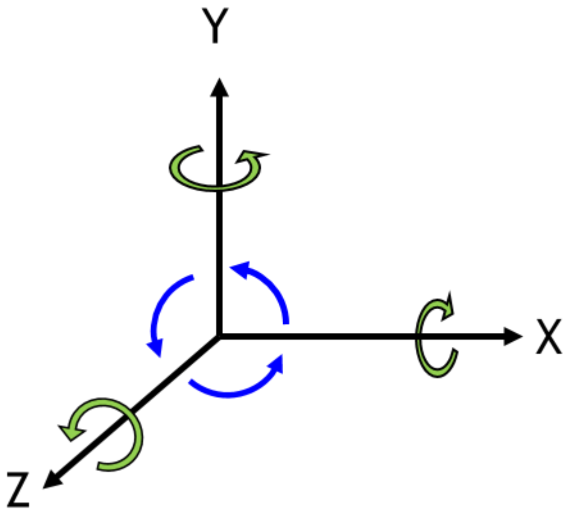
Transformation & Representing Orientations

Fall 2023
10/12/2023

Kyoung Shin Park
Computer Engineering
Dankook University

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



OpenGL

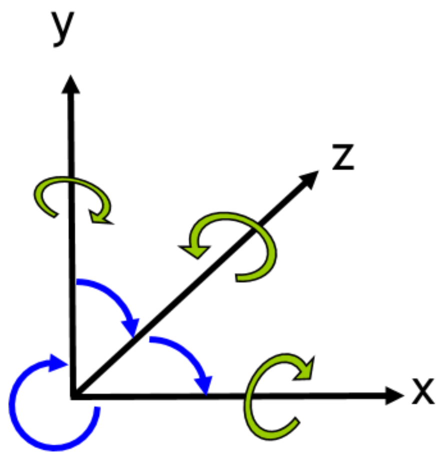
LHS Coordinate Systems

- Left Hand Coordinate System (LHS) – z+ inside the screen
- 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



Unity

Homogeneous Coordinates

- 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)

Unity Matrix

Unity Matrix4x4

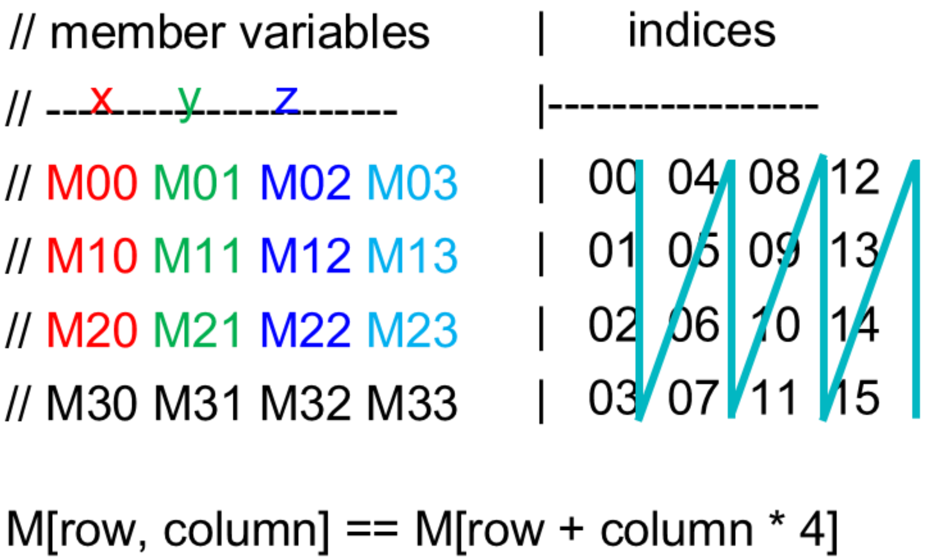
- A transformation matrix can perform arbitrary linear 3D transformations (i.e. translation, rotation, scale, shear etc.) and perspective transformations using homogenous coordinates.
- You rarely use matrices in scripts; most often using **Vector3, Quaternion**, and functionality of **Transform** class is more straightforward.

$$p' = M * p = \begin{pmatrix} m_0 & m_4 & m_8 & m_{12} \\ m_1 & m_5 & m_9 & m_{13} \\ m_2 & m_6 & m_{10} & m_{14} \\ m_3 & m_7 & m_{11} & m_{15} \end{pmatrix} \begin{bmatrix} v_0 \\ v_1 \\ v_2 \\ v_3 \end{bmatrix}$$

Unity Matrix

Unity Matrix4x4

- The elements must be specified in **column-major order**.
  - i.e. the **position** of a transformation matrix is in the **last column**, and the **first three columns** contain **x**, **y**, **and z-axis**.
  - Data is accessed as: **row + (column*4)**. Matrices can be indexed like 2D arrays but note that in an expression like mat[a, b], a refers to the row index, while b refers to the column index.



```
// member variables        |     indices
// ---x----y-----z-------  |----------------
// M00 M01 M02 M03         |  00  04  08  12
// M10 M11 M12 M13         |  01  05  09  13
// M20 M21 M22 M23         |  02  06  10  14
// M30 M31 M32 M33         |  03  07  11  15

M[row, column] == M[row + column * 4]
```

Unity Matrix Column-Major Order

- Unity uses 4x4 matrix and 4x1 vector for transformation
  - v = (2, 6, -3, 1)
  - M = translate 10 units in x-axis
  - **v′ = M** * **v** = (12, 6, -3, 1)

$$\begin{pmatrix} M00*vx+M01*vy+M02*vz+M03*vw \\ M10*vx+M11*vy+M12*vz+M13*vw \\ M20*vx+M21*vy+M22*vz+M23*vw \\ M30*vx+M31*vy+M32*vz+M33*vw \end{pmatrix} = \begin{pmatrix} M00 & M01 & M02 & M03 \\ M10 & M11 & M12 & M13 \\ M20 & M21 & M22 & M23 \\ M30 & M31 & M32 & M33 \end{pmatrix} \begin{pmatrix} vx \\ vy \\ vz \\ vw \end{pmatrix}$$

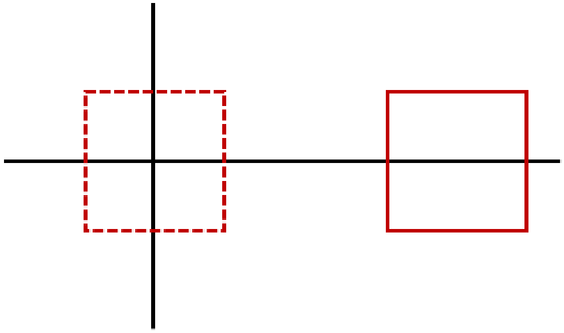
Transformation

- Geometric transformation refers to **a function that transforms a group of points describing a geometric object to new points**.
- 2D transformation
  - Translation
  - Rotation
  - Scaling

Translation

- Translation
  - Move the transform by (dx, dy, dz)
  - 2D translation uses dz = 0.0

```
// Create a translation matrix in Unity
Matrix4x4 T = Matrix4x4.Translate(new Vector3(dx, dy, dz));
```



Translation

- Translation

```
// Unity Transform.Translate
transform.position = new Vector(0, -0.5f, 0); // set position
transform.Translate(translation); // move in Vector3 translation
transform.Translate(dx, dy, dz); // move by (dx,dy,dz)

// Get matrix from the Transform
 var matrix = transform.localToWorldMatrix;
 // Get position from the matrix last column
 var position = new Vector3(matrix[0,3], matrix[1,3], matrix[2,3]);
 // Get position from the matrix last column
 var position = matrix.GetPosition();
```

Rotation

- Rotation
  - Unity uses **Quaternion** for rotation

```
// Create a rotation matrix 30 degrees around the y-axis
Quaternion rotation = Quaternion.Euler(0, 30, 0);
Matrix4x4 R = Matrix4x4.Rotate(rotation);
// Set the rotation using Euler angles (in degree)
transform.eulerAngles = new Vector3 (30, 60, 45);
transform.rotation = Quaternion.Euler(30, 60, 45);
transform.localRotation = Quaternion.Euler(30, 60, 45);
// Add an amount of rotation to an object every time it's called
transform.Rotate(new Vetor3(0, 90, 0)); // add 90 degrees to y-axis
transform.Rotate(0, 90, 0);
```

$$R^{-1}(\theta) = R(-\theta)$$

$$R^{-1}(\theta) = R^{T}(\theta)$$

Rotation

- Rotation in Z-axis

  $x' = x \cos\theta - y \sin\theta$

  $y' = x \sin\theta + y \cos\theta$

  $z' = z$

```
// Create a Rz matrix
Matrix4x4 rz = Matrix4x4.Rotate(Quaternion.Euler(0, 0, 45));
```

Rotation

- Rotation in X-axis

  $x' = x$

  $y' = y \cos\theta - z \sin\theta$

  $z' = y \sin\theta + z \cos\theta$

```
// Create a Rx matrix
Matrix4x4 rx = Matrix4x4.Rotate(Quaternion.Euler(30, 0, 0));
```

Rotation

- Rotation in Y-axis

  $x' = x \cos\theta + z \sin\theta$

  $y' = y$

  $z' = -x \sin\theta + z \cos\theta$

  ```
  // Create a Ry matrix
  Matrix4x4 ry = Matrix4x4.Rotate(Quaternion.Euler(0, 60, 0));
  ```

Rotation

- Rotation in arbitrary axis in Unity

```
// Create a Ra matrix Euler(30, 60, 45)
Matrix4x4 ra = Matrix4x4.Rotate(Quaternion.Euler(30, 60, 45));
// Create a Rb matrix rz -> rx -> ry
Matrix4x4 rz = Matrix4x4.Rotate(Quaternion.Euler(0, 0, 45));
Matrix4x4 rx = Matrix4x4.Rotate(Quaternion.Euler(30, 0, 0));
Matrix4x4 ry = Matrix4x4.Rotate(Quaternion.Euler(0, 60, 0));
Matrix4x4 rb = ry * rx * rz; // Z → X → Y
Matrix4x4 rc = rz * rx * ry; // Y → X → Z
// ra == rb matrix
if (ra == rb) Debug.Log("ra == rb");
// rb != rc matrix
if (rb != rc) Debug.Log("rb != rc");
```

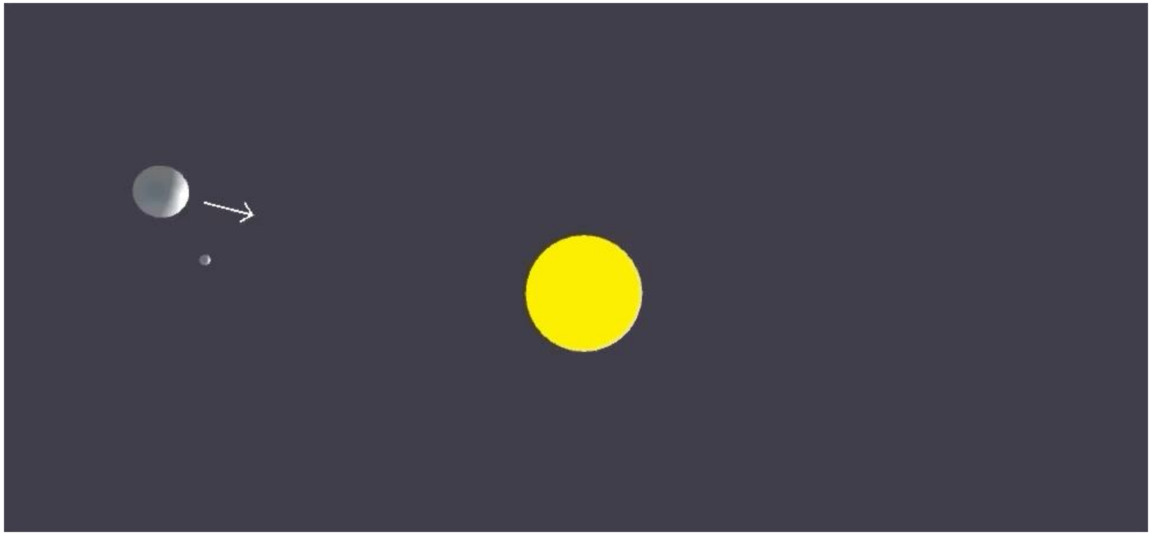
Rotation

- Local rotation vs World rotation

```
// Unity Transform.eulerAngles or Transform.localEulerAngles
Vector3 worldRotation = transform.eulerAngles;
Vector3 localRotation = transform.localEulerAngles;
```

- Transform.RotateAround

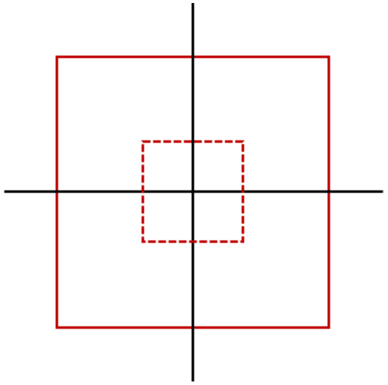
```
// Unity Transform.RotateAround to rotate an object around a
   target position in world space and the y-axis by 10 degrees
Transform.RotateAround(target.position, Vector3.up, 10 *
   Time.deltaTime);
```



Scale

- Scale
  - Transform the size by sx, sy, sz on the x-axis, y-axis, z-axis.
  - If the scale factor>1, it scales up. If 0<scale factor<=1, it scales down. If the scale factor<0, it becomes reflection.

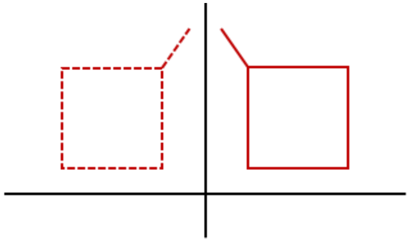
```
// Create a scaling matrix in Unity
Matrix4x4 m = Matrix4x4.Scale(new Vector(sx, sy, sz));
// Scale the transform relatives to the GameObject parent
obj.transform.localScale = new Vector(-0.5f, -0.5f, 1);
```



Reflection

- Reflection

```
// Create a reflection matrix about the yz-plane (the plane x = 0)
Matrix4x4 m = Matrix4x4.Scale(new Vector3(-1, 0, 0));
// Create a reflection matrix about the xz-plane (the plane y = 0)
Matrix4x4 m = Matrix4x4.Scale(new Vector3(0, -1, 0));
// Create a reflection matrix about the xy-plane (the plane z = 0)
Matrix4x4 m = Matrix4x4.Scale(new Vector3(0, 0, -1));
// Create a reflection matrix over (0, 0, 0)
Matrix4x4 m = Matrix4x4.Scale(new Vector3(-1, -1, -1));
```



Composing Transformation

- Composing transformation is a process of forming one transformation by applying several transformation in sequence.
- If you want to transform one point, apply one transformation at a time or multiply the matrix and then multiply this matrix by the point.

$$Q = (M3 \cdot (M2 \cdot (M1 \cdot P))) = \underbrace{M3 \cdot M2 \cdot M1}_{M} \cdot P$$

(pre-multiply)

Composing Transformation

```
// translation, rotation, scale
Vector3 translation = new Vector3(1.5, 0, 0);
Quaternion rotation = Quaternion.Euler(0, 0, 45);
Vector3 scale = new Vector3(0.2, 0.2, 0.2);
// Create a composing transformation Scale -> Rotate -> Translate
Matrix4x4 m1 = Matrix4x4.TRS(translation, rotation, scale);
// Create a composing transformation Scale -> Rotate -> Translate
Matrix4x4 m2 = Matrix4x4.Translate(translation) *
Matrix4x4.Rotate(rotation) * Matrix4x4.Scale(scale);
if (m1 == m2) Debug.Log(“m1 == m2”);
// Create a composing transformation Translate -> Rotate -> Scale
Matrix4x4 m3 = Matrix4x4. Matrix4x4.Scale(scale) *
Matrix4x4.Rotate(rotation) * Translate(translation);
if (m2 != m3) Debug.Log(“m2 != m3”);
// Get a new position by a composing transformation matrix, m1
Vector3 pos = new Vector3(5, 0, 0);
Vector3 newpos = m1.MultiplyPoint(pos);
```

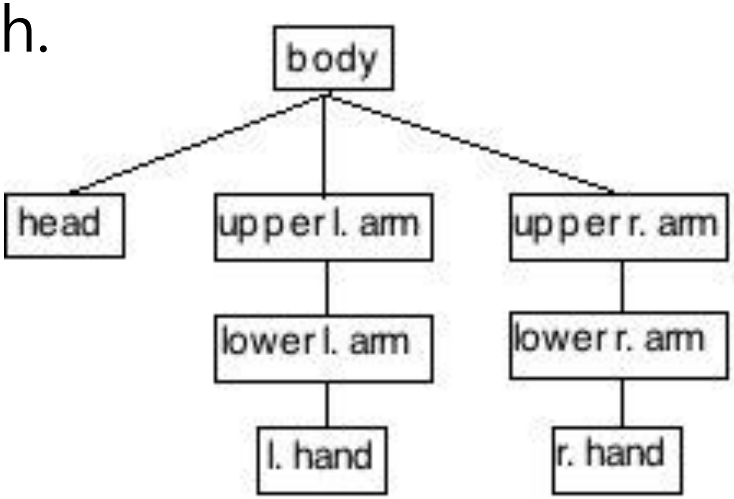
Transformation Order Matters!

- The multiplication of the transformation matrix is not commutative.
- Even if the transformation matrix is the same, it may have completely different results depending on the order of multiplication.

```
// Original cube at the origin(0, 0, 0)
RT = Rz(45) * T(1.5, 0, 0) // T first, then Rz
TR = T(1.5, 0, 0) * Rz(45) // Rz first, then T
TRS = T(1,2,-3) * Rz(45) * S(0.2, 0.2, 0.2) // S → R → T
SRT = S(0.2, 0.2, 0.2) * Rz(45) * T(1,2,-3) // T→ R → S
TRS != SRT // Transformation Matrix Order Matter!
```

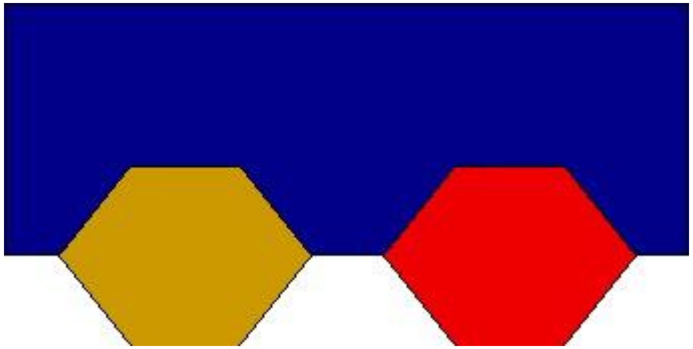
Transformation Hierarchy

- Hierarchical transformations are often expressed as a tree structure of transformations.
- To design a three-dimensional character, we use a hierarchical transformation made of rigid body parts.
- For more flexible 3D character design, a number of hierarchical transformations should be properly mixed.
- These layers are the same as the basis for the scene graph.



Hierarchical Transformation

- Hierarchical transformation can be thought of as belonging to another transformation.
- Hierarchical transformation is used as transformation of one object relative to other objects.
- For example, a car hierarchical transformation with a body and two wheels. It can be seen that when the car moves, the two wheels located in relative positions on the car body also move with the body.
- The two wheels are made to be affected by the transformation of the car body, and the wheels are not transformed separately.



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..

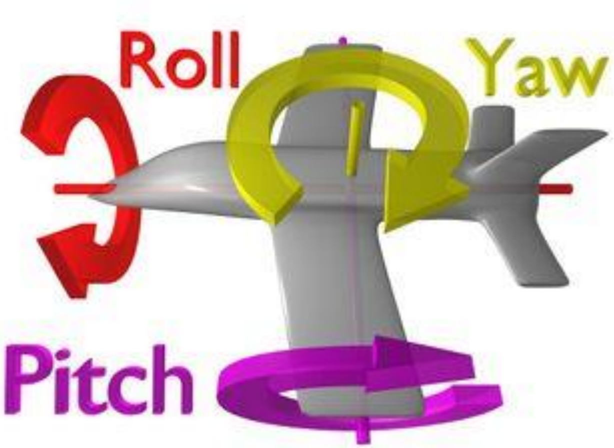
Euler Angles

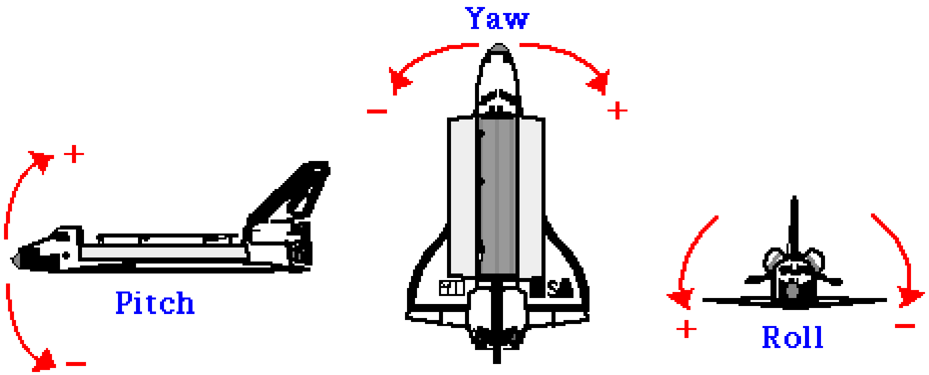
- Euler Angles
  - Represent any arbitrary orientation as three rotations about three mutually perpendicular axes (rotation about X, Y, Z)
  - Sometimes described as “Yaw, Pitch, Roll” or similar
  - A sequence of rotations around principle axes is called an ***Euler Angle Sequence***
- Axis order
 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





Euler Angles to Matrix Conversion

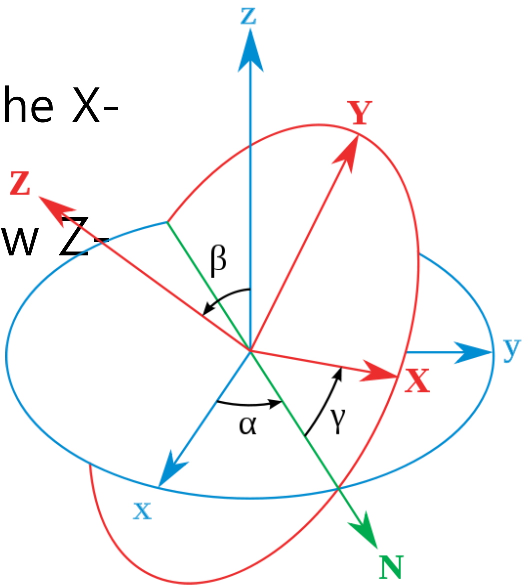
- Any orientation can be achieved by composing three elemental rotations
  - i.e., Any rotation matrix can be decomposed as a product of three elemental rotation matrices.

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

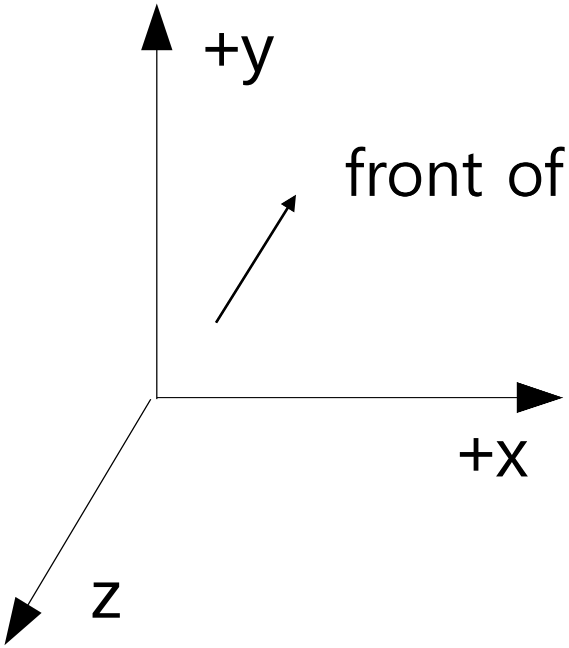
Euler Angle Order

- 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.

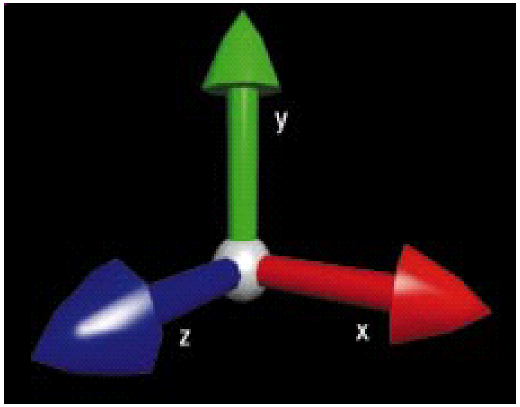


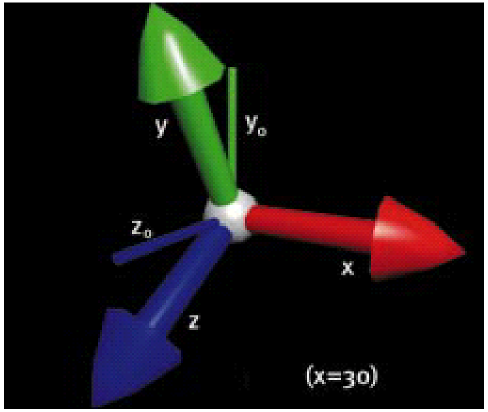
Rotations not uniquely defined with Euler Angles

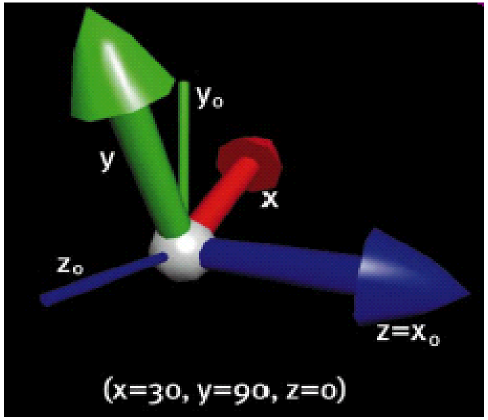
- Rotations are not uniquely defined with Euler Angles.
- Cartesian coordinates are independent of each other.
  - Arbitrary position = x-axis position + y-axis position + z-axis position
- **Euler angles do not act independently of each other.**
  - Arbitrary orientation = x-axis rotation matrix * y-axis rotation matrix * z-axis rotation matrix
  - For example, (θx, θy, θz) (180, 0, 180) == (0, 180, 0)

Gimbal Lock

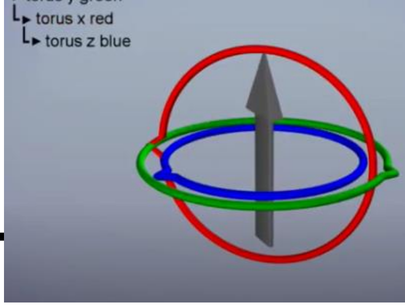
- One potential problem is '**gimbal lock**'.
- '**Gimbal Lock**' results when two axes effectively line up, resulting in a temporary loss of a degree of freedom. Change to one of the angles affect to the entire system.
  - This is related to the singularities in longitude that you get at the north and 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.



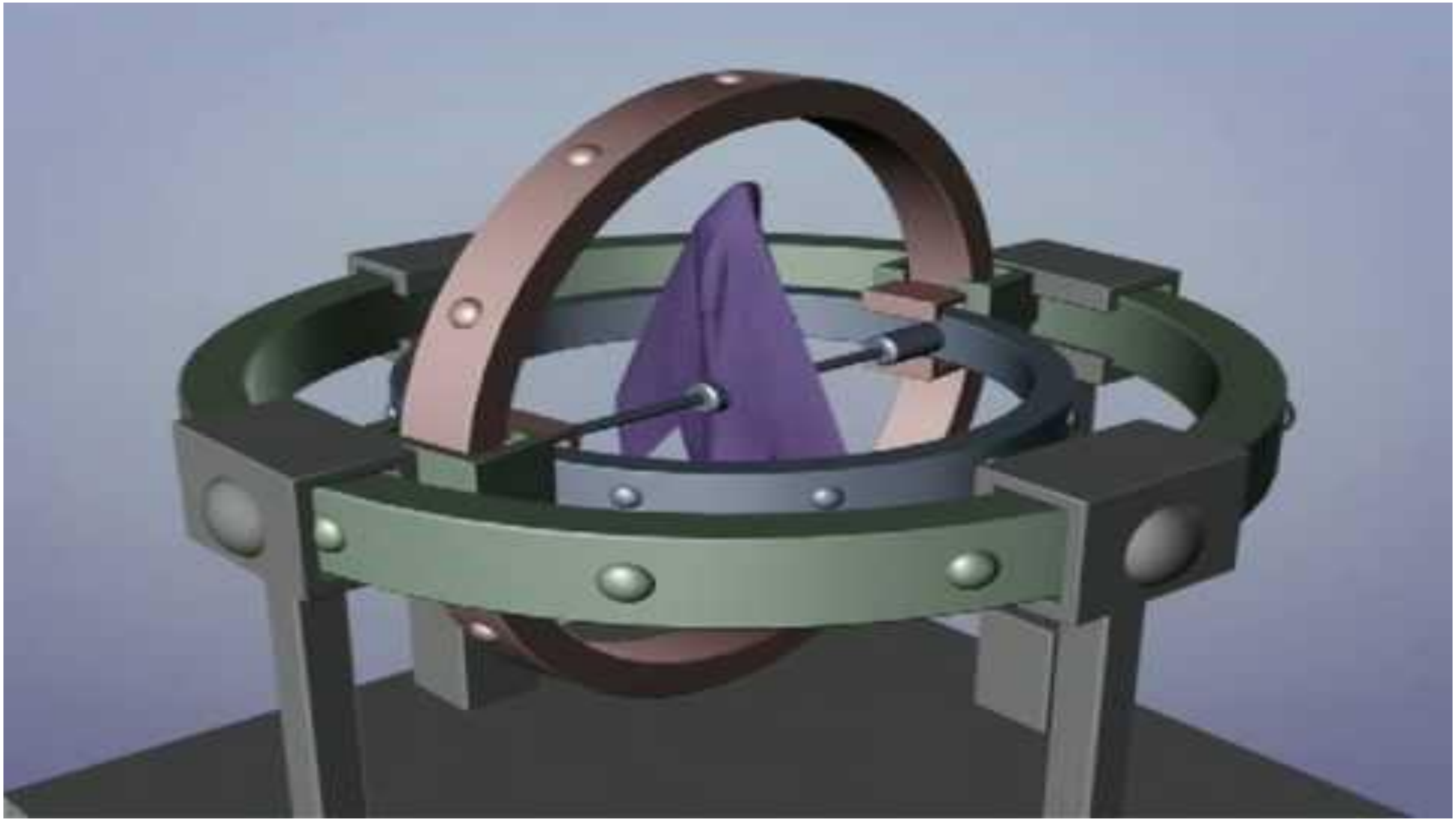




Gimbal Lock

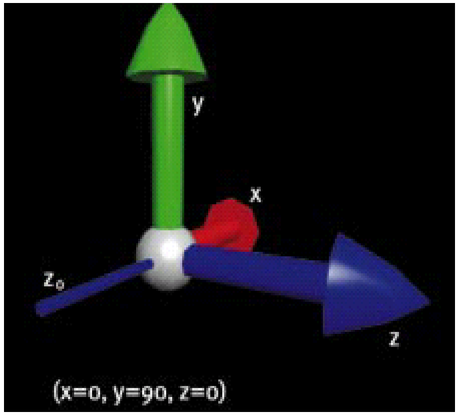


- https://www.youtube.com/watch?v=zc8b2Jo7mno

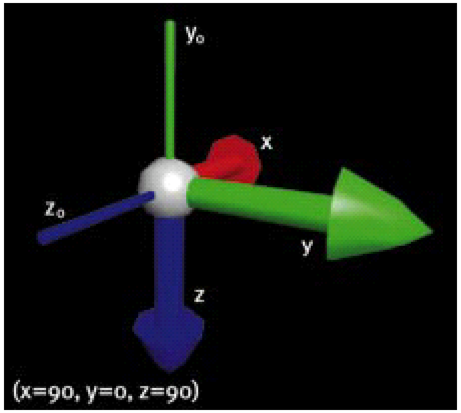


Problem with Interpolating Euler Angles

- 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

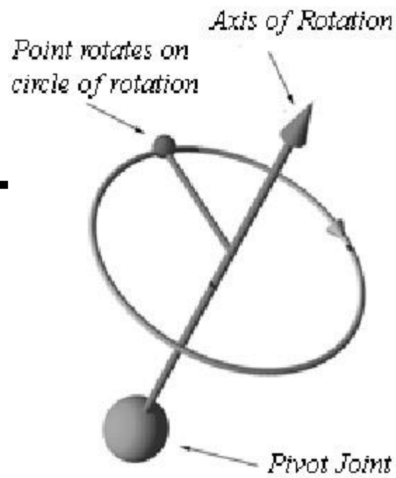
- Euler angles are used in a lot of applications, but they tend to require some rather arbitrary decisions.
- They also do not interpolate in a consistent way (but this isn’t always bad).
- **They can suffer from Gimbal lock and related problems.**
- There is no simple way to concatenate rotations.
-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

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**:



$$R = \mathrm{I}\cos\theta + \mathbf{Symmetric}\ \ (1-\cos\theta) + \mathbf{Skew}\ \ \sin\theta$$

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

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

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 **v,** where

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

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

$$s = w$$

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:

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

Quaternion as Rotations

- A quaternion can represent a rotation by an angle **θ** around a unit axis **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 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:

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

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

- 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

$$\mathbf{q}\mathbf{q}' = (xi + yj + zk + w)(x'i + y'j + z'k + w')$$

$$= \langle s\mathbf{v}' + s'\mathbf{v} + \mathbf{v}'\times\mathbf{v}, ss' - \mathbf{v}\cdot\mathbf{v}' \rangle$$

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

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)

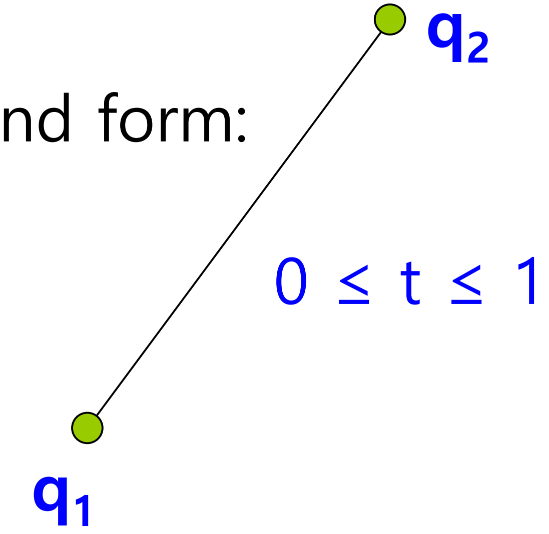
- If we want to do a direct interpolation between two quaternions **p** and **q** by alpha:

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

$$\text{where } 0 \le t \le 1$$

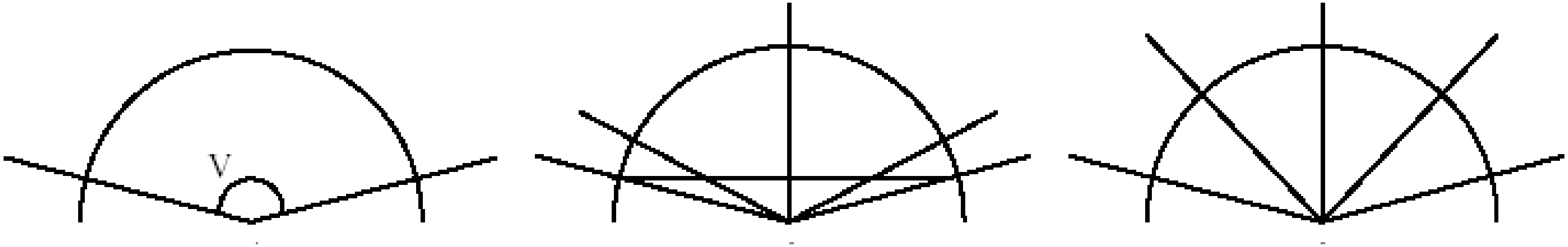
- Note that the Lerp operation can be thought of as a weighted average (convex)
- We could also write it in it’s additive blend form:

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



Why SLERP?

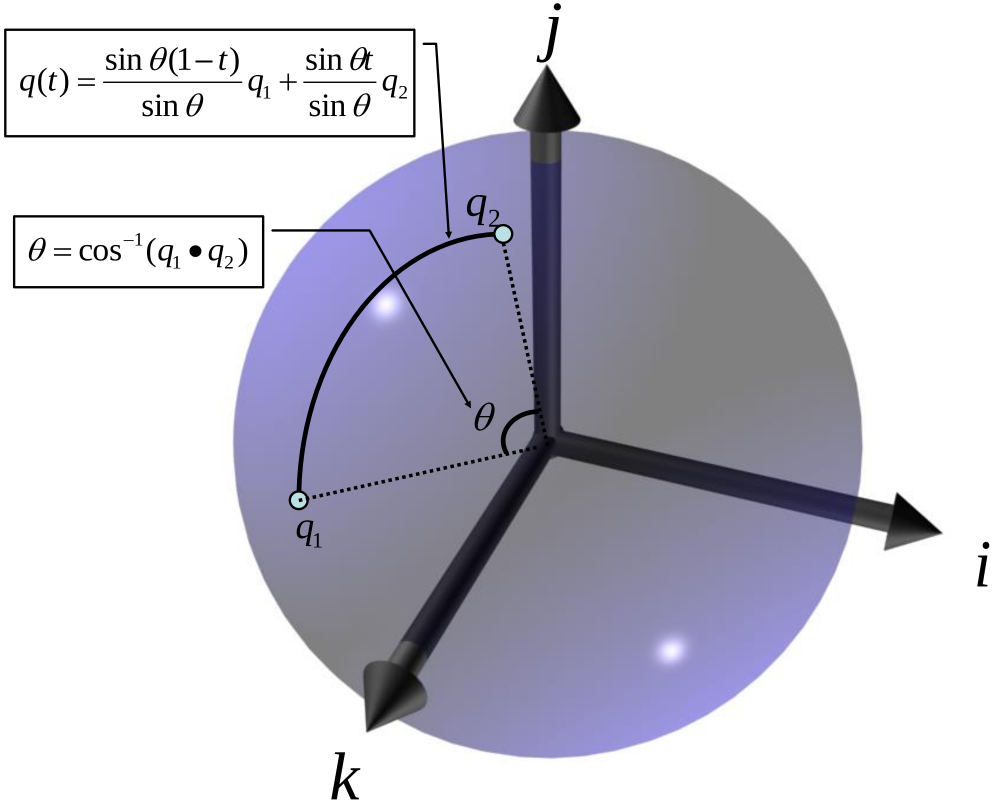
- The set of quaternions live on the unit hypersphere. The direct interpolation between quaternions would stray from the hypersphere.



- 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.'



$$q(t) = \frac{\sin\theta(1-t)}{\sin\theta} q_1 + \frac{\sin\theta t}{\sin\theta} q_2$$

$$\theta = \cos^{-1}(q_1 \bullet q_2)$$

Spherical Linear Interpolation

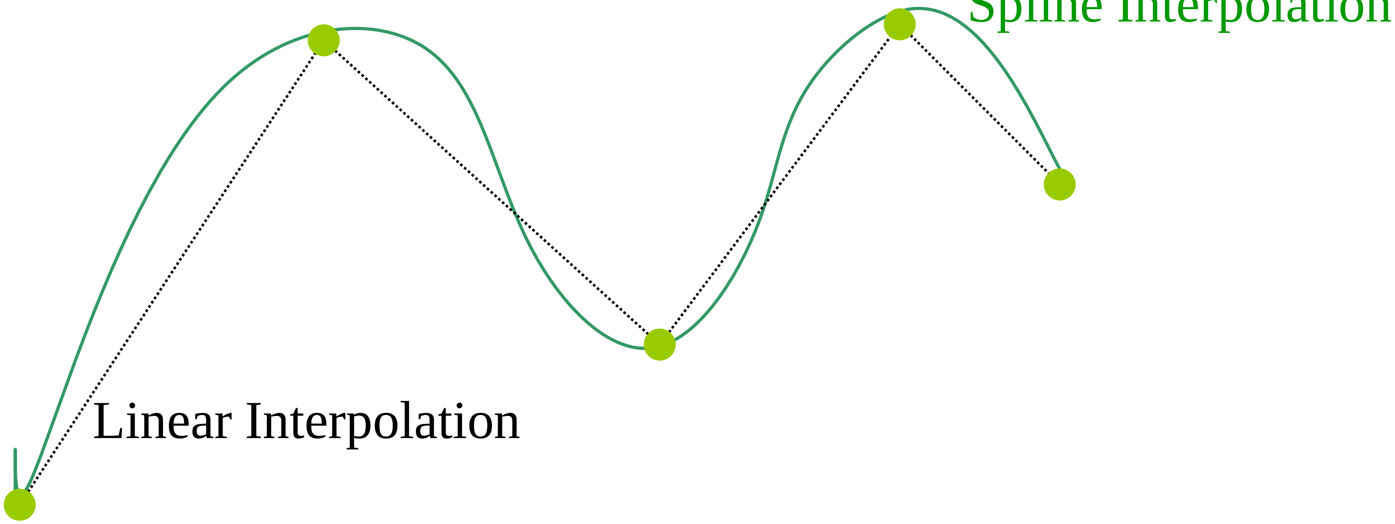
- Remember that there are two redundant vectors in quaternion space for every unique orientation in 3D space
- What is the difference between:

  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

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

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

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

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

  - a_i, a_{i+1} are inner quadrangle quaternions between q1 and q2. And you have to choose carefully so that continuity is guaranteed across segments.

Unity Quaternion

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

// yaw(y)/pitch(x)/roll(z) -> quaternion
static Quaternion Euler(float x, float y, float z);

// axis/angle -> quaternion
static Quaternion AxisAngle(float angle, Vector3 axis);

// lookat(forward) -> quaternion
static Quaternion LookRotation(Vector3 forward, Vector3 upward =
    Vector3.up);

// fromDirection/toDirection -> quaternion
static Quaternion FromToRotation(Vector3 from, Vector3 to);
```

Unity 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);
```

Example: SimpleTransform

- Create an EmptyGameObject
- Add a C# script component, **CubeTransform**, on an EmptyGameObject
  - White cube (original)

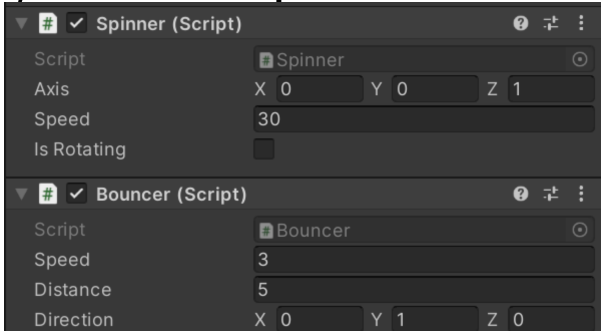
  - Red cube with T->R->S
  - Blue cube with S->R->T

  - Green cube with Rz->Rx->Ry
  - Cyan cube with Ry->Rx->Rz

  - Yellow cube with euler1 (0, 180, 0) -> euler2 (180, 0, 180)
  - Magenta cube with quat1 180도 (0, 1, 0) -> quat2 180도 (1, 0, 1)

Example: SimpleTransform

- Create Cube, Plane, Capsule
  - Cube (0, 0, 0)
  - Plane Position (0, -1.5 0) Scale (10, 10, 10)
  - Capsule Position (-6, 1, -3)
- Add a C# script component, **Mover**, on a cube
  - W/S-key to move forward/backward
  - A/D-key to pan left/right
  - L/R-key to lookat camera or rotate around camera
- Add a C# script component, **Spinner**, on a capsule
- Add a C# script component, **Bouncer**, on a capsule.



☑ Spinner (Script)

Script: Spinner

Axis: X 0 Y 0 Z 1

Speed: 30

Is Rotating: □

☑ Bouncer (Script)

Script: Bouncer

Speed: 3

Distance: 5

Direction: X 0 Y 1 Z 0

```
public class Mover: MonoBehaviour {
    void Update () {
        if (Input.GetKey(KeyCode.W)) {
            transform.Translate(10 * Vector3.forward * Time.deltaTime);
        } else if (Input.GetKey(KeyCode.S)) {
            transform.Translate(10 * Vector3.back * Time.deltaTime);
        } else if (Input.GetKey(KeyCode.A)) {
            transform.Rotate(-90 * Vector3.up * Time.deltaTime);
        } else if (Input.GetKey(KeyCode.D)) {
            transform.Rotate(90 * Vector3.up * Time.deltaTime);
        } else if (Input.GetKey(KeyCode.L)) {
            transform.LookAt(Camera.main.transform.position);
        } else if (Input.GetKey(KeyCode.R)) {
            isRotating = !isRotating;
        } if (isRotating) {
            transform.RotateAround(Camera.main.transform.position, Vector3.up,
90 * Time.deltaTime);
        }
    }
}
```

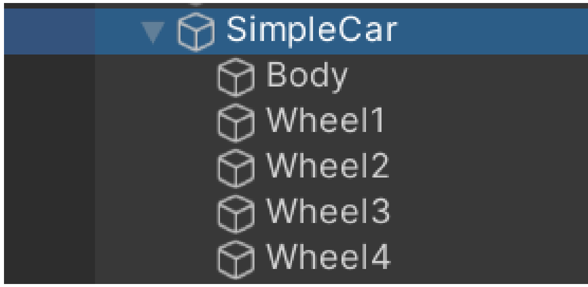
```
public class Spinner : MonoBehaviour {
    public Vector3 axis = new Vector3(0, 0, 1f);
    public float speed = 30.0f;
    public bool isRotating = false;

    // Update is called once per frame
    void Update () {
        if (Input.GetKey(KeyCode.Space)) {
            isRotating = !isRotating;
        }
        // Rotate the object around its local coordinate system
        transform.Rotate(axis, speed * Time.deltaTime);
    }
}
```

```
public class Bouncer : MonoBehaviour {
    public float speed = 3.0f;
    public float distance = 5f;
    public Vector3 direction = Vector3.up; // move up/down
    Vector3 pos;
    void Start() {
        pos = transform.position;
    }
    void Update() {
        // Calculate what the new XYY position will be
        float newX = Mathf.Sin(Time.time * speed) * distance * direction.x + pos.x;
        float newY = Mathf.Sin(Time.time * speed) * distance * direction.y + pos.y;
        float newZ = Mathf.Sin(Time.time * speed) * distance * direction.z + pos.z;
        // Set the object's position to the new calculated XYZ
        transform.position = new Vector3(newX, newY, newZ);
    }
}
```

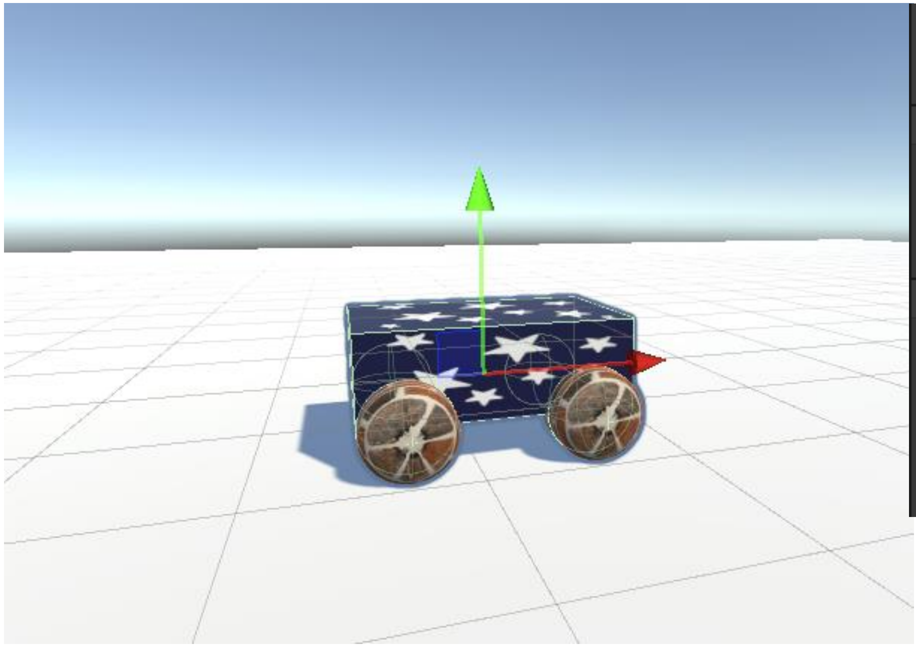
Example: SimpleCar

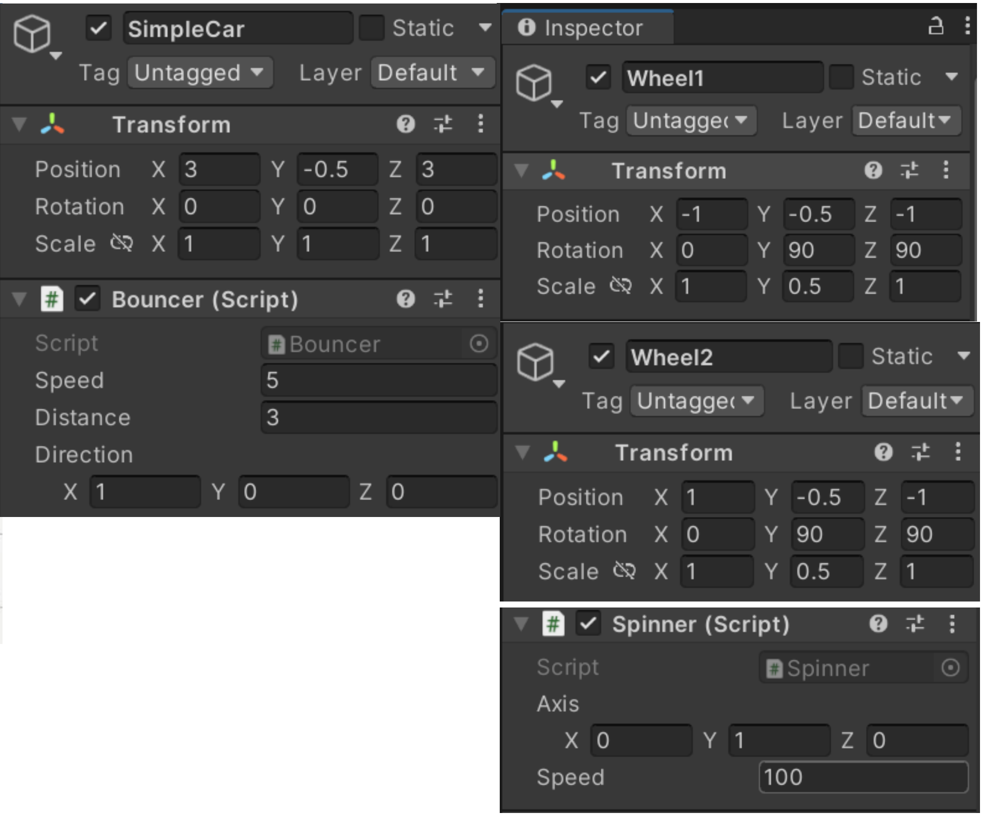
- Create Plane, EmptyGameObject (called SimpleCar)
  - Plane Position (0, -1.5 0) Scale (10, 10, 10)
  - Add a C# script component, **Bouncer**, on SimpleCar.
    - speed = 5
    - distance = 3
    - **direction (1, 0, 0)** // left/right movement



- Create Cube (called Body), Capsule (called Wheel1/2/3/4) under SimpleCar
  - Body Scale (3, 1, 2)
  - Wheel1 Position(-1, -0.5, -1) Rotation(0, 90, 90) Scale(1, 0.5, 1)
  - Wheel2 Position(1, -0.5, -1) Rotation(0, 90, 90) Scale(1, 0.5, 1)
  - Wheel3 Position(-1, -0.5, 1) Rotation(0, 90, 90) Scale(1, 0.5, 1)
  - Wheel4 Position(1, -0.5, 1) Rotation(0, 90, 90) Scale(1, 0.5, 1)
  - Add a C# script component, **Spinner** on wheel1/2/3/4
    - speed = 100

Example: SimpleCar





☑ SimpleCar □ Static
Tag Untagged Layer Default

Transform
Position X 3 Y -0.5 Z 3
Rotation X 0 Y 0 Z 0
Scale X 1 Y 1 Z 1

☑ Bouncer (Script)
Script Bouncer
Speed 5
Distance 3
Direction
X 1 Y 0 Z 0

Inspector

☑ Wheel1 □ Static
Tag Untagged Layer Default

Transform
Position X -1 Y -0.5 Z -1
Rotation X 0 Y 90 Z 90
Scale X 1 Y 0.5 Z 1

☑ Wheel2 □ Static
Tag Untagged Layer Default

Transform
Position X 1 Y -0.5 Z -1
Rotation X 0 Y 90 Z 90
Scale X 1 Y 0.5 Z 1

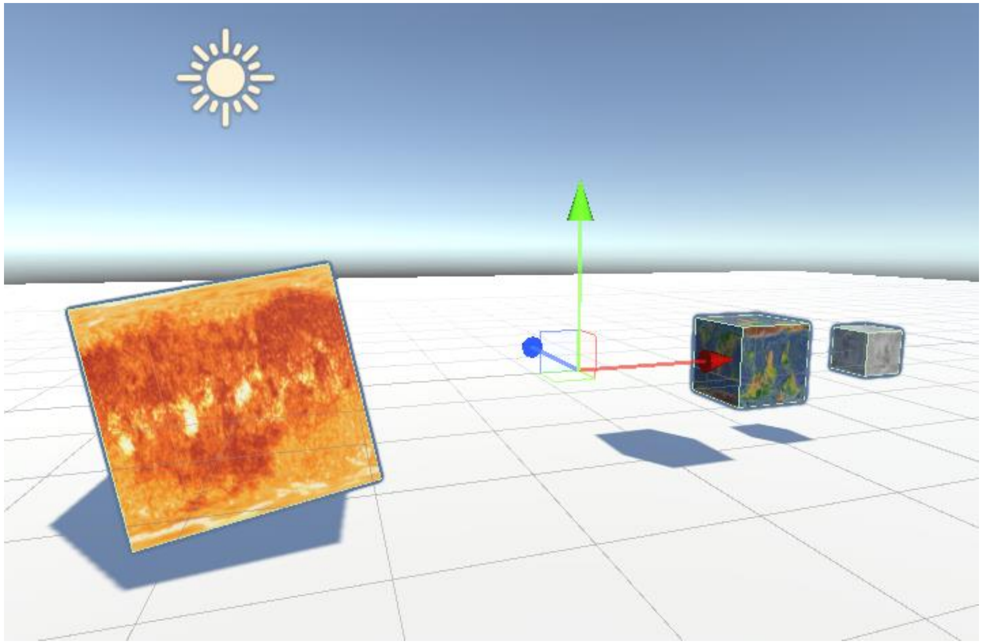
☑ Spinner (Script)
Script Spinner
Axis
X 0 Y 1 Z 0
Speed 100

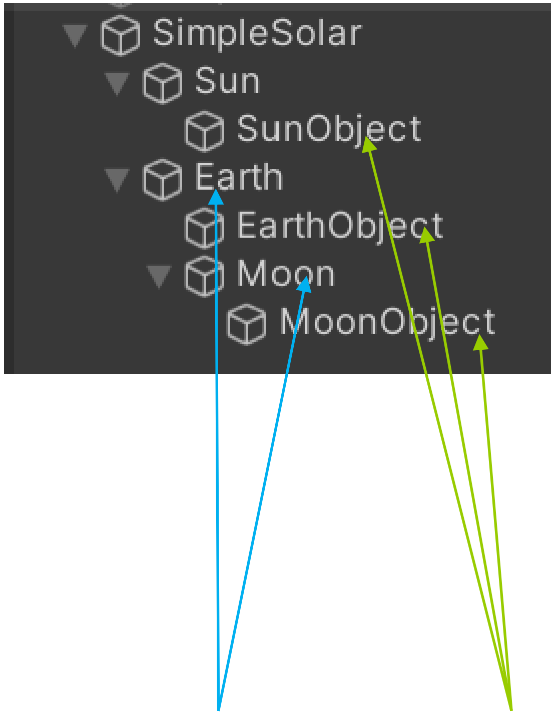
Example: SimpleSolar

- Create EmptyGameObject(SimpleSolar)
- Create EmptyGameObject(Sun) Rotation(10, 0, 10)
  - Under Sun, Create Cube(SunObject) Scale (2,2,2)
    - Add **Spinner** script on SunObject, Speed=10
- Create EmptyGameObject(Earth) Position(6, 0, 0)
  - Add **Orbit** script on Earth(ObjectToOrbit=Sun, Speed=40)
  - Under Earth, Create Cube(EarthObject)
    - Add **Spinner** script on EarthObject, speed=60
  - Under Earth, Create EmptyGameObject(Moon) Position(2, 0, 0)
    - Add **Orbit** script on Moon(ObjectToOrbit=Earth, Speed=90)
    - Under Moon, Create Cube(MoonObject) Scale(0.6, 0.6, 0.6)
      - Add **Spinner** script on MoonObject, speed=120

```
public class Orbit : MonoBehaviour {
    float angle;  Vector3 direction;
    float radius; // orbit distance
    public GameObject objectToOrbit; // object that we will orbit around
    public float speed = 30.0f; // orbit degrees per second
    void Start () {
        direction = transform.position - objectToOrbit.transform.position;
        radius = Vector3.Distance(objectToOrbit.transform.position,
transform.position);
    }
    void Update () {
        angle += speed * Time.deltaTime;
        if (angle > 360) angle -= 360;
        Vector3 orbit = Vector3.forward * radius;
        orbit = Quaternion.LookRotation(direction) * Quaternion.Euler(0, angle,
0) * orbit;
        transform.position = objectToOrbit.transform.position + orbit;
    }
}
```

Example: SimpleSolar





SimpleSolar
Sun
SunObject
Earth
EarthObject
Moon
MoonObject

Orbit

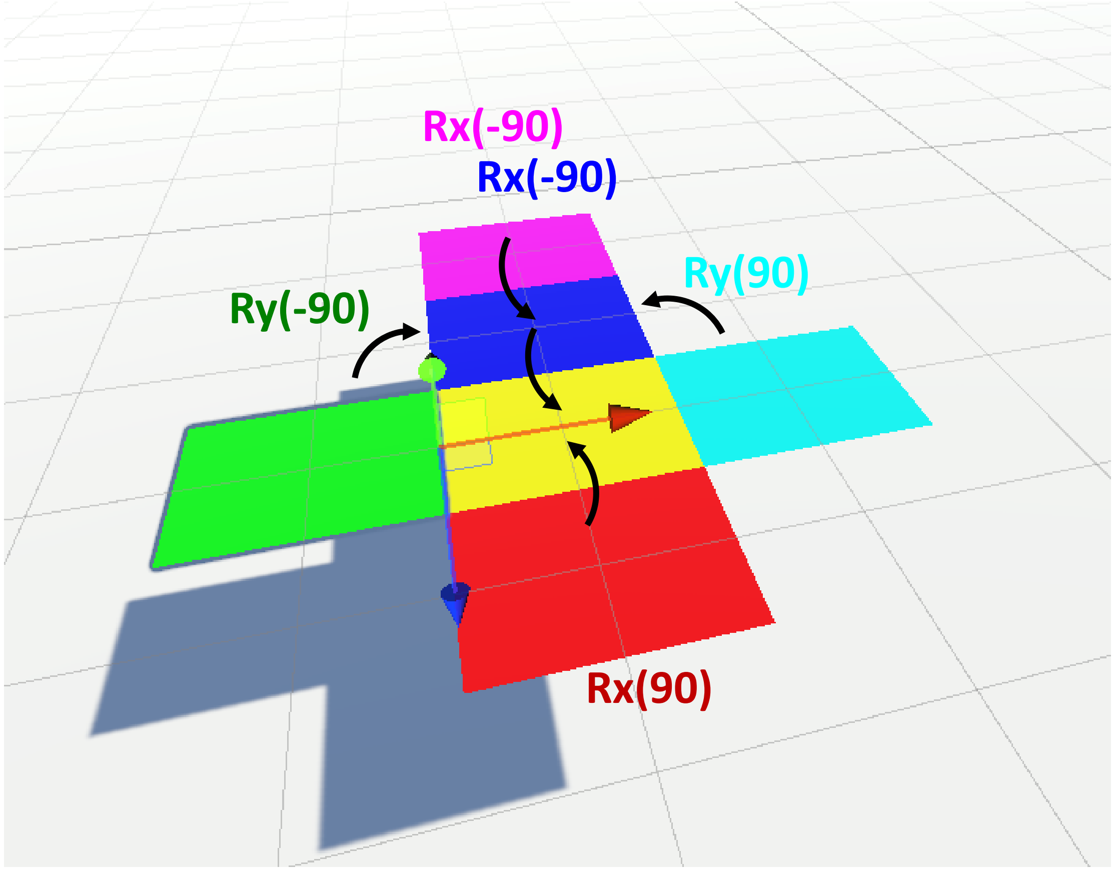
Spinner

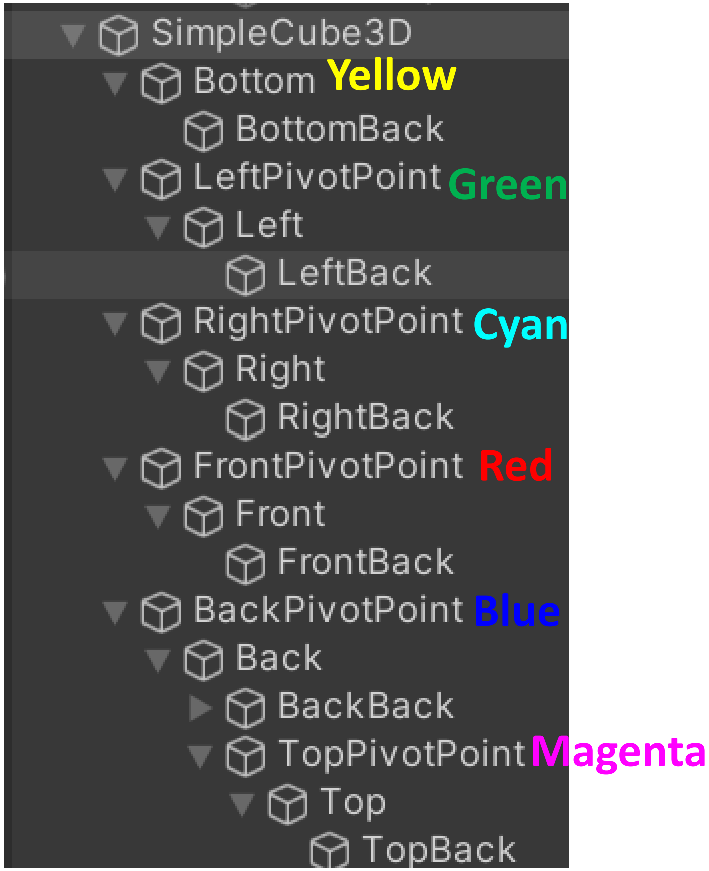
Example: SimpleCube3D

- Create EmptyGameObject(SimpleCube3D) **Rotation(90,0,0)**
- Create **Quad** for **Bottom**, **Left**, **Right**, **Front**, **Back**, **Top** (Create quads for Back-face culling)
- Create EmptyParent(LeftPivotPoint) for Left
  - LeftPivotPoint Position(-0.5,0,0), Left Position(-0.5,0,0)
  - Add **RotateFromTo** on LeftPivotPoint, From(0,0,0) **To(0,-90,0)**
- RightPivotPoint Position(0.5,0,0), Right Position(0.5,0,0)
  - Add **RotateFromTo** on RightPivotPoint, From(0,0,0) **To(0,90,0)**
- FrontPivotPoint Position(0,-0.5,0), Front Position(0,-0.5,0)
  - Add **RotateFromTo** on FrontPivotPoint, From(0,0,0) **To(90,0,0)**
- BackPivotPoint Position(0,0.5,0), Back Position(0,0.5,0)
  - Add **RotateFromTo** on BackPivotPoint, From(0,0,0) **To(-90,0,0)**
  - TopPivotPoint Position(0,0.5,0), Top Position(0,0.5,0)
    - Add **RotateFromTo** on TopPivotPoint, From(0,0,0) **To(-90,0,0)**

```
public class RotateFromTo : MonoBehaviour {
    public bool state = false;
    public float smoot = 2f;
    public Vector3 fromRotation = Vector3.zero;
    public Vector3 toRotation = Vector3.zero;
    void Update()  {
        if (state)  {
            Quaternion targetRotation = Quaternion.Euler(toRotation.x, toRotation.y,
toRotation.z);
            transform.localRotation = Quaternion.Slerp(transform.localRotation,
targetRotation, smoot * Time.deltaTime);
        }  else {
            Quaternion targetRotation2 = Quaternion.Euler(fromRotation.x,
fromRotation.y, fromRotation.z);
            transform.localRotation = Quaternion.Slerp(transform.localRotation,
targetRotation2, smoot * Time.deltaTime);
        }
        if (Input.GetKey(KeyCode.Space)) state = !state; // toggle state
    }
}
```

Example: SimpleCube3D





- SimpleCube3D
  - Bottom Yellow
    - BottomBack
  - LeftPivotPoint Green
    - Left
      - LeftBack
  - RightPivotPoint Cyan
    - Right
      - RightBack
  - FrontPivotPoint Red
    - Front
      - FrontBack
  - BackPivotPoint Blue
    - Back
      - BackBack
      - TopPivotPoint Magenta
        - Top
          - TopBack

Hierarchical Transformation

- The Hierarchy and Parent-child relationships – Unity Official Tutorials https://www.youtube.com/watch?v=0ZDZaKrofmc