Transformation & Representing Orientations

305890
Spring 2011
4/4/2011
Kyoung Shin Park

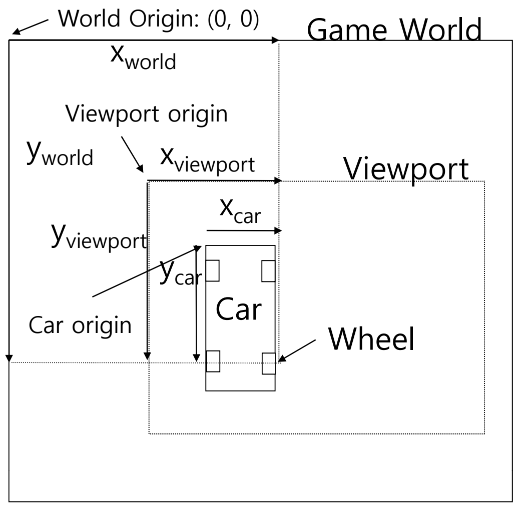
Outline

- Coordinate Systems
- Transformation
  - Translate
  - Rotate
  - Scale
- Orientation
  - Euler Angles
  - Rotation Matrix
  - Quaternion

Multiple Coordinate Space

- Use more than one coordinate system to specify coordinates – multiple coordinate space
  - Why need ?
- Some Useful Coordinate Spaces
  - World coordinate
  - Object(Local) coordinate
  - Camera(Viewport) coordinate
  - Inetial Coordinate

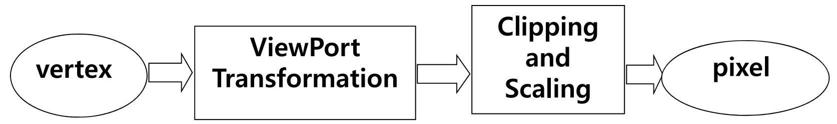
Multiple Coordinate Space: Example



- World coordinate of Wheel = (x_{world}, y_{world})
- Object coordinate of Wheel to the car = (x_{car}, y_{car})
- Camera coordinate of Wheel =($x_{viewport}$, $y_{viewport}$)

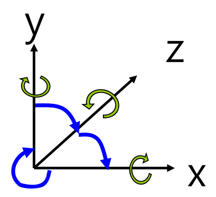
Multiple Coordinate Space: Example

- In different situation, we use different coordinates of the wheel
- We can calculate the world coordinate of wheel from coordinate of car and local coordinate of wheel to the car, why ?
- Rendering pipeline



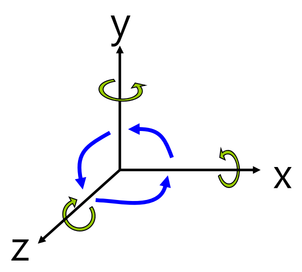
LHS Coordinate Systems

- Left Hand Coordinate System (LHS) – z+ forward
- Clockwise rotation
- If X-axis rotation,
  Y->Z rotation is positive
- If Y-axis rotation,
  Z->X rotation is positive
- If Z-axis rotation,
  X->Y rotation is positive



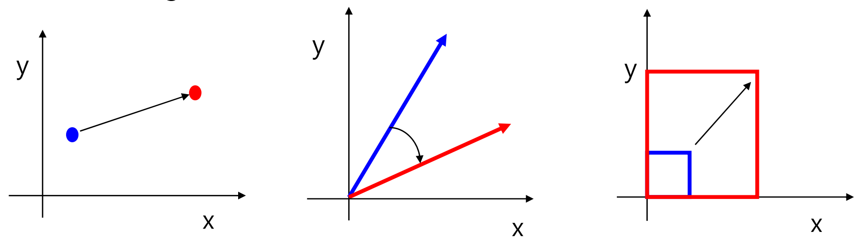
RHS Coordinate Systems

- Right Hand Coordinate System (RHS) – z+ coming out of the screen
- Counter clockwise rotation
- If X-axis rotation,
  Y->Z rotation is positive
- If Y-axis rotation,
  Z->X rotation is positive
- If Z-axis rotation,
  X->Y rotation is positive



Transformation

- Geometric transformations are **functions that map points from one place to another**.
- 2D transformation
  - Translation
  - Rotation
  - Scaling



Transformation

- Direct3D/XNA uses 4x4 matrix and 1x4 vector for transformation
  - v_{1x4} = (2, 6, -3, 1)
  - T_{4x4} = translate 10 units in x-axis
  - **$v' = v_{1x4}\,T_{4x4}$** = (12, 6, -3, 1)

$$\begin{bmatrix} x' & y' & z' & 1 \end{bmatrix} = \begin{bmatrix} x & y & z & 1 \end{bmatrix} \begin{bmatrix} M_{11} & M_{12} & M_{13} & M_{14} \\ M_{21} & M_{22} & M_{23} & M_{24} \\ M_{31} & M_{32} & M_{33} & M_{34} \\ M_{41} & M_{42} & M_{43} & M_{44} \end{bmatrix}$$

$$x' = (x \times M_{11}) + (y \times M_{21}) + (z \times M_{31}) + (1 \times M_{41})$$
$$y' = (x \times M_{12}) + (y \times M_{22}) + (z \times M_{32}) + (1 \times M_{42})$$
$$z' = (x \times M_{13}) + (y \times M_{23}) + (z \times M_{33}) + (1 \times M_{43})$$

Transformation

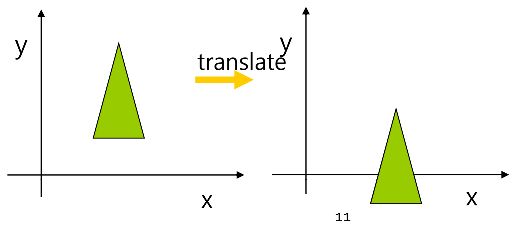
- Why 3D computer graphics uses 4x4 matrix?
  - Because it can express all kinds of transformation matrices (including translation, shearing, reflection, etc)
  - It also allows transformations to be concatenated easily (by multiplying their matrices)
- Non-homogeneous/Homogeneous coordinates convert
  - (x, y, z) → (x, y, z, 1)
  - (x/w, y/w, z/w) ← (x, y, z, w)

Translation

- Translation
  - $T^{-1}(p)=T(-p)$

// create a translation matrix
Matrix Matrix.CreateTranslation(px, py, pz);

$$T = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ px & py & pz & 1 \end{bmatrix}$$



Rotation

- Rotation
  - $R^{-1}(p)=R^{T}(p)$
  - Angle in radian

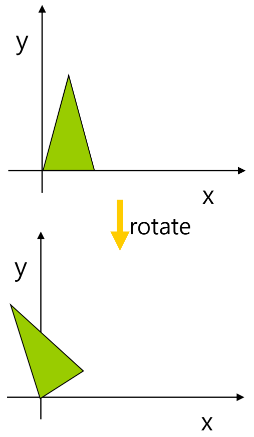
// create a matrix that can be used to rotate a set of vertices around the x/y/z-axis
Matrix Matrix.CreateRotationX(angle);
Matrix Matrix.CreateRotationY(angle);
Matrix Matrix.CreateRotationZ(angle);

3D Rotation Matrix

$$R_x(\theta)=\begin{bmatrix}1 & 0 & 0 & 0\\ 0 & \cos\theta & \sin\theta & 0\\ 0 & -\sin\theta & \cos\theta & 0\\ 0 & 0 & 0 & 1\end{bmatrix}$$

$$R_y(\theta)=\begin{bmatrix}\cos\theta & 0 & -\sin\theta & 0\\ 0 & 1 & 0 & 0\\ \sin\theta & 0 & \cos\theta & 0\\ 0 & 0 & 0 & 1\end{bmatrix}$$

$$R_z(\theta)=\begin{bmatrix}\cos\theta & \sin\theta & 0 & 0\\ -\sin\theta & \cos\theta & 0 & 0\\ 0 & 0 & 1 & 0\\ 0 & 0 & 0 & 1\end{bmatrix}$$

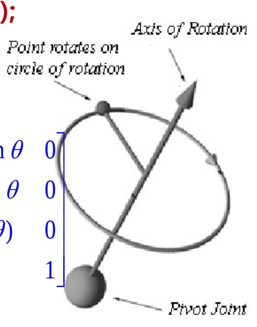


Rotation

- Rotation
  - Angle in radian

// create a matrix that rotates around an arbitrary vector
Matrix Matrix.CreateFromAxisAngle(vec, angle);

$$R_\theta=\begin{bmatrix}\cos\theta+x^2(1-\cos\theta) & xy(1-\cos\theta)+z\sin\theta & xz(1-\cos\theta)-a_y\sin\theta & 0\\ xy(1-\cos\theta)-z\sin\theta & \cos\theta+y^2(1-\cos\theta) & yz(1-\cos\theta)+x\sin\theta & 0\\ xz(1-\cos\theta)+y\sin\theta & yz(1-\cos\theta)-x\sin\theta & \cos\theta+z^2(1-\cos\theta) & 0\\ 0 & 0 & 0 & 1\end{bmatrix}$$

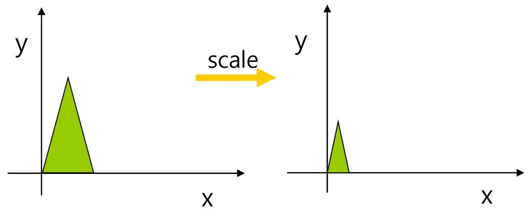


Scaling

- Scaling
  - $S^{-1}(sx, sy, sz)=S(1/sx, 1/sy, 1/sz)$

// create a scaling matrix
Matrix Matrix.CreateScale(3); // scaling in all axis by 3
Matrix Matrix.CreateScale(sx, sy, sz);

$$S=\begin{bmatrix}sx & 0 & 0 & 0\\ 0 & sy & 0 & 0\\ 0 & 0 & sz & 0\\ 0 & 0 & 0 & 1\end{bmatrix}$$



Inverse Transformation Matrix

$$T^{-1}(p)\;=\;\begin{pmatrix}1 & 0 & 0 & 0\\ 0 & 1 & 0 & 0\\ 0 & 0 & 1 & 0\\ -px & -py & -pz & 1\end{pmatrix}$$

$$S^{-1}(s)\;=\;\begin{pmatrix}1/sx & 0 & 0 & 0\\ 0 & 1/sy & 0 & 0\\ 0 & 0 & 1/sz & 0\\ 0 & 0 & 0 & 1\end{pmatrix}$$

$$R_X^{-1}(\theta)\;=\;\begin{pmatrix}1 & 0 & 0 & 0\\ 0 & \cos\theta & -\sin\theta & 0\\ 0 & \sin\theta & \cos\theta & 0\\ 0 & 0 & 0 & 1\end{pmatrix}$$

$$R_Y^{-1}(\theta)\;=\;\begin{pmatrix}\cos\theta & 0 & \sin\theta & 0\\ 0 & 1 & 0 & 0\\ -\sin\theta & 0 & \cos\theta & 0\\ 0 & 0 & 0 & 1\end{pmatrix}$$

$$R_Z^{-1}(\theta)\;=\;\begin{pmatrix}\cos\theta & -\sin\theta & 0 & 0\\ \sin\theta & \cos\theta & 0 & 0\\ 0 & 0 & 1 & 0\\ 0 & 0 & 0 & 1\end{pmatrix}$$

Composing Transformation

- For example, transforms a vector p=[5, 0, 0, 1]
  - scale 1/5
  - rotate π/4 in y-axis
  - translate (1, 2, -3)
  - Then, Q=S(1/5, 1/5, 1/5) R_y(PI/4) T(1,2,-3)
- p' = pQ = [1.707, 2, -3707, 1]

$$SR_yT = \begin{pmatrix} 1/5 & 0 & 0 & 0 \\ 0 & 1/5 & 0 & 0 \\ 0 & 0 & 1/5 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} .707 & 0 & -.707 & 0 \\ 0 & 1 & 0 & 0 \\ .707 & 0 & .707 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 1 & 2 & -3 & 1 \end{pmatrix}$$

$$= \begin{pmatrix} .1414 & 0 & -.1414 & 0 \\ 0 & 1 & 0 & 0 \\ .1414 & 0 & .1414 & 0 \\ 1 & 2 & -3 & 1 \end{pmatrix} = Q$$

Transformation

- Vector3.Transform
  - Transforms a Vector3 (or array of Vec3 by a specified Matrix or Quaternion)

Vector3 Vector3.Transform(Vector3, Matrix);
Vector3 Vector3.Transform(ref Vector3, ref Matrix, out Vector3);
Vector3 Vector3.Transform(Vector3, Quaternion);
Vector3 Vector3.Transform(ref Vector3, ref Quaternion, out Vector3);
Vector3 Vector3.Transform(Vector3[], int, ref Matrix, Vec3[], int, int);
Vector3 Vector3.Transform(Vector3[], int, ref Quaternion, Vec3[], int, int);
Vector3 Vector3.Transform(Vector3[], ref Matrix, Vector3[]);
Vector3 Vector3.Transform(Vector3[], ref Quaternion, Vector3[]);

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..

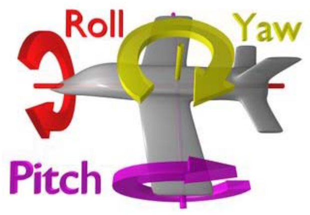
Euler Angles

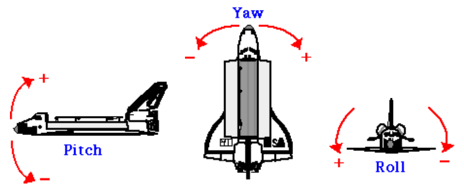
- Euler Angles
  - Represent any arbitrary orientation as three rotations about three mutually perpendicular axes (rotation about X, Y, Z)
  - Sometimes described as "Yaw, Pitch, Roll" or similar
  - A sequence of rotations around principle axes is called an ***Euler Angle Sequence***
- Axis order
  - 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), … 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 OpenGL/Direct3D/XNA





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.

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

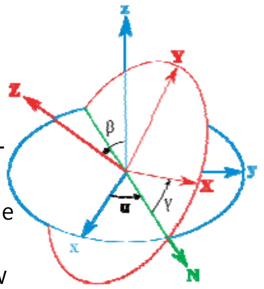
$$= \begin{bmatrix} c_y c_z & c_y s_z & -s_y \\ s_x s_y c_z - c_x s_z & s_x s_y s_z + c_x c_z & s_x c_y \\ c_x s_y c_z + s_x s_z & c_x s_y s_z - s_x c_z & c_x c_y \end{bmatrix}$$

Euler Angle Order

- 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

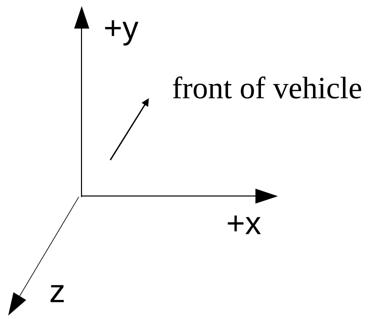
Euler Angle Order

- 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.

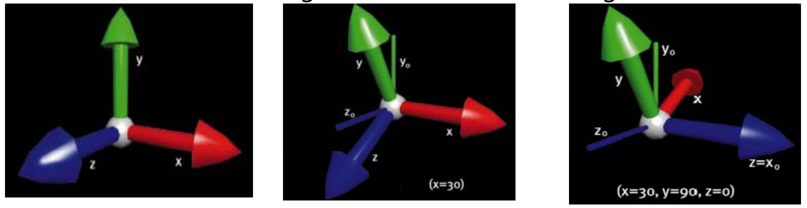


Rotations not uniquely defined with Euler Angles

- 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)

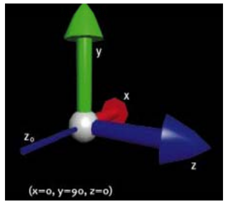
Gimbal Lock

- 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.

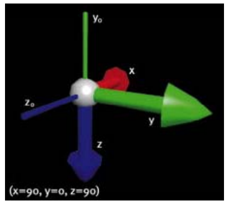


Problem with Interpolating Euler Angles

- 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)

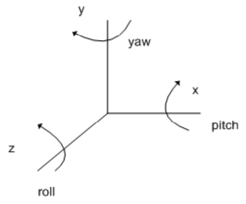
Euler Angles

- 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).**

Matrix.CreateFromYawPitchRoll

- Matrix.CreateFromYawPitchRoll

```
// Yaw/Pitch/Roll -> Rotation Matrix
Matrix Matrix.CreateFromYawPitchRoll
                         (float yaw,      // by y-axis (in radians)
                          float pitch,    // by x-axis (in radians)
                          float roll);    // by z-axis (in radians)
```



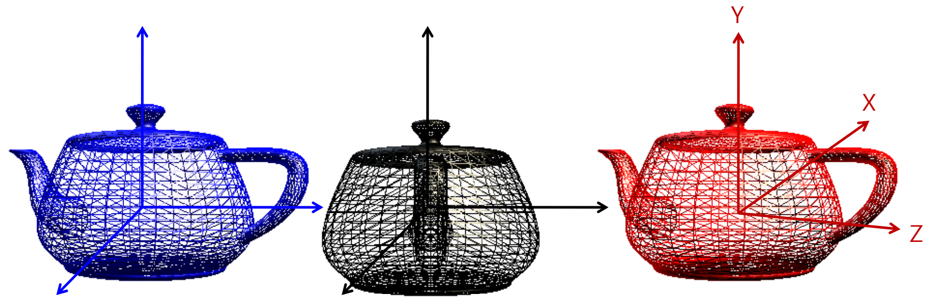
Matrix.CreateRotationX/Y/Z

- CreateFromYawPitchRoll vs. CreateRotationX/Y/Z
  - CreateFromYawPitchRoll – rotations in local coordinate system
  - CreateRotationX/Y/Z multiplication – rotations in world coordinate system

```
Matrix R1, R2, Rx, Ry, Rz;
Ry = Matrix.CreateRotationY(MathHelper.ToRadians(60.0));
Rx = Matrix.CreateRotationX(MathHelper.ToRadians(30.0));
Rz = Matrix.CreateRotationZ(MathHelper.ToRadians(45.0));
R1 = Ry * Rx * Rz;
R2 = Matrix.CreateFromYawPitchRoll(MathHelper.ToRadians(60.0),
                                   MathHelper.ToRadians(30.0),
                                   MathHelper.ToRadians(45));
```

R1 != R2

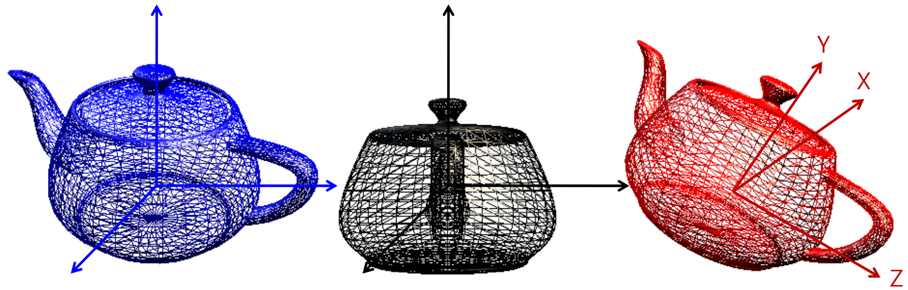
YawPitchRoll vs. RotationX/Y/Z



R1 = Y-axis rotation 60

R2 = Yaw 60

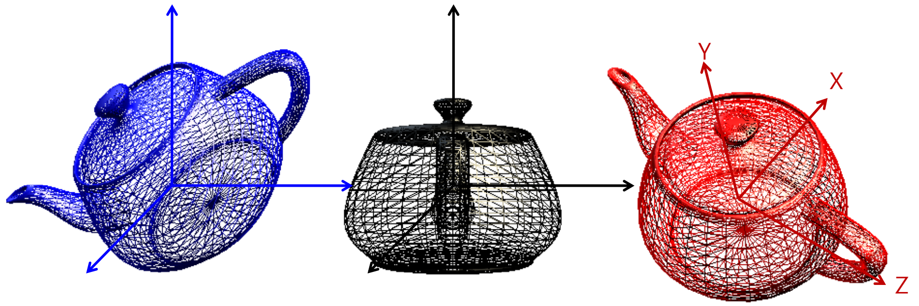
YawPitchRoll vs. RotationX/Y/Z



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 q around an arbitrary unit axis **a**:

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

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.CreateFromAxisAngle

- Matrix.CreateFromAxisAngle

```
Vector3 axis(0, 1, 0);
float angle = 60;
Matrix R3 = Matrix.CreateFromAxisAngle(axis,
                                       MathHelper.ToRadians(angle));
```

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!

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

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

$$i = jk = -kj$$

$$j = ki = -ik$$

$$k = ij = -ji$$

Quaternions (Scalar/Vector)

- Quaternions are written as the combination of a scalar value s and a vector value **v,** where

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

$$v = [x, y, z]$$

$$s = w$$

Identity Quaternions

- 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 Quaternions

- 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:

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

Quaternions as Rotations

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

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

or

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

- If **a** has unit length, then **q** will also has unit length

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

- 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 :

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

Matrix to Quaternion

- To convert a matrix to a quaternion:

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

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

- If w=0, then the division is undefined. First, determining which q0, q1,q2, q3 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 **q** represents a rotation and **q’** represents a rotation, then **qq’** represents **q** rotated by **q’**
- This follows very similar rules as matrix multiplication (I.e., non-commutative) 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 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

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

- 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)

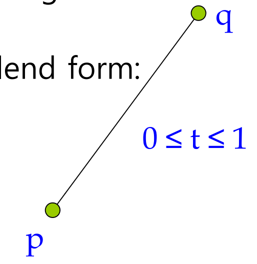
Linear Interpolation (LERP)

- If we want to do a direct interpolation between two quaternions **p** and **q** by alpha:

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

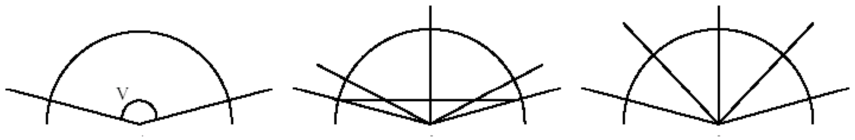
- Note that the Lerp operation can be thought of as a weighted average (convex)
- We could also write it in it's additive blend form:

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



Why SLERP?

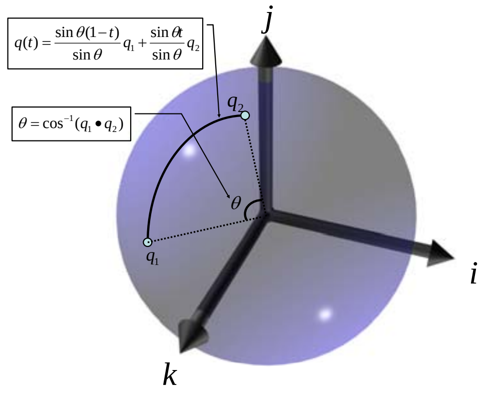
- The set of quaternions live on the unit hypersphere. The direct interpolation between quaternions would stray from the hypersphere.



- 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

- 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.'


$q(t)=\frac{\sin\theta(1-t)}{\sin\theta}q_1+\frac{\sin\theta t}{\sin\theta}q_2$
$\theta=\cos^{-1}(q_1\bullet q_2)$
j
q_2
θ
q_1
i
k

Spherical Linear Interpolation

- We define the spherical linear interpolation of two quaternions **p** and **q** by alpha:

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

$$where\ \theta=\operatorname{acos}(\mathbf{p}\cdot\mathbf{q})$$

- NOTE: if p, q are more than 90 degrees apart, it takes shorter path.

Spherical Linear Interpolation

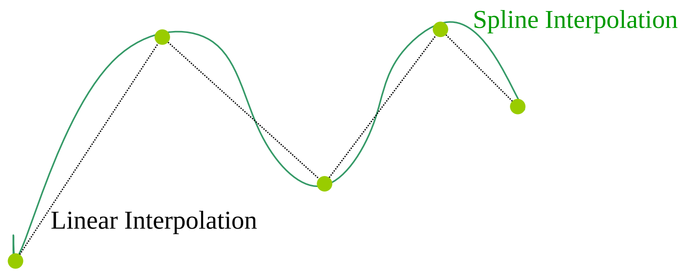
- Remember that there are two redundant vectors in quaternion space for every unique orientation in 3D space
- What is the difference between:

  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?

- 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

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

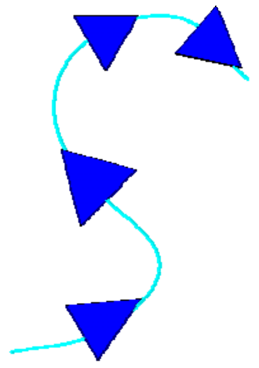
$$a_i = q_i * \exp\left(\frac{-\log(q_i^{-1} * q_{i-1}) + \log(q_i^{-1} * q_{i+1})}{4}\right)$$

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

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.

Path Animation



Path Controlled Translation & Rotation

XNA Quaternion

- Quaternion methods

```
// q* (conjugate of a quaternion)
Quaternion p;
p.Conjugate();

// pq (multiply two quaternions)
Quaternion Quaternion.Multiply(Quaternion p,
                               Quaternion q);

// p · q (dot product of two quaternions)
float Quaternion.Dot(Quaternion p,
                     Quaternion q);
```

XNA Quaternion

```
// yaw/pitch/roll -> quaternion
Quaternion Quaternion.CreateFromYawPitchRoll
                    (float yaw, float pitch, float roll);

// rotation matrix -> quaternion
Quaternion Quaternion.CreateFromRotationMatrix (Matrix matrix);

// axis/angle -> quaternion
Quaternion Quaternion.CreateFromAxisAngle
                    (Vector3 axis, float angle);
```

XNA Quaternion

```
// quaternion -> axis/angle
void QuaternionToAxisAngle(ref Quaternion q,
                           out Vector3 axis, out float angle);
{
    angle = (float)Math.Acos(q.W);
    float scale = 1.0f / (float)Math.Sin(angle);
    angle *= 2.0f;
    axis = new Vector3(-q.X * scale, -q.Y * scale, -q.Z * scale);
}

// quaternion -> rotation matrix
Matrix Matrix.CreateFromQuaternion(Quaternion quaternion);

// transform a vector by quaternion
Vector3 Vector3.Transform(Vector3 value, Quaternion rotation);
```

XNA Quaternion

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

```
Quaternion Quaternion.Slerp
                    (Quaternion quaternion1,
                     Quaternion quaternion2,
                     float amount);
```

// lerp(q_1, q_2, t) linear interpolation between two quaternions

```
Quaternion Quaternion.Lerp
                    (Quaternion quaternion1,
                     Quaternion quaternion2,
                     float amount);
```

XNA Interpolation

- // Catmull Rom Spline Interpolation

```
Vector3 Vector3.CatmullRom(Vector3 value1,
                           Vector3 value2,
                           Vector3 value3,
                           Vector3 value4,
                           float amount);
```

  // Hermite Spline Interpolation

```
Vector3 Vector3.Hermite(Vector3 value1,
                        Vector3 value2,
                        Vector3 value3,
                        Vector3 value4,
                        float amount);
```