Representing Orientations

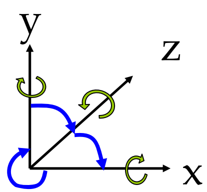
305890
2008년 봄학기
4/2/2007
박경신

Outline

- Orientation
- Euler Angles
- Rotation Matrix
- Quaternion

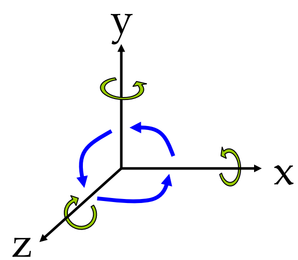
LHS Coordinate Systems

- Left Hand Coordinate System (LHS) – z+가 화면 안으로 들어가는 방향.
- Clockwise 방향으로 rotation을 함.
- X-축으로 rotation을 하면,
  Y->Z 방향의 회전이 positive
- Y-축으로 rotation을 하면,
  Z->X 방향의 회전이 positive
- Z-축으로 rotation을 하면,
  X->Y 방향의 회전이 positive



RHS Coordinate Systems

- Right Hand Coordinate System (RHS) – z+가 화면 앞으로 튀어나오는 방향.
- Counter clockwise 방향으로 rotation을 함.
- X-축으로 rotation을 하면,
  Y->Z 방향의 회전이 positive
- Y-축으로 rotation을 하면,
  Z->X 방향의 회전이 positive
- Z-축으로 rotation을 하면,
  X->Y 방향의 회전이 positive



Representing Orientations

- 3차원 회전 표현하는 방식은 여러가지가 존재한다.
- Euler angles (오일러각 방식)– 가장 간단한 방법
- Rotation vectors (axis/angle 각축 방식)
- Rotation matrices (회전행렬 방식)
- Quaternions (사원수 방식)
- 그 외에 다수..

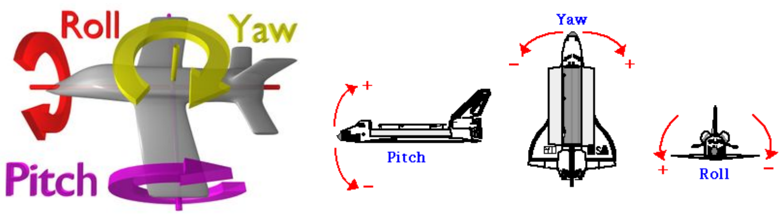
Euler Angles

- Euler Angles
  - 3차원 직교 좌표계의 좌표축인 x, y, z축마다 축에 대한 회전각을 지정한 순서대로 곱해서 사용하면 임의의 방향을 나타낼 수 있다.
  - 2차원 평면에서 물체의 방향을 지정해주려면 단지 하나의 각도 값이면 충분하지만 3차원 공간에서는 최소한 3개의 각도값이 필요하다.
  - x, y, z축에 대해 θ_x, θ_y, θ_z 각도로 회전한다.
- Axis order는 중요하지 않음
  - 오일러 앵글은 특정 회전축의 조합 하나를 말하는 게 아니라 3차원 공간에서 임의의 방향을 나타낼 수 있는 조합이라면 다 오일러 앵글.
  - (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

- Yaw, Pitch, Roll로 표현
- DirectX/OpenGL에서는 Yaw (rotation about Y), Pitch (X), Roll (Z) 회전 조합을 사용한다.



Euler Angles to Matrix Conversion

- 오일러 앵글로부터 회전행렬을 만들기 위해 일정한 순서대로 회전행렬을 곱한다.

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

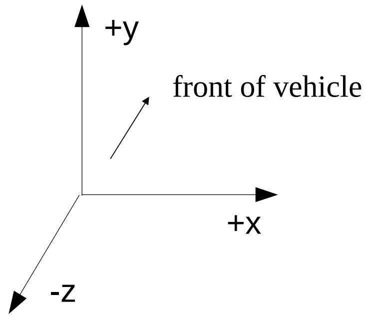
$$= \begin{bmatrix} c_y c_z & c_y s_z & -s_y \\ s_x s_y c_z - c_x s_z & s_x s_y s_z + c_x c_z & s_x c_y \\ c_x s_y c_z + s_x s_z & c_x s_y s_z - s_x c_z & c_x c_y \end{bmatrix}$$

Euler Angle Order

- 행렬 곱셈은 교한법칙이 성립하지 않는다. 즉, 순서가 중요하다.
- 12가지의 x, y, z 회전 적용순서 중에서 하나를 선택해야 한다.
- 가장 좋은 순서는 응용에 따라 다를 수 있다.
- 3차원 그래픽스의 경우 (x, y, z) 회전조합을 많이 사용한다.
- 물리의 강체역학에서는 (z, x, z) 회전조합을 많이 사용한다.

Vehicle Orientation Using Euler Angles

- 일반적으로 자동차의 경우, roll (z), pitch (x), yaw (y) 순서로 한다.
- 움직이는 물체가 땅 위를 돌아다니는 응용의 경우, Euler angles 방식이 가장 간단하고 직관적인 방법이다.

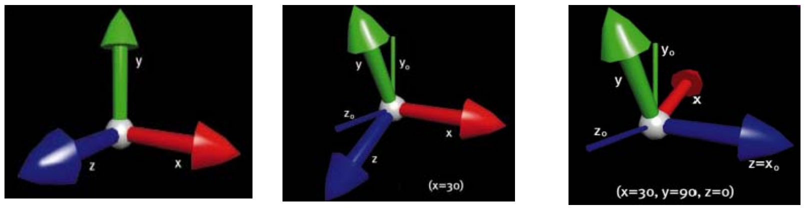


Rotations not uniquely defined with Euler Angles

- 그러나 문제는 수학적인 견지에서 볼 때 이러한 임의의 3 각도 값이 서로 독립적이지 않다는 것이다.
- Cartesian coordinates은 서로 완전히 독립적이다. 임의의 위치는 x축 위치성분 + y축 위치성분 + z축 위치성분
- 위치와는 달리 Euler angles은 독립적이지 않다. 임의의 방향 = x축 회전행렬 * y축 회전행렬 * z축 회전행렬
- 예를 들어, (z, x, y) = (90, 45, 45) = (45, 0, -45)
- 임의의 방향을 설정할 때 3 축의 회전을 조합해야 하는데 이것이 직관적으로 되질 않고, 현재의 특정 방향에서 다음 방향으로 바꾸려고 할 때 각 회전 값을 얼만큼 변화시키면 되는지 애매하다.

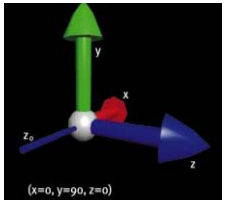
Gimbal Lock

- 오일러 앵글을 사용하는 시스템은 gimbal lock 문제를 갖는다.
- 'Gimbal Lock'은 같은 방향으로 객체의 두 회전 축이 겹치는 현상을 말한다. – losing a degree of freedom under certain rotations
  - 예를 들어, xyz 회전순서를 사용하는 경우, y가 90도로 회전한 순간부터 3개의 회전축 중 하나가 사라지게 되는 상황이다.
  - 왜냐하면 x성분이 이미 평가가 됐기 때문에 다른 두 축으로 이동되지 않고, x와 z축이 서로 같은 축을 향해 가리키게 된다.

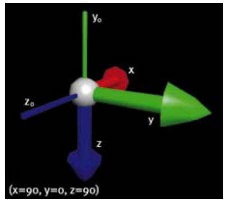


Problem with Interpolating Euler Angles

- 오일러 앵글을 사용했을 때, 임의의 3 각도 값이 서로 독립적이지 않기 때문에 2개의 오일러 앵글 간의 각도 값을 보간(interpolating)할 때도 문제가 생긴다.
  - 12가지 회전조합 중에 어느 것을 사용하느냐에 따라 각자 다른 방향으로 각도를 보간할 수 있다.
  - 또한, (x, y, z) 회전조합의 경우 (0, 0, 0)에서 (0, 180, 0)로 움직이는 경우와 (0, 0, 0)에서 (180, 0, 180)으로 움직이는 경우 서로 완전히 다른 방향으로 보간하여 회전한다.



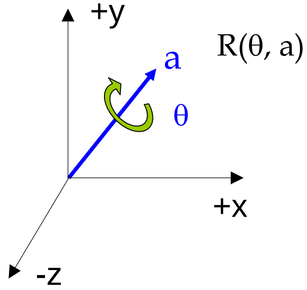
Halfway between (0,0,0) and (0,180,0)



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

Rotation Vectors and Axis/Angle

- 임의의 회전축 (axis)에 대한 하나의 회전각도 (angle) 4개의 숫자로 표현한다. 임의의 회전축을 나타내는 단위벡터 a (x, y, z)와 단위 벡터 주위로 회전각도를 나타내는 θ (0~360)값으로 구성된다.
- Gimbal lock문제는 피할 수 있으므로 Euler angle 방법보다는 좀 더 낫다. 그러나 2개의 회전 사이를 보간하는 경우 회전이 부드럽지 못한 문제가 여전히 존재한다.
- 궁극적으로 axis/angle을 하나의 3D rotation vector로 바꿀 수 있다.



Axis/Angle to Matrix

- 회전축(axis)/각(angle)로부터 다음과 같은 회전행렬 (rotation matrix)을 만든다.

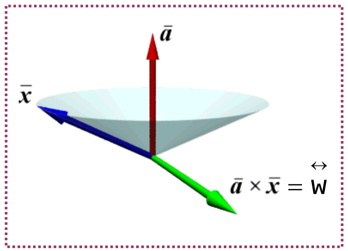
$$R=\begin{bmatrix}1&0&0\\0&1&0\\0&0&1\end{bmatrix}\cos\theta+\begin{bmatrix}a_x^2&a_xa_y&a_xa_z\\a_xa_y&a_y^2&a_ya_z\\a_xa_z&a_ya_z&a_z^2\end{bmatrix}(1-\cos\theta)+\begin{bmatrix}0&-a_z&a_y\\a_z&0&-a_x\\-a_y&a_x&0\end{bmatrix}\sin\theta$$

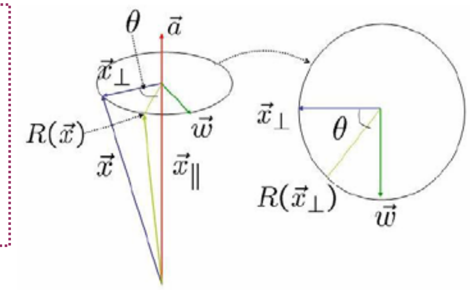
$$=\begin{bmatrix}a_x^2+\cos\theta(1-a_x^2)&a_xa_y(1-\cos\theta)-a_z\sin\theta&a_xa_z(1-\cos\theta)+a_y\sin\theta\\a_xa_y(1-\cos\theta)+a_z\sin\theta&a_y^2+\cos\theta(1-a_y^2)&a_ya_z(1-\cos\theta)-a_x\sin\theta\\a_xa_z(1-\cos\theta)-a_y\sin\theta&a_ya_z(1-\cos\theta)+a_x\sin\theta&a_y^2+\cos\theta(1-a_y^2)\end{bmatrix}$$

3D Rotation as Vector Components

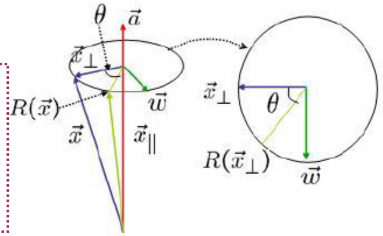
- Rotate a point by θ about an arbitrary axis, a =[a_x, a_y, a_z]

$$\begin{bmatrix}x'\\y'\\z'\end{bmatrix}=\left(\mathbf{Symmetric}\left(\begin{bmatrix}a_x\\a_y\\a_z\end{bmatrix}\right)(1-\cos\theta)+\mathbf{Skew}\left(\begin{bmatrix}a_x\\a_y\\a_z\end{bmatrix}\right)\sin\theta+\mathbf{I}\cos\theta\right)\begin{bmatrix}x\\y\\z\end{bmatrix}$$





3D Rotation as Vector Components



$$\begin{aligned}\vec{w} &= \vec{a} \times \vec{x}_\perp \\ &= \vec{a} \times (\vec{x} - \vec{x}_{||}) \\ &= (\vec{a} \times \vec{x}) - (\vec{a} \times \vec{x}_{||}) \\ &= \vec{a} \times \vec{x}\end{aligned}$$

$$R(\vec{x}_\perp) = \cos\theta\,\vec{x}_\perp + \sin\theta\,\vec{w}$$

$$\begin{aligned}\vec{x}_{||} &= (\vec{a}\cdot\vec{x})\vec{a} \\ \vec{x}_\perp &= \vec{x} - \vec{x}_{||} = \vec{x} - (\vec{a}\cdot\vec{x})\vec{a}\end{aligned}$$

$$\begin{aligned}R(\vec{x}) &= R(\vec{x}_{||}) + R(\vec{x}_\perp) \\ &= R(\vec{x}_{||}) + \cos\theta\,\vec{x}_\perp + \sin\theta\,\vec{w} \\ &= (\vec{a}\cdot\vec{x})\vec{a} + \cos\theta(\vec{x} - (\vec{a}\cdot\vec{x})\vec{a}) + \sin\theta\,\vec{w} \\ &= \cos\theta\,\vec{x} + (1-\cos\theta)(\vec{a}\cdot\vec{x})\vec{a} + \sin\theta(\vec{a}\times\vec{x})\end{aligned}$$

3D Rotation as Vector Components

$$\begin{bmatrix} x' \\ y' \\ z' \end{bmatrix} = \left(\mathbf{Symmetric}\left(\begin{bmatrix} a_x \\ a_y \\ a_z \end{bmatrix}\right)(1-\cos\theta) + \mathbf{Skew}\left(\begin{bmatrix} a_x \\ a_y \\ a_z \end{bmatrix}\right)\sin\theta + \mathbf{I}\cos\theta \right)\begin{bmatrix} x \\ y \\ z \end{bmatrix}$$

- The vector *a* specifies the axis of rotation. This axis vector must be normalized.
- The rotation angle is given by θ.
- The basic idea is that *any rotation can be decomposed into weighted contributions from three different vectors*.

3D Rotation as Vector Components

- *The symmetric matrix of a vector* generates a vector in the direction of the axis.
- The symmetric matrix is composed of the outer product of a row vector and an column vector of the same value.

$$\text{Symmetric}\left(\begin{bmatrix} a_x \\ a_y \\ a_z \end{bmatrix}\right) = \begin{bmatrix} a_x \\ a_y \\ a_z \end{bmatrix}\begin{bmatrix} a_x & a_y & a_z \end{bmatrix} = \begin{bmatrix} a_x^2 & a_x a_y & a_x a_z \\ a_x a_y & a_y^2 & a_y a_z \\ a_x a_z & a_y a_z & a_z^2 \end{bmatrix}$$

$$\text{Symmetric}\left(\begin{bmatrix} a_x \\ a_y \\ a_z \end{bmatrix}\right)\begin{bmatrix} x \\ y \\ z \end{bmatrix} = \bar{a}(\bar{a}\cdot\bar{x})$$

3D Rotation as Vector Components

- *Skew symmetric matrix of a vector* generates a vector that is perpendicular to both the axis and it's input vector.

$$\text{Skew}\left(\begin{bmatrix} a_x \\ a_y \\ a_z \end{bmatrix}\right) = \begin{bmatrix} 0 & -a_z & a_y \\ a_z & 0 & -a_x \\ -a_y & a_x & 0 \end{bmatrix}$$

$$\text{Skew}(\bar{a})\bar{x} = \bar{a}\times\bar{x}$$

Axis/Angle Representation

- Euler angle의 경우 미지수가 3개이나 Axis-angle의 경우에는 4개의 미지수 (axis, angle)을 사용한다. 이는 3DOF를 4DOF로 표현한 꼴이 된다.
- Euler's theorem에 따르면 회전을 표현하기 위해 오직 3개의 미지수가 필요하다고 했다. 즉, Axis/angle의 방식이 redundancy를 갖고 있다는 의미이다.
- 이 redundancy는 axis vector의 크기이다. 그러나 벡터의 크기는 아무런 정보를 갖고 있지 않기 때문에 axis vector를 정규화 (normalize) 함으로써 redundancy를 제거할 수 있다.

Matrix Representation

- 3x3 rotation matrix를 사용하여 회전을 표현할 수 있다.
- 그러나, 이것은 3개의 미지수를 사용하는 대신 9개의 미지수를 사용해야 한다는 의미이다. 즉, 이 경우 6 extra degrees of freedom이 필요하다.
- NOTE: 대부분의 그래픽스의 경우 4x4 행렬을 사용한다.

Matrix Representation

- Rotation matrix 회전 표현 방법이 기하 정보 (geometric data)에 적용하기에 가장 적합하다. 따라서 여러 가지 회전표현 방법을 사용하더라도 최종적으로 rotation matrix로 변환하여 쓰도록 한다.
- 그러나, rotation matrix로 그냥 사용하기엔 여러 가지 문제점들이 있다.
- 계산이 지속되면서 numerical errors가 축적된다.
- 데이터량에서 16개의 float을 사용해야 하므로 Storage가 문제시 된다.
- 2 rotation matrices 간의 보간(interpolation)도 어렵다.

Quaternions

- 사원수(Quaternion)란 3차원 그래픽스에서 회전을 표현할 때, 행렬 대신에 사용하는 수학적 개념으로 사원수는 4차원 복소수 공간 (complex space) 벡터이다.
- 실제로 회전의 표현에 있어서 가장 효과적인 방법이다.
- 사원수 (quaternion)는 4개의 구성요소로 표현한다.

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

Quaternions (Imaginary Space)

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

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

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

$$i = jk = -kj$$

$$j = ki = -ik$$

$$k = ij = -ji$$

Quaternions (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 Quaternions

- 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 Quaternions

- 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.

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

Quaternions 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 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 (ax, ay, az)과 각도(θ)에 의한 표현으로 변환:

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

$$ax = x/scale$$

$$ay = y/scale$$

$$az = z/scale$$

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

Matrix to Quaternion

- 행렬에서 사원수로 변환하려면 아래와 같은 식을 사용한다.

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

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

- 만약 w=0이면 나눗셈 계산이 안 된다. 따라서 먼저 x, y, z, w중에 가장 큰 숫자를 찾는다. 그리고 그것을 이용하여 행렬에서 사원수를 계산한다.

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

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

Basic Quaternion Mathematics

- 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)
  - $q^{-1} = q^* / |q|$
  - $q\,q^{-1} = q^{-1}\,q = 1$

Basic Quaternion Mathematics

- Exponential of quaternion
  - $\exp(v\ \theta) = v \sin\theta + \cos\theta$
- Logarithm of quaternion
  - $\log(q) = \log(v \sin\theta + \cos\theta) = \log(\exp(v\ \theta)) = v\ \theta$

    where $q = [v \sin\theta\ ,\ \cos\theta]$

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)

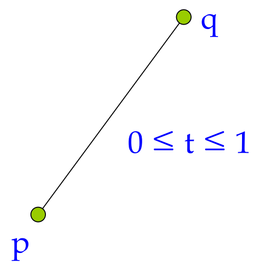
Linear Interpolation (LERP)

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

  $\mathrm{Lerp}(\mathbf{p}, \mathbf{q}, t) = (1-t)\mathbf{p} + (t)\mathbf{q}$
  where $0 \le t \le 1$

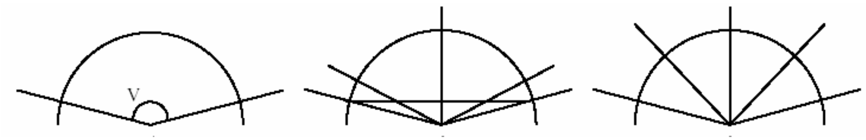
- 선형보간 공식의 또 다른 표현:

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



Why SLERP?

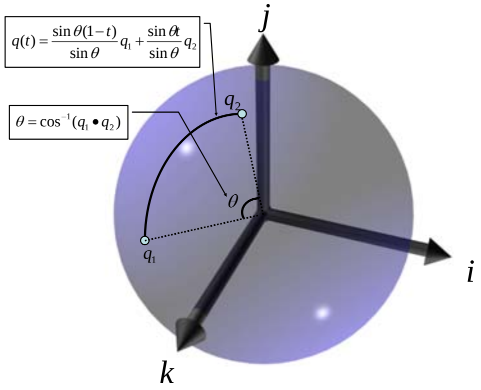
- 사원수의 공간은 구면공간의 성격을 띠는데, 이를 선형 보간하게 되면 속도 오차가 생긴다. 즉, 선형 보간의 특성상 가운데 부분에서 빨리 지나가게 된다. 그에 반해 구면 선형 보간은 일정한 속도를 유지한다.



- 그림은 Lerp 과 Slerp을 사용했을 때의 차이를 보여준다.
  - 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

- 구면 선형 보간 (spherical linear interpolation)은 벡터 q_1가 길이를 유지한 채로 회전해서 q_2가 되었다고 했을 때 회전한 그 사이 값을 보간하는 방법이다.



Spherical Linear Interpolation

- 두 단위 사원수 간의 구면 선형 보간 (spherical linear interpolation)은 아래와 같이 정의한다.

$$Slerp(\mathbf{p}, \mathbf{q}, t) = \frac{\sin((1-t)\theta)}{\sin\theta}\mathbf{p} + \frac{\sin(t\theta)}{\sin\theta}\mathbf{q}$$
$$where\ \theta = \mathrm{acos}(\mathbf{p} \cdot \mathbf{q})$$

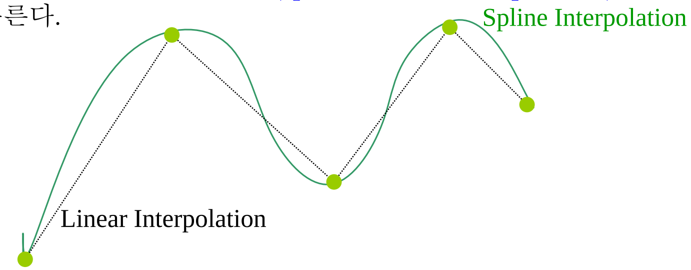
- 만약 p, q가 90도 이상 떨어져있다면, 짧은 경로를 선택한다.

Spherical Linear Interpolation

- 3차원 공간에서의 각 회전에 대해 사원수 공간에서 2개의 중복 벡터가 존재한다.
- Slerp(**p**, **q**, t) 과 Slerp(-**p**, **q**, t) 차이는 무엇일까?
  - 둘 중 하나는 구면에서 90도보다 작게 움직이는 것이고, 다른 하나는 90도 보다 크게 움직이는 것이다. 짧은 경로로 회전하는 것과 긴 경로로 회전하는 것에 해당한다.
  - 짧은 경로를 선택하는 것이 좋기 때문에 두 벡터의 dot product가 음수이면 두 벡터 중 하나를 음수로 바꿔준다.

Why SQUAD?

- Slerp (구면 선형 보간)은 두 방향 사이를 보간하는 데 더할 나위 없이 좋은 함수이다.
  - 두 개 이상의 방향, q_0, q_1, q_2, q_3, ..q_n이 있고 q_0부터 q_1, q_1 부터 q_2, 하는 식으로 q_n 까지 보간하고자 하면 slerp 함수를 그냥 연달아 호출하면 된다. 그런데 이렇게 하면 방향 보간 중에 갑작스럽게 방향이 바뀔 수 있다 (at the control points).
  - 따라서 더 좋은 보간 방법으로 스플라인 (spline)을 사용하는 방법이 있다. 이를 구면 입방 보간 (spherical cubic interpolation)이라 부른다.



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.

D3DXQUATERNION

- 사원수 구조체

```
Typedef struct D3DXQUATERNION {
    FLOAT x;
    FLOAT y;
    FLOAT z;
    FLOAT w;
} D3DXQUATERNION;

q.x = sin(theta/2) * axis.x
q.y = sin(theta/2) * axis.y
q.z = sin(theta/2) * axis.z
q.w = cos(theta/2)
```

D3DXQUATERNION

- DX9 함수 구현

```
// q*
D3DXQUATERNION * D3DXQuaternionConjugate
                    (D3DXQUATERNION *pOut,
                     CONST D3DXQUATERNION *pQ);

// q1q2
D3DXQUATERNION *D3DXQuaternionMultiply
                    (D3DXQUATERNION *pOut,
                     CONST D3DXQUATERNION *pQ1,
                     CONST D3DXQUATERNION *pQ2);

// q1·q2
FLOAT D3DXQuaternionDot
                    (CONST D3DXQUATERNION *pQ1,
                     CONST D3DXQUATERNION *pQ2);
```

D3DXQUATERNION

- // yaw/pitch/roll -> quaternion

```
D3DXQUATERNION * D3DXQuaternionRotationYawPitchRoll
                    (D3DXQUATERNION *pOut,
                     FLOAT Yaw,
                     FLOAT Pitch,
                     FLOAT Roll);

// rotation matrix -> quaternion
D3DXQUATERNION * D3DXQuaternionRotationMatrix
                    (D3DXQUATERNION * pOut,
                     CONST D3DXMATRIX *pM);
```

D3DXQUATERNION

- // quaternion -> axis/angle

```
void D3DXQuaternionToAxisAngle
                          (CONST D3DXQUATERNION *pOut,
                            D3DXVECTOR3 *pAxis,
                            FLOAT *pAngle);

// quaternion -> rotation matrix
D3DXMATRIX *D3DXMatrixRotationQuaternion
                          (D3DXMATRIX *pOut,
                           CONST D3DXQUATERNION *pQ);
```

D3DXQUATERNION

- // slerp(q_1, q_2, t)

```
D3DXQUATERNION *D3DXQuaternionSlerp
                          (D3DXQUATERNION *pOut,
                           CONST D3DXQUATERNION *pQ1,
                           CONST D3DXQUATERNION *pQ2,
                           FLOAT t);
```