openGL 좌표계 변환

GLM 라이브러리 사용하기
좌표계 변환

GLM 라이브러리 사용하기

- GLM (GL Mathematics)
  - GLM (GL Mathematics)는 OpenGL Shading Language를 기반으로 하는 그래픽스 소프트웨어에서 사용할 수 있는 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 사용하지말고 “glm::” 문법을 사용하여 함수들을 호출하도록 한다.
    - 참조 사이트: https://glm.g-truc.net/0.9.9/index.html

  - GLM 라이브러리 사용하기
    - 헤더파일 포함하기

```
#include <glm/glm.hpp>
#include <glm/ext.hpp>
#include <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()
  - glm::vec4()
  - glm::vec3()

- 초기화
  - glm ::mat4 (1.0); // GLM 0.9.9 버전부터는 초기화된 기본 행렬이 단위 행렬이 아니라 0으로 초기화
    // 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 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

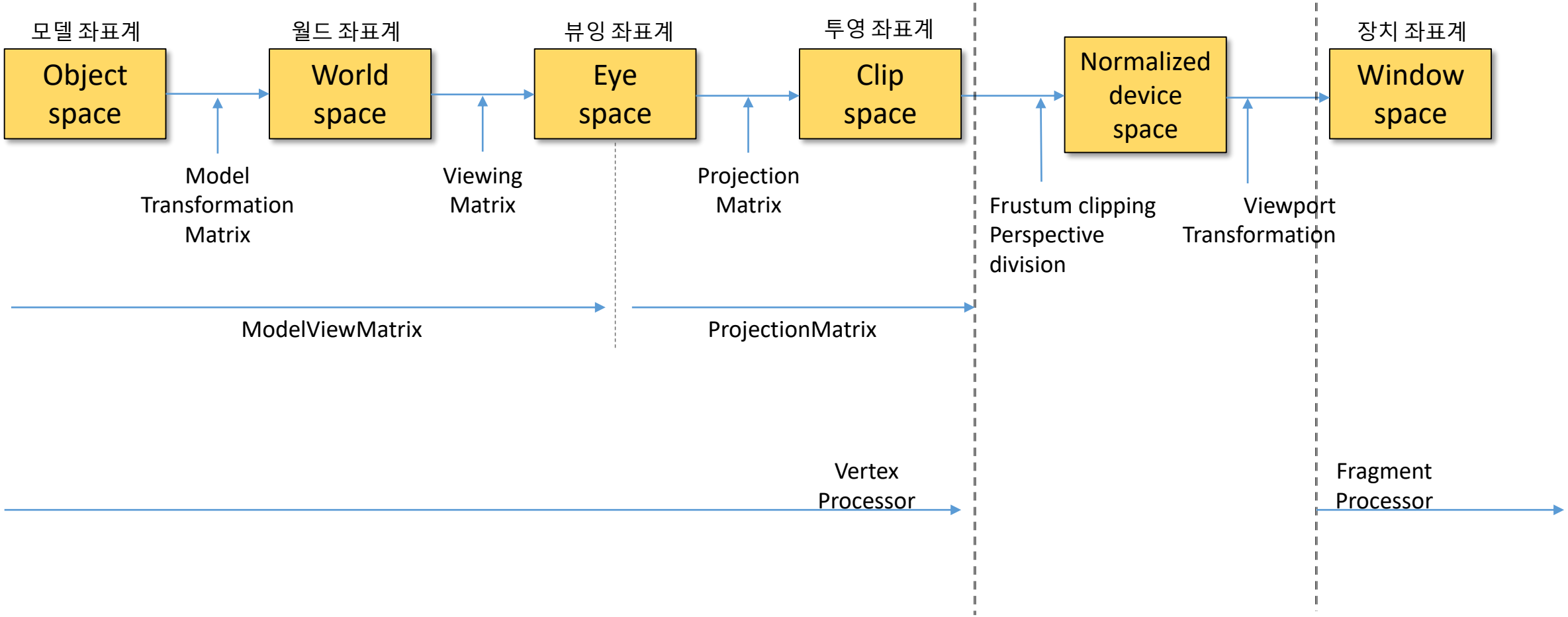
- 삼각 함수
  - vec4f glm::sin(vec4f const &angle);
  - vec4f glm::cos(vec4f const &angle);
  - vec4f glm::tan(vec4f const &angle);
  - vec4f glm::degrees (vec4f const &radians);
  - vec4f glm::radians (vec4f const &degrees);

- 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 ) );
}
```

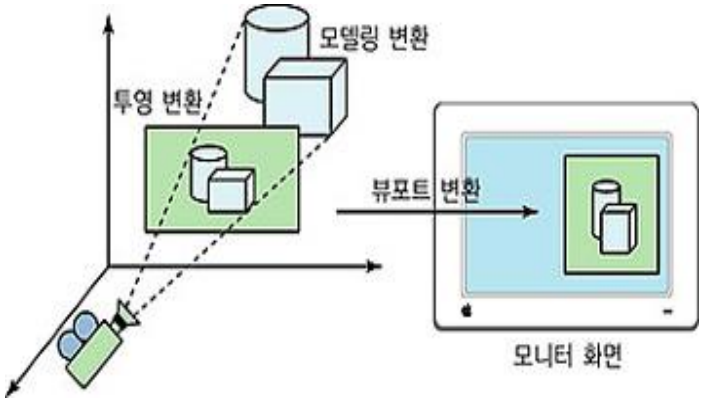
3차원 좌표계

- 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, 회전시켜주는 변환
    - 모델링 변환을 적용하는 순서에 따라 결과 값은 달라진다.
    - 물체를 뒤로 옮기는 것 = 좌표축을 앞으로 옮기는 것
  - Viewing Transformation (관측 변환, 뷰잉 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



변환 행렬

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

$$\begin{bmatrix} a & b & c & d \\ e & f & g & h \\ i & j & k & l \\ m & n & o & p \end{bmatrix} * \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} ax + by + cz + dw \\ ex + fy + gz + hw \\ ix + jy + kz + lw \\ m + ny + oz + pw \end{bmatrix}$$

  - 응용 프로그램에서 GLM 라이브러리 사용하면

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

  - GLSL에서

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

변환 행렬

- 이동 (Translation) 행렬: (dx, dy, dz) – 이동 벡터 값

$$\begin{bmatrix} 1 & 0 & 0 & dx \\ 0 & 1 & 0 & dy \\ 0 & 0 & 1 & dz \\ 0 & 0 & 0 & 1 \end{bmatrix} * \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} dx + x \\ dy + y \\ dz + z \\ 1 \end{bmatrix}$$

  - GLM 함수
    - glm::mat4 glm::**translate** (glm::mat4 const &M, glm::vec3 const & Translation);
      - M: 이동 행렬
      - Translation: 이동 변환 인자

    - 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

변환 행렬

- 신축 (Scaling) 행렬: (sx, sy, sz) – 신축률

$$\begin{bmatrix} sx & 0 & 0 & 0 \\ 0 & sy & 0 & 0 \\ 0 & 0 & sz & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} * \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} sx * x \\ sy * y \\ sz * z \\ 1 \end{bmatrix}$$

  - GLM 함수
    - glm::mat4 glm::**scale** (glm::mat4 const &M, glm::vec3 const& Factors);
      - M: 신축 행렬
      - Factors: 신축률

    - 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

변환 행렬

- 회전 (Rotation) 행렬: θ – 회전각

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

  - 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)

    - 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);                    // 단위 행렬로 초기화
rotMatrix = glm::rotate (rotMatrix, glm::radians(angle), glm::vec3 (0.0f, 1.0f, 0.0f));
myVec = rotMatrix * myVec;
```

  - GLSL
    - vec4 rotateMatrix = myMatrix * myVector

변환 행렬

- 누적 변환
  - TransformationVector = TranslateMatrix * RotationMatrix * ScaleMatrix * OriginalVector;
  - 변환 순서에 따라 다른 결과값이 나온다.
  - GLM

    ```
    glm::mat4 myModelMatrix = myTranslationMatrix * myRotationMatrix * myScaleMatrix;
    glm::vec4 myTransformeVector = myModelMatrix * myOriginalVector;
    ```

  - GLSL

    ```
    mat4 transform = mat2 * mat1;
    vec4 myTransformVector = transform * myVector
    ```

좌표계 변환

- 각 단계에 필요한 변환 행렬들을 생성하여 합성 변환 행렬을 만든다.
  - **$V_{clip} = M_{projection} * M_{view} * M_{model} * V_{local}$**

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

  - 뷰포트 변환은 glViewport 함수를 사용하여 화면 좌표에 매핑한다.

- 변환 적용
  - 변환 행렬 사용
  - 응용 프로그램에서 모든 변환을 하나의 행렬에 결합한다
    - GLM 함수 사용: translate, rotate, scale 함수
    - 또는 직접 행렬 연산을 통하여 하나의 행렬로 결합한다.
  - 결합된 변환 행렬을 버텍스 세이더에 적용
    - 버텍스 세이더의 좌표값에 변환 행렬을 적용
    - 변환 행렬값을 uniform 변수로 선언하여 응용 프로그램에서 값 저장

좌표계 변환

- 변환 적용 예)
  - (1, 1, 0) 만큼 이동 후 ½ 크기로 축소하기
  - 응용 프로그램

```
//--- 변환 행렬 만들기
glUseProgram (shaderProgram]);
glm::vec4 v (1.0, 0.0, 0.0, 1.0f);
glm:: mat4 transformMatrix (1.0f);
transformMatrix = glm::translate (transformMatrix, glm::vec3 (1.0f, 1.0f, 0.0f));          //--- 이동
transformMatrix = glm::scale (transformMatrix, glm::vec3 (0.5, 0.5, 0.5));                 //--- 신축

//--- 변환 행렬 값을 버텍스 세이더로 보내기
unsigned int transformLocation = glGetUniformLocation (shaderProgram, "transform");
glUniformMatrix4fv (transformLocation, 1, GL_FALSE, glm::value_ptr (transformMatrix));
```

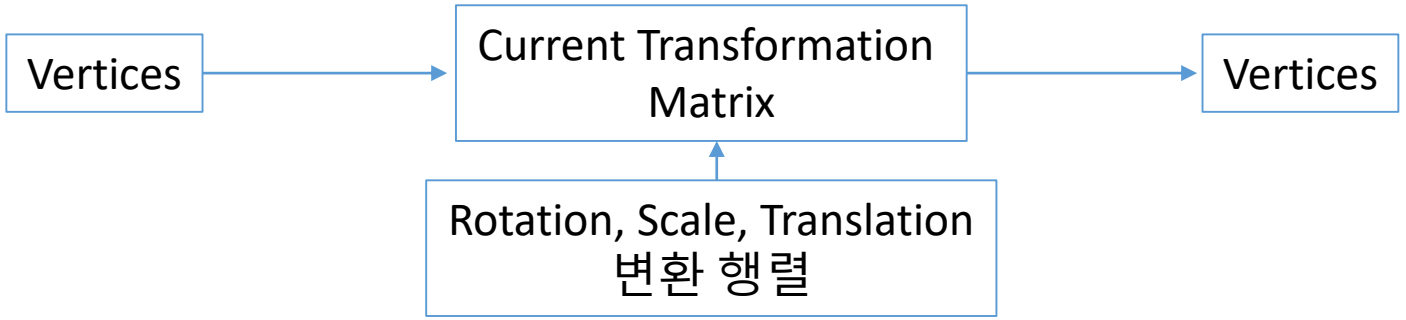
  - 버텍스 세이더
    - 변환 행렬 적용하기

```
#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 세이더로 데이터를 전달하는 한 방법
- 모든 단계의 모든 세이더에서 접근 가능한 전역 변수
- 필요한 세이더에서 전역 변수 형태로 선언한 후 사용
- 세이더가 아니라 응용 프로그램에서 값을 설정할 수 있고, 세이더에서는 디폴트 값으로 초기화할 수 있다

모델링 변환

- 객체들을 월드 좌표계의 기준으로 배치하기
  - **M_{model}** 행렬을 사용하여 모델링 변환을 적용
    - M_{model} 행렬: 객체의 위치와 방향을 월드에 배치하기 위해 이동, 스케일, 회전하는 변환 행렬



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

- openGL 에서 모델링 변환 행렬들은 객체에 설정된 반대 순서로 적용된다.
  - 마지막 변환이 정점 데이터에 먼저 적용된다.

모델링 변환

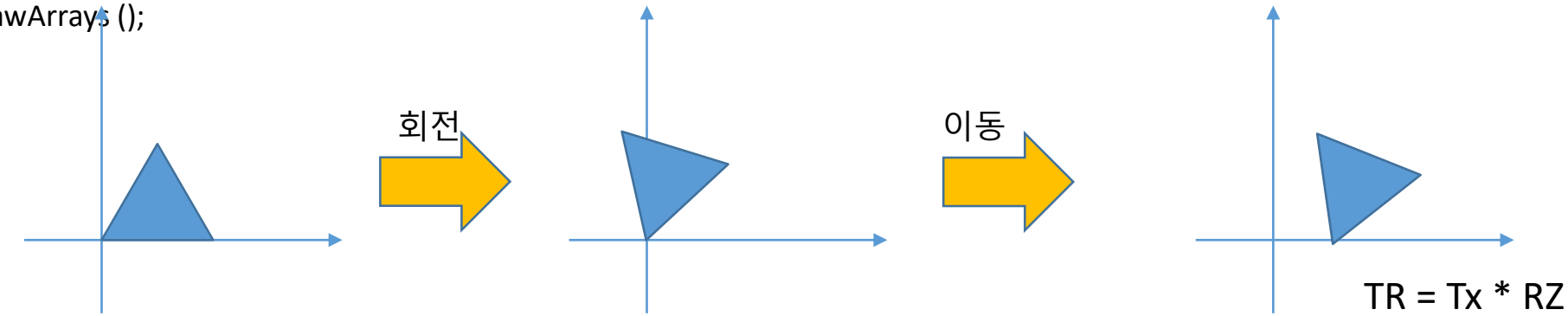
- 모델링 변환 적용 예)
  - 사각형을 x축으로 0.1, y축으로 0.5만큼 이동

```
//--- 응용 프로그램
void drawScene ()
{
        glUseProgram (shaderProgram);
        glm::mat4 model = glm::mat4(1.0f);

        model = glm::translate (model, glm::vec3(0.1f, 0.5f, 0.0f));                    //--- model 행렬에 이동 변환 적용
        unsigned int modelLocation = glGetUniformLocation (shaderProgram, "modelTransform");
                                                                                        //--- 버텍스 세이더에서 modelTransform 변수 위치 가져오기
        glUniformMatrix4fv (modelLocation, 1, GL_FALSE, glm::value_ptr (model)); //--- modelTransform 변수에 변환 값 적용하기
        glBindVertexArray (VAO);
        glDrawArrays (GL_TRIANGLES, 0, 3);                                              //--- 도형 그리기
        glutSwapBuffers ();
}


//--- 버텍스 세이더
#version 330 core
layout (location = 0) in vec3 vPos;                                 //--- 응용 프로그램에서 받아온 도형 좌표값

uniform mat4 modelTransform;                                        //--- 모델링 변환 행렬: uniform 변수로 선언

void main()
{
        gl_Position = modelTransform * vec4(vPos, 1.0);             //--- 좌표값에 modelTransform 변환을 적용한다.
}
```

모델링 변환

- 1개 이상의 변환을 적용하는 경우
  - 예) x축으로 0.5 이동하고 z축에 대하여 45도 회전한다.

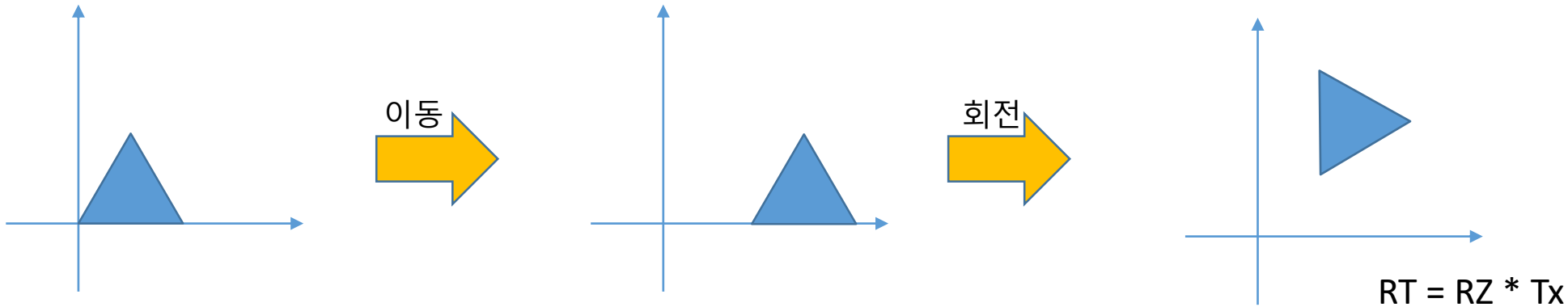
```
//--- 응용 프로그램
void drawScene ()
{
        glUseProgram (shaderProgram);
        glm::mat4 Tx = glm::mat4 (1.0f);                          //--- translation matrix
        glm::mat4 Rz = glm::mat4 (1.0f);                          //--- rotation matrix
        glm::mat4 TR = glm::mat4 (1.0f);                          //--- translation matrix
        Tx = glm::translate (Tx, glm::vec3 (0.5, 0.0, 0.0));      //--- x축으로 translation
        Rz = glm::rotate (Rz, glm::radians(45.0), glm::vec3 (0.0, 0.0, 1.0));   //--- z축에대하여 회전

        TR = Tx * Rz;                                             //--- 합성 변환 행렬: rotate -> translate

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



모델링 변환

- 변환 적용 시 순서에 따라 다른 결과가 나온다.

```
void drawScene ()
{
        glUseProgram (shaderProgram);
        glm::mat4 TR = glm::mat4 (1.0f);                                    //--- transformation matrix
        glm::mat4 Rz = glm::mat4 (1.0f);                                    //--- rotation matrix
        glm::mat4 Tx = glm::mat4 (1.0f);                                    //--- translation matrix
        Tx = glm::translate (Tx, glm::vec3 (0.5, 0.0, 0.0));                //--- x축으로 translation
        Rz = glm::rotate (Rz, glm::radians(45.0), glm::vec3 (0.0, 0.0, 1.0));   //--- z축에대하여 회전

        TR = Rz * Tx ;                                                      //--- 합성 변환 행렬: translate -> rotate

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



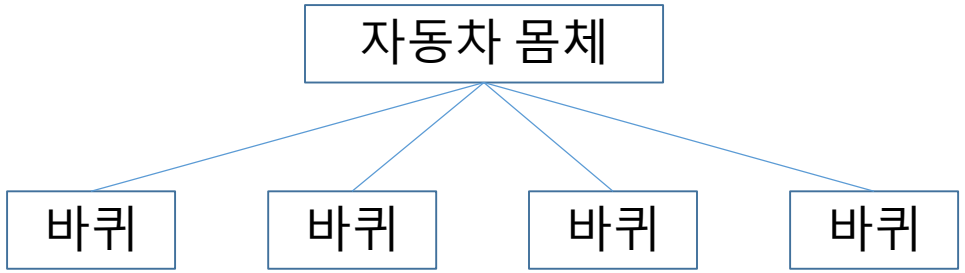
함수 프로토타입

- 유니폼 변수 다루기
  - GLint glGetUniformLocation (GLuint program, const GLchar *name);
    - 프로그램에서 uniform 변수의 위치를 가져온다.
      - program: 세이더 프로그램 이름
      - name: uniform 변수 이름
    - 리턴값: uniform 변수 위치 (-1: 위치를 찾지 못함)

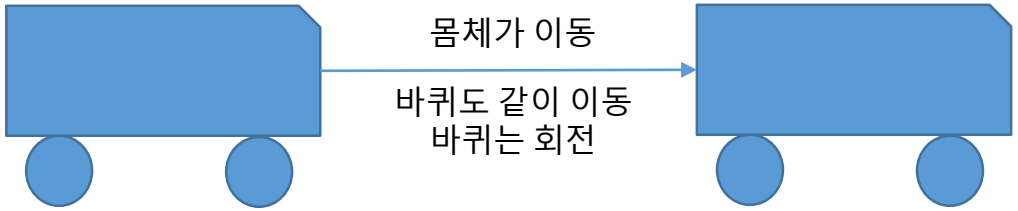
  - void glUniform{1|2|3|4}(f|i|ui} (GLuint location, {GLfloat v0, GLfloat v1, GLfloat v2, GLfloat v3});
    - glUniform1f, glUniform2f, glUniform3f, glUniform4f…
    - 현재 프로그램에서 uniform 변수의 값을 명시
      - location: 수정할 uniform 변수의 위치
      - vo, v1, v2, v3: 사용될 uniform 변수 값

계층적 변환

- 계층적 변환 (Hierarchical Transformation)
  - 한 객체의 변환을 다른 객체들의 변환에 소속시키는 것
  - 예) 자동차의 이동
    - 자동차 몸체가 이동한다.
    - 자동차 바퀴는 몸체가 이동한 곳에서 회전한다.



    - 계층적 구조를 만들기 위해서
      - 몸체 변환
      - 몸체 그리기
      - 변환 상태 저장
      - 앞바퀴 변환 적용
      - 앞바퀴 그리기
      - 몸체 변환으로 전환
      - 뒷바퀴 변환 적용
      - 뒷바퀴 그리기



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
    - …

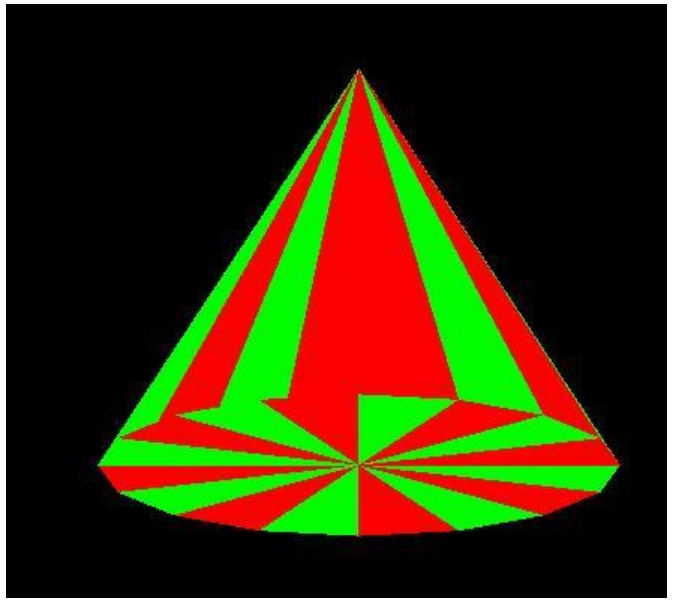
은면 제거

- 은면 제거
  - 3차원 장면을 2차원 평면에 투영시키면 물체들이 중첩될 수 있는데, 관측자의 시점에서 가까운 면은 보이고 깊이가 큰 면은 가려져 보이도록 은면 제거를 한다.

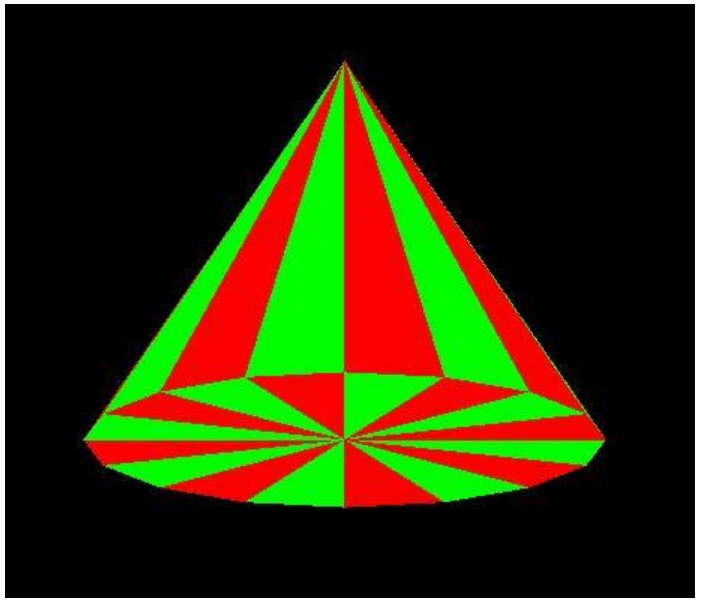
  - 깊이 검사 (depth test)를 사용한다.
    - 윈도우 초기화 시 깊이 검사 모드 설정
      - glutInitDisplayMode (GLUT_DOUBLE | GLUT_RGB | ***GLUT_DEPTH***)
    - 깊이 버퍼를 클리어한다
      - glClear (GL_COLOR_BUFFER_BIT | ***GL_DEPTH_BUFFER_BIT***);
    - 깊이 검사를 설정:
      - **glEnable** (***GL_DEPTH_TEST***);
    - 깊이 검사를 해제:
      - **glDisable** (***GL_DEPTH_TEST***);

  - 메인 프로그램에서 깊이 검사를 설정한다.

은면 제거



Depth test를 안 했을 때



Depth test를 했을 때

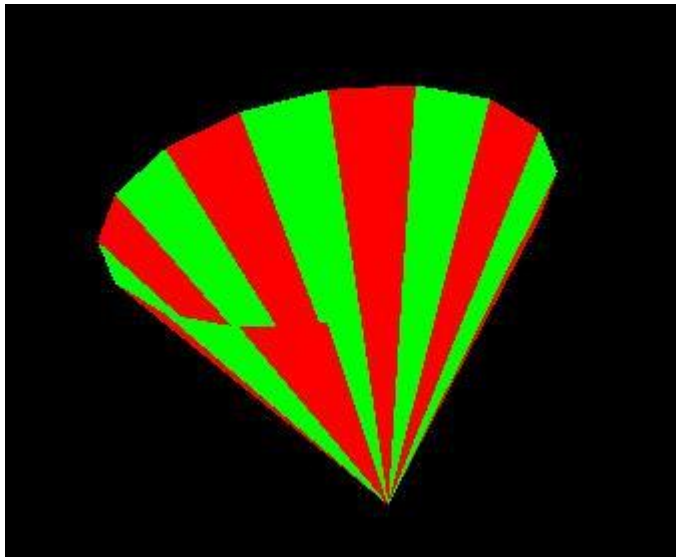
은면 제거

- 컬링 (culling)
  - 후면을 선별(backface culling)하여 **뒷면을 모두 제거**할 수 있다.
    - Winding을 이용하여 폴리곤의 앞면과 뒷면을 구분한다.
    - 시계 반대방향으로 winding되는 폴리곤이 앞면이다.

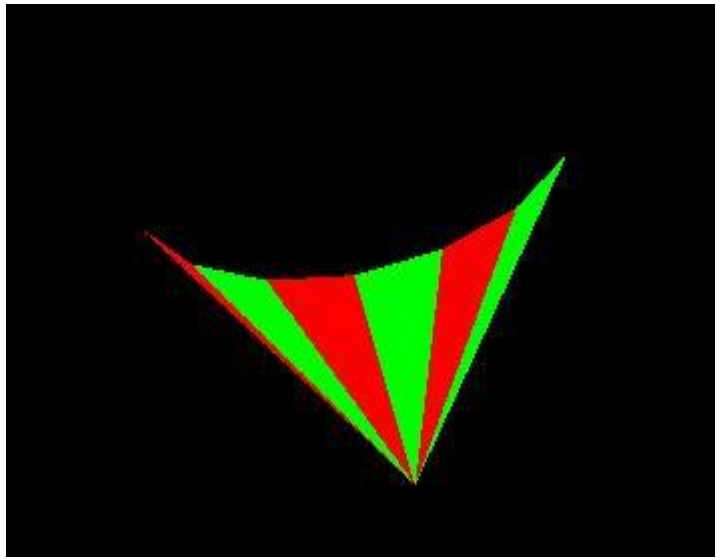
      - 컬링 설정: glEnable (**GL_CULL_FACE**);
      - 컬링 해제: glDisable (**GL_CULL_FACE**);

      - 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

은면 제거



Culling을 안 했을 때



Culling을 했을 때

3차원 객체 그리기: GLU 모델

- GLU 라이브러리를 이용하여 모델링 하기
  - 2차 곡선 (Quadrics)을 이용한다.
    - GLUquadricObj * gluNewQuadric ();
      - Quadric Object 를 생성
    - void gluQuadricDrawStyle (GLUqadric *quadObject, GLenum drawStyle);
      - 도형의 스타일 지정하기
      - drawStyle: GLU_FILL / GLU_LINE / GLU_SILHOUETTE / GLU_POINT
    - void gluQuadricNormals (GLUqadric *quadObject, GLenum normals);
      - 법선 벡터 제어, 빛에 대한 영향 결정
      - Normals: GLU_NONE / GLU_FLAT / GLU_SMOOTH
    - void gluquadricOrientation (GLUqadric *quadObject, GLenum orientation);
      - 법선 벡터의 내부 및 외부 등과 같은 방향 지정
      - Orientation: GLU_OUTSIDE / GLU_INSIDE
    - void gluDeleteQuadric (GLUqadric *quadObject);
      - 객체 삭제하기

3차원 객체 그리기: GLU 모델

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

void gluSphere (GLUquadric **qobj*, GLdouble *radius*, GLint *slices*, GLint *stacks*);		
Parameters Help	*qobj*	// gluNewQuadric으로 생성된 Quadric Object
	radius	// Sphere의 반지름(Radius)
	slices	// Z축을 중심으로 하는 Subdivisions의 개수(경도(Longitude)와 유사)
	stacks	// Z축을 따르는 Subdivisions의 개수(위도(Latitude)와 유사)

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

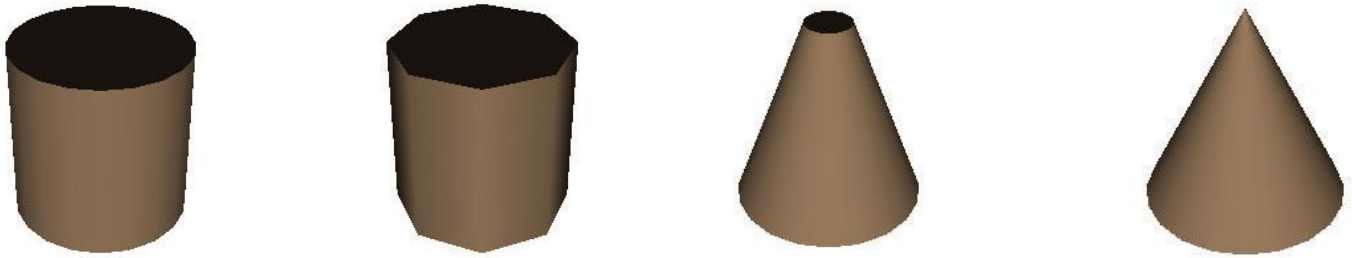
(B) gluSphere (obj, 1.0, 10, 10);

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

3차원 객체 그리기: GLU 모델

- 실린더 생성하기
  - void gluCylinder (GLUquadirc *qobj, GLdouble baseRadium, GLdouble topRadius, GLdouble height, GLint slices, GLint stack);

void gluCylinder (GLUquadric *qobj, GLdouble baseRadius, GLdouble topRadius, GLdouble height, GLint slices, GLint stacks);		
Parameters Help	*qobj*	// gluNewQuadric로 생성되어진 Quadric Object
	baseRadius	// z=0에 있는 Cylinder의 반지름(Radius)
	topRadius	// z=height에 있는 Cylinder의 반지름(Radius)
	height	// Cylinder의 높이(Height)
	slices	// Z축을 중심으로 하는 회전 Subdivisions의 개수
	stacks	// Z축을 따르는 Subdivisions의 개수



(A) (B) (C) (D)

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

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

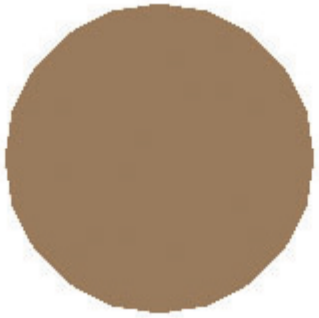
(C) gluCylinder (obj, 1.0, 0.3, 2.0, 20, 8);

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

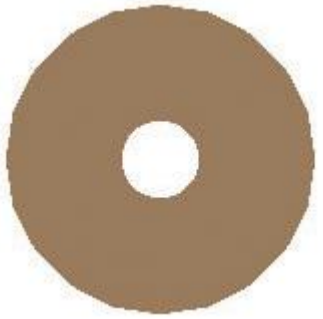
3차원 객체 그리기: GLU 모델

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

void gluDisk (GLUquadric **qobj*, GLdouble *innerRadius*, GLdouble *outerRadius*, GLint *slices*, GLint *loops*);		
Parameters Help	*qobj*	// gluNewQuadric으로 생성된 Quadric Object
	innerRadius	// Disk의 안쪽 반지름(Radius)
	outerRadius	// Disk의 바깥쪽 반지름(Radius)
	slices	// Z축을 중심으로 하는 Subdivisions의 개수
	loops	// Disk가 세분화되는 동심원의 개수



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

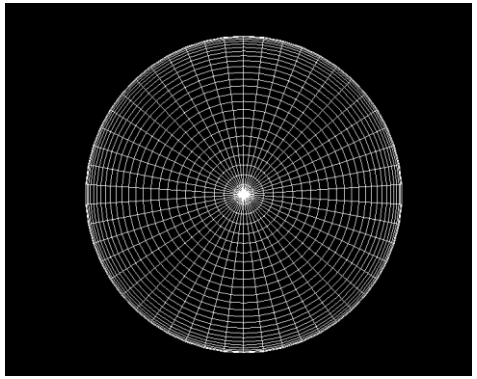


(B) gluDisk(obj, 0.5, 2.0, 20, 3);

3차원 객체 그리기: GLU 모델

- 모델 생성 예)

```
GLUquadricObj *qobj;

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

   qobj = gluNewQuadric ();                          // 객체 생성하기
   gluQuadricDrawStyle( qobj, GLU_LINE );            // 도형 스타일
   gluQuadricNormals( qobj, GLU_SMOOTH );            //→ 생략 가능
   gluQuadricOrientation( qobj, GLU_OUTSIDE );       //→ 생략 가능
   gluSphere( qobj, 1.5, 50, 50 );                   // 객체 만들기

   glutSwapBuffers();
}
```



.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

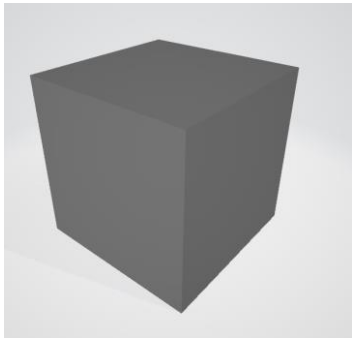
- 파일 포맷 (#으로 시작하는 줄은 주석)

```
# 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 슬라이스 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

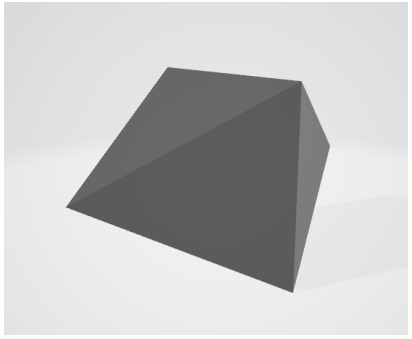
- 육면체

```
# 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

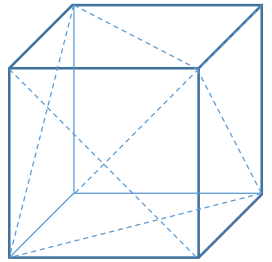
```
# OBJ file created by ply_to_obj.c
#
g Object001

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
…
```

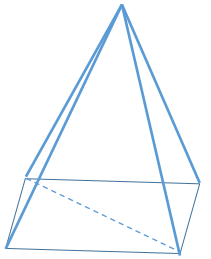
.obj 파일 읽기 샘플 (버전에 따라 수정 필요)

```
#include <stdio.h>

void ReadObj (FILE *objFile)
{
    //--- 1. 전체 버텍스 개수 및 삼각형 개수 세기
    char count[100];
    int vertexNum = 0;
    int faceNum = 0;

    while (!feof (objFile)) {
        fscanf (objFile, "%s", count);
        if (count[0] == 'v' && count[1] =='₩0')
                vertexNum += 1;
        else if (count[0] == 'f' && count[1] =='₩0')
                faceNum += 1;
        memset (count, '₩0', sizeof (count));              // 배열 초기화
    }


    //--- 2. 메모리 할당
    vec4 *vertex;
    vec4 *face;
    int vertIndex = 0;
    int faceIndex = 0;

    vertex = (vec4 *)malloc (sizeof (vec4) * vertexNum);
    face    = (vec4 *)malloc (sizeof (vec4) * faceNum);

    //--- 3. 할당된 메모리에 각 버텍스, 페이스 정보 입력
    while (!feof (objFile)) {
        fscanf (objFile, "%s", bind);
        if (bind[0] == 'v' && bind[1] == '₩0') {
                fscanf (objFile, "%f %f %f",
                        &vertex[vertIndex].x, &vertex[vertIndex].y,
                        &vertex[vertIndex].z);
                vertIndex ++;
        }


        else if (bind[0] == 'f' && bind[1] == '₩0') {
                fscanf (objFile, "%f %f %f",
                        &face[faceIndex].x, &face[faceIndex].y, &face[faceIndex].z);
                faceIndex ++;
        }
    }

}
```

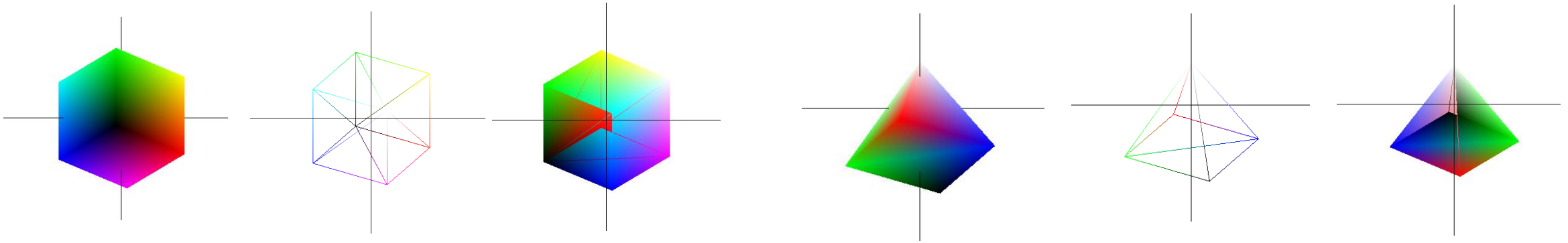
실습 9

- 꼭지점을 이용하여 3차원 객체 그리기
  - 화면 중앙에 좌표계 (x축, y축, z축)을 그린다.
  - 키보드 명령에 따라 육면체 혹은 사각뿔을 그린다.
    - 객체는 X축으로 30도, y축으로 -30도 회전해 있다.
  - 키보드 명령
    - C: 육면체
    - P: 사각 뿔 (피라미드 모양으로 바닥은 사각형, 옆면은 삼각형)
    - H: 은면제거 적용/해제
    - y/Y: y축을 기준으로 양/음 방향으로 회전
    - w/W: 와이어 객체/솔리드 객체



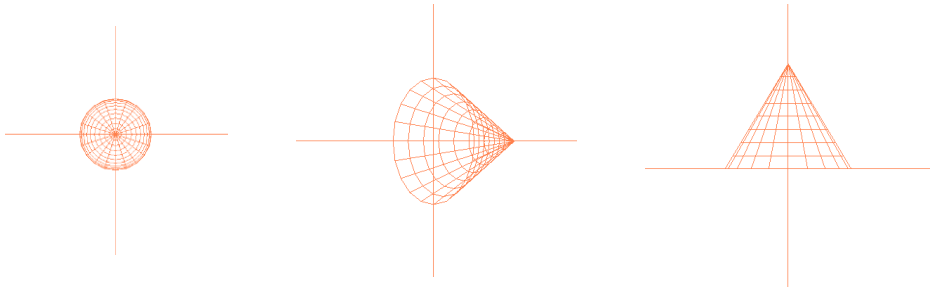


  - 각 꼭지점에 다른 색상을 설정한다.



실습 10

- 모델링 변환하기
  - 화면 중앙에 좌표계를 그린다.
  - 키보드 명령에 따라 화면 중앙에 다른 3차원 객체를 그린다. 크기는 [0.0, 1.0]사이에서 그린다.
    - 1: 구
    - 2: 원뿔 (뾰족한 부분이 z축을 따라)
    - 3: 원뿔 (뾰족한 부분이 y축을 따라)
    - 4: 디스크
    - 5: 실린더
  - 키보드 명령으로 객체를 이동한다.
    - w/W: 솔리드/와이어 모델
    - ←/→/↑/↓: 좌/우/상/하로 객체를 이동한다.
    - y/Y: y축으로 양/음 방향으로 자전한다.
    - r/R: 좌표계의 y축으로 양/음 방향으로 공전한다.
    - C: 모든 변환 리셋
    - 위의 명령어를 입력했을 때, 좌표계는 이동하지 않는다.



실습 11

실습 12

뷰잉 변환

- 월드 좌표계를 유저의 시점인 view space로 변환
  - View space: 월드 좌표를 유저의 시점 앞에 있는 좌표로 변환한 결과
    - 즉, 카메라의 관점에서 바라보는 공간
    - 카메라 초기값: 좌표계의 원점에 위치
    - 모델링 변환 함수들을 사용하여 카메라의 위치를 바꿀 수 있다.
    - 즉, 카메라의 위치 변환은 객체들의 위치 변환과 반대로 변환하는 것과 같다.
      - 카메라를 오른쪽으로 이동 == 객체를 왼쪽으로 이동
  - 카메라 위치 지정방법
    - 카메라를 원점으로부터 뒤로 이동시키는 방법
    - 또는 물체를 카메라의 앞으로 이동시키는 방법
  - 연속된 회전과 이동으로 카메라 위치를 지정할 수 있다.

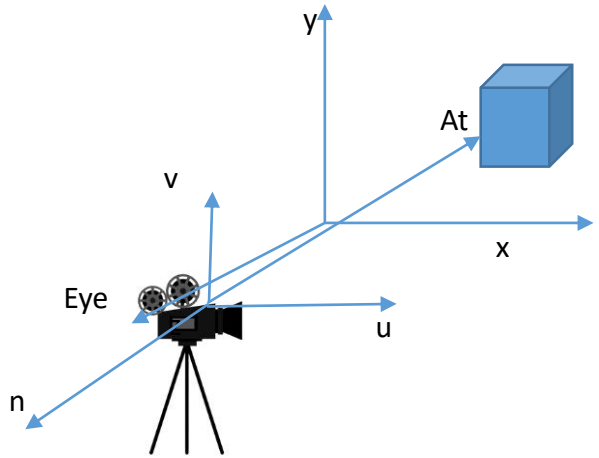
뷰잉 변환

- 뷰잉 변환 적용하기
  - 뷰잉 변환 행렬 M_{view} 값을 설정하여 버텍스 세이더에 적용한다.
  - 물체를 이동하여 카메라의 위치를 지정한다.

```
//--- 응용 프로그램
void drawScene ()
{
	glUseProgram (shaderProgram);
	glm::mat4 view = glm::mat4(1.0f);
	view = glm::translate (view, glm::vec3(0.0f, 0.0f, -3.0f));                    //--- z축으로 -3만큼 이동
	unsigned int viewLocation = glGetUniformLocation (shaderProgram, "viewTransform"); //--- 버텍스 세이더에서 viewTransform  변수위치
	glUniformMatrix4fv (viewLocation, 1, GL_FALSE, &view[0][0]);                   //--- viewTransform 변수에 변환값 적용하기
	glBindVertexArray (VAO);
	glDrawArrays (GL_TRIANGLES, 0, 3);
	glutSwapBuffers ();
}

//--- 버텍스 세이더
#version 330 core
layout (location = 0) in vec3 vPos;

uniform mat4 modelTransform;
uniform mat4 viewTransform;

void main()
{
		gl_Position = viewTransform * modelTransform * vec4(vPos, 1.0);
}
```

뷰잉 변환

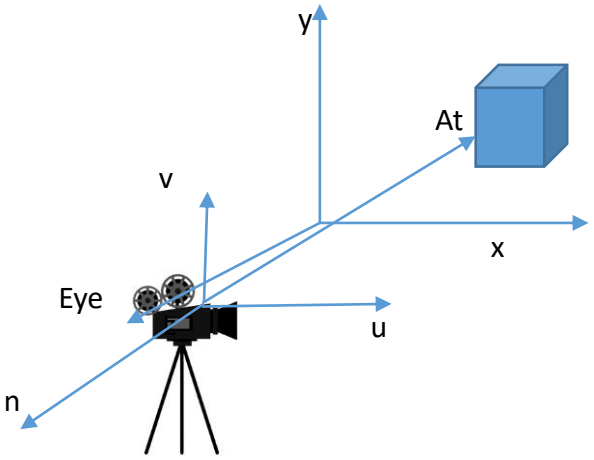
- 카메라 공간
  - 다음의 세 가지 파라미터를 가진다.
    - Eye: 월드 공간에서의 카메라의 위치
    - At: 월드 공간에서 카메라가 바라보는 기준점
    - Up: 카메라의 상단이 가리키는 방향,
      - 대부분의 경우 Up은 월드 공간의 y축



  - 카메라 공간 설정을 위한 3차원 뷰잉 변환 행렬은
    - $T = \begin{bmatrix} 1 & 0 & 0 & -tx \\ 0 & 1 & 0 & -ty \\ 0 & 0 & 1 & -tz \\ 0 & 0 & 0 & 1 \end{bmatrix}$ $R = \begin{bmatrix} ux & uy & uz & 0 \\ vx & vy & vz & 0 \\ nx & ny & nz & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$
    - $M_{view} = \begin{bmatrix} ux & uy & uz & -txu \\ vx & vy & vz & -tyv \\ nx & ny & nz & -tzn \\ 0 & 0 & 0 & 1 \end{bmatrix}$

뷰잉 변환

- 카메라 공간
  - 다음의 세 가지 파라미터를 가진다.
    - Eye: 월드 공간에서의 카메라의 위치
    - At: 월드 공간에서 카메라가 바라보는 기준점
    - Up: 카메라의 상단이 가리키는 방향,
      - 대부분의 경우 Up은 월드 공간의 y축



  - 단위 벡터, 외적 등을 이용하여 카메라의 파라미터를 설정한다.
    - 카메라 객체 좌표계 설정
      - 카메라 위치:
        - glm::vec3 cameraPos = glm::vec3 (0.0f, 0.0f, 5.0f); //--- (0.0, 0.0, 5.0)위치에 카메라를 놓음
      - 카메라 방향: 카메라가 바라보는 방향 (n)
        - glm::vec3 cameraTarget = glm::vec3 (0.0f, 0.0f, 0.0f); //--- 카메라가 바라보는 곳
        - glm::vec3 cameraDirection = glm::normalize (cameraPos-cameraTarget); //--- 카메라 방향 벡터: z축 양의 방향
      - 카메라의 오른쪽 축 (u)
        - 위쪽 벡터와 카메라 방향 벡터와의 외적
        - glm::vec3 up = glm::vec3 (0.0f, 1.0f, 0.0f);
        - glm::vec3 cameraRight = glm::normalize (glm::cross (up, cameraDirection));
      - 카메라의 위쪽 축 (v)
        - glm::vec3 cameraUp = glm::cross (cameraDirection, cameraRight); //--- 카메라의 위쪽 방향

함수 프로토타입

- 카메라 설정 함수
  - glm::mat4 glm::**lookAt** (vec3 const &cameraPos, vec3 const &cameraDirection, vec3 const &cameraUp);
    - cameraPos: 카메라의 위치
    - cameraDirection: 카메라가 바라보는 기준점
    - cameraUp: 카메라의 상단이 가리키는 방향
  - 카메라 변환 위해서는
    - 위의 세 인자값 변경
    - 변환 한수를 적용하여 카메라 위치 및 방향 변경 가능
  - 사용 예)
    - 카메라를 (0, 0, 3)위치에 두고 z축의 음의 방향으로 카메라를 놓는다.
    - glm::lookat (glm::vec3 (0.0f, 0.0f, 3.0f), glm::vec3 (0.0f, 0.0f, 0.0f), glm::vec3 (0.0f, 1.0f, 0.0f));

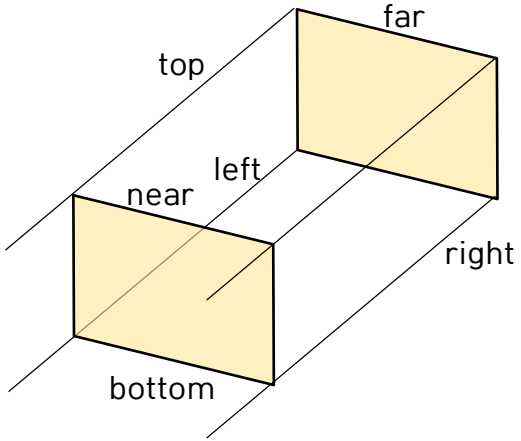
뷰잉 변환

- 카메라의 위치 설정 예)

```
//--- 응용 프로그램
void drawScene ()
{
    glUseProgram (shaderProgram);
    glm::vec3 cameraPos   = glm::vec3(0.0f, 0.0f,  5.0f);
    glm::vec3 cameraDirection = glm::vec3(0.0f, 0.0f, 0.0f);
    glm::vec3 cameraUp    = glm::vec3(0.0f, 1.0f,  0.0f);
    glm::mat4 view = glm::mat4 (1.0f);

    view = glm::lookAt (cameraPos, cameraDirection, cameraUp);
    unsigned int viewLocation = glGetUniformLocation (shaderProgram, "viewTransform");
    glUniformMatrix4fv (viewLocation, 1, GL_FALSE, &view[0][0]);
    glBindVertexArray (VAO);
    glDrawArrays (GL_TRIANGLES, 0, 3);                                    //--- 도형 그리기
    glutSwapBuffers ();
}


//--- 버텍스 세이더
#version 330 core
layout (location = 0) in vec3 vPos;

uniform mat4 modelTransform;
uniform mat4 viewTransform;

void main()
{
        gl_Position = viewTransform * modelTransform * vec4(vPos, 1.0);
}
```

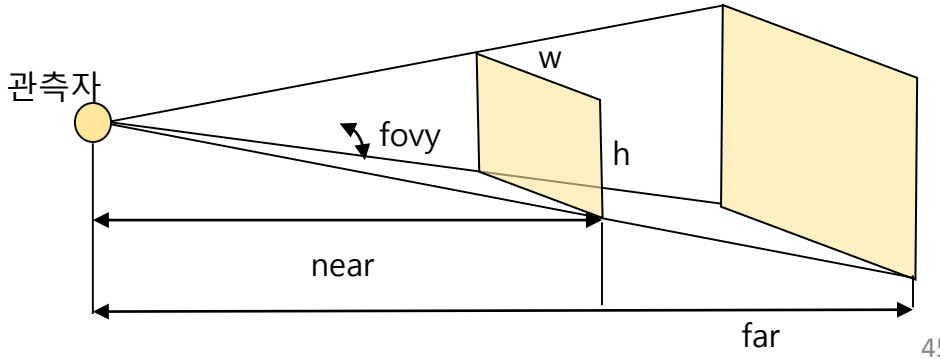
투영 변환

- 객체가 놓이는 공간 설정
  - $M_{Projection}$ 행렬: 투영 공간을 설정하는 변환 행렬
    - 절두체 (frustum)의 투영 공간: 직육면체 또는 피라미드 형태의 절두체
    - 공간 설정은 대개 바뀌지 않으므로 초기화 함수에서 한 번 적용하는 것이 좋음



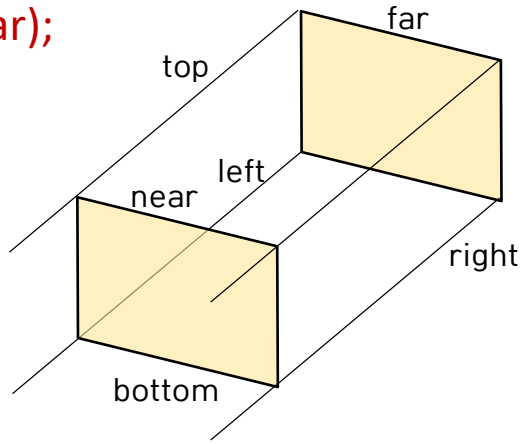
  - 직각 투영
    - 투영 공간이 직육면체: orthographic projection 행렬이 적용된다.
    - 공간 외에 있는 객체들은 clip된다.
    - 2D 평면에 똑바로 매핑되고 원근감이 없다.

  - 원근 투영
    - 투영 공간이 피라미드 형태: perspective projection 행렬이 적용된다.
    - 원근감을 가진다.



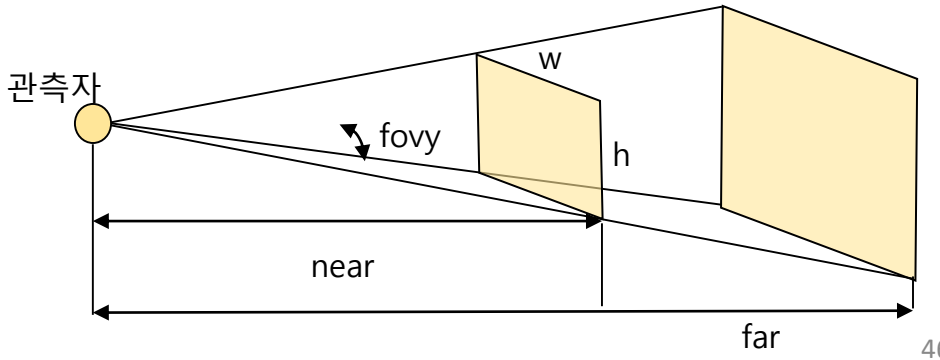
함수 프로토타입

- 직각 투영 볼륨
  - glm::mat4 glm::ortho (float left, float right, float bottom, float top, float near, float far);
    - left, right: 절두체의 왼쪽, 오른쪽 좌표
    - bottom, top: 절두체의 맨 밑과 맨 위의 좌표
    - near, far: 가까운 평면과 먼 평면



- 원근 투영 볼륨
  - glm:: mat4 glm::perspective (float fovy, float aspect, float near, float far);
    - fovy: 뷰잉 각도(라디언), 뷰잉 공간이 얼마나 큰지를 설정
    - aspect: 종횡비 (앞쪽의 클리핑 평면의 폭(w)을 높이(h)로 나눈 값)
      - 종횡비: 화면의 가로방향에 대한 단위 길이를 나타내는 픽셀수에 대한 세로방향의 단위길이를 나타내는 픽셀 수의 비율.

      예) 종횡비가 0.5: 가로길이의 두 픽셀이 세로길이의 한 픽셀에 대응한다.
    - near: 관측자에서부터 가까운 클리핑 평면까지의 거리 (항상 양의 값)
    - far: 관측자에서 먼 클리핑 평면까지의 거리 (항상 양의 값)



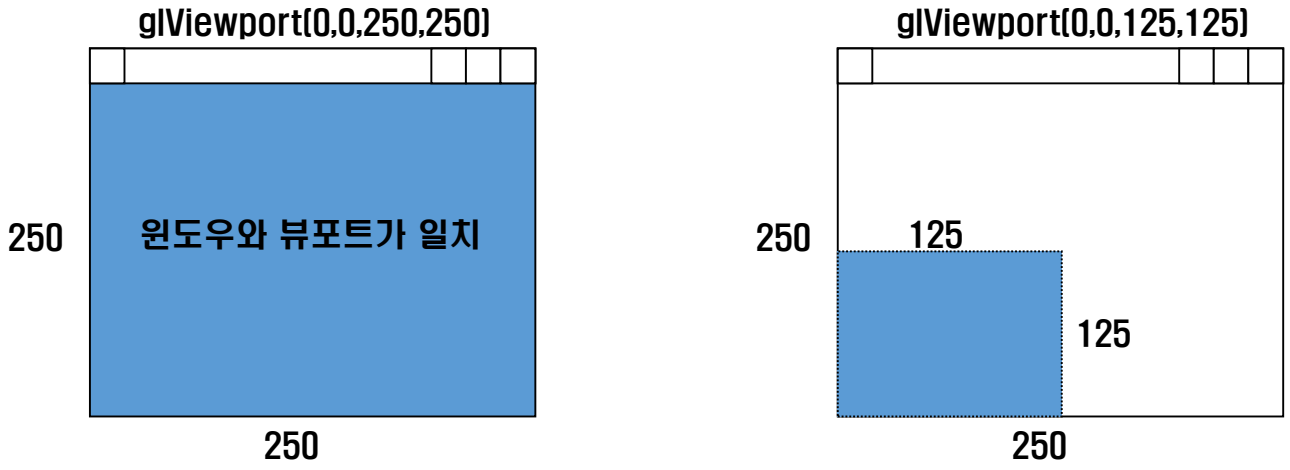
투영 변환

- 투영 변환 예)
  - 원근 투영

```
//--- 응용 프로그램
void drawScene ()
{
	glUseProgram (shaderProgram);
	glm::mat4 projection = glm::mat4(1.0f);
	projection = glm::perspective (glm::radians(45.0f), (float)SCR_WIDTH / (float)SCR_HEIGHT, 0.1f, 100.0f);

	unsigned int projectionLocation = glGetUniformLocation (shaderProgram, “projectionTransform”);
	glUniformMatrix4fv (projectionLocation, 1, GL_FALSE, &projection[0][0]);
	glBindVertexArray (VAO);
	glDrawArrays (GL_TRIANGLES, 0, 3);                                    //--- 도형 그리기
	glutSwapBuffers ();
}

//--- 버텍스 세이더
#version 330 core
layout (location = 0) in vec3 vPos;

uniform mat4 modelTransform;
uniform mat4 viewTransform;
uniform mat4 projectionTransform;

void main()
{
	gl_Position = projectionTransform * viewTransform * modelTransform * vec4(vPos, 1.0);
}
```

뷰포트 변환

- 화면의 최종적으로 출력될 영역 설정
- void **glViewport** (GLint x, GLint y, GLsizei width, GLsizei height);
  - 윈도우의 영역을 설정한다.
    - x, y: 뷰포트 사각형의 왼쪽 아래 좌표
    - width, height: 뷰포트의 너비와 높이



- Reshape 함수가 있을 때, 대개 그 함수에 포함시킨다.

모델링 좌표값에 변환 적용하기

- 버텍스 세이더에 변환 행렬을 적용
  - 메인 프로그램에서 변환 행렬 설정
    - M_{model} 행렬

```
glm::mat4 model;
model = glm::rotate(model, glm::radians(30.0f), glm::vec3(1.0f, 0.0f, 0.0f));          //--- x축에 대하여 30도 회전
```

    - M_{view} 행렬

```
glm::mat4 view;
view = glm::translate(view, glm::vec3(0.0f, 0.0f, -3.0f));          //--- 카메라를 z축으로 3만큼 이동, 카메라와 객체 변환은 반대
```

    - $M_{projection}$ 행렬

```
glm::mat4 projection;
projection = glm::perspective(glm::radians(45.0f), screenWidth / screenHeight, 0.1f, 100.0f);     //--- 원근 투영 적용
```

모델링 좌표값에 변환 적용하기

- 버텍스 세이더에 변환 행렬 적용

```
#version 330 core
layout (location = 0) in vec3 vPos;       //--- 객체의 원래 버텍스 좌표값
layout (location = 1) in vec3 vColor;     //--- 객체의 색상값

uniform mat4 model;                       //--- 모델링 변환값: 응용 프로그램에서 전달 -> uniform 변수로 선언: 변수 이름“model”로 받아옴
uniform mat4 view;                        //--- 뷰잉 변환값: 응용 프로그램에서 전달 -> uniform 변수로 선언: 변수 이름“view”로 받아옴
uniform mat4 projection;                  //--- 투영 변환값: 응용 프로그램에서 전달 -> uniform 변수로 선언: 변수 이름“projection”로 받아옴
out vec3 passColor;

void main()
{
     gl_Position = projection * view * model * vec4(vPos, 1.0);          //--- 변환은 ← 방향으로 적용
                                                                         //--- vPos: 객체의 원래 좌표값
     passColor = vColor;
}
```

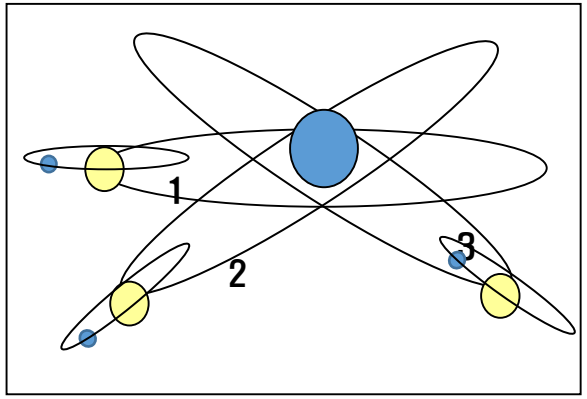
모델링 좌표값에 변환 적용하기

- 응용 프로그램의 그리기 함수에서 버텍스 세이더에 uniform으로 선언되어 있는 변환 행렬 값을 전달

```
void drawScene ()
{
	glUseProgram (shaderProgram);
	glBindVertexArray (VAO);
	//--- 버텍스 세이더에서 각 변환 행렬값을 받아온다.
	int modelLoc = glGetUniformLocation (shaderProgram,  “model”);
	int viewLoc = glGetUniformLocation (shaderProgram, “view”);
	int projLoc = glGetUniformLocation (shaderProgram,  “projection”);

	//--- 모델링 변환, 뷰잉 변환, 투영 변환 행렬을 설정한 후, 버텍스 세이더에 저장한다.
	glm::mat4 transform = glm::mat4 (1.0f);
	transform = glm::rotate (transform, glm::radians(yAngle), glm::vec3(0.0f, 1.0, 0.0f));
	glUniformMatrix4fv (modelLoc, 1, GL_FALSE, glm::value_ptr(transform));

	glm::mat4 view = glm::mat4 (1.0f);
	view = glm::lookAt (cameraPos, cameraDirection, cameraUp);
	glUniformMatrix4fv (viewLoc, 1, GL_FALSE, &view[0][0]);

	glm::mat4 projection = glm::mat4 (1.0f);
	projection = glm::perspective (glm::radians(60.0f), (float)g_window_w / (float)g_window_h, 0.1f, 200.0f);
	glUniformMatrix4fv (projLoc, 1, GL_FALSE, &projection[0][0]);

	//--- 객체를 그린다.
	glDrawArrays (GL_TRIANGLES, 0, 12);
}
```

실습 13

- 중심의 구를 중심으로 3개의 구가 다른 방향의 경로를 따라 회전하는 애니메이션 제작, 각 구에는 그 구를 중심으로 달이 공전한다. 이때, 회전 경로는 원을 사용하여 그리도록 한다.
  - 은면제거, 원근 투영을 적용한다.
  - 경로 1: xz 평면
  - 경로 2: xz 평면이 반시계방향으로 45도 기울어져 있다.
  - 경로 3: xz 평면이 시계방향으로 45도 기울어져 있다.

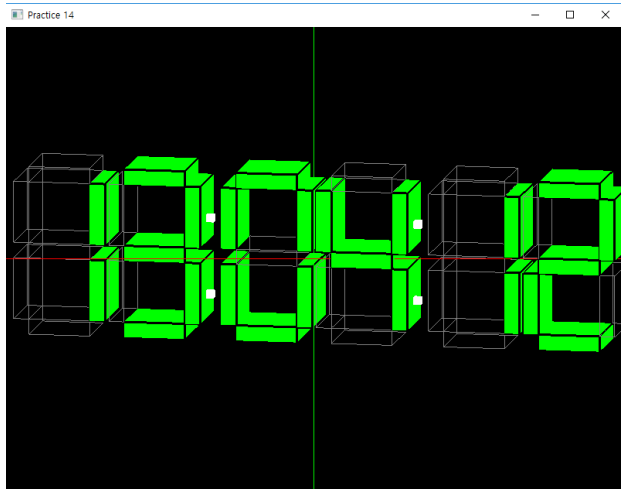


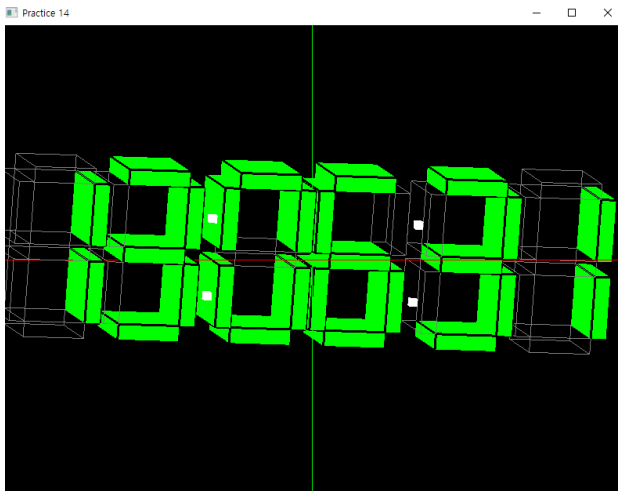
  - 3개의 구는 다른 속도로 중심의 구를 공전한다.
  - 3개의 구에는 각각 공전하는 달을 가지고 있다.
  - 달의 궤도는 공전 궤도와 평행하다.
  - 메뉴를 이용하여 구의 모델을 선택할 수 있게 한다.
    - 솔리드 모델 / 와이어 모델

  - 키보드 명령
    - w/a/s/d: 위의 도형들을 좌/우/상/하로 이동
    - z/x: 위의 도형들을 앞/뒤로 이동
    - y/Y: 전체 객체들을 y축으로 양/음 방향으로 회전

실습 14

- 육면체를 이용하여 시간 출력하기
  - 육면체를 사용하여 시계를 구현한다.
  - x축으로 30도, y축으로 30도 회전 시킨 방향으로 초기화 되어있다.
  - 다음의키보드 명령을 수행한다.
    - y/Y: 화면 전체 y축에 대하여 양/음 방향으로 회전한다.
    - z/Z: 화면 전체를 z축으로 양/음 방향으로 이동한다.





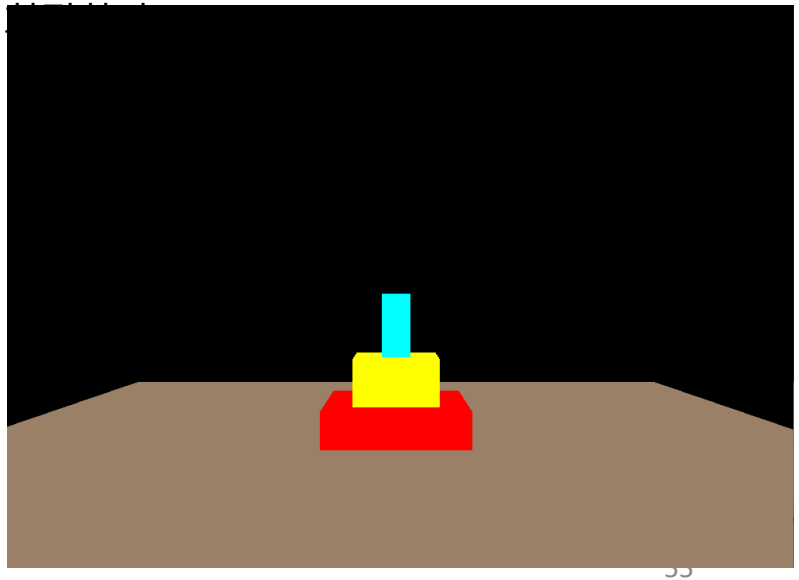
  - 현재 시간 가져오기

```
time_t          now = time(0);
struct tm       curr_time;
localtime_s     (&curr_time, &now);

// correct_wtime.h
struct tm
{
    int tm_sec;         // seconds after the minute - [0, 60] including leap second
    int tm_min;         // minutes after the hour - [0, 59]
    int tm_hour;        // hours since midnight - [0, 23]
    int tm_mday;        // day of the month - [1, 31]
    int tm_mon;         // months since January - [0, 11]
    int tm_year;        // years since 1900
    int tm_wday;        // days since Sunday - [0, 6]
    int tm_yday;        // days since January 1 - [0, 365]
    int tm_isdst;       // daylight savings time flag
};
```

실습 15

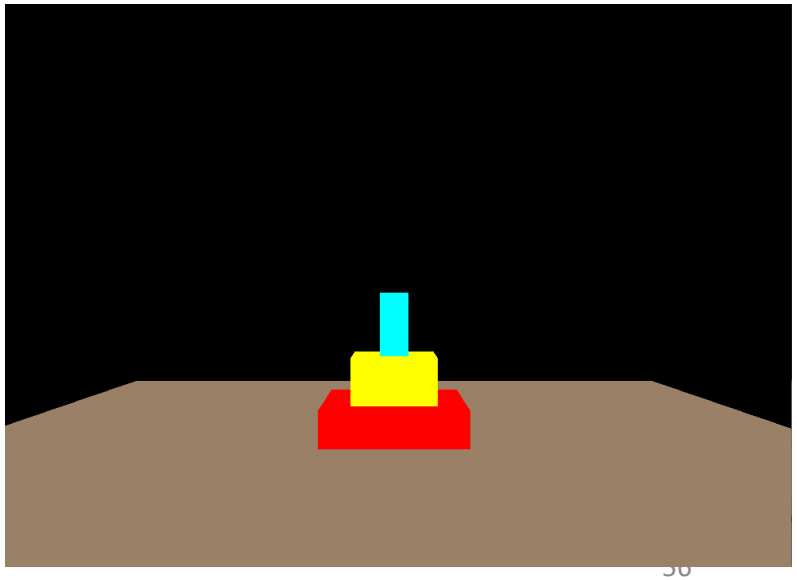
실습 16

- 이동하는 크레인 만들기
  - 원근 투영과 은면제거를 넣는다.
  - 크레인을 올릴 바닥을 그린다.
  - 3개의 육면체를 이용하여 크레인을 만든다.
    - 크레인은 바닥위에 놓여있다.
    - 각 몸체는 다른 색으로 설정한다.
  - 키보드 명령
    - b/B: 크레인의 아래 몸체가 y축에 대하여 양/음 방향으로 회전한다.
      - 아래 몸체가 회전하면 중앙 몸체와 맨 위의 팔은 같이 회전한다.
    - m/M: 크레인의 중앙 몸체가 x축에 대하여 양/음 방향으로 회전한다.
      - 회전 각도는 90 ~ 180도 사이로 정하고, 중앙 몸체가 회전하면 맨 위의 팔도 같이
    - t/T: 크레인의 맨 위 팔이 z축에 대하여 양/음 방향으로 회전한다.
      - 회전 각도는 90~180도 사이로 정한다.
    - s/S: 움직임 멈추기
    - c/C: 모든 움직임이 초기화된다.
    - Q: 프로그램 종료하기



실습 17

- 카메라 넣기
  - 앞의 크레인 실습에 카메라를 적용한다.
  - 원점에 크레인을 렌더링하고 크레인은 좌우로 이동하고 있다.
  - 크레인을 바라보는 카메라 설정
    - 키보드 명령으로 카메라 이동
      - z/Z: 앞뒤로 이동
      - x/X: 좌우로 이동
      - r/R: 카메라 기준 y축에 대하여 회전



실습 18