Getting Started (2)

GPU Programming

2022학년도
2학기

ⓒ nvidia

Hello Triangle

Graphics Pipeline

- 그래픽스 파이프라인
  - OpenGL 내에서 다루는 많은 데이터는 3D 공간상에 존재
  - 최종 출력이 될 화면과 윈도우 창은 2D 픽셀
  - 이러한 3D 좌표를 2D 좌표로,
    이를 다시 2D 픽셀로 변환하는 일련의 작업이 그래픽스 파이프라인

- 셰이더 (shader)
  - 그래픽스 파이프라인은 병렬화가 쉬움
  - 파이프라인의 각 단계별로
    GPU 내 수백~수천 개의 프로세싱 코어에
    셰이더라는 작은 프로그램을 올림
  - OpenGL에서는 셰이더 프로그래밍을 위해
    OpenGL Shading Language (GLSL) 사용

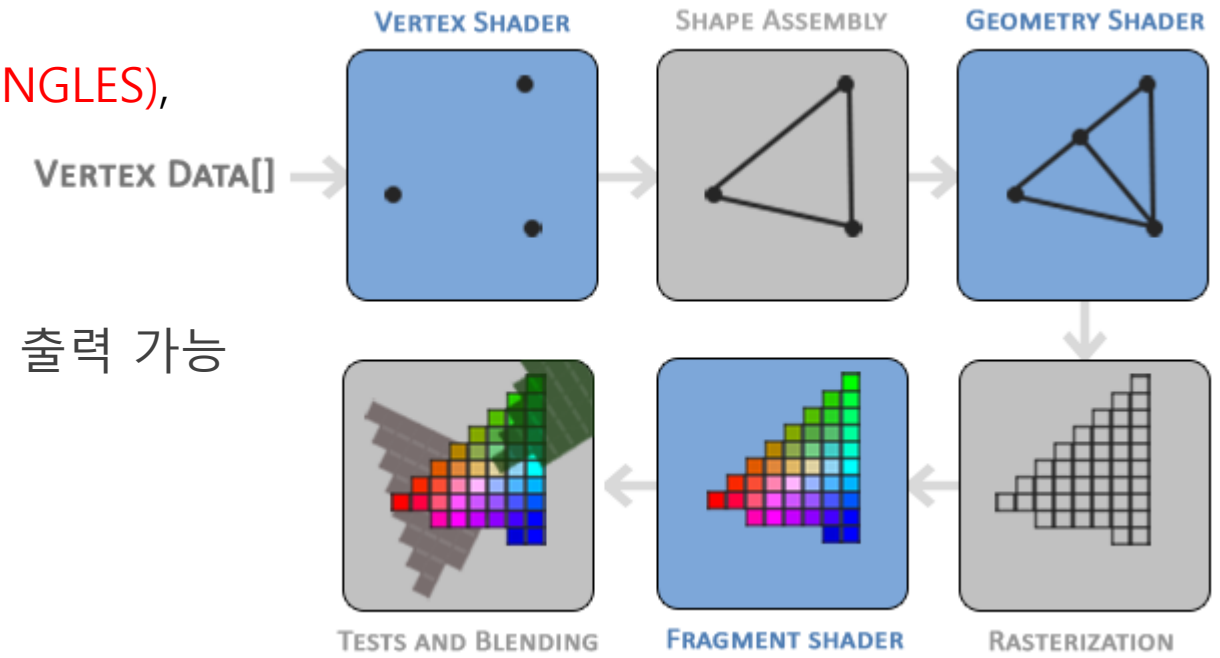
nVIDIA RTX3080Ti의 구조

Graphics Pipeline

- Vertex shader
  - 3D 좌표값(위치, 컬러 등)으로 표현되는 프리미티브(primitive)의 정점(vertex)을 조작
  - Transformation(변환)을 통해 물체의 이동이나 좌표계의 변환 가능

- Primitive assembly
  - Vertex로부터 primitive(도형)를 조립
  - Primitive type에는 점(GL_POINTS), 삼각형(GL_TRIANGLES), 선(GL_LINE_STRIP) 등이 있음

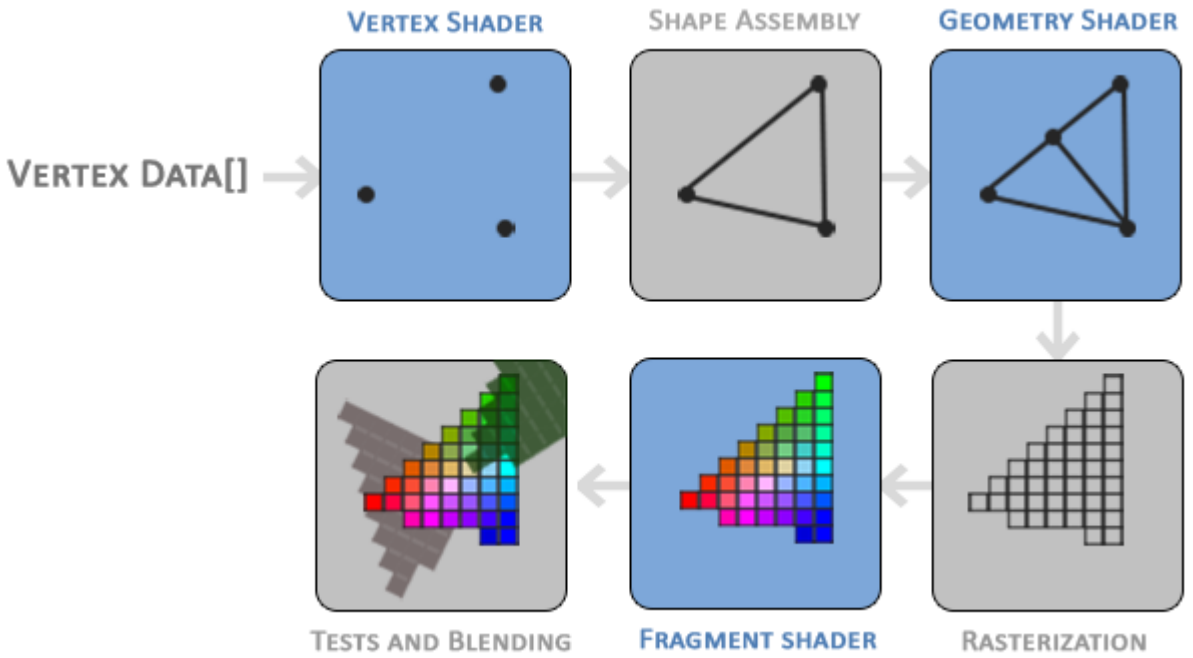
- Geometry shader (optional)
  - 입력 primitive를 조작하여 새로운 primitive를 생성, 출력 가능
  - Cube mapping, shadow volume 등에 사용 가능



Graphics Pipeline

- Rasterization
  - 프리미티브를 화면의 픽셀과 매핑해주는 작업.
    프리미티브가 삼각형이라면, 삼각형 내부를 픽셀 단위의 프래그먼트(fragment)로 채우는 작업이 됨
  - 참고로, 최종적으로 프레임버퍼에 쓰여진 데이터를 픽셀(pixel)
    그 픽셀을 렌더링하기 전에 필요한 모든 데이터를 프래그먼트(fragment)라 함
  - Fragment shader로 fragment를 보내기 전,
    clipping이 수행되어 view 밖의 fragment를 제거

- Fragment shader (또는 pixel shader)
  - 픽셀 단위의 프래그먼트(fragment)를 조작
  - 픽셀의 최종 컬러를 계산 (광원, 그림자, 빛의 색 등)
  - 대부분의 경우 이 단계에서 많은 연산 시간을 소요
    (높아지는 해상도, 복잡해지는 광원 효과 등)

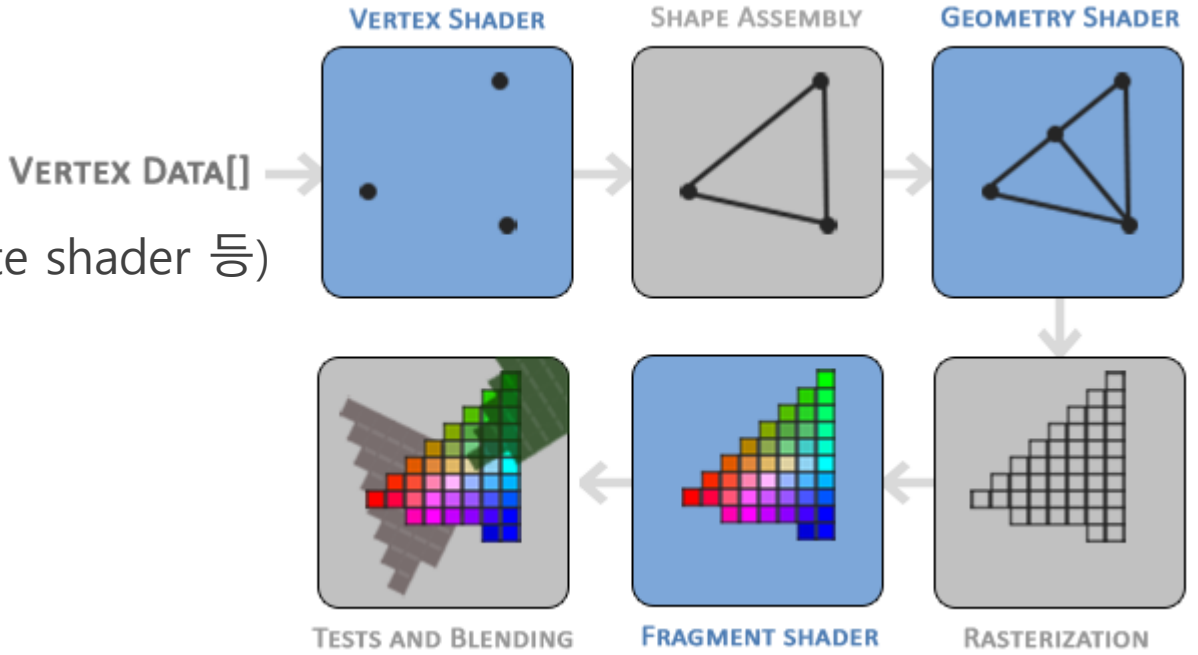


Graphics Pipeline

- Tests and blending
  - 깊이(depth) 값과 스텐실(stencil) 값을 토대로 해당 픽셀이 최종적으로 화면에 그려질지 말지 결정
  - Alpha(투명도) 값이 존재할 경우, 반투명 효과를 위해 다른 픽셀과 색을 섞을 수 있음 (blending)

- 하늘색 shader는 programmable한 부분으로, 개발자가 직접 shader를 생성해야 함
  - Vertex 및 fragment shader는 필수
  - 다른 shader는 선택 (geometry, tessellation, compute shader 등)

- 회색 부분은 fixed pipeline으로, OpenGL의 함수 호출을 통해 설정 가능



Vertex Input

- NDC (normalized device coordinates)
  - [-1.0, 1.0] 범위를 가지는 좌표계
  - 화면 좌표계의 2D 픽셀로 변환되기 전의 모든 좌표들은 NDC 상에 존재해야 나중에 화면 상에 보여지게 됨

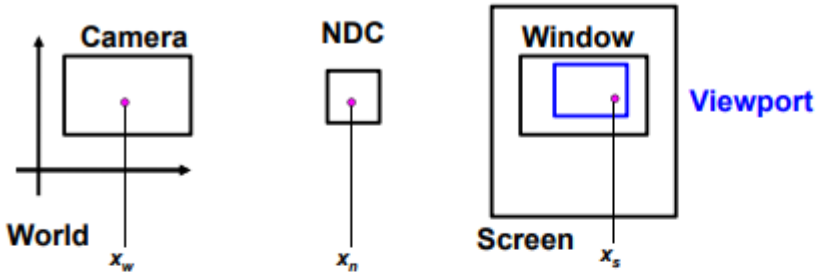
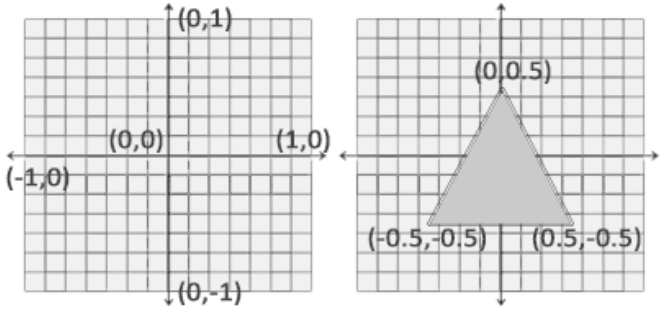
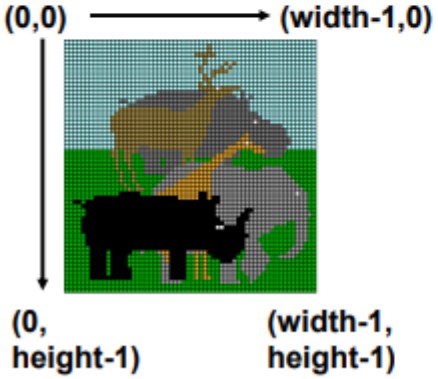


Figure 3.1: This shows a mapping from a viewable region in the world through a camera to the viewport in our screen space pass through the intermediate space, normalized device coordinate (NDC).

- 3개의 vertex 선언 예시

```
float vertices[] = {
    -0.5f, -0.5f, 0.0f,
     0.5f, -0.5f, 0.0f,
     0.0f,  0.5f, 0.0f
};
```





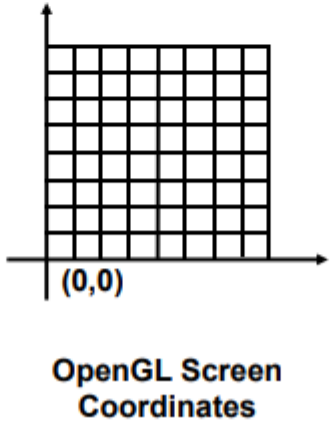


Figure 3.2: This shows two different conventions of screen coordinate spaces.

Vertex Input

- Vertex buffer objects (VBOs)
  - GPU 상에 vertex data를 위한 메모리 공간을 할당
  - 이후 대량의 vertex data를 한꺼번에 효과적으로 GPU로 전달 가능

- VBO 사용법
  - 버퍼를 생성하고 ID를 VBO 할당

```
unsigned int VBO;
glGenBuffers(1, &VBO);
```

```
glBindBuffer(GL_ARRAY_BUFFER, VBO);
```

  - VBO의 버퍼 유형인 GL_ARRAY_BUFFER로 바인딩하면,
    이후 GL_ARRAY_BUFFER형태의 모든 버퍼는 현재 바인딩된 버퍼(VBO)를 사용하게 됨
  - 미리 정의된 vertex data를 버퍼의 메모리에 복사.
    마지막 파라미터는 드라이버에 알려주는 성능 힌트임 (데이터가 한번 보내면 거의 바뀌지 않음).

```
glBufferData(GL_ARRAY_BUFFER, sizeof(vertices), vertices, GL_STATIC_DRAW);
```

Vertex Shader

- Shader는 보통 별도의 .glsl 파일에 기술하나, C소스 코드 내에 문자열로 저장할 수도 있음
- C와 비슷해 보이나, C 문법이 아닌 GLSL 문법을 따름

- 가장 간단한 vertex shader 예제

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

void main()
{
    gl_Position = vec4(aPos.x, aPos.y, aPos.z, 1.0);
}
```

```
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";
```

  - OpenGL 3.3 Core-profile을 사용
  - 첫번째로(location =0)로 입력(in) 받는 데이터의 타입이 vec3 (3차원 벡터)이고, 이를 aPos라는 변수로 접근
  - 최종 출력 위치 gl_position에, 입력받은 3차원 벡터와 1.0을 합한 vec4(4차원 벡터) 값을 저장

Compiling a shader

- Vertex shader 객체 생성

```
unsigned int vertexShader;
vertexShader = glCreateShader(GL_VERTEX_SHADER);
```

- 앞의 소스코드가 포함된 문자열을 vertex shader 객체의 소스로 지정 후 컴파일

```
glShaderSource(vertexShader, 1, &vertexShaderSource, NULL);
glCompileShader(vertexShader);
```

- 컴파일이 제대로 되었는지 확인
  - 안 되었으면 왜 안되었는지 로그 출력

```
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

- 가장 간단한 Fragment shader 예제
  - vec4 형태의 FragColor 변수에 저장된 값이 출력(out)
  - Fragment의 결과물로는 [0.0, 1.0]의 RGBA 컬러값을 사용 (A가 1.0이면 불투명, 0.0이면 완전 투명

```
#version 330 core
out vec4 FragColor;

void main()
{
    FragColor = vec4(1.0f, 0.5f, 0.2f, 1.0f);
}
```

- FS의 컴파일도 VS와 거의 유사함
  - GL_FRAGMENT_SHADER 상수 사용

```
unsigned int fragmentShader;
fragmentShader = glCreateShader(GL_FRAGMENT_SHADER);
glShaderSource(fragmentShader, 1, &fragmentShaderSource, NULL);
glCompileShader(fragmentShader);
```

Shader program

- FS와 VS가 생성되었으면 두 셰이더를 연결해야 함
- 셰이더 프로그램 객체 생성

```
unsigned int shaderProgram;
shaderProgram = glCreateProgram();
```

- 이 프로그램에 VS와 FS를 첨부 후 서로 연결

```
glAttachShader(shaderProgram, vertexShader);
glAttachShader(shaderProgram, fragmentShader);
glLinkProgram(shaderProgram);
```

- 링킹 오류 확인

```
glGetProgramiv(shaderProgram, GL_LINK_STATUS, &success);
if(!success) {
    glGetProgramInfoLog(shaderProgram, 512, NULL, infoLog);
    ...
}
```

- 링킹까지 완료되면, 위 프로그램을 사용하겠다고 선언

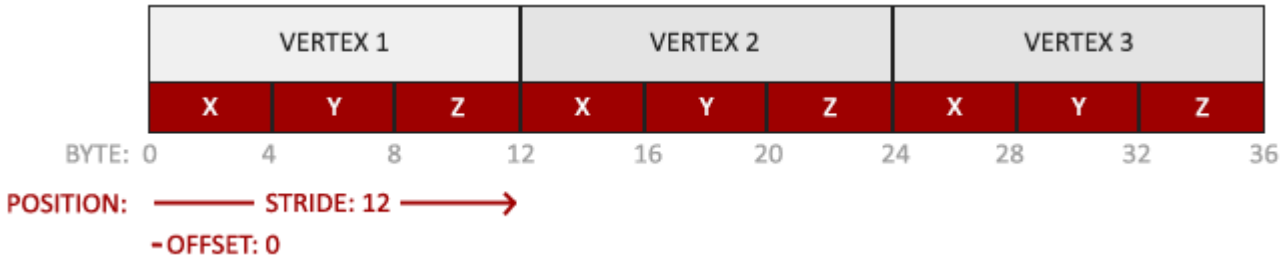
```
glUseProgram(shaderProgram);
```

- 셰이더가 필요 없어지면 삭제

```
glDeleteShader(vertexShader);
glDeleteShader(fragmentShader);
```

Linking Vertex Attributes

- 앞서 선언한 하나의 삼각형을 구성하는 vertex data의 구조




```
float vertices[] = {
    -0.5f, -0.5f, 0.0f,
     0.5f, -0.5f, 0.0f,
     0.0f,  0.5f, 0.0f
};
```

- 메모리 상에 올라온 데이터의 구조가 어떻게 해석되어야 하는지 OpenGL에 전달

```
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 3 * sizeof(float), (void*)0);
glEnableVertexAttribArray(0);
```

  - 1번째 파라미터: 설정한 vertex의 attribute(속성)의 인덱스 지정. 즉, (location=0)에 데이터 전달
  - 2번째, 3번째 파라미터: Vertex 속성의 크기 및 데이터의 타입. 즉, vec3 형태로 데이터 전달
  - 4번째 파라미터: 데이터를 [0, 1] 또는 [-1, 1] 사이로 정규화할지 말지 결정
  - 5번째 파라미터: Stride(인접한 vertex 속성 간의 byte offset). Padding이 존재하는 경우 이를 설정 가능. 0으로 놓으면 padding 없이 빽빽하게 데이터가 저장됨을 의미
  - 6번째 파라미터: 버퍼에서 데이터가 시작하는 위치의 offset으로, void*형으로 변환

Linking Vertex Attributes

- 총정리 – OpenGL에서 객체를 그리는 방법

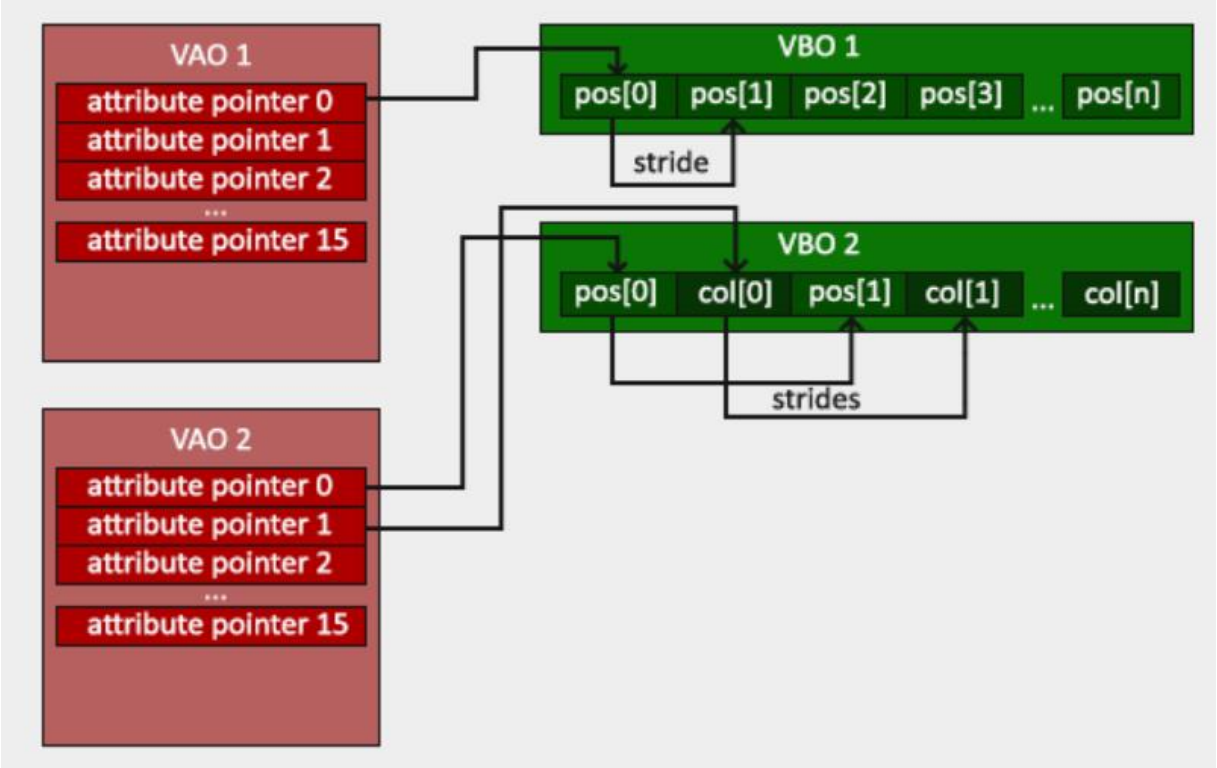
```
// 0. copy our vertices array in a buffer for OpenGL to use
glBindBuffer(GL_ARRAY_BUFFER, VBO);
glBufferData(GL_ARRAY_BUFFER, sizeof(vertices), vertices, GL_STATIC_DRAW);
// 1. then set the vertex attributes pointers
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 3 * sizeof(float), (void*)0);
glEnableVertexAttribArray(0);
// 2. use our shader program when we want to render an object
glUseProgram(shaderProgram);
// 3. now draw the object
someOpenGLFunctionThatDrawsOurTriangle();
```

- 만약 객체의 개수가 많다면?
  - Vertex attribute pointer를 따로 저장하는 VAO(vertex array object)를 사용하면 효율적

Vertex Array Object

- VAO는 VBO와 같이 바인딩 가능
- Vertex 속성 포인터를 구성할 때 딱 한 번 VAO에 저장된 vertex 속성을 호출
  - 다른 VAO를 바인딩함으로써 vertex 데이터와 속성들을 손쉽게 교체 가능
- Core Profile은 VAO 사용 요구

- VAO에 저장되는 내용
  - glEnableVertexAttribArray()/ glDisableVertexAttribArray() 호출 내용
  - glVertexAttribPointer() 설정 내용
  - glVertexAttribPointer()에 의해 vertex attributes와 연결된 VBOs



Vertex Array Object

- VAO + VBO 사용 코드

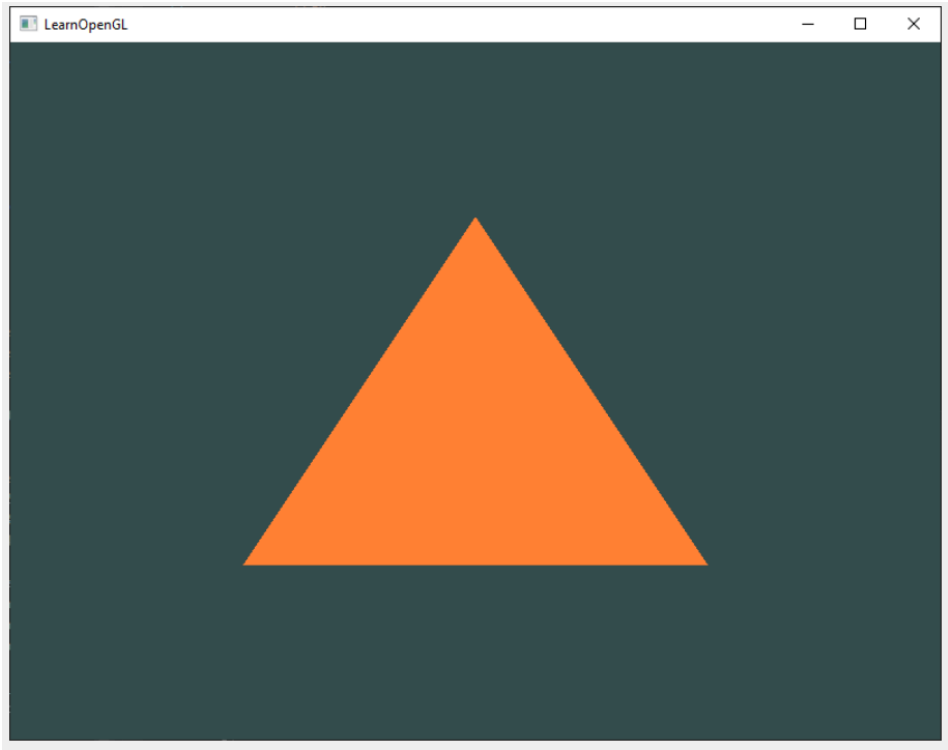
```
// ..:: Initialization code (done once (unless your object frequently changes)) :: ..
// 1. bind Vertex Array Object
glBindVertexArray(VAO);
// 2. copy our vertices array in a buffer for OpenGL to use
glBindBuffer(GL_ARRAY_BUFFER, VBO);
glBufferData(GL_ARRAY_BUFFER, sizeof(vertices), vertices, GL_STATIC_DRAW);
// 3. then set our vertex attributes pointers
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 3 * sizeof(float), (void*)0);
glEnableVertexAttribArray(0);


[...]

// ..:: Drawing code (in render loop) :: ..
// 4. draw the object
glUseProgram(shaderProgram);
glBindVertexArray(VAO);
someOpenGLFunctionThatDrawsOurTriangle();
```

The triangle we've all been waiting for

- 현재 활성화된 shader 프로그램과,
  VAO에 의해 간접적으로 바인딩된 vertex 데이터로,
  0번부터 총 3개의 vertex를 이용해 삼각형을 그림

```
glUseProgram(shaderProgram);
glBindVertexArray(VAO);
glDrawArrays(GL_TRIANGLES, 0, 3);
```

- 전체 소스 코드 (실습 때 사용)
  - Code Viewer. Source code: src/1.getting_started/2.1.hello_triangle/hello_triangle.cpp (learnopengl.com)



Element Buffer Objects

- Vertex를 공유하는 object에 유용하게 사용 가능 (triangle mesh 등)
- 기존 방법으로 사각형 그리기
  - 인접한 삼각형 2개의 vertex를 각각 설정

```
float vertices[] = {
    // first triangle
     0.5f,  0.5f, 0.0f,  // top right
     0.5f, -0.5f, 0.0f,  // bottom right
    -0.5f,  0.5f, 0.0f,  // top left
    // second triangle
     0.5f, -0.5f, 0.0f,  // bottom right
    -0.5f, -0.5f, 0.0f,  // bottom left
    -0.5f,  0.5f, 0.0f   // top left
};
```

- Indexed drawing 사용
  - 사각형을 구성하는 vertex 4개를 정의한 후, 두 삼각형은 이를 인덱싱(indexing)하도록 함

```
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
};
```

- Indexed drawing 사용시에는 EBO를 사용

Element Buffer Objects

- EBO 사용법은 VBO와 유사

```
unsigned int EBO;
glGenBuffers(1, &EBO);
```

```
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, EBO);
glBufferData(GL_ELEMENT_ARRAY_BUFFER, sizeof(indices), indices, GL_STATIC_DRAW);
```

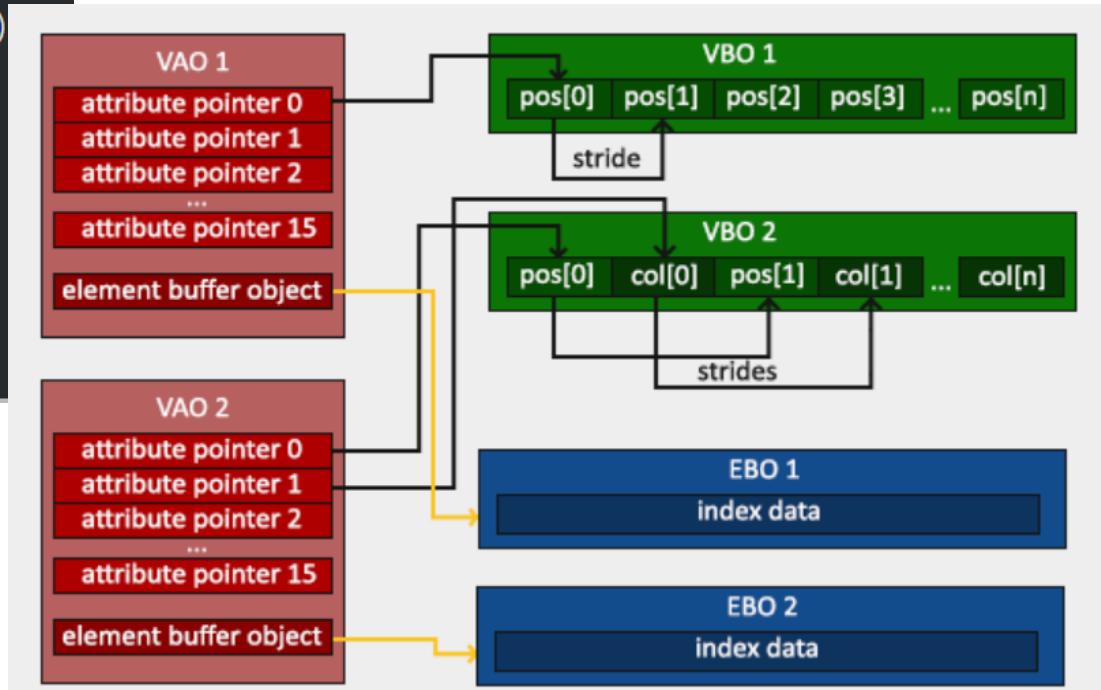
- 단, glDrawArrays() 대신 glDrawElements() 함수 호출

```
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, EBO);
glDrawElements(GL_TRIANGLES, 6, GL_UNSIGNED_INT, 0);
```

  - 삼각형들을, 최종적으로 6개의 vertex를 이용해 그리고, 인덱스는 자연수로 되어 있으며, index offset은 0

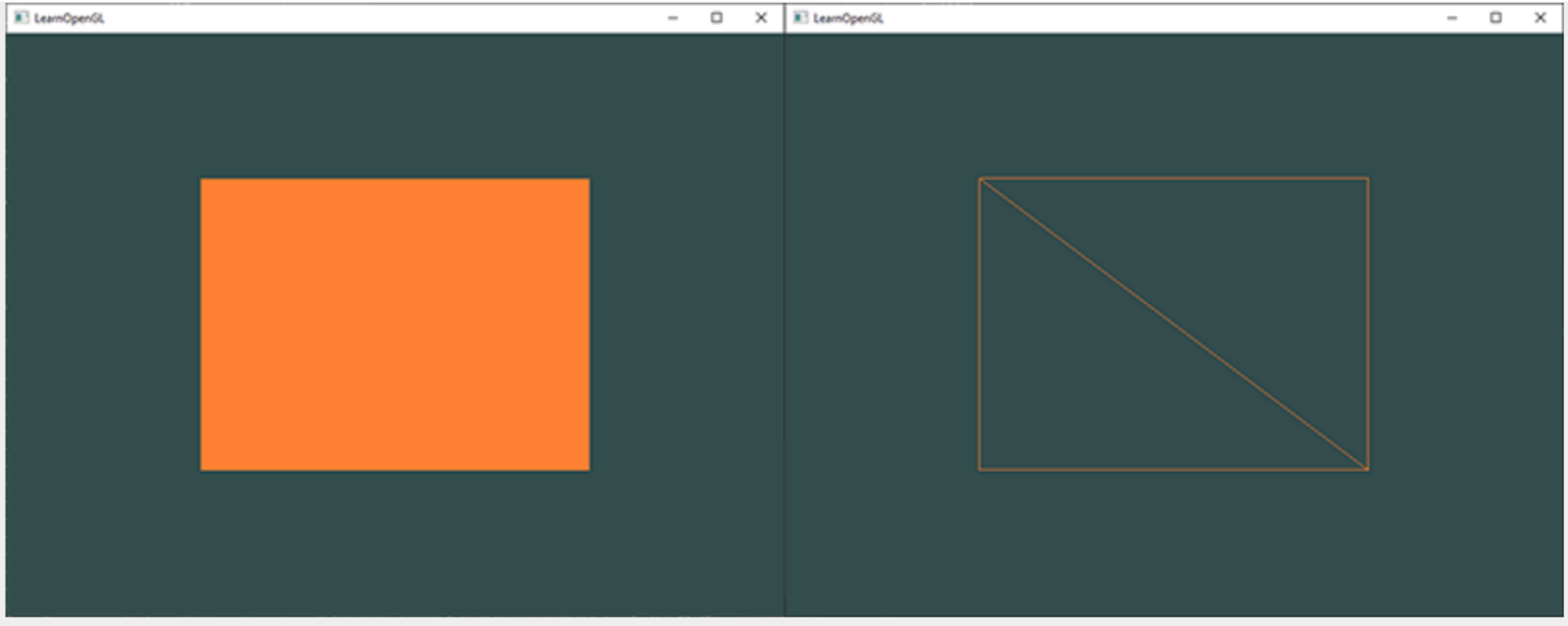
Element Buffer Objects

```
// ..:: Initialization code :: ..
// 1. bind Vertex Array Object
glBindVertexArray(VAO);
// 2. copy our vertices array in a vertex buffer for OpenGL to use
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);
```



Element Buffer Objects

- 출력 결과
  - 왼쪽은 기본값인 glPolygonMode(GL_FRONT_AND_BACK, GL_FILL);를 사용 하였을 때
  - 오른쪽은 wireframe을 그리기 위해 glPolygonMode(GL_FRONT_AND_BACK, GL_LINE);사용시



ⓒ nvidia

Shaders

Shaders & GLSL

- GPU 상에서 그래픽스 파이프라인의 특정 부분을 맡아 각자 실행되는 작은 프로그램
  - 입/출력값 이외에 셰이더 간에 서로 의사소통 할 수 있는 방법은 없음

- GLSL
  - OpenGL에서 셰이더는 GLSL로 작성. Vector/matrix 연산에 유용한 기능 포함
  - 기본 구조

```
#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;
}
```

glGet

- VS에서 지원되는 최대 vertex attribute의 개수를 알고 싶다면?
  - GPU별로 다를 수 있지만 최소 16개의 vec4를 지원

```
int nrAttributes;
glGetIntegerv(GL_MAX_VERTEX_ATTRIBS, &nrAttributes);
std::cout << "Maximum nr of vertex attributes supported: " << nrAttributes << std::endl;
```

- 위와 같이 GPU 스펙상의 특정 속성을 알고 싶으면, glGet*() 함수들을 이용하면 됨
  - [glGet - OpenGL 4 Reference Pages (khronos.org)]
  - [NVIDIA GeForce RTX 3080 performance in GFXBench - unified graphics benchmark based on DXBenchmark (DirectX) and GLBenchmark (OpenGL ES)]

3D Graphics Performance of NVIDIA GeForce RTX 3080

Graphics | Compute | Info

Direct3D API	DirectX 11 Feature Level 11.1
OpenGL API	NVIDIA GeForce RTX 3080/PCIe/SSE2

GL_ALIASED_LINE_WIDTH_MAX	10
GL_ALIASED_LINE_WIDTH_MIN	1
GL_ALIASED_LINE_WIDTH_RANGE	1 10
GL_ALIASED_POINT_SIZE_MAX	2047
GL_ALIASED_POINT_SIZE_MIN	1
GL_ALIASED_POINT_SIZE_RANGE	1 2047
GL_ALPHA_BITS	0
GL_BLUE_BITS	8
GL_COMPRESSED_TEXTURE_FORMATS	GL_COMPRESSED_RGB_S3TC_DXT1_EXT GL_COMPRESSED_RGBA_S3TC_DXT3_ANGLE GL_COMPRESSED_RGBA_S3TC_DXT5_ANGLE GL_PALETTE4_RGB8_OES GL_PALETTE4_RGBA8_OES GL_PALETTE4_R5_G6_B5_OES GL_PALETTE4_RGBA4_OES GL_PALETTE4_RGB5_A1_OES GL_PALETTE8_RGB8_OES GL_PALETTE8_RGBA8_OES GL_PALETTE8_R5_G6_B5_OES GL_PALETTE8_RGBA4_OES

Types

- int, float, double, uint, bool
- Vectors (n은 1~4 사이의 숫자)
  - vecn: the default vector of n floats
  - bvecn: a vector of n booleans
  - ivecn: a vector of n integers
  - uvecn: a vector of n unsigned integers
  - dvecn: a vector of n double components
- Vector의 첫번째~네번째 요소는 xyzw로 접근 가능 (추가로 컬러는 rgba, 텍스쳐 좌표는 stpq 가능)

- Vector datatype은 swizzling 지원

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

- 다른 vector 생성자의 파라미터로도 벡터를 넘길 수 있음

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

Ins and outs

- 각 셰이더는 in과 out 키워드로 입출력을 지정해야 함

Vertex shader

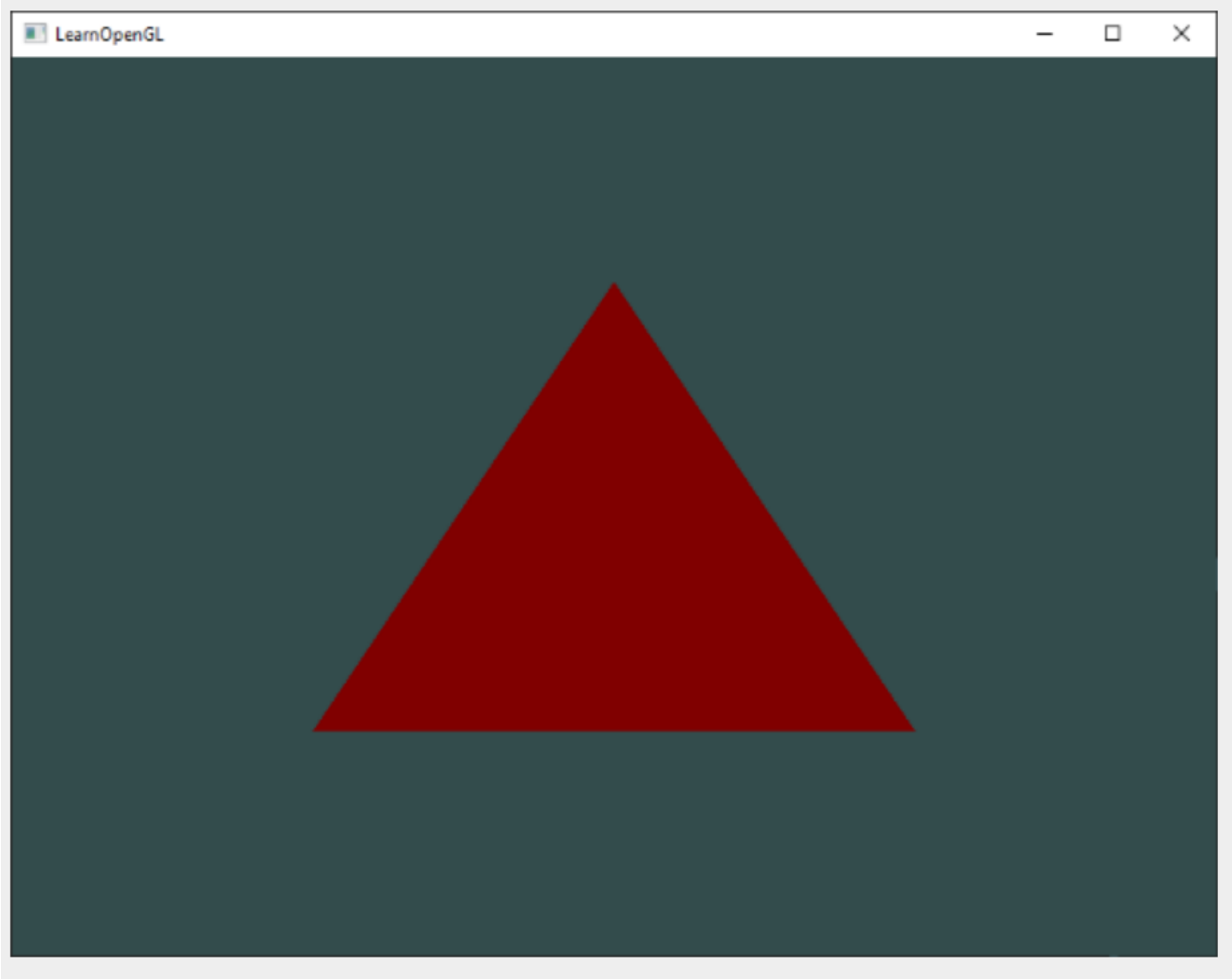
```
#version 330 core
layout (location = 0) in vec3 aPos; // the position variable has attribute position 0

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;
}
```

Ins and outs

- 실행 결과



Uniforms

- CPU에서 GPU로 보내는 셰이더 상의 전역 변수들
- 셰이더 상에서는 uniform 키워드를 사용하여 uniform 변수임을 명시

```
#version 330 core
out vec4 FragColor;

uniform vec4 ourColor; // we set this variable in the OpenGL code.

void main()
{
    FragColor = ourColor;
}
```

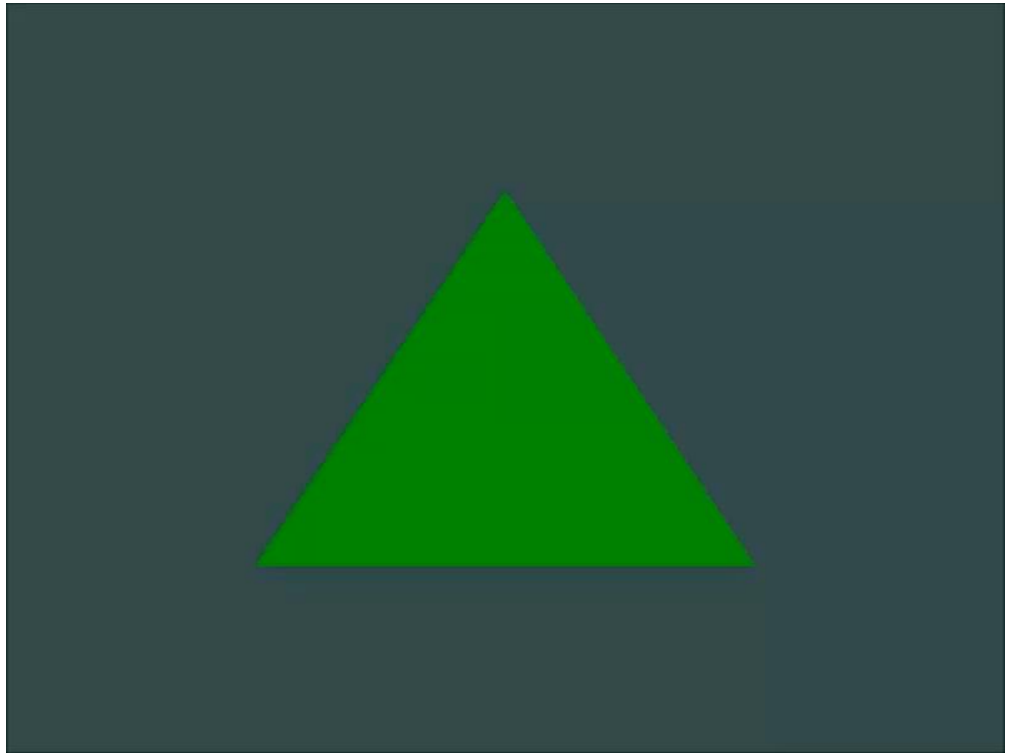
- C 코드에서는 glGetUniformLocation으로 해당 변수의 위치를 얻고, glUniform*() 함수로 값 지정
  - 시간에 따라 ourColor의 색상을 바꾸는 코드

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

Uniforms

- glUniform*() 함수들과 같이, OpenGL은 C 상에서도 함수 오버로딩을 지원하도록 설계
  - 접미사에 따라 파라미터의 형식이 달라짐
  - 예) f: float, i:정수, ui:자연수, 3f:3개의 float, fv: float vector

- 실행 결과



More attributes!

- Vertex별로 색상을 다르게 지정하는 예시
  - 위치값 외에 색상값을 추가로 속성으로 포함
  - VS 상에 이 두 속성을 입력으로 받음을 명시하고, FS로 색상값 출력

```
float vertices[] = {
    // 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
};
```

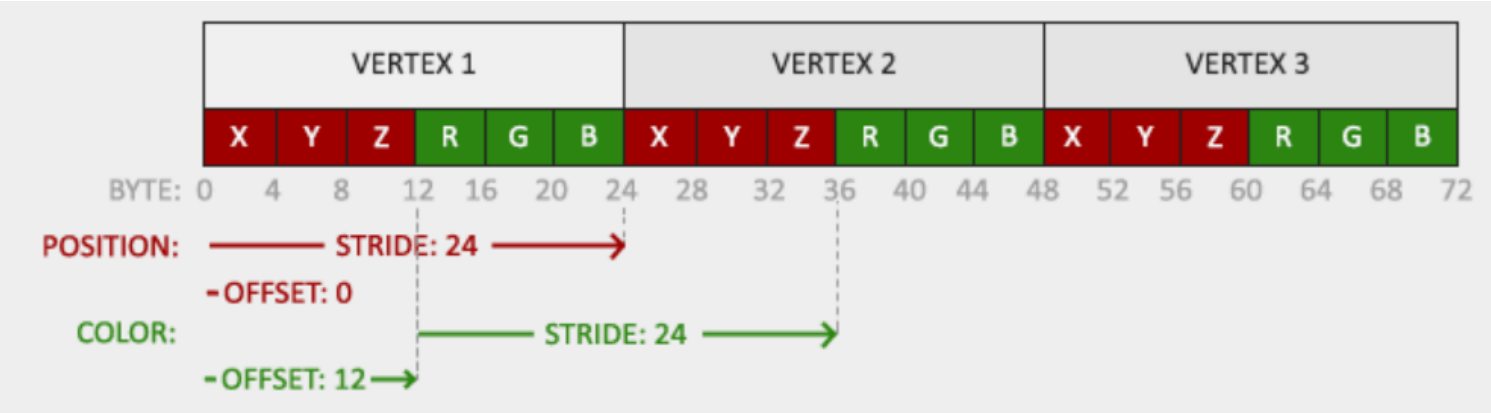
```
#version 330 core
layout (location = 0) in vec3 aPos;   // the position variable has attribute position 0
layout (location = 1) in vec3 aColor; // the color variable has attribute position 1

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
}
```

  - FS에서는 입력받은 색상값을 그대로 출력
    단, 이 색상값은 fragment의 위치에 따라 보간(interpolation) 되어짐

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

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

More attributes!

- VBO 설정

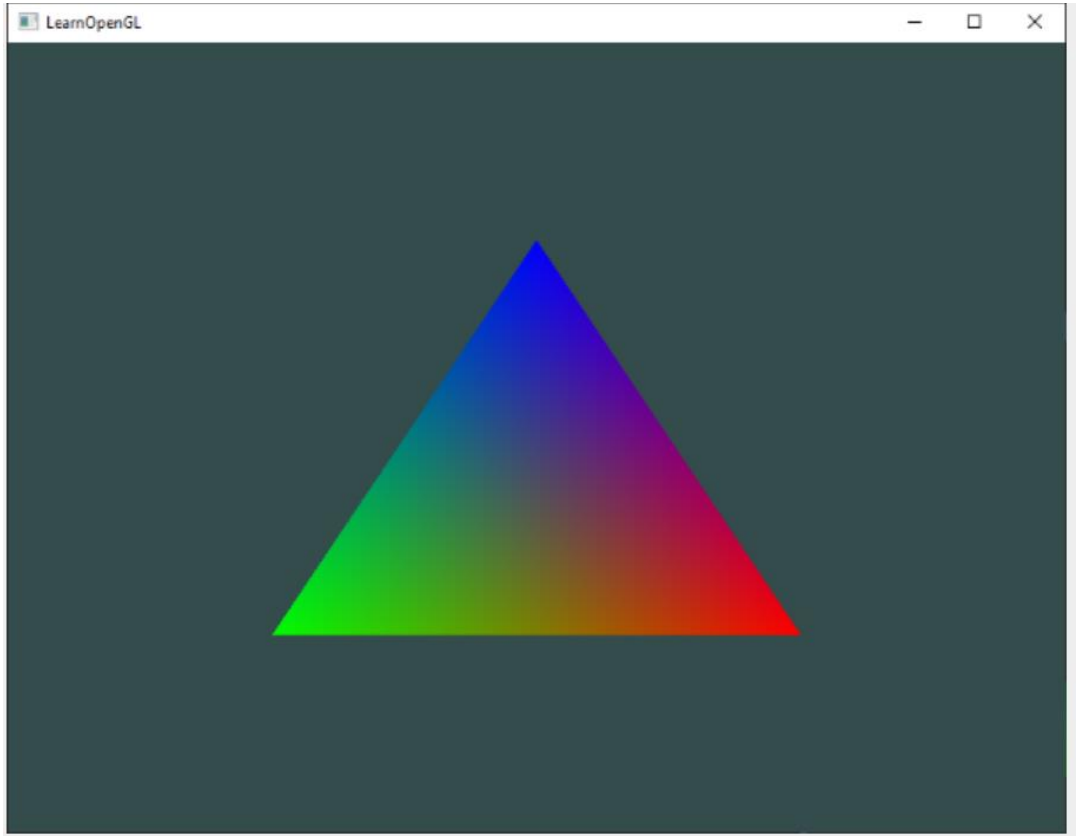



```
// position attribute
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 6 * sizeof(float), (void*)0);
glEnableVertexAttribArray(0);
// color attribute
glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, 6 * sizeof(float), (void*)(3* sizeof(float)));
glEnableVertexAttribArray(1);
```

More attributes!

- 최종 소스 코드
  - Code Viewer. Source code: src/1.getting_started/3.2.shaders_interpolation/shaders_interpolation.cpp (learnopengl.com)

- 실행 결과



Our own shader class

- 반복적인 셰이더 작성&컴파일과 관련된 코드를 한 데 모아 놓은 클래스
  - 별도 셰이더 파일을 이용한 컴파일 기능도 제공
  - OOP의 클래스 개념과 셰이더 프로그램에 대한 개념만 있으면 쉽게 이해 가능

- 아래 코드를 참조하여 필요시 이용
  - Code Viewer. Source code: includes/learnopengl/shader_s.h
  - Code Viewer. Source code: src/1.getting_started/3.3.shaders_class/shaders_class.cpp (learnopengl.com)

```
// 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
```

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

ⓒ nvidia

마무리

마무리

- 이번 시간에는 아래와 같은 내용을 살펴보았습니다.
  - 그래픽스 파이프라인
  - OpenGL로 삼각형을 그리는 방법
  - 셰이더 프로그램의 개요

- 실습 문제
  - 마지막 예제와 바로 직전 예제를 합쳐서,
    uniform을 이용해 RGB로 보간된 삼각형의 밝기가 계속 바뀌도록 해 보세요

- 다음 주차에는 아래 내용을 다룹니다.
  - Textures & Transformations