# Representing Orientations

514780
2018년 가을학기
10/18/2018
단국대학교 박경신

## 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

- 3D orientation 표현 방식은 여러가지가 존재한다.
  - Euler angles (오일러각) – 가장 간단한 방법
  - Rotation vectors (axis/angle)
  - Rotation matrices (회전행렬)
  - Quaternions (사원수)

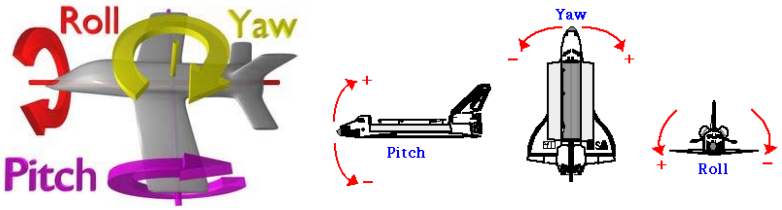
## Euler Angles

- Euler Angles
  - 3차원 공간의 X, Y, Z 축에 대한 회전각을 지정한 순서대로 곱해서 사용하여 arbitrary orientation을 나타낸다.
  - 2차원 평면에서 물체의 방향을 지정해주려면 단지 하나의 각도면 충분하지만 3차원에서는 최소한 3개의 각도 값이 필요하다.
  - X,Y,Z축에 대한 $\theta_x$, $\theta_y$, $\theta_z$ 회전을 ***Euler Angle Sequence***라 부른다.
- Axis order
  - 오일러각은 특정 회전축의 조합 하나를 말하는 것이 아니라 3차원 공간에서 임의의 방향을 나타낼 수 있는 조합을 모두 가리킨다.
  - (y, x, z), (x, y, z), (z, x, y), … 12가지 모두 사용 가능하다.

| | | | |
|---|---|---|---|
| XYZ | XZY | XYX | XZX |
| YXZ | YZX | YXY | YZY |
| ZXY | ZYX | ZXZ | ZYZ |

## Euler Angles

- 오일러각은 Yaw, Pitch, Roll로도 표현한다.
- Yaw (rotation about Y), Pitch (X), Roll (Z) 회전조합으로 OpenGL/DirectX 등에서 사용한다.




## Euler Angles to Matrix Conversion

- 3D orientation을 표현하기 위해 오일러각 회전행렬의 곱으로 회전행렬을 만들어낸다.

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

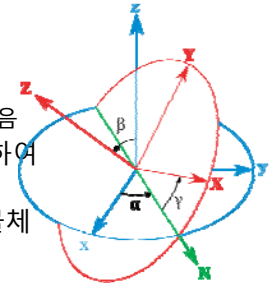
$$= \begin{bmatrix} c_y c_z & c_y s_z & -s_y \\ s_x s_y c_z - c_x s_z & s_x s_y s_z + c_x c_z & s_x c_y \\ c_x s_y c_z + s_x s_z & c_x s_y s_z - s_x c_z & c_x c_y \end{bmatrix}$$

## Euler Angle Order

- 행렬의 곱셈은 교환법칙이 성립하지 않는다. 즉, 순서가 중요하다.
- *12가지 X,Y,Z 회전 적용 순서 중에서 하나를 선택해서 사용해야한다.*
- 오일러각 회전조합의 순서는 응용프로그램에 따라 다를 수 있다.
  - 3차원 컴퓨터 그래픽스에서는 XYZ 회전조합을 많이 사용한다.
  - 강체물리역학에서는 ZXZ 회전조합을 많이 사용한다.

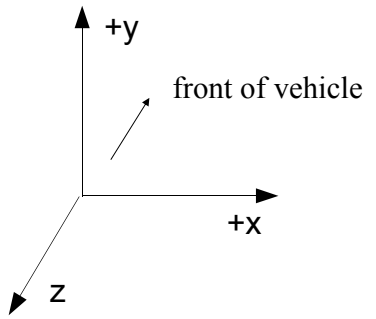
## Euler Angle Order

- ZXZ convention은 공간좌표계(space coordinates) XYZ에서
  - XYZ (fixed) system is shown in blue.
  - XYZ (rotated) system is shown in red.
  - The line of nodes, N, is shown in green.
  - (Z-회전) Z축을 회전축으로 α 만큼 X-Y 좌표축을 회전
    - X-축이 N과 일치
  - (X-회전) 회전된 좌표축 X-축(이제 N-축)을 회전축으로 하여 β만큼 Z-Y 좌표축을 회전
    - Z-축이 최종 회전으로 가능, X-축은 N으로 남음
  - (Z-회전) 새로운 Z-축으로 다시 회전축으로 하여 γ만큼 X-Y 좌표축을 회전
  - 이와 같이 ZXZ 순서로 차례로 회전시키면, 물체 좌표계(object coordinates) XYZ를 얻는다



## Vehicle Orientation Using Euler Angles

- 일반적으로 Vehicle 움직임을 표현하는 경우, yaw (y), pitch (x), roll (z) 순서를 사용한다.
- 지면 위를 돌아다니는 자동차의 경우, yaw (y), pitch (x) 순서를 사용한다.

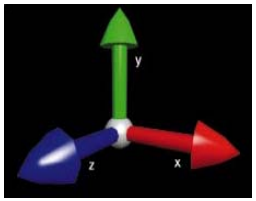


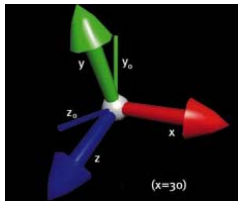

## Rotations not uniquely defined with Euler Angles

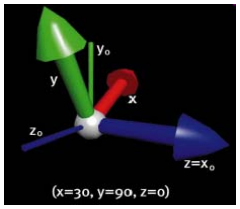
- 그러나, 문제는 수학적인 견지에서 볼 때 이러한 임의의 3축 각도 값이 서로 독립적이지 않다는 점이다.
- Cartesian coordinates는 서로 독립적이다. 즉, 임의의 점 위치는 X, Y, Z 축 위치로 표현 가능하다.
  - Arbitrary position = x-axis position + y-axis position + z-axis position
- 그러나, Euler angles은 독립적이지 않다.
  - Arbitrary orientation = x-axis rotation matrix * y-axis rotation matrix * z-axis rotation matrix
  - 예를 들어, (z, x, y) = (90, 45, 45) = (45, 0, -45)
  - 임의의 방향을 설정할 때 3축의 회전을 조합해야 하는데 이것이 직관적으로 되질 않고, 현재의 특정 방향에서 다음 방향으로 바꾸려고 할 때 각 회전 값을 얼만큼 변화시켜야 하는지 애매하다.

## Gimbal Lock

- Euler angles은 '**gimbal lock**' 문제를 갖는다.
- '**Gimbal Lock**' 이란, 같은 방향으로 객체의 두 회전 축이 겹치는 현상을 말한다 (즉, losing a degree of freedom under certain rotations). 그래서 한 축에 대한 회전이 다른 축에도 영향을 미치게 되는 현상이다.
  - XYZ 회전순서를 사용하는 경우, Y축으로 90 회전한 순간부터 **3개의 회전 축 중 하나가 사라지게 되는 상황이다**.
  - 왜냐하면 X 성분이 이미 평가가 됐기 때문에 다른 두 축으로 이동되지 않고, X와 Z축이 서로 같은 축을 향해 가리키게 됨으로써, 이 후 Z축 회전이 X축 회전에 영향을 주게 되는 현상이다.

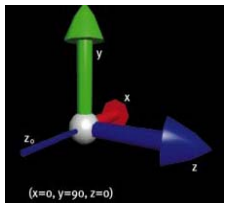


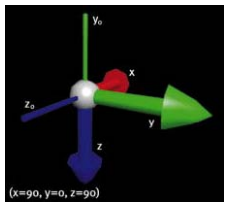




## Problem with Interpolating Euler Angles

- Euler angles은 3 축에 대해 서로 독립적이지 않기 때문에 2개의 오일러각 간의 각도 값을 보간(interpolate)할 때에도 문제가 생긴다.
  - Euler angle (0,180,0) = (180, 0, 180)
  - 따라서, (0, 0, 0) -> (0, 180, 0)로 보간하는 경우와 (0, 0, 0) -> (180, 0, 180)으로 보간하는 경우 중간값은 (0, 90, 0)과 (90, 0, 90)으로 완전히 다른 방향으로 회전하게 된다.




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

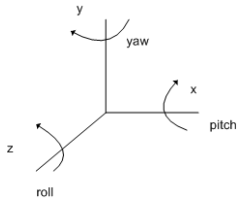



Halfway between (0,0, 0) and (180,0, 180)

## 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

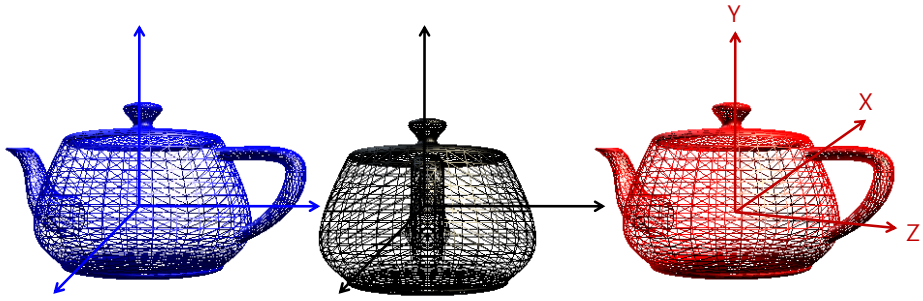
- glm::yawPitchRoll vs. glm::rotate (X/Y/Z)
  - YawPitchRoll – rotations in local coordinate system
  - Rotate (X/Y/Z) multiplication – rotations in world coordinate system

```
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

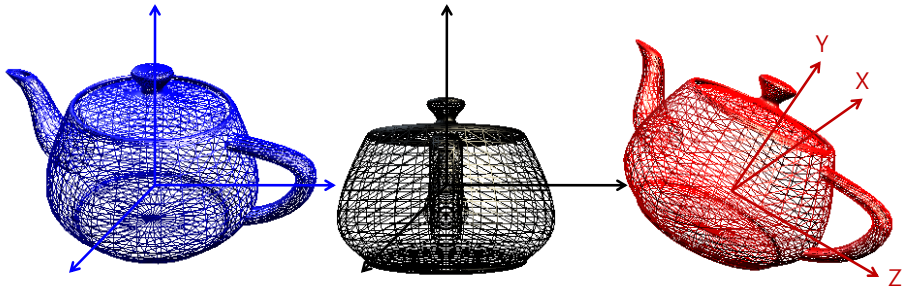
## YawPitchRoll vs. RotationX/Y/Z




R1 = Y-axis rotation 60

R2 = Yaw 60

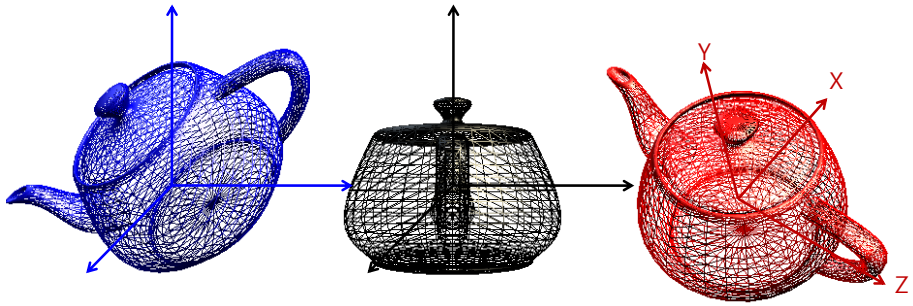
## YawPitchRoll vs. RotationX/Y/Z




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

R2 = Yaw 60
Pitch 30

## YawPitchRoll vs. RotationX/Y/Z




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

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 **a**:

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

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

## Quaternions

- Quaternions (사원수)란 3차원 컴퓨터 그래픽스에서 회전을 표현할 때 사용되는 수학적 개념으로 4차원 복소수 공간 (complex space) 벡터이다.
- 실제로 회전 표현하는데 있어서 가장 효과적인 방법이다.
- Quaternion (사원수) 표현

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

## Quaternions (Imaginary Space)

- 사원수는 실제로 복소수(complex numbers)의 확장이다.
- 4개의 구성 요소 중에 하나는 실수 (scalar number)이고 다른 세 개는 허수의 공간 I, j, k에 있는 복소수이다.

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

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

$$i = jk = -kj$$

$$j = ki = -ik$$

$$k = ij = -ji$$

## Quaternion (Scalar/Vector)

- 사원수는 또한 스칼라 값 s와 벡터 값 v 로 표현된다.

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

$$v = (x, y, z)$$

$$s = w$$

## Identity Quaternion

- 벡터와는 달리 2개의 항등 사원수 (Identity quaternion)가 있다.
- 곱셈 항등 사원수 (multiplication identity quaternion) – 그래서 이 곱셈 항등 사원수와 곱해진 어떤 사원수도 변하지 않는다:

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

- 덧셈 단위 사원수 (addition identity quaternion) – 여기서는 사용하지 않는다:

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

## Unit Quaternion

- 사원수 연산의 편리함을 위하여 단위 사원수 (unit length quaternion)을 사용한다.
- 단위 사원수 (unit length quaternion)는 사원수의 크기가 1이다. 이것은 4차원 공간에서 단위 길이를 가지는 구 (hypersphere)의 surface (즉, 4차원 공간에서의 3차원 부피)를 형성하는 벡터를 이룬다.

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

- 사원수의 정규화 (normalization)은 아래와 같이 구한다.

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

## Quaternion as Rotations

- 사원수는 벡터의 회전과 밀접한 관계가 있는데 회전축 (axis **a**) 와 각도 (angle θ)로 나타낼 수 있다.

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

*or*

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

- 회전축 **a** 가 단위길이를 갖는다면, 사원수 **q**도 마찬가지로 단위길이를 갖는다.

## Quaternions as Rotations

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

$$= \sqrt{a_x^2 \sin^2\frac{\theta}{2} + a_y^2 \sin^2\frac{\theta}{2} + a_z^2 \sin^2\frac{\theta}{2} + \cos^2\frac{\theta}{2}}$$

$$= \sqrt{\sin^2\frac{\theta}{2}\left(a_x^2 + a_y^2 + a_z^2\right) + \cos^2\frac{\theta}{2}}$$

$$= \sqrt{\sin^2\frac{\theta}{2}|\mathbf{a}|^2 + \cos^2\frac{\theta}{2}} = \sqrt{\sin^2\frac{\theta}{2} + \cos^2\frac{\theta}{2}}$$

$$= \sqrt{1} = 1$$

## Quaternion to Rotation Matrix

- 최종적으로 얻어진 사원수를 실제 프로그램에서 회전에 사용하기 위해서는 다음과 같은 행렬로 변환:

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

- 단위 사원수가 $x^2+y^2+z^2+w^2=1$를 갖는 점을 이용하여, 회전행렬을 좀더 줄이면:

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

## Quaternion to Axis/Angle

- 사원수를 3차원 공간에서의 임의 회전축 a ($a_x$, $a_y$, $a_z$)과 각도(θ)에 의한 표현으로 변환:

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

$$ax = x / scale$$

$$ay = y / scale$$

$$az = z / scale$$

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

## Matrix to Quaternion

- 행렬에서 사원수로 변환:

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

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

- 만약 w=0이면, 나눗셈 연산이 이루어질 수 없다. First, determining which q0, q1,q2, q3 is the largest, computing that component using the diagonal of the matrix.

## Quaternion Dot Product

- 두 개의 사원수 간의 내적 (dot product)은 두 개의 벡터 간의 내적과 같은 방식으로 계산하면 된다.

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

## Quaternion Multiplication

- 단위 사원수는 3차원 공간에서의 한 방향을 표현하기 때문에, 두 개의 단위 사원수 간의 곱은 두 개의 단위 회전을 결합한 회전을 나타내는 단위 사원수가 된다.
- 사원수의 곱은 순서가 중요하다. 사원수의 곱은 교환법칙이 성립되지 않는다. qq′ ≠ q′q

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

## 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|
  - $|\mathbf{q}| = \sqrt{x^2 + y^2 + z^2 + w^2}$
- Conjugate of quaternion, q* (켤레 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, flm::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

- 사원수는 키 프레임 (key frames)간에 회전 보간 (interpolation)을 가장 효과적으로 표현할 수 있다.

```
alpha = fraction value in between frame0 and frame1
q1 = Euler2Quaternion(frame0)
q2 = Euler2Quaternion(frame1)
qr = QuaternionInterpolation(q1, q2, alpha)
qr.Quaternion2Euler()
```

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

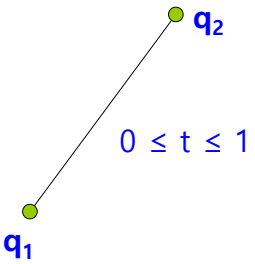
## Linear Interpolation (LERP)

- 가장 쉬운 방식으로 두 개의 사원수간의 선형보간 (linear interpolation) 방식이 있다.

Lerp($\mathbf{q_1}$, $\mathbf{q_2}$, t) = (1-t) $\mathbf{q_1}$ + (t) $\mathbf{q_2}$
where 0 ≤ t ≤ 1

- 선형보간 공식의 또 다른 표현:

Lerp($\mathbf{q_1}$, $\mathbf{q_2}$, t) = $\mathbf{q_1}$ + t($\mathbf{q_2}$ − $\mathbf{q_1}$)

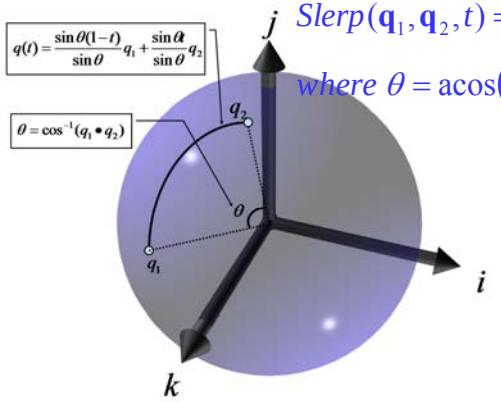



## Spherical Linear Interpolation (SLERP)

- 구면 선형 보간 (spherical linear interpolation)은 벡터 $q_1$가 길이를 유지한 채로 회전해서 $q_2$가 되었다고 했을 때 회전한 그 사이 값을 보간하는 방법이다.

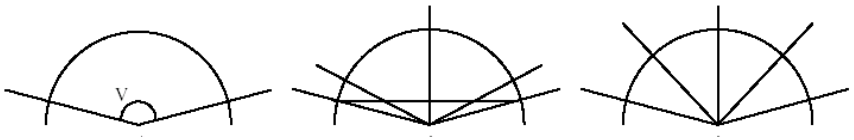
$$Slerp(\mathbf{q}_1, \mathbf{q}_2, t) = \frac{\sin((1-t)\theta)}{\sin\theta}\mathbf{q}_1 + \frac{\sin(t\theta)}{\sin\theta}\mathbf{q}_2$$

$$where\ \theta = \arccos(\mathbf{q}_1 \cdot \mathbf{q}_2)$$



## Why SLERP?

- 사원수의 공간의 구면 공간 (hypersphere)의 성격을 띠는데, 이를 선형 보간하게 되면 속도 오차가 생긴다.
- 즉, 선형 보간의 특성상 가운데 부분이 빨리 지나가게 된다. 그에 반해 구면 선형 보간은 일정한 속도를 유지한다.




- Lerp 과 Slerp의 차이점
  - 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

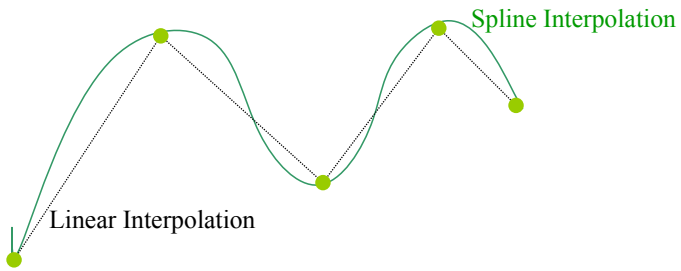
- Remember that there are two redundant vectors in quaternion space for every unique orientation in 3D space
- What is the difference between:

  Slerp(**p**, **q**, t) and Slerp(-**p**, **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?

- Slerp은 2개의 사원수 간의 보간을 해준다.
- 하지만, 여러 개의 사원수 간의 보간에서 Slerp을 사용하면 갑작스런 변화가 생긴다. 따라서, 여러 개의 사원수 간의 보간을 하기 위해서는 SQUAD 같은 spline interpolation을 사용해야 한다.




## Spherical Cubic Interpolation (SQUAD)

- 두 단위 사원수 $q_i$, $q_{i+1}$ 사이에 $a_i$, $a_{i+1}$이라는 사원수를 도입한다. 구면 입방 보간 (spherical cubic interpolation)은 아래와 같이 정의한다.

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

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

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

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

- $a_i$ 들은 초기 방향들에서의 접선 방향 (tangent orientation)을 표시하는데 사용된다.

## 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);

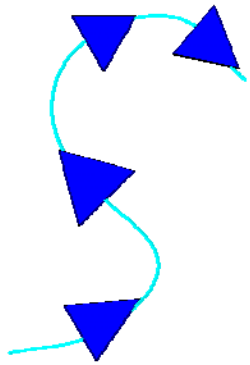
## Catmull-Rom Spline Interpolation

- Given n+1 control points {$P_0$, $P_1$, .. $P_n$}, 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**