

Representing Orientations

Fall 2021

10/12/2021

Kyoung Shin Park
Computer Engineering
Dankook University

Orientation

- We will define *orientation* to mean an **object's** instantaneous rotational configuration.
- Think of it as the rotational equivalent of position
- Direction
 - Vector has a direction but not orientation
- Rotation
 - An orientation is given by a rotation from identity orientation
- Angular Displacement
 - The amount of rotation is angular displacement

Representing Orientations

- Is there a simple means of representing a 3D orientation (analogous to Cartesian coordinates)?
 - Not really
- There are several popular options though:
 - Euler angles – the simplest
 - Rotation vectors (axis/angle)
 - Rotation matrices
 - Quaternions
 - etc..

Euler Angles

□ Euler Angles

- Represent any arbitrary orientation as three rotations about three mutually perpendicular axes (rotation about X, Y, Z)
- Sometimes described as “Yaw, Pitch, Roll” or similar
- A sequence of rotations around principle axes is called an *Euler Angle Sequence*

□ Axis order

- Euler angles represent three composed rotations that move a reference frame to a given referred frame.
- Euler angles are used in a lot of applications, but they tend to require some rather arbitrary decisions.
- $(y, x, z), (x, y, z), (z, x, y), \dots$ can be used

XYZ

XZY

XYX

XZX

YXZ

YZX

YXY

YZY

ZXY

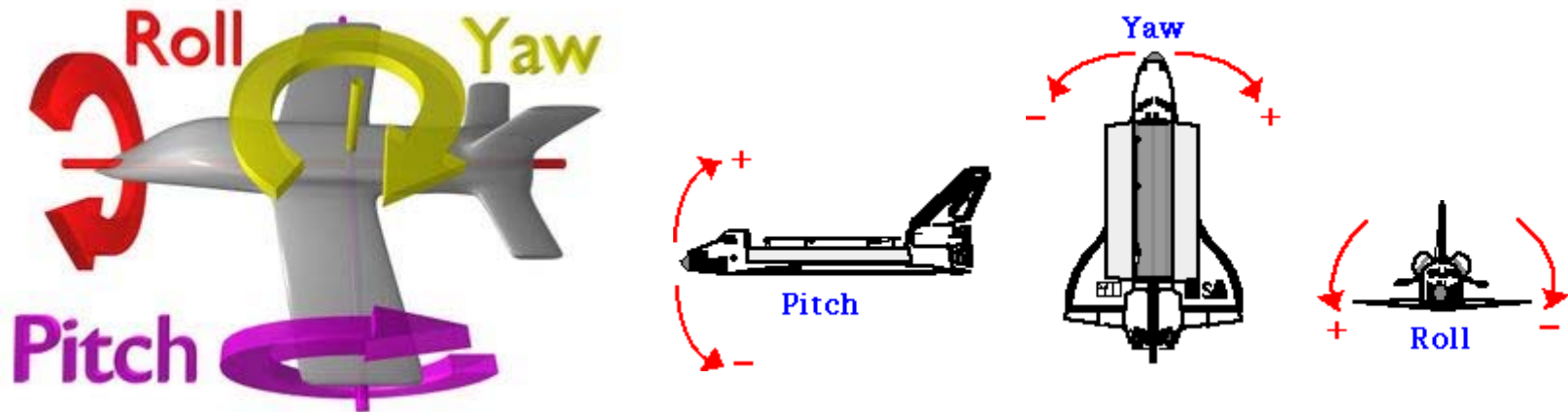
ZYX

ZXZ

ZYZ

Euler Angles

- Yaw, Pitch, Roll
- Yaw (rotation about Y), Pitch (X), Roll (Z) sequence is used in computer graphics.



Euler Angles to Matrix Conversion

- Any orientation can be achieved by composing three elemental rotations
 - i.e., Any rotation matrix can be decomposed as a product of three elemental rotation matrices.

$$\begin{aligned} \mathbf{R}_x \cdot \mathbf{R}_y \cdot \mathbf{R}_z &= \begin{bmatrix} 1 & 0 & 0 \\ 0 & c_x & s_x \\ 0 & -s_x & c_x \end{bmatrix} \cdot \begin{bmatrix} c_y & 0 & -s_y \\ 0 & 1 & 0 \\ s_y & 0 & c_y \end{bmatrix} \cdot \begin{bmatrix} c_z & s_z & 0 \\ -s_z & c_z & 0 \\ 0 & 0 & 1 \end{bmatrix} \\ &= \begin{bmatrix} c_y c_z & c_y s_z & -s_y \\ s_x s_y c_z - c_x s_z & s_x s_y s_z + c_x c_z & s_x c_y \\ c_x s_y c_z + s_x s_z & c_x s_y s_z - s_x c_z & c_x c_y \end{bmatrix} \end{aligned}$$

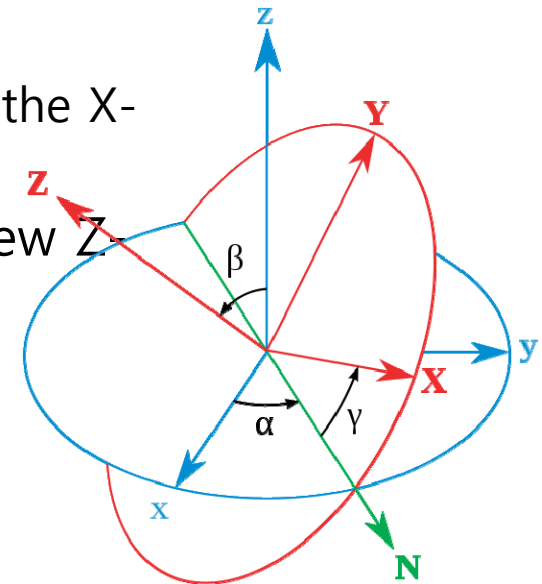
Euler Angle Order

- *As matrix multiplication is not commutative, The order of operations is important.*
- *Rotations are assumed to be relative to fixed world axes, rather than local to the object.*
- One can think of them as being local to the object if the sequence order is reversed.
- Euler angle can be used differently by applications.
 - XYZ convention is widely used in 3D graphics
 - ZXZ convention is used in rigid-body dynamics

Euler Angle Order

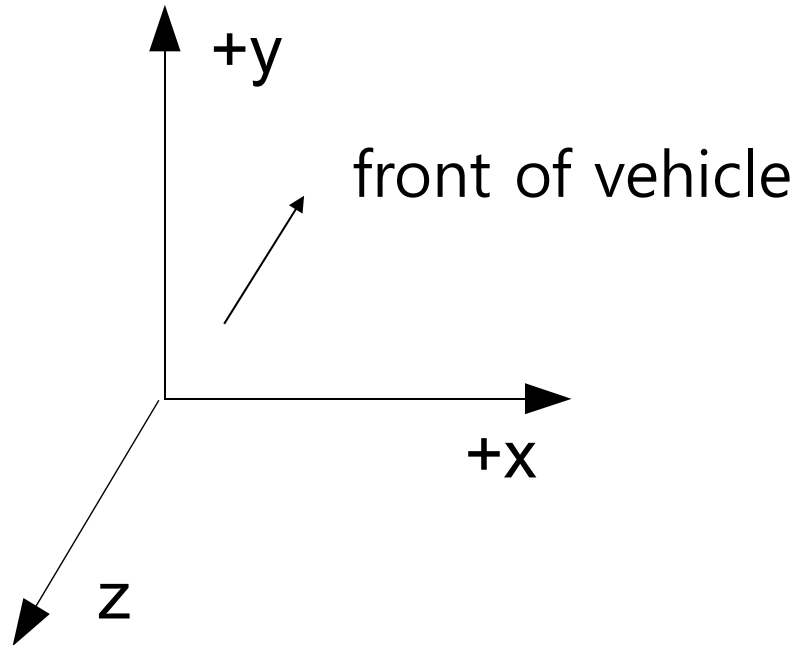
□ ZXZ convention

- XYZ (fixed) system is shown in blue.
- XYZ (rotated) system is shown in red.
- The line of nodes, N, is shown in green.
- (Z-rotation) Rotate about the Z-axis by α .
 - The X-axis now lies on the line of nodes, N
- (X-rotation) Rotate again about the rotated X-axis (i.e., N) by β .
 - The Z-axis is now in its final orientation, and the X-axis remains on the line of nodes
- (Z-rotation) Rotate a third time about the new Z-axis by γ .



Vehicle Orientation Using Euler Angles

- Generally, for vehicles, it is convenient to rotate in roll (z), pitch (x) and then yaw (y) order.
- In situations where there is a definite ground plane, Euler angles can actually be an intuitive representation.

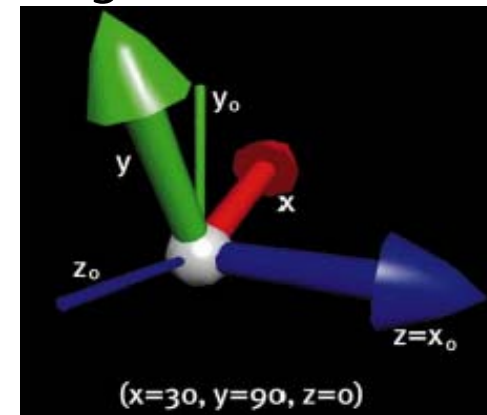
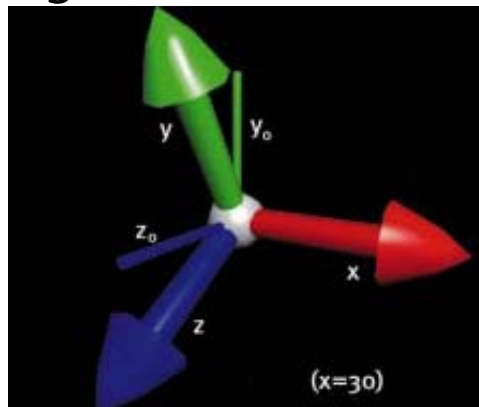
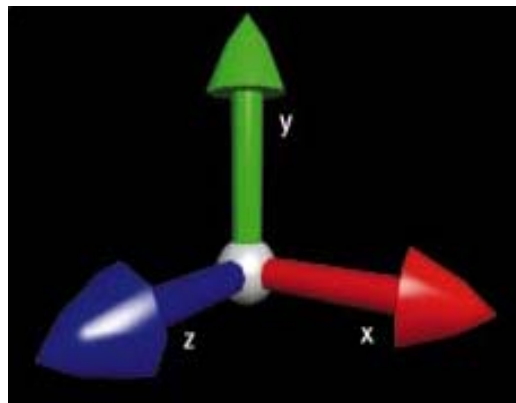


Rotations not uniquely defined with Euler Angles

- Rotations are not uniquely defined with Euler Angles.
- Cartesian coordinates are independent of each other.
 - Arbitrary position = x-axis position + y-axis position + z-axis position
- Euler angles do not act independently of each other.
 - Arbitrary orientation = x-axis rotation matrix * y-axis rotation matrix * z-axis rotation matrix
 - For example, $(z, x, y) = (90, 45, 45) = (45, 0, -45)$

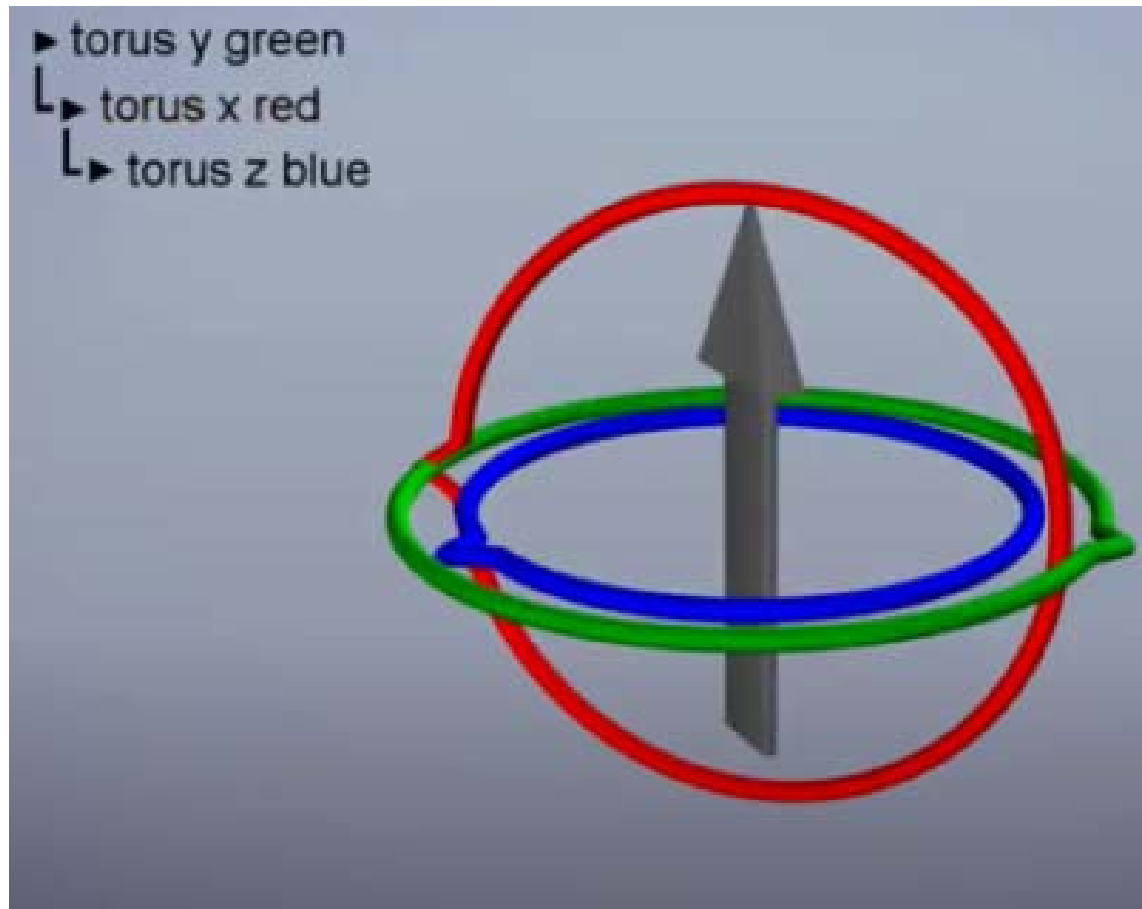
Gimbal Lock

- ❑ One potential problem is '**gimbal lock**'.
- ❑ '**Gimbal Lock**' results when two axes effectively line up, resulting in a temporary loss of a degree of freedom. Change to one of the angles affect to the entire system.
 - This is related to the singularities in longitude that you get at the north and south poles.
 - Rotate 30 about X, then rotate 90 about Y. The current Z-axis is in line with X0-axis. This is what we call 'gimbal lock' situation.
 - Any further rotation about the Z-axis affects the same degree of freedom as rotating about the X-axis – losing the third DOF.



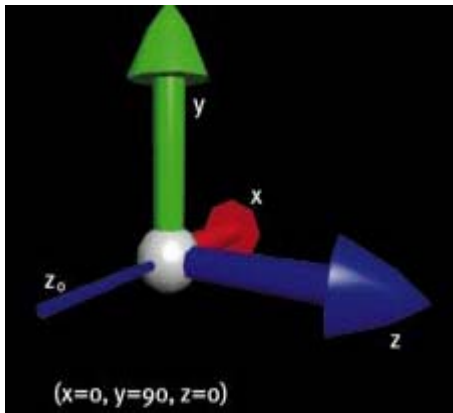
Gimbal Lock

- <https://www.youtube.com/watch?v=zc8b2Jo7mno>

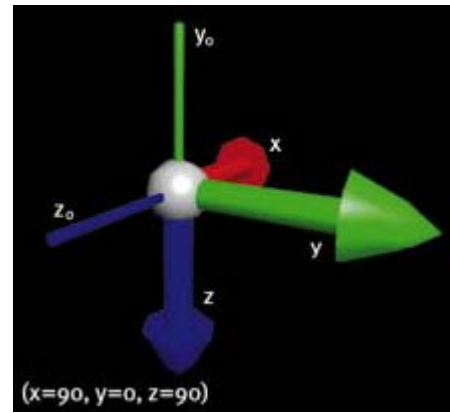


Problem with Interpolating Euler Angles

- The second problem is with generating the in-between frames, due to the fact that the Euler angles do not act independently of each other.
- Let say you have the object with $(0,180,0)$ of rotation angles, and the next keyframe rotation angles is in $(0,0,0)$
 - $(180,0,180)$ represents the same orientation of $(0,180,0)$
 - But, the halfway between $(0,180,0)$ and $(0,0,0)$ is not same orientation of the halfway between $(180,0,180)$ and $(0,0,0)$



Halfway between
 $(0,0,0)$ and $(0,180,0)$



Halfway between
 $(0,0,0)$ and $(180,0,180)$

Euler Angles

- Euler angles are used in a lot of applications, but they tend to require some rather arbitrary decisions.
- They also do not interpolate in a consistent way (but this isn't always bad).
- **They can suffer from Gimbal lock and related problems.**
- There is no simple way to concatenate rotations.
- Conversion to/from a matrix requires several trigonometry operations.
- **They are compact (requiring only 3 numbers).**

glm::yawPitchRoll

□ glm::yawPitchRoll

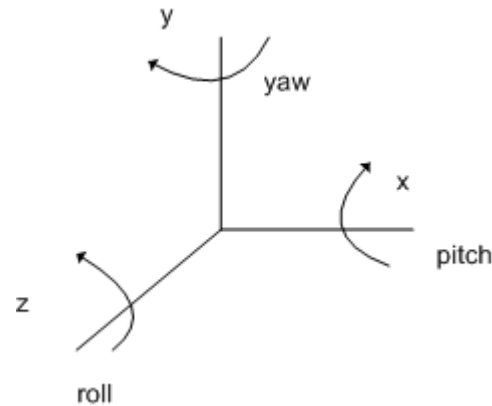
// Yaw/Pitch/Roll -> Rotation Matrix

glm::yawPitchRoll(yaw, pitch, roll) ;

float yaw, // by y-axis (in radians)

float pitch, // by x-axis (in radians)

float roll // by z-axis (in radians)



glm::rotate

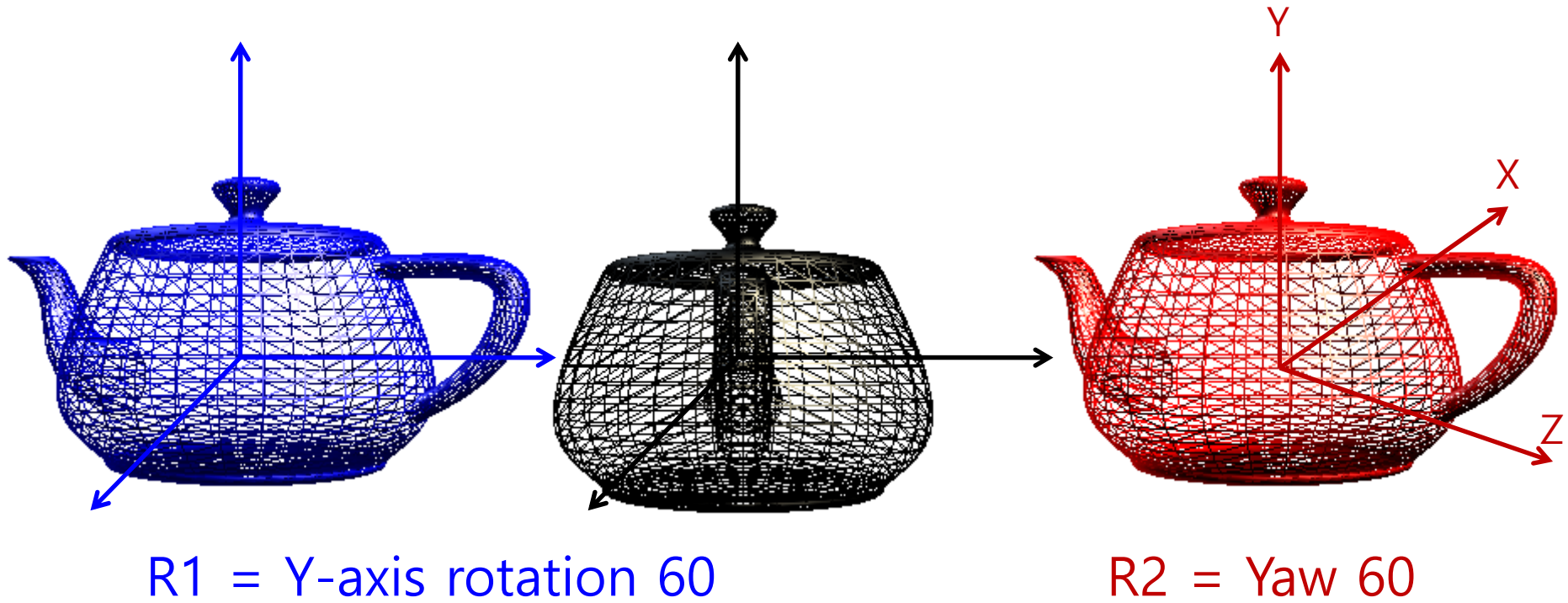
- glm::yawPitchRoll vs. glm::rotate (X/Y/Z)
 - YawPitchRoll – rotations in local coordinate system
 - Rotate (X/Y/Z) multiplication – rotations in world coordinate system

```
glm::mat4 R1, R2, Rx, Ry, Rz;  
Ry = glm::rotate(glm::mat4(1), 60, glm::vec3(0, 1, 0));  
Rx = glm::rotate(glm::mat4(1), 30, glm::vec3(1, 0, 0));  
Rz = glm::rotate(glm::mat4(1), 45, glm::vec3(0, 0, 1));  
R1 = Rz * Rx * Ry;
```

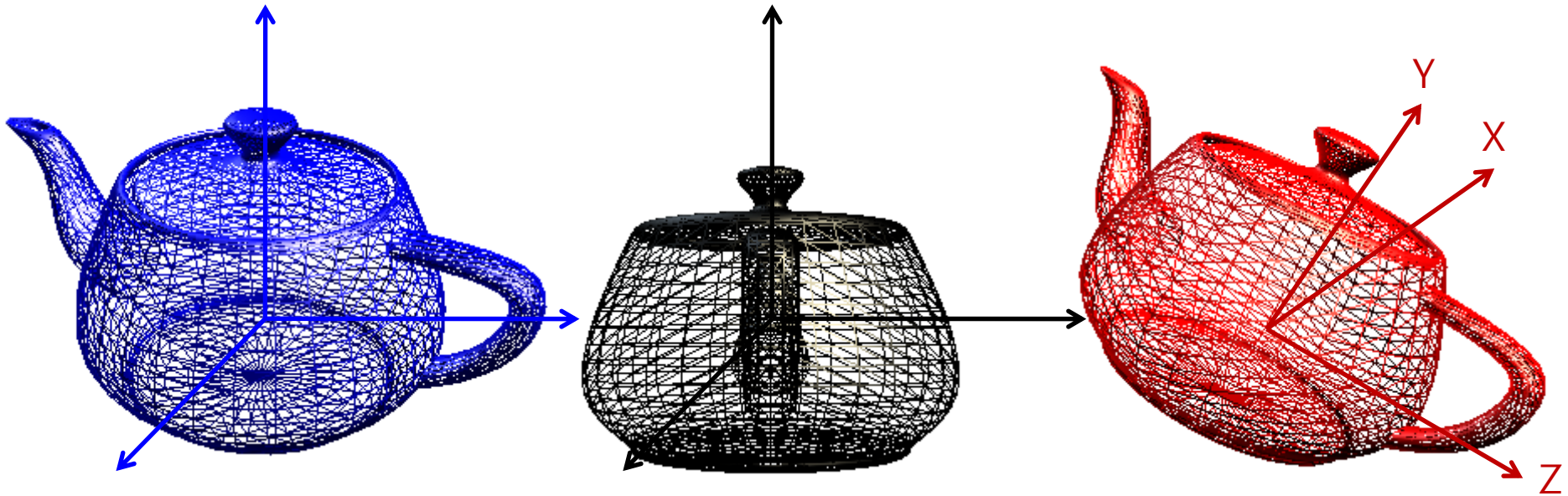
```
R2 = glm::yawPitchRoll(60, 30, 45);
```

R1 != R2

YawPitchRoll vs. RotationX/Y/Z



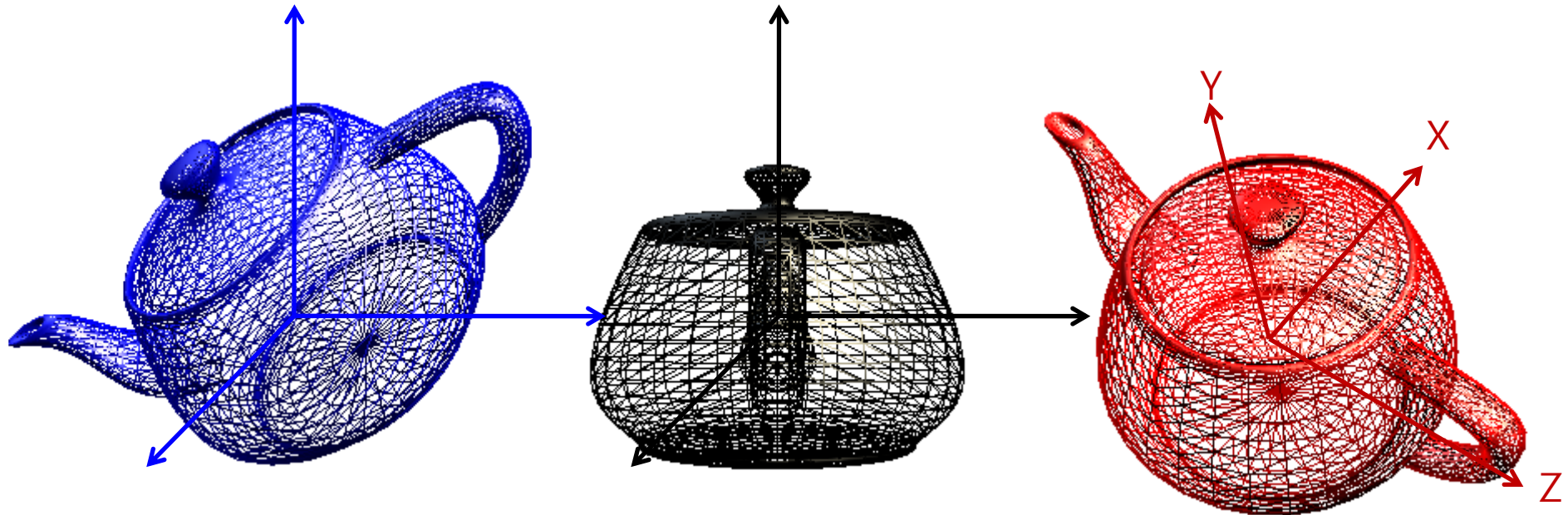
YawPitchRoll vs. RotationX/Y/Z



R1 = X-axis rotation 30
* Y-axis rotation 60

R2 = Yaw 60
Pitch 30

YawPitchRoll vs. RotationX/Y/Z



R1 = Z-axis rotation 45
* X-axis rotation 30
* Y-axis rotation 60

R2 = Yaw 60
Pitch 30
Roll 45

Rotation Vectors and Axis/Angle

- Euler's Theorem also shows that any two orientations can be related by a single rotation about some axis (not necessarily a principle axis).
- This means that we can represent an arbitrary orientation as a rotation about some unit axis by some angle (4 numbers) (Axis/Angle form).
- Alternately, we can scale the axis by the angle and compact it down to a single 3D vector (Rotation vector).

Axis/Angle to Matrix

- To generate a matrix as a rotation θ around an arbitrary unit axis \mathbf{a} :

$$\begin{bmatrix} a_x^2 + \cos \theta(1 - a_x^2) & a_x a_y(1 - \cos \theta) + a_z \sin \theta & a_x a_z(1 - \cos \theta) - a_y \sin \theta \\ a_x a_y(1 - \cos \theta) - a_z \sin \theta & a_y^2 + \cos \theta(1 - a_y^2) & a_y a_z(1 - \cos \theta) + a_x \sin \theta \\ a_x a_z(1 - \cos \theta) + a_y \sin \theta & a_y a_z(1 - \cos \theta) - a_x \sin \theta & a_z^2 + \cos \theta(1 - a_z^2) \end{bmatrix}$$

```
glm::vec3 axis(0, 1, 0);  
float angle = glm::radians(60);  
glm::mat4 R = glm::rotate(glm::mat4(1), angle, axis);
```

Rotation Vectors

- ❑ To convert a scaled rotation vector to a matrix, one would have to extract the magnitude out of it and then rotate around the normalized axis
- ❑ Normally, rotation vector format is more useful for representing angular velocities and angular accelerations, rather than angular position (orientation)

Axis/Angle Representation

- ❑ Storing an orientation as an axis and an angle uses 4 numbers, but Euler's theorem says that we only need 3 numbers to represent an orientation
- ❑ Mathematically, this means that we are using 4 degrees of freedom to represent a 3 degrees of freedom value
- ❑ This implies that there is possibly extra or redundant information in the axis/angle format
- ❑ The redundancy manifests itself in the magnitude of the axis vector. The magnitude carries no information, and so it is redundant. To remove the redundancy, we choose to normalize the axis, thus *constraining* the extra degree of freedom

Matrix Representation

- We can use a 3x3 matrix to represent an orientation as well.
- This means we now have 9 numbers instead of 3, and therefore, we have 6 extra degrees of freedom.
- NOTE: We don't use 4x4 matrices here, as those are mainly useful because they give us the ability to combine translations. We will just think of 3x3 matrices.

Matrix Representation

- Those extra 6 DOFs manifest themselves as 3 scales (x, y, and z) and 3 shears (xy, xz, and yz)
- If we assume the matrix represents a *rigid* transform (orthonormal), then we can constrain the extra 6 DOFs

$$|\mathbf{a}| = |\mathbf{b}| = |\mathbf{c}| = 1$$

$$\mathbf{a} = \mathbf{b} \times \mathbf{c}$$

$$\mathbf{b} = \mathbf{c} \times \mathbf{a}$$

$$\mathbf{c} = \mathbf{a} \times \mathbf{b}$$

Matrix Representation

- Matrices are usually the most computationally efficient way to apply rotations to geometric data, and so most orientation representations ultimately need to be converted into a matrix in order to do anything useful.
- Why then, shouldn't we just always use matrices?
 - Numerical issues
 - Storage issues
 - User interaction issues
 - Interpolation issues

Quaternions

- ❑ Quaternions are an interesting mathematical concept with a deep relationship with the foundations of algebra and number theory
- ❑ Invented by W.R.Hamilton in 1843
- ❑ In practice, they are most useful as a means of representing orientations
- ❑ A quaternion has 4 components

$$\mathbf{q} = \langle x \quad y \quad z \quad w \rangle$$

Quaternions (Imaginary Space)

- Quaternions are actually an extension to complex numbers.
- Of the 4 components, one is a 'real' scalar number, and the other 3 form a vector in imaginary ijk space!

$$\mathbf{q} = xi + yj + zk + w$$

$$i^2 = j^2 = k^2 = ijk = -1$$

$$i = jk = -kj$$

$$j = ki = -ik$$

$$k = ij = -ji$$

Quaternion (Scalar/Vector)

- Quaternions are written as the combination of a scalar value s and a vector value \mathbf{v} , where

$$\mathbf{q} = \langle \mathbf{v}, s \rangle$$

$$\mathbf{v} = (x, y, z)$$

$$s = w$$

Identity Quaternion

- Unlike vectors, there are two identity quaternions.
- The multiplication identity quaternion is

$$\mathbf{q} = \langle 0, 0, 0, 1 \rangle = 0i + 0j + 0k + 1$$

- The addition identity quaternion (which we do not use) is

$$\mathbf{q} = \langle 0, 0, 0, 0 \rangle$$

Unit Quaternion

- For convenience, we will use only unit length quaternions, as they will make things a little easier

$$|\mathbf{q}| = \sqrt{x^2 + y^2 + z^2 + w^2} = 1$$

- These correspond to the set of vectors that form the 'surface' of a 4D hyper-sphere of radius 1
- The 'surface' is actually a 3D volume in 4D space, but it can sometimes be visualized as an extension to the concept of a 2D surface on a 3D sphere
- Quaternion normalization:

$$q = \frac{q}{|\mathbf{q}|} = \frac{q}{\sqrt{x^2 + y^2 + z^2 + w^2}}$$

Quaternion as Rotations

- A quaternion can represent a rotation by an angle θ around a unit axis \mathbf{a} (a_x, a_y, a_z) :

$$\mathbf{q} = \left[a_x \sin \frac{\theta}{2}, a_y \sin \frac{\theta}{2}, a_z \sin \frac{\theta}{2}, \cos \frac{\theta}{2} \right]$$

or

$$\mathbf{q} = \left[\mathbf{a} \sin \frac{\theta}{2}, \cos \frac{\theta}{2} \right]$$

- If \mathbf{a} has unit length, then \mathbf{q} will also has unit length

Quaternions as Rotations

$$\begin{aligned} |\mathbf{q}| &= \sqrt{x^2 + y^2 + z^2 + w^2} \\ &= \sqrt{a_x^2 \sin^2 \frac{\theta}{2} + a_y^2 \sin^2 \frac{\theta}{2} + a_z^2 \sin^2 \frac{\theta}{2} + \cos^2 \frac{\theta}{2}} \\ &= \sqrt{\sin^2 \frac{\theta}{2} (a_x^2 + a_y^2 + a_z^2) + \cos^2 \frac{\theta}{2}} \\ &= \sqrt{\sin^2 \frac{\theta}{2} |\mathbf{a}|^2 + \cos^2 \frac{\theta}{2}} = \sqrt{\sin^2 \frac{\theta}{2} + \cos^2 \frac{\theta}{2}} \\ &= \sqrt{1} = 1 \end{aligned}$$

Quaternion to Rotation Matrix

- Equivalent rotation matrix representing a quaternion is:

$$\begin{bmatrix} x^2 - y^2 - z^2 + w^2 & 2xy - 2wz & 2xz + 2wy \\ 2xy + 2wz & -x^2 + y^2 - z^2 + w^2 & 2yz - 2wx \\ 2xz - 2wy & 2yz + 2wx & -x^2 - y^2 + z^2 + w^2 \end{bmatrix}$$

- Using unit quaternion that $x^2 + y^2 + z^2 + w^2 = 1$, we can reduce the matrix to:

$$\begin{bmatrix} 1 - 2y^2 - 2z^2 & 2xy - 2wz & 2xz + 2wy \\ 2xy + 2wz & 1 - 2x^2 - 2z^2 & 2yz - 2wx \\ 2xz - 2wy & 2yz + 2wx & 1 - 2x^2 - 2y^2 \end{bmatrix}$$

Quaternion to Axis/Angle

- To convert a quaternions to a rotation axis, a (ax, ay, az) and an angle θ :

$$scale = \sqrt{x^2 + y^2 + z^2} \quad or \quad \sin(\mathbf{acos}(w))$$

$$ax = x / scale$$

$$ay = y / scale$$

$$az = z / scale$$

$$\theta = 2\mathbf{acos}(w)$$

Matrix to Quaternion

- To convert a matrix to a quaternion:

$$w = \frac{\sqrt{m_{11} + m_{22} + m_{33} + 1}}{2}$$

$$x = \frac{m_{23} - m_{32}}{4w} \quad y = \frac{m_{31} - m_{13}}{4w} \quad z = \frac{m_{12} - m_{21}}{4w}$$

- If $w=0$, then the division is undefined. First, determining which q_0, q_1, q_2, q_3 is the largest, computing that component using the diagonal of the matrix.

Quaternion Dot Product

- The dot product of two quaternions works in the same way as the dot product of two vectors:

$$\mathbf{p} \cdot \mathbf{q} = x_p x_q + y_p y_q + z_p z_q + w_p w_q = |\mathbf{p}| |\mathbf{q}| \cos \varphi$$

- The angle between two quaternions in 4D space is half the angle one would need to rotate from one orientation to the other in 3D space.

Quaternion Multiplication

- If \mathbf{q} represents a rotation and \mathbf{q}' represents a rotation, then $\mathbf{q}\mathbf{q}'$ represents \mathbf{q} rotated by \mathbf{q}'
- This follows very similar rules as matrix multiplication (i.e., non-commutative) $\mathbf{q}\mathbf{q}' \neq \mathbf{q}'\mathbf{q}$

$$\begin{aligned}\mathbf{q}\mathbf{q}' &= (xi + yj + zk + w)(x'i + y'j + z'k + w') \\ &= \langle s\mathbf{v}' + s'\mathbf{v} + \mathbf{v}' \times \mathbf{v}, ss' - \mathbf{v} \cdot \mathbf{v}' \rangle\end{aligned}$$

Quaternion Multiplication

- Note that two unit quaternions multiplied together will result in another unit quaternion
- This corresponds to the same property of complex numbers
- Remember that multiplication by complex numbers can be thought of as a rotation in the complex plane
- Quaternions extend the planar rotations of complex numbers to 3D rotations in space

Quaternion Operations

- Negation of quaternion, $-q$
 - $-[v \ s] = [-v \ -s] = [-x, -y, -z, -w]$
- Addition of two quaternion, $p + q$
 - $p + q = [pv, ps] + [qv, qs] = [pv + qv, ps + qs]$
- Magnitude of quaternion, $|q|$
 - $|q| = \sqrt{x^2 + y^2 + z^2 + w^2}$
- Conjugate of quaternion, q^* (켈레 사원수)
 - $q^* = [v \ s]^* = [-v \ s] = [-x, -y, -z, w]$
- Multiplicative inverse of quaternion, q^{-1} (역수) $q q^{-1} = q^{-1} q = 1$
 - $q^{-1} = q^*/|q|$
- Exponential of quaternion
 - $\exp(v \ q) = v \sin q + \cos q$
- Logarithm of quaternion $q = [v \sin q, \cos q]$
 - $\log(q) = \log(v \sin q + \cos q) = \log(\exp(v \ q)) = v \ q$

glm::quaternion

- // rotation matrix (4x4) -> quaternion

```
glm::quat quat1 = glm::quat_cast(matrix1);
```

- // axis/angle -> quaternion

```
glm::quat quat1 = glm::axisAngle((float)M_PI/2.0, glm::vec3(0, 1, 0));
```

- // quaternion -> Euler angle (yaw/pitch/roll)

```
glm::vec3 euler = glm::eulerAngles(quat1); // XYZ
```

```
float yaw = glm::yaw(quat1); // Y
```

```
float pitch = glm::pitch(quat1); // X
```

```
float roll = glm::roll(quat1); // Z
```

- // quaternion -> rotation matrix (4x4)

```
glm::mat4 R = glm::mat4_cast(quat1);
```

- // rotate a vector by a quaternion

```
glm::vec4 vec = glm::rotate(quat1, glm::vec4(1, 2, 3, 1));
```

- // rotate a quaternion by axis/angle

```
glm::quat q = glm::rotate(quat1, (float)M_PI/2.0, glm::vec3(0, 1, 0));
```

Quaternion Interpolation

- One of the key benefits of using a quaternion representation is the ability to interpolate between key frames.

alpha = fraction value in between frame0 and frame1

q1 = Euler2Quaternion(frame0)

q2 = Euler2Quaternion(frame1)

qr = QuaternionInterpolation(q1, q2, alpha)

qr.Quaternion2Euler()

- Quaternion Interpolation
 - Linear Interpolation (LERP)
 - Spherical Linear Interpolation (SLERP)
 - Spherical Cubic Interpolation (SQUAD)

Linear Interpolation (LERP)

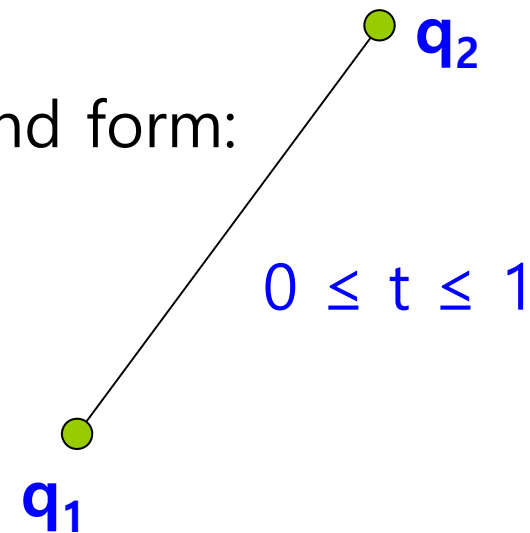
- If we want to do a direct interpolation between two quaternions \mathbf{p} and \mathbf{q} by alpha:

$$\text{Lerp}(\mathbf{p}, \mathbf{q}, t) = (1-t)\mathbf{p} + (t)\mathbf{q}$$

where $0 \leq t \leq 1$

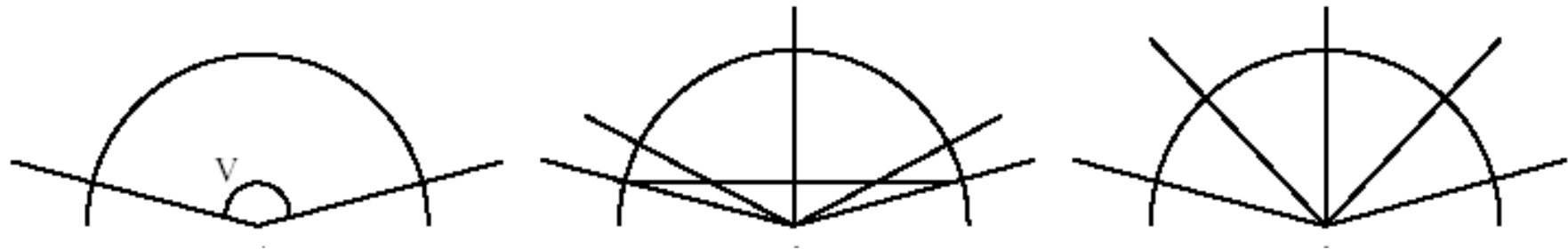
- Note that the Lerp operation can be thought of as a weighted average (convex)
- We could also write it in it's additive blend form:

$$\text{Lerp}(\mathbf{q}_1, \mathbf{q}_2, t) = \mathbf{q}_1 + t(\mathbf{q}_2 - \mathbf{q}_1)$$



Why SLERP?

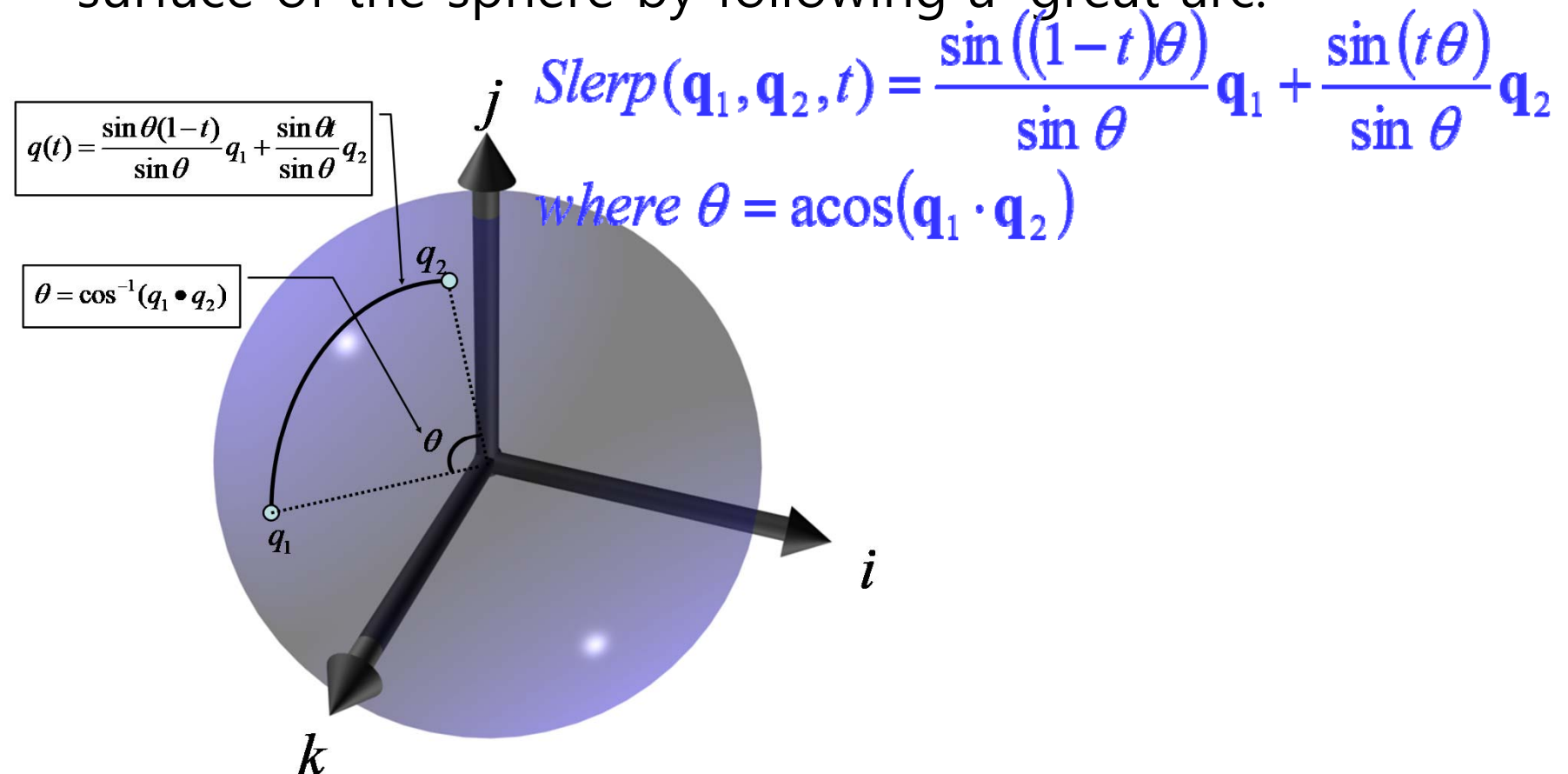
- The set of quaternions live on the unit hypersphere. The direct interpolation between quaternions would stray from the hypersphere.



- An illustration in the plane of the difference between Lerp and Slerp
 - The interpolation covers the angle v in three steps
 - [Lerp] The secant across is split in four equal pieces The corresponding angles are shown
 - [Slerp] The angle is split in four equal angles

Spherical Linear Interpolation (SLERP)

- If we want to interpolate between two points on a sphere (or hypersphere), we will travel across the surface of the sphere by following a 'great arc.'

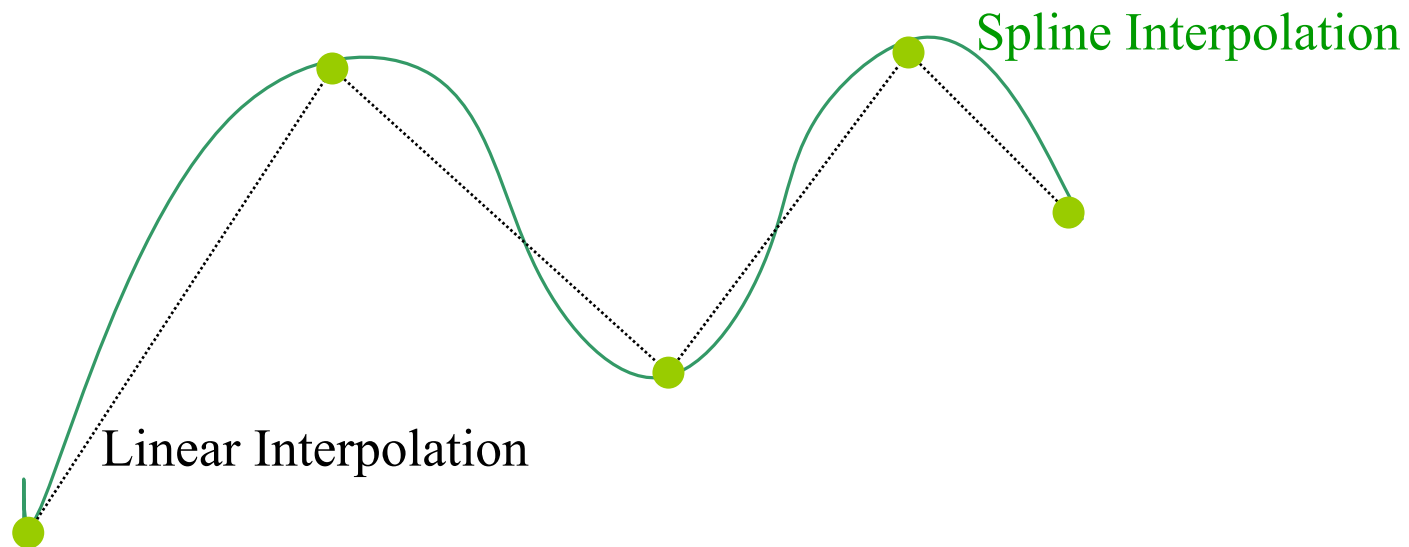


Spherical Linear Interpolation

- Remember that there are two redundant vectors in quaternion space for every unique orientation in 3D space
- What is the difference between:
 $\text{Slerp}(\mathbf{p}, \mathbf{q}, t)$ and $\text{Slerp}(-\mathbf{p}, \mathbf{q}, t)$?
 - One of these will travel less than 90 degrees while the other will travel more than 90 degrees across the sphere
 - This corresponds to rotating the 'short way' or the 'long way'
 - Usually, we want to take the short way, so we negate one of them if their dot product is < 0

Why SQUAD?

- ❑ Slerp produces smooth interpolation, but it always follows a great arc connecting two quaternions – i.e. the animations change directions abruptly at the control points. To smoothly interpolate through a series of quaternions, use splines.



Spherical Cubic Interpolation (SQUAD)

- To achieve C^2 continuity between curve segments, a cubic interpolation must be done.
- Squad does a cubic interpolation between four quaternions by t

$$\text{Squad}(q_i, q_{i+1}, a_i, a_{i+1}, t)$$

$$= \text{slerp}(\text{slerp}(q_i, q_{i+1}, t), \text{slerp}(a_i, a_{i+1}, t), 2t(1-t))$$

$$a_i = q_i * \exp\left(\frac{-\log(q_i^{-1} * q_{i-1}) + \log(q_i^{-1} * q_{i+1})}{4}\right)$$

$$a_{i+1} = q_{i+1} * \exp\left(\frac{-\log(q_{i+1}^{-1} * q_i) + \log(q_{i+1}^{-1} * q_{i+2})}{4}\right)$$

- a_i, a_{i+1} are inner quadrangle quaternions between q_1 and q_2 . And you have to choose carefully so that continuity is guaranteed across segments.

glm::quaternion

- // slerp(q_1, q_2, t) spherical linear interpolation between two quaternions, q_1, q_2 according to t

```
glm::quat quat3 = glm::mix(quat1, quat2, alpha);
```

- // squad(q_1, q_2, s_1, s_2, t) spherical cubic interpolation

```
glm::quat quat3 = glm::squad(quat1, quat2, s1, s2, alpha);
```

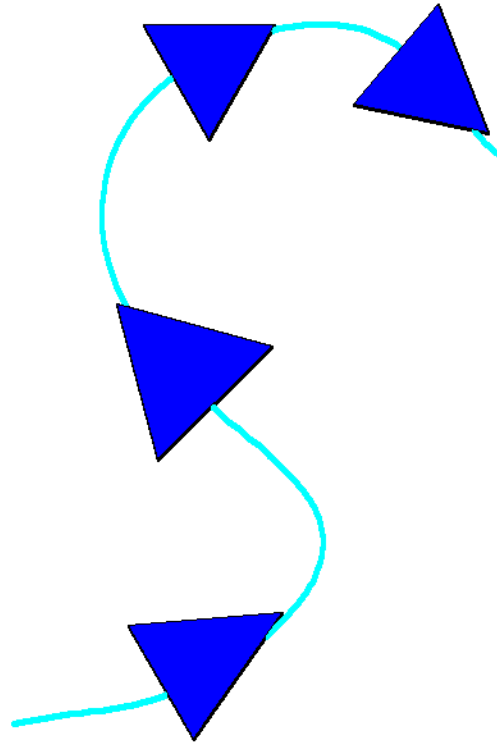
Catmull-Rom Spline Interpolation

- Given $n+1$ control points $\{P_0, P_1, \dots, P_n\}$, you wish to find a curve that interpolates these control points (and passes through them all), and is local in nature (i.e. if one of the control points is moved, it only affects the curve locally) – Catmull-Rom Spline.
- The Catmull-Rom Spline takes a set of keyframe points to describe a smooth piecewise cubic curve that passes through all the points. In order to use this routine we need four keyframe points.
- Given four keyframe points, P_0, P_1, P_2, P_3 , the curve passes through P_1 at $t=0$ and it passes through P_2 at $t=1$ ($0 < t < 1$).
- The tangent vector at a point P is parallel to the line joining P 's two surrounding points.

Catmull-Rom Spline Interpolation

- // Catmull Rom Spline Interpolation
glm::vec3 position = glm::catmullRom(vec1, vec2, vec3,
vec4, alpha);

Path Animation



Path Controlled Translation & Rotation