

# OpenGL 좌표계 변환 1

2024년 2학기

# GLM 라이브러리 사용하기

- GLM (GL Mathematics)
  - GLM (GL Mathematics)는 OpenGL Shading Language를 기반으로 하는 그래픽 소프트웨어에서 사용할 수 있는 GLSL 기반의 **header-only C++ 수학 라이브러리**
  - 모던 openGL은 변환 관련 함수들과 카메라 함수 등이 더 이상 지원되지 않는다. 따라서 이런 작업들을 하기 위하여 glm 이 제공하는 함수들을 사용한다.
    - 함수들을 사용할 때 네임스페이스를 사용하지 말고 "glm::" 문법을 사용하여 함수들을 호출하도록 한다.
    - 참조 사이트: <https://glm.g-truc.net/0.9.9/index.html>
- GLM 라이브러리 사용하기
  - 헤더파일 포함하기

```
#include <gl/glm/glm.hpp>
#include <gl/glm/ext.hpp>
#include <gl/glm/gtc/matrix_transform.hpp>
```
- 기본 데이터 타입
  - Vector type: `vec{2|3|4}`, `bvec{2|3|4}`, `ivec{2|3|4}`, `uvec{2|3|4}`
  - Matrix type: `dmat{2|3|4}`, `dmat2x{2|3|4}`, `dmat3x{2|3|4}`, `dmat4x{2|3|4}`, `mat{2|3|4}`, `mat2x{2|3|4}`, `mat3x{2|3|4}`, `mat4x{2|3|4}`

# GLM 라이브러리 사용하기

- 생성자

- `glm::mat4();` //--- 4x4 행렬
- `Glm::vec4();` //--- 4-요소 벡터
- `glm::vec3();` //--- 3-요소 벡터

- 초기화

- `glm::mat4 (1.0);` //--- GLM 0.9.9 버전부터는 초기화된 기본 행렬이 단위 행렬이 아니라 0으로 초기화되어 있음  
//--- 사용하기 위해서는 `glm::mat4 M = glm::mat4 (1.0f);`와 같이 행렬을 초기화해야한다.

- 행렬 곱셈

- `glm::mat4() * glm::mat4();`
- `glm::mat4() * glm::vec4;`
- `glm::mat4() * glm::vec4 (glm::vec3, 1);`

- 변환 함수

- `glm::mat4 glm::rotate (glm::mat4 const&m, float angle, glm::vec3 const& axis);`
- `glm::mat4 glm::scale (glm::mat4 const&m, glm::vec3 const& factors);`
- `glm::mat4 glm::translate (glm::mat4 const&m, glm::vec3 const & translation);`

# GLM 라이브러리 사용하기

- 뷰잉 볼륨

- glm::mat4 glm::ortho (float left, float right, float bottom, float top, float near, float far);
- glm::mat4 glm::frustum (float left, float right, float bottom, float top, float near, float far);
- glm::mat4 glm::perspective (float fovy, float aspect, float near, float far);

- 카메라 조정

- glm::mat4 glm::lookAt (glm::vec3 const &eye, glm::vec3 const&look, glm::vec3 const &up);

- 수학 함수 (genType: float, integer, scalar 또는 vector types)

- genType glm::abs (genType x);
- vec4f glm::ceil (vec4f x);
- genType glm::clamp (genType x, genType minVal, genType maxVal);
- int glm::floatBitsToInt (float const &v);
- vec4f glm::floor (vec4f const &x);
- vec3f glm::cross (vec3f const &x, vec3f const &y);
- float glm::distance (vec4f const &p0, vec4f const &p1);
- float glm::dot (vec4f const &x, vec4f const &y);
- float glm::length (vec4f const &x);
- vec4f glm::normalize (vec4f const&x);

# GLM 라이브러리 사용하기

- 삼각 함수

- `float glm::sin (float const &angle);`
- `float glm::cos (float const &angle);`
- `float glm::tan (float const &angle);`
- `float glm::degrees (float const &radians);`
- `float glm::radians (float const &degrees);`

- Input 인자의 데이터 주소 가져오기

- `genType glm::value_ptr (genType const &vec);`

- `#include <glm/gtc/type_ptr.hpp>`

- 사용 예)

```
void f () {  
    glm::vec3 aVector( 3 );  
    glm::mat4 someMatrix( 1.0f );  
    glUniform3fv ( uniformLoc, 1, glm::value_ptr ( aVector ) );  
    glUniformMatrix4fv ( uniformMatrixLoc, 1, GL_FALSE, glm::value_ptr (someMatrix ) );  
    //--- 또는, glUniformMatrix4fv ( uniformMatrixLoc, 1, GL_FALSE, &someMatrix[0][0] );  
}
```

# 동차 좌표

- 동차 좌표 (Homogeneous Coordinates)
  - 벡터 공간: 크기와 방향이 같으면 같은 벡터로 취급
  - 어파인 공간: 크기와 방향이 같아도 시작 위치가 다르면 다름
    - 점과 벡터를 동족으로 취급하여 벡터 공간을 확장: 점과 벡터와의 덧셈 추가됨
    - 그러나, 점과 벡터의 표현이 다름
      - $v = (x, y, z)$
      - $p = (x, y, z, w)$
  - 3차원을 하나 올려 4개의 요소로 표시해서 동일 방법으로 표시
    - $(x, y, z) \rightarrow (x, y, z, w)$ 
      - $w = 0 \rightarrow$  벡터
      - $w = 1 \rightarrow$  점
    - 벡터와 점을 동일한 방법으로 표현
      - $v = (x, y, z, 0)$  //--- 벡터
      - $P = (x, y, z, 1)$  //--- 점
    - 모든 기하 변환 행렬을 곱셈으로 표시할 수 있음
  - 3차원 동차 좌표를 4차원  $(x, y, z, w)$  좌표로 표시하면  $\rightarrow$  3차원 실제 좌표는  $(x/w, y/w, z/w)$   
 $(1, 2, 4, 1) == (2, 4, 8, 2) == (5, 10, 20, 5)$

# 변환 행렬

- 변환 행렬
  - 3차원 변환 시, 동차 좌표계를 이용한 4x4 행렬 사용
  - 행렬 곱셈 순서: 행렬 x 좌표값 = 변형된 좌표값

$$\begin{bmatrix} a & b & c & d \\ e & f & g & h \\ i & j & k & l \\ m & n & o & p \end{bmatrix} * \begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix} = \begin{bmatrix} ax + by + cz + dw \\ ex + fy + gz + hw \\ ix + jy + kz + lw \\ mx + ny + oz + pw \end{bmatrix}$$

- 응용 프로그램에서

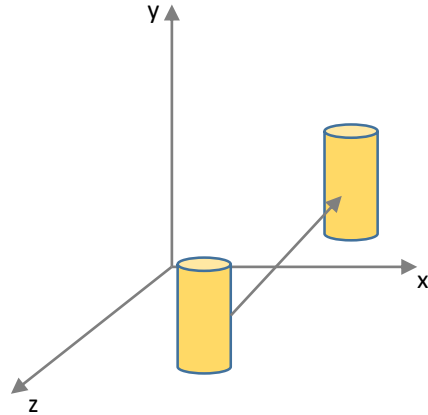
```
glm::mat4 myMatrix;
glm::vec4 myVector;
glm::vec4 transformedVector = myMatrix * myVector;
```
- GLSL에서

```
mat4 myMatrix;
vec4 myVector;
vec4 transformedVector = myMatrix * myVector;
```

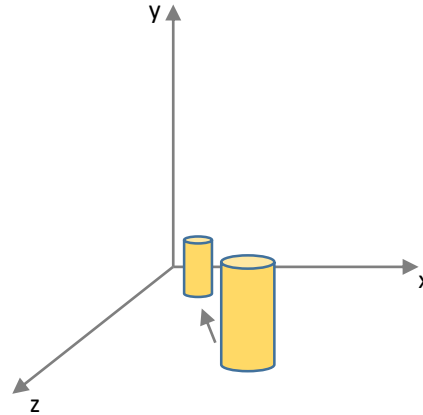
# 기하 변환

- 기하 변환 (Geometric Transformation)

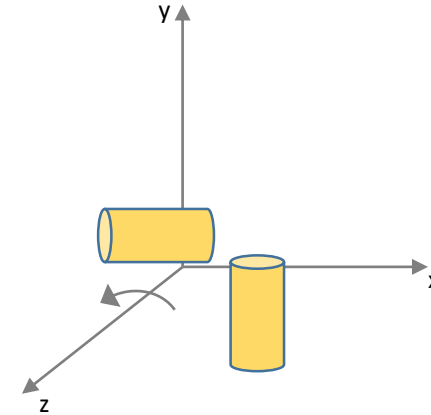
- 물체 또는 좌표계 변환의 기본으로 하나 또는 그 이상의 기하학적 객체를 나타내는 점들을 새로운 위치로 옮기는 것
- 행렬로 표현됨
- 기본 기하 변환
  - 이동 (Translation):  $(x', y', z') = (x + dx, y + dy, z + dz)$
  - 회전 (Rotation):  $(x', y', z') = (x\cos\theta - y\sin\theta, x\sin\theta + y\cos\theta, z)$
  - 신축 (Scale) :  $(x', y', z') = (sx \cdot x, sy \cdot y, sz \cdot z)$



이동



신축



z축 회전



# 기하 변환

- 이동 (Translation): 물체를 구성하는 정점을 동일한 양만큼 움직이기 →  $(d_x, d_y, d_z)$ : 이동 벡터 값

$$\begin{bmatrix} 1 & 0 & 0 & d_x \\ 0 & 1 & 0 & d_y \\ 0 & 0 & 1 & d_z \\ 0 & 0 & 0 & 1 \end{bmatrix} * \begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix} = \begin{bmatrix} d_x + x \\ d_y + y \\ d_z + z \\ 1 \end{bmatrix}$$

- GLM 함수

- `glm::mat4 glm::translate (glm::mat4 const &M, glm::vec3 const & Translation);`
  - `M`: 입력된 이동 행렬
  - `Translation`: 이동 변환 인자
  - 리턴값: 3 vector 요소 Translation 으로 생성된 4x4 행렬

- 사용 예) (2.0, 1.0, 3.0)의 좌표를 x축으로 0.1, y축으로 0.2, z축으로 0.4 이동

```
glm::vec4 myVec(2.0, 1.0, 3.0, 1.0);  
glm::mat4 transMatrix = glm::mat4 (1.0f);           //--- 단위 행렬로 초기화  
transMatrix = glm::translate (transMatrix, glm::vec3 (0.1, 0.2, 0.4));  
myVec = transMatrix * myVec;
```

- GLSL

- `vec4 transMatrix = myMatrix * myVector`

# 기하 변환

- 신축 (Scaling) 행렬: 한 점을 기준으로 물체에 크기 조절 변환  $\rightarrow (s_x, s_y, s_z)$ : 신축률

$$\begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} * \begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix} = \begin{bmatrix} s_x * x \\ s_y * y \\ s_z * z \\ 1 \end{bmatrix}$$

- GLM 함수

- `glm::mat4 glm::scale (glm::mat4 const &M, glm::vec3 const& Factors);`

- `M`: 신축 행렬
    - `Factors`: 신축률
    - 리턴값: 3 scalar 값 Factors로 생성된 4x4 행렬

- 사용 예) (2.0, 1.0, 3.0)의 좌표를 x축으로 0.5, y축으로 1.5배 신축

```
glm::vec4 myVec(2.0, 1.0, 3.0, 1.0);
```

```
glm::mat4 scaleMatrix = glm::mat4 (1.0f); //--- 단위 행렬로 초기화
```

```
scaleMatrix = glm::scale (scaleMatrix, glm::vec3 (0.5, 1.5, 1.0));
```

```
myVec = scaleMatrix * myVec;
```

- GLSL

- `vec4 scaleMatrix = myMatrix * myVector`

# 기하 변환

- 회전 (Rotation) 행렬: 기준축을 중심으로 객체가 놓여있는 방향을 변환 →  $\theta$ : 회전각

$$\begin{bmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} * \begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix} = \begin{bmatrix} x\cos\theta - y\sin\theta \\ x\sin\theta + y\cos\theta \\ z \\ 1 \end{bmatrix}$$

- GLM 함수

- `glm::mat4 glm::rotate` (`glm::mat4 const &M`, `float Angle`, `glm::vec3 const& Axis`);
  - `M`: 회전 행렬
  - `Angle`: 회전 각도 (라디안 값)
  - `Axis`: 회전 축 (normalize된 값으로 적용)
  - 리턴값: 축 벡터와 각도에서 생성된 4x4 행렬

회전

- 사용 예) (2.0, 1.0, 3.0)의 좌표를 y축에 대해 각도 30도만큼 회전

```
glm::vec4 myVec(2.0, 1.0, 3.0, 1.0);  
glm::mat4 rotMatrix = glm::mat4 (1.0f);  
rotMatrix = glm::rotate (rotMatrix, glm::radians(30.0f), glm::vec3 (0.0f, 1.0f, 0.0f));  
myVec = rotMatrix * myVec;
```

//--- 단위 행렬로 초기화

- GLSL

- `vec4 rotateMatrix = myMatrix * myVector`

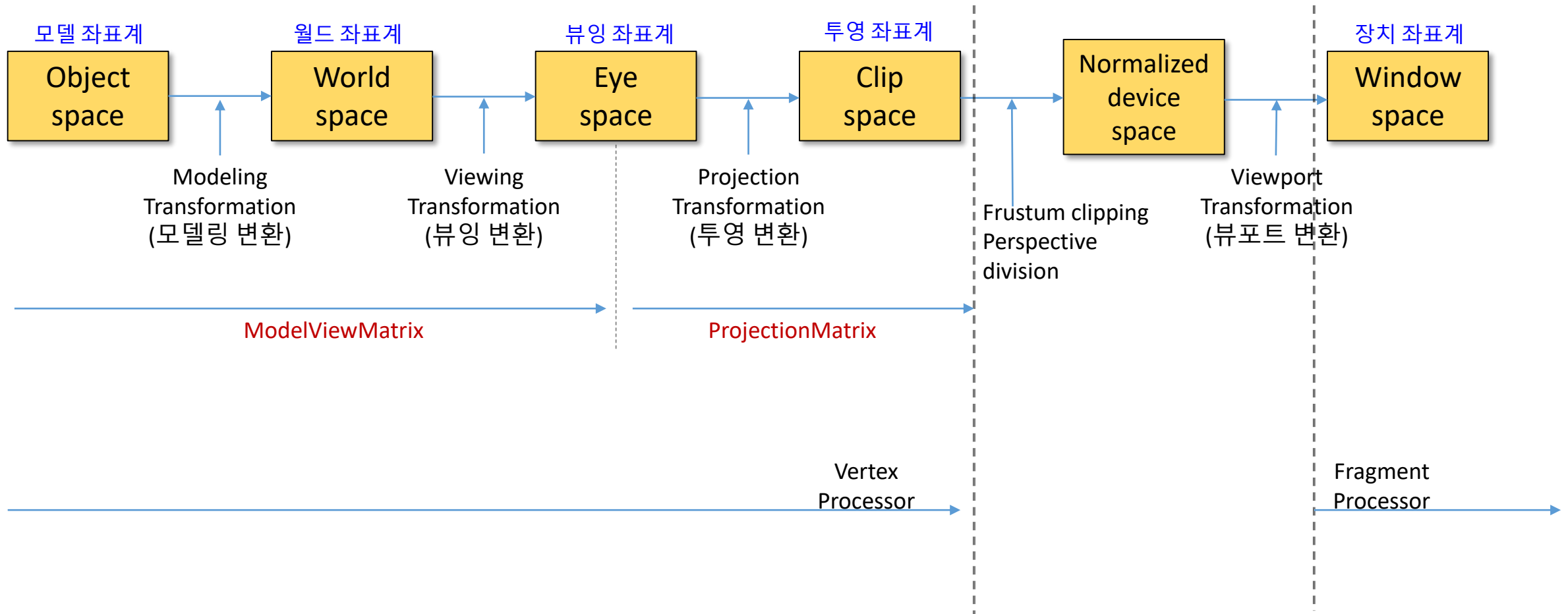
# 함수 프로토타입

- 유니폼 변수 다루기
  - GLint **glGetUniformLocation** (GLuint program, const GLchar \*name);
    - 프로그램에서 uniform 변수의 위치를 가져온다.
      - program: 셰이더 프로그램 이름
      - name: uniform 변수 이름
    - 리턴값: uniform 변수 위치 (-1: 위치를 찾지 못함)
  - void **glUniform{1|2|3|4}{f|i|ui}** (GLuint location, {GLfloat v0, GLfloat v1, GLfloat v2, GLfloat v3});
    - glUniform1f, glUniform2f, glUniform3f, glUniform4f...
    - 현재 프로그램에서 uniform 변수의 값을 명시
      - location: 수정할 uniform 변수의 위치
      - vo, v1, v2, v3: 사용될 uniform 변수 값
  - Void **glUniformMatrix4fv** (GLuint location, GLsizei count, GLboolean transpose, const GLfloat \*value);
    - location: 수정할 uniform 변수의 위치
    - count: 수정할 행렬의 개수 ( 1: 수정할 유니폼 변수가 행렬의 배열이 아니면)
    - transpose: 전치 행렬이 적용될 지 여부 (아니라면 GL\_FALSE)
    - value: 지정된 유니폼 변수를 수정할 때 사용할 배열에 대한 포인터 값

# 3차원 좌표계

- 좌표계 변환

- 물체는 좌표계에 따라 새로운 좌표값으로 바뀌어 최종적으로 화면에 그려진다.



# 좌표계 변환

- 좌표계 변환

- Modeling Transformation (모델링 변환):

- 3차원 공간에서 그래픽스 객체를 이동, 신축, 회전 등의 기하 변환 하는 작업
- 모델링 변환을 적용하는 순서에 따라 결과 값은 달라진다.
- 물체를 뒤로 옮기는 것 = 좌표축을 앞으로 옮기는 것

- Viewing Transformation (뷰잉 변환, 관측 변환):

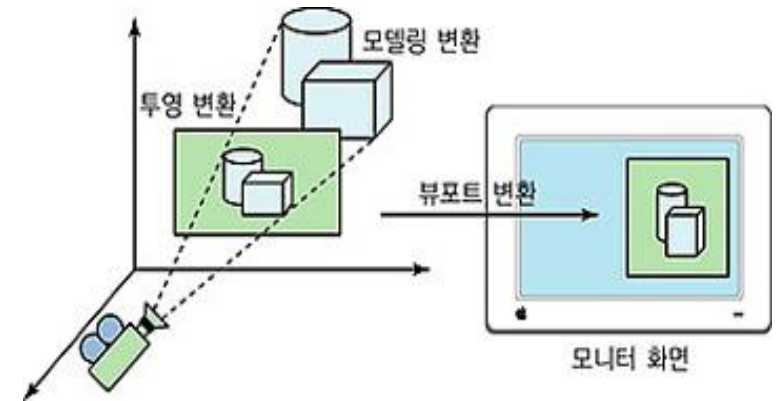
- 관측자의 시점(viewpoint)을 설정하는 변환 (장면을 보는 위치를 결정)
- 카메라의 위치를 잡는 것과 같은 효과를 내는 변환
- 원하는 곳에 원하는 방향으로 관측점을 놓을 수 있다.
- 기본적으로 관측점은 (0, 0, 0)이다. (z축의 음의 방향은 모니터의 안쪽)

- Projection Transformation (투영 변환):

- 3차원 그래픽스 객체를 2차원 평면으로 투영시키는 투영 변환

- Viewport Transformation (뷰포트 변환):

- 투영된 그림이 출력될 위치와 크기를 정의하는 변환
- 윈도우에 나타날 최종 화면의 크기 조절



# 좌표계 변환

- 각 단계에 필요한 변환 행렬들을 생성 → 객체의 좌표값에 적용 → 변환 좌표값

- $V_{clip} = M_{projection} * M_{view} * M_{model} * V_{local}$

- $V_{local}$  : 객체의 좌표값
- $M_{model}$  : 모델링 변환
- $M_{view}$  : 뷰잉 변환
- $M_{projection}$  : 투영 변환
- $V_{clip}$  : 클립 좌표값

- 뷰포트 변환은 glViewport 함수를 사용하여 화면 좌표에 매핑한다.

- 변환 적용

- 변환 함수 (glm 함수) 사용
- 응용 프로그램에서 단계별로 필요한 모든 변환들을 각각 하나의 행렬로 만든다.
  - GLM 함수 사용: translate, rotate, scale 함수
  - 또는 직접 행렬 연산을 통하여 하나의 행렬로 결합한다.
- 결합된 단계별 변환 행렬을 버텍스 셰이더에 적용
  - 버텍스 셰이더에서 좌표값에 변환 행렬을 적용
  - 변환 행렬값을 **uniform** 변수로 선언하여 응용 프로그램에서 셰이더로 값 전송

# 좌표계 변환

- 변환 적용 예) x축으로 1.0, y축으로 1.0 만큼 이동하기

- 응용 프로그램

//--- 변환 행렬 만들기

```
glUseProgram (shaderProgram));
```

```
glm::mat4 transformMatrix (1.0f);
```

```
transformMatrix = glm::translate (transformMatrix, glm::vec3 (1.0f, 1.0f, 0.0f)); //--- 이동
```

//--- 변환 행렬 값을 버텍스 셰이더로 보내기

```
unsigned int transformLocation = glGetUniformLocation (shaderProgram, "transform");
```

```
glUniformMatrix4fv (transformLocation, 1, GL_FALSE, glm::value_ptr (transformMatrix));
```

- 버텍스 셰이더

//--- 변환 행렬 적용하기

```
#version 330 core
```

```
layout (location = 0) in vec3 vPos;
```

//--- 객체의 좌표값

```
uniform mat4 transform;
```

// ---변환 행렬: uniform으로 선언하여 응용 프로그램에서 값을 저장한다.

```
void main()
```

```
{
```

```
    gl_Position = transform *vec4 (vPos, 1.0f);
```

//--- 객체의 좌표에 변환 행렬을 적용한다.

```
}
```

**uniform:** CPU위의 응용프로그램에서 GPU 위의 셰이더로 데이터를 전달하는 한 방법

- 모든 단계의 모든 셰이더에서 접근 가능한 전역 변수
- 필요한 셰이더에서 전역 변수 형태로 선언한 후 사용
- 셰이더가 아니라 응용 프로그램에서 값을 설정할 수 있고, 셰이더에서는 디폴트 값으로 초기화할 수 있다

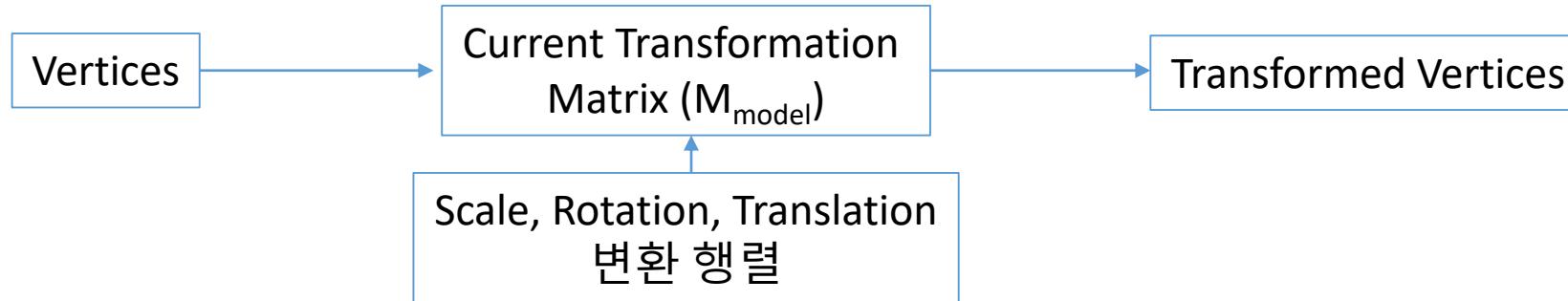


# 1) 모델링 변환

- 모델링 변환: 객체들을 월드 좌표계의 기준으로 배치하는 변환

- $M_{\text{model}}$  행렬을 사용하여 모델링 변환을 적용

- $M_{\text{model}}$  행렬: 객체의 위치와 방향을 월드에 배치하기 위해 이동, 스케일, 회전하는 변환 행렬



- 기하변환을 사용하여 필요한 변환 행렬을 적용하여 합성 변환 행렬을 만든 후, 그 행렬을 객체의 버텍스에 적용한다.

# 1) 모델링 변환

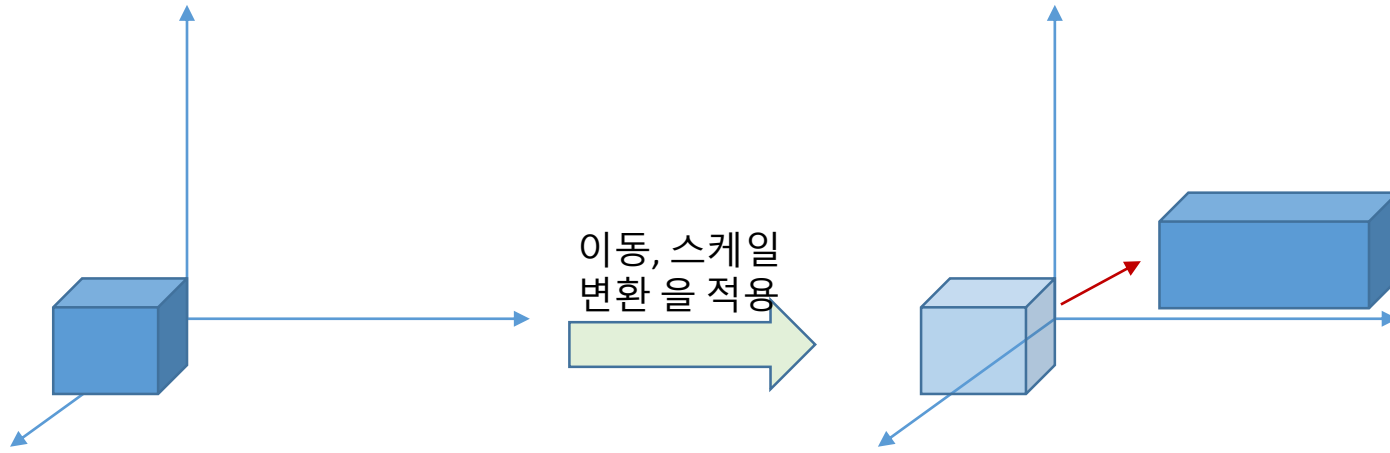
- OpenGL 에서 모델링 변환 행렬들은 적용 순서에 따라 좌표계 또는 좌표 변환이 적용된다.

$$P' = S \cdot R \cdot T \cdot P$$

좌표계 변환

← 물체 (좌표) 변환

- 오픈지엘에서 모델링 변환 행렬은 객체에 설정된 반대 순서로 적용된다.
- 즉, 마지막 변환 (즉, 기하함수 호출 바로 전에 쓰인 것)이 정점 데이터에 적용된다.



# 1) 모델링 변환

- 모델링 변환 적용 예) 사각형을 x축으로 0.1, y축으로 0.5만큼 이동

//--- 응용 프로그램

```
void drawScene ()
```

```
{
```

```
    glUseProgram (shaderProgram);
```

```
    glm::mat4 model = glm::mat4(1.0f);
```

//--- 적용할 모델링 변환 행렬 만들기

```
model = glm::translate (model, glm::vec3(0.1f, 0.5f, 0.0f));
```

//--- model 행렬에 이동 변환 적용

//--- 셰이더 프로그램에서 **modelTransform** 변수 위치 가져오기

```
unsigned int modelLocation = glGetUniformLocation (shaderProgram, "modelTransform");
```

//--- **modelTransform** 변수에 변환 값 적용하기

```
glUniformMatrix4fv (modelLocation, 1, GL_FALSE, glm::value_ptr (model));
```

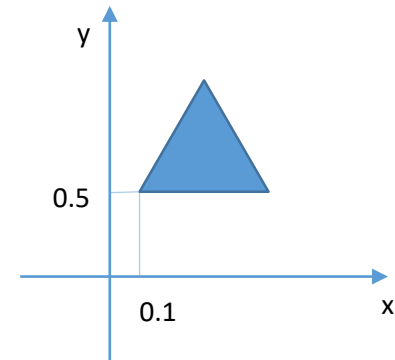
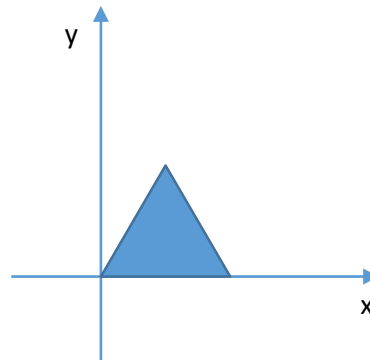
//--- 도형 그리기

```
glBindVertexArray (VAO);
```

```
glDrawArrays (GL_TRIANGLES, 0, 6);
```

```
glutSwapBuffers ();
```

```
}
```



# 1) 모델링 변환

- 모델링 변환 적용 예)

//--- 버텍스 셰이더

#version 330 core

layout (location = 0) in vec3 vPos;

uniform mat4 **modelTransform**;

void main()

{

gl\_Position = **modelTransform** \* vec4(vPos, 1.0);

}

//--- 응용 프로그램에서 받아온 도형 좌표값

//--- 모델링 변환 행렬: uniform 변수로 선언

//--- 좌표값에 modelTransform 변환을 적용한다.

# 1) 모델링 변환

- 모델링 변환 적용 예) 사각형을 x축으로 0.5배, y축으로 2.0배 스케일

//--- 응용 프로그램

```
void drawScene ()
```

```
{
```

```
    glUseProgram (shaderProgram);
```

```
    glm::mat4 model = glm::mat4(1.0f);
```

//--- 적용할 모델링 변환 행렬 만들기

```
model = glm::scale (model, glm::vec3 (0.5, 2.0, 1.0));
```

//--- model 행렬에 스케일 변환 적용

//--- 셰이더 프로그램에서 **modelTransform** 변수 위치 가져오기

```
unsigned int modelLocation = glGetUniformLocation (shaderProgram, "modelTransform");
```

//--- **modelTransform** 변수에 변환 값 적용하기

```
glUniformMatrix4fv (modelLocation, 1, GL_FALSE, glm::value_ptr (model));
```

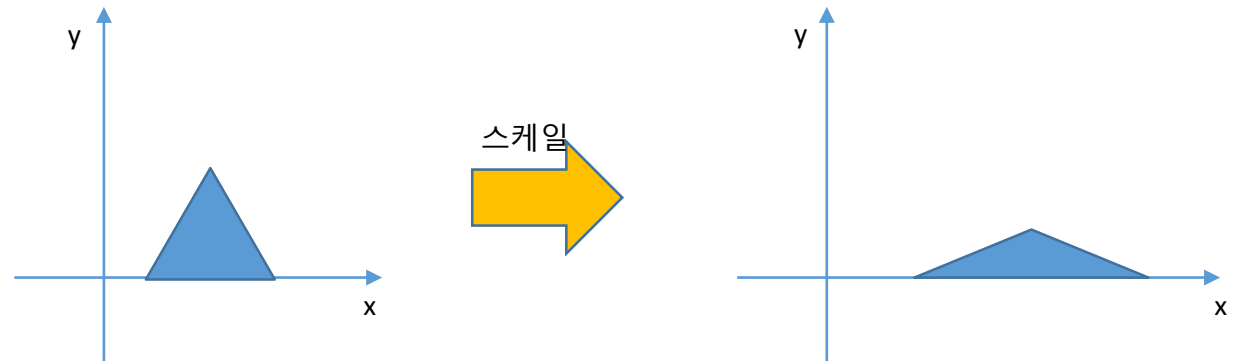
//--- 도형 그리기

```
glBindVertexArray (VAO);
```

```
glDrawArrays (GL_TRIANGLES, 0, 6);
```

```
glutSwapBuffers ();
```

```
}
```



# 1) 모델링 변환

- 모델링 변환 적용 예)

//--- 버텍스 셰이더

#version 330 core

layout (location = 0) in vec3 vPos;

uniform mat4 **modelTransform**;

void main()

{

gl\_Position = **modelTransform** \* vec4(vPos, 1.0);

}

//--- 응용 프로그램에서 받아온 도형 좌표값

//--- 모델링 변환 행렬: uniform 변수로 선언

//--- 좌표값에 modelTransform 변환을 적용한다.

# 1) 모델링 변환

- 1개 이상의 변환을 적용하는 경우 예) z축에 대하여 45도 회전하고, x축으로 0.5 이동 (회전 후 이동)

//--- 응용 프로그램

```
void drawScene ()
```

```
{
```

```
    glUseProgram (shaderProgram);
```

```
    glm::mat4 Tx = glm::mat4 (1.0f);
```

```
    glm::mat4 Rz = glm::mat4 (1.0f);
```

```
    glm::mat4 TR = glm::mat4 (1.0f);
```

```
    Tx = glm::translate (Tx, glm::vec3 (0.5, 0.0, 0.0));
```

```
    Rz = glm::rotate (Rz, glm::radians(45.0f), glm::vec3 (0.0, 0.0, 1.0));
```

```
    TR = Tx * Rz;
```

```
    unsigned int modelLocation = glGetUniformLocation (shaderProgram, "modelTransform"); //--- 버텍스 셰이더에서 모델링 변환 위치 가져오기
```

```
    glUniformMatrix4fv (modelLocation, 1, GL_FALSE, glm::value_ptr (TR)); //--- modelTransform 변수에 변환 값 적용하기
```

```
    glBindVertexArray (VAO);
```

```
    glDrawArrays (GL_TRIANGLES, 0, 3);
```

```
}
```

//--- 이동 행렬 선언

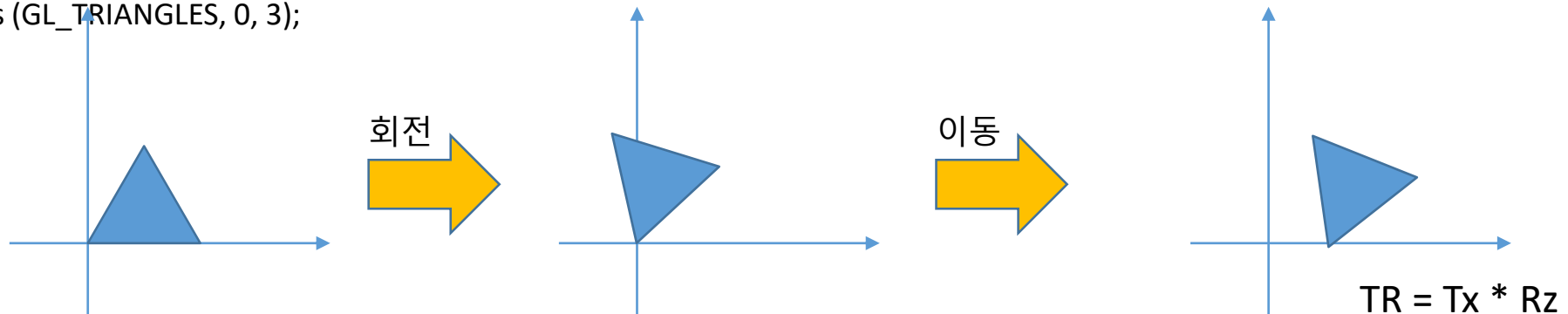
//--- 회전 행렬 선언

//--- 합성 변환 행렬

//--- x축으로 이동 행렬

//--- z축에 대하여 회전 행렬

//--- 합성 변환 행렬: 회전 → 이동



# 1) 모델링 변환

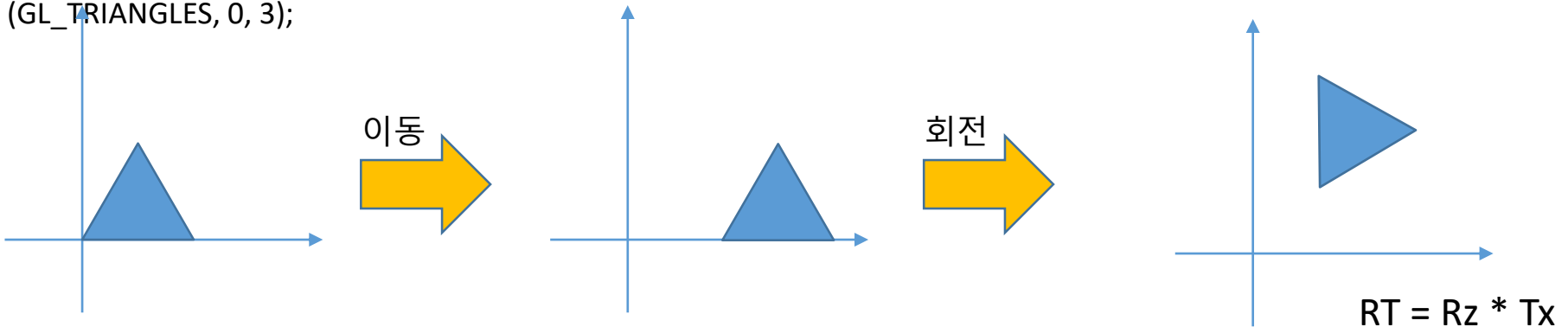
- 변환 적용 시 순서에 따라 다른 결과가 나온다: 이동 후 회전

```
void drawScene ()
{
    glUseProgram (shaderProgram);
    glm::mat4 Rz = glm::mat4 (1.0f);           //--- 회전 행렬 선언
    glm::mat4 Tx = glm::mat4 (1.0f);           //--- 이동 행렬 선언
    glm::mat4 TR = glm::mat4 (1.0f);           //--- 합성 변환 행렬

    Tx = glm::translate (Tx, glm::vec3 (0.5, 0.0, 0.0)); //--- x축으로 이동 행렬
    Rz = glm::rotate (Rz, glm::radians(45.0f), glm::vec3 (0.0, 0.0, 1.0)); //--- z축에 대하여 회전 행렬

    TR = Rz * Tx; //--- 합성 변환 행렬: 이동 → 회전

    unsigned int modelLocation = glGetUniformLocation (shaderProgram, "modelTransform"); //--- 버텍스 셰이더에서 모델링 변환 위치 가져오기
    glUniformMatrix4fv (modelLocation, 1, GL_FALSE, glm::value_ptr (TR)); //--- modelTransform 변수에 변환 값 적용하기
    glBindVertexArray (VAO);
    glDrawArrays (GL_TRIANGLES, 0, 3);
}
```





# 1) 모델링 변환

- 한 개의 행렬에 변환 행렬을 누적해서 만들 수도 있다.

```
void drawScene ()  
{  
    glUseProgram (shaderProgram);  
    glm::mat4 TR = glm::mat4 (1.0f);  
    TR = glm::translate (TR, glm::vec3 (0.5, 0.0, 0.0));  
    TR = glm::rotate (TR, glm::radians(45.0f), glm::vec3 (0.0, 0.0, 1.0));
```

```
//--- 합성 변환 행렬: 변환 행렬을 한 개의 행렬에 누적한다.  
//--- x축으로 이동 행렬  
//--- z축에 대하여 회전 행렬
```

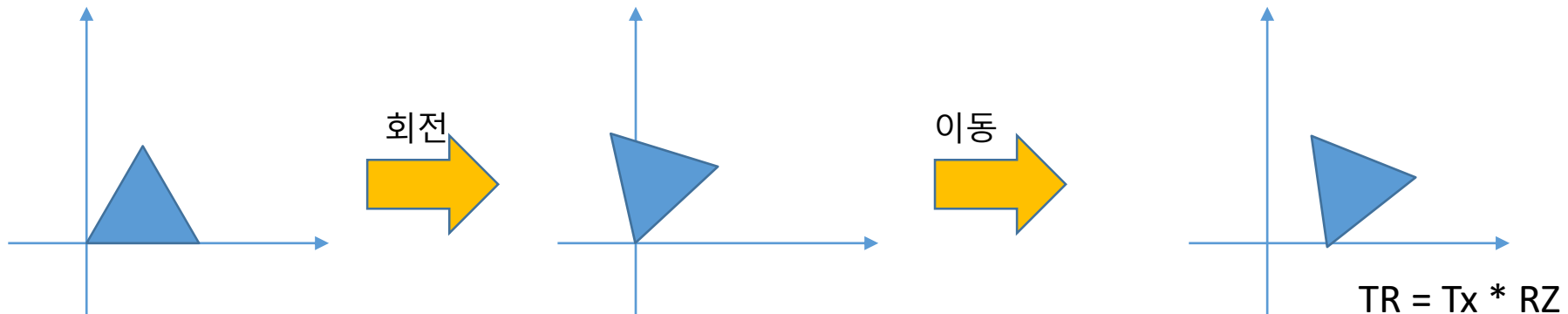
//---위의 결과는 아래의 방식으로 적용해도 같은 변환 결과

```
//--- glm::mat4 Rz = glm::rotate (Rz, glm::radians(45.0f), glm::vec3 (0.0, 0.0, 1.0));  
//--- glm::mat4 Tx = glm::translate (Tx, glm::vec3 (0.5, 0.0, 0.0));  
//--- TR = Tx * Rz ;
```

```
//--- 회전 행렬  
//--- 이동 행렬  
//--- 합성 변환 행렬: 회전 → 이동
```

```
unsigned int modelLocation = glGetUniformLocation (shaderProgram, "modelTransform"); //--- 버텍스 셰이더에서 모델링 변환 위치 가져오기  
glUniformMatrix4fv (modelLocation, 1, GL_FALSE, glm::value_ptr (TR)); //--- modelTransform 변수에 변환 값 적용하기  
glBindVertexArray (VAO);  
glDrawArrays (GL_TRIANGLES, 0, 3);
```

```
}
```



# <OpenGL 상태 관리>

- OpenGL에서의 상태 및 상태 관리
  - 대부분의 상태들은 (라이팅, 텍스처링, 은면 제거, 안개 효과 등) 디폴트로 비활성화(disable)되어 있다.
  - 상태를 활성화(켜거나)하거나 비활성화(끄는)하는 명령어
    - void **glEnable** (GLenum cap);
      - 지정한 기능을 활성화한다.
    - void **glDisable** (GLenum cap);
      - 지정한 기능을 비활성화 한다.
  - 활성화 여부를 체크하는 명령어
    - GLboolean **glIsEnabled** (GLenum cap);
  - cap:
    - GL\_BLEND: 픽셀 블렌딩 연산을 수행 (glBlendFunc)
    - GL\_CULL\_FACE: 앞면 혹은 뒷면을 향하는 폴리곤을 선별 (glCullFace)
    - GL\_DEPTH\_TEST: 깊이를 비교
    - GL\_DITHER: 컬러의 디더링 수행
    - GL\_LINE\_SMOOTH: 선의 안티알리아싱 효과
    - GL\_STENCIL\_TEST: 스텐실 테스트
    - GL\_LINE\_SMOOTH, GL\_POLYGON\_SMOOTH: 선, 면 안티앨리어싱
    - ...

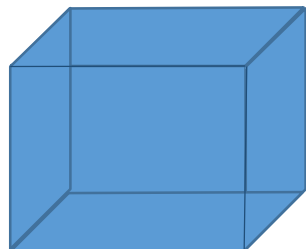
# 상태 관리: 은면 제거

- 은면 제거

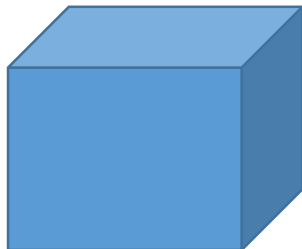
- 3차원 장면을 2차원 평면에 투영시키면 물체들이 중첩될 수 있다.
  - 은면제거를 하지 않으면 → 그려진 순서대로 보인다.
  - 은면제거를 하면 → 관측자의 시점에서 가까운 면이 보이고 가까운 면에 가려진 면은 안보이게 그린다.

- 응용 프로그램에서 깊이 검사 (depth test)를 설정한다.

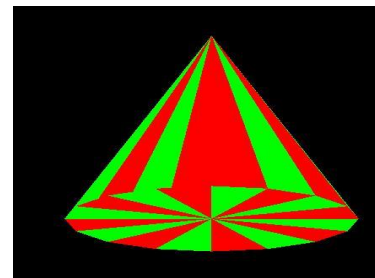
- 윈도우 초기화 시 깊이 검사 모드 설정
  - `glutInitDisplayMode ( GLUT_DOUBLE | GLUT_RGB | GLUT_DEPTH )`
- 깊이 버퍼를 클리어한다
  - `glClear ( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );`
- 깊이 검사를 설정:
  - `glEnable (GL_DEPTH_TEST);`
- 깊이 검사를 해제:
  - `glDisable (GL_DEPTH_TEST);`



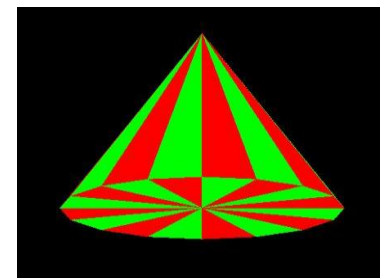
Depth test를 안 했을 때



Depth test를 했을 때



Depth test를 안 했을 때



Depth test를 했을 때

# 상태 관리: 은면 제거

- 사용 예)

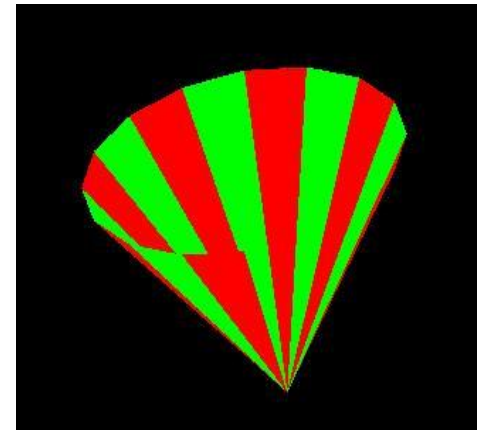
```
void main (int argc, char **argv)
{
    glutInit(&argc, argv);
    glutInitDisplayMode (GLUT_DOUBLE | GLUT_RGBA | GLUT_DEPTH);    //--- 깊이 검사 모드도 같이 설정
    ...

    glEnable (GL_DEPTH_TEST);    //--- 상태 설정은 필요한 곳에서 하면 된다.
}

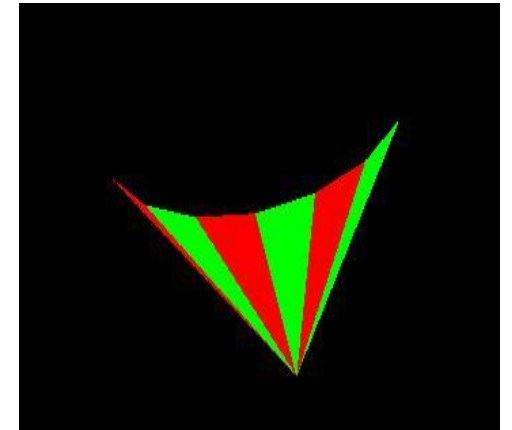
void drawScene ()
{
    glClearColor(0.0, 0.0, 0.0, 1.0f);
    glClear (GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);    //--- 깊이 버퍼를 클리어한다.
    ...
}
```

# 상태 관리: 은면 제거

- 컬링 (culling)
  - 후면을 선별(backface culling)하여 **뒷면을 모두 제거**할 수 있다.
    - Winding (꼭짓점이 진행되는 방향)을 이용하여 폴리곤의 앞면과 뒷면을 구분한다.
    - 시계 반대방향으로 windin되는 폴리곤이 앞면이다.
      - 컬링 설정: glEnable (**GL\_CULL\_FACE**);
      - 컬링 해제: glDisable (**GL\_CULL\_FACE**);
  - void **glFrontFace** (GLenum mode);
    - 폴리곤의 어느 면이 앞면 또는 뒷면인지 정의한다.
    - 장면이 닫힌 객체로 구성되어 있을 때 그 객체의 내부 연산은 불필요한데, 폴리곤의 어느 면이 앞면인지를 결정할 수 있다.
    - GLenum mode: GL\_CW – 시계방향, GL\_CCW – 반시계 방향
    - glFrontFace (GL\_CW): 시계 방향을 앞면으로
    - glFrontFace (GL\_CCW): 반시계 방향을 앞면으로
- 사용 예)
  - 응용 프로그램의 필요한 부분에 아래 함수를 호출한다.
    - glEnable (GL\_CULL\_FACE);



Culling을 안 했을 때



Culling을 했을 때

## <3차원 객체 그리기>

- GLU 라이브러리를 이용하여 모델링 하기
  - 2차 곡선 (Quadrics)을 이용한다.
    - `GLUquadricObj * gluNewQuadric ();`
      - Quadric Object 를 생성
    - `void gluQuadricDrawStyle (GLUquadric *quadObject, GLenum drawStyle);`
      - 도형의 스타일 지정하기: 솔리드 스타일 / 와이어 프레임 / 선으로 외부 모서리만 / 점
      - drawStyle: GLU\_FILL / GLU\_LINE / GLU\_SILHOUETTE / GLU\_POINT
    - `void gluQuadricNormals (GLUquadric *quadObject, GLenum normals);`
      - 법선 벡터 제어, 빛에 대한 영향 결정: 빛의 효과 없음 / 면이 각인것처럼 보임 / 면이 부드럽게 보임
      - Normals: GLU\_NONE / GLU\_FLAT / GLU\_SMOOTH
    - `void gluquadricOrientation (GLUquadric *quadObject, GLenum orientation);`
      - 법선 벡터의 방향 지정: 물체에 대해서 법선 벡터를 바깥쪽으로 지정 / 안쪽으로 지정
      - Orientation: GLU\_OUTSIDE / GLU\_INSIDE
    - `void gluDeleteQuadric (GLUquadric *quadObject);`
      - 객체 삭제하기

# 3차원 객체 그리기: GLU 모델

- GLU 라이브러리를 이용하여 모델링 하기
  - 구 생성하기
    - void gluSphere (GLUquadric \*qobj, GLdouble radius, GLint slices, GLint stacks);

```
void gluSphere ( GLUquadric *qobj, GLdouble radius, GLint slices, GLint stacks );
```

Parameters	<i>qobj</i>	// gluNewQuadric으로 생성된 Quadric Object
	<i>radius</i>	// Sphere의 반지름(Radius)
Help	<i>slices</i>	// Z축을 중심으로 하는 Subdivisions의 개수(경도(Longitude)와 유사)
	<i>stacks</i>	// Z축을 따르는 Subdivisions의 개수(위도(Latitude)와 유사)



(A) gluSphere (oZbj, 1.0, 5, 5);



(B) gluSphere (obj, 1.0, 10, 10);



(C) gluSphere (obj, 1.0, 20, 20);

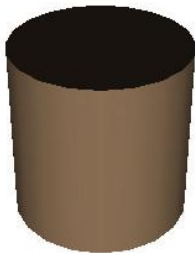
# 3차원 객체 그리기: GLU 모델

- 실린더 생성하기

- `void gluCylinder (GLUquadric *qobj, GLdouble baseRadius, GLdouble topRadius, GLdouble height, GLint slices, GLint stack);`

```
void gluCylinder ( GLUquadric *qobj, GLdouble baseRadius, GLdouble topRadius,  
                  GLdouble height, GLint slices, GLint stacks );
```

Parameters Help	<i>qobj</i>	// gluNewQuadric로 생성되어진 Quadric Object
	<i>baseRadius</i>	// z=0에 있는 Cylinder의 반지름(Radius)
	<i>topRadius</i>	// z=height에 있는 Cylinder의 반지름(Radius)
	<i>height</i>	// Cylinder의 높이(Height)
	<i>slices</i>	// Z축을 중심으로 하는 회전 Subdivisions의 개수
	<i>stacks</i>	// Z축을 따르는 Subdivisions의 개수



(A)

(A) `gluCylinder (obj, 1.0, 1.0, 2.0, 20, 8);`



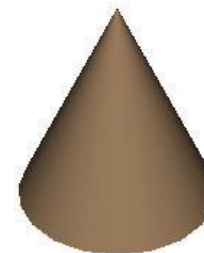
(B)

(C) `gluCylinder (obj, 1.0, 0.3, 2.0, 20, 8);`



(C)

(B) `gluCylinder (obj, 1.0, 1.0, 2.0, 8, 8);`



(D)

(D) `gluCylinder (obj, 1.0, 0.0, 2.0, 20, 8);`



# 3차원 객체 그리기: GLU 모델

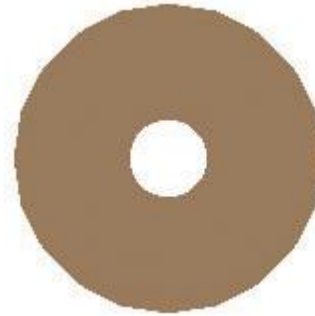
- 디스크 생성하기
  - void gluDisk (GLUquadric \*qobj, GLdouble innerRadius, GLdouble outerRadius, GLint slices, GLint loops);

```
void gluDisk ( GLUquadric *qobj, GLdouble innerRadius, GLdouble outerRadius, GLint slices,  
              GLint loops );
```

Parameters Help	<i>qobj</i>	// gluNewQuadric으로 생성된 Quadric Object
	<i>innerRadius</i>	// Disk의 안쪽 반지름(Radius)
	<i>outerRadius</i>	// Disk의 바깥쪽 반지름(Radius)
	<i>slices</i>	// Z축을 중심으로 하는 Subdivisions의 개수
	<i>loops</i>	// Disk가 세분화되는 동심원의 개수



(A) gluDisk(obj, 0.0, 2.0, 20, 3);



(B) gluDisk(obj, 0.5, 2.0, 20, 3);

# 3차원 객체 그리기: GLU 모델

- 모델 생성 예)

```
GLUquadricObj *qobj;
```

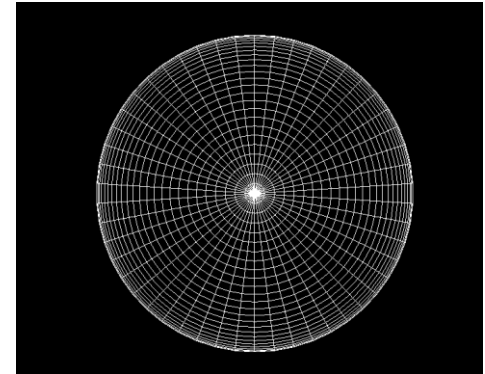
```
void drawScene ( ) {  
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
```

```
    qobj = gluNewQuadric ();  
    gluQuadricDrawStyle( qobj, GLU_LINE );  
    gluQuadricNormals( qobj, GLU_SMOOTH );  
    gluQuadricOrientation( qobj, GLU_OUTSIDE );  
    gluSphere( qobj, 1.5, 50, 50 );
```

```
    glutSwapBuffers();
```

```
}
```

```
// 객체 생성하기  
// 도형 스타일  
//→ 생략 가능  
//→ 생략 가능  
// 객체 만들기
```



# .obj 파일 다루기

- obj 파일
  - 3차원 그래픽 이미지가 저장된 파일의 형태 중 하나로 "Wavefront Technologies"의 고유 파일 포맷
  - 가장 오래되고 기본적으로 사용되는 3차원 모델 표현 파일
    - 오브젝트의 좌표를 기록하고 각 버텍스의 정보를 포함
    - 코드 내에서 직접 입력했던 좌표 및 정보를 가지고 있는 하나의 파일
    - 확장자가 .obj
- 파일 종류
  - obj 파일: 폴리곤을 구성하는 정보 저장
    - 각 버텍스의 좌표값
    - 텍스처 좌표값의 UV 값
    - 각 버텍스의 노말값
    - 폴리곤의 면을 구성하는 꼭지점과 텍스처 좌표값의 리스트
  - mtl (Material Template Library)파일: 재질과 텍스처에 관한 정보
- 파일 만들기
  - 맥스나 3차원 모델링 제작 프로그램을 통해 만든다.
- 파일 읽기
  - 전체 버텍스 개수 및 삼각형 개수 세기
  - 해당 개수만큼 메모리 할당
  - 할당된 메모리에 각 버텍스 및 면 정보 입력

# .obj 파일 다루기

- 파일 포맷 (#으로 시작하는 줄은 주석)

# t

# **v**: List of geometric vertices, with (x, y, z [,w]) coordinates, w is optional and defaults to 1.0.

v 0.123 0.234 0.345 1.0

v ... ..

# **vt**: List of texture coordinates, in (u, [v ,w]) coordinates, these will vary between 0 and 1, v and w are optional and default to 0.

vt 0.500 1 [0]

vt ... ..

# **vn**: List of vertex normals in (x,y,z) form; normals might not be unit vectors.

vn 0.707 0.000 0.707

vn ... ..

# **f**: Polygonal face element (vertex/texture coordinate/vertex normal) (texture coord.가 생략되면 두 개의 슬래쉬 사용 (/))

f 1 2 3

f 3/1 4/2 5/3

f 6/4/1 3/5/3 7/6/5

f 7//1 8//2 9//3

f ... ..

**g**: group name

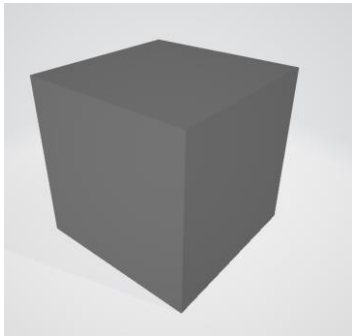
g cube

# .obj 샘플 파일

- 육면체  
# cube.obj  
#

g cube

```
v 0.0 0.0 0.0  
v 0.0 0.0 1.0  
v 0.0 1.0 0.0  
v 0.0 1.0 1.0  
v 1.0 0.0 0.0  
v 1.0 0.0 1.0  
v 1.0 1.0 0.0  
v 1.0 1.0 1.0
```



```
vn 0.0 0.0 1.0  
vn 0.0 0.0 -1.0  
vn 0.0 1.0 0.0  
vn 0.0 -1.0 0.0  
vn 1.0 0.0 0.0  
vn -1.0 0.0 0.0
```

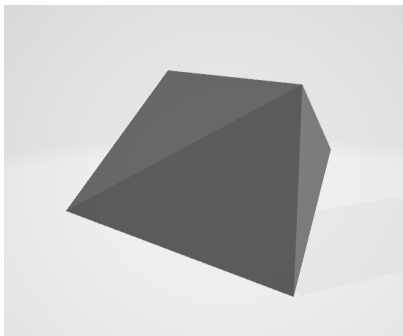
```
f 1//2 7//2 5//2  
f 1//2 3//2 7//2  
f 1//6 4//6 3//6  
f 1//6 2//6 4//6  
f 3//3 8//3 7//3  
f 3//3 4//3 8//3  
f 5//5 7//5 8//5  
f 5//5 8//5 6//5  
f 1//4 5//4 6//4  
f 1//4 6//4 2//4  
f 2//1 6//1 8//1  
f 2//1 8//1 4//1
```

- 다이아몬드  
# diamond.obj

g Object001

```
v 0.000000E+00 0.000000E+00 78.0000  
v 45.0000 45.0000 0.000000E+00  
v 45.0000 -45.0000 0.000000E+00  
v -45.0000 -45.0000 0.000000E+00  
v -45.0000 45.0000 0.000000E+00  
v 0.000000E+00 0.000000E+00 -78.0000
```

```
f 1 2 3  
f 1 3 4  
f 1 4 5  
f 1 5 2  
f 6 5 4  
f 6 4 3  
f 6 3 2  
f 6 2 1  
f 6 1 5
```



- 주전자  
# OBJ file created by ply\_to\_obj.c  
#

g Object001

```
v 40.6266 28.3457 -1.10804  
v 40.0714 30.4443 -1.10804  
v 40.7155 31.1438 -1.10804  
v 42.0257 30.4443 -1.10804  
v 43.4692 28.3457 -1.10804  
v 37.5425 28.3457 14.5117  
v 37.0303 30.4443 14.2938
```

```
...  
vn -0.966742 -0.255752 9.97231e-09  
vn -0.966824 0.255443 3.11149e-08  
vn -0.092052 0.995754 4.45989e-08  
vn 0.68205 0.731305 0  
vn 0.870301 0.492521 -4.87195e-09  
vn -0.893014 -0.256345 -0.369882  
vn -0.893437 0.255997 -0.369102
```

```
...  
f 7 6 1  
f 1 2 7  
f 8 7 2  
f 2 3 8  
f 9 8 3  
f 3 4 9  
f 10 9 4  
f 4 5 10  
f 12 11 6  
f 6 7 12  
f 13 12 7  
...
```



# .obj 파일 읽기 샘플 (버전에 따라 수정 필요)

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
```

```
typedef struct {
    float x, y, z;
} Vertex;
```

```
typedef struct {
    unsigned int v1, v2, v3;
} Face;
```

```
typedef struct {
    Vertex* vertices;
    size_t vertex_count;
    Face* faces;
    size_t face_count;
} Model;
```

```
void read_newline(char* str) {
    char* pos;
    if ((pos = strchr(str, '\n')) != NULL)
        *pos = '\0';
}
```

```
void read_obj_file(const char* filename, Model* model) {
    FILE* file;
    fopen_s(&file, filename, "r");
    if (!file) {
        perror("Error opening file");
        exit(EXIT_FAILURE);
    }
}
```

```
char line[MAX_LINE_LENGTH];
model->vertex_count = 0;
model->face_count = 0;
```

```
while (fgets(line, sizeof(line), file)) {
    read_newline(line);
```

```
    if (line[0] == 'v' && line[1] == ' ')
        model->vertex_count++;
    else if (line[0] == 'f' && line[1] == ' ')
        model->face_count++;
}
```

```
fseek(file, 0, SEEK_SET);
```

```
model->vertices = (Vertex*)malloc(model->vertex_count * sizeof(Vertex));
model->faces = (Face*)malloc(model->face_count * sizeof(Face));
```

```
size_t vertex_index = 0; size_t face_index = 0;
```

```
while (fgets(line, sizeof(line), file)) {
    read_newline(line);
```

```
    if (line[0] == 'v' && line[1] == ' ') {
        int result = sscanf_s(line + 2, "%f %f %f", &model->vertices[vertex_index].x,
            &model->vertices[vertex_index].y,
            &model->vertices[vertex_index].z);
        vertex_index++;
    }
```

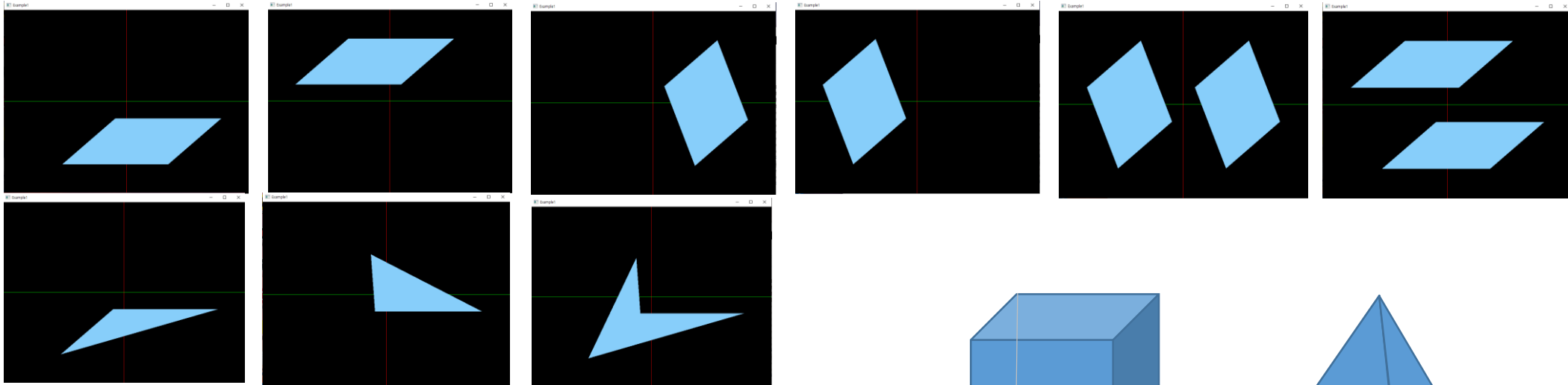
```
    else if (line[0] == 'f' && line[1] == ' ') {
        unsigned int v1, v2, v3;
        int result = sscanf_s(line + 2, "%u %u %u", &v1, &v2, &v3);
        model->faces[face_index].v1 = v1 - 1; // OBJ indices start at 1
        model->faces[face_index].v2 = v2 - 1;
        model->faces[face_index].v3 = v3 - 1;
        face_index++;
    }
```

```
    }
}
fclose(file);
```

```
}
```

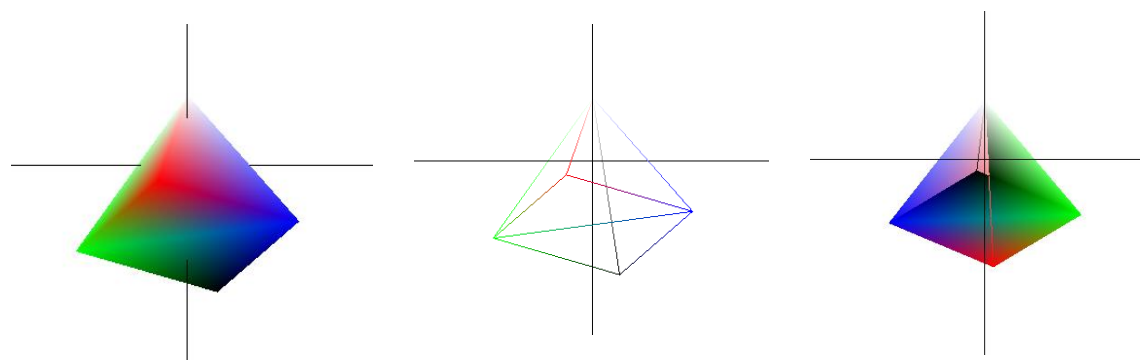
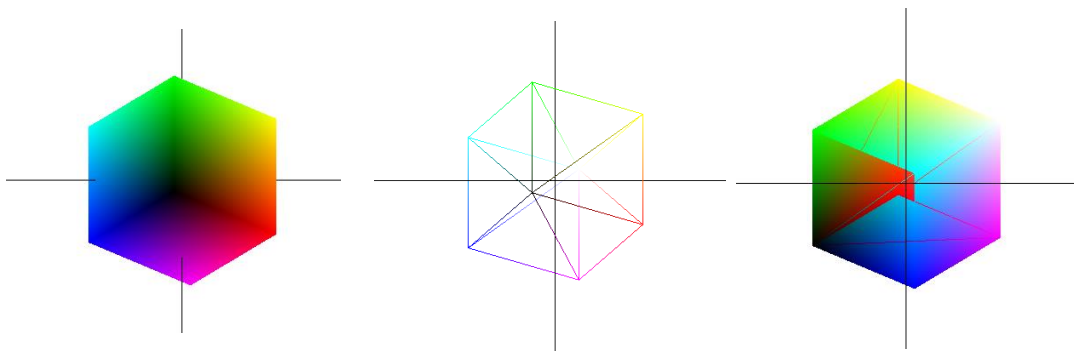
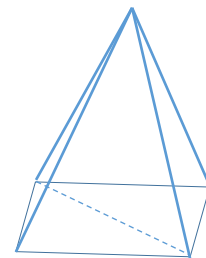
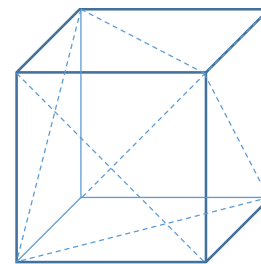
# 실습 13

- 3차원 객체, 육면체 또는 사면체의 면을 명령어에 따라 그리기
  - 화면 중앙에 화면 좌표계 (x축, y축)을 그린다.
  - 모든 객체들은 x축으로 10도, y축으로 10도 회전해서 그린다. (3차원 도형이라 약간 기울어지게 그린다)
  - 다음의 키보드 명령어에 따라 도형을 구성하는 각 면을 그린다. 각 면마다 색상을 정해 해당 색상으로 그린다.
    - 1/2/3/4/5/6: 육면체의 각 면을 그린다.
    - 7/8/9/0 : 사면체의 각 면을 그린다.
    - c: 육면체에서 랜덤한 2개의 면을 그린다.
    - t: 사면체에서 랜덤한 2개의 면을 그린다.
    - 좌표계는 [-1.0, 1.0]이고, 크기는 원하는 크기로 설정



# 실습 14

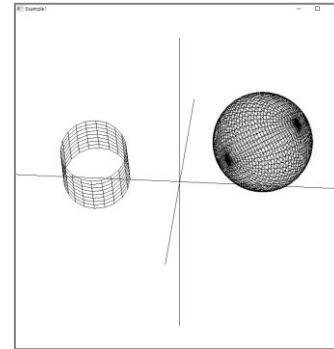
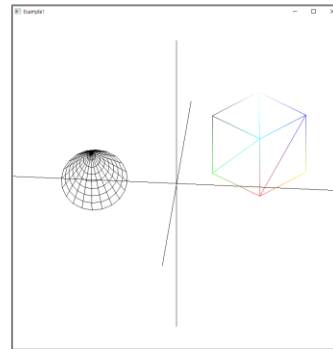
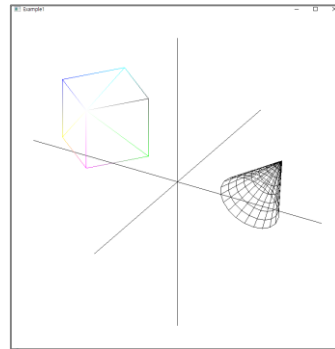
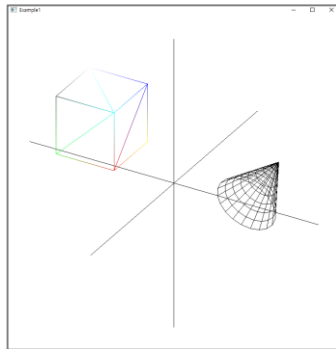
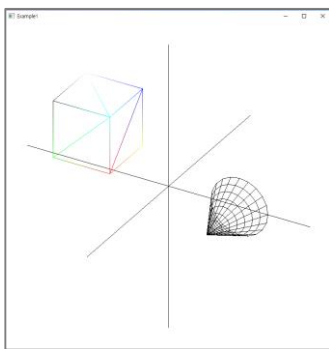
- 꼭지점을 이용하여 3차원 객체 그리기
  - 화면 중앙에 화면 좌표계 (x축, y축)을 그린다. 좌표계는 이동하지 않고 고정되어 있다.
  - 키보드 명령에 따라 육면체 혹은 사각뿔을 그린다. 각 꼭지점에 다른 색상 설정한다.
    - 객체는 x축으로 30도, y축으로 -30도 회전해 있다. (회전 각도는 변경 가능)
  - 키보드 명령
    - c: 육면체 ([-1.0, 1.0] 사이의 값으로 꼭지점 설정하여 그리기)
    - p: 사각 뿔 (피라미드 모양으로 바닥은 사각형, 옆면은 삼각형)
    - h: 은면제거 적용/해제
    - w/W: 와이어 객체/솔리드 객체
    - x/X: x축을 기준으로 양/음 방향으로 회전 애니메이션 (자전)
    - y/Y: y축을 기준으로 양/음 방향으로 회전 애니메이션 (자전)
    - ←/→/↑/↓: 좌/우/상/하로 객체를 이동한다. (x축, y축으로 이동)
    - s: 초기 위치로 리셋 (자전 애니메이션도 멈추기)





# 실습 15

- 모델링 변환하기
  - 화면의 중앙에  $x, y, z$ 축을 그린다. 좌표계와 3차원 객체들은 모두  $x, y$ 축에 대해 같은 각도로 기울어져 있다.
  - 화면의 좌, 우에 3차원 객체를 그리고, 키보드 명령으로 객체 변환 하기
    - 우: 육면체
    - 좌: 원뿔 (또는, 구)
  - 키보드 명령
    - $x/X/y/Y$ : 객체의  $x/y$ 축에 대하여 각각 양/음 방향으로 회전하기 (두 객체가 모두 자전)
      - 1/2/3: 1을 누르면 좌측의 객체만, 2를 누르면 우측의 객체만, 3을 누르면 모두 자전한다.
    - $r/R$ : 두 객체를 화면 중앙의  $y$ 축에 대하여 양/음 방향으로 회전하기 (공전)
    - $c$ : 두 도형을 다른 도형으로 바꾼다.
    - $s$ : 초기화 하기



# 이번 주에는

- 변환 적용하기
  - GLM 라이브러리
  - 모델링 변환
  - 상태 설정: 은면제거, 컬링
- 다음 시간에는
  - 뷰잉변환, 투영변환, 뷰포트 변환