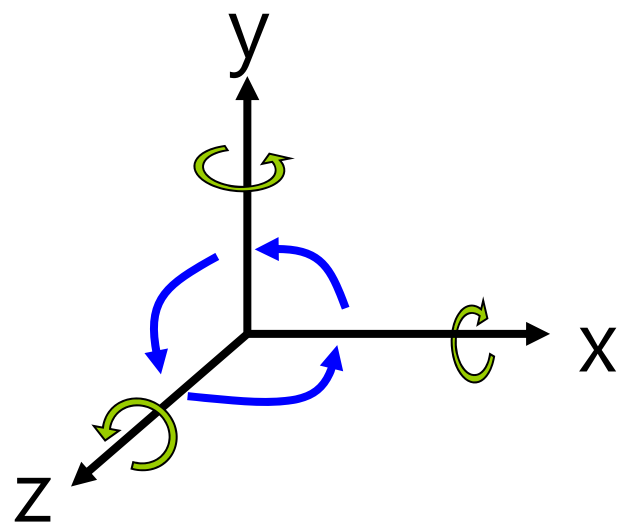
Geometric Objects and Transformation

527970
Fall 2020
10/15/2020

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



Matrix Operations

- OpenGL Matrix
  - The elements of the 4x4 matrix M must be specified in **column-major order**.

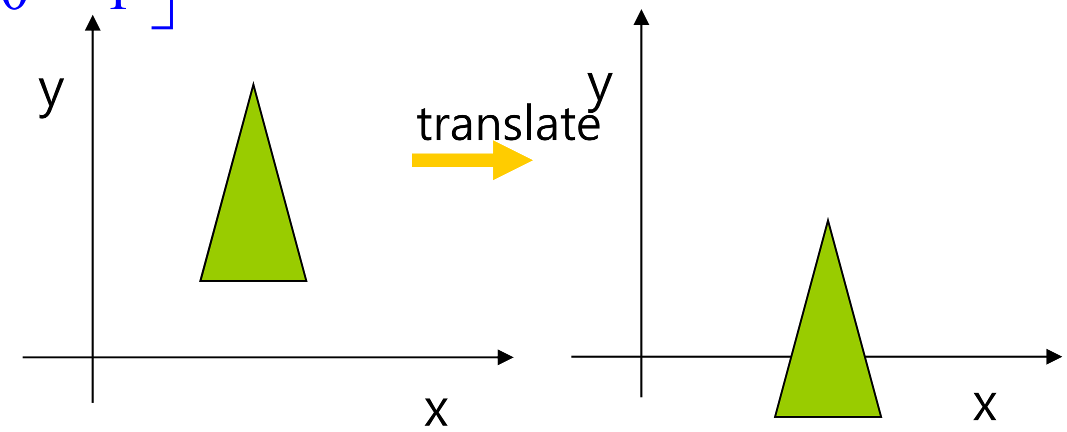
$$p' = M * p = \begin{pmatrix} m_0 & m_4 & m_8 & m_{12} \\ m_1 & m_5 & m_9 & m_{13} \\ m_2 & m_6 & m_{10} & m_{14} \\ m_3 & m_7 & m_{11} & m_{15} \end{pmatrix} \begin{bmatrix} v_0 \\ v_1 \\ v_2 \\ v_3 \end{bmatrix}$$

Translation

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

```
glm::mat4 T = glm::translate(glm::mat4(1.0f), glm::vec3(0.5f, -0.2f, 0));
```

$$p' = Tp \qquad T = \begin{bmatrix} 1 & 0 & 0 & dx \\ 0 & 1 & 0 & dy \\ 0 & 0 & 1 & dz \\ 0 & 0 & 0 & 1 \end{bmatrix}$$



Rotation

- Rotation
  - Rotate by angle around the arbitrary axis (x,y,z) (OpenGL uses degree & glm uses radian)
  - 2D rotation is used as the z-axis (0, 0, 1) rotation.

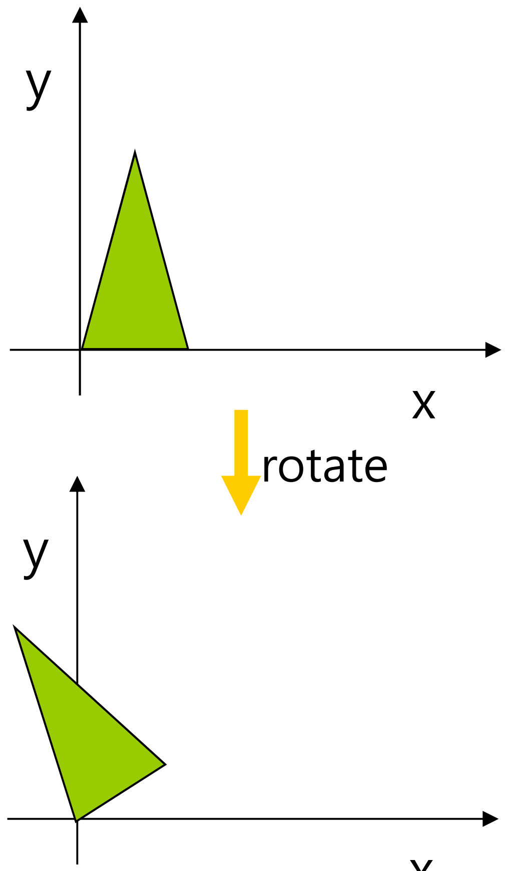
```
glm::mat4 Rx = glm::rotate(glm::mat4(1.0f), 30.0f, glm::vec3(1, 0, 0);
glm::mat4 Ry = glm::rotate(glm::mat4(1.0f), 60.0f, glm::vec3(0, 1, 0);
glm::mat4 Rz = glm::rotate(glm::mat4(1.0f), 45.0f, glm::vec3(0, 0, 1);
glm::mat4 Ra = glm::rotate(glm::mat4(1.0f), 45.0f, glm::vec3(1, 1, 1);
```

Rotation

$$p' = R_x p \qquad R_x = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta & 0 \\ 0 & \sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$p' = R_y p \qquad R_y = \begin{bmatrix} \cos\theta & 0 & \sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

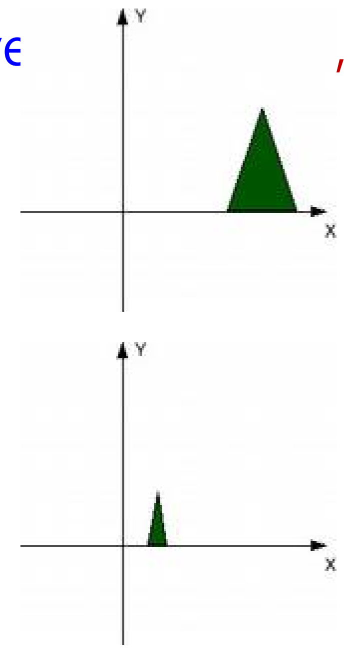
$$p' = R_z p \qquad R_z = \begin{bmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$



Scale

- Scale
  - Transform the size by sx on the x-axis, sy on the y-axis, sz on the 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.
  - 2D scaling uses z=1.

```
glm::mat4 S = glm::scale(glm::mat4(1.0f), glm::ve          , 1.0f));
```

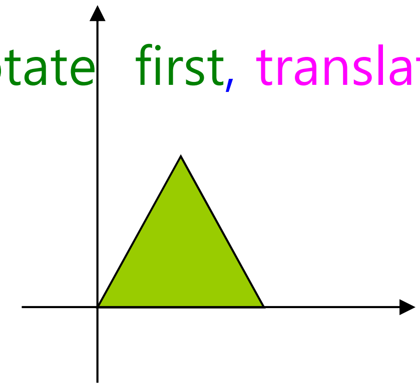


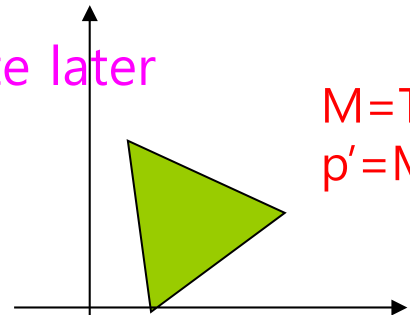
$$p' = Sp \qquad S = \begin{bmatrix} sx & 0 & 0 & 0 \\ 0 & sy & 0 & 0 \\ 0 & 0 & sz & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Transformation Order

- In OpenGL, modeling transformation matrices are applied in the reverse order set on the object.

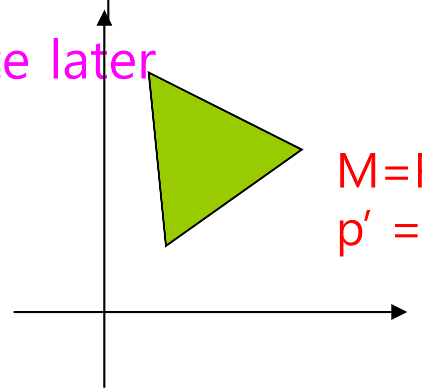
```
glm::mat4 Tx = glm::translate(glm::mat4(1.0f), glm::vec3(0.5, 0, 0));
glm::mat4 Rz =glm::rotate(glm::mat4(1.0f), 45, glm::vec3(0, 0, 1));

glm::mat4 TR = Tx * Rz; // rotate  first, translate later
drawTriangle(TR);
```





M=T*R
p'=Mp

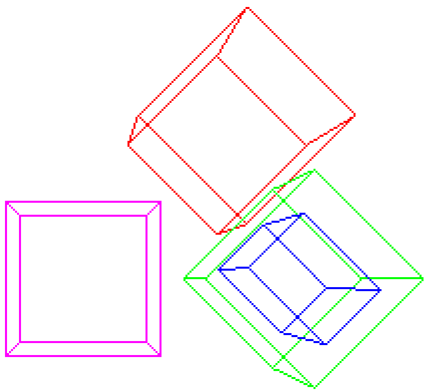
```
glm::mat4 RT = Rz * Tx; // translate first, rotate later
drawTriangle(RT);
```



M=R*T
p' = Mp

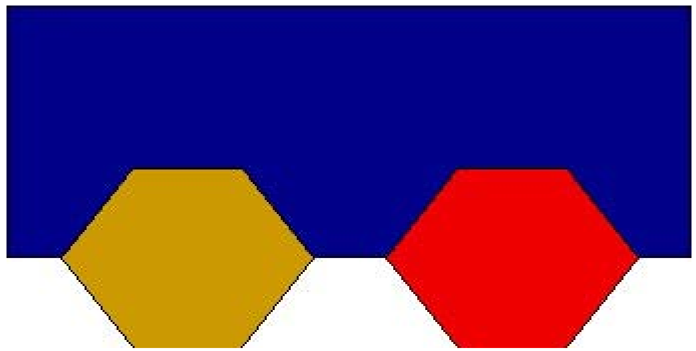
Transformation Order

```
glm::mat4 T = glm::translate(glm::mat4(1.0f), glm::vec3(1.5f, 0.0f, 0.0f));
glm::mat4 R = glm::rotate(glm::mat4(1.0f), 45.0f, glm::vec3(0.0f, 0.0f, 1.0f));
glm::mat4 S = glm::scale(glm::mat4(1.0f), glm::vec3(0.5f, 0.7f, 1.0f));

drawCube();

glm::mat4 RT = R * T; // p' = R * T * p (red)
drawCube();

glm::mat4 TR = T * R; // p' =  T * R * p (green)
drawCube();

glm::mat4 TRS = T * R * S; // p' = T * R * S * p (blue)
drawCube();
```



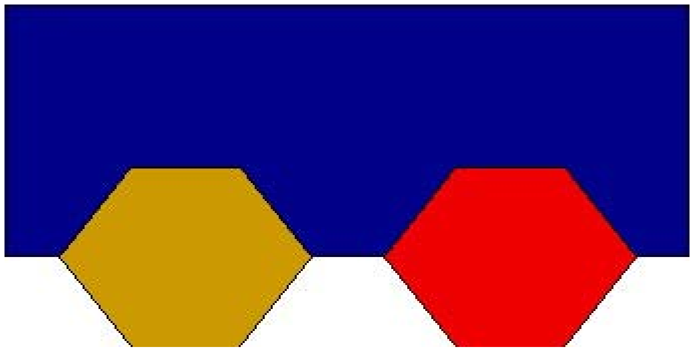
Hierarchical Transformations

- 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:
  - Apply body transformation
  - Draw body
  - Save state
  - Apply front wheel transformation
  - Draw wheel
  - Restore saved state
  - Apply rear wheel transformation
  - Draw wheel



Hierarchical Transformations

- In addition, 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.

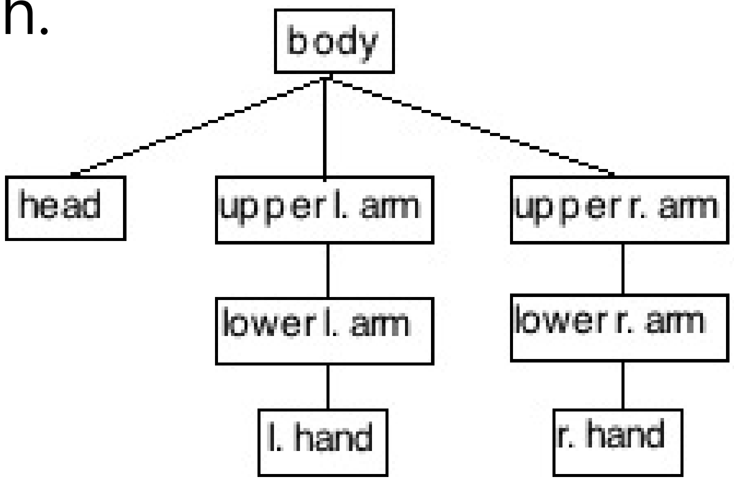


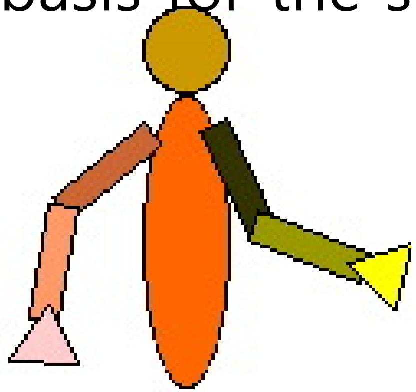
Example: Car

```
bodyTransform = glm::translate(glm::mat4(1.0f), position); // car position
wheelMatrix = MVP * bodyTransform;
body.draw();
wheelTransform[0] = glm::translate(glm::mat4(1.0f), glm::vec3(-0.5f, 0.0f, -0.5f)) *
    glm::rotate(glm::mat4(1.0f), angle, glm::vec3(0.0f, 0.0f, 1.0f));
wheelMatrix = MVP * bodyTransform * wheelTransform[0];
wheel.draw();
wheelTransform[1] = glm::translate(glm::mat4(1.0f), glm::vec3(-0.5f, 0.0f,  0.5f)) *
    glm::rotate(glm::mat4(1.0f), angle, glm::vec3(0.0f, 0.0f, 1.0f));
wheelMatrix = MVP * bodyTransform * wheelTransform[1];
wheel.draw();
wheelTransform[2] = glm::translate(glm::mat4(1.0f), glm::vec3( 0.5f, 0.0f, -0.5f)) *
    glm::rotate(glm::mat4(1.0f), angle, glm::vec3(0.0f, 0.0f, 1.0f));
wheelMatrix = MVP * bodyTransform * wheelTransform[2];
wheel.draw();
wheelTransform[3] = glm::translate(glm::mat4(1.0f), glm::vec3( 0.5f, 0.0f,  0.5f)) *
    glm::rotate(glm::mat4(1.0f), angle, glm::vec3(0.0f, 0.0f, 1.0f));
wheelMatrix = MVP * bodyTransform * wheelTransform[3];
wheel.draw();
```

Transformation Hierarchy

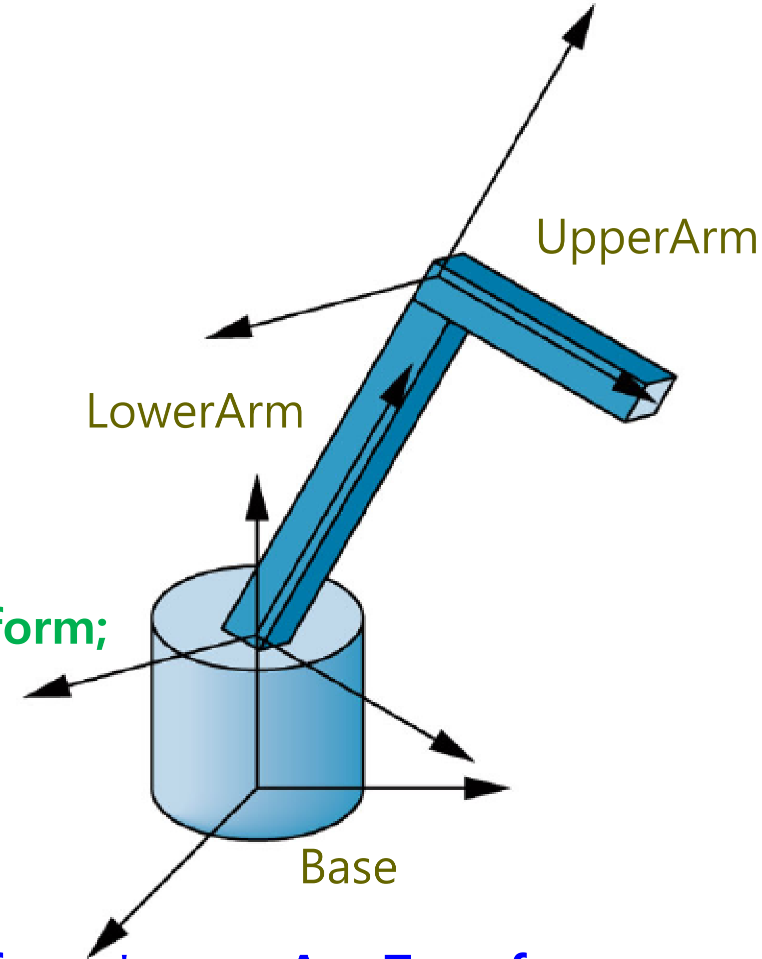
- Hierarchical transformations are often expressed as a tree structure of transformations.
- To design a three-dimensional character, we use a hierarchical transformation made of rigid body parts.
- For more flexible 3D character design, a number of hierarchical transformations should be properly mixed.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





Example: Robot

```
void SimpleRobot::draw(glm::mat4 MVP)
{
// base
glm::mat4 baseMatrix =
    MVP * baseTransform;
base.draw();

// lowerArm
glm::mat4 lowerArmMatrix =
    MVP * baseTransform * lowerArmTransform;
arm.draw();

// upperArm
glm::mat4 upperArmMatrix  =
    MVP * baseTransform * lowerArmTransform * upperArmTransform;
arm.draw();
}
```

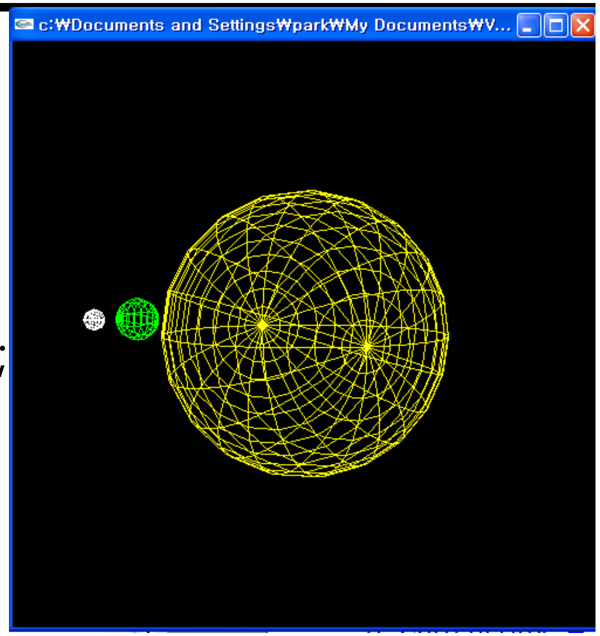


Example: Robot

```
SimpleRobot::SimpleRobot(Program* p_) {
    theta = 0.0f; phi = 0.0f; psi = 0.0f; // base, lower/upper arm rotation
    base = Cylinder(2.0f, 3.0f, 16);
    arm = Parallelpiped(glm::vec3(-0.25f, 0.0f, -0.25f), glm::vec3(0.5f, 0.0f,
            0.0f), glm::vec3(0.0f, 0.0f, 0.5f), glm::vec3(0.0f, 3.0f, 0.0f));
}
bool SimpleRobot::update(float deltaTime) {
    baseTransform = glm::translate(glm::mat4(1.0f), glm::vec3(0.0f, -1.5f,
    0.0f)) * glm::rotate(glm::mat4(1.0f), theta, glm::vec3(0.0f, 1.0f, 0.0f));
    lowerArmTransform = glm::translate(glm::mat4(1.0f), glm::vec3(0.0f,
    1.5f, 0.0f)) * glm::rotate(glm::mat4(1.0f), phi, glm::vec3(0.0f, 0.0f, 1.0f));
    upperArmTransform = glm::translate(glm::mat4(1.0f), glm::vec3(0.0f,
    3.0f, 0.0f)) * glm::scale(glm::mat4(1.0f), glm::vec3(1.0f, 0.5f, 1.0f)) *
    glm::rotate(glm::mat4(1.0f), psi, glm::vec3(0.0f, 0.0f, 1.0f));
return true;
}
```

Example: Solar

```
const float SimpleSolar::SunRadius = 4.0f;
const float SimpleSolar::EarthRadius = 1.0f;
const float SimpleSolar::MoonRadius = 0.5f;
const float SimpleSolar::EarthDistanceFromSun = 10.0f;
const float SimpleSolar::MoonDistanceFromEarth = 2.0f;

SimpleSolar::SimpleSolar(Program* p_)
{   p = p_;
    sunSpin = 0.0f;// sun spin
    earthSpin = 0.0f;// earth spin
    earthOrbit = 0.0f;// earth orbit around the sun
    moonSpin = 0.0f;// moon spin
    moonOrbit = 0.0f;// moon orbit around the earth
    sun = Sphere(SunRadius, 16, 16);
    earth = Sphere(EarthRadius, 16, 16);
    moon = Sphere(MoonRadius, 16, 16);
}
```



Example: Solar

```
bool SimpleSolar::update(float deltaTime)
{
// The Sun spins by rotating it about y-axis
sunSpin += (float) (deltaTime) * 0.01f;
sunTransform = glm::rotate(glm::mat4(1.0f), sunSpin, glm::vec3(0.0f, 1.0f,
    0.0f));

// The Earth spins on its own axis and orbits the Sun
earthSpin += (float) (deltaTime) * 0.05f;
earthOrbit += (float) (deltaTime) * 0.01f;
earthTransform = glm::rotate(glm::mat4(1.0f), earthOrbit, glm::vec3(0.0f,
    1.0f, 0.0f))
* glm::translate(glm::mat4(1.0f), glm::vec3(0.0f, 0.0f,
    EarthDistanceFromSun))
* glm::rotate(glm::mat4(1.0f), earthSpin, glm::vec3(0.0f, 1.0f, 0.0f));
```

Example: Solar

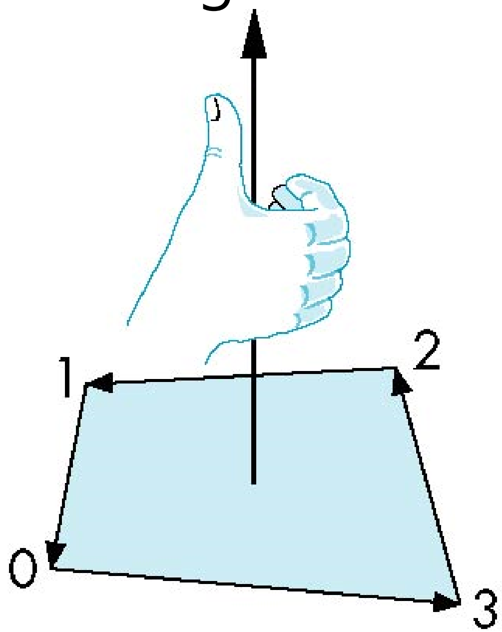
```
// The Moon spins on its own axis and orbits the Earth (that orbits the Sun)
moonSpin += (float) (deltaTime) * 0.07f;
moonOrbit += (float) (deltaTime) * 0.08f;
moonTransform = glm::rotate(glm::mat4(1.0f), earthOrbit, glm::vec3(0.0f,
    1.0f, 0.0f))
* glm::translate(glm::mat4(1.0f), glm::vec3(0.0f, 0.0f,
    EarthDistanceFromSun))
* glm::rotate(glm::mat4(1.0f), moonOrbit, glm::vec3(0.0f, 1.0f, 0.0f))
* glm::translate(glm::mat4(1.0f), glm::vec3(0.0f, 0.0f,
    MoonDistanceFromEarth))
* glm::rotate(glm::mat4(1.0f), moonSpin, glm::vec3(0.0f, 1.0f, 0.0f));

return true;
}
```

Example: Solar

```
void SimpleSolar::draw(glm::mat4 MVP) {
    glm::mat4 sunMatrix = MVP * sunTransform;
    p->useProgram();
    p->setUniform("gMVP", sunMatrix);
    glVertexAttrib3f(1, 1, 1, 0); // yellow - sun
    sun.draw();
    glm::mat4 earthMatrix = MVP * earthTransform;
    p->useProgram();
    p->setUniform("gMVP", earthMatrix);
    glVertexAttrib3f(1, 0, 1, 0); // green - earth
    earth.draw();
    glm::mat4 moonMatrix = MVP * moonTransform;
    p->useProgram();
    p->setUniform("gMVP", moonMatrix);
    glVertexAttrib3f(1, 0.5, 0.5, 0.5); // gray - moon
    moon.draw();
}
```

Modeling a Cube

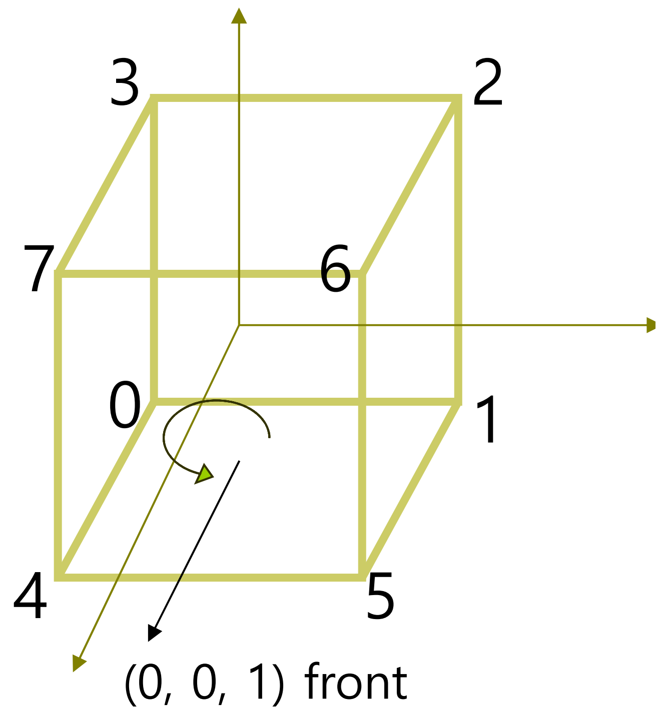
- In OpenGL, the winding order of vertices $\{v_0, v_3, v_2, v_1\}$ and $\{v_1, v_0, v_3, v_2\}$ creates the same polygon. However, the vertex winding order $\{v_1, v_2, v_3, v_0\}$ is different.
- Since OpenGL uses the right-handed coordinate system, if a vertex is defined with a counter-clockwise encirclement, it creates a normal vector facing outward.



Modeling a Cube

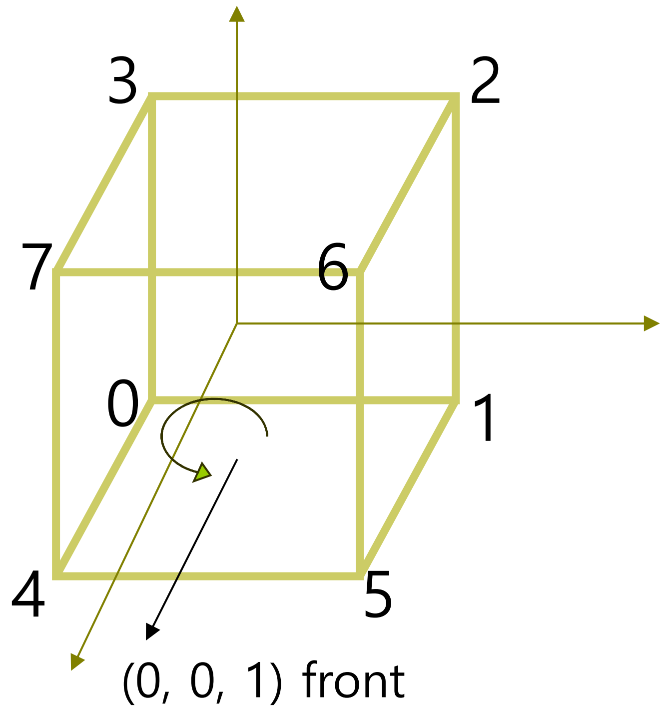
- Draw a cube using the vertex list and the index list.

```
GLfloat cubeVertices[][3] = {
        {-1.0,-1.0,-1.0},    { 1.0,-1.0,-1.0},
        { 1.0, 1.0,-1.0},    {-1.0, 1.0,-1.0},
        {-1.0,-1.0, 1.0},    { 1.0,-1.0, 1.0},
        { 1.0, 1.0, 1.0},    {-1.0, 1.0, 1.0}
};

GLfloat cubeNormals[][3] = {
        { 0.0,  0.0,  1.0},   // front
        { 0.0,  0.0, -1.0},   // back
        {-1.0,  0.0,  0.0},   // left
        { 1.0,  0.0,  0.0},   // right
        { 0.0,  1.0,  0.0},   // top
        { 0.0, -1.0,  0.0},   // bottom
};
```



Modeling a Cube

```
GLint cubeIndices[] = {
        4, 5, 6, 4, 6, 7, // front
        1, 0, 3, 1, 3, 2, // back
        0, 4, 7, 0, 7, 3, // left
        5, 1, 2, 5, 2, 6, // right
        7, 6, 2, 7, 2, 3, // top
        0, 1, 5, 0, 5, 4 // bottom
};
```



Model Files

- 3D model types
  - Wavefront (.obj)
  - Inventor (.iv)
  - VRML / X3D
  - 3D Studio (.3ds)
  - OpenFlight (.flt)
  - …
- The 3D object model contains the following:
  - Geometry data – vertex positions, faces
  - Colors/material properties
  - Textures
  - Transformations

Wavefront OBJ Files

- An OBJ file is a plain text file that contains vertices, polygon faces, materials, and many other information.
- Each line starts with a token that tells us what kind of information it has, such as vertex, normal vector, and texture.
  - v *x y z*
    - Vertex position
  - vn *x y z*
    - Vertex normal
  - vt *u v*
    - texture coordinate
  - f *v1 v2 v3* ..
    - Face (list of vertex numbers)
  - Mtllib *file.mtl*
    - File containing material descriptions
  - Usemtl *name*
    - Current material to apply to geometry