CS482 Lab Session

Quaternion and Arcball

2016. 9. 28

VISUAL COMPUTING Lab

Rotation

Rotation

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

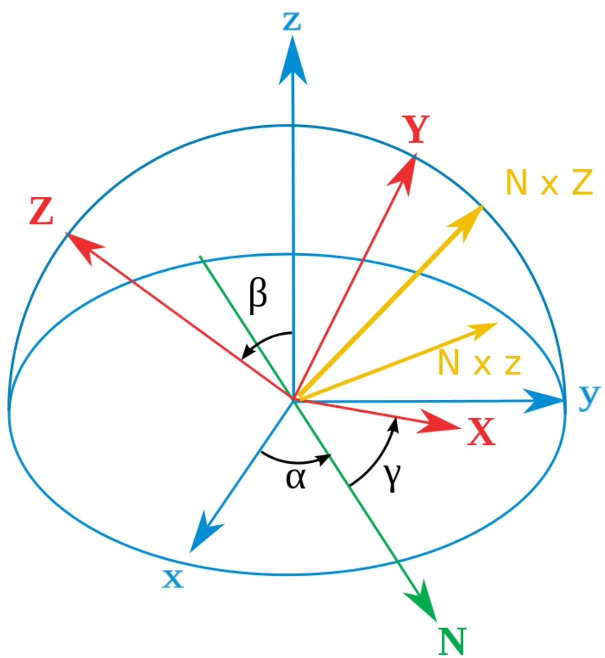
```java
if (e.getAction() == MotionEvent.ACTION_MOVE) {
    switch (mode) {
        case 0:
            if (count == 1) {
                // Rotate world
                float[] rot = temp1;

                Matrix.setIdentityM(rot, 0);
                Matrix.rotateM(rot, 0, dx, 0, 1, 0);
                Matrix.rotateM(rot, 0, dy, 1, 0, 0);
                Matrix.multiplyMM(temp2, 0, rot, 0, mRenderer.mViewRotationMatrix, 0);
                System.arraycopy(temp2, 0, mRenderer.mViewRotationMatrix, 0, 16);
            } else if (count == 2) {
                // Translate world
                Matrix.translateM(mRenderer.mViewTranslationMatrix, 0, dx/ 100, -dy / 100, 0);
            }
            break;
```

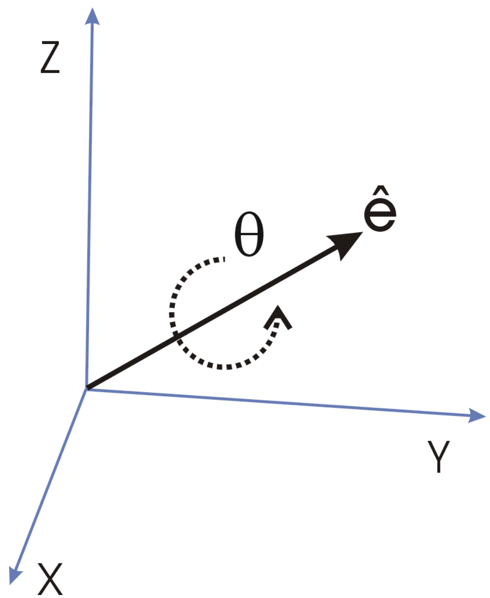
Rotate ‘dx’ degree w.r.t. Y-axis
Rotate ‘dy’ degree w.r.t. X-axis

Rotation

- Euler Angles vs. Quaternions



Rotation using Euler Angles



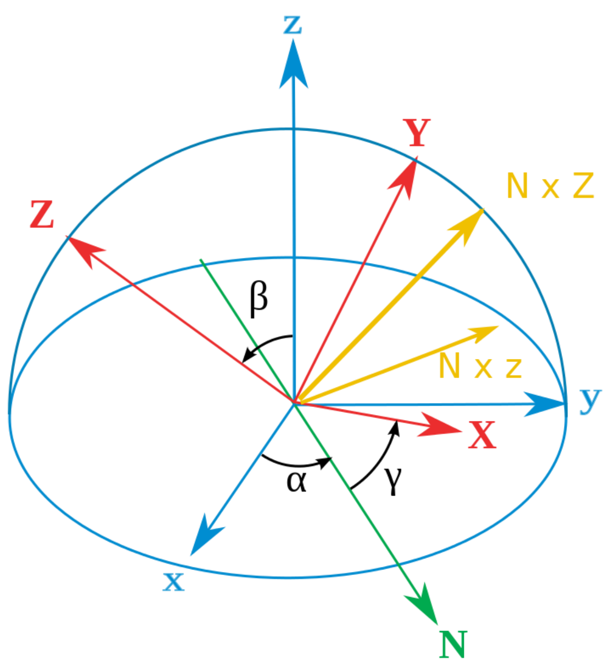
Rotation using Quaternions

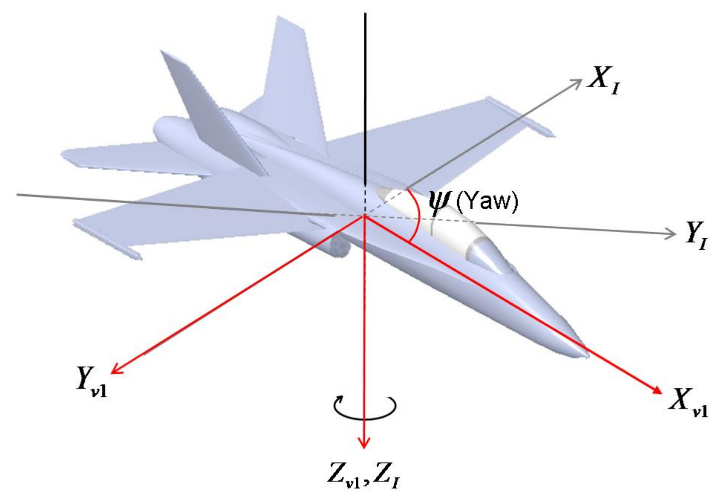
Reference: http://www.opengl-tutorial.org/intermediate-tutorials/tutorial-17-quaternions/

VISUAL COMPUTING Lab

Euler Angles

- 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, α, β, γ)
- 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.



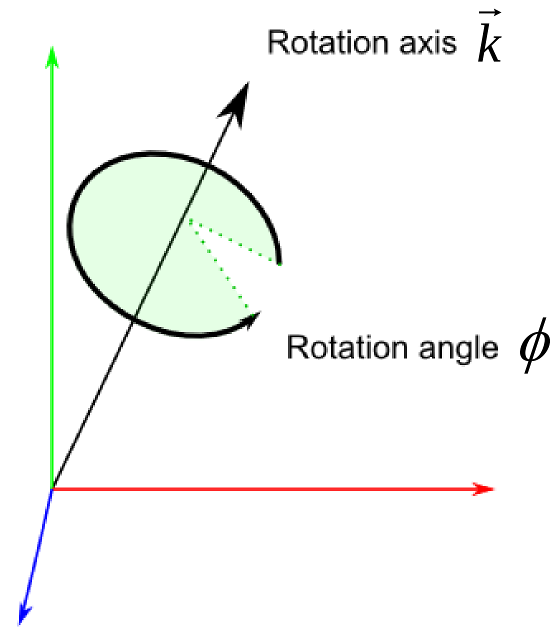


Euler Angles

- 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
  - Different angles make the same rotation
    - E.g., -180° and 180°
  - 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.

Quaternions

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



Quaternions

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

```java
if (e.getAction() == MotionEvent.ACTION_MOVE) {
    switch (mode) {
        case 0:
            if (count == 1) {
                // Rotate world
                float[] rot = temp1;

                Matrix.setIdentityM(rot, 0);
                Matrix.rotateM(rot, 0, dx, 0, 1, 0);
                Matrix.rotateM(rot, 0, dy, 1, 0, 0);
                Matrix.multiplyMM(temp2, 0, rot, 0, mRenderer.mViewRotationMatrix, 0);
                System.arraycopy(temp2, 0, mRenderer.mViewRotationMatrix, 0, 16);
            } else if (count == 2) {
                // Translate world
                Matrix.translateM(mRenderer.mViewTranslationMatrix, 0, dx/ 100, -dy / 100, 0);
            }
            break;
```

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()

KAIST

```java
public static void setRotateM(float[] rm, int rmOffset,
        float a, float x, float y, float z) {
    rm[rmOffset + 3] = 0;
    rm[rmOffset + 7] = 0;
    rm[rmOffset + 11]= 0;
    rm[rmOffset + 12]= 0;
    rm[rmOffset + 13]= 0;
    rm[rmOffset + 14]= 0;
    rm[rmOffset + 15]= 1;
    a *= (float) (Math.PI / 180.0f);
    float s = (float) Math.sin(a);
    float c = (float) Math.cos(a);
    if (1.0f == x && 0.0f == y && 0.0f == z) {
        rm[rmOffset + 5] = c;   rm[rmOffset + 10]= c;
        rm[rmOffset + 6] = s;   rm[rmOffset + 9] = -s;
        rm[rmOffset + 1] = 0;   rm[rmOffset + 2] = 0;
        rm[rmOffset + 4] = 0;   rm[rmOffset + 8] = 0;
        rm[rmOffset + 0] = 1;
    } else if (0.0f == x && 1.0f == y && 0.0f == z) {
        rm[rmOffset + 0] = c;   rm[rmOffset + 10]= c;
        rm[rmOffset + 8] = s;   rm[rmOffset + 2] = -s;
        rm[rmOffset + 1] = 0;   rm[rmOffset + 4] = 0;
        rm[rmOffset + 6] = 0;   rm[rmOffset + 9] = 0;
        rm[rmOffset + 5] = 1;
    } else if (0.0f == x && 0.0f == y && 1.0f == z) {
        rm[rmOffset + 0] = c;   rm[rmOffset + 5] = c;
        rm[rmOffset + 1] = s;   rm[rmOffset + 4] = -s;
        rm[rmOffset + 2] = 0;   rm[rmOffset + 6] = 0;
        rm[rmOffset + 8] = 0;   rm[rmOffset + 9] = 0;
        rm[rmOffset + 10]= 1;
    } else {
        float len = length(x, y, z);
        if (1.0f != len) {
            float recipLen = 1.0f / len;
            x *= recipLen;
            y *= recipLen;
            z *= recipLen;
        }
        float nc = 1.0f - c;
        float xy = x * y;
        float yz = y * z;
        float zx = z * x;
        float xs = x * s;
        float ys = y * s;
        float zs = z * s;
        rm[rmOffset +  0] = x*x*nc +  c;
        rm[rmOffset +  4] =  xy*nc - zs;
        rm[rmOffset +  8] =  zx*nc + ys;
        rm[rmOffset +  1] =  xy*nc + zs;
        rm[rmOffset +  5] = y*y*nc +  c;
        rm[rmOffset +  9] =  yz*nc - xs;
        rm[rmOffset +  2] =  zx*nc - ys;
        rm[rmOffset +  6] =  yz*nc + xs;
        rm[rmOffset + 10] = z*z*nc +  c;
    }
}
```

Rotation using Euler angles

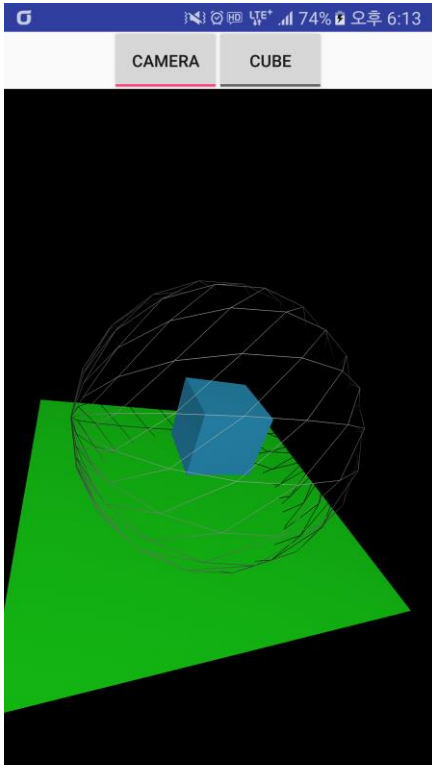
Rotation using Quaternions

VISUAL COMPUTING Lab

Trackball and Arcball

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)



Trackball and Arcball

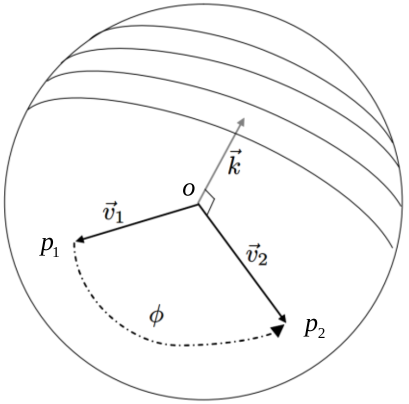
- Scenario
  - A user touches on the screen at some pixel s_1 over the sphere in the image
  - The user drags to some other pixel s_2 over the sphere
- Definition

$$\overrightarrow{v_1} = (p_1 - o)$$

$$\overrightarrow{v_2} = (p_2 - o)$$

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

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



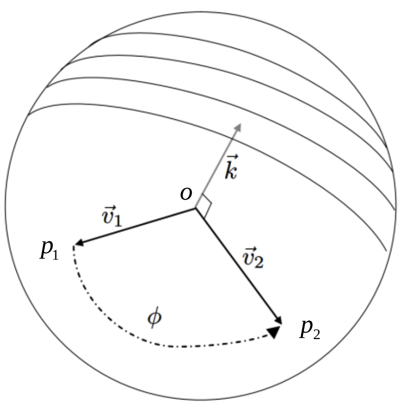
p_1 : 3D point of s_1

p_2 : 3D point of s_2

o : 3D point of the center of the sphere

Trackball and Arcball

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



$$\overrightarrow{v_1} = (p_1 - o)$$

$$\overrightarrow{v_2} = (p_2 - o)$$

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

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

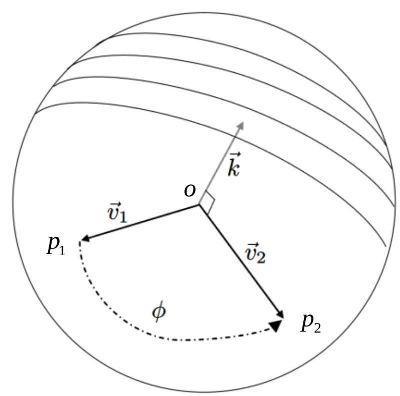
Implementation

- 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()*



$$\overrightarrow{v_1}=(p_1-o)$$
$$\overrightarrow{v_2}=(p_2-o)$$
$$\phi=\cos^{-1}(\overrightarrow{v_1}\cdot\overrightarrow{v_2})$$
$$\vec{k}=normalize(\overrightarrow{v_1}\times\overrightarrow{v_2})$$

Task

- 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’