is required than that for real time motion. This need for finer animation usually is achieved by interpolating the Euler angles (if these have been specified). Unfortunately the interpolation of Euler angles can give non-smooth interpolated orientations and hence a jerky motion.

Quaternions are currently the preferred method for dealing with rotations and are covered in the next section.

### **3.7.4 Quaternions (Mathematical properties)**

Quaternions are a 19<sup>th</sup> century mathematical device which also have use in representing rotations. Quaternions are an outgrowth of complex numbers.

Complex numbers define a number to represent the square root of negative one. This number is usually called  $i$  (often electrical engineers call it  $j$ ).  $i$  is often referred to as the pure imaginary number.

$$\mathbf{i}^2 = -1.$$

Quaternions introduce two further imaginary numbers,  $j$  and  $k$ .

$$\mathbf{j}^2 = -1,$$

$$\mathbf{k}^2 = -1.$$

These new numbers while similar to  $i$  are not the same as  $i$  due to the following properties.

$$\mathbf{ij} = \mathbf{k} \quad \mathbf{ki} = \mathbf{j} \quad \mathbf{jk} = \mathbf{i}$$

$$\mathbf{ji} = -\mathbf{k} \quad \mathbf{ik} = -\mathbf{j} \quad \mathbf{kj} = -\mathbf{i}.$$

A quaternion number can be expressed as

$$q = a + b\mathbf{i} + c\mathbf{j} + d\mathbf{k}.$$

It is also common to express a quaternion as

$$q = (s, \mathbf{v})$$

$$= s + v_x\mathbf{i} + v_y\mathbf{j} + v_z\mathbf{k}$$

where  $s$  is referred to as the scalar part of the quaternion and  $\mathbf{v}$  is referred to as the vector part of the quaternion with axes  $\mathbf{i}$ ,  $\mathbf{j}$ ,  $\mathbf{k}$ .

### 3.7.4.1 Length

The length of a quaternion is defined by

$$\|q\| = \text{sqr}t(s^2 + v_x^2 + v_y^2 + v_z^2).$$

If the length of a quaternion is one (unity) then the quaternion is called a *unit quaternion*. Unit quaternions are of interest because they are the ones that represent rotations in 3D space.

### 3.7.4.2 Multiplication

Two quaternions  $q_1$  and  $q_2$  are multiplied together to produce another quaternion by

$$q_1 q_2 = (s_1 s_2 - \mathbf{v}_1 \bullet \mathbf{v}_2, \quad s_1 \mathbf{v}_2 + s_2 \mathbf{v}_1 + \mathbf{v}_1 \times \mathbf{v}_2).$$

Note that the cross product term means that multiplication of quaternions is not commutative, that is

$$q_1 q_2 \neq q_2 q_1.$$

The identity for multiplication is given by (alternatively, what is the quaternion,  $I$ , such that  $qI = q$ )

$$I = (1, 0, 0, 0).$$

The inverse for multiplication is given by (alternatively, what is the quaternion,  $q^{-1}$ , such that  $q q^{-1} = I$ )

$$q^{-1} = \frac{1}{\|q\|^2} (s, -\mathbf{v}).$$

Mathematically this means that the quaternions form a group. (It can be shown that the quaternions form a division ring.)

### 3.7.4.2 Conjugate

The conjugate of a quaternion,  $q = (s, \mathbf{v})$  is defined by

$$\bar{q} = (s, -\mathbf{v}).$$

Conjugates have a few useful properties such as

$$\begin{aligned} q \bar{q} &= (\|q\|^2, \mathbf{0}), \\ \bar{q} &= \|q\|^2 q^{-1}. \end{aligned}$$

It is interesting to note that for a unit quaternion the second property above reduces to the conjugate equaling the inverse. Thus, since only unit quaternions are used in computer graphics, this gives a fast method of calculating the inverse quaternion.

### **3.7.5 Convert from Euler rotation to a Quaternion**

The quaternion that is equivalent to a rotation,  $\theta$ , about a unit axis,  $\mathbf{n}$ , is given by (see [1] for a proof of this statement)

$$q = (\cos(\theta/2), \sin(\theta/2) \mathbf{n}).$$

### **3.7.6 Combining rotations using Quaternions**

Two (or more) rotations, which are expressed as quaternions, can be combined by multiplication. Since multiplication of quaternions yields a quaternion we are guaranteed that the result is a quaternion. It is possible to prove that the resulting quaternion is the correct combined rotation.

The order of multiplication is important since quaternion multiplication is not commutative. This makes intuitive sense as the order of rotations is important even when we aren't using quaternions to express the rotations. The quaternions are multiplied in the same order that the equivalent rotation matrices are combined.

### **3.7.7 Convert from Quaternion to Matrix**

In order to work with OpenGL we need to specify matrix manipulations. Thus we need to convert from the quaternions to a matrix which can be multiplied onto the current stack. There are two ways of achieving this conversion. The first is to directly generate the appropriate matrix. The second (and easier alternative) is to generate an angle and axis of rotation.

A rotation expressed as a unit quaternion can be re-expressed in matrix form as

$$\begin{bmatrix} 1 - 2v_y^2 - 2v_z^2 & 2v_x v_y - 2sv_z & 2v_x v_z + 2sv_y & 0 \\ 2v_x v_y + 2sv_z & 1 - 2v_x^2 - 2v_z^2 & 2v_y v_z - 2sv_x & 0 \\ 2v_x v_z - 2sv_y & 2v_y v_z + 2sv_x & 1 - 2v_x^2 - 2v_y^2 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

See [1] for a derivation of the above expression.

### 3.7.8 Convert from Quaternion to Axis-Angle

To extract an angle,  $\theta$ , and axis,  $\mathbf{n}$ , from the quaternion,  $q=(s,\mathbf{v})$ , which are suitable for a call to `glRotate*`, the following is used.

$$\theta = 2 \arccos(s),$$

$$\mathbf{n} = \mathbf{v}.$$

Many references on quaternions will scale the axis  $\mathbf{n}$  to be unit. This is unnecessary as `glRotate*` does not require a unit length axis to be supplied.

### 3.7.9 Example

Let us now examine the example in Figure 1 but this time expressed in terms of quaternions. This example is a roll of  $\theta_x$  around the x-axis, followed by a 90 degree roll around the y-axis, and finally a roll of  $\theta_z$  around the z-axis. Section 3.7.5 gives us the expression necessary to convert the Euler angles into quaternions. Expressed as unit quaternions these rotations are:

$$q_x = (\cos(\theta_x / 2) \quad \sin(\theta_x / 2) \quad 0 \quad 0)$$

$$q_y = (\cos(90^\circ / 2) \quad 0 \quad \sin(90^\circ / 2) \quad 0)$$

$$q_z = (\cos(\theta_z / 2) \quad 0 \quad 0 \quad \sin(\theta_z / 2))$$

These rotations can be combined (by multiplication) to form one effective rotation.

$$q = q_z q_y q_x.$$

The following facts are useful for doing the above quaternion multiplication:

$$\cos(45^\circ) = \sin(45^\circ)$$

$$\cos(-\theta) = \cos(\theta)$$

$$\sin(-\theta) = -\sin(\theta)$$

$$\cos(A + B) = \cos(A) \cos(B) - \sin(A) \sin(B)$$

$$\sin(A + B) = \sin(A) \cos(B) + \cos(A) \sin(B)$$

$$\cos^2(\theta) + \sin^2(\theta) = 1$$

The actual multiplication is left as an exercise to the reader. The result is

$$q = \left( \frac{1}{\sqrt{2}} \cos\left(\frac{\theta_z - \theta_x}{2}\right) \quad \frac{-1}{\sqrt{2}} \sin\left(\frac{\theta_z - \theta_x}{2}\right) \quad \frac{1}{\sqrt{2}} \cos\left(\frac{\theta_z - \theta_x}{2}\right) \quad \frac{1}{\sqrt{2}} \sin\left(\frac{\theta_z - \theta_x}{2}\right) \right)$$

It is possible to check that this expression for the total quaternion rotation is correct (again this is left as an exercise to the reader). Using the expression given in Section 3.7.7 the quaternion can be converted into its equivalent rotation matrix. This rotation

matrix can then be compared (as seen to be identical) to the matrix derived in Section 3.7.2.1.

To use the resulting quaternion back in OpenGL we convert the result into an axis-angle formulation and call `glRotatef`. The angle and axis are given by

$$\begin{aligned} \text{angle} &= \arccos\left(\frac{1}{\sqrt{2}} \cos\left(\frac{\theta_z - \theta_x}{2}\right)\right) \\ n_x &= \frac{-1}{\sqrt{2}} \sin\left(\frac{\theta_z - \theta_x}{2}\right) \\ n_y &= \frac{1}{\sqrt{2}} \cos\left(\frac{\theta_z - \theta_x}{2}\right) \\ n_z &= \frac{1}{\sqrt{2}} \sin\left(\frac{\theta_z - \theta_x}{2}\right) \end{aligned}$$

### 3.7.10 Interpolation

If interpolation between quaternions is needed for animation purposes, the first approach that you may think about is to simply linearly interpolate the quaternions. Unfortunately a linear interpolation will not preserve the unit lengths of the quaternions. This means that instead of staying on the surface of the unit 3-sphere in the 4D quaternion space, we have taken a shortcut through the sphere. This shortcut manifests itself as a changing of rotational velocity of the scene (or objects in the scene). The solution to this is to use a spherical linear interpolation. Spherical linear interpolation is commonly abbreviated to `slerp`.

### 3.7.11 References

[1] “Advanced Animation and Rendering Techniques – Theory and Practice”, Alan Watt and Mark Watt, ACM Press, 1992