





**CS482 Lab Session**  
HelloOpenGL3D  
2016. 9. 28



**Rotation**

2



# Rotation

KAIST

- Last week, we have implemented object rotation in OpenGL ES with Android

```

59     if (e.getAction() == MotionEvent.ACTION_MOVE) {
60         switch (mode) {
61             case 0:
62                 if (count == 1) {
63                     // Rotate world
64                     float[] rot = temp1;
65
66                     Matrix.setIdentityM(rot, 0);
67                     Matrix.rotateM(rot, 0, dx, 0, 1, 0);
68                     Matrix.rotateM(rot, 0, 0, dy, 1, 0);
69                     Matrix.multiplyMM(temp2, 0, rot, 0, mRenderer.mViewRotationMatrix, 0);
70                     System.arraycopy(temp2, 0, mRenderer.mViewRotationMatrix, 0, 16);
71                 } else if (count == 2) {
72                     // Translate world
73                     Matrix.translateM(mRenderer.mViewTranslationMatrix, 0, dx/100, -dy/100, 0);
74                 }
75                 break;

```

Rotate 'dx' degree w.r.t. Y-axis  
Rotate 'dy' degree w.r.t. X-axis

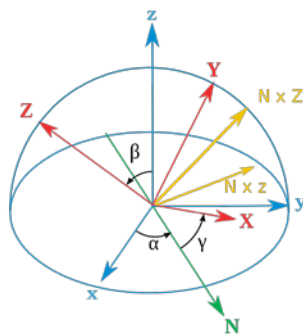
3

VISUAL  
COMPUTING Lab

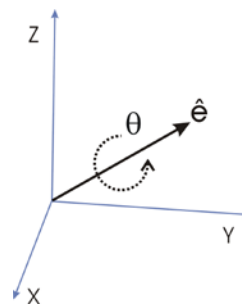
# Rotation

KAIST

- Euler Angles vs. Quaternions



Rotation using Euler Angles



Rotation using Quaternions

Reference: <http://www.opengl-tutorial.org/intermediate-tutorials/tutorial-17-quaternions/>

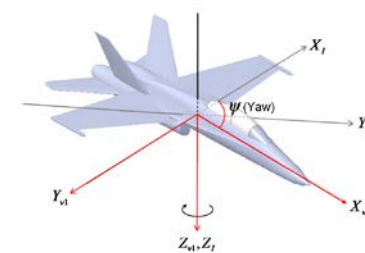
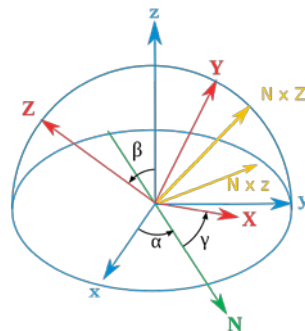
4

VISUAL  
COMPUTING Lab

## Euler Angles

KAIST

- Euler angles are the easiest way to think of a rotation.
- You basically store three rotations around the X, Y and Z axes. (For example,  $\alpha$ ,  $\beta$ ,  $\gamma$ )
- These 3 rotations are then applied successively, usually in this order: first Y, then Z, then X (but not necessarily).
- Using a different order yields different results.



COMPUTING Lab

## Euler Angles

KAIST

- Drawbacks
  - Interpolating smoothly between 2 orientations is hard. Naively interpolating the X,Y and Z angles will be ugly.
    - <https://www.youtube.com/watch?v=bnINsb0we7g>
  - Applying several rotations is complicated and imprecise
  - A well-known problem, the “**Gimbal Lock**”, will sometimes block your rotations, and other singularities which will flip your model upside-down.
    - Gimbal Lock: [https://en.wikipedia.org/wiki/Gimbal\\_lock](https://en.wikipedia.org/wiki/Gimbal_lock)
  - Different angles make the same rotation
    - E.g.,  $-180^\circ$  and  $180^\circ$
  - Usually the right order is YZX, but if you use a library with a different order, you’ll be in trouble.
  - Some operations are complicated
    - E.g., rotation of N degrees around a specific axis.

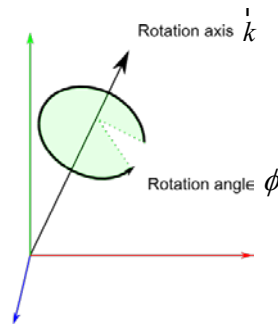
6

VISUAL COMPUTING Lab

# Quaternions

KAIST

- A quaternion is a set of 4 numbers,  $[i\ j\ k\ w]$ , which represents rotations.
  - Given a **RotationAxis**  $\vec{k}$  and a **RotationAngle**  $\phi$
  - $i = k.x * \sin(\phi / 2)$
  - $j = k.y * \sin(\phi / 2)$
  - $k = k.z * \sin(\phi / 2)$
  - $w = \cos(\phi / 2)$



7

VISUAL COMPUTING Lab

# Quaternions

KAIST

- How to implement?
  - You don't need to!
  - `Matrix.rotateM()` does it for you
  - <https://developer.android.com/reference/android/opengl/Matrix.html>

```

59         if (e.getAction() == MotionEvent.ACTION_MOVE) {
60             switch (mode) {
61                 case 0:
62                     if (count == 1) {
63                         // Rotate world
64                         float[] rot = temp1;
65
66                         Matrix.setIdentityM(rot, 0);
67                         Matrix.rotateM(rot, 0, dx, 0, 1, 0);
68                         Matrix.rotateM(rot, 0, dy, 1, 0, 0);
69                         Matrix.multiplyMM(temp2, 0, rot, 0, mRenderer.mViewRotationMatrix, 0);
70                         System.arraycopy(temp2, 0, mRenderer.mViewRotationMatrix, 0, 16);
71                     } else if (count == 2) {
72                         // Translate world
73                         Matrix.translateM(mRenderer.mViewTranslationMatrix, 0, dx / 100, -dy / 100, 0);
74                     }
75                     break;

```

8

VISUAL COMPUTING Lab

# Quaternions

- Matrix.rotateM()

## rotateM

Added in API level 1

```
void rotateM (float[] m,
             int mOffset,
             float a,
             float x,
             float y,
             float z)
```

Rotates matrix m in place by angle a (in degrees) around the axis (x, y, z).

Parameters	
m	float: source matrix
mOffset	int: index into m where the matrix starts
a	float: angle to rotate in degrees
x	float: X axis component
y	float: Y axis component
z	float: Z axis component

# Matrix.rotateM()

```
580 public static void setRotateM(float[] m, int mOffset,
581 float a, float x, float y, float z) {
582     m[mOffset + 3] = 0;
583     m[mOffset + 7] = 0;
584     m[mOffset + 11] = 0;
585     m[mOffset + 12] = 0;
586     m[mOffset + 13] = 0;
587     m[mOffset + 14] = 0;
588     m[mOffset + 15] = 1;
589     a = (float) (Math.PI / 180.0f);
590     float s = (float) Math.sin(a);
591     float c = (float) Math.cos(a);
592     if (1.0f == x && 0.0f == y && 0.0f == z) {
593         m[mOffset + 5] = c; m[mOffset + 10] = c;
594         m[mOffset + 6] = s; m[mOffset + 9] = -s;
595         m[mOffset + 1] = 0; m[mOffset + 2] = 0;
596         m[mOffset + 4] = 0; m[mOffset + 8] = 0;
597         m[mOffset + 0] = 1;
598     } else if (0.0f == x && 1.0f == y && 0.0f == z) {
599         m[mOffset + 0] = c; m[mOffset + 10] = c;
600         m[mOffset + 8] = s; m[mOffset + 2] = -s;
601         m[mOffset + 1] = 0; m[mOffset + 4] = 0;
602         m[mOffset + 6] = 0; m[mOffset + 9] = 0;
603         m[mOffset + 5] = 1;
604     } else if (0.0f == x && 0.0f == y && 1.0f == z) {
605         m[mOffset + 0] = c; m[mOffset + 5] = c;
606         m[mOffset + 1] = s; m[mOffset + 4] = -s;
607         m[mOffset + 2] = 0; m[mOffset + 6] = 0;
608         m[mOffset + 8] = 0; m[mOffset + 9] = 0;
609         m[mOffset + 10] = 1;
610     } else {
611         float len = length(x, y, z);
612         if (1.0f != len) {
613             float recipLen = 1.0f / len;
614             x *= recipLen;
615             y *= recipLen;
616             z *= recipLen;
617         }
618         float nc = 1.0f - c;
619         float xy = x + y;
620         float yz = y + z;
621         float zx = z + x;
622         float xs = x + s;
623         float ys = y + s;
624         float zs = z + s;
625         m[mOffset + 0] = xc*nc + c;
626         m[mOffset + 4] = xy*nc - zs;
627         m[mOffset + 8] = zy*nc + ys;
628         m[mOffset + 1] = xy*nc + zs;
629         m[mOffset + 5] = yz*nc + c;
630         m[mOffset + 9] = yz*nc - xs;
631         m[mOffset + 2] = zx*nc - ys;
632         m[mOffset + 6] = yz*nc + xs;
633         m[mOffset + 10] = z*nc + c;
634     }
635 }
```

Rotation using Euler angles

Rotation using Quaternions

KAIST

# Trackball and Arcball

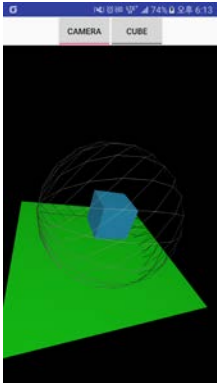
11

VISUAL COMPUTING Lab

KAIST

## Trackball and Arcball

- How can we link screen touch to object rotation?
- We want the feeling of pushing a sphere around
- A sphere can be a cue for rotation
- We want path invariant (Arcball)



12

VISUAL COMPUTING Lab

## Trackball and Arcball

KAIST

- Scenario
  - A user touches on the screen at some pixel  $s_1$  over the sphere in the image
  - The user drags to some other pixel  $s_2$  over the sphere

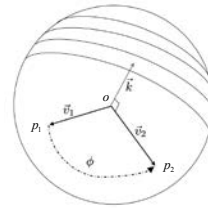
- Definition

$$v_1 = (p_1 - o)$$

$$v_2 = (p_2 - o)$$

$$\phi = \cos^{-1}(v_1 \cdot v_2)$$

$$k = \text{normalize}(v_1 \times v_2)$$



$p_1$ : 3D point of  $s_1$

$p_2$ : 3D point of  $s_2$

$o$ : 3D point of the center of the sphere

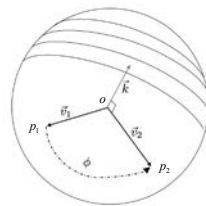
13

VISUAL  
COMPUTING Lab

## Trackball and Arcball

KAIST

- Trackball
  - $M$  is the rotation of  $\phi$  degrees about the axis  $k$
  - Actual feeling of grabbing and dragging a sphere
- Arcball
  - $M$  is the rotation of  $2\phi$  degrees about the axis  $k$
  - Path independent rotation



$$v_1 = (p_1 - o)$$

$$v_2 = (p_2 - o)$$

$$\phi = \cos^{-1}(v_1 \cdot v_2)$$

$$k = \text{normalize}(v_1 \times v_2)$$

14

VISUAL  
COMPUTING Lab

## Implementation

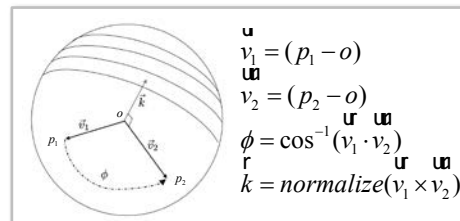
KAIST

- Given the  $(x,y)$  window coordinates of touch, the  $z$  coordinate on the sphere can be solved using

$$(x - c_x)^2 + (y - c_y)^2 + (z - 0)^2 = r^2, \quad z > 0$$

- $[c_x, c_y, 0]^T$  is the window coordinates (in pixels) of the center of the sphere

- Use *Matrix.rotateM()*



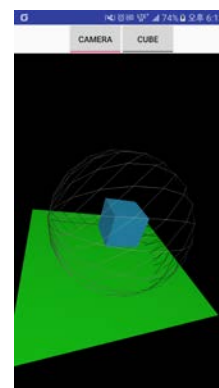
15

VISUAL COMPUTING Lab

## Task

KAIST

- Implement Arcball for
  - Rotating CAMERA
  - Rotating CUBE
- Press 'CAMERA' button for manipulating an Arcball for 'CAMERA'
- Press 'CUBE' button for manipulating an Arcball for 'CUBE'



16

VISUAL COMPUTING Lab