

Open GL

-  OpenGL 3.3 Basics
- Basic OpenGL Pipeline
- OpenGL Shading Language (GLSL) Basics
- Viewing Pipeline

OpenGL Basics

- OpenGL : 여러 사람들이 공통으로 만든 graphic API에 대한 규약
 - OpenGL은 C 언어 기반이지만 sample code는 C++
- 강의에서는 3.3 버전 core를 사용
 - 3.3 버전은 backward compatibility가 있어 3.3 버전을 배워도 최신 버전 쓰는데 문제 없음
- OpenGL extensions
 - 개발한 graphic feature를 ARB(Architecture Review Board) 승인을 받기 전 사용할 수 있도록 제공하는 방법
 - 문제점) OpenGL Core에 없어서 하우스 키핑 과정이 복잡함 --> function/low level call을 여러 번 해야 함
- State machine (OpenGL context)
 - 한번 state를 저장하면 바꾸기 전까지 뒷 코드에도 state가 영향을 미침

OS-specific Support Libs

- OpenGL은 어떤 운영체계에서도 사용 가능하기 때문에 운영체계 의존적인 일을 해줄 방법이 필요함
- Window, canvas 만들어야 그림을 그릴 수 있는데 이것은 OpenGL이 해주지 않음
- 과거에는 이것들을 직접 코딩했지만, 지금은 support library를 사용하면 됨
→ GLFW, GLAD

GLFW

1. **Window 생성:** 화면 크기 조정, 모니터와 관련된 기능 제공
2. **User Interface 이벤트 처리:** 사용자 인터페이스(키보드 입력, 드래그, ...) 관련 이벤트 처리 기능 제공

GLAD

1. OpenGL 함수와 Extension을 사용할 때 필요한 Low level 작업

Hello Window

-  window를 만들어보자.



-  1. Header (GLFW/GLAD)
- GLFW, GLAD 헤더를 포함

```
#include <glad/glad.h>
#include <GLFW/glfw3.h>

#include <iostream>
void framebuffer_size_callback(GLFWwindow* window, int width, int height);
void processInput(GLFWwindow *window);

// settings
const unsigned int SCR_WIDTH = 800;
const unsigned int SCR_HEIGHT = 600;
```

Header (GLFW/GLAD)

-  2. GLFW init

```

const unsigned int SCR_WIDTH = 800;
const unsigned int SCR_HEIGHT = 600;

int main()
{
    // glfw: initialize and configure
    // -----
    glfwInit();
    glfwWindowHint(GLFW_CONTEXT_VERSION_MAJOR, 3); // openGL core version 3.3
    glfwWindowHint(GLFW_CONTEXT_VERSION_MINOR, 3);
    glfwWindowHint(GLFW_OPENGL_PROFILE, GLFW_OPENGL_CORE_PROFILE);

#ifndef __APPLE__
    glfwWindowHint(GLFW_OPENGL_FORWARD_COMPAT, GL_TRUE); // uncomment this statement to fix compilation on OS X
#endif

    // glfw window creation
    // -----
    GLFWwindow* window = glfwCreateWindow(SCR_WIDTH, SCR_HEIGHT, "LearnOpenGL", NULL, NULL); Window 생성
    if (window == NULL) Window 생성됐는지 확인
    {
        std::cout << "Failed to create GLFW window" << std::endl;
        glfwTerminate();
        return -1;
    } 현재 Window Target 이라 선언
    glfwMakeContextCurrent(window);
    glfwSetFramebufferSizeCallback(window, framebuffer_size_callback);
}

```

GLFW init

GLFW init

1. Window 생성
2. 이벤트 메시지 처리

Framebuffer 처음 생성 시, jump 해서 이벤트 처리

```

// glfw: whenever the window size changed (by OS or user resize) this callback function executes
// -----
void framebuffer_size_callback(GLFWwindow* window, int width, int height)
{
    // make sure the viewport matches the new window dimensions; note that width and
    // height will be significantly larger than specified on retina displays.
    glViewport(0, 0, width, height);
}

```

Window Callback

Viewport 생성

- Viewport의 width, height는 frame buffer 사이즈와 동일

3. GLAD init + 4. Render Loop

```

// glad: load all OpenGL function pointers
// -----
if (!gladLoadGLLoader((GLADloadproc)glfwGetProcAddress))
{
    std::cout << "Failed to initialize GLAD" << std::endl;
    return -1;
}

// render loop
// -----
while (!glfwWindowShouldClose(window)) Window close 될 때까지 무한 루프
{
    // input
    // -----
    processInput(window); 1. Interface input 처리

    // render
    // -----
    glClearColor(0.2f, 0.3f, 0.3f, 1.0f); 2. Rendering
    glClear(GL_COLOR_BUFFER_BIT); 0-1 사이 Normalized 된 Color 값으로 clear

    // glfw: swap buffers and poll IO events (keys pressed/released, mouse moved etc.)
    // -----
    glfwSwapBuffers(window); 3. Double buffering (swapping)
    glfwPollEvents();
}

// glfw: terminate, clearing all previously allocated GLFW resources.
// -----
glfwTerminate(); Window terminate 하면서 끝
return 0;
}

```

GLAD init

GLAD init

- 여러 function pointer 들 불러옴

Render Loop

Render Loop

- 무엇을 렌더링 할지 정하는 부분으로, 우리가 중점적으로 신경 써야 하는 부분!
- 과정
 1. 사용자 input 처리
 2. Rendering
 3. Double buffering

* 지금까지의 코드들은 대부분 기계적으로 들어가야 하는 내용으로 바꿀 일이 별로 없음

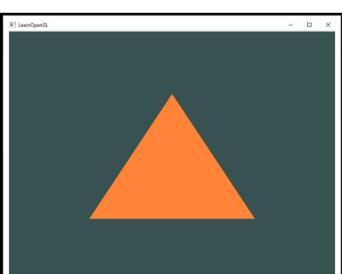
```

// process all input: query GLFW whether relevant keys are pressed/released this frame and react accordingly
// -----
void processInput(GLFWwindow *window)
{
    if(glfwGetKey(window, GLFW_KEY_ESCAPE) == GLFW_PRESS) Escape 키 → true: close (Loop 정지)
}

```

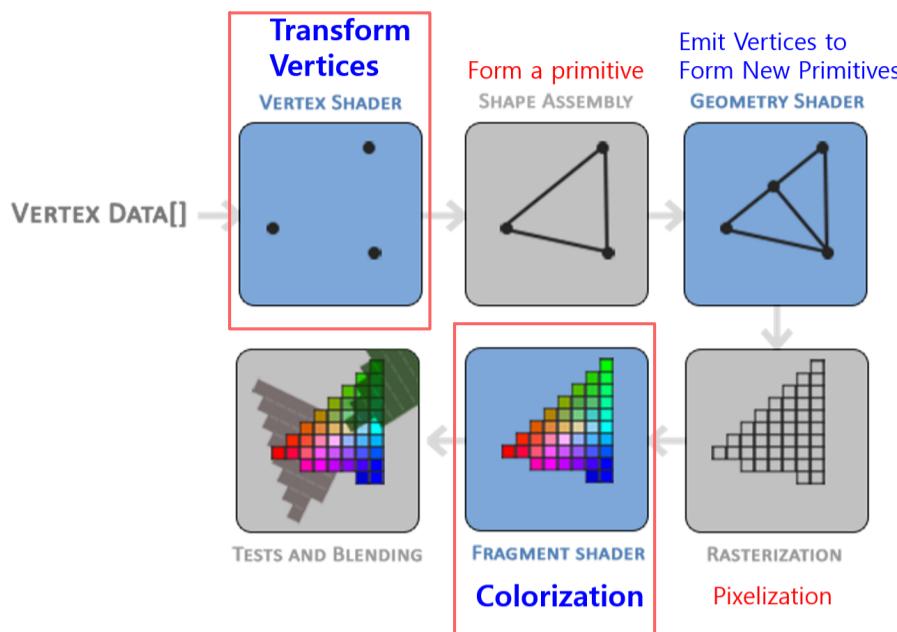
Hello Triangle

 삼각형을 그려보자 → OpenGL graphics pipeline에 대해 알아야 함.



OpenGL Graphics Pipeline

- Basic concept에서 배운 pipeline 보다 세분화 되어 있음.
- Input은 3차원 꼭짓점 데이터로, position 외의 color, texture data도 들어갈 수 있음.
- 파란색 부분은 shader (**OpenGL Shading Language, GLSL**)를 사용해서 사용자가 구현



1. Vertex shader

- Transform vertices
- Modeling, viewing, projection, viewport transformation이 발생
- 각 vertex에 개별적, 병렬적으로 적용
- 점 정보로는 여러 해석이 가능하기 때문에 shape assembly 단계가 필요

2. Shape Assembly

- 점으로 어떤 shape를 만드는지 결정

3. Geometry Shader

- Primitive를 만드는 과정
- 수업시간에는 다루지 않음

4. Rasterization

- 렌더링하기 위해 pixelization

5. Fragment Shader

- Colorization 즉, Pixel의 색을 정함

6. Tests and Blending

- 물체가 여러 개일 경우 뒤덮을 때 어디가 가려지는지, 알파 blending 되는지 결정

Rendering Loop

```
// render loop
// -----
while (!glfwWindowShouldClose(window))
{
    // input
    // -----
    processInput(window);

    // render
    // -----
    glClearColor(0.2f, 0.3f, 0.3f, 1.0f);
    glClear(GL_COLOR_BUFFER_BIT);

    // draw our first triangle
    glUseProgram(shaderProgram);
    glBindVertexArray(VAO); // seeing as we only have a single VAO there's no need to bind it every time, but we'll do it for demonstration purposes
    glDrawArrays(GL_TRIANGLES, 0, 3);
    // glBindVertexArray(0); // no need to unbind it every time

    // glfw: swap buffers and poll IO events (keys pressed/released, mouse moved etc.)
    // -----
    glfwSwapBuffers(window);
    glfwPollEvents();
}
```

- Specify shaders to use
- Provide vertex data
- Draw

삼각형 그리는 절차

- 2개의 shader (vertex, fragment) 중 어느 shader를 사용할 것인지 결정
- Vertex data를 넘겨줌
- 그리기

Vertex/fragment shader 정의 + shader Load & compile

Character string

```
const char *vertexShaderSource = "#version 330 core\n"
"layout (location = 0) in vec3 aPos;\n"
"void main()\n"
"{\n"
"    gl_Position = vec4(aPos.x, aPos.y, aPos.z, 1.0);\n"
"}\0";
const char *fragmentShaderSource = "#version 330 core\n"
"out vec4 FragColor;\n"
"void main()\n"
"{\n"
"    FragColor = vec4(1.0f, 0.5f, 0.2f, 1.0f);\n"
"}\0";
```

개별 파일

```
#version 330 core
layout (location = 0) in vec3 aPos;
void main()
{
    gl_Position = vec4(aPos.x, aPos.y, aPos.z, 1.0);

#version 330 core
out vec4 FragColor;
void main()
{
    FragColor = vec4(1.0f, 0.5f, 0.2f, 1.0f);
```

Vertex shader

- 이 코드에선 Vertex input을 그대로 출력
- 즉, 아무 transformation이 없는 상태 (단지, homogeneous coordinate)

Fragment shader

- r = 1.0, g = 0.5, b = 0.2 색으로 통일

```
// build and compile our shader program
// -----
// vertex shader
int vertexShader = glCreateShader(GL_VERTEX_SHADER); 1. Vertex shader 생성
glShaderSource(vertexShader, 1, &vertexShaderSource, NULL); 2. vertexShaderSource 불러오기
glCompileShader(vertexShader); 3. Compile
// check for shader compile errors
int success;
char infoLog[512];
glGetShaderiv(vertexShader, GL_COMPILE_STATUS, &success);
if (!success)
{
    glGetShaderInfoLog(vertexShader, 512, NULL, infoLog);
    std::cout << "ERROR::SHADER::VERTEX::COMPILATION_FAILED\n" << infoLog << std::endl;
}
// fragment shader
int fragmentShader = glCreateShader(GL_FRAGMENT_SHADER); 1. Fragment shader 생성
glShaderSource(fragmentShader, 1, &fragmentShaderSource, NULL); 2. fragmentShaderSource 불러오기
glCompileShader(fragmentShader); 3. Compile
// check for shader compile errors
glGetShaderiv(fragmentShader, GL_COMPILE_STATUS, &success);
if (!success)
{
    glGetShaderInfoLog(fragmentShader, 512, NULL, infoLog);
    std::cout << "ERROR::SHADER::FRAGMENT::COMPILATION_FAILED\n" << infoLog << std::endl;
}
```

Shader Load & Compile

Vertex/fragment shader 정의

- 보통은 개별 파일로 저장하지만, 이 코드에서는 character string 형태로 정의
- Shader Load & Compile

- Rendering loop 들어가기 전에 shader를 load, compile 해주어야 함

Link Shader

```
int shaderProgram = glCreateProgram(); 전체 Shader 프로그램 생성
glAttachShader(shaderProgram, vertexShader);
glAttachShader(shaderProgram, fragmentShader); 위에서 정의한
glLinkProgram(shaderProgram); link vertexShader, fragmentShader 사용
// check for linking errors
glGetProgramiv(shaderProgram, GL_LINK_STATUS, &success);
if (!success) {
    glGetProgramInfoLog(shaderProgram, 512, NULL, infoLog);
    std::cout << "ERROR::SHADER::PROGRAM::LINKING_FAILED\n" << infoLog << std::endl;
}
glDeleteShader(vertexShader);
glDeleteShader(fragmentShader);
```

Link Shader

Link Shader

- 여러 shader (Vertex, Fragment shader...) 를 하나의 Shader Program 으로 결합하는 과정
- 어떤 shader 를 link 하냐에 따라 서로 다른 shader 를 사용할 수 있음

```
// draw our first triangle
glUseProgram(shaderProgram);
glBindVertexArray(VAO); VAO 쓰겠다고 선언 have a single VAO there's no need to bind it every time, but we'll
glDrawArrays(GL_TRIANGLES, 0, 3); Rendering
// glBindVertexArray(0); // no need to unbind it every time
```

- Specify shaders to use
- Provide vertex data
- Draw

Main rendering loop 내부

- 위에서 정의한 `ShaderProgram` 을 것을 선언
(Rendering 할 준비 완료)
- 2. Provide vertex data 가 필요
(다음 페이지에서 구체적 설명)
- VAO 쓰겠다고 선언
- Rendering
 - `GL_TRIANGLES`: 삼각형을 그릴 것을 shape assembly

지금까지, shader 만들고 → load & Compile → `ShaderProgram` 에 link → 사용
이제, 삼각형 데이터를 전달해야 함.

Prepare Vertex Data

```
// set up vertex data (and buffers!) and configure vertex attributes
float vertices[] = {
    -0.5f, -0.5f, 0.0f, // left
    0.5f, -0.5f, 0.0f, // right 삼각형 꼭짓점 위치 좌표
    0.0f, 0.5f, 0.0f // top
};

unsigned int VBO, VAO;
 glGenVertexArrays(1, &VAO);
 glGenBuffers(1, &VBO);

// bind the Vertex Array Object first, then bind and set vertex buffer(s), and then configure vertex attribute(s).
 glBindVertexArray(VAO); 1. Bind VAO

glBindBuffer(GL_ARRAY_BUFFER, VBO); 2. Bind VBO, Vertices 데이터를 VBO에 복사
glBufferData(GL_ARRAY_BUFFER, sizeof(vertices), vertices, GL_STATIC_DRAW);
3. VAO 가 VBO 의 데이터 형태를 선언 (Vertex attribute pointers)
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 3 * sizeof(float), (void*)0); → 0 번째 속성으로 설정,
Component 3 개(x, y, z), 데이터 타입,
Normalize 여부 Stride, Offset
// note that this is allowed, the call to glVertexAttribPointer registered VBO as the vertex attribute's bound vertex
glBindBuffer(GL_ARRAY_BUFFER, 0);

// You can unbind the VAO afterwards so other VAO calls won't accidentally modify this VAO, but this rarely happens.
// VAOs requires a call to glBindVertexArray anyways so we generally don't unbind VAOs (nor VBOs) when it's not direc
glBindVertexArray(0);
```



Prepare Vertex Data

Prepare Vertex Data

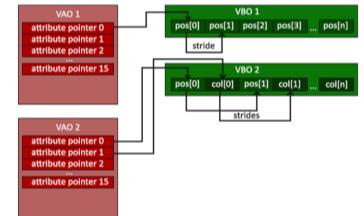
- Vertex data 전달 해주어야 함
- VBO, VAO 사용
- 실제로 **Rendering** 할 때는 VAO 사용

Vertex Buffer Object (VBO)

- (실제 데이터 저장) Vertex data 저장

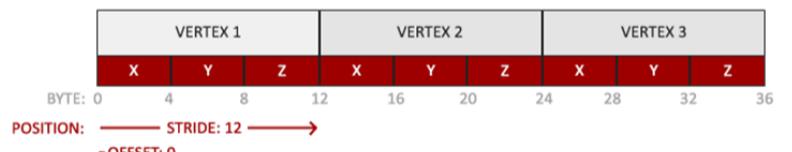
Vertex Array Object (VAO)

- (데이터를 관리) data 의 의미, 형태를 저장



Sequence

- Bind VAO
- Bind VBO and copy vertices
- Configure vertex attributes in VBO using VAO
- Drawing using VAO



Vertex 는 위와 같은 형태로 VBO에 들어감

- Stride:** 첫 번째 vertex 의 시작으로부터, 두 번째 vertex 시작까지 건너 뛰어야 하는 양
- Offset:** 첫 번째 vertex 가 시작하는 위치

Attribute 가 여러 개인 경우

- Position, Color, TexCoords

```
GLfloat data[] = {
    // Position // Color    Z // TexCoords Y   Z   X   Y
    1.0f, 0.0f, 0.5f, 0.5f, 0.5f, 0.0f, 0.5f, 0.0f, 1.0f, 0.2f, 0.8f, 0.0f, 0.0f, 1.0f
};

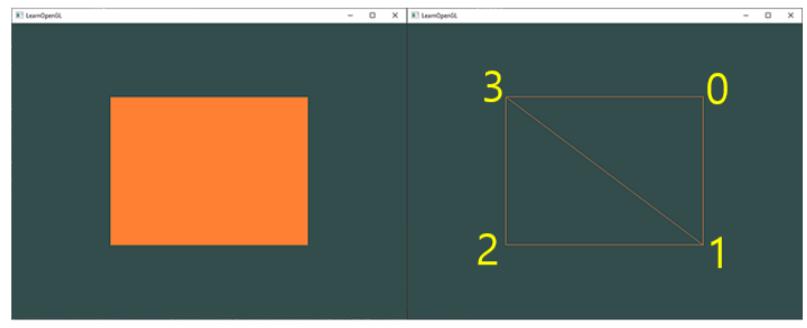
glEnableVertexAttribArray(0);
glVertexAttribPointer(0, 2, GL_FLOAT, GL_FALSE,
    7 * sizeof(GLfloat), (GLvoid*)0);
glEnableVertexAttribArray(1);
glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE,
    7 * sizeof(GLfloat), (GLvoid*)(2 * sizeof(GLfloat)));
glEnableVertexAttribArray(2);
glVertexAttribPointer(2, 2, GL_FLOAT, GL_FALSE,
    7 * sizeof(GLfloat), (GLvoid*)(5 * sizeof(GLfloat)));
```

Indexed Face-set Rendering using Element Buffer Object (EBO)

사각형을 그려보자

- 렌더링 할 때, direct vertex position 으로 렌더링 하지 않고 index 를 사용

```
float vertices[] = {
    0.5f, 0.5f, 0.0f, // top right
    0.5f, -0.5f, 0.0f, // bottom right
    -0.5f, -0.5f, 0.0f, // bottom left
    -0.5f, 0.5f, 0.0f // top left
};
unsigned int indices[] = { // note that we start from 0!
    0, 1, 3, // first triangle
    1, 2, 3 // second triangle
};
```

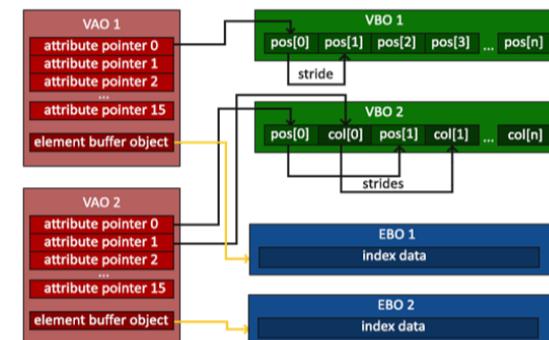


```
// ...: Initialization code :: ..
// 1. bind Vertex Array Object
glBindVertexArray(VAO);
// 2. copy our vertices array in a vertex buffer for OpenGL to use           Index copy
glBindBuffer(GL_ARRAY_BUFFER, VBO);
glBufferData(GL_ARRAY_BUFFER, sizeof(vertices), vertices, GL_STATIC_DRAW);
// 3. copy our index array in a element buffer for OpenGL to use
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, EBO);
glBufferData(GL_ELEMENT_ARRAY_BUFFER, sizeof(indices), indices, GL_STATIC_DRAW);
// 4. then set the vertex attributes pointers
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 3 * sizeof(float), (void*)0);
 glEnableVertexAttribArray(0);

[...]

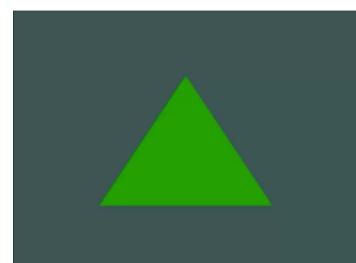
// ...: Drawing code (in render loop) :: ..
glUseProgram(shaderProgram);
glBindVertexArray(VAO);
glDrawElements(GL_TRIANGLES, 6, GL_UNSIGNED_INT, 0);
glBindVertexArray(0);
```

- 사각형은 삼각형 2개로 나눠서 그림
- 이 경우 중복되는 꽈짓점이 생기고, 이 vertex 들에 대해 여러 번 데이터를 보내야 하는 비효율적인 상황 발생
⇒ **index** 를 사용
- 앞의 코드에서는 VBO 에 vertex position 을 저장했고, 이 코드에서는 추가로 EBO 에 index 를 저장함
- VAO 가 VBO, EBO 모두를 point 함



OpenGL Shading Language (GLSL) Basics

- Vertex shader, fragment shader: 실질적으로 프로그래밍을 많이 할 곳
- 시간에 따라 계속 색이 변하는 삼각형을 그려보자



Typical structure

```
#version version_number
in type in_variable_name;
in type in_variable_name;

out type out_variable_name;

uniform type uniform_name;

void main()
{
    // process input(s) and do some weird graphics stuff
    ...
    // output processed stuff to output variable
    out_variable_name = weird_stuff_we_processed;
}
```

1. 버전 선언

2. 변수 선언

- **in**: shader 의 input
- **out**: shader 의 output
- **uniform**: shader 뿐만 아니라 **OpenGL Host** 도 접근 가능한 글로벌 변수

in, out 은 vertex/fragment 마다 다 다른 값
uniform 은 전부 같은 값

3. 메인

- Shader function

Vector Type

```
vec2 someVec;
vec4 differentVec = someVec.xyxx; //swizzling
vec3 anotherVec = differentVec.zyw;
vec4 otherVec = someVec.xxxx + anotherVec.yxzy;
```

```
vec2 vect = vec2(0.5, 0.7);
vec4 result = vec4(vect, 0.0, 0.0);
vec4 otherResult = vec4(result.xyz, 1.0);
```

- **Swizzling** operation: dimension 쉽게 늘리고 줄일 수 있게 해주는 연산

- **vec_n**: the default vector of _n floats
 - _n에 따라 component가 몇 개인지 결정
 - component 1-_x, 2-_y, 3-_z, 4-_w
- **bvec_n**: a vector of _n booleans
- **ivec_n**: a vector of _n integers
- **uvec_n**: a vector of _n unsigned integers
- **dvec_n**: a vector of _n double precisions

Matrix Types

```
#version 330 core
layout (location = 0) in vec3 aPos;
layout (location = 1) in vec2 aTexCoord;

out vec2 TexCoord;

uniform mat4 model;
uniform mat4 view;
uniform mat4 projection;

void main()
{
    gl_Position = projection * view * model * vec4(aPos, 1.0f);
    TexCoord = vec2(aTexCoord.x, aTexCoord.y);
}
```

- **mat_n**: $n \times n$ matrix of floats
- **dmat_n**: $n \times n$ matrix of doubles

In and Out Variables

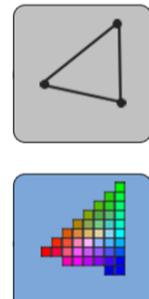
실제 코딩할 때는 shader 파일 (Vertex shader → .vs, Fragment shader → .fs) 파일을 따로 만듭니다

Vertex shader

```
#version 330 core
layout (location = 0) in vec3 aPos; // the position variable has attribute position 0
out vec4 vertexColor; // specify a color output to the fragment shader
void main()
{
    gl_Position = vec4(aPos, 1.0); // see how we directly give a vec3 to vec4's constructor
    vertexColor = vec4(0.5, 0.0, 0.0, 1.0); // set the output variable to a dark-red color
}
```

Fragment shader

```
#version 330 core
out vec4 FragColor;
in vec4 vertexColor; // the input variable from the vertex shader (same name and same type)
void main()
{
    FragColor = vertexColor;
}
```



Vertex shader

- **location**
 - 몇 번째 데이터를 사용할 것인가
 - 예시
 - location = 0, 첫 번째 데이터 사용 (position x, y, z)
 - location = 1, 두 번째 데이터 사용 (color r, g, b)

location을 사용한 연결:

- VAO 설정 중에서 `glEnableVertexAttribArray(0);`을 통해 0번 속성 (location 0)을 활성화하면, OpenGL은 VAO에 설정된 구조에 따라 해당 데이터를 Shader의 0번 슬롯으로 전달하게 됨
- Shader 파일에서 `layout(location = 0)`로 0번 슬롯에 있는 데이터를 참조하겠다고 선언

Fragment shader

- **vertexColor**
 - Vertex shader에서 무조건 출력해야 하는 output
 - modeling, viewing, projection 한 결과

Fragment shader

- **vertexColor**
 - Vertex shader의 output인 `vertexColor`을 input으로 받음

Uniform Variable

이제, 시간에 따라 계속 색이 바뀌도록 Fragment Shader를 설정합니다

Fragment shader 코드

```
#version 330 core
out vec4 FragColor;
uniform vec4 ourColor; // we set this variable in the OpenGL code.
void main()
{
    FragColor = ourColor;
}
```

OpenGL Host에서 접근 가능하도록, Uniform 변수로 수정합니다

OpenGL 코드

```
float timeValue = glfwGetTime();
float greenValue = (sin(timeValue) / 2.0f) + 0.5f;           ourColor 에 접근
int vertexColorLocation = glGetUniformLocation(shaderProgram, "ourColor");
glUseProgram(shaderProgram);
glUniform4f(vertexColorLocation, 0.0f, greenValue, 0.0f, 1.0f); ourColor 에 Setting
```

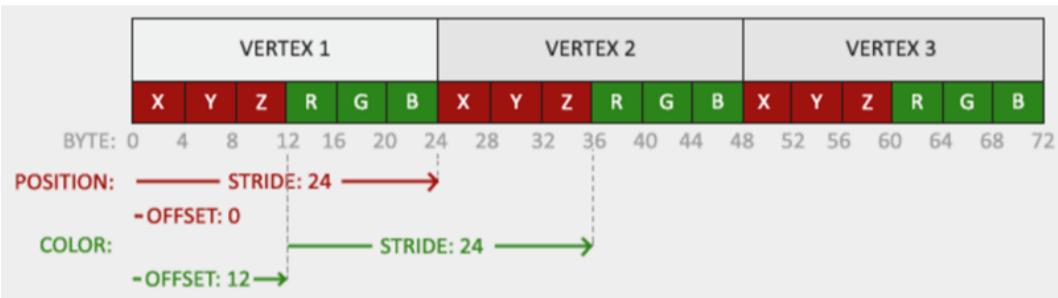
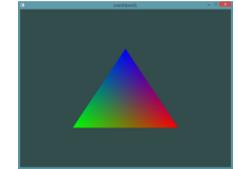
r = 0.0, g = `greenValue`로 계속 바꿔줌, b = 0.0, a = 1.0
→ 현재 시간에 따라 색이 바뀌는 삼각형 완성

More

💡 삼각형을 무지개 색으로 채워주자

```
float vertices[] = {    각 꼭짓점의 좌표 및 Color
    // positions          // colors
    0.5f, -0.5f, 0.0f,  1.0f, 0.0f, 0.0f,      // bottom right
    -0.5f, -0.5f, 0.0f,  0.0f, 1.0f, 0.0f,      // bottom left
    0.0f,  0.5f, 0.0f,  0.0f, 0.0f, 1.0f        // top
};
```

Vertex data in host with positions/colors



VBO memory layout

```
// position attribute
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 6 * sizeof(float), (void*)0);
glEnableVertexAttribArray(0);
// color attribute
glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, 6 * sizeof(float), (void*)(3 * sizeof(float)))
glEnableVertexAttribArray(1);
```

Configure VAO

```
#version 330 core
layout (location = 0) in vec3 aPos; // the position variable has attribute position 0
layout (location = 1) in vec3 aColor; // the color variable has attribute position 1

out vec3 ourColor; // output a color to the fragment shader

void main()
{
    gl_Position = vec4(aPos, 1.0);
    ourColor = aColor; // set ourColor to the input color we got from the vertex data
}
```

Vertex Shader

```
#version 330 core
out vec4 FragColor;
in vec3 ourColor;

void main()
{
    FragColor = vec4(ourColor, 1.0);
}
```

Fragment Shader

shader.h

💡 shader 파일 (.vs, .fs)

File open/load 과정이 필요한데, shader.h 헤더는 이 과정을 도와줌

`#include <learnopengl/shader_s.h>`

Include

`// build and compile our shader program
Shader ourShader("3_3.shader.vs", "3_3.shader.fs"); // you can name your shader files however you like`

Load and compile

`// render the triangle
ourShader.use();
glBindVertexArray(VAO);`

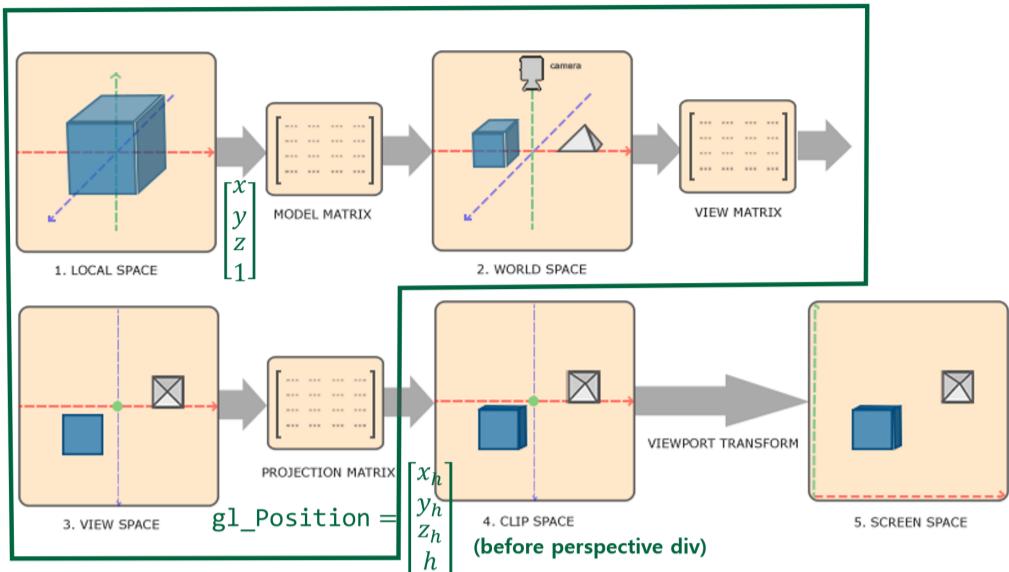
Use

Viewing Pipeline



- Basic concept에서 본 pipeline 과 동일
- Projection 까지가 vertex shader에서 하는 일
- input, output은 둘 다 homogeneous coordinate
- 각 matrix들을 곱해가면서 계산

Vertex Shader



회전하는 3D 육면체를 만들어보자

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

```
//  
float vertices[] = {  
    -0.5f, -0.5f, -0.5f,  0.0f,  0.0f,  
    0.5f, -0.5f, -0.5f,  1.0f,  0.0f,  
    0.5f,  0.5f, -0.5f,  1.0f,  1.0f,  
    0.5f,  0.5f, -0.5f,  1.0f,  1.0f,  
    -0.5f,  0.5f, -0.5f,  0.0f,  1.0f,  
    -0.5f, -0.5f, -0.5f,  0.0f,  0.0f,  
  
    -0.5f, -0.5f,  0.5f,  0.0f,  0.0f,  
    0.5f, -0.5f,  0.5f,  1.0f,  0.0f,  
    0.5f,  0.5f,  0.5f,  1.0f,  1.0f,  
    0.5f,  0.5f,  0.5f,  1.0f,  1.0f,  
    -0.5f,  0.5f,  0.5f,  0.0f,  1.0f,  
    -0.5f, -0.5f,  0.5f,  0.0f,  0.0f,  
  
    -0.5f,  0.5f,  0.5f,  1.0f,  0.0f,  
    -0.5f,  0.5f, -0.5f,  1.0f,  1.0f,  
    -0.5f, -0.5f, -0.5f,  0.0f,  1.0f,  
    -0.5f, -0.5f,  0.5f,  0.0f,  0.0f,  
    -0.5f,  0.5f,  0.5f,  1.0f,  0.0f,  
  
    0.5f,  0.5f,  0.5f,  1.0f,  0.0f,
```



GLM include

- GLM 라이브러리 (OpenGL Mathematics library) 사용
 - Linear algebra 등 수학 포함

Data

- **Modeling coordinate 기준 좌표**
 - 육면체의 꼭짓점 데이터를 input 으로 받아야 함
 - 해당 예제는 Index face-set (EBO) 대신, vertex position 을 넣어주는 방식 사용
 - 6개의 면 → 하나의 면은 2개의 삼각형 → 삼각형은 3개의 Vertex
⇒ $6 \times 2 \times 3 = 36$ 개의 좌표 필요
 - 이 좌표들을 Vertex shader 에 넘겨줌

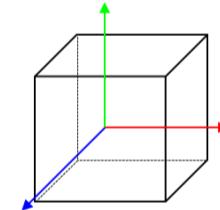
```

    0.5f,  0.5f, -0.5f,  1.0f,  1.0f,
    0.5f, -0.5f, -0.5f,  0.0f,  1.0f,
    0.5f, -0.5f, -0.5f,  0.0f,  1.0f,
    0.5f, -0.5f,  0.5f,  0.0f,  0.0f,
    0.5f,  0.5f,  0.5f,  1.0f,  0.0f,
-0.5f, -0.5f, -0.5f,  0.0f,  1.0f,
    0.5f, -0.5f, -0.5f,  1.0f,  1.0f,
    0.5f, -0.5f,  0.5f,  1.0f,  0.0f,
    0.5f, -0.5f,  0.5f,  1.0f,  0.0f,
-0.5f, -0.5f,  0.5f,  0.0f,  0.0f,
-0.5f, -0.5f, -0.5f,  0.0f,  1.0f,
-0.5f,  0.5f, -0.5f,  0.0f,  1.0f,
    0.5f,  0.5f, -0.5f,  1.0f,  1.0f,
    0.5f,  0.5f,  0.5f,  1.0f,  0.0f,
    0.5f,  0.5f,  0.5f,  1.0f,  0.0f,
-0.5f,  0.5f,  0.5f,  0.0f,  0.0f,
-0.5f,  0.5f, -0.5f,  0.0f,  1.0f
};

unsigned int VBO, VAO;

```

36 vertices (6F*2Tri*3Verts) defined
in modeling coordinate



```

// render loop
while (!glfwWindowShouldClose(window))
{
    // input
    // -----
    processInput(window);

    // render
    // -----
    glClearColor(0.2f, 0.3f, 0.3f, 1.0f);
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT); // also clear the depth buffer now!

    // bind textures on corresponding texture units
    glBindTexture(GL_TEXTURE0);
    glBindTexture(GL_TEXTURE_2D, texture1);
    glBindTexture(GL_TEXTURE1);
    glBindTexture(GL_TEXTURE_2D, texture2);

    // activate shader
    ourShader.use();

    // create transformations 단위 행렬로 Initialize
    glm::mat4 model = glm::mat4(1.0f); // make sure to initialize matrix to identity matrix first
    glm::mat4 view = glm::mat4(1.0f); // 회전 각도 회전축
    glm::mat4 projection = glm::mat4(1.0f);
    model = glm::rotate(model, (float)glfwGetTime(), glm::vec3(0.5f, 1.0f, 0.0f));
    view = glm::translate(view, glm::vec3(0.0f, 0.0f, -3.0f));
    projection = glm::perspective(glm::radians(45.0f), (float)SCR_WIDTH / (float)SCR_HEIGHT, 0.1f, 100.0f);
    // retrieve the matrix uniform locations
    unsigned int modelLoc = glGetUniformLocation(ourShader.ID, "model");
    unsigned int viewLoc = glGetUniformLocation(ourShader.ID, "view");
    // pass them to the shaders (3 different ways)
    glUniformMatrix4fv(modelLoc, 1, GL_FALSE, glm::value_ptr(model));
    glUniformMatrix4fv(viewLoc, 1, GL_FALSE, &view[0][0]);
    // note: currently we set the projection matrix each frame, but since the projection matrix rarely changes it's of
    ourShader.setMat4("projection", projection);

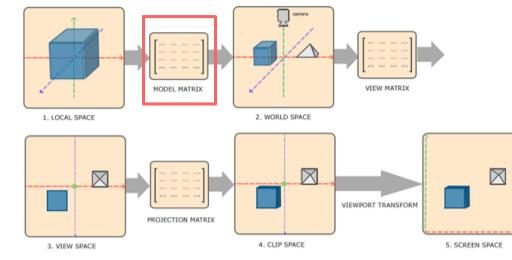
    // render box
    glBindVertexArray(VAO);
    glDrawArrays(GL_TRIANGLES, 0, 36);

    // glfw: swap buffers and poll IO events (keys pressed/released, mouse moved etc.)
    // -----
    glfwSwapBuffers(window);
    glfwPollEvents();
}

// optional: de-allocate all resources once they've outlived their purpose:
// -----
glDeleteVertexArrays(1, &VAO);
glDeleteBuffers(1, &VBO);

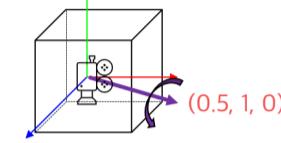
// glfw: terminate, clearing all previously allocated GLFW resources.
// -----
glfwTerminate();
return 0;
}

```



model

- `glm::rotate` 함수 사용
- 회전 각도, 회전축 정보를 담고 있음
- `model = model * R`



Vertex shader

```

#version 330 core
layout (location = 0) in vec3 aPos;
layout (location = 1) in vec2 aTexCoord;

out vec2 TexCoord;

uniform mat4 model;
uniform mat4 view;
uniform mat4 projection;

void main()
{
    gl_Position = projection * view * model * vec4(aPos, 1.0f);
    TexCoord = vec2(aTexCoord.x, aTexCoord.y);
}

```

💡 여기까지 rotation 적용됨. 육면체는 계속 돌게 될 거임.
근데, 카메라 위치를 옮겨야 돌아가는 걸 볼 수 있게 되는 거임.
즉, 물체로부터 조금 뒤로 이동해야 함

⇒ viewing transformation



```

// render loop
while (!glfwWindowShouldClose(window))
{
    // input
    // -----
    processInput(window);

    // render
    // -----
    glClearColor(0.2f, 0.3f, 0.3f, 1.0f);
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT); // also clear the depth buffer now!

    // bind textures on corresponding texture units
    glBindTexture(GL_TEXTURE0);
    glBindTexture(GL_TEXTURE_2D, texture1);
    glBindTexture(GL_TEXTURE1);
    glBindTexture(GL_TEXTURE_2D, texture2);

    // activate shader
    ourShader.use();

    // create transformations
    glm::mat4 model = glm::mat4(1.0f); // make sure to initialize matrix to identity matrix first
    glm::mat4 view = glm::mat4(1.0f); // 카메라의 위치 결정
    glm::mat4 projection = glm::mat4(1.0f);
    model = glm::rotate(model, (float)glfwGetTime(), glm::vec3(0.5f, 1.0f, 0.0f));
    view = glm::translate(view, glm::vec3(0.0f, 0.0f, -3.0f));
    projection = glm::perspective(glm::radians(45.0f), (float)SCR_WIDTH / (float)SCR_HEIGHT, 0.1f, 100.0f);
    // retrieve the matrix uniform locations
    unsigned int modelLoc = glGetUniformLocation(ourShader.ID, "model");
    unsigned int viewLoc = glGetUniformLocation(ourShader.ID, "view");
    // pass them to the shaders (3 different ways)
    glUniformMatrix4fv(modelLoc, 1, GL_FALSE, glm::value_ptr(model));
    glUniformMatrix4fv(viewLoc, 1, GL_FALSE, &view[0][0]);
    // note: currently we set the projection matrix each frame, but since the projection matrix rarely changes it's of
    ourShader.setMat4("projection", projection);

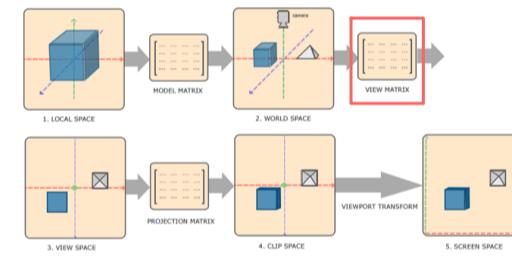
    // render box
    glBindVertexArray(VAO);
    glDrawArrays(GL_TRIANGLES, 0, 36);

    // glfw: swap buffers and poll IO events (keys pressed/released, mouse moved etc.)
    // -----
    glfwSwapBuffers(window);
    glfwPollEvents();
}

// optional: de-allocate all resources once they've outlived their purpose:
// -----
glDeleteVertexArrays(1, &VAO);
glDeleteBuffers(1, &VBO);

// glfw: terminate, clearing all previously allocated GLFW resources.
// -----
glfwTerminate();
return 0;
}

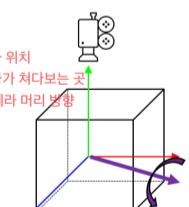
```



view

- 카메라의 위치 결정
- `glm::translate` 함수 사용
= `glm::lookAt` 함수와 동일
- 카메라의 default position = 원점
→ 카메라의 위치 z 방향으로 3 이동

view = `glm::lookAt(glm::vec3(0.0f, 0.0f, 3.0f), glm::vec3(0.0f, 0.0f, 0.0f), glm::vec3(0.0f, 1.0f, 0.0f));`
카메라 위치
카메라가 쳐다보는 곳
카메라 머리 방향



💡 카메라를 움직이는 viewing 과 modeling 으로 오브젝트를 움직이는 것의 결과가 같아서 OpenGL에서 view 안 하고 그냥 model에서 다 처리하는 경우도 있음 (해당 코드가 그런 경우임)

Vertex shader

```

#version 330 core
layout (location = 0) in vec3 aPos;
layout (location = 1) in vec2 aTexCoord;

out vec2 TexCoord;

uniform mat4 model;
uniform mat4 view;
uniform mat4 projection;

void main()
{
    gl_Position = projection * view * model * vec4(aPos, 1.0f);
    TexCoord = vec2(aTexCoord.x, aTexCoord.y);
}

```

💡 Projection

3차원 상에서 렌더링된 것을 어떻게 2차원으로 투사할 것인가?

```
// render loop
while (!glfwWindowShouldClose(window))
{
    // input
    // -----
    processInput(window);

    // render
    // -----
    glClearColor(0.2f, 0.3f, 0.3f, 1.0f);
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT); // also clear the depth buffer now!

    // bind textures on corresponding texture units
    glBindTexture(GL_TEXTURE0, texture1);
    glBindTexture(GL_TEXTURE1, texture2);
    glBindTexture(GL_TEXTURE_2D, texture2);

    // activate shader
    ourShader.use();

    // create transformations
    glm::mat4 model      = glm::mat4(1.0f); // make sure to initialize matrix to identity matrix first
    glm::mat4 view       = glm::mat4(1.0f);
    glm::mat4 projection = glm::mat4(1.0f);
    model = glm::rotate(model, (float)glfwGetTime(), glm::vec3(0.5f, 1.0f, 0.0f));
    view = glm::translate(view, glm::vec3(0.0f, 0.0f, -3.0f));
    projection = glm::perspective(glm::radians(45.0f), (float)SCR_WIDTH / (float)SCR_HEIGHT, 0.1f, 100.0f);
    // retrieve the matrix uniform locations
    unsigned int modelLoc = glGetUniformLocation(ourShader.ID, "model");
    unsigned int viewLoc = glGetUniformLocation(ourShader.ID, "view");
    // pass them to the shaders (3 different ways)
    glUniformMatrix4fv(modelLoc, 1, GL_FALSE, glm::value_ptr(model));
    glUniformMatrix4fv(viewLoc, 1, GL_FALSE, &view[0][0]);
    // note: currently we set the projection matrix each frame, but since the projection matrix rarely changes it's often better to just set it once and leave it there
    ourShader.setMat4("projection", projection); // model, view 와 다르게
    // render box
    glBindVertexArray(VAO);
    glDrawArrays(GL_TRIANGLES, 0, 36);

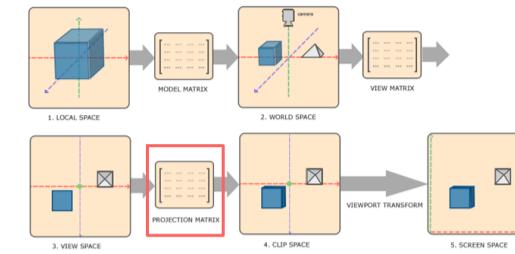
    // glfw: swap buffers and poll IO events (keys pressed/released, mouse moved etc.)
    // -----
    glfwSwapBuffers(window);
    glfwPollEvents();
}

// optional: de-allocate all resources once they've outlived their purpose:
// -----
glDeleteVertexArrays(1, &VAO);
glDeleteBuffers(1, &VBO);

// glfw: terminate, clearing all previously allocated GLFW resources.
// -----
glfwTerminate();
return 0;
}
```

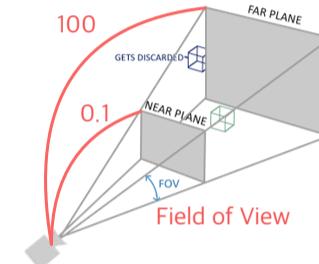
Near plane 거리, Far plane 거리

OpenGL 코드에서 uniform 변수 model, view, projection 넘겨줌



projection

- Perspective, orthographic 2가지가 있으나 이 코드에선 `glm::perspective` 사용
- FOV, aspect ratio, near plane/far plane 까지 거리 정보를 담고 있음
- view volume 밖은 clipping



Vertex shader

```
#version 330 core
layout (location = 0) in vec3 aPos;
layout (location = 1) in vec2 aTexCoord;

out vec2 TexCoord;

uniform mat4 model;
uniform mat4 view;
uniform mat4 projection;

void main()
{
    gl_Position = projection * view * model * vec4(aPos, 1.0f);
    TexCoord = vec2(aTexCoord.x, aTexCoord.y);
}
```

💡 (변형) 코드에서 model transform 2번 이상일 때

- 모델에 추가적으로 T 와 R 을 곱해보자

```
model = glm::translate(model, glm::vec3(1.0f, 0.0f, 0.0f));
model = glm::rotate(model, (float)glfwGetTime(), glm::vec3(0.5f, 1.0f, 0.0f));
```

```
model = glm::rotate(model, (float)glfwGetTime(), glm::vec3(0.5f, 1.0f, 0.0f));
model = glm::translate(model, glm::vec3(1.0f, 0.0f, 0.0f));
```

- 코드 순서가 translate, rotate면 → 실제 순서는 rotate, translate ⇒ 자전 효과
 - model x T x R ⇒ Modeling coordinate** 를 기준으로 Rotate 후에 Translate 하니까, 자전 효과
 - R 은 곧 Post-multiplication, **Modeling coordinate** 를 기준으로 회전

- 코드 순서가 rotate, translate면 → 실제 순서는 translate, rotate ⇒ 공전 효과
 - model x R x T ⇒ World coordinate** 를 기준으로 Translate 한 후에 Rotate 하니까, 공전 효과
 - R 은 곧 pre-multiplication, **World coordinate** 를 기준으로 회전