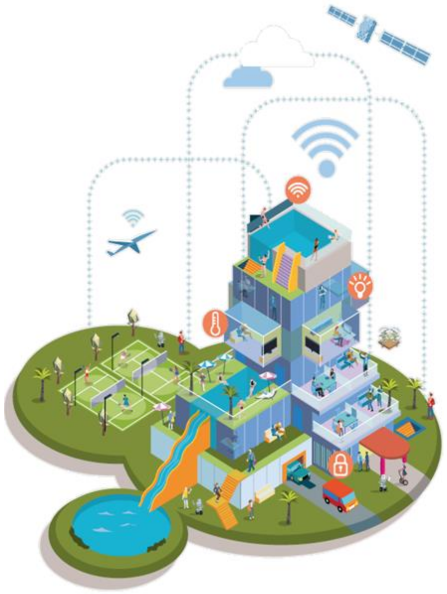
컴퓨터그래픽스
Computer Graphics

Quaternion



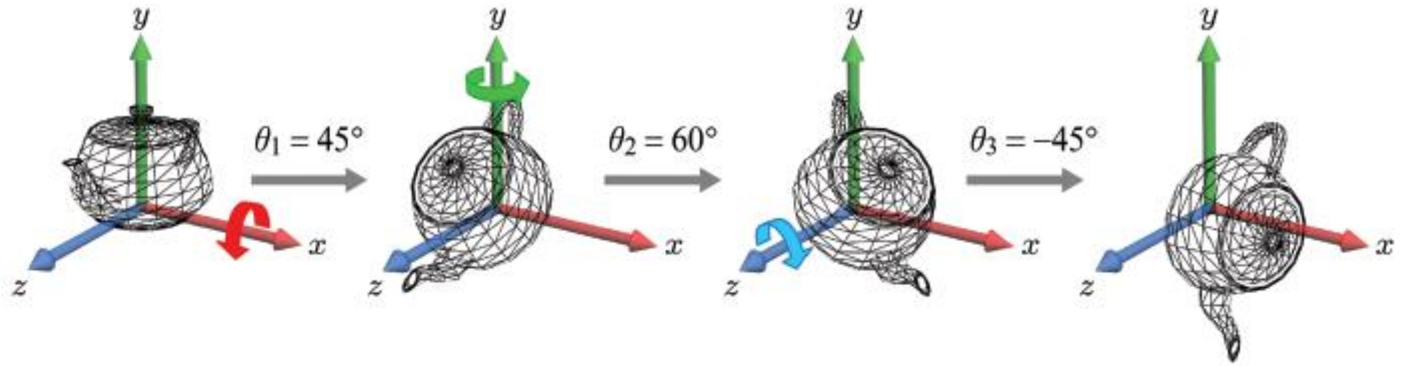
부산대학교 정보·의생명공학대학
정보컴퓨터공학부

This class...

- Euler Transforms and Quaternions
    - Euler Transforms
    - Keyframe Animation
    - Quaternion
    - 3D rotation through Quaternions
    - Interpolation of Quaternions

Euler Transforms

- When we successively rotate an object about the principal axes, the object acquires an arbitrary orientation. This method of determining an object's orientation is called *Euler transform*, and the rotations angles, $(\theta_1, \theta_2, \theta_3)$, are called the *Euler angles*.



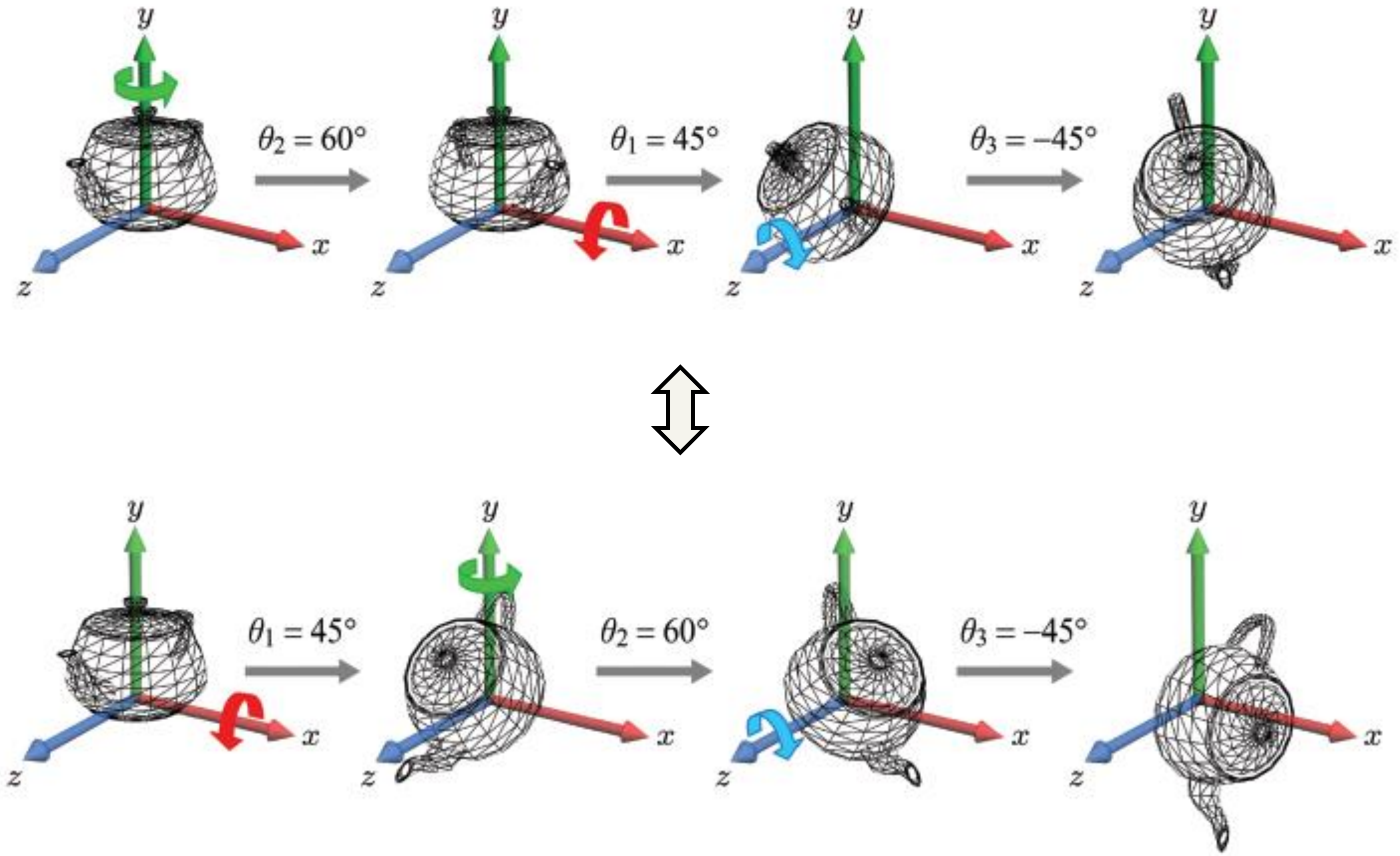
- Concatenating three matrices produces a single matrix defining an arbitrary orientation.

$$R_z(-45°)R_y(60°)R_x(45°) = \begin{pmatrix} \frac{\sqrt{2}}{2} & \frac{\sqrt{2}}{2} & 0 \\ -\frac{\sqrt{2}}{2} & \frac{\sqrt{2}}{2} & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} \frac{1}{2} & 0 & \frac{\sqrt{3}}{2} \\ 0 & 1 & 0 \\ -\frac{\sqrt{3}}{2} & 0 & \frac{1}{2} \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 \\ 0 & \frac{\sqrt{2}}{2} & -\frac{\sqrt{2}}{2} \\ 0 & \frac{\sqrt{2}}{2} & \frac{\sqrt{2}}{2} \end{pmatrix}$$

$$= \begin{pmatrix} \frac{\sqrt{2}}{4} & \frac{2+\sqrt{3}}{4} & \frac{-2+\sqrt{3}}{4} \\ -\frac{\sqrt{2}}{4} & \frac{2-\sqrt{3}}{4} & \frac{-2-\sqrt{3}}{4} \\ -\frac{\sqrt{3}}{2} & \frac{\sqrt{2}}{4} & \frac{\sqrt{2}}{4} \end{pmatrix}$$

Euler Transforms (cont'd)

- The rotation axes are not necessarily taken in the order of *x*, *y*, and *z*. Shown below is the order of *y*, *x*, and *z*. Observe that the teapot has a different orientation from the previous one.

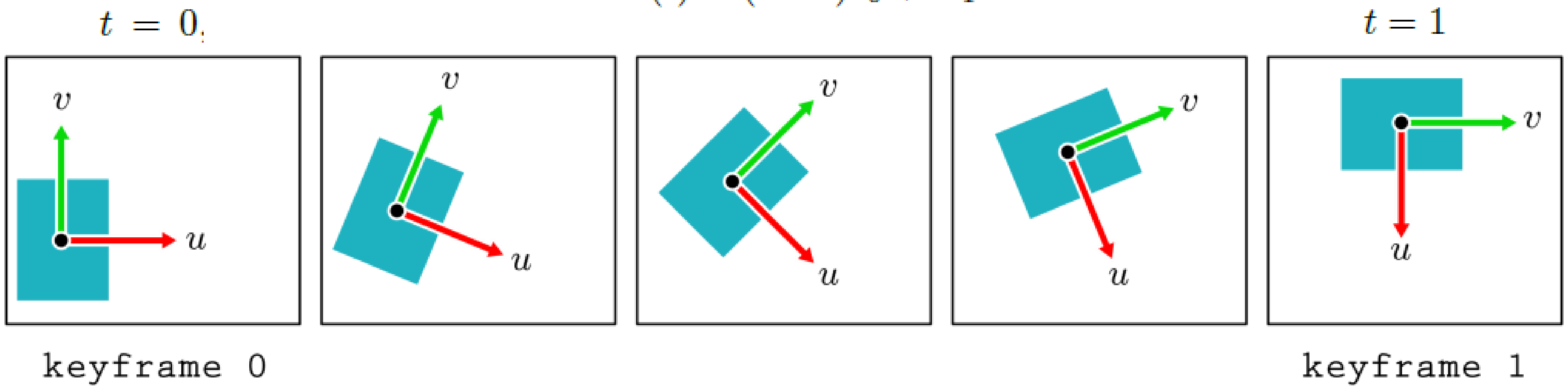


Keyframe Animation in 2D

- In the traditional hand-drawn cartoon animation, the senior key artist would draw the *keyframes*, and the junior artist would fill the *in-between frames*.
- For a 30-fps computer animation, for example, much fewer than 30 frames are defined per second. They are the keyframes. In real-time computer animation, the in-between frames are automatically filled at run time.
- The key data are assigned to the keyframes, and they are *interpolated* to generate the in-between frames.
- In the example, the center position *p* and orientation angle θ are interpolated.

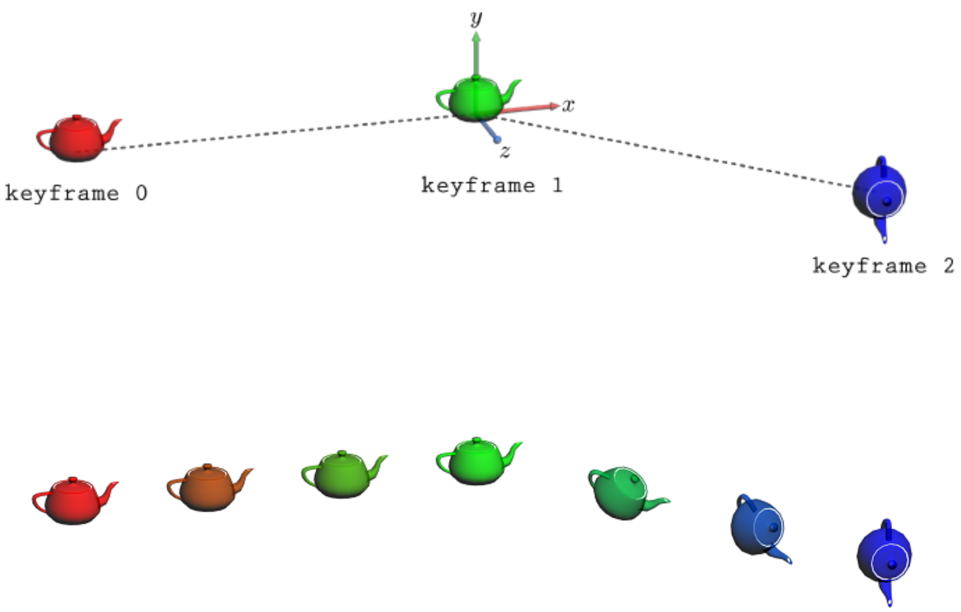
$$p(t) = (1-t)p_0 + tp_1$$

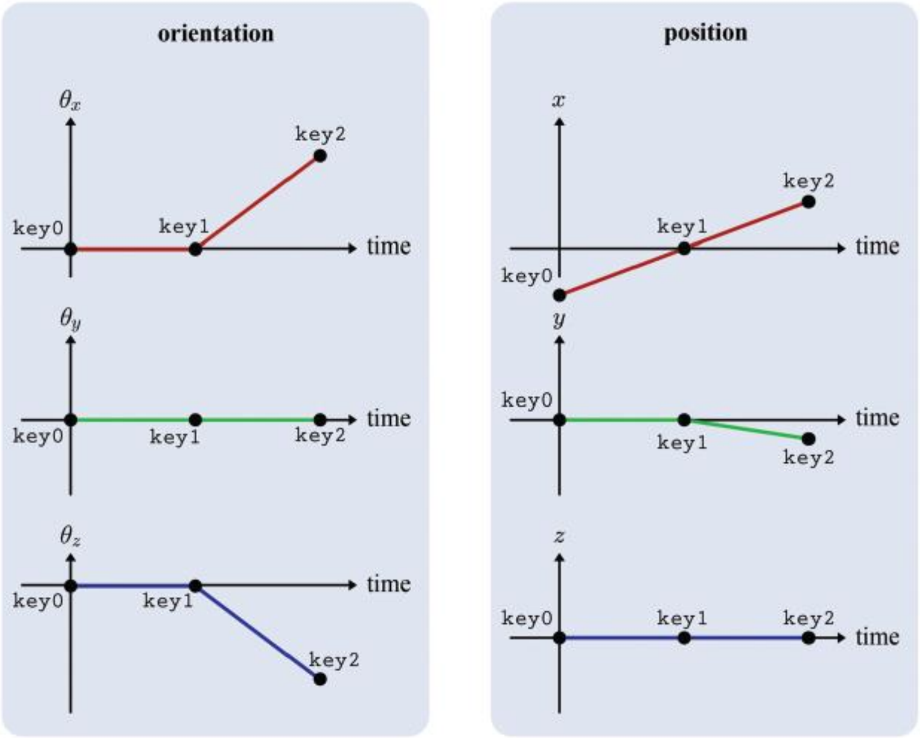
$$\theta(t) = (1-t)\theta_0 + t\theta_1$$



Keyframe Animation in 3D

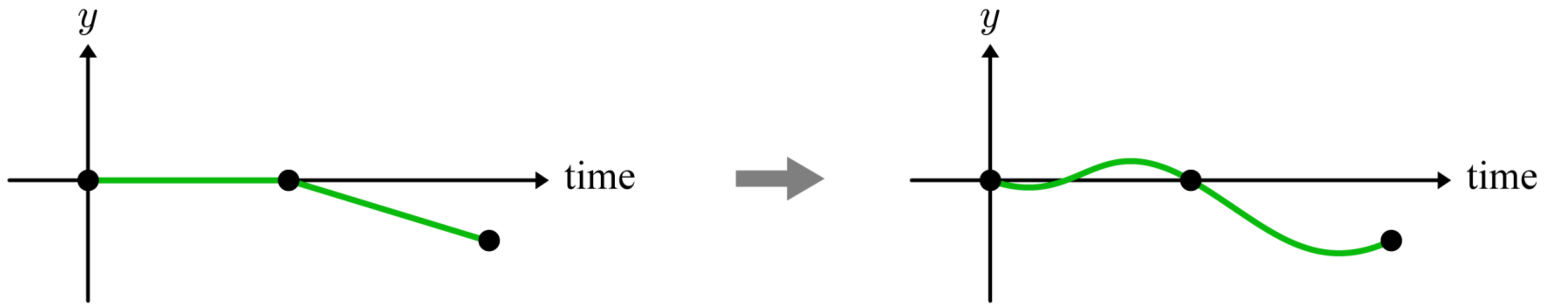
- Keyframe animation in 3D: Seven teapot instances are defined by sampling the graphs seven times.



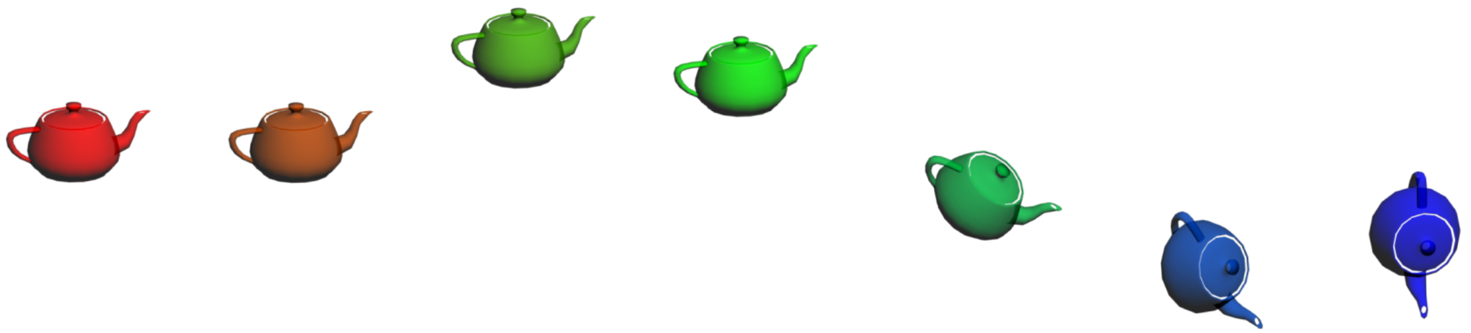


Keyframe Animation in 3D (cont'd)

- Smoother animation may often be obtained using a higher-order interpolation.



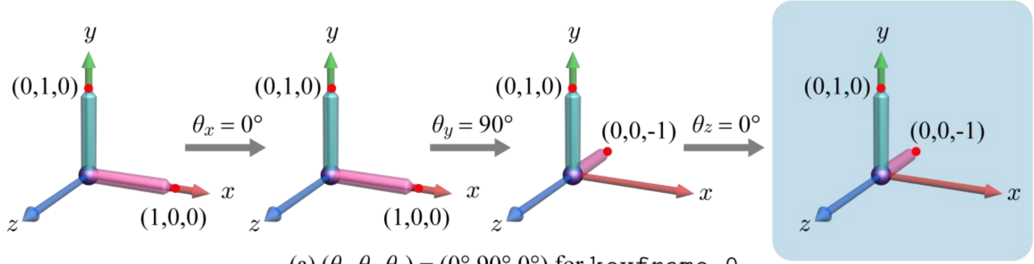
(a)



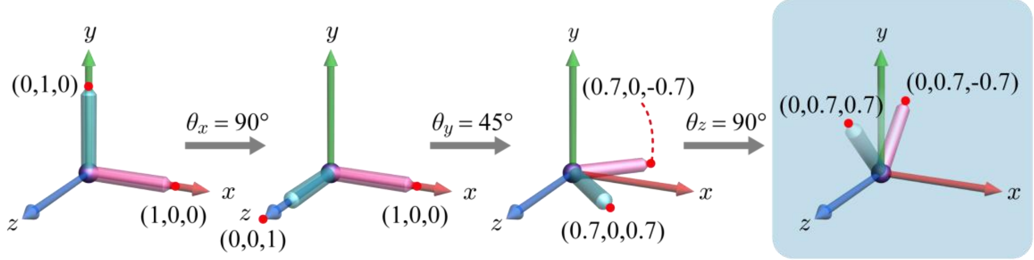
(b)

A Problem of Euler Angles

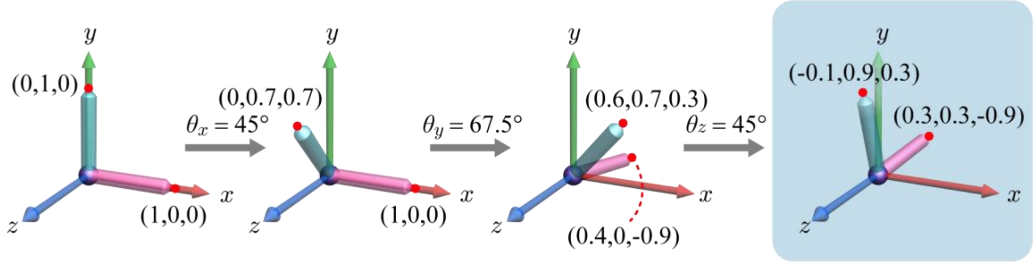
- Euler angles are not always correctly interpolated and so are not suitable for keyframe animation



(a) $(\theta_x,\theta_y,\theta_z) = (0°,90°,0°)$ for `keyframe 0`



(b) $(\theta_x,\theta_y,\theta_z) = (90°,45°,90°)$ for `keyframe 1`



(c) Interpolated Euler angles $(\theta_x,\theta_y,\theta_z) = (45°,67.5°,45°)$

Quaternion

- A quaternion is an extended complex number.

$$q_x i + q_y j + q_z k + q_w = (q_x, q_y, q_z, q_w) = (\mathbf{q}_v, q_w)$$

$$\begin{aligned} i^2 &= j^2 = k^2 = -1 \\ ij &= k, ji = -k \\ jk &= i, kj = -i \\ ki &= j, ik = -j \end{aligned}$$

$$\begin{aligned} \mathbf{p} &= (p_x, p_y, p_z, p_w) \\ \mathbf{q} &= (q_x, q_y, q_z, q_w) \\ \mathbf{pq} &= (p_x i + p_y j + p_z k + p_w)(q_x i + q_y j + q_z k + q_w) \\ &= (p_x q_w + p_y q_z - p_z q_y + p_w q_x) i + \\ &\quad (-p_x q_z + p_y q_w + p_z q_x + p_w q_y) j + \\ &\quad (p_x q_y - p_y q_x + p_z q_w + p_w q_z) k + \\ &\quad (-p_x q_x - p_y q_y - p_z q_z + p_w q_w) \end{aligned}$$

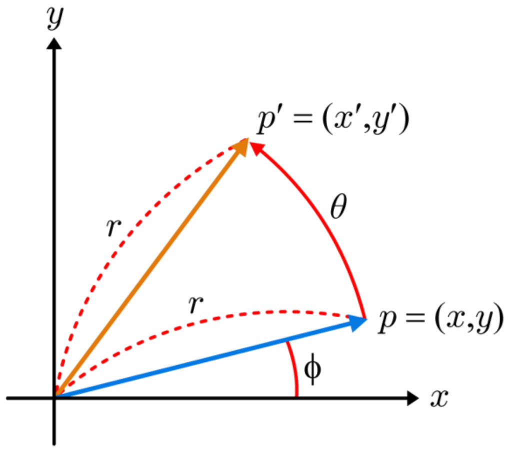
- Conjugate $\mathbf{q}^* = (-\mathbf{q}_v, q_w)$
  $= (-q_x, -q_y, -q_z, q_w)$
  $= -q_x i - q_y j - q_z k + q_w$

- It is easy to show that **(pq)*****=q*****p***.
- Magnitude: If the magnitude of a quaternion is 1, it's called a *unit quaternion.*

$$\|\mathbf{q}\| = \sqrt{q_x^2 + q_y^2 + q_z^2 + q_w^2}$$

2D Rotation through Complex Numbers

- Recall 2D rotation



$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} cos\theta & -sin\theta \\ sin\theta & cos\theta \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix}$$
$$= \begin{pmatrix} xcos\theta - ysin\theta \\ xsin\theta + ycos\theta \end{pmatrix}$$

- Let us represent (x,y) by a complex number x+yi, and denote it by **p**.
- Given the rotation angle θ, let us consider a unit-length complex number, $cos\theta$+$sin\theta i$. We denote it by **q**. Then, we have the following:

$$\begin{aligned} \mathbf{pq} &= (x + yi)(cos\theta + sin\theta i) \\ &= (xcos\theta - ysin\theta) + (xsin\theta + ycos\theta)i \end{aligned}$$

- Surprisingly, the real and imaginary parts of **pq** represent the rotated coordinates.

3D Rotation through Quaternions

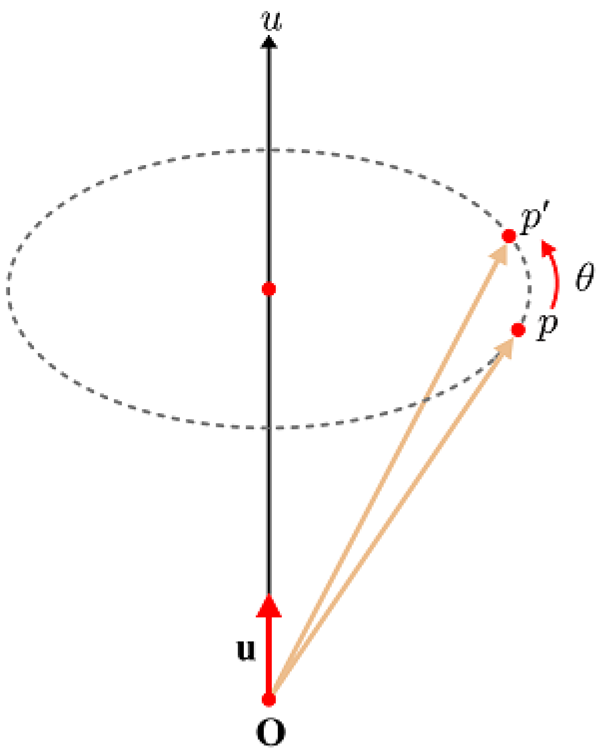
- As extended complex numbers, quaternions can be used to describe 3D rotation.
- Consider rotating a 3D vector *p* about an axis *u* by an angle θ. Represent both "the vector to be rotated" and "the rotation" in quaternions.
  - Define a quaternion **p** using *p*.

$$\begin{aligned}\mathbf{p} &= (\mathbf{p}_v, p_w)\\ &= (p, 0)\end{aligned}$$

  - Define a *unit quaternion* **q** using *u* and θ. (The axis *u* is divided by its length to make a unit vector **u**.)

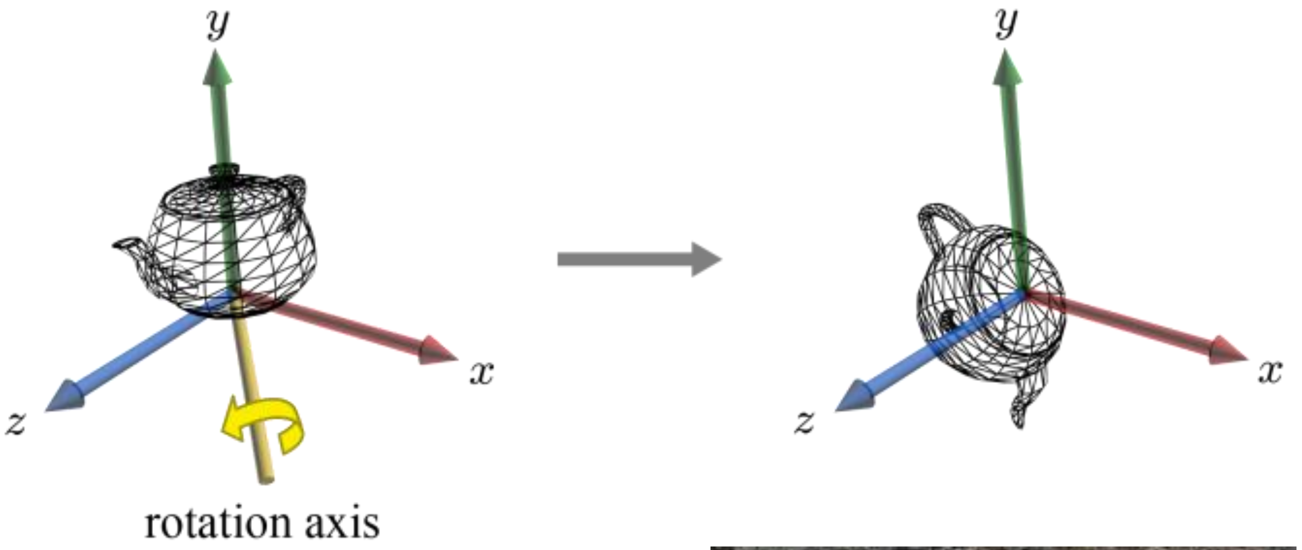
$$\begin{aligned}\mathbf{q} &= (\mathbf{q}_v, q_w)\\ &= (sin\frac{\theta}{2}\mathbf{u}, cos\frac{\theta}{2})\end{aligned}$$

  - Compute **qpq***. Then, its *imaginary part* represents the rotated vector.



3D Rotation through Quaternions (cont'd)

- Quaternions enable rotations about arbitrary axes.



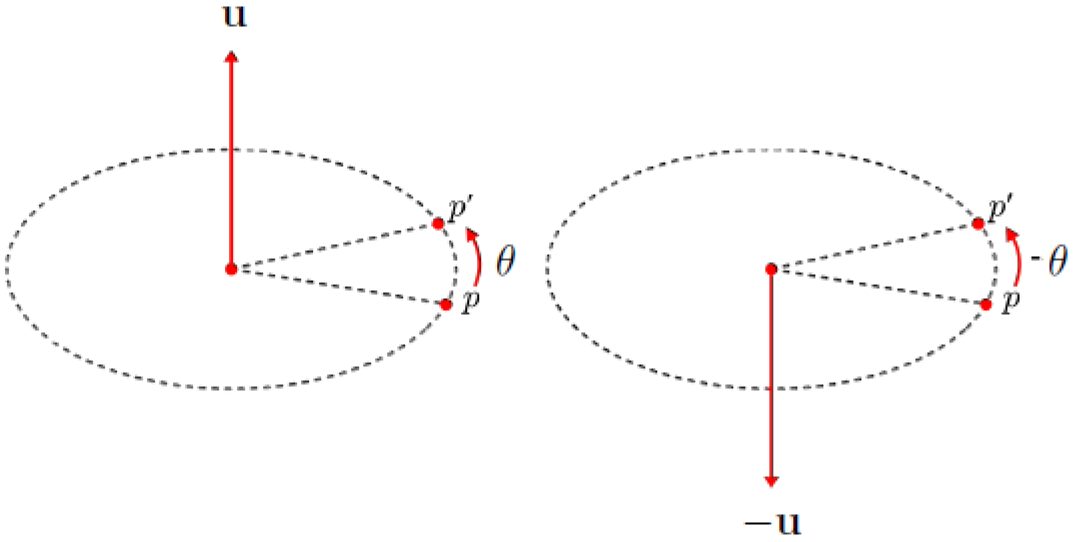
- https://eater.net/quaternions

3D Rotation through Quaternions (cont'd)

- Let $\mathbf{p}'$ denote $\mathbf{qpq^*}$. It represents the rotated vector p'. Consider rotating p' by another quaternion $\mathbf{r}$. The combined rotation is represented in $\mathbf{rq}$.

$$\begin{aligned} \mathbf{r}\mathbf{p}'\mathbf{r}^* &= \mathbf{r}(\mathbf{q}\mathbf{p}\mathbf{q}^*)\mathbf{r}^* \\ &= (\mathbf{r}\mathbf{q})\mathbf{p}(\mathbf{q}^*\mathbf{r}^*) \\ &= (\mathbf{r}\mathbf{q})\mathbf{p}(\mathbf{r}\mathbf{q})^* \end{aligned}$$

- "Rotation about $\mathbf{u}$ by θ" is identical to "rotation about $-\mathbf{u}$ by $-\theta$."



$$\mathbf{q} = (sin\tfrac{\theta}{2}\mathbf{u}, cos\tfrac{\theta}{2})$$

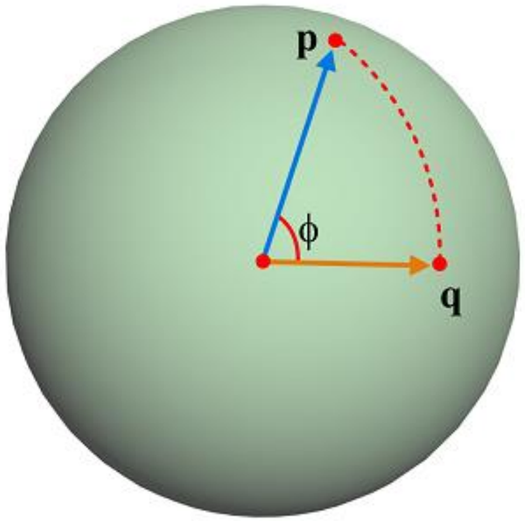
$$\begin{aligned} \mathbf{r} &= (sin\tfrac{-\theta}{2}(-\mathbf{u}), cos\tfrac{-\theta}{2}) \\ &= (sin\tfrac{\theta}{2}\mathbf{u}, cos\tfrac{\theta}{2}) \end{aligned}$$

Interpolation of Quaternions

- Consider two unit quaternions, **p** and **q**, which represent rotations. They can be interpolated using parameter *t* in the range of [0,1]:

$$\frac{sin(\phi(1-t))}{sin\phi}\mathbf{p} + \frac{sin(\phi t)}{sin\phi}\mathbf{q}$$

$$cos\phi = \mathbf{p} \cdot \mathbf{q} = (p_x, p_y, p_z, p_w) \cdot (q_x, q_y, q_z, q_w) = p_x q_x + p_y q_y + p_z q_z + p_w q_w$$



- This is called *spherical linear interpolation* (slerp).

Quaternion and Matrix

- A quaternion **q** representing a rotation can be converted into a matrix form. If **q** = (q_x,q_y,q_z,q_w), the rotation matrix is defined as follows:

$$\begin{pmatrix} 1-2(q_y^2+q_z^2) & 2(q_xq_y-q_wq_z) & 2(q_xq_z+q_wq_y) & 0 \\ 2(q_xq_y+q_wq_z) & 1-2(q_x^2+q_z^2) & 2(q_yq_z-q_wq_x) & 0 \\ 2(q_xq_z-q_wq_y) & 2(q_yq_z+q_wq_x) & 1-2(q_x^2+q_y^2) & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

- Conversely, given a rotation matrix, we can compute its quaternion. It requires us to extract {q_x,q_y,q_z,q_w} given the above matrix.
  - Compute the sum of all diagonal elements.

$$4-4(q_x^2+q_y^2+q_z^2) = 4-4(1-q_w^2) = 4q_w^2$$

  - So, we obtain q_w.
  - Subtract m_{12} from m_{21} of the above matrix.

$$m_{21}-m_{12} = 2(q_xq_y+q_wq_z)-2(q_xq_y-q_wq_z) = 4q_wq_z$$

  - As we know q_w, we can compute q_z. Similarly, we can compute q_x and q_y.

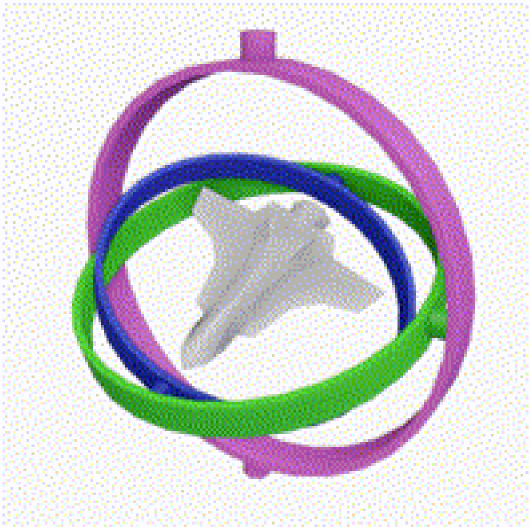
Quaternion - Summary

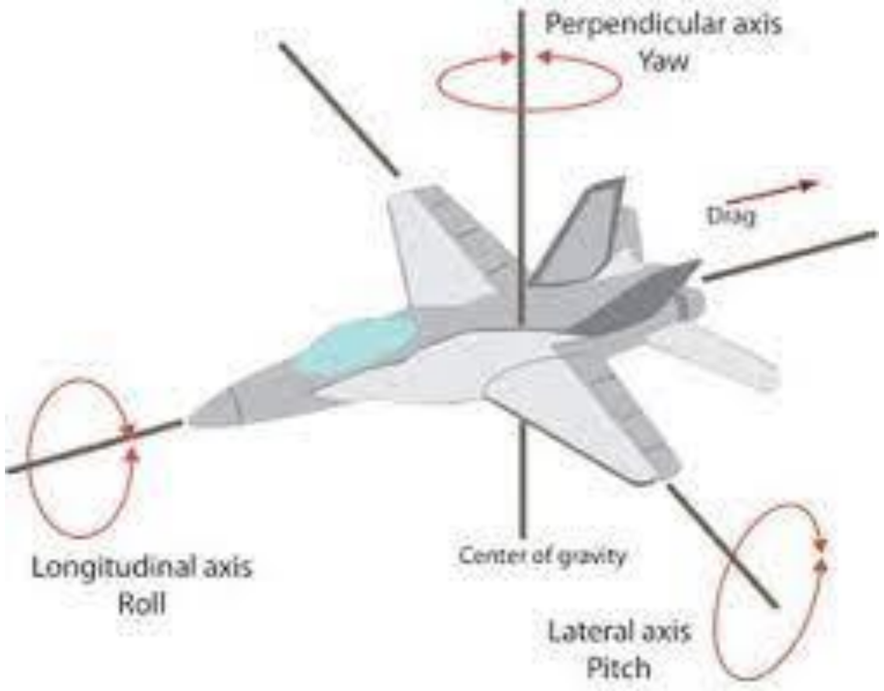
- Summary
  - An arbitrary 3D rotation is represented in a quaternion as well as in Euler transform.
  - Quaternions are correctly interpolated through slerp.
  - A quaternion can be converted into a rotation matrix.

- Given quaternions for the keyframes,
  - spherically interpolate them for the in-between frames, and
  - convert each interpolated quaternion into a rotation matrix.

- Unity references
  - https://docs.unity3d.com/2022.3/Documentation/Manual/QuaternionAndEulerRotationsInUnity.html
  - https://docs.unity3d.com/ScriptReference/Quaternion.html
  - https://docs.unity3d.com/ScriptReference/Quaternion.Slerp.html
  - https://docs.unity3d.com/ScriptReference/Matrix4x4.Rotate.html

Gimbal lock





https://youtu.be/kB7iE8Udq5g?si=sc8LX-I8X-AN-UwR

Gimbal lock

Loss of a degree of freedom with Euler angles [edit]

A rotation in 3D space can be represented numerically with matrices in several ways. One of these representations is:

$$R = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\alpha & -\sin\alpha \\ 0 & \sin\alpha & \cos\alpha \end{bmatrix} \begin{bmatrix} \cos\beta & 0 & \sin\beta \\ 0 & 1 & 0 \\ -\sin\beta & 0 & \cos\beta \end{bmatrix} \begin{bmatrix} \cos\gamma & -\sin\gamma & 0 \\ \sin\gamma & \cos\gamma & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

An example worth examining happens when $\beta = \frac{\pi}{2}$. Knowing that $\cos\frac{\pi}{2} = 0$ and $\sin\frac{\pi}{2} = 1$, the above expression becomes equal to:

$$R = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\alpha & -\sin\alpha \\ 0 & \sin\alpha & \cos\alpha \end{bmatrix} \begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ -1 & 0 & 0 \end{bmatrix} \begin{bmatrix} \cos\gamma & -\sin\gamma & 0 \\ \sin\gamma & \cos\gamma & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Carrying out matrix multiplication:

$$R = \begin{bmatrix} 0 & 0 & 1 \\ \sin\alpha & \cos\alpha & 0 \\ -\cos\alpha & \sin\alpha & 0 \end{bmatrix} \begin{bmatrix} \cos\gamma & -\sin\gamma & 0 \\ \sin\gamma & \cos\gamma & 0 \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 1 \\ \sin\alpha\cos\gamma + \cos\alpha\sin\gamma & -\sin\alpha\sin\gamma + \cos\alpha\cos\gamma & 0 \\ -\cos\alpha\cos\gamma + \sin\alpha\sin\gamma & \cos\alpha\sin\gamma + \sin\alpha\cos\gamma & 0 \end{bmatrix}$$

And finally using the trigonometry formulas:

$$R = \begin{bmatrix} 0 & 0 & 1 \\ \sin(\alpha+\gamma) & \cos(\alpha+\gamma) & 0 \\ -\cos(\alpha+\gamma) & \sin(\alpha+\gamma) & 0 \end{bmatrix}$$

Source: https://en.wikipedia.org/wiki/Gimbal_lock

slerp

[Note: Proof of spherical linear interpolation]

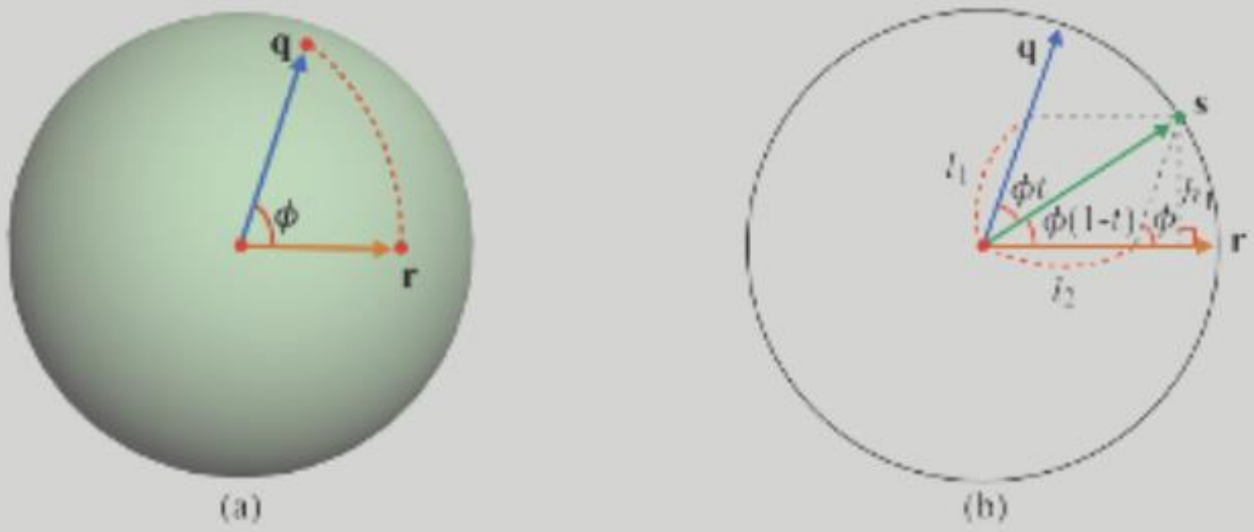


Fig. 11.11: Spherical linear interpolation on the 4D unit sphere: (a) Shortest arc between $\mathbf{q}$ and $\mathbf{r}$. (b) Spherical linear interpolation of $\mathbf{q}$ and $\mathbf{r}$ returns $\mathbf{s}$.

The set of all possible quaternions makes up a 4D unit sphere. Fig. 11.11-(a) illustrates $\mathbf{q}$ and $\mathbf{r}$ on the sphere. Note that the interpolated quaternion must lie on the shortest arc connecting $\mathbf{q}$ and $\mathbf{r}$. Fig. 11.11-(b) shows the cross section of the unit sphere. It is in fact the great circle defined by $\mathbf{q}$ and $\mathbf{r}$. The

slerp

interpolated quaternion is denoted by $\mathbf{s}$ and is defined by the parallelogram rule:

$$\mathbf{s} = l_1\mathbf{q} + l_2\mathbf{r} \tag{11.23}$$

In Fig. 11.11-(b), $sin\phi = \frac{h_1}{l_1}$, and therefore $l_1 = \frac{h_1}{sin\phi}$. As $h_1 = sin(\phi(1-t))$, we can compute l_1 as follows:

$$l_1 = \frac{sin(\phi(1-t))}{sin\phi} \tag{11.24}$$

Similarly, l_2 is computed as follows:

$$l_2 = \frac{sin(\phi t)}{sin\phi} \tag{11.25}$$

When we insert Equations (11.24) and (11.25) into Equation (11.23), we obtain the slerp function presented in Equation (11.22).

$$\frac{sin(\phi(1-t))}{sin\phi}\mathbf{q} + \frac{sin(\phi t)}{sin\phi}\mathbf{r} \tag{11.22}$$

Conversion from a quaternion to a rotation matrix

Consider two quaternions, $\mathbf{p} = (p_x, p_y, p_z, p_w)$ and $\mathbf{q} = (q_x, q_y, q_z, q_w)$. Their multiplication returns another quaternion:

$$\begin{aligned}\mathbf{pq} &= (p_x i + p_y j + p_z k + p_w)(q_x i + q_y j + q_z k + q_w) \\ &= (p_x q_w + p_y q_z - p_z q_y + p_w q_x)i + \\ &\quad (-p_x q_z + p_y q_w + p_z q_x + p_w q_y)j + \\ &\quad (p_x q_y - p_y q_x + p_z q_w + p_w q_z)k + \\ &\quad (-p_x q_x - p_y q_y - p_z q_z + p_w q_w)\end{aligned} \tag{11.8}$$

[Note: Conversion from a quaternion to a rotation matrix]

Notice that each component of $\mathbf{pq}$ presented in Equation (11.8) is a linear combination of p_x, p_y, p_z and p_w. Therefore, $\mathbf{pq}$ can be represented by a matrix-vector multiplication form:

$$\mathbf{pq} = \begin{pmatrix} q_w & q_z & -q_y & q_x \\ -q_z & q_w & q_x & q_y \\ q_y & -q_x & q_w & q_z \\ -q_x & -q_y & -q_z & q_w \end{pmatrix} \begin{pmatrix} p_x \\ p_y \\ p_z \\ p_w \end{pmatrix} = M_{\mathbf{q}}\mathbf{p} \tag{11.27}$$

where $M_{\mathbf{q}}$ is a 4×4 matrix built upon the components of $\mathbf{q}$. Each component of $\mathbf{pq}$ in Equation (11.8) is also a linear combination of q_x, q_y, q_z and q_w, and therefore $\mathbf{pq}$ can be represented by another matrix-vector multiplication

Conversion from a quaternion to a rotation matrix

form:

$$\mathbf{pq} = \begin{pmatrix} p_w & -p_z & p_y & p_x \\ p_z & p_w & -p_x & p_y \\ -p_y & p_x & p_w & p_z \\ -p_x & -p_y & -p_z & p_w \end{pmatrix} \begin{pmatrix} q_x \\ q_y \\ q_z \\ q_w \end{pmatrix} = N_{\mathbf{p}}\mathbf{q} \tag{11.28}$$

where $N_{\mathbf{p}}$ is a 4×4 matrix built upon the components of $\mathbf{p}$.

Then, $\mathbf{qpq}^*$ in Equation (11.15) is expanded as follows:

$$\begin{aligned} \mathbf{qpq}^* &= (\mathbf{qp})\mathbf{q}^* \\ &= M_{\mathbf{q}^*}(\mathbf{qp}) \\ &= M_{\mathbf{q}^*}(N_{\mathbf{q}}\mathbf{p}) \\ &= (M_{\mathbf{q}^*}N_{\mathbf{q}})\mathbf{p} \\ &= \begin{pmatrix} q_w & -q_z & q_y & -q_x \\ q_z & q_w & -q_x & -q_y \\ -q_y & q_x & q_w & -q_z \\ q_x & q_y & q_z & q_w \end{pmatrix} \begin{pmatrix} q_w & -q_z & q_y & q_x \\ q_z & q_w & -q_x & q_y \\ -q_y & q_x & q_w & q_z \\ -q_x & -q_y & -q_z & q_w \end{pmatrix} \begin{pmatrix} p_x \\ p_y \\ p_z \\ p_w \end{pmatrix} \end{aligned} \tag{11.29}$$

$M_{\mathbf{q}^*}N_{\mathbf{q}}$ returns a 4×4 matrix. Consider its first element, $(q_w^2 - q_z^2 - q_y^2 + q_x^2)$. As $\mathbf{q}$ is a unit quaternion, $q_x^2 + q_y^2 + q_z^2 + q_w^2 = 1$. The first element is rewritten as $(1 - 2(q_y^2 + q_z^2))$. When all of the 4×4 elements are processed in similar manners, $M_{\mathbf{q}^*}N_{\mathbf{q}}$ is proven to be $\mathbf{M}$ in Equation (11.26).

Thank You!

Slides are modified from

Introduction to Computer Graphics with OpenGL ES (J. Han)