MASSEY UNIVERSITY
TE KUNENGA KI PŪREHUROA
UNIVERSITY OF NEW ZEALAND

159.709 Computer Graphics

Quaternions

Daniel Playne
d.p.playne@massey.ac.nz

Orientation

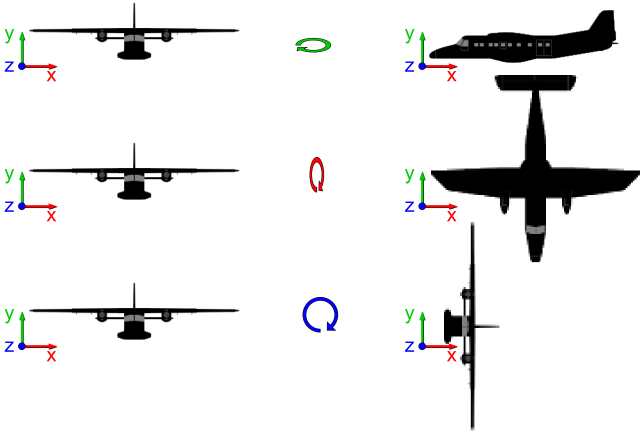
The ***orientation*** of an object describes the rotational part of the position of the object (the other part being the translation).

So far we have defined the orientation of an object by rotating it around the three principle axes ($\vec{\mathbf{x}}$, $\vec{\mathbf{y}}$, $\vec{\mathbf{z}}$).

When the rotations we want are limited to a single axis, it is relatively simple to construct an orientation matrix.

Orientation

For example:



Dornier Image - License: Public Domain, Author: Greg Goebel

Orientation

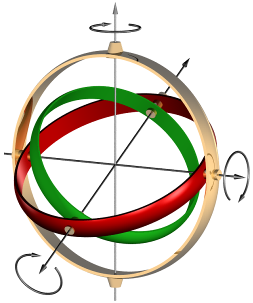
The orientation of an object can be defined by combining these different rotations. The result will depend on the order these rotations are combined - which is correct?

$$R = R_x R_y R_z \quad \text{or} \quad R = R_z R_x R_y \quad \text{or} \quad R = R_z R_y R_x$$

This method of representing orientation is like building a set of three gimbals for the object.

Gimbal

A gimbal allows an object to rotate around a single axis, three can be combined to allow an object to be rotated to any orientation.



Gimbal Image - License: CC-BY-SA-4.0, Author: Svjo, Source:

https://commons.wikimedia.org/wiki/File:Gimbal-3-axis-brass.png

Gimbal-Lock

There is a well-known problem with using this method of representing an orientation called ***gimbal-lock*** which occurs when two of the gimbals are in a parallel configuration.

The gimbals are not exactly locked as they can still move freely, however the object cannot be freely rotated one axis without reorienting the other gimbals first.

The same problem can occur in our 3D graphics programs.

Orientations

The problem arises because we are trying to store and manipulate orientations as a series of rotations. What we need is a way to represent an orientation.

One approach is to store an orientation as a matrix. Each subsequent transformation is applied to the matrix and are accumulated together.

Frame of Reference

At this point it is helpful to consider what order these transforms should be combined together.

If we wish to rotate our current transform matrix R around the x-axis with the transform R_x which order should these matrices be multiplied together?

$$R = RR_x \quad \text{or} \quad R = R_xR$$

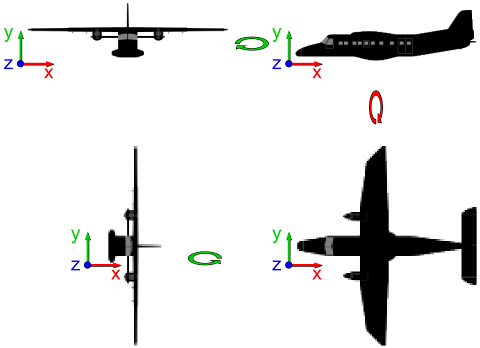
Frame of Reference

The answer will depend on what outcome we are trying to achieve, namely whether we are trying to rotate our object around a ***global*** or ***local*** frame-of-reference.

Rotations with a ***global*** frame-of-reference are performed in ***world-space*** while ***local*** rotations are performed within the ***model-space***.

Frame of Reference

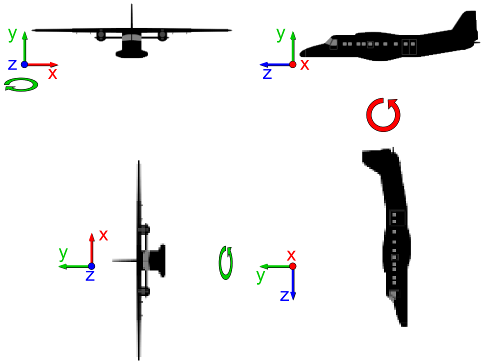
Sequence of rotations in ***global*** frame-of-reference.



Dornier Image - License: Public Domain, Author: Greg Goebel

Frame of Reference

Same sequence of rotations in ***local*** frame-of-reference.



Dornier Image - License: Public Domain, Author: Greg Goebel

Frame of Reference

Applying a rotation R in the ***global*** frame of reference can be achieved by applying multiplying the new rotation by the rotation matrix R_g as follows:

$$R_g = R * R_g$$

Applying a rotation R in the ***local*** frame of reference can be achieved by multiplying in the opposite order:

$$R_l = R_l * R$$

Frame of Reference

For example, the rotations from the example are as follows:

$$
\begin{aligned}
R_y &= \begin{bmatrix} 0 & 0 & -1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{bmatrix} \\
R_x &= \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & -1 \\ 0 & 1 & 0 \end{bmatrix} \\
R'_y &= \begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ -1 & 0 & 0 \end{bmatrix}
\end{aligned}
$$

Frame of Reference

Multiplied in ***global*** order:

$$\begin{aligned} R_{global} &= R'_y R_x R_y \\ &= \begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ -1 & 0 & 0 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & -1 \\ 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} 0 & 0 & -1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{bmatrix} \\ &= \begin{bmatrix} 0 & 1 & 0 \\ -1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix} \end{aligned}$$

Frame of Reference

And in ***local*** order:

$$\begin{aligned} R_{global} &= R_y R_x R'_y \\ &= \begin{bmatrix} 0 & 0 & -1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & -1 \\ 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ -1 & 0 & 0 \end{bmatrix} \\ &= \begin{bmatrix} 0 & -1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix} \end{aligned}$$

Orientations

The main problem with storing orientations in a matrix is that successive floating-point operations can start to introduce errors.

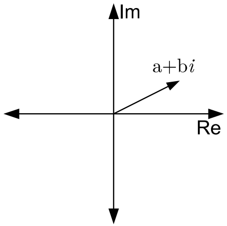
While these errors will be small initially, over time the matrix may no longer represent an orientation and start to introduce other, unintended transformations such as skewing and scaling.

Solving this problem requires periodically re-orthonormalising the matrix to remove these errors - need different approach.

Complex Numbers

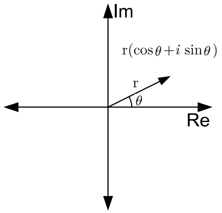
In two-dimensions there is a completely separate way to represent rotations - complex numbers.

Complex numbers can be represented by a real and an imaginary part - e.g. $z = a + bi$:



Complex Numbers

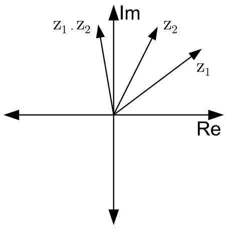
Another representation of a complex number is the polar form $z = r(\cos\theta + i\sin\theta)$:



Complex Numbers

Consider the multiplication of two complex numbers $z_1 = a + bi$ and $z_2 = c + di$:

$$
\begin{aligned}
z_1 \cdot z_2 &= (a + bi) \cdot (c + di) \\
&= a \cdot c + ia \cdot d + b \cdot ci + b \cdot di^2 \\
&= (a \cdot c - b \cdot d) + (a \cdot d + b \cdot c)i
\end{aligned}
$$



Complex Numbers

The importance of complex number multiplication may be more obvious in polar form:

$$\begin{aligned}
& z_1 \cdot z_2 \\
= \quad & r_1(\cos\theta + i\sin\theta) \cdot r_2(\cos\phi + i\sin\phi) \\
= \quad & r_1 \cdot r_2(\cos\theta\cos\phi + i\cos\theta\sin\phi + i\sin\theta\cos\phi + i^2\sin\theta\sin\phi) \\
= \quad & r_1 \cdot r_2((\cos\theta\cos\phi - \sin\theta\sin\phi) + i(\cos\theta\sin\phi + \sin\theta\cos\phi)) \\
= \quad & r_1 \cdot r_2(\cos(\theta+\phi) + i\sin(\theta+\phi))
\end{aligned}$$

Complex Numbers

If both complex numbers have unit length $r_1 = r_2 = 1$ then this becomes:

$$\begin{aligned} z_1 \cdot z_2 &= (\cos\theta + i\sin\theta)\cdot(\cos\phi + i\sin\phi) \\ &= (\cos(\theta+\phi) + i\sin(\theta+\phi)) \end{aligned}$$

Rotations in two-dimensions can be represented by complex numbers and combined by multiplication - is there an equivalent for three-dimensions?

Quaternions

As it turns out there is no three-dimensional equivalent for complex-numbers. However, there is for four-dimensional equivalent - ***quaternions***.

A quaternion has the following representation:

$$q = q_w + q_x i + q_y j + q_z k$$

Where q_w is the real part and $q_x i$, $q_y j$, $q_z k$ are the imaginary parts (note: $i^2 = j^2 = k^2 = ijk = -1$).

Quaternions

The basis elements i, j and k also have the following relationships to each other.

$$\begin{aligned} ij &= k \\ ji &= -k \\ jk &= i \\ kj &= -i \\ ki &= j \\ ik &= -j \end{aligned}$$

Quaternions

Quaternions have the following relationship to a rotation by angle θ in three-dimensions around the axis $\vec{\mathbf{v}} = (v_x, v_y, v_z)$:

$$q = \begin{pmatrix} q_w \\ q_x \\ q_y \\ q_z \end{pmatrix} = \begin{pmatrix} \cos(\frac{\theta}{2}) \\ \sin(\frac{\theta}{2}) v_x \\ \sin(\frac{\theta}{2}) v_y \\ \sin(\frac{\theta}{2}) v_z \end{pmatrix}$$

Quaternions

A ***unit*** quaternion is a quaternion where the magnitude $|q| = \sqrt{q_x^2 + q_y^2 + q_z^2 + q_w^2}$ is equal to 1.

The ***identity*** quaternion is $i = (1, 0, 0, 0)$ where $iq = qi = q$.

The ***inverse*** q^{-1} of a quaternion $q = (q_w, q_x, q_y, q_z)$ where $q^{-1}q = i$ can be calculated by: $q^{-1} = \frac{(q_w, -q_x, -q_y, -q_z)}{q_w^2 + q_x^2 + q_y^2 + q_z^2}$.

A ***pure*** quaternion is a quaternion with a zero real term $q = (0, q_x, q_y, q_z)$.

Quaternions

Quaternion multiplication can be calculated as follows:

$$\begin{pmatrix} a_w \\ a_x \\ a_y \\ a_z \end{pmatrix} \begin{pmatrix} b_w \\ b_x \\ b_y \\ b_z \end{pmatrix} = \begin{pmatrix} a_w \cdot b_w - a_x \cdot b_x - a_y \cdot b_y - a_z \cdot b_z \\ a_w \cdot b_x + a_x \cdot b_w + a_y \cdot b_z - a_z \cdot b_y \\ a_w \cdot b_y + a_y \cdot b_w + a_z \cdot b_x - a_x \cdot b_z \\ a_w \cdot b_z + a_z \cdot b_w + a_x \cdot b_y - a_y \cdot b_x \end{pmatrix}$$

Note that quaternion multiplication is associative $(a \cdot b) \cdot c = a \cdot (b \cdot c)$ but not commutative $(a \cdot b \neq b \cdot a)$

Quaternions

Quaternions can be used to rotate vectors as follows:

Express a vector $\vec{\mathbf{v}}$ as a ***pure*** quaternion $v = (0, \vec{\mathbf{v}})$.

Multiplying this quaternion by a ***unit*** quaternion $q = (\cos\theta, \sin\theta\hat{\mathbf{q}})$:

$$\begin{aligned} v' &= qv \\ &= (\cos\theta, \sin\hat{\mathbf{q}})(0, \vec{\mathbf{v}}) \\ &= (-\sin\theta\hat{\mathbf{q}}\cdot\vec{\mathbf{v}}, \cos\theta\vec{\mathbf{v}} + \sin\theta\hat{\mathbf{q}}\times\vec{\mathbf{v}}) \end{aligned}$$

Which will not be a ***pure*** quaternion (when $\hat{\mathbf{q}}\cdot\vec{\mathbf{v}} \neq 0$ and $\sin\theta \neq 0$).

Quaternions

A vector (represented by a ***pure*** quaternion) can be rotated by multiplying by a ***unit*** quaternion and its inverse in the following order:

$$\begin{aligned}
v &= (0, \vec{\mathbf{v}}) \\
q &= (\cos\theta, \sin\theta\hat{\mathbf{q}}) \\
q^{-1} &= (\cos\theta, -\sin\theta\hat{\mathbf{q}}) \\
v' &= qvq^{-1}
\end{aligned}$$

Quaternions in GLM

GLM provides an implementation for quaternions, this is provided by the data type `glm::quat` with the constructor `glm::quat(w,x,y,z)`.

A quaternion representing a 90° rotation around the x-axis can be constructed as follows:

```
glm::quat q = glm::quat(glm::cos(glm::radians(90.0f/2.0f)),
                        glm::sin(glm::radians(90.0f/2.0f)), 0, 0);
```

Quaternions in GLM

One way to rotate a vector in GLM is to represent it as a quaternion and perform the multiplication by `q` and the inverse `qi` manually:

```cpp
// Quaternion
glm::quat q = glm::quat(glm::cos(glm::radians(90.0f/2.0f)),
                        glm::sin(glm::radians(90.0f/2.0f)), 0, 0);
// Inverse
glm::quat qi = glm::quat(glm::cos(glm::radians(90.0f/2.0f)),
                        -glm::sin(glm::radians(90.0f/2.0f)), 0, 0);
// Vector (as pure quaternion)
glm::quat v = glm::quat(0, 1, 1, 1);

// Rotated vector
glm::quat r = q * v * qi;

// Prints 'quat(1.0, -1.0, 1.0, 0.0)' in the format (x,y,z,w)
std::cout << glm::to_string(r) << std::endl;
```

Quaternions in GLM

A more convenient way is to use the GLM operators to multiply a quaternion by a vector:

```cpp
// Quaternion
glm::quat q = glm::quat(glm::cos(glm::radians(90.0f/2.0f)),
                        glm::sin(glm::radians(90.0f/2.0f)), 0, 0);

// Vector
glm::vec4 v = glm::vec4(1, 1, 1, 0);

// Rotated vector
glm::vec4 r = q * v;

// Prints 'vec4(1.0, -1.0, 1.0, 0.0)' in the format (x,y,z,w)
std::cout << glm::to_string(r) << std::endl;
```

Quaternions

The rotations from two quaternions can be combined by multiplying them together.

It is important to remember that quaternion multiplication is not commutative. Just like matrices, the order the quaternions are applied matters.

The order quaternions are multiplied will define whether rotations are accumulated in ***global*** or ***local*** frame of reference.

Quaternions

The main difference between accumulating rotations this way and using matrices is that quaternions are easy to normalise.

When the length of a quaternion starts to drift away from 1 then it will no longer be representing a simple rotation. This can be detected by comparing the length to one (or more efficiently the length squared) and the quaternion renormalised if it goes outside a certain range.