

[CSE4170 기초 컴퓨터 그래픽스]

2019년도 1학기

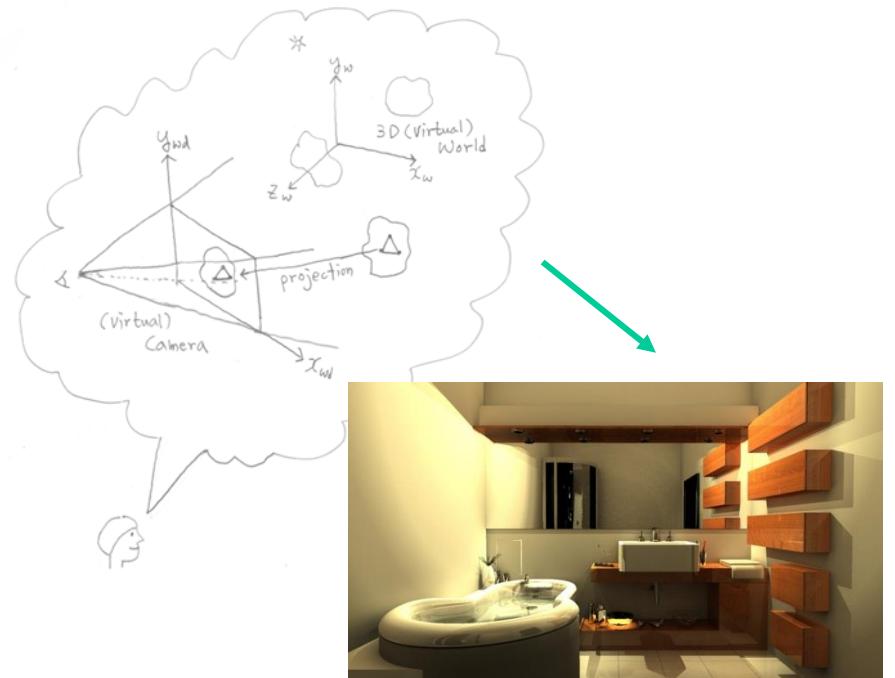
강의자료 II

Introduction to 3D Rendering

Rendering Algorithms

What is Rendering?

- 상상으로 존재하는 가상의 3D 세상으로부터 가상의 카메라를 사용하여 원하는 내용을 촬영하는 상황을 컴퓨터를 사용하여 시뮬레이션 하여 해당하는 2D 이미지를 생성해주는 과정임.
- 궁극적으로 rendering equation과 volume rendering equation에 의해 기술되는 물리적인 현상을 “**실용적인**” 렌더링 알고리즘을 사용하여 “**효과적으로**” 시뮬레이션 하여 “**사실적인**” 영상 생성.

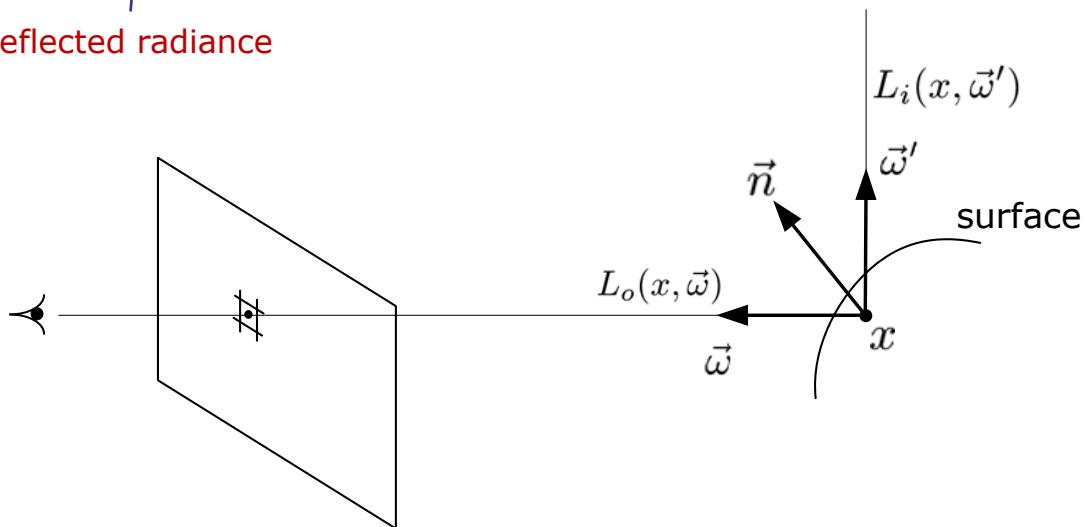


Rendering and Volume Rendering Equations

- The mathematical basis for all global illumination algorithms
- **Rendering equation for surfaces**

$$\begin{aligned} L_o(x, \vec{\omega}) &= L_e(x, \vec{\omega}) + L_r(x, \vec{\omega}) \\ &= L_e(x, \vec{\omega}) + \int_{\Omega} f_r(x, \vec{\omega}', \vec{\omega}) L_i(x, \vec{\omega}') (\vec{\omega}' \cdot \vec{n}) d\vec{\omega}' \end{aligned}$$

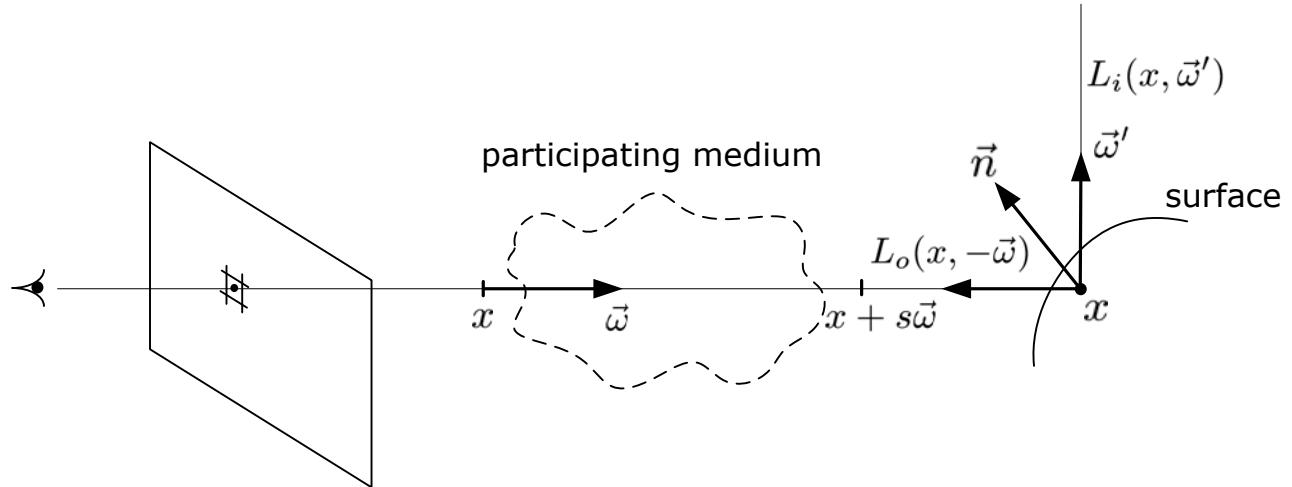
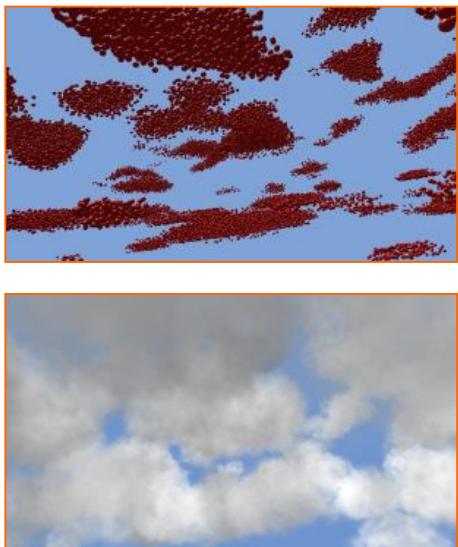
outgoing radiance emitted radiance reflected radiance



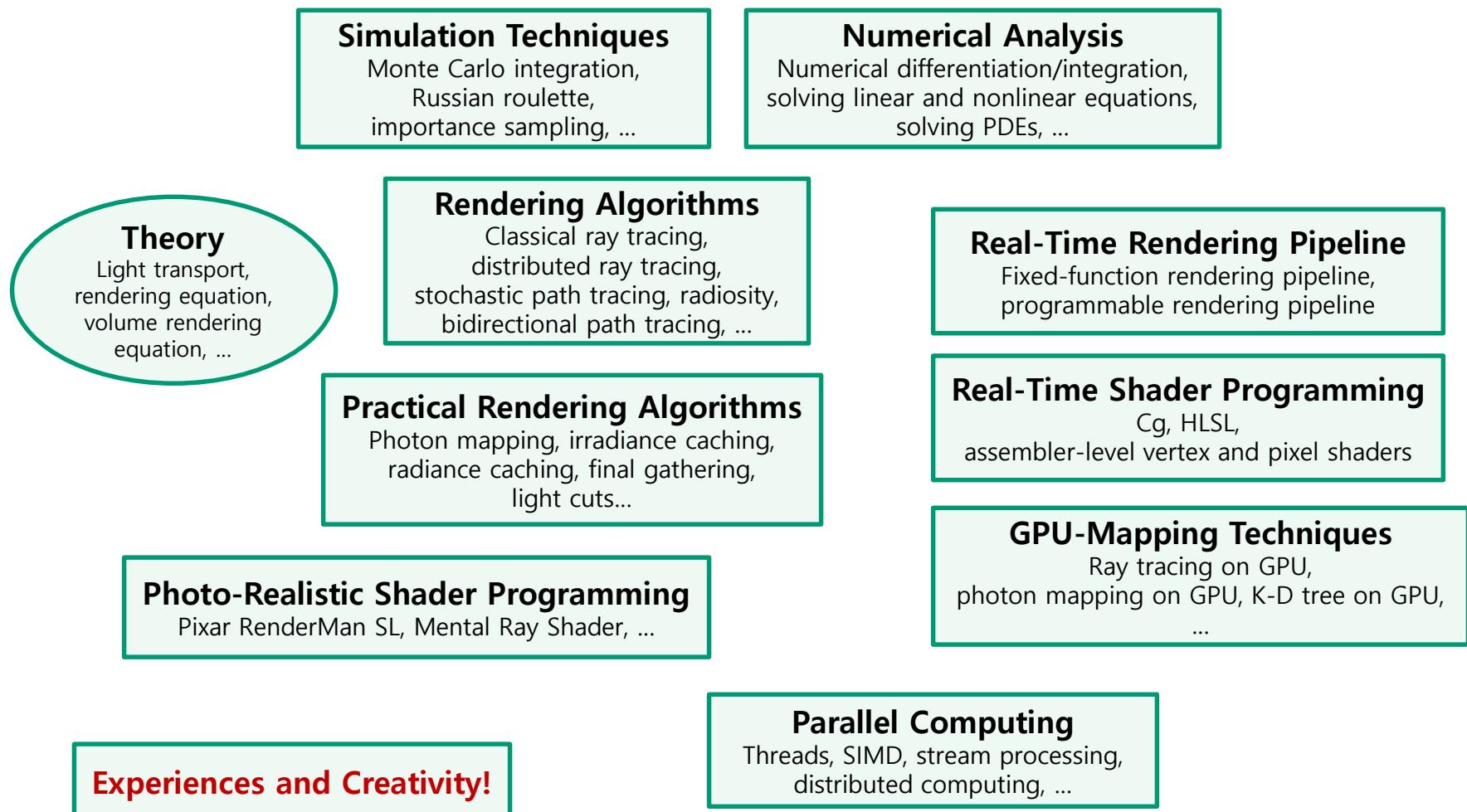
- Volume rendering equation for participating media

$$L(x, \vec{\omega}) = L(x + s\vec{\omega}, \vec{\omega}) e^{-\int_0^s \sigma_t(x + s'\vec{\omega}) ds'} + \int_0^s L_e(x + s'\vec{\omega}) e^{-\int_0^{s'} \sigma_t(x + t\vec{\omega}) dt} ds' \\ + \int_0^s \left\{ e^{-\int_0^{s'} \sigma_t(x + t\vec{\omega}) dt} \sigma_s(x + s'\vec{\omega}) \int_{\Omega_{4\pi}} p(x + s'\vec{\omega}, \vec{\omega}', \vec{\omega}) L(x + s'\vec{\omega}, \vec{\omega}') d\vec{\omega}' \right\} ds'$$

Outgoing radiance Incoming radiance
 Extinction due to absorption and out-scattering In-scattering Emission



What should we know for the photo-realistic real-time rendering?



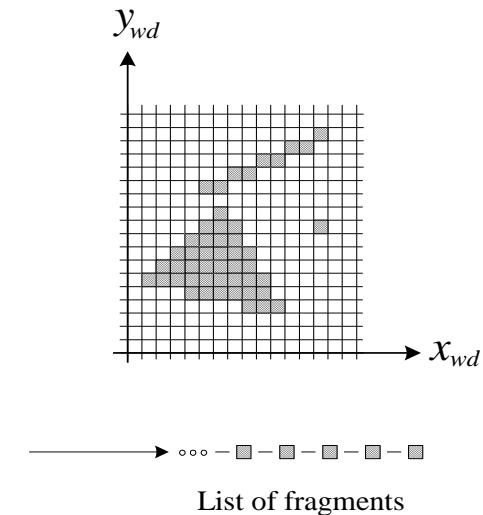
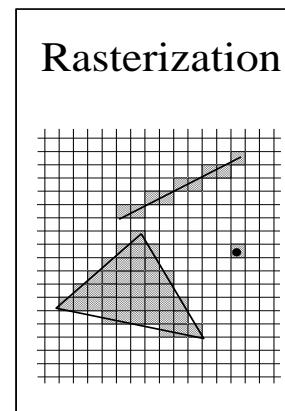
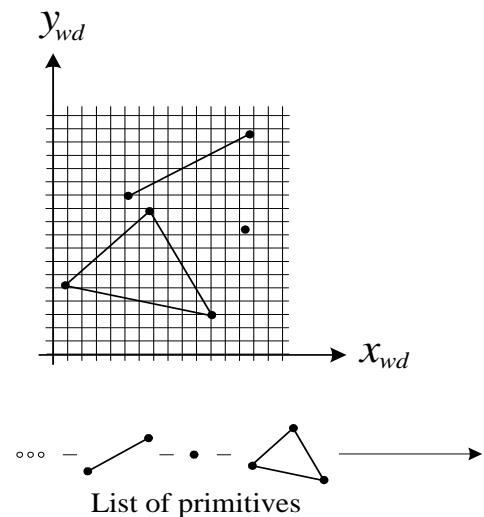
Rendering Algorithms that are Frequently Used

- **OpenGL (DirectX) rendering pipeline**
 - Object space method
 - Easily accelerated by massively parallel streaming processors
 - Not easy to generate global illumination effects
 - Usually have been used for interactive 3D graphics applications
- **Pixar REYES (Render Everything You Ever Saw) architecture**
 - Object space method
 - Not easy to generate global illumination effects
 - Usually have been used for offline 3D animations/special effects
- **Ray tracing**
 - Image space method
 - Naturally generate global illumination effects
 - Usually have been used for offline 3D animations/special effects
 - Now accelerated by massively parallel streaming processors
 - Currently being actively studied as an alternative for real-time rendering

Real-time Rendering Pipeline

- **Real-time rendering pipeline**은 rasterization 과정을 중심으로 geometry data를 처리하는 전반부와 pixel (fragment) data를 처리하는 후반부 계산으로 나누어짐.
- **Rasterization**: 점, 선분, 다각형 등 꼭지점으로 표현된 geometric primitive들을 이미지를 구성하는 pixel (fragment) 단위로 재구성하는 과정
 - Geometry data의 attribute들이 어떻게 fragment의 속성으로 전달이 될까?

To generate one image frame



Per-Primitive Operations
Per-vertex Operations

Per-Fragment Operations
Per-pixel Operations

APIs for Real-Time Rendering

- **OpenGL (Open Graphics Library) & GLSL (OpenGL Shading Language)**
 - ★ <http://www.khronos.org>
- **OpenGL ES (Open Graphics Library for Embedded System) & GLSL**
 - ★ <http://www.khronos.org>
- **Vulkan**
 - ★ <http://www.khronos.org>
- DirectX & HLSL (High Level Shading Language)
 - ★ <http://www.microsoft.com/windows/directx/>
- CUDA (Compute Unified Device Architecture)
 - ★ http://www.nvidia.com/object/cuda_home.html
- OpenCL (Open Computing Language)
 - ★ <http://www.khronos.org/opencl/>

OpenGL 4.5 Rendering Pipeline

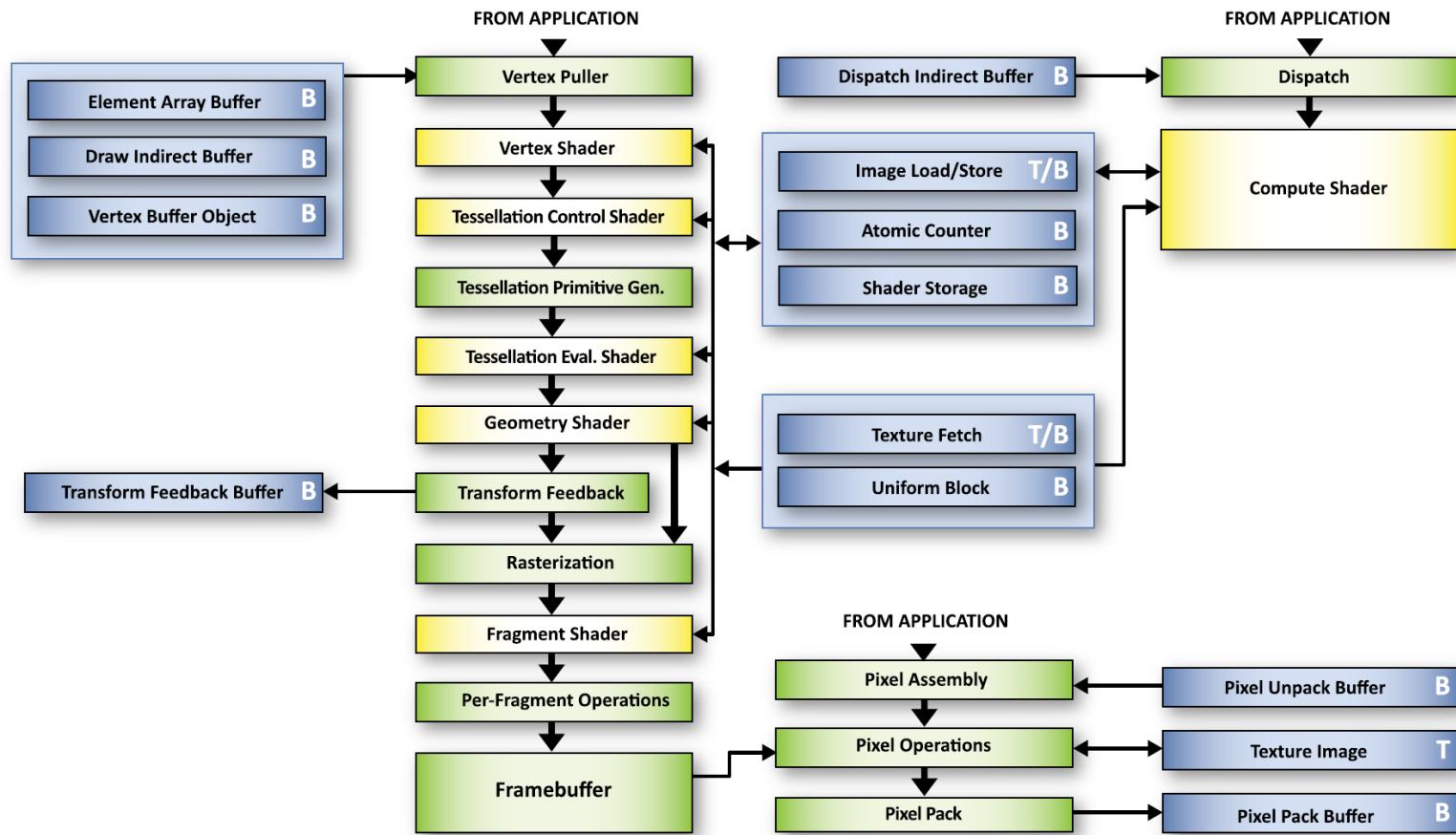
(From *OpenGL 4.5 API Reference Card*)

OpenGL Pipeline

A typical program that uses OpenGL begins with calls to open a window into the framebuffer into which the program will draw. Calls are made to allocate a GL context which is then associated with the window, then OpenGL commands can be issued.

The heavy black arrows in this illustration show the OpenGL pipeline and indicate data flow.

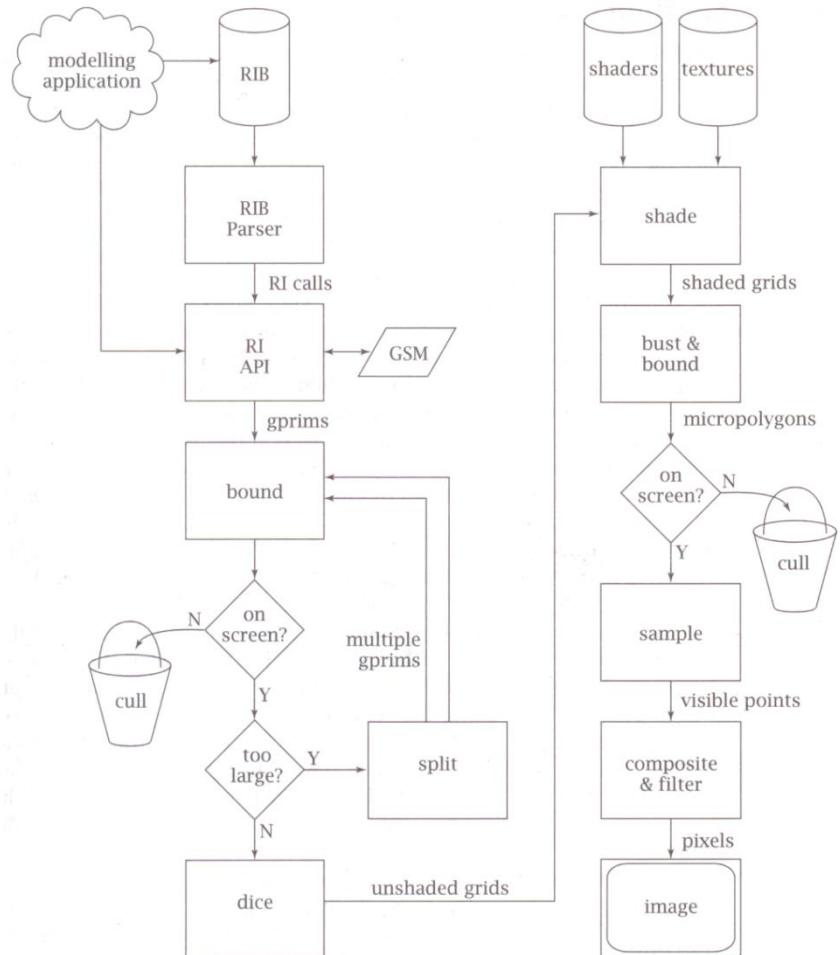
- Blue blocks indicate various buffers that feed or get fed by the OpenGL pipeline.
- Green blocks indicate fixed function stages.
- Yellow blocks indicate programmable stages.
- T Texture binding
- B Buffer binding



PIXAR REYES Architecture

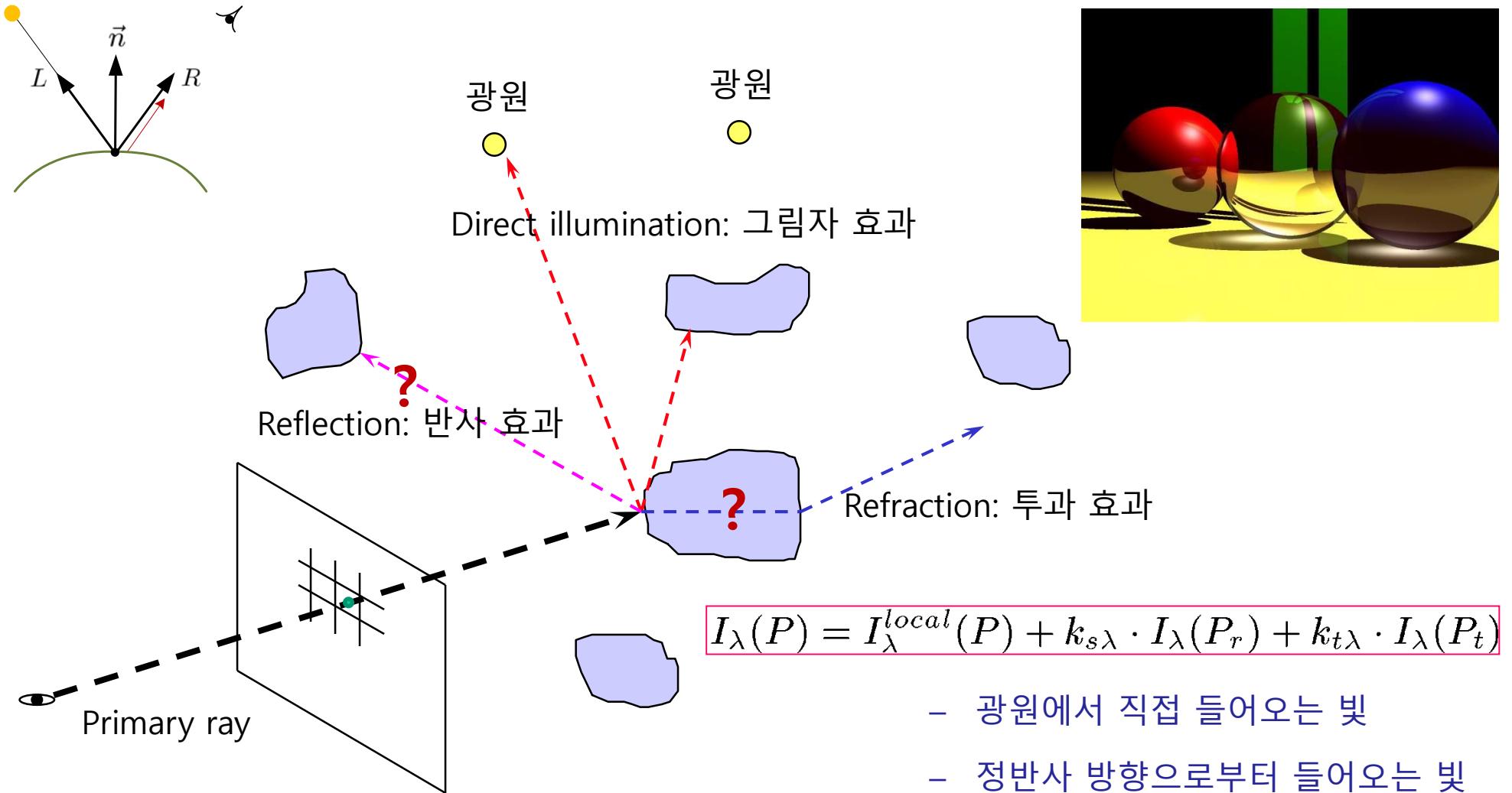
- Developed in the mid-1980s at Lucas film and in used at Pixar
- A scanline rendering algorithm
 - Splits and dices graphics primitives into micropolygons.
 - Performs a shading and visibility calculations on micropolygons.
 - Uses stochastic sampling with a Z buffer.
 - Use various mapping techniques for shading.

"The Reyes image rendering architecture" by R. Cook et a. (SIGGRAPH '87)
"Rendering antialiased shadows with depth maps" by W. Reeves (SIGGRAPH '87)



(From *Advanced RenderMan: Creating CGI for Motion Pictures* by A. Apodaca and L. Gritz)

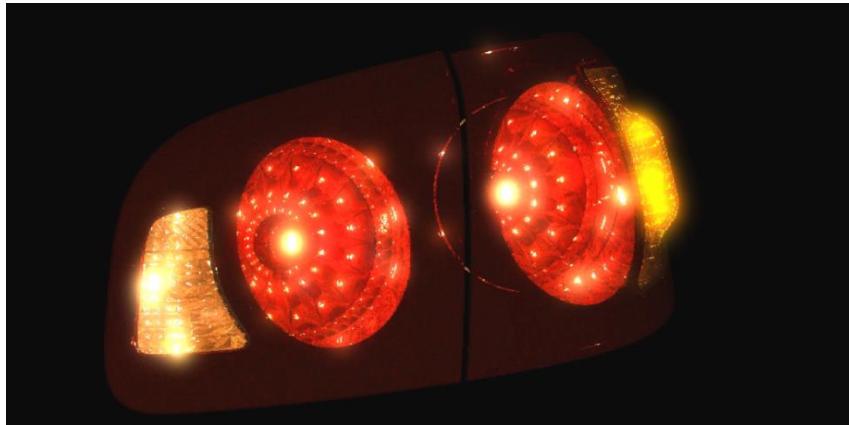
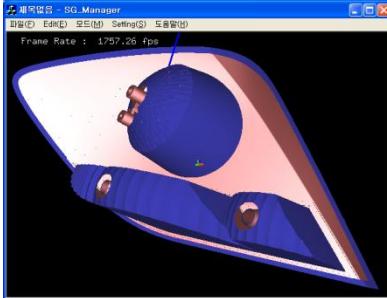
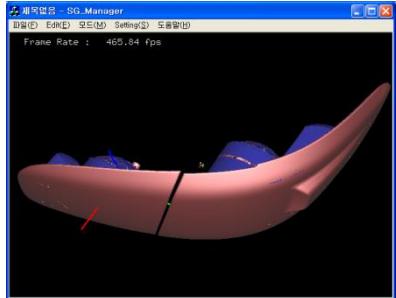
Ray Tracing



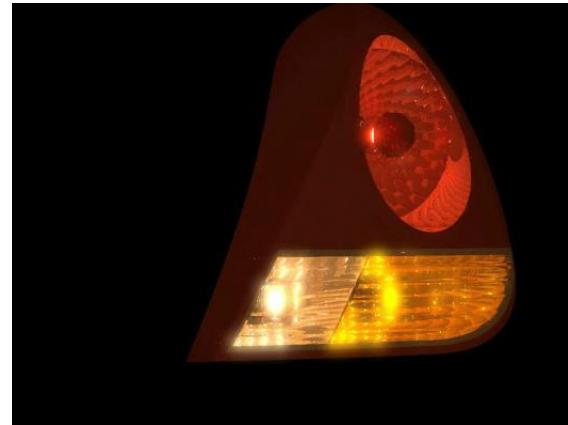
Classical (Whitted-Style) Ray Tracing

Real-Time Ray Tracing on GPU

- Example in Production Rendering



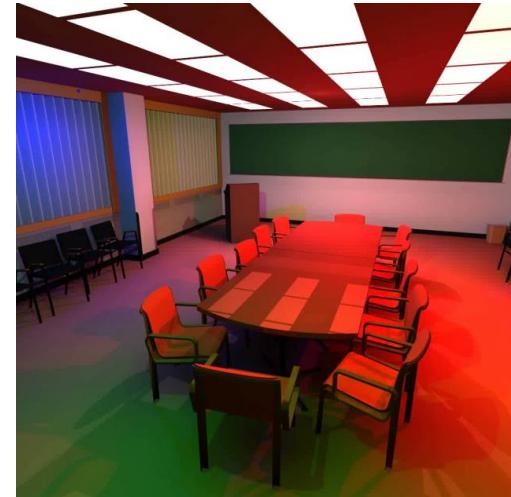
20-30 min's per image (2006)



10-15 frames per second (2010)

Real-Time/Interactive/Fast Global Illumination

- 어떤 렌더링 알고리즘을 사용하건 어떻게 하면 “고속으로” global illumination (GI) 효과를 생성할 수 있을까?



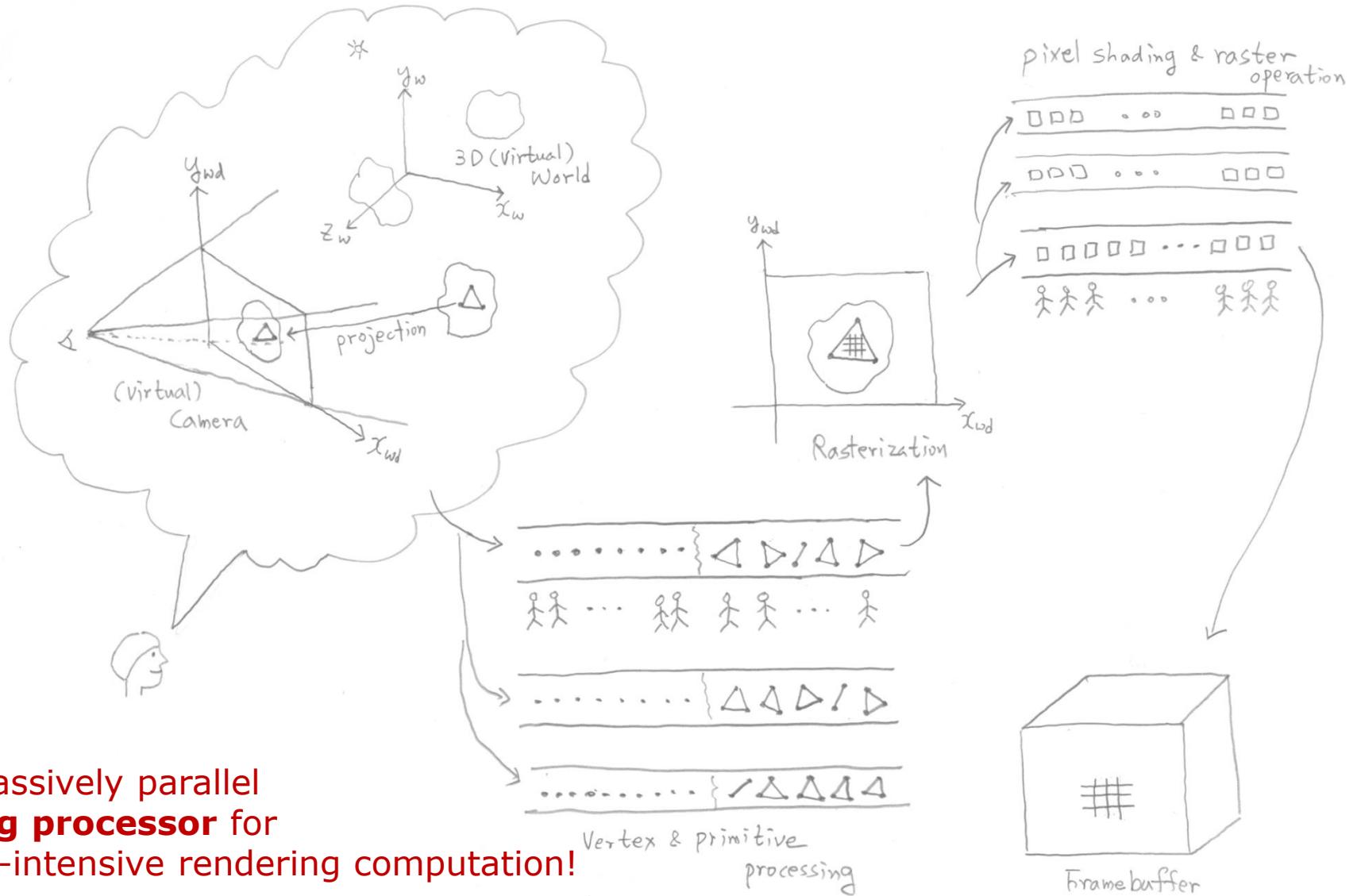
[CSE4170 기초 컴퓨터 그래픽스]

2019년도 1학기

강의자료 II

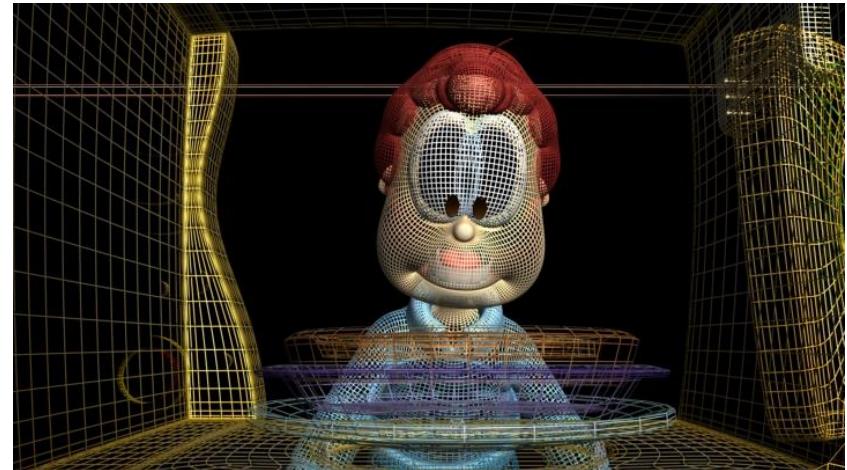
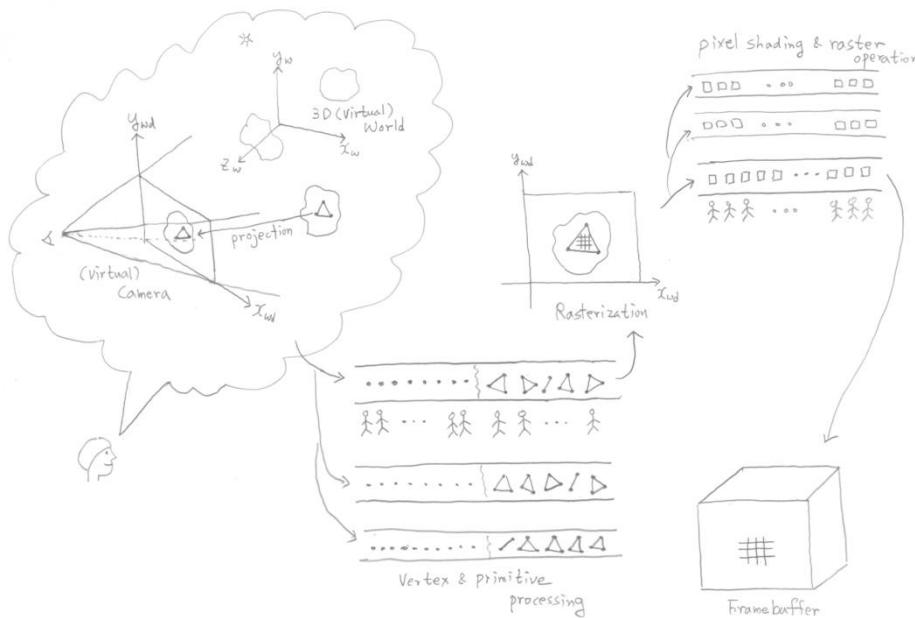
Geometry Computation in
OpenGL Rendering Pipeline

Conventional Real-Time Rendering Pipeline on GPU



Before and After Rasterization

- ✓ **래스터화(Rasterization) 과정:** 점, 선분, 다각형 등 꼭지점으로 표현된 기하 프리미티브(geometric primitive)들을 이미지를 구성하는 픽셀(프래그먼트) 단위로 재구성(분할)하는 과정
 - ✓ 이 과정 전후로 그래픽스 레이터의 형태에 근본적인 변화가 발생함.



Before Rasterization



After Rasterization

3D Rendering Pipeline: High-Level Overview

vertex stream → Vertex Shader

- **vertex stream → Primitive Assembly**
- **primitive stream → Geometry Shader**
- **primitive stream → Clipping & Setup**
- **Rasterization → fragment stream**
- **Fragment Shader → fragment stream**
- **Raster Operation → Frame Buffer**

A current view on the pipeline

- Understand **the types of streaming data:** vertex/(geometric) primitive vs fragment(pixel).
- Understand the **user-programmable shader parts:** vertex/geometry/ fragment (pixel)/compute.
- Understand the still **fixed-function parts:** primitive assembly, clipping & setup, raster operation, ...

1. Application/Scene

- Scene/Geometry database traversal
- Movement of objects, and aiming and movement of view camera
- Animated movement of object models
- Description of the contents of the 3D world
- Object Visibility Check including possible Occlusion Culling
- Select Level of Detail (LOD)

2. Geometry

- Transforms (rotation, translation, scaling)
- Transform from Model Space to World Space (Direct3D)
- Transform from World Space to View Space
- View Projection
- Trivial Accept/Reject Culling
- Back-Face Culling (can also be done later in Screen Space)
- Lighting
- Perspective Divide - Transform to Clip Space
- Clipping
- Transform to Screen Space

3. Triangle Setup

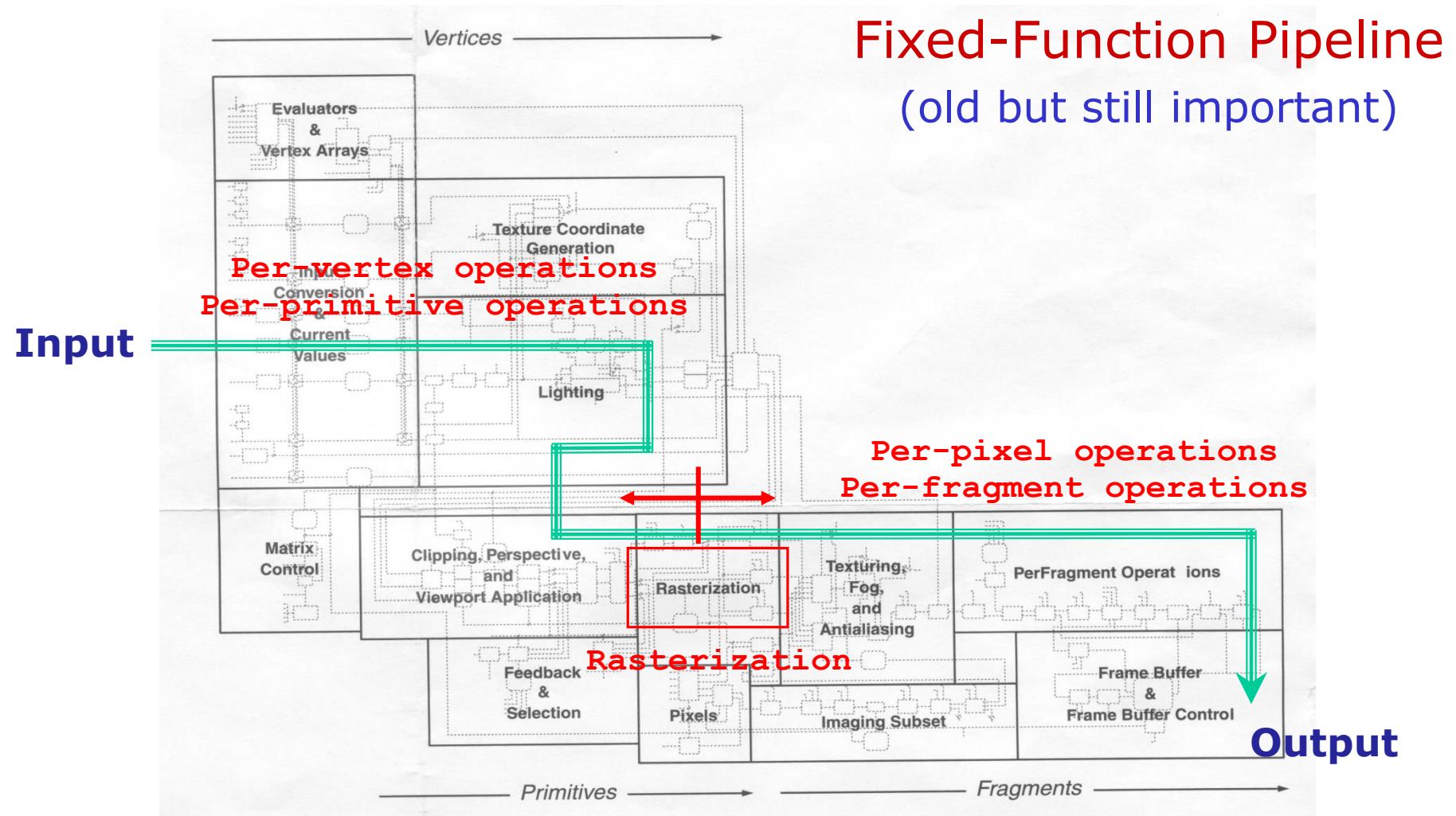
- Back-face Culling (or can be done in view space before lighting)
- Slope/Delta Calculations
- Scan-Line Conversion

4. Rendering / Rasterization

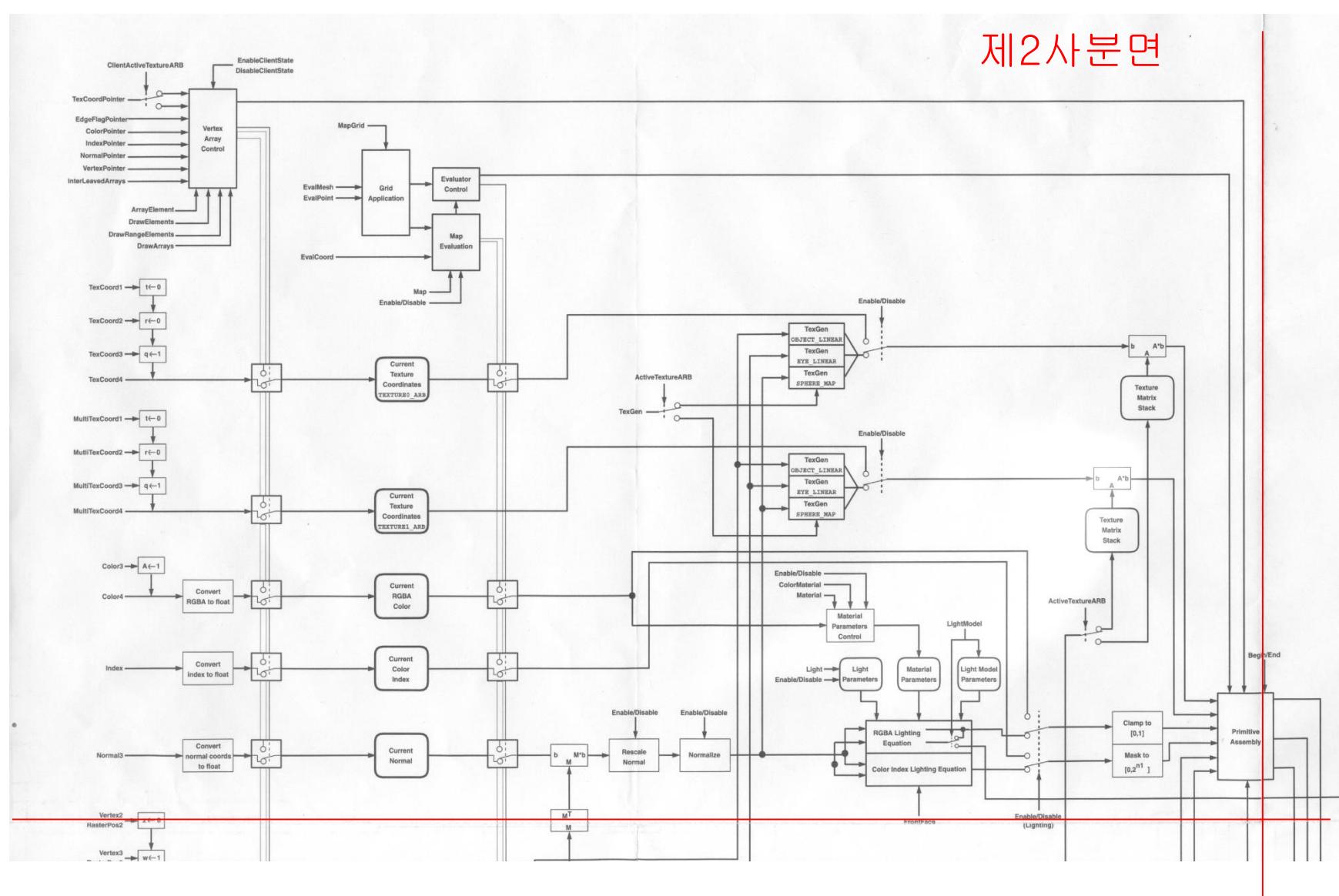
- Shading
- Texturing
- Fog
- Alpha Transparency Tests
- Depth Buffering
- Antialiasing (optional)
- Display

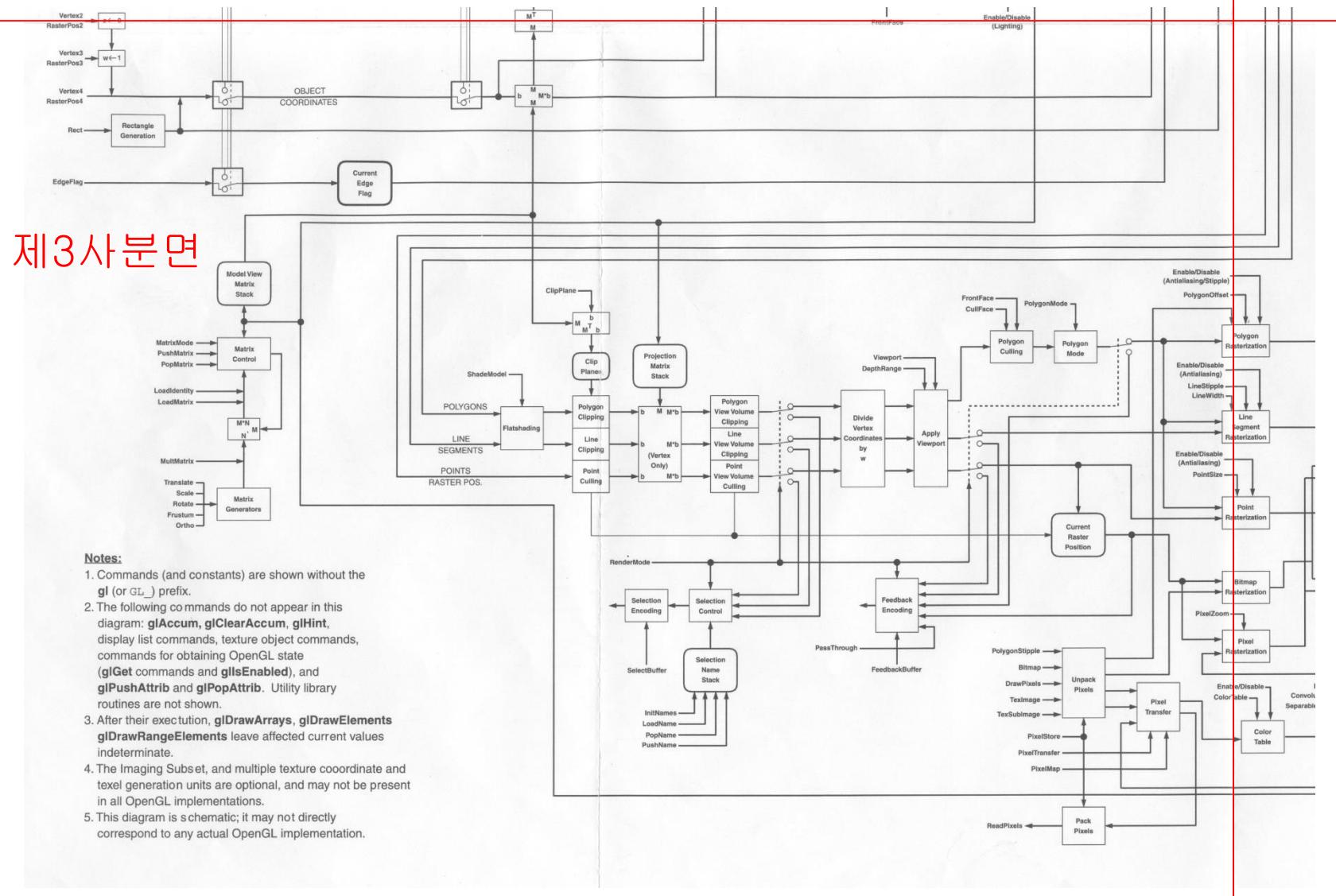
<http://www.extremetech.com/article2/0,2845,1154772,00.asp>

Example: OpenGL Architecture (Version 1.2)

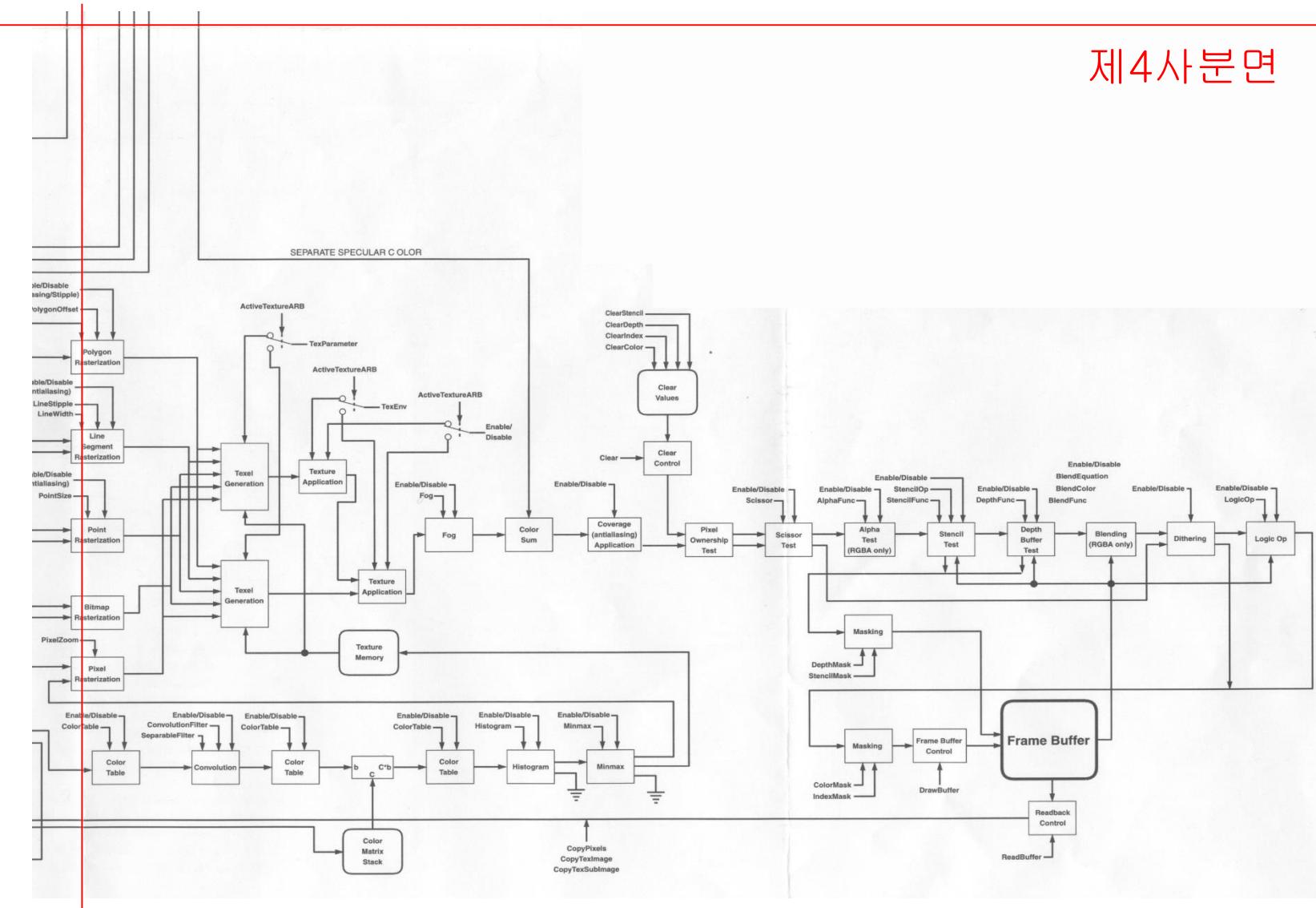


제2사분면

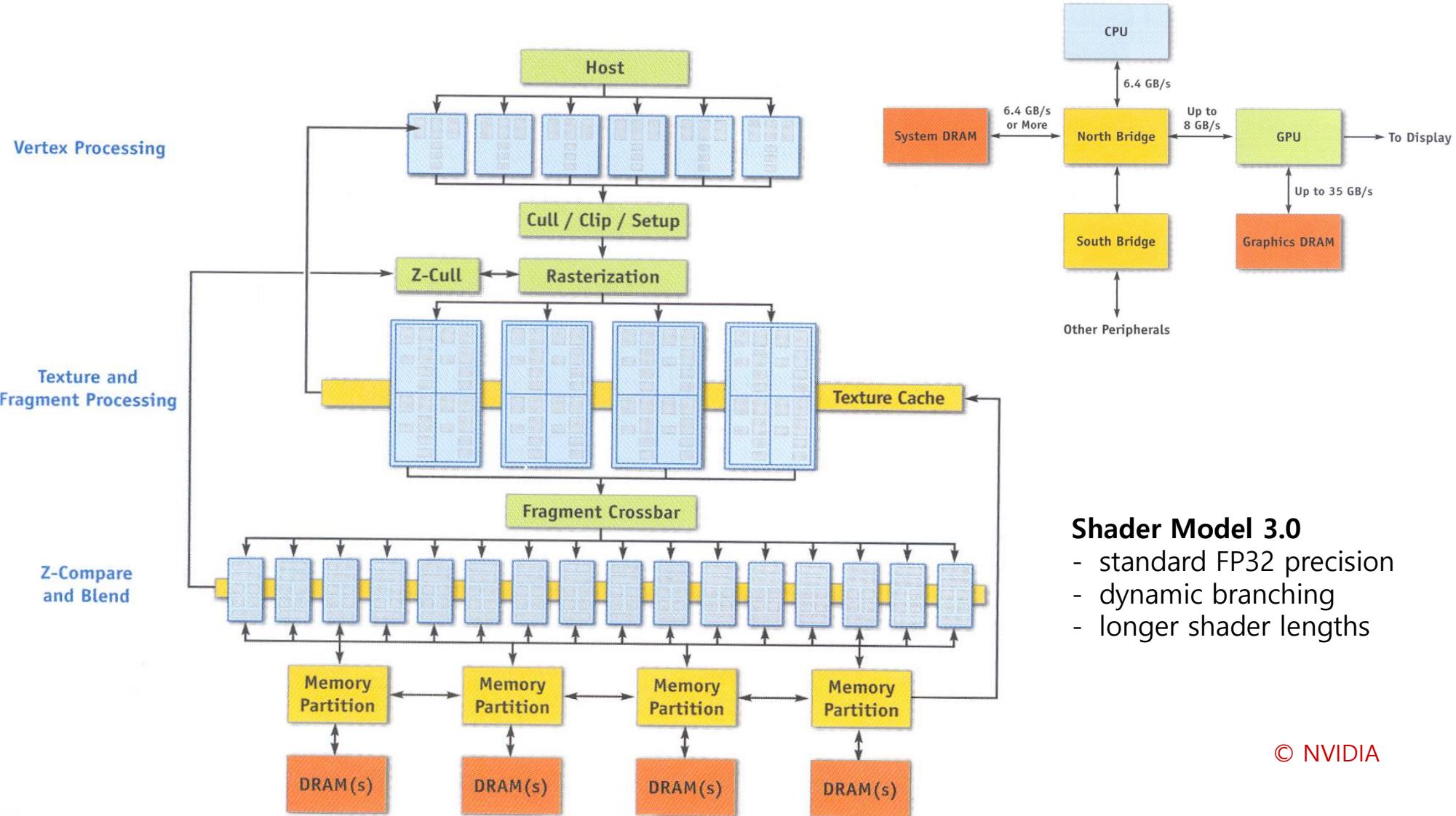




제4사분면

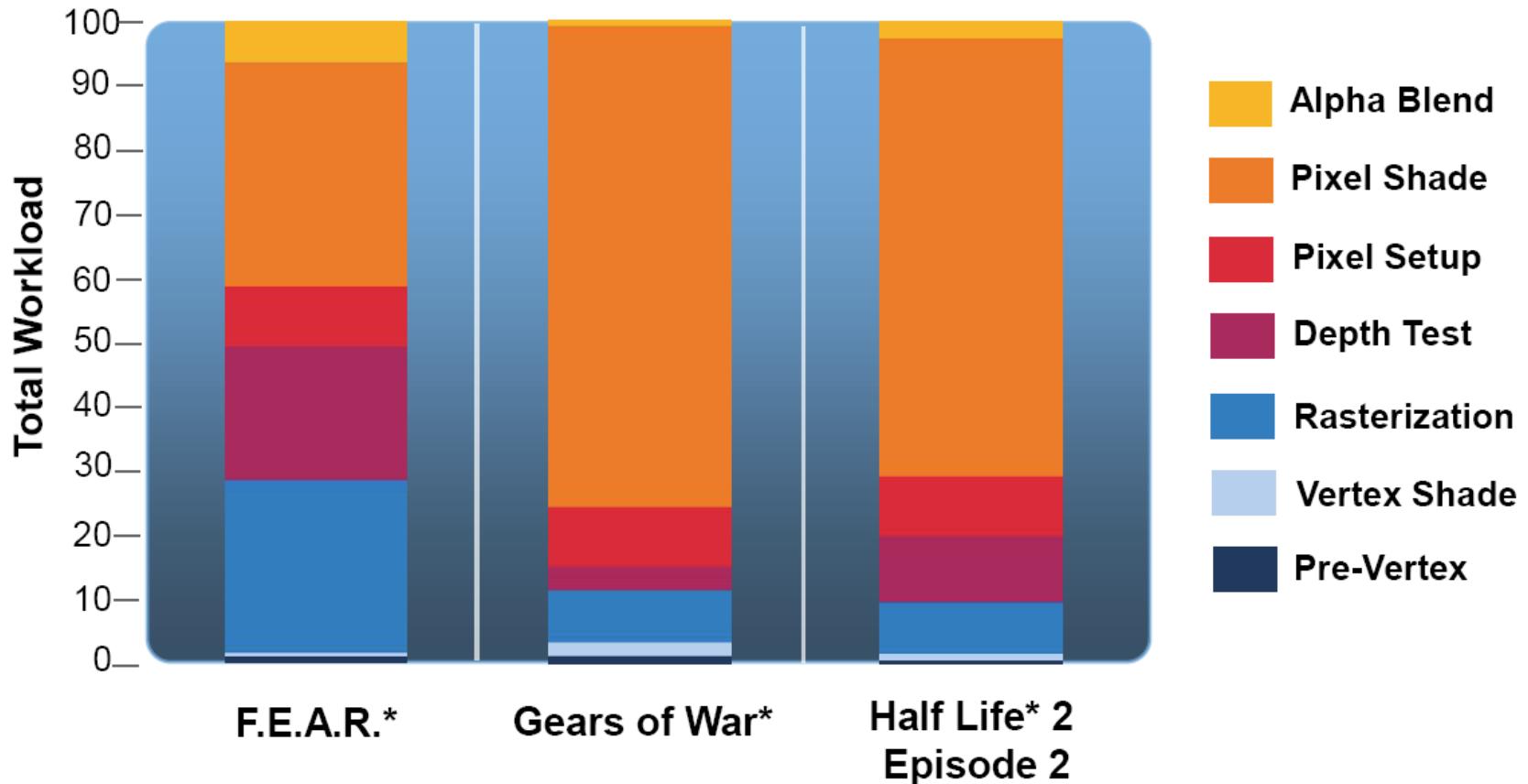


Example: GeForce 6 Series (NV40) - 6800 Ultra (2004)



Example: Dissection of the Rendering Computation

과연 각 단계가 application, geometry, 그리고 rasterizer stage 중 어느 단계에 해당할까?

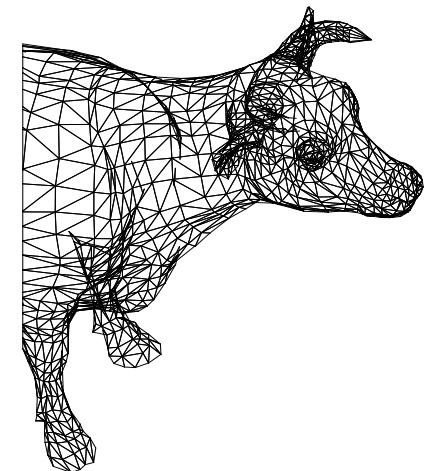


From Seiler et al., "Larrabee: A many-core x86 architecture for visual computing," SIGGRAPH 2008.

Input and Output of 3D Rendering Pipeline

- **Input: scene description**

- Polygonal models
 - **Primitives:** points, line segments, polygons(triangles)
 - Per-vertex attributes
 - » Position (x, y, z, 1), Normal (x, y, z, 0), Color (r, g, b, a), Texture coordinates (s, t, r, q), and any vertex properties whatever they are.
 - Type of primitive
- Camera
 - Position, orientation, field of view, etc.
- Shading parameters
 - Material properties, light properties, shading model, etc.
- Textures
- Etc.



- **Output: raster image**

Rendering Pipeline의 부하 분석 예

Why unify?



Heavy Geometry
Workload Perf = 4

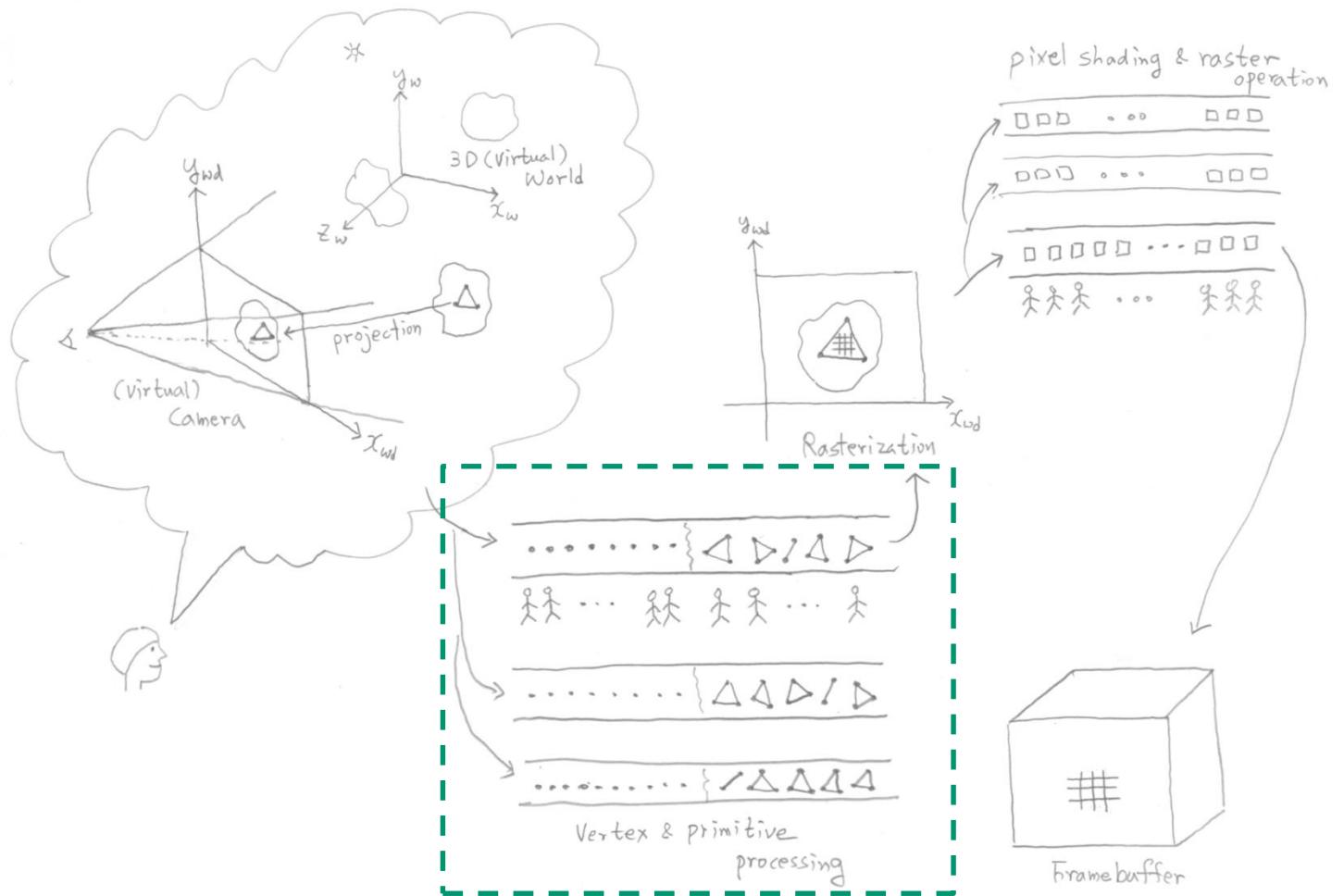


Heavy Pixel
Workload Perf = 8

Figure 15. Fixed shader performance characteristics

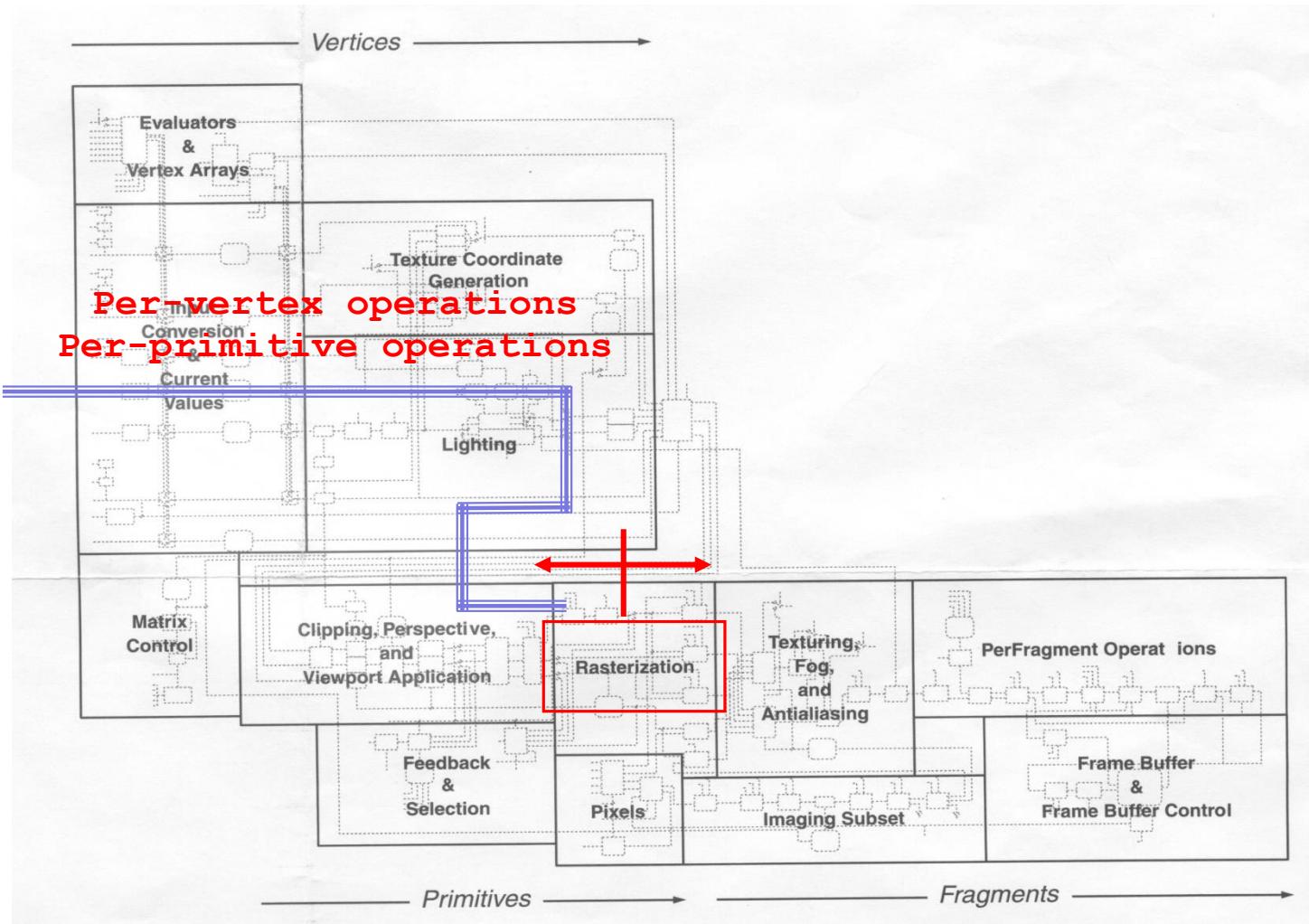
From Technical Brief: NVIDIA GeForce 8800 GPU Architecture Overview, 2006.

OpenGL Geometry Pipeline



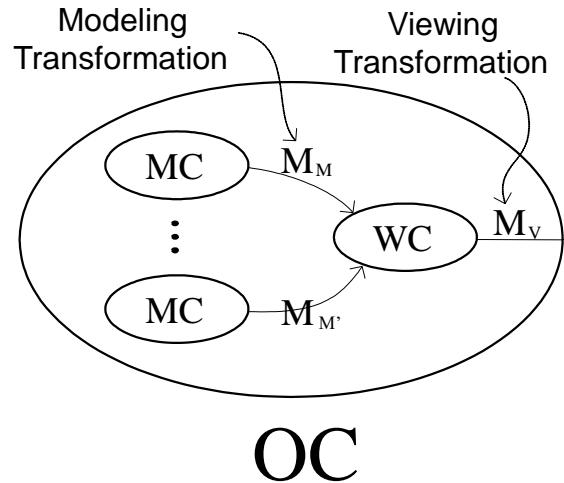
Geometry Stage in OpenGL System (Fixed-Function)

Polygonal
models



Geometry Stage in Fixed-Function OpenGL Pipeline

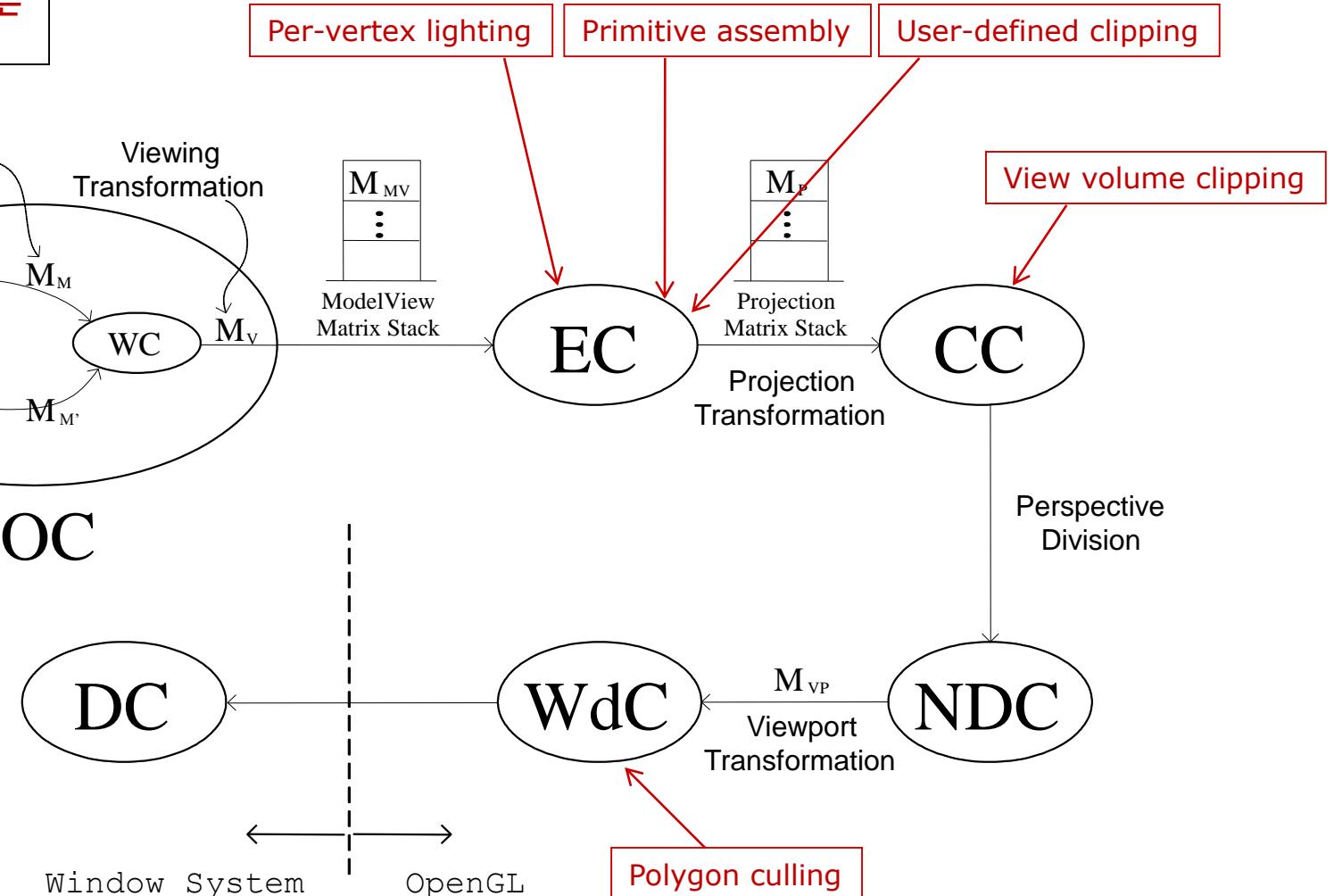
Geometry stage의 골격을 이루는 geometry pipeline을 이해하자.



Per-vertex lighting

Primitive assembly

User-defined clipping

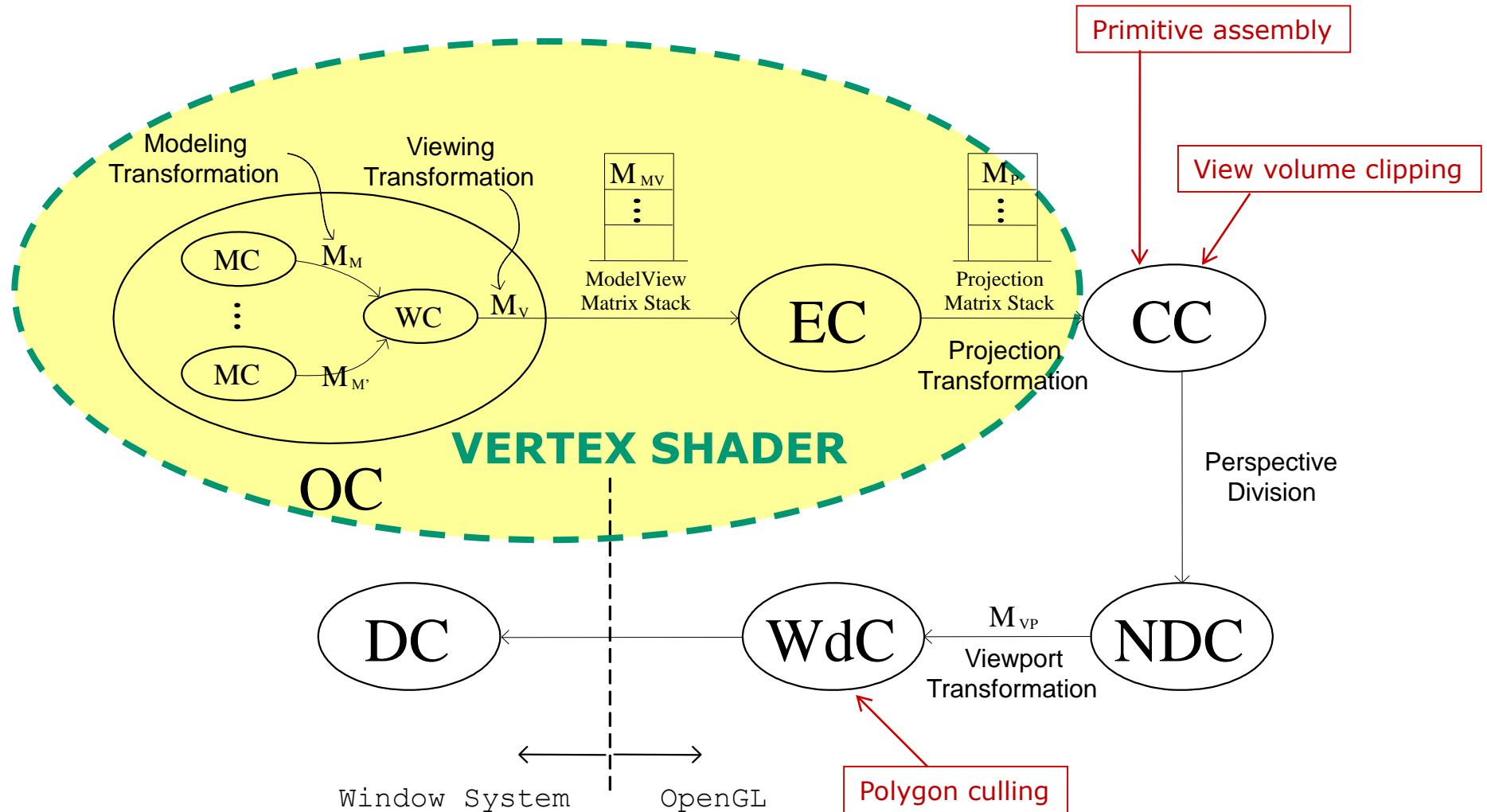


Basic computations in geometry stage

- Model and view transform
- **Lighting**
- **Primitive assembly**
- Projection transform
- **View volume clipping**
- Screen mapping
- **Polygon culling**
- Etc.

최신 GPU에서는 geometry stage의 어느 부분이 vertex shader 및 geometry shader 형태로 프로그램이 가능할까?

Geometry Stage in Programmable OpenGL Pipeline

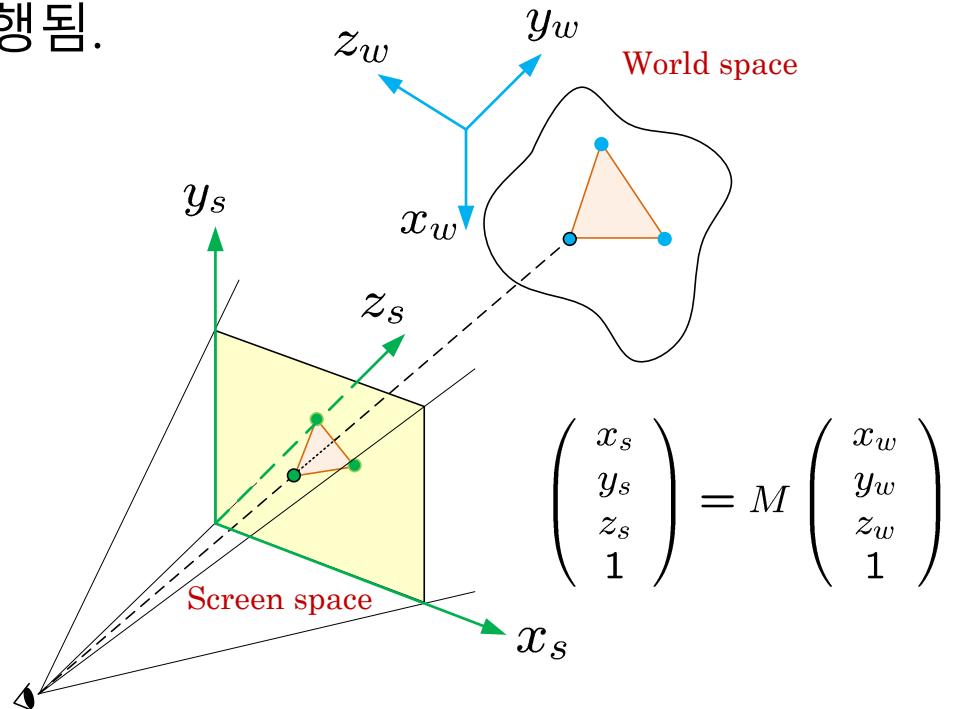


Geometry stage의 골격을 이루는
geometry pipeline을 이해하자.

당분간 vertex shader에 집중할 예정임.

3D Viewing

- 가상의 세상에 카메라 인자를 설정하였을 때, 물체를 구성하는 **3차원 공간의 점이 2차원 평면인 화면의 어느 지점으로 투영이 되는지 결정**해주는 계산 과정으로, **geometry stage**의 중추를 이룸.
- 3차원 뷰잉(3D viewing)**이라고 하는 3차원 공간에서 2차원 공간으로의 기하 변환(geometric transform) 계산이 수행됨.

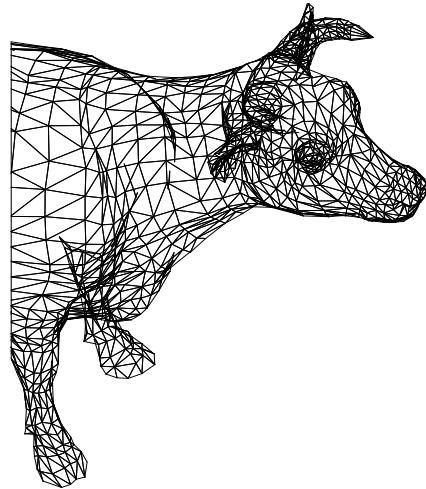


Vertex, Geometric Primitives, and Polygonal Model

Vertex and its Attributes

- 기하 물체를 구성하는 꼭지점(vertex)의 여러 성질을 기술함.
 - Conventional vertex attributes
 - 위치, 법선 벡터, 색깔, 텍스춰 좌표, ...

☞ 예1: NVIDIA NV30



Vertex Attribute Register Number	Conventional Per-vertex Parameter	Conventional Per-vertex Parameter Command	Conventional Component Mapping
0 (OPOS)	vertex position	Vertex	x, y, z, w
1 (WGHT)	vertex weights	VertexWeightEXT	w, 0, 0, 1
2 (NRML)	normal	Normal	x, y, z, 1
3 (COLO)	primary color	Color	r, g, b, a
4 (COL1)	secondary color	SecondaryColorEXT	r, g, b, 1
5 (FOGC)	fog coordinate	FogCoordEXT	fc, 0, 0, 1
6	-	-	-
7	-	-	-
8 (TEX0)	texture coord 0	MultiTexCoord(GL_TEXTURE0_ARB, ...)	s, t, r, q
9 (TEX1)	texture coord 1	MultiTexCoord(GL_TEXTURE1_ARB, ...)	s, t, r, q
10 (TEX2)	texture coord 2	MultiTexCoord(GL_TEXTURE2_ARB, ...)	s, t, r, q
11 (TEX3)	texture coord 3	MultiTexCoord(GL_TEXTURE3_ARB, ...)	s, t, r, q
12 (TEX4)	texture coord 4	MultiTexCoord(GL_TEXTURE4_ARB, ...)	s, t, r, q
13 (TEX5)	texture coord 5	MultiTexCoord(GL_TEXTURE5_ARB, ...)	s, t, r, q
14 (TEX6)	texture coord 6	MultiTexCoord(GL_TEXTURE6_ARB, ...)	s, t, r, q
15 (TEX7)	texture coord 7	MultiTexCoord(GL_TEXTURE7_ARB, ...)	s, t, r, q



예2: NVIDIA G80

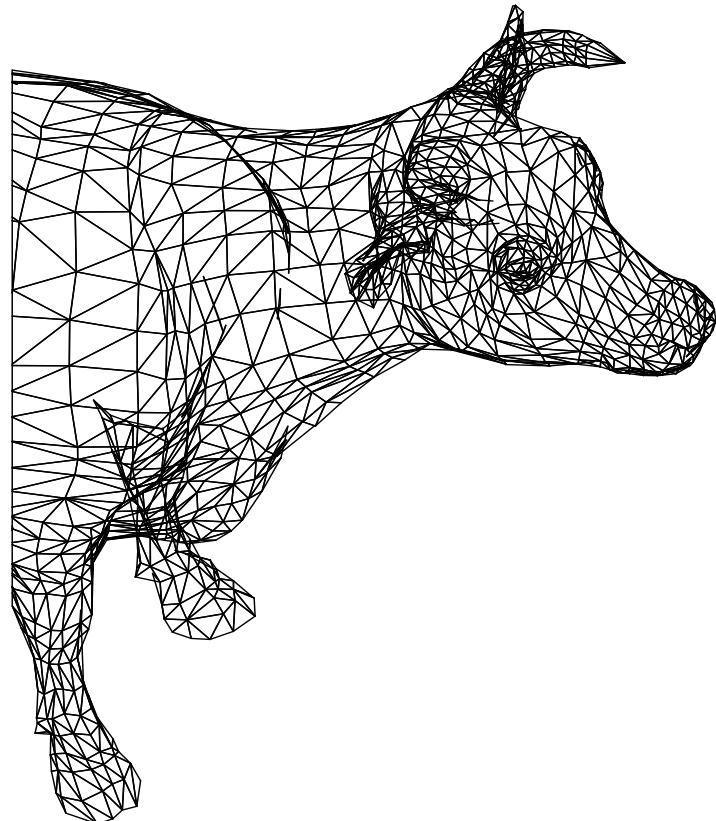
Vertex Attribute Binding	Components	Underlying State
vertex.position	(x,y,z,w)	object coordinates
vertex.normal	(x,y,z,1)	normal
vertex.color	(r,g,b,a)	primary color
vertex.color.primary	(r,g,b,a)	primary color
vertex.color.secondary	(r,g,b,a)	secondary color
vertex.fogcoord	(f,0,0,1)	fog coordinate
vertex.texcoord	(s,t,r,q)	texture coordinate, unit 0
vertex.texcoord[n]	(s,t,r,q)	texture coordinate, unit n
vertex.attrib[n]	(x,y,z,w)	generic vertex attribute n
vertex.id	(id,-,-,-)	vertex identifier (integer)
vertex.instance	(i,-,-,-)	primitive instance number (integer)
vertex.texcoord[n..o]	(x,y,z,w)	array of texture coordinates
vertex.attrib[n..o]	(x,y,z,w)	array of generic vertex attributes

- User-defined attributes

- 위의 전통적인 vertex attribute 외에 shader 기반의 렌더링 파이프라인에서는 (필요 시)
“어떠한 부류”의 속성도 각 꼭지점에 설정할 수 있음.

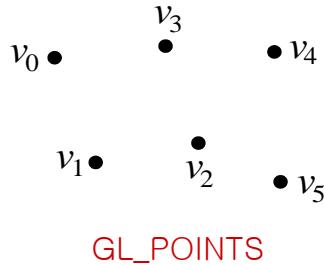
Geometric Primitives

- 가상의 공간을 구성하는 각 물체를 표현하기 위하여 어떠한 base primitive를 사용할 것인가?
 - 실시간 그래픽스에서는 주로 가장 단순한 형태의 표현 방법인 **선형 프리미티브 (linear primitive)**들을 사용함.
 - 점 (point)
 - 선분 (line)
 - 삼각형 (triangle)
- 실시간 렌더링 파이프라인에서 사용되는 주요 geometric primitive들에 대해 조사해보자.

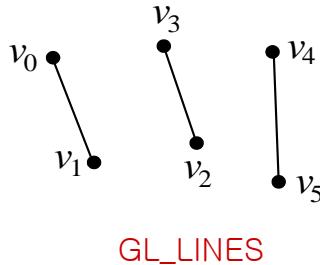


OpenGL 지원 Geometric Primitive 예

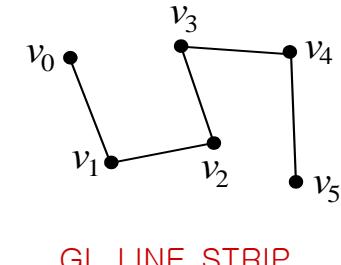
- 방식: vertex를 나열하면 사용자가 설정한 모드의 방식으로 geometric primitive로 primitive assembly가 됨.



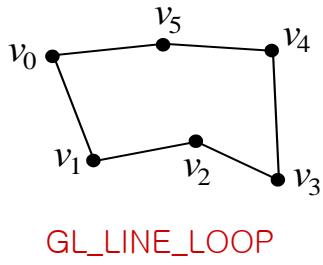
GL_POINTS



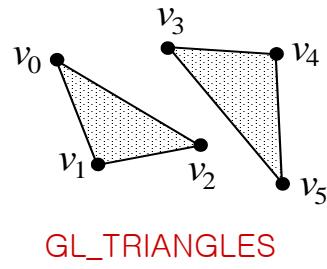
GL_LINES



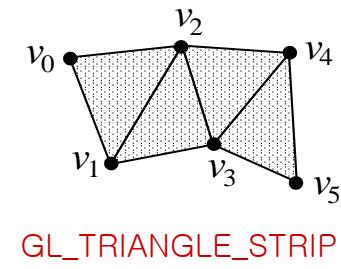
GL_LINE_STRIP



GL_LINE_LOOP

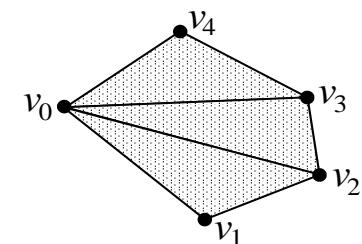


GL_TRIANGLES



GL_TRIANGLE_STRIP

GL_POINTS, **GL_LINE_STRIP**, **GL_LINE_LOOP**, **GL_LINES**,
GL_LINE_STRIP_ADJACENCY, **GL_LINES_ADJACENCY**,
GL_TRIANGLE_STRIP, **GL_TRIANGLE_FAN**,
GL_TRIANGLES, **GL_TRIANGLE_STRIP_ADJACENCY**,
GL_TRIANGLES_ADJACENCY, **GL_PATCHES**



GL_TRIANGLE_FAN

Feeding Vertex Stream into OpenGL Pipeline

(Fixed-Function Pipeline)

glVertex*() and glBegin(*)/glEnd() Functions

- **glVertex*()** 함수 void glVertex{234}{sifd}[v](TYPE coords);

- 꼭지점의 좌표(position)을 설정하는 함수로서, 현재까지 설정되어 있는 다른 꼭지점 속성들과 함께 한 개의 꼭지점을 생성함.

Vertex Attribute Register Number	Conventional Per-vertex Parameter	Conventional Per-vertex Parameter Command	Conventional Component Mapping
0 (OPOS)	vertex position	Vertex	x, y, z, w
1 (WGHT)	vertex weights	VertexWeightEXT	w, 0, 0, 1
2 (NRML)	normal	Normal	x, y, z, 1
3 (COLO)	primary color	Color	r, g, b, a
4 (COL1)	secondary color	SecondaryColorEXT	r, g, b, 1
5 (FOGC)	fog coordinate	FogCoordEXT	fc, 0, 0, 1
6	-	-	-
7	-	-	-
8 (TEX0)	texture coord 0	MultiTexCoord(GL_TEXTURE0_ARB, ...)	s, t, r, q
9 (TEX1)	texture coord 1	MultiTexCoord(GL_TEXTURE1_ARB, ...)	- - - -

```
glTexCoord2f(0.2f, 0.5f);
glNormal3f(-1.0f, 0.0f, 0.0f);
glVertex3f(3.1, 9.0, -1.4);
```

```
glTexCoord2f(0.23f, 0.42f);
glNormal3f(-0.9f, 0.435f, 0.0f);
glVertex3f(3.72, 8.5, -1.12);
```

...



OPOS	3.1, 9.0, -1.4
NRML	-1.0, 0.0, 0.0
TEX0	0.2, 0.5



OPOS	3.72, 8.5, -1.12
NRML	-0.9, 0.435, 0.0
TEX0	0.23, 0.42

- **glBegin() / glEnd()** 함수

- glVertex*() 함수를 반복적으로 호출하여 vertex stream 생성함.
- glBegin(*) 함수의 인자에 따라 primitive assembly 과정에서 vertex stream의 꼭지점들을 어떻게 조합하여 geometric primitive들을 생성할 지가 결정됨.

```
glBegin(GL_LINES);  
    glVertex3f(3.0, 2.5, -1.5); // v0  
    glVertex3f(3.0, 2.1, -1.2); // v1  
    glVertex3f(2.9, 1.5, -1.0); // v2  
    glVertex3f(2.5, 1.5, -1.5); // v3  
    :  
    glEnd();
```

꼭지점 기술 예: 고전적인 방법

```
void draw_box_frame(void) {
    glDisable(GL_LIGHTING);

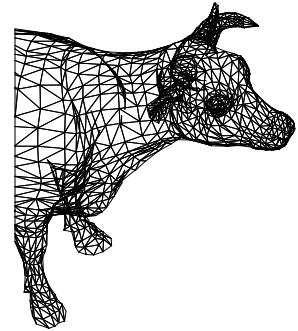
    glPolygonMode(GL_FRONT_AND_BACK, GL_LINE);
    glColor3f(1.0, 0.0, 0.0);

    glMatrixMode(GL_MODELVIEW);
    glPushMatrix();
    glTranslatef(-x_center, -y_center, -z_center);
    glScalef(bbox, bbox, bbox);
    glBegin(GL_QUADS);
        glVertex3f(1.5, 1.5, 1.5); glVertex3f(-1.5, 1.5, 1.5); glVertex3f(-1.5, 1.5, -1.5); glVertex3f(1.5, 1.5, -1.5);
        glVertex3f(1.5, 1.5, 1.5); glVertex3f(1.5, 1.5, -1.5); glVertex3f(1.5, -1.5, -1.5); glVertex3f(1.5, -1.5, 1.5);
        glVertex3f(1.5, 1.5, -1.5); glVertex3f(-1.5, 1.5, -1.5); glVertex3f(-1.5, -1.5, -1.5); glVertex3f(1.5, -1.5, -1.5);
        glVertex3f(-1.5, 1.5, -1.5); glVertex3f(-1.5, 1.5, 1.5); glVertex3f(-1.5, -1.5, 1.5); glVertex3f(-1.5, -1.5, -1.5);
        glVertex3f(1.5, 1.5, 1.5); glVertex3f(1.5, -1.5, 1.5); glVertex3f(-1.5, -1.5, 1.5); glVertex3f(-1.5, 1.5, 1.5);
        glVertex3f(1.5, -1.5, 1.5); glVertex3f(1.5, -1.5, -1.5); glVertex3f(-1.5, -1.5, -1.5); glVertex3f(-1.5, -1.5, 1.5);
    glEnd();
    glPopMatrix();
    glEnable(GL_LIGHTING);
}
```

```
for (k = 0; k < num_cows; k++) {
    glPushMatrix();
    glTranslatef(pos[k][0], pos[k][1], pos[k][2]);
    for (i = 0; i < npoly; i++) {
        glBegin(GL_POLYGON);
        for (j = 0; j < object[i].nvertex; j++) {
            glNormal3fv(object[i].normal[j]);
            glVertex3fv(object[i].vertex[j])
        }
        glEnd();
    }
    glPopMatrix();
}
```

- OpenGL에서의 다각형에 대한 요구 사항
 - 선분들이 서로 교차하면 안됨.
 - 볼록 다각형만 사용해야 함.
 - 꼭지점들이 한 평면상에 존재해야 함.
- ◆ 다각형으로서 일반적으로 이러한 조건을 만족시키는 삼각형을 사용함 → triangular mesh

Feeding Vertex Stream into OpenGL Pipeline (Programmable Pipeline)



How to Draw Virtual World in OpenGL

- 보편적인 OpenGL 렌더링 과정

- ① OpenGL 함수를 사용하여 렌더링에 필요한 여러 가지 인자들을 설정함.
 - ✓ OpenGL은 **state machine**임.
- ② Drawing command인 glDrawArrays() 또는 glDrawElements() 함수와 같은 rendering command를 사용하여 렌더링을 구동하면, ①에서 설정한 인자들을 사용하여 렌더링 계산을 한 후 결과를 framebuffer에 저장함.
 - ✓ 다른 rendering commands에 대해서도 알아보자.

- 기하 물체의 설정 과정

- 위의 ①번 과정에서 가장 중요한 작업 중의 하나는 렌더링을 하고자 하는 3차원 세상을 구성하는 개개의 기하 물체를 설정하는 것임.
- ① 각 기하 물체를 구성하는 꼭지점들을 하나씩 나열하는데 필요한 정보를 제공함.
 - Vertex attribute 정보를 어떤 메모리 영역에서 어떻게 가져올 지에 대한 정보 설정
- ② 다음 꼭지점들을 어떻게 조합하여 geometric primitive(point, line segment, triangle) 형태로 primitive assembly할 지에 대한 방식을 설정함.
 - glDrawArrays() 또는 glDrawElements() 등의 drawing command 내에서 꼭지점을 몇 개 사용할 지와 어떠한 형태의 geometric primitive들로 조립할 지 모드를 설정함.

Vertex Shader and Primitive Assembly

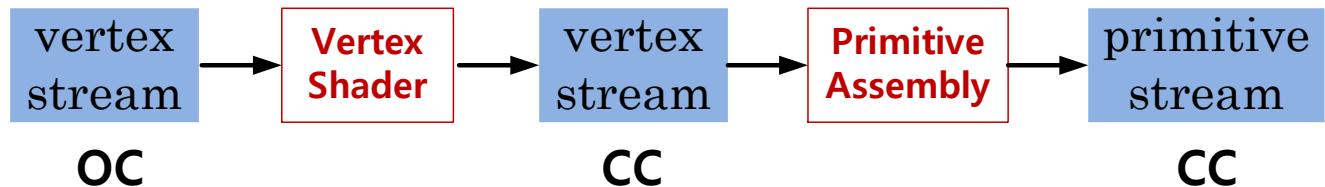
glDrawArrays() 함수와 같은 drawing command로 렌더링을 구동하면, 먼저 기하 물체를 구성하는 각 꼭지점들에 대해 독립적으로 vertex shader가 수행되며, 이후 primitive assembly 과정을 통해 geometric primitive로 조립이 되어 렌더링 계산이 진행됨.

- **Vertex Shader** (from Wikipedia)

- Handles the processing of individual vertices by altering their attributes.
- Receives a single vertex from the vertex stream and generates a single vertex to the output vertex stream: **one-vertex-in/one-vertex_out**.
- **The most essential task of vertex shader** is to transform the coordinate of vertex from Object Coordinate(OC) to Clip Coordinate(CC).

- **Primitive Assembly** (from Wikipedia)

- The stage in the OpenGL rendering pipeline where **a stream of vertices is divided into a sequence of individual base primitives**.
- After **some minor processing**, they are passed along to the rasterizer to be rendered.



Example of Vertex Shader

- Vertex shader 예

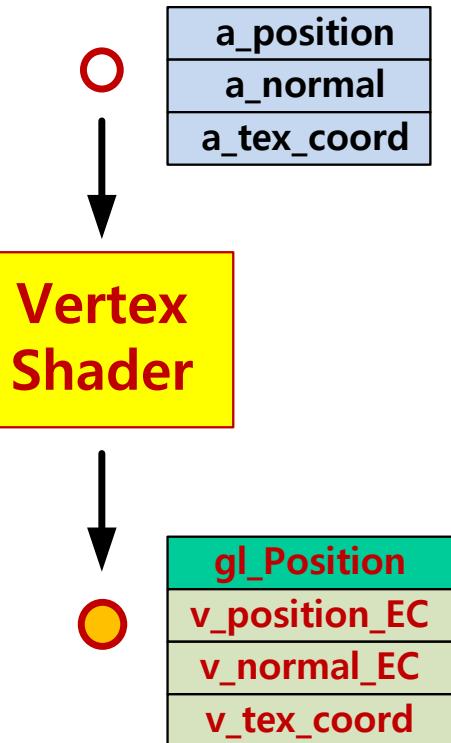
```
#version 330

uniform mat4 u_ModelViewProjectionMatrix;
...
layout (location = 0) in vec3 a_position;
layout (location = 1) in vec3 a_normal;
layout (location = 2) in vec2 a_tex_coord;

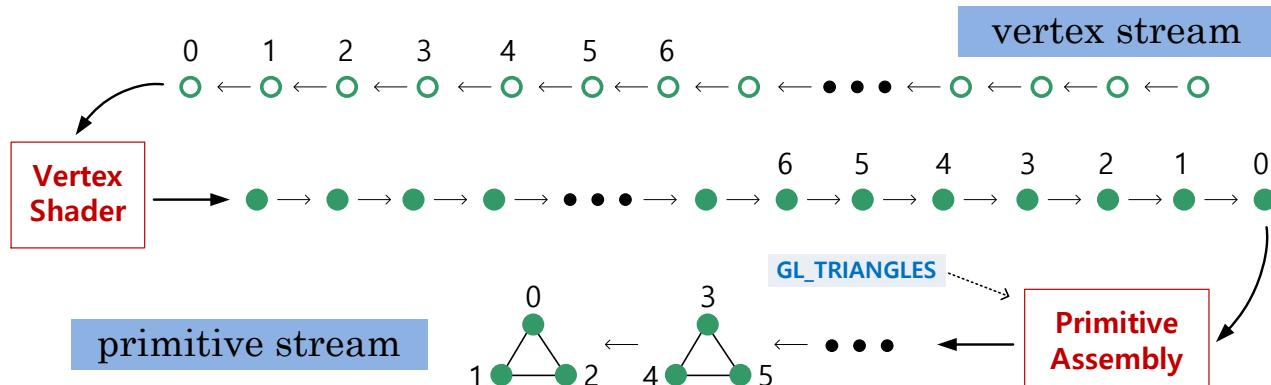
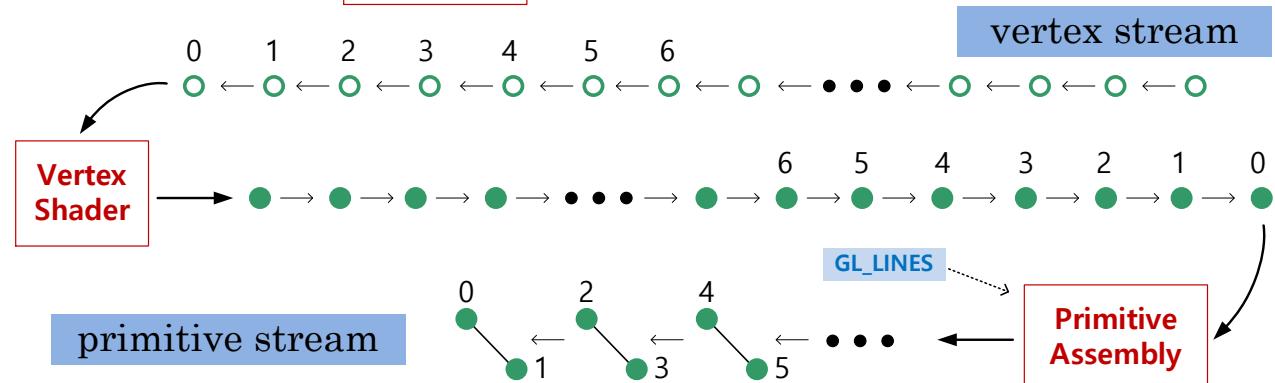
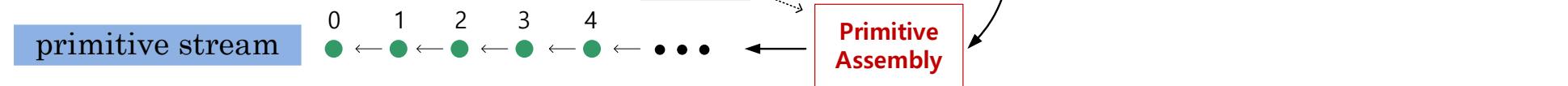
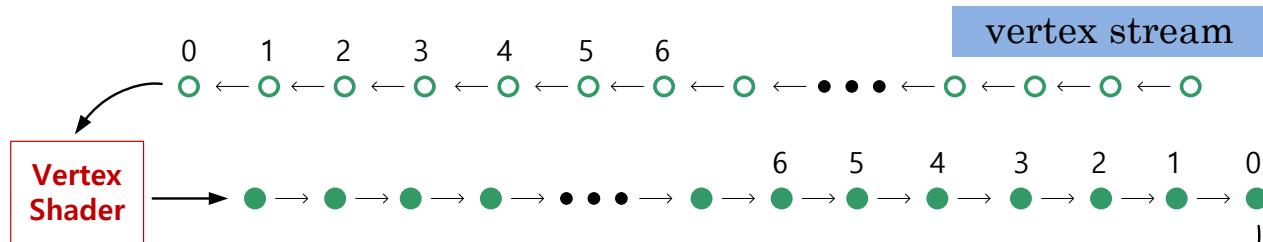
out vec3 v_position_EC;
out vec3 v_normal_EC;
out vec2 v_tex_coord;

void main(void) {
    v_position_EC = vec3(u_ModelViewMatrix*vec4(a_position, 1.0f));
    v_normal_EC = normalize(u_ModelViewMatrixInvTrans*a_normal);
    v_tex_coord = a_tex_coord;

    gl_Position = u_ModelViewProjectionMatrix*vec4(a_position, 1.0f);
}
```



Examples of Primitive Assembly



Code Example of Object Description

```
#define BUFFER_OFFSET(offset) ((GLvoid *) (offset))

GLfloat *object_vertices; // pointer to vertex array data of object in CPU memory
int object_n_triangles; // number of triangles in object

GLuint object_VBO;
glGenBuffers(1, &object_VBO);

glBindBuffer(GL_ARRAY_BUFFER, object_VBO);
glBufferData(GL_ARRAY_BUFFER, object_n_triangles*3*8*sizeof(float), object_vertices, GL_STATIC_DRAW);

glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 8*sizeof(float), BUFFER_OFFSET(0));
 glEnableVertexAttribArray(0);

glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, 8*sizeof(float), BUFFER_OFFSET(3*sizeof(float)));
 glEnableVertexAttribArray(1);

glVertexAttribPointer(2, 2, GL_FLOAT, GL_FALSE, 8*sizeof(float), BUFFER_OFFSET(6*sizeof(float)));
 glEnableVertexAttribArray(2);

glBindBuffer(GL_ARRAY_BUFFER, 0);

...

glBindBuffer(GL_ARRAY_BUFFER, object_VBO);
glDrawArrays(GL_TRIANGLES, 0, 3*object_n_triangles);
glBindBuffer(GL_ARRAY_BUFFER, 0);
```

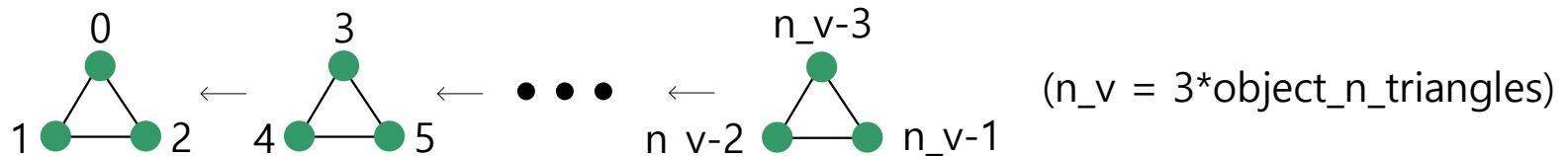
Vertex Array in Client's Address Space (CPU Memory)

- **Vertex array**
 - Vertex attribute data placed into an array in client's address space (CPU memory)

- **In this example,**

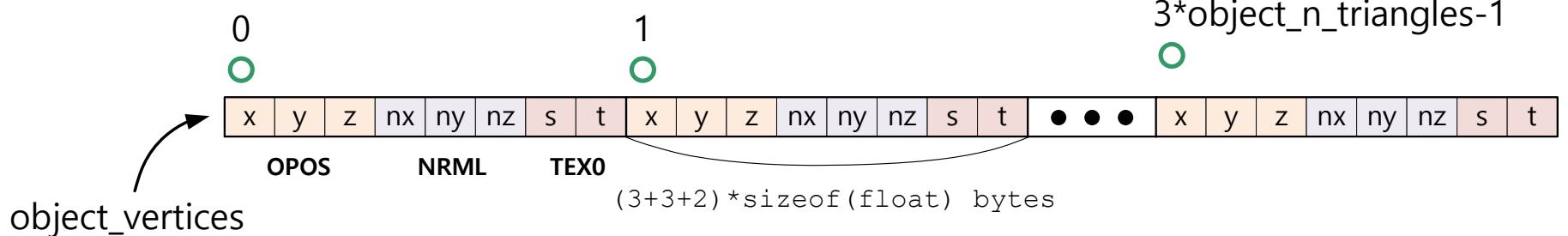
geometric object

```
GLfloat *object_vertices; // pointer to  
int object_n_triangles; // number of tri
```



$$(n_v = 3 \cdot \text{object_n_triangles})$$

vertex array



Vertex Buffer Object (VBO)

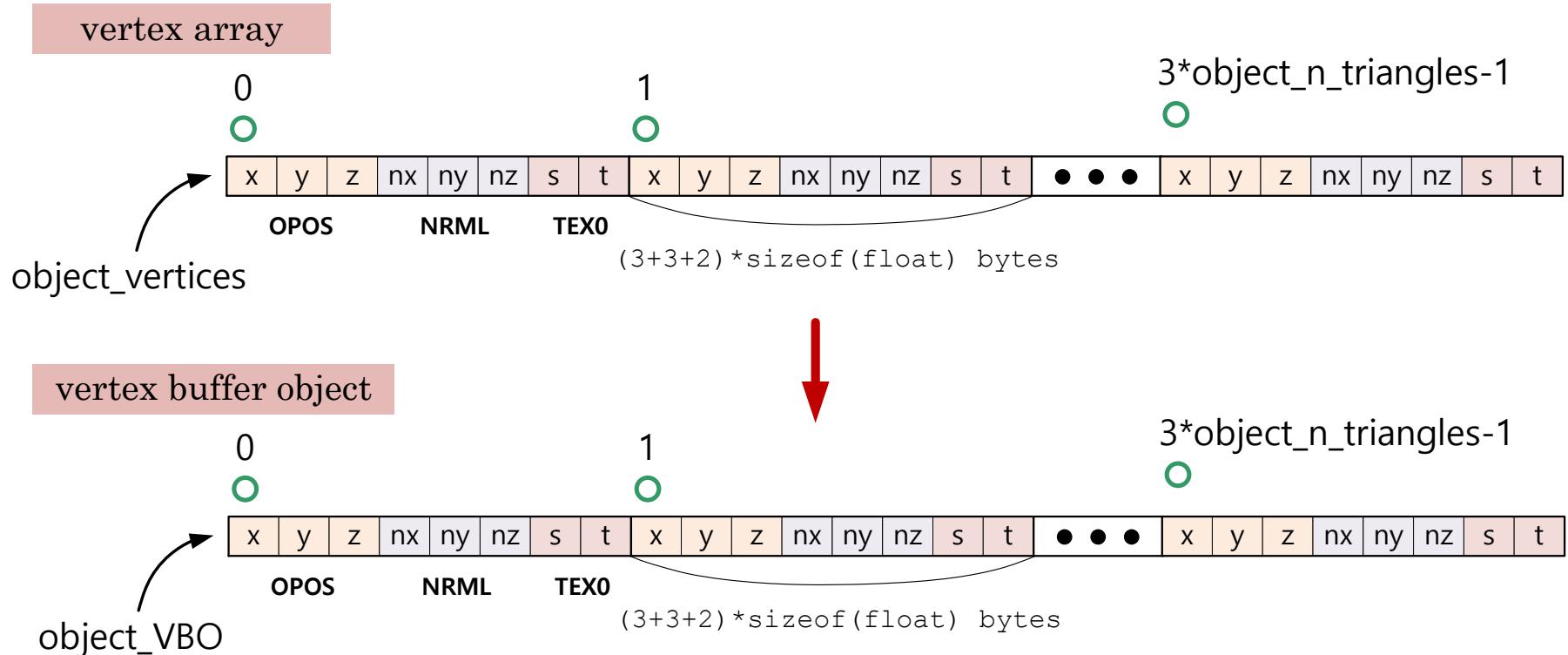
- **Buffer object**
 - Memory that the OpenGL server allocates and owns in server's address space (GPU memory)
 - Almost all data passed into OpenGL is done by storing the data in a buffer object.
- **Vertex buffer object**
 - A buffer object that holds various data related to a collection of vertices
 - ✓ 다른 종류의 데이터를 저장해주는 buffer object들도 다수 존재
- **In this example,**
 1. 사용할 buffer object의 이름을 받아옴.

```
GLuint object_VBO;  
glGenBuffers(1, &object_VBO);
```
 2. (처음 이 함수가 호출된다면 이 buffer object를 생성한 다음) 이 buffer object를 active 상태로 바꿈
 - 이후 buffer object에 영향을 미치는 모든 명령어는 이 active한 buffer object에 영향을 미침.
 - GL_ARRAY_BUFFER로 현재 buffer object가 VBO임을 명시함.

```
glBindBuffer(GL_ARRAY_BUFFER, object_VBO);
```

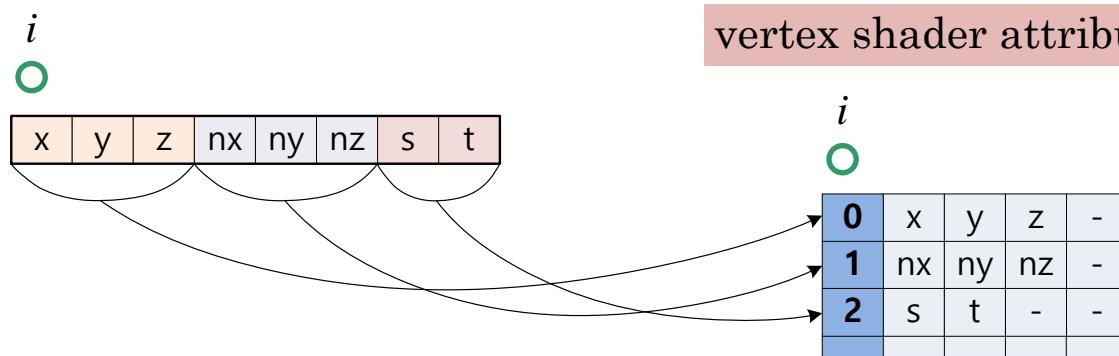
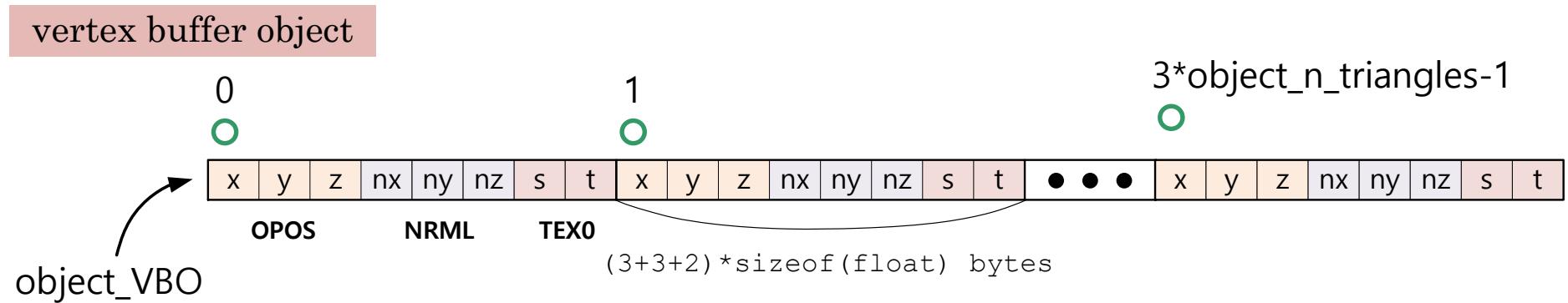
3. CPU 쪽의 vertex array 데이터를 GPU 쪽의 buffer object로 로드 함.

```
glBufferData(GL_ARRAY_BUFFER,  
             object_n_triangles*3*8*sizeof(float), // the amount of storage needed  
             object_vertices, // a pointer to a client memory for vertex array  
             GL_STATIC_DRAW);
```

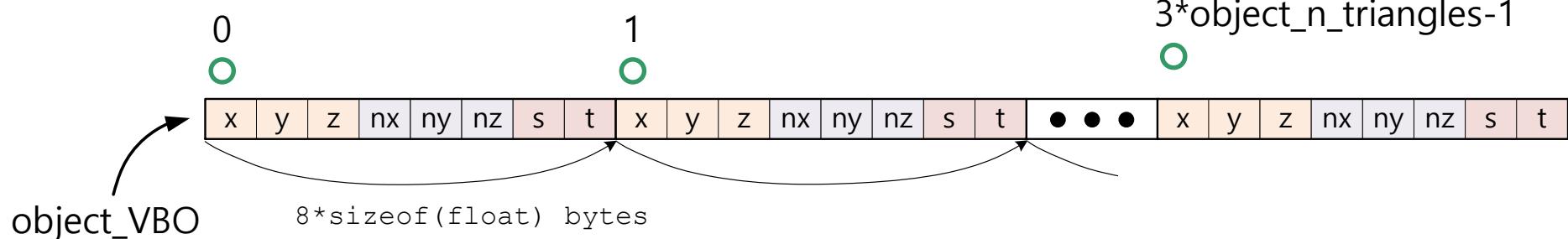


Association of Vertex Attributes with Vertex Shader Variables

- Vertex shader는 vertex stream의 vertex 한 개를 input으로 받아 처리한 후 그 vertex를 output으로 내보냄.
- 각 vertex에 대한 vertex attribute가 vertex shader의 어떤 variable에 대응되는지를 지정해야 함.



vertex buffer object



- In this example,

```
glVertexAttribPointer(0, // shader attribute location
                     3, // the number of components to be used
                     GL_FLOAT, // type of each element in the array
                     GL_FALSE, // normalization indicator
                     8*sizeof(float), // the byte offset between consecutive
                                       // elements in the array
                     BUFFER_OFFSET(0)); // the offset from the start of
                           // buffer object

glEnableVertexAttribArray(0); // enable the vertex array associated with
                           // the current variable

glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, 8*sizeof(float),
                     BUFFER_OFFSET(3*sizeof(float)));
glEnableVertexAttribArray(1);

glVertexAttribPointer(2, 2, GL_FLOAT, GL_FALSE, 8*sizeof(float),
                     BUFFER_OFFSET(6*sizeof(float)));
glEnableVertexAttribArray(2);
```

Drawing with glDrawArray(*) Function

- 지금까지의 과정을 통하여 vertex stream을 생성할 준비를 마침.
- glDrawArray(*) 함수를 사용하여 vertex data를 그래픽스 파이프라인으로 보냄.
 - 이 함수의 첫 번째 인자를 통하여 vertex들을 어떻게 조합할지(primitive assembly)를 결정함.

```
glBindBuffer(GL_ARRAY_BUFFER, object_VBO);
glDrawArrays(GL_TRIANGLES, // what kind of primitive is to be contructed
             0, // use the elements in the array starting at index first
             3*object_n_triangles // the number of vertices to be processed
);
glBindBuffer(GL_ARRAY_BUFFER, 0);
```

- ✓ 지금까지의 과정을 통하여 어떻게 vertex stream이 생성되어 vertex shader로 보내지는지, 그리고 어떻게 primitive assembly가 되는지에 대하여 다시 한 번 생각해볼 것.

Vertex Array Object (VAO)

- **Vertex array object**
 - An OpenGL object that describes how the vertex attributes are stored in a Vertex buffer object (VBO).
 - Encapsulates all the data that is associated with the vertex processor.
 - Is not the actual object storing the vertex data, but the descriptor of the vertex data.

• In this example,

```
GLuint object_VAO;
GLuint object_VBO;

glGenVertexArrays(1, &object_VAO);
glBindVertexArray(object_VAO);

glBindBuffer(GL_ARRAY_BUFFER, object_VBO);
glBufferData(GL_ARRAY_BUFFER, object_n_trianlges*3*8*sizeof(float), object_vertices, GL_STATIC_DRAW);
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 8*sizeof(float), BUFFER_OFFSET(0));
 glEnableVertexAttribArray(0);
glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, 8*sizeof(float), BUFFER_OFFSET(3*sizeof(float)));
 glEnableVertexAttribArray(1);
glVertexAttribPointer(2, 2, GL_FLOAT, GL_FALSE, 8*sizeof(float), BUFFER_OFFSET(6*sizeof(float)));
 glEnableVertexAttribArray(2);
 glBindBuffer(GL_ARRAY_BUFFER, 0);

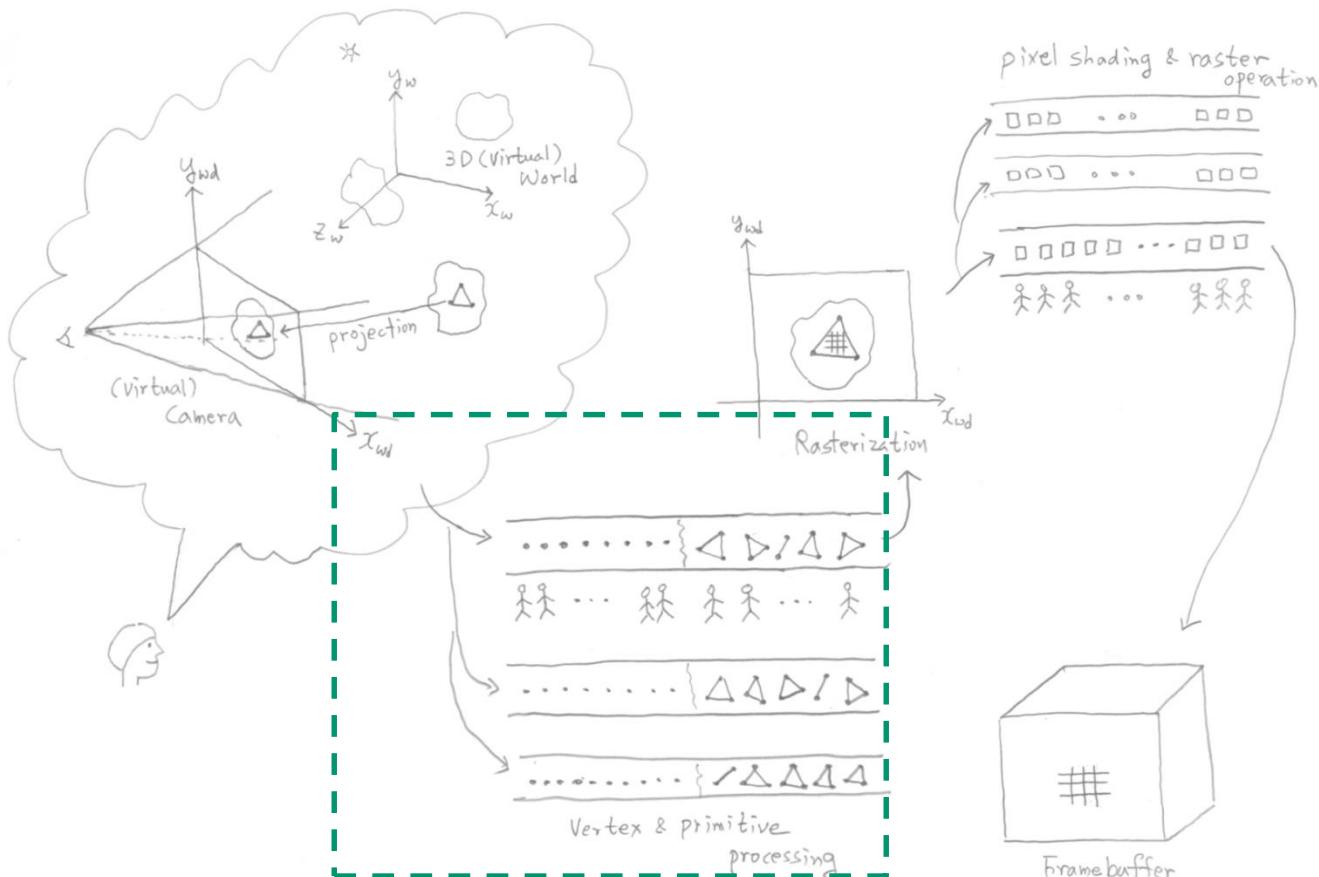
glBindVertexArray(0);
```

```
glBindVertexArray(object_VAO);
glDrawArrays(GL_TRIANGLES, 0, 3*object_n_triangles);
glBindVertexArray(0);
```

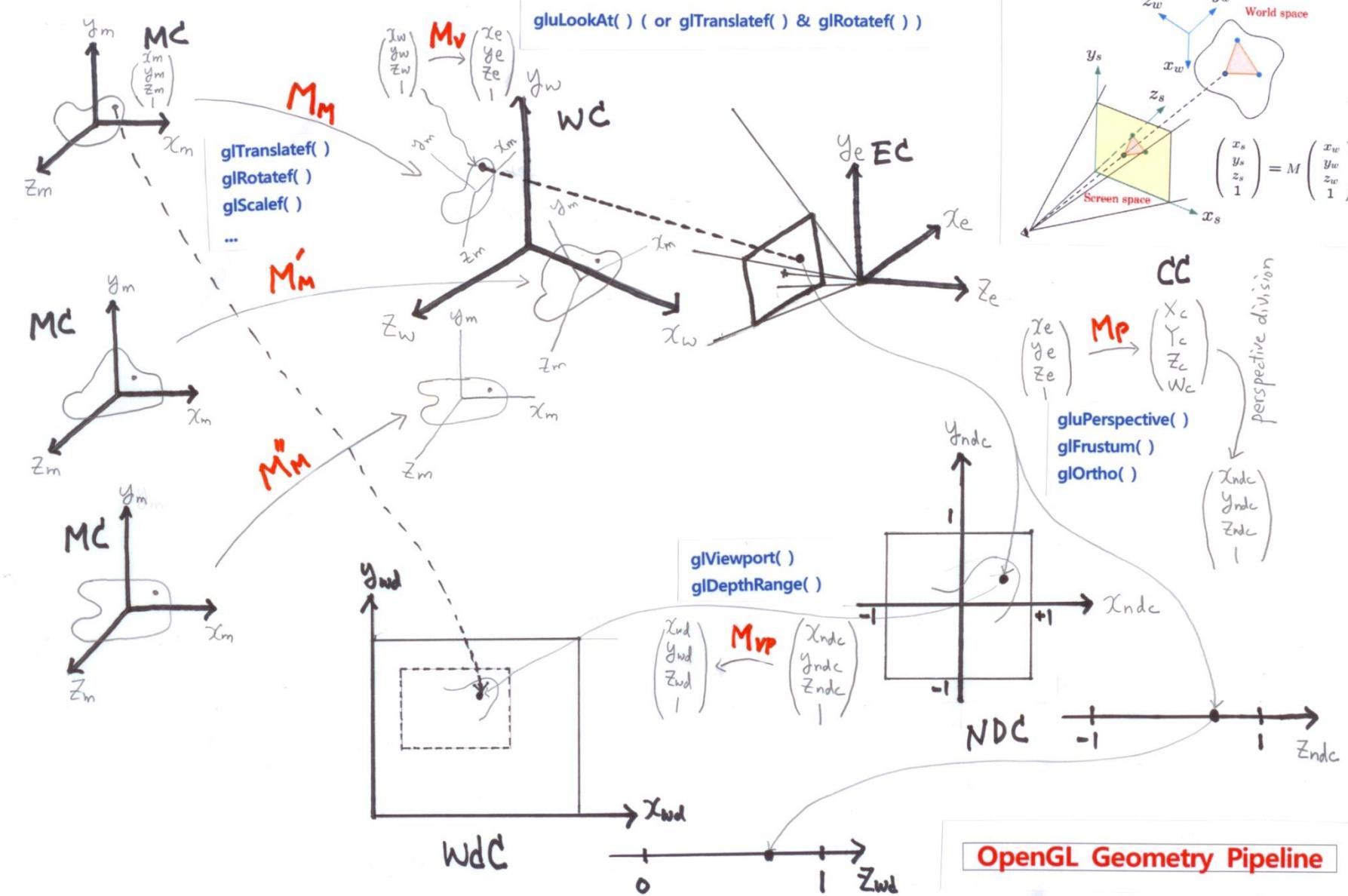
From Modeling Coordinates to World Coordinates: Modeling Transformation

3D Viewing: the Most Important Geometry Processing before Rasterization

- Virtual world에 존재하는 polygonal model(3D)의 geometric primitive들(또는 그것을 구성하는 vertex들) 각각이 virtual camera의 필름 면에 해당하는 image plane(2D)의 어느 지점에 projection되는지를 결정하는 과정을 어떻게 설계하고 구현할 것인가?
 - Real-time rendering system 개발 측면 vs application programming 측면



OpenGL Geometry Pipeline Overview



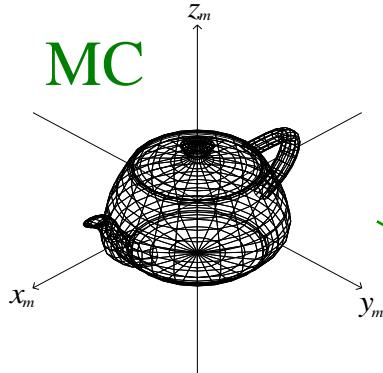
3D Coordinate Systems in OpenGL

- 모델링 좌표계(Modeling Coordinates, MC)

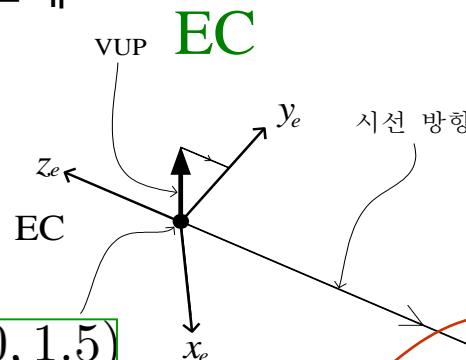
- 가상의 세상에 존재하는 각 물체들이 정의가 되는 좌표계
- 일반적으로 각각의 물체를 자신만의 좌표계에서 모델링한 후 세상에 배치함.

- 세상 좌표계(World Coordinates, WC)

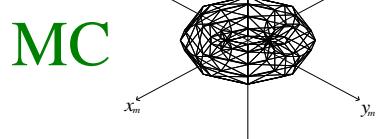
- 가상의 세상의 기준이 되는 좌표계



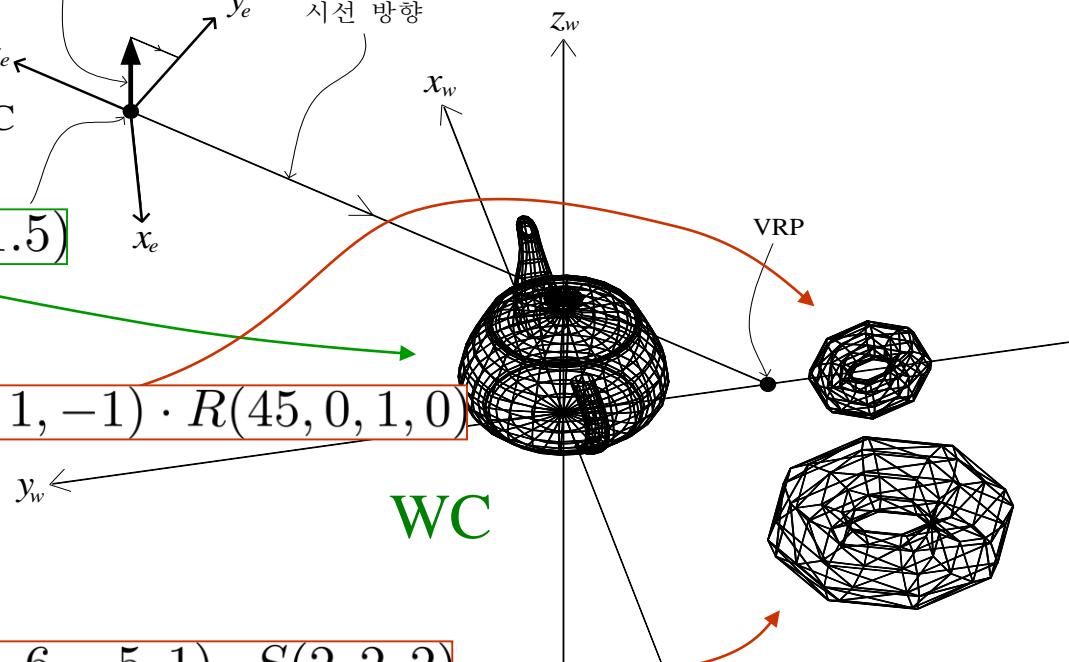
$$M_M = T(0.0, 0.0, 1.5)$$



$$M_M = T(0, -6, 0) \cdot R(-45, 0, 1, 0) \cdot S(1, 1, -1) \cdot R(45, 0, 1, 0)$$

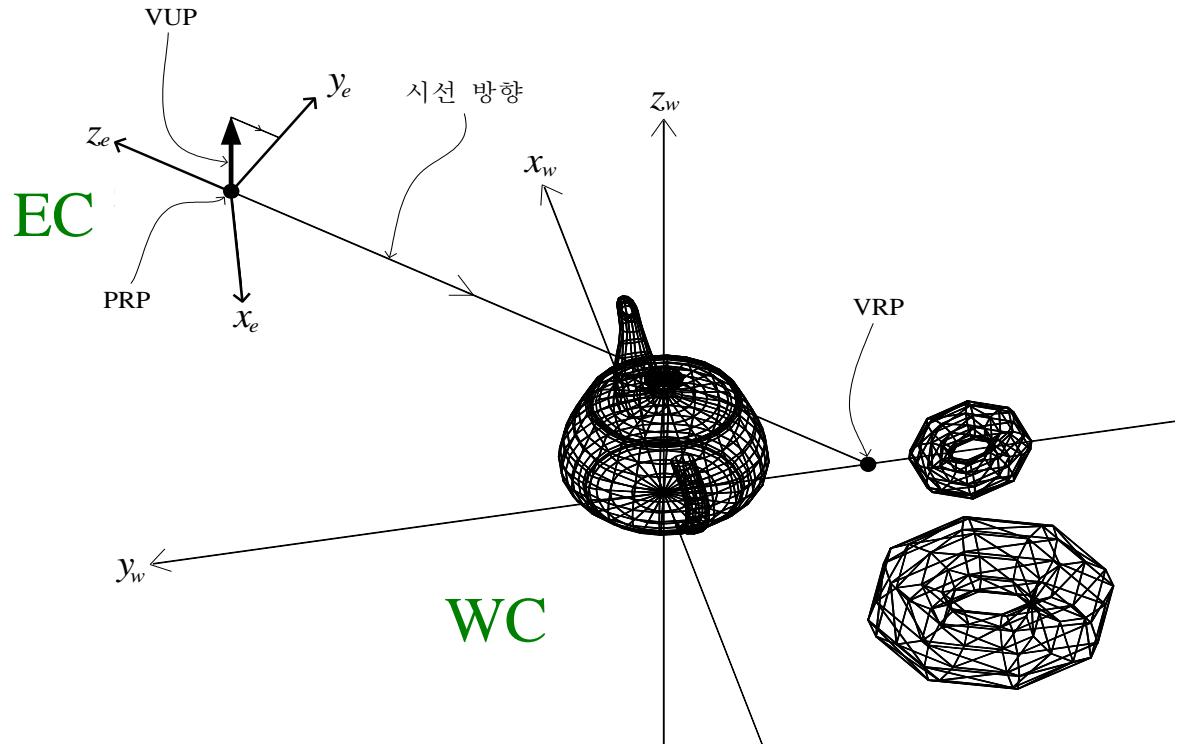
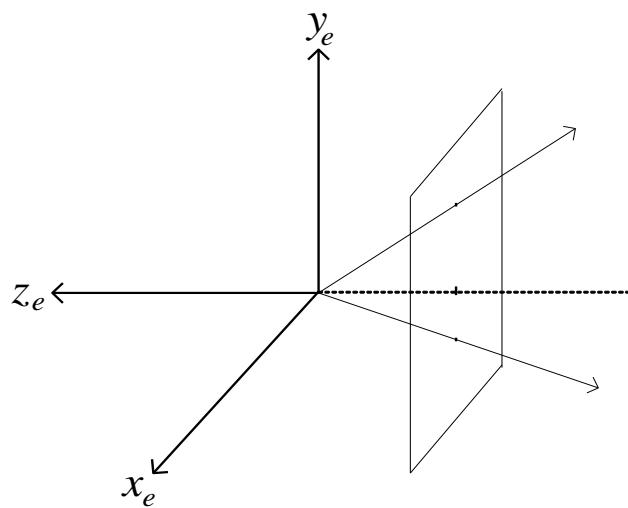


$$M_M = T(-6, -5, 1) \cdot S(2, 2, 2)$$



• 눈 좌표계(Eye Coordinates, EC)

- 가상의 카메라를 기준으로 하는 좌표계로서, Camera Coordinates라고 하기도 함.
- 가상의 세상에 존재하는 점이 카메라를 기준으로 왼쪽으로, 오른쪽으로, 그리고 앞으로 얼마나큼 떨어져 있는가 하는 정보를 제공함.

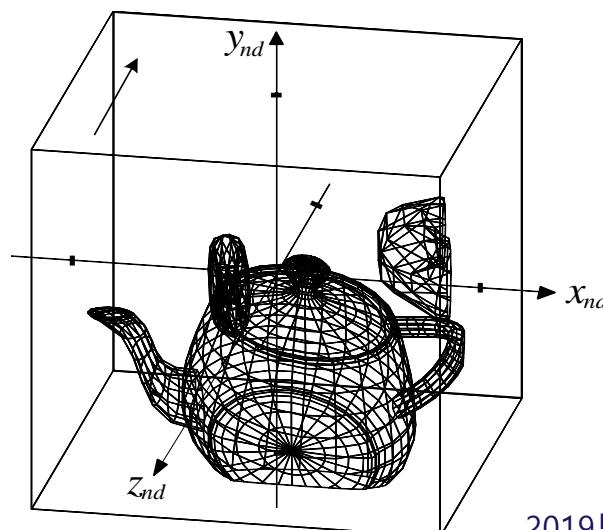


• 절단 좌표계(Clip Coordinates, CC)

- Graphics rendering pipeline 과정에서 투영 변환(projection transformation)이 수행된 직후의 좌표계.
- Affine space에서 한 점이 (x, y, z) 로 표현되는 다른 OpenGL 좌표계와는 달리 3차원 공간의 점이 (X, Y, Z, W) 의 homogeneous coordinates로 표현되는 projective space에 존재함.
- 이후 원근 나눗셈(perspective division) 계산을 통해 다시 affine space에서 정의되는 NDC로 변환 됨.

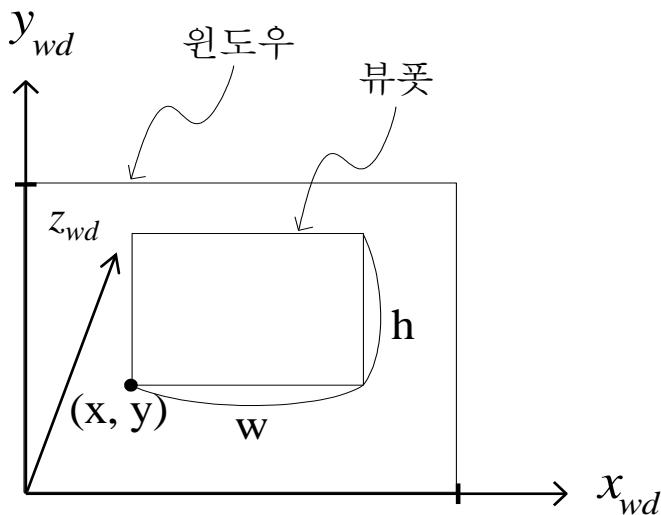
• 정규 디바이스 좌표계(Normalized Device Coordinates, NDC)

- '정규화된 3차원 필름'에 해당하는 좌표계
- 필름 상에서의 위치 (x, y) 외에도 깊이 정보인 z 값을 유지하는 3차원 좌표계임.



- **윈도우 좌표계(Window Coordinates, WdC)**

- 사용자가 화면에 띄운 윈도우의 픽셀 위치의 기준이 되는 좌표계
- 개념적으로 윈도우는 2D 공간에서 정의되나, graphics rendering pipeline에서는 한 점의 위치에 해당하는 (x, y) 좌표 외에 그 점이 화면에서 얼마나 떨어져 있는가 하는 깊이 정보 z 도 유지하는 3차원 좌표계임.



OpenGL의 Geometry Pipeline과 사진 촬영과의 비교

- 모델링 변환(Modeling transformation)
 - 피사체 및 조명 배치
 - 모델링 좌표계(MC) → 세상 좌표계(WC)
- 뷰잉 변환(Viewing transformation)
 - 카메라의 위치와 방향 설정
 - 세상 좌표계(WC) → 눈 좌표계(EC)
- 투영 변환(Projection transformation)
 - 사용할 렌즈 결정 및 촬영 대상 결정
 - 카메라의 셔터를 누름
 - 눈 좌표계(EC) → 절단 좌표계(CC) → 정규 디바이스 좌표계(NDC)
- 뷰포트 변환(Viewport transformation)
 - 사진의 크기 결정 및 인화
 - 정규 디바이스 좌표계(NDC) → 윈도우 좌표계(WdC)

OpenGL의 Geometry Pipeline 요약

- OpenGL geometry pipeline은

1. 자신만의 고유한 좌표계에서 정의된 물체를 구성하는 각 geometric primitive들을 가상의 세상에 배치한 후,
2. 가상의 세상에 가상의 카메라를 배치한 후,
3. 카메라 렌즈의 성질을 설정한 후,
4. 셔터를 눌러 촬영을 하여,
5. 카메라의 필름에 상이 맺힌 후,
6. 결과적으로 각 geometric primitive들이 인화지에 해당하는 윈도우에 투영되는 위치를 구하는

과정을 수학적으로 모델링한 절차임.

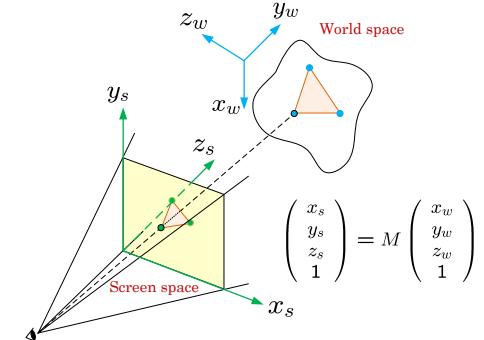
- 이 과정을 이해하기 위하여 다음과 같은 개념에 대한 정확한 이해가 요구됨.

- 좌표계(Coordinate system)

- Affine space vs projective space

- 기하변환(Geometric transformation)

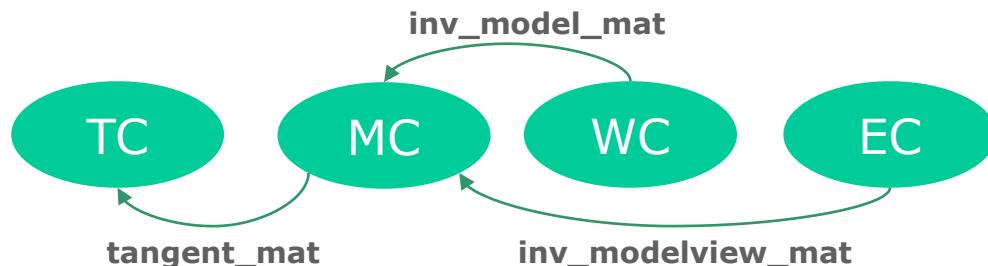
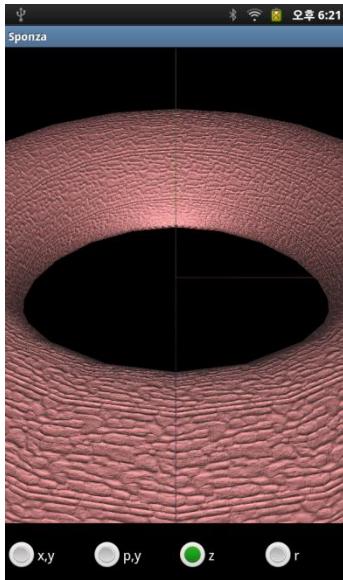
- Affine transformation vs projective transformation



기하 계산 과정의 정확한 이해에 대한 중요성

- 기본적으로 기하 파이프라인은 화면에 물체가 어떻게 투영되는가에 관한 것임.
- 하지만, 기하계산 이후의 쉐이딩(shading) 과정, 즉 색깔 계산 과정에서도 기하 파이프라인의 개념이 매우 중요하게 사용이 됨.

Bump mapping



$$N = \begin{pmatrix} n_x \\ n_y \\ n_z \end{pmatrix} \quad B = \begin{pmatrix} b_x \\ b_y \\ b_z \end{pmatrix}$$

$$\text{MC} \quad z_m \quad \text{TC} \quad T = \begin{pmatrix} t_x \\ t_y \\ t_z \end{pmatrix}$$

$$x_m \quad y_m \quad X_m \longrightarrow X_t$$

$$M_{MT}^p = \begin{bmatrix} t_x & t_y & t_z \\ b_x & b_y & b_z \\ n_x & n_y & n_z \end{bmatrix}$$

$$M_{MT}^n = ((M_{MT}^p)^{-1})^t = \begin{bmatrix} t_x & t_y & t_z \\ b_x & b_y & b_z \\ n_x & n_y & n_z \end{bmatrix}$$

$$X_t = M_{MT}^n \cdot X_m$$

$$N = \begin{pmatrix} n_x \\ n_y \\ n_z \end{pmatrix} \quad B = \begin{pmatrix} b_x \\ b_y \\ b_z \end{pmatrix}$$

$$\text{WC} \quad \text{EC}$$

$$L \quad N^* \quad V$$

$$\text{MC} \quad T = \begin{pmatrix} t_x \\ t_y \\ t_z \end{pmatrix}$$

$$N = \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix} \quad B = \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix}$$

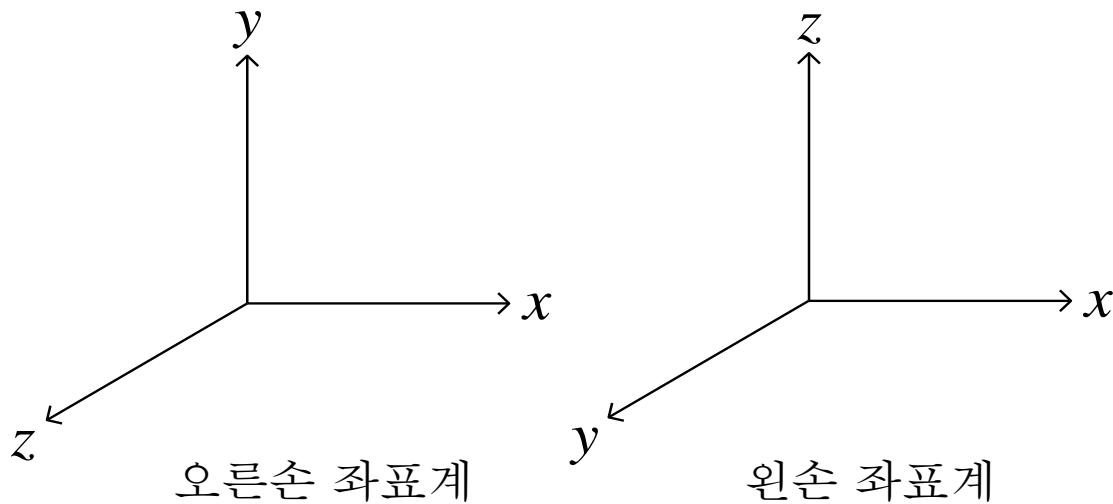
$$\text{TC (SLC)} \quad N^* \quad V$$

$$T = \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix}$$

3D Coordinate Systems and Homogenous Coordinates

Cartesian Coordinate System and its Orientation

- 3차원 좌표 (x , y , z)의 의미
- 오른손 (right-handed) 및 왼손 좌표계(left-handed coordinate system)



- 차이점 및 용도

Homogeneous Coordinate System (동차 좌표계)

- 유클리드 기하학(Euclid geometry)과 투영 기하학(projective geometry)
 - 유클리드 기하학
 - 유클리드 공간(Euclidean space), 아핀 공간(affine space)
 - 2차원 공간의 점 $\rightarrow (x, y)$
 - 3차원 공간의 점 $\rightarrow (x, y, z)$
 - 투영 기하학
 - 투영 공간(projective space)
 - 2차원 공간의 점 $\rightarrow (X, Y, W)$
 - 3차원 공간의 점 $\rightarrow (X, Y, Z, W)$
- 컴퓨터과학 수준의 동차 좌표계의 정의
 - *A coordinate system in which there is an extra dimension, used most commonly in computer science to specify whether the given coordinates represent a **vector** (if the last coordinate is zero) or a **point** (if the last coordinate is non-zero).*

From WIKIPEDIA: The Free Encyclopedia

- 2차원 투영 공간에서 두 점 (X, Y, W) 과 (X', Y', W') 은 다음과 같은 조건을 만족할 때 같은 점을 표현한다고 간주함.

$(X, Y, W) \equiv (X', Y', W')$ if and only if

$$(X, Y, W) = \alpha(X', Y', W') = (\alpha X', \alpha Y', \alpha W'), \alpha \neq 0$$

- 예: $(2, -3, 1) \equiv (4, -6, 2) \equiv (-1, 1.5, -0.5)$

- 아핀 공간 (x, y) 와 투영 공간 (X, Y, W) 의 관계
 - $W \neq 0$ 인 경우
 - $(X, Y, W) \equiv (\frac{X}{W}, \frac{Y}{W}, 1) \rightarrow (X, Y, W) \equiv (x, y) = (\frac{X}{W}, \frac{Y}{W})$
 - 아핀 공간의 모든 점은 W 가 0이 아닌 투영공간의 점에 해당. 즉 아핀 공간은 투영공간에 포함.
 - $W = 0$ 인 경우
 - 이러한 점을 무한대 점(point at infinity)라 함.
 - 이러한 점은 아핀 공간에 존재하지 않는 점으로서, 아핀 공간에서는 위치에 해당하는 점이 아니라 방향에 해당하는 벡터로 보면 편리함.
- 3차원 투영공간
 - 점의 좌표가 (X, Y, Z, W) 로 표현됨.
 - 2차원 투영공간의 개념이 자연스럽게 확장이 됨.
 - 컴퓨터 그래픽스 분야의 기본이 되는 기하 변환의 원리를 이해하는데 큰 도움이 됨.

- 투영공간은 아핀 공간보다 더 '완전한' 공간이라 할 수 있음.
 - 무한(infinity)의 개념을 도입.
 - 아핀 공간에서의 여러 기하학적인 예외가 사라짐.
 - 위치에 해당하는 점과 방향에 해당하는 벡터의 구별이 사라짐.
 - 기타 등등
- 예: 두 직선은 한 점에서 만남

- 아핀 공간:

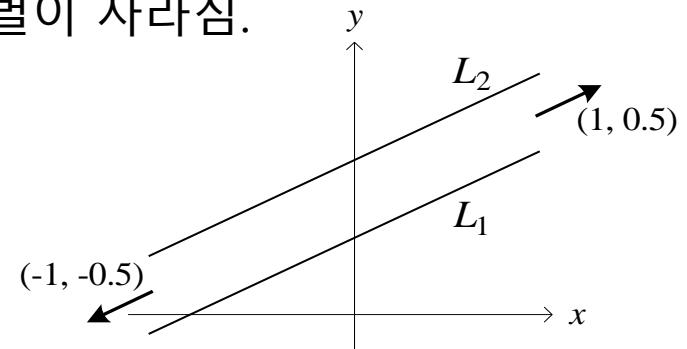
$$L_1 : 2x - 4y + 4 = 0, L_2 : 2x - 4y + 8 = 0$$

- 교점 \rightarrow ???

- 투영 공간:

$$L_1 : 2X - 4Y + 4W = 0, L_2 : 2X - 4Y + 8W = 0$$

- 교점 $\rightarrow (X, \frac{X}{2}, 0) \quad (X \neq 0)$

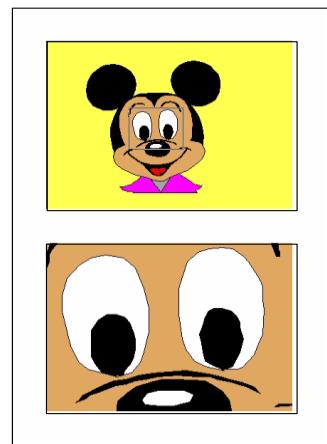
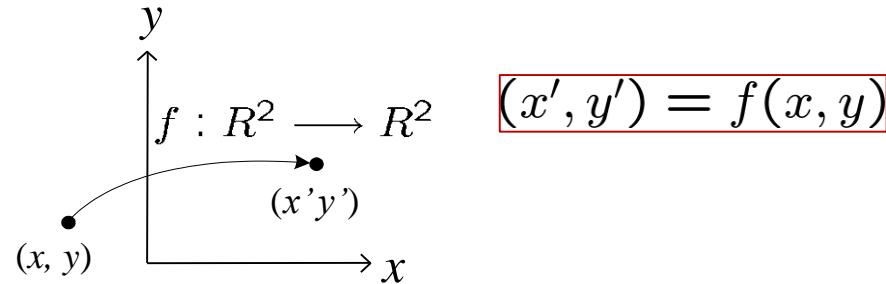


Geometric Transformation

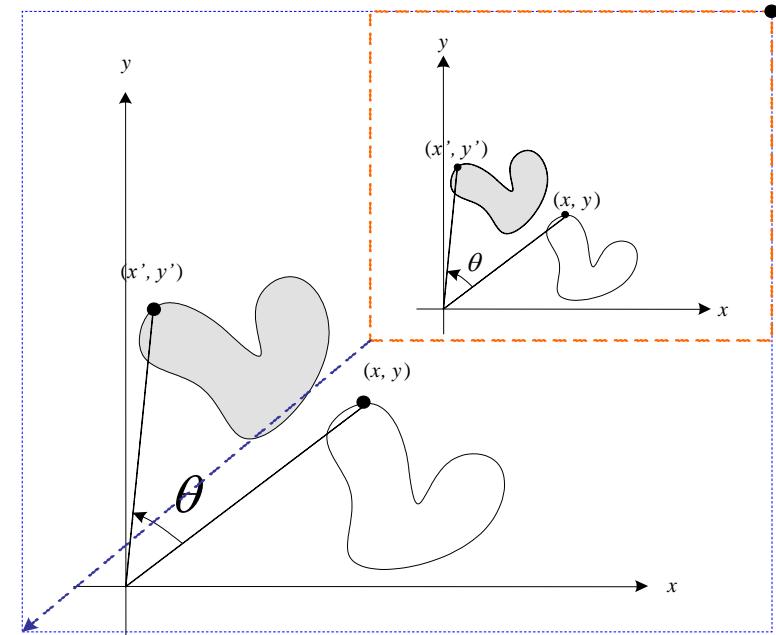
- Affine Transformation

2D Geometric Transformation

- 기하 변환
 - 컴퓨터 그래픽스 분야에서 가장 기본이 되는 요소.
 - 수학적(개념적) 정의



2차원 변환 응용 예 1



2차원 변환 응용 예 2

2D Affine Transformation

- 정의: A transformation $f : R^2 \rightarrow R^2$ is called an **2D affine transformation** if a point $p = (x \ y)^t$ is transformed to another point $p' = (x' \ y')^t$ such that $p' = f(p) = Ap + v$, where

$$A = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \in R^{2 \times 2} \text{ and } v = \begin{pmatrix} v_1 \\ v_2 \end{pmatrix}.$$

A linear transformation followed by a translation

- Matrix notation:

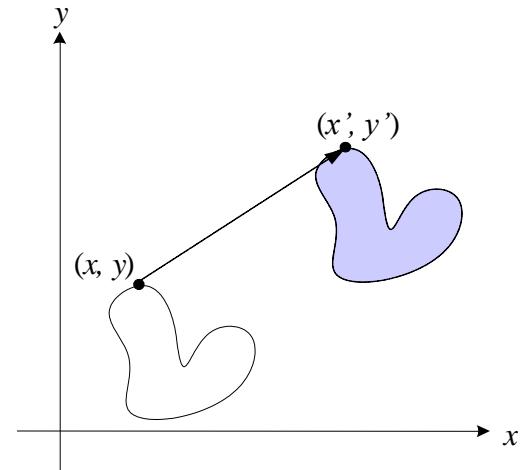
$$\begin{pmatrix} x' \\ y' \\ 1 \end{pmatrix} = \begin{pmatrix} a_{11} & a_{12} & v_1 \\ a_{21} & a_{22} & v_2 \\ 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}$$

$$\begin{array}{ccc} p' & M & p \end{array}$$

2차원 아핀 변환의 예

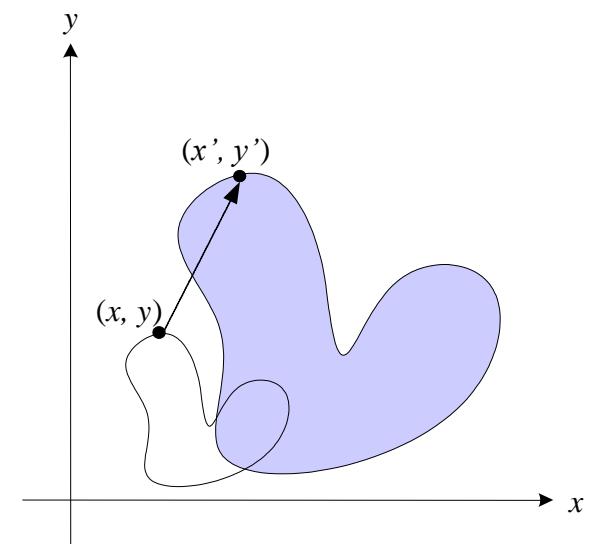
- **Translation:** 물체의 이동

$$M = \begin{pmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \\ 0 & 0 & 1 \end{pmatrix} \equiv T(t_x, t_y)$$



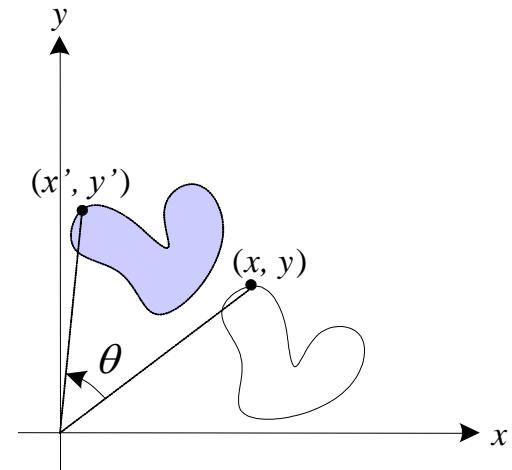
- **Scaling:** 원점을 중심으로 한 물체의 크기 변화

$$M = \begin{pmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{pmatrix} \equiv S(s_x, s_y)$$



- **Rotation:** 원점을 중심으로 한 물체의 회전

$$M = \begin{pmatrix} \cos(\theta) & -\sin(\theta) & 0 \\ \sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 1 \end{pmatrix} \equiv R(\theta)$$



- **Shearing:** 축 방향으로의 물체의 찌그러트림

$$M = \begin{pmatrix} 1 & sh_x & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \equiv SH_x(sh_x)$$

x-shearing

$$M = \begin{pmatrix} 1 & 0 & 0 \\ sh_y & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \equiv SH_y(sh_y)$$

y-shearing

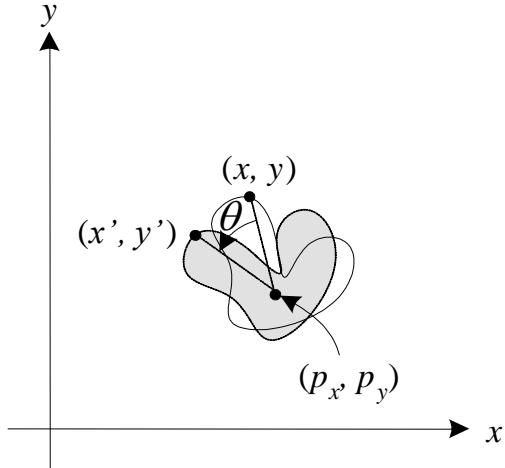
- **Reflection:** 기본축을 중심으로 한 물체의 반사
 - 예: x 축에 대한 반사.

$$M = \begin{pmatrix} 1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \equiv Rf_{y=0}$$

- ★ 2차원 그래픽스에서 사용되는 대부분의 좌표변환은 translation, scaling, rotation, shearing 등의 합성으로 표현 가능.

Example 1: Composition of 2D Transformation

- 문제: 주어진 점을 중심으로 한 회전 변환



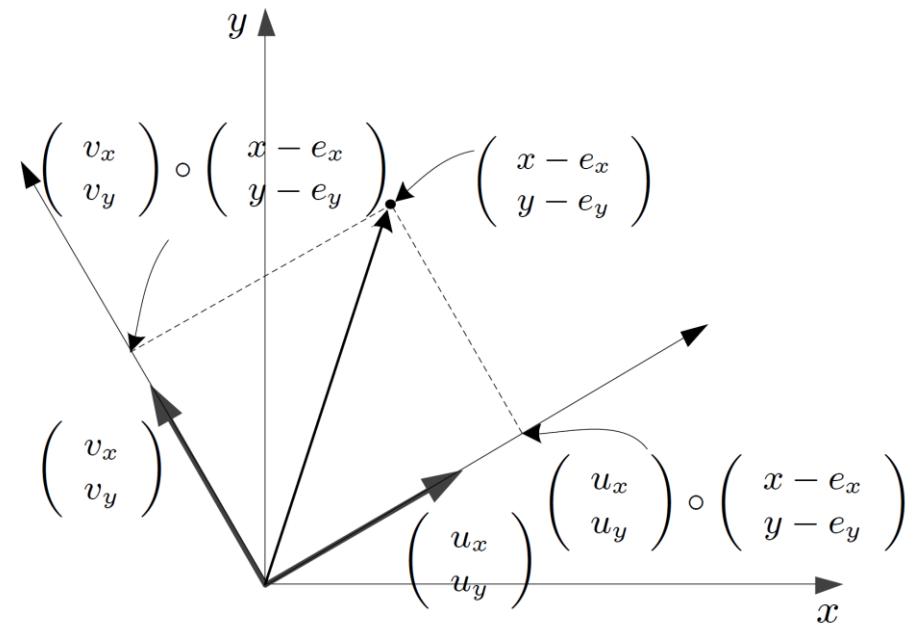
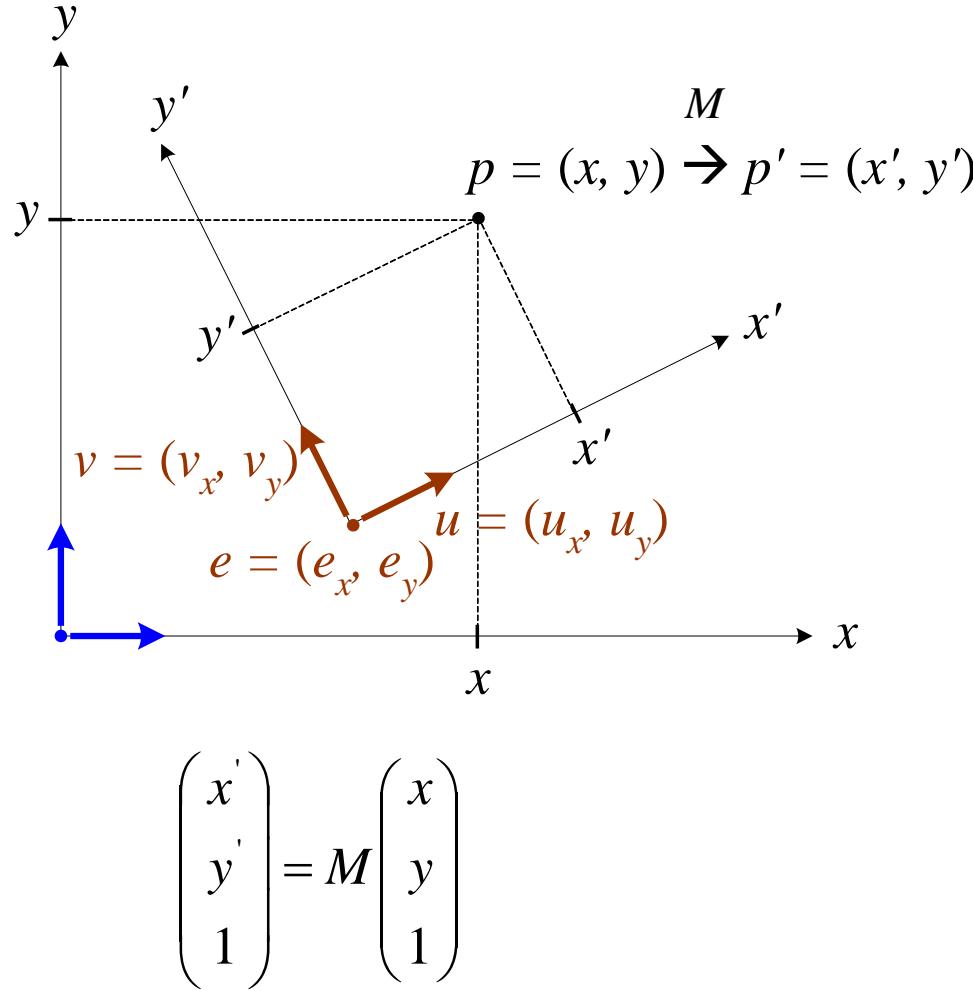
$$M = M_3 \cdot M_2 \cdot M_1 = T(p_x, p_y) \cdot R(\theta) \cdot T(-p_x, -p_y)$$

☞ 응용

- 주어진 점을 중심으로 한 크기 변환
- 주어진 직선에 대한 반사 변환
- 기타

Example 2: Composition of 2D Transformation

- 문제: 두 좌표계간의 좌표 변환

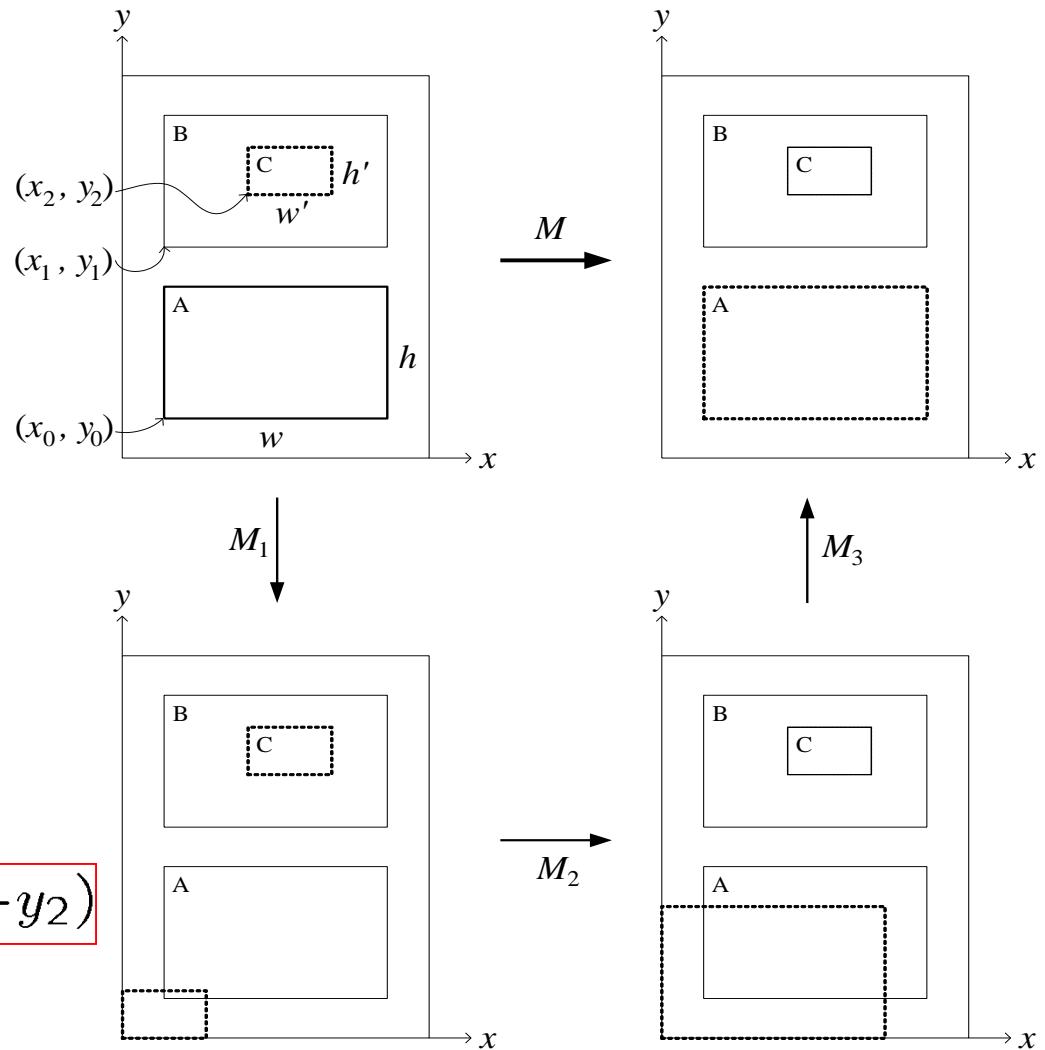
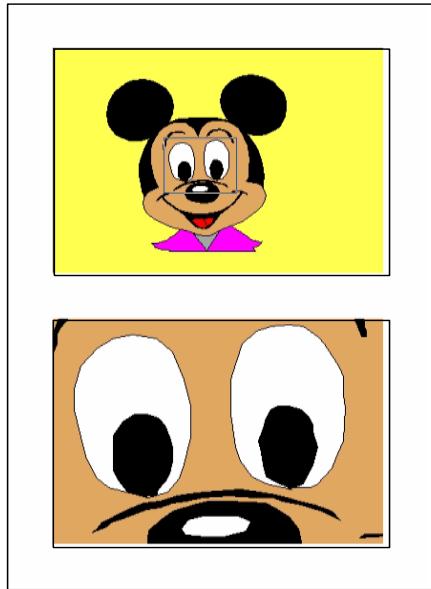


$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} u_x & u_y & 0 \\ v_x & v_y & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x - e_x \\ y - e_y \\ 1 \end{bmatrix}$$

$$M = R \cdot T(-e_x, -e_y)$$

Example 3: Composition of 2D Transformation

- 문제: 선택 영역의 확대

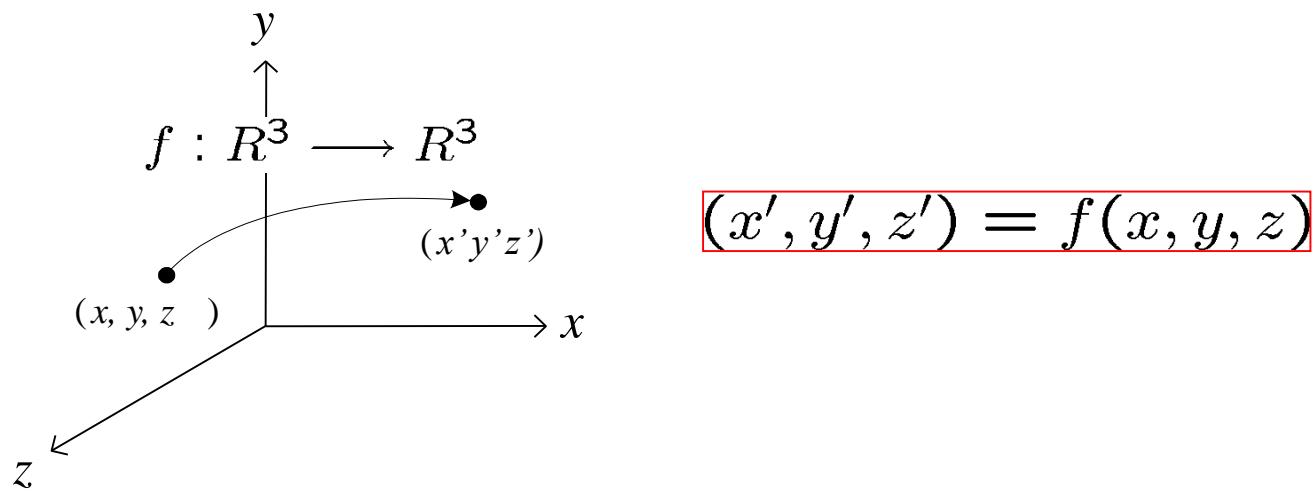


$$M = M_3 \cdot M_2 \cdot M_1$$

$$= T(x_0, y_0) \cdot S\left(\frac{w}{w'}, \frac{h}{h'}\right) \cdot T(-x_2, -y_2)$$

3D Geometric Transformation

- 기하 변환
 - 컴퓨터 그래픽스 분야에서 가장 기본이 되는 요소.
 - 수학적(개념적) 정의



3D Affine Transformation

- 정의: 기하 변환 중 다음과 같은 방식으로 주어진 점을 변환하면 (3차원) 아핀 변환이라 함.

$$p = (x \ y \ z)^t \longrightarrow p' = (x' \ y' \ z')^t$$

$$p' = Ap + v, \quad A = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} \in R^{3 \times 3}, \quad v = \begin{pmatrix} v_1 \\ v_2 \\ v_3 \end{pmatrix} \in R^3$$



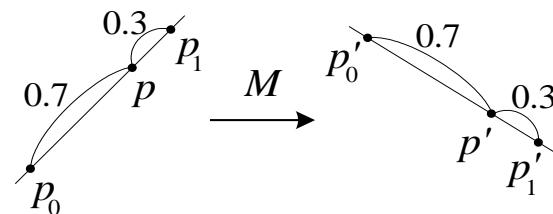
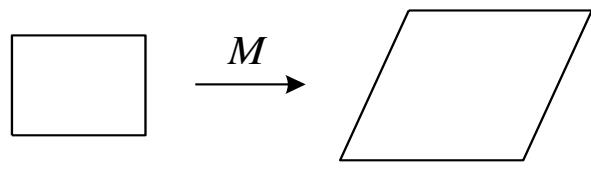
$$\begin{pmatrix} x' \\ y' \\ z' \\ 1 \end{pmatrix} = \begin{bmatrix} a_{11} & a_{12} & a_{13} & v_1 \\ a_{21} & a_{22} & a_{23} & v_2 \\ a_{31} & a_{32} & a_{33} & v_3 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}$$

$$\boxed{p' = Mp}$$

- 특징

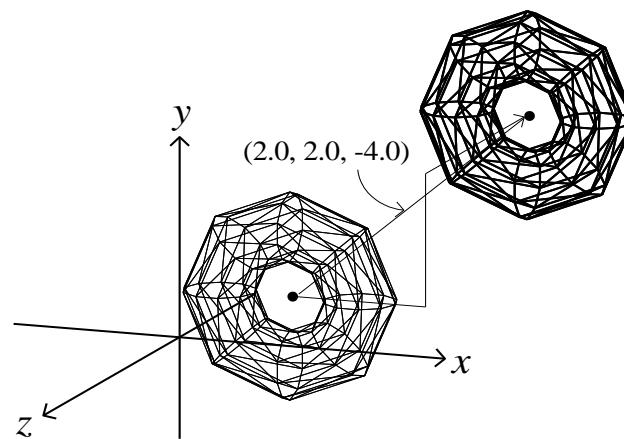
- 기하 변환 중 컴퓨터 그래픽스 분야에서 보편적으로 가장 많이 쓰이는 변환 형태
→ 이동 변환, 회전 변환, 크기 변환 등.
- 12개의 자유도를 가짐. 이 값들의 내용에 따라 다양하게 변환이 됨.
- $p' = Mp$ 형태로 표현하였을 때 M 의 네 번째 행은 항상 $(0 \ 0 \ 0 \ 1)$ 임

- 평행성의 유지와 비율의 보존



- 기본 아핀 변환
 - 이동 변환(Translation)

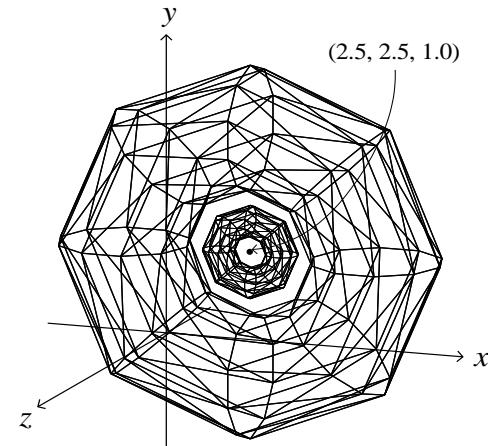
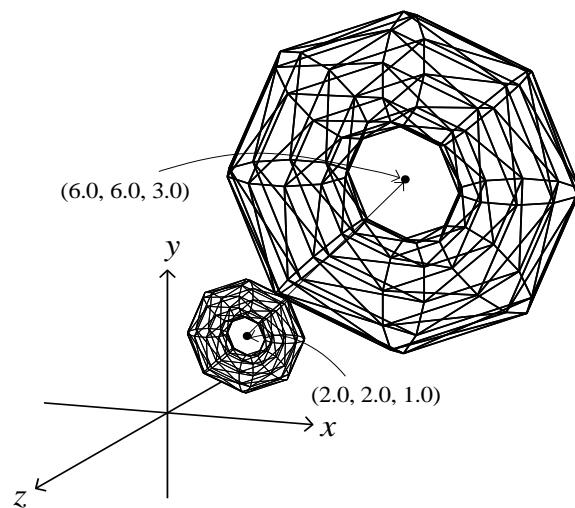
$$T(t_x, t_y, t_z) = \begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$



- 크기 변환(Scaling)

$$S(s_x, s_y, s_z) = \begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

- 등방성 크기 변환과 비등방성 크기 변환

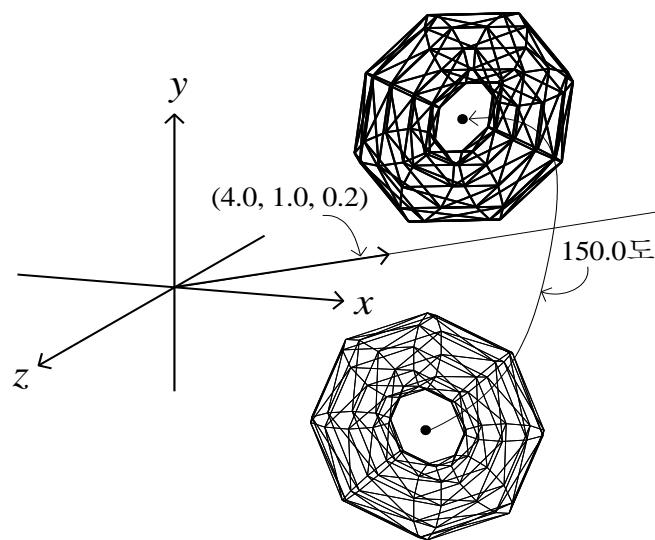
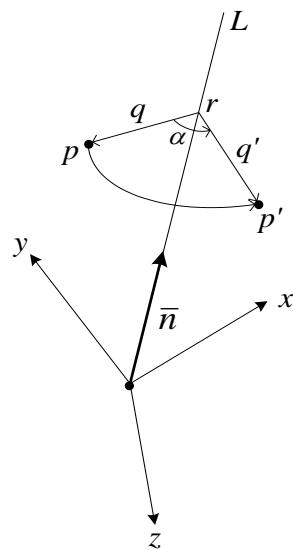


회전 변환은 회전축 방향과 회전 각도로 설정함

- 회전 변환(Rotation)

$$R(\alpha, n_x, n_y, n_z) = \begin{bmatrix} \bar{n}_x^2(1 - c) + c & \bar{n}_x\bar{n}_y(1 - c) - \bar{n}_z s & \bar{n}_z\bar{n}_x(1 - c) + \bar{n}_y s & 0 \\ \bar{n}_x\bar{n}_y(1 - c) + \bar{n}_z s & \bar{n}_y^2(1 - c) + c & \bar{n}_y\bar{n}_z(1 - c) - \bar{n}_x s & 0 \\ \bar{n}_z\bar{n}_x(1 - c) - \bar{n}_y s & \bar{n}_y\bar{n}_z(1 - c) + \bar{n}_x s & \bar{n}_z^2(1 - c) + c & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

where $c = \cos(\alpha)$, $s = \sin(\alpha)$, $(\bar{n}_x, \bar{n}_y, \bar{n}_z) = \frac{(n_x, n_y, n_z)}{|(n_x, n_y, n_z)|}$



$$R_x(\alpha) = R(\alpha, 1, 0, 0)$$

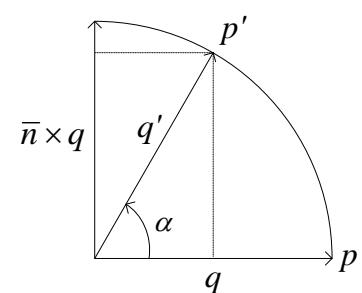
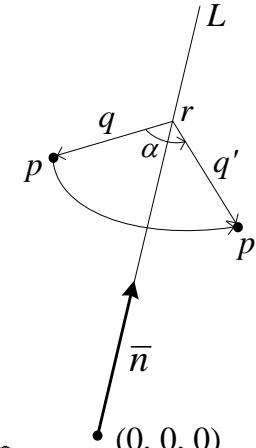
$$R_y(\alpha) = R(\alpha, 0, 1, 0)$$

$$R_z(\alpha) = R(\alpha, 0, 0, 1)$$

회전 변환 행렬의 유도

$$p = (x \ y \ z)^t, \bar{n} = (\bar{n}_x \ \bar{n}_y \ \bar{n}_z)^t \rightarrow p' = (x' \ y' \ z')^t ?$$

$$\begin{aligned}
 r &= (\bar{n} \cdot p)\bar{n} \\
 p' &= r + q' \\
 q' &= q \cos \alpha + (\bar{n} \times q) \sin \alpha \\
 q &= p - r = p - (\bar{n} \cdot p)\bar{n} \\
 \bar{n} \times q &= \bar{n} \times (p - (\bar{n} \cdot p)\bar{n}) = \bar{n} \times p
 \end{aligned}$$



$$\begin{aligned}
 p' &= r + q' = (\bar{n} \cdot p)\bar{n} + (p - (\bar{n} \cdot p)\bar{n}) \cos \alpha + (\bar{n} \times p) \sin \alpha \\
 &\rightarrow p' = (\bar{n} \cdot \bar{n}^t)p + \{\cos \alpha(I - \bar{n} \cdot \bar{n}^t)\}p + \{\sin \alpha S\}p
 \end{aligned}$$

$$R = \bar{n} \cdot \bar{n}^t + \cos \alpha(I - \bar{n} \cdot \bar{n}^t) + \sin \alpha S \quad (S = \begin{bmatrix} 0 & -\bar{n}_z & \bar{n}_y \\ \bar{n}_z & 0 & -\bar{n}_x \\ -\bar{n}_y & \bar{n}_x & 0 \end{bmatrix})$$

$$p' = Rp'$$

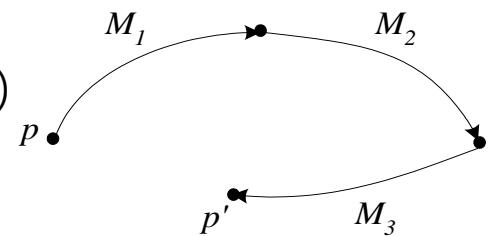
아핀 변환과 행렬 계산

- 기하 변환 vs 행렬 계산

- 변환의 합성(composition)과 행렬의 곱셈(multiplication)

$$p' = (M_3 M_2 M_1)p$$

$$T(t_x + u_x, t_y + u_y, t_z + u_z) = T(u_x, u_y, u_z)T(t_x, t_y, t_z)$$

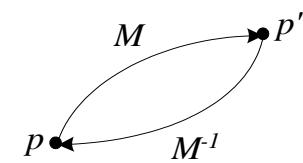


- 역변환(inverse transformation)과 역행렬(inverse matrix)

$$T^{-1}(t_x, t_y, t_z) = T(-t_x, -t_y, -t_z)$$

$$S^{-1}(s_x, s_y, s_z) = S\left(\frac{1}{s_x}, \frac{1}{s_y}, \frac{1}{s_z}\right)$$

$$R^{-1}(\alpha, n_x, n_y, n_z) = R(-\alpha, n_x, n_y, n_z)$$

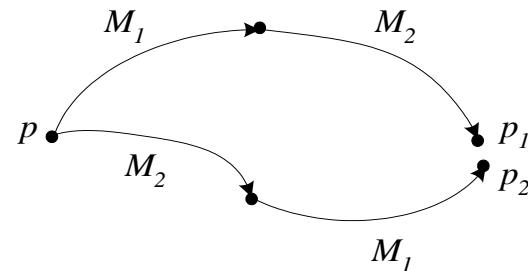


- 역변환의 존재와 행렬의 계수(rank)

- 변환의 순서와 행렬 곱셈의 교환 법칙

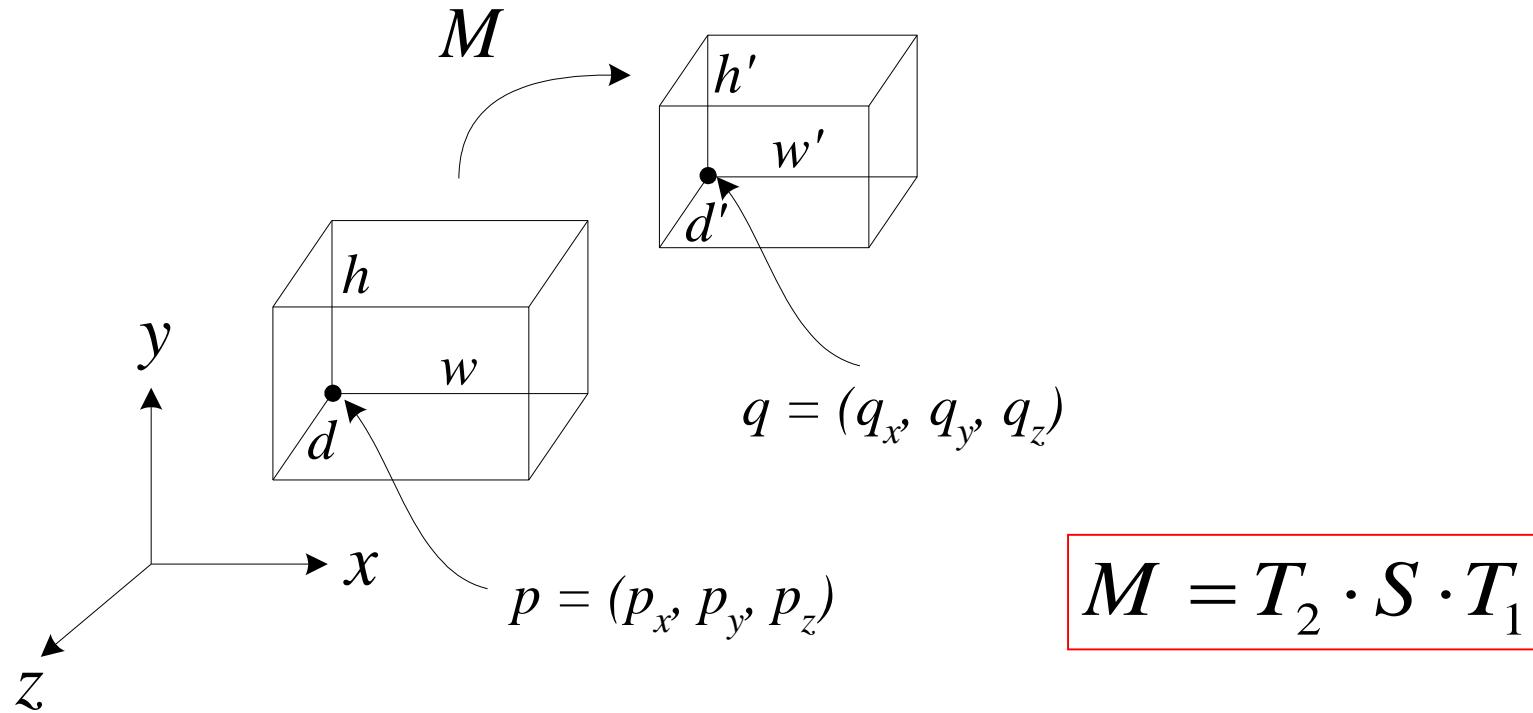
$$M_2 M_1 \stackrel{?}{=} M_1 M_2$$

When is a composition commutable?



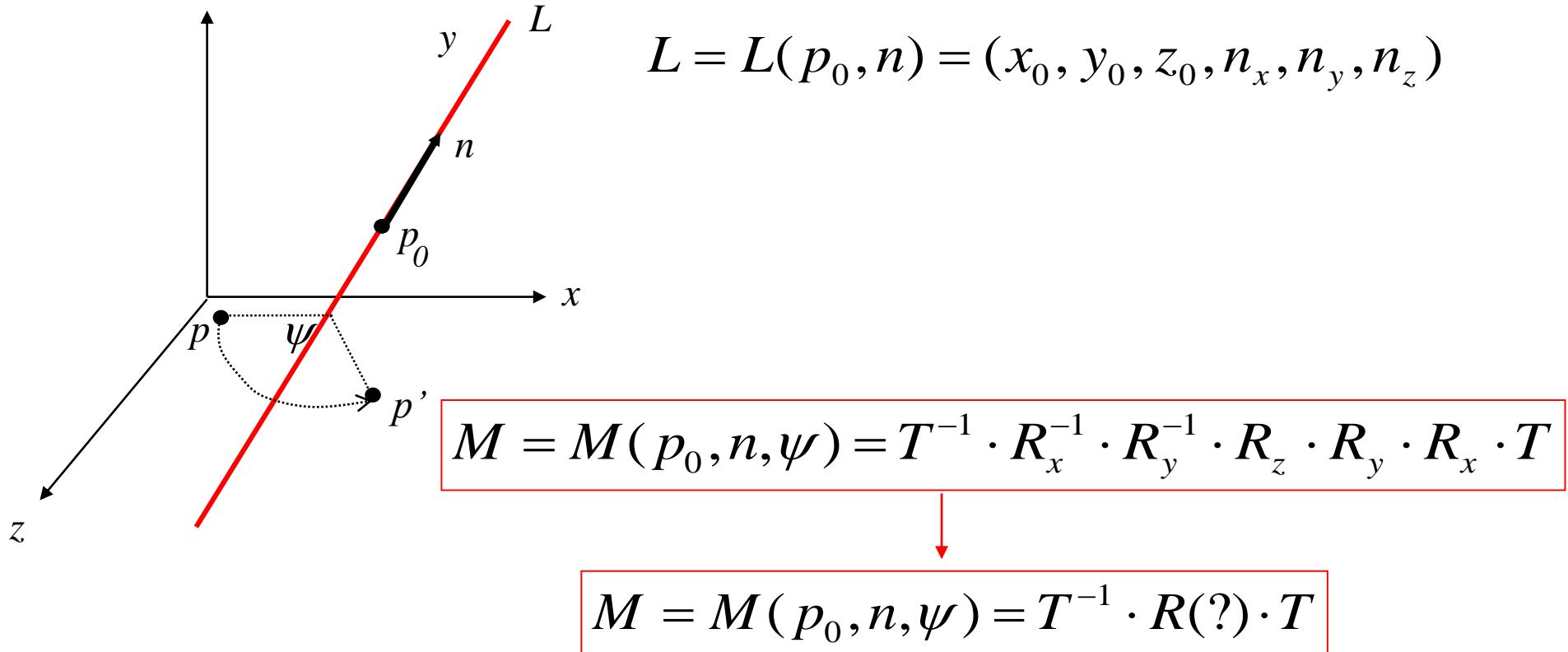
Example 1: Composition of 3D Transformation

- 문제: 두 육면체 간의 매핑



Example 2: Composition of 2D Transformation

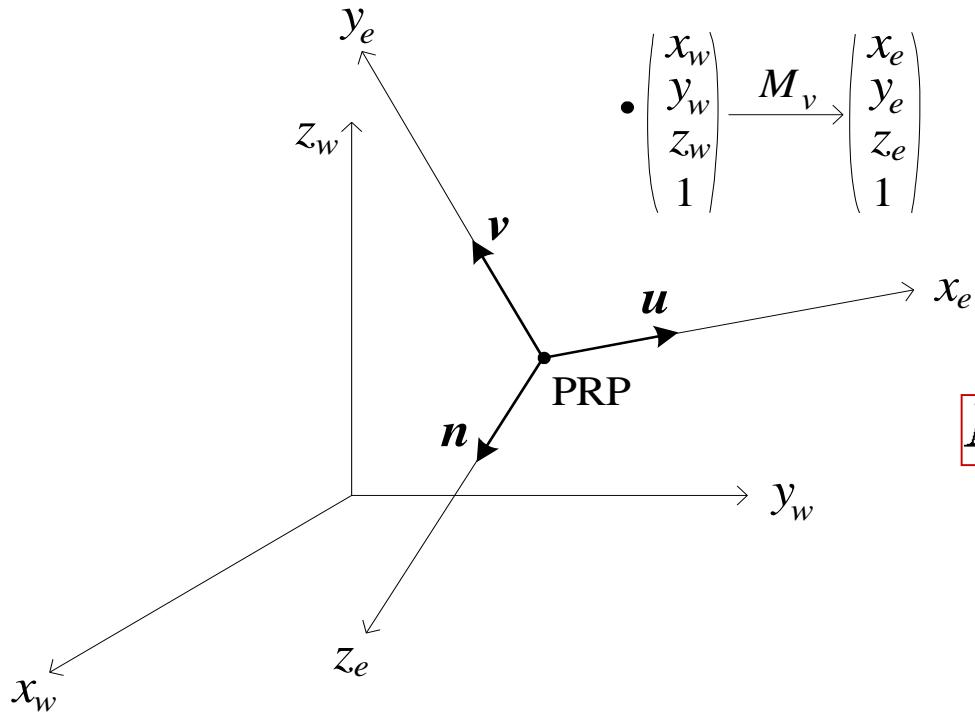
- 문제: 임의의 직선 둘레로의 회전 변환



- ☞ 회전 변환과 직교 행렬간의 관계
- ☞ 회전 변환 행렬들의 곱은 역시 회전임.

Example 3: Composition of 2D Transformation

- 문제: 두 좌표계간의 좌표 변환

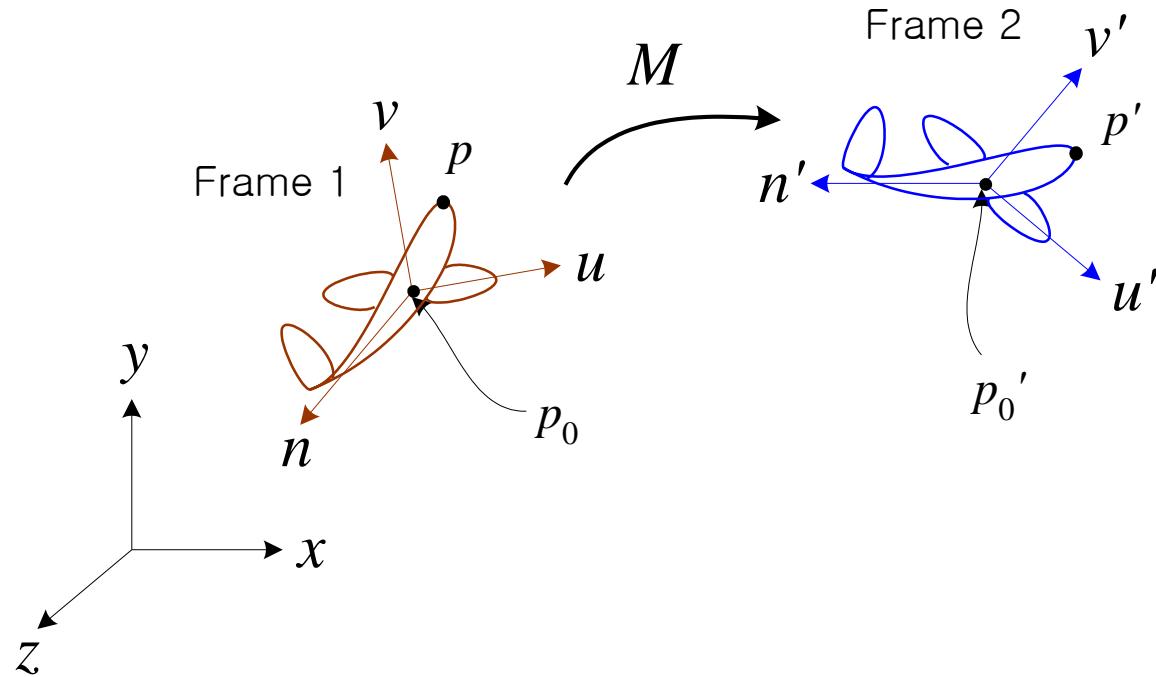


$$M_v = R \cdot T(-e_x, -e_y, -e_z)$$

$$M_v = \begin{bmatrix} u_x & u_y & u_z & 0 \\ v_x & v_y & v_z & 0 \\ n_x & n_y & n_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 & 0 & -e_x \\ 0 & 1 & 0 & -e_y \\ 0 & 0 & 1 & -e_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \equiv R \cdot T(-e_x, -e_y, -e_z)$$

Example 4: Composition of 2D Transformation

- 문제: 비행기의 이동



$$M = T(p'_0) \begin{bmatrix} u'_x & v'_x & n'_x & 0 \\ u'_y & v'_y & n'_y & 0 \\ u'_z & v'_z & n'_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} u_x & u_y & u_z & 0 \\ v_x & v_y & v_z & 0 \\ n_x & n_y & n_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} T(-p_0)$$

Reviews on Linear Algebra

Let V be a vector space over R .

1. Let W be a nonempty subset of V . W is a subspace if and only if $v_1, v_2 \in W \rightarrow \alpha_1 v_1 + \alpha_2 v_2 \in W$.
2. Given v_1, v_2, \dots, v_m from V , the smallest subspace of V containing all of the given vectors is called the subspace *spanned* by the v_i : $\text{Span}\{v_1, v_2, \dots, v_m\} = \{\sum_{i=1}^k \alpha_i v_i | \alpha_i \in R\}$.
3. Linear combination: $v = \alpha_1 v_1 + \alpha_2 v_2 + \dots + \alpha_m v_m$.
4. The elements v_1, v_2, \dots, v_m of V are said to be *linearly dependent* if there exist scalars α_i , not all zero, such that $\alpha_1 v_1 + \alpha_2 v_2 + \dots + \alpha_m v_m = 0$.
5. The v_i are *linearly independent* if they are not linearly dependent; that is, $\alpha_1 v_1 + \alpha_2 v_2 + \dots + \alpha_m v_m = 0$ only if $v_1 = v_2 = \dots = v_m = 0$.
6. A *basis* for V is a linearly independent set of vectors from V which spans V .
7. Dimension, Basis, Standard Basis, Change-of-Basis-Matrix, **Viewing Transformation**, Camera Setting

Rigid Body Transformation and Orthogonal Matrix

- 강체 변환(rigid body transformation)
 - 물체 구성 요소의 길이와 각도가 보존이 되는 변환. 즉 물체의 형태와 보존되는 변환.
 - 예: 비행기가 날라가는 변환
- 강체 변환에 대한 행렬의 성질
 - 두 벡터의 내적(inner product)과 그의 성질
$$u = (u_1 \ u_2 \ \cdots \ u_n)^t, \ v = (v_1 \ v_2 \ \cdots \ v_n)^t$$
$$u \cdot v \equiv u_1v_1 + u_2v_2 + \cdots + u_nv_n = |u||v| \cos \theta$$
 - 직교(orthogonal) 벡터
 - m 개의 n 차원 공간 벡터 v_1, v_2, \dots, v_m 에 대해 $v_i \cdot v_j = 0 \ \forall i \neq j$ 일 경우.
 - 정규직교(orthonormal) 벡터
 - m 개의 n 차원 공간 벡터 v_1, v_2, \dots, v_m 에 대해 다음 조건을 만족할 경우.
$$v_i \cdot v_j = 0 \ \forall i \neq j, \ v_i \cdot v_i = 1 \ \forall i, j$$
 - 기하학적인 의미

- 직교 행렬(orthogonal matrix)

- n 행 m 열 행렬 M 이 $M^{-1} = M^t$ 인 경우.
- 직관적 의미

- 강체 변환과 직교 행렬

- 3차원 공간에서의 임의의 아핀 변환에 해당하는 4행 4열 행렬 M 의 왼쪽 위의 3행 3열 부행렬이 직교 행렬이고 행렬식(determinant)이 +1일 경우 이 변환은 강체 변환임.
- 임의의 강체 변환에 대한 행렬은 다음과 같이 표현 가능.

$$M = T(t_x, t_y, t_z) \cdot R(\alpha, n_x, n_y, n_z) = \begin{pmatrix} r_{11} & r_{12} & r_{13} & t_x \\ r_{21} & r_{22} & r_{23} & t_y \\ r_{31} & r_{32} & r_{33} & t_z \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

- 역변환도 역시 강체 변환임.

$$\begin{aligned} M^{-1} &= (T(t_x, t_y, t_z) \cdot R(\alpha, n_x, n_y, n_z))^{-1} = R(\alpha, n_x, n_y, n_z)^{-1} \cdot T(t_x, t_y, t_z)^{-1} \\ &= R(-\alpha, n_x, n_y, n_z) \cdot T(-t_x, -t_y, -t_z) \end{aligned}$$

$$R(\alpha, n_x, n_y, n_z)^{-1} = R(-\alpha, n_x, n_y, n_z) = R(\alpha, n_x, n_y, n_z)^t$$

- 다음을 생각해볼 것.

$$M^{-1} = T(t_x^*, t_y^*, t_z^*) \cdot R(\alpha, n_x, n_y, n_z)^t, \quad \text{where } \begin{pmatrix} t_x^* \\ t_y^* \\ t_z^* \end{pmatrix} = -R_{3 \times 3}^t \begin{pmatrix} t_x \\ t_y \\ t_z \end{pmatrix}$$

Affine Transformation for Point, Vector, and Plane

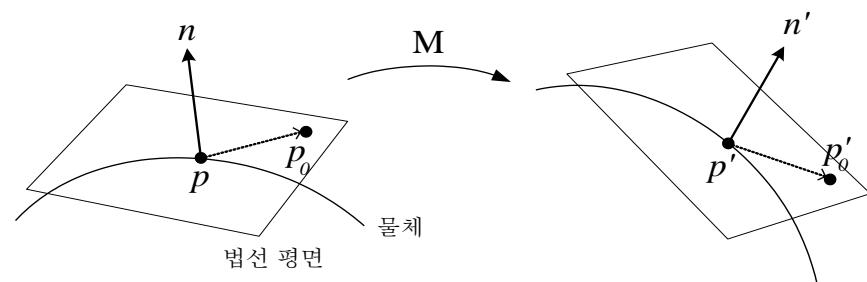
- 점(point)에 대한 변환

$$\begin{pmatrix} x' \\ y' \\ z' \\ 1 \end{pmatrix} = M \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}$$

- 주어진 점에 대한 아핀 변화 M 을 상이한 부류의 기하 개념인 벡터와 평면에 적용할 때에는 그에 맞는 방식을 사용해야 함!

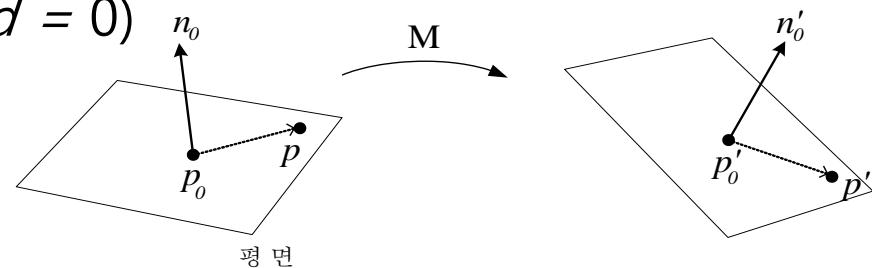
- 벡터(vector)에 대한 변환

$$\begin{pmatrix} n'_x \\ n'_y \\ n'_z \\ 0 \end{pmatrix} = (M^{-1})^t \begin{pmatrix} n_x \\ n_y \\ n_z \\ 0 \end{pmatrix}$$



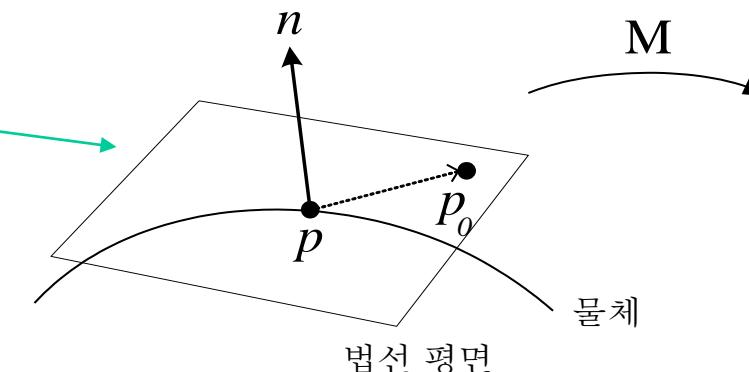
- 평면(plane)에 대한 변환 ($ax + by + cz + d = 0$)

$$(a' \ b' \ c' \ d') = (a \ b \ c \ d) \cdot M^{-1}$$



벡터에 대한 변환 공식 유도

$$\begin{aligned} p &= (x \ y \ z \ 1)^t \\ p_0 &= (x_0 \ y_0 \ z_0 \ 1)^t \\ n &= (n_x \ n_y \ n_z \ 0)^t \\ n^t \cdot (p_0 - p) &= 0 \end{aligned}$$



$$\begin{aligned} n^t \cdot (M^{-1} \cdot p'_0 - M^{-1} \cdot p') &= (n^t \cdot M^{-1}) \cdot (p'_0 - p') \\ &= \underbrace{\{(M^{-1})^t \cdot n\}^t}_{\text{이 벡터가 } n'\text{이어야 함}} \cdot (p'_0 - p') \\ &= 0 \end{aligned}$$

$$\begin{aligned} p' &= (x' \ y' \ z' \ 1)^t \\ p'_0 &= (x'_0 \ y'_0 \ z'_0 \ 1)^t \\ n' &= (n'_x \ n'_y \ n'_z \ 0)^t \\ p' &= M \cdot p \\ p'_0 &= M \cdot p_0 \end{aligned}$$

$$n' = (M^{-1})^t \cdot n$$

- 만약 M 이 아핀 변환이라면

$$M = \begin{bmatrix} M_{33} & v_t \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad \rightarrow \quad M^{-1} = \begin{bmatrix} M_{33}^{-1} & -M_{33}^{-1}v_t \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

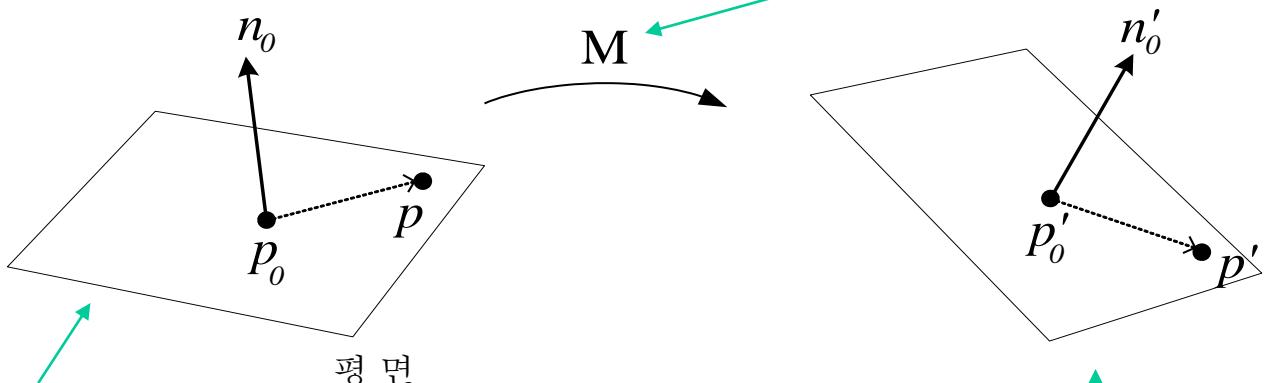
$$\begin{pmatrix} n'_x \\ n'_y \\ n'_z \end{pmatrix} = (M_{33}^{-1})^t \begin{pmatrix} n_x \\ n_y \\ n_z \end{pmatrix}$$

- 만약 M 이 강체 변환이라면

$$(M_{33}^{-1})^t = M_{33} \text{ 이므로} \quad \begin{pmatrix} n'_x \\ n'_y \\ n'_z \end{pmatrix} = M_{33} \begin{pmatrix} n_x \\ n_y \\ n_z \end{pmatrix}$$

평면에 대한 아핀 변환 공식 유도

$$M = \begin{bmatrix} M_{33} & v_t \\ 0 & 0 & 0 & 1 \end{bmatrix}$$



$$a \cdot x + b \cdot y + c \cdot z + d = 0$$

$$p_0 = (x_0 \ y_0 \ z_0)^t$$

$$n_0 = (n_{0x} \ n_{0y} \ n_{0z})^t \rightarrow n_0^t \cdot (p - p_0) = n_0^t \cdot p + (-n_0^t \cdot p_0) = 0$$

$$p = (x \ y \ z)^t$$

$$a' \cdot x + b' \cdot y + c' \cdot z + d' = 0$$

$$p'_0 = (x'_0 \ y'_0 \ z'_0)^t \quad p'_0 = M_{33} \cdot p_0 + v_t$$

$$n'_0 = (n'_{0x} \ n'_{0y} \ n'_{0z})^t \rightarrow$$

$$p' = (x' \ y' \ z')^t$$

$$n'_0 = (M_{33}^{-1})^t \cdot n$$

$$(a' \ b' \ c') = n_0'^t$$

$$d' = -n_0'^t \cdot p_0'$$

이므로

$$\begin{aligned}
 (a' \ b' \ c' &\quad d') = (n_0'^t & -n_0'^t \cdot p_0') \\
 &= (n_0^t \cdot M_{33}^{-1} & -n_0^t \cdot M_{33}^{-1} \cdot (M_{33} \cdot p_0 + v_t)) \\
 &= (n_0^t \cdot M_{33}^{-1} & -n_0^t \cdot p_0 - n_0^t \cdot M_{33}^{-1} \cdot v_t) \\
 &= (n_0^t & -n_0^t \cdot p_0) \cdot \begin{bmatrix} M_{33}^{-1} & -M_{33}^{-1} \cdot v_t \\ 0 & 0 & 0 & 1 \end{bmatrix} \\
 &= (a \ b \ c & d) \cdot \begin{bmatrix} M_{33}^{-1} & -M_{33}^{-1} \cdot v_t \\ 0 & 0 & 0 & 1 \end{bmatrix} \\
 &= (a \ b \ c & d) \cdot M^{-1}
 \end{aligned}$$

Geometric Transformation in OpenGL

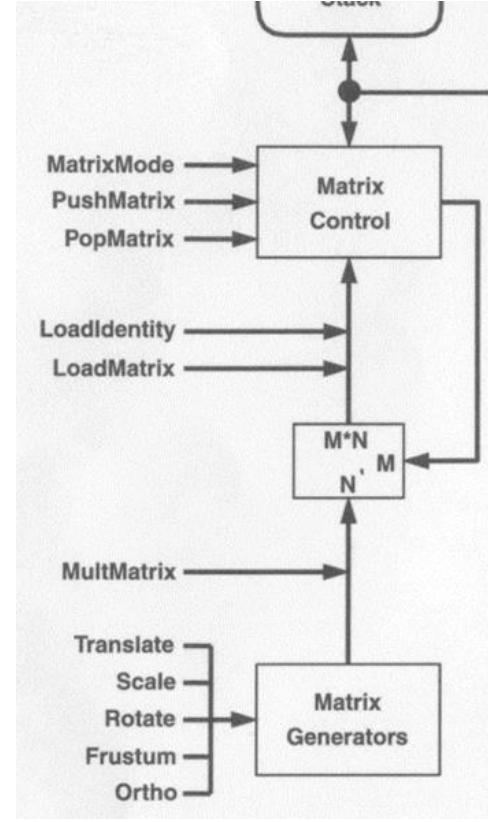
- Compatibility Profile

OpenGL의 기하 계산 관련 함수

- OpenGL의 행렬 스택
 - 모델뷰 행렬 스택(ModelView matrix stack),
투영 행렬 스택(Projection matrix stack),
텍스춰 행렬 스택(Texture matrix stack),
색깔 행렬 스택(Color matrix stack) 등
 - 스택의 각 원소는 4행 4열 행렬을 저장.
 - 앞의 세 스택의 경우 각 원소에는 3차원 변환 행렬을 저장.

$$m = \begin{bmatrix} m[0] & m[4] & m[8] & m[12] \\ m[1] & m[5] & m[9] & m[13] \\ m[2] & m[6] & m[10] & m[14] \\ m[3] & m[7] & m[11] & m[15] \end{bmatrix}$$

- OpenGL의 기하 계산 관련 함수는 대부분 행렬 스택을 지정하거나 스택의 탑에 있는 원소에 영향을 미치는 방식으로 작동.
- 스택 (stack)
 - 대표적인 LIFO(Last-In-First-Out) 지원 자료 구조
 - 푸쉬(push)와 팝(pop) 연산, 스택의 탑(top)
 - 그래픽스 프로그래밍의 방식이 LIFO 구조와 밀접한 관계가 있음.
 - 그래픽스 프로그래밍에 쓰이는 스택은 일반 스택과는 약간 다른 방식으로 동작.

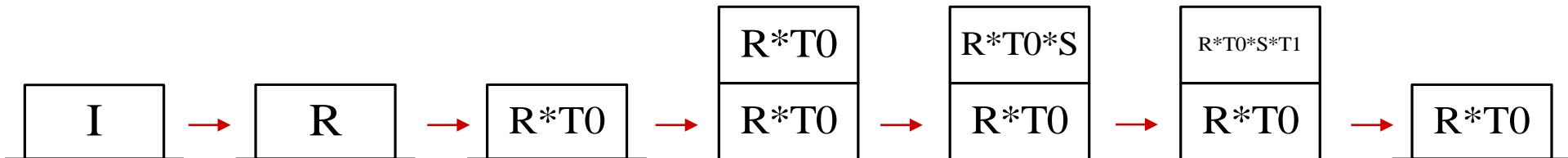


OpenGL 기하 파이프라인 관련 주요 함수 (Compatibility Profile)

OpenGL 함수	용도
<code>Void glMatrixMode(GLenum mode);</code>	
<code>void glLoadMatrixf(const GLfloat *m);</code>	
<code>void glLoadIdentity(void);</code>	
<code>void glMultMatrixf(const GLfloat *m);</code>	
<code>void glTranslatef(GLfloat x, GLfloat y, GLfloat z);</code>	
<code>void glScalef(GLfloat x, GLfloat y, GLfloat z);</code>	
<code>void glRotatef(GLfloat angle, GLfloat x, GLfloat y, GLfloat z);</code>	
<code>void glOrtho(GLdouble left, GLdouble right, GLdouble bottom, GLdouble top, GLdouble zNear, GLdouble zFar);</code>	
<code>void glFrustum(GLdouble left, GLdouble right, GLdouble bottom, GLdouble top, GLdouble zNear, GLdouble zFar);</code>	
<code>void gluPerspective(GLdouble fovy, GLdouble aspect, GLdouble zNear, GLdouble zFar);</code>	
<code>void gluLookAt(GLdouble ex, GLdouble ey, GLdouble ez, GLdouble cx, GLdouble cy, GLdouble cz, GLdouble ux, GLdouble uy, GLdouble uz);</code>	
<code>void glViewport(GLint x, GLint y, GLsizei width, GLsizei height);</code>	
<code>void glDepthRange(GLclampd zNear, GLclampd zFar);</code>	
<code>void glPushMatrix(void);</code>	
<code>void glPopMatrix(void);</code>	

OpenGL 스택의 사용 예

```
glMatrixMode(GL_MODELVIEW) ; // Line (a)
glLoadIdentity() ; // Line (b) : I = unit matrix
glRotatef(90.0, 0.0, 1.0, 0.0) ; // Line (c) : R = R(90, 0, 1, 0)
glTranslatef(2.0, 0.0, 0.0) ; // Line (d) : T0 = T(2, 0, 0)
glPushMatrix() ; // Line (e)
glScalef(2.0, 2.0, 2.0) ; // Line (f) : S = S(2, 2, 2)
glTranslatef(-2.0, 3.0, 0.0) ; // Line (g) : T1 = T(-2, 3, 0)
draw_object();
glPopMatrix() ; // Line (h)
```



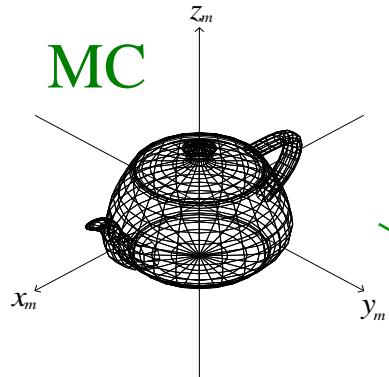
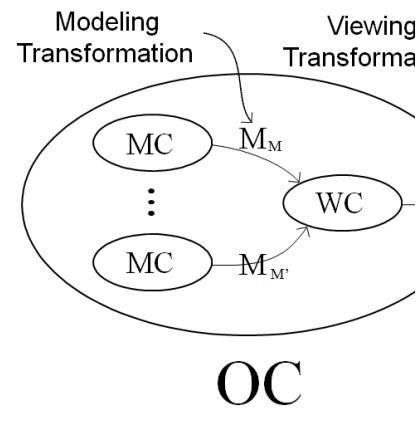
모델링 좌표계와 모델링 변환

- 모델링 좌표계(Modeling Coordinates, MC)

- 가상의 세상에 존재하는 각 물체들이 정의가 되는 좌표계

- 모델링 변환(Modeling Transformation)

- 모델링 좌표계에서 정의된 물체를 가상의 세상 공간인 세상 좌표계로 보내 수는 변환.



$$M_M = T(0.0, 0.0, 1.5)$$

EC

VUP

EC

z_e

y_e

x_e

E

z_e

y_e

x_e

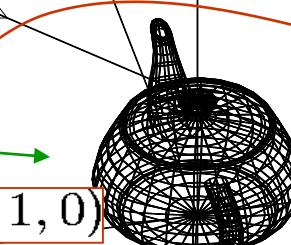
시선 방향



VRP

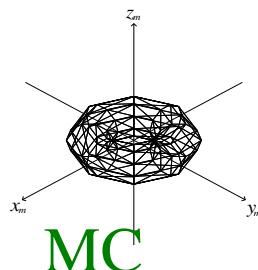
VRP

VRP



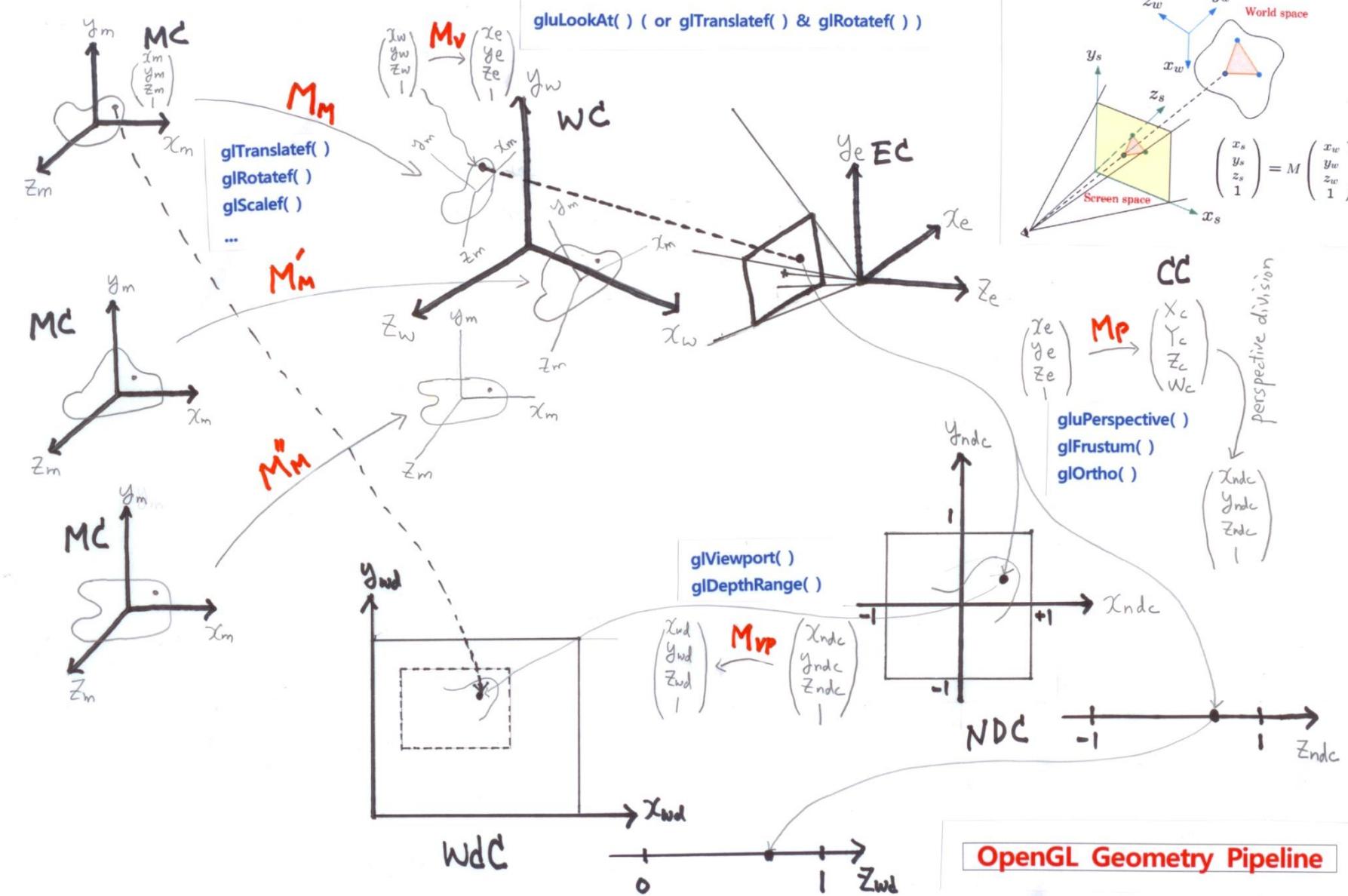
WC

$$M_M = T(0, -6, 0) \cdot R(-45, 0, 1, 0) \cdot S(1, 1, -1) \cdot R(45, 0, 1, 0)$$



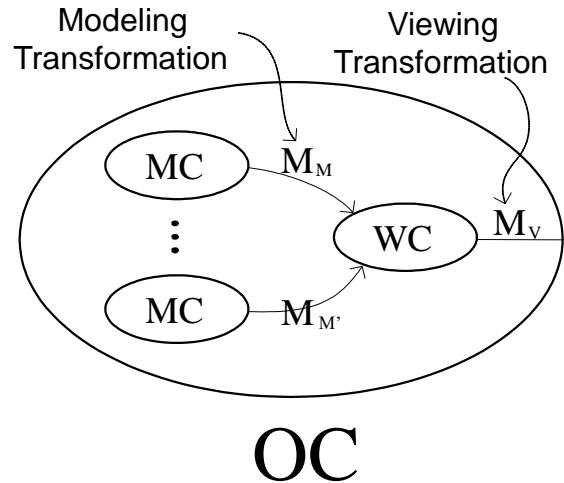
$$M_M = T(-6, -5, 1) \cdot S(2, 2, 2)$$

OpenGL Geometry Pipeline Overview



Geometry Stage in Fixed-Function OpenGL Pipeline

Geometry stage의 골격을 이루는 geometry pipeline을 이해하자.



Basic computations in geometry stage

- Model and view transform
- **Lighting**
- **Primitive assembly**
- Projection transform
- **View volume clipping**
- Screen mapping
- **Polygon culling**
- Etc.

Per-vertex lighting Primitive assembly User-defined clipping

View volume clipping

최신 GPU에서는 geometry stage의 어느 부분이 vertex shader 및 geometry shader 형태로 프로그램이 가능할까?

OpenGL을 통한 3차원 모델링 변환 예

```
void draw_world(void) {  
    draw_axes_in_WC();  
  
    glMatrixMode(GL_MODELVIEW);  
  
    glColor3f(1.0, 1.0, 0.0);  
    glPushMatrix();  
    glTranslatef(0.0, 0.0, 1.5);  
    draw_object_in_MC(teapot, npolytp); // draw teapot  
    glPopMatrix();  
  
    glColor3d(1.0, 0.0, 1.0);  
    glPushMatrix();  
    glTranslatef(-6.0, -5.0, 1.0);  
    glScalef(2.0, 2.0, 2.0);  
    draw_object_in_MC(donut, npolydn); // draw big  
                                     donut  
    glPopMatrix();
```

```
glColor3d(0.0, 1.0, 1.0);  
glPushMatrix();  
glTranslatef(0.0, -6.0, 0.0);  
glRotatef(-45.0, 0.0, 1.0, 0.0); // x<->z  
glScalef(1.0, 1.0, -1.0);  
glRotatef(45.0, 0.0, 1.0, 0.0);  
draw_object_in_MC(donut, npolydn); // draw  
small donut  
glPopMatrix();  
}
```

$$M_V \cdot T(0, 0, 1.5)$$

$$M_V \cdot T(-6, -5, 1) \cdot S(2, 2, 2)$$

모델뷰 행렬 스택 탑의 내용

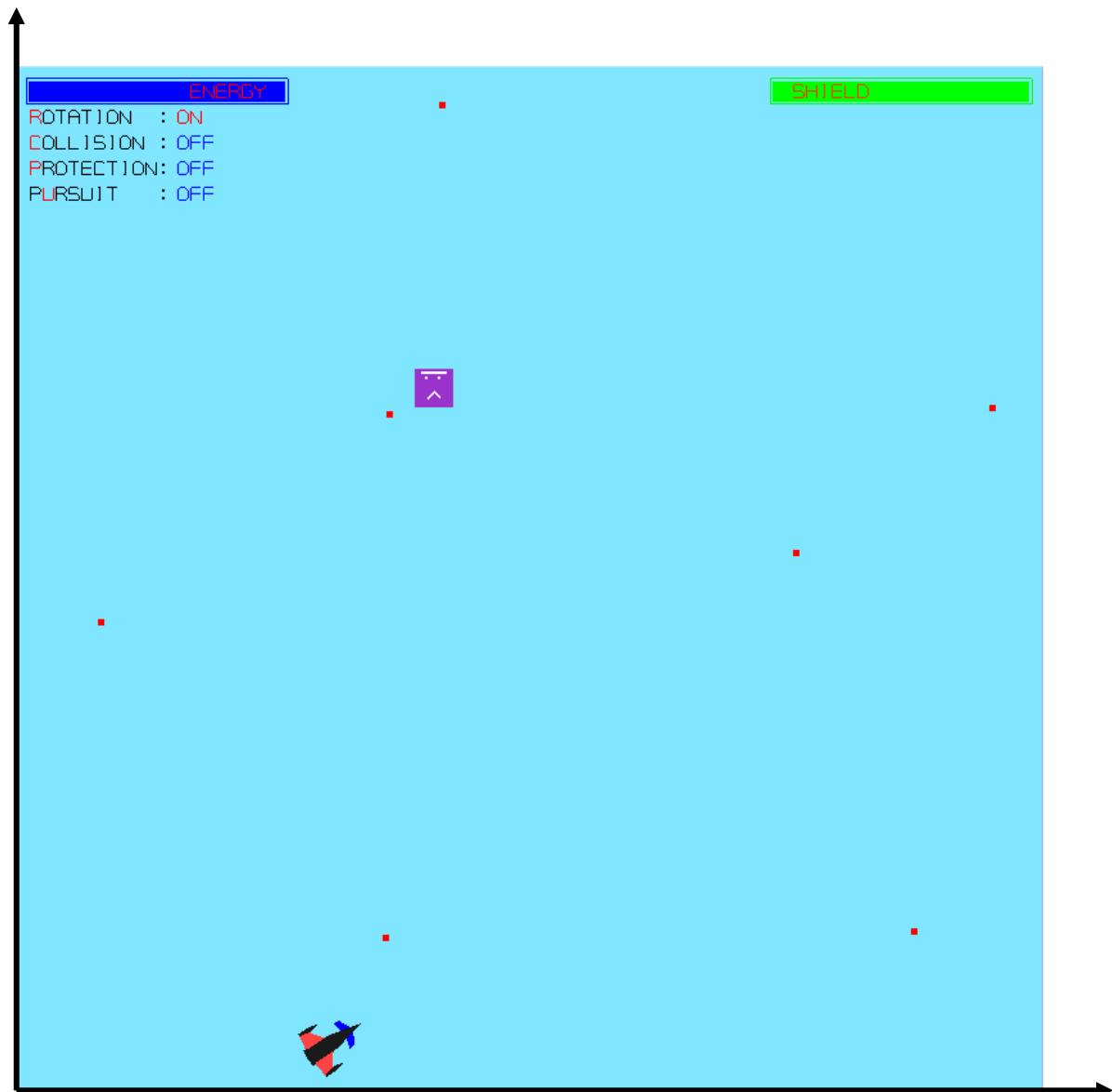
$$M_V \cdot T(0, -6, 0) \cdot R(-45, 0, 1, 0) \cdot S(1, 1, -1) \cdot R(45, 0, 1, 0)$$

```
void draw_object_in_MC(polygon *object, int npoly) {
    int i, j;

    for (i = 0; i < npoly; i++) {
        glBegin(GL_POLYGON);
        for (j = 0; j < object[i].nvertex; j++) {
            glVertex3f(object[i].poly[j][0], object[i].poly[j][1], object[i].poly[j][2]);
        }
        glEnd();
    }
}
```

OpenGL을 통한 2차원 모델링 변환 예 1

```
void display(void) {  
    glClear(GL_COLOR_BUFFER_BIT);  
  
    glPushMatrix();  
    glTranslatef(airx, airy, 0.0);  
    glScalef(scale, scale, 0.0);  
    glRotatef(rotate, 0.0, 0.0, 1);  
    draw_airplane(); // Airplane  
    glPopMatrix();  
  
    glPushMatrix();  
    glTranslatef(monx, mony, 0.0);  
    draw_monster(); // Monster  
    glPopMatrix();  
  
    glPushMatrix();  
    draw_missile(); // Missile  
    glPopMatrix();  
  
    ...  
    glutSwapBuffers();  
}
```



```
void draw_monster() {
    glColor3f(0.6, 0.2, 0.8);
    glLineWidth(2.0);

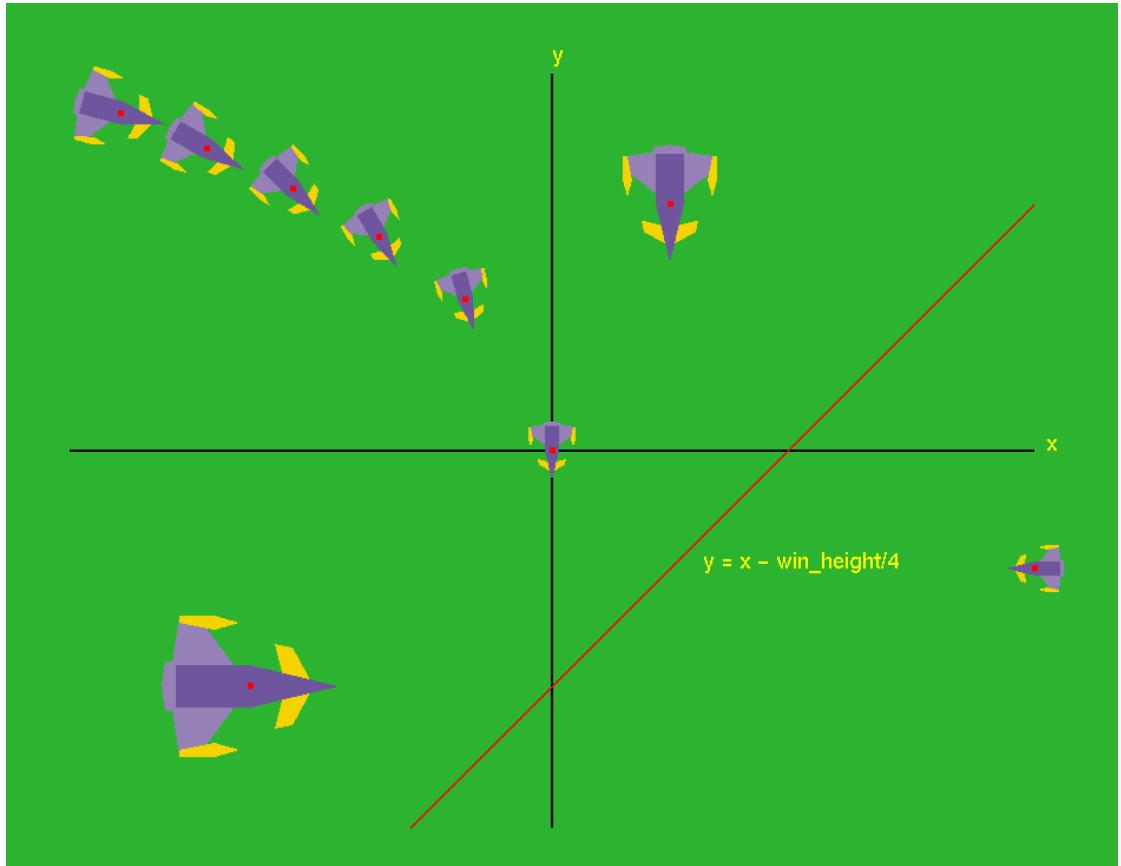
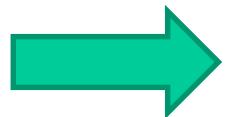
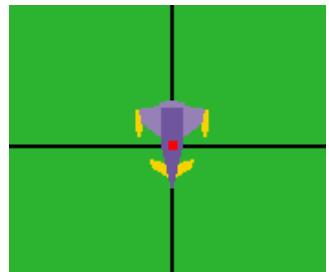
    glBegin(GL_POLYGON);
        glVertex2i(15, -15); glVertex2i(-15, -15);
        glVertex2i(-15, 15); glVertex2i(15, 15);
    glEnd();
    glColor3f(1.0, 1.0, 1.0);
    if (indanger) {
        glBegin(GL_LINES);
            glVertex2i(-10, -12); glVertex2f(10, -12);
            glVertex2i(-5, 8); glVertex2i(0, 3);
            glVertex2i(0, 3); glVertex2i(5, 8);
        glEnd();
    }
    else {
        glBegin(GL_LINES);
            glVertex2i(-10, -12); glVertex2f(10, -12);
            glVertex2i(-6, 3); glVertex2i(0, 8);
            glVertex2i(0, 8); glVertex2i(6, 3);
        glEnd();
    }
    glPointSize(2.0);
    glBegin(GL_POINTS);
        glVertex2i(-6, -8); glVertex2i(4, -8);
    glEnd();
}
```

```
void draw_airplain() {
    ...
}

void draw_missile() {
    ...
}
```

OpenGL을 통한 2차원 모델링 변환 예 II

- 모델링 좌표계(MC)를 기준으로 주어진 비행기 데이터에 대해 다음과 같은 그림을 그려보자.



```

void display (void) {
    float x, r, s, delx, delr, dels;

    glClear(GL_COLOR_BUFFER_BIT);
    draw_axes();
    draw_text();
    draw_line();

    draw_airplane(); // 0

    glPushMatrix();
    glTranslatef(-win_width/4.0,
                 -win_height/4.0, 0.0);
    glRotatef(90.0, 0.0, 0.0, 1.0);
    glScalef(3.0, 3.0, 1.0);
    draw_airplane(); // 1
    glPopMatrix();

    glPushMatrix();
    glTranslatef(win_width/2.5,
                -win_height/8.0, 0.0);
    glRotatef(270.0, 0.0, 0.0, 1.0);
    draw_airplane(); // 2
    glPopMatrix();
}

```

```

glPushMatrix();
glTranslatef(win_height/4.0, 0.0, 0.0);
glRotatef(45.0, 0.0, 0.0, 1.0);
glScalef(1.0, -1.0, 1.0);
glRotatef(-45.0, 0.0, 0.0, 1.0);
glTranslatef(-win_height/4.0, 0.0, 0.0);
glTranslatef(win_width/2.5, -win_height/8.0, 0.0);
glScalef(2.0, 2.0, 1.0);
glTranslatef(-win_width/2.5, win_height/8.0, 0.0);

glTranslatef(win_width/2.5, -win_height/8.0, 0.0);
glRotatef(270.0, 0.0, 0.0, 1.0);
draw_airplane(); // 3
glPopMatrix();

delx = win_width/14.0; delr = 15.0; dels = 1.1;
x = -delx; r = delr; s = dels;
for (int i = 0; i < 5; i++, x -= delx, r += delr, s *= dels) {
    glPushMatrix();
    glTranslatef(x, 15.0*sqrt((double) -x), 0.0);
    glRotatef(r, 0.0, 0.0, 1.0);
    glScalef(s, s, 1.0);
    draw_airplane(); // 4
    glPopMatrix();
}
glFlush();
}

```

Simplification of Geometric Transformations

```
glPushMatrix();
glTranslatef(win_height/4.0, 0.0, 0.0);
glRotatef(45.0, 0.0, 0.0, 1.0);
glScalef(1.0, -1.0, 1.0);
glRotatef(-45.0, 0.0, 0.0, 1.0);
glTranslatef(-win_height/4.0, 0.0, 0.0);
glTranslatef(win_width/2.5, -win_height/8.0, 0.0);
glScalef(2.0, 2.0, 1.0);
glTranslatef(-win_width/2.5, +win_height/8.0, 0.0);

glTranslatef(win_width/2.5, -win_height/8.0, 0.0);
glRotatef(270.0, 0.0, 0.0, 1.0);
draw_airplane(); // 3
glPopMatrix();
```

$$T(t_x, t_y)S(1, -1) = S(1, -1)T(?, ?)$$
$$R(\theta)S(1, -1) = S(1, -1)R(?)$$
$$R(\theta)T(t_x, t_y) = T(?, ?)R(\theta)$$
$$T(t_x, t_y)R(\theta) = R(\theta)T(?, ?)$$

어떻게 위의 기하 변환을 세 개의 기하 변환의 합성으로 표현할 수 있을까?

```
glPushMatrix();
glTranslatef(win_height/8.0, win_width/2.5-win_height/4.0, 0.0);
glRotatef(180.0, 0.0, 0.0, 1.0);
glScalef(2.0, -2.0, 1.0);
draw_airplane(); // 3
glPopMatrix();
```

Geometric Transformation in OpenGL

- Core Profile

GLM (OpenGL Mathematics)

- A header only, **free-software/open-source C++ mathematics library for graphics software based on the OpenGL Shading Language (GLSL)**
 - Provides classes and functions designed and implemented with the same naming conventions and functionalities than GLSL so that when a programmer knows GLSL, he knows GLM as well which makes it really easy to use.
 - Released under the MIT license.

➤ 참고

- Shader를 사용하는 방식인 OpenGL Core Profile에서는 기존의 Compatibility Profile에서와는 달리 행렬 및 기하 계산에 관련된 gl 함수들이 모두 제거됨.
 - glLoadIdentity(*), glPushMatrix(), glTranslatef(*), glOrtho(*), glFrustum(*), ect.
 - 따라서 이에 대한 부분을 glm과 같은 부류의 수학 전용 라이브러리를 사용하여 수행해야 함.
 - 설치 방법은 조교가 설명할 예정임.
-
- 정확한 함수 사용을 위하여 매뉴얼 "glm Manual (Version 0.9.7)"을 참조할 것.

GLM Core Features

- Core GLM features can be included
 - either using `<glm/glm.hpp>`
 - or using individual headers to allow faster user program compilations.
- ✓ Including `<glm/glm.hpp>` provides all the GLSL features implemented by GLM.

헤더 파일 이름	용도
<code><glm/vec2.hpp></code>	<code>vec2, bvec2, dvec2, ivec2</code> and <code>uvec2</code>
<code><glm/vec3.hpp></code>	<code>vec3, bvec3, dvec3, ivec3</code> and <code>uvec3</code>
<code><glm/vec4.hpp></code>	<code>vec4, bvec4, dvec4, ivec4</code> and <code>uvec4</code>
<code><glm/mat2x2.hpp></code>	<code>mat2, dmat2</code>
<code><glm/mat2x3.hpp></code>	<code>mat2x3, dmat2x3</code>
<code><glm/mat2x4.hpp></code>	<code>mat2x4, dmat2x4</code>
<code><glm/mat3x2.hpp></code>	<code>mat3x2, dmat3x2</code>
<code><glm/mat3x3.hpp></code>	<code>mat3, dmat3</code>
<code><glm/mat3x4.hpp></code>	<code>mat3x4, dmat2</code>

GLM Core Features (계속)

헤더 파일 이름	용도
<code><glm/mat4x2.hpp></code>	mat4x2, dmat4x2
<code><glm/mat4x3.hpp></code>	mat4x3, dmat4x3
<code><glm/mat4x4.hpp></code>	mat4, dmat4
<code><glm/common.hpp></code>	all the GLSL common functions
<code><glm/exponential.hpp></code>	all the GLSL exponential functions
<code><glm/geometry.hpp></code>	all the GLSL geometry functions
<code><glm/integer.hpp></code>	all the GLSL integer functions
<code><glm/matrix.hpp></code>	all the GLSL matrix functions
<code><glm/packing.hpp></code>	all the GLSL packing functions
<code><glm/trigonometric.hpp></code>	all the GLSL trigonometric functions
<code><glm/vector_relational.hpp></code>	all the GLSL vector relational functions

GLM Extensions

- GLM extends the core GLSL feature set with extensions, including quaternion, transformation, spline, matrix inverse, color spaces, etc.
- ✓ To include an extension, include only the dedicated header file.
 - Once included, the features are added to the GLM namespace.

헤더 파일 이름	용도
<code><glm/gtc/bitfield.hpp></code>	Fast bitfield operations on scalar and vector variables
<code><glm/gtc/color_space.hpp></code>	Conversion between linear RGB to sRGB and sRGB to linear RGB
<code><glm/gtc/constants.hpp></code>	Provide a list of built-in constants
<code><glm/gtc/epsilon.hpp></code>	Approximate equal and not equal comparisons with selectable epsilon
<code><glm/gtc/integer.hpp></code>	Provide integer variants of GLM core functions
<code><glm/gtc/matrix_access.hpp></code>	Define functions to access rows or columns of a matrix easily
<code><glm/gtc/matrix_integer.hpp></code>	Provide integer matrix types
<code><glm/gtc/matrix_inverse.hpp></code>	Define additional matrix inverting functions
<code><glm/gtc/matrix_transform.hpp></code>	Define functions that generate common transformation matrices.

GLM Extensions (계속)

헤더 파일 이름	용도
<code><glm/gtc/noise.hpp></code>	Define 2D, 3D and 4D procedural noise functions
<code><glm/gtc/packing.hpp></code>	Convert scalar and vector types to packed formats
<code><glm/gtc/quaternion.hpp></code>	Define a quaternion type and several quaternion operations
<code><glm/gtc/random.hpp></code>	Generate random number from various distribution methods
<code><glm/gtc/reciprocal.hpp></code>	Provide hyperbolic functions: secant, cosecant, cotangent, etc
<code><glm/gtc/round.hpp></code>	Rounding operation on power of two and multiple values
<code><glm/gtc/type_precision.hpp></code>	Add vector and matrix types with defined precisions
<code><glm/gtc/type_ptr.hpp></code>	Handle the interaction between pointers and vector, matrix types
<code><glm/gtc/ulp.hpp></code>	Allow the measurement of the accuracy of a function against a reference implementation
<code><glm/gtc/vec1.hpp></code>	Add *vec1 types

효율적인 헤더 파일의 삽입

- GLM makes a heavy usage of C++ templates, significantly increasing the compile time for including header files.
 - If possible, limit inclusion to header and source files that actually use it.

```
// Include GLM core features
#include <glm/vec3.hpp>
#include <glm/vec4.hpp>
#include <glm/mat4x4.hpp>
// Include GLM extensions
#include <glm/gtc/matrix_transform.hpp>
glm::mat4 transform(glm::vec2 const & Orientation, glm::vec3 const & Translate, glm::vec2 const & Up) {
    glm::mat4 Projection = glm::perspective(45.0f, 4.0f / 3.0f, 0.1f, 100.0f);
    glm::mat4 ViewTranslate = glm::translate(glm::mat4(1.0f), Translate);
    glm::mat4 ViewRotateX = glm::rotate(ViewTranslate, Orientation.y, Up);
    glm::mat4 View = glm::rotate(ViewRotateX, Orientation.x, Up);
    glm::mat4 Model = glm::mat4(1.0f);
    return Projection * View * Model;
}
```

- 참고: <glm/gtc/matrix_transform.hpp>는 glm의 core feature 사용을 위하여 다음 header file 등을 삽입함.

- <glm/vec3.hpp>, <glm/vec4.hpp>, <glm/mat4x4.hpp>, etc.

GLM 제공 벡터/행렬 타입

종류	제공 타입 (여기서는 float 타입만 고려)	정의 헤더*
2D vector	glm::vec2	<glm/vec2.hpp>
3D vector	glm::vec3	<glm/vec3.hpp>
4D vector	glm::vec4	<glm/vec4.hpp>
2x2 matrix	glm::mat2	<glm/mat2x2.hpp>
2x3 matrix	glm::mat2x3	<glm/mat2x3.hpp>
2x4 matrix	glm::mat2x4	<glm/mat2x4.hpp>
3x2 matrix	glm::mat3x2	<glm/mat3x2.hpp>
3x3 matrix	glm::mat3	<glm/mat3x3.hpp>
3x4 matrix	glm::mat3x4	<glm/mat3x4.hpp>
4x2 matrix	glm::mat4x2	<glm/mat4x2.hpp>
4x3 matrix	glm::mat4x3	<glm/mat4x3.hpp>
4x4 matrix	glm::mat4	<glm/mat4x4.hpp>

* 이 타입들은 glm의 core feature로서 개별적으로 헤더를 삽입해도 되고, <glm/glm.hpp>을 삽입하면 모든 타입이 한 번에 정의됨. 또한 다른 헤더, 예를 들어, <glm/gtc/matrix_transform.hpp>를 삽입하면 그에 필요한 벡터/행렬 관련 헤더가 같이 삽입됨.

OpenGL에서의 벡터와 행렬 개념

```
vec3 velocity = vec3 ( 0.0 , 2.0 , 3.0 ) ; // initialize a vec3 with 3 floats  
ivec steps = ivec3 ( velocity ) ; // convert vec3 to ivec  
  
vec4 color;  
vec3 RGB = vec3 ( color ) ; // color vector is truncated so RGB only has three elements  
vec3 white = vec3 ( 1.0 ) ; // white = ( 1.0 , 1.0 , 1.0 )  
vec4 translucent = vec4 ( white, 0.5 ) ; // translucent = ( 1.0, 1.0, 1.0, 0.5 )  
  
mat3 identity = mat3 ( 1.0 ) ; // identity matrix  
mat2 diagonal = mat2 ( 2.0 ) ; // diagonal matrix  
// matrices are specified in column-major order  
mat3 M = mat3 ( 1.0, 2.0, 3.0,    // first column  
                  4.0, 5.0, 6.0,    // second column  
                  7.0, 8.0, 9.0 ) ; // third column  
Vec3 column1 = vec3 ( 1.0, 2.0, 3.0 ) ;  
Vec3 column2 = vec3 ( 4.0, 5.0, 6.0 ) ;  
Vec3 column3 = vec3 ( 7.0, 8.0, 9.0 ) ;  
M = mat3 ( column1, column2, column3 ); // compose a matrix of three columns
```

Transparent Types

void	no function return value
bool	Boolean
int, uint	signed/unsigned integers
float	single-precision floating-point scalar
double	double-precision floating scalar
vec2, vec3, vec4	floating point vector
dvec2, dvec3, dvec4	double precision floating-point vectors
bvec2, bvec3, bvec4	Boolean vectors
ivec2, ivec3, ivec4 uvec2, uvec3, uvec4	signed and unsigned integer vectors
mat2, mat3, mat4	2x2, 3x3, 4x4 float matrix
mat2x2, mat2x3, mat2x4	2-column float matrix of 2, 3, or 4 rows
mat3x2, mat3x3, mat3x4	3-column float matrix of 2, 3, or 4 rows
mat4x2, mat4x3, mat4x4	4-column float matrix of 2, 3, or 4 rows
dmat2, dmat3, dmat4	2x2, 3x3, 4x4 double-precision float matrix
dmat2x2, dmat2x3, dmat2x4	2-col. double-precision float matrix of 2, 3, 4 rows
dmat3x2, dmat3x3, dmat3x4	3-col. double-precision float matrix of 2, 3, 4 rows
dmat4x2, dmat4x3, dmat4x4	4-column double-precision float matrix of 2, 3, 4 rows

- Vector component accessors

Component Accessors	Description
(x , y , z , w)	Components associated with positions
(r , g , b , a)	Components associated with colors
(s , t , p , q)	Components associated with texture coordinates

- Accessing elements in vectors and matrices

- Vectors support a named-component method and an array-like method
- Matrices support an array-like method

```
vec3 luminance = color.rrr ;  
color = color.abgr ; // reverse the components of a color  
  
vec4 color = otherColor.rgz ; // Error : "z" is from a different group  
  
vec2 pos ;  
float zPos = pos.z ; // Error : no "z" component in 2D vectors  
  
mat4 m = mat4 ( 2.0 ) ;  
vec4 zVec = m [ 2 ] ; // get column 2 of the matrix  
float yScale = m [ 1 ] [ 1 ] ; // or m[1].y works as well
```

본 과목에서 자주 사용할 GL 함수 대체 GLM 함수들 예

기존 OpenGL 함수	방법 또는 GLM 제공 함수 (여기서는 float 타입만 고려함)	참고
glLoadIdentity	glm::mat4(1.0) or glm::mat4();	1
glMultMatrix	glm::mat4() * glm::mat4();	1
glLoadTransposeMatrix	glm::transpose(glm::mat4());	1
glMultTransposeMatrix	glm::mat4() * glm::transpose(glm::mat4());	1
glTranslate	glm::mat4 glm::translate(glm::mat4 const & m, glm::vec3 const & translation);	2
glScale	glm::mat4 glm::scale(glm::mat4 const & m, glm::vec3 const & factors);	2
glRotate	glm::mat4 glm::rotate(glm::mat4 const & m, float angle, glm::vec3 const & axis);	2
glFrustum	glm::mat4 glm::frustum(float left, float right, float bottom, float top, float zNear, float zFar);	2
glOrtho	glm::mat4 glm::ortho(float left, float right, float bottom, float top, float zNear, float zFar);	2
gluLookAt	glm::mat4 glm::lookAt(glm::vec3 const & eye, glm::vec3 const & center, glm::vec3 const & up);	2
gluOrtho2D	glm::mat4 glm::ortho(float left, float right, float bottom, float top);	2
gluPerspective	glm::mat4 perspective(float fovy, float aspect, float zNear, float zFar);	2
inverseTranspose	glm::mat4 inverseTranspose(glm::mat4 const & m);	3

참고 1: #include <glm/glm.hpp>, 참고 2: #include <glm/gtc/matrix_transform.hpp>, 참고 3: #include <glm/gtc/matrix_inverse.hpp>

glm::translate(*) & scale(*) 함수의 구현

```
namespace glm {  
    ...  
    template <typename T, precision P>  
    GLM_FUNC_QUALIFIER tmat4x4<T, P> translate( tmat4x4<T, P> const & m, tvec3<T, P> const & v ) {  
        tmat4x4<T, P> Result(m);  
        Result[3] = m[0] * v[0] + m[1] * v[1] + m[2] * v[2] + m[3];  
        return Result;  
    }  
  
    template <typename T, precision P>  
    GLM_FUNC_QUALIFIER tmat4x4<T, P> scale( tmat4x4<T, P> const & m, tvec3<T, P> const & v ) {  
        tmat4x4<T, P> Result(uninitialize);  
        Result[0] = m[0] * v[0];  
        Result[1] = m[1] * v[1];  
        Result[2] = m[2] * v[2];  
        Result[3] = m[3];  
        return Result;  
    }  
}
```

$$T(t_x, t_y, t_z) = \begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$S(s_x, s_y, s_z) = \begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

glm::rotate(*) 함수의 구현

```
namespace glm {  
    ...  
    template <typename T, precision P>  
    GLM_FUNC_QUALIFIER tmat4x4<T, P> rotate( tmat4x4<T, P> const & m, T angle, tvec3<T, P> const & v ) {  
        T const a = angle; T const c = cos(a); T const s = sin(a);  
        tvec3<T, P> axis(normalize(v)); tvec3<T, P> temp((T(1) - c) * axis);  
        tmat4x4<T, P> Rotate(uninitialize);  
        Rotate[0][0] = c + temp[0] * axis[0];  
        Rotate[0][1] = 0 + temp[0] * axis[1] + s * axis[2]; Rotate[0][2] = 0 + temp[0] * axis[2] - s * axis[1];  
        Rotate[1][0] = 0 + temp[1] * axis[0] - s * axis[2]; Rotate[1][1] = c + temp[1] * axis[1];  
        Rotate[1][2] = 0 + temp[1] * axis[2] + s * axis[0];  
        Rotate[2][0] = 0 + temp[2] * axis[0] + s * axis[1]; Rotate[2][1] = 0 + temp[2] * axis[1] - s * axis[0];  
        Rotate[2][2] = c + temp[2] * axis[2];  
  
        tmat4x4<T, P> Result(uninitialize);  
        Result[0] = m[0] * Rotate[0][0] + m[1] * Rotate[0][1] + m[2] * Rotate[0][2];  
        Result[1] = m[0] * Rotate[1][0] + m[1] * Rotate[1][1] + m[2] * Rotate[1][2];  
        Result[2] = m[0] * Rotate[2][0] + m[1] * Rotate[2][1] + m[2] * Rotate[2][2];  
        Result[3] = m[3];  
        return Result;  
    }  
}
```

$$R(\alpha, n_x, n_y, n_z) = \begin{bmatrix} \bar{n}_x^2(1 - c) + c & \bar{n}_x\bar{n}_y(1 - c) - \bar{n}_z s & \bar{n}_z\bar{n}_x(1 - c) + \bar{n}_y s & 0 \\ \bar{n}_x\bar{n}_y(1 - c) + \bar{n}_z s & \bar{n}_y^2(1 - c) + c & \bar{n}_y\bar{n}_z(1 - c) - \bar{n}_x s & 0 \\ \bar{n}_z\bar{n}_x(1 - c) - \bar{n}_y s & \bar{n}_y\bar{n}_z(1 - c) + \bar{n}_x s & \bar{n}_z^2(1 - c) + c & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Code Examples: Matrix Transform 1

```
#include <glm/glm.hpp>
#include <glm/gtc/matrix_transform.hpp>

int foo() {
    glm::vec4 Position = glm::vec4(glm::vec3(0.0f), 1.0f);
    glm::mat4 Model = glm::translate(glm::mat4(1.0f), glm::vec3(1.0f));
    glm::vec4 Transformed = Model * Position;
    ...
    return 0;
}
```

Code Examples: Matrix Transform 2

```
#include <glm/vec3.hpp> // glm::vec3
#include <glm/vec4.hpp> // glm::vec4, glm::ivec4
#include <glm/mat4x4.hpp> // glm::mat4
#include <glm/gtc/matrix_transform.hpp> // glm::translate, glm::rotate, glm::scale, glm::perspective
#include <glm/gtc/type_ptr.hpp> // glm::value_ptr

void func(GLuint LocationMVP, float Translate, glm::vec2 const & Rotate) {
    glm::mat4 Projection = glm::perspective(glm::radians(45.0f), 4.0f / 3.0f, 0.1f, 100.f);
    glm::mat4 ViewTranslate = glm::translate(glm::mat4(1.0f), glm::vec3(0.0f, 0.0f, -Translate));
    glm::mat4 ViewRotateX = glm::rotate(ViewTranslate, Rotate.y, glm::vec3(-1.0f, 0.0f, 0.0f));
    glm::mat4 View = glm::rotate(ViewRotateX, Rotate.x, glm::vec3(0.0f, 1.0f, 0.0f));
    glm::mat4 Model = glm::scale(glm::mat4(1.0f), glm::vec3(0.5f));
    glm::mat4 MVP = Projection * View * Model;
    glUniformMatrix4fv(LocationMVP, 1, GL_FALSE, glm::value_ptr(MVP));
}
```

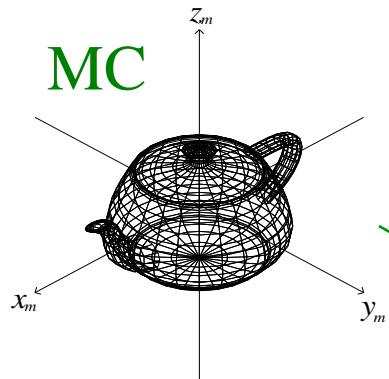
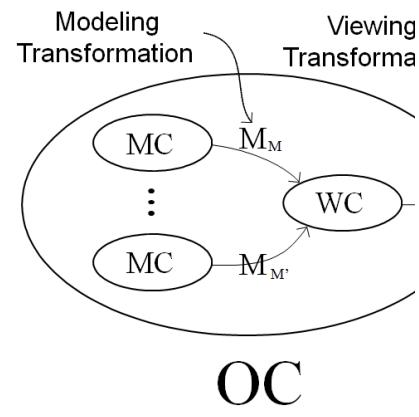
모델링 좌표계와 모델링 변환

- 모델링 좌표계(Modeling Coordinates, MC)

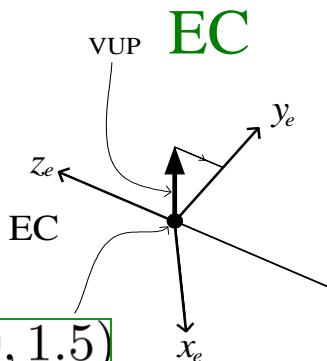
- 가상의 세상에 존재하는 각 물체들이 정의가 되는 좌표계

- 모델링 변환(Modeling Transformation)

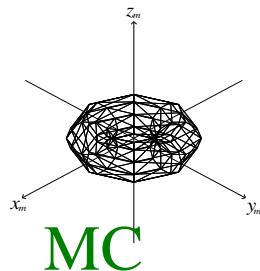
- 모델링 좌표계에서 정의된 물체를 가상의 세상 공간인 세상 좌표계로 보내 수는 변환.



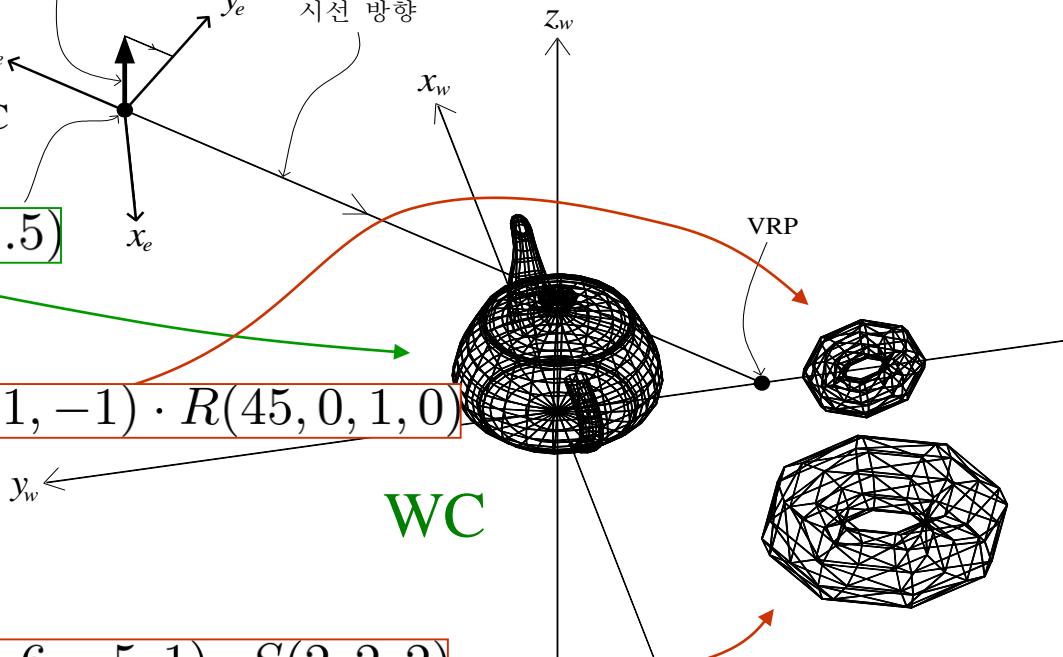
$$M_M = T(0.0, 0.0, 1.5)$$



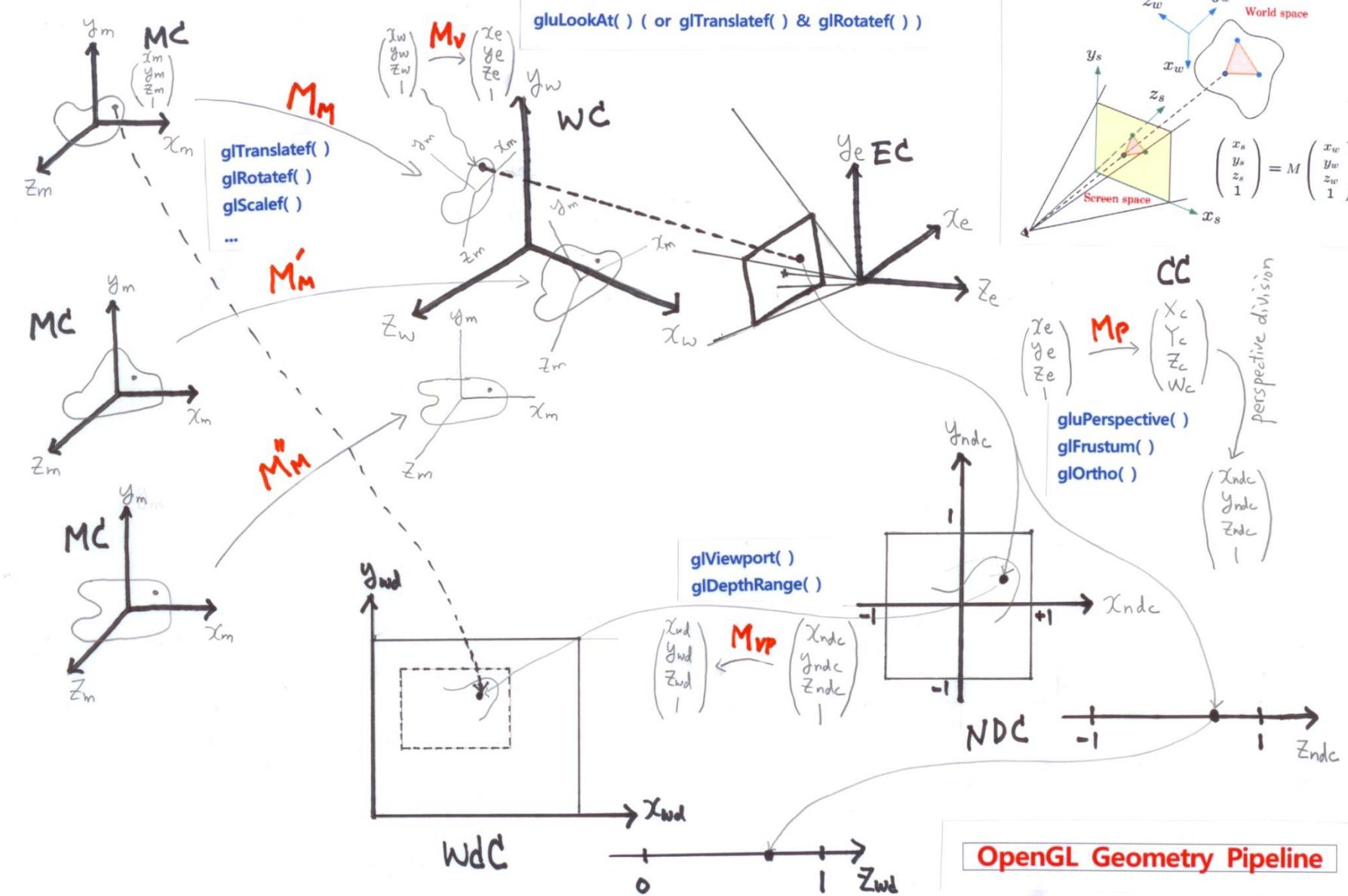
$$M_M = T(0, -6, 0) \cdot R(-45, 0, 1, 0) \cdot S(1, 1, -1) \cdot R(45, 0, 1, 0)$$



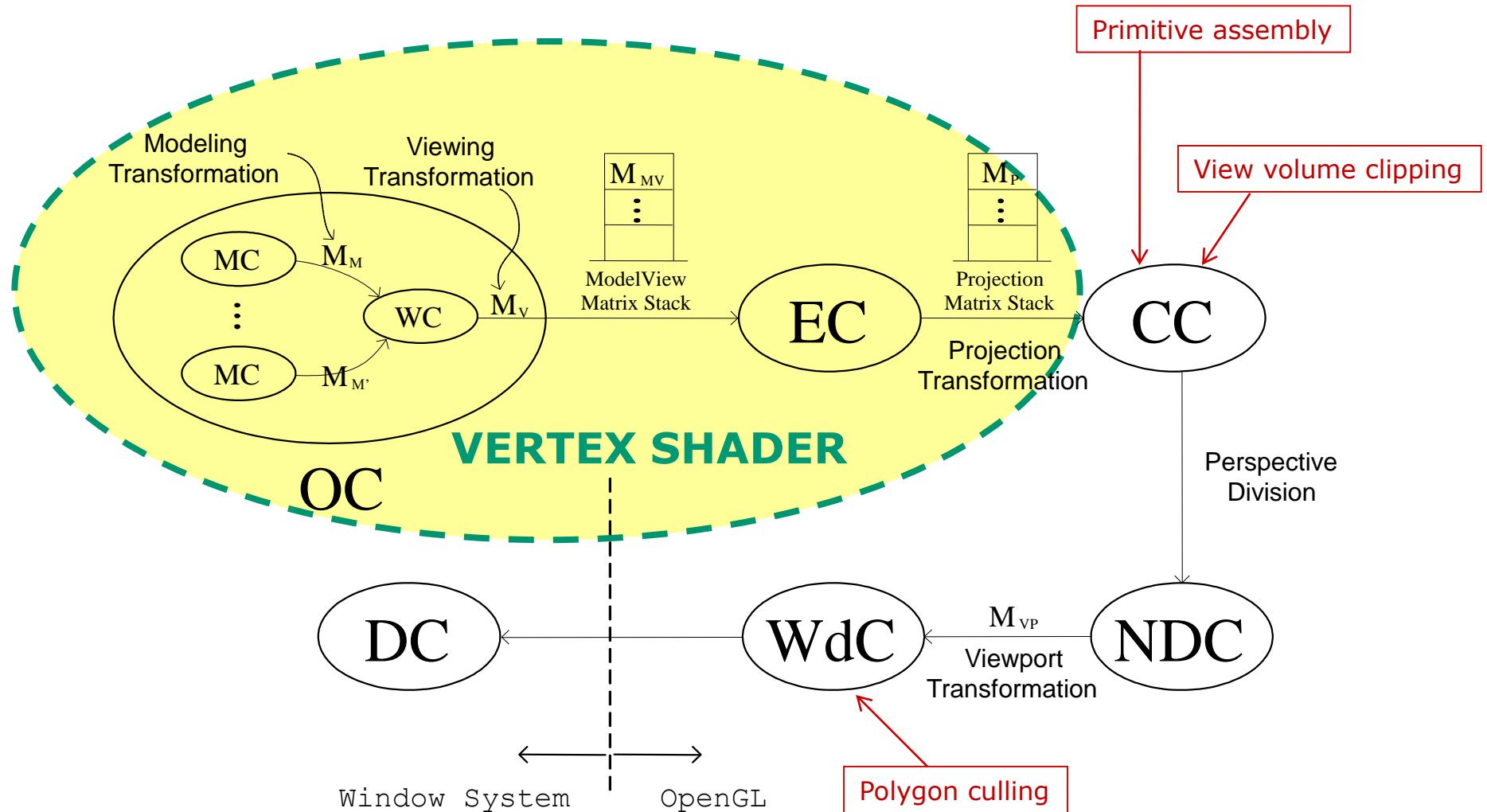
$$M_M = T(-6, -5, 1) \cdot S(2, 2, 2)$$



OpenGL Geometry Pipeline Overview



Geometry Stage in Programmable OpenGL Pipeline



Geometry stage의 골격을 이루는
geometry pipeline을 이해하자.

당분간 vertex shader에 집중할 예정임.

OpenGL을 통한 3차원 모델링 변환 예

```
//#include <glm/glm.hpp>
#include <glm/gtc/matrix_transform.hpp> //translate, rotate, scale, lookAt, ortho, perspective, etc.
glm::mat4 ModelViewProjectionMatrix;
glm::mat4 ModelViewMatrix, ViewMatrix, ProjectionMatrix;

void display(void) {
    glClear(GL_COLOR_BUFFER_BIT);

    ModelViewMatrix = glm::scale(ViewMatrix, glm::vec3(50.0f, 50.0f, 50.0f));
    ModelViewProjectionMatrix = ProjectionMatrix * ModelViewMatrix;
    glUniformMatrix4fv(loc_ModelViewProjectionMatrix, 1, GL_FALSE, &ModelViewProjectionMatrix[0][0]);
    glLineWidth(2.0f);
    draw_axes();
    glLineWidth(1.0f);

ModelViewMatrix = glm::translate(ViewMatrix, glm::vec3(0.0f, 0.0f, 1.5f));
    ModelViewProjectionMatrix = ProjectionMatrix * ModelViewMatrix;
    glUniformMatrix4fv(loc_ModelViewProjectionMatrix, 1, GL_FALSE, &ModelViewProjectionMatrix[0][0]);
    glUniform3f(loc_primitive_color, 1.0f, 1.0f, 0.0f);
draw_geom_obj(GEOM_OBJ_ID_TEAPOT);
```

$$M_V \cdot T(0, 0, 1.5)$$

$$M_V \cdot T(-6, -5, 1) \cdot S(2, 2, 2)$$

```
ModelViewMatrix = glm::translate(ViewMatrix, glm::vec3(-6.0f, -5.0f, 1.0f));
ModelViewMatrix = glm::scale(ModelViewMatrix, glm::vec3(2.0f, 2.0f, 2.0f)),
ModelViewProjectionMatrix = ProjectionMatrix * ModelViewMatrix;
glUniformMatrix4fv(loc_ModelViewProjectionMatrix, 1, GL_FALSE, &ModelViewProjectionMatrix[0][0]);
glUniform3f(loc_primitive_color, 1.0f, 0.0f, 1.0f);
draw_geom_obj(GEOM_OBJ_ID_DONUT);
```

```
ModelViewMatrix = glm::translate(ViewMatrix, glm::vec3(0.0f, -6.0f, 0.0f));
ModelViewMatrix = glm::rotate(ModelViewMatrix, -45.0f*TO_RADIAN, glm::vec3(0.0f, 1.0f, 0.0f));
ModelViewMatrix = glm::scale(ModelViewMatrix, glm::vec3(1.0f, 1.0f, -1.0f));
ModelViewMatrix = glm::rotate(ModelViewMatrix, 45.0f*TO_RADIAN, glm::vec3(0.0f, 1.0f, 0.0f));
ModelViewProjectionMatrix = ProjectionMatrix * ModelViewMatrix;
glUniformMatrix4fv(loc_ModelViewProjectionMatrix, 1, GL_FALSE, &ModelViewProjectionMatrix[0][0]);
glUniform3f(loc_primitive_color, 0.0f, 1.0f, 1.0f);
draw_geom_obj(GEOM_OBJ_ID_DONUT);

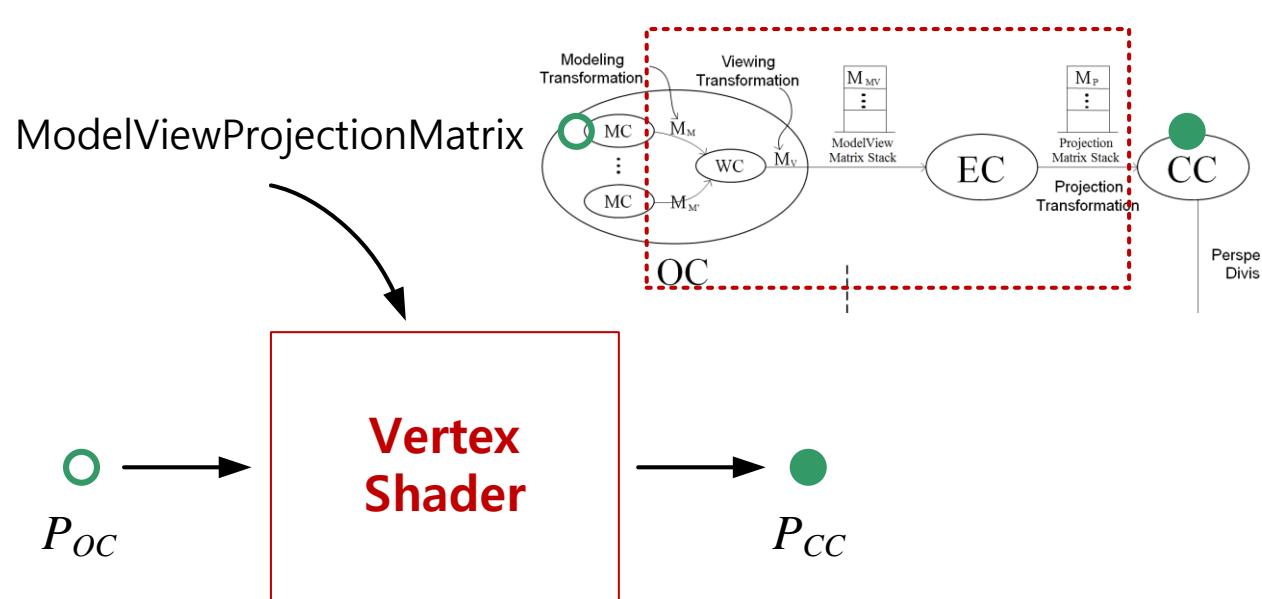
}
glFlush();
```

$$M_V \cdot T(0, -6, 0) \cdot R(-45, 0, 1, 0) \cdot S(1, 1, -1) \cdot R(45, 0, 1, 0)$$

```

void draw_geom_obj(int geom_obj_ID) {
    glBindVertexArray(geom_obj_VAO[geom_obj_ID]);
    glDrawArrays(GL_TRIANGLES, 0, 3 * geom_obj_n_triangles[geom_obj_ID]);
    glBindVertexArray(0);
}

```



```

#version 330
vertex shader

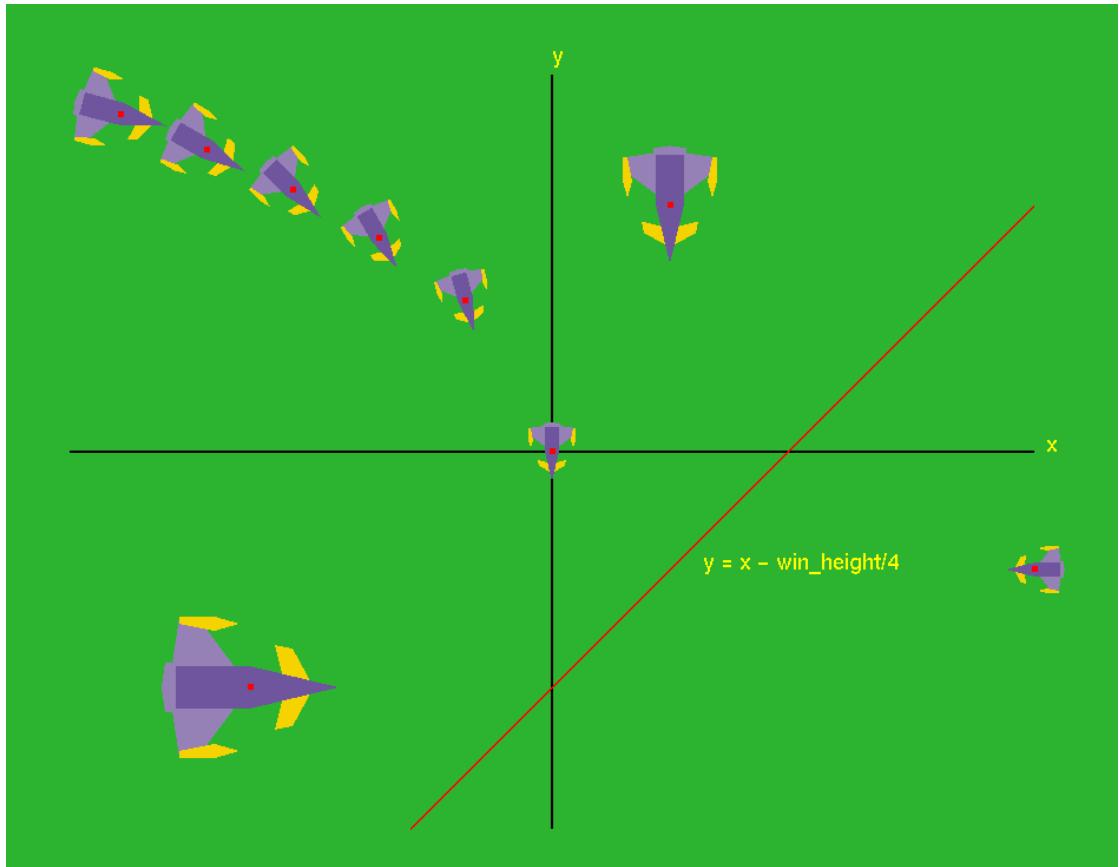
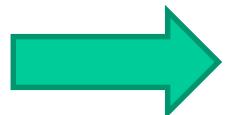
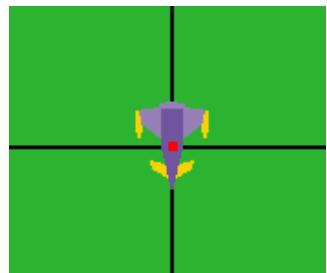
uniform mat4 ModelViewProjectionMatrix;
uniform vec3 primitive_color;
layout (location = 0) in vec4 a_position;
out vec4 color;

void main(void) {
    color = vec4(primitive_color, 1.0f);
    gl_Position =
        ModelViewProjectionMatrix
        * a_position;
}

```

OpenGL을 통한 2차원 모델링 변환 예

- 모델링 좌표계(MC)를 기준으로 주어진 비행기 데이터에 대해 다음과 같은 그림을 그려보자.



```
void display(void) {
    int i;
    float x, r, s, delx, delr, dels;
    glm::mat4 ModelMatrix;
```

```
//#include <glm/glm.hpp>
#include <glm/gtc/matrix_transform.hpp>
glm::mat4 ModelViewProjectionMatrix;
glm::mat4 ViewMatrix, ProjectionMatrix, ViewProjectionMatrix;
```

```
glClear(GL_COLOR_BUFFER_BIT);
```

```
ModelMatrix = glm::mat4(1.0f);
ModelViewProjectionMatrix = ViewProjectionMatrix * ModelMatrix;
glUniformMatrix4fv(loc_ModelViewProjectionMatrix, 1, GL_FALSE, &ModelViewProjectionMatrix[0][0]);
draw_axes();
draw_line();
draw_airplane(); // 0
```

```
ModelMatrix = glm::translate(glm::mat4(1.0f), glm::vec3(-win_width / 4.0f, -win_height / 4.0f, 0.0f));
ModelMatrix = glm::rotate(ModelMatrix, 90.0f*TO_RADIAN, glm::vec3(0.0f, 0.0f, 1.0f));
ModelMatrix = glm::scale(ModelMatrix, glm::vec3(3.0f, 3.0f, 1.0f));
ModelViewProjectionMatrix = ViewProjectionMatrix * ModelMatrix;
glUniformMatrix4fv(loc_ModelViewProjectionMatrix, 1, GL_FALSE, &ModelViewProjectionMatrix[0][0]);
draw_airplane(); // 1
```

```
ModelMatrix = glm::translate(glm::mat4(1.0f), glm::vec3(win_width / 2.5f, -win_height / 8.0f, 0.0f));
ModelMatrix = glm::rotate(ModelMatrix, 270.0f*TO_RADIAN, glm::vec3(0.0f, 0.0f, 1.0f));
ModelViewProjectionMatrix = ViewProjectionMatrix * ModelMatrix;
glUniformMatrix4fv(loc_ModelViewProjectionMatrix, 1, GL_FALSE, &ModelViewProjectionMatrix[0][0]);
draw_airplane(); // 2
```

```
ModelMatrix = glm::translate(glm::mat4(1.0f), glm::vec3(win_height / 4.0f, 0.0f, 0.0f));
ModelMatrix = glm::rotate(ModelMatrix, 45.0f*TO_RADIAN, glm::vec3(0.0f, 0.0f, 1.0f));
ModelMatrix = glm::scale(ModelMatrix, glm::vec3(1.0f, -1.0f, 1.0f));
ModelMatrix = glm::rotate(ModelMatrix, -45.0f*TO_RADIAN, glm::vec3(0.0f, 0.0f, 1.0f));
ModelMatrix = glm::translate(ModelMatrix, glm::vec3(-win_height / 4.0f, 0.0f, 0.0f));
ModelMatrix = glm::translate(ModelMatrix, glm::vec3(win_width / 2.5f, -win_height / 8.0f, 0.0f));
ModelMatrix = glm::scale(ModelMatrix, glm::vec3(2.0f, 2.0f, 1.0f));
ModelMatrix = glm::translate(ModelMatrix, glm::vec3(-win_width / 2.5f, win_height / 8.0f, 0.0f));

ModelMatrix = glm::translate(ModelMatrix, glm::vec3(win_width / 2.5f, -win_height / 8.0f, 0.0f));
ModelMatrix = glm::rotate(ModelMatrix, 270.0f*TO_RADIAN, glm::vec3(0.0f, 0.0f, 1.0f));
ModelViewProjectionMatrix = ViewProjectionMatrix * ModelMatrix;
glUniformMatrix4fv(loc_ModelViewProjectionMatrix, 1, GL_FALSE, &ModelViewProjectionMatrix[0][0]);
draw_airplane(); // 3
```

```

delx = win_width/14.0f; delr = 15.0f; dels = 1.1f;
x = -delx; r = delr; s = dels;
for (i = 0; i < 5; i++, x -= delx, r += delr, s *= dels) {
    ModelMatrix = glm::translate(glm::mat4(1.0f), glm::vec3(x, 15.0f*sqrtf(-x), 0.0f));
    ModelMatrix = glm::rotate(ModelMatrix, r*TO_RADIAN, glm::vec3(0.0f, 0.0f, 1.0f));
ModelMatrix = glm::scale(ModelMatrix, glm::vec3(s, s, 1.0f));
    ModelViewProjectionMatrix = ViewProjectionMatrix * ModelMatrix;
    glUniformMatrix4fv(loc_ModelViewProjectionMatrix, 1, GL_FALSE, &ModelViewProjectionMatrix[0][0]);
draw_airplane(); // 4
}
glFlush();
}

```

#version 330

vertex shader

```

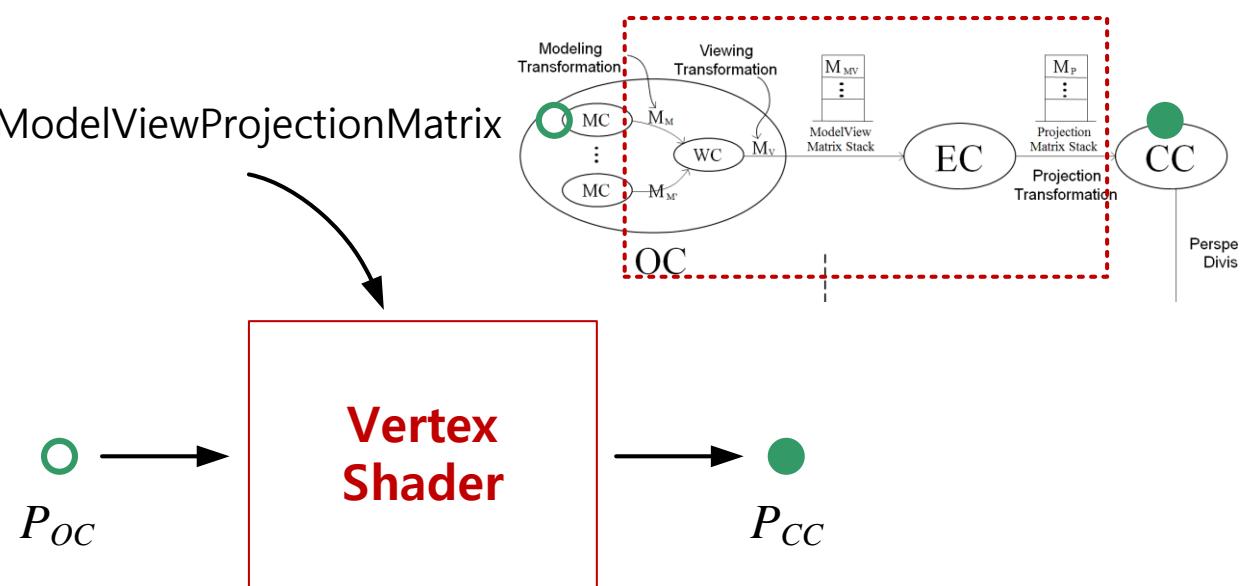
uniform mat4 ModelViewProjectionMatrix;
uniform vec3 primitive_color;
layout (location = 0) in vec4 a_position;
out vec4 color;

```

```

void main(void) {
    color = vec4(primitive_color, 1.0f);
    gl_Position =
        ModelViewProjectionMatrix
            * a_position;
}

```



Simplification of Geometric Transformations

```
glPushMatrix();
glTranslatef(win_height/4.0, 0.0, 0.0);
glRotatef(45.0, 0.0, 0.0, 1.0);
glScalef(1.0, -1.0, 1.0);
glRotatef(-45.0, 0.0, 0.0, 1.0);
glTranslatef(-win_height/4.0, 0.0, 0.0);
glTranslatef(win_width/2.5, -win_height/8.0, 0.0);
glScalef(2.0, 2.0, 1.0);
glTranslatef(-win_width/2.5, +win_height/8.0, 0.0);

glTranslatef(win_width/2.5, -win_height/8.0, 0.0);
glRotatef(270.0, 0.0, 0.0, 1.0);
draw_airplane(); // 3
glPopMatrix();
```

$$T(t_x, t_y)S(1, -1) = S(1, -1)T(?, ?)$$
$$R(\theta)S(1, -1) = S(1, -1)R(?)$$
$$R(\theta)T(t_x, t_y) = T(?, ?)R(\theta)$$
$$T(t_x, t_y)R(\theta) = R(\theta)T(?, ?)$$

어떻게 위의 기하 변환을 세 개의 기하 변환의 합성으로 표현할 수 있을까?

```
glPushMatrix();
glTranslatef(win_height/8.0, win_width/2.5-win_height/4.0, 0.0);
glRotatef(180.0, 0.0, 0.0, 1.0);
glScalef(2.0, -2.0, 1.0);
draw_airplane(); // 3
glPopMatrix();
```

Geometric Transformation

- Projection Transformation

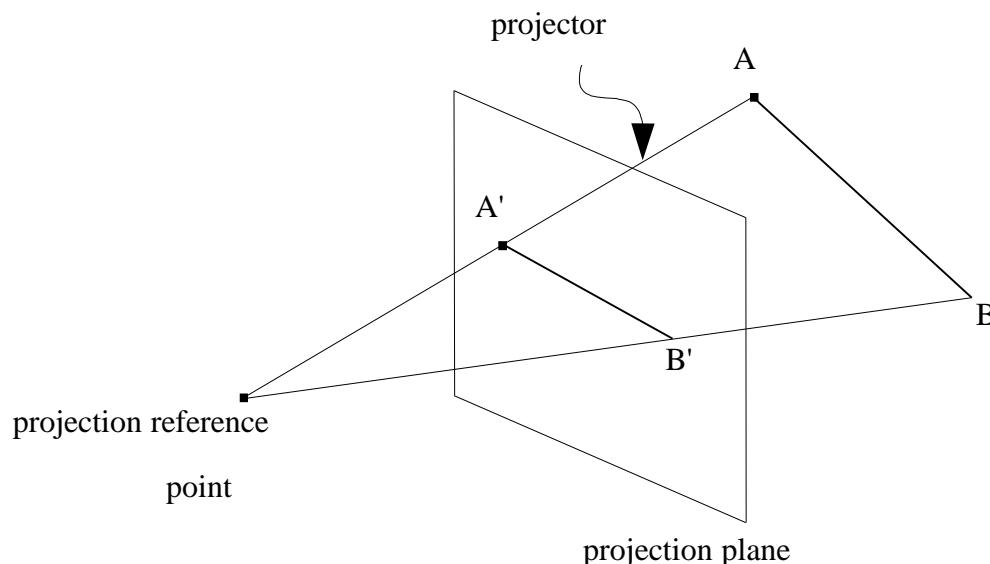
평면 기하 투영(Planar Geometric Projection)

- **투영 변환(projection transformation)**

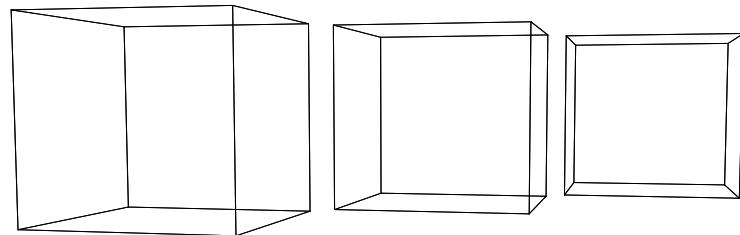
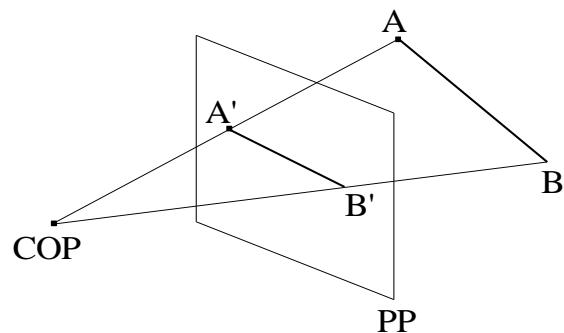
- n 차원 공간의 점을 m 차원 공간의 점으로 바꾸어주는 변환 ($n > m$).
- 렌더링 파이프라인에서는 (가상의 물체가 존재하는) 3차원 공간에서 (이미지 화면에 해당하는) 2차원 공간으로의 투영 변환을 고려.

- **평면 기하 투영**

- 투영 참조점, 투영 평면, 그리고 투영선

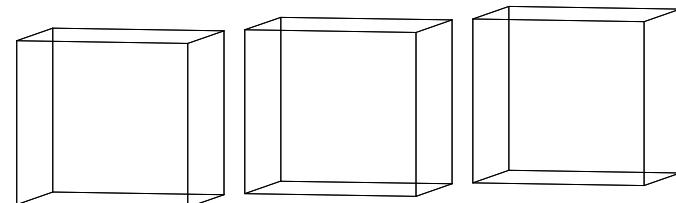
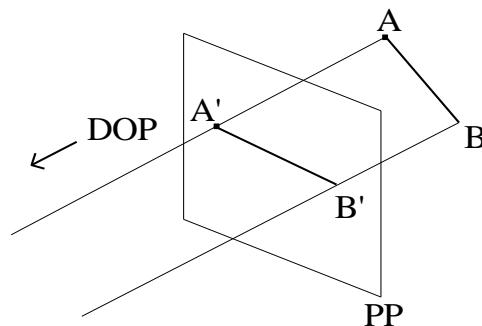


- 평면 기하 투영의 분류
 - 원근 투영(perspective projection)
 - 투영 참조점이 투영 평면에서 유한 거리만큼 떨어진 투영.
 - 투영 참조점의 W좌표가 0이 아닌 경우.
 - 이 경우 투영 참조점을 투영 중심(center of projection)이라 함.
 - 평행성이 깨지면서 원근감이 생성됨 → 원근 축소(perspective foreshortening)
 - 비아핀 변환

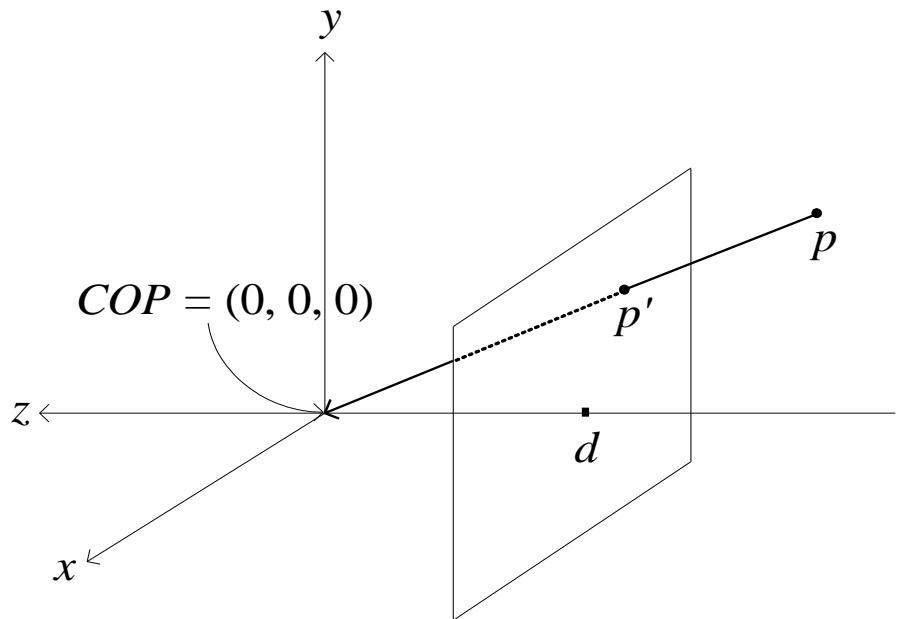


- 평행 투영(parallel projection)

- 투영 참조점이 투영 평면에서 특정 방향으로 무한 거리만큼 떨어진 투영.
- 투영 참조점의 W좌표가 0인 경우.
- 이 경우 투영 참조점을 투영 방향(direction of projection)이라 함.
- 투영 방향에 따라 직교 투영(orthographic projection)과 경사 투영(oblique projection)으로 분류됨.
- 아핀 변환



• 원근 투영의 예 1



$$\longrightarrow M_{pers} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & \frac{1}{d} & 0 \end{bmatrix}$$

원근 나눗셈(perspective division)

* z/d 의 의미

$$\begin{pmatrix} \frac{dx}{d} \\ \frac{dy}{d} \\ \frac{dz}{d} \\ 1 \end{pmatrix} \leftarrow \begin{pmatrix} x \\ y \\ z \\ \frac{z}{d} \end{pmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & \frac{1}{d} & 0 \end{bmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}$$

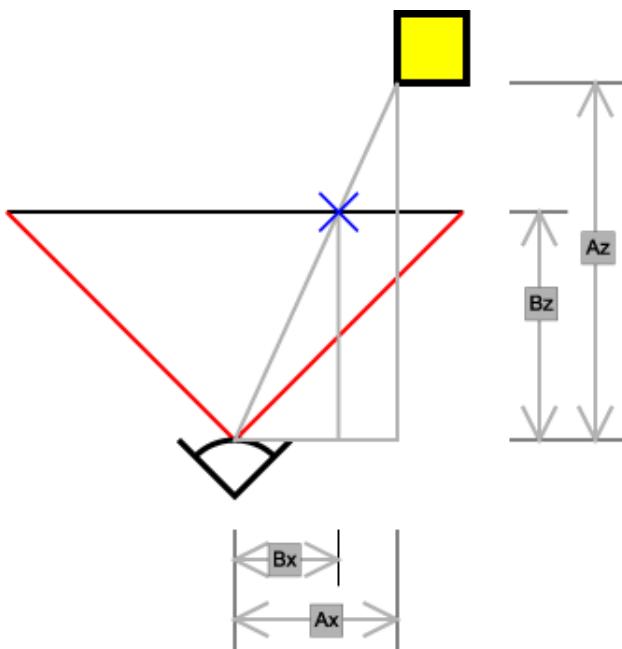
- 원근 투영의 예 2

$$COP = (0, 0, 0)$$

$$PP. ax + by + cz = 1$$

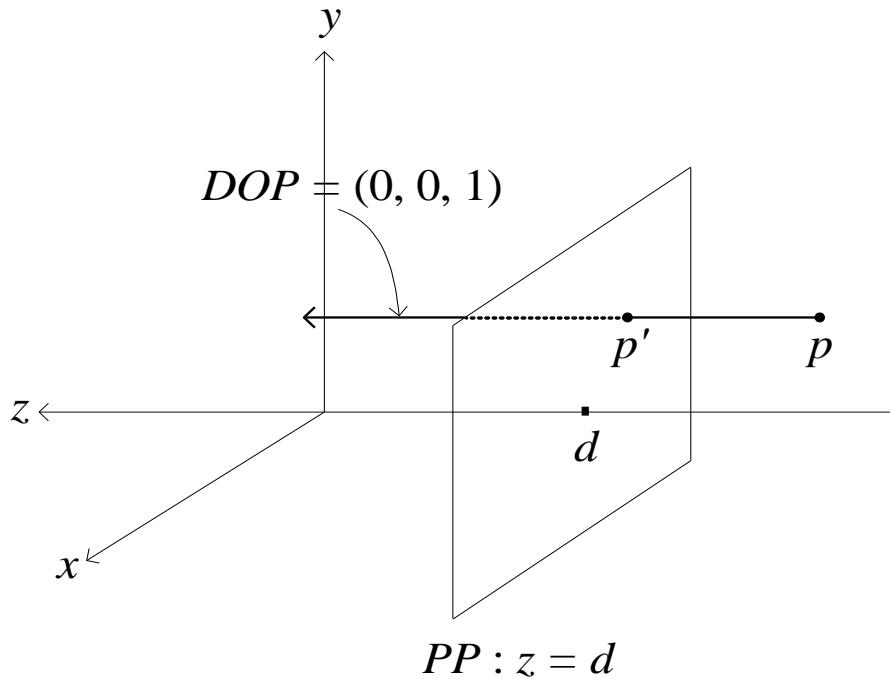


$$M_{pers} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ a & b & c & 0 \end{bmatrix}$$



From WIKIPEDIA: The Free Encyclopedia

- 평행 투영의 예 1



$$M_{ortho} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & d \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

- 평행 투영의 예 2

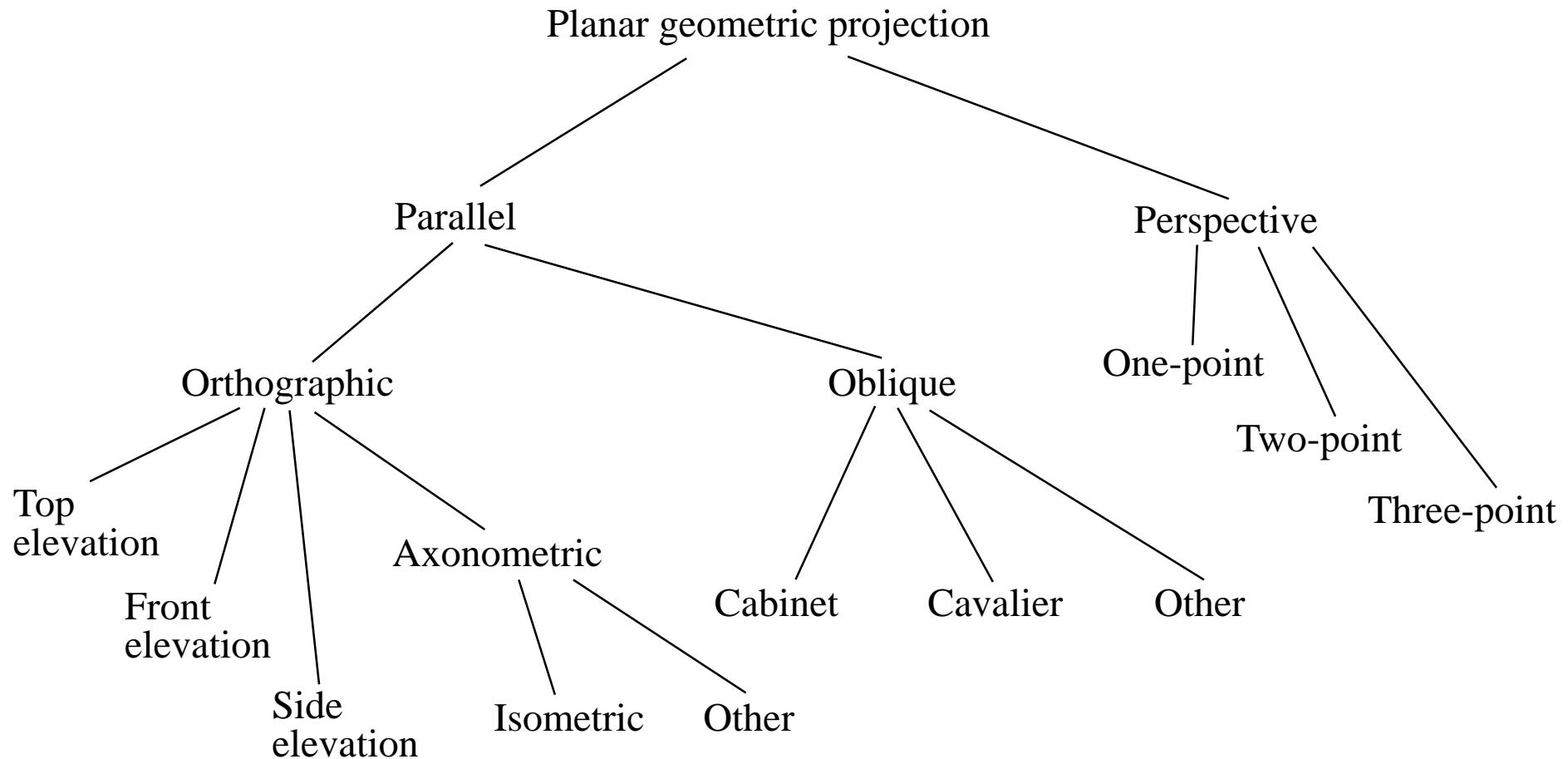
$$DOP = (-a, -b, -c)$$

$$PP: ax + by + cz = 1$$



$$M_{ortho} = \begin{bmatrix} e & f & g & h \\ i & j & k & l \\ m & n & o & p \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

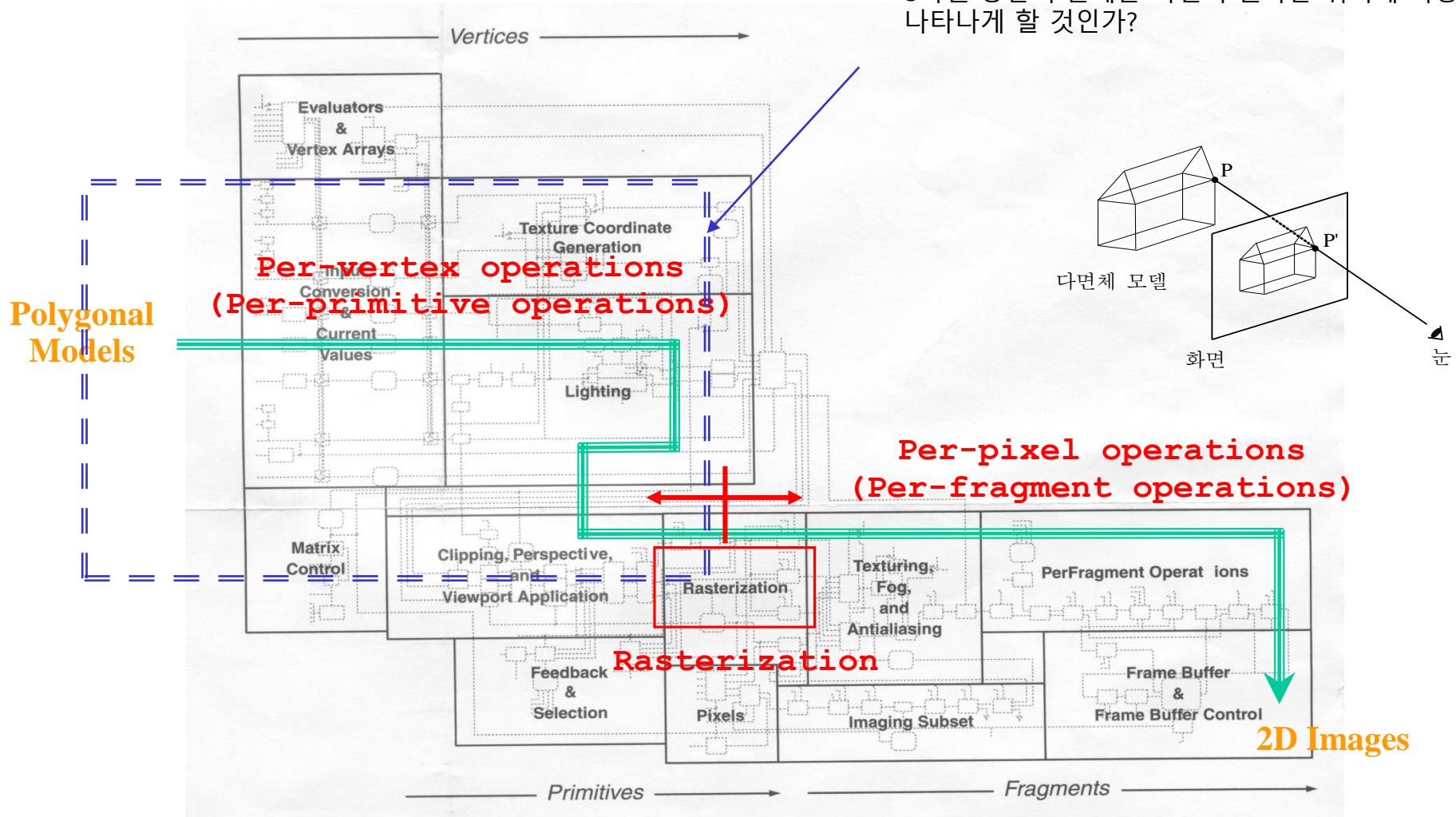
투영의 분류



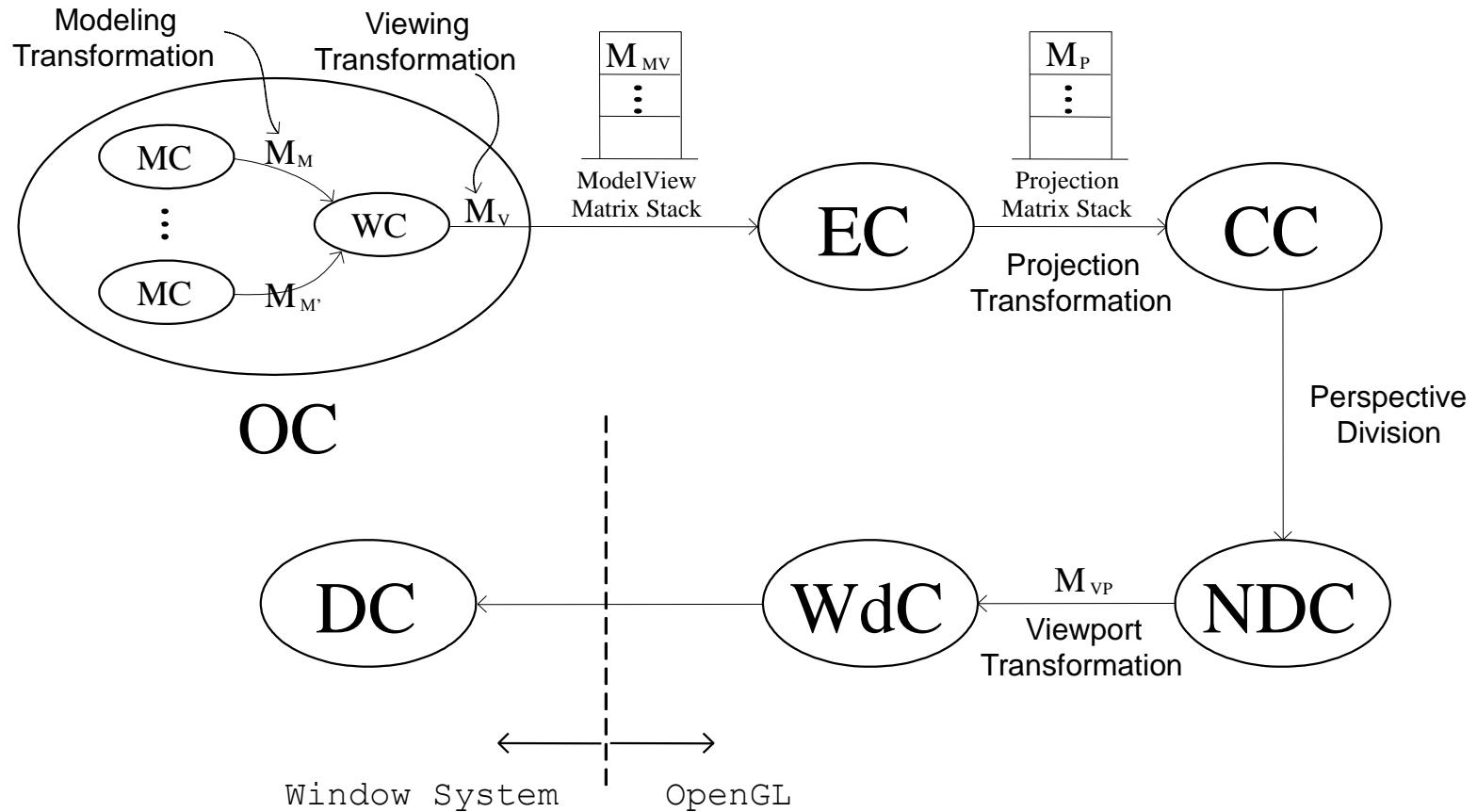
3D Viewing in OpenGL

OpenGL Rendering Pipeline

- 기하 파이프라인은 이 부분의 근간을 이룸.
- 3차원 뷰잉(3D viewing) 계산 수행.
- 3차원 공간의 물체를 화면의 원하는 위치에 어떻게 나타나게 할 것인가?



OpenGL Geometry Pipeline I



OpenGL의 기하 파이프라인과 사진 촬영과의 비교

- **모델링 변환(Modeling transformation)**

- 피사체 및 조명 배치
- 모델링 좌표계(MC) → 세상 좌표계(WC)

세상 좌표계 (World Coordinates, WC)

-- 가상의 세상이 존재하는 좌표계

- **뷰잉 변환(Viewing transformation)**

- 카메라의 위치와 방향 설정
- 세상 좌표계(WC) → 눈 좌표계(EC)

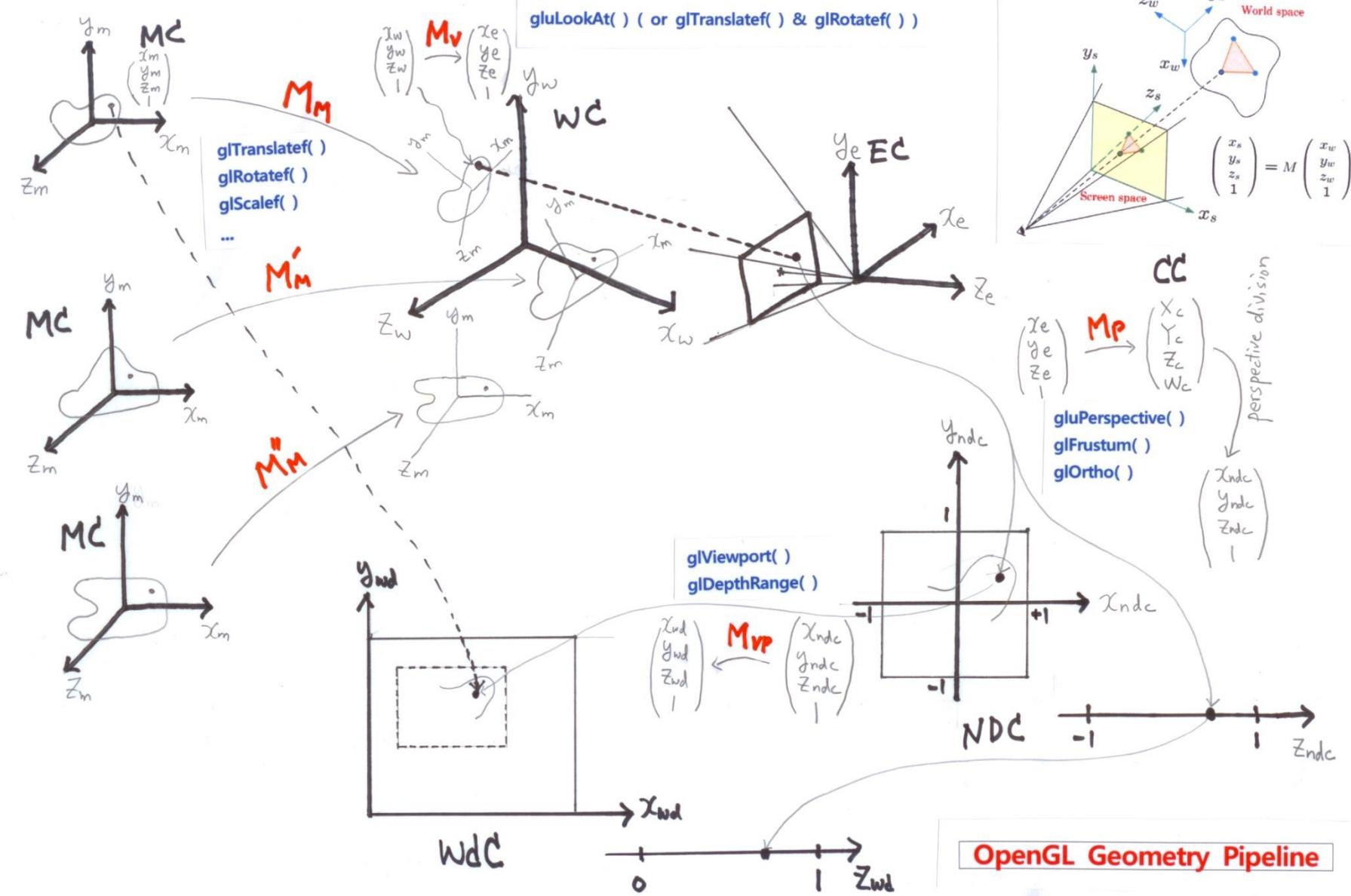
- **투영 변환(Projection transformation)**

- 사용할 렌즈 결정 및 촬영 대상 결정
- 카메라의 셔터를 누름
- 눈 좌표계(EC) → 절단 좌표계(CC) → 정규 디바이스 좌표계(NDC)

- **뷰포트 변환(Viewport transformation)**

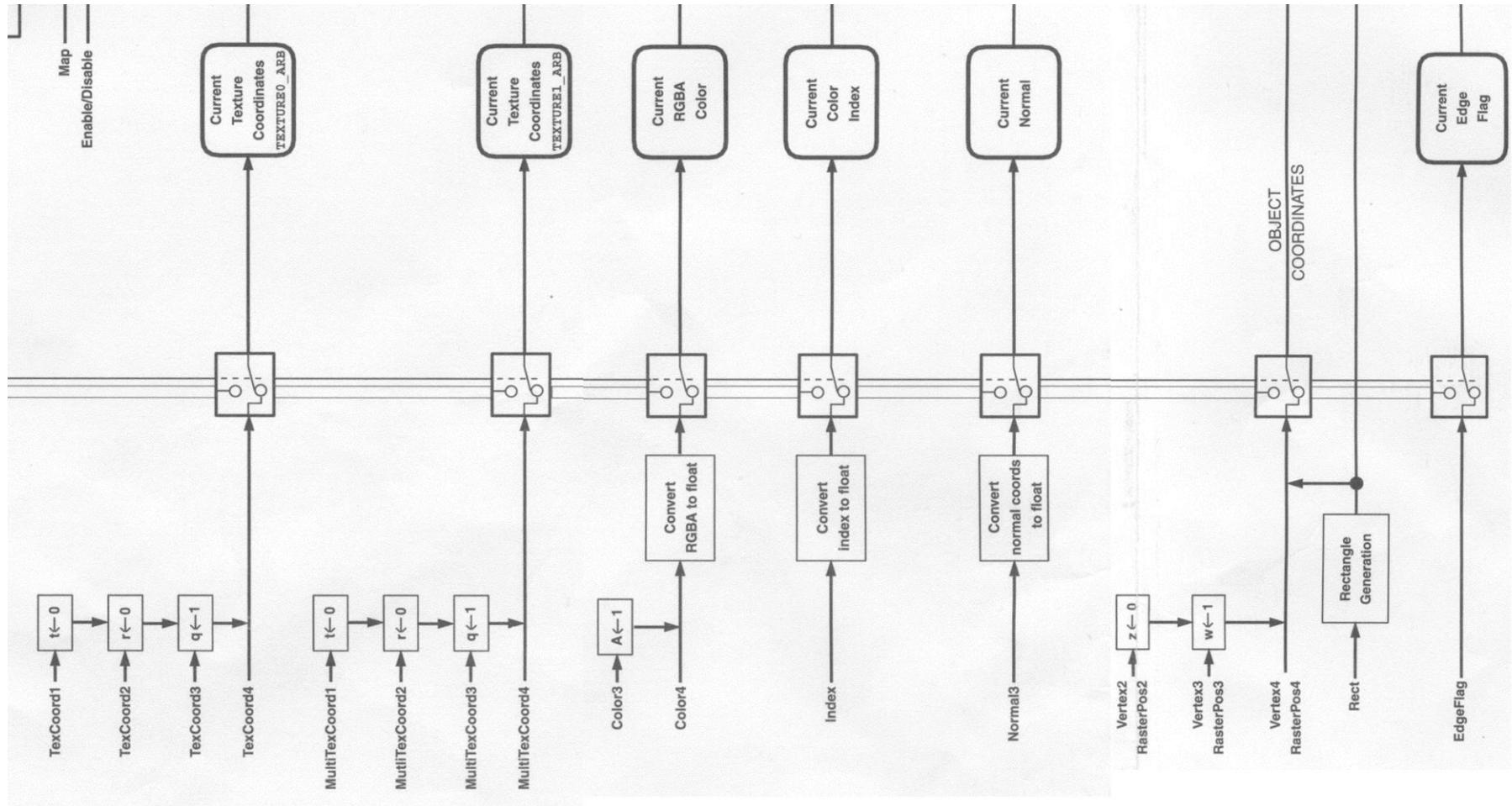
- 사진의 크기 결정 및 인화
- 정규 디바이스 좌표계(NDC) → 윈도우 좌표계(WdC)

OpenGL Geometry Pipeline II



기하 파이프라인의 구동 (Compatibility Profile)

- Specification of per-vertex attributes



- `glBegin(*)`;과 `glEnd()`; 문장 사이에서 `glVertex*(*)` 등의 함수를 통하여 꼭지점들의 속성을 설정하면(고전적인 방법임), 각 꼭지점 별로 속성들이 조합이 되어 파이프라인으로 흘러 들어감.
- 앞 부분 (지금의 vertex shader 부분)에서 각 꼭지점에 대한 독립적인 처리가 수행된 후, Primitive Assembly 과정을 통하여 해당 타입의 기하 프리미티브로 조립이 됨.
- 이후 각 프리미티브 단위로 기하 파이프라인을 따라 흘러가면서 처리가 됨.

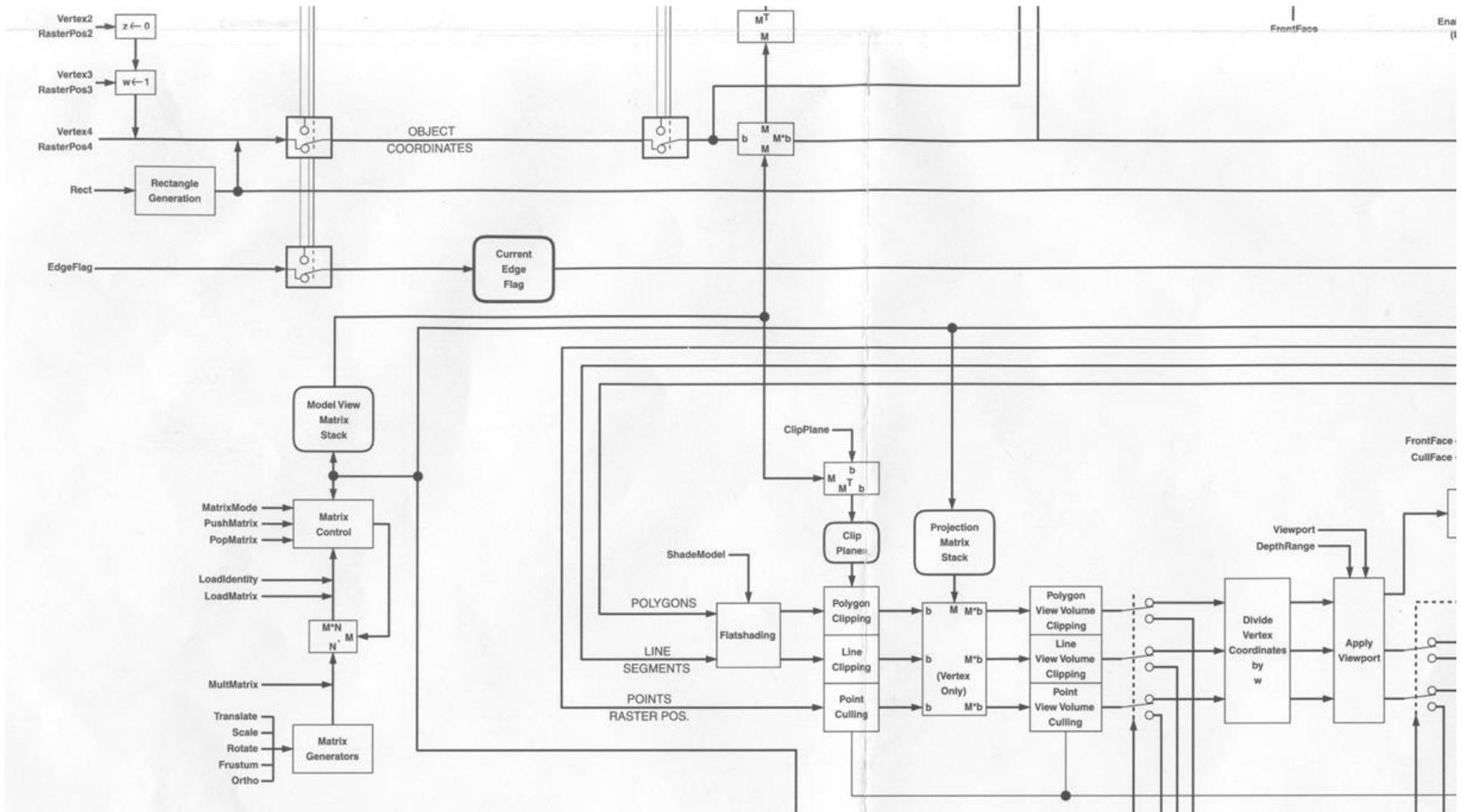
```

for (k = 0; k < num_cows; k++) {
    glPushMatrix();
    glTranslatef(pos[k][0], pos[k][1], pos[k][2]);
    for (i = 0; i < npoly; i++) {
        glBegin(GL_POLYGON);
        for (j = 0; j < object[i].nvertex; j++) {
            glNormal3fv(object[i].normal[j]);
            glColor3fv(object[i].color[j]);
            glMultiTexCoord2fv(GL_TEXTURE0,
                               object[i].texcoord0[j]);
            glVertex3fv(object[i].vertex[j]);
        }
        glEnd();
    }
    glPopMatrix();
}

```

Vertex Attribute Register Number	Conventional Per-vertex Parameter	Conventional Per-vertex Parameter Command	Conventional Component Mapping
0 (OPOS)	vertex position	Vertex	x, y, z, w
1 (WGHT)	vertex weights	VertexWeightEXT	w, 0, 0, 1
2 (NRML)	normal	Normal	x, y, z, 1
3 (COLO)	primary color	Color	r, g, b, a
4 (COL1)	secondary color	SecondaryColorEXT	r, g, b, 1
5 (FOGC)	fog coordinate	FogCoordEXT	fc, 0, 0, 1
6	-	-	-
7	-	-	-
8 (TEX0)	texture coord 0	MultiTexCoord(GL_TEXTURE0_ARB, ...)	s, t, r, q
9 (TEX1)	texture coord 1	MultiTexCoord(GL_TEXTURE1_ARB, ...)	s, t, r, q
10 (TEX2)	texture coord 2	MultiTexCoord(GL_TEXTURE2_ARB, ...)	s, t, r, q
11 (TEX3)	texture coord 3	MultiTexCoord(GL_TEXTURE3_ARB, ...)	s, t, r, q
12 (TEX4)	texture coord 4	MultiTexCoord(GL_TEXTURE4_ARB, ...)	s, t, r, q
13 (TEX5)	texture coord 5	MultiTexCoord(GL_TEXTURE5_ARB, ...)	s, t, r, q
14 (TEX6)	texture coord 6	MultiTexCoord(GL_TEXTURE6_ARB, ...)	s, t, r, q
15 (TEX7)	texture coord 7	MultiTexCoord(GL_TEXTURE7_ARB, ...)	s, t, r, q

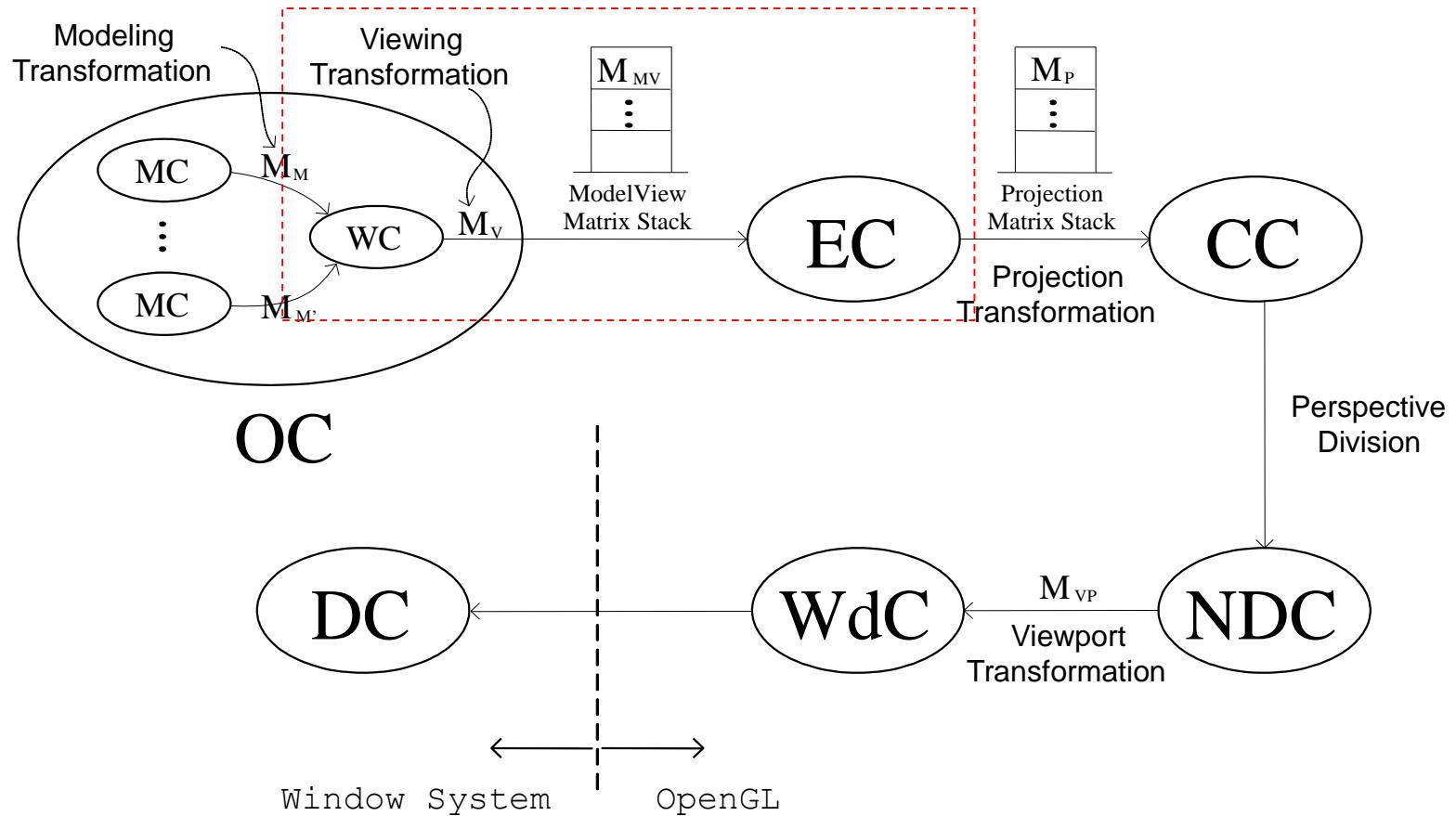
- 각 꼭지점 좌표는 기하 파이프라인을 타고 가면서 1. 모델뷰 행렬 스택의 탑에 있는 행렬 M_{MV} (= $M_V * M_M$), 2. 투영 행렬 스택의 탑에 있는 행렬 M_P , 그리고 3. 뷰포트 변환에 해당하는 M_{MV} 에 순서대로 곱해지면서 윈도우 좌표계로 변환이 됨.



From World Coordinates to Eye Coordinates: Viewing Transformation

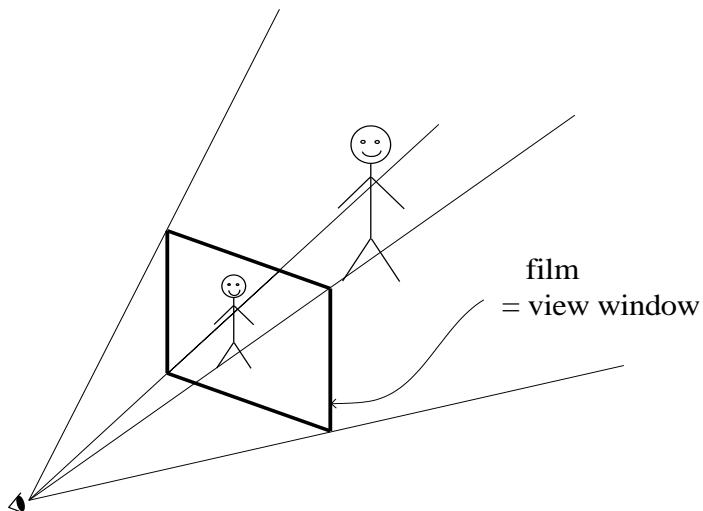
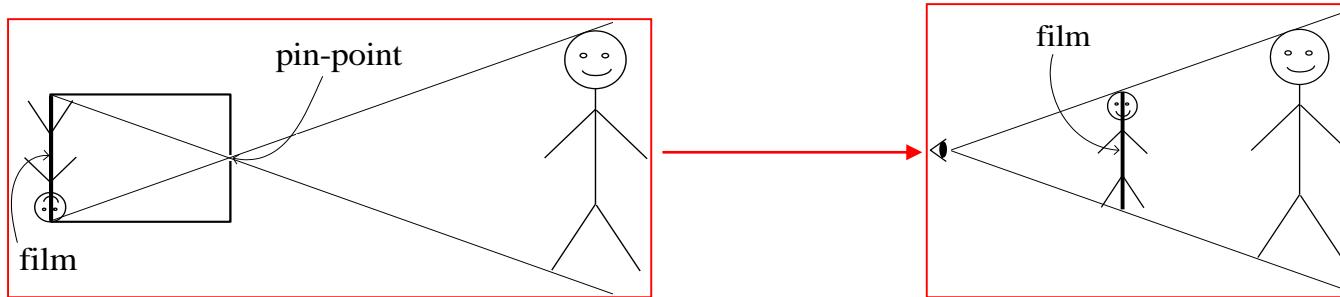
OpenGL의 기하 파이프라인과 사진 촬영과의 비교

- 모델링 변환(Modeling transformation)
 - 피사체 및 조명 배치
 - 모델링 좌표계(MC) → 세상 좌표계(WC)
- 뷰잉 변환(Viewing transformation)
 - 카메라의 위치와 방향 설정 ← 카메라의 위치와 방향은 WC 기준으로 설정!
 - 세상 좌표계(WC) → 눈 좌표계(EC)
- 투영 변환(Projection transformation)
 - 사용할 렌즈 결정 및 촬영 대상 결정
 - 카메라의 셔터를 누름
 - 눈 좌표계(EC) → 절단 좌표계(CC) → 정규 디바이스 좌표계(NDC)
- 뷰포트 변환(Viewport transformation)
 - 사진의 크기 결정 및 인화
 - 정규 디바이스 좌표계(NDC) → 윈도우 좌표계(WdC)

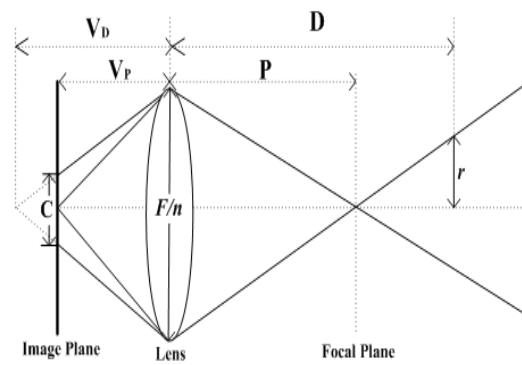


바늘 구멍 카메라(Pinhole Camera)

- A very simple camera that uses a pinhole instead of a lens
- Mathematically, the pinhole has no area → unrealistic



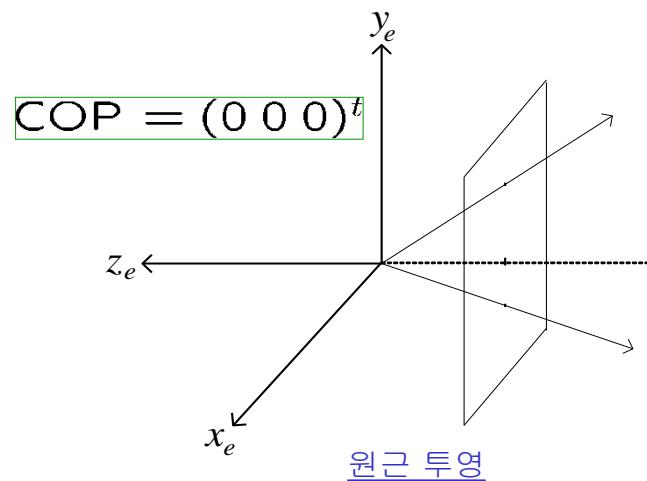
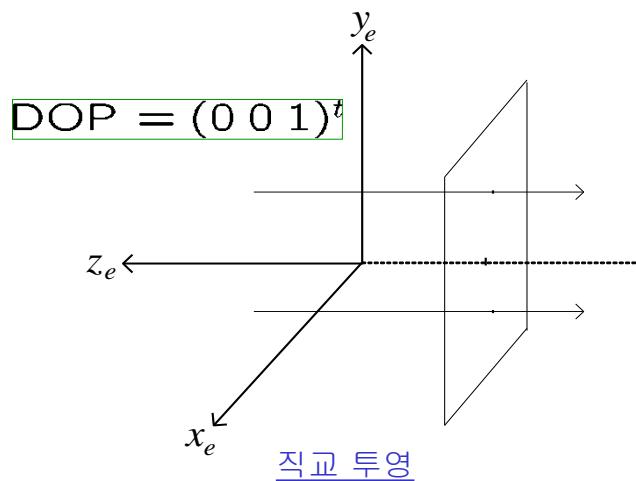
- 실제 카메라의 렌즈는 면적을 가짐! → 피사계 심도(depth of field) 효과



눈좌표계와 뷰잉 변환

- **눈 좌표계(Eye Coordinates, EC)**

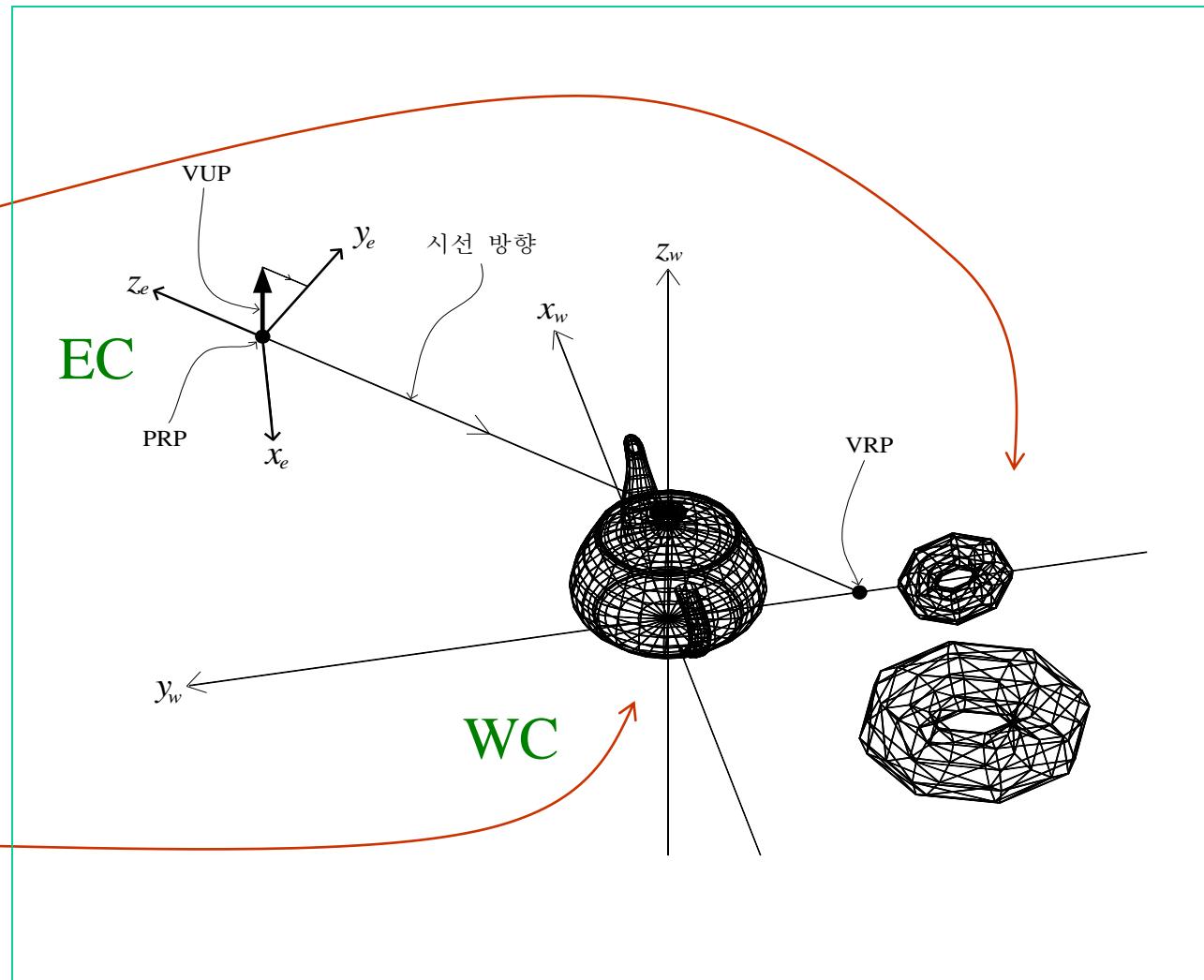
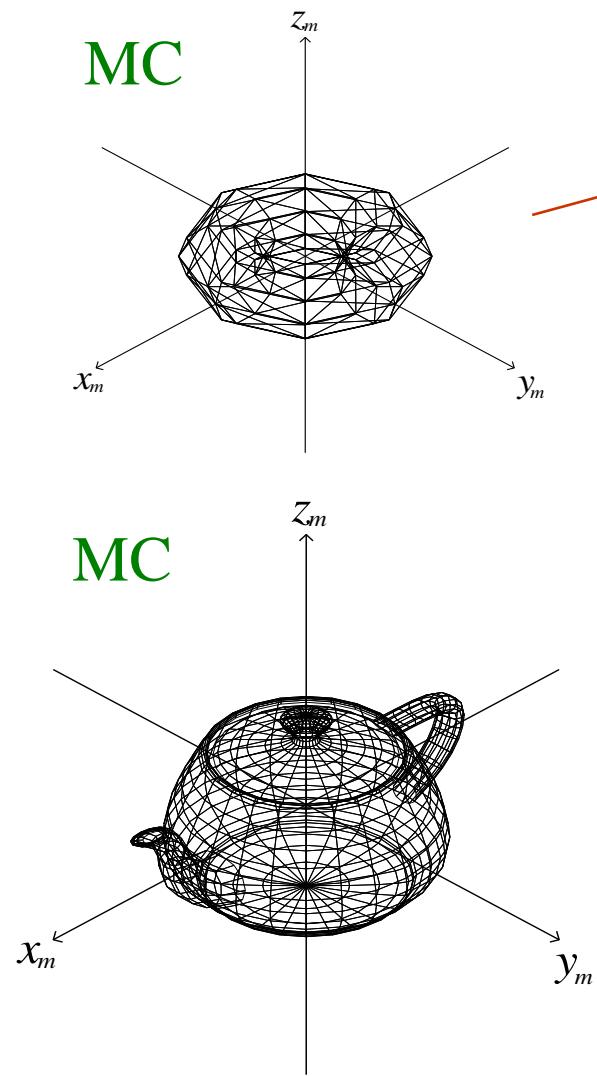
- 카메라 좌표계(Camera Coordinates)라고 부르기도 함.
- 카메라를 기준으로 하는 좌표계로서, 주어진 점이 카메라 기준점에서 왼쪽으로, 오른쪽으로, 그리고 앞으로 얼마나큼 떨어져 있는가 하는 정보를 제공함.
- OpenGL에서는 음의 z_e 축 방향으로 바라봄.



- **뷰잉 변환(viewing transformation)**

- `gluLookAt(*)` 함수에 의해 눈 좌표계가 결정.
- 세상 좌표계를 기준으로 설정된 점의 좌표를 눈 좌표계를 기준으로 바꾸어주는 변환

세상 좌표계에서의 카메라의 설정



- 목표: 가상의 세상에 피사체를 배치한 상태에서 **카메라의 위치와 방향을 결정해야 함.**
- **카메라의 위치와 방향 설정 방법**
 - ① 직접 이동 변환과 회전 변환을 사용하여 합성
 - 추후 설명
 - ② void **gluLookAt(GLdouble ex, GLdouble ey, GLdouble ez, GLdouble cx, GLdouble cy, GLdouble cz, GLdouble ux, GLdouble uy, GLdouble uz); (Compatibility Profile)**
or
glm::mat4 **glm::lookAt(glm::vec3 const & eye, glm::vec3 const & center, glm::vec3 const & up); (Core Profile - glm)** 함수 사용
- ★ 근본적으로 위의 두 방법은 같은 방법으로서 아래와 같은 기하 변환을 설정하는데, 초보자들에게는 후자가 사용하기 편리함.

$$M_v = R \cdot T(-e_x, -e_y, -e_z)$$

gluLookAt(*) 함수를 사용한 뷰잉 변환의 설정

```
void gluLookAt(GLdouble ex, GLdouble ey, GLdouble ez, GLdouble cx, GLdouble cy,  
               GLdouble cz, GLdouble ux, GLdouble uy, GLdouble uz);
```

- 투영 참조점(**Projection Reference Point**)의 설정: $\text{PRP} \equiv (e_x \ e_y \ e_z)^t$
 - 뷰 참조점(**View Reference Point**)의 설정: $\text{VRP} \equiv (c_x \ c_y \ c_z)^t$
 - 뷰 상향 벡터(**View-Up Vector**)의 설정: $\text{VUP} \equiv (u_x \ u_y \ u_z)^t$
- ★ 위의 세 인자의 설정을 통하여 1. 카메라의 위치, 2. 카메라가 세상을 바라보는 방향, 그리고 3. 영상의 위쪽 방향(따라서 오른쪽 방향)이 결정이 됨.

뷰잉 변환의 유도

- 눈 좌표계의 결정 과정

$$1. \text{ VPN} \equiv \text{PRP} - \text{VRP}$$

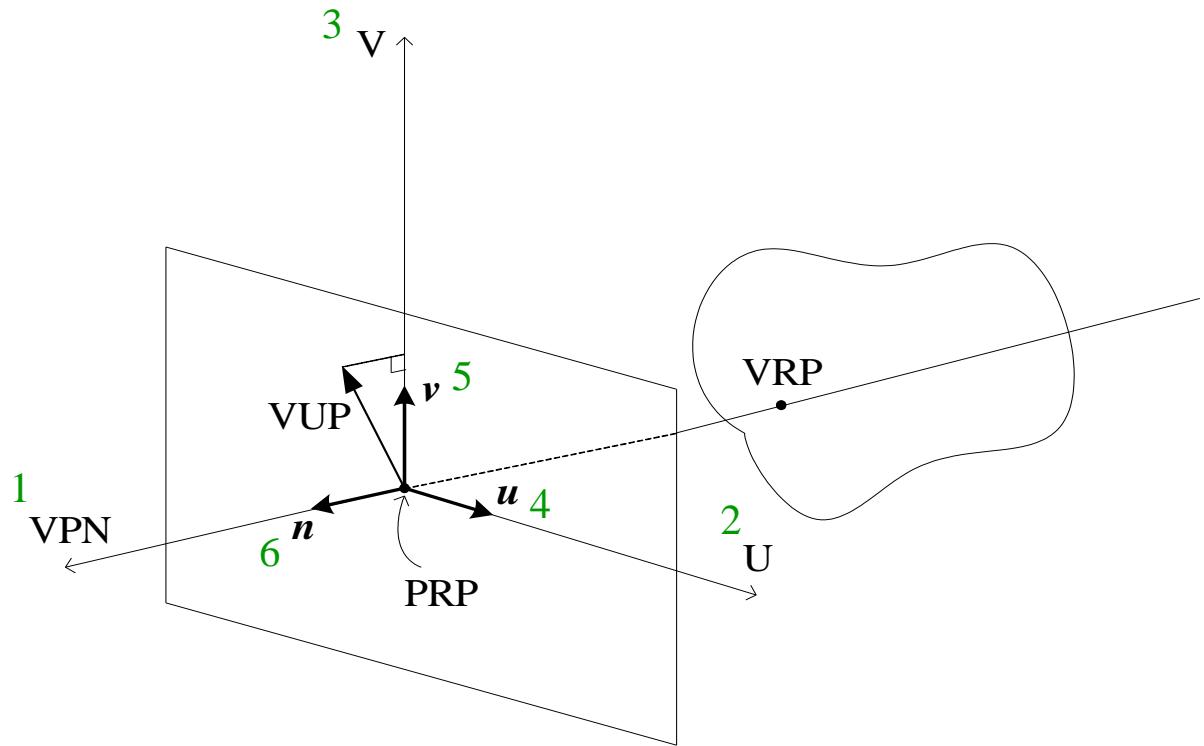
$$2. \text{ U} \equiv \text{VUP} \times \text{VPN}$$

$$3. \text{ V} \equiv \text{VPN} \times \text{U}$$

$$4. \text{ u} = \frac{\text{U}}{|\text{U}|} \equiv (u_x \ u_y \ u_z)^t$$

$$5. \text{ v} = \frac{\text{V}}{|\text{V}|} \equiv (v_x \ v_y \ v_z)^t$$

$$6. \text{ n} = \frac{\text{VPN}}{|\text{VPN}|} \equiv (n_x \ n_y \ n_z)^t$$

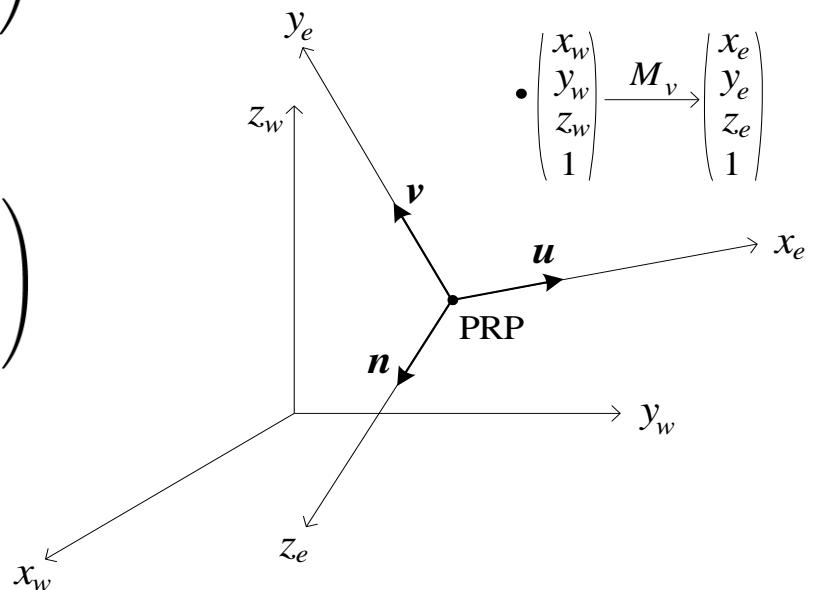


- 뷰잉 변환 M_v 의 계산

$$\begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix} + x_w \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix} + y_w \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix} + z_w \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix} = \begin{pmatrix} e_x \\ e_y \\ e_z \end{pmatrix} + x_e \begin{pmatrix} u_x \\ u_y \\ u_z \end{pmatrix} + y_e \begin{pmatrix} v_x \\ v_y \\ v_z \end{pmatrix} + z_e \begin{pmatrix} n_x \\ n_y \\ n_z \end{pmatrix}$$

→ $\begin{pmatrix} x_w \\ y_w \\ z_w \end{pmatrix} = \begin{pmatrix} e_x \\ e_y \\ e_z \end{pmatrix} + \begin{bmatrix} u_x & v_x & n_x \\ u_y & v_y & n_y \\ u_z & v_z & n_z \end{bmatrix} \begin{pmatrix} x_e \\ y_e \\ z_e \end{pmatrix}$

$$\begin{aligned} \begin{pmatrix} x_e \\ y_e \\ z_e \end{pmatrix} &= \begin{bmatrix} u_x & v_x & n_x \\ u_y & v_y & n_y \\ u_z & v_z & n_z \end{bmatrix}^{-1} \left(\begin{pmatrix} x_w \\ y_w \\ z_w \end{pmatrix} - \begin{pmatrix} e_x \\ e_y \\ e_z \end{pmatrix} \right) \\ &= \begin{bmatrix} u_x & u_y & u_z \\ v_x & v_y & v_z \\ n_x & n_y & n_z \end{bmatrix} \left(\begin{pmatrix} x_w \\ y_w \\ z_w \end{pmatrix} - \begin{pmatrix} e_x \\ e_y \\ e_z \end{pmatrix} \right) \end{aligned}$$



$$\begin{bmatrix} u_x & u_y & u_z \\ v_x & v_y & v_z \\ n_x & n_y & n_z \end{bmatrix} \begin{pmatrix} e_x \\ e_y \\ e_z \end{pmatrix} \equiv \begin{pmatrix} t_x \\ t_y \\ t_z \end{pmatrix}$$

→

$$\begin{pmatrix} x_e \\ y_e \\ z_e \\ 1 \end{pmatrix} = \begin{bmatrix} u_x & u_y & u_z & -t_x \\ v_x & v_y & v_z & -t_y \\ n_x & n_y & n_z & -t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{pmatrix} x_w \\ y_w \\ z_w \\ 1 \end{pmatrix}$$

$$M_v = \begin{bmatrix} u_x & u_y & u_z & 0 \\ v_x & v_y & v_z & 0 \\ n_x & n_y & n_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 & 0 & -e_x \\ 0 & 1 & 0 & -e_y \\ 0 & 0 & 1 & -e_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \equiv R \cdot T(-e_x, -e_y, -e_z)$$

회전 변환

이동 변환

☺ 뷰잉 변환은 강체 변환임!

gluLookAt(*) 함수의 구현

```

void gluLookAt(GLdouble ex, GLdouble ey, GLdouble ez,
               GLdouble cx, GLdouble cy, GLdouble cz,
               GLdouble ux, GLdouble uy, GLdouble uz)
{
    GLdouble m[16]; GLdouble x[3], y[3], z[3]; GLdouble mag;

    /* Make rotation matrix */

    /* Z vector */
    z[0] = ex - cx; z[1] = ey - cy; z[2] = ez - cz;
    mag = sqrt( z[0]*z[0] + z[1]*z[1] + z[2]*z[2] );
    if (mag) { /* mpichler, 19950515 */
        z[0] /= mag; z[1] /= mag; z[2] /= mag;
    }

    /* Y vector */
    y[0] = ux; y[1] = uy; y[2] = uz;

    /* X vector = Y cross Z */
    x[0] = y[1]*z[2] - y[2]*z[1]; x[1] = -y[0]*z[2] + y[2]*z[0];
    x[2] = y[0]*z[1] - y[1]*z[0];

    /* Recompute Y = Z cross X */
    y[0] = z[1]*x[2] - z[2]*x[1]; y[1] = -z[0]*x[2] + z[2]*x[0];
    y[2] = z[0]*x[1] - z[1]*x[0];
}

```

```

mag = sqrt( x[0]*x[0] + x[1]*x[1] + x[2]*x[2] );
if (mag) {
    x[0] /= mag; x[1] /= mag; x[2] /= mag;
}

mag = sqrt( y[0]*y[0] + y[1]*y[1] + y[2]*y[2] );
if (mag) {
    y[0] /= mag; y[1] /= mag; y[2] /= mag;
}

#define M(row,col) m[col*4+row]
M(0,0) = x[0]; M(0,1) = x[1]; M(0,2) = x[2]; M(0,3) = 0.0;
M(1,0) = y[0]; M(1,1) = y[1]; M(1,2) = y[2]; M(1,3) = 0.0;
M(2,0) = z[0]; M(2,1) = z[1]; M(2,2) = z[2]; M(2,3) = 0.0;
M(3,0) = 0.0; M(3,1) = 0.0; M(3,2) = 0.0; M(3,3) = 1.0;
#undef M

glMultMatrixd( m );

/* Translate Eye to Origin */
glTranslated( -ex, -ey, -ez );
}

```

$$M_v = \begin{bmatrix} u_x & u_y & u_z & 0 \\ v_x & v_y & v_z & 0 \\ n_x & n_y & n_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 & 0 & -e_x \\ 0 & 1 & 0 & -e_y \\ 0 & 0 & 1 & -e_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

glm::lookAt(*) 함수의 구현

```

namespace glm {
    ...
    template <typename T, precision P>
    GLM_FUNC_QUALIFIER tmat4x4<T, P> lookAt(
        tvec3<T, P> const & eye,
        tvec3<T, P> const & center,
        tvec3<T, P> const & up )
    {
        #ifdef GLM_LEFT_HANDED
            return lookAtLH(eye, center, up);
        #else
            return lookAtRH(eye, center, up);
        #endif
    }

    template <typename T, precision P>
    GLM_FUNC_QUALIFIER tmat4x4<T, P> lookAtRH(
        tvec3<T, P> const & eye,
        tvec3<T, P> const & center,
        tvec3<T, P> const & up )
    {
        tvec3<T, P> const f(normalize(center - eye));
        tvec3<T, P> const s(normalize(cross(f, up)));
        tvec3<T, P> const u(cross(s, f));
    }
}

```

```

tmat4x4<T, P> Result(1);
Result[0][0] = s.x;
Result[1][0] = s.y;
Result[2][0] = s.z;

Result[0][1] = u.x;
Result[1][1] = u.y;
Result[2][1] = u.z;

Result[0][2] = -f.x;
Result[1][2] = -f.y;
Result[2][2] = -f.z;

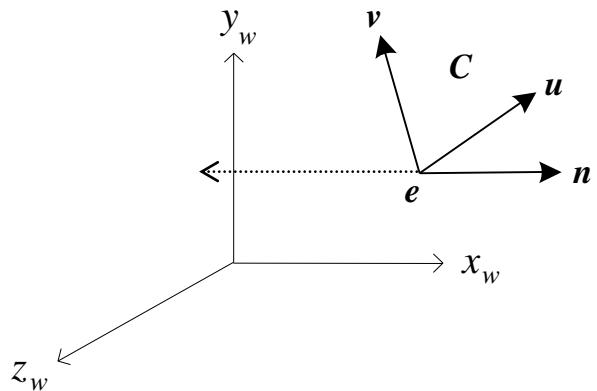
Result[3][0] = -dot(s, eye);
Result[3][1] = -dot(u, eye);
Result[3][2] = dot(f, eye);
return Result;
}

```

s (side)	→ u vector
u (up)	→ v vector
f (front)	→ - n vector

$$M_v = \begin{bmatrix} s_x & s_y & s_z & 0 \\ u_x & u_y & u_z & 0 \\ -f_x & -f_y & -f_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 & 0 & -e_x \\ 0 & 1 & 0 & -e_y \\ 0 & 0 & 1 & -e_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

세상 좌표계에서의 카메라의 표현



$$\mathbf{e} = (e_x, e_y, e_z)$$

$$\mathbf{u} = (u_x, u_y, u_z), \mathbf{v} = (v_x, v_y, v_z), \mathbf{n} = (n_x, n_y, n_z)$$

$$R = \begin{bmatrix} u_x & u_y & u_z & 0 \\ v_x & v_y & v_z & 0 \\ n_x & n_y & n_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

```
/* Viewing Transformation */
```

```
glMultMatrixf(R);
glTranslatef(-ex, -ey, -ez);
```

- ☺ 카메라의 위치와 방향에 대한 정보를 위의 \mathbf{C} 와 같이 세상 공간에서 이동 변환과 회전 변환을 통하여 자유롭게 움직이는 하나의 프레임으로 표현할 수 있음.

뷰잉 변환의 예 (Compatibility Profile)

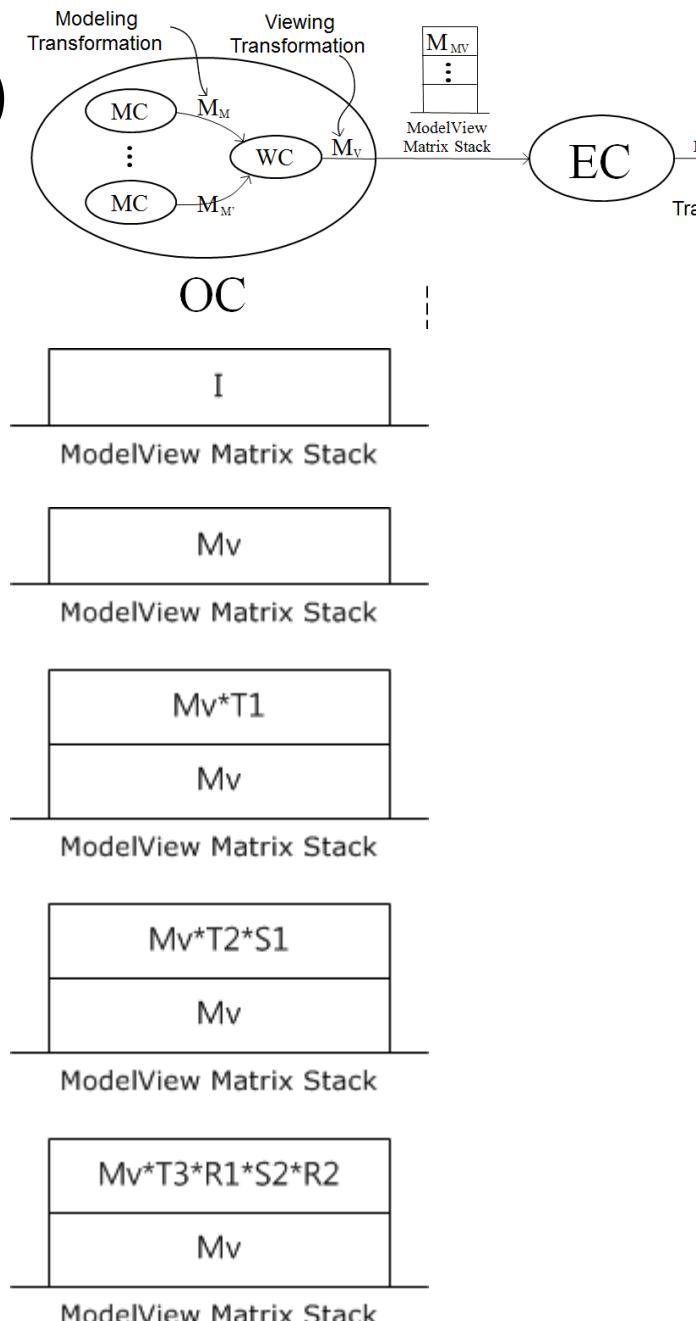
```
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
gluLookAt(2.0, 8.0, 8.0, 0.0, -4.0, 0.0, 0.0, 2.0, 0.0); // Mv
```

```
draw_axes_in_WC();
```

```
glMatrixMode(GL_MODELVIEW);
	glColor3f(1.0, 1.0, 0.0);
	glPushMatrix();
	 glTranslatef(0.0, 0.0, 1.5); // T1
	 draw_object_in_MC(teapot, npolytp);
	glPopMatrix();

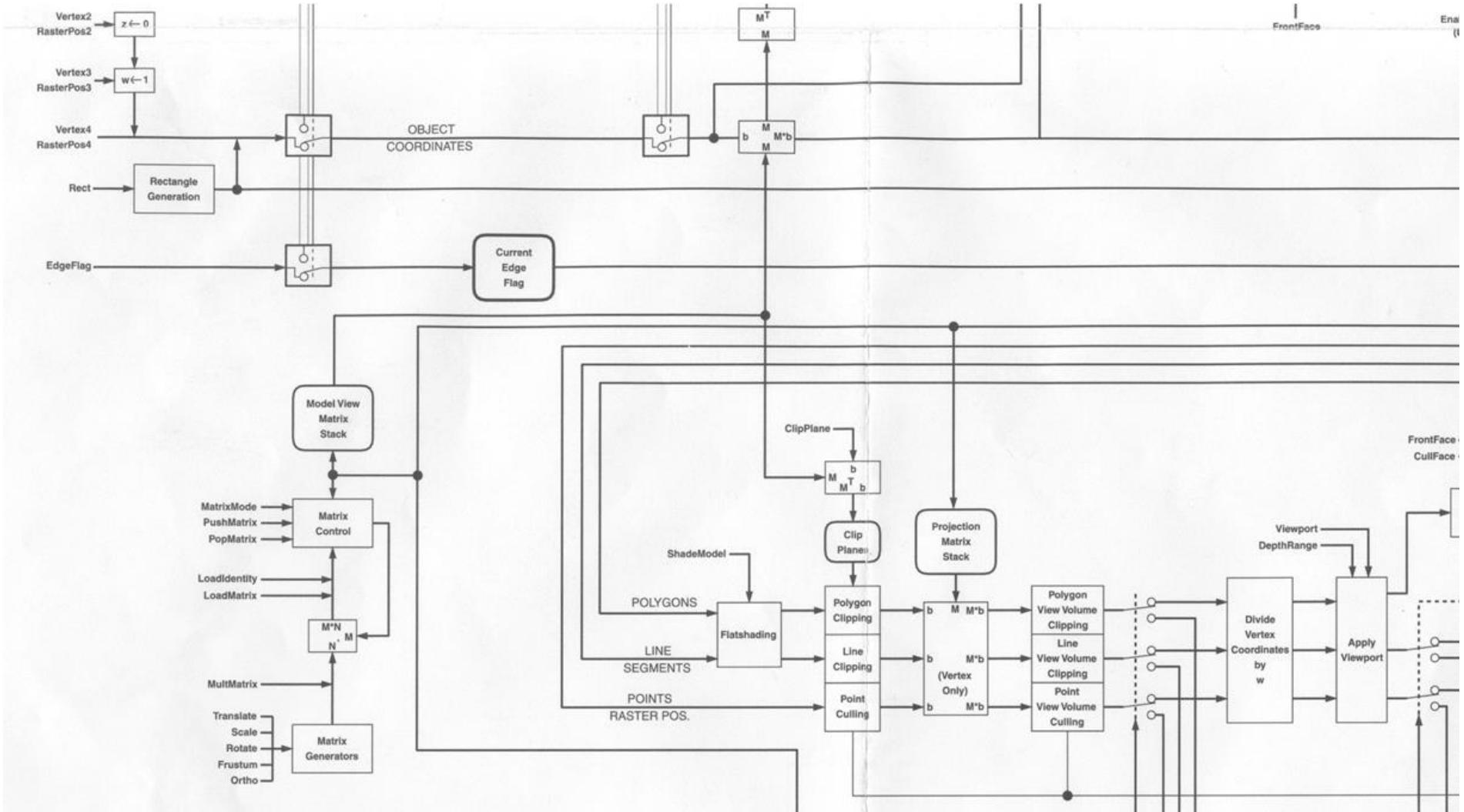
	glColor3d(1.0, 0.0, 1.0);
	glPushMatrix();
	 glTranslatef(-6.0, -5.0, 1.0); glScalef(2.0, 2.0, 2.0); // T2, S1
	 draw_object_in_MC(donut, npolydn);
	glPopMatrix();

	glColor3d(0.0, 1.0, 1.0);
	glPushMatrix();
	 glTranslatef(0.0, -6.0, 0.0); glRotatef(-45.0, 0.0, 1.0, 0.0); // x-->z T3, R1
	 glScalef(1.0, 1.0, -1.0); glRotatef(45.0, 0.0, 1.0, 0.0); // S2, R2
	 draw_object_in_MC(donut, npolydn);
	glPopMatrix();
```



- ◎ 실제로 기하 변환이 수행되는 순서와 해당하는 OpenGL 함수의 호출 순서는 정반대임!
 - ◎ `glPushMatrix()` 함수는 현재 행렬 스택의 탑의 내용을 보존하고 싶을 때, 그리고 `glPopMatrix()`은 더 이상 현재 행렬 스택의 탑의 내용이 필요 없어 제거하고 싶을 때 사용.
- 모델링 변환은 '계층적 구조'를 사용하여 복잡한 물체를 효율적으로 정의할 수 있도록 해줌.

Modeling and Viewing Transformation in OpenGL



뷰잉 변환의 예 (Core Profile)

```
void initialize_OpenGL(void) {
    ...
    // set up the initial viewing transformation
    ViewMatrix = glm::lookAt(glm::vec3(2.0f, 8.0f, 8.0f), glm::vec3(0.0f, -4.0f, 0.0f), glm::vec3(0.0f, 0.0f, 1.0f));
}

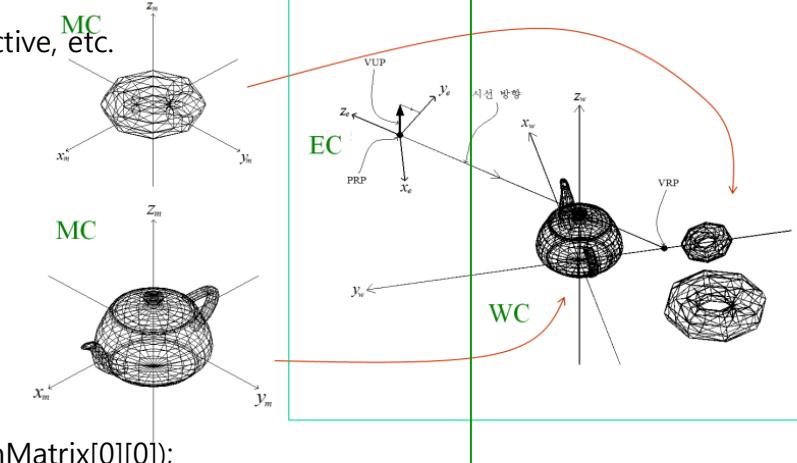
//#include <glm/glm.hpp>
#include <glm/gtc/matrix_transform.hpp> //translate, rotate, scale, lookAt, ortho, perspective, etc.
glm::mat4 ModelViewProjectionMatrix;
glm::mat4 ModelViewMatrix, ViewMatrix, ProjectionMatrix;

void display(void) {
    glClear(GL_COLOR_BUFFER_BIT);

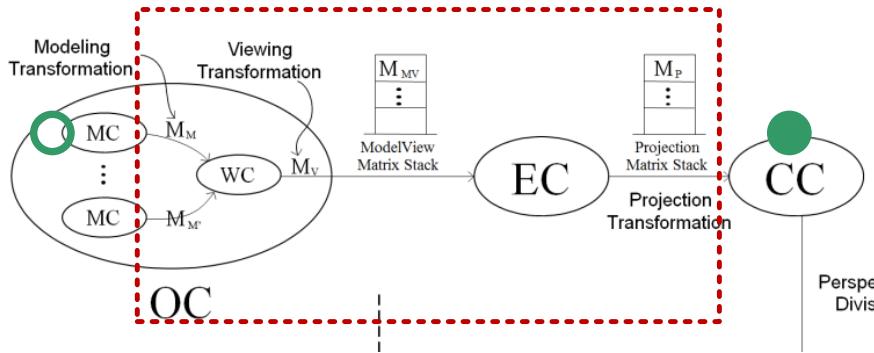
    ModelViewMatrix = glm::scale(ViewMatrix, glm::vec3(50.0f, 50.0f, 50.0f));
    ModelViewProjectionMatrix = ProjectionMatrix * ModelViewMatrix;
    glUniformMatrix4fv(loc_ModelViewProjectionMatrix, 1, GL_FALSE, &ModelViewProjectionMatrix[0][0]);
    glLineWidth(2.0f);
    draw_axes();
    glLineWidth(1.0f);

    ModelViewMatrix = glm::translate(ViewMatrix, glm::vec3(0.0f, 0.0f, 1.5f));
    ModelViewProjectionMatrix = ProjectionMatrix * ModelViewMatrix;
    glUniformMatrix4fv(loc_ModelViewProjectionMatrix, 1, GL_FALSE, &ModelViewProjectionMatrix[0][0]);
    glUniform3f(loc_primitive_color, 1.0f, 1.0f, 0.0f);
    draw_geom_obj(GEOM_OBJ_ID_TEAPOT);
}
```

- ✓ 뷰잉 변환 행렬은 필요할 때마다 계산
 - 카메라가 고정되어 있을 때
 - 카메라가 부정기적으로 움직일 때
 - 카메라가 매 프레임 움직일 때



ModelViewProjectionMatrix



```
#version 330
vertex shader

uniform mat4 ModelViewProjectionMatrix;
uniform vec3 primitive_color;

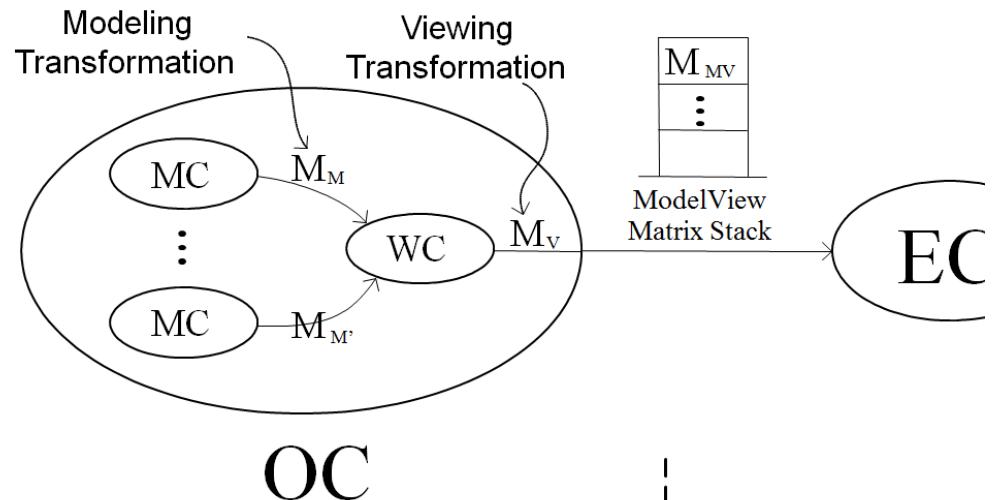
layout (location = 0) in vec4 a_position;
out vec4 color;

void main(void) {
    color = vec4(primitive_color, 1.0f);
    gl_Position =
        ModelViewProjectionMatrix
        * a_position;
}
```

물체 좌표계 (Object Coordinates, OC)

- OpenGL에서는 개념적으로 MC와 WC를 하나의 좌표계, 즉 OC로 간주함
- OC를 모델링 좌표계(Modeling Coordinates, MC)와 세상 좌표계(World Coordinates, WC)로 나누어 생각하면 편리.
 - 만약 모든 기하 물체들이 WC에서 정의가 되어있으면, OC는 WC에 해당하며, 뷰잉 변환만 수행하면 됨.
 - 만약 MC에 정의된 물체들을 모델링 변환을 통하여 WC로 보내주는 과정이 있다면, 모델링 변환과 뷰잉 변환 모두 필요함.
- 각 물체를 그리기 전에 ModelView Matrix Stack에는 적절히 M_{MV} 를 설정을 해주어야 함.

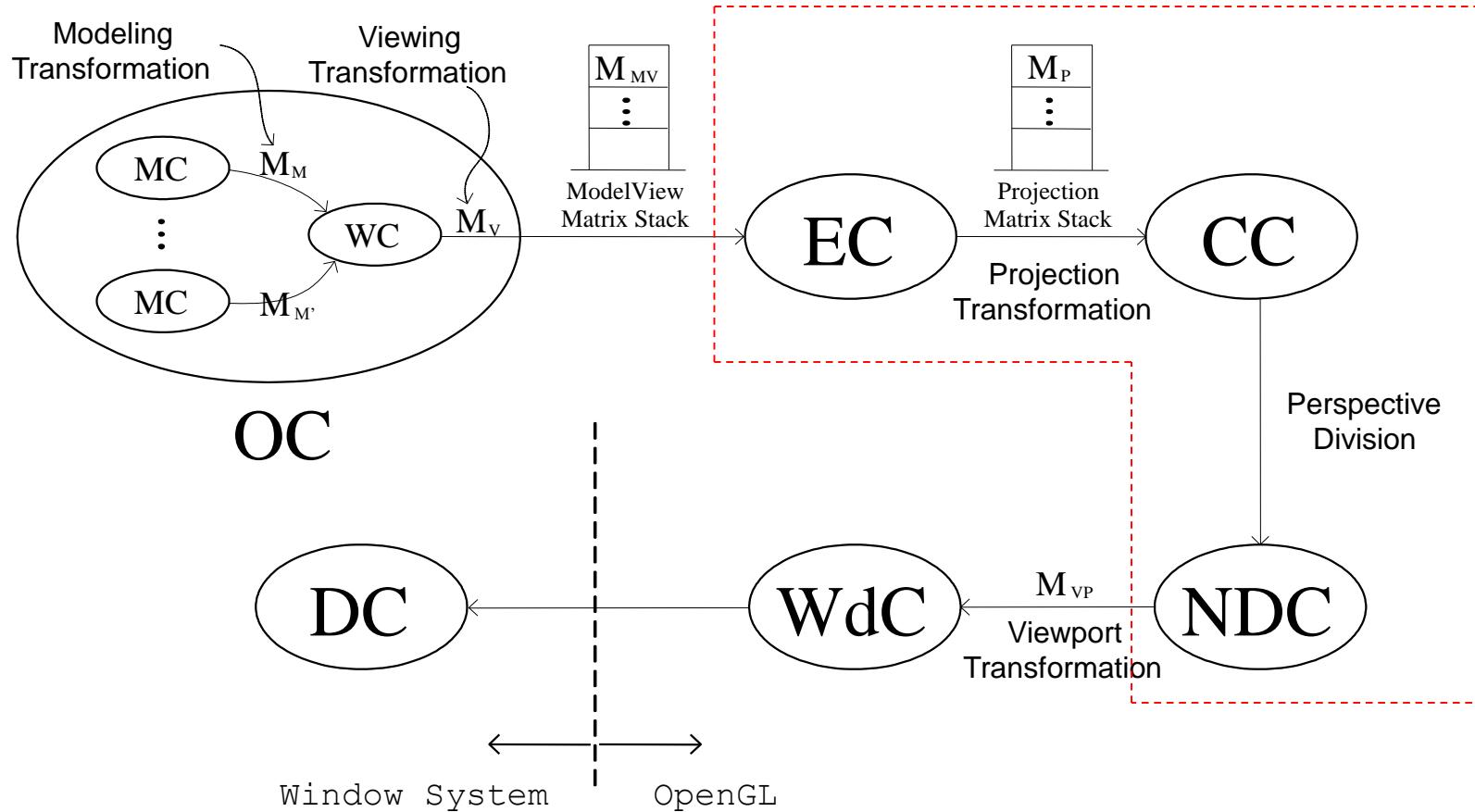
$$M_{MV} = M_V \cdot M_M$$



From Eye Coordinates to Normalized Device Coordinates: Projection Transformation

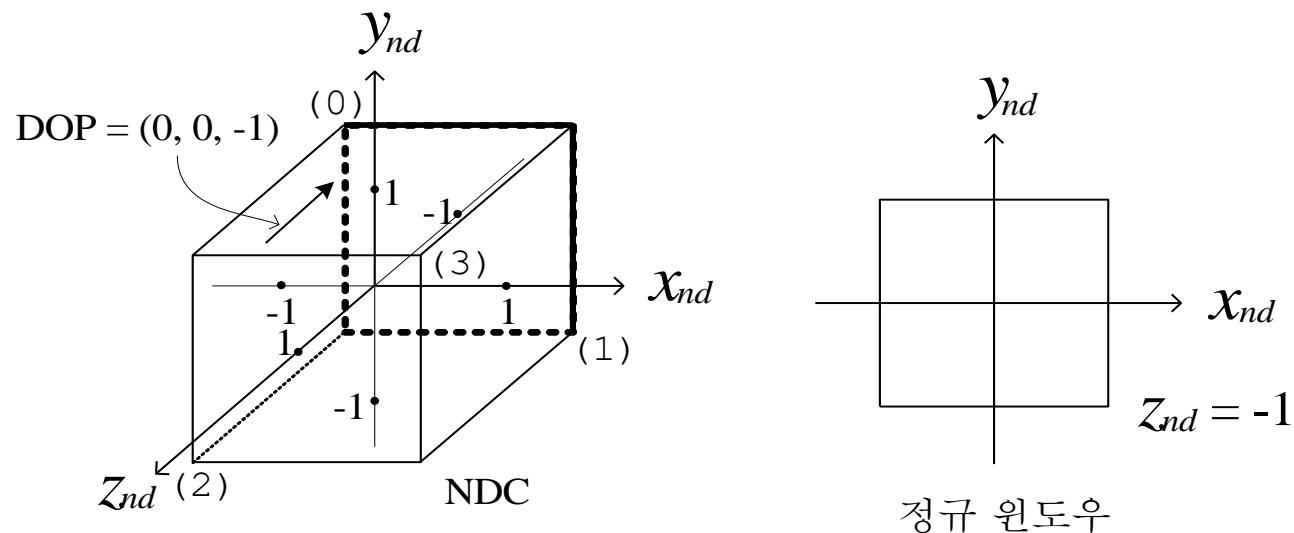
OpenGL의 기하 파이프라인과 사진 촬영과의 비교

- 모델링 변환(Modeling transformation)
 - 피사체 및 조명 배치
 - 모델링 좌표계(MC) → 세상 좌표계(WC)
- 뷰잉 변환(Viewing transformation)
 - 카메라의 위치와 방향 설정
 - 세상 좌표계(WC) → 눈 좌표계(EC)
- 투영 변환(Projection transformation)
 - 사용할 렌즈 결정 및 촬영 대상 결정 ← 카메라의 속성은 EC 기준으로 설정!
 - 카메라의 셔터를 누름
 - 눈 좌표계(EC) → 절단 좌표계(CC) → 정규 디바이스 좌표계(NDC)
- 뷰포트 변환(Viewport transformation)
 - 사진의 크기 결정 및 인화
 - 정규 디바이스 좌표계(NDC) → 윈도우 좌표계(WdC)



정규 디바이스 좌표계(Normalized Device Coordinates, NDC)

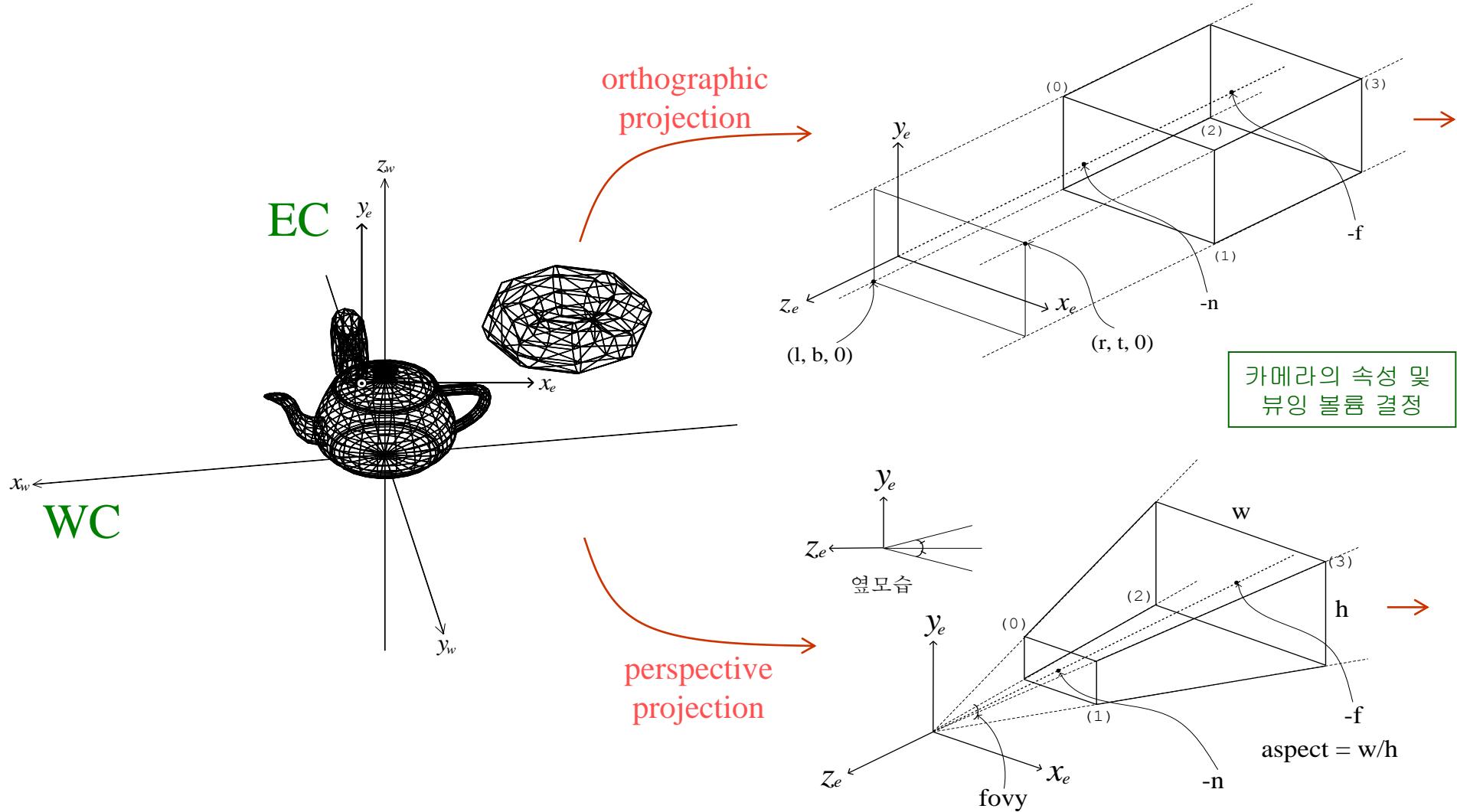
- '정규화된 3차원' 필름에 해당하는 정육면체 $[-1,1] \times [-1,1] \times [-1,1]$
 - 2차원 상 정보 (x_{nd}, y_{nd}) 와 깊이 정보 z_{nd} 포함.
- $(0, 0, -1)$ 방향으로 직교 투영함.
- 눈 좌표계에서 결정된 뷰잉 볼륨 영역이 정규 디바이스 좌표계의 정육면체 영역으로 매핑이 됨 → **뷰 매핑(view mapping)**
 - 3차원 공간에서 3차원 공간(2차원 공간이 아니라)으로의 매핑 변환 → **투영 변환**

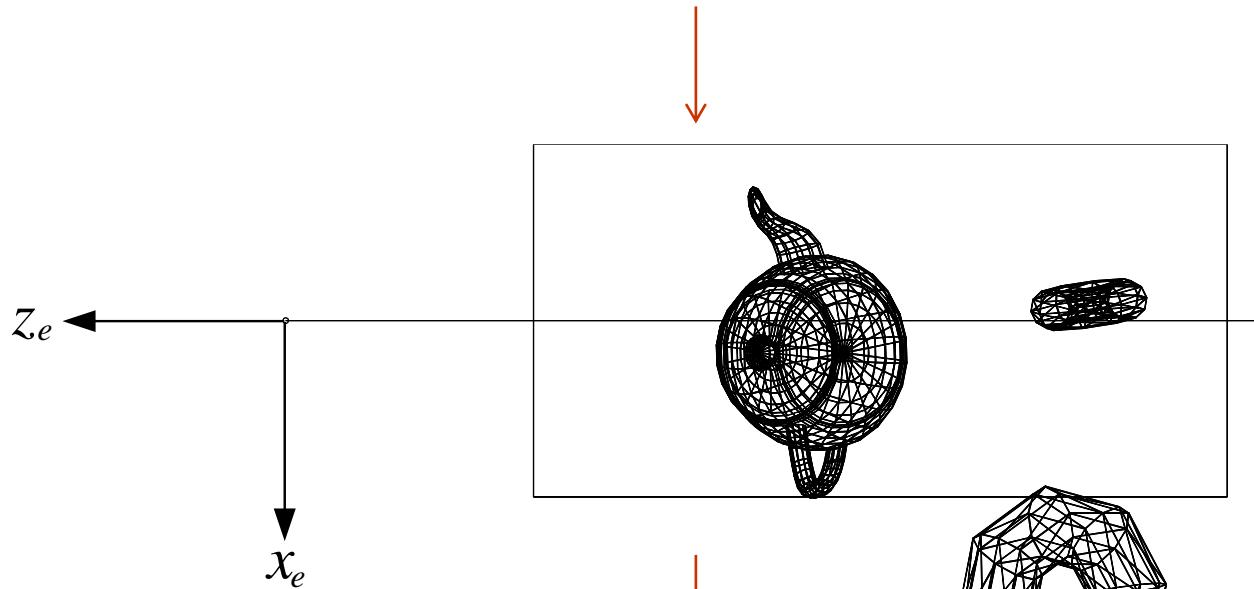


카메라의 속성 결정

- 줌 렌즈를 사용하듯이 사진에 찍힐 3차원 영역을 결정.
- 눈 좌표계 공간에서 **뷰잉 볼륨(viewing volume)** 또는 **뷰 볼륨(view volume)**을 설정하는 것이 목적
- 뷔잉 볼륨의 영역이 정규 디바이스 좌표계(NDC)의 정육면체 영역으로 **뷰 맵핑(view mapping)**됨. → **투영 변환(Projection Transformation)**
- OpenGL에서는 기본적으로 **직교 투영과 원근 투영**을 제공함.
 - 단 경사 투영과 같은 다른 형태의 평면 기하 투영도 변환 행렬을 직접 계산함으로써 구현 가능함.

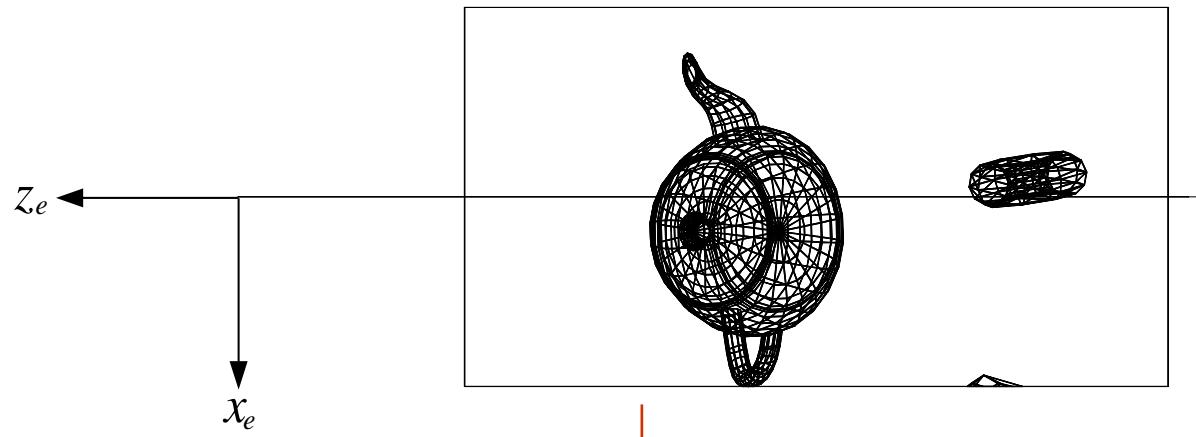
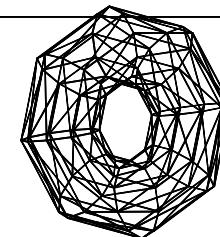
투영 변환 과정 개관

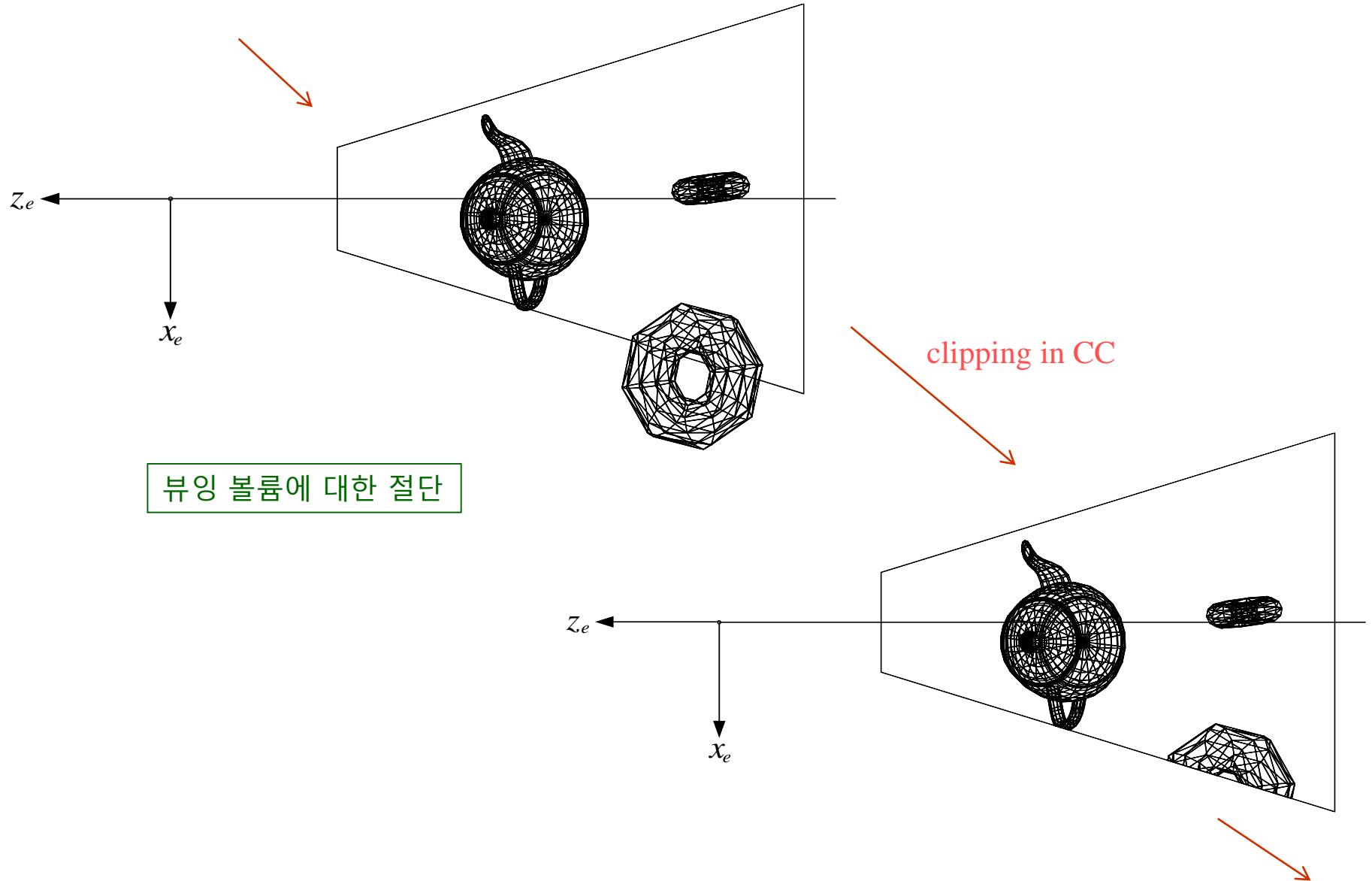


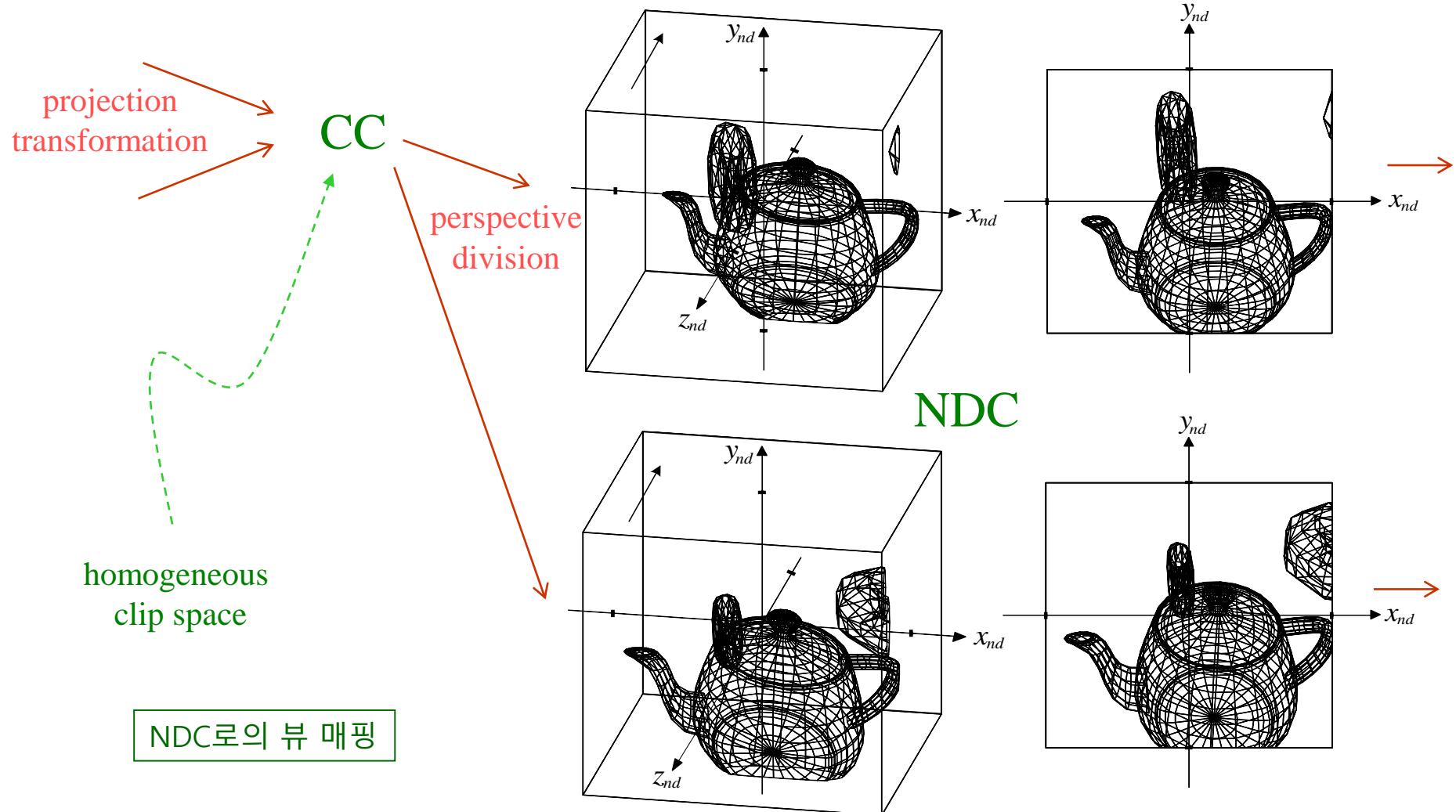


clipping in CC

뷰잉 볼륨에 대한 절단



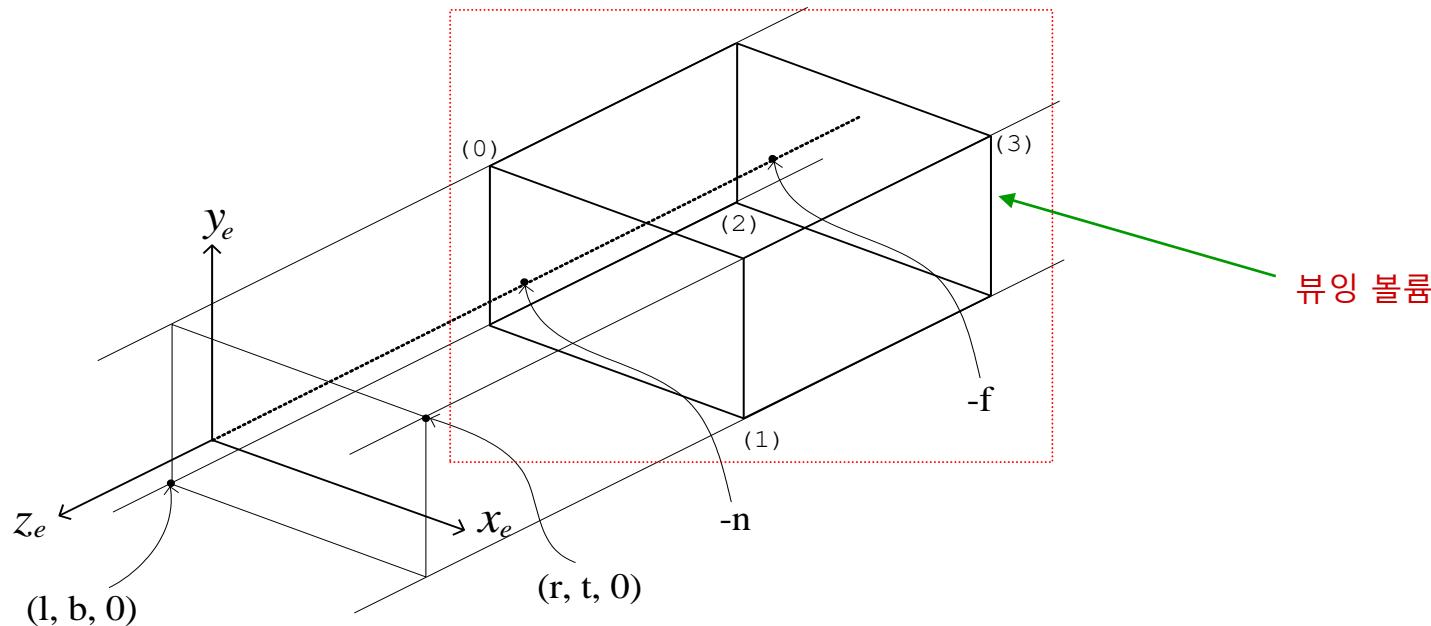




OpenGL에서의 직교 투영

```
void glOrtho(GLdouble l, GLdouble r, GLdouble b, GLdouble t, GLdouble n, GLdouble f);
```

```
glm::mat4 glm::ortho(float left, float right, float bottom, float top, float zNear, float zFar);
```



$$M_{ortho} = \begin{bmatrix} \frac{2}{r-l} & 0 & 0 & -\frac{r+l}{r-l} \\ 0 & \frac{2}{t-b} & 0 & -\frac{t+b}{t-b} \\ 0 & 0 & -\frac{2}{f-n} & -\frac{f+n}{f-n} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

glm::ortho(*) 함수의 구현

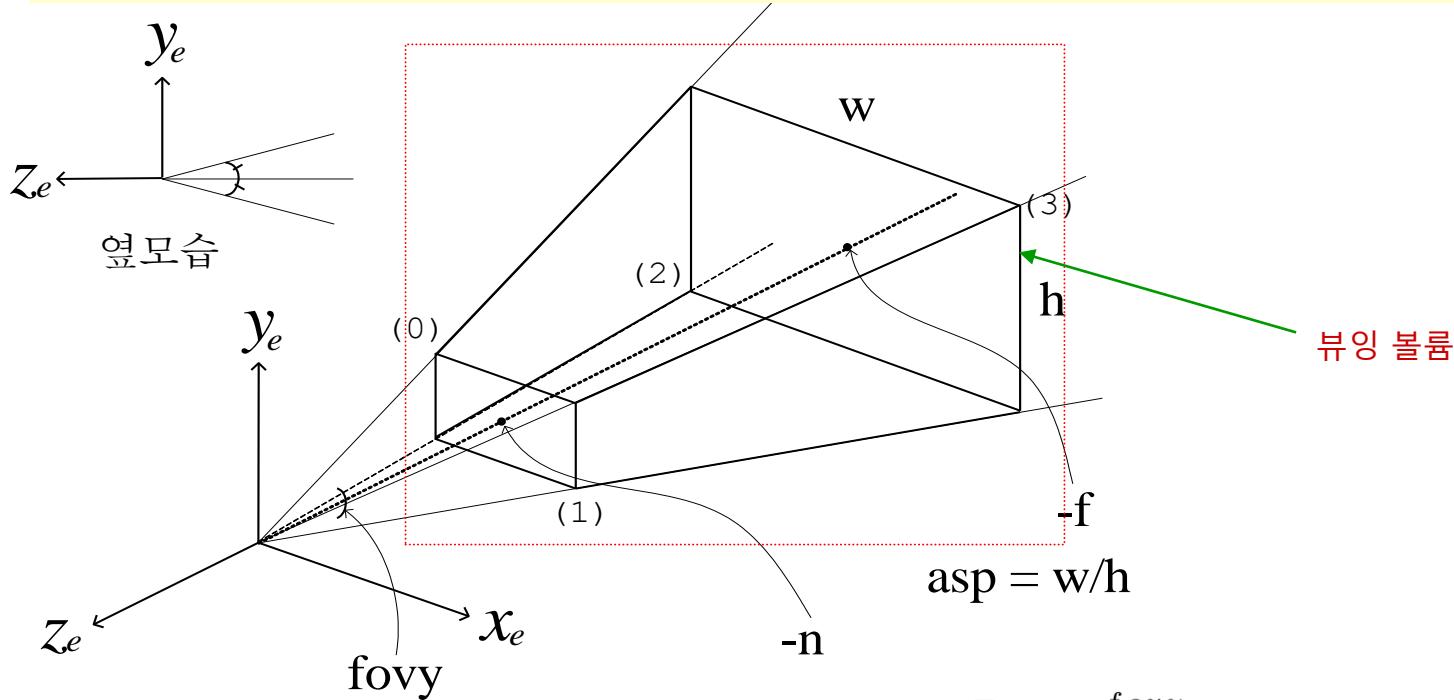
```
namespace glm {  
    ...  
    template <typename T>  
    GLM_FUNC_QUALIFIER tmat4x4<T, defaultp> ortho(T left, T right, T bottom, T top,  
                                              T zNear, T zFar ) {  
        tmat4x4<T, defaultp> Result(1);  
  
        Result[0][0] = static_cast<T>(2) / (right - left);  
        Result[1][1] = static_cast<T>(2) / (top - bottom);  
        Result[2][2] = - static_cast<T>(2) / (zFar - zNear);  
        Result[3][0] = - (right + left) / (right - left);  
        Result[3][1] = - (top + bottom) / (top - bottom);  
        Result[3][2] = - (zFar + zNear) / (zFar - zNear);  
  
        return Result;  
    }  
}
```

$$M_{ortho} = \begin{bmatrix} \frac{2}{r-l} & 0 & 0 & -\frac{r+l}{r-l} \\ 0 & \frac{2}{t-b} & 0 & -\frac{t+b}{t-b} \\ 0 & 0 & -\frac{2}{f-n} & -\frac{f+n}{f-n} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

OpenGL에서의 원근 투영 I: gluPerspective(*)

```
void gluPerspective(GLdouble fovy, GLdouble asp, GLdouble n, GLdouble f);
```

```
glm::mat4 perspective(float fovy, float aspect, float zNear, float zFar);
```



$$M_{pers} = \begin{bmatrix} \frac{\cot(\frac{fovy}{2})}{asp} & 0 & 0 & 0 \\ 0 & \cot(\frac{fovy}{2}) & 0 & 0 \\ 0 & 0 & -\frac{f+n}{f-n} & -\frac{2nf}{f-n} \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

glm::perspective(*) 함수의 구현

```
namespace glm {  
    ...  
  
    template <typename T>  
    GLM_FUNC_QUALIFIER tmat4x4<T, defaultp>  
        perspective( T fovy, T aspect, T zNear,  
                      T zFar ) {  
  
        #ifdef GLM_LEFT_HANDED  
            return perspectiveLH(fovy, aspect, zNear, zFar);  
        #else  
            return perspectiveRH(fovy, aspect, zNear, zFar);  
        #endif  
    }  
  
    template <typename T>  
    GLM_FUNC_QUALIFIER tmat4x4<T, defaultp>  
        perspectiveRH( T fovy, T aspect, T zNear, T zFar ) {  
        assert(abs(aspect - std::numeric_limits<T>::epsilon())  
              > static_cast<T>(0));  
    }  
}
```

```
T const tanHalfFovy = tan(fovy / static_cast<T>(2));  
  
tmat4x4<T, defaultp> Result(static_cast<T>(0));  
  
Result[0][0] = static_cast<T>(1) / (aspect * tanHalfFovy);  
Result[1][1] = static_cast<T>(1) / (tanHalfFovy);  
Result[2][2] = - (zFar + zNear) / (zFar - zNear);  
Result[2][3] = - static_cast<T>(1);  
Result[3][2] = - (static_cast<T>(2) * zFar * zNear)  
                / (zFar - zNear);  
  
return Result;  
}
```

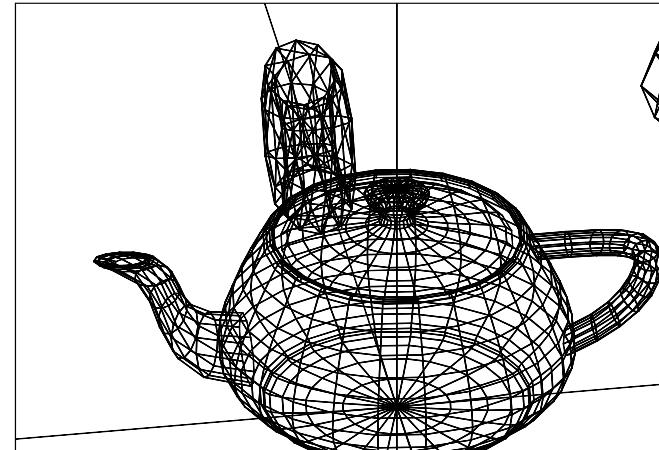
$$M_{pers} = \begin{bmatrix} \frac{\cot(\frac{fovy}{2})}{asp} & 0 & 0 & 0 \\ 0 & \cot(\frac{fovy}{2}) & 0 & 0 \\ 0 & 0 & -\frac{f+n}{f-n} & -\frac{2nf}{f-n} \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

- 앞 절단 평면(front clipping plane)과 뒤 절단 평면(back clipping plane)으로 유한 영역 결정
 - 가급적 이 두 평면의 간격은 좁히는 것이 좋음.
- 상의 가로-세로 비율(aspect ratio)이 결정이 되며, 추후 윈도우 좌표계에서 설정이 되는 뷰포트의 비율과 일치하지 않으면 공간의 왜곡이 생김.
- 투영 변환 함수를 호출하면
 - ① 뷰잉 볼륨이 결정이 된 후,
 - ② 눈 좌표계에서 정규 디바이스 좌표계로의 투영 변환 행렬 M_p 가 계산되어,
 - ③ 현재 행렬 스택의 탑의 오른쪽에 곱해짐.

투영 변환의 예 (Compatibility Profile)

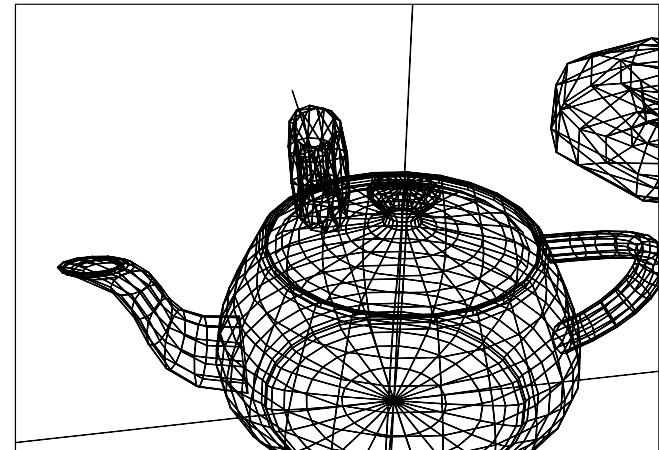
직교 투영

```
glMatrixMode(GL_PROJECTION);  
glLoadIdentity();  
glOrtho(-3.6, 3.6, -2.7, 2.7, 5, 19.0);
```



원근 투영

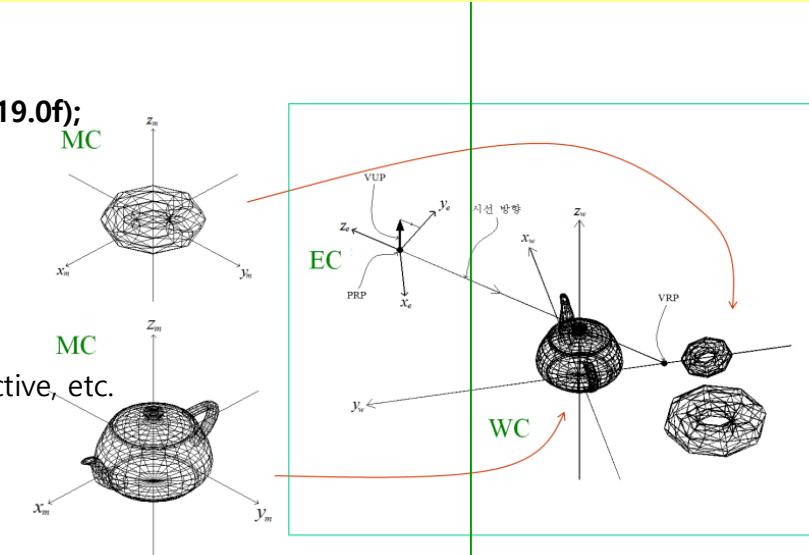
```
glMatrixMode(PROJECTION);  
glLoadIdentity();  
gluPerspective(28.0, 800.0/600.0, 5.0, 19.0);
```



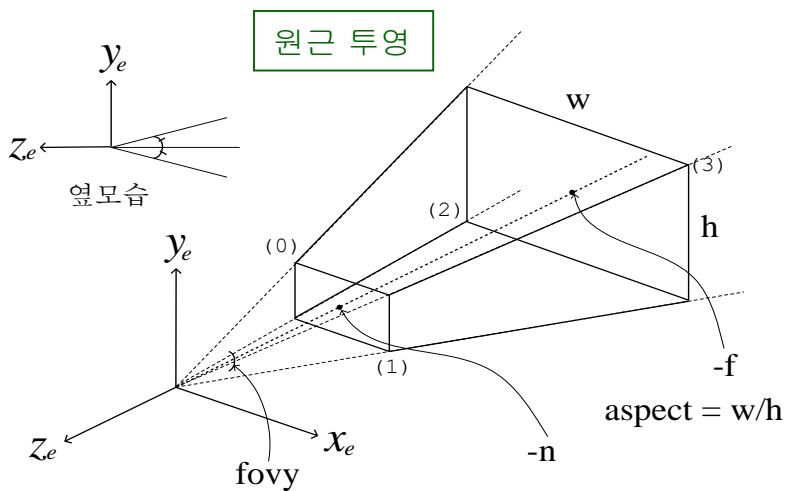
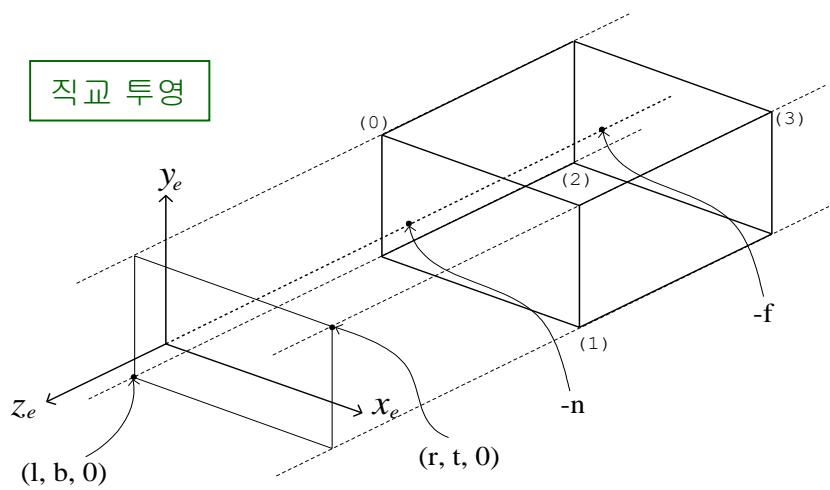
투영 변환의 예 (Core Profile)

```
void reshape(int width, int height) {  
    // set up the initial viewport transformation  
    glViewport(0, 0, width, height);  
  
    window_aspect_ratio = (float)width / height;  
    // set up the initial projection transformation  
    ProjectionMatrix = glm::perspective(28.0f*TO_RADIAN, window_aspect_ratio, 5.0f, 19.0f);  
  
    glutPostRedisplay();  
}  
  
-----  
//#include <glm/glm.hpp>  
#include <glm/gtc/matrix_transform.hpp> //translate, rotate, scale, lookAt, ortho, perspective, etc.  
glm::mat4 ModelViewProjectionMatrix;  
glm::mat4 ModelViewMatrix, ViewMatrix, ProjectionMatrix;  
  
void display(void) {  
    glClear(GL_COLOR_BUFFER_BIT);  
    ...  
  
    ModelViewMatrix = glm::translate(ViewMatrix, glm::vec3(0.0f, 0.0f, 1.5f));  
    ModelViewProjectionMatrix = ProjectionMatrix * ModelViewMatrix;  
    glUniformMatrix4fv(loc_ModelViewProjectionMatrix, 1, GL_FALSE, &ModelViewProjectionMatrix[0][0]);  
    glUniform3f(loc_primitive_color, 1.0f, 1.0f, 0.0f);  
    draw_geom_obj(GEOM_OBJ_ID_TEAPOT);  
}
```

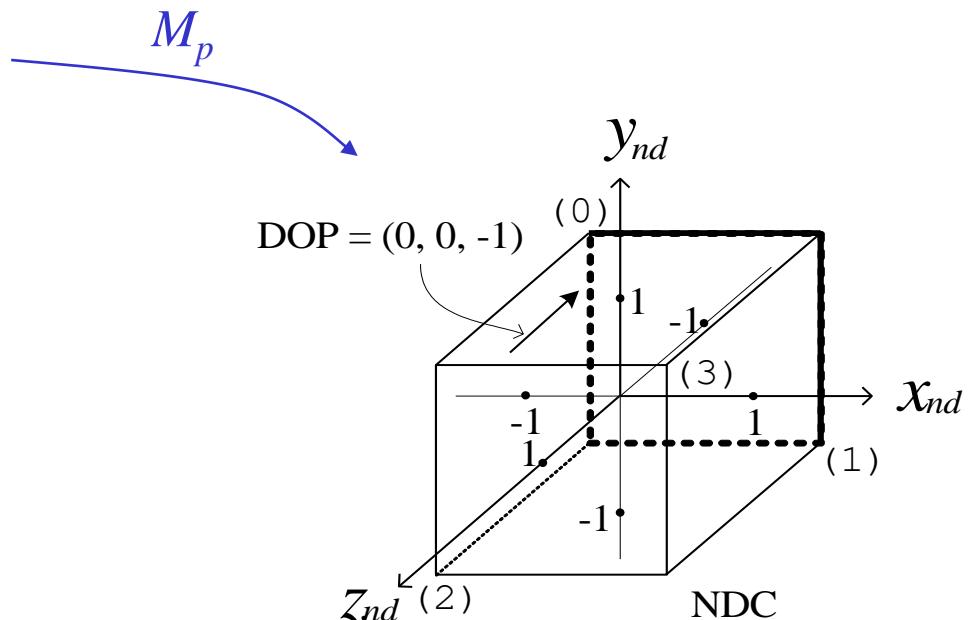
- ✓ 투영 변환 행렬은 필요할 때마다 계산
 - 처음 윈도우가 화면에 뜰 때
 - 윈도우의 크기가 변할 때
 - 줌-인/줌-아웃 연산을 통해 카메라 인자가 변할 때
 - 기타



뷰 매핑: 투영 변환 M_p



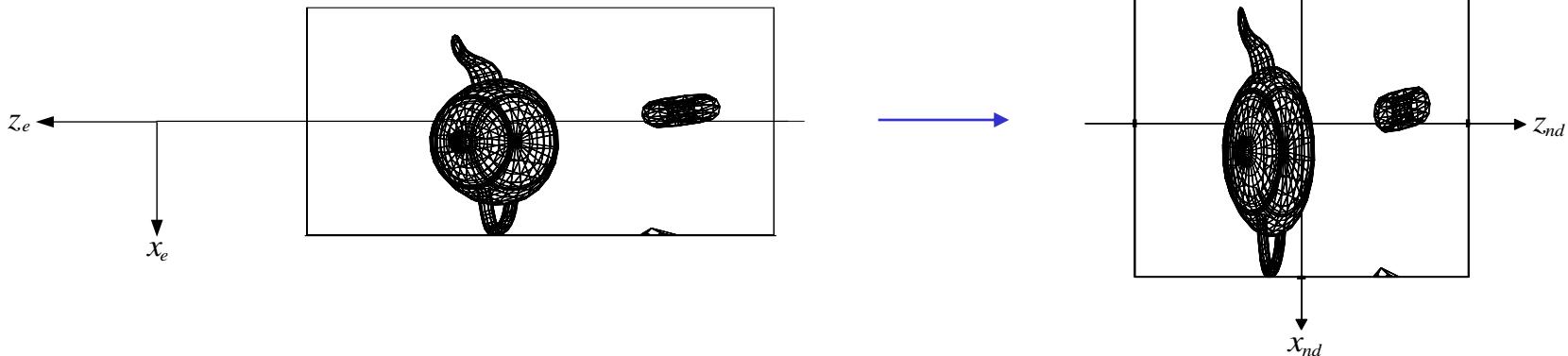
- 직육면체 → 정육면체, 직교 투영 → 직교 투영
- 아핀 변환



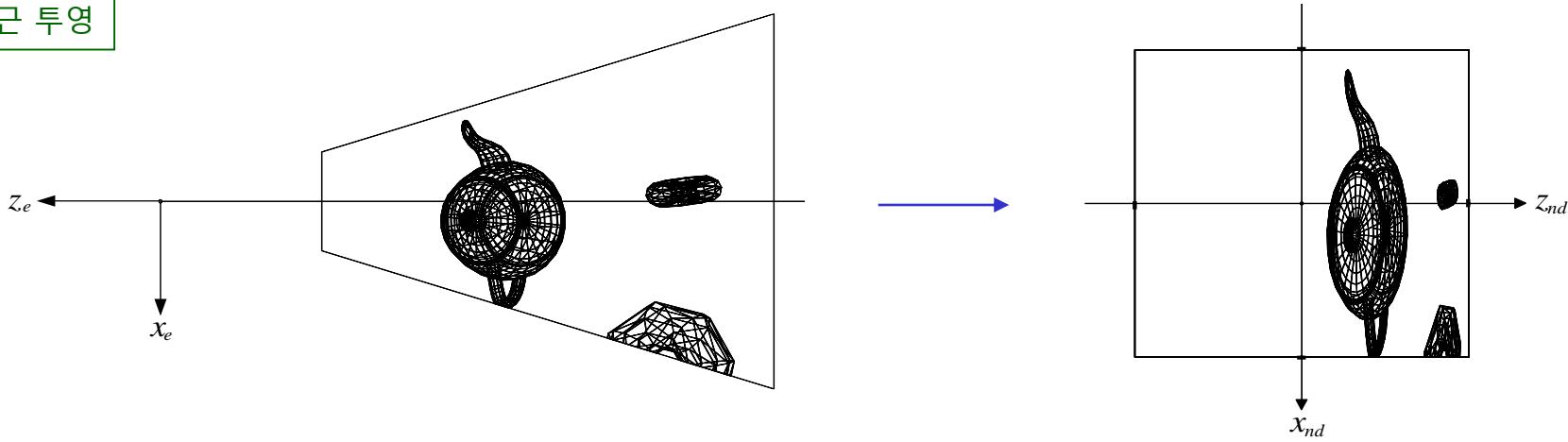
- M_p
- 사각뿔 → 정육면체, 원근 투영 → 직교 투영
 - 비아핀 변환: 원근 변환(perspective trans.)
 - 한 점으로 모이던 투영선들이 평행해짐 → 원근감 생성
 - 원근 나눗셈(perspective division)
 - 양의 z_e 축 방향으로 공간이 밀림.

뷰 매핑 과정(위에서 본 모습)

직교 투영

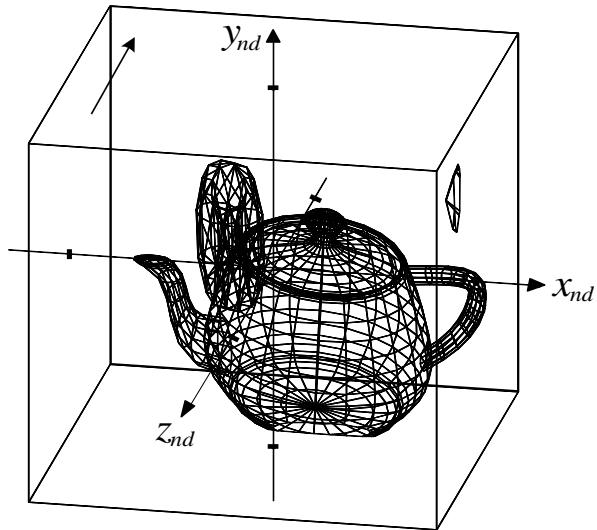


원근 투영

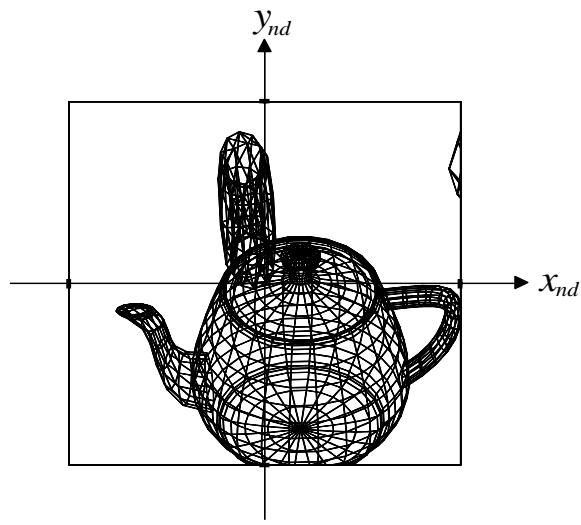
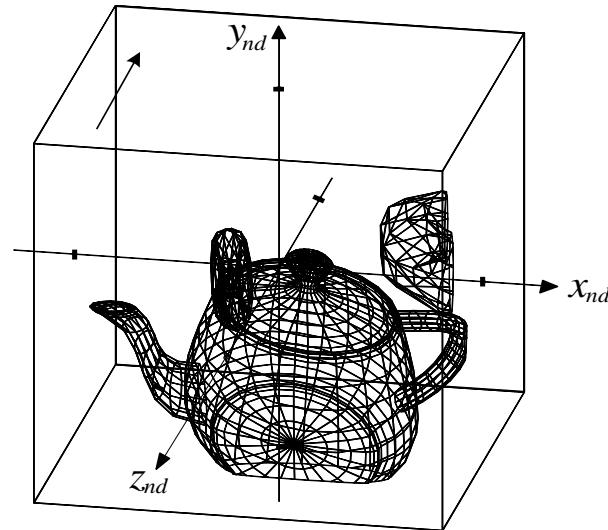


뷰 매핑 후의 정규 디바이스 좌표계 공간의 모습

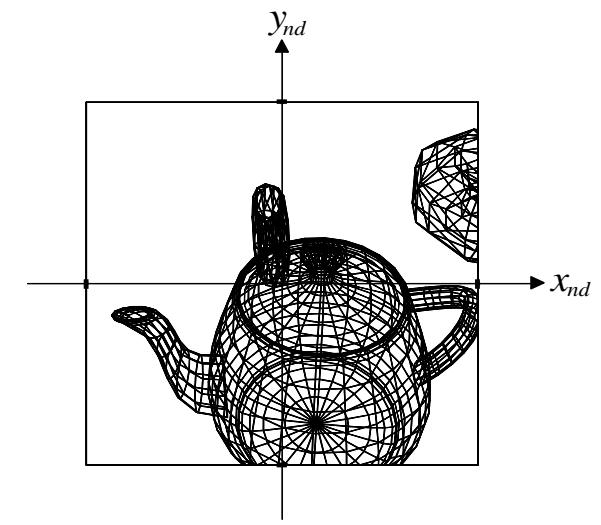
직교 투영



원근 투영



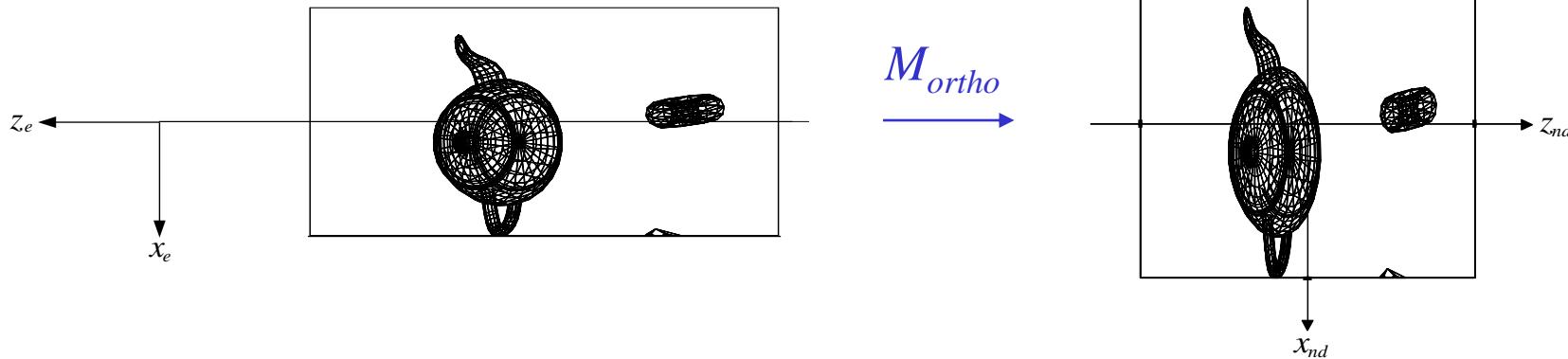
정규 윈도우



투영 변환의 M_p 유도: 직교 변환 M_{ortho}

$$\begin{aligned}
 M_{ortho} &= S(1, 1, -1) \cdot S\left(\frac{2}{r-l}, \frac{2}{t-b}, \frac{2}{f-n}\right) \cdot T\left(-\frac{l+r}{2}, -\frac{b+t}{2}, \frac{n+f}{2}\right) \\
 &= \begin{bmatrix} \frac{2}{r-l} & 0 & 0 & -\frac{r+l}{r-l} \\ 0 & \frac{2}{t-b} & 0 & -\frac{t+b}{t-b} \\ 0 & 0 & -\frac{2}{f-n} & -\frac{f+n}{f-n} \\ 0 & 0 & 0 & 1 \end{bmatrix}
 \end{aligned}$$

☺ 직교 변환은 아핀 변환
 → 평행성 유지와 비율의 보존
 → 원근감 X
 ☺ 3차원공간 → 3차원 공간
 → 행렬의 계수 = 4

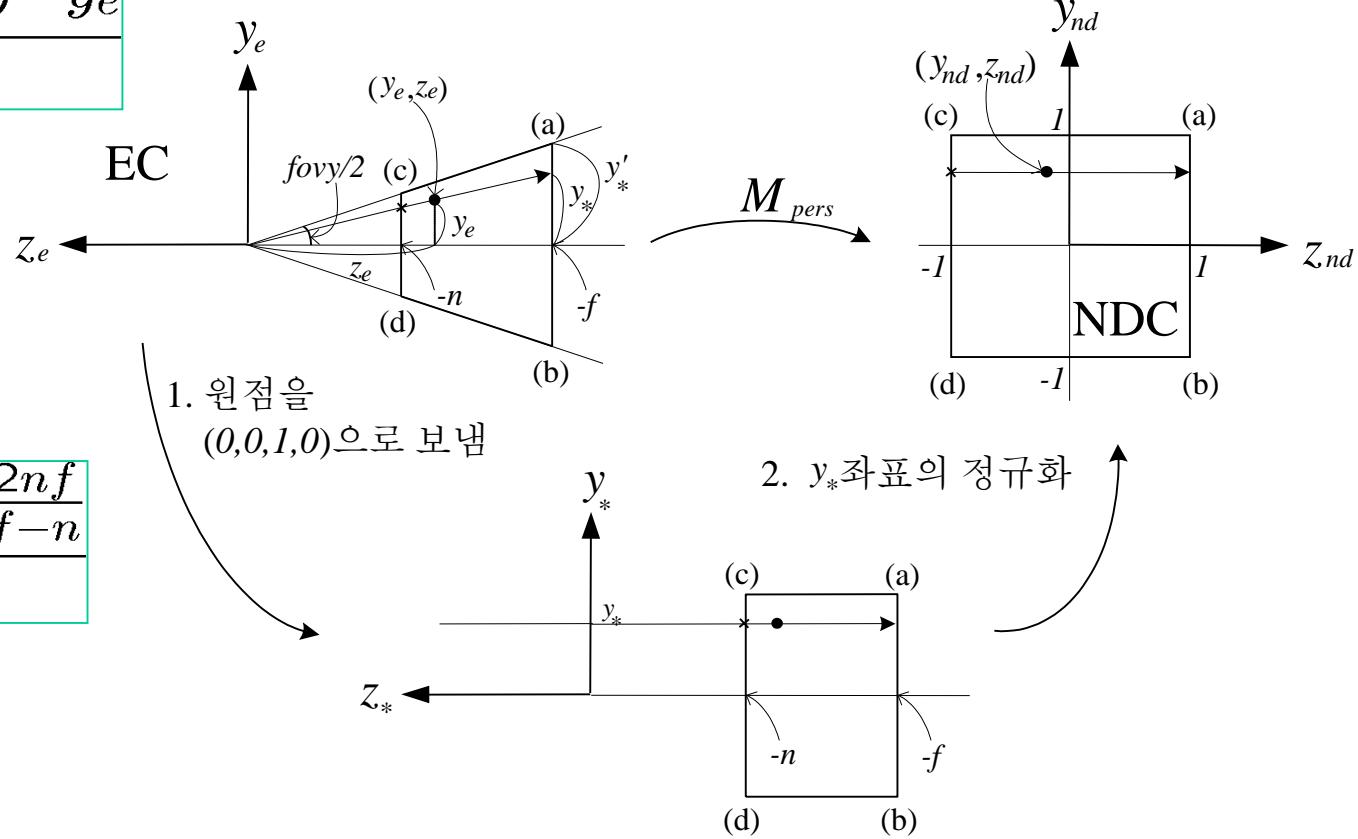


투영 변환의 M_p 유도: 원근 변환 M_{pers}

$$x_{nd} = \frac{x_*}{x'_*} = \frac{\cot(\frac{fovy}{2})}{asp} \cdot x_e$$

$$y_{nd} = \frac{y_*}{y'_*} = \frac{\cot(\frac{fovy}{2}) \cdot y_e}{-z_e}$$

$$z_{nd} = \frac{-\frac{f+n}{f-n}z_e - \frac{2nf}{f-n}}{-z_e}$$



$x_c = \frac{\cot(\frac{fov_y}{2})}{asp} \cdot x_e, y_c = \cot(\frac{fov_y}{2}) \cdot y_e, z_c = -\frac{f+n}{f-n} \cdot z_e - \frac{2nf}{f-n}, w_c = -z_e$ 라 하면

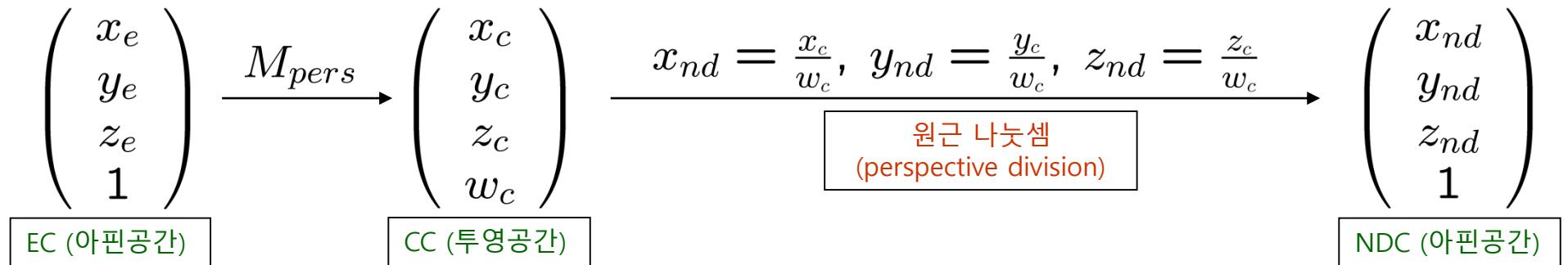
$$\begin{pmatrix} x_c \\ y_c \\ z_c \\ w_c \end{pmatrix} = \begin{bmatrix} \frac{\cot(\frac{fov_y}{2})}{asp} & 0 & 0 & 0 \\ 0 & \cot(\frac{fov_y}{2}) & 0 & 0 \\ 0 & 0 & -\frac{f+n}{f-n} & -\frac{2nf}{f-n} \\ 0 & 0 & -1 & 0 \end{bmatrix} \begin{pmatrix} x_e \\ y_e \\ z_e \\ 1 \end{pmatrix} \equiv M_{pers} \begin{pmatrix} x_e \\ y_e \\ z_e \\ 1 \end{pmatrix}$$

M_{pers}

$$x_{nd} = \frac{x_c}{w_c}, y_{nd} = \frac{y_c}{w_c}, z_{nd} = \frac{z_c}{w_c}$$

OpenGL 원근 변환의 특징

- 직관적인 변환 과정



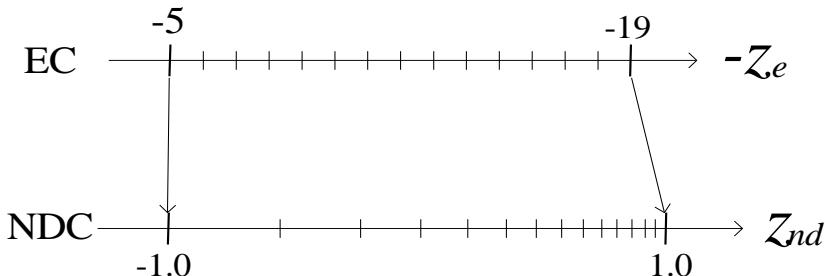
- 비아핀 변환: M_{pers} 의 네 번째 행은 $(0 \ 0 \ -1 \ 0)$ 임.
- 절단 좌표계(Clip Coordinates, CC)에서 뷰잉 볼륨에 대한 절단 계산이 수행이 됨.
- 원근 나눗셈의 의미
 - 원근감 생성 ($w_c = -z_e$ 의 의미)
 - 눈 좌표계의 원점에 있는 PRP를 z_e 축 방향의 무한대로 보내줌.

원근 투영을 할 경우 EC와 NDC 사이에는 z축 방향으로 공간이 왜곡됨.
 → 추후 rasterization 과정에서 perspective correction이 필요함.

- z 축 방향으로의 공간의 찌그러짐

- NDC(또는 NDC와 선형 관계에 있는 WdC)에서는 해당 구간이 균일하게 샘플링됨
→ 샘플링 간격의 불균일 성.
- 만약 32비트 Z-버퍼를 사용할 경우 2^{32} 개의 구간으로 샘플링됨.
- 가능하면 앞, 뒤 절단 평면의 간격을 좁힐 것.

$$z_{nd} = \frac{-\frac{f+n}{f-n}z_e - \frac{2nf}{f-n}}{-z_e}$$

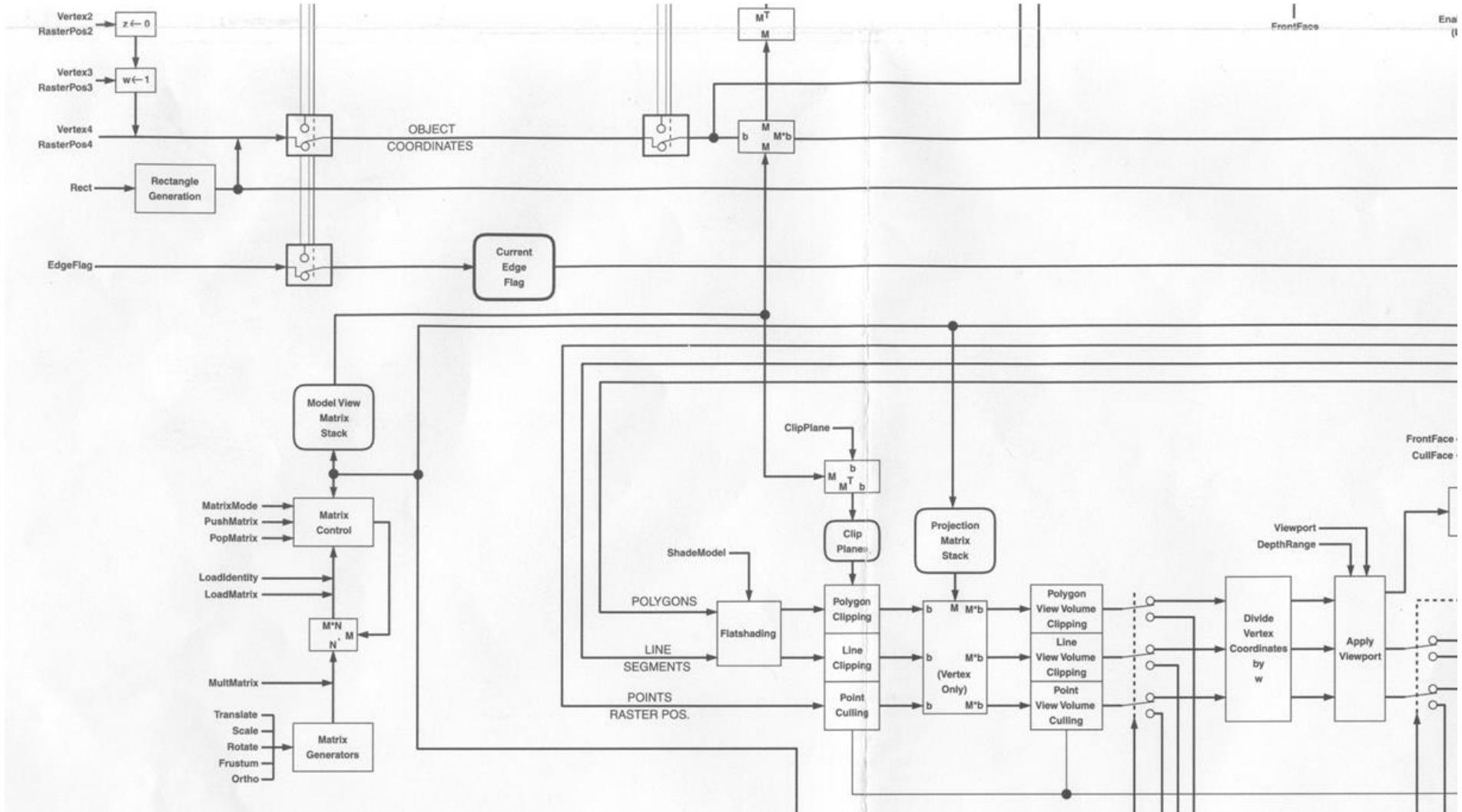


- 두 투영 변환의 비교

- 직교 투영
 - 아핀 공간 \rightarrow 아핀 공간
 - 아핀 변환
- 원근 투영
 - 아핀 공간 \rightarrow 투영 공간 \rightarrow 아핀 공간
 - 비아핀 변환

투영 공간 관점에서는 차이가 없음!

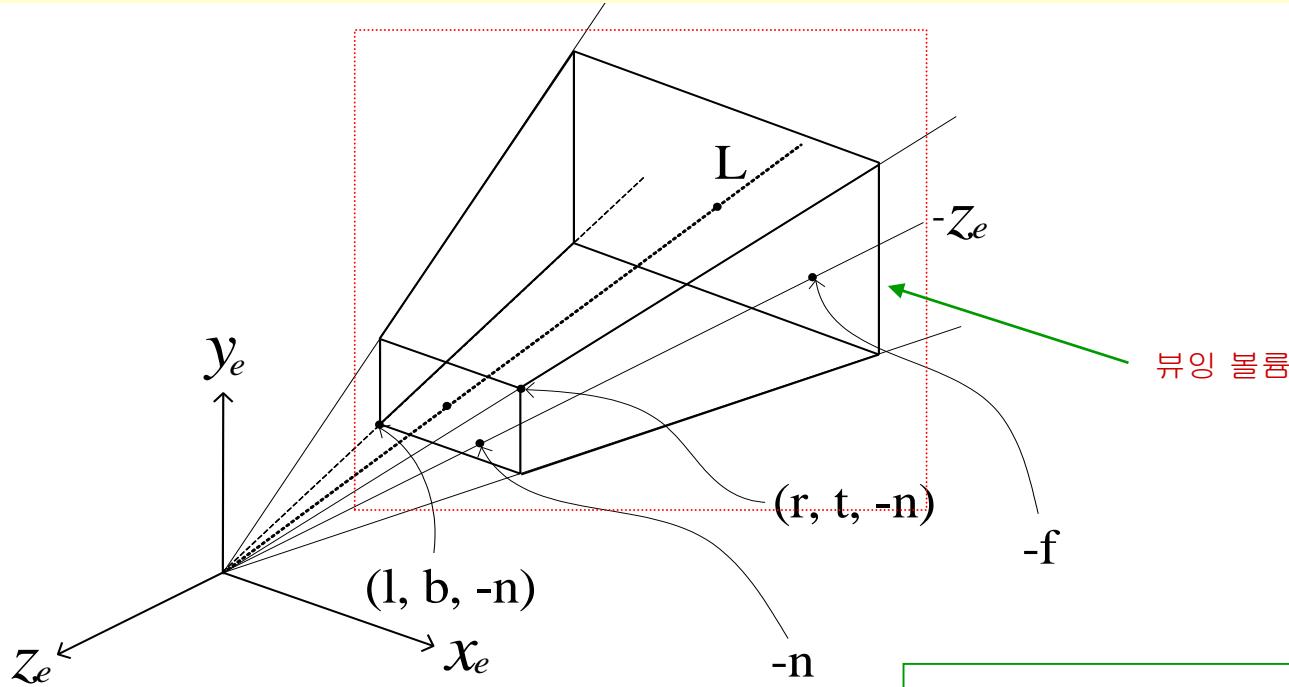
Projection Transformation in OpenGL



OpenGL에서의 원근 투영 II: glFrustum(*)

void **glFrustum**(GLdouble l, GLdouble r, GLdouble b, GLdouble t, GLdouble n, GLdouble f);

glm::mat4 **glm::frustum**(float left, float right, float bottom, float top, float zNear, float zFar);



$$M' = \begin{bmatrix} 1 & 0 & \frac{r+l}{2n} & 0 \\ 0 & 1 & \frac{t+b}{2n} & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$M_{frus} = M_{pers} \cdot M' = \begin{bmatrix} \frac{2n}{r-l} & 0 & \frac{r+l}{r-l} & 0 \\ 0 & \frac{2n}{t-b} & \frac{t+b}{t-b} & 0 \\ 0 & 0 & -\frac{f+n}{f-n} & -\frac{2nf}{f-n} \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

glm::frustum(*) 함수의 구현

```
namespace glm {  
    ...  
  
    template <typename T>  
    GLM_FUNC_QUALIFIER tmat4x4<T, defaulttp> frustum(  
        T left, T right, T bottom, T top, T nearVal, T farVal ) {  
        tmat4x4<T, defaulttp> Result(0);  
        Result[0][0] = (static_cast<T>(2) * nearVal) / (right - left);  
        Result[1][1] = (static_cast<T>(2) * nearVal) / (top - bottom);  
        Result[2][0] = (right + left) / (right - left);  
        Result[2][1] = (top + bottom) / (top - bottom);  
        Result[2][2] = -(farVal + nearVal) / (farVal - nearVal);  
        Result[2][3] = static_cast<T>(-1);  
        Result[3][2] = -(static_cast<T>(2) * farVal * nearVal) / (farVal - nearVal);  
        return Result;  
    }  
}
```

$$M_{frus} = M_{pers} \cdot M' = \begin{bmatrix} \frac{2n}{r-l} & 0 & \frac{r+l}{r-l} & 0 \\ 0 & \frac{2n}{t-b} & \frac{t+b}{t-b} & 0 \\ 0 & 0 & -\frac{f+n}{f-n} & -\frac{2nf}{f-n} \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

gluPerspective(*) 함수의 구현

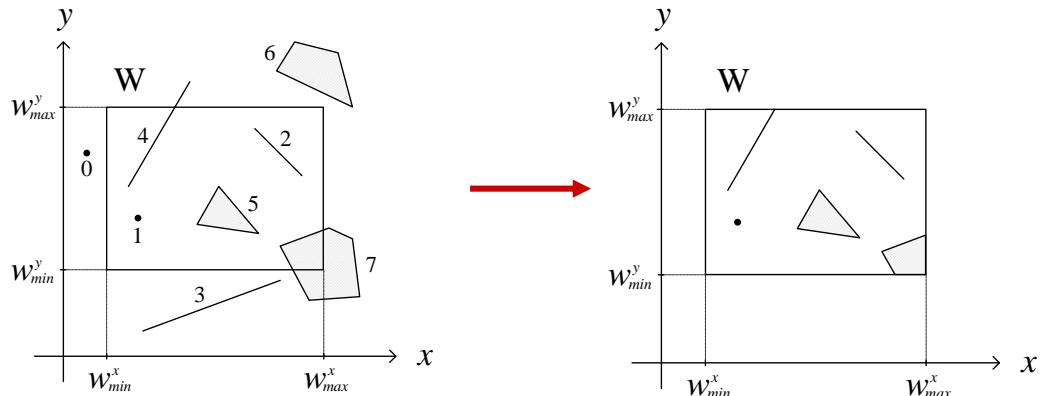
```
#define PI 3.14159265
void gluPerspective(GLdouble fovy, GLdouble asp,
                     GLdouble n, GLdouble f) {
    GLdouble ll, rr, bb, tt;

    tt = n*tan(fovy*PI/360.0);
    bb = -tt;
    ll = bb*asp;
    rr = tt*asp;

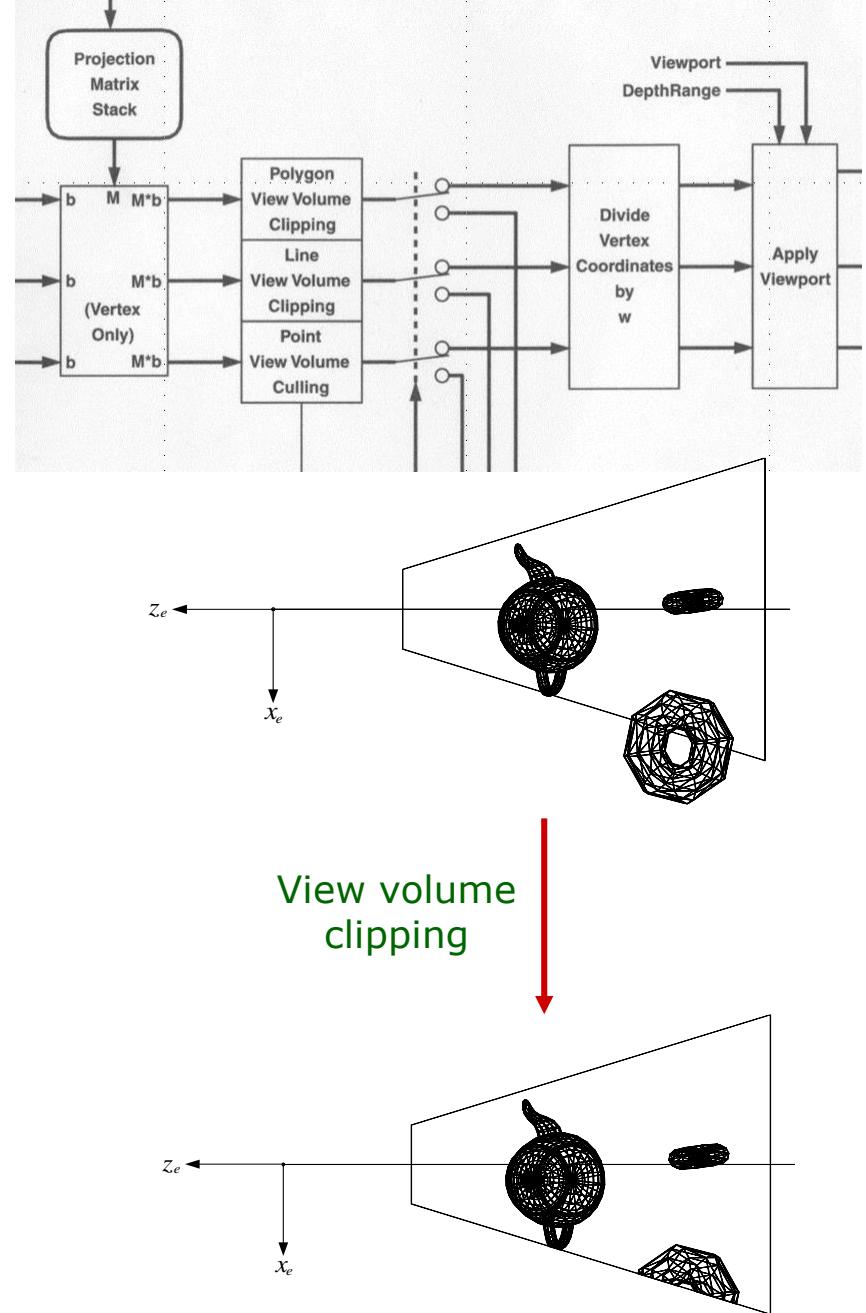
    glFrustum(ll, rr, bb, tt, n, f);
}
```

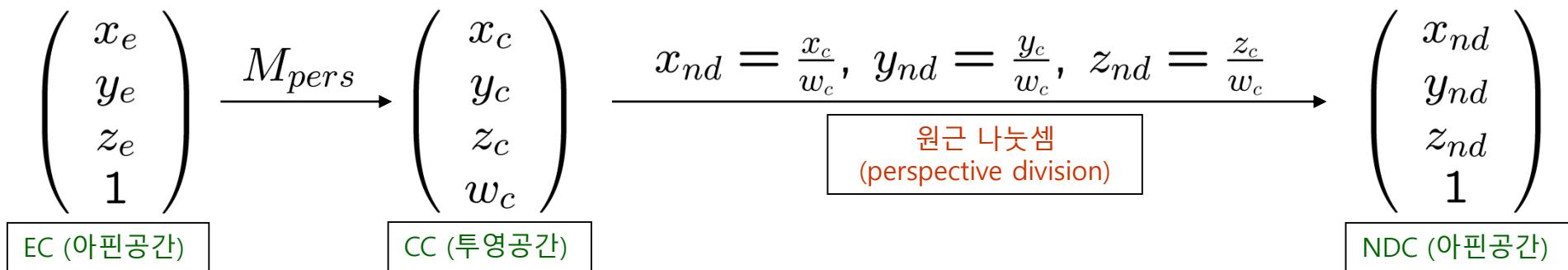
절단 좌표계(CC)에서의 절단

- 2차원 윈도우에 대한 절단



- 점의 절단 방법 → 간단
- 선분의 절단 방법 → 예: Cohen-Sutherland algorithm
- 다각형의 절단 방법 → 예: Sutherland-Hodgeman algorithm
- ❖ 교재 참조





어느 좌표계에서 view volume clipping을 하는 것이 효율적일까?

- OpenGL에서는
 - 동차 좌표 (x_c, y_c, z_c, w_c) 를 사용하는 절단 좌표계로 기하 변환된 각 프리미티브들에 대해, 사용자가 눈 좌표계에서 설정한 뷰 볼륨의 경계 면에 해당하는 여섯 개의 평면 조건

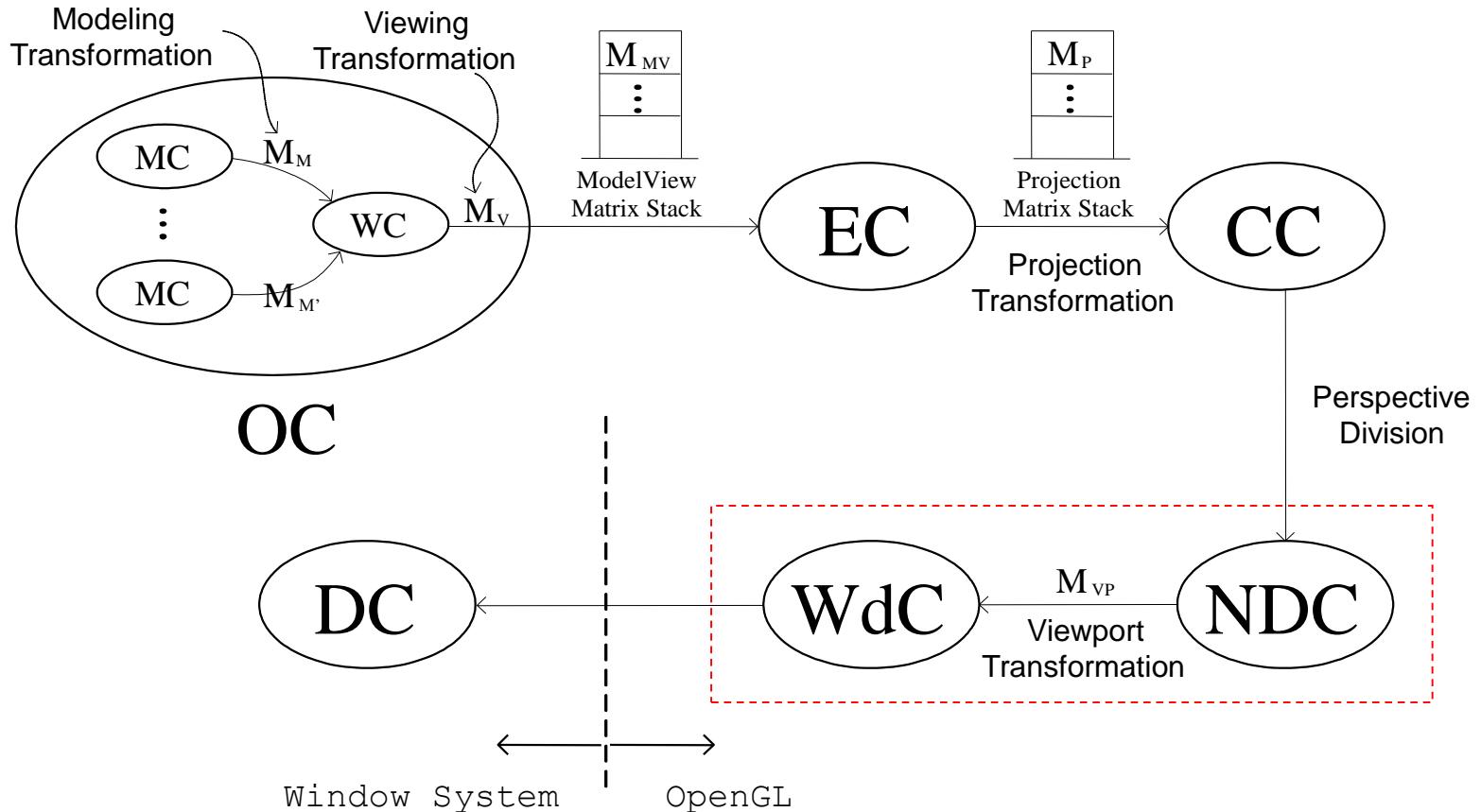
$$x_c \geq -w_c, x_c \leq w_c, y_c \geq -w_c, y_c \leq w_c, z_c \geq -w_c, z_c \leq w_c$$
 이 정의하는 뷰 볼륨 영역

$$-w_c \leq x_c \leq w_c, -w_c \leq y_c \leq w_c, -w_c \leq z_c \leq w_c$$
 에 대해, 절단 과정을 수행.
 - ☞ 2차원 공간에서의 절단 알고리즘은 자연스럽게 3차원 동차 좌표계 공간의 방법으로 확장이 됨.
 - ☞ 선분이나 다각형의 각 선분이 절단되어 일부분만 남을 경우 원래의 꼭지점에 대한 속성들이 새로운 꼭지점으로 적절히 선형 보간이 되어야 함.
 - ☞ 절단 후 살아 남은 꼭지점에 대해 원근 나눗셈 연산을 수행하여 정규 디바이스 좌표계의 $-1 \leq x_{nd} \leq 1, -1 \leq y_{nd} \leq 1, -1 \leq z_{nd} \leq 1$ 영역으로 매핑함.

From Normalized Device Coordinates to Window Coordinates: Viewport Transformation

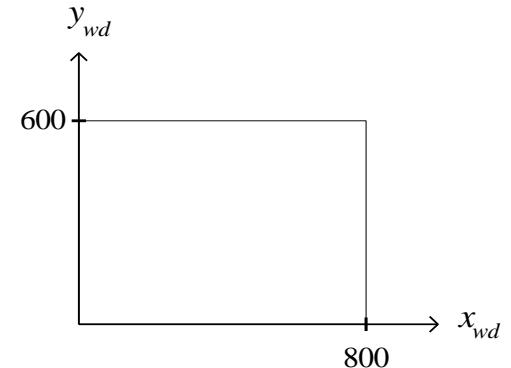
OpenGL의 기하 파이프라인과 사진 촬영과의 비교

- 모델링 변환(Modeling transformation)
 - 피사체 및 조명 배치
 - 모델링 좌표계(MC) → 세상 좌표계(WC)
- 뷰잉 변환(Viewing transformation)
 - 카메라의 위치와 방향 설정
 - 세상 좌표계(WC) → 눈 좌표계(EC)
- 투영 변환(Projection transformation)
 - 사용할 렌즈 결정 및 촬영 대상 결정
 - 카메라의 셔터를 누름
 - 눈 좌표계(EC) → 절단 좌표계(CC) → 정규 디바이스 좌표계(NDC)
- 뷰포트 변환(Viewport transformation)
 - 사진의 크기 결정 및 인화
 - 정규 디바이스 좌표계(NDC) → 윈도우 좌표계(WdC)



물체 좌표계와 윈도우 좌표계

- OpenGL의 기하 파이프라인은 물체 좌표계에서 출발하여 윈도우 좌표계까지의 변환에 해당함.
- **윈도우 좌표계(Window Coordinates, WdC)**
 - 사용자가 화면에 띄운 윈도우에 해당하는 좌표계.
 - 만들려고 하는 영상, 즉 사진을 인화하려는 인화지에 해당하는 좌표계임.
 - 실제 영상이 나타나는 뷰포트를 설정해주어야 함.



윈도우 좌표계와 뷰포트

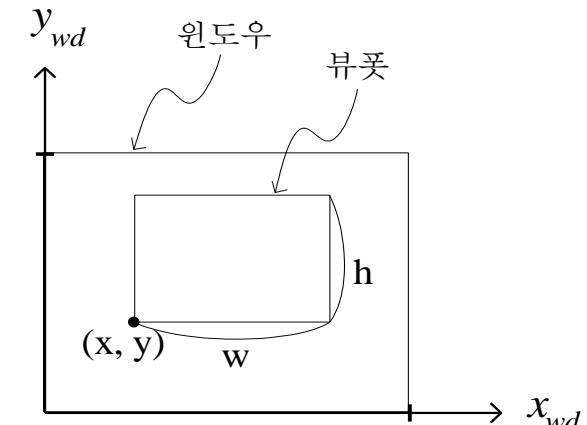
- NDC상의 '정규화된' 필름에 맺힌 상을 WdC상의 윈도우라는 인화지에 뷰포트(viewport)이라 하는 특정 부분을 설정하여 사진을 인화.

- **뷰포트의 설정**

```
void glViewport(GLint x, GLint y, GLsizei w, GLsizei h);
```

- NDC의 정규 윈도우 $[-1, 1] \times [-1, 1]$ 영역이
WdC의 뷰포트 $[x, x+w] \times [y, y+h]$ 영역으로 매핑이 됨.

$$(x_{nd}, y_{nd}) \longrightarrow (x_{wd}, y_{wd})$$



- 깊이값 범위(depth range)의 설정

```
void glDepthRange(GLclampd n, GLclampd f);
```

- NDC의 $[-1, 1]$ 구간이 WdC의 $[n, f]$ 구간으로 매핑이 됨.

$$z_{nd} \longrightarrow z_{wd}$$

뷰포트 변환(Viewport Transformation)

- 정규 디바이스 좌표계에서 윈도우 좌표계로의 변환
- 유도

$$o_x = x + \frac{w}{2}, o_y = y + \frac{h}{2} \text{ 라 하면}$$

$$\begin{aligned}x_{wd} &= \frac{w}{2} \cdot x_{nd} + o_x \\y_{wd} &= \frac{h}{2} \cdot y_{nd} + o_y \\z_{wd} &= \frac{f-n}{2} \cdot z_{nd} + \frac{f+n}{2}\end{aligned}$$



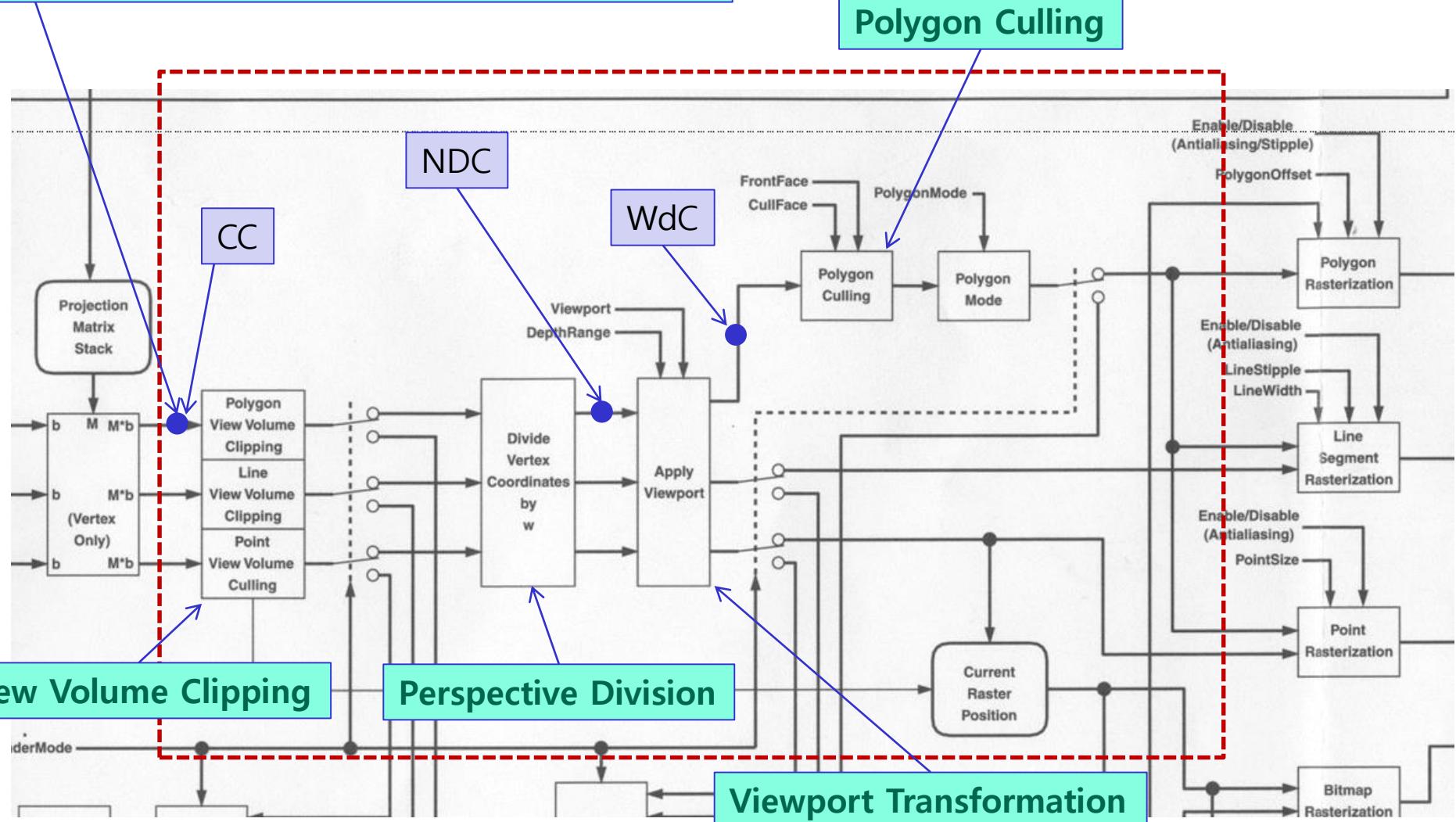
$$M_{VP} = \begin{bmatrix} \frac{w}{2} & 0 & 0 & o_x \\ 0 & \frac{h}{2} & 0 & o_y \\ 0 & 0 & \frac{f-n}{2} & \frac{f+n}{2} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

☺ 2차원 화면에 해당하는 윈도우 좌표계까지 변환이 되어도, 아직 z 좌표 정보, 즉 깊이 정보 값을 유지함 → 추후 은면 제거(hidden surface removal)를 위한 깊이 버퍼링(depth buffering) 등의 연산에 사용됨.

Geometry Computation after Vertex Shader

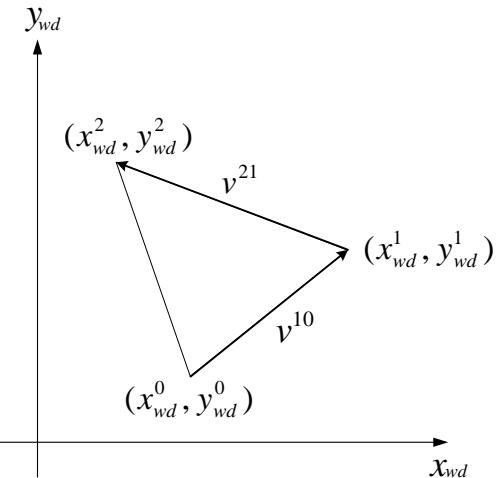
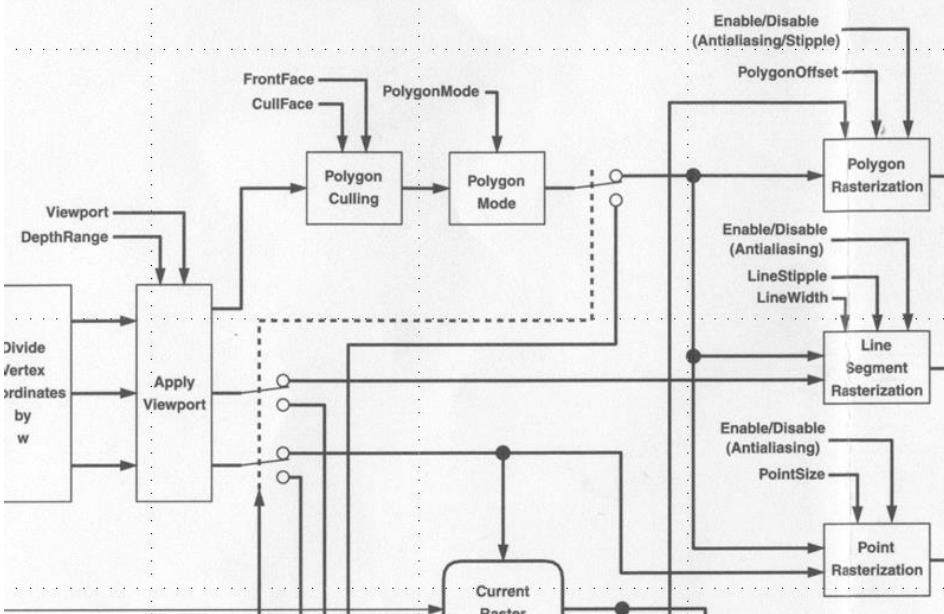
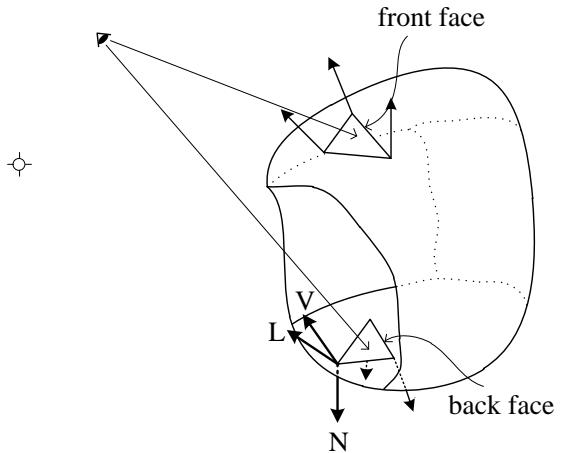
이 시점에서 Primitive Assembly 연산 수행 후
(필요에 따라) Geometry Shader가 수행됨

이 부분은 fixed-function 형태로 남아 있음



Polygon Culling in WdC

```
glFrontFace(GL_CCW);  
glCullFace(GL_BACK);  
 glEnable(GL_CULL_FACE);
```



$$a = \frac{1}{2} \sum_{i=0}^{n-1} (x_{wd}^i y_{wd}^{i+1} - x_{wd}^{i+1} y_{wd}^i)$$

어느 좌표계에서 polygon culling을 하는 것이 효율적일까?

Summary of OpenGL 3D Viewing

OpenGL의 기하 파이프라인과 사진 촬영과의 비교

- **모델링 변환(Modeling transformation)**

- 피사체 및 조명 배치
- 모델링 좌표계(MC) → 세상 좌표계(WC)

세상 좌표계 (World Coordinates, WC)

-- 가상의 세상이 존재하는 좌표계

- **뷰잉 변환(Viewing transformation)**

- 카메라의 위치와 방향 설정
- 세상 좌표계(WC) → 눈 좌표계(EC)

- **투영 변환(Projection transformation)**

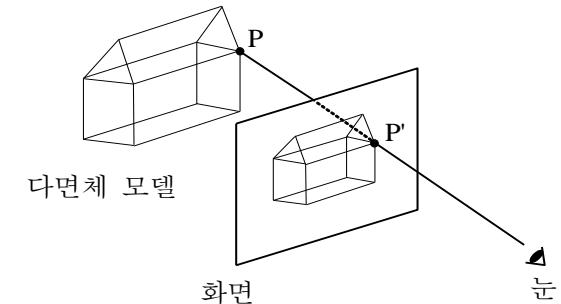
- 사용할 렌즈 결정 및 촬영 대상 결정
- 카메라의 셔터를 누름
- 눈 좌표계(EC) → 절단 좌표계(CC) → 정규 디바이스 좌표계(NDC)

- **뷰포트 변환(Viewport transformation)**

- 사진의 크기 결정 및 인화
- 정규 디바이스 좌표계(NDC) → 윈도우 좌표계(WdC)

OpenGL의 좌표계 및 기하 변환의 의미

- 그래픽스 프로그래밍에 있어 **3D viewing**의 목적
 - 렌더링 하고자 하는 물체의 각 꼭지점 (x_o, y_o, z_o)을 프로그래머가 설정한 윈도우의 어느 지점 (x_{wd}, y_{wd})에 떨어지게 할 것인가를 설정.
- OpenGL의 각 좌표계 공간에서의 좌표의 의미
 - 물체 좌표계(OC)
 - 모델링 좌표계(MC) → 세상 좌표계(WC)
 - 눈 좌표계(EC)
 - 절단 좌표계(CC)
 - 정규 디바이스 좌표계(NDC)
 - 윈도우 좌표계(WdC)



◎ 기하 파이프라인을 따라 꼭지점이 흘러가면서 꼭지점의 의미가 물체 좌표계에서의 의미에서 점진적으로 윈도우 좌표계에서의 의미로 전환됨.

$$M_{MV} = M_V * M_M \rightarrow M_P \rightarrow M_{VP}$$

◎ 프로그래머는 OpenGL 함수 호출을 통하여 변환이 올바르게 수행 될 수 있도록 각 변환 행렬 설정하는 것이 기하 파이프라인 관련 프로그래밍의 목적임.

OpenGL 기하 변환 관련 프로그래밍의 목적 (Compatibility Profile)

- ① 적절한 모델링 변환과 뷰잉 변환을 나타내는 모델뷰 행렬 M_{MV} 를 모델뷰 행렬 스택의 탑에 올려놓고,
 - ② 원하는 투영 변환에 해당하는 투영 행렬 M_p 를 투영 행렬 스택의 탑에 올려놓은 다음,
 - ③ 마지막으로 뷰포트 변환에 해당하는 M_{VP} 행렬을 적절하게 설정하는 것임.
-
- glVertex*(*) 함수를 통하여 물체 좌표계상에서의 꼭지점의 좌표를 기술하면 꼭지점 좌표가 $M_{MV} \rightarrow M_p \rightarrow M_{VP}$ 순서대로 각 행렬에 곱해져 물체 좌표계로부터 윈도우 좌표계까지의 좌표 변환 계산이 수행이 됨.
 - gluLookAt(*) 함수를 수행하면 바로 M_V 행렬이 계산이 되어 현재 행렬 스택의 탑에 있는 행렬의 오른쪽에 곱해짐.
 - 만약 모델링 변환을 사용할 경우($M_{MV} = M_V * M_M$) 스택의 성격 상 뷰잉 변환이 먼저 설정이 되어야 함(직관적인 순서와는 반대).

3차원 뷰잉 관련 코드 예시

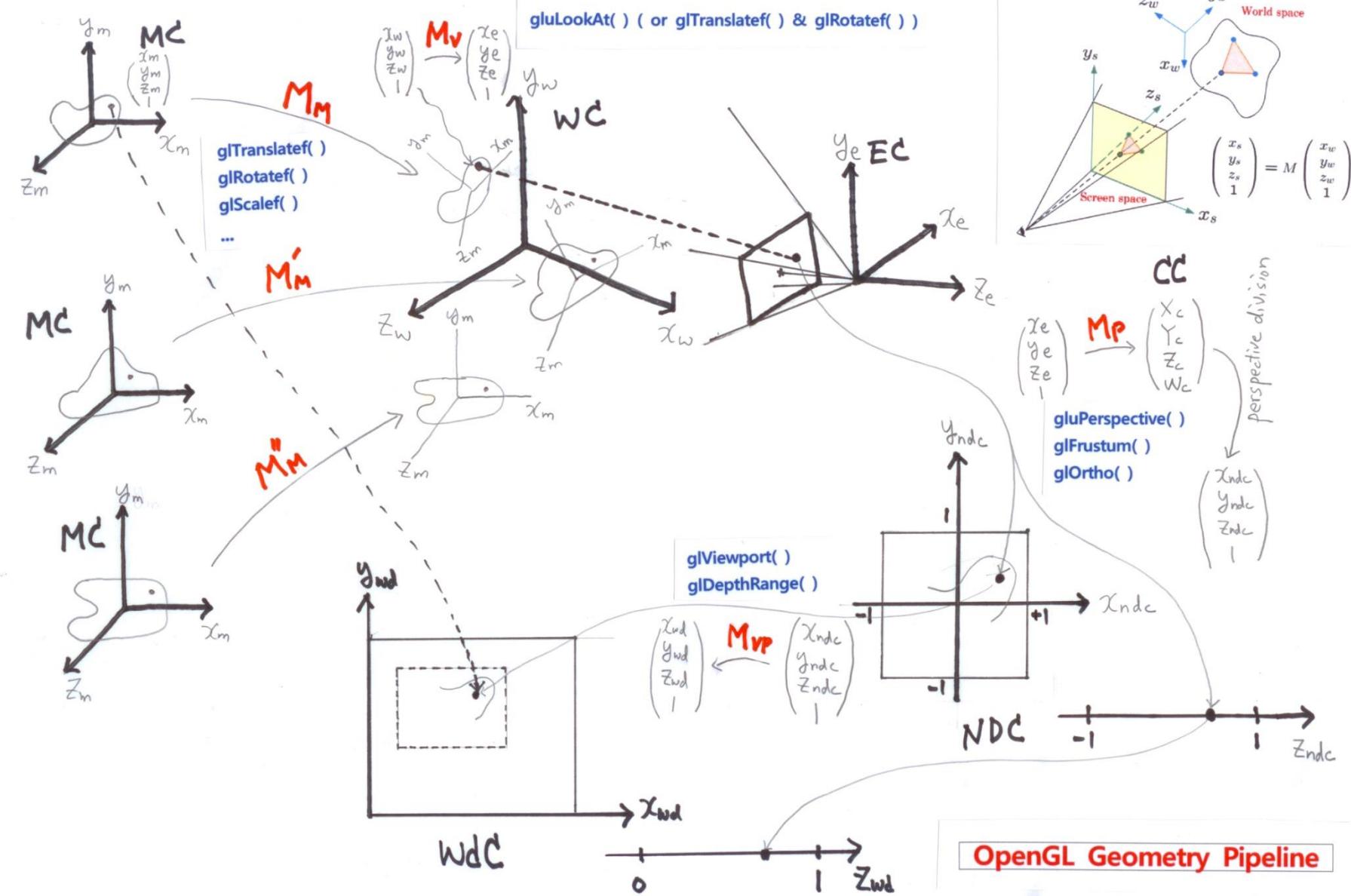
```
glViewport(0, 0, 400, 300); // Viewport  
transformation  
  
glMatrixMode(GL_PROJECTION); // Projection  
transformation  
glLoadIdentity();  
glOrtho(-5.0, 5.0, -5.0, 5.0, 0.0, 1000.0);  
  
glMatrixMode(GL_MODELVIEW); // Viewing  
transformation  
glLoadIdentity();  
gluLookAt(10.0, 10.0, 10.0, 0.0, 0.0, 0.0, 0.0, 1.0,  
0.0);  
  
glPushMatrix();  
glRotatef(angle, 0.0, 1.0, 0.0); // Modeling  
trans. 1  
...  
...
```

```
draw_object1();  
glPushMatrix();  
glScalef(2.0, 2.0, 2.0); // Modeling  
trans. 2  
...  
draw_object2();  
glRotatef(40.0*angle, 1.0, 0.0, 0.0);  
// Modeling trans. 3  
...  
draw_object3();  
glPopMatrix();  
glTranslatef(0.15, 0.3, 0.0);  
// Modeling trans. 4  
...  
draw_object4();  
glPopMatrix();
```

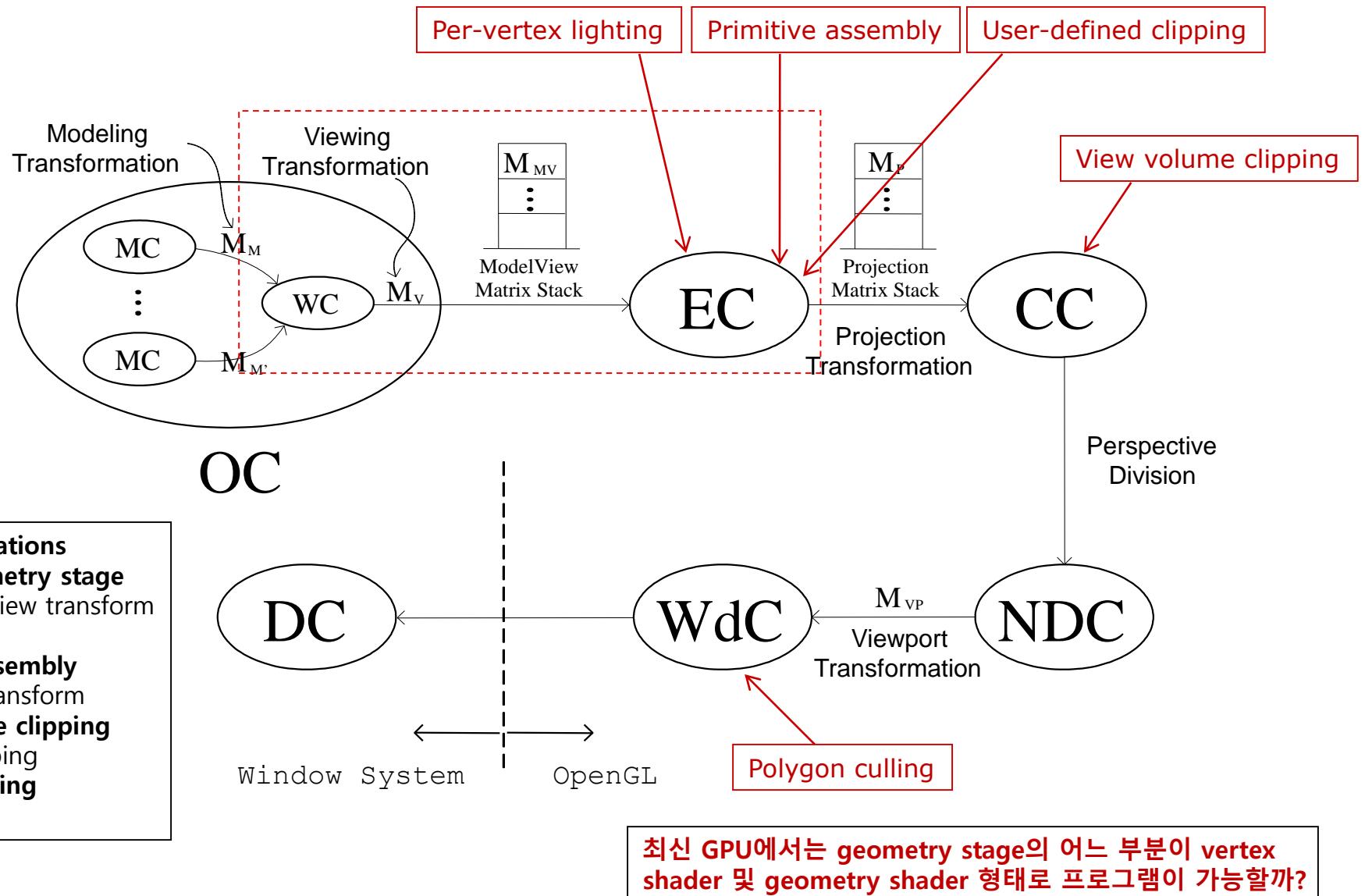
OpenGL 기하 변환 관련 프로그래밍의 목적(Core Profile)

- ① 원하는 뷰풋 변환을 설정한 후 (M_{VP} 설정),
 - ② 원하는 투영 변환에 해당하는 투영 행렬 M_P 를 계산한 다음,
 - ③ 원하는 뷰잉 변환에 해당하는 뷰잉 행렬 M_V 를 계산한 다음,
 - ④ 그리려고 하는 각 물체에 대한 모델링 변환에 해당하는 모델링 행렬 M_M 을 계산한 다음,
 - ⑤ 이 세 행렬을 모두 곱한 $M_{MVP} = M_P M_V M_M$ 를 계산하여 vertex shader에게 uniform variable 형태로 보내줌.
-
- `glDrawArray(*)` 함수를 통하여 꼭지점 연결 정보와 함께 vertex stream을 그래픽스 파이프라인에 넣어주면, 먼저 각 꼭지점에 대해 꼭지점의 좌표가 $M_M \rightarrow M_V \rightarrow M_P$ 순서대로 각 행렬에 곱해져 OC(MC/WC)로부터 CC까지의 좌표 변환 계산이 수행이 됨.
 - 이후, primitive assembly 연산이 일어난 후 (geometry shader를 거쳐) perspective division을 통해 NDC로 넘어간 꼭지점에 대해 위에서 설정한 뷰풋 변환 행렬 M_{VP} 곱해져 최종 목적지인 WdC로 가게 됨.

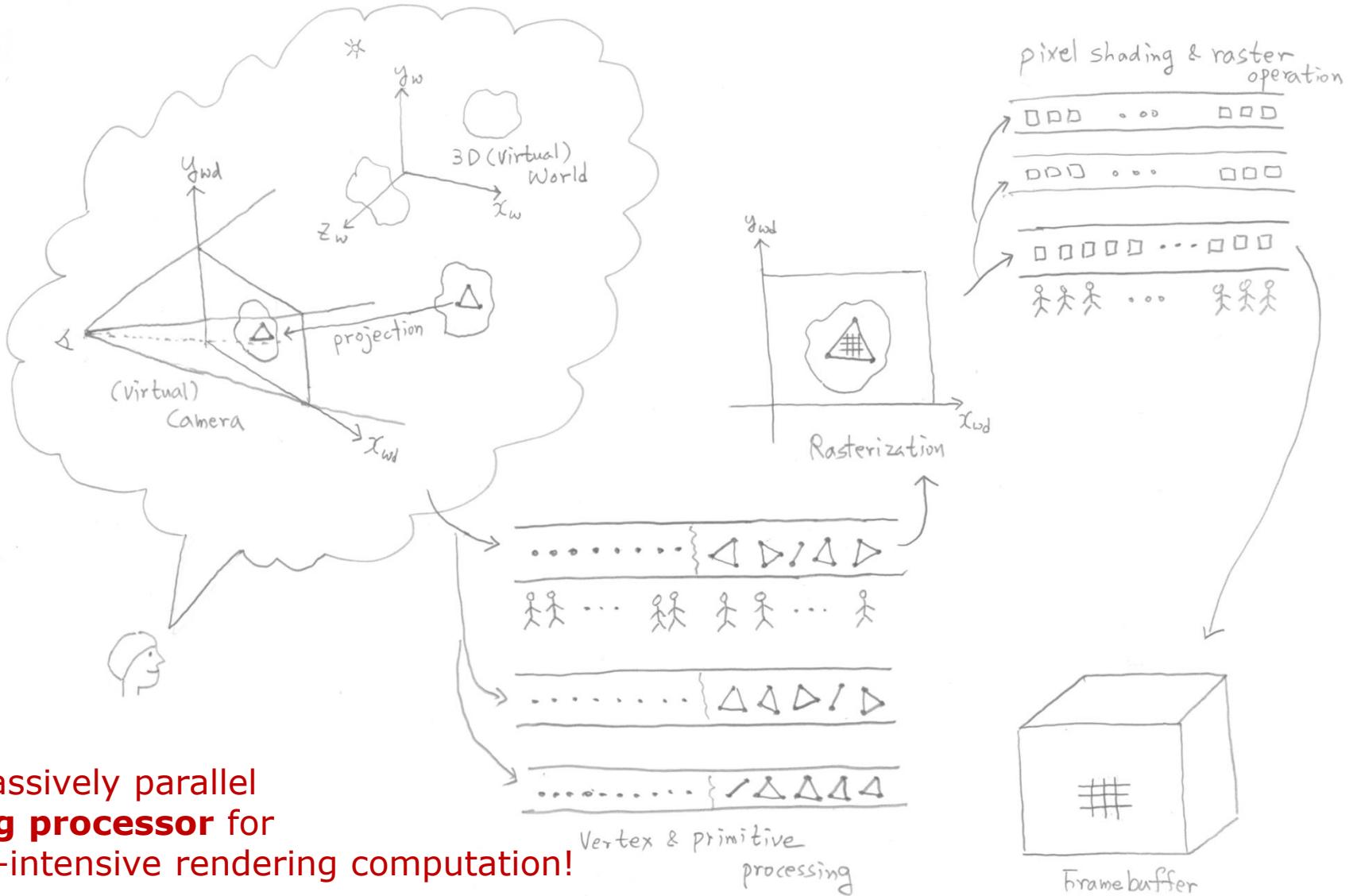
Summary: OpenGL Geometry Pipeline II



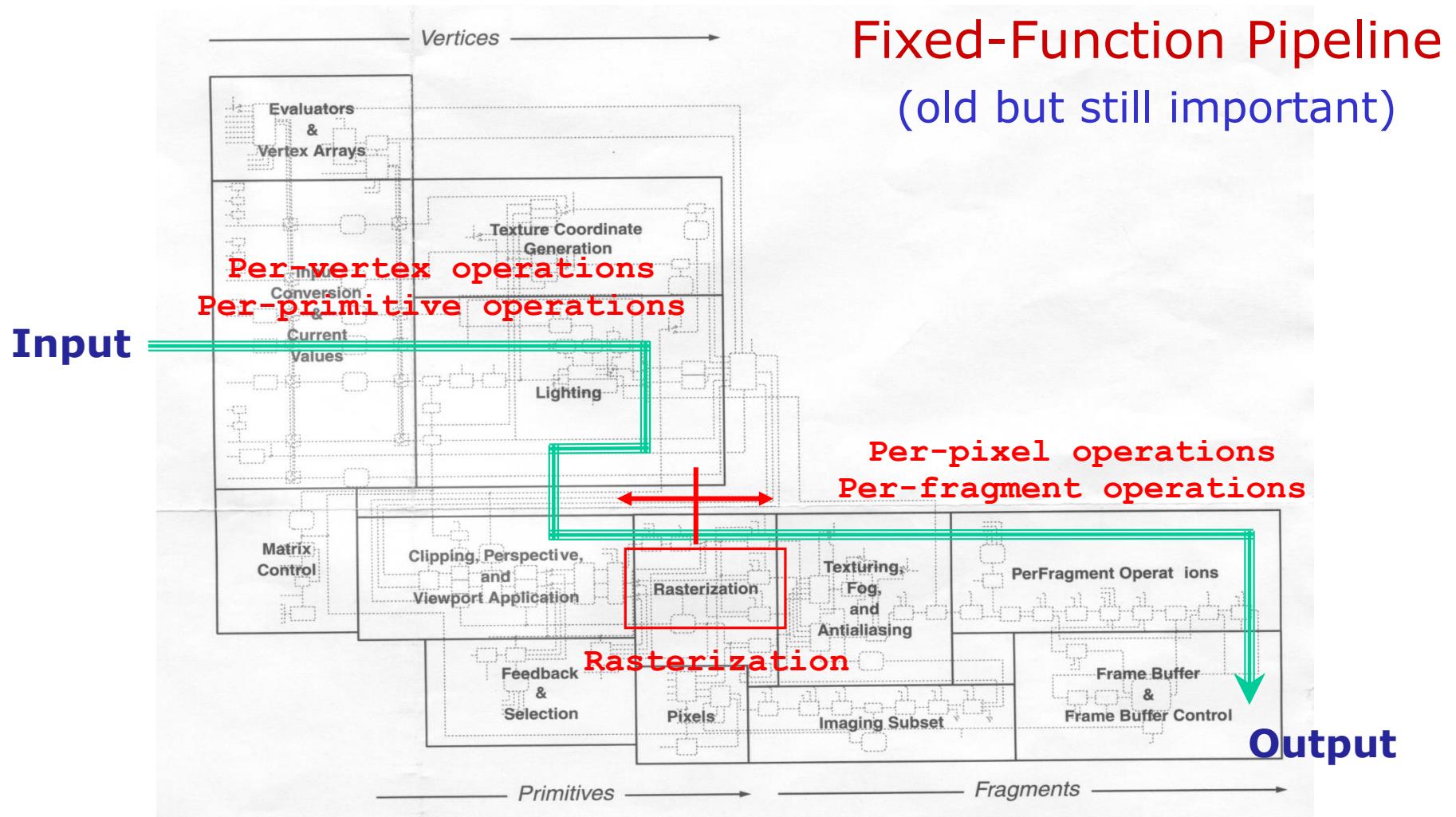
Geometry Stage in Fixed-Function OpenGL Pipeline



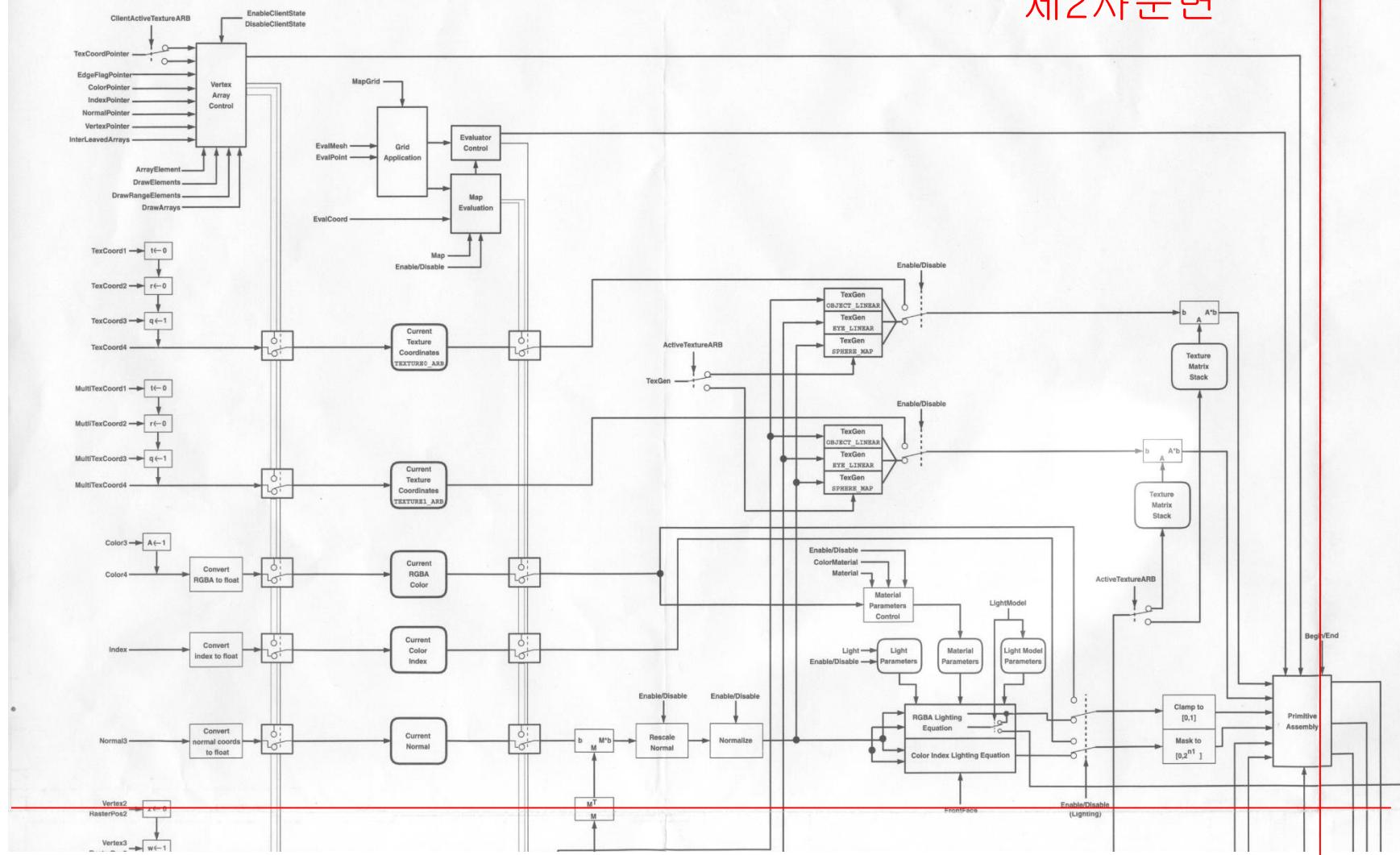
Conventional Real-Time Rendering Pipeline on GPU

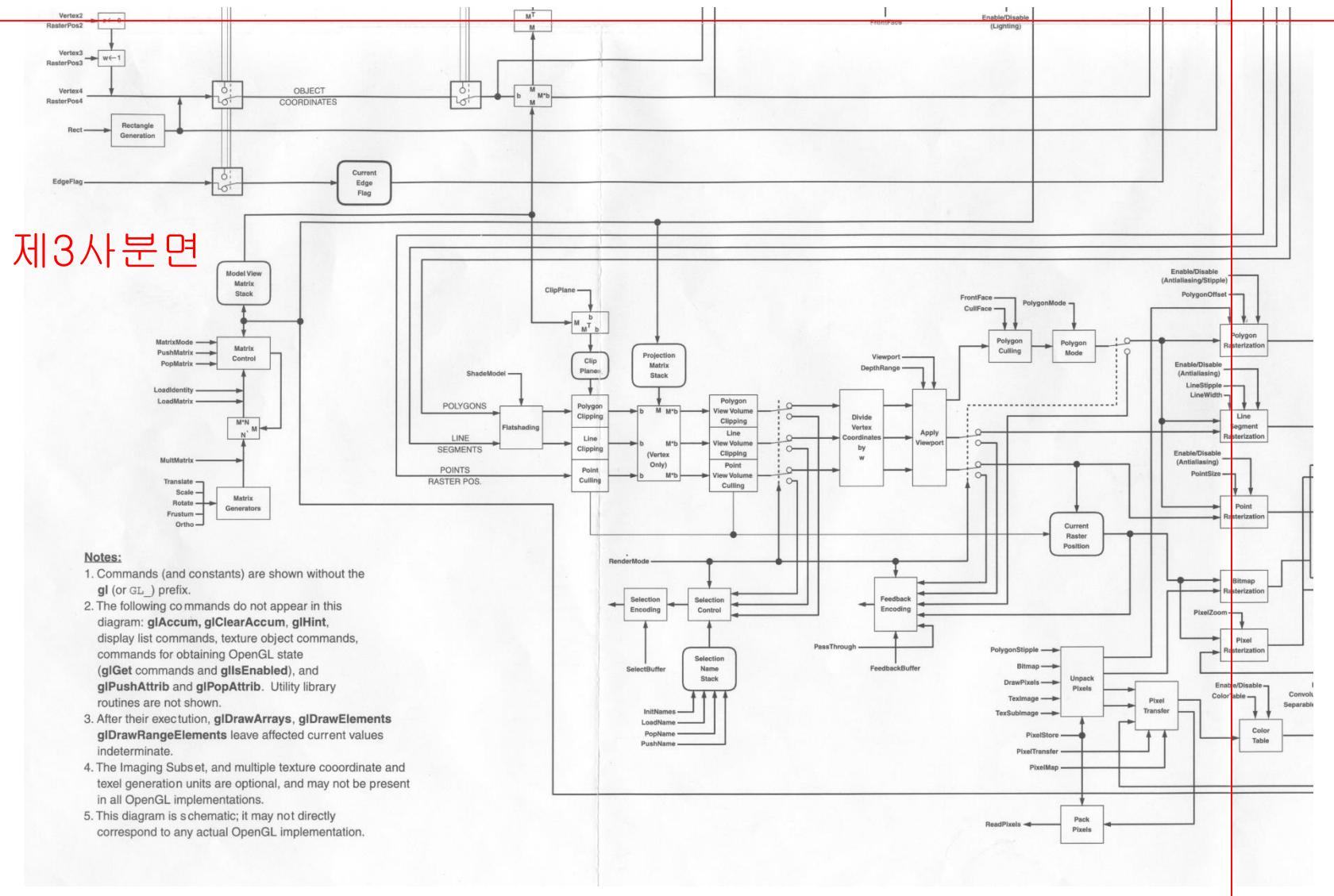


OpenGL Architecture (Version 1.2)

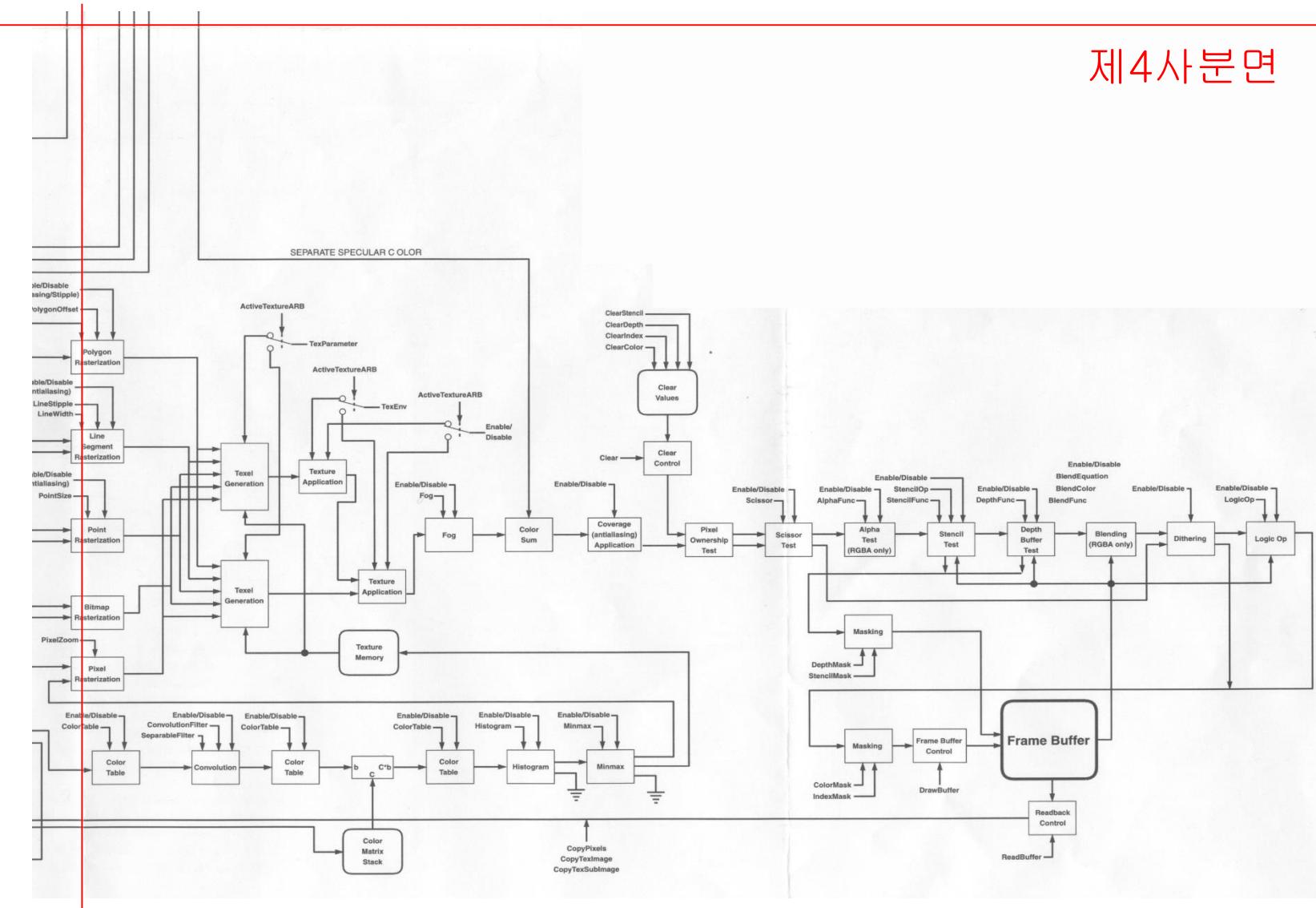


제2사분면

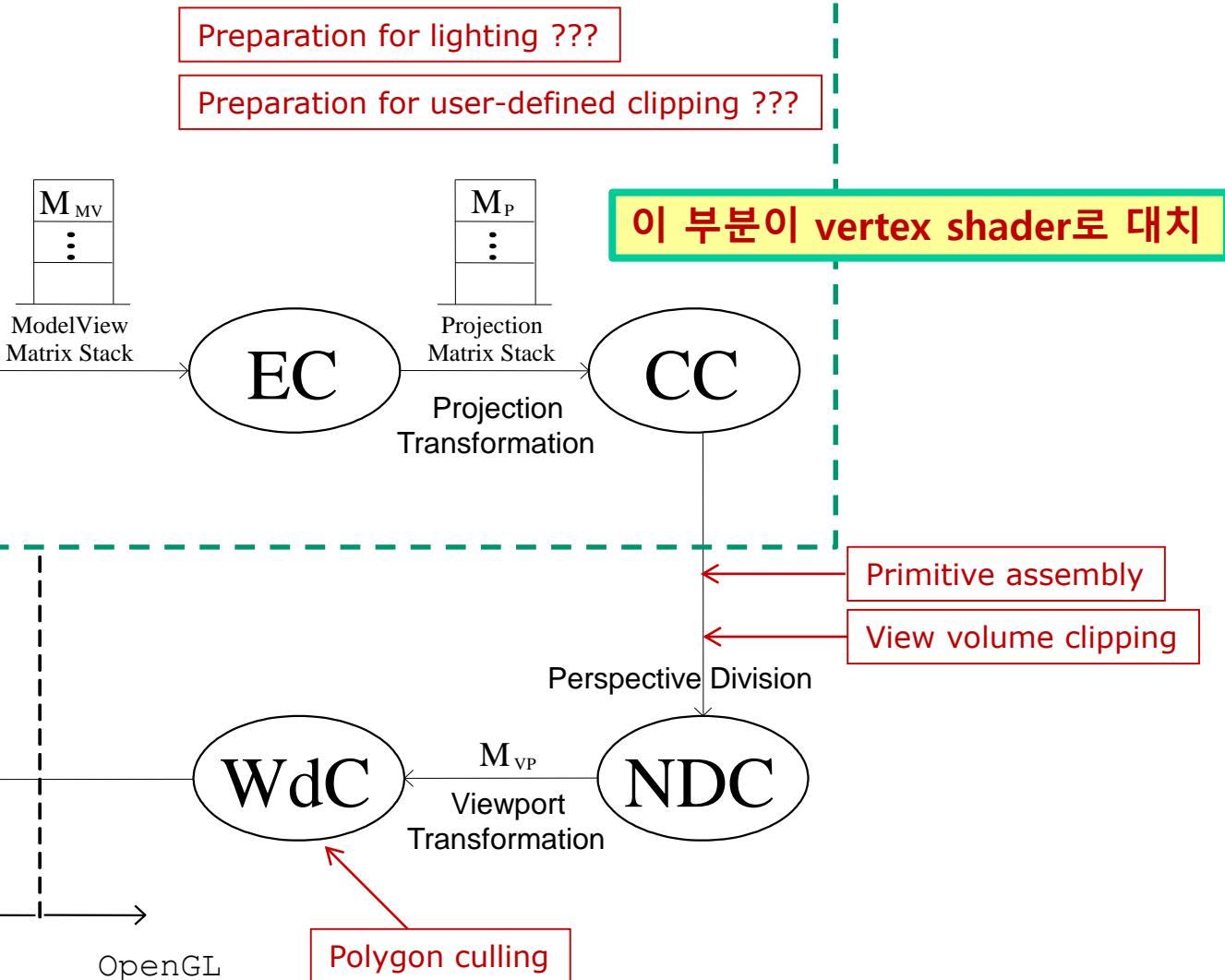
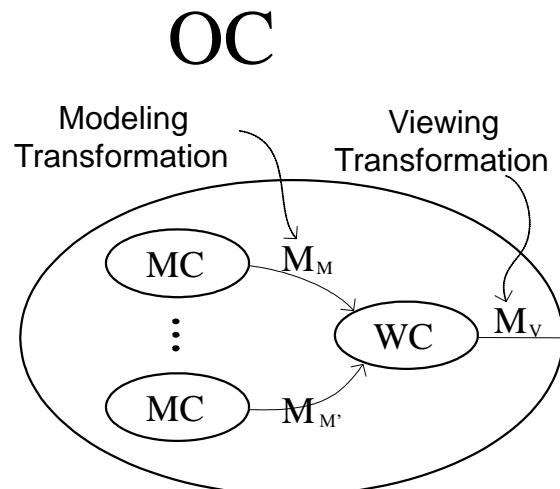




제4사분면



Vertex Shader in Programmable OpenGL Pipeline



최신 GPU에서는 geometry stage의 어느 부분이 vertex shader 및 geometry shader 형태로 프로그램이 가능할까?

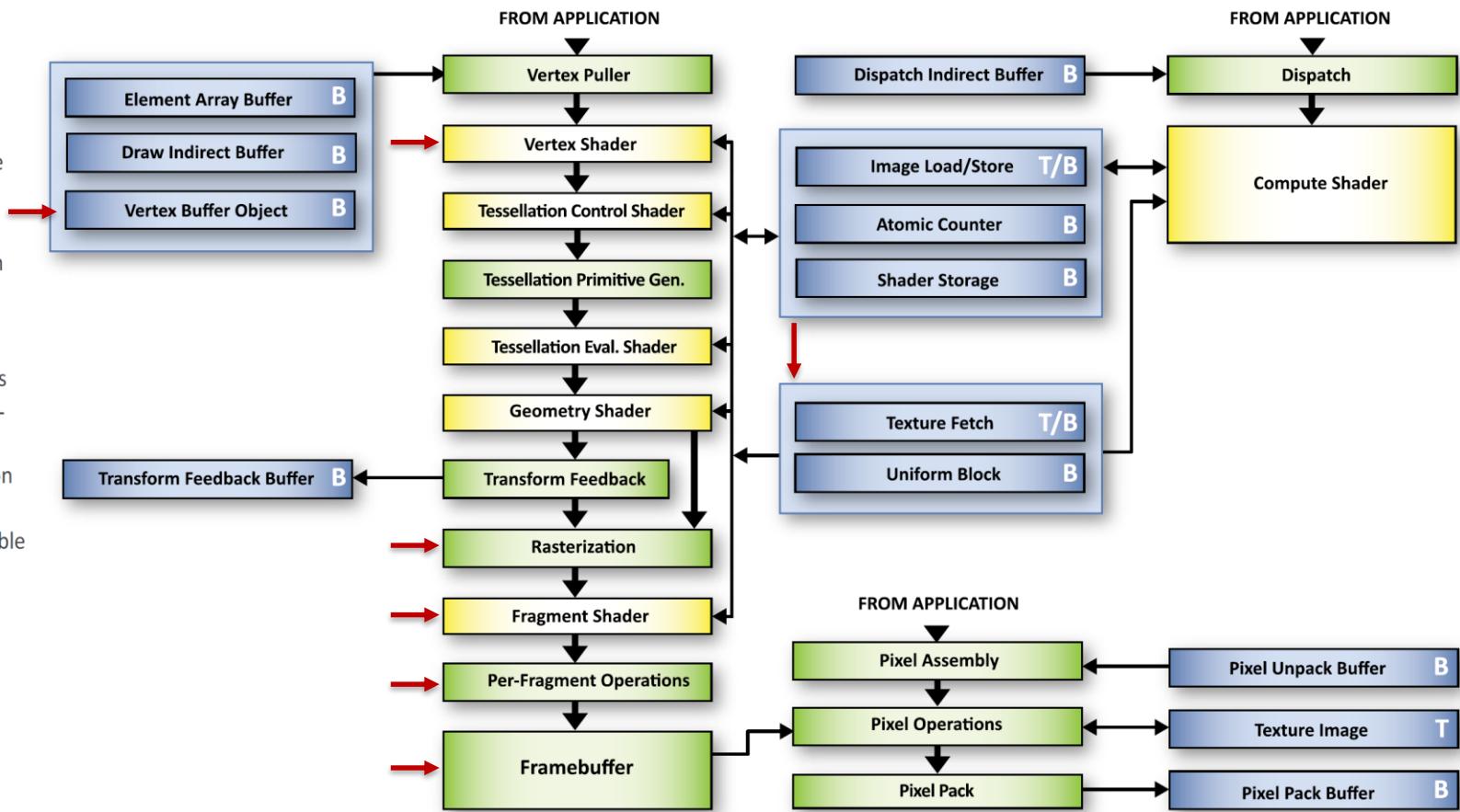
The Newest OpenGL (from OpenGL 4.5 Reference Card)

OpenGL Pipeline

A typical program that uses OpenGL begins with calls to open a window into the framebuffer into which the program will draw. Calls are made to allocate a GL context which is then associated with the window, then OpenGL commands can be issued.

The heavy black arrows in this illustration show the OpenGL pipeline and indicate data flow.

- Blue blocks indicate various buffers that feed or get fed by the OpenGL pipeline.
- Green blocks indicate fixed function stages.
- Yellow blocks indicate programmable stages.
- T Texture binding
- B Buffer binding



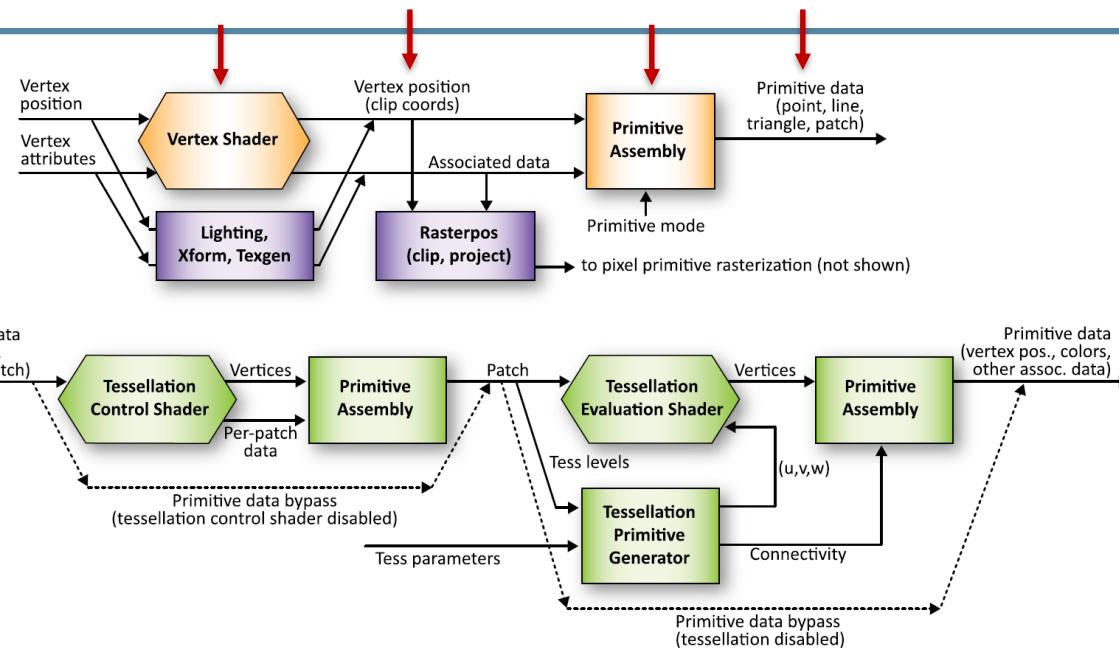
수업 시간에 배운 내용이 OpenGL 4.5 rendering pipeline의 어느 부분(→)에 해당하는지 살펴보자!

Vertex & Tessellation Details

Each vertex is processed either by a vertex shader or fixed-function vertex processing (compatibility only) to generate a transformed vertex, then assembled into primitives. Tessellation (if enabled) operates on patch primitives, consisting of a fixed-size collection of vertices, each with per-vertex attributes and associated per-patch attributes. Tessellation control shaders (if enabled) transform an input patch and compute per-vertex and per-patch attributes for a new output patch.

A fixed-function primitive generator subdivides the patch according to tessellation levels computed in the tessellation control shaders or specified as fixed values in the API (TCS disabled). The tessellation evaluation shader computes the position and attributes of each vertex produced by the tessellator.

- Orange blocks indicate features of the Core specification.
- Purple blocks indicate features of the Compatibility specification.
- Green blocks indicate features new or significantly changed with OpenGL 4.x.



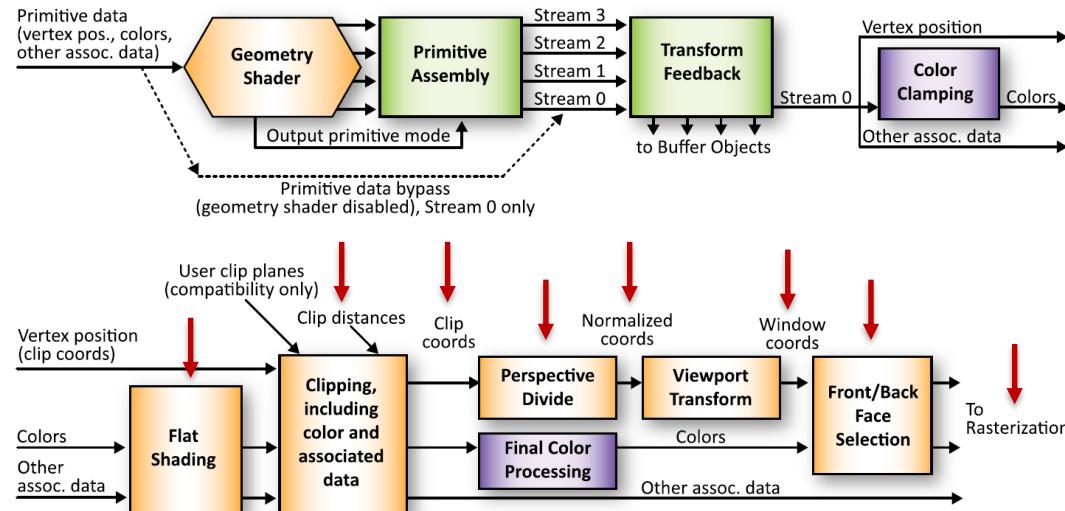
Geometry & Follow-on Details

Geometry shaders (if enabled) consume individual primitives built in previous primitive assembly stages. For each input primitive, the geometry shader can output zero or more vertices, with each vertex directed at a specific vertex stream. The vertices emitted to each stream are assembled into primitives according to the geometry shader's output primitive type.

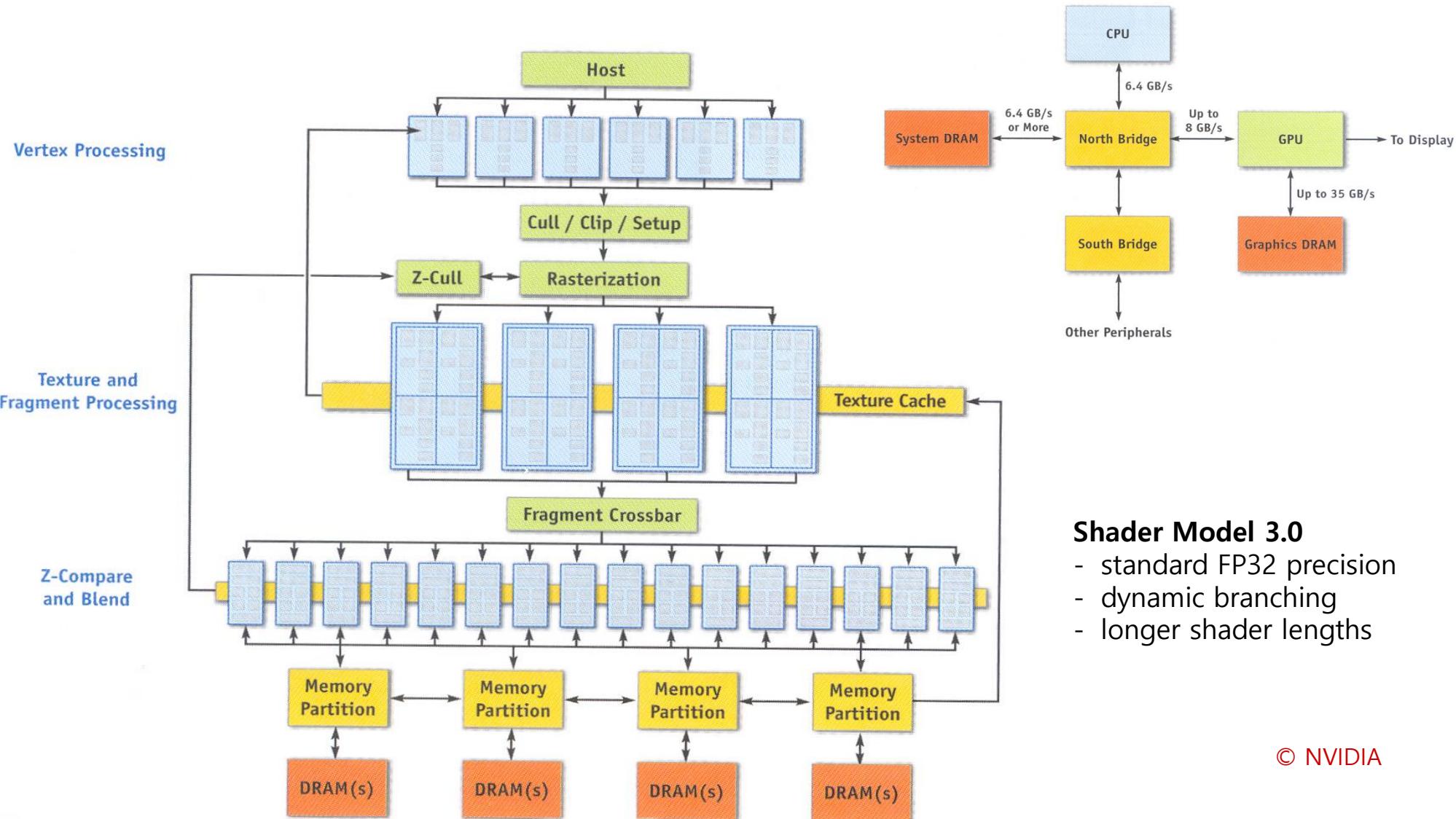
Transform feedback (if active) writes selected vertex attributes of the primitives of all vertex streams into buffer objects attached to one or more binding points.

Primitives on vertex stream zero are then processed by fixed-function stages, where they are clipped and prepared for rasterization.

- Orange blocks indicate features of the Core specification.
- Purple blocks indicate features of the Compatibility specification.
- Green blocks indicate features new or significantly changed with OpenGL 4.x.



2004년: GeForce 6 Series (NV40) - 6800 Ultra



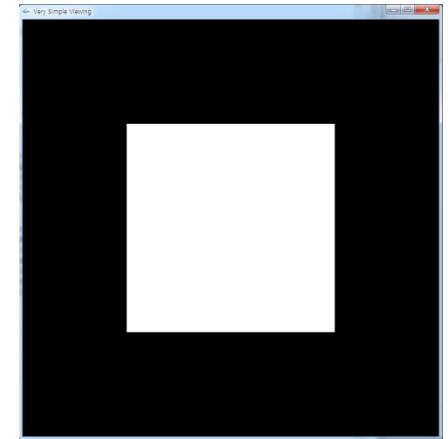
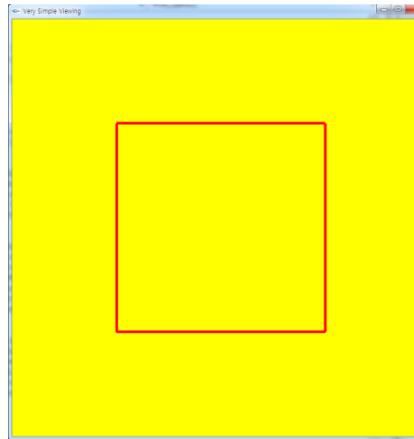
간단한 3D 뷰잉을 사용하는 프로그램 예

```
#include <GL/glut.h>
void display (void) {
    glClear(GL_COLOR_BUFFER_BIT);
    glBegin(GL_POLYGON);
    glVertex3f(0.5, 0.5, 0.0);
    glVertex3f(-0.5, 0.5, 0.0);
    glVertex3f(-0.5, -0.5, 0.0);
    glVertex3f(0.5, -0.5, 0.0);
    glEnd();
    glFlush();
}

void keyboard (unsigned char key, int x, int y) {
    switch (key) { case 'q': exit(0); break; }
}

void reshape(int width, int height) {
    glViewport(0.0, 0.0, width, height);
}

void init_OpenGL(void) {
    glPolygonMode(GL_FRONT_AND_BACK, GL_LINE);
    glClearColor(1.0, 1.0, 0.0, 1.0);
    glColor3f(1.0, 0.0, 0.0); glLineWidth(5.0);
}
```

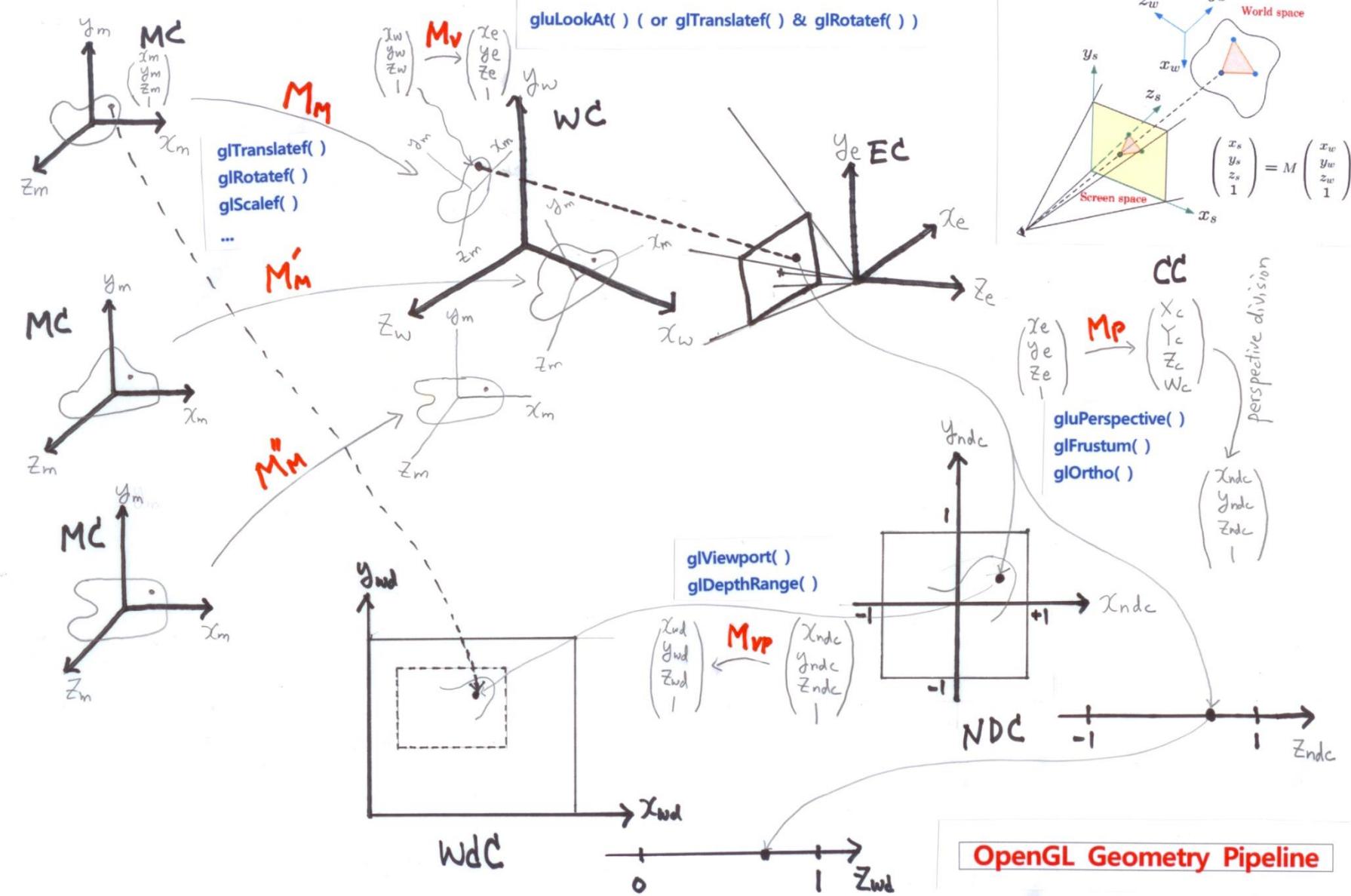


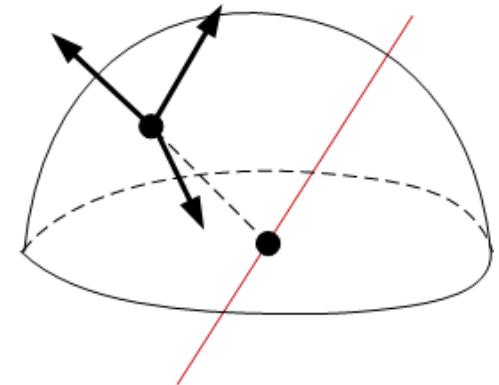
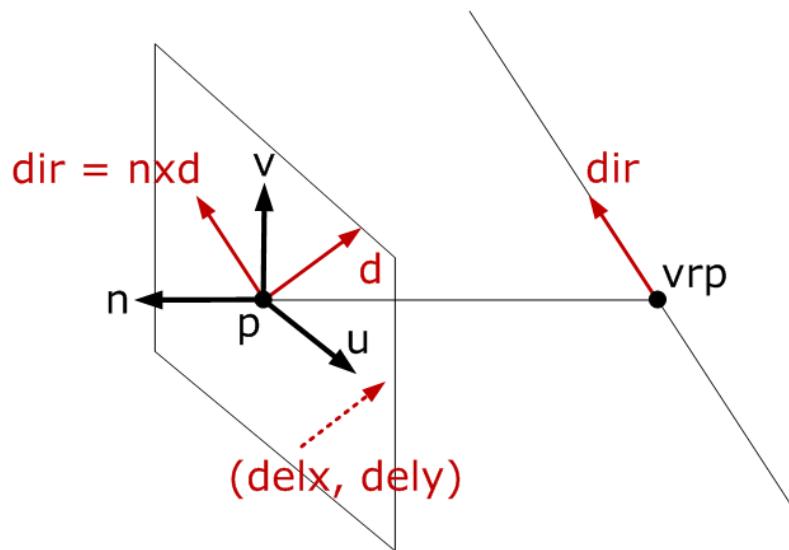
```
void init_windows(void) {
    glutInitDisplayMode(GLUT_RGB);
    glutInitWindowSize(800, 800);
    glutCreateWindow("Very Simple Viewing");
    glutDisplayFunc(display);
    glutKeyboardFunc(keyboard);
    glutReshapeFunc(reshape);
}
```

```
void main(int argc, char **argv) {
    glutInit(&argc, argv);
    init_windows(); // Line 1
    init_OpenGL(); // Line 2
    glutMainLoop();
}
```

만약 Line 1과 Line 2의 순서를 바꾼다면?

Summary: OpenGL Geometry Pipeline II





$$leng = \sqrt{delx^2 + dely^2}$$

$$\mathbf{p}^* = R_{3 \times 3}(\mathbf{p} - \mathbf{vrp}) + \mathbf{vrp}$$

$$\mathbf{d} = \frac{delx}{leng}\mathbf{u} + \frac{dely}{leng}\mathbf{v}$$

$$\mathbf{dir} = \mathbf{n} \times \mathbf{d}$$

$$R_{3 \times 3} = R(a \cdot leng, \mathbf{dir})_{3 \times 3}$$

$$\mathbf{u}^* = R_{3 \times 3}\mathbf{u}$$

$$\mathbf{v}^* = R_{3 \times 3}\mathbf{v}$$

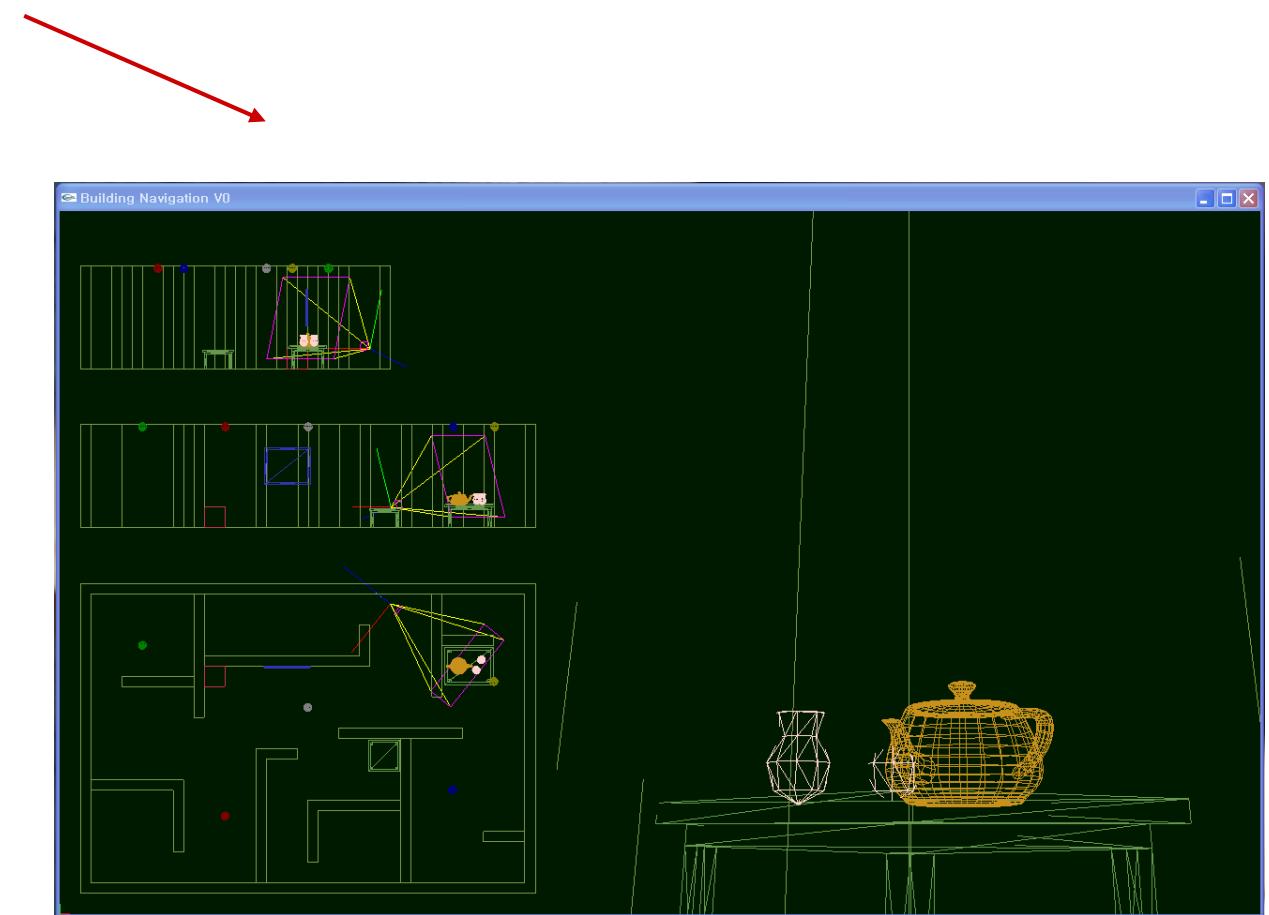
$$\mathbf{n}^* = R_{3 \times 3}\mathbf{n}$$

EXERCISE: 간단한 뷰잉 과정 연습

- 목적:
 - OpenGL 프로그래밍을 통하여 3차원 뷰잉 과정에 대한 이해를 높임.
 - 모델링 좌표계 상에서 설계되어 있는 다양한 기본 물체를 세상에 배치해봄.
- 구현 내용 (본 프로그램의 샘플 executable는 수업 홈페이지에 있음)
 - 본 수업에서 제공하는 기본 물체 (새로운 물체 추가할 경우 추가 점수)들을 다양하게 세상에 배치하여 건물 내부를 설계한 후, 그 내용을 이름이 `scene.txt`인 파일에 저장.
 - `scene.txt` 파일에서 전체 장면에 대한 정보를 읽어 들여 건물 내부에 대하여
 1. top view,
 2. front view,
 3. side view, 그리고
 4. camera view를화면에 도시. 이때 앞의 세 view에 대해서는 현재 카메라에 대한 정보를 도시해야 하며, 프로그램이 처음 수행될 때에는 카메라 인자는 본인이 설정한 디폴트 값을 사용

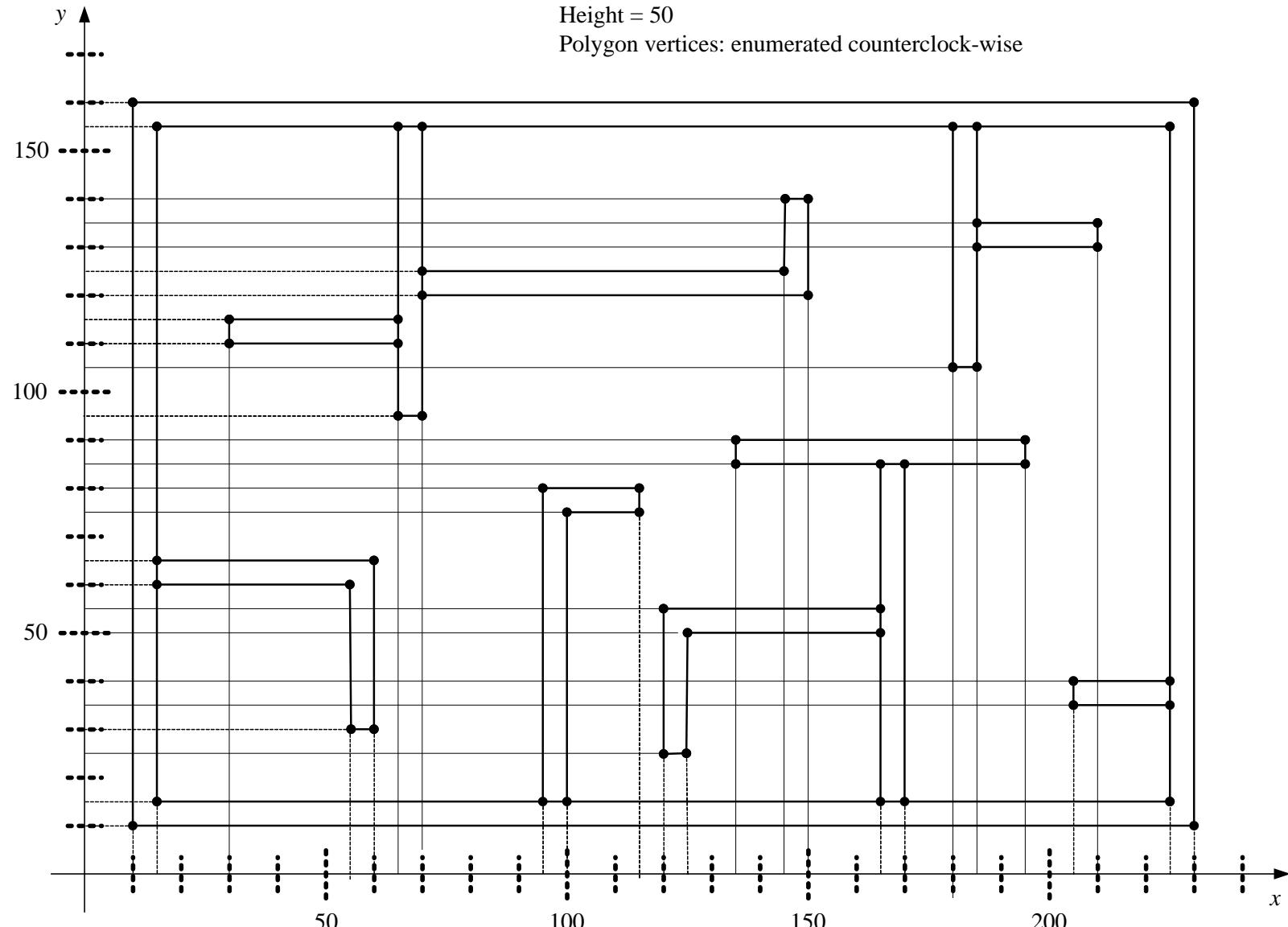
- 'C' 키를 누른 후 다음과 같이 카메라에 대한 정보를 입력하면, 새로운 카메라 인자를 사용하여 화면의 내용을 다시 그려줌.

```
[Projection Reference Point]:  
160.0 150.0 10.0  
[View Reference Point]:  
210.0 110.0 30.0  
[View-Up Vector]:  
0.0 0.0 1.0  
[Field of View in y]:  
75.0  
[Aspect]:  
1.0  
[Near Clipping Plane]:  
10  
[Far Clipping Plane]:  
80
```



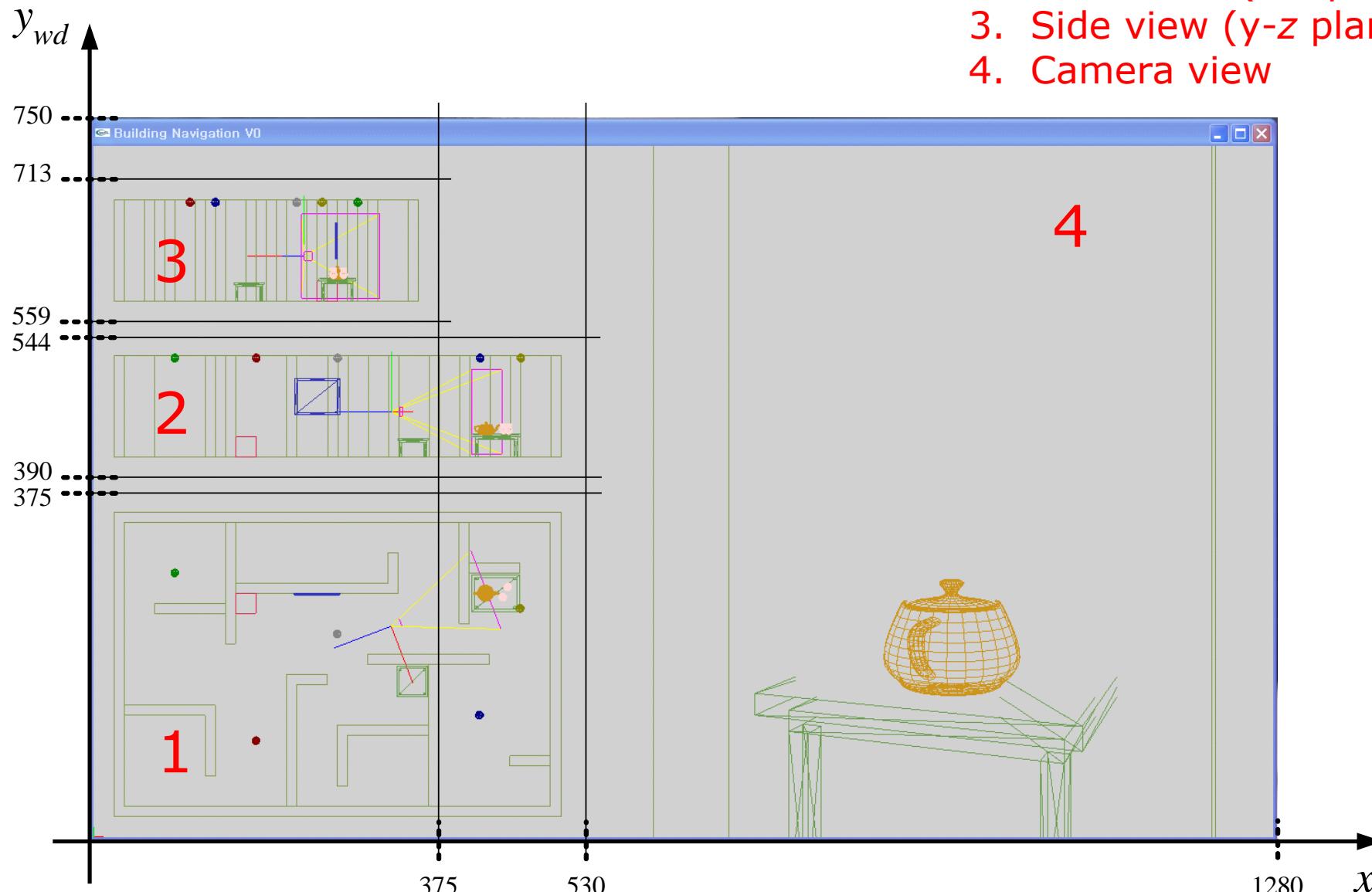
- 사용자가 왼쪽 마우스 버튼을 누른 상태에서 움직일 경우 카메라는 다음과 같은 방식으로 움직여야 한다.
 - 위쪽 방향: 카메라 프레임이 EC의 $-z$ 축 방향으로 이동
 - 아래쪽 방향: 카메라 프레임이 EC의 z 축 방향으로 이동
 - 오른쪽 방향: 카메라 프레임이 EC의 y 축 둘레로 오른쪽 방향으로 회전
 - 왼쪽 방향: 카메라 프레임이 EC의 y 축 둘레로 왼쪽 방향으로 회전
- 사용자가 오른쪽 마우스 버튼을 누른 상태에서 움직일 경우 카메라는 다음과 같은 방식으로 움직여야 한다.
 - 위쪽 방향: 카메라 프레임이 EC의 y 축 방향으로 이동
 - 아래쪽 방향: 카메라 프레임이 EC의 $-y$ 축 방향으로 이동
- 'q' 키를 누르면 프로그램 종료.

건물 내부의 구조



Viewport의 배치

1. Top view (x - y plane)
2. Front view (x - z plane)
3. Side view (y - z plane)
4. Camera view



자료 구조

- 다음과 같은 자료 구조를 바탕으로 필요 시 확장 및 변환을 할 것.

```
#define NUM_MAX_ATTRI 8

typedef struct _polygon {
    int nvertex;
    float face[MAX_VERTICES] [NUM_MAX_ATTRI];
} Polygon;

typedef struct _material {
    float emission[4], ambient[4], diffuse[4],
          specular[4], exponent;
} Material;

typedef struct _xform {
    float matrix[16];
} Xform;

typedef struct _object {
    int type;
    int nface;
    Polygon *polyhedron;
    float xmin, xmax, ymin, ymax, zmin, zmax;
    int ncopy;
    Material *material;
    Xform *xform;
} Object;
```

```
typedef struct _light {
    float position[4], ambient[4], diffuse[4],
          specular[4];
} Light;

typedef struct _scene {
    int nobject;
    char *fname[256];
    Object *objects;
    int nlight;
    Light *lights;
} Scene;

typedef struct _cam {
    float pos[3];
    float uaxis[3], vaxis[3], naxis[3];
    GLfloat mat[16];
    int move, upanddown;
    GLdouble fovy, aspect, near_c, far_c;
} Camera;
```

구현 예

이 문장이 반드시 필요한가?

```
void draw_scene(Scene *scene,
                Camera *cam) {
    int i, j;

    glViewport(530, 0, 750, 750);

    glMatrixMode(GL_PROJECTION);
glLoadIdentity();
    gluPerspective(cam->fovy, cam->aspect,
                   cam->near_c, cam->far_c);

    glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
glMultMatrixf(cam->mat);
    glTranslatef(-cam->pos[X],
                 -cam->pos[Y], -cam->pos[Z]);

    draw_axis();
}
```

A

이 두 문장을 하나의 OpenGL 문장으로 대체하려면?

이 세 문장을 A 지점으로 옮기면 어떤 일이 일어날까?

이 문장의 역할은?

```
for (i = 0; i < scene->nobject; i++) {
    for (j = 0; j <
         scene->objects[i].ncopy;
         j++) {
        glPushMatrix();
        glMultMatrixf((GLfloat *)
          &(scene->objects[i].xform[j]));
        draw_object(
            &(scene->objects[i]));
        glPopMatrix();
    }
    for (i = 0; i < scene->nlight; i++) {
        glPushMatrix();
        glColor3fv(
            scene->lights[i].diffuse);
        glTranslatef(...);
        glutSolidSphere(2.0, 10, 10);
        glPopMatrix();
    }
}
```

이런 glut 함수를 활용할 것.

입력 장면 파일 형식

```
6 ←  
data/Building1.txt  
1  
0.0 0.0 0.0 1.0 ]  
0.4 0.4 0.4 1.0 ]  
0.5 0.6 0.3 1.0  
0.6 0.6 0.6 1.0 ]  
55.0  
1.0 0.0 0.0 0.0 ]  
0.0 1.0 0.0 0.0 ]  
0.0 0.0 1.0 0.0 ]  
0.0 0.0 0.0 1.0 ]  
data/Box.txt  
...  
data/Frame.txt  
...  
data/Table.txt  
2  
0.0 0.0 0.0 1.0 ]  
...  
0.5 0.5 0.5 1.0 ]  
15.0  
0.5 0.0 0.0 157.0 ]  
...  
0.0 0.0 0.0 1.0 ]  
0.0 0.0 0.0 1.0 ]  
...  
0.5 0.5 0.5 1.0 ]  
55.0  
0.8 0.0 0.0 198.0 ]  
...  
0.0 0.0 0.0 1.0 ]
```

```
data/Teapotn.txt  
...  
data/Jar.txt  
...  
5 ←  
120.0 100.0 49.0 1.0 ]  
0.7 0.7 0.7 1.0 ]  
0.5 0.5 0.5 1.0 ]  
0.8 0.8 0.8 1.0 ]  
80.0 47.5 49.0 1.0  
...  
0.5 0.0 0.0 1.0  
40.0 130.0 49.0 1.0  
...  
0.0 0.5 0.0 1.0  
190.0 60 49.0 1.0  
...  
0.0 0.0 0.5 1.0  
210.0 112.5 49.0 1.0  
...  
0.5 0.5 0.0 1.0
```

다면체 데이터 파일 형식

T ← **V** or **N** or **T**

53

4

10.000000 10.000000 0.000000 0.000000 0.000000 1.000000 0.000000 0.000000
230.000000 10.000000 0.000000 0.000000 0.000000 1.000000 1.000000 0.000000
230.000000 160.000000 0.000000 0.000000 0.000000 1.000000 1.000000 1.000000
10.000000 160.000000 0.000000 0.000000 0.000000 1.000000 0.000000 1.000000

4

10.000000 160.000000 50.000000 0.000000 0.000000 -1.000000 0.000000 0.000000
230.000000 160.000000 50.000000 0.000000 0.000000 -1.000000 1.000000 0.000000
230.000000 10.000000 50.000000 0.000000 0.000000 -1.000000 1.000000 1.000000
10.000000 10.000000 50.000000 0.000000 0.000000 -1.000000 0.000000 1.000000

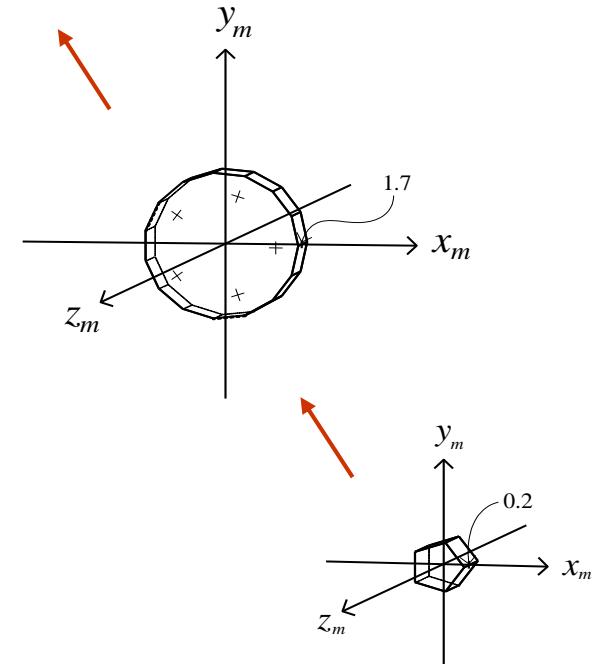
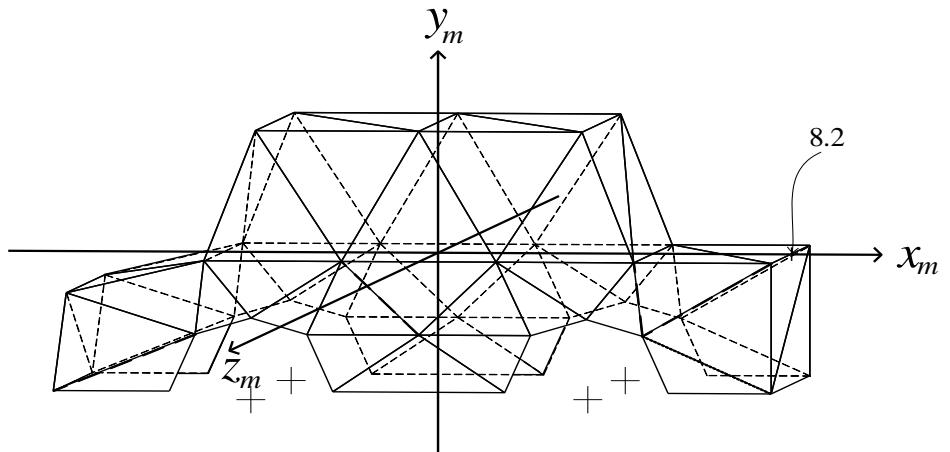
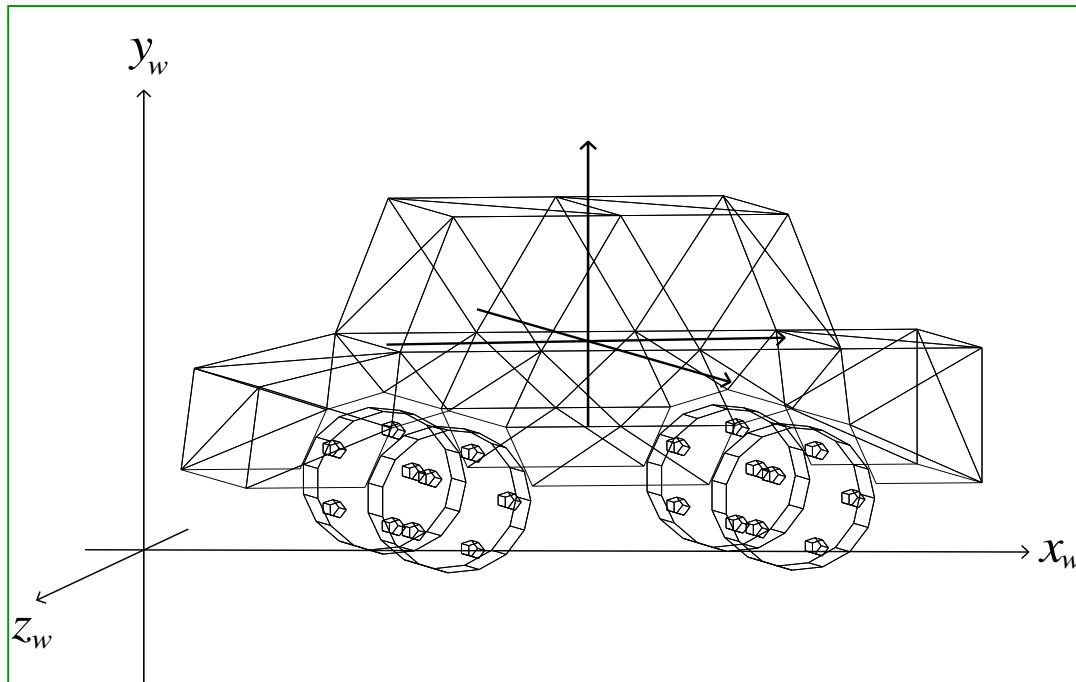
...

V **N** **T**

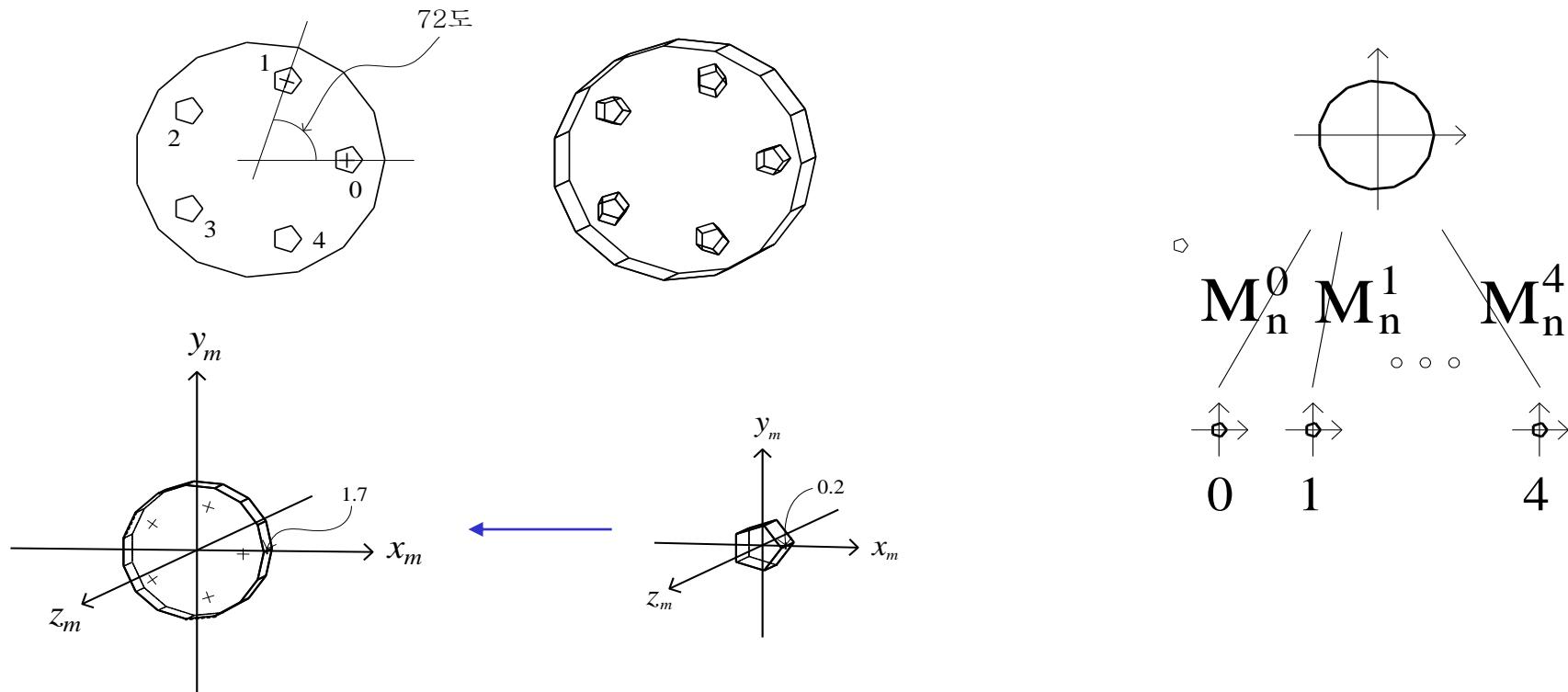
Hierarchical Modeling

[예1] 계층적 구조를 가지는 자동차

- 계층적 구조(Hierarchical Structure)
 - 구성 요소간의 종속 관계 존재
 - 자동차가 움직임에 따라 몸체, 바퀴, 나사가 종속적으로 움직임.
- 종속성을 어떻게 표현할 것인가?
 - 자료 구조 및 알고리즘



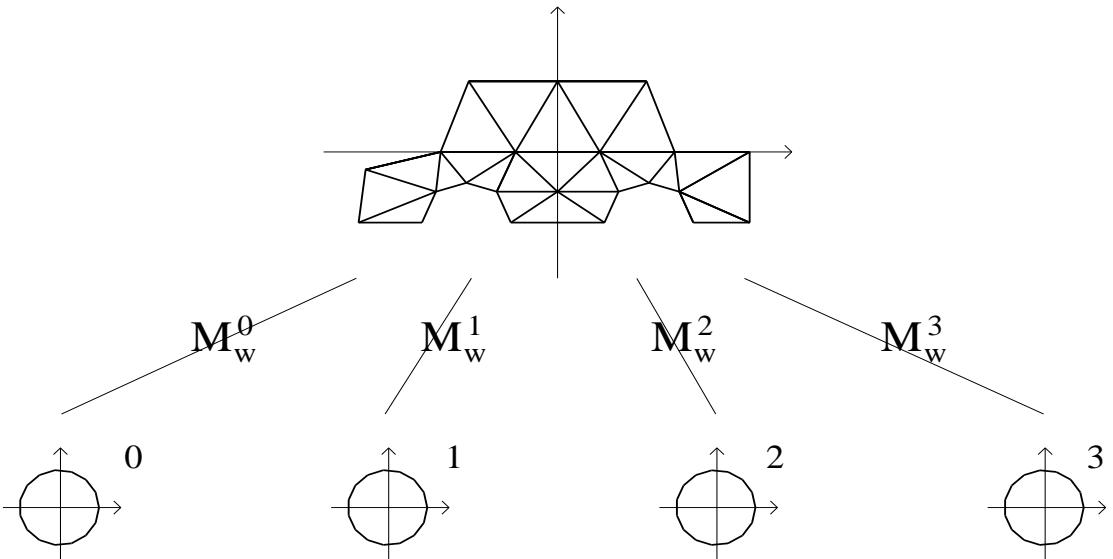
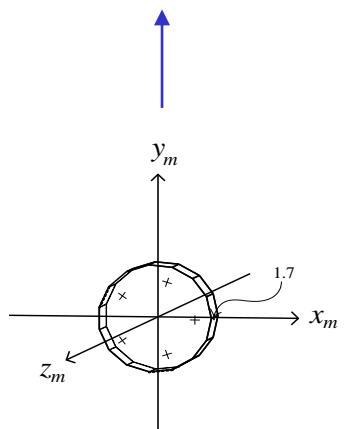
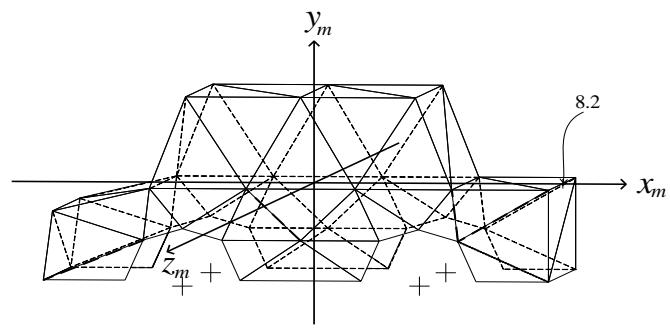
바퀴와 나사와의 관계



$$M_n^i = R(72 \cdot i, 0, 0, 1) \cdot T(1.2, 0, 1) \quad (i = 0, 1, \dots, 4)$$

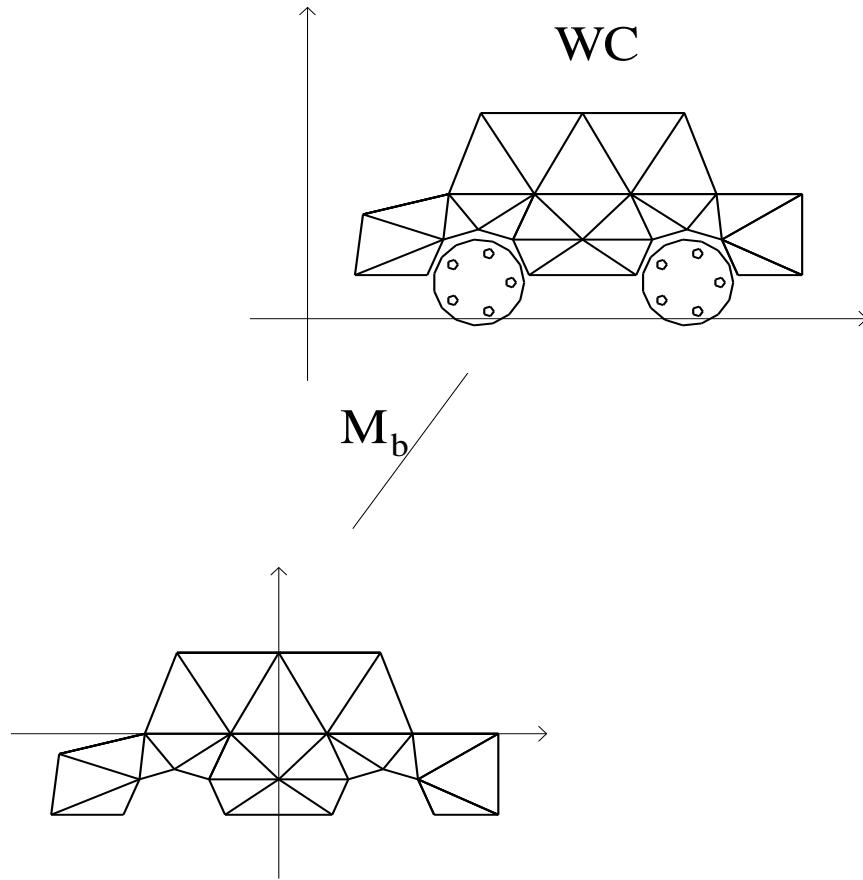
```
glRotatef(72.0*i, 0.0, 0.0, 1.0);
glTranslatef(rad-0.5, 0, ww);
// rad = 1.7, ww = 1.0
```

몸체와 바퀴와의 관계



$$\boxed{\begin{aligned}M_w^0 &= T(-3.9, -3.5, 4.5), \\M_w^1 &= T(3.9, -3.5, 4.5), \\M_w^2 &= T(-3.9, -3.5, -4.5) \cdot S(1, 1, -1), \\M_w^3 &= T(3.9, -3.5, -4.5) \cdot S(1, 1, -1)\end{aligned}}$$

세상과 몸체와의 관계

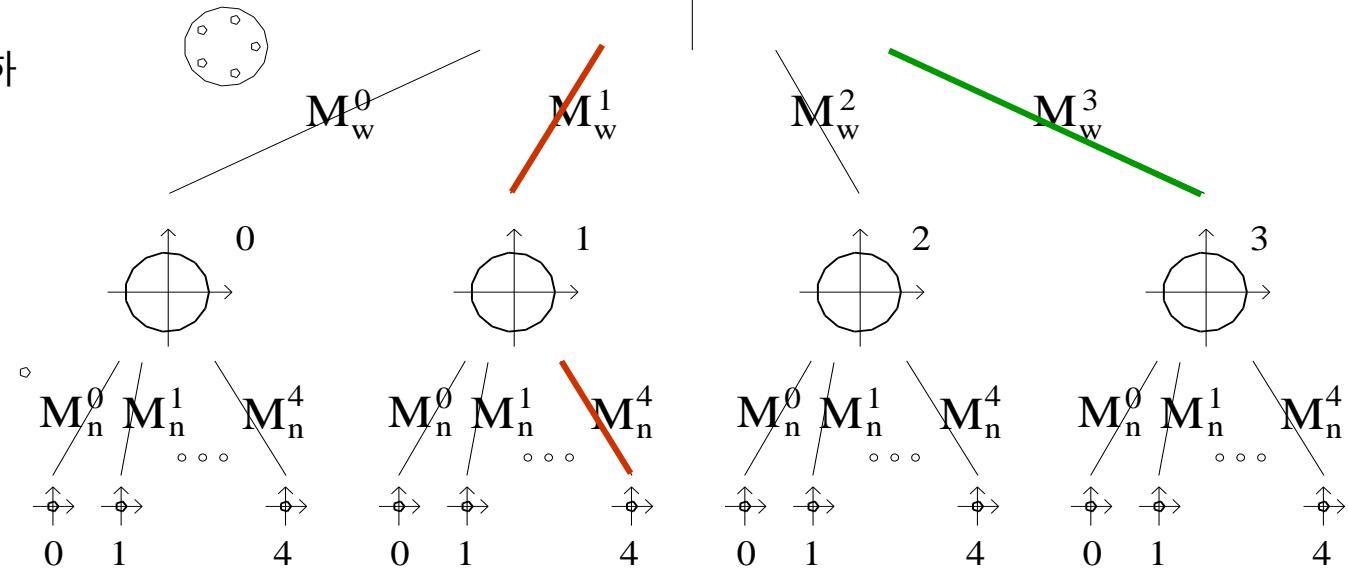


자동차의 계층적 구조

- 트리를 사용하여 종속 관계를 계층적으로 표현.
- 각 물체는 자신의 부모 노드로의 변환만 기억하고 있으면 됨.
- 각 물체는 자신부터 루트 노트까지의 경로상의 행렬을 순서대로 곱하여 세상 좌표계로 배치가 됨.
 - 1번 바퀴의 4번 나사의 경우

$$M_M = M_b \cdot M_w^1 \cdot M_n^4$$

- 개념적으로 각 노드는 해당 기하 물체에 대한 포인터 저장.



계층적 구조의 구현

- 전체 자동차를 세상에 배치하기 위하여
 - 트리를 깊이 우선 탐색(Depth First Search, DFS)을 하면서 각 노드를 그림.
 - 스택을 사용하면 트리를 효과적으로 탐색하면서 물체를 세상에 배치할 수 있음.
 - ① 아래로 내려 갈 때 변에 있는 행렬을 푸쉬하고,
 - ② 다시 위로 올라올 때 그 행렬을 팝하며,
 - ③ 현재 노드에 기하 물체가 붙어 있으면 스택에 있는 행렬들을 순서대로 곱하여 세상 좌표계에 배치함.
 - OpenGL을 사용하여 구현을 할 경우 OpenGL에서 제공하는 스택 연산을 특성을 고려하여 트리를 탐색하여야 함.

OpenGL을 사용한 깊이 우선 탐색 (Compatibility Profile)

```
void draw_wheel_and_nut(void) {
    int i;

    draw_wheel(); // draw wheel object
    for (i = 0; i < 5; i++) {
        // for the i-th nut
        glPushMatrix();
        glRotatef(72.0*i, 0.0, 0.0, 1.0);
        glTranslatef(rad-0.5, 0, ww);
        // rad = 1.7, ww = 1.0
        draw_nut(); // Line (d)
        // draw nut object
        glPopMatrix();
    }

    void draw_car(void) {
        draw_body(); // draw body object

        glPushMatrix(); // for wheel 0
        glTranslatef(-3.9, -3.5, 4.5);
        draw_wheel_and_nut(); // Line (c)
        glPopMatrix();
    }
}
```

```
glPushMatrix(); // for wheel 1
glTranslatef(3.9, -3.5, 4.5);
draw_wheel_and_nut();
glPopMatrix();

glPushMatrix(); // for wheel 2
glTranslatef(-3.9, -3.5, -4.5);
glScalef(1.0, 1.0, -1.0);
draw_wheel_and_nut();
glPopMatrix();

glPushMatrix(); // for wheel 3
glTranslatef(3.9, -3.5, -4.5);
glScalef(1.0, 1.0, -1.0);
draw_wheel_and_nut();
glPopMatrix();
}
```

```

void draw_world(void) {
    draw_axes();

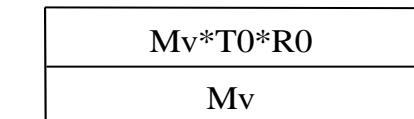
    glPushMatrix(); // Line (a)
    // from body to world
    glTranslatef(10.0, 5.0, 0.0);
    glRotatef(15.0, 0.0, 1.0, 0.0);
    draw_car(); // Line (b)
    glPopMatrix();
}

```

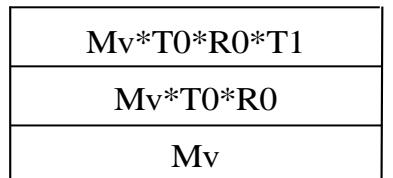
- 자동차 바퀴가 회전한다면 이를 어떻게 처리할 수 있을까?



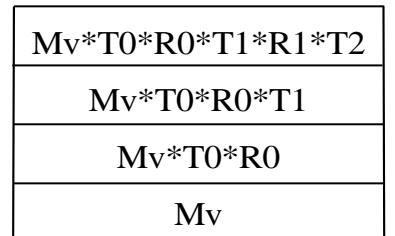
Line (a) 수행 직전



Line (b) 수행 직전



Line (c) 수행 직전



Line (d) 수행 직전

OpenGL을 사용한 깊이 우선 탐색 (Core Profile)

```
void display(void) {
    glm::mat4 ModelMatrix_big_cow, ModelMatrix_small_cow;
    glm::mat4 ModelMatrix_big_box, ModelMatrix_small_box;
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    ...

    ModelMatrix_CAR_BODY = glm::rotate(glm::mat4(1.0f), -rotation_angle_car,
                                       glm::vec3(0.0f, 1.0f, 0.0f));
    ModelMatrix_CAR_BODY = glm::translate(ModelMatrix_CAR_BODY,
                                         glm::vec3(20.0f, 4.89f, 0.0f));
ModelMatrix_CAR_BODY = glm::rotate(ModelMatrix_CAR_BODY, 90.0f*TO_RADIAN,
                                         glm::vec3(0.0f, 1.0f, 0.0f));

Mb

    ModelViewProjectionMatrix = ViewProjectionMatrix * ModelMatrix_CAR_BODY;
    glUniformMatrix4fv(loc_ModelViewProjectionMatrix, 1, GL_FALSE,
                      &ModelViewProjectionMatrix[0][0]);

draw_car_dummy();

    glutSwapBuffers();
}
```

```

void draw_car_dummy(void) {
    glUniform3f(loc_primitive_color, 0.498f, 1.000f, 0.831f);
    draw_geom_obj(GEOM_OBJ_ID_CAR_BODY); // draw body
    glLineWidth(2.0f);
    draw_axes(); // draw MC axes of body
    glLineWidth(1.0f);

    ModelMatrix_CAR_DRIVER = glm::translate(ModelMatrix_CAR_BODY,
                                             glm::vec3(-3.0f, 0.5f, 2.5f));
    ModelMatrix_CAR_DRIVER = glm::rotate(ModelMatrix_CAR_DRIVER, TO_RADIAN*90.0f,
                                         glm::vec3(0.0f, 1.0f, 0.0f));
    ModelViewProjectionMatrix = ViewProjectionMatrix * ModelMatrix_CAR_DRIVER;
    glUniformMatrix4fv(loc_ModelViewProjectionMatrix, 1, GL_FALSE,
                       &ModelViewProjectionMatrix[0][0]);
    glLineWidth(5.0f);
    draw_axes(); // draw camera frame at driver seat
    glLineWidth(1.0f);

    ModelMatrix_CAR_WHEEL = glm::translate(ModelMatrix_CAR_BODY,
                                           glm::vec3(-3.9f, -3.5f, 4.5f));
    
$$M_b \cdot M_w^0$$

    ModelViewProjectionMatrix = ViewProjectionMatrix * ModelMatrix_CAR_WHEEL;
    glUniformMatrix4fv(loc_ModelViewProjectionMatrix, 1, GL_FALSE,
                       &ModelViewProjectionMatrix[0][0]);
    draw_wheel_and_nut(); // draw wheel 0
    ...
}

```

```

#define rad 1.7f
#define ww 1.0f

void draw_wheel_and_nut() {
    int i;

    glUniform3f(loc_primitive_color, 0.000f, 0.808f, 0.820f);
    draw_geom_obj(GEOM_OBJ_ID_CAR_WHEEL); // draw wheel

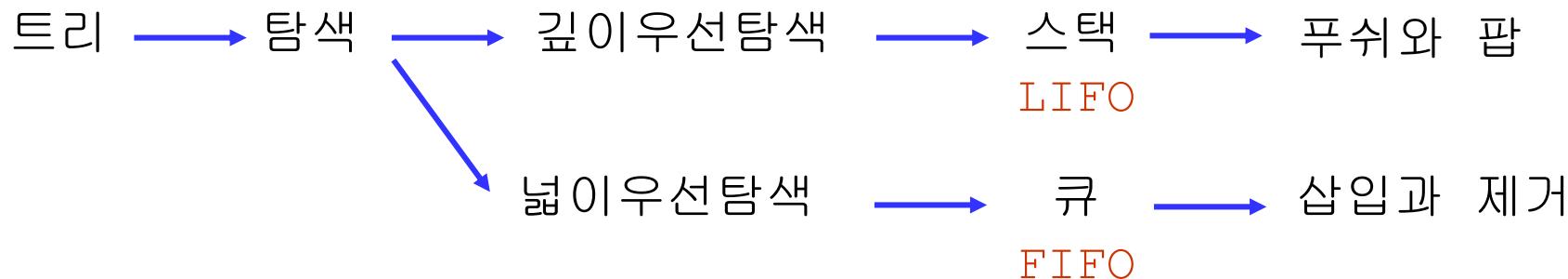
    for (i = 0; i < 5; i++) {
        ModelMatrix_CAR_NUT = glm::rotate(ModelMatrix_CAR_WHEEL, TO_RADIAN*72.0f*i,
                                           glm::vec3(0.0f, 0.0f, 1.0f));
        ModelMatrix_CAR_NUT = glm::translate(ModelMatrix_CAR_NUT,
                                             glm::vec3(rad - 0.5f, 0.0f, ww));
 $M_b \cdot M_w^0 \cdot M_n^i$ 

        ModelViewProjectionMatrix = ViewProjectionMatrix * ModelMatrix_CAR_NUT;
        glUniformMatrix4fv(loc_ModelViewProjectionMatrix, 1, GL_FALSE,
                           &ModelViewProjectionMatrix[0][0]);
    }

    glUniform3f(loc_primitive_color, 0.690f, 0.769f, 0.871f);
    draw_geom_obj(GEOM_OBJ_ID_CAR_NUT); // draw i-th nut
}
}

```

종속성의 성질을 가지는 정보의 표현

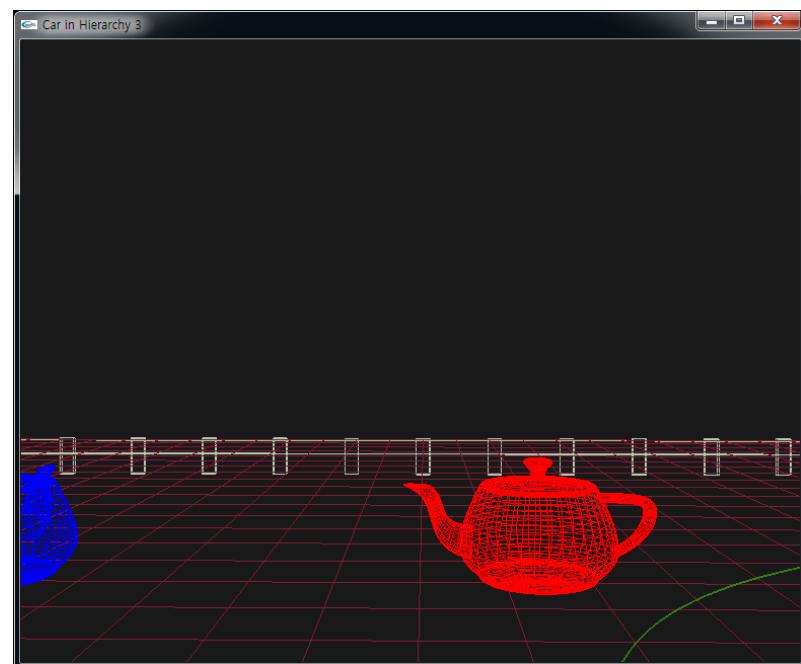
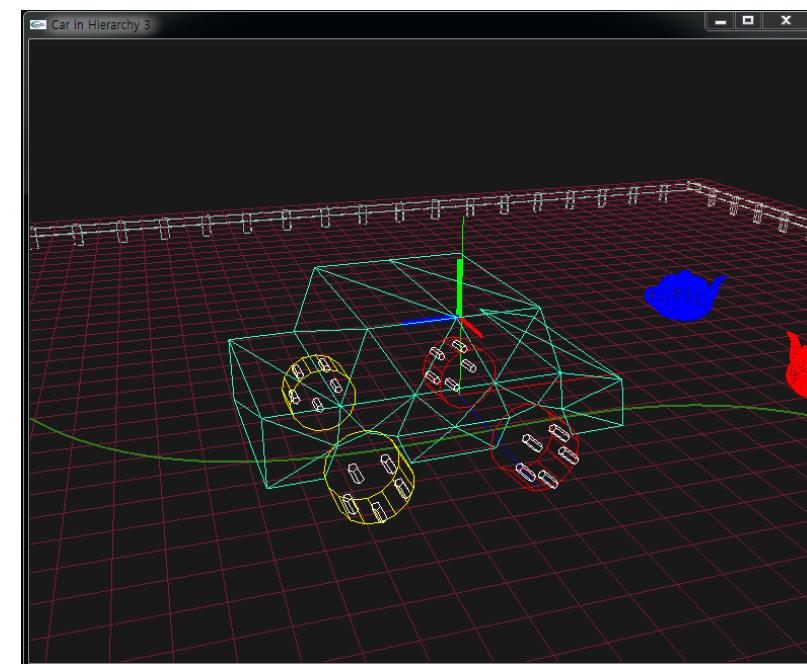


- 자동차에 존재하는 종속 관계를 트리를 사용하여 계층적으로 표현.
- 모델링 변환을 위하여 자동차에 대한 트리를 스택을 사용하여 깊이 우선 탐색.
- 프로그래밍 언어의 블록 구조(block structure)에 내재하는 계층적 종속성.

계층적 모델링을 사용하는 프로그램 예 (Compatibility Profile)

- 기능

이에 해당하는 프로그램을 Core Profile을 사용하여 구현하여 보자!



```

#include <stdio.h>
#include <math.h>
#include <GL/glut.h>

#define MAX_POLY 200
#define MAX_VERT 20
#define MAX_PATH 1000

typedef struct { int nvertex; float poly[MAX_VERT][3]; } mypolygon;

mypolygon body[MAX_POLY], wheel[MAX_POLY], nut[MAX_POLY];
int npolyb, npolyw, npolyn;

float path[MAX_PATH][3];
int npath, path_exist;

double dist;

int prev_i, cur_i = 0;
int rightbuttonpressed = 0;
int camera_mode =
    0;

void read_object(char *file, mypolygon *object, int *npoly) {
    ...
}

```

```

void read_path(char *file) {
    ...
}

void read_objects(void) {
    read_object("mbody.dat", body, &npolyb);
    read_object("mwheel.dat", wheel, &npolyw);
    read_object("mnut.dat", nut, &npolyn);
    read_path("path.dat");
}

void draw_axes(void) {
    glBegin(GL_LINES);
    glColor3f(1.0, 0.0, 0.0);
    glVertex3f(0.0, 0.0, 0.0); glVertex3f(10.0, 0.0, 0.0);

    ...
    glEnd();
}

```

```

void draw_camera_frame(void) {
    glLineWidth(5.0);
    glBegin(GL_LINES);
    glColor3f(1.0, 0.0, 0.0);
    glVertex3f(0.0, 0.0, 0.0); glVertex3f(3.0, 0.0, 0.0);

    ...
    glEnd();
}

void draw_path(void) {
    int i;

    glColor3f(0.2, 0.5, 0.1);
    glLineWidth(2.0);

    glBegin(GL_LINE_STRIP);
    for (i = 0; i < npath; i++)
        glVertex3f(path[i][0], path[i][1], path[i][2]);
    glEnd();

    glLineWidth(1.0);
}

```

```

void draw_ground(void) {
    float x, z;

    glColor3f(0.5, 0.1, 0.2);
    for (x = -50.0; x <= 50.0; x += 2.0) {
        glBegin(GL_LINES); // Use GL_QUADS !
        glVertex3f(x, 0.0, -50.0); glVertex3f(x, 0.0, 50.0);
        glEnd();
    }

    for (z = -50.0; z <= 50.0; z += 2.0) {
        glBegin(GL_LINES);
        glVertex3f(-50.5, 0.0, z); glVertex3f(50.0, 0.0, z);
        glEnd();
    }

    glColor3f(1.0, 0.0, 0.0);
    glPushMatrix();
    glTranslatef(20.0, 1.0, 5.0); glRotatef(90.0, 0.0, 1.0, 0.0);
    glutWireTeapot(2.0);
    glPopMatrix();
    glColor3f(0.0, 0.0, 1.0);
    glPushMatrix();
    glTranslatef(20.0, 1.0, -9.0);
    glutWireTeapot(2.0);
    glPopMatrix();
}

```

```

void draw_body(void) {
    int i, j;

    glPushMatrix();
    glTranslatef(-3.0, 0.5, 2.5); glRotatef(90.0, 0.0, 1.0, 0.0);
    draw_camera_frame();
    glPopMatrix();

    glColor3d(0.2, 1.0, 0.8);
    for (i = 0; i < npolyb; i++) {
        glBegin(GL_POLYGON);
        for (j = 0; j < body[i].nvertex; j++) {
            glVertex3f(body[i].poly[j][0], body[i].poly[j][1],
                      body[i].poly[j][2]);
        }
        glEnd();
    }
}

void draw_wheel(float angle) {
    int i, j;

    if (angle == 0.0) glColor3d(1.0, 1.0, 0.0);
    else if (angle > 0.0) glColor3d(1.0, 0.0, 0.0);
    else glColor3f(0.0, 0.0, 1.0);
}

```

```

for (i = 0; i < npolyw; i++) {
    glBegin(GL_POLYGON);
    for (j = 0; j < wheel[i].nvertex; j++) {
        glVertex3f(wheel[i].poly[j][0], wheel[i].poly[j][1],
                  wheel[i].poly[j][2]);
    }
    glEnd();
}

void draw_nut(void) {
    int i, j;

    glColor3d(1.0, 1.0, 1.0);
    for (i = 0; i < npolyn; i++) {
        glBegin(GL_POLYGON);
        for (j = 0; j < nut[i].nvertex; j++) {
            glVertex3f(nut[i].poly[j][0], nut[i].poly[j][1], nut[i].poly[j][2]);
        }
        glEnd();
    }
}

```

```

#define rad 1.7
#define ww 1.0
void draw_wheel_and_nut(float angle) {
    int i;

    draw_wheel(angle); // draw wheel object
    for (i = 0; i < 5; i++) {
        // nut i
        glPushMatrix();
        glRotatef(72.0*i, 0.0, 0.0, 1.0);
        glTranslatef(rad-0.5, 0, ww); // rad = 1.7, ww = 1.0
        draw_nut(); // draw nut object
        glPopMatrix();
    }

#define TO_DEG 57.29579
void normalize_vec3(float *v) { ... }

float dot_prod_vec3(float *u, float *v) { ... }

float compute_length_mul_two_vec3(float *u, float *v) { ... }

float angle_between_two_vec3(float *u, float *v) { ... }

void cross_prod_vec3(float *u, float *v, float *n) { ... }

```

```

float wheel_rot_angle_in_y(void) {
    int i;
    float angle, before[3], after[3], result[3];

    angle = 0.0;
    if ((cur_i < 50) || (cur_i >= npath - 50)) return(angle);
    for (i = 0; i < 3; i++) before[i] = path[cur_i][i] - path[cur_i-50][i];
    for (i = 0; i < 3; i++) after[i] = path[cur_i+50][i] - path[cur_i][i];

    angle = 1.2*TO_DEG*angle_between_two_vec3(before, after);
    cross_prod_vec3(before, after, result);

    if (result[1] < 0) angle *= -1.0;
    return(angle);
}

float wheel_rot_angle_in_z(void) {
    static int angle = 0.0;

    angle += 0.8*180.0*dist/(3.141592*rad);
    if (angle >= 360.0) angle -= 360.0;

    return(angle);
}

```

```

void draw_car_correct(void) {
    float angle_y = 0.0, angle_z = 0.0;
    angle_y = wheel_rot_angle_in_y();
    angle_z = wheel_rot_angle_in_z();

    draw_body(); // draw body object
    glPushMatrix();
    glTranslatef(-3.9, -3.5, 4.5); glRotatef(angle_y, 0.0, 1.0, 0.0);
    glRotatef(angle_z, 0.0, 0.0, 1.0);
    draw_wheel_and_nut(angle_y); // wheel 0
    glPopMatrix();

    glPushMatrix();
    glTranslatef(3.9, -3.5, 4.5); glRotatef(angle_z, 0.0, 0.0, 1.0);
    draw_wheel_and_nut(0.0); // wheel 1
    glPopMatrix();

    glPushMatrix();
    glTranslatef(-3.9, -3.5, -4.5); glRotatef(angle_y, 0.0, 1.0, 0.0);
    glRotatef(angle_z, 0.0, 0.0, 1.0); glScalef(1.0, 1.0, -1.0);
    draw_wheel_and_nut(angle_y); // wheel 2
    glPopMatrix();

    glPushMatrix();
    glTranslatef(3.9, -3.5, -4.5); glRotatef(angle_z, 0.0, 0.0, 1.0);
    glScalef(1.0, 1.0, -1.0);
    draw_wheel_and_nut(0.0); // wheel 3
    glPopMatrix();
}

```

```

void set_up_rot_mat(float *m, float *minv, int i) {
    GLfloat u[3], v[3], n[3];

    v[0] = v[2] = 0.0; v[1] = 1.0;
    if (i == 0) {
        u[0] = path[0][0] - path[1][0]; u[1] = path[0][1] - path[1][1];
        u[2] = path[0][2] - path[1][2];
    }
    else {
        u[0] = path[i-1][0] - path[i][0]; u[1] = path[i-1][1] - path[i][1];
        u[2] = path[i-1][2] - path[i][2];
    }
    normalize_vec3(u);
    cross_prod_vec3(u, v, n);
    m[0] = u[0]; m[1] = u[1]; m[2] = u[2]; m[3] = 0.0;
    m[4] = v[0]; m[5] = v[1]; m[6] = v[2]; m[7] = 0.0;
    m[8] = n[0]; m[9] = n[1]; m[10] = n[2]; m[11] = 0.0;
    m[12] = m[13] = m[14] = 0.0; m[15] = 1.0;

    minv[0] = u[0]; minv[1] = v[0]; minv[2] = n[0]; minv[3] = 0.0;
    minv[4] = u[1]; minv[5] = v[1]; minv[6] = n[1]; minv[7] = 0.0;
    minv[8] = u[2]; minv[9] = v[2]; minv[10] = n[2]; minv[11] = 0.0;
    minv[12] = minv[13] = minv[14] = 0.0; minv[15] = 1.0;
}

```

```

void draw_fence(GLfloat r, GLfloat g, GLfloat b) { ... }
void draw_fences(void) { ... }

void draw_world (void) {
GLfloat m[16], minv[16];

set_up_rot_mat(m, minv, cur_i);

glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
if (camera_mode == 0) {
    gluLookAt(-15.0, 20.0, 40.0, path[cur_i][0], 4.89,
              path[cur_i][2], 0.0, 1.0, 0.0);
}
else {
    glRotatef(-90.0, 0.0, 1.0, 0.0); glTranslatef(3.0, -0.5, -2.5);

    glMultMatrixf(minv);
    glTranslatef(-path[cur_i][0], -4.89 , -path[cur_i][2]);
}
draw_fences(); draw_ground(); draw_axes();
if (path_exist) draw_path();

glPushMatrix(); // from body to world
glTranslatef(path[cur_i][0], 4.89 , path[cur_i][2]);
glMultMatrixf(m);
draw_car_correct();
glPopMatrix();
}

```

```

void render(void) {
    glClear(GL_COLOR_BUFFER_BIT);
    draw_world();
    glutSwapBuffers();
}

void keyboard (unsigned char key, int x, int y) {
    switch (key) {
        case 'q': exit(0); break;
        case 'v': camera_mode = 1 - camera_mode;
                    glutPostRedisplay(); break;
    }
}

int prevx_mouse;
void mousepress(int button, int state, int x, int y) { ... }

#define MOUSESENSITIVITY 0
void mousemove(int x, int y) {
    double deltax;

    if (rightbuttonpressed) {
        deltax = (x - prevx_mouse) >> MOUSESENSITIVITY;
        prevx_mouse = x;
        if ((cur_i + deltax > 0) && (cur_i + deltax < npath-1)) {
            prev_i = cur_i; cur_i += deltax;
            dist = ... // distance between path[cur_i] and path[prev_i]
            glutPostRedisplay();
        }
    }
}

```

```
void reshape(int width, int height) {  
    glViewport(0, 0, width, height);  
  
    glMatrixMode(GL_PROJECTION);  
    glLoadIdentity();  
    gluPerspective(40.0, width/(double) height, 1.0, 150.0);  
}  
  
void init_OpenGL(void) {
```

```
    printf("***** MOVE MOUSE HORIZONTALLY WITH RIGHT  
    MOUSEBUTTON PRESSED...\\n");
```

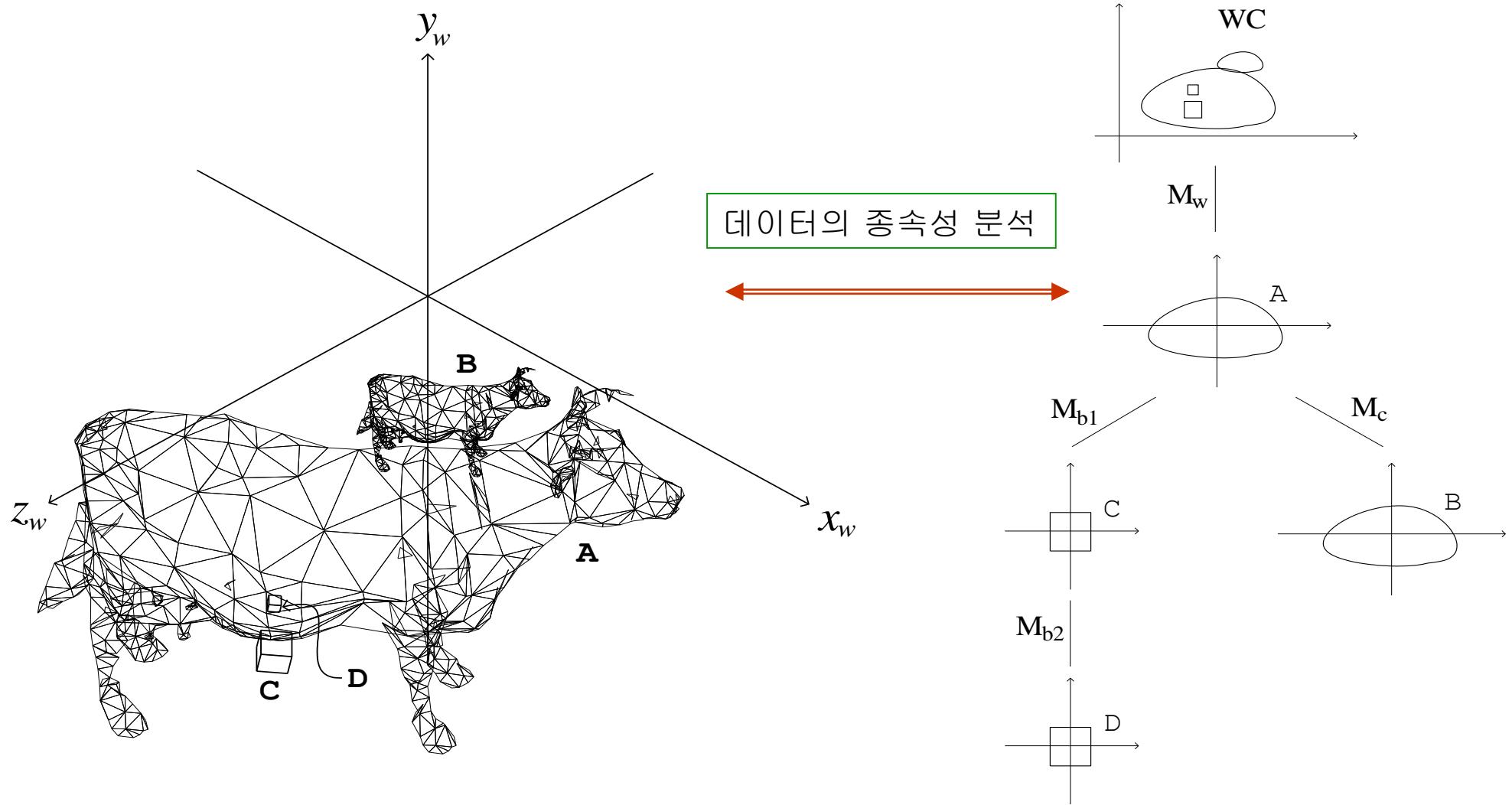
```
    glClearColor(0.1, 0.1, 0.1, 1.0);  
    glPolygonMode(GL_FRONT_AND_BACK, GL_LINE);  
    glFrontFace(GL_CCW);  
    glCullFace(GL_BACK);  
    glEnable(GL_CULL_FACE);
```

```
}
```

```
void init_windows(void) {  
    glutInitDisplayMode(GLUT_RGB | GLUT_DOUBLE);  
    glutInitWindowSize(1280*0.6, 1024*0.6);  
    glutCreateWindow("Car in Hierarchy 3");  
    glutDisplayFunc(render);  
    glutKeyboardFunc(keyboard);  
    glutMouseFunc(mousepress);  
    glutMotionFunc(mousemove);  
    glutReshapeFunc(reshape);  
}
```

```
void main(int argc, char **argv) {  
    read_objects();  
    glutInit(&argc, argv);  
    init_windows();  
    init_OpenGL();  
    glutMainLoop();  
}
```

[예2] 회전하는 어미소와 송아지 (Compatibility Profile)

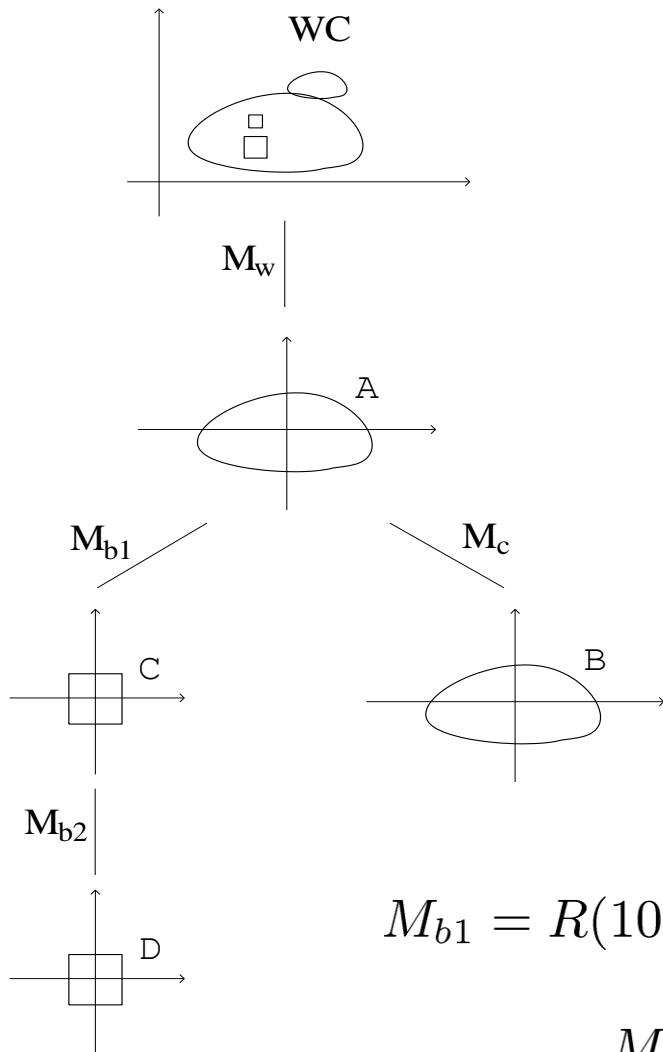


```
Void glutIdleFunc(void (*func) (void));
```

```
void next_frame(void) {  
    angle = ((int) (angle + 1.0)) % 360;  
    glutPostRedisplay();  
}
```

```
void register_callback(void) {  
    glutDisplayFunc(render);  
    glutKeyboardFunc(keyboard);  
    glutIdleFunc(next_frame);  
}
```

매번 그림을 그릴 때마다 angle 값이 0과 359 사이에서 1도씩 증가.



$$M_w = R(\text{angle}, 0, 1, 0) \cdot T(0, 0, 5) \cdot S(6.5, 6.5, 6.5)$$

$$M_c = T(0.15, 0.3, 0) \cdot S(0.3, 0.3, 0.3)$$

$$M_{b1} = R(10 \cdot \text{angle}, 1, 0, 0) \cdot T(-0.1, 0, 0.3) \cdot S(0.025, 0.025, 0.025)$$

$$M_{b2} = R(40 \cdot \text{angle}, 1, 0, 0) \cdot T(0, 4, 0) \cdot S(0.4, 0.4, 0.4)$$

```
void render(void) {
    glClearColor(0.0, 0.0, 0.0, 1.0);
    glClear(GL_COLOR_BUFFER_BIT);

    glViewport(0, 0, 600, 600);

    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    glOrtho(-5.0, 5.0, -5.0, 5.0, 0.0, 1000.0);

    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
    gluLookAt(10.0, 10.0, 10.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0);

    draw_axis();
```

$$M_V \cdot R(angle, 0, 1, 0) \cdot T(0, 0, 5) \cdot S(6.5, 6.5, 6.5) \cdot R(10 \cdot angle, 1, 0, 0) \cdot T(-0.1, 0, 0.3) \\ \cdot S(0.025, 0.025, 0.025) \cdot R(40 \cdot angle, 1, 0, 0) \cdot T(0, 4, 0) \cdot S(0.4, 0.4, 0.4)$$

```

glPushMatrix();
    glRotatef(angle, 0.0, 1.0, 0.0);
    glTranslatef(0.0, 0.0, 5.0);
    glScalef(6.5, 6.5, 6.5);
    draw_cow(1.0, 0.0, 0.0); // 어미소 A

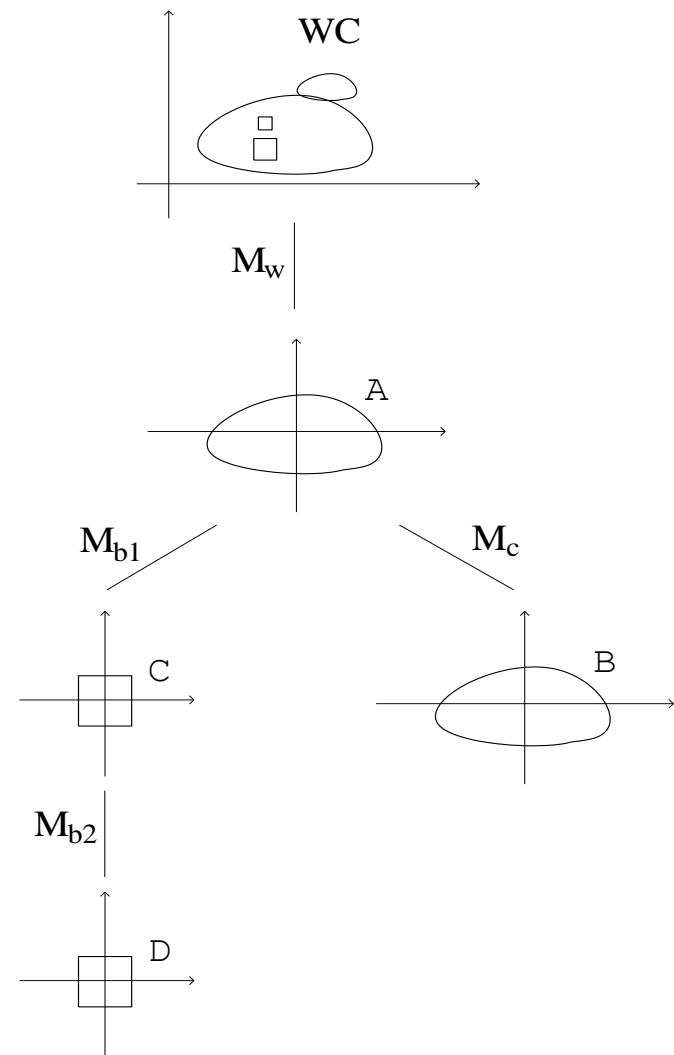
    glPushMatrix();
        glRotatef(10.0*angle, 1.0, 0.0, 0.0);
        glTranslatef(-0.1, 0.0, 0.3);
        glScalef(0.025, 0.025, 0.025);
        draw_box(); // 큰 육면체 C

        glRotatef(40.0*angle, 1.0, 0.0, 0.0);
        glTranslatef(0.0, 4.0, 0.0);
        glScalef(0.4, 0.4, 0.4);
        draw_box(); // 작은 육면체 D
    glPopMatrix();

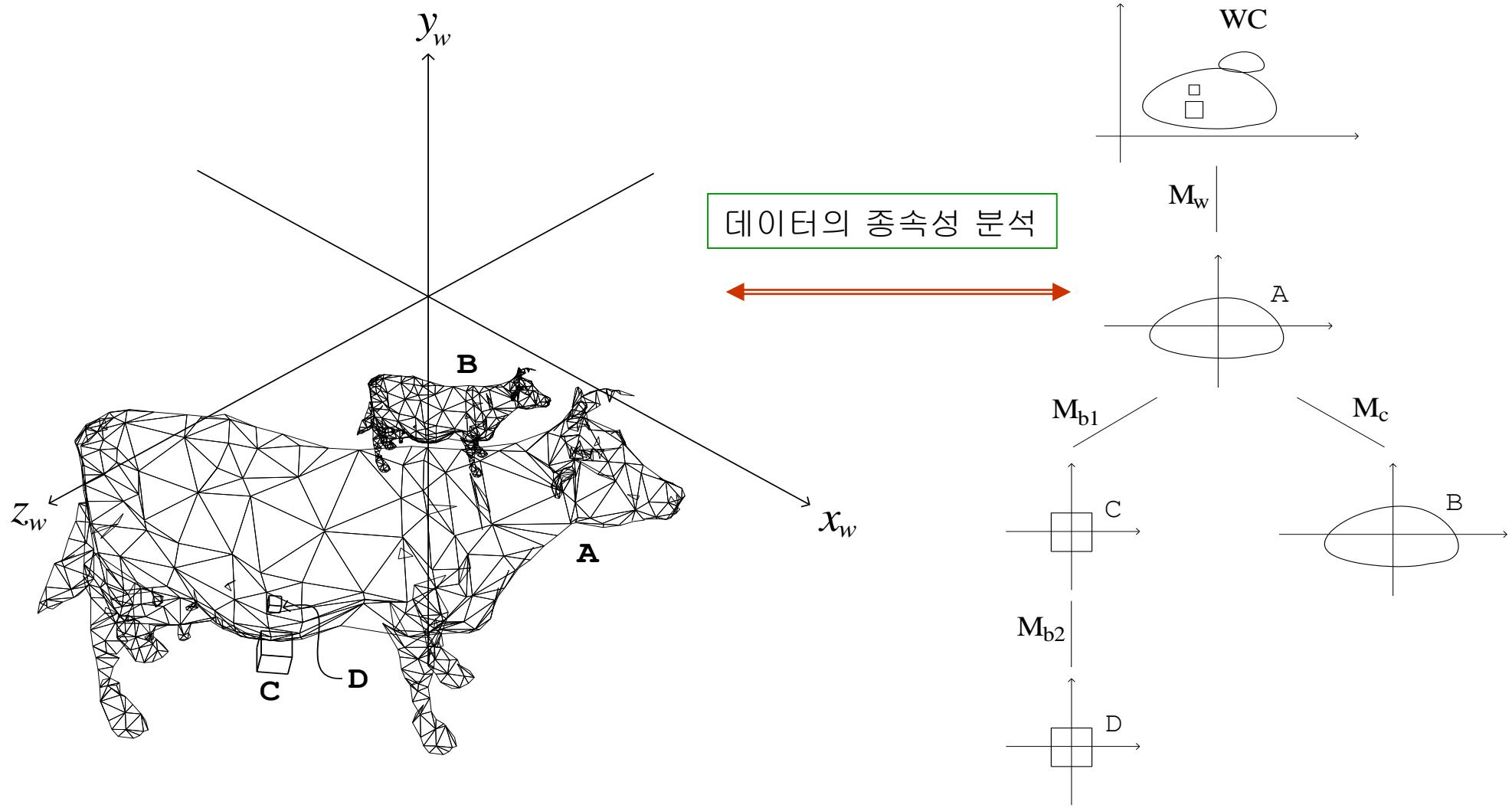
    glTranslatef(0.15, 0.3, 0.0);
    glScalef(0.3, 0.3, 0.3);
    draw_cow(0.0, 0.0, 1.0); // 송아지 B
glPopMatrix();

    glutSwapBuffers();
} // End of render()

```



[예2] 회전하는 어미소와 송아지 (Core Profile)

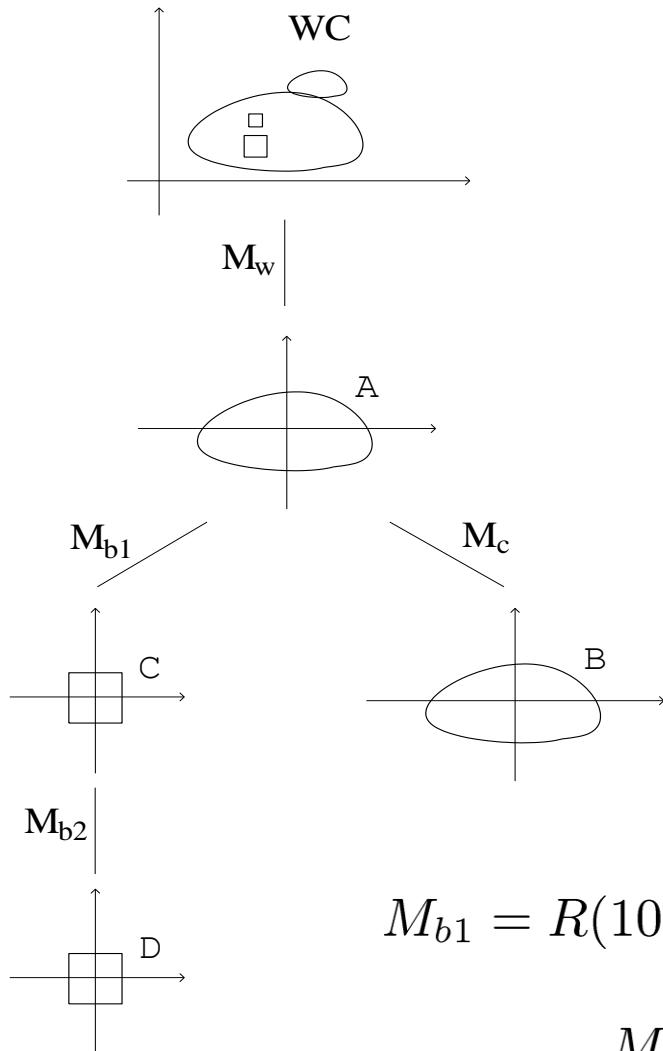


```
void glutTimerFunc(unsigned int msecs, void (*func)(int value), value);
```

```
void timer_scene(int timestamp_scene) {
    rotation_angle_cow = (timestamp_scene
                           % 360) * TO_RADIAN;
    glutPostRedisplay();
    glutTimerFunc(100, timer_scene,
                  (timestamp_scene + 1) % INT_MAX);
}

void register_callbacks(void) {
    ...
    glutTimerFunc(100, timer_scene, 0);
    ...
}
```

매번 그림을 그릴 때마다 `rotation_angle_cow` 값이 0과 359 사이에서 1도씩 증가.



$$M_w = R(\text{angle}, 0, 1, 0) \cdot T(0, 0, 5) \cdot S(6.5, 6.5, 6.5)$$

$$M_c = T(0.15, 0.3, 0) \cdot S(0.3, 0.3, 0.3)$$

$$M_{b1} = R(10 \cdot \text{angle}, 1, 0, 0) \cdot T(-0.1, 0, 0.3) \cdot S(0.025, 0.025, 0.025)$$

$$M_{b2} = R(40 \cdot \text{angle}, 1, 0, 0) \cdot T(0, 4, 0) \cdot S(0.4, 0.4, 0.4)$$

```

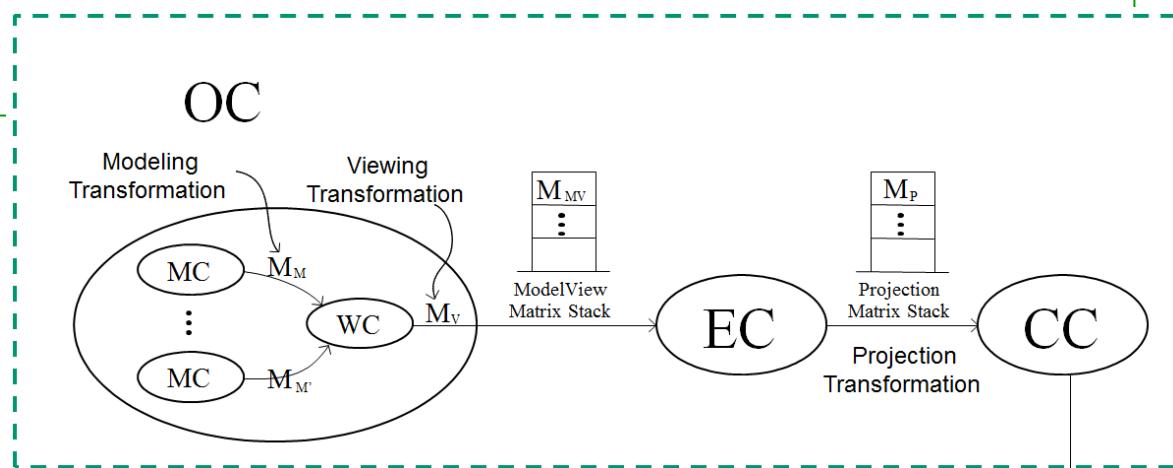
glm::mat4 ModelViewProjectionMatrix;
glm::mat4 ViewProjectionMatrix, ViewMatrix, ProjectionMatrix;

...
float rotation_angle_cow = 0.0f;

void display(void) {
    glm::mat4 ModelMatrix_big_cow, ModelMatrix_small_cow;
    glm::mat4 ModelMatrix_big_box, ModelMatrix_small_box;
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    ModelViewProjectionMatrix = glm::scale(ViewProjectionMatrix,
                                            glm::vec3(1.0f, 1.0f, 1.0f));
    glUniformMatrix4fv(loc_ModelViewProjectionMatrix, 1, GL_FALSE,
                       &ModelViewProjectionMatrix[0][0]);
    glLineWidth(2.0f);
    draw_axes();
    glLineWidth(1.0f);
}

```

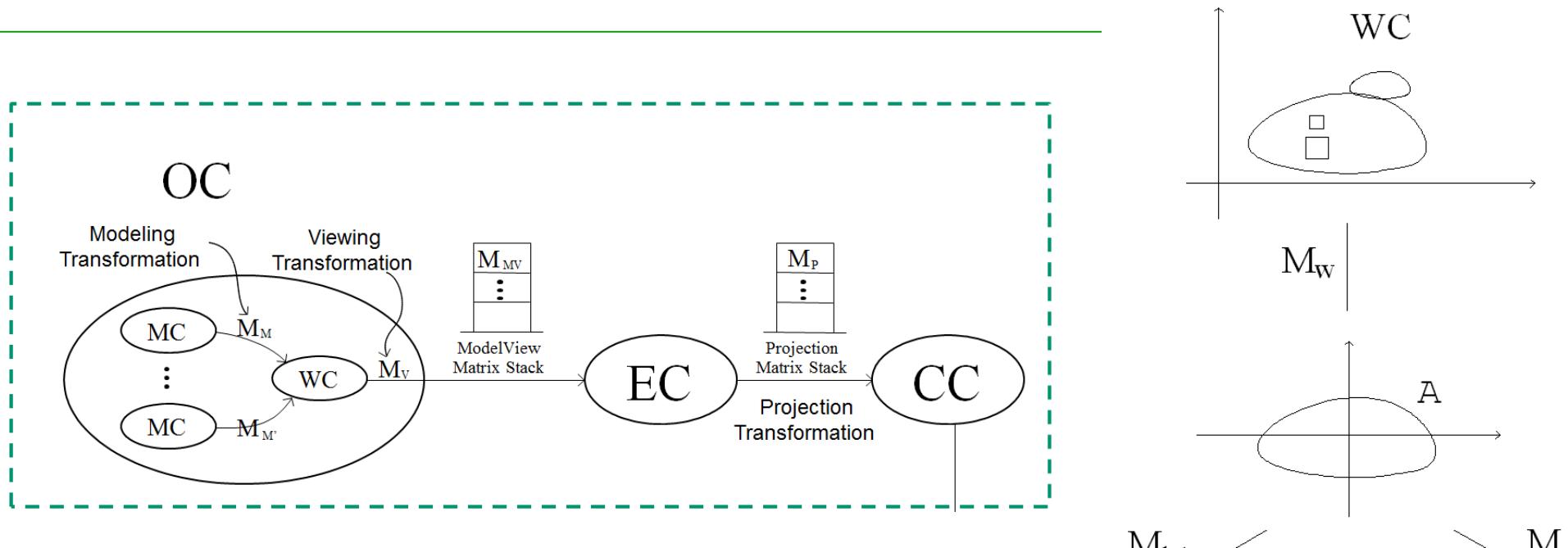


$$\begin{aligned}
 & M_V \cdot R(\text{angle}, 0, 1, 0) \cdot T(0, 0, 5) \cdot S(6.5, 6.5, 6.5) \cdot R(10 \cdot \text{angle}, 1, 0, 0) \cdot T(-0.1, 0, 0.3) \\
 & \cdot S(0.025, 0.025, 0.025) \cdot R(40 \cdot \text{angle}, 1, 0, 0) \cdot T(0, 4, 0) \cdot S(0.4, 0.4, 0.4)
 \end{aligned}$$

```

ModelMatrix_big_cow = glm::rotate(glm::mat4(1.0f), rotation_angle_cow,
                                  glm::vec3(0.0f, 1.0f, 0.0f));
ModelMatrix_big_cow = glm::translate(ModelMatrix_big_cow, glm::vec3(0.0f, 0.0f, 5.0f));
ModelMatrix_big_cow = glm::scale(ModelMatrix_big_cow, glm::vec3(6.5f, 6.5f, 6.5f));
ModelViewProjectionMatrix = ViewProjectionMatrix * ModelMatrix_big_cow;
glUniformMatrix4fv(loc_ModelViewProjectionMatrix, 1, GL_FALSE,
&ModelViewProjectionMatrix[0][0]);
glUniform3f(loc_primitive_color, 0.482f, 0.408f, 0.933f);
draw_geom_obj(GEOM_OBJ_ID_COW); // draw the bigger cow(A)

```

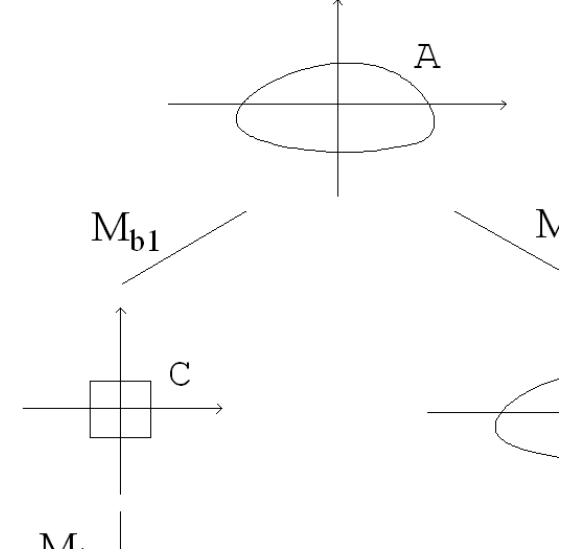


$$\begin{aligned}
 M_V \cdot R(angle, 0, 1, 0) \cdot T(0, 0, 5) \cdot S(6.5, 6.5, 6.5) \cdot R(10 \cdot angle, 1, 0, 0) \cdot T(-0.1, 0, 0.3) \\
 \cdot S(0.025, 0.025, 0.025) \cdot R(40 \cdot angle, 1, 0, 0) \cdot T(0, 4, 0) \cdot S(0.4, 0.4, 0.4)
 \end{aligned}$$

```

ModelMatrix_big_box = glm::rotate(ModelMatrix_big_cow, 10.0f*rotation_angle_cow,
                                  glm::vec3(1.0f, 0.0f, 0.0f));
ModelMatrix_big_box = glm::translate(ModelMatrix_big_box,
                                    glm::vec3(-0.1f, 0.0f, 0.3f));
ModelMatrix_big_box = glm::scale(ModelMatrix_big_box,
                                 glm::vec3(0.025f, 0.025f, 0.025f));
ModelViewProjectionMatrix = ViewProjectionMatrix * ModelMatrix_big_box;
glUniformMatrix4fv(loc_ModelViewProjectionMatrix, 1, GL_FALSE,
&ModelViewProjectionMatrix[0][0]);
glUniform3f(loc_primitive_color, 0.498f, 1.000f, 0.831f);
draw_geom_obj(GEOM_OBJ_ID_BOX); // draw the bigger box (C)
    
```

IVI_W

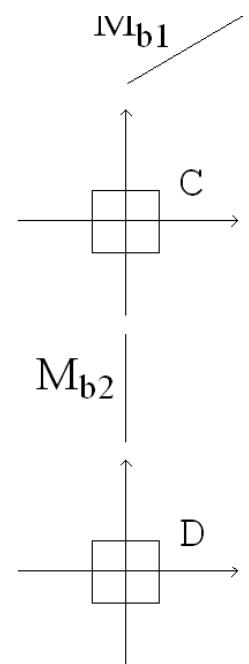
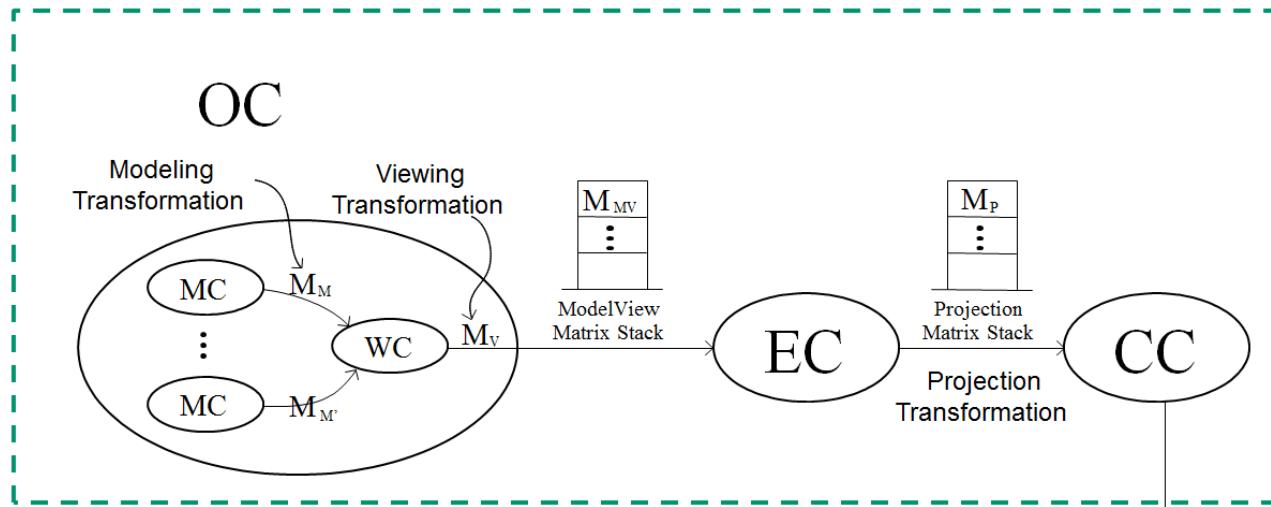


$$\begin{aligned}
 M_V \cdot R(angle, 0, 1, 0) \cdot T(0, 0, 5) \cdot S(6.5, 6.5, 6.5) \cdot R(10 \cdot angle, 1, 0, 0) \cdot T(-0.1, 0, 0.3) \\
 \cdot S(0.025, 0.025, 0.025) \cdot R(40 \cdot angle, 1, 0, 0) \cdot T(0, 4, 0) \cdot S(0.4, 0.4, 0.4)
 \end{aligned}$$

```

ModelMatrix_small_box = glm::rotate(ModelMatrix_big_box, 40.0f*rotation_angle_cow,
                                    glm::vec3(1.0f, 0.0f, 0.0f));
ModelMatrix_small_box = glm::translate(ModelMatrix_small_box,
                                       glm::vec3(0.0f, 4.0f, 0.0f));
ModelMatrix_small_box = glm::scale(ModelMatrix_small_box, glm::vec3(0.4f, 0.4f, 0.4f));
ModelViewProjectionMatrix = ViewProjectionMatrix * ModelMatrix_small_box;
glUniformMatrix4fv(loc_ModelViewProjectionMatrix, 1, GL_FALSE,
                   &ModelViewProjectionMatrix[0][0]);
glUniform3f(loc_primitive_color, 1.000f, 0.271f, 0.000f);
draw_geom_obj(GEOM_OBJ_ID_BOX); // draw the smaller box (D)

```

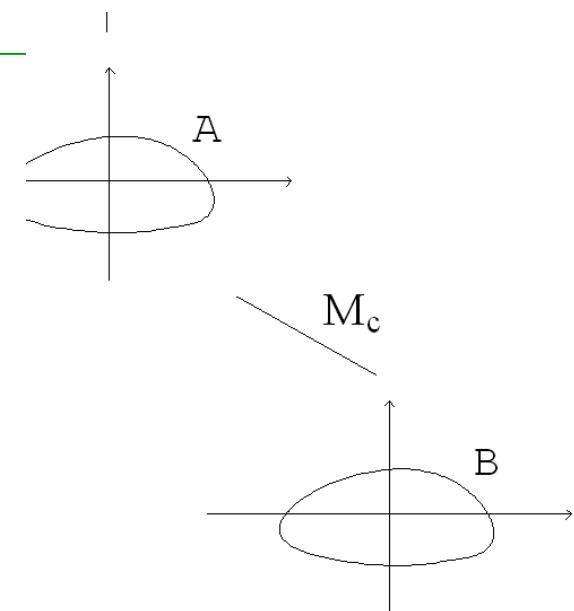
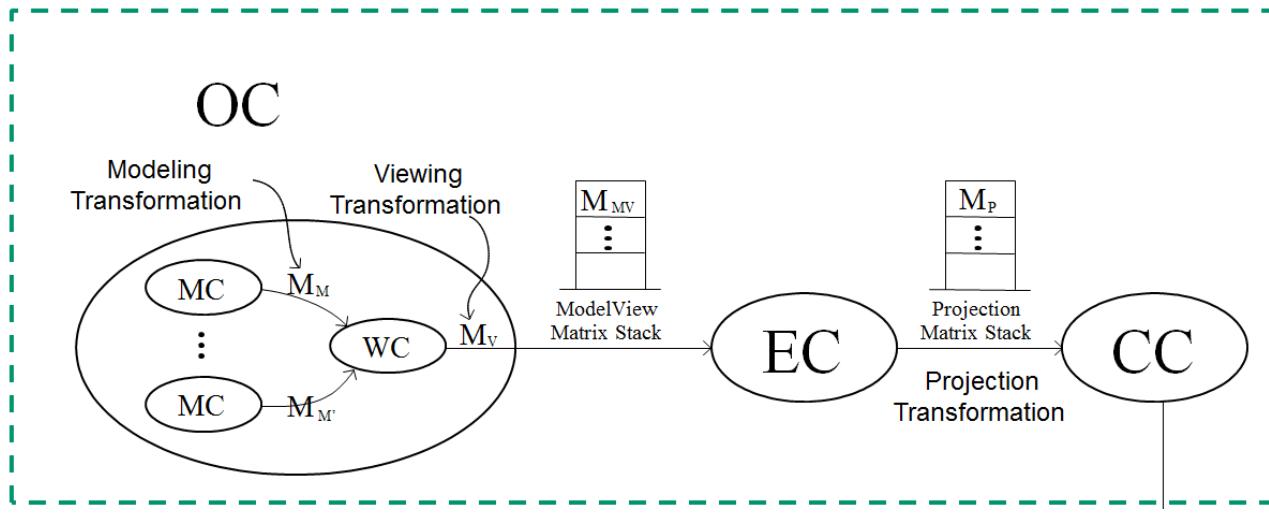


```

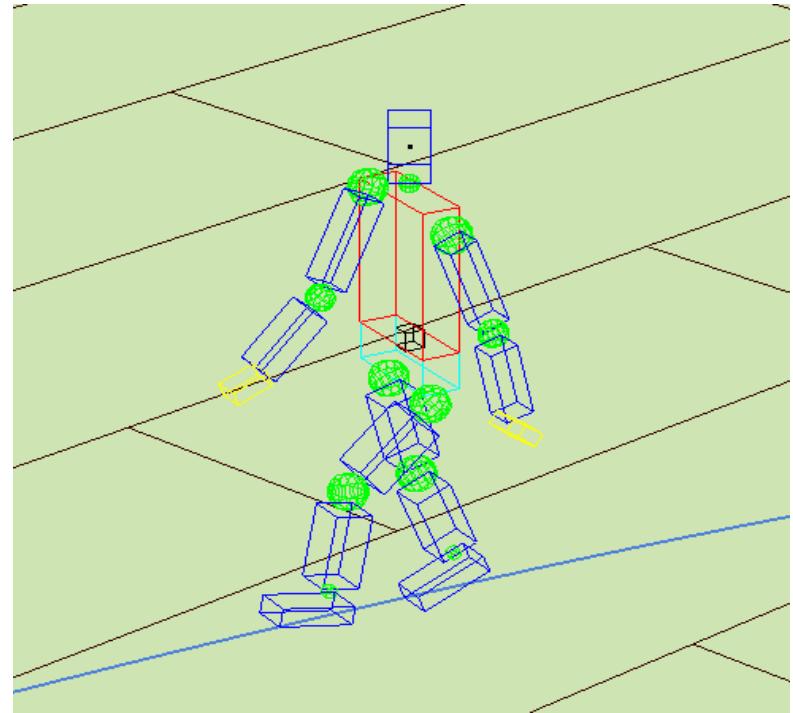
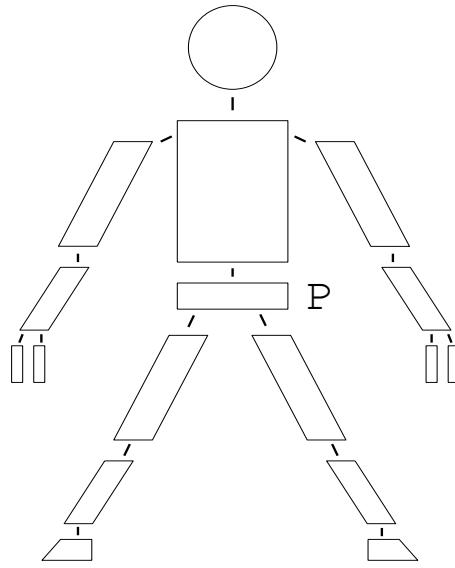
ModelMatrix_small_cow = glm::translate(ModelMatrix_big_cow,
                                         glm::vec3(0.15f, 0.3f, 0.0f));
ModelMatrix_small_cow = glm::scale(ModelMatrix_small_cow, glm::vec3(0.3f, 0.3f, 0.3f));
ModelViewProjectionMatrix = ViewProjectionMatrix * ModelMatrix_small_cow;
glUniformMatrix4fv(loc_ModelViewProjectionMatrix, 1, GL_FALSE,
&ModelViewProjectionMatrix[0][0]);
glUniform3f(loc_primitive_color, 0.867f, 0.627f, 0.867f);
draw_geom_obj(GEOM_OBJ_ID_COW); // draw the smaller cow (B)

glutSwapBuffers();
}

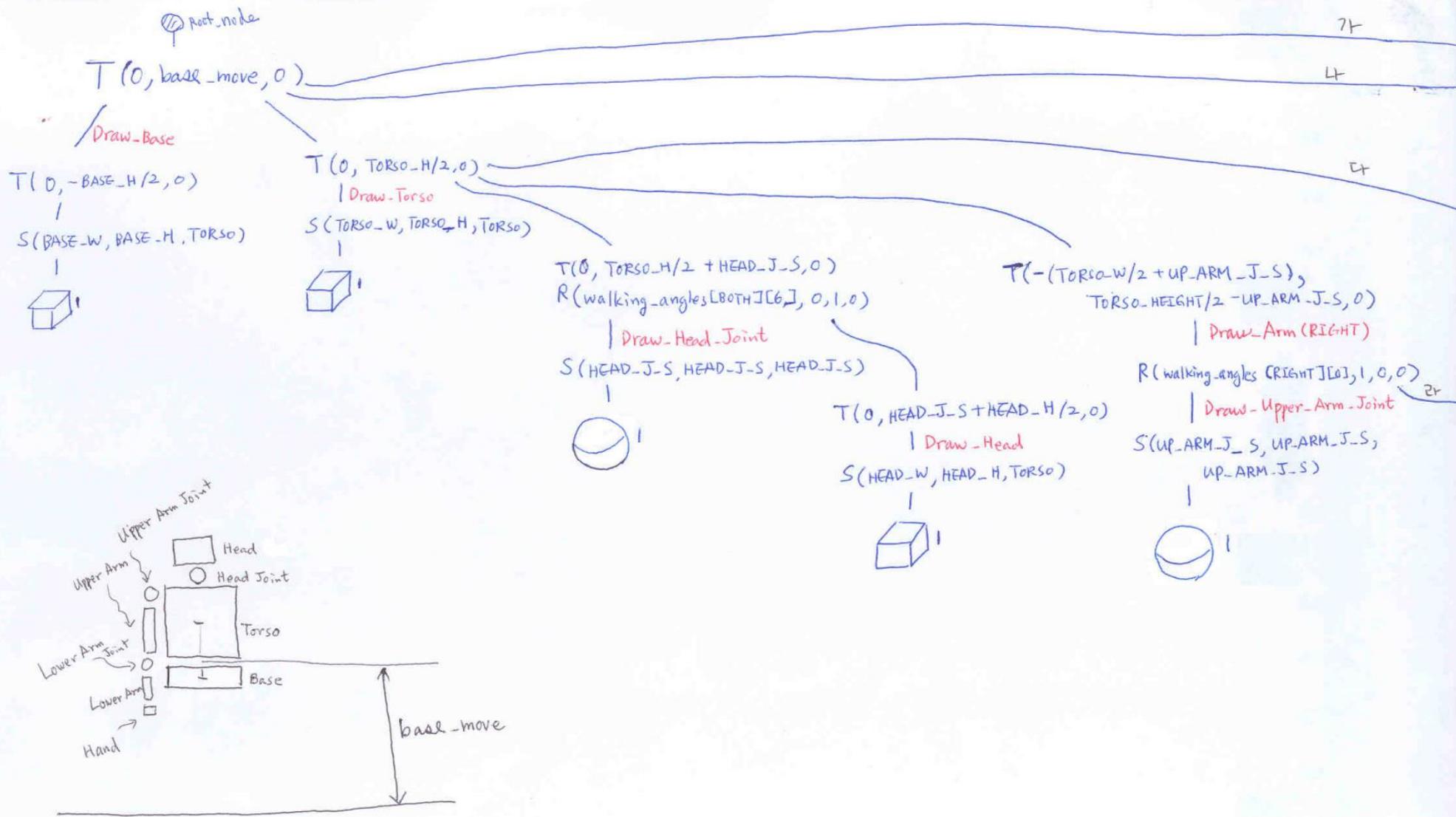
```

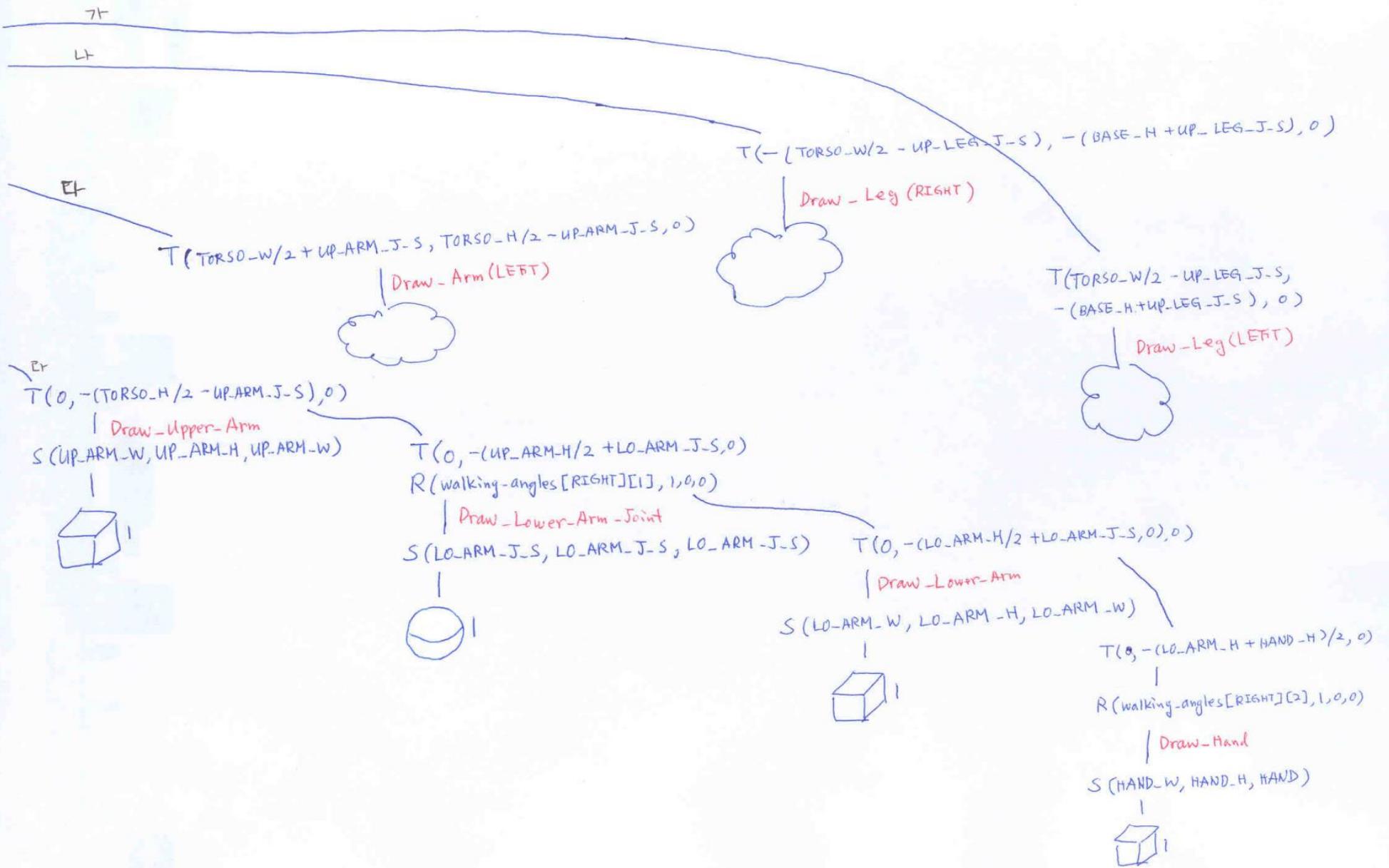


[예3] 계층적 구조를 가지는 인체 (Compatibility Profile)



Courtesy of The Developers Gallery: A 3D Case Study Using OpenGL by F. Chatzinitikos





계층적 모델링을 위한 코드 예

```
float walking_angles[2][6]; double base_move;
```

```
#define FOOT_JOINT_SIZE  
#define FOOT_HEIGHT  
#define FOOT_WIDTH  
#define FOOT  
#define UP_ARM_HEIGHT  
#define UP_ARM_WIDTH  
#define UP_ARM_JOINT_SIZE  
#define LO_ARM_HEIGHT  
#define LO_ARM_WIDTH  
#define LO_ARM_JOINT_SIZE  
#define HAND_HEIGHT  
#define HAND_WIDTH  
#define HAND  
#define FINGER_SIZE  
#define TORSO_WIDTH  
#define TORSO_HEIGHT  
#define TORSO  
#define HEAD_WIDTH  
#define HEAD_HEIGHT  
#define HEAD_JOINT_SIZE  
#define BASE_WIDTH  
#define BASE_HEIGHT  
#define UP_LEG_HEIGHT  
#define UP_LEG_JOINT_SIZE  
#define UP_LEG_WIDTH  
#define LO_LEG_HEIGHT  
#define LO_LEG_WIDTH  
#define LO_LEG_JOINT_SIZE  
#define LEG_HEIGHT  
                  HEAD_JOINT_SIZE  
                  FOOT_JOINT_SIZE * 2.0  
                  LO_LEG_WIDTH  
                  FOOT_WIDTH * 2.0  
                  TORSO_HEIGHT * 0.625  
                  TORSO_WIDTH/4.0  
                  HEAD_JOINT_SIZE * 2.0  
                  TORSO_HEIGHT * 0.5  
                  UP_ARM_WIDTH  
                  UP_ARM_JOINT_SIZE * 0.75  
                  LO_ARM_HEIGHT / 2.0  
                  LO_ARM_WIDTH  
                  LO_ARM_WIDTH / 2.0  
                  0.1  
                  TORSO_HEIGHT * 0.75  
                  0.8  
                  TORSO_WIDTH / 3.0  
                  HEAD_HEIGHT * 0.93  
                  TORSO_HEIGHT * 0.375  
                  HEAD_HEIGHT/6  
                  TORSO_WIDTH  
                  TORSO_HEIGHT / 4.0  
                  LO_ARM_HEIGHT  
                  UP_ARM_JOINT_SIZE  
                  UP_LEG_JOINT_SIZE * 2.0  
                  UP_LEG_HEIGHT  
                  UP_LEG_WIDTH  
                  UP_LEG_JOINT_SIZE  
                  UP_LEG_HEIGHT + LO_LEG_HEIGHT + FOOT_HEIGHT \\  
                  + 2*(FOOT_JOINT_SIZE+UP_LEG_JOINT_SIZE+LO_LEG_JOINT_SIZE)
```

```
void display(void) {
    GLfloat m[16], m_inv[16];

    glClear(GL_COLOR_BUFFER_BIT);

    set_up_rot_mat(m, m_inv, cur_i);
    if (view_from_head_cam)
        set_head_camera(m_inv);
    else set_world_camera();

    draw_world();

    glPushMatrix();
    glTranslatef(path[cur_i][0], 0.0, path[cur_i][2]);
    glMultMatrixf(m);

    Draw_Model();
    glPopMatrix();

    glutSwapBuffers();
}
```

animate_body() → display() → animate_body() → display() → ...

```

void Draw_Model(void) {
    glPushMatrix(); // Top to Base
    glTranslatef(0.0, base_move, 0.0);
    draw_CWireCube(0,0,0,0.1);
    Draw_Base();

    glPushMatrix(); // Base to Torso
    glTranslatef(0.0, TORSO_HEIGHT/2.0, 0.0);
    Draw_Torso();

    glPushMatrix(); // Torso to Head Joint
    glTranslatef(0.0, TORSO_HEIGHT/2.0 + HEAD_JOINT_SIZE, 0.0);
    glRotatef(walking_angles[BOTH][6], 0.0, 1.0, 0.0);
    Draw_Head_Joint();

    glPushMatrix(); // Head Joint to Head
    glTranslatef(0.0, HEAD_JOINT_SIZE + HEAD_HEIGHT/2.0, 0.0);
    Draw_Head();
    glPopMatrix(); // Back to Head Joint from Head
    glPopMatrix(); // Back to Torso from Head Joint

    glPushMatrix(); // Torso to Upper (Right) Upper Arm Joint
    glTranslatef(-(TORSO_WIDTH/2.0+UP_ARM_JOINT_SIZE),
                TORSO_HEIGHT/2.0 - UP_ARM_JOINT_SIZE, 0.0);
    Draw_Arm(RIGHT);
    glPopMatrix(); // Back to Torso from Upper Arm Joint
}

```

```
glPushMatrix(); // Torso to Upper (Left) Upper Arm Joint
    glTranslatef(TORSO_WIDTH/2.0+UP_ARM_JOINT_SIZE,
                TORSO_HEIGHT/2.0-UP_ARM_JOINT_SIZE, 0.0);
    Draw_Arm(LEFT);
glPopMatrix(); // Back to Torso from Upper Arm Joint
glPopMatrix(); // Back to Base from Torso

glPushMatrix(); // Base to (Right) Upper Leg Joint
    glTranslatef(-(TORSO_WIDTH/2.0 - UP_LEG_JOINT_SIZE),
                -(BASE_HEIGHT+UP_LEG_JOINT_SIZE), 0.0);
    Draw_Leg(RIGHT);
glPopMatrix(); // Back to Base from Upper Leg Joint

glPushMatrix(); // Base to (Left) Upper Leg Joint
    glTranslatef(TORSO_WIDTH/2.0 - UP_LEG_JOINT_SIZE,
                -(BASE_HEIGHT+UP_LEG_JOINT_SIZE), 0.0);
    Draw_Leg(LEFT);
glPopMatrix(); // Back to Base from Upper Leg Joint
glPopMatrix(); // Back to Top from Base
}
```

```

void draw_CWireCube(GLfloat r, GLfloat g,
                    GLfloat b, GLdouble size) {
    glColor3f(r, g, b);
    glutWireCube(size);
}

void Draw_Base(void) {
    glPushMatrix();
    glTranslatef(0.0, -BASE_HEIGHT/2.0, 0.0);
    glPushMatrix();
    glScalef(BASE_WIDTH, BASE_HEIGHT, TORSO);
    draw_CWireCube(0.0, 1.0, 1.0, 1.0);
    glPopMatrix();
    glPopMatrix();
}

void Draw_Torso(void) {
    glPushMatrix();
    glScalef(TORSO_WIDTH, TORSO_HEIGHT, TORSO);
    draw_CWireCube(1.0, 0.0, 0.0, 1.0);
    glPopMatrix();
}

void Draw_Head_Joint(void) {
    glPushMatrix();
    glScalef(HEAD_JOINT_SIZE, HEAD_JOINT_SIZE,
             HEAD_JOINT_SIZE);
    draw_CWireSphere(0.0, 1.0, 0.0, 1.0);
    glPopMatrix();
}

```

```

void Draw_Head(void) {
    glPushMatrix();
    glScalef(HEAD_WIDTH, HEAD_HEIGHT, TORSO);
    draw_CWireCube(0.0, 0.0, 1.0, 1.0);
    glPopMatrix();
}

void Draw_Upper_Arm_Joint(void) {
    glPushMatrix();
    glScalef(UP_ARM_JOINT_SIZE,
             UP_ARM_JOINT_SIZE, UP_ARM_JOINT_SIZE);
    draw_CWireSphere(0.0, 1.0, 0.0, 1.0);
    glPopMatrix();
}

void Draw_Upper_Arm(void) {
    glPushMatrix();
    glScalef(UP_ARM_WIDTH, UP_ARM_HEIGHT,
             UP_ARM_WIDTH);
    draw_CWireCube(0.0, 0.0, 1.0, 1.0);
    glPopMatrix();
}

void Draw_Lower_Arm_Joint(void) {
    glPushMatrix();
    glScalef(LO_ARM_JOINT_SIZE,
             LO_ARM_JOINT_SIZE, LO_ARM_JOINT_SIZE);
    draw_CWireSphere(0.0, 1.0, 0.0, 1.0);
    glPopMatrix();
}

void Draw_Lower_Arm(void) {
    glPushMatrix();
    glScalef(LO_ARM_WIDTH, LO_ARM_HEIGHT,
             LO_ARM_WIDTH);
    draw_CWireCube(0.0, 0.0, 1.0, 1.0);
    glPopMatrix();
}

```

```

void Draw_Arm(int side) {
    glRotatef(walking_angles[side][0],
              1.0, 0.0, 0.0);
    Draw_Upper_Arm_Joint();

    glPushMatrix(); // Upper Arm Joint
                    to Upper Arm
    glTranslatef(0.0, -(TORSO_HEIGHT/2.0-
                      UP_ARM_JOINT_SIZE), 0.0);
    Draw_Upper_Arm();

    glPushMatrix(); // Upper Arm
                    to Lower Arm Joint
    glTranslatef(0.0, -(UP_ARM_HEIGHT/2.0 +
                      LO_ARM_JOINT_SIZE), 0.0);
    glRotatef(walking_angles[side][1],
              1.0, 0.0, 0.0);
    Draw_Lower_Arm_Joint();

    glPushMatrix(); // Lower Arm Joint
                    to Lower Arm
    glTranslatef(0.0, -(LO_ARM_HEIGHT/2.0 +
                      LO_ARM_JOINT_SIZE), 0.0);
    Draw_Lower_Arm();

    glPushMatrix(); // Lower Arm to Hand
    glTranslatef(0.0, -(LO_ARM_HEIGHT +
                      HAND_HEIGHT)/2.0, 0.0);
    glRotatef(walking_angles[side][2],
              1.0, 0.0, 0.0);
    Draw_Hand();
    glPopMatrix(); // Back to Lower Arm
                   from Hand
}

```

```

glPopMatrix(); // Back to Lower Arm Joint
               from Lower Arm
glPopMatrix(); // Back to Upper Arm
               from Lower Arm Joint
glPopMatrix(); // Back to Upper Arm Joint
               from Upper Arm
}

void Draw_Hand(void) {
    glPushMatrix();
    glScalef(HAND_WIDTH, HAND_HEIGHT, HAND);
    draw_CWireCube(1.0, 1.0, 0.0, 1.0);
    glPopMatrix();
}

float walking_angles[2][7];
// various angles for the current frame

void display(void) {
    ...
    glPushMatrix();
    glTranslatef(path[cur_i][0], 0.0,
                path[cur_i][2]);
    glMultMatrixf(m);
    Draw_Model();
    glPopMatrix();
    ...
}

```

어떻게 자연스러운 움직임을 생성할 것인가?

- 컴퓨터 애니메이션 분야의 중요 연구 문제
 - 수작업
 - 간단한 프로그램
 - Key -frame 기법
 - 정역학 및 동역학 이론 적용
 - Motion capture 기법 적용
 - ...

- 이 예제에서는 간단한 코드를 통하여 움직임 생성

- Read data for `head`, `upbody`, `lobody`, `l_uparm`, `l_loarm`, `l_hand`, `l_upleg`, `l_loleg`, `l_foot`, `r_uparm`, `r_loarm`, `r_hand`, `r_upleg`, `r_loleg`, `r_foot` from `DATA.TXT` into `init_angle`, `angles[4]` in `Read_Data_From_File()` of `INOUT.C` called by `init_program()` of `MAIN.C`.

`DATA.TXT`

```
0.0 0.0 0.0 -40.0 -8.0 0.0 30.0 0.0 0.0 40.0 -8.0 50.0 -30.0 0.0 0.0
0.0 0.0 0.0 20.0 -17.0 -30.0 15.0 15.0 0.0 -25.0 0.0 10.0 5.0 5.0 0.0
0.0 0.0 0.0 20.0 -15.0 80.0 20.0 -15.0 0.0 -15.0 -32.0 -10.0 -65.0 85.0 0.0
0.0 0.0 0.0 15.0 32.0 -10.0 5.0 10.0 0.0 -20.0 15.0 -80.0 -25.0 -40.0 0.0
0.0 0.0 0.0 25.0 0.0 10.0 20.0 -10.0 0.0 -20.0 17.0 30.0 25.0 -50.0 0.0
```

```
anim_angles init_angles, angles[4];
```

```
typedef struct {
    float head;
    float upbody;
    float lobody;
    float l_uparm;
    float l_loarm;
    float l_hand;
    float l_upleg;
    float l_loleg;
    float l_foot;
    float r_uparm;
    float r_loarm;
    float r_hand;
    float r_upleg;
    float r_loleg;
    float r_foot;
} anim_angles;
```

- In `animate_body()` of **ANIM.C**, which is an idle callback function, update `walking_angles[2][6]` for the next animation frame.

```

void animate_body() {
    static frames = FRAMES, zoom_f1 = 0, flag = 1;
    float l_upleg_add, l_loleg_add, r_upleg_add, r_loleg_add, l_uparm_add,
          l_loarm_add, l_hand_add, r_uparm_add, r_loarm_add, r_hand_add;

    switch (flag) {
        case 1:
            l_upleg_add = angles[0].l_upleg / FRAMES; r_upleg_add = angles[0].r_upleg / FRAMES;
            l_loleg_add = angles[0].l_loleg / FRAMES; r_loleg_add = angles[0].r_loleg / FRAMES;
            l_uparm_add = angles[0].l_uparm / FRAMES; l_loarm_add = angles[0].l_loarm / FRAMES;
            l_hand_add = angles[0].l_hand / FRAMES; r_uparm_add = angles[0].r_uparm / FRAMES;
            r_hand_add = angles[0].r_hand / FRAMES; r_loarm_add = angles[0].r_loarm / FRAMES;

            walking_angles[LEFT][3] += r_upleg_add; walking_angles[RIGHT][3] += l_upleg_add;
            walking_angles[LEFT][4] += r_loleg_add; walking_angles[RIGHT][4] += l_loleg_add;
            walking_angles[LEFT][0] += l_uparm_add; walking_angles[LEFT][1] += l_loarm_add;
            walking_angles[LEFT][2] += l_hand_add; walking_angles[RIGHT][0] += r_uparm_add;
            walking_angles[RIGHT][1] += r_loarm_add; walking_angles[RIGHT][2] += r_hand_add;

            langle_count --= l_upleg_add; langle_count2 -= l_loleg_add;
            rangle_count --= r_upleg_add; rangle_count2 -= r_loleg_add;
            base_move = find_base_move(langle_count, langle_count2, rangle_count, rangle_count2);
            frames--;
            if (frames == 0) { flag = 2; frames = FRAMES; }
        break;
    ...
}

```

- 1 움직임 사이클 = 8개의 부분 움직임
 - 각 부분 움직임 = 각 부분은 FRAME(여기서는 20) 조각

- 아래 함수의 역할은?

```
double find_base_move(double langle_up, double langle_lo, double rangle_up,
                      double rangle_lo) {
    double result1, result2, first_result, second_result, radians_up, radians_lo;

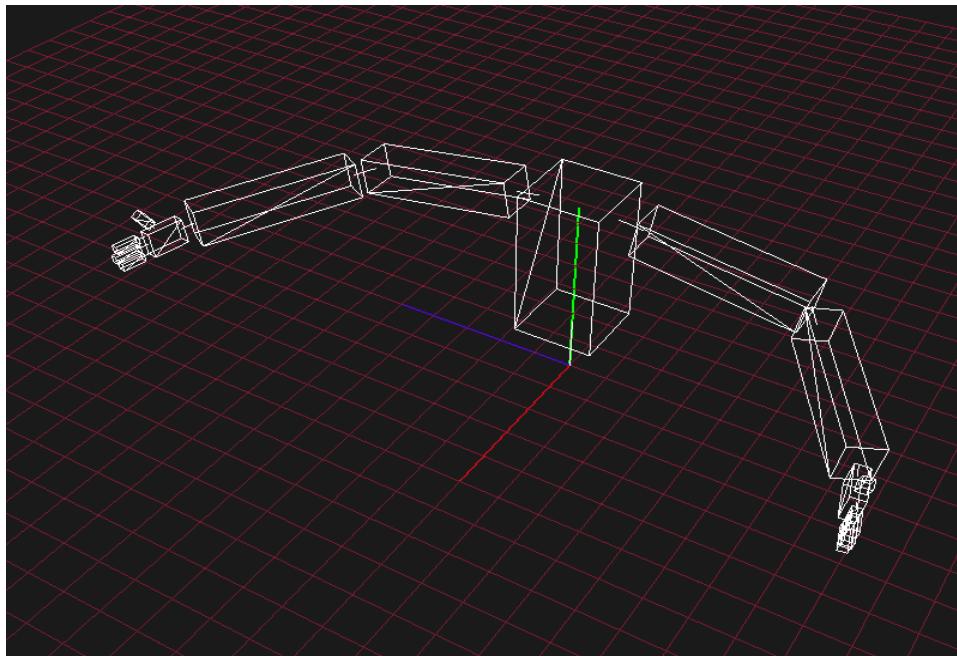
    radians_up = (PI*langle_up)/180.0;
    radians_lo = (PI*langle_lo-langle_up)/180.0;
    result1 = (UP_LEG_HEIGHT + 2*UP_LEG_JOINT_SIZE) * cos(radians_up);
    result2 = (LO_LEG_HEIGHT + 2*(LO_LEG_JOINT_SIZE+FOOT_JOINT_SIZE)+FOOT_HEIGHT)
              * cos(radians_lo);
    first_result = LEG_HEIGHT - (result1 + result2);

    radians_up = (PI*rangle_up)/180.0;
    radians_lo = (PI*rangle_lo-rangle_up)/180.0;
    result1 = (UP_LEG_HEIGHT + 2*UP_LEG_JOINT_SIZE) * cos(radians_up);
    result2 = (LO_LEG_HEIGHT + 2*(LO_LEG_JOINT_SIZE+FOOT_JOINT_SIZE)+FOOT_HEIGHT)
              * cos(radians_lo);
    second_result = LEG_HEIGHT - (result1 + result2);

    if (first_result <= second_result)
        return (- first_result);
    else
        return (- second_result);
}
```

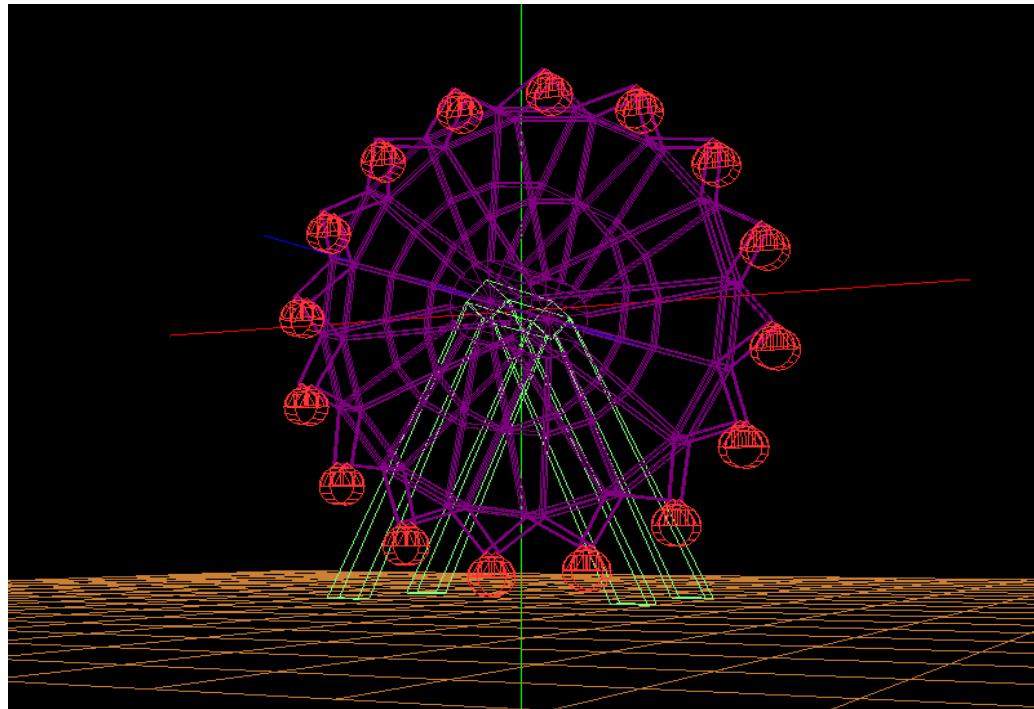
[예4] 로봇 팔의 계층적 구성

- 프로그래밍 숙제에서 구현하여 볼 것.



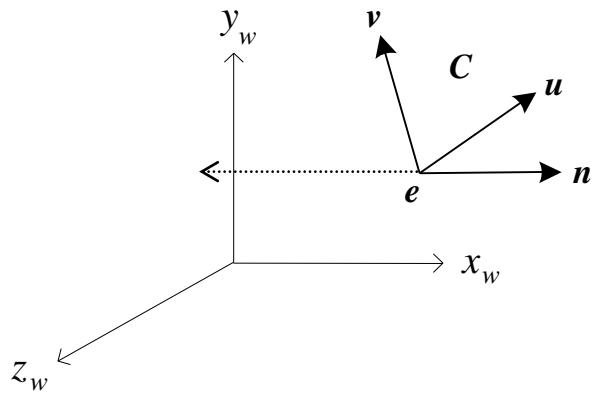
[예5] 놀이 기구의 구성

- 최소한의 기하 데이터만 가지고 아래와 같은 장면을 렌더링 하려면?



Camera Manipulation

세상 좌표계에서의 카메라의 표현



$$\mathbf{e} = (e_x, e_y, e_z)$$
$$\mathbf{u} = (u_x, u_y, u_z), \mathbf{v} = (v_x, v_y, v_z), \mathbf{n} = (n_x, n_y, n_z)$$

$$R = \begin{bmatrix} u_x & u_y & u_z & 0 \\ v_x & v_y & v_z & 0 \\ n_x & n_y & n_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

/ Viewing Transformation */*

```
glMultMatrixf(R);
```

```
glTranslatef(-ex, -ey, -ez);
```

$$M_v = R \cdot T(-e_x, -e_y, -e_z)$$

- ☺ 카메라의 위치와 방향에 대한 정보를 위의 \mathbf{C} 와 같이 세상 공간에서 이동 변환과 회전 변환을 통하여 자유롭게 움직이는 하나의 프레임으로 표현할 수 있음.

Camera 관련 코드 예

$$\mathbf{e} = (e_x, e_y, e_z)$$

$$\mathbf{u} = (u_x, u_y, u_z), \mathbf{v} = (v_x, v_y, v_z), \mathbf{n} = (n_x, n_y, n_z)$$

$$R = \begin{bmatrix} u_x & u_y & u_z & 0 \\ v_x & v_y & v_z & 0 \\ n_x & n_y & n_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$M_v = R \cdot T(-e_x, -e_y, -e_z)$$

```
typedef struct _Camera {
    float pos[3];
    float uaxis[3], vaxis[3], naxis[3];
    float fovy, aspect_ratio, near_c, far_c;
    int move;
} Camera;

Camera camera;

void set_ViewMatrix_from_camera_frame(void) {
    ViewMatrix = glm::mat4(camera.uaxis[0], camera.vaxis[0], camera.naxis[0], 0.0f,
                           camera.uaxis[1], camera.vaxis[1], camera.naxis[1], 0.0f,
                           camera.uaxis[2], camera.vaxis[2], camera.naxis[2], 0.0f,
                           0.0f, 0.0f, 0.0f, 1.0f);
    ViewMatrix = glm::translate(ViewMatrix, glm::vec3(-camera.pos[0], -camera.pos[1],
                                                       -camera.pos[2]));
}
```

Camera 관련 코드 예 (Core Profile)

```
void initialize_camera(void) {
    camera.pos[0] = 20.0f; camera.pos[1] = 1.0f;   camera.pos[2] = 2.0f;
    camera.uaxis[0] = camera.uaxis[1] = 0.0f; camera.uaxis[2] = -1.0f;
    camera.vaxis[0] = camera.vaxis[2] = 0.0f; camera.vaxis[1] = 1.0f;
    camera.naxis[1] = camera.naxis[2] = 0.0f; camera.naxis[0] = 1.0f;

    camera.move = 0;
    camera.fovy = 30.0f, camera.aspect_ratio = 1.0f; camera.near_c = 5.0f;
    camera.far_c = 10000.0f;

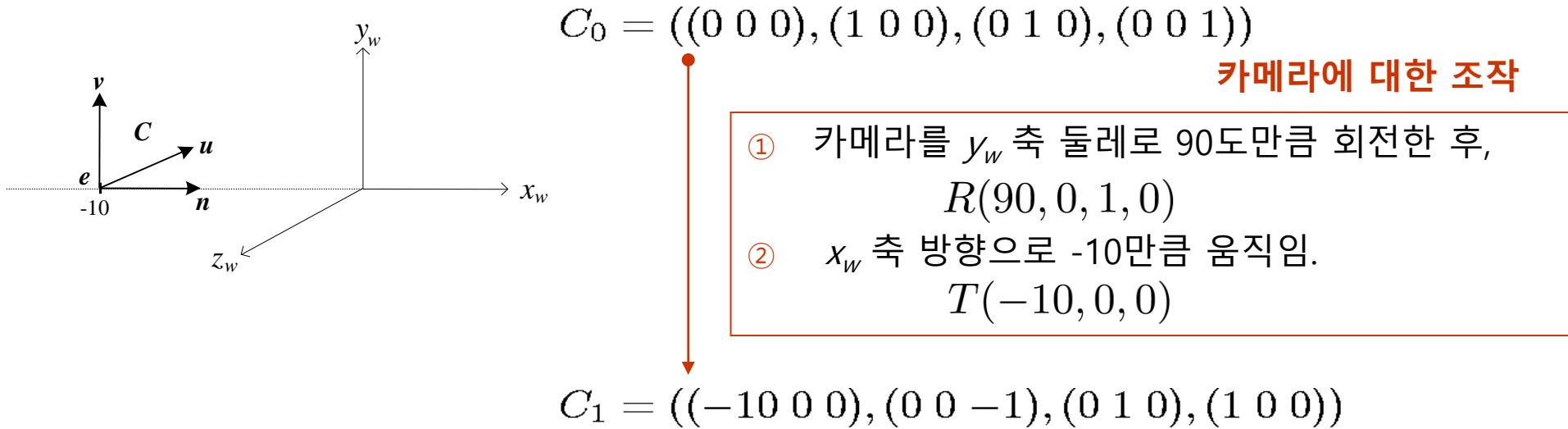
    set_ViewMatrix_from_camera_frame();
    ViewProjectionMatrix = ProjectionMatrix * ViewMatrix;
}

...

void reshape(int width, int height) {
    glViewport(0, 0, width, height);

    camera.aspect_ratio = (float)width / height;
    ProjectionMatrix = glm::perspective(TO_RADIANT*camera.fovy, camera.aspect_ratio,
                                         camera.near_c, camera.far_c);
    ViewProjectionMatrix = ProjectionMatrix * ViewMatrix;
    glutPostRedisplay();
}
```

카메라에 대한 조작과 뷰잉 변환간의 관계



뷰잉 변환

$$M_V = R(-90, 0, 1, 0) \cdot T(10, 0, 0) = R^{-1}(90, 0, 1, 0) \cdot T^{-1}(-10, 0, 0)$$

OpenGL 코드

```
glRotatef(-90.0, 0.0, 1.0, 0.0);  
glTranslatef(10.0, 0.0, 0.0);
```

카메라 움직임의 표현 (Compatibility Profile)

- ① 카메라의 초기 상태에서 시작하여 카메라에 대하여 다음과 같은 순서로 변환을 가하였다면,

$$M_1 \rightarrow M_2 \rightarrow \dots \rightarrow M_n$$

[교재 4.4 (247쪽)]

- ② 뷰잉 변환은 다음과 같고,

$$M_V = (M_n \cdot \dots \cdot M_2 \cdot M_1)^{-1} = M_1^{-1} \cdot M_2^{-1} \cdot \dots \cdot M_n^{-1}$$

- ③ 이를 구현하는 OpenGL 프로그램은 다음과 같은 형태를 가지게 된다.

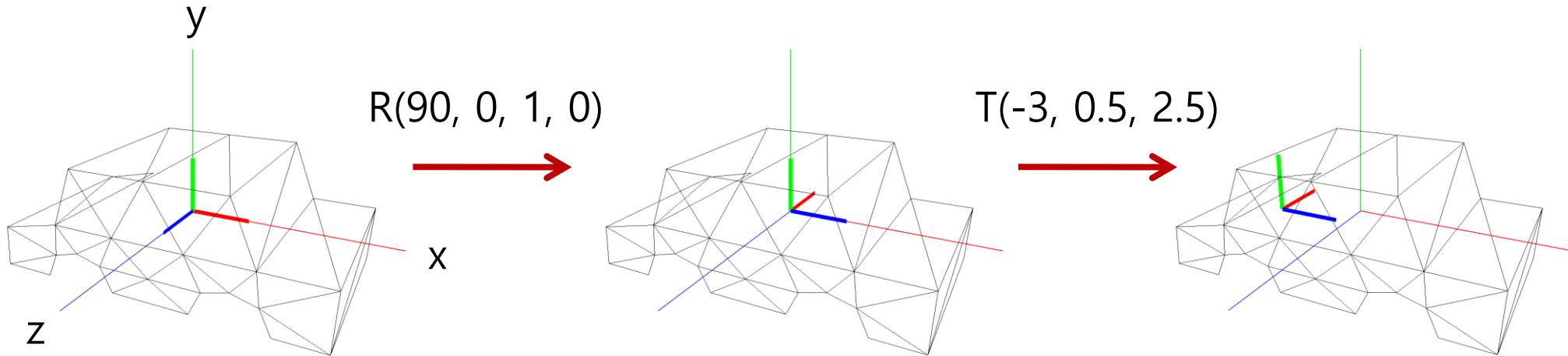
```
glMultMatrixf(M1-1);
```

```
glMultMatrixf(M2-1);
```

```
:
```

```
glMultMatrixf(Mn-1);
```

운전석 카메라 프레임 (e , (u , v , n))에 대한 조작

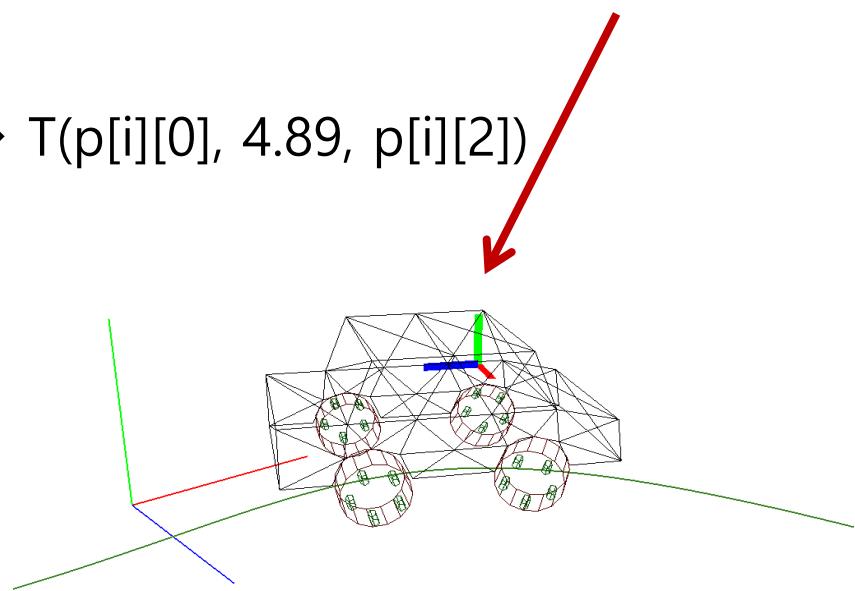


카메라 프레임을 body의 MC로 보내자!

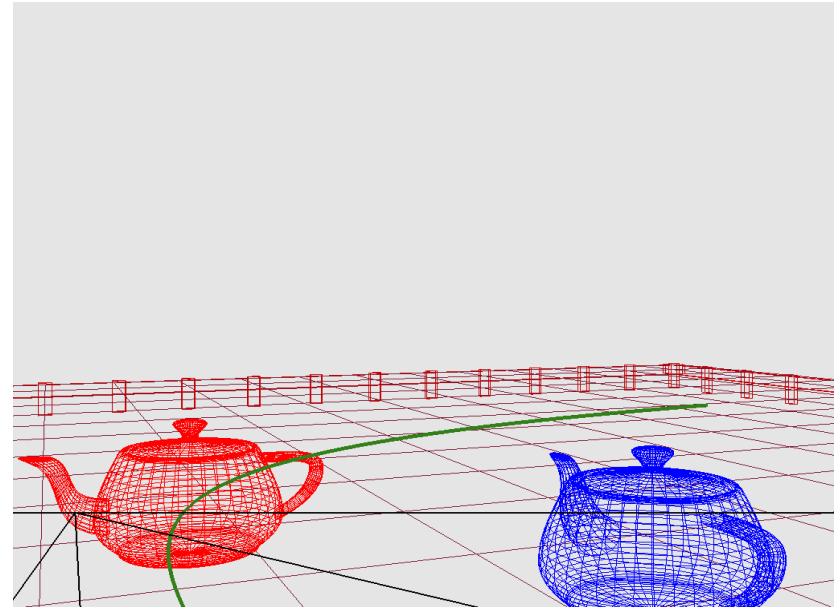
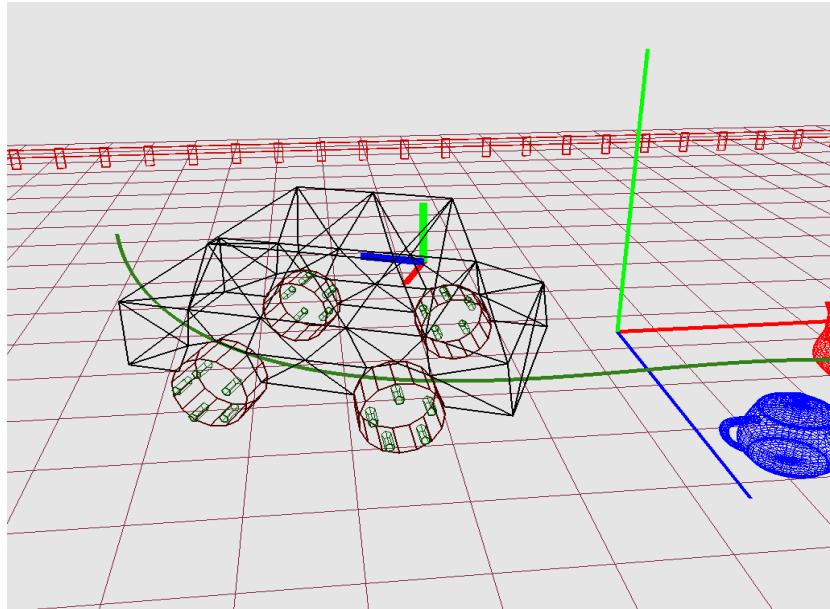
$M(R) \rightarrow T(p[i][0], 4.89, p[i][2])$

Viewing Transform

```
glRotatef(-90.0, 0.0, 1.0, 0.0);
glTranslatef(3.0, -0.5, -2.5);
glMultMatrixf(Minv);
glTranslatef(-p[i][0], -4.89, -p[i][2]);
```



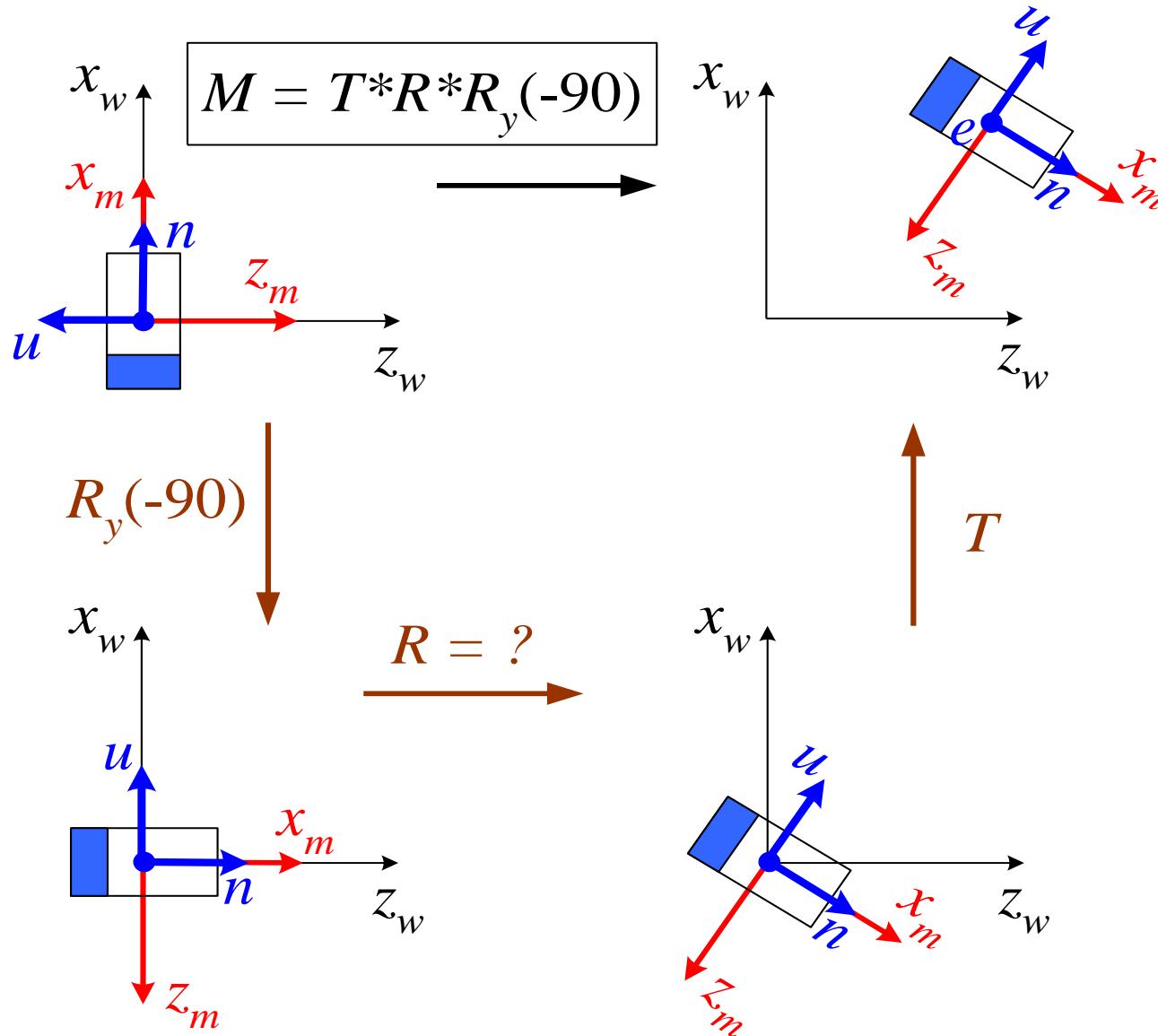
운전석에서 바라본 세상



Viewing Transform

```
glRotatef(-90.0, 0.0, 1.0, 0.0);  
glTranslatef(3.0, -0.5, -2.5);  
glMultMatrixf(Minv);  
glTranslatef(-p[i][0], -4.89, -p[i][2]);
```

자동차 기준 프레임 (e , (u , v , n))에 대한 모델링 변환



카메라 움직임의 표현 (Core Profile)

- ① 카메라의 초기 상태에서 시작하여 카메라에 대하여 다음과 같은 순서로 변환을 가하였다면,

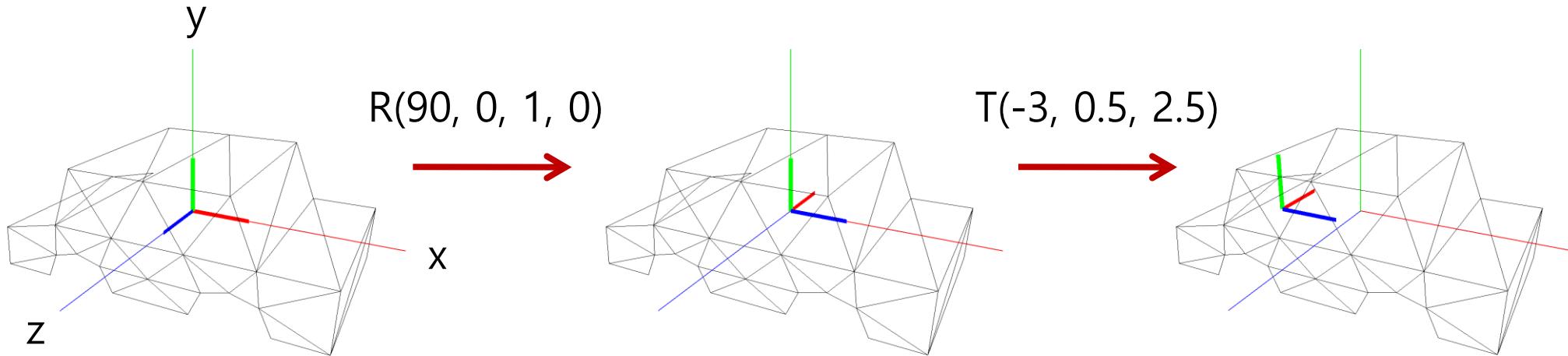
$$M_1 \rightarrow M_2 \rightarrow \dots \rightarrow M_n$$

- ② 뷰잉 변환은 다음과 같다.

$$M_V = (M_n \cdot \dots \cdot M_2 \cdot M_1)^{-1} = M_1^{-1} \cdot M_2^{-1} \cdot \dots \cdot M_n^{-1}$$

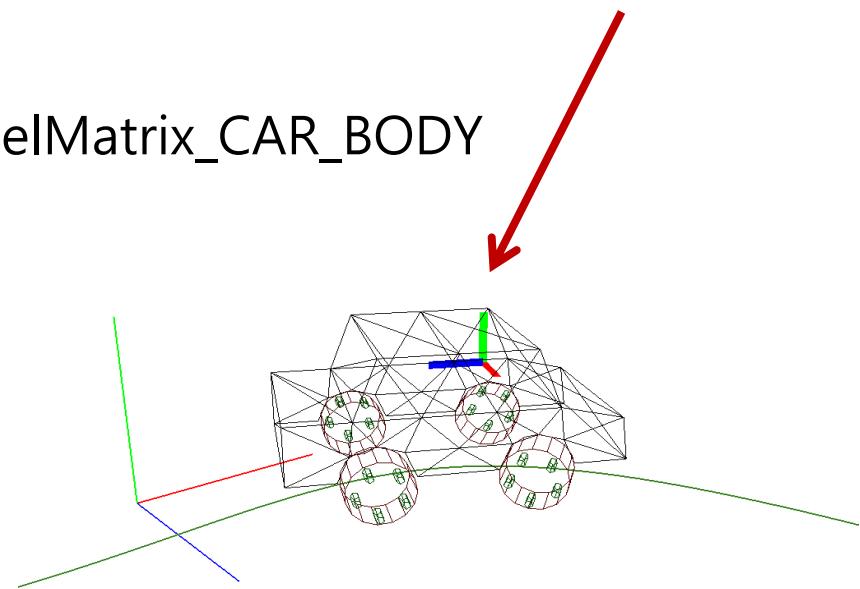
- ③ 이를 구현하는 OpenGL 프로그램은 다음 슬라이드에서 설명.

운전석 카메라 프레임 (e , (u , v , n))에 대한 조작



카메라 프레임을 body의 MC로 보내자!

ModelMatrix_CAR_BODY



```

glm::mat4 ModelMatrix_CAR_BODY, ModelMatrix_CAR_DRIVER, Matrix_CAMERA_driver_inverse;

...
ModelMatrix_CAR_BODY_to_DRIVER = glm::translate(glm::mat4(1.0f),
                                                glm::vec3(-3.0f, 0.5f, 2.5f));
ModelMatrix_CAR_BODY_to_DRIVER = glm::rotate(ModelMatrix_CAR_BODY_to_DRIVER,
                                              TO_RADIAN*90.0f, glm::vec3(0.0f, 1.0f, 0.0f));

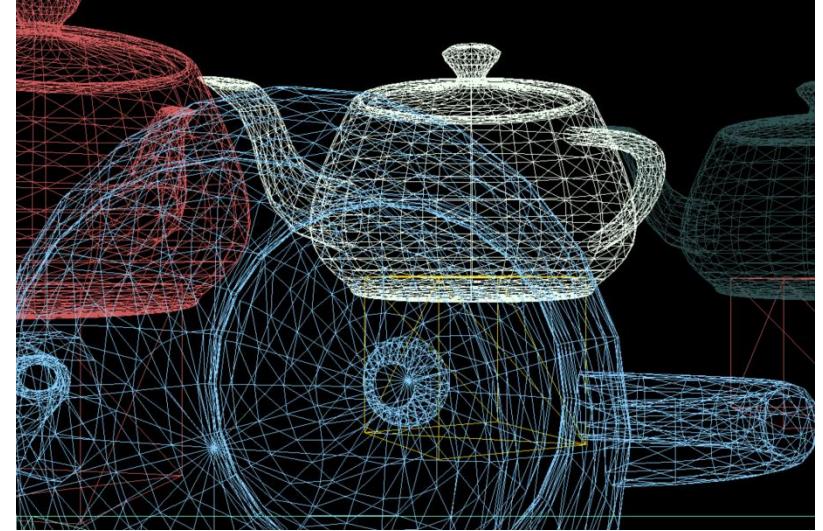
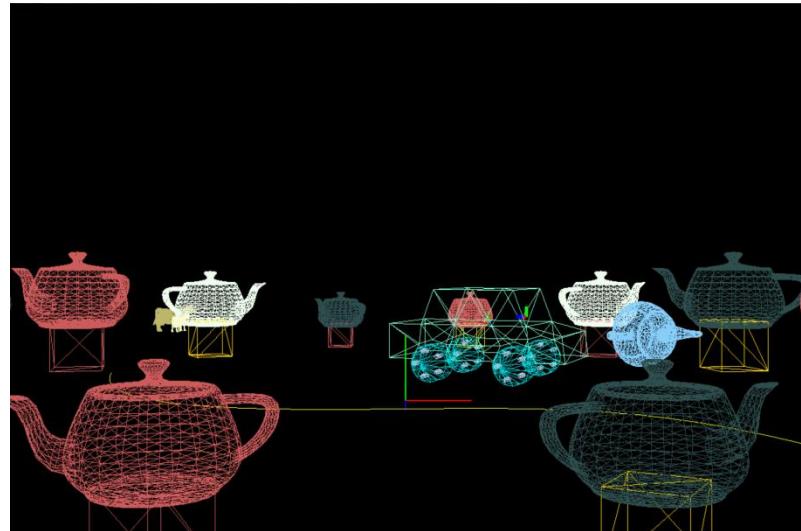
ModelMatrix_CAR_BODY = glm::rotate(glm::mat4(1.0f), -rotation_angle_car,
                                   glm::vec3(0.0f, 1.0f, 0.0f));
ModelMatrix_CAR_BODY = glm::translate(ModelMatrix_CAR_BODY,
                                      glm::vec3(20.0f, 4.89f, 0.0f));
ModelMatrix_CAR_BODY = glm::rotate(ModelMatrix_CAR_BODY, 90.0f*TO_RADIAN,
                                       glm::vec3(0.0f, 1.0f, 0.0f));

Matrix_CAMERA_driver_inverse = ModelMatrix_CAR_BODY *
                                    ModelMatrix_CAR_BODY_to_DRIVER;

ViewMatrix = glm::affineInverse(Matrix_CAMERA_driver_inverse);

ViewProjectionMatrix = ProjectionMatrix * ViewMatrix;

```

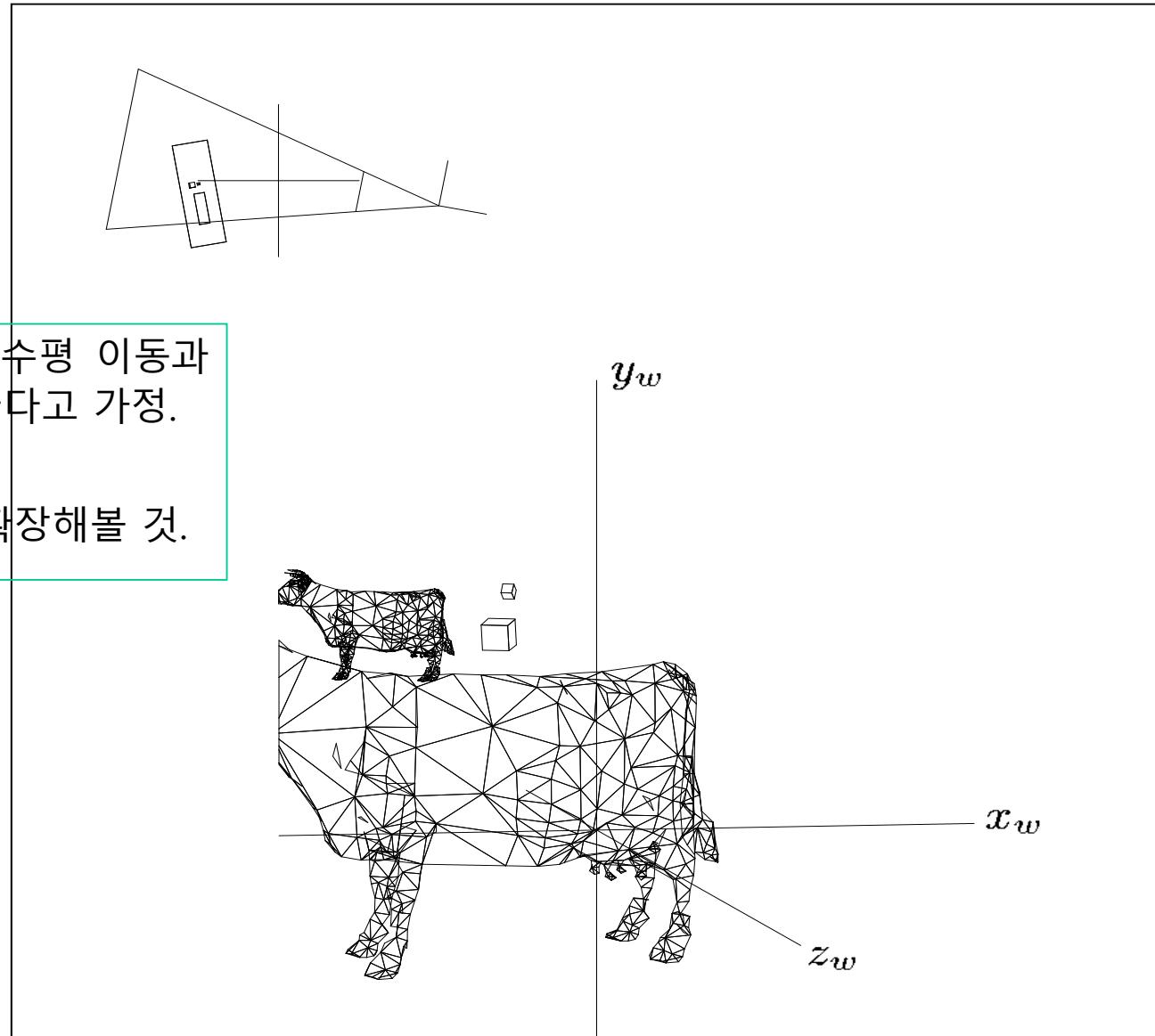


Viewing Transform

```
void set_ViewMatrix_for_driver(void) {  
    glm::mat4 Matrix_CAMERA_driver_inverse;  
  
    Matrix_CAMERA_driver_inverse = ModelMatrix_CAR_BODY *  
        ModelMatrix_CAR_BODY_to_DRIVER;  
  
    ViewMatrix = glm::affineInverse(Matrix_CAMERA_driver_inverse);  
    ViewProjectionMatrix = ProjectionMatrix * ViewMatrix;  
}
```

카메라에 대한 조작 예 1 (Compatibility Profile)

- ☺ 편의상 카메라는 전후 방향의 수평 이동과 카메라 v 축 둘레로의 회전만 있다고 가정.
- ☺ 카메라 움직임에 대한 기능을 확장해볼 것.



카메라의 정의

```
typedef struct _cam {
    float pos[3];
    float uaxis[3], vaxis[3], naxis[3];
    GLfloat mat[16];
    int move;
    GLdouble fovy, aspect, near_c, far_c;
} Cam;
Cam cam;

#define M(row,col) m[col*4+row] // C/C++ way
void set_rotate_mat(GLfloat *m) {
    M(0,0)=cam.uaxis[X]; M(0,1)=cam.uaxis[Y]; M(0,2)=cam.uaxis[Z]; M(0,3)=0.0;
    M(1,0)=cam.vaxis[X]; M(1,1)=cam.vaxis[Y]; M(1,2)=cam.vaxis[Z]; M(1,3)=0.0;
    M(2,0)=cam.naxis[X]; M(2,1)=cam.naxis[Y]; M(2,2)=cam.naxis[Z]; M(2,3)=0.0;
    M(3,0)=0.0;   M(3,1)=0.0;   M(3,2)=0.0;   M(3,3)=1.0;
}

void init_camera(void) {
    cam.pos[X]=20.0; cam.pos[Y]= 1.0;  cam.pos[Z] = 2.0;
    cam.uaxis[X] = cam.uaxis[Y] = 0.0; cam.uaxis[Z] = -1.0;
    cam.vaxis[X] = cam.vaxis[Z] = 0.0; cam.vaxis[Y] = 1.0;
    cam.naxis[Y] = cam.naxis[Z] = 0.0; cam.naxis[X] = 1.0;
    set_rotate_mat(cam.mat);
    cam.move = 0;
    cam.fovy = 30.0, cam.aspect = 1.0; cam.near_c = 5.0; cam.far_c = 20.0;
}
```

뷰잉 변환

```
void draw_scene(void) {
    glViewport(150, 0, 500, 500);

    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluPerspective(cam.fovy, cam.aspect, cam.near_c, cam.far_c);

    glMatrixMode(GL_MODELVIEW); // Viewing transformation
    glLoadIdentity();
    glMultMatrixf(cam.mat);
    glTranslatef(-cam.pos[X], -cam.pos[Y], -cam.pos[Z]);

    draw_axis();
    draw_world(SCENE);
}
```

마우스 입력에 따른 카메라 프레임의 수정

```
void motion(int x, int y) {
    int delx, dely;
    if (!cam.move) return;
    delx = prevx - x; dely = prevy - y;
    prevy = y; prevx = x;

    renew_cam_pos(dely);
    renew_cam_ori_y(delx);
    glutPostRedisplay();
}
```

```
#define CAM_TSPEED 0.05
void renew_cam_pos(int del) {
    cam.pos[X] += CAM_TSPEED*del*(-cam.naxis[X]);
    cam.pos[Y] += CAM_TSPEED*del*(-cam.naxis[Y]);
    cam.pos[Z] += CAM_TSPEED*del*(-cam.naxis[Z]);
}
```

```

#define TO_RADIAN 0.01745329
void get_rotation_mat(float angle, float m[3][3]) {
    // R(angle, 0.0, 1.0, 0.0)
    float rad_angle, sina, cosa;

    rad_angle = TO_RADIAN*angle;
    sina = sin(rad_angle); cosa = cos(rad_angle);

    m[0][0] = cosa;      m[0][1] = 0.0;      m[0][2] = sina;
    m[1][0] = 0.0;      m[1][1] = 1.0;      m[1][2] = 0.0;
    m[2][0] = -sina;    m[2][1] = 0.0;      m[2][2] = cosa;
}

#define CAM_RSPEED 0.1
void renew_cam_ori_y(int angle) {
    float m[3][3], tmpX, tmpY, tmpZ;

    get_rotation_mat(CAM_RSPEED*angle, m);

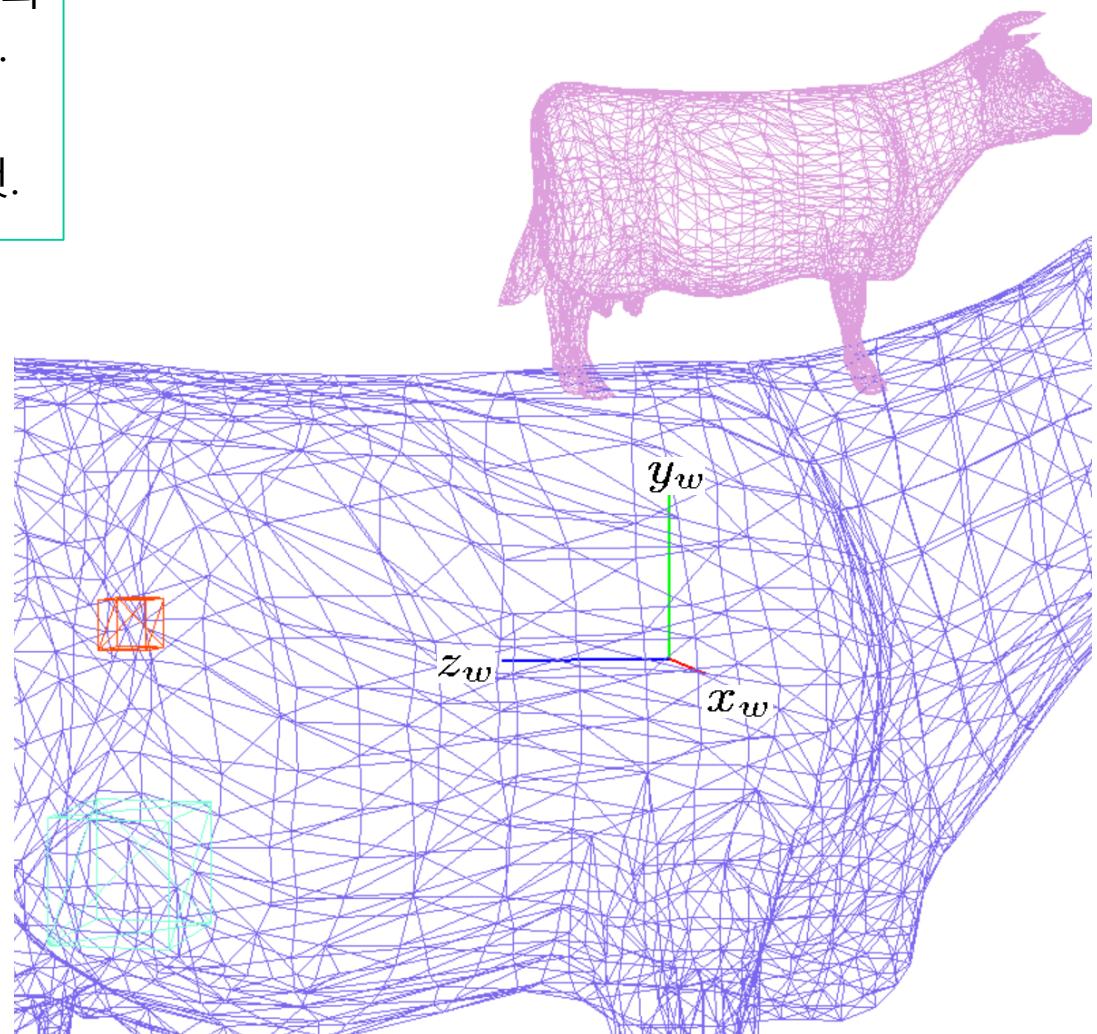
    cam.uaxis[X] = m[0][0]* (tmpX=cam.uaxis[X]) + m[0][1]* (tmpY=cam.uaxis[Y])
                  + m[0][2]* (tmpZ=cam.uaxis[Z]);
    cam.uaxis[Y] = m[1][0]*tmpX + m[1][1]*tmpY + m[1][2]*tmpZ;
    cam.uaxis[Z] = m[2][0]*tmpX + m[2][1]*tmpY + m[2][2]*tmpZ;
    cam.naxis[X] = m[0][0]* (tmpX=cam.naxis[X]) + m[0][1]* (tmpY=cam.naxis[Y])
                  + m[0][2]* (tmpZ=cam.naxis[Z]);
    cam.naxis[Y] = m[1][0]*tmpX + m[1][1]*tmpY + m[1][2]*tmpZ;
    cam.naxis[Z] = m[2][0]*tmpX + m[2][1]*tmpY + m[2][2]*tmpZ;

    set_rotate_mat(cam.mat);
}

```

카메라에 대한 조작 예 1 (Core Profile)

- ☺ 편의상 카메라는 전후 방향의 수평 이동과 카메라 v 축 둘레로의 회전만 있다고 가정.
- ☺ 카메라 움직임에 대한 기능을 확장해볼 것.



Camera의 정의

$$\mathbf{e} = (e_x, e_y, e_z)$$

$$\mathbf{u} = (u_x, u_y, u_z), \mathbf{v} = (v_x, v_y, v_z), \mathbf{n} = (n_x, n_y, n_z)$$

$$R = \begin{bmatrix} u_x & u_y & u_z & 0 \\ v_x & v_y & v_z & 0 \\ n_x & n_y & n_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$M_v = R \cdot T(-e_x, -e_y, -e_z)$$

```
typedef struct _Camera {
    float pos[3];
    float uaxis[3], vaxis[3], naxis[3];
    float fovy, aspect_ratio, near_c, far_c;
    int move;
} Camera;

Camera camera;

void set_ViewMatrix_from_camera_frame(void) {
    ViewMatrix = glm::mat4(camera.uaxis[0], camera.vaxis[0], camera.naxis[0], 0.0f,
                          camera.uaxis[1], camera.vaxis[1], camera.naxis[1], 0.0f,
                          camera.uaxis[2], camera.vaxis[2], camera.naxis[2], 0.0f,
                          0.0f, 0.0f, 0.0f, 1.0f);
    ViewMatrix = glm::translate(ViewMatrix, glm::vec3(-camera.pos[0], -camera.pos[1],
                                                     -camera.pos[2]));
}
```

Camera 관련 코드 예 (Core Profile)

```
void initialize_camera(void) {
    camera.pos[0] = 20.0f; camera.pos[1] = 1.0f;   camera.pos[2] = 2.0f;
    camera.uaxis[0] = camera.uaxis[1] = 0.0f; camera.uaxis[2] = -1.0f;
    camera.vaxis[0] = camera.vaxis[2] = 0.0f; camera.vaxis[1] = 1.0f;
    camera.naxis[1] = camera.naxis[2] = 0.0f; camera.naxis[0] = 1.0f;

    camera.move = 0;
    camera.fovy = 30.0f, camera.aspect_ratio = 1.0f; camera.near_c = 5.0f;
    camera.far_c = 10000.0f;

    set_ViewMatrix_from_camera_frame();
    ViewProjectionMatrix = ProjectionMatrix * ViewMatrix;
}

...
void reshape(int width, int height) {
    glViewport(0, 0, width, height);

    camera.aspect_ratio = (float)width / height;
    ProjectionMatrix = glm::perspective(TO_RADIAN*camera.fovy, camera.aspect_ratio,
                                         camera.near_c, camera.far_c);
    ViewProjectionMatrix = ProjectionMatrix * ViewMatrix;
    glutPostRedisplay();
}
```

Display Callback Function

```
void display(void) {
    glm::mat4 ModelMatrix_big_cow, ModelMatrix_small_cow;
    glm::mat4 ModelMatrix_big_box, ModelMatrix_small_box;
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    ModelViewProjectionMatrix = glm::scale(ViewProjectionMatrix,
                                            glm::vec3(1.0f, 1.0f, 1.0f));
    glUniformMatrix4fv(loc_ModelViewProjectionMatrix, 1, GL_FALSE,
                       &ModelViewProjectionMatrix[0][0]);
    glLineWidth(2.0f);
    draw_axes();
    glLineWidth(1.0f);

    ModelMatrix_big_cow = glm::rotate(glm::mat4(1.0f), rotation_angle_cow,
                                      glm::vec3(0.0f, 1.0f, 0.0f));
    ModelMatrix_big_cow = glm::translate(ModelMatrix_big_cow
                                         glm::vec3(0.0f, 0.0f, 5.0f));
    ModelMatrix_big_cow = glm::scale(ModelMatrix_big_cow, glm::vec3(6.5f, 6.5f, 6.5f));
    ModelViewProjectionMatrix = ViewProjectionMatrix * ModelMatrix_big_cow;
    glUniformMatrix4fv(loc_ModelViewProjectionMatrix, 1, GL_FALSE,
                       &ModelViewProjectionMatrix[0][0]);
    glUniform3f(loc_primitive_color, 0.482f, 0.408f, 0.933f);
    draw_geom_obj(GEOM_OBJ_ID_COW); // draw the bigger cow
    ...
}
```

Motion and Mouse Callback Functions

```
int prevx, prevy;

void motion(int x, int y) { // motion callback
    if (!camera.move) return;

    renew_cam_position(prevy - y);
    renew_cam_orientation_rotation_around_v_axis(prevx - x);
    prevx = x; prevy = y;

    set_ViewMatrix_from_camera_frame();
    ViewProjectionMatrix = ProjectionMatrix * ViewMatrix;

    glutPostRedisplay();
}

void mouse(int button, int state, int x, int y) { // mouse callback
    if ((button == GLUT_LEFT_BUTTON)) {
        if (state == GLUT_DOWN) {
            camera.move = 1; prevx = x; prevy = y;
        }
    } else if (state == GLUT_UP) camera.move = 0;
}
}
```

마우스 입력에 따른 카메라 프레임의 설정

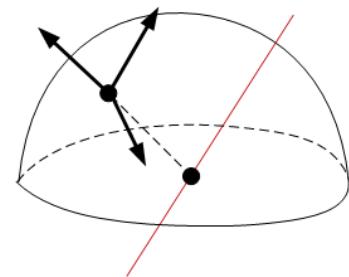
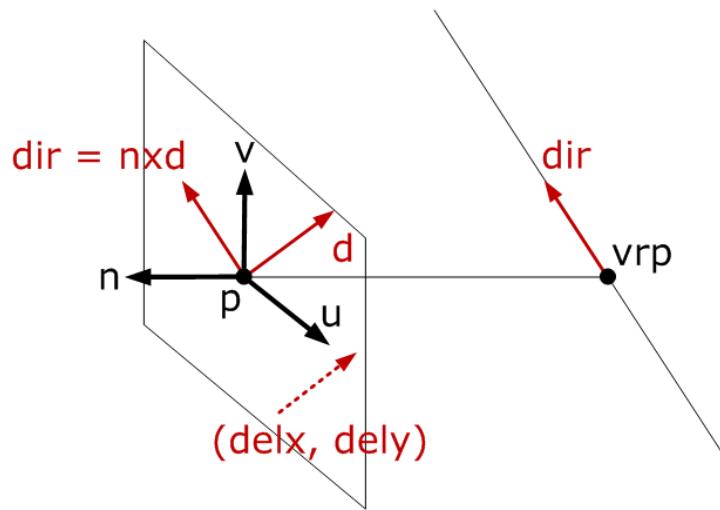
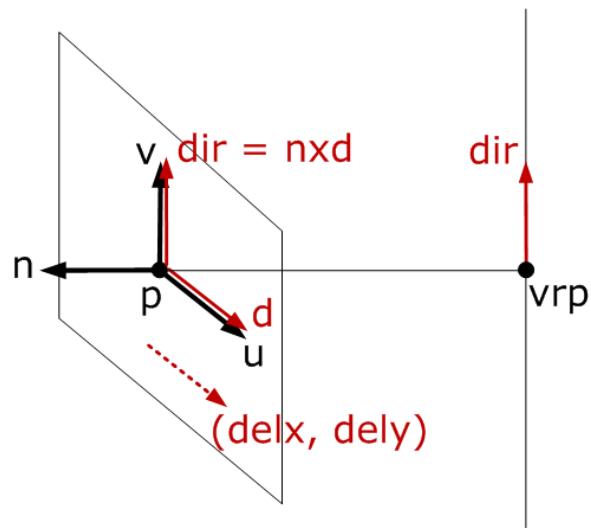
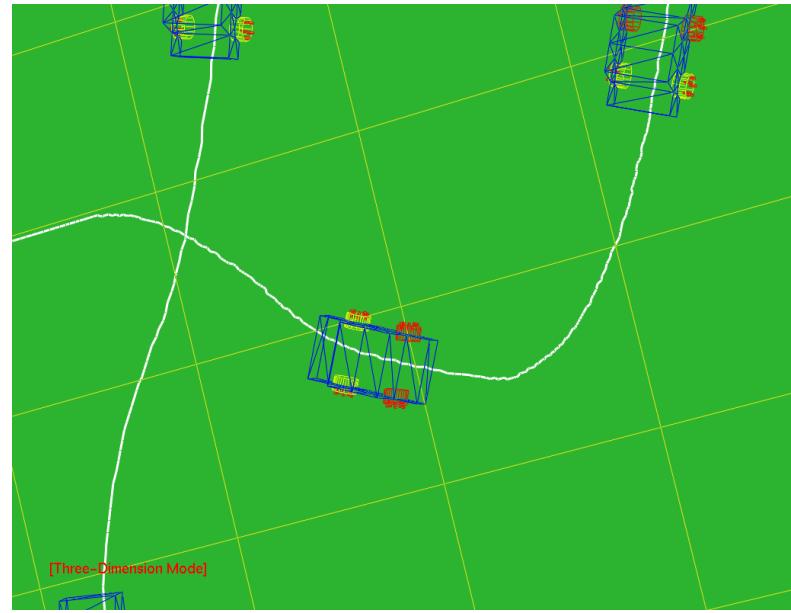
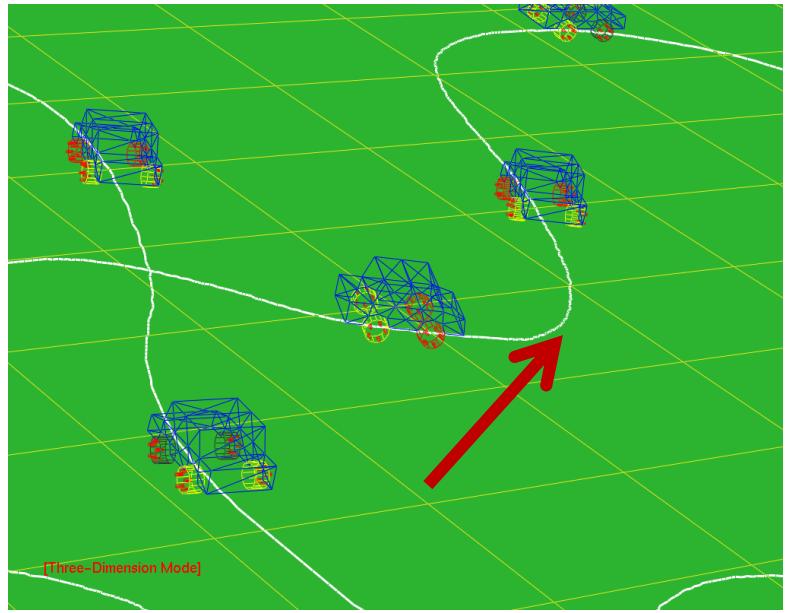
```
#define CAM_TSPEED 0.05f
void renew_cam_position(int del) {
    camera.pos[0] += CAM_TSPEED*del*(-camera.naxis[0]);
    camera.pos[1] += CAM_TSPEED*del*(-camera.naxis[1]);
    camera.pos[2] += CAM_TSPEED*del*(-camera.naxis[2]);
}

#define CAM_RSPEED 0.1f
void renew_cam_orientation_rotation_around_v_axis(int angle) {
    // let's get a help from glm
    glm::mat3 RotationMatrix;
    glm::vec3 direction;

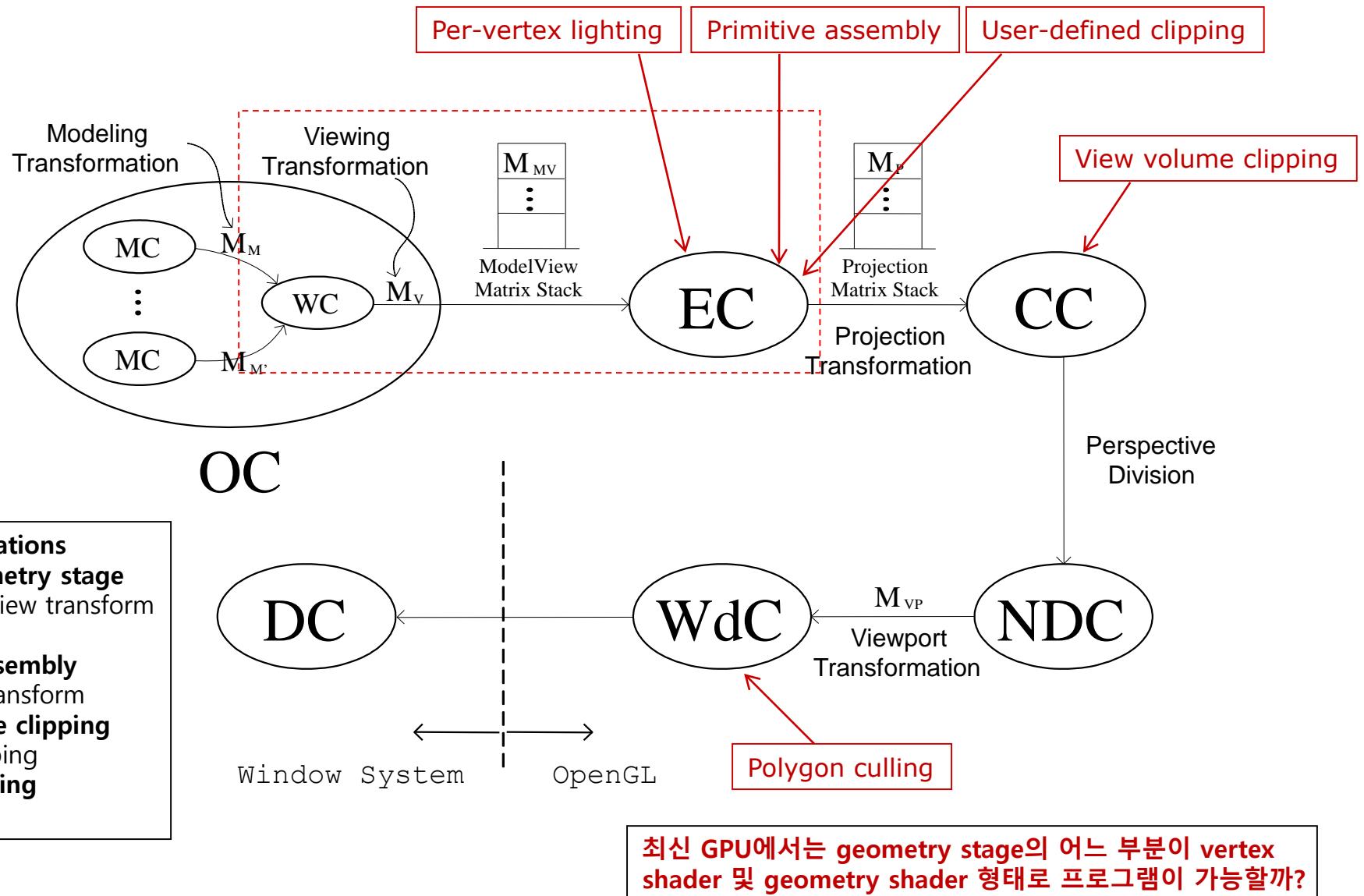
    RotationMatrix = glm::mat3(glm::rotate(glm::mat4(1.0), CAM_RSPEED*TO_RADIAN*angle,
                                             glm::vec3(0.0f, 1.0f, 0.0f)));

    direction = RotationMatrix * glm::vec3(camera.uaxis[0], camera.uaxis[1],
                                            camera.uaxis[2]);
    camera.uaxis[0] = direction.x; camera.uaxis[1] = direction.y;
    camera.uaxis[2] = direction.z;
    direction = RotationMatrix
        * glm::vec3(camera.naxis[0], camera.naxis[1], camera.naxis[2]);
    camera.naxis[0] = direction.x; camera.naxis[1] = direction.y;
    camera.naxis[2] = direction.z;
}
```

카메라에 대한 조작 예 2



Geometry Stage in Fixed-Function OpenGL Pipeline



OpenGL에서의 법선 벡터 설정

```
void glNormal3d(GLdouble x, GLdouble y, GLdouble z);
```

- 현재 법선 벡터(current normal vector)를 설정
- glVertex*() 함수를 호출할 경우 현재 법선 벡터, 현재 색깔, 현재 텍스춰 좌표 등이 꼭지점 좌표에 붙어 기하 파이프라인으로 흘러감.
- glNormal*() 함수 호출시 항상 모델뷰 행렬 스택의 탑에 있는 M_{MV} 에 의해 눈 좌표계로 변환이 됨. → 눈 좌표계에서 조명 계산에 사용이 됨.
- M_{MV} 의 내용에 따라 벡터의 크기가 변할 수 있음. 주로 단위 법선 벡터(unit normal vector)를 사용함.
 - glEnable(GL_NORMALIZE);

Vertex Attribute Register Number	Conventional Per-vertex Parameter	Conventional Per-vertex Parameter Command	Conventional Component Mapping
0 (OPOS)	vertex position	Vertex	x, y, z, w
1 (WGHT)	vertex weights	VertexWeightEXT	w, 0, 0, 1
2 (NRML)	normal	Normal	x, y, z, 1
3 (CULO)	primary color	Color	r, g, b, a
4 (COL1)	secondary color	SecondaryColorEXT	r, g, b, 1
5 (FOGC)	fog coordinate	FogCoordEXT	fc, 0, 0, 1
6	-	-	-
7	-	-	-
8 (TEX0)	texture coord 0	MultiTexCoord(GL_TEXTURE0_ARB, ...)	s, t, r, q
9 (TEX1)	texture coord 1	MultiTexCoord(GL_TEXTURE1_ARB, ...)	s, t, r, q
10 (TEX2)	texture coord 2	MultiTexCoord(GL_TEXTURE2_ARB, ...)	s, t, r, q
11 (TEX3)	texture coord 3	MultiTexCoord(GL_TEXTURE3_ARB, ...)	s, t, r, q
12 (TEX4)	texture coord 4	MultiTexCoord(GL_TEXTURE4_ARB, ...)	s, t, r, q
13 (TEX5)	texture coord 5	MultiTexCoord(GL_TEXTURE5_ARB, ...)	s, t, r, q
14 (TEX6)	texture coord 6	MultiTexCoord(GL_TEXTURE6_ARB, ...)	s, t, r, q
15 (TEX7)	texture coord 7	MultiTexCoord(GL_TEXTURE7_ARB, ...)	s, t, r, q

```
glBegin(GL_POLYGON);
    glNormal3d(-0.9927, 0.0000, -0.1201);
    glVertex3d(1.3813, 0.0000, 2.4546);
    glNormal3d(-0.9465, 0.0000, -0.3225);
    glVertex3d(1.4000, 0.0000, 2.4000);
    glNormal3d(-0.9140, 0.2452, -0.3229);
    glVertex3d(1.350741, -0.375926, 2.400000);
    glNormal3d(-0.9588, 0.2572, -0.1201);
    glVertex3d( 1.3327, -0.3709, 2.4546);
    glEnd();
```

[CSE4170 기초 컴퓨터 그래픽스]

2019년도 1학기

강의자료 II

OpenGL Shader Programming

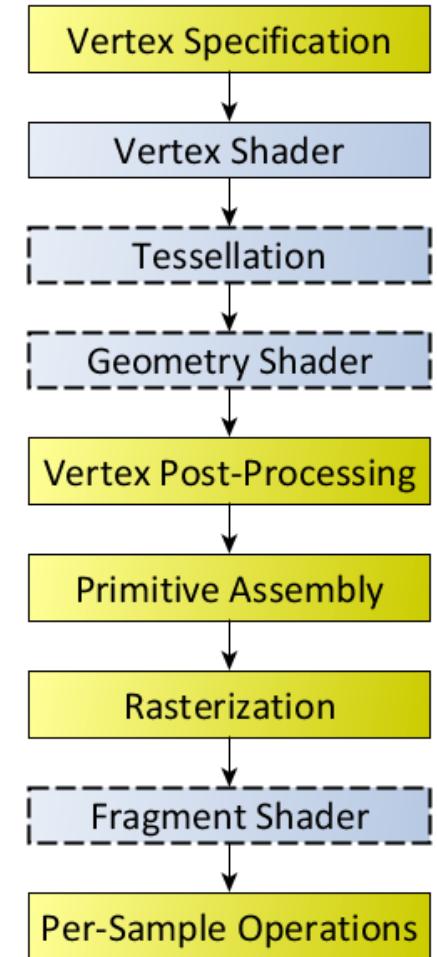
OpenGL Shading Language 4.3

Shader and OpenGL

- OpenGL before and including version 3.0
 - fixed-function pipeline
- Starting with version 3.1
 - fixed-function pipeline was removed from the core profile
 - shaders became mandatory
- GLSL (OpenGL Shading Language)
 - programming language specially designed for graphics
 - similar to the “C” language, with a little C++ mixed in

OpenGL's Programmable Pipeline

- Vertex shading stage
 - Receive the vertex data
 - Processes each vertex separately
 - Mandatory for all OpenGL programs
- Tessellation shading stage
 - Optional stage generates additional geometry
 - Receives the output of the vertex shading stage
 - Does further processing of the received vertices



© www.opengl.org/wiki

- Geometry shading stage
 - Optional stage modifies entire geometric primitives
 - Operates on individual geometric primitives
 - Receives the output of the vertex shading stage or tessellation shading stage
 - Generates more geometry from the input primitive
 - Changes the type of geometric primitive
 - discards the geometry
- Fragment shading stage
 - Processes the individual fragments
 - Receives the output of rasterizer
 - Must have a shader bound to it
 - Fragment's color and depth values are computed
- Compute shading stage
 - Not part of the graphical pipeline
 - Processes generic work items

Preprocessor

Preprocessor [3.3]

Preprocessor Directives

```
#      #define    #elif     #if      #else
#extension #version  #ifdef   #ifndef  #undef
#error     #include   #line    #endif   #pragma
```

Preprocessor Operators

#version 430 #version 430 <i>profile</i>	Required when using version 4.30. <i>profile</i> is core, compatibility, or es.
#extension <i>extension_name</i> : <i>behavior</i> #extension all : <i>behavior</i>	<ul style="list-style-type: none">• <i>behavior</i>: require, enable, warn, disable• <i>extension_name</i>: extension supported by compiler, or “all”

Predefined Macros

__LINE__	Decimal integer constants. __FILE__ says which source string is being processed.
__VERSION__	Decimal integer, e.g.: 430
GL_core_profile	Defined as 1
GL_es_profile	1 if the implementation supports the es profile
GL_compatibility_profile	Defined as 1 if the implementation supports the compatibility profile.

- Macro definition

```
#define NUM_ELEMENTS 10 // define a single value
#define Lpos(n) gl_LightSource[(n)].position // define with parameters
```

```
#ifdef NUM_ELEMENTS
...
#endif
```

- Conditional code

- Use the **#ifdef** directive
- or use the defined operator with the **#if** or **#elif** directives

```
#if defined ( NUM_ELEMENTS ) && NUM_ELEMENTS > 3
...
#elif NUM_ELEMENTS < 7
...
#endif
```

- Optimization Compiler Option
 - Optimization is enabled by default

```
#pragma optimize(on) // enable optimization
#pragma optimize(off) // disable optimization
```

- Debug Compiler Option
 - Debugging is disabled by default

```
#pragma debug(on) // enable debugging
#pragma debug(off) // disable debugging
```

- **#line** directive and **#error** directive
 - **#line** directive controls diagnostic numbering
 - **#error** directive insert text into the shader information log

```
#line 0
#error line 0 error
#line 25
#error line 25 error
#line 0 1
#error source 1, line 0 error
#line 30 2
#error source 2, line 30 error
```

```
0(0) : error C0000: line 0 error
0(25) : error C0000: line 25 error
1(0) : error C0000: source 1, line 0 error
2(30) : error C0000: source 2, line 30 error
```

Line number
 Source string number

Variable Types: Scalars, Vectors, and Matrices

- Variable names
 - Use letters, numbers, underscore character(_)
 - A digit, a underscore can not be the first character
 - Cannot contain consecutive underscores
 - Reserved in GLSL
- Transparent types
 - Internal form is exposed
 - The shader code gets to assume what they look like internally

Basic data types in GLSL

Types [4.1]	
Transparent Types	
void	no function return value
bool	Boolean
int, uint	signed/unsigned integers
float	single-precision floating-point scalar
double	double-precision floating scalar
vec2, vec3, vec4	floating point vector
dvec2, dvec3, dvec4	double precision floating-point vectors
bvec2, bvec3, bvec4	Boolean vectors
ivec2, ivec3, ivec4 uvec2, uvec3, uvec4	signed and unsigned integer vectors
mat2, mat3, mat4	2x2, 3x3, 4x4 float matrix
mat2x2, mat2x3, mat2x4	2-column float matrix of 2, 3, or 4 rows
mat3x2, mat3x3, mat3x4	3-column float matrix of 2, 3, or 4 rows
mat4x2, mat4x3, mat4x4	4-column float matrix of 2, 3, or 4 rows
dmat2, dmat3, dmat4	2x2, 3x3, 4x4 double-precision float matrix
dmat2x2, dmat2x3, dmat2x4	2-col. double-precision float matrix of 2, 3, 4 rows
dmat3x2, dmat3x3, dmat3x4	3-col. double-precision float matrix of 2, 3, 4 rows
dmat4x2, dmat4x3, dmat4x4	4-column double-precision float matrix of 2, 3, 4 rows

Floating-Point Opaque Types

sampler{1D,2D,3D}	1D, 2D, or 3D texture
image{1D,2D,3D}	
samplerCube	cube mapped texture
imageCube	
sampler2DRect	rectangular texture
image2DRect	
sampler{1D,2D}Array	1D or 2D array texture
image{1D,2D}Array	
samplerBuffer	buffer texture
imageBuffer	
sampler2DMS	2D multi-sample texture
image2DMS	
sampler2DMSArray	2D multi-sample array texture
image2DMSArray	
samplerCubeArray	cube map array texture
imageCubeArray	
sampler1DShadow	1D or 2D depth texture with comparison
sampler2DShadow	
sampler2DRectShadow	rectangular tex. / compare
sampler1DArrayShadow	1D or 2D array depth texture with comparison
sampler2DArrayShadow	
samplerCubeShadow	cube map depth texture with comparison
samplerCubeArrayShadow	cube map array depth texture with comparison

Signed Integer Opaque Types

isampler[1,2,3]D	integer 1D, 2D, or 3D texture
iimage[1,2,3]D	integer 1D, 2D, or 3D image
isamplerCube	integer cube mapped texture
iimageCube	integer cube mapped image
isampler2DRect	int. 2D rectangular texture

Continue ↑

Signed Integer Opaque Types (cont'd)

iimage2DRect	int. 2D rectangular image
isampler[1,2]DArray	integer 1D, 2D array texture
iimage[1,2]DArray	integer 1D, 2D array image
isamplerBuffer	integer buffer texture
iimageBuffer	integer buffer image
isampler2DMS	int. 2D multi-sample texture
iimage2DMS	int. 2D multi-sample image
isampler2DMSArray	int. 2D multi-sample array tex.
iimage2DMSArray	int. 2D multi-sample array image
isamplerCubeArray	int. cube map array texture
iimageCubeArray	int. cube map array image

Unsigned Integer Opaque Types

atomic_uint	uint atomic counter
usampler[1,2,3]D	uint 1D, 2D, or 3D texture
uimage[1,2,3]D	uint 1D, 2D, or 3D image
usamplerCube	uint cube mapped texture
uimageCube	uint cube mapped image
usampler2DRect	uint rectangular texture
uimage2DRect	uint rectangular image
usampler[1,2]DArray	1D or 2D array texture
iimage[1,2]DArray	1D or 2D array image
usamplerBuffer	uint buffer texture
uimageBuffer	uint buffer image
usampler2DMS	uint 2D multi-sample texture
uimage2DMS	uint 2D multi-sample image
usampler2DMSArray	uint 2D multi-sample array tex.

Continue ↑

Unsigned Integer Opaque Types (cont'd)

uimage2DMSArray	uint 2D multi-sample array image
usamplerCubeArray	uint cube map array texture
uimageCubeArray	uint cube map array image

Opaque types

- Internal form is not exposed
- Markers that reference the real data
- “sampler types” represent a texture bound to the OpenGL context
- “image types” refer to an image stored within a texture
- “atomic counter types” are limited form of buffer image variable

- Variable scoping
 - Variable declared outside of function definition have global scope
 - Variable declared within a set of curly braces exist within the scope of those braces only
 - Loop iteration variables are only scoped for the body of the loop

```
for ( int i = 0 ; i < 10 ; ++i ){  
    // loop body  
}
```

- Variable initialization
 - Integer literal constants
 - octal(base 8, 0-), decimal(base 10), hexadecimal(base 16, 0x-/0X-)
 - Minus sign before a numeric value negates the constant
 - Trailing "u"/"U" denotes an unsigned integer value
 - Floating-point literals
 - Must include decimal point
 - Optionally include an "f"/"F"(float), "lf"/"lf"(double) suffix

```
double pi = 3.141592f;
```

Constructors

Implicit Conversions

int	->	uint	uvec2	->	dvec2
int, uint	->	float	uvec3	->	dvec3
int, uint, float	->	double	uvec4	->	dvec4
ivec2	->	uvec2	vec2	->	dvec2
ivec3	->	uvec3	vec3	->	dvec3
ivec4	->	uvec4	vec4	->	dvec4
ivec2	->	vec2	mat2	->	dmat2
ivec3	->	vec3	mat3	->	dmat3
ivec4	->	vec4	mat4	->	dmat4
uvec2	->	vec2	mat2x3	->	dmat2x3
uvec3	->	vec3	mat2x4	->	dmat2x4
uvec4	->	vec4	mat3x2	->	dmat3x2
ivec2	->	dvec2	mat3x4	->	dmat3x4
ivec3	->	dvec3	mat4x2	->	dmat4x2
ivec4	->	dvec4	mat4x3	->	dmat4x3

Aggregation of Basic Types

Arrays	float[3] foo; float foo[3]; int a [3][2]; // Structures, blocks, and structure members // can be arrays. Arrays of arrays supported.
Structures	struct type-name { members } struct-name[]; // optional variable declaration
Blocks	in/out/uniform block-name { // interface matching by block name optionally-qualified members } instance-name[]; // optional instance name, optionally an array

- Implicit conversions

- Fewer implicit conversion between values (more safe than C++)

```
int f = false; // compilation error
```

- Explicit conversions

- Function with the same name as a type returns value of that type

➤ **float, double, uint, int, bool ...**

```
float f = 10.0;
```

```
int ten = int ( f );
```

- Function overloading

- Vector and matrix constructors

```
vec3 velocity = vec3 ( 0.0 , 2.0 , 3.0 ) ; // initialize a vec3 with 3 floats
ivec steps = ivec3 ( velocity ) ; // convert vec3 to ivec

vec4 color;
vec3 RGB = vec3 ( color ) ; // color vector is truncated so RGB only has three elements

vec3 white = vec3 ( 1.0 ) ; // white = ( 1.0 , 1.0 , 1.0 )
vec4 translucent = vec4 ( white, 0.5 ) ; // translucent = ( 1.0, 1.0, 1.0, 0.5 )

mat3 identity = mat3 ( 1.0 ) ; // identity matrix    identity =  $\begin{pmatrix} 1.0 & 0.0 & 0.0 \\ 0.0 & 1.0 & 0.0 \\ 0.0 & 0.0 & 1.0 \end{pmatrix}$ 
mat2 diagonal = mat2 ( 2.0 ) ; // diagonal matrix      diagonal =  $\begin{pmatrix} 2.0 & 0.0 \\ 0.0 & 2.0 \end{pmatrix}$ 

// matrices are specified in column-major order
mat3 M = mat3 ( 1.0, 2.0, 3.0,
                  4.0, 5.0, 6.0,
                  7.0, 8.0, 9.0 ) ; // first column
                           // second column
                           // third column
M =  $\begin{pmatrix} 1.0 & 4.0 & 7.0 \\ 2.0 & 5.0 & 8.0 \\ 3.0 & 6.0 & 9.0 \end{pmatrix}$ 

vec3 column1 = vec3 ( 1.0, 2.0, 3.0 ) ;
vec3 column2 = vec3 ( 4.0, 5.0, 6.0 ) ;
vec3 column3 = vec3 ( 7.0, 8.0, 9.0 ) ;
M = ( column1, column2, column3); // compose a matrix of three columns
```

- Structure constructors
 - Group collections of different types
 - Automatically creates a new type, implicitly defines a constructor function

```
struct Particle {  
    float lifetime ;  
    vec3 position ;  
    vec3 velocity ;  
};  
Particle p = Particle ( 10.0 , pos , vel ) ; // pos , vel are vec3s
```

- Array constructors
 - Negative array indices, positive indices out of range are not permitted
 - GLSL 4.3 allows a multidimensional array

```
float coeff [ 3 ] ; // an array or 3 floats  
float [ 3 ] coeff ; // same thing  
int indices [ ] ; // array can be declared unsized. Redeclare later with a size  
  
float coeff [ 3 ] = float [ 3 ] ( 2.38 , 3.14 , 42.0 ) ;
```

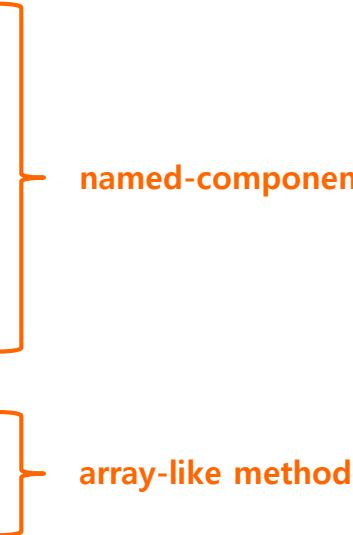
Vector and Matrix Components

- Vector component accessors

Component Accessors	Description
(x, y, z, w)	Components associated with positions
(r, g, b, a)	Components associated with colors
(s, t, p, q)	Components associated with texture coordinates

- Accessing elements in vectors and matrices
 - Vectors support a named-component method and an array-like method
 - Matrices support an array-like method

```
vec3 luminance = color.rrr ;  
color = color.abgr ; // reverse the components of a color  
  
vec4 color = otherColor.rgz ; // Error : "z" is from a different group  
  
vec2 pos ;  
float zPos = pos.z ; // Error : no "z" component in 2D vectors  
  
mat4 m = mat4 ( 2.0 ) ;  
vec4 zVec = m [ 2 ] ; // get column 2 of the matrix  
float yScale = m [ 1 ] [ 1 ] ; // or m[1].y works as well
```



named-component method

array-like method

Operations and Constructors

Vector & Matrix [5.4.2]

.length() for matrices returns number of columns

.length() for vectors returns number of components

```
mat2(vec2, vec2);           // 1 col./arg.  
mat2x3(vec2, float, vec2, float); // col. 2  
dmat2(dvec2, dvec2);        // 1 col./arg.  
dmat3(dvec3, dvec3, dvec3); // 1 col./arg.
```

Structure Example [5.4.3]

.length() for structures returns number of members

```
struct light {members;};  
light lightVar = light(3.0, vec3(1.0, 2.0, 3.0));
```

Array Example [5.4.4]

```
const float c[3];  
c.length()          // will return the integer 3
```

Matrix Examples [5.6]

Examples of access components of a matrix with array subscripting syntax:

```
mat4 m;           // m is a matrix  
m[1] = vec4(2.0); // sets 2nd col. to all 2.0  
m[0][0] = 1.0;   // sets upper left element to 1.0  
m[2][3] = 2.0;   // sets 4th element of 3rd col. to 2.0
```

Examples of operations on matrices and vectors:

```
m = f * m;      // scalar * matrix component-wise  
v = f * v;      // scalar * vector component-wise  
v = v * v;      // vector * vector component-wise  
m = m +/ - m;  // matrix +/- matrix comp.-wise  
m = m * m;      // linear algebraic multiply  
f = dot(v, v); // vector dot product  
v = cross(v, v); // vector cross product
```

Structure & Array Operations [5.7]

Select structure fields or length() method of an array using the period (.) operator. Other operators:

.	field or method selector
== !=	equality
=	assignment
[]	indexing (arrays only)

Array elements are accessed using the array subscript operator ([]), e.g.:

```
diffuseColor += lightIntensity[3]*NdotL;
```



```
mat3x4 m ;
```

```
int c = m.length () ;
```

```
// number of columns in m : 3
```

```
int r = m [ 0 ].length () ;
```

```
// number of components in column vector 0 : 4
```

```
float coeff [ 3 ] [ 5 ] ;
```

```
coeff.length () ; // returns the constant 3
```

```
coeff [ 2 ] ; // a one-dimensional array of size 5
```

```
coeff [ 3 ].length () ; // returns the constant 5
```

Qualifiers

- Types can also have modifiers that affect their behavior
- uniform** storage qualifier
 - Variable's value will be specified by the application
 - Value does not change across the primitive being processed
 - Shader cannot write to **uniform** variable

```
#version 430
```

Vertex Shader

```
uniform mat4 u_ModelViewProjectionMatrix ;
uniform vec3 u_primitive_color ;

layout (location = 0) in vec4 a_position ; // input into a shader stage
out vec4 v_color ; // output from a shader stage

void main ( void ) {
    v_color = vec4 ( u_primitive_color, 1.0f ) ;
    gl_Position = u_ModelViewProjectionMatrix * a_position;
}
```

```
GLint loc_ModelViewProjectionMatrix = glGetUniformLocation( h_ShaderProgram, "u_ModelViewProjectionMatrix" ) ; Application
GLint loc_primitive_color = glGetUniformLocation( h_ShaderProgram, "u_primitive_color" ) ;
```

Qualifiers

Storage Qualifiers [4.3]

Declarations may have one storage qualifier.

none	(default) local read/write memory, or input parameter
const	read-only variable
in	linkage into shader from previous stage
out	linkage out of a shader to next stage
uniform	linkage between a shader, OpenGL, and the application
buffer	accessible by shaders and OpenGL API
shared	compute shader only, shared among work items in a local work group

```
GLfloat axes_color[3] = { 1.0f , 0.0f , 0.0f } ; Application
glUniform3fv ( loc_primitive_color , 1 , axes_color ) ;

glm :: mat4 ModelViewProjectionMatrix ;
glUniformMatrix4fv( loc_ModelViewProjectionMatrix, 1,
GL_FALSE, &ModelViewProjectionMatrix[0][0] ) ;
```

① obtain the index of the uniform variable

② set the value of the uniform variable

- Layout qualifiers
 - Used for vertex shader input assignment

```
layout ( location = 0 ) in vec4 a_position ; // set a_position attribute location to zero
```

Vertex Shader

the same value

```
glVertexAttribPointer ( 0 , 3 , GL_FLOAT, GL_FALSE, n_bytes_per_vertex, BUFFER_OFFSET ( 0 ) );
```

Application

- Used for fragment shader output assignment

```
layout ( location = 0 ) out final_color;  
// set the final_color's fragment color to zero
```

Fragment Shader

Layout Qualifiers [4.4]

`layout(layout-qualifiers) block-declaration`
`layout(layout-qualifiers) in/out/uniform`
`layout(layout-qualifiers) in/out/uniform declaration`

Input Layout Qualifiers [4.4.1]

For all shader stages:
`location = integer-constant`

Tessellation Evaluation

triangles, quads, equal_spacing, isolines,
`fractional_{even,odd}_spacing`, cw, ccw,
`point_mode`

Geometry Shader

points, lines, {lines,triangles}_adjacency,
 triangles, invocations = `integer-constant`

Fragment Shader

For redeclaring built-in variable `gl_FragCoord`:
`origin_upper_left`, `pixel_center_integer`

For `in` only (not with variable declarations):
`early_fragment_tests`

Compute Shader

`local_size_x = integer-constant`,
`local_size_y = integer-constant`,
`local_size_z = integer-constant`

Output Layout Qualifiers [4.4.2]

For all shader stages:
`location = integer-constant`,
`index = integer-constant`

Tessellation Control

`vertices = integer-constant`

Geometry Shader

points, `line_strip`, `triangle_strip`,
`max_vertices = integer-constant`,
`stream = integer-constant`

Fragment Shader

`depth_any`, `depth_greater`,
`depth_less`, `depth_unchanged`

Uniform Variable Layout Qualifiers [4.4.3]
`location = integer-constant`

Subroutine Function Layout Qualifiers [4.4.4]
`index = integer-constant`

Storage Block Layout Qualifiers [4.4.5]

Layout qualifier identifiers for uniform blocks:
`shared`, `packed`, `std140`, `std340`,
`{row, column}_major`,
`binding = integer-constant`

Opaque Uniform Layout Qualifiers [4.4.6]

Used to bind opaque uniform variables to
 specific buffers or units.
`binding = integer-constant`

Atomic Counter Layout Qualifiers

`binding = integer-constant`,
`offset = integer-constant`

(Continued on next page >)

- Parameter qualifiers
 - No concept of a pointer or reference
 - **in** : the value will be copied into the parameter when the function is called
 - **out** : the value will be copied out into the variable after the function's execution is completed
 - **inout** : combines both

```
void MyFunction ( in float inputValue, out int outputValue,
                  inout float inAndOutValue )
{
    inputValue = 0.0 ;
    outputValue = int ( inAndOutValue + inputValue ) ;
    inAndOutValue = 3.0 ;
}
void main ( )
{
    float in1 = 10.5 ;
    int out1 = 5 ;
    float out2 = 10.0 ;
    MyFunction ( in1, out1, out2 ) ;
}
```

[After function call]
 in1 : 10.5
 out1 : 10
 out2 : 3.0

Parameter Qualifiers [4.6]
 Input values copied in at function call time,
 output values copied out at function return.

none	(default) same as in
in	for function parameters passed into function
const	for function parameters that cannot be written to
out	for function parameters passed back out of function, but not initialized when passed in
inout	for function parameters passed both into and out of a function

- Precision qualifiers
 - Supported for compatibility with OpenGL ES 2.0.
 - No functional effects
- Computational invariance
 - Two identical computations in different shaders may result in different results because compiler's optimizations may reorder instructions
 - Values may lose precision when written to a register
 - Values can be calculated in a different way
 - **invariant, precise** qualifier is applied to any shader output variable guaranteeing the same result from the same expression

Precision Qualifiers [4.7]

Precision qualifiers have no effect on precision; they aid code portability with OpenGL ES:

highp, mediump, lowp

Invariant Qualifiers Examples [4.8]

These are for vertex, tessellation, geometry, and fragment languages.

#pragma STDGL invariant(all)	force all output variables to be invariant
invariant gl_Position;	qualify a previously declared variable
invariant centroid out vec3 Color;	qualify as part of a variable declaration

Precise Qualifier [4.9]

Ensures that operations are executed in stated order with operator consistency. For example, a fused multiply-add cannot be used in the following; it requires two identical multiplies, followed by an add.

```
precise out vec4 Position = a * b + c * d;
```

Interface Blocks

- Shader variables organized into blocks of variables
- In/Out blocks
 - Input and output variables are organized into in and out blocks
 - Visually verify interface matches between stages, link separate programs easier

```
in Lighting {  
    vec3 normal ;  
    vec2 bumpCoord ;  
};
```

- Buffer blocks
 - Shader storage buffers are organized into buffer blocks
 - Shader can both read and write the members of the buffer block
 - Use a uniform block when there is no need to write to a buffer

```
buffer BufferObject { // create a read-writable buffer  
    int mode ;      // preamble members  
    vec4 points [ ] ; // size can be established just before rendering  
};
```

Interface Blocks [4.3.9]

Input, **output**, **uniform**, and **buffer** variable declarations can be grouped. For example:

```
uniform Transform {  
    mat4 ModelViewMatrix;  
    // allowed restatement qualifier  
    uniform mat3 NormalMatrix;  
};
```

- Uniform blocks
 - Uniform variables are organized into uniform blocks
 - Optimize both accessing uniform variables and enabling sharing of uniform values across shader programs

```
#version 430
```

```
uniform Uniforms {
    vec3 translation;
    float scale;
}
```

Vertex Shader

```
#version 430
```

```
uniform Uniforms {
    vec3 translation;
    float scale;
}
```

Fragment Shader

...

// find the uniform buffer index for "Uniforms"

```
GLuint uboIndex = glGetUniformBlockIndex ( program,
"Uniforms" );
```

// determine the block's sizes

```
GLint uboSize;
```

```
glGetActiveUniformBlockiv ( program, uboIndex,
GL_UNIFORM_BLOCK_DATA_SIZE, &uboSize );
```

```
GLvoid *buffer = malloc ( uboSize );
```

...

Application

```
...
// find index of a particular named uniform variable
glGetUniformIndices(program, NumUniforms, names, indices);
// get the offset and size for that particular index
glGetActiveUniformsiv(program, NumUniforms, indices,
GL_UNIFORM_OFFSET, offset );
glGetActiveUniformsiv(program, NumUniforms, indices,
GL_UNIFORM_SIZE, size );
glGetActiveUniformsiv(program, NumUniforms, indices,
GL_UNIFORM_TYPE, type );
/* copy the uniform values into the buffer */
...
```

// initialize and associate a buffer object with uniform block

```
GLuint ubo;
 glGenBuffers(1, &ubo);
 glBindBuffer(GL_UNIFORM_BUFFER, ubo);
 glBufferData(GL_UNIFORM_BUFFER, uboSize, buffer,
 GL_STATIC_RAW);
 glBindBufferBase(GL_UNIFORM_BUFFER, uboIndex, ubo);
...
```

Application

Built-In Inputs, Outputs

- Vertex shader built-in input variables
 - **gl_VertexID** : the index of the vertex currently being processed
 - **gl_InstanceID** : the index of the current instance when doing some form of instanced rendering
- Vertex shader built-in output variables
 - Defined as an interface block
 - **gl_Position** : the clip-space output position of the current vertex
 - **gl_PointSize** : the pixel width/height of the point being rasterized
 - **gl_ClipDistance** : allows the shader to set the distance from the vertex to each user-defined clipping half-space

Built-In Variables [7]	
Vertex Language	
Inputs	in int gl_VertexID; in int gl_InstanceID;
Outputs	out gl_PerVertex { vec4 gl_Position; float gl_PointSize; float gl_ClipDistance[]; };

- Fragment shader built-in input variables
 - **gl_FragCoord** : the location of the fragment in window space
 - **gl_FrontFacing** : true if this fragment was generated by the front-face
 - **gl_ClipDistance** : interpolated clipping plane half-spaces, as output for vertices from the last vertex processing stage
 - **gl_PointCoord** : the location within a point primitive that defines the position of the fragment relative to the side of the point
 - **gl_PrimitiveID** : the index of the current primitive being rendered
 - **gl_SampleID** : an integer identifier for the current sample that this fragment is rasterized for
 - **gl_SamplePosition** : the location of the current sample for the fragment within the pixel's area
 - **gl_SampleMask** : a bitfield for the sample mask of the fragment being generated
 - **gl_Layer** : either 0 or the layer number for this primitive output by the Geometry Shader

Fragment Language	
Inputs	
Outputs	
	<pre>in vec4 gl_FragCoord; in bool gl_FrontFacing; in float gl_ClipDistance[]; in vec2 gl_PointCoord; in int gl_PrimitiveID; in int gl_SampleID; in vec2 gl_SamplePosition; in int gl_SampleMask[]; in int gl_Layer; in int gl_ViewportIndex;</pre>
	<pre>out float gl_FragDepth; out int gl_SampleMask[];</pre>

- **gl_viewportIndex** : either 0 or the viewport index for this primitive output by the Geometry Shader
- Fragment shader built-in output variables
 - **gl_FragDepth** : fragment's depth
 - **gl_SampleMask** : defines the sample mask for the fragment when performing multisampled rendering

Built-In Constants

Built-In Constants [7.3]

The following are provided to all shaders. The actual values are implementation-dependent, but must be at least the value shown.

```
const ivec3 gl_MaxComputeWorkGroupCount[] = {65535, 65535, 65535};  
const ivec3 gl_MaxComputeLocalWorkSize[] = {1024, 1024, 64};  
const int gl_MaxComputeUniformComponents = 1024;  
const int gl_MaxComputeTextureImageUnits = 16;  
const int gl_MaxComputeImageUniforms = 8;  
const int gl_MaxComputeAtomicCounters = 8;  
const int gl_MaxComputeAtomicCounterBuffers = 1;  
const int gl_MaxVertexAttribs = 16;  
const int gl_MaxVertexUniformComponents = 1024;  
const int gl_MaxVaryingComponents = 60;  
const int gl_MaxVertexOutputComponents = 64;  
const int gl_MaxGeometryInputComponents = 64;  
const int gl_MaxGeometryOutputComponents = 128;  
const int gl_MaxFragmentInputComponents = 128;  
const int gl_MaxVertexTextureImageUnits = 16;  
const int gl_MaxCombinedTextureImageUnits = 80;  
const int gl_MaxTextureImageUnits = 16;  
const int gl_MaxImageUnits = 8;  
const int gl_MaxCombinedImageUnitsAndFragmentOutputs = 8;  
const int gl_MaxImageSamples = 0;  
const int gl_MaxVertexImageUniforms = 0;  
const int gl_MaxTessControlImageUniforms = 0;  
const int gl_MaxTessEvaluationImageUniforms = 0;  
const int gl_MaxGeometryImageUniforms = 0;  
const int gl_MaxFragmentImageUniforms = 8;  
const int gl_MaxCombinedImageUniforms = 8;  
const int gl_MaxFragmentUniformComponents = 1024;  
const int gl_MaxDrawBuffers = 8;  
const int gl_MaxClipDistances = 8;  
const int gl_MaxGeometryTextureImageUnits = 16;
```

```
const int gl_MaxGeometryOutputVertices = 256;  
const int gl_MaxGeometryTotalOutputComponents = 1024;  
const int gl_MaxGeometryUniformComponents = 1024;  
const int gl_MaxGeometryVaryingComponents = 64;  
const int gl_MaxTessControlInputComponents = 128;  
const int gl_MaxTessControlOutputComponents = 128;  
const int gl_MaxTessControlTextureImageUnits = 16;  
const int gl_MaxTessControlUniformComponents = 1024;  
const int gl_MaxTessControlTotalOutputComponents = 4096;  
const int gl_MaxTessEvaluationInputComponents = 128;  
const int gl_MaxTessEvaluationOutputComponents = 128;  
const int gl_MaxTessEvaluationTextureImageUnits = 16;  
const int gl_MaxTessEvaluationUniformComponents = 1024;  
const int gl_MaxTessPatchComponents = 120;  
const int gl_MaxPatchVertices = 32;  
const int gl_MaxTessGenLevel = 64;  
const int gl_MaxViewports = 16;  
const int gl_MaxVertexUniformVectors = 256;  
const int gl_MaxFragmentUniformVectors = 256;  
const int gl_MaxVaryingVectors = 15;  
const int gl_MaxVertexAtomicCounters = 0;  
const int gl_MaxTessControlAtomicCounters = 0;  
const int gl_MaxTessEvaluationAtomicCounters = 0;  
const int gl_MaxGeometryAtomicCounters = 0;  
const int gl_MaxFragmentAtomicCounters = 8;  
const int gl_MaxCombinedAtomicCounters = 8;  
const int gl_MaxAtomicCounterBindings = 1;  
const int gl_MaxVertexAtomicCounterBuffers = 0;  
const int gl_MaxTessControlAtomicCounterBuffers = 0;  
const int gl_MaxTessEvaluationAtomicCounterBuffers = 0;  
const int gl_MaxGeometryAtomicCounterBuffers = 0;  
const int gl_MaxFragmentAtomicCounterBuffers = 1;  
const int gl_MaxCombinedAtomicCounterBuffers = 1;  
const int gl_MaxAtomicCounterBufferSize = 16384;  
const int gl_MinProgramTexelOffset = -8;  
const int gl_MaxProgramTexelOffset = 7;
```

Operators

- Arithmetic operations
 - Operator precedence
 - The type of operands must be the same
 - Vectors, matrices must be of the same dimension
- Overloaded operators

```
vec2 a, b, c ;  
mat2 m, u, v ;  
c = a * b ; // c = (a.x*b.x, a.y*b.y)  
m = u * v ; // m = (u00*v00+u01*v10  u00*v01+u01*v11  
//           u01*v00+u11*v10  u10*v01+u11*v11)
```

Operators and Expressions [5.1]

The following operators are numbered in order of precedence. Relational and equality operators evaluate to Boolean. Also see lessThan(), equal(), etc.

1.	()	parenthetical grouping
2.	[] () .++ --	array subscript function call, constructor, structure field, selector, swizzle postfix increment and decrement

3.	++ -- + - ~ !	prefix increment and decrement unary
4.	* / %	multiplicative
5.	+ -	additive
6.	<< >>	bit-wise shift
7.	<> <= >=	relational
8.	== !=	equality
9.	&	bit-wise and
10.	^	bit-wise exclusive or
11.		bit-wise inclusive or
12.	&&	logical and
13.	^^	logical exclusive or
14.		logical inclusive or
15.	? :	selects an entire operand.
16.	= += -= *= /= %=<<=>>= &= ^= =	assignment arithmetic assignments
17.	,	sequence

Statements

- Flow control
 - GLSL's logical control structures are **if-else** and **switch** statement
 - The else clause is optional, multiple statements require a block
 - Switch statements support "fall-through" cases
 - No statements are allowed before the first case
- Looping constructs
 - Supports the familiar "C" form of **for**, **while**, and **do ... while** loops
- Flow-control statements
 - **break** : terminates execution of the block of a loop
 - **continue** : terminate the current iteration
 - **return [result]** : returns from the current subroutine
 - **discard** : available only in fragment programs. discards the current fragment

Iteration and Jumps [6.3-4]

Function	call by value-return
Iteration	for (;;) { break, continue } while () { break, continue } do { break, continue } while () ;
Selection	if () {} if () {} else {} switch () { case integer: ... break; ... default: ... }
Entry	void main()
Jump	break, continue, return (There is no 'goto')
Exit	return in main() discard // Fragment shader only

Shader Subroutines

Statements and Structure

Subroutines [6.1.2]

Subroutine type variables are assigned to functions through the `UniformSubroutinesuiv` command in the OpenGL API.

Declare types with the `subroutine` keyword:

```
subroutine returnType subroutineTypeName(type0  
    arg0,  
    type1 arg1, ..., typen argn);
```

Associate functions with subroutine types of matching declarations by defining the functions with the `subroutine` keyword and a list of subroutine types the function matches:

```
subroutine(subroutineTypeName0, ...,  
          subroutineTypeNameN)  
returnType functionName(type0 arg0,  
                      type1 arg1, ..., typen argn){ ... }  
// function body
```

Declare subroutine type variables with a specific subroutine type in a subroutine uniform variable declaration:

```
subroutine uniform subroutineTypeName  
    subroutineVarName;
```

- Bind a function call to one of possible function definitions

- Similar to function pointers in C
- Select alternate implementations at runtime

#version 430

return type

parameters

1

2

3

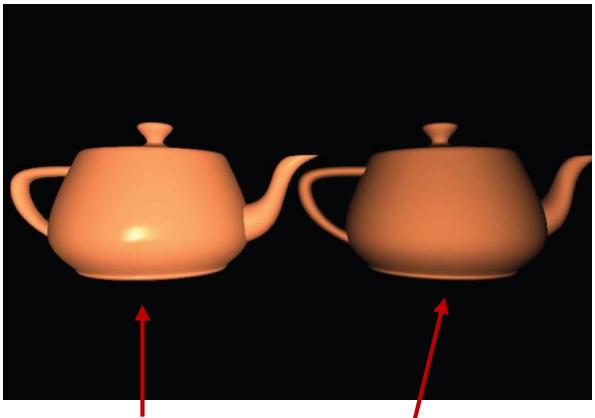
Vertex Shader

```
subroutine vec3 shadeModelType (vec4 position , vec3 normal) ;  
subroutine uniform shadeModelType shadeModel ;  
...  
subroutine ( shadeModelType ) vec3 phongModel ( vec4 position , vec3 norm )  
{  
    ... // ambient, diffuse and specular (ADS) shading calculations go here  
}  
subroutine ( shadeModelType ) vec3 diffuseOnly ( vec4 position, vec3 norm )  
{  
    ... // diffuse shading calculation go here  
}
```

1. define the subroutine type
2. Specify the subroutine uniform variable
3. Define the set of subroutines

4. Call one of the two subroutine functions by utilizing the subroutine uniform

```
...
void main ( )
{
    ...
    LightIntensity = shadeModel ( eyePosition , eyeNorm ) ; 4
    ...
}
```



using "phongModel"

using "diffuseOnly"

Vertex Shader

```
...
GLuint adsIndex = glGetSubroutineIndex ( programHandle ,
                                         GL_VERTEX_SHADER, "phongModel" ) ;
GLuint diffuseIndex = glGetSubroutineIndex ( programHandle,
                                              GL_VERTEX_SHADER, "diffuseOnly" ) ;
glUniformSubroutinesuiv ( GL_VERTEX_SHADER , 1 , &adsIndex ) ;
... // Render the left teapot
glUniformSubroutinesuiv ( GL_VERTEX_SHADER , 1 , &diffuseIndex ) ;
... // Render the right teapot
...
```

Application

1

2

1. Determine the indices of the subroutines inside of the shader
2. Specify which subroutine should be executed

Built-In Functions

Built-In Functions

Angle & Trig. Functions [8.1]

Functions will not result in a divide-by-zero error. If the divisor of a ratio is 0, then results will be undefined. Component-wise operation. Parameters specified as *angle* are in units of radians. Tf=float, vecn.

Tf radians[Tf degrees]	degrees to radians
Tf degrees[Tf radians]	radians to degrees
Tf sin(Tf angle)	sine
Tf cos(Tf angle)	cosine
Tf tan(Tf angle)	tangent
Tf asin(Tf x)	arc sine
Tf acos(Tf x)	arc cosine
Tf atan(Tf y, Tf x)	arc tangent
Tf atan(Tf y_over_x)	arc tangent
Tf sinh(Tf x)	hyperbolic sine
Tf cosh(Tf x)	hyperbolic cosine
Tf tanh(Tf x)	hyperbolic tangent
Tf asinh(Tf x)	hyperbolic sine
Tf acosh(Tf x)	hyperbolic cosine
Tf atanh(Tf x)	hyperbolic tangent

Exponential Functions [8.2]

Component-wise operation. Tf=float, vecn. Td=double, dvecn. Tfd=Tf, Td

Tf pow(Tf x, Tf y)	x^y
Tf exp(Tf x)	e^x
Tf log(Tf x)	ln
Tf exp2(Tf x)	2^x
Tf log2(Tf x)	\log_2
Tfd sqrt(Tfd x)	square root
Tfd inversesqrt(Tfd x)	inverse square root

Common Functions [8.3]

Component-wise operation. Tf=float, vecn. Tb=bool, bvecn. Ti=int, ivecн. Tu=uint, uvecn. Td=double, dvecn. Tfd=Tf, Td. Tiu=Ti, Tu.

Returns absolute value:

Tfd abs(Tfd x) Ti abs(Ti x)

Returns -1.0, 0.0, or 1.0:

Tfd sign(Tfd x) Ti sign(Ti x)

Returns nearest integer $\leq x$:

Tfd floor(Tfd x)

Returns nearest integer with absolute value \leq absolute value of x:

Tfd trunc(Tfd x)

Returns nearest integer, implementation-dependent rounding mode:

Tfd round(Tfd x)

Returns nearest integer, 0.5 rounds to nearest even integer:

Tfd roundEven(Tfd x)

Returns nearest integer $\geq x$:

Tfd ceil(Tfd x)

Returns x - floor(x):

Tfd fract(Tfd x)

Returns modulus:

Tfd mod(Tfd x, Tfd y)

Td mod(Td x, double y)

Tf mod(Tf x, float y)

Returns separate integer and fractional parts:

Tfd modf(Tfd x, out Tfd i)

Returns minimum value:

Tfd min(Tfd x, Tfd y)

Tiu min(Tiu x, Tiu y)

Tf min(Tf x, float y)

Ti min(Ti x, int y)

Td min(Td x, double y)

Tu min(Tu x, uint y)

(Continue ↴)

Common Functions (cont.)

Returns maximum value:

Tfd max(Tfd x, Tfd y) Tiu max(Tiu x, Tiu y)

Tf max(Tf x, float y) Ti max(Ti x, int y)

Td max(Td x, double y) Tu max(Tu x, uint y)

Returns min(max(x, minValue), maxValue):

Tfd clamp(Tfd x, Tfd minValue, Tfd maxValue)

Tf clamp(Tf x, float minValue, float maxValue)

Td clamp(Td x, double minValue, double maxValue)

Tiu clamp(Tiu x, Tiu minValue, Tiu maxValue)

Ti clamp(Ti x, int minValue, int maxValue)

Tu clamp(Tu x, uint minValue, uint maxValue)

Returns linear blend of x and y:

Tfd mix(Tfd x, Tfd y, Tfd a)

Tf mix(Tf x, Tf y, float a)

Td mix(Td x, Td y, double a)

Returns true if components in *a* select components from *y*, else from *x*:

Tfd mix(Tfd x, Tfd y, Tf a)

Returns 0.0 if *x* < *edge*, else 1.0:

Tfd step(Tfd edge, Tfd x)

Td step(double edge, Td x)

Tf step(float edge, Tf x)

Clamps and smoothes:

Tfd smoothstep(Tfd edge0, Tfd edge1, Tfd x)

Tf smoothstep(float edge0, float edge1, Tf x)

Td smoothstep(double edge0, double edge1, Td x)

Returns true if *x* is NaN:

Tb isnan(Tfd x)

Returns true if *x* is positive or negative infinity:

Tb isinf(Tfd x)

Returns signed int or uint value of the encoding of a float:

Ti floatBitsToInt(Tf value)

Tu floatBitsToUint(Tf value)

Returns float value of a signed int or uint encoding of a float:

Tf intBitsToFloat(Ti value) Tf uintBitsToFloat(Tu value)

Computes and returns $a * b + c$. Treated as a single operation when using precise:

Tfd fma(Tfd a, Tfd b, Tfd c)

Splits *x* into a floating-point significand in the range [0.5, 1.0] and an integer exponent of 2:

Tfd frexp(Tfd x, out Ti exp)

Builds a floating-point number from *x* and the corresponding integral exponent of 2 in *exp*:

Tfd ldexp(Tfd x, in Ti exp)

Floating-Point Pack/Unpack [8.4]

These do not operate component-wise.

Converts each comp. of *v* into 8- or 16-bit ints, packs results into the returned 32-bit unsigned integer:

uint packUnorm2x16(vec2 v) uint packUnorm4x8(vec4 v)
uint packSnorm2x16(vec2 v) uint packSnorm4x8(vec4 v)

Unpacks 32-bit *p* into two 16-bit uints, four 8-bit uints, or signed ints. Then converts each component to a normalized float to generate a 2- or 4-component vector:

vec2 unpackUnorm2x16(uint p)
vec2 unpackSnorm2x16(uint p)
vec4 unpackUnorm4x8(uint p)
vec4 unpackSnorm4x8(uint p)

Packs components of *v* into a 64-bit value and returns a double-precision value:

double packDouble2x32(ivec2 v)

Returns a 2-component vector representation of *v*:

ivec2 unpackDouble2x32(double v)

Returns a uint by converting the components of a two-component floating-point vector:

uint packHalf2x16(vec2 v)

Returns a two-component floating-point vector:

vec2 unpackHalf2x16(uint v)

Built-In Functions

Geometric Functions [8.5]

These functions operate on vectors as vectors, not component-wise. Tf=float, vecn. Td=double, dvecn. Tfd= float, vecn, double, dvecn.

float length(Tf x)	length of vector
double length(Td x)	
float distance(Tf p0, Tf p1)	distance between points
double distance(Td p0, Td p1)	
float dot(Tf x, Tf y)	dot product
double dot(Td x, Td y)	
vec3 cross(vec3 x, vec3 y)	
dvec3 cross(dvec3 x, dvec3 y)	cross product
Tfd normalize(Tfd x)	normalize vector to length 1
Tfd faceforward(Tfd N, Tfd I, Tfd Nref)	returns N if dot(Nref, I) < 0, else -N
Tfd reflect(Tfd I, Tfd N)	reflection direction $I - 2 * \text{dot}(N, I) * N$
Tfd refract(Tfd I, Tfd N, float eta)	refraction vector

Matrix Functions [8.6]

N and M are 1, 2, 3, 4.

mat matrixCompMult(mat x, mat y)	component-wise multiply
dmat matrixCompMult(dmat x, dmat y)	
matN outerProduct(vecN c, vecN r)	outer product (where $N \neq M$)
dmatN outerProduct(dvecN c, dvecN r)	
matNxM outerProduct(vecM c, vecN r)	outer product
dmatNxM outerProduct(dvecM c, dvecN r)	
matN transpose(matN m)	
dmatN transpose(dmatN m)	transpose
matNxM transpose(matMxN m)	
dmatNxM transpose(dmatMxN m)	transpose (where $N \neq M$)
float determinant(matN m)	
double determinant(dmatN m)	determinant
matN inverse(matN m)	
dmatN inverse(dmatN m)	inverse

Vector Relational Functions [8.7]

Compare x and y component-wise. Sizes of the input and return vectors for any particular call must match. Tvec=vecn, ivecN, ivecN.

bvecn lessThan(Tvec x, Tvec y)	<
bvecn lessThanEqual(Tvec x, Tvec y)	<=
bvecn greaterThan(Tvec x, Tvec y)	>
bvecn greaterThanEqual(Tvec x, Tvec y)	>=
bvecn equal(Tvec x, Tvec y)	==
bvecn equal(bvecn x, bvecn y)	
bvecn notEqual(Tvec x, Tvec y)	!=
bvecn notEqual(bvecn x, bvecn y)	
bool any(bvecn x)	true if any component of x is true
bool all(bvecn x)	true if all comps. of x are true
bvecn not(bvecn x)	logical complement of x

Integer Functions [8.8]

Component-wise operation. Tu:uint, ivecN. Ti=int, ivecN. Tu=int, ivecN, uint, ivecN.

Adds 32-bit uint x and y, returning the sum modulo 2^{32} : Tu uaddCarry(Tu x, Tu y, out Tu carry)	
Subtracts y from x, returning the difference if non-negative, otherwise 2^{32} plus the difference: Tu usubBorrow(Tu x, Tu y, out Tu borrow)	

(Continued ...)

Integer Functions (cont.)

Multiples 32-bit integers x and y, producing a 64-bit result:

```
void umulExtended(Tu x, Tu y, out Tu msb, out Tu lsb)
void imulExtended(Ti x, Ti y, out Ti msb, out Ti lsb)
```

Extracts bits [offset, offset + bits - 1] from value, returns them in the least significant bits of the result:
Ti bitfieldExtract(Tiu value, int offset, int bits)

Returns the reversal of the bits of value:
Ti bitfieldReverse(Tiu value)

Inserts the bits least-significant bits of insert into base:
Ti bitfieldInsert(Tiu base, Tiu insert, int offset, int bits)

Returns the number of bits set to 1:
Ti bitCount(Tiu value)

Returns the bit number of the least significant bit:
Ti findLSB(Tiu value)

Returns the bit number of the most significant bit:
Ti findMSB(Tiu value)

```
int atomicOP(inout int mem, int data)
```

Image Functions [8.12]

In these image functions, IMAGE_PARAMS may be one of the following:

```
gimage1D image, int P
gimage2D image, ivec2 P
gimage3D image, ivec3 P
gimage2DRect image, ivec2 P
gimageCube image, ivec3 P
gimageBuffer image, int P
gimage1DArray image, ivec2 P
gimage2DArray image, ivec3 P
gimageCubeArray image, ivec3 P
gimage2DMs image, ivec2 P int sample
gimage2DMsArray image, ivec3 P, int sample
```

Returns the dimensions of the images or images:

```
int imageSize(gimage1D, Buffer) image
ivec2 imageSize(gimage2D, Cube, Rect, 1DArray,
2DMS) image
ivec3 imageSize(gimageCube, 2D, 2DMS) Array image
vec3 imageSize(gimage3D image)
```

Loads texel at the coordinate P from the image unit image:

```
gvec4 imageLoad(readonly IMAGE_PARAMS)
```

Stores data into the texel at the coordinate P from the image specified by image:

```
void imageStore(writeonly IMAGE_PARAMS, gvec4 data)
```

(Continued on next page >)

Atomic Counter Functions [8.10]

Returns the value of an atomic counter.

Atomically increments c then returns its prior value:
uint atomicCounterIncrement atomic_uint c)

Atomically decrements c then returns its prior value:
uint atomicCounterDecrement(atomic_uint c)

Atomically returns the counter for c:
uint atomicCounter(atomic_uint c)

Atomic Memory Functions [8.11]

Operates on individual integers in buffer-object or shared-variable storage. OP is Add, Min, Max, And, Or, Xor, Exchange, or CompSwap.

```
uint atomicOP(inout uint mem, uint data)
```

Built-In Functions (cont.)

Image Functions (cont.)

Adds the value of *data*

to the contents of the selected texel:

```
uint imageAtomicAdd(IMAGE_PARAMS, uint data)  
int imageAtomicAdd(IMAGE_PARAMS, int data)
```

Takes the minimum of the value of *data* and the contents of the selected texel:

```
uint imageAtomicMin(IMAGE_PARAMS, uint data)  
int imageAtomicMin(IMAGE_PARAMS, int data)
```

Takes the maximum of the value *data* and the contents of the selected texel:

```
uint imageAtomicMax(IMAGE_PARAMS, uint data)  
int imageAtomicMax(IMAGE_PARAMS, int data)
```

Performs a bit-wise AND of the value of *data* and the contents of the selected texel:

```
uint imageAtomicAnd(IMAGE_PARAMS, uint data)  
int imageAtomicAnd(IMAGE_PARAMS, int data)
```

Performs a bit-wise OR of the value of *data* and the contents of the selected texel:

```
uint imageAtomicOr(IMAGE_PARAMS, uint data)  
int imageAtomicOr(IMAGE_PARAMS, int data)
```

(Continue ↓)

Integer Functions (cont'd)

Performs a bit-wise exclusive OR of the value of *data* and the contents of the selected texel:

```
uint imageAtomicXor(IMAGE_PARAMS, uint data)  
int imageAtomicXor(IMAGE_PARAMS, int data)
```

Copies the value of *data*:

```
uint imageAtomicExchange(IMAGE_PARAMS, uint data)  
int imageAtomicExchange(IMAGE_PARAMS, int data)
```

Compares the value of *compare* and contents of selected texel. If equal, the new value is given by *data*; otherwise, it is taken from the original value loaded from texel:

```
uint imageAtomicCompSwap(IMAGE_PARAMS,  
    uint compare, uint data)  
int imageAtomicCompSwap(IMAGE_PARAMS, int compare,  
    int data)
```

Fragment Processing Functions [8.13]

Available only in fragment shaders.

Tf=float, vecn.

Derivative fragment-processing functions

Tf <code>dFdx(Tf p)</code>	derivative in x
Tf <code>dFdy(Tf p)</code>	derivative in y
Tf <code>fwidth(Tf p)</code>	sum of absolute derivative in x and y; <code>abs(dFdx(p)) + abs(dFdy(p));</code>

(Continue ↓)

Interpolation fragment-processing functions

Return value of *interpolant* sampled inside pixel and the primitive:

```
Tf interpolateAtCentroid(Tf interpolant)
```

Return value of *interpolant* at location of sample # *sample*:

```
Tf interpolateAtSample(Tf interpolant, int sample)
```

Return value of *interpolant* sampled at fixed offset *offset* from pixel center:

```
Tf interpolateAtOffset(Tf interpolant, vec2 offset)
```

Noise Functions [8.14]

Returns noise value. Available to fragment, geometry, and vertex shaders. *n* is 2, 3, or 4:

```
float noise1(Tf x)           vecn noisen(Tf x)
```

Geometry Shader Functions [8.15]

Only available in geometry shaders.

Emits values of output variables to current output primitive stream *stream*:

```
void EmitStreamVertex(int stream)
```

Completes current output primitive stream *stream* and starts a new one:

```
void EndStreamPrimitive(int stream)
```

(Continue ↓)

Geometry Shader Functions (cont'd)

Emits values of output variables to the current output primitive:

```
void EmitVertex()
```

Completes output primitive and starts a new one:

```
void EndPrimitive()
```

Other Shader Functions [8.16-17]

See diagram on page 11 for more information.

Synchronizes across shader invocations:

```
void barrier()
```

Controls ordering of memory transactions issued by a single shader invocation:

```
void memoryBarrier()
```

Controls ordering of memory transactions as viewed by other invocations in a compute work group:

```
void groupMemoryBarrier()
```

Order reads and writes accessible to other invocations:

```
void memoryBarrierAtomicCounter()
```

```
void memoryBarrierShared()
```

```
void memoryBarrierBuffer()
```

```
void memoryBarrierImage()
```

GLSL 4.3 프로그램 작성 예 1

-- OpenGL 3D Geometric Transformations



Shaders: Just Trivial

<Vertex Shader>

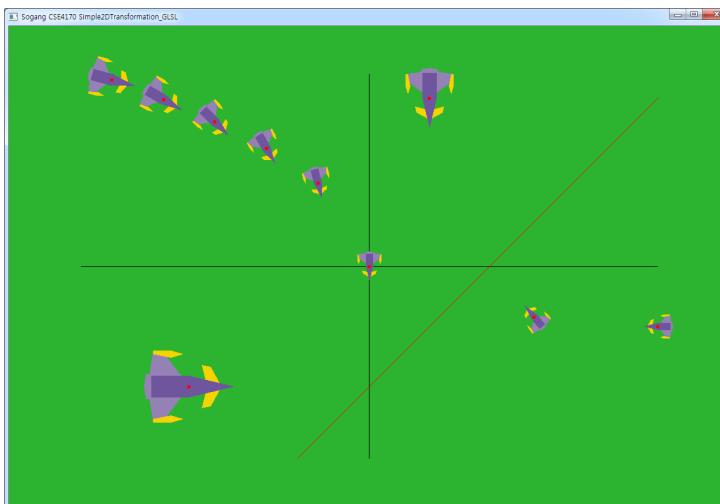
```
uniform mat4 u_ModelViewProjectionMatrix;  
uniform vec3 u_primitive_color;  
  
layout (location = 0) in vec4 a_position;  
out vec4 v_color;  
  
void main(void) {  
    v_color = vec4(u_primitive_color, 1.0f);  
    gl_Position = u_ModelViewProjectionMatrix * a_position;  
}
```

단순히 CC로 기하 변환 수행

<Fragment Shader>

```
in vec4 v_color;  
  
layout (location = 0) out vec4 final_color;  
  
void main(void) {  
    final_color = v_color;  
}
```

단순히 현재 설정된 색깔을 사용



Drawing a Frame

```
void display(void) {  
    int i;  
    float x, r, s, delx, delr, dels;  
    glm::mat4 ModelMatrix;  
    glClear(GL_COLOR_BUFFER_BIT);  
  
    ModelMatrix = glm::mat4(1.0f);  
    ModelViewProjectionMatrix = ViewProjectionMatrix * ModelMatrix;  
    glUniformMatrix4fv(loc_ModelViewProjectionMatrix, 1, GL_FALSE, &ModelViewProjectionMatrix[0][0]);  
    draw_axes();  
    draw_line();  
    draw_airplane();  
  
    ModelMatrix = glm::translate(glm::mat4(1.0f), glm::vec3(centerx, centery, 0.0f));  
    ModelMatrix = glm::rotate(ModelMatrix, rotate_angle, glm::vec3(0.0f, 0.0f, 1.0f));  
    ModelViewProjectionMatrix = ViewProjectionMatrix * ModelMatrix;  
    glUniformMatrix4fv(loc_ModelViewProjectionMatrix, 1, GL_FALSE, &ModelViewProjectionMatrix[0][0]);  
    draw_airplane(); // 0  
  
    ModelMatrix = glm::translate(glm::mat4(1.0f), glm::vec3(-win_width / 4.0f, -win_height / 4.0f, 0.0f));  
    ModelMatrix = glm::rotate(ModelMatrix, 90.0f*TO_RADIAN, glm::vec3(0.0f, 0.0f, 1.0f));  
    ModelMatrix = glm::scale(ModelMatrix, glm::vec3(3.0f, 3.0f, 1.0f));  
    ModelViewProjectionMatrix = ViewProjectionMatrix * ModelMatrix;  
    glUniformMatrix4fv(loc_ModelViewProjectionMatrix, 1, GL_FALSE, &ModelViewProjectionMatrix[0][0]);  
    draw_airplane(); // 1
```

```
ModelMatrix = glm::translate(glm::mat4(1.0f), glm::vec3(win_width / 2.5f, -win_height / 8.0f, 0.0f));
ModelMatrix = glm::rotate(ModelMatrix, 270.0f*TO_RADIAN, glm::vec3(0.0f, 0.0f, 1.0f));
ModelViewProjectionMatrix = ViewProjectionMatrix * ModelMatrix;
glUniformMatrix4fv(loc_ModelViewProjectionMatrix, 1, GL_FALSE, &ModelViewProjectionMatrix[0][0]);
draw_airplane(); // 2
```

```
ModelMatrix = glm::translate(glm::mat4(1.0f), glm::vec3(win_height / 4.0f, 0.0f, 0.0f));
ModelMatrix = glm::rotate(ModelMatrix, 45.0f*TO_RADIAN, glm::vec3(0.0f, 0.0f, 1.0f));
ModelMatrix = glm::scale(ModelMatrix, glm::vec3(1.0f, -1.0f, 1.0f));
ModelMatrix = glm::rotate(ModelMatrix, -45.0f*TO_RADIAN, glm::vec3(0.0f, 0.0f, 1.0f));
ModelMatrix = glm::translate(ModelMatrix, glm::vec3(-win_height / 4.0f, 0.0f, 0.0f));
ModelMatrix = glm::translate(ModelMatrix, glm::vec3(win_width / 2.5f, -win_height / 8.0f, 0.0f));
ModelMatrix = glm::scale(ModelMatrix, glm::vec3(2.0f, 2.0f, 1.0f));
ModelMatrix = glm::translate(ModelMatrix, glm::vec3(-win_width / 2.5f, win_height / 8.0f, 0.0f));
```

```
ModelMatrix = glm::translate(ModelMatrix, glm::vec3(win_width / 2.5f, -win_height / 8.0f, 0.0f));
ModelMatrix = glm::rotate(ModelMatrix, 270.0f*TO_RADIAN, glm::vec3(0.0f, 0.0f, 1.0f));
ModelViewProjectionMatrix = ViewProjectionMatrix * ModelMatrix;
glUniformMatrix4fv(loc_ModelViewProjectionMatrix, 1, GL_FALSE, &ModelViewProjectionMatrix[0][0]);
draw_airplane(); // 3
```

```

delx = win_width/14.0f; delr = 15.0f; dels = 1.1f;
x = -delx; r = delr; s = dels;
for (i = 0; i < 5; i++, x -= delx, r += delr, s *= dels) {
    ModelMatrix = glm::translate(glm::mat4(1.0f), glm::vec3(x, 15.0f*sqrtf(-x), 0.0f));
    ModelMatrix = glm::rotate(ModelMatrix, r*TO_RADIAN, glm::vec3(0.0f, 0.0f, 1.0f));
    glTranslatef(x, 15.0f*sqrtf(-x), 0.0f);
    ModelMatrix = glm::scale(ModelMatrix, glm::vec3(s, s, 1.0f));
    ModelViewProjectionMatrix = ViewProjectionMatrix * ModelMatrix;
    glUniformMatrix4fv(loc_ModelViewProjectionMatrix, 1, GL_FALSE, &ModelViewProjectionMatrix[0][0]);
    draw_airplane(); // 4
}
glFlush();
}

```



OpenGL Mathematics (GLM)

- GLM is a header only C++ mathematics library for graphics software based on the GLSL
- To use GLM, a programmer only has to include `<glm/glm.hpp>`
- GLM extends the core GLSL feature set with extensions. These extensions include : quaternion, transformation, spline, matrix inverse, color spaces, etc.
- GLM replacements for deprecated OpenGL functions

```
glRotate{f, d}:
glm::mat4 glm::rotate(
    glm::mat4 const & m,
    float angle,
    glm::vec3 const & axis);

glm::dmat4 glm::rotate(
    glm::dmat4 const & m,
    double angle,
    glm::dvec3 const & axis);
```

From `GLM_GTC_matrix_transform` extension: `<glm/gtc/matrix_transform.hpp>`

OpenGL Mathematics (GLM)

```
glScale{f, d}:
glm::mat4 glm::scale(
    glm::mat4 const & m,
    glm::vec3 const & factors);

glm::dmat4 glm::scale(
    glm::dmat4 const & m,
    glm::dvec3 const & factors);
```

From GLM_GTC_matrix_transform extension: <glm/gtc/matrix_transform.hpp>

```
glTranslate{f, d}:
glm::mat4 glm::translate(
    glm::mat4 const & m,
    glm::vec3 const & translation);

glm::dmat4 glm::translate(
    glm::dmat4 const & m,
    glm::dvec3 const & translation);
```

From GLM_GTC_matrix_transform extension: <glm/gtc/matrix_transform.hpp>

```
glLoadIdentity:
glm::mat4(1.0) or glm::mat4();
glm::dmat4(1.0) or glm::dmat4();
```

From GLM core library: <glm/glm.hpp>

```
glMultMatrix{f, d}:
glm::mat4() * glm::mat4();
glm::dmat4() * glm::dmat4();
```

From GLM core library: <glm/glm.hpp>

OpenGL Mathematics (GLM)

glLoadTransposeMatrix{f, d}:

```
glm::transpose(glm::mat4());
glm::transpose(glm::dmat4());
```

From GLM core library: <glm/glm.hpp>

glMultTransposeMatrix{f, d}:

```
glm::mat4() * glm::transpose(glm::mat4());
glm::dmat4() * glm::transpose(glm::dmat4());
```

From GLM core library: <glm/glm.hpp>

glFrustum:

```
glm::mat4 glm::frustum(
    float left, float right,
    float bottom, float top,
    float zNear, float zFar);
```

```
glm::dmat4 glm::frustum(
    double left, double right,
    double bottom, double top,
    double zNear, double zFar);
```

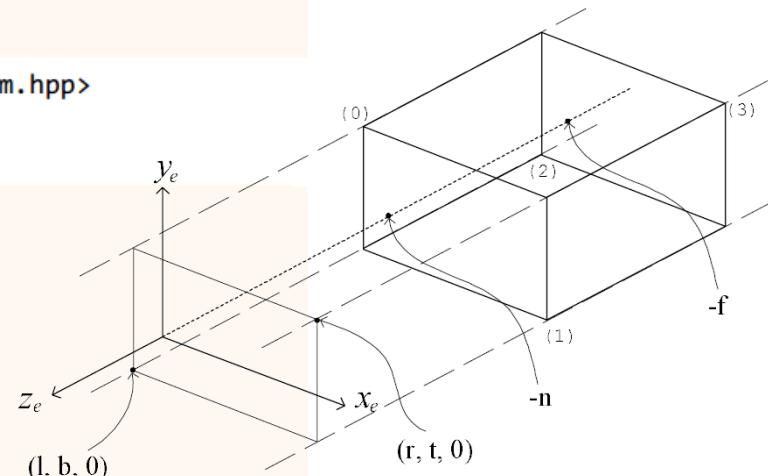
From GLM_GTC_matrix_transform extension: <glm/gtc/matrix_transform.hpp>

glOrtho:

```
glm::mat4 glm::ortho(
    float left, float right,
    float bottom, float top,
    float zNear, float zFar);
```

```
glm::dmat4 glm::ortho(
    double left, double right,
    double bottom, double top,
    double zNear, double zFar);
```

From GLM_GTC_matrix_transform extension: <glm/gtc/matrix_transform.hpp>



OpenGL Mathematics (GLM)

- GLM replacements for GLU functions

gluLookAt:

```
glm::mat4 glm::lookAt(
    glm::vec3 const & eye,
    glm::vec3 const & center,
    glm::vec3 const & up);

glm::dmat4 glm::lookAt(
    glm::dvec3 const & eye,
    glm::dvec3 const & center,
    glm::dvec3 const & up);
```

From GLM_GTC_matrix_transform extension: <glm/gtc/matrix_transform.hpp>

gluOrtho2D:

```
glm::mat4 glm::ortho(
    float left, float right, float bottom, float top);

glm::dmat4 glm::ortho(
    double left, double right, double bottom, double top);
```

From GLM_GTC_matrix_transform extension: <glm/gtc/matrix_transform.hpp>

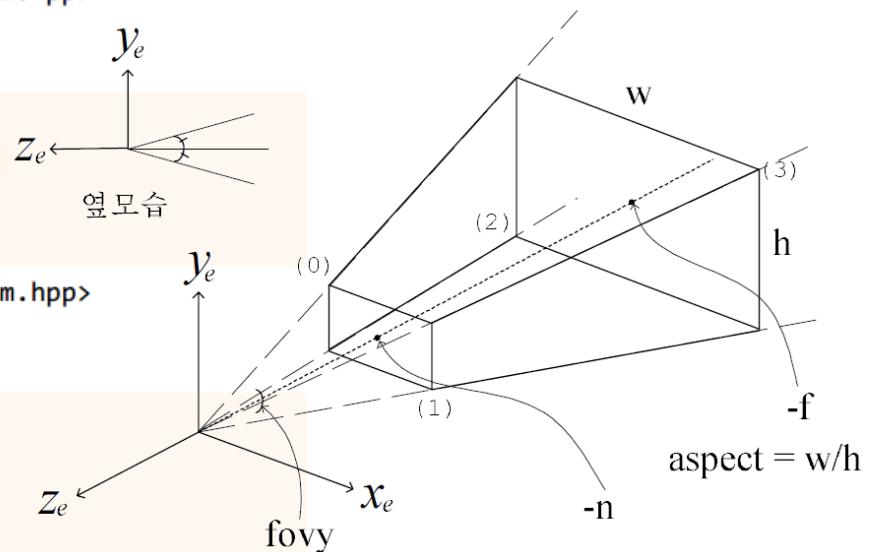
gluPerspective:

```
glm::mat4 perspective(
    float fovy, float aspect, float zNear, float zFar);

glm::dmat4 perspective(
    double fovy, double aspect, double zNear, double zFar);
```

One difference between GLM and GLU is that fovy is expressed in radians in GLM instead of degrees.

From GLM_GTC_matrix_transform extension: <glm/gtc/matrix_transform.hpp>



OpenGL Mathematics (GLM)

gluPickMatrix:

```
glm::mat4 pickMatrix(
    glm::vec2 const & center,
    glm::vec2 const & delta,
    glm::ivec4 const & viewport);

glm::dmat4 pickMatrix(
    glm::dvec2 const & center,
    glm::dvec2 const & delta,
    glm::ivec4 const & viewport);
```

From GLM_GTC_matrix_transform extension: <glm/gtc/matrix_transform.hpp>

gluProject:

```
glm::vec3 project(
    glm::vec3 const & obj,
    glm::mat4 const & model,
    glm::mat4 const & proj,
    glm::{i, ' '}vec4 const & viewport);

glm::dvec3 project(
    glm::dvec3 const & obj,
    glm::dmat4 const & model,
    glm::dmat4 const & proj,
    glm::{i, ' ', d}vec4 const & viewport);
```

From GLM_GTC_matrix_transform extension: <glm/gtc/matrix_transform.hpp>

gluUnProject:

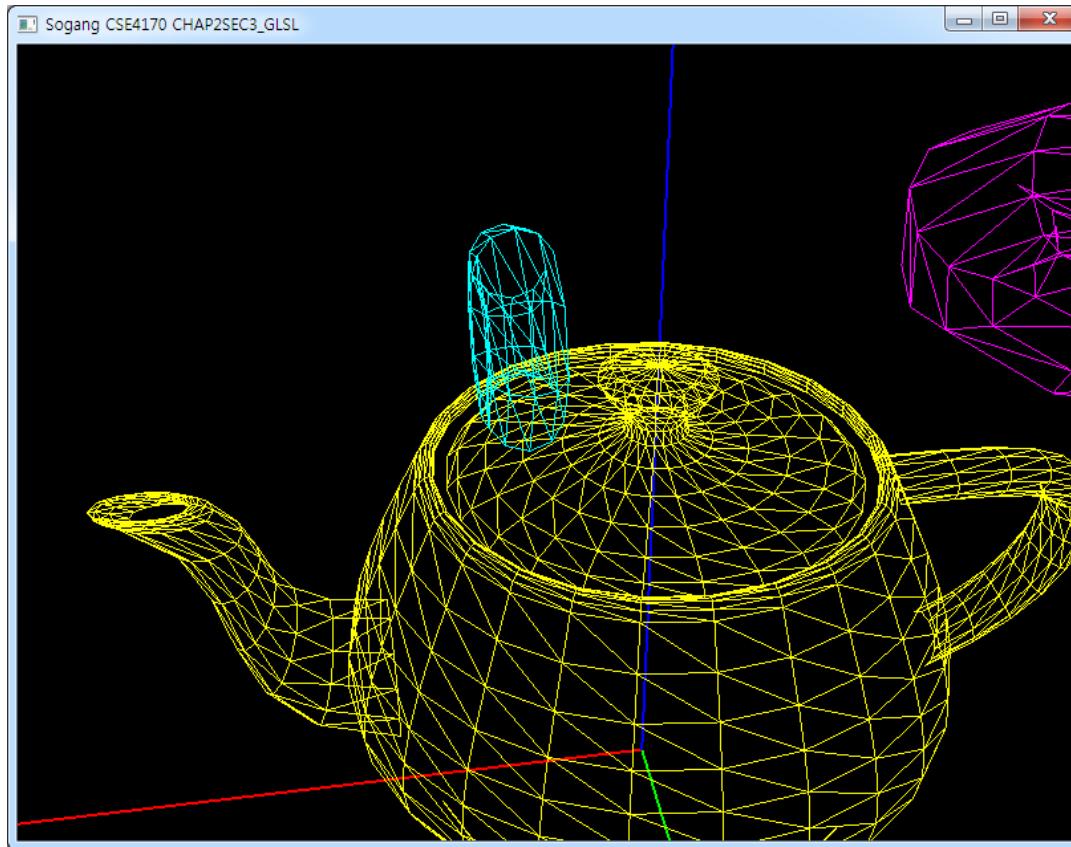
```
glm::vec3 unProject(
    glm::vec3 const & win,
    glm::mat4 const & model,
    glm::mat4 const & proj,
    glm::{i, ' '}vec4 const & viewport);

glm::dvec3 unProject(
    glm::dvec3 const & win,
    glm::dmat4 const & model,
    glm::dmat4 const & proj,
    glm::{i, ' ', d}vec4 const & viewport);
```

From GLM_GTC_matrix_transform extension: <glm/gtc/matrix_transform.hpp>

GLSL 4.3 프로그램 작성 예 2

-- Simple OpenGL 3D Viewing



Shaders: Just Trivial

<Vertex Shader>

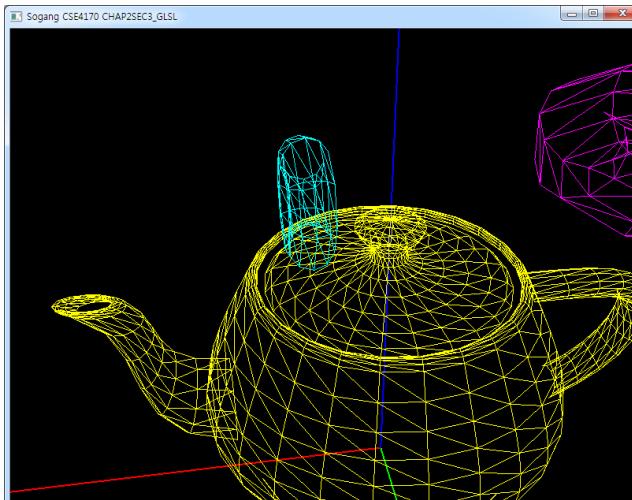
```
uniform mat4 u_ModelViewProjectionMatrix;  
uniform vec3 u_primitive_color;  
  
layout (location = 0) in vec4 a_position;  
out vec4 v_color;  
  
void main(void) {  
    v_color = vec4(u_primitive_color, 1.0f);  
    gl_Position = u_ModelViewProjectionMatrix * a_position;  
}
```

단순히 CC로 기하 변환 수행

<Fragment Shader>

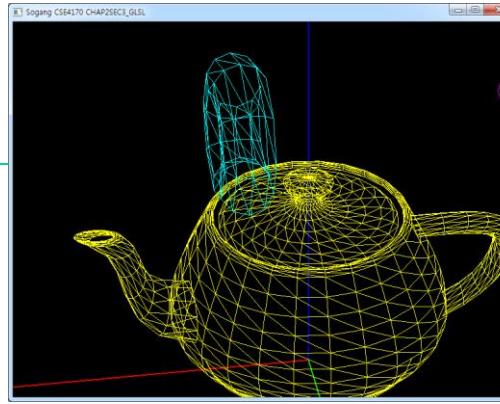
```
in vec4 v_color;  
  
layout (location = 0) out vec4 final_color;  
  
void main(void) {  
    final_color = v_color;  
}
```

단순히 현재 설정된 색깔을 사용

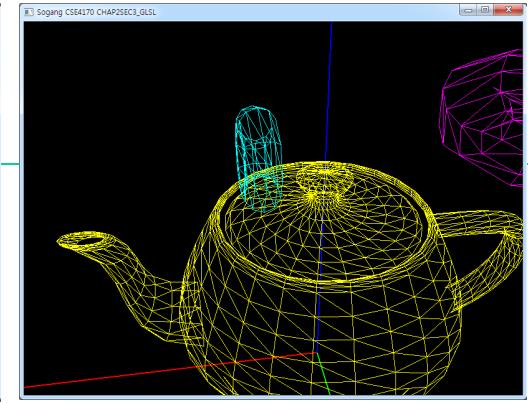


Keyboard callback

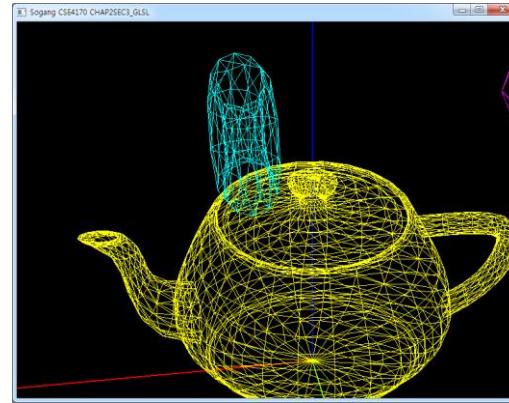
```
void keyboard(unsigned char key, int x, int y) {  
    switch (key) {  
        case 'q': // Incur destruction callback for cleanups.  
            glutLeaveMainLoop();  
            break;  
        case 'o':  
            // orthographic projection  
            ProjectionMatrix = glm::ortho(-2.7f*window_aspect_ratio, 2.7f*window_aspect_ratio, -2.7f, 2.7f, 5.0f, 19.0f);  
            glutPostRedisplay();  
            break;  
        case 'p':  
            // perspective projection  
            ProjectionMatrix = glm::perspective(28.0f*TO_RADIAN, window_aspect_ratio, 5.0f, 19.0f);  
            glutPostRedisplay();  
            break;  
        case 'c':  
            cull_face_mode = 1 - cull_face_mode;  
            if (cull_face_mode)  
                glEnable(GL_CULL_FACE); // turn the face-culling feature on  
            else  
                glDisable(GL_CULL_FACE); // turn the face-culling feature off  
            glutPostRedisplay();  
            break;  
    }  
}
```



orthographic projection



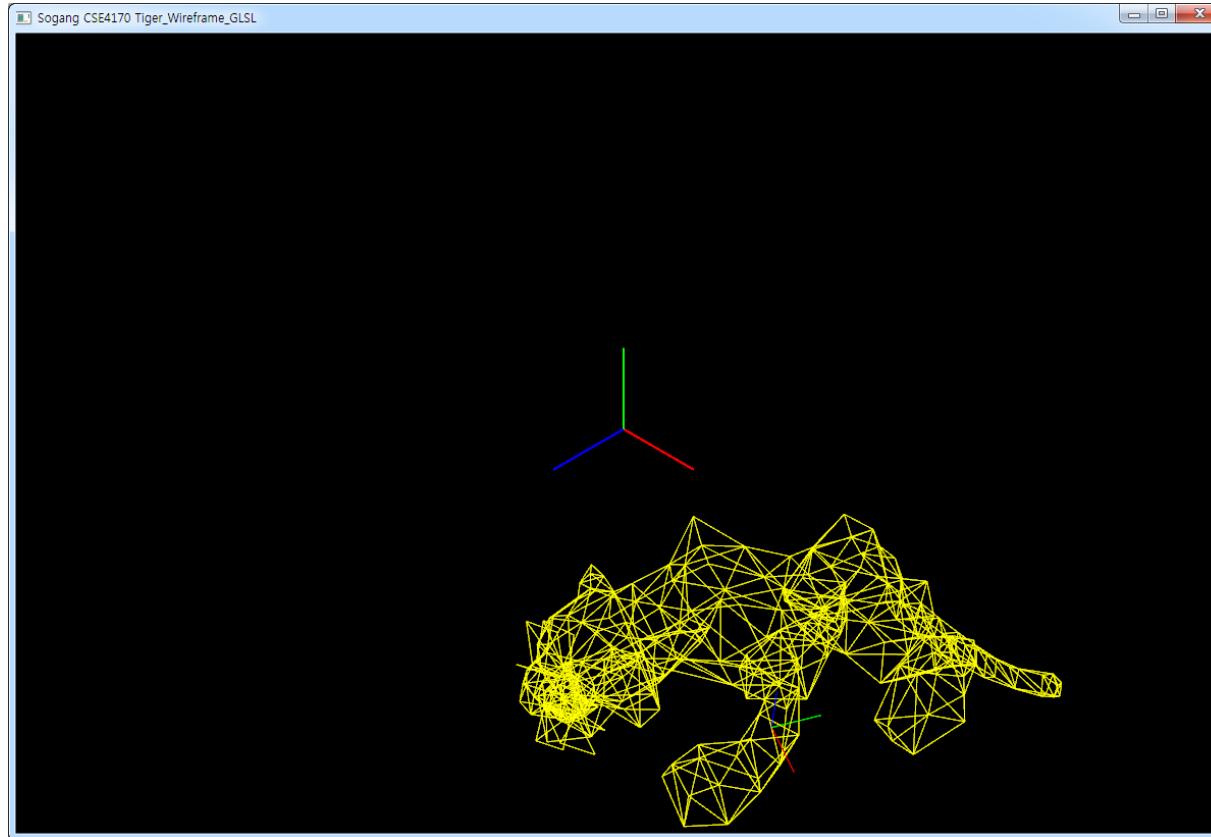
Perspective projection



glDisable(GL_CULL_FACE)

GLSL 4.3 프로그램 작성 예 3

-- Simple OpenGL 3D Viewing



Shaders: Just Trivial

<Vertex Shader>

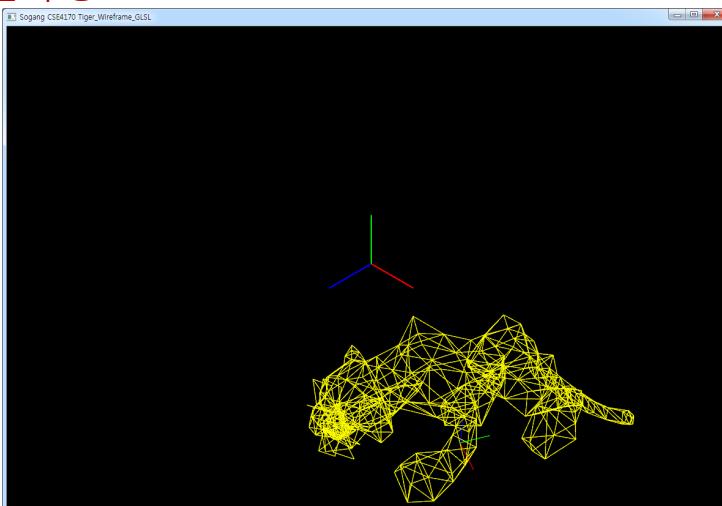
```
uniform mat4 u_ModelViewProjectionMatrix;  
uniform vec3 u_primitive_color;  
  
layout (location = 0) in vec4 a_position;  
out vec4 v_color;  
  
void main(void) {  
    v_color = vec4(u_primitive_color, 1.0f);  
    gl_Position = u_ModelViewProjectionMatrix * a_position;  
}
```

단순히 CC로 기하 변환 수행

<Fragment Shader>

```
in vec4 v_color;  
  
layout (location = 0) out vec4 final_color;  
  
void main(void) {  
    final_color = v_color;  
}
```

단순히 현재 설정된 색깔을 사용



Entering main()

```
void main ( int argc, char *argv[] ) {  
    ...  
    // initialize the freeGLUT library  
    glutInit(&argc, argv);  
    // configures the type of window we want to use  
    // window use the RGBA color space, double-buffering to clean up our display and multisampling  
    glutInitDisplayMode(GLUT_RGBA | GLUT_DOUBLE | GLUT_MULTISAMPLE);  
    // specifies the size of the window  
    glutInitWindowSize(1200, 800);  
    // specifies the type of OpenGL context  
    // request an OpenGL Version 4.3 core profile  
    glutInitContextVersion(4, 3);  
    glutInitContextProfile(GLUT_CORE_PROFILE);  
    // create a window matching the display mode  
    glutCreateWindow(program_name);  
    ...  
    initialize_renderer();  
    // specify what happens when a user closes a window  
    // the application exits the main loop when the window is closed and execution continues  
    glutSetOption(GLUT_ACTION_ON_WINDOW_CLOSE, GLUT_ACTION_GLUTMAINLOOP_RETURNS);  
    // infinite loop  
    glutMainLoop();  
}
```

```
void initialize_renderer(void) {  
    register_callbacks();  
    prepare_shader_program();  
    initialize_OpenGL();  
    prepare_scene();  
}
```

```

void register_callbacks(void) {
    // call display() when it's time to render another frame
    glutDisplayFunc(display);
    // call keyboard() when a key is pressed on the keyboard
    glutKeyboardFunc(keyboard);
    // call reshape() when the program is first started, and window is resized
    glutReshapeFunc(reshape);
    // call timer_scene(0) in at least 100 milliseconds
    glutTimerFunc(100, timer_scene, 0);
    // call cleanup() when a window is closed
    glutCloseFunc(cleanup);
}

```

```

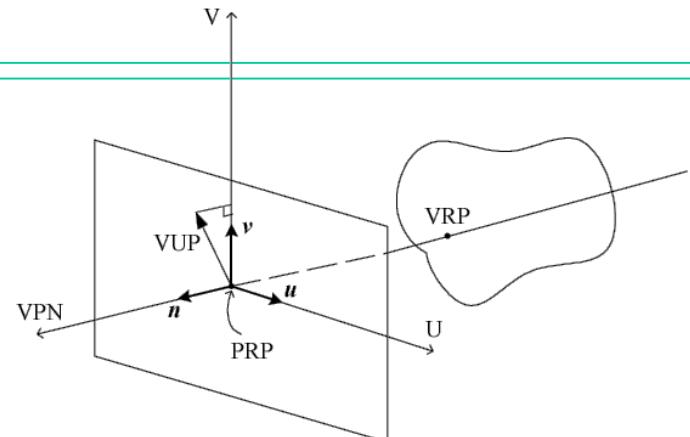
void initialize_OpenGL(void) {
    // activate multisampling
    glEnable(GL_MULTISAMPLE);
    // select a polygon rasterization mode
    glPolygonMode(GL_FRONT_AND_BACK, GL_LINE);
    // clear framebuffer to black
    glClearColor(0.0f, 0.0f, 0.0f, 1.0f);
    // set the view matrix
    ViewMatrix = glm::lookAt(glm::vec3(300.0f, 300.0f, 300.0f), glm::vec3(0.0f, 0.0f, 0.0f), glm::vec3(0.0f, 1.0f, 0.0f));
}

```

```

void timer_scene(int timestamp_scene) {
    cur_frame_tiger = timestamp_scene % N_TIGER_FRAMES;
    rotation_angle_tiger = (timestamp_scene % 360)*TO_RADIAN;
    glutPostRedisplay(); // flag for redraw
    glutTimerFunc(100, timer_scene, (timestamp_scene + 1) % INT_MAX);
}

```



Initialization

```
GLuint axes_VBO, axes_VAO;  
GLfloat axes_vertices[6][3] = { { 0.0f, 0.0f, 0.0f }, { 1.0f, 0.0f, 0.0f }, { 0.0f, 0.0f, 0.0f },  
                               { 0.0f, 1.0f, 0.0f }, { 0.0f, 0.0f, 0.0f }, { 0.0f, 0.0f, 1.0f } };  
GLfloat axes_color[3][3] = { { 1.0f, 0.0f, 0.0f }, { 0.0f, 1.0f, 0.0f }, { 0.0f, 0.0f, 1.0f } };  
void prepare_axes(void) { // Draw coordinate axes.  
    // Initialize vertex buffer object.  
    glGenBuffers(1, &axes_VBO);  
  
    glBindBuffer(GL_ARRAY_BUFFER, axes_VBO);  
    glBufferData(GL_ARRAY_BUFFER, sizeof(axes_vertices), &axes_vertices[0][0], GL_STATIC_DRAW);  
  
    // Initialize vertex array object.  
    glGenVertexArrays(1, &axes_VAO);  
    glBindVertexArray(axes_VAO);  
  
    glBindBuffer(GL_ARRAY_BUFFER, axes_VBO); // binds axes_VBO to the GL_ARRAY_BUFFER  
    // attribute index 0 gets its vertex array data from axes_VBO  
    glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 0, BUFFER_OFFSET(0));  
  
    glEnableVertexAttribArray(0);  
    glBindBuffer(GL_ARRAY_BUFFER, 0);  
    glBindVertexArray(0);  
}
```

Vertex Array Object and Vertex Buffer Object

- Vertex Array Object
 - OpenGL Object that stores all of the state needed to supply vertex data
 - Usual creation, destruction, and binding functions : **glGenVertexArrays**, **glDeleteVertexArrays** and **glBindVertexArray**
 - Array access is enabled and disabled by binding the VAO

```
void glEnableVertexAttribArray(GLuint index);
void glDisableVertexAttribArray(GLuint index);
```

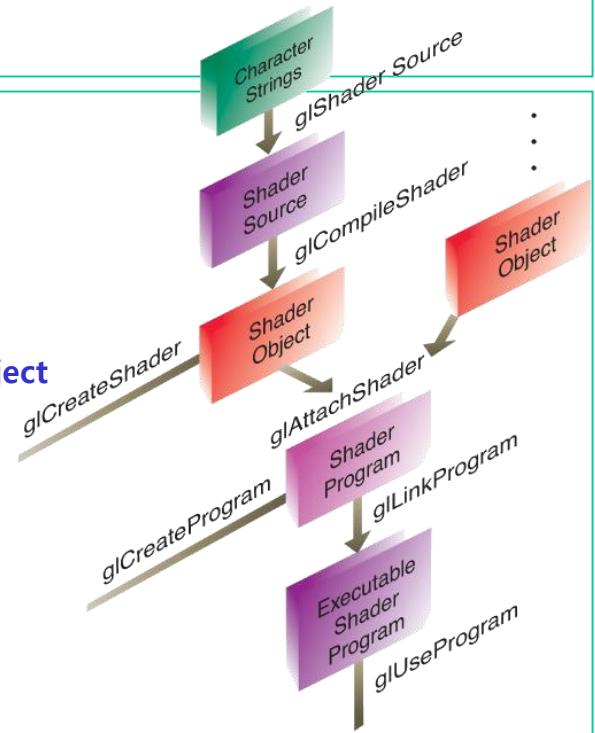
Specifies that the vertex array associated with variable *index* be enabled or disabled. *index* must be a value between zero and GL_MAX_VERTEX_ATTRIBS – 1.

- Vertex Buffer Object
 - Buffer object which is used as the source for vertex array data

Compiling shaders

```
void prepare_shader_program(void) {  
    ShaderInfo shader_info[3] = { { GL_VERTEX_SHADER, "Shaders/simple.vs.glsl" },  
                                { GL_FRAGMENT_SHADER, "Shaders/simple.fs.glsl" }, { GL_NONE, NULL } };  
    h_ShaderProgram = LoadShaders(shader_info);  
    glUseProgram(h_ShaderProgram); // use the linked shader program  
  
    loc_ModelViewProjectionMatrix = glGetUniformLocation(h_ShaderProgram, "u_ModelViewProjectionMatrix");  
    loc_primitive_color = glGetUniformLocation(h_ShaderProgram, "u_primitive_color");  
}
```

```
GLuint LoadShaders( ShaderInfo* shaders ) {  
    ...  
    GLuint program = glCreateProgram(); // create a shader object  
    ...  
    const GLchar* source = ReadShader( entry->filename );  
    glShaderSource( shader, 1, &source, NULL ); // associate the source code with the object  
    glCompileShader( shader ); // compile a shader object's source  
    glGetShaderiv( shader, GL_COMPILE_STATUS, &compiled );  
    glAttachShader( program, shader ); // attach a shader object to the program  
    ...  
    glLinkProgram( program ); // link the objects for an executable program  
    glGetProgramiv( program, GL_INFO_LOG_LENGTH, &len );  
    ...  
}
```



© OpenGL Programming Guide by Dave Shreiner et al.

Drawing a Frame

```
void display(void) {  
    glClear(GL_COLOR_BUFFER_BIT);  
  
    ModelViewMatrix = glm::scale(ViewMatrix, glm::vec3(50.0f, 50.0f, 50.0f));  
    ModelViewProjectionMatrix = ProjectionMatrix * ModelViewMatrix;  
    glUniformMatrix4fv(loc_ModelViewProjectionMatrix, 1, GL_FALSE, &ModelViewProjectionMatrix[0][0]);  
    glLineWidth(2.0f);  
    draw_axes();  
    glLineWidth(1.0f);  
    ...  
    glutSwapBuffers();  
}
```

```
void draw_axes(void) {  
    glBindVertexArray(axes_VAO);  
    glUniform3fv(loc_primitive_color, 1, axes_color[0]);  
    glDrawArrays(GL_LINES, 0, 2); // sends vertex data to the OpenGL pipeline  
    glUniform3fv(loc_primitive_color, 1, axes_color[1]);  
    glDrawArrays(GL_LINES, 2, 2);  
    glUniform3fv(loc_primitive_color, 1, axes_color[2]);  
    glDrawArrays(GL_LINES, 4, 2);  
    glBindVertexArray(0);  
}
```

```
void glDrawArrays(GLenum mode, GLint first, GLsizei count);
```

Constructs a sequence of geometric primitives using the elements from the currently bound vertex array starting at *first* and ending at *first + count - 1*. *mode* specifies what kinds of primitives are constructed and is one of GL_POINTS, GL_LINES, GL_LINE_STRIP, GL_LINE_LOOP, GL_TRIANGLES, GL_TRIANGLE_STRIP, GL_TRIANGLE_FAN, and GL_PATCHES.