목록

유니티(Unity)를 활용한

그래픽스 프로그래밍

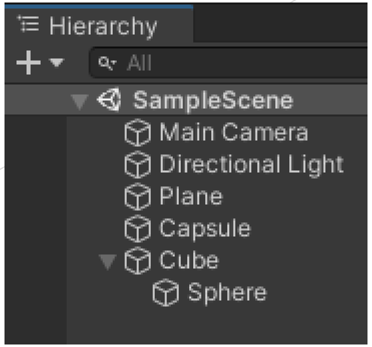
07 Transformation & Representing Orientations

3

Transformation in Unity

Transformation in Unity

- Create Plane, Capsule, Cube
  - Plane Scale (10, 10, 10)
  - Capsule Position (0, 1, 0)
  - Cube Position (5, 1, 0)
- Create Sphere under Cube
  - Sphere Scale (0.7, 0.7, 0.7) Position (1, 0, 0)


Hierarchy

All

SampleScene

Main Camera

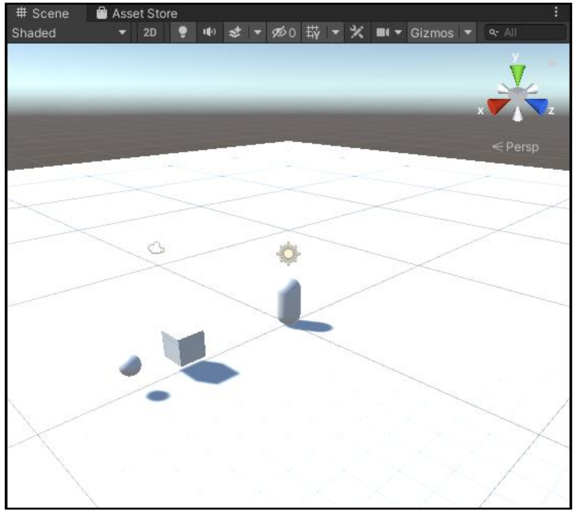
Directional Light

Plane

Capsule

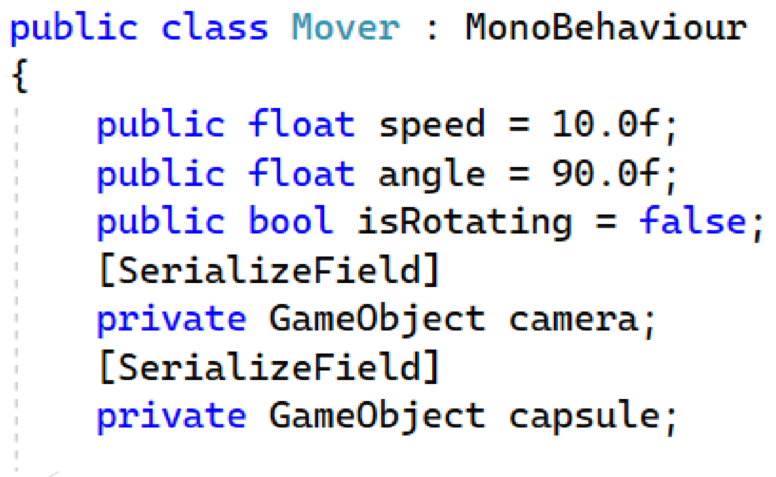
Cube

Sphere



Transformation in Unity

- Add a C# script component, called Mover, on a cube.

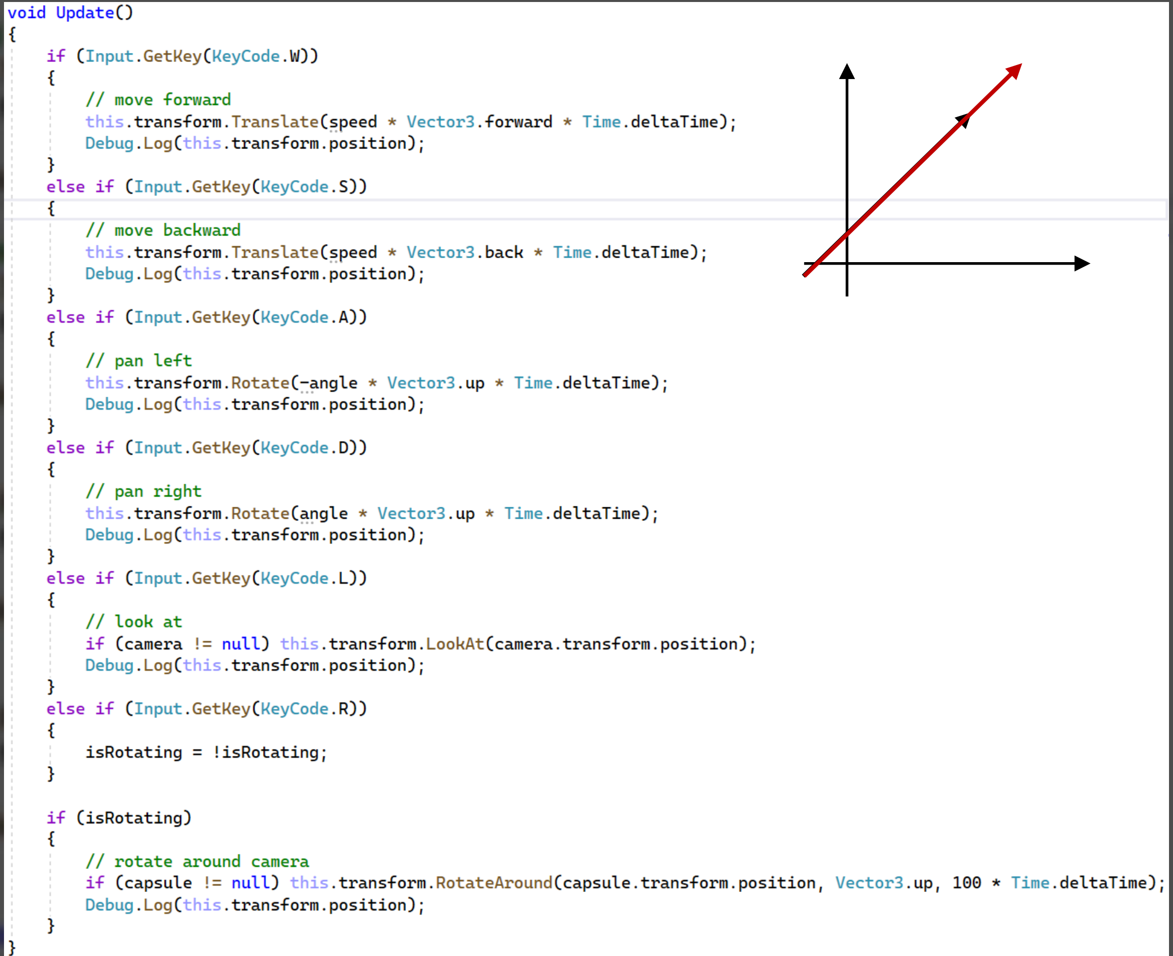


```
public class Mover : MonoBehaviour
{
    public float speed = 10.0f;
    public float angle = 90.0f;
    public bool isRotating = false;
    [SerializeField]
    private GameObject camera;
    [SerializeField]
    private GameObject capsule;
```

Drag and drop “MainCamera” (in Hierarchy view) to “camera” (in inspector view) of Mover

Drag and drop “Capsule” (in Hierarchy view) to “capsule” (in inspector view) of Mover

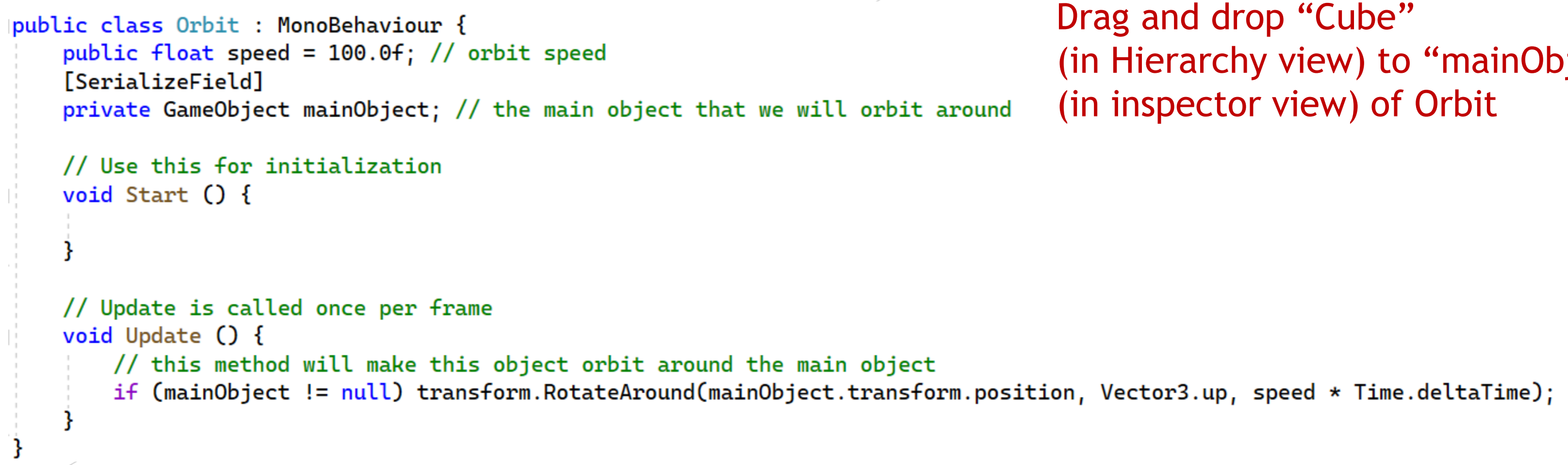
Transformation in Unity



```
void Update()
{
    if (Input.GetKey(KeyCode.W))
    {
        // move forward
        this.transform.Translate(speed * Vector3.forward * Time.deltaTime);
        Debug.Log(this.transform.position);
    }
    else if (Input.GetKey(KeyCode.S))
    {
        // move backward
        this.transform.Translate(speed * Vector3.back * Time.deltaTime);
        Debug.Log(this.transform.position);
    }
    else if (Input.GetKey(KeyCode.A))
    {
        // pan left
        this.transform.Rotate(-angle * Vector3.up * Time.deltaTime);
        Debug.Log(this.transform.position);
    }
    else if (Input.GetKey(KeyCode.D))
    {
        // pan right
        this.transform.Rotate(angle * Vector3.up * Time.deltaTime);
        Debug.Log(this.transform.position);
    }
    else if (Input.GetKey(KeyCode.L))
    {
        // look at
        if (camera != null) this.transform.LookAt(camera.transform.position);
        Debug.Log(this.transform.position);
    }
    else if (Input.GetKey(KeyCode.R))
    {
        isRotating = !isRotating;
    }

    if (isRotating)
    {
        // rotate around camera
        if (capsule != null) this.transform.RotateAround(capsule.transform.position, Vector3.up, 100 * Time.deltaTime);
        Debug.Log(this.transform.position);
    }
}
```

Transformation in Unity

- Add a C# script component, called Orbit, on a sphere.

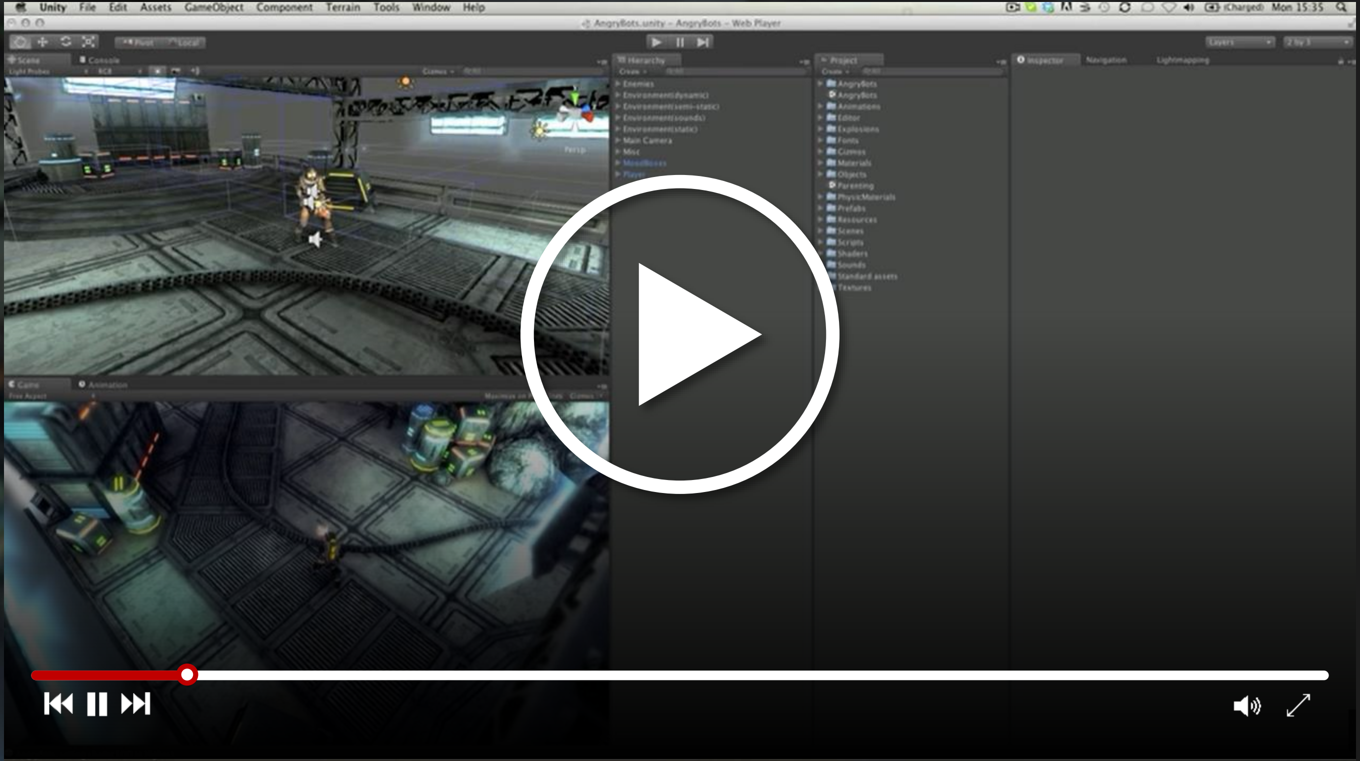


```
public class Orbit : MonoBehaviour {
    public float speed = 100.0f; // orbit speed
    [SerializeField]
    private GameObject mainObject; // the main object that we will orbit around

    // Use this for initialization
    void Start () {

    }

    // Update is called once per frame
    void Update () {
        // this method will make this object orbit around the main object
        if (mainObject != null) transform.RotateAround(mainObject.transform.position, Vector3.up, speed * Time.deltaTime);
    }
}
```

Drag and drop “Cube”
(in Hierarchy view) to “mainObject”
(in inspector view) of Orbit

Hierarchical Transformation

- The Hierarchy and Parent-child relationships - Unity Official Tutorials



영상 출처 : https://www.youtube.com/watch?v=0ZDZaKrofmc

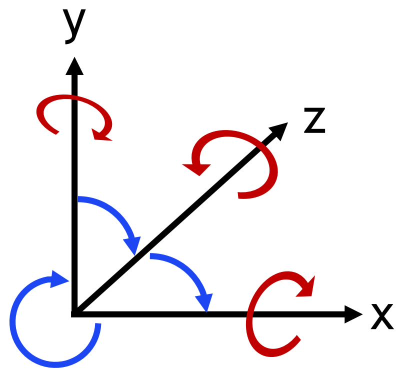
유니티(Unity)를 활용한

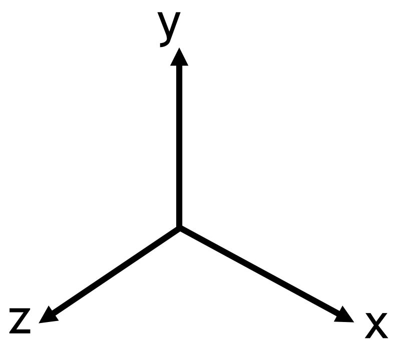
그래픽스 프로그래밍

07 Transformation & Representing Orientations

LHS Coordinate System (Unity)

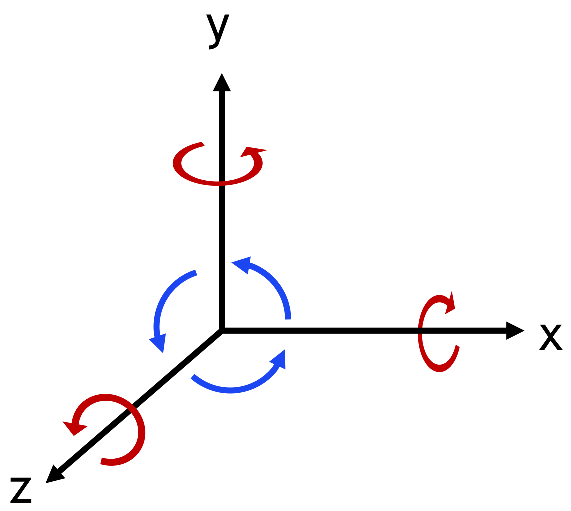
- Left Hand Coordinate System (LHS) - z+ forward
- Clockwise rotation
- If X-axis rotation
  - Y → Z rotation is positive
- If Y-axis rotation
  - Z → X rotation is positive
- If Z-axis rotation
  - X → Y rotation is positive





RHS Coordinate System

- 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



1

Transformation

Homogeneous Coordinates

- Why 3D computer graphics uses 4 x 4 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
  - $(x, y, z) \rightarrow (x, y, z, 1)$ $(2x, 2y, 2z, 2)$
  - $(x / \boxed{w}, y / \boxed{w}, z / \boxed{w}) \leftarrow (x, y, z, w)$

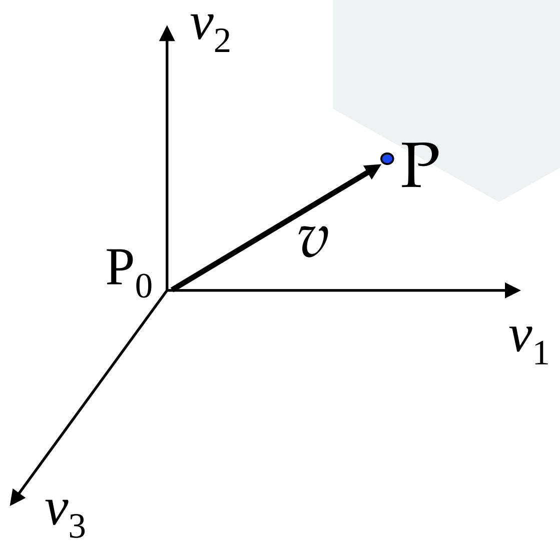
=1

Homogeneous Coordinates

$$\text{Vector } v = \Sigma \alpha_i v_i = \begin{pmatrix} \alpha_1 & \alpha_2 & \alpha_3 & 0 \end{pmatrix} \begin{pmatrix} v_1 \\ v_2 \\ v_3 \\ P_0 \end{pmatrix}$$

$$\text{Point } P = P_0 + \Sigma \alpha_i v_i = \begin{pmatrix} \alpha_1 & \alpha_2 & \alpha_3 & 1 \end{pmatrix} \begin{pmatrix} v_1 \\ v_2 \\ v_3 \\ P_0 \end{pmatrix}$$

$$P = \begin{pmatrix} \alpha_1 \\ \alpha_2 \\ \alpha_3 \\ 1 \end{pmatrix}, \quad v = \begin{pmatrix} \alpha_1 \\ \alpha_2 \\ \alpha_3 \\ 0 \end{pmatrix}$$



Homogeneous Coordinates

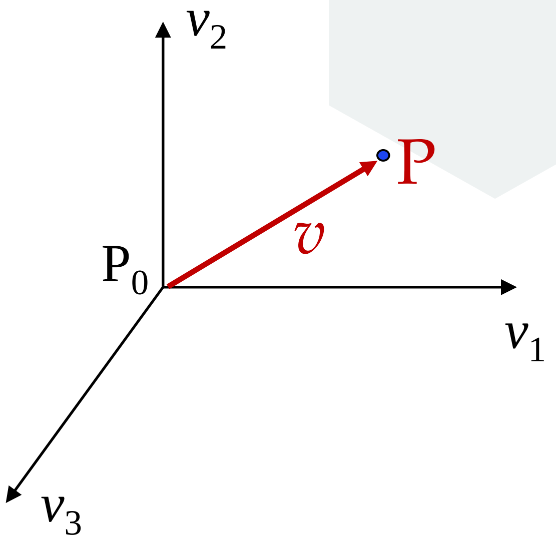
$$= \alpha_1 v_1 + \alpha_2 v_2 + \alpha_3 v_3$$

$$\text{Vector } v = \Sigma \alpha_i v_i = \begin{pmatrix} \alpha_1 & \alpha_2 & \alpha_3 & 0 \end{pmatrix} \begin{pmatrix} v_1 \\ v_2 \\ v_3 \\ P_0 \end{pmatrix}$$

$$= \alpha_1 v_1 + \alpha_2 v_2 + \alpha_3 v_3 + p_0$$

$$\text{Point } P = P_0 + \Sigma \alpha_i v_i = \begin{pmatrix} \alpha_1 & \alpha_2 & \alpha_3 & 1 \end{pmatrix} \begin{pmatrix} v_1 \\ v_2 \\ v_3 \\ P_0 \end{pmatrix}$$

$$P = \begin{pmatrix} \alpha_1 \\ \alpha_2 \\ \alpha_3 \\ 1 \end{pmatrix}, \quad v = \begin{pmatrix} \alpha_1 \\ \alpha_2 \\ \alpha_3 \\ 0 \end{pmatrix}$$



Affine Transformation

- The affine transformation maintains collinearity.
  - That is, every affine transformation preserves lines. All points on a line exist on the transformed line.
- Also, it maintains the ratio of distance.
  - That is, the midpoint of a line is located at the midpoint of the transformed line segment.
- $P' = f(P)$
- $P' = f(\alpha P_1 + \beta P_2) = \alpha f(P_1) + \beta f(P_2)$

Affine Transformation

- Most transformation in computer graphics are affine transformation. Affine transformation include translation, rotation, scaling, shearing.

- The transformed point P' (x', y', z') can be expressed as a linear combination of the original point P (x, y, z), i.e.

$$x' = \alpha_{11}x + a_{12}y + a_{13}z + \alpha_{14}$$

$$\begin{pmatrix} \boxed{x'} \\ y' \\ Z' \\ 1 \end{pmatrix} = \begin{pmatrix} \alpha_{11} & \alpha_{12} & \alpha_{13} & \alpha_{14} \\ \alpha_{21} & \alpha_{22} & \alpha_{23} & \alpha_{24} \\ \alpha_{31} & \alpha_{32} & \alpha_{33} & \alpha_{34} \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}$$

Affine Transformation

- The transformed point P' (x', y', z') can be expressed as a linear combination of the original point P (x, y, z), i.e.,

$$\begin{bmatrix} \boxed{x'} \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} \boxed{\alpha_{11}\, x + \alpha_{12}\, y + \alpha_{13}} \\ \alpha_{21}\, x + \alpha_{22}\, y + \alpha_{23} \\ 1 \end{bmatrix}$$

$$\Downarrow$$

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} \boxed{\alpha_{11} \quad \alpha_{12} \quad \alpha_{13}} \\ \alpha_{21} \quad \alpha_{22} \quad \alpha_{23} \\ 0 \quad\ \ 0 \quad\ \ 1 \end{bmatrix} * \begin{bmatrix} \boxed{\begin{matrix} x \\ y \\ 1 \end{matrix}} \end{bmatrix}$$

Affine Transformation

- The transformed point P' (x', y', z') can be expressed as a linear combination of the original point P (x, y, z), i.e.,

$$\begin{bmatrix} x' \\ \boxed{y'} \\ 1 \end{bmatrix} = \begin{bmatrix} \alpha_{11}\, x + \alpha_{12}\, y + \alpha_{13} \\ \boxed{\alpha_{21}\, x + \alpha_{22}\, y + \alpha_{23}} \\ 1 \end{bmatrix}$$

$$\Downarrow$$

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} \alpha_{11} & \alpha_{12} & \alpha_{13} \\ \alpha_{21} & \alpha_{22} & \alpha_{23} \\ 0 & 0 & 1 \end{bmatrix} * \begin{bmatrix} \boxed{\begin{matrix} x \\ y \\ 1 \end{matrix}} \end{bmatrix}$$

Affine Transformation

- The transformed point P' (x', y', z') can be expressed as a linear combination of the original point P (x, y, z), i.e.,

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} \alpha_{11} x + \alpha_{12} y + \alpha_{13} \\ \alpha_{21} x + \alpha_{22} y + \alpha_{23} \\ 1 \end{bmatrix}$$

$$\Downarrow$$

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} \alpha_{11} & \alpha_{12} & \alpha_{13} \\ \alpha_{21} & \alpha_{22} & \alpha_{23} \\ 0 & 0 & 1 \end{bmatrix} * \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

Unity Matrix Column - Major Order

- Matrices in Unity are column major.

```
// 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

- Matrices in Unity are column major.

```
// member variables       |     indices
// ---------------------  |----------------
// 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

- Matrices in Unity are column major.

```
// member variables      |     indices
// ---------------------  |----------------
// 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 4 x 4 matrix and 4 x 1 vector for transformation
  - v = (2, 6, -3, 1)
  - M = translate 10 units in x-axis
  - v' = M * v = (12, 6, -3, 1)

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

Unity Matrix Column - Major Order

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

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

Unity Matrix Column - Major Order

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

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

Geometric Transformation

- Geometric transformation refers to a function that transforms a group of points describing a geometric object to new points.
- At this time, the points are transformed to a new position while maintaining the relationship between the vertices of the objects.
- Basic transformation
  - Translation
  - Rotation
  - Scaling

Translation

- Translation

// create a translation matrix

```
Matrix4x4 m = Matrix4x4.Translate(new Vector3(dx, dy, dz));
```

$$P' = Tp \qquad T = \boxed{\begin{pmatrix} 1 & 0 & 0 & \underset{=1}{dx} \\ 0 & 1 & 0 & \underset{=2}{dy} \\ 0 & 0 & 1 & \underset{=3}{dz} \\ 0 & 0 & 0 & 1 \end{pmatrix}} \qquad T^{-1} = \begin{pmatrix} 1 & 0 & 0 & -dx \\ 0 & 1 & 0 & -dy \\ 0 & 0 & 1 & -dz \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Translation

- Translation

```
// create a translation matrix
Matrix4x4 m = Matrix4x4.Translate(new Vector3(dx, dy, dz));
```

$$P' = Tp \qquad T = \begin{pmatrix} 1 & 0 & 0 & dx \\ 0 & 1 & 0 & dy \\ 0 & 0 & 1 & dz \\ 0 & 0 & 0 & 1 \end{pmatrix} \qquad T^{-1} = \boxed{\begin{pmatrix} 1 & 0 & 0 & -dx \\ 0 & 1 & 0 & -dy \\ 0 & 0 & 1 & -dz \\ 0 & 0 & 0 & 1 \end{pmatrix}}$$

Translation

- Translation

```
// create a translation matrix
Matrix4x4 m = Matrix4x4.Translate(new Vector3(dx, dy, dz));
```

$$P' = Tp \qquad T = \begin{pmatrix} 1 & 0 & 0 & dx \\ 0 & 1 & 0 & dy \\ 0 & 0 & 1 & dz \\ 0 & 0 & 0 & 1 \end{pmatrix} \qquad T^{-1} = \begin{pmatrix} 1 & 0 & 0 & -dx \\ 0 & 1 & 0 & -dy \\ 0 & 0 & 1 & -dz \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Translation

- Translation
  - Matrices in Unity are column major; i.e. the position of a transformation matrix is in the last column, and the first three columns contain x, y, and z-axes.

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

$$T = \begin{pmatrix} 1 & 0 & 0 & dx \\ 0 & 1 & 0 & dy \\ 0 & 0 & 1 & dz \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Translation

- Translation
  - Matrices in Unity are column major; i.e. the position of a transformation matrix is in the last column, and the first three columns contain x, y, and z-axes.

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

$$
T = \begin{array}{cccc} x & y & z & \\ \hline 1 & 0 & 0 & dx \\ 0 & 1 & 0 & dy \\ 0 & 0 & 1 & dz \\ 0 & 0 & 0 & 1 \end{array}
$$

Translation

- Translation
  - Matrices in Unity are column major; i.e. the position of a transformation matrix is in the last column, and the first three columns contain x, y, and z-axes.

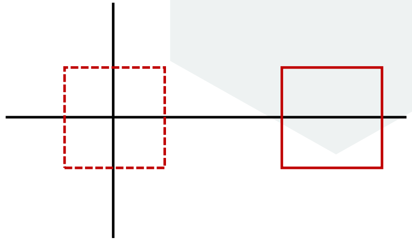
```
// get matrix from the Transform
var matrix = transform.localToWorldMatrix;
// get position from the last column
var position = new Vector3(matrix[0,3], matrix[1,3], matrix[2,3]);
// get position from the last column
var position = matrix.GetPosition();
```

$$T = \begin{pmatrix} 1 & 0 & 0 & dx \\ 0 & 1 & 0 & dy \\ 0 & 0 & 1 & dz \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Translation

- Translation
  - Matrices in Unity are column major; i.e. the position of a transformation matrix is in the last column, and the first three columns contain x, y, and z-axes.

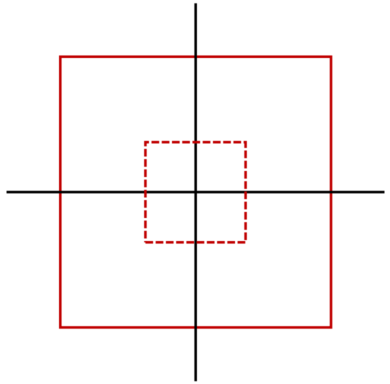
```
// get matrix from the Transform
var matrix = transform.localToWorldMatrix;
// get position from the last column
var position = new Vector3(matrix[0,3], matrix[1,3], matrix[2,3]);
// get position from the last column
var position = matrix.GetPosition();
```



Scale

- Scale

```
// create a scaling matrix
Matrix4x4 m = Matrix4x4.Scale(new Vector3(sx, sy, sz));
```



Scale

- Scale

  // create a scaling matrix

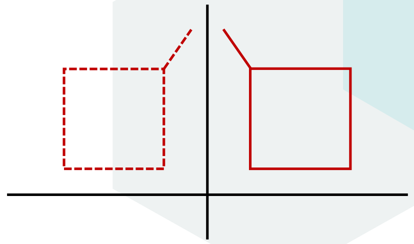
  Matrix4x4 m = Matrix4x4.Scale(new Vector3(sx, sy, sz));

$$P' = Sp \qquad S = \begin{pmatrix} sx & 0 & 0 & 0 \\ 0 & sy & 0 & 0 \\ 0 & 0 & sz & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \qquad S^{-1} = \begin{pmatrix} \frac{1}{sx} & 0 & 0 & 0 \\ 0 & \frac{1}{sx} & 0 & 0 \\ 0 & 0 & \frac{1}{sx} & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Reflection

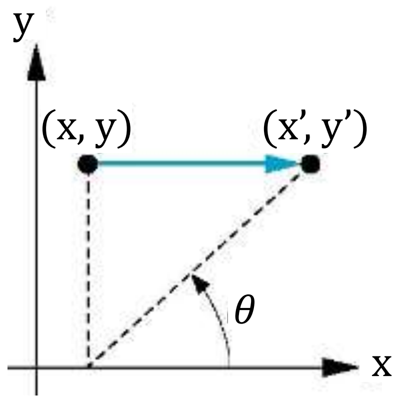
- Reflection

```
// 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));
```



Shear

- Shear



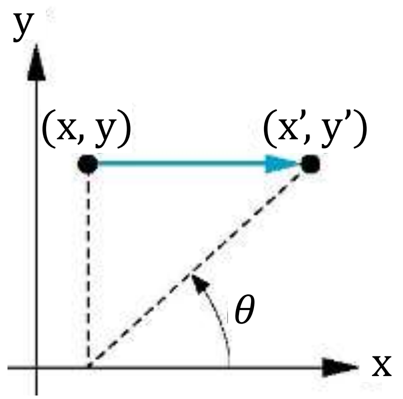
$$H_{xy}(\theta) = \begin{pmatrix} 1 & \cot\theta & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$$x' = x + y \cot\theta + 0$$
$$y' = y + 0 \cdot x + 0$$
$$z' = z + 0 \cdot x + 0 \cdot y$$

$$\tan\theta = \frac{y}{x' - x} \Rightarrow \cot\theta = \frac{x' - x}{y}$$

Shear

- Shear



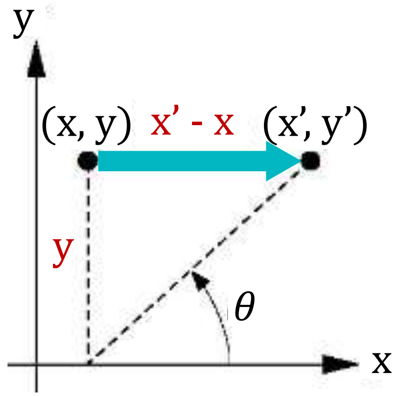
$$\mathrm{H}_{xy}(\theta) = \begin{pmatrix} \boxed{1 \quad \cot\theta \quad 0 \quad 0} \\ \boxed{0 \quad 1 \quad 0 \quad 0} \\ \boxed{0 \quad 0 \quad 1 \quad 0} \\ 0 \quad 0 \quad 0 \quad 1 \end{pmatrix}$$

$$x' = \boxed{x + y \cot \theta}$$
$$y' = \boxed{y}$$
$$z' = \boxed{z}$$

$$\tan\theta = \frac{y}{x'-x} \Rightarrow \cot\theta = \frac{x'-x}{y}$$

Shear

- Shear



x' ≠ x
y' = y

$$H_{xy}(\theta) = \begin{pmatrix} 1 & \cot\theta & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$$x' = x + y \cot \theta$$
$$y' = y$$
$$z' = z$$

$$\boxed{\tan\theta = \frac{y}{x' - x} \Rightarrow \cot\theta = \frac{x' - x}{y}}$$

$$\cot\theta \cdot y = x' - x,\ x' = x + \cot\theta \cdot y$$

Rotation

- Rotation
  - Unity uses Quaternion for rotation

```
// create a matrix that can be used to rotate a set of vertices
    around the x / y / z-axis (Euler angle in degree)
// a rotation 30 degrees around the y-axis
Quaternion rotation = Quaternion.Euler(0, 30, 0);
Matrix4x4 m = Matrix4x4.Rotate(rotation);
```

Rotation

- Rotation in Z-axis (Unity LHS CW)
  - $x' = x\,\boxed{\cos\theta}\,\boxed{-}\,y\,\boxed{\sin\theta\ □}$
  - $y' = x\,\boxed{\sin\theta}\,+\,y\,\boxed{\cos\theta\ □}$
  - $z' = z$

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

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

$$\begin{pmatrix} x' \\ y' \\ z' \\ 1 \end{pmatrix} = \begin{pmatrix} \boxed{\cos\theta} & \boxed{-\sin\theta} & 0 & 0 \\ \boxed{\sin\theta} & \boxed{\cos\theta} & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}$$

$$P' = R_z(\theta) \cdot P$$

Rotation

- Rotation in X-axis (Unity LHS CW)
  - $y' = \boxed{y \cos\theta - z \sin\theta}$
  - $z' = \boxed{y \sin\theta + z \cos\theta}$
  - $x' = \boxed{x}$

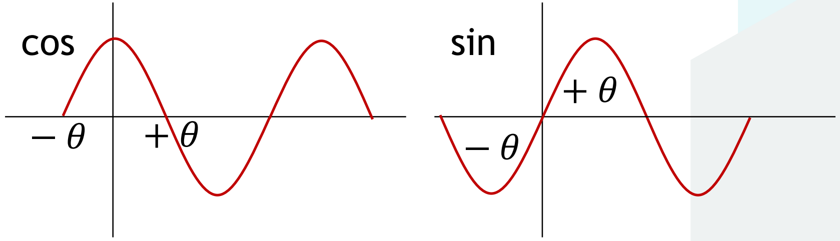
```
// create a Rx matrix
Matrix4x4 rx = Matrix4x4.Rotate(Quaternion.Euler(30, 0, 0));
```

$$\begin{pmatrix} x' \\ y' \\ z' \\ 1 \end{pmatrix} = \begin{pmatrix} \boxed{1} & 0 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta & 0 \\ 0 & \sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}$$

$$P' = R_x(\theta) \cdot P$$

Rotation

- Rotation in Y-axis (Unity LHS CW)
  - x’ = $\boxed{x \cos\theta + z \sin\theta}$
  - z’ = $\boxed{-x \sin\theta + z \cos\theta}$
  - y’ = $\boxed{y}$



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

$$\begin{pmatrix} x' \\ y' \\ z' \\ 1 \end{pmatrix} = \begin{pmatrix} \boxed{\cos\theta} & 0 & \boxed{\sin\theta} & 0 \\ 0 & \boxed{1} & 0 & 0 \\ \boxed{-\sin\theta} & 0 & \boxed{\cos\theta} & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}$$

$$P' = R_y(\theta) \cdot P$$

Rotation

- Rotation in arbitrary axis (Unity LHS CW)

```
// create a Ra matrix
Matrix4x4 ra = Matrix4x4.Rotate(Quaternion.Euler(30, 60, 45));
// create a Rb matrix
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
// ra == rb matrix
// ra == rb
```

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.

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

(pre-multiply)

Composing Transformation

- Matrix multiplication is associative.

$$M3 \cdot M2 \cdot M1 = (M3 \cdot M2) \cdot M1 = M3 \cdot (M2 \cdot M1)$$

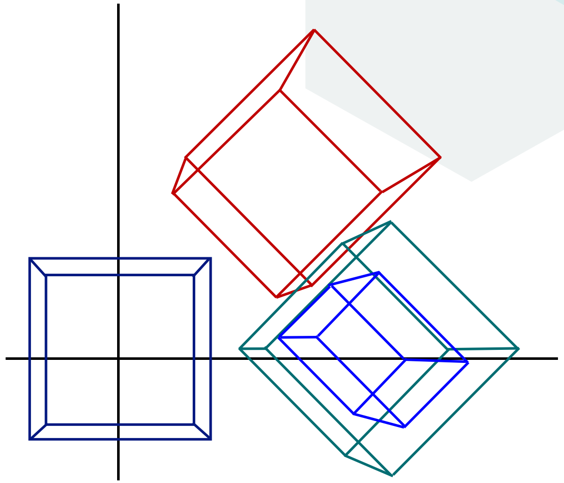
- Matrix multiplication is not commutative.

$$A \cdot B != B \cdot A$$

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.

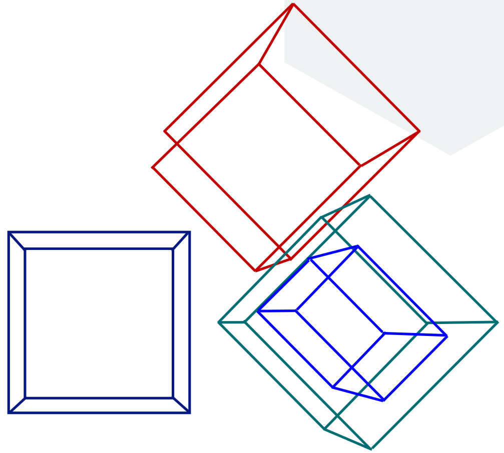
```
// original cube at the origin(0, 0, 0)
RT = Rz(45) * T(1.5, 0, 0) // T first, then Rz
TR = T(1.5, 0, 0) * Rz(45) // Rz first, then T
TRS = T(1,2,-3) * Rz(45) * S(1/5, 1/5, 1/5) // S → R → T
SRT = S(1/5, 1/5, 1/5) * Rz(45) * T(1,2,-3) // T→ R→S
TRS! = SRT // Transformation Matrix Order Matter!
```



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(1/5, 1/5, 1/5) // S → R → T
SRT = S(1/5, 1/5, 1/5) * Rz(45) * T(1,2,-3) // T→ R→S
TRS! = SRT // Transformation Matrix Order Matter!
```



Composing Transformation

- TRS matrix
  - For example, transforms a position p=(5, 0, 0)
  - scale 1/5
  - rotate π/4 in y-axis
  - translate (1, 2, -3)
  - Then, M = T(1, 2, -3) * Ry(45) * S(1/5, 1/5, 1/5)
- p’ = Mp = (1.707, 2, -3.707)

Composing Transformation

- Vector3 scale = new Vector3 (0.2, 0.2, 0.2);
- Quaternion rotation = Quaternion.Euler (0, 45, 0);
- Vector3 translation = new Vector3 (1, 2, -3);

```
// create a composing transformation Scale → Rotate → Translate
Matrix4x4 m = Matrix4x4.TRS(translation, rotation, scale);

// create a composing transformation Scale → Rotate → Translate
Matrix4x4 m2 = Matrix4x4.Translate(translation) *
Matrix4x4.Rotate(rotation) * Matrix4x4.Scale(scale);
//m == m2

// get a new position by a composing transformation matrix, m
Vector3 pos = new Vector3(5, 0, 0);
Vector3 newpos = m.MultiplyPoint(pos);
```

MultiplyPoint

Matrix4x4.MultiplyPoint	Transforms a position by this matrix (generic)
Matrix4x4.MultiplyPoint3x4	Transforms a position by this matrix (fast)
Matrix4x4.MultiplyVector	Transforms a vector by this matrix

MultiplyPoint

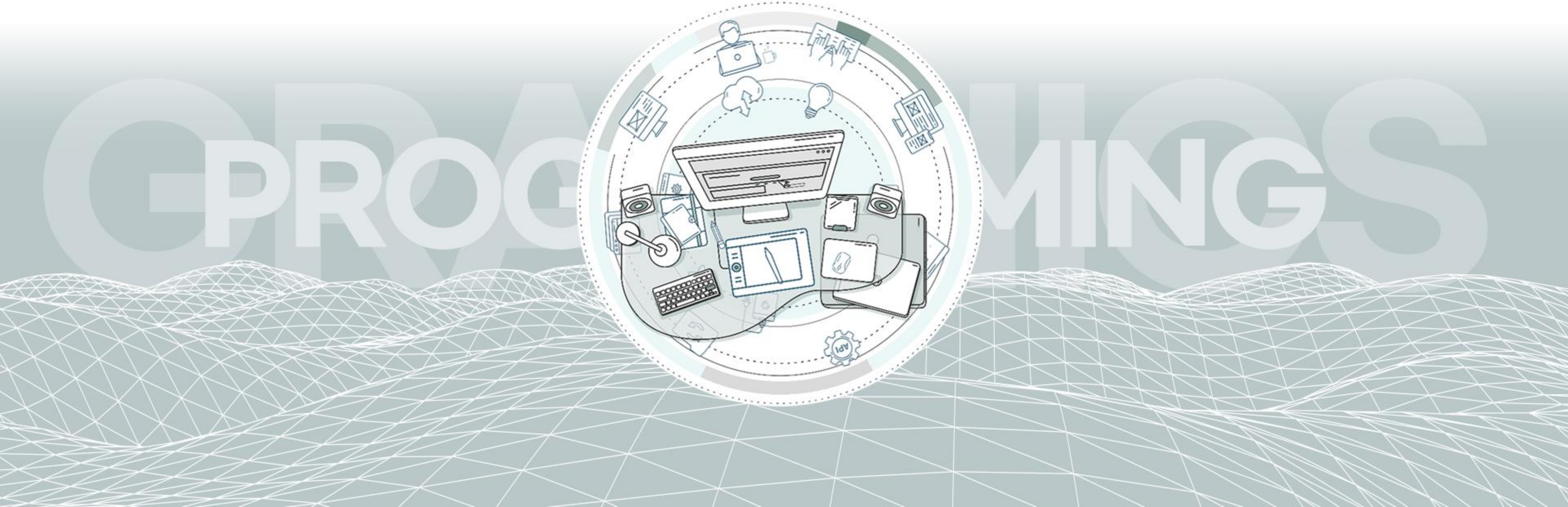
Matrix4x4.MultiplyPoint	Transforms a position by this matrix (generic)
Matrix4x4.MultiplyPoint3x4	Transforms a position by this matrix (fast)
Matrix4x4.MultiplyVector	Transforms a vector by this matrix

MultiplyPoint

Matrix4x4.MultiplyPoint	Transforms a position by this matrix (generic)
Matrix4x4.MultiplyPoint3x4	Transforms a position by this matrix (fast)
Matrix4x4.MultiplyVector	Transforms a vector by this matrix

2

Representing Orientation



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

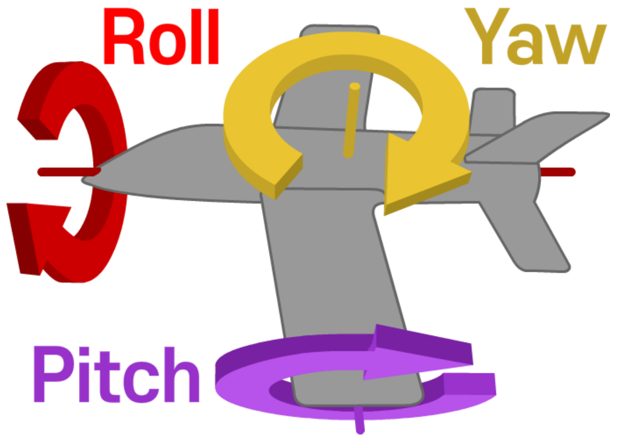
Euler Angles

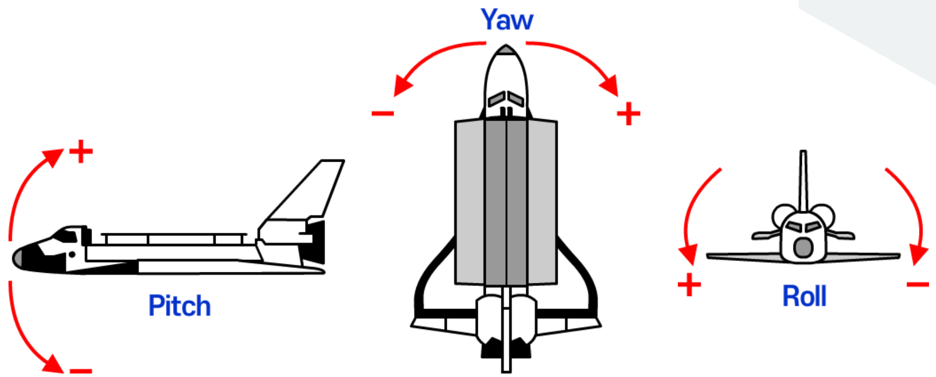
- Axis order
  - Euler angles represent three composed rotations that move a reference frame to a given referred frame.
  - Euler angles are used in a lot of applications, but they tend to require some rather arbitrary decisions.
  - (y, x, z), (x, y, z), (z, x, y), … can be used

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/DirectX/Unity





이미지 출처 :https://en.wikiversity.org/wiki/Flight_dynamics
https://www.faa.gov/

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.

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

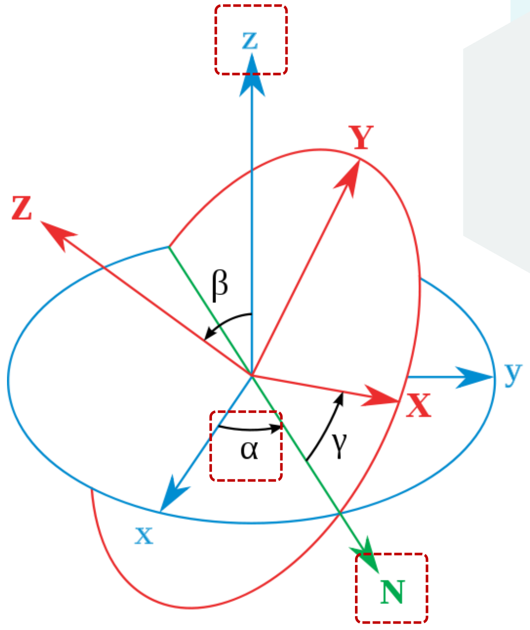
$$= \begin{bmatrix} c_y c_z & c_y s_z & -s_y \\ s_x s_y c_z - c_x s_z & s_x s_y s_z + c_x c_z & s_x c_y \\ c_x s_y c_z + s_x s_z & c_x s_y s_z - s_x c_z & c_x c_y \end{bmatrix}$$

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

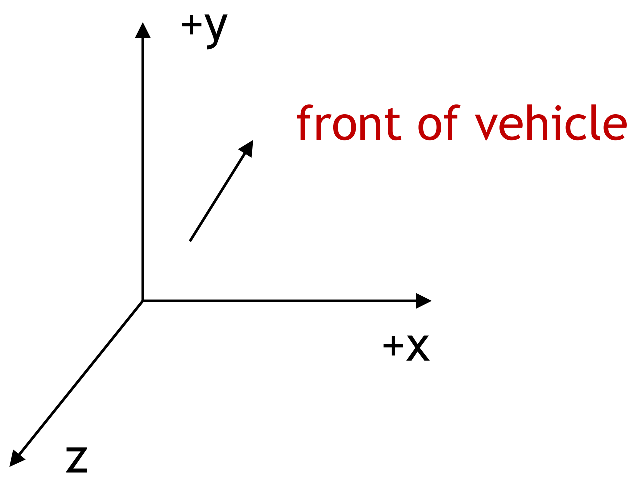
- ZXZ convention
  - XYZ (fixed) system is shown in blue.
  - XYZ (rotated) system is shown in red.
  - The line of nodes, N, is shown in green.
  - (Z-rotation) Rotate about the Z-axis by α.
    - The X-axis now lies on the line of nodes, N
  - (X-rotation) Rotate again about the rotated X-axis (i.e., 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.



이미지 출처 : https://en.wikipedia.org/wiki/Euler_angles

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.

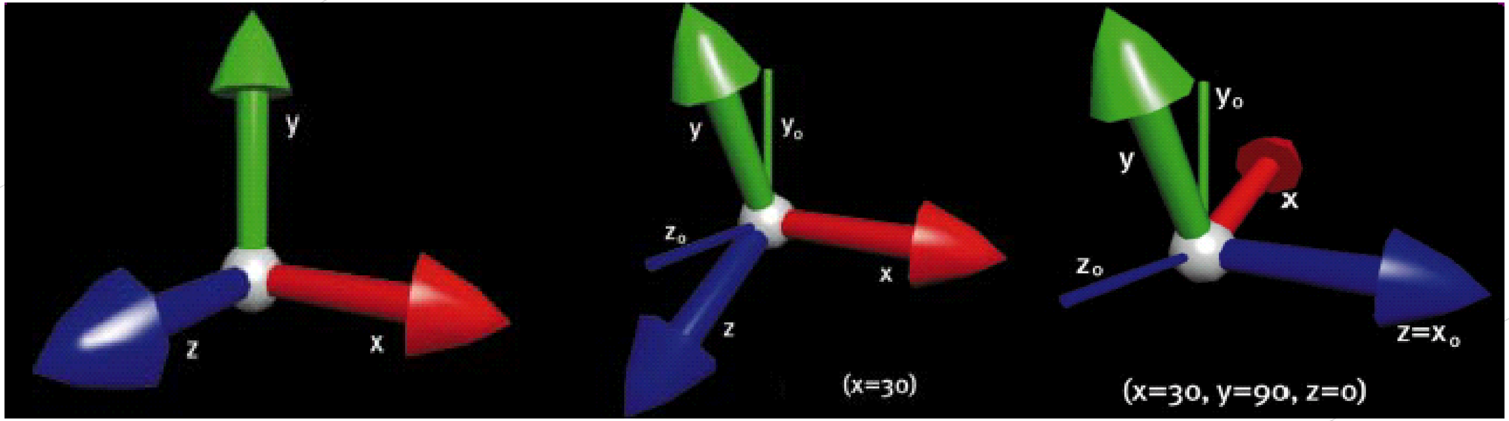


Rotations not uniquely defined with Euler Angles

- 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)

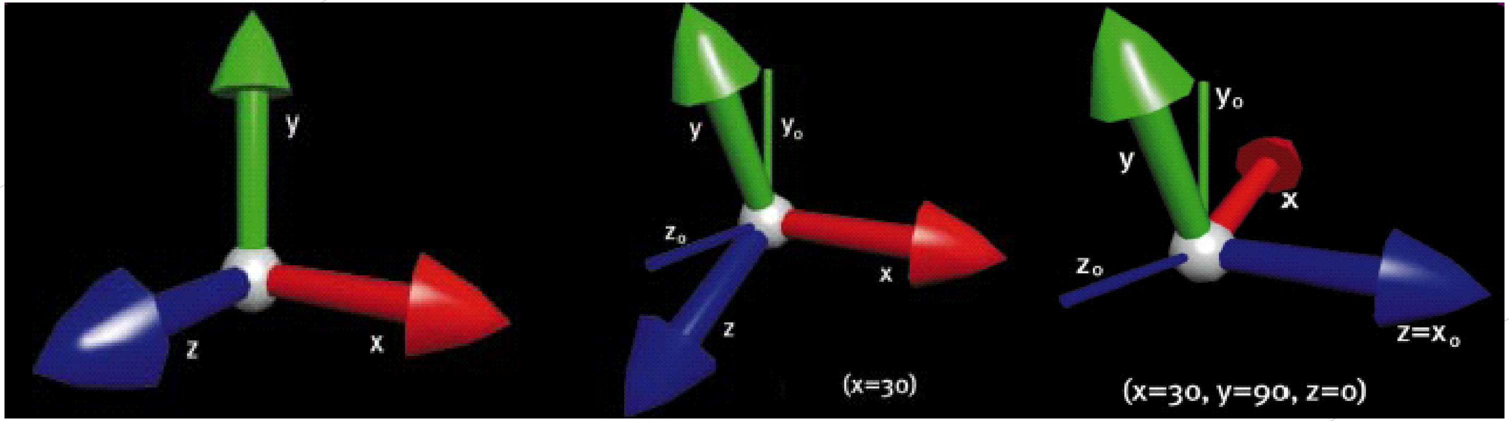
Gimbal Lock

- 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.



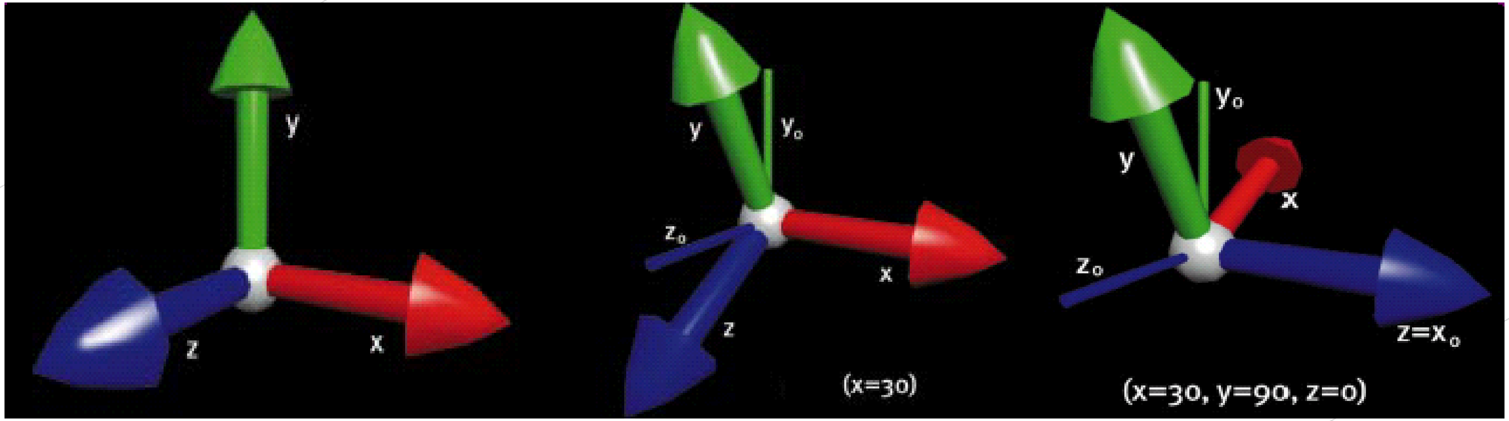
Gimbal Lock

- 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.
  - ➢ Rotate 30 about X, then rotate 90 about Y. The current Z-axis is in line with X_0-axis. This is what we call 'gimbal lock' situation.



Gimbal Lock

- One potential problem is '<u>gimbal lock</u>'.
- '<u>Gimbal Lock</u>' 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.
  - Any further rotation about the Z-axis affects the same degree of freedom as rotating about the X-axis - losing the third DOF.



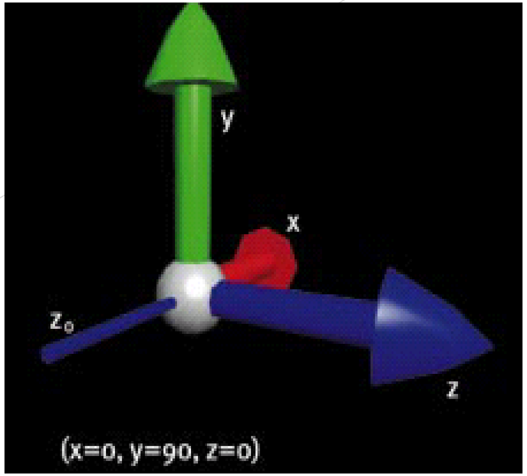
Gimbal Lock

- Euler (gimbal lock) Explained

영상 출처 : 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)

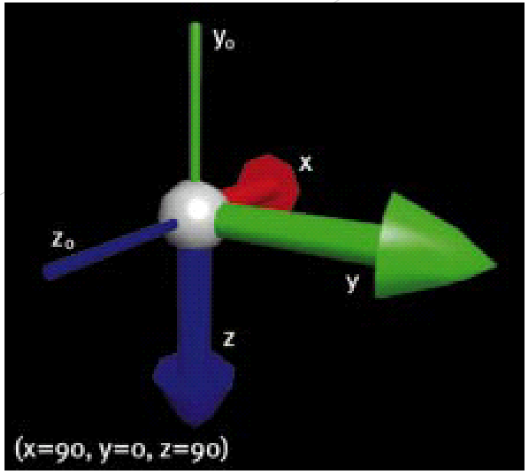


Halfway between (0,0,0) and (0,180,0)

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)
  - 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 (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).

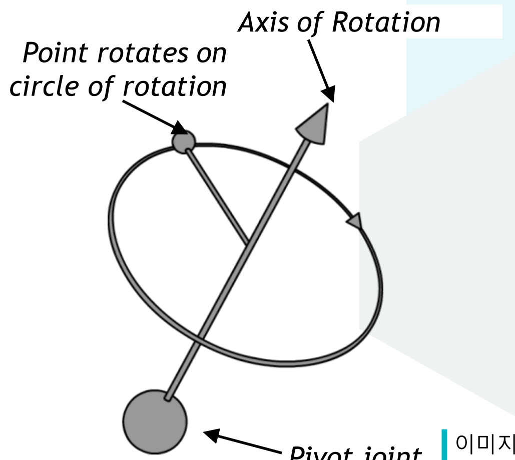
Rotation Vectors and Axis/Angle

- 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).

Rotation Vectors and Axis/Angle

- To generate a matrix as a rotation q around an arbitrary unit axis **a** :

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



이미지 출처 :---

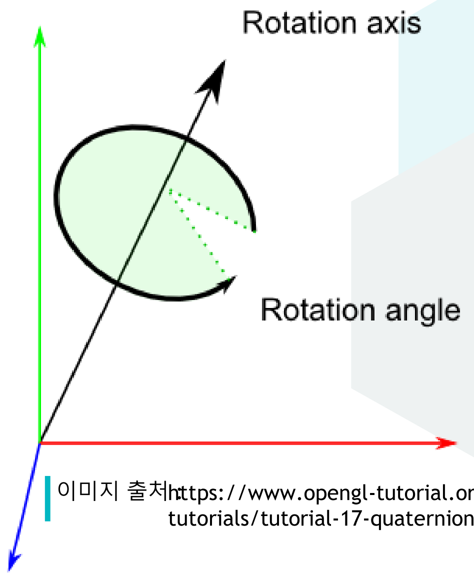
to. 교
출처
부탁드

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

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

Rotation Vectors and Axis/Angle

- To generate a matrix as a rotation q around an arbitrary unit axis **a** :



이미지 출처 https://www.opengl-tutorial.org/intermediate-tutorials/tutorial-17-quaternions/

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

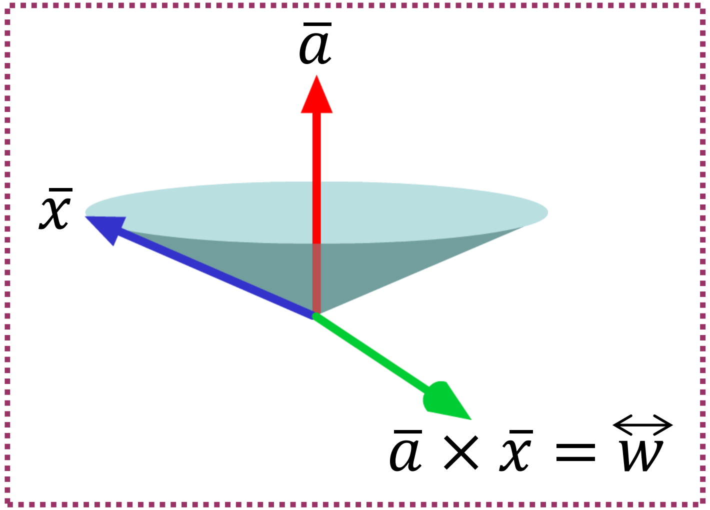
$$= \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}\cos\theta + \begin{bmatrix} a_x^2 & a_xa_y & a_xa_z \\ a_xa_y & a_y^2 & a_ya_z \\ a_xa_z & a_ya_z & a_z^2 \end{bmatrix}(1-\cos\theta) + \begin{bmatrix} 0 & -a_z & a_y \\ a_z & 0 & -a_x \\ -a_y & a_x & 0 \end{bmatrix}\sin\theta$$

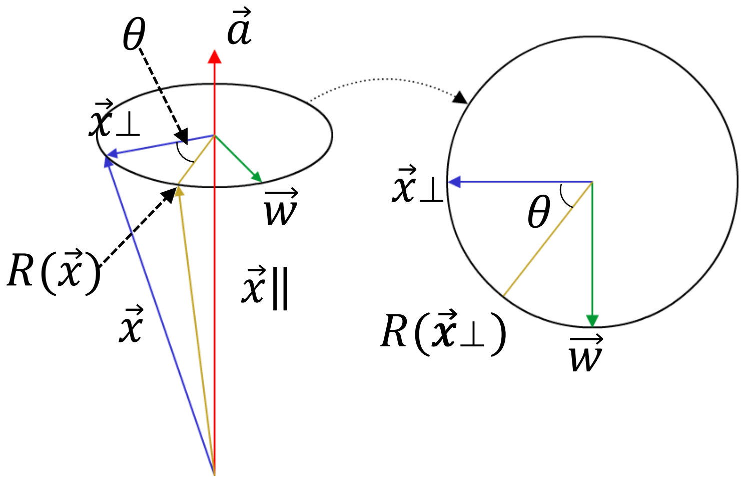
$$= \begin{bmatrix} a_x^2 + \cos\theta\,(1-a_x^2) & a_xa_y(1-\cos\theta) - a_z\sin\theta & a_xa_z(1-\cos\theta) - a_y\sin\theta \\ a_xa_y(1-\cos\theta) + a_z\sin\theta & a_y^2 + \cos\theta\,(1-a_y^2) & a_ya_z(1-\cos\theta) - a_x\sin\theta \\ a_xa_z(1-\cos\theta) - a_y\sin\theta & a_ya_z(1-\cos\theta)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

3D Rotation as Vector Components

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

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

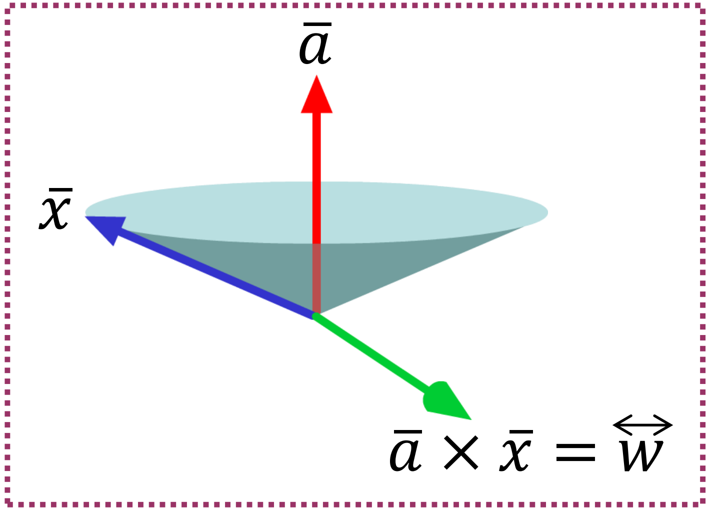


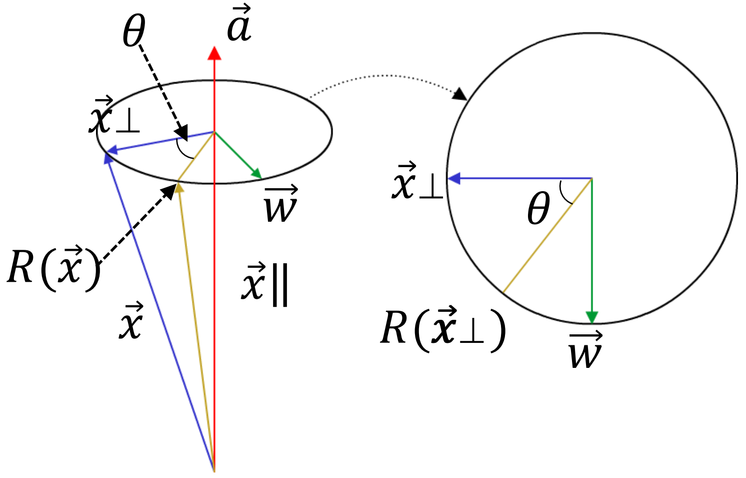


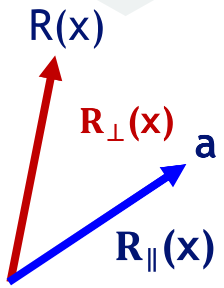
3D Rotation as Vector Components

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

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

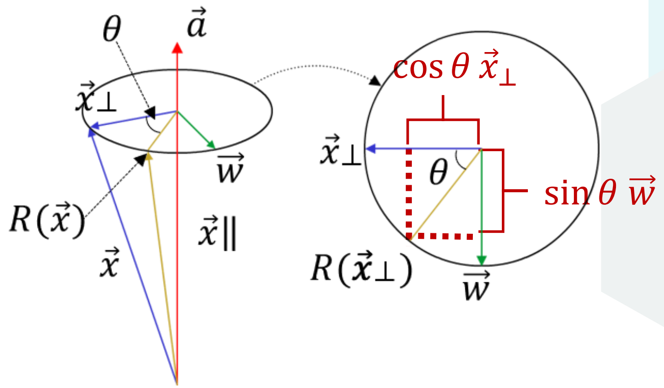






3D Rotation as Vector Components

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



$$\vec{x}_\parallel = (\vec{a}\cdot\vec{x})\vec{a}$$

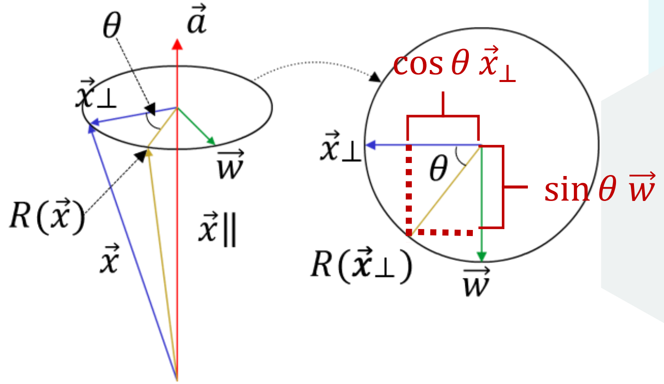
$$\vec{x}_\perp = \vec{x} - \vec{x}_\parallel = \vec{x} - (\vec{a}\cdot\vec{x})\vec{a}$$

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

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

3D Rotation as Vector Components

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

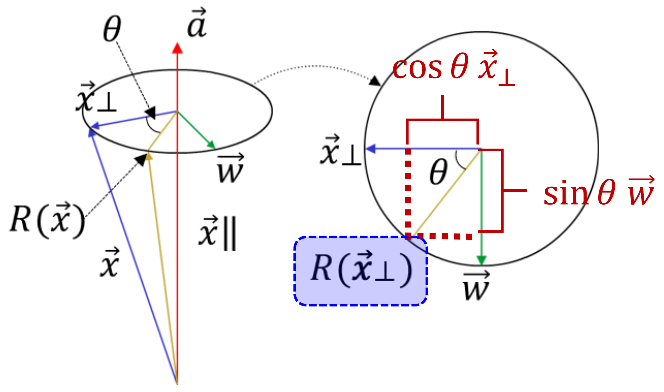


$$\mathrm{R}(\vec{x}_\perp) = \cos\theta\,\vec{x}_\perp + \sin\theta\,\vec{w}$$

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

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

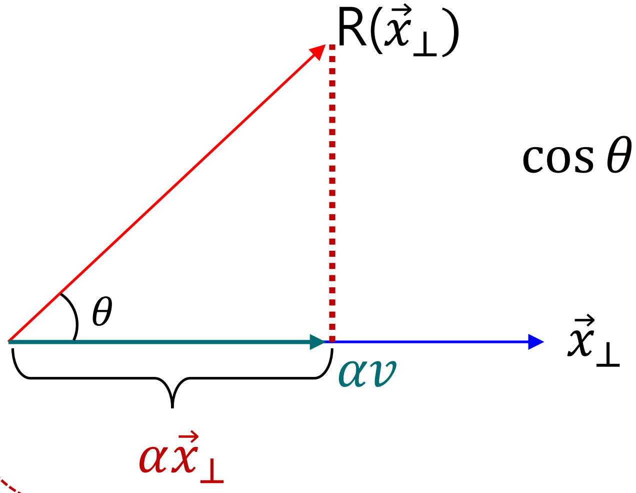
3D Rotation as Vector Components



$$\cos\theta\,\|R(x_\perp)\|\hat{x}_\perp$$

$$= \cos\theta\,\|\cancel{x_\perp}\|\,\frac{x_\perp}{\|\cancel{x_\perp}\|}$$

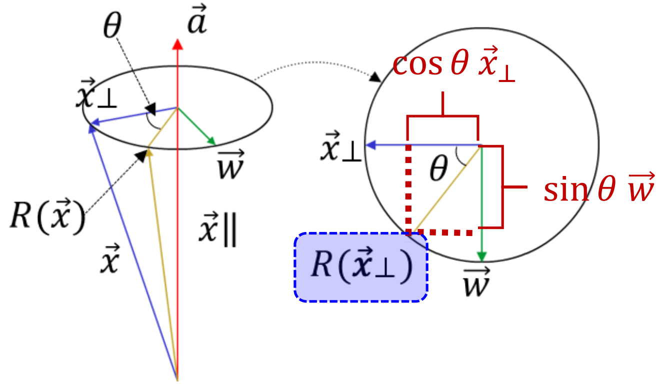
$$= \cos\theta\,x_\perp$$



$$\cos\theta = \frac{\|\alpha x_\perp\|}{\|R(x_\perp)\|}$$

$$= \|\alpha x_\perp\| = \cos\theta\,\|R(x_\perp)\|$$

3D Rotation as Vector Components



$$\cos\theta\,\|R(x_\perp)\|\hat{x}_\perp$$

$$= \cos\theta\,\|\cancel{x_\perp}\|\,\frac{x_\perp}{\|\cancel{x_\perp}\|}$$

$$= \cos\theta\,x_\perp$$

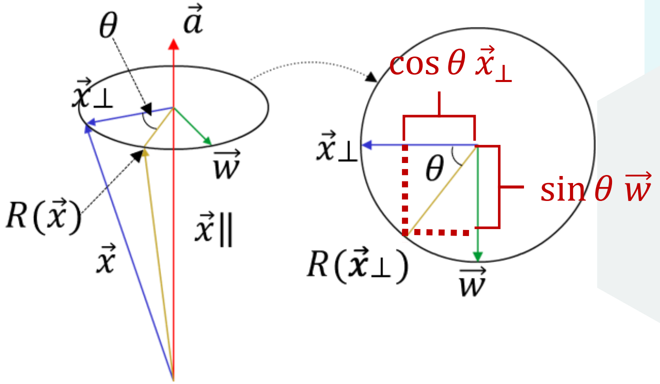
$$\text{Sin}\,\theta\,\|R(x_\perp)\|\hat{w} \qquad \|R(x_\perp)\| = \|x_\perp\| = \|w\|$$

$$= \text{Sin}\,\theta\,\|w\|$$

$$= \text{Sin}\,\theta\,w$$

3D Rotation as Vector Components

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



$$\vec{x}_\parallel = (\vec{a} \cdot \vec{x})\vec{a}$$

$$\vec{x}_\perp = \vec{x} - \vec{x}_\parallel = \vec{x} - (\vec{a} \cdot \vec{x})\vec{a}$$

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

$$\begin{aligned}\mathrm{R}(\vec{x}) &= \mathrm{R}(\vec{x}_\parallel) + \mathrm{R}(\vec{x}_\perp) \\ &= \mathrm{R}(\vec{x}_\parallel) + \cos\theta\, \vec{x}_\perp + \sin\theta\, \vec{w} \\ &= (\vec{a} \cdot \vec{x})\vec{a} + \cos\theta\, (\vec{x} - (\vec{a} \cdot \vec{x})\vec{a}) + \sin\theta\, \vec{w} \\ &= \cos\theta\, \vec{x} + (1 - \cos\theta)\underbrace{(\vec{a} \cdot \vec{x})\vec{a}}_{\text{Symmetric}} + \sin\theta\, \underbrace{(\vec{a} \times \vec{x})}_{\text{Skew}}\end{aligned}$$

3D Rotation as Vector Components

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

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

3D Rotation as Vector Components

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

$$\textbf{Symmetric}\left(\begin{bmatrix} a_x \\ a_y \\ a_z \end{bmatrix}\right) = \underbrace{\begin{bmatrix} a_x \\ a_y \\ a_z \end{bmatrix}}_{3\times 1} \underbrace{[a_x\ a_y\ a_z]}_{1\times 3} = \underbrace{\begin{bmatrix} a_x^2 & a_x a_y & a_x a_z \\ a_x a_y & a_y^2 & a_y a_z \\ a_x a_z & a_y a_z & a_z^2 \end{bmatrix}}_{3\times 3} \begin{bmatrix} x \\ y \\ z \end{bmatrix}$$

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

$$x' = a_x^2 x + a_x a_y y + a_x a_z z$$
$$y' = a_x a_y x + a_y^2 y + a_y a_z z$$
$$z' = a_x a_z x + a_y a_z y + a_z^2 z$$

3D Rotation as Vector Components

Symmetric $= (a \cdot x)a$

$$\Big\downarrow$$

$$(a_x^x + a_y^y + a_z^z)\begin{pmatrix} a_x \\ a_y \\ a_z \end{pmatrix}$$

$$x' = a_x^2 x + a_x a_y y + a_x a_z z$$

$$y' = a_x a_y x + a_y^2 y + a_y a_z z$$

$$z' = a_x a_z x + a_y a_z y + a_z^2 z$$

3D Rotation as Vector Components

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

$$\mathbf{Symmetric}\left(\begin{bmatrix} a_x \\ a_y \\ a_z \end{bmatrix}\right) = \begin{bmatrix} a_x \\ a_y \\ a_z \end{bmatrix} [a_x \ a_y \ a_z] = \boxed{\begin{bmatrix} a_x^2 & a_x a_y & a_x a_z \\ a_x a_y & a_y^2 & a_y a_z \\ a_x a_z & a_y a_z & a_z^2 \end{bmatrix}} \begin{bmatrix} x \\ y \\ z \end{bmatrix}$$

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

$$\begin{aligned} x' &= a_x^2 x + a_x a_y y + a_x a_z z \\ y' &= a_x a_y x + a_y^2 y + a_y a_z z \\ z' &= a_x a_z x + a_y a_z y + a_z^2 z \end{aligned}$$

3D Rotation as Vector Components

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

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

$$\mathbf{Skew}(\bar{a}) = \bar{a} \times \bar{x} \qquad \begin{pmatrix} a_x \\ a_y \\ a_z \end{pmatrix} \times \begin{pmatrix} x \\ y \\ z \end{pmatrix}$$

$$= \begin{pmatrix} a_y z & - a_z y \\ -a_x z + a_z x \\ a_x y & - a_y x \end{pmatrix}$$

3D Rotation as Vector Components

- First, consider a rotation by 0 :

$$\mathbf{Rotate}\left(\begin{bmatrix} a_x \\ a_y \\ a_z \end{bmatrix}, 0\right) = \cancel{\begin{bmatrix} a_x^2 & a_x a_y & a_x a_z \\ a_x a_y & a_y^2 & a_y a_z \\ a_x a_z & a_y a_z & a_z^2 \end{bmatrix}}(1-1) + \cancel{\begin{bmatrix} 0 & -a_z & a_y \\ a_z & 0 & -a_x \\ -a_y & a_x & 0 \end{bmatrix}}0 + \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}1 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

- For instance, a rotation about the x-axis:

$$\mathbf{Rotate}\left(\begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}, \theta\right) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}(1-\cos\theta) + \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & -1 \\ 0 & 1 & 0 \end{bmatrix} \operatorname{Sin}\theta + \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \cos\theta$$

$$\mathbf{Rotate}\left(\begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}, \theta\right) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\theta & -\operatorname{Sin}\theta \\ 0 & \operatorname{Sin}\theta & \cos\theta \end{bmatrix}$$

3D Rotation as Vector Components

- For instance, a rotation about the y-axis

$$\textbf{Rotate}\left(\begin{bmatrix}0\\1\\0\end{bmatrix}, \theta\right) = \begin{bmatrix}0 & 0 & 0\\0 & 1 & 0\\0 & 0 & 0\end{bmatrix}(1 - \cos\theta) + \begin{bmatrix}0 & 0 & 1\\0 & 0 & 0\\-1 & 0 & 0\end{bmatrix}\text{Sin}\,\theta + \begin{bmatrix}1 & 0 & 0\\0 & 1 & 0\\0 & 0 & 1\end{bmatrix}\cos\theta$$

$$\textbf{Rotate}\left(\begin{bmatrix}0\\1\\0\end{bmatrix}, \theta\right) = \begin{bmatrix}\cos\theta & 0 & \text{Sin}\,\theta\\0 & 1 & 0\\-\text{Sin}\,\theta & 0 & \cos\theta\end{bmatrix}$$

- For instance, a rotation about the z-axis

$$\textbf{Rotate}\left(\begin{bmatrix}0\\0\\1\end{bmatrix}, \theta\right) = \begin{bmatrix}0 & 0 & 0\\0 & 0 & 0\\0 & 0 & 1\end{bmatrix}(1 - \cos\theta) + \begin{bmatrix}0 & -1 & 0\\1 & 0 & 0\\0 & 0 & 0\end{bmatrix}\text{Sin}\,\theta + \begin{bmatrix}1 & 0 & 0\\0 & 1 & 0\\0 & 0 & 1\end{bmatrix}\cos\theta$$

$$\textbf{Rotate}\left(\begin{bmatrix}0\\0\\1\end{bmatrix}, \theta\right) = \begin{bmatrix}\cos\theta & -\text{Sin}\,\theta & 0\\\text{Sin}\,\theta & \cos\theta & 0\\0 & 0 & 1\end{bmatrix}$$

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!

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

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

$$i = jk = -kj$$

$$j = ki = -ik$$

$$k = ij = -ji$$

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 (Imaginary Space)

- 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

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

- 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

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

Quaternions as Rotations

- A quaternion can represent a rotation by an angle q around a unit axis a (ax, ay, az)

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

$$or$$

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

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

Quaternions as Rotations

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

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

$$= \sqrt{\sin^2\frac{\theta}{2}\underbrace{\boxed{\left(a_x^2 + a_y^2 + a_z^2\right)}}_{=1} + \cos^2\frac{\theta}{2}}$$

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

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

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

$$Scale = \sqrt{x^2 + y^2 + z^2} \text{ or } Sin(\text{acos}(w))$$

$$ax = {}^{x}/_{Scale}$$

$$ay = {}^{y}/_{Scale}$$

$$az = {}^{z}/_{Scale}$$

$$\theta = 2acos(w)$$

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 q_0, q_1,q_2, q_3 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

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

- 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

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

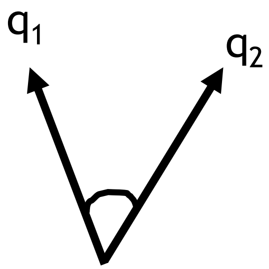
Basic Quaternion Mathematics

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

    where q = [v sin q, cos q]

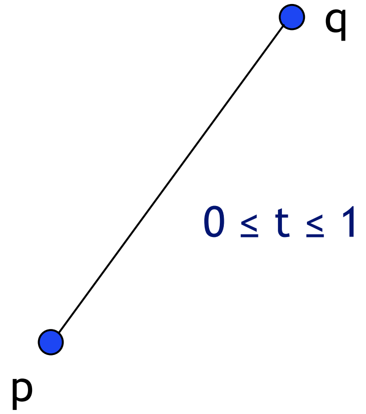
Quaternion Interpolation

- 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
  - q_1 = Euler2Quaternion(frame0)
  - q_2 = Euler2Quaternion(frame1)
  - q_r = QuaternionInterpolation(q_1, q_2, alpha)
  - q_r.Quaternion2Euler()
- Quaternion Interpolation
  - Linear Interpolation (LERP)
  - Spherical Linear Interpolation (SLERP)
  - Spherical Cubic Interpolation (SQUAD)



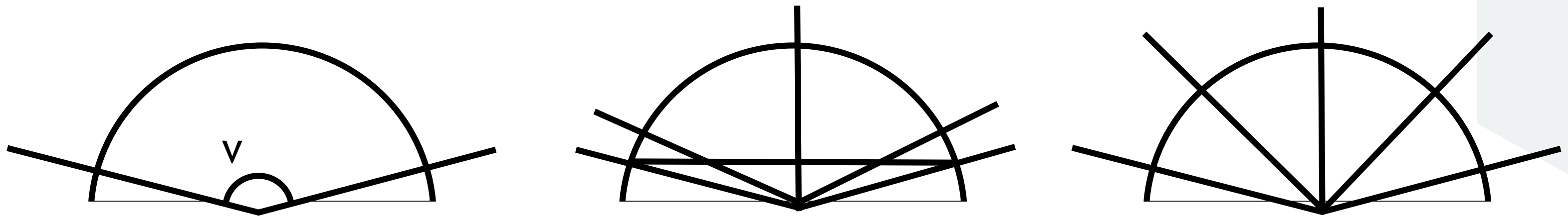
Linear Interpolation (LERP)

- If we want to do a direct interpolation between two quaternions p and q by alpha
  - Lerp(p, q, t) = (1-t)p + (t)q
  - where $0 \leq t \leq 1$
- 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
  - Lerp(p, q, t) = p + t(q - p)



Why SLERP?

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



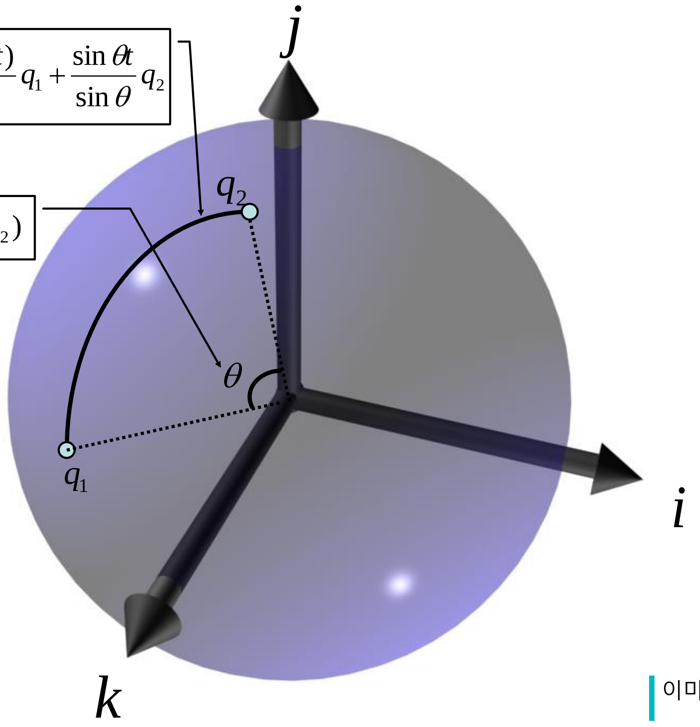
- An illustration in the plane of the difference between Lerp and Slerp
  - The interpolation covers the angle v in three steps
  - [Lerp] The secant across is split in four equal pieces The corresponding angles are shown
  - [Slerp] The angle is split in four equal angles

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.'

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

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



to. 교수님
출처 확인
부탁드립니다.
혹시 고화질
이미지가 있다면
함께 전달
부탁드립니다.

이미지 출처 :---

Spherical Linear Interpolation

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

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

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

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

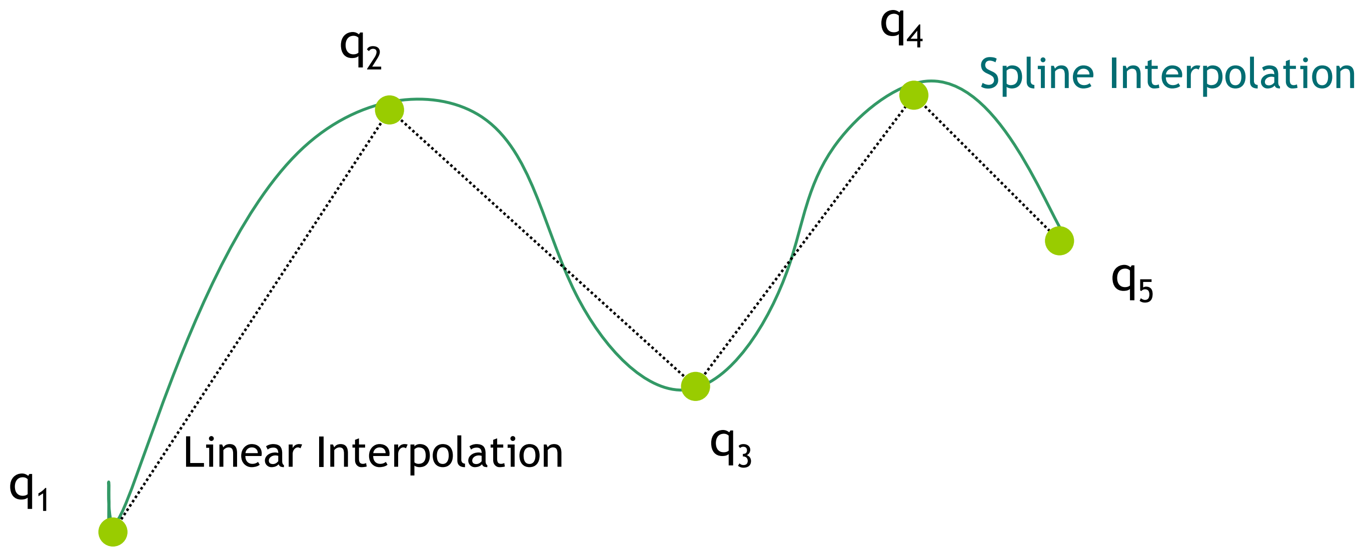
Spherical Linear Interpolation

- 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

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

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

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

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);

Unity Quaternion

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

Catmull-Rom Spline Interpolation

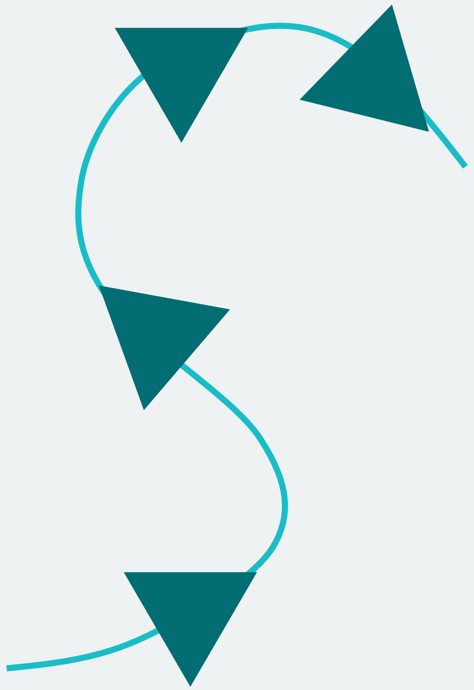
- Given n+1 control points $\{P_0, P_1, .. P_n\}$, you wish to find a curve that interpolates these control points (and passes through them all), and is local in nature (i.e. if one of the control points is moved, it only affects the curve locally) - Catmull-Rom Spline.

- 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