

OpenGL 셰이더 사용하기

2024-2 컴퓨터 그래픽스

OpenGL 그래픽스 파이프라인

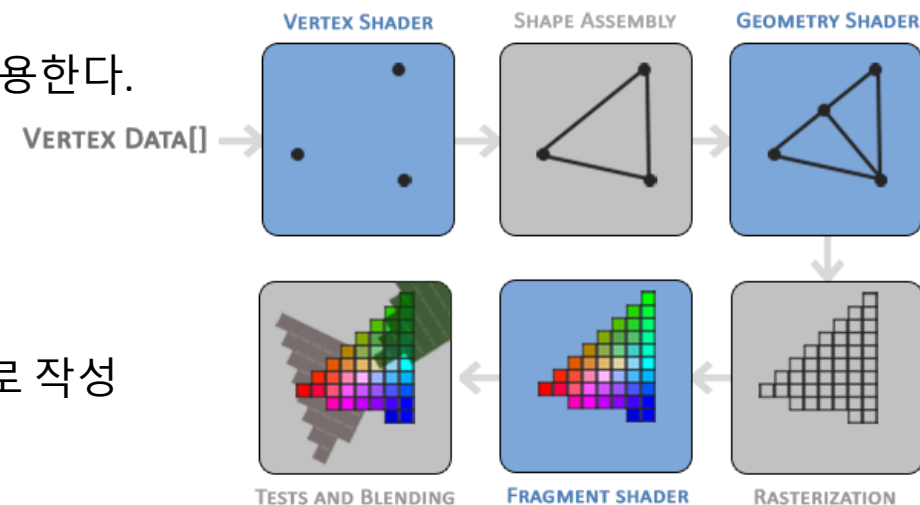
- OpenGL 그래픽스 파이프라인

- 3D 좌표와 데이터를 2D 화면의 픽셀로 변환하는 일련의 단계

- 3D 좌표를 2D 화면에 표시하기 위한 변환 과정
 - 여러 단계로 구성되어 있고, 각 단계는 이전 단계의 출력을 입력으로 사용한다.
 - GPU에서 병렬 처리를 통해 효율적으로 실행된다.

- 주요 단계

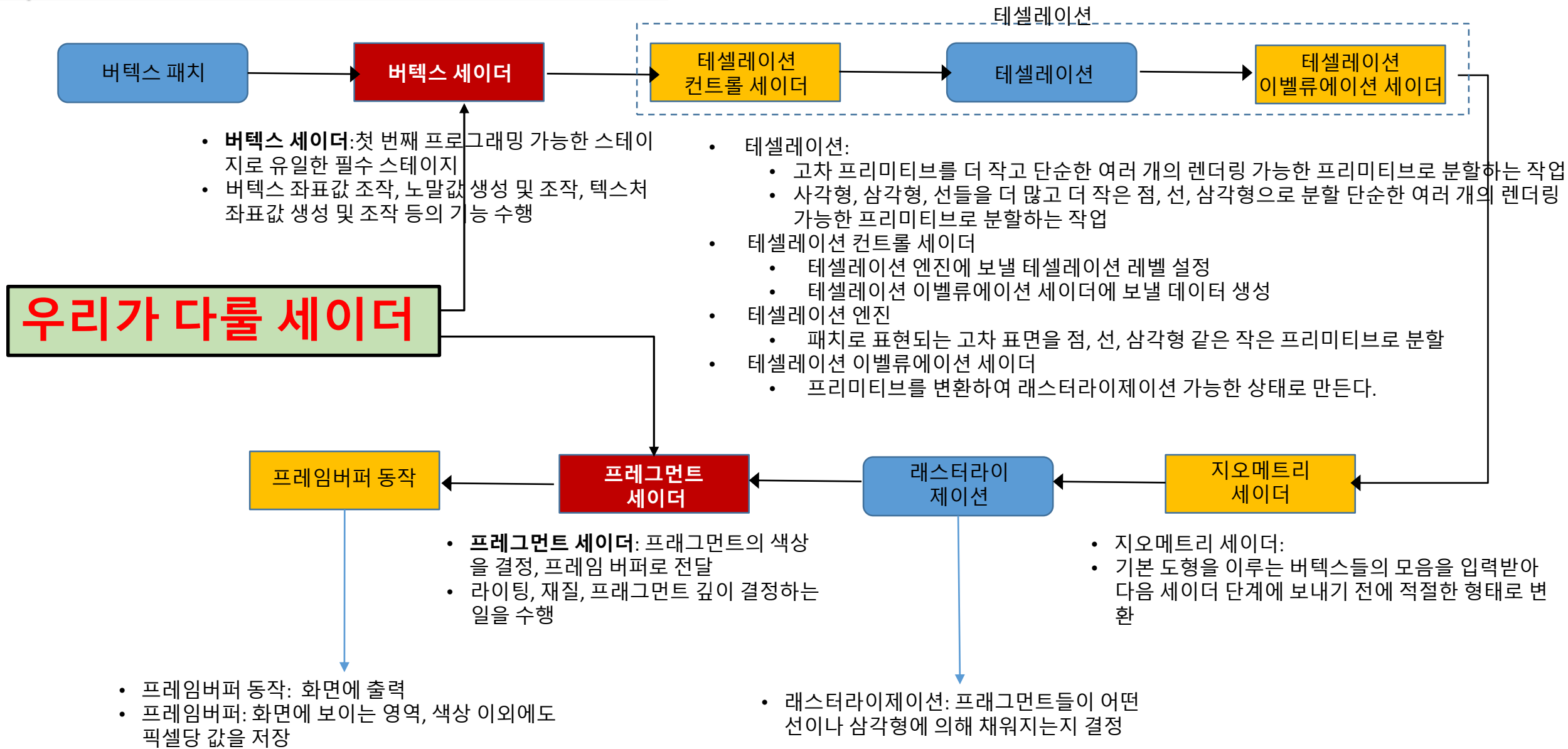
- Vertex Data: 정점 데이터를 정의하고 Vertex Shader에 전달
 - Vertex Shader: 정점 위치 변환 및 조작, 프로그래밍 가능한 단계로 GLSL로 작성
 - Primitive Assembly: 정점들을 연결하여 기본 도형(삼각형 등) 생성
 - Rasterization: 도형을 픽셀로 변환하는 과정
 - Fragment Shader: 각 픽셀(프래그먼트)의 최종 색상 결정, 프로그래밍 가능한 단계로 GLSL로 작성
 - Blending: 프래그먼트를 프레임 버퍼와 결합



- 특징: 셰이더를 통해 파이프라인의 일부 단계를 프로그래밍 가능

- 셰이더: 그래픽스 파이프라인의 여러 단계에서 사용되는 프로그램을 총칭하는 일반적인 용어로 GPU의 특성을 활용하여 고성능 그래픽 처리와 병렬 연산을 가능하게 하는 중요 기술

OpenGL 그래픽스 파이프라인



OpenGL 그래픽스 파이프라인

- 그래픽스 파이프라인에서
 - 모던 OpenGL (OpenGL 3.1 이상) 프로그래밍은
 - 모든 정점 (vertex)와 속성 (attribute) 정보를 배열(array)로 지정한다.
 - Vertex buffer Object와 Vertex Array Object 사용
 - 이 배열을 GPU에 보내어 저장하고 렌더링에 사용한다.
 - 이번 학기는 오픈지엘 3.3 이용하여 Vertex shader와 Fragment shader 구현
 - 두 shader 코드들은 메인 프로그램과는 분리해서 각각 따로 작성한다.

Shader 프로그램 구조

- 프로그램 구조

```
#version version_number core           //--- 사용 버전 선언

in type in_variable_name;              //--- 변수 선언
out type out_variable_name;
uniform type uniform_variable_name;

void main(void)                         //--- 메인 함수: 인자값 없음
{
    //--- 입력 값을 처리
    //--- 필요한 그래픽 작업 수행
    ...
    //--- 출력 변수 저장
    out_variable_name = output data;
}
```

- 오픈지엘의 두 가지 프로파일

- 코어 프로파일 (core profile): 기존의 삭제된 legacy function 을 사용하지 않음
- 호환성 프로파일 (compatibility profile): 오픈지엘의 모든 버전과 하위 호환성 유지

Shader 프로그램 구조

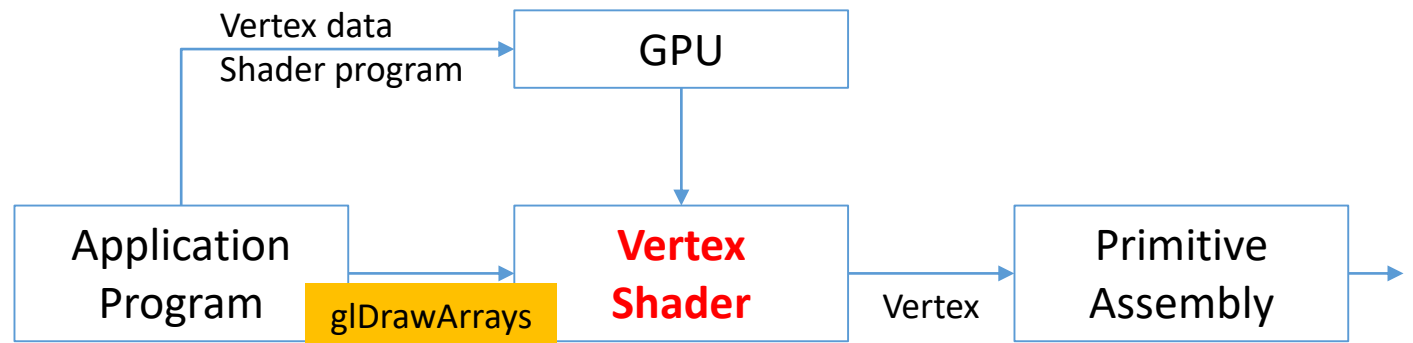
- **Vertex Shader**

- 그리기 명령어에 의해 vertex array object로부터 버텍스 속성을 설정
- 버텍스 변환
- 텍스처 좌표 설정

```
//--- 버텍스 셰이더 예)
#version 330 core

in vec3 vPos;

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



Shader 프로그램 구조

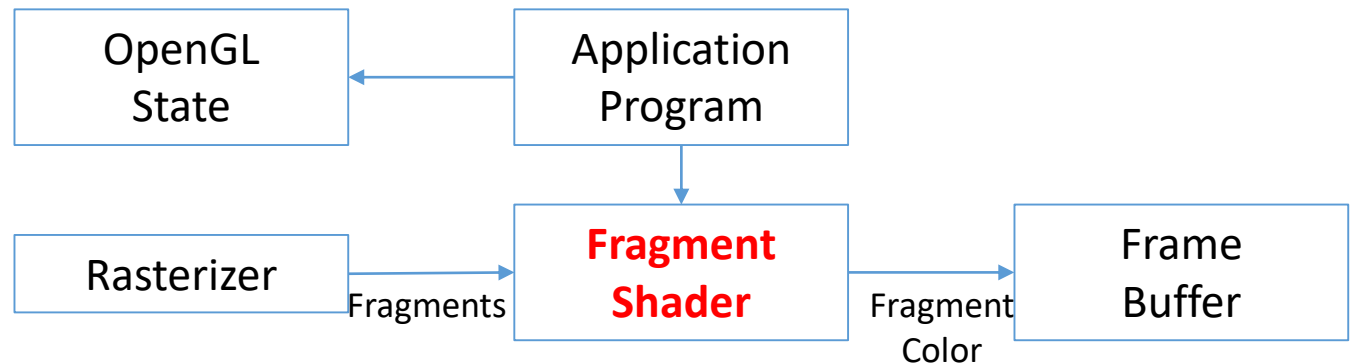
- **Fragment Shader**

- Rasterizer에서 생성된 프래그먼트를 일련의 색상과 깊이 값으로 처리
- 색상 설정
- 조명 설정

```
//--- 프래그먼트 셰이더 예)
#version 330 core

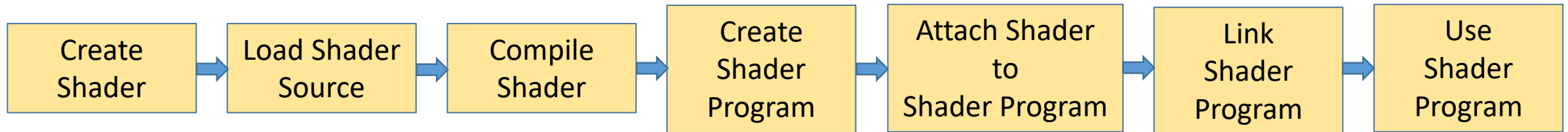
out vec4 FragColor;

void main ()
{
    FragColor = vec4 (1.0, 0.0, 0.0, 1.0);
}
```



셰이더 작성하기

- openGL 셰이더: GLSL(openGL Shader Language)로 작성한다.
 - GLSL:
 - C언어를 기초로 한 상위 레벨 셰이딩 언어로 병렬 실행에 적합한 언어
 - 행렬(Matrices)과 벡터(Vectors) 타입이 기본 타입
 - GLSL에는 포인터 개념이 없음: 동적 할당이 없음
 - 함수 오버로딩을 지원
 - 재귀 호출이 허용되지 않음
 - C언어의 구조체 사용 가능
- 셰이더 처리 과정



GLSL 기본 문법

- 데이터 타입
 - 기본 C언어 스칼라 타입: **int**, **float**, **double**, **uint**, **bool**
 - 벡터 타입: **vec**
 - 지원하는 모든 스칼라 타입에 대한 벡터 타입 지원, 접미어 n 은 인자 개수, $2 \leq n \leq 4$
 - **vec n** / **dvec n** / **bvec n** / **ivec n** / **uvec n**
 - Vec2, vec3, vec4: 2D, 3D, 4D float-타입 벡터
 - 벡터 요소들 필드로 대응되는 구조체로 접근 가능
 - x, y, z, w (좌표값) 또는 r, g, b, a (색상) 또는 s, t, p, q (텍스처 좌표)
 - 벡터의 요소들은 배열처럼 접근 가능
 - `vec4 foo; float x = foo[0]; float y = foo[1];...`
 - 스위즐링 (swizzling)
 - 특정 필드들을 묶어 벡터로 표현하는 것
 - 행렬 타입: **mat**
 - float와 double 타입에 대한 행렬 지원, 접미어 n 과 m 은 각각 행과 열의 개수, $2 \leq n \leq 4, 2 \leq m \leq 4$
 - **mat n** / **mat $m \times n$** / **dmat n** / **dmat $m \times n$**
 - mat2, mat3, mat4: 2x2, 3x3, 4x4 매트릭스(행렬)
 - 행렬은 배열처럼 다룰 수 있고, 열 우선으로 구성
 - 텍스처 샘플러: **sampler**
 - 텍스처 값 (texel)에 접근이 가능한 샘플러 타입
 - **sampler1D**, **sampler2D**, **sampler3D**

GLSL 기본 문법

- 생성자 (Constructor)
 - 변수의 초기화는 C++ 생성자 방식 이용
 - `vec3 aPos = vec3 (0.0f, 0.5f, 0.0f);`
 - 벡터 생성자는 저장할 값의 숫자를 지정한다.
 - `vec3 position (1.0);` `//--- vec3 position (1.0, 1.0, 1.0);`
 - 벡터에 하나의 스칼라값을 지정하면 벡터의 모든 요소에 할당
 - `vec4 whiteColor = vec4 (1.0f);` `//--- vec4 whiteColor (1.0, 1.0, 1.0, 1.0);`
 - 스칼라와 벡터, 행렬을 생성자 내에서 혼합해 사용할 수 있다.
 - `vec4 aColor = vec4 (r, vec2(g, b), a);` `//--- vec4 aColor (r, g, b, a)`
 - 배열 생성자는 다음의 문법에 따라 생성자를 호출한다.
 - `const float array[3] = float[3] (2.5, 7.0, 1.5);`
 - `const vec4 vertex[3] = vec4[3] (vec4(0.25, -0.25, 0.5, 1.0), vec4(-0.25, -0.25, 0.5, 1.0), vec4(0.25, 0.25, 0.5, 1.0));`
 - 행렬은 열 우선으로 구성되고, 단일 스칼라 값을 지정하는 경우 대각 행렬이 됨 (대각 요소 외에는 0으로 채워짐)
 - `mat3 m = mat3 (`
 `1.1, 2.1, 3.1, //--- 첫번째 열`
 `1.2, 2.2, 3.2, //--- 두번째 열`
 `1.3, 2.3, 3.3); //--- 세번째 열`
 - `mat3 m = mat3 (1.0);` `//--- 대각선이 1.0이고 다른 요소들은 0.0인 3x3 행렬`

GLSL 기본 문법

- 연산자

- C 언어에서 사용하는 대부분의 연산자가 지원된다.

- 산술 연산자: +, -, *, /, %

- 비트 연산자: &, |, ^, ~, <<, >>

- 논리 연산자: !, &&, ||

- 비교 연산자: <, <=, >, >=, ==, !=

- 삼항 연산자: ?

- 대입 연산자: =, +=, -=

- 대부분의 연산자는 C언어의 우선순위 규칙과 동일하게 연산자 사용 가능

- 이항 연산일 경우, 벡터 연산은 요소별 연산으로 진행

- 예)

```
vec3 a = vec3(1.0, 2.0, 3.0);  
vec3 b = vec3(0.1, 0.2, 0.3);  
vec3 c = a + b;           // = vec3(1.1, 2.2, 3.3)  
vec3 d = a * b;           // = vec3(0.1, 0.4, 0.9)
```

- 예)

```
mat2 a = mat2(1., 2., 3., 4.);  
mat2 b = mat2(10., 20., 30., 40.);  
mat2 c = a * b;           // = mat2( 1. * 10. + 3. * 20., 2. * 10. + 4. * 20.,  
                           //      1. * 30. + 3. * 40., 2. * 30. + 4. * 40.)
```

- 예)

```
vec3 a = vec3(1.0, 2.0, 3.0);  
mat3 m = mat3(1.0);  
float s = 10.0;  
vec3 b = s * a;           // vec3(10.0, 20.0, 30.0)  
vec3 c = a * s;           // vec3(10.0, 20.0, 30.0)  
mat3 m2 = s * m;         // = mat3(10.0)  
mat3 m3 = m * s;         // = mat3(10.0)
```

GLSL 기본 문법

- 예) 행렬-벡터 곱셈 (벡터는 행렬의 오른쪽에서 곱셈 진행해야 함)

```
vec2 v = vec2(10., 20.);
```

```
mat2 M = mat2(1., 2., 3., 4.);
```

```
vec2 w = M * v;
```

```
// = vec2(1. * 10. + 3. * 20., 2. * 10. + 4. * 20.)
```

$$Mv = \begin{bmatrix} m_{11} & m_{12} \\ m_{21} & m_{22} \end{bmatrix} \begin{bmatrix} v_1 \\ v_2 \end{bmatrix} = \begin{bmatrix} m_{11}v_1 + m_{12}v_2 \\ m_{21}v_1 + m_{22}v_2 \end{bmatrix}$$

GLSL 기본 문법

- 연산자

- 비트 연산은 허용되지 않음
- 스위즐(swizzling) 연산자 (C의 선택 연산자(.))
 - 행렬 및 벡터형으로부터 복수의 구성 요소들을 선택
 - [] 또는 . 을 이용하여 벡터 및 행렬 요소에 접근 가능

- vec4 타입의 변수는 각각

- x, y, z, w

→ 좌표값으로 사용할 때

- 또는 r, g, b, a

→ 색상으로 사용할 때

- 또는 s, t, p, q

→ 텍스처 좌표값으로 사용할 때

의 요소로 사용가능하다

- vec4 a = vec3 (0.0f, 1.0f, 2.0f, 3.0f);

//--- a[2] == a.z == a.b == a.p → 아무 요소로나 사용 가능

- 요소 선택자를 이용하여 재배치 및 복제, 일부 요소 수정 가능

- vec4 a;

- vec4 b = a.xyxx;

- vec3 c = b.zwx;

- vec4 d = a.xxyy + b.yxzy;

GLSL 기본 문법

- 한정자 (qualifier)
 - 한정자는 변수나 함수의 형식 인자의 앞에 붙여 해당 변수나 인자의 특성을 결정짓는다.
 - 세이더로 정보를 주고 받을 때는 변수를 사용:
 - Built-in 변수
 - 사용자 정의 변수: 선언할 때 가장 앞에 한정자를 붙여 변수의 종류를 결정한다.
 - Storage 한정자
 - const, in, out (varying), uniform
 - Layout 한정자
 - layout
- 한정자 (qualifier)
 - const
 - 상수
 - 세이더에 의해 실행 중에 변수가 변경되는 것을 방지
 - 사용 예)
 - `const float one = 1.0;`
 - `const vec3 origin = vec3 (1.0, 2.0, 3.0);`

GLSL 기본 문법

- 한정자 (qualifier)

- in/out (varying)

- 입력/출력 변수로 CPU로부터 데이터를 전달받고 각 셰이더의 연산 결과를 전달할 때 사용
 - 입력 변수는 이전 셰이더 스테이지와 연결, 출력 변수는 다음 셰이더 스테이지로 연결
 - 연결된 스테이지의 입력-출력 변수는 같은 이름으로 사용
 - 3.0 이전에는 varying으로 사용, 그 이후 버전에서는 varying 대신 **in/out**으로 사용
 - in: 함수 내부에서 읽기만 가능, 기본 한정자
 - out: 함수 시작 시점에는 정의되지 않은 값을 가지며, 함수 종료 시점에 가진 값을 호출자에게 전달함 (함수 내부에서 호출자로의 쓰기만 가능)

- 사용 예)

- **//--- vertex shader**

```
in vec3 position;           //--- CPU로부터 받는 정점 속성
const vec4 red = vec4 (1.0, 0.0, 0.0, 0.0);
out vec3 color;             //--- 프래그먼트 셰이더로 보내기
```

```
void main ()
{
    gl_Position = vec4 (position, 1.0);
    color = red;
}
```

- **//--- fragment shader**

```
in vec3 color;              //--- 버텍스 셰이더에서 받기
out vec4 outColor;          //--- 프레임 버퍼로 보내기
```

```
void main ()
{
    outColor = vec4 (color, 1.0);
}
```

GLSL 기본 문법

- 한정자 (qualifier)
 - **uniform**: CPU위의 응용프로그램에서 GPU 위의 셰이더로 데이터를 전달하는 한 방법
 - 모든 단계의 셰이더에서 접근 가능한 전역 변수
 - 필요한 셰이더에서 전역 변수 형태로 선언한 후 사용
 - 셰이더가 아니라 응용 프로그램에서 값을 설정할 수 있고, 셰이더에서는 디폴트 값으로 초기화할 수 있다.
 - 리셋을 하거나 업데이트를 하기 전까지 그 값을 계속 유지
- 사용 예)
 - 변수 선언
 - 변수 선언 시 타입, 이름과 함께 uniform을 추가한다.
 - 예) `uniform vec4 outColor;`
 - 사용되지 않는 uniform 변수는 오류를 발생시킬 수 있다.
 - 값 가져오기
 - uniform 변수의 이름을 사용하여 위치(index)를 가져와서 값을 수정 가능
 - `GLuint glGetUniformLocation (GLuint program, const GLchar *name);`
 - uniform 변수의 위치를 가져온다.
 - program: 프로그램 이름
 - name: uniform 변수 이름
 - 리턴값: uniform 변수 위치 (-1: 위치를 찾지 못함)
 - `void glUniform{1|2|3|4}{f|i|ui} (GLuint location, {GLfloat v0, GLfloat v1, GLfloat v2, GLfloat v3});`
 - 현재 프로그램에서 uniform 변수의 값을 명시
 - location: 수정할 uniform 변수의 위치
 - v0, v1, v2, v3: 사용될 uniform 변수 값

GLSL 기본 문법

- 사용 예) uniform 변수를 사용하여 색상 설정하기
 - 프래그먼트 셰이더에서 uniform 변수 선언

```
#version 330 core
out vec4 FragColor;
```

```
uniform vec4 outColor;
```

//--- 메인 프로그램에서 변수 설정

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

- 메인 프로그램에서 변수에 값 지정

```
float r=0.2, g=0.3, b=0.7;
```

//--- shaderProgram에서 "outColor"라는 이름의 uniform 변수의 위치를 가져온다.

```
int vColorLocation = glGetUniformLocation(shaderProgram, "outColor");
```

```
glUseProgram(shaderProgram);
```

//--- 사용 셰이더 프로그램 설정

//--- vColorLocation에 (outColor) 이름의 uniform 변수의 값에 r, g, b, 1.0값을 저장

```
glUniform4f(vColorLocation, r, g, b, 1.0);
```

GLSL 기본 문법

- 한정자 (qualifier)

- layout

- 셰이더 간에 연결되고 CPU코드와도 통신이 필요할 때 사용
 - 1개 이상의 속성을 가진 데이터들이 저장된 버퍼의 속성 순서를 설정
 - 형태: **layout (qualifier, qualifier2 = value, ...) variable definition**
 - 버텍스 셰이더:
 - **layout** (location = 1) in vec3 vColor; //--- 버텍스 셰이더 입력 1번 위치에 vColor 값 지정
 - **layout** (location = 2) in vec4 values[4]; //--- 버텍스 셰이더 입력 2, 3, 4, 5번 위치에 values[0]~values[3] 지정
 - **glBindAttribLocation** 함수를 사용하여 layout으로 설정된 변수의 입력 위치 사용
 - 프래그먼트 셰이더:
 - **layout** (location = 6) out vec4 outColor; //--- 프래그먼트 셰이더에서 출력할 데이터 6번에 outColor 지정
 - **glBindFragDataLocation** 함수를 사용하여 layout으로 설정된 변수 사용

- attribute

- 버텍스 셰이더에서만 사용할 수 있고, 버텍스 셰이더에 각 정점 입력을 지정하기 위해 사용
 - OpenGL 응용 프로그램으로부터 값을 전달받을 수 있다.
 - position, normal, texture coordinate, color 등의 정보가 전달
 - built-in attribute들 (버텍스 셰이더): gl_Color, gl_Normal, gl_Vertex, gl_Position, gl_MultiTexCoord2...

GLSL 기본 문법

- 사용 예) layout을 이용하여 2개의 속성 변수 설정하기
 - 버텍스 셰이더에서 속성 설정

```
#version 330 core
layout (location = 0) in vec3 aPos;           //--- 위치 변수: attribute position 0
layout (location = 1) in vec3 aColor;         //--- 컬러 변수: attribute position 1

out vec3 ourColor;                             //--- 컬러를 fragment shader로 출력

void main()
{
    gl_Position = vec4(aPos, 1.0);
    ourColor = aColor;                         //--- 컬러를 ourColor에 설정
}
```

GLSL 기본 문법

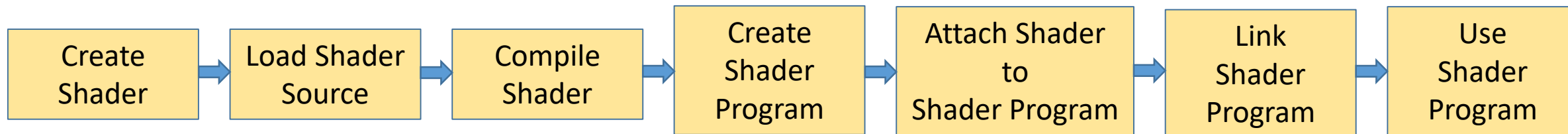
- GLSL에 이미 지정되어 있는 built-in 변수들
 - 버텍스 셰이더:
 - vec4 `gl_Position`: 해당 시점의 버텍스 position
 - int `gl_VertexID`: 해당 시점에 처리될 버텍스 인덱스, 이 인덱스를 사용하여 각 버텍스에 대해 다른 위치를 지정할 수 있다.
 - `glDrawArrays()`에 입력으로 들어온 두번째 인자부터 count의 수만큼 한개씩 증가하게 된다.
 - 프래그먼트 셰이더
 - vec4 `gl_FragColor`: 프래그먼트 색상값
 - vec4 `gl_FragCoord`: 윈도우 공간에서 프래그먼트의 좌표값
 - float `gl_FragDepth`: 프래그먼트 깊이 값

GLSL 기본 문법

- GLSL에 저장되어 있는 built-in 함수들
 - 삼각함수
 - `sin, cos, tan`
 - `asin, acos, atan`
 - 수학함수
 - `pow, log, exp, sqrt, abs, max, min, round, mod, clamp, mix, step, smoothstep`
 - `mix (a, b, t): $a + t(b-a)$`
 - `step (limit, a): 0 when $a < \text{limit}$, 1 when $a > \text{limit}$`
 - `smoothstep (a0, a1, b): 0 when $b < a0$, 1 when $b > a1$`
 - 기하학 계산 함수
 - `length, distance, dot product, cross product, normalize`
- 함수
 - 함수의 선언 및 정의는 C 문법과 거의 동일하게 지원, 사용자 정의 함수를 만들 수 있다.
 - C 문법과 거의 동일, 함수 overloading이 가능, 재귀 호출 할 수 없다, 포인터 타입이 없어 인자는 모두 call-by-value 형태로 전달
- 그 외
 - 조건문, 반복문, 배열, 구조체 등은 C 문법과 거의 동일하게 지원됨

셰이더 프로그램 만들기

- OpenGL 셰이딩 언어 GLSL을 사용하여 셰이더 작성
 - GLSL의 컴파일러는 OpenGL에 내장되어 있음
 - 셰이더 프로그램을 만들려면
 - 1) 셰이더 코드 작성: 필요한 셰이더 코드 작성
 - 버텍스 셰이더와 프래그먼트 셰이더 메인 코드 작성 → 버텍스 셰이더 메인 코드, 프래그먼트 셰이더 메인 코드 작성
 - 2) 셰이더 객체 만들기: 작성한 버텍스 셰이더, 프래그먼트 셰이더 코드로 셰이더 객체 만들기
 - 셰이더 객체 만들기 → 버텍스 셰이더 객체와 프래그먼트 셰이더 객체 만들기: **glCreateShader**
 - 셰이더 객체에 셰이더 코드 붙이기 → 버텍스 셰이더와 프래그먼트 셰이더 코드를 각각 객체에 붙이기: **glShaderSource**
 - 셰이더 객체 컴파일하기 → 버텍스 셰이더 객체와 프래그먼트 셰이더 객체 컴파일하기: **glCompileShader**
 - 3) 셰이더 프로그램 만들기: 버텍스 셰이더 객체와 프래그먼트 셰이더 객체를 링크하여 셰이더 프로그램 만들기
 - 셰이더 프로그램 생성 → 셰이더 객체를 링크 할 셰이더 프로그램 생성: **glCreateProgram**
 - 셰이더 프로그램에 셰이더 객체들을 붙이기 → 버텍스셰이더와 프래그먼트셰이더 객체를 셰이더 프로그램에 붙이기: **glAttachShader**
 - 셰이더 프로그램 링크 → 셰이더 프로그램을 링크: **glLinkProgram**
 - 4) 셰이더 프로그램 활성화: 링크된 셰이더 프로그램 사용하도록 셰이더 프로그램 활성화 하기
 - 셰이더 객체들이 링크된 셰이더 프로그램을 활성화: **glUseProgram**



셰이더 코드

- 셰이더 코드 작성하기: Vertex shader
 - 버텍스 셰이더: 버텍스를 배치하여 객체의 위치를 정한다.
 - 개별 정점의 처리를 하는 렌더링 파이프라인의 프로그래밍 가능한 첫번째 셰이더 단계
 - 그리기 명령에 의해 정점 배열 객체에서 지정된 정점 속성 데이터가 제공된다.
 - 정점 스트림에서 단일 정점을 수신하고, 출력 정점 스트림에 단일 정점을 생성한다. 입력 정점에서 출력 정점으로 1:1 매핑
 - 투영 공간으로의 변환을 수행

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

셰이더 코드

- 셰이더 코드 작성하기: Fragment shader
 - 프래그먼트 셰이더: 픽셀의 색상을 정하고 프레임 버퍼로 그 값을 전달한다.
 - 래스터화에 의해 생성된 fragment(조각)을 일련의 색상과 단일 깊이 값으로 처리하는 셰이더 단계
 - 각 픽셀에 대한 fragment 값이 생성: 위치, 정점당 출력 색상값 등이 포함
 - 프래그먼트 셰이더는 단일 프래그먼트를 입력으로 사용하고 단일 프래그먼트를 출력으로 생성

```
#version 330 core
out vec4 color;
void main ()
{
    color = vec4 (1.0, 0.0, 0.0, 1.0);
}
```


셰이더 객체 만들기

- 셰이더 객체 만들기: 버텍스 셰이더 객체, 프래그먼트 셰이더 객체를 각각 만든다.
 - 응용 프로그램에서 진행한다

```
GLchar * vertexSource;           //--- 셰이더 코드 저장할 문자열

//--- 버텍스 셰이더 읽어 저장하고 컴파일 하기
//--- filetobuf: 사용자정의 함수로 텍스트를 읽어서 문자열에 저장하는 함수

vertexSource = filetobuf ("vertex.glsl");
vertexShader = glCreateShader (GL_VERTEX_SHADER);
glShaderSource (vertexShader, 1, &vertexSource, NULL);
glCompileShader (vertexShader);
```

셰이더 프로그램 만들고 활성화 하기

- 만들어진 한 개 이상의 셰이더 객체를 결합하여 한 개의 셰이더 프로그램으로 만든다.
 - 셰이더 객체를 만든 후 셰이더 프로그램을 만든다.

```
GLuint shaderID;  
shaderID = glCreateProgram();
```

```
//--- 셰이더 프로그램 만들기
```

```
glAttachShader (shaderID, vertexShader);  
glAttachShader (shaderID, fragmentShader);
```

```
//--- 셰이더 프로그램에 버텍스 셰이더 붙이기  
//--- 셰이더 프로그램에 프래그먼트 셰이더 붙이기
```

```
glLinkProgram (shaderID);
```

```
//--- 셰이더 프로그램 링크하기
```

- 셰이더 프로그램 활성화

```
glUseProgram (shaderID);
```

함수 프로토타입

- 작성된 셰이더 코드를 응용 프로그램에서 읽어와 런타임시에 셰이더 소스코드를 동적으로 컴파일
 - 셰이더 객체 생성
 - GLuint **glCreateShader** (GLenum shaderType);
 - 빈 셰이더 객체를 생성하여 리턴한다.
 - shaderType: 생성할 셰이더 타입
 - GL_VERTEX_SHADER, GL_FRAGMENT_SHADER, GL_COMPUTE_SHADER, GL_TESS_CONTROL_SHADER, GL_TESS_EVALUATION_SHADER, GL_GEOMETRY_SHADER
 - 리턴 값: 셰이더 오브젝트
 - 셰이더 코드 읽어오기
 - void **glShaderSource** (GLuint shader, GLsizei count, const GLchar **string, const GLint *length);
 - 셰이더 소스코드를 셰이더 객체로 전달해서 복사본을 유지한다.
 - shader: 셰이더 오브젝트
 - count: string 배열의 구성요소의 개수
 - string: 소스코드가 저장되어 있는 배열 이름
 - length: 소스코드 크기
 - 셰이더 컴파일
 - void **glCompileShader** (GLuint shader);
 - 셰이더 객체에 포함된 소스코드를 컴파일한다.
 - shader: 컴파일 할 셰이더 객체

함수 프로토타입

- 여러 셰이더를 결합하여 한 개의 셰이더 프로그램으로 링크
 - 셰이더 프로그램 만들기
 - GLuint **glCreateProgram** ();
 - 셰이더 객체에 붙일 프로그램 객체를 생성한다.
 - 셰이더 객체들을 프로그램에 첨부
 - void **glAttachShader** (GLuint program, GLuint shader);
 - 셰이더 객체를 프로그램 객체에 붙인다.
 - program: 셰이더를 붙일 프로그램 객체
 - shader: 셰이더 객체
 - 셰이더 프로그램 링크
 - void **glLinkProgram** (GLuint program);
 - 프로그램 객체에 붙인 모든 셰이더 객체를 링크한다.
 - program: 링크할 프로그램 객체
 - 셰이더 객체 삭제하기
 - void **glDeleteShader** (GLuint shader);
 - 셰이더 객체를 삭제한다. 셰이더가 프로그램 객체에 링크되면, 프로그램이 바이너리 코드를 보관하며 셰이더는 더 이상 필요없게 된다.
 - shader: 삭제 할 셰이더

함수 프로토타입

- 프로그램 활성화
 - 셰이더 프로그램 사용
 - `void glUseProgram (GLuint program);`
 - 현재 렌더링 상태에 프로그램 객체를 활성화한다.
 - program: 실행 할 프로그램

함수 프로토타입

- 셰이더 상태 가져오기

- void **glGetShaderiv** (GLuint shader, GLenum pname, GLint *params);

- 셰이더 정보 가져오기

- shader: 셰이더 객체

- pname: 객체 파라미터

- GL_SHADER_TYPE, GL_DELETE_STATUS, GL_COMPILE_STATUS, GL_INFO_LOG_LENGTH, GL_SHADER_SOURCE_LENGTH)

- params: 리턴 값

- GL_SHADER_TYPE: 셰이더 타입 리턴 (GL_VERTEX_SHADER / GL_FRAGMENT_SHADER)

- GL_DELETE_STATUS: 셰이더가 삭제됐으면 GL_TRUE, 아니면 GL_FALSE

- GL_COMPILE_STATUS: 컴파일이 성공했으면 GL_TRUE, 아니면 GL_FALSE

- GL_INFO_LOG_LENGTH: 셰이더의 INFORMATION LOG 크기

- GL_SHADER_SOURCE_LENGTH: 셰이더 소스 크기

- 에러 발생 시, GL_INVALID_VALUE, GL_INVALID_OPERATION, GL_INVALID_ENUM

- void **glGetShaderInfoLog** (GLuint shader, GLsizei maxLength, GLsizei *length, GLchar *infoLog);

- 셰이더 객체의 information log 가져오기

- shader: 셰이더 객체

- maxLength: information log 크기

- length: infoLog 길이

- infoLog: information log

함수 프로토타입

- 프로그램 상태 가져오기
 - void **glGetProgramiv** (GLuint program, GLenum pname, GLint *params);
 - 프로그램 객체 정보 가져오기
 - program: 프로그램
 - pname: 객체 파라미터
 - GL_DELETE_STATUS, GL_LINK_STATUS, GL_VALIDATE_STATUS, GL_INFO_LOG_LENGTH, GL_ATTACHED_SHADERS, GL_ACTIVE_ATOMIC_COUNTER_BUFFERS, GL_ACTIVE_ATTRIBUTES, GL_ACTIVE_ATTRIBUTE_MAX_LENGTH, GL_ACTIVE_UNIFORMS, GL_ACTIVE_UNIFORM_BLOCKS, GL_ACTIVE_UNIFORM_BLOCK_MAX_NAME_LENGTH, GL_ACTIVE_UNIFORM_MAX_LENGTH, GL_COMPUTE_WORK_GROUP_SIZE, GL_PROGRAM_BINARY_LENGTH, GL_TRANSFORM_FEEDBACK_BUFFER_MODE, GL_TRANSFORM_FEEDBACK_VARYINGS, GL_TRANSFORM_FEEDBACK_VARYING_MAX_LENGTH, GL_GEOMETRY_VERTICES_OUT, GL_GEOMETRY_INPUT_TYPE, and GL_GEOMETRY_OUTPUT_TYPE.
 - params: 리턴 값
 - GL_DELETE_STATUS, GL_LINK_STATUS, GL_VALIDATE_STATUS, GL_ATTACHED_SHADERS...
 - void **glGetProgramInfoLog** (GLuint program, GLsizei maxLength, GLsizei *length, GLchar infoLog);
 - 프로그램 객체 information log 가져오기
 - program: 프로그램
 - maxLength: information log 크기
 - length: infoLog 길이
 - infoLog: information log
- 이 함수들은 GL 2.0 이상에서 지원됨

셰이더 사용하여 도형 그리기

- 셰이더 프로그램을 만든 후, 셰이더 프로그램을 사용하여 도형을 그린다.

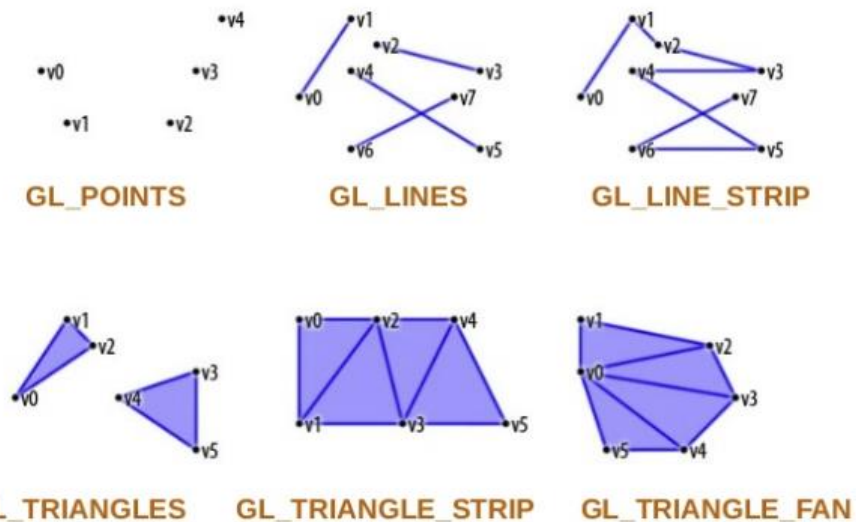
```
void DrawScene ()  
{  
    //--- 필요한 작업 진행  
  
    glUseProgram (shaderProgramID);  
  
    glDrawArrays (GL_TRIANGLES, 0, 3);    //--- 도형 그리기  
  
    glutSwapBuffers ();  
}
```


함수 프로토타입

- 배열 데이터로부터 프리미티브 렌더링
 - `void glDrawArrays (GLenum mode, GLint first, GLsizei count);`
 - 배열 데이터로부터 프리미티브 렌더링 하기
 - mode: 렌더링 할 프리미티브의 종류
 - GL_POINTS, GL_LINE_STRIP, GL_LINE_LOOP, GL_LINES, GL_TRIANGLE_STRIP, GL_TRIANGLE_FAN, GL_TRIANGLES 등
 - first: 배열에서 도형의 시작 인덱스
 - count: 렌더링 할 인덱스 개수
 - 프리미티브 (primitive): 오픈지엘 렌더링의 기본 단위로 이용 가능한 가장 단순한 요소 → 점, 선, 삼각형

- Primitive 타입

- Points: [GL_POINTS](#)
- Line
 - Lines: [GL_LINES](#)
 - Line strips: [GL_LINE_STRIP](#)
 - Line loops: [GL_LINE_LOOP](#)
- Triangle
 - Triangles: [GL_TRIANGLES](#)
 - Triangle strips: [GL_TRIANGLE_STRIP](#)
 - Triangle fans: [GL_TRIANGLE_FAN](#)



Primitive types

- GL에서 그리기
 - void **glDrawElements** (GLenum mode, GLsizei count, GLenum type, const GLvoid *indices);
 - 배열 데이터로부터 프리미티브 렌더링 하기, 배열 데이터의 인덱스를 사용
 - mode: 렌더링 할 프리미티브의 종류
 - GL_POINTS, GL_LINE_STRIP, GL_LINE_LOOP, GL_LINES, GL_TRIANGLE_STRIP, GL_TRIANGLE_FAN, GL_TRIANGLES 등
 - count: 렌더링할 요소의 개수
 - type: indices 값의 타입
 - GL_UNSIGNED_BYTE, GL_UNSIGNED_SHORT, 또는 GL_UNSIGNED_INT
 - indices: 바인딩 되는 버퍼의 데이터 저장소에 있는 배열의 첫 번째 인덱스 오프셋

함수 프로토타입

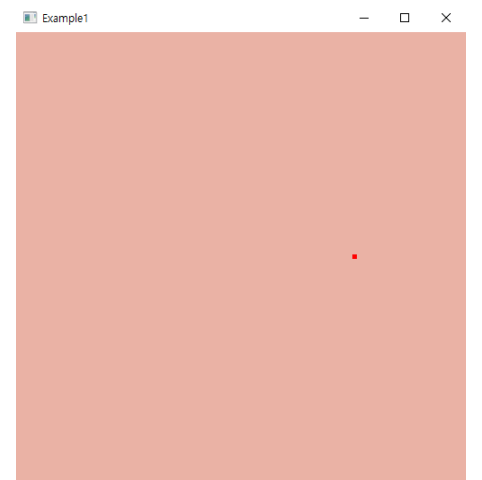
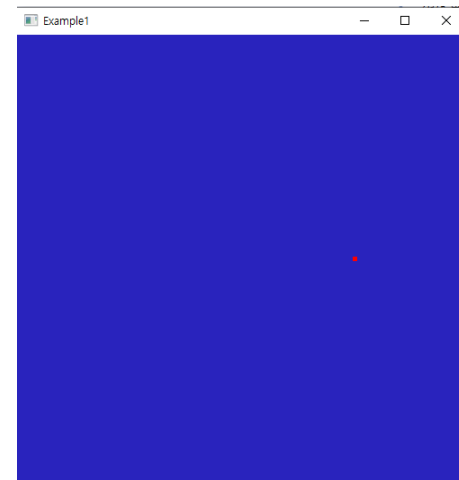
- 기본 속성 바꾸기
 - void **glPointSize** (GLfloat size);
 - 점 크기 설정
 - size: 점의 크기 (초기값: 1)
 - void **glLineWidth** (GLfloat width);
 - 선의 굵기 조정
 - width: 선의 굵기 (초기값: 1)
 - void **glPolygonMode** (GLenum face, GLenum mode);
 - 폴리곤 모드 설정
 - face: 모드를 설정할 면 (GL_FRONT_AND_BACK)
 - mode: 모드
 - GL_POINT, GL_LINE, GL_FILL

<셰이더 사용하여 도형 그리기>

1. 점 그리기
 - 고정된 좌표값의 한 개의 점을 고정된 색상으로 그리기
2. 삼각형 그리기
 - 셰이더에 고정된 좌표값, 고정된 색상으로 삼각형 그리기
3. 삼각형 그리기
 - 메인 프로그램에서 좌표값을 정하여 버텍스 셰이더로 보내서 고정된 색상으로 삼각형 그리기
4. 삼각형 그리기
 - 좌표값과 색상을 메인프로그램에서 설정하기
 - 좌표값과 색상을 버텍스 셰이더로 보내고, 버텍스 셰이더에서 색상을 프래그먼트 셰이더로 보내서 삼각형 그리기
 - 좌표값과 색상을 각각 버텍스 셰이더와 프래그먼트 셰이더로 보내서 삼각형 그리기

1. 첫 번째 셰이더 프로그램: 화면에 점 찍기

- 화면에 점 찍기: 셰이더 내에 좌표값과 색상을 고정하여 사용하기
- 배경색을 칠하고 화면에 고정된 위치에 점 찍기
 - 버텍스 셰이더와 프래그먼트 셰이더는 각각 별개의 파일로 저장하여 사용
 - 버텍스 셰이더: vertex.glsl
 - 프래그먼트 셰이더: fragment.glsl
 - 파일의 확장자는 다르게 설정 가능 (우리는 glsl로 설정하기로 함)
- 셰이더 코드 작성하기
 - 버텍스 셰이더: 점의 좌표값 설정 ➔ (0.5, 0.0, 0.0)
 - 프래그먼트 셰이더: 고정 색상 설정 ➔ (1.0, 0.0, 0.0)
- 메인 프로그램 작성하기
 - 화면 띄우기
 - 버텍스 셰이더 객체와 프래그먼트 셰이더 객체 만들기
 - 셰이더 프로그램 만들어 두 셰이더 객체 연결하기
 - 출력 콜백 함수에서 그리기 함수 호출하여 점 찍기



셰이더 코드

- 버텍스 셰이더: *vertex.glsl* 파일로 저장

```
#version 330 core
```

```
void main()
```

```
{  
    gl_Position = vec4(0.5, 0.0, 0.0, 1.0);    //--- (0.5, 0.0, 0.0) 으로 좌표값 고정  
}
```

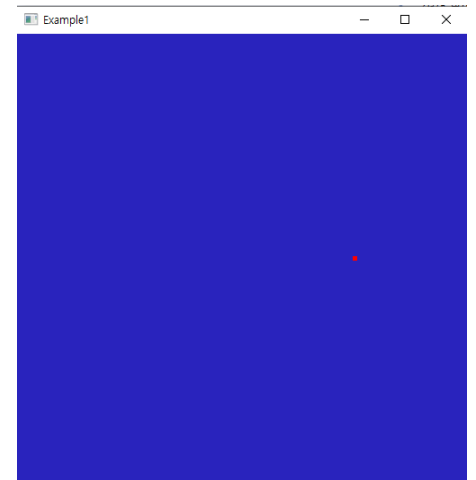
- 프래그먼트 셰이더: *fragment.glsl* 파일로 저장

```
#version 330 core
```

```
out vec4 color;
```

```
void main ()
```

```
{  
    color = vec4 (1.0, 0.0, 0.0, 1.0);    //--- 빨강색으로 색상 고정  
}
```



셰이더 사용하여 점 찍기: 메인 함수

```
//--- 필요한 헤더파일 선언
//--- 아래 5개 함수는 사용자 정의 함수 임
void make_vertexShaders ();
void make_fragmentShaders ();
GLuint make_shaderProgram ();
GLvoid drawScene ();
GLvoid Reshape ( int w, int h );

//--- 필요한 변수 선언
GLint width, height;
GLuint shaderProgramID; //--- 셰이더 프로그램 이름
GLuint vertexShader;    //--- 버텍스 셰이더 객체
GLuint fragmentShader; //--- 프래그먼트 셰이더 객체

//--- 메인 함수
void main (int argc, char** argv)      //--- 윈도우 출력하고 콜백함수 설정
{
    width = 500;
    height = 500;

    //--- 윈도우 생성하기
    glutInit(&argc, argv);
    glutInitDisplayMode (GLUT_DOUBLE | GLUT_RGBA);
    glutInitWindowPosition (100, 100);
    glutInitWindowSize (width, height);
    glutCreateWindow ("Example1");
```

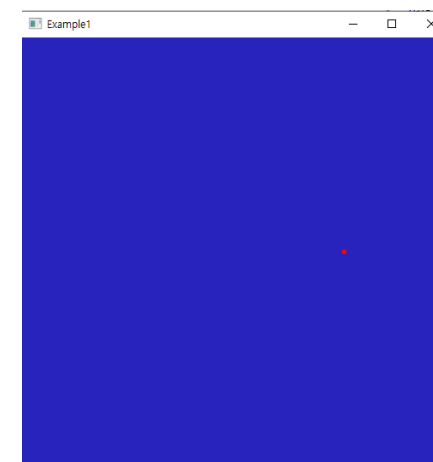
```
//--- GLEW 초기화하기
glewExperimental = GL_TRUE;
glewInit ();

//--- 셰이더 읽어와서 셰이더 프로그램 만들기
make_vertexShaders ();      //--- 버텍스 셰이더 만들기
make_fragmentShaders ();   //--- 프래그먼트 셰이더 만들기
shaderProgramID = make_shaderProgram ();
                           //--- 셰이더 프로그램 만들기

glutDisplayFunc (drawScene); //--- 출력 콜백 함수
glutReshapeFunc (Reshape);

glutMainLoop ();

}
```



셰이더 객체만들기

//--- 버텍스 셰이더 객체 만들기

```
void make_vertexShaders ()
{
    GLchar * vertexSource;

    //--- 버텍스 셰이더 읽어 저장하고 컴파일 하기
    //--- filetoBuf: 사용자정의 함수로 텍스트를 읽어서 문자열에 저장하는 함수

    vertexSource = filetoBuf ("vertex.glsl");
    vertexShader = glCreateShader (GL_VERTEX_SHADER);
    glShaderSource (vertexShader, 1, &vertexSource, NULL);
    glCompileShader (vertexShader);

    GLint result;
    GLchar errorLog[512];
    glGetShaderiv (vertexShader, GL_COMPILE_STATUS, &result);
    if (!result)
    {
        glGetShaderInfoLog (vertexShader, 512, NULL, errorLog);
        std::cerr << "ERROR: vertex shader 컴파일 실패\n" << errorLog << std::endl;
        return;
    }
}
```

//--- 프래그먼트 셰이더 객체 만들기

```
void make_fragmentShaders ()
{
    GLchar *fragmentSource;

    //--- 프래그먼트 셰이더 읽어 저장하고 컴파일하기
    fragmentSource = filetoBuf ("fragment.glsl"); // 프래그먼트 셰이더 읽어오기
    fragmentShader = glCreateShader (GL_FRAGMENT_SHADER);
    glShaderSource (fragmentShader, 1, &fragmentSource, NULL);
    glCompileShader (fragmentShader);

    GLint result;
    GLchar errorLog[512];
    glGetShaderiv (fragmentShader, GL_COMPILE_STATUS, &result);
    if (!result)
    {
        glGetShaderInfoLog (fragmentShader, 512, NULL, errorLog);
        std::cerr << "ERROR: frag_shader 컴파일 실패\n" << errorLog << std::endl;
        return;
    }
}
```


셰이더 프로그램 만들기

//--- 셰이더 프로그램 만들고 셰이더 객체 링크하기

GLuint **make_shaderProgram** ()

{

GLuint shaderID;

shaderID = **glCreateProgram**();

glAttachShader (shaderID, vertexShader);

glAttachShader (shaderID, fragmentShader);

glLinkProgram (shaderID);

glDeleteShader (vertexShader);

glDeleteShader (fragmentShader);

glGetProgramiv (shaderID, GL_LINK_STATUS, &result);

if (!result) {

glGetProgramInfoLog (shaderID, 512, NULL, errorLog);

std::cerr << "ERROR: shader program 연결 실패\n" << errorLog << std::endl;

return false;

}

glUseProgram (shaderID);

return **shaderID**;

}

//--- 셰이더 프로그램 만들기

//--- 셰이더 프로그램에 버텍스 셰이더 붙이기

//--- 셰이더 프로그램에 프래그먼트 셰이더 붙이기

//--- 셰이더 프로그램 링크하기

//--- 셰이더 객체를 셰이더 프로그램에 링크했으므로, 셰이더 객체 자체는 삭제 가능

// ---셰이더가 잘 연결되었는지 체크하기

//--- 만들어진 셰이더 프로그램 사용하기

//--- 여러 개의 셰이더프로그램 만들 수 있고, 그 중 한개의 프로그램을 사용하려면

//--- glUseProgram 함수를 호출하여 사용 할 특정 프로그램을 지정한다.

//--- 사용하기 직전에 호출할 수 있다.

메인/출력 함수 만들기

//--- 출력 콜백 함수

GLvoid **drawScene** ()

```
{  
    GLfloat rColor, gColor, bColor;  
  
    rColor = gColor = 0.0;  
    bColor = 1.0;  
    glClearColor(rColor, gColor, bColor, 1.0f);  
    glClear(GL_COLOR_BUFFER_BIT);
```

glUseProgram (shaderID);

glPointSize(5.0);

glDrawArrays (GL_POINTS, 0, 1);

glutSwapBuffers();

}

//--- 다시그리기 콜백 함수

GLvoid **Reshape** (int w, int h)

```
{  
    glViewport ( 0, 0, w, h );  
}
```

//--- 콜백 함수: 그리기 콜백 함수

//--- 배경색을 파랑색으로 설정

//--- 렌더링하기: 0번 인덱스에서 1개의 버텍스를 사용하여 점 그리기

// 화면에 출력하기

//--- 콜백 함수: 다시 그리기 콜백 함수

파일에서 문자열 읽어오기 샘플

- 파일에서 문자열 읽어오기 샘플 코드 (워밍업에서 구현한 본인의 코드 사용 가능)

```
#define _CRT_SECURE_NO_WARNINGS    //-- 프로그램 맨 앞에 선언할 것
#include <stdlib.h>
#include <stdio.h>
char* filetobuf (const char *file)
{
    FILE *fptr;
    long length;
    char *buf;

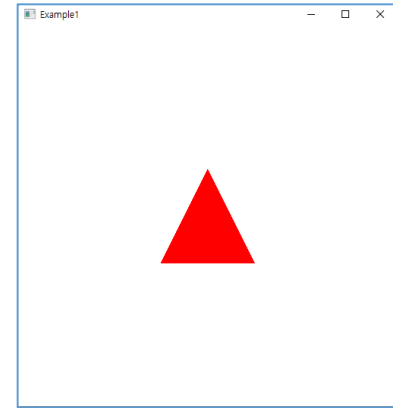
    fptr = fopen (file, "rb");           // Open file for reading
    if (!fptr)                          // Return NULL on failure
        return NULL;

    fseek (fptr, 0, SEEK_END);          // Seek to the end of the file
    length = ftell (fptr);               // Find out how many bytes into the file we are
    buf = (char*) malloc (length+1);    // Allocate a buffer for the entire length of the file and a null terminator
    fseek (fptr, 0, SEEK_SET);          // Go back to the beginning of the file
    fread (buf, length, 1, fptr);       // Read the contents of the file in to the buffer
    fclose (fptr);                      // Close the file
    buf[length] = 0;                   // Null terminator

    return buf;                         // Return the buffer
}
```

2. 두번째 셰이더 프로그램: 셰이더 사용하여 삼각형 그리기 (1)

- 중앙에 빨간색 삼각형 그리기:
 - 버텍스 셰이더: 셰이더 내에서 삼각형의 좌표값을 직접 설정
 - 프래그먼트 셰이더: 빨간색 고정하여 사용하기
- 화면 중앙에 삼각형 그리기
 - 버텍스 셰이더: 삼각형 좌표값 설정
 - 프래그먼트 셰이더: 도형 색상 설정
- 버텍스 셰이더와 프래그먼트 셰이더는 각각 별개의 파일로 저장하여 사용
 - 버텍스 셰이더: vertex.glsl
 - 프래그먼트 셰이더: fragment.glsl
- 메인 프로그램 작성하기
 - 화면 띄우기
 - 버텍스 셰이더 객체와 프래그먼트 셰이더 객체 만들기
 - 셰이더 프로그램 만들어 두 셰이더 객체 연결하기
 - 출력 콜백 함수에서 그리기 함수 호출하여 삼각형 그리기



셰이더 사용하여 삼각형 그리기

```
//--- 응용 프로그램의 삼각형 그리기 함수: 메인 함수, 그리기 함수
//--- 필요한 헤더파일 선언
//--- 아래 5개 함수는 사용자 정의 함수 임
void make_vertexShaders ();
void make_fragmentShaders ();
GLuint make_shaderProgram ();
GLvoid drawScene ();
GLvoid Reshape ( int w, int h );

//--- 필요한 변수 선언
GLint width, height;
GLuint shaderProgramID; //--- 셰이더 프로그램 이름
GLuint vertexShader;    //--- 버텍스 셰이더 객체
GLuint fragmentShader; //--- 프래그먼트 셰이더 객체

//--- 메인 함수
void main (int argc, char** argv)      //--- 윈도우 출력하고 콜백함수 설정
{
    width = 500;
    height = 500;
    //--- 윈도우 생성하기
    glutInit(&argc, argv);
    glutInitDisplayMode (GLUT_DOUBLE | GLUT_RGBA);
    glutInitWindowPosition (100, 100);
    glutInitWindowSize (width, height);
    glutCreateWindow ("Example1");
```

```
//--- GLEW 초기화하기
glewExperimental = GL_TRUE;
glewInit ();

//--- 셰이더 읽어와서 셰이더 프로그램 만들기
make_vertexShaders ();      //--- 버텍스 셰이더 만들기
make_fragmentShaders ();    //--- 프래그먼트 셰이더 만들기
shaderProgramID = make_shaderProgram ();
                                //--- 셰이더 프로그램 만들기

glutDisplayFunc (drawScene);    //--- 출력 콜백 함수
glutReshapeFunc (Reshape);

glutMainLoop ();
}
```

세이더 사용하여 삼각형 그리기

//--- 출력 콜백 함수

void DrawScene ()

{

GLfloat rColor, gColor, bColor;

rColor = nColor = 0.0;

gColor = 1.0;

//--- 배경색을 초록색으로 설정

glClearColor(rColor, gColor, bColor, 1.0f);

glClear (GL_COLOR_BUFFER_BIT);

glUseProgram (**shaderProgramID**);

glDrawArrays (GL_TRIANGLES, 0, 3);

//--- 삼각형 그리기: 0번 인덱스부터 3개의 버텍스를 사용하여 삼각형 그리기

glutSwapBuffers();

}

//--- 다시그리기 콜백 함수

GLvoid **Reshape** (int w, int h)

{

glViewport (0, 0, w, h);

}

//--- 콜백 함수: 다시 그리기 콜백 함수

셰이더 사용하여 삼각형 그리기

//--- 버텍스 셰이더 객체 만들기

```
void make_vertexShaders ()
{
    GLchar * vertexSource;

    vertexSource = filetobuf ("vertex.glsl"); //--- 버텍스셰이더 읽어오기

    //--- 버텍스 셰이더 읽어 저장하고 컴파일 하기
    vertexShader = glCreateShader (GL_VERTEX_SHADER);
    glShaderSource (vertexShader, 1, &vertexSource, NULL);
    glCompileShader (vertexShader);

    GLint result;
    GLchar errorLog[512];
    glGetShaderiv (vertexShader, GL_COMPILE_STATUS, &result);
    if (!result)
    {
        glGetShaderInfoLog (vertexShader, 512, NULL, errorLog);
        std::cerr << "ERROR: vertex shader error\n" << errorLog << std::endl;
        return;
    }
}
```

//--- 프래그먼트 셰이더 객체 만들기

```
void make_fragmentShaders ()
{
    GLchar *fragmentSource;

    fragmentSource = filetobuf ("fragment.glsl"); //--- 프래그먼트셰이더 읽어오기

    //--- 프래그먼트 셰이더 읽어 저장하고 컴파일하기
    fragmentShader = glCreateShader (GL_FRAGMENT_SHADER);
    glShaderSource (fragmentShader, 1, &fragmentSource, NULL);
    glCompileShader (fragmentShader);

    glGetShaderiv (fragmentShader, GL_COMPILE_STATUS, &result);
    if (!result)
    {
        glGetShaderInfoLog (fragmentShader, 512, NULL, errorLog);
        std::cerr << "ERROR: fragment shader error\n" << errorLog << std::endl;
        return;
    }
}
```

셰이더 사용하여 삼각형 그리기

//--- 셰이더 프로그램 만들고 셰이더 객체 링크하기

```
GLuint make_shaderProgram
```

```
{
```

```
    GLuint shaderID = glCreateProgram();
```

//--- 셰이더 프로그램 만들기

```
    glAttachShader (shaderID, vertexShader);
```

//--- 셰이더 프로그램에 버텍스 셰이더 붙이기

```
    glAttachShader (shaderID, fragmentShader);
```

//--- 셰이더 프로그램에 프래그먼트 셰이더 붙이기

```
    glLinkProgram (shaderID);
```

//--- 셰이더 프로그램 링크하기

```
    glDeleteShader (vertexShader);
```

//--- 셰이더 프로그램에 링크하여 셰이더 객체 자체는 삭제 가능

```
    glDeleteShader (fragmentShader);
```

```
    glGetProgramiv (shaderID, GL_LINK_STATUS, &result);
```

// ---셰이더가 잘 연결되었는지 체크하기

```
    if (!result) {
```

```
        glGetProgramInfoLog (shaderID, 512, NULL, errorLog);
```

```
        cerr << "ERROR: shader program 연결 실패\n" << errorLog << endl;
```

```
        return false;
```

```
    }
```

```
    return shaderID;
```

```
}
```

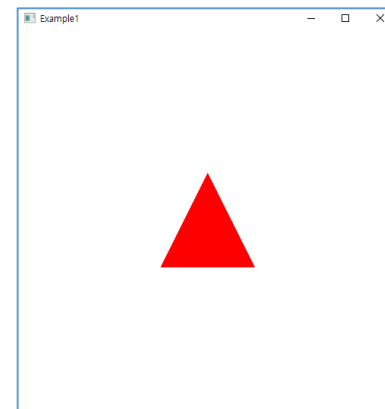

셰이더 사용하여 삼각형 그리기

- 버텍스 셰이더: *vertex.glsl* 파일로 저장

```
#version 330 core
```

```
void main()
{
    const vec4 vertex[3] = vec4[3] (vec4(-0.25, -0.25, 0.5, 1.0),
                                     vec4(0.25, -0.25, 0.5, 1.0),
                                     vec4(0.0, 0.25, 0.5, 1.0));

    gl_Position = vertex [gl_VertexID];
}
```



- **gl_VertexID**: 내장 변수로 해당 시점에 처리될 버텍스의 인덱스
 - gl_VertexID 입력값은 glDrawArrays()에 입력으로 들어간 첫 번째 인자값의 프리미티브 형태에 따라 두번째 인자값부터 시작해서 세 번째 인자인 count만큼의 버텍스까지 한 번에 하나씩 증가한다.
 - gl_VertexID의 값에 기반하여 각 버텍스에 다른 위치를 할당할 수 있다.
- **gl_Position**: 내장 변수로 해당 시점에 그릴 버텍스의 출력위치
vec4 (0.0, 0.0, 0.5, 1.0)을 할당하면 화면의 중앙에 위치한다.

셰이더 사용하여 삼각형 그리기

- 프래그먼트 셰이더: *fragment.glsl* 파일로 저장

```
#version 330 core
```

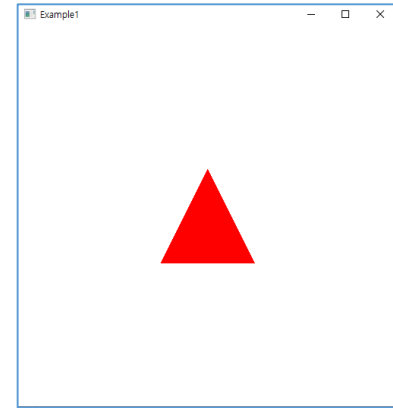
```
out vec4 color;
```

```
void main ()
```

```
{
```

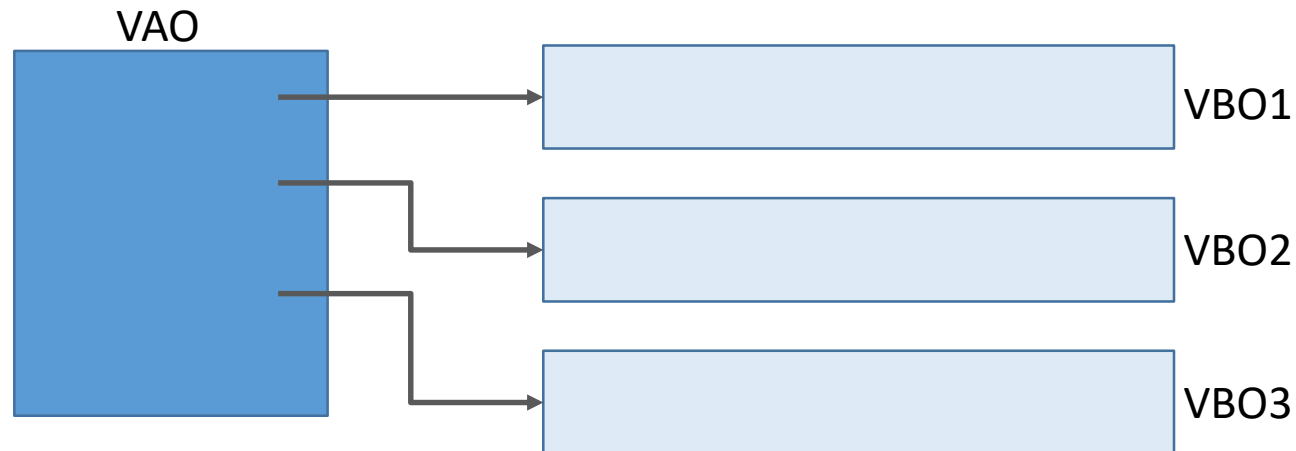
```
    color = vec4 (1.0, 0.0, 0.0, 1.0);
```

```
}
```



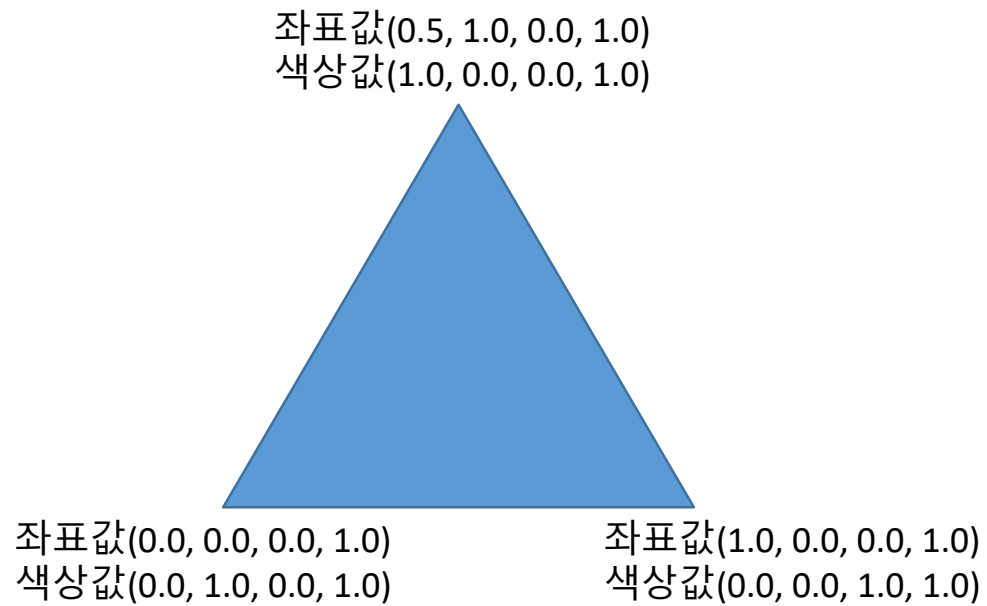
3. 세번째 셰이더 프로그램: 셰이더 사용하여 삼각형 그리기 (2)

- 응용 프로그램에서 버텍스 데이터를 셰이더로 보내서 삼각형 그리기
 - 위치, 색상 등의 속성을 셰이더에서 고정하지 않고, 버퍼에 저장하여 셰이더로 보내기
- **Vertex Buffer Objects (VBO)**
 - 버텍스 데이터를 GPU 메모리에 복사하기 위해 사용되는 배열
 - GL buffer object는 이 버텍스 배열을 저장, 초기화, 렌더링을 할 수 있게 한다.
- **Vertex Array Object (VAO)**
 - 하나의 오브젝트를 구성하는 속성들을 개별 VBO에 저장하고 하나의 VAO로 묶어서 사용한다.
 - VAO에는 버텍스 데이터가 직접 저장되는 것이 아니라 연결 정보만 저장
 - VAO는 VBO에 저장된 데이터 타입과 어떤 속성 변수가 데이터를 가져가게 되는지 저장



Vertex Specifications

- 버텍스의 속성들
 - 위치 (position)
 - 색상 (color)
 - 텍스처 좌표 (texture coordinates)
 - 그 외 데이터 들



```
//--- vertex attribute: position
```

```
const float vertexPosition [] =  
{  
    0.5, 1.0, 0.0,  
    0.0, 0.0, 0.0,  
    1.0, 0.0, 0.0  
};
```

```
//--- vertex attribute: color
```

```
const float vertexColor [] =  
{  
    1.0, 0.0, 0.0,  
    0.0, 1.0, 0.0,  
    0.0, 0.0, 1.0  
};
```

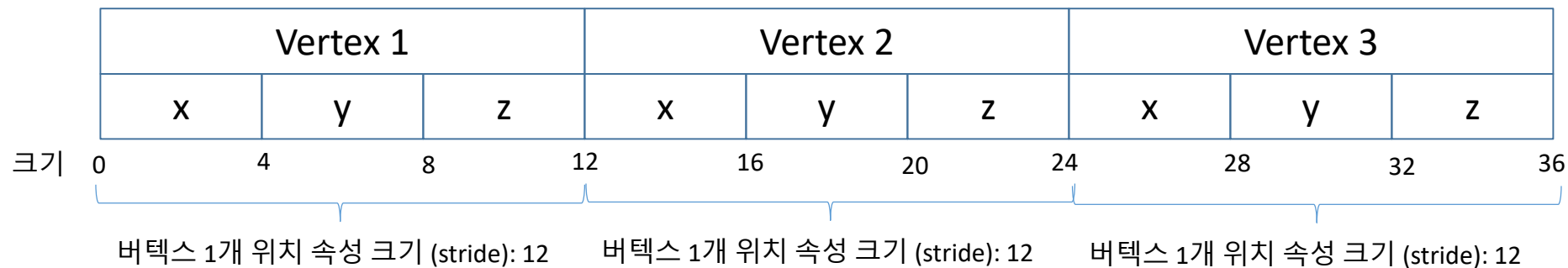
```
// 두 속성을 한 개의 배열에 저장 -----
```

```
//--- vertex attributes: position color
```

```
const float vertexData [] =  
{  
    0.5, 1.0, 0.0,    1.0, 0.0, 0.0,  
    0.0, 0.0, 0.0,    0.0, 1.0, 0.0,  
    1.0, 0.0, 0.0,    0.0, 0.0, 1.0  
};
```

Vertex Attribute

- 버텍스 속성 연결
 - 입력 데이터의 어느 부분이 vertex shader의 어떠한 정점 속성과 맞는지 직접 지정해야 함
- 예) 삼각형의 위치 값
 - 한 개의 버텍스는 3개의 실수 값 (x, y, z 좌표값)으로 구성
 - 각 실수값은 4바이트로 구성
 - 즉, 한 개의 버텍스 속성 값(위치)은 12 바이트 (3개 실수 * 4바이트)로 구성



- 위의 속성 값을 설정해야 함

Vertex Buffer Object

- **Vertex Buffer Object (VBO)**

- 버텍스 데이터를 저장하기 위한 메모리 버퍼
 - 버텍스의 다양한 속성들을 저장한다.
 - Position, Normal, Vector, Color 등
 - VBO당 한 개 또는 여러 개의 속성을 저장할 수 있다.

- 대용량 자료를 GPU에 보내줄 수 있음

- 버퍼를 생성하고 바인드하고 실제 데이터를 넣어줌

```
GLuint VBO;
```

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

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

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

//--- 버퍼 id를 생성

//--- 버퍼 객체에 저장할 데이터 타입 지정

//--- 바인드 후에 호출하는 모든 버퍼는 바인딩 된 버퍼를 사용한다.

//--- 사용자가 정의한 데이터를 현재 바인딩된 버퍼에 복사한다.

- 현재 바인딩된 버퍼가 몇 번째 attribute 버퍼인지 어떤 속성을 갖는지 알려주기위해 속성 포인터 설정하고 사용

```
glVertexAttribPointer (0, 3, GL_FLOAT, GL_FALSE, 0, 0);
```

```
glEnableVertexAttribArray (0);
```

Vertex Array Object

- **Vertex Array Object (VAO)**

- 한 개 또는 그 이상의 VBO를 포함하는 오브젝트로 렌더링할 완전한 객체의 정보들을 저장한다.
 - 하나의 오브젝트를 구성하는 위치, 색상같은 vertex 속성들을 개별 Vertex Buffer Object(VBO)에 저장하고 하나의 VAO로 묶는다.
 - 하나의 VAO에 여러 개의 VBO를 가질 수 있다.
- VAO에는 버텍스 데이터가 직접 저장되는게 아니라 연결 정보만 저장
 - VAO는 VBO에 저장된 데이터 타입과 어떤 속성 변수가 데이터를 가져가게 되는지 저장
- 보통 하나의 매쉬마다 하나의 VAO를 사용함

- VAO를 생성하고 바인드 함

```
GLuint VAO;
```

```
glGenVertexArrays (1, & VAO);
```

```
glBindVertexArray (VAO);
```

```
//--- 버텍스 array 생성
```

```
//--- VAO를 가진다는 의미이고 아직 실제 데이터는 없음
```

함수 프로토타입

- Vertex Array Object (VAO), Vertex Buffer Object (VBO) 생성 및 바인딩 함수
 - void **glGenBuffers** (GLsizei n, GLuint *buffers);
 - 버퍼 오브젝트 (VBO) 이름 생성
 - n: 생성할 이름 개수
 - buffers: 버퍼 오브젝트 이름을 가리키는 배열
 - void **glBindBuffer** (GLenum target, GLuint buffer);
 - 버퍼 오브젝트를 바인드 한다.
 - Target: 바인드할 버퍼 타겟 타입
 - GL_ARRAY_BUFFER: 버텍스 속성
 - GL_ELEMENT_ARRAY_BUFFER: 버텍스 배열 인덱스
 - GL_TEXTURE_BUFFER: 텍스처 데이터 버퍼
 - Buffer: 버퍼 오브젝트 이름
 - void **glBufferData** (GLenum target, GLsizeiptr size, const GLvoid *data, GLenum usage);
 - 버퍼 오브젝트의 데이터를 생성
 - target: 바인드할 버퍼 타겟 타입
 - GL_ARRAY_BUFFER, GL_ELEMENT_ARRAY_BUFFER, GL_TEXTURE_BUFFER...
 - size: 버퍼 오브젝트의 크기
 - data: 저장할 데이터를 가리키는 포인터
 - usage: 저장한 데이터를 사용할 패턴
 - GL_STATIC_DRAW: 한번 버텍스 데이터 업데이트 후 변경이 없는 경우 사용
 - GL_STREAM_DRAW: 데이터가 그려질 때마다 변경
 - GL_DYNAMIC_DRAW: 버텍스 데이터가 자주 바뀌는 경우 (애니메이션), 버텍스 데이터가 바뀔 때마다 다시 업로드 된다

함수 프로토타입

- void **glGenVertexArrays** (GLsizei n, GLuint *arrays);
 - 버텍스 배열 오브젝트 (VAO) 이름 생성
 - n: 생성할 VAO 개수
 - arrays: VAO 저장할 배열 이름
- void **glBindVertexArray** (GLuint array);
 - VAO를 바인드한다.
 - array: 바인드할 버텍스 배열의 이름

함수 프로토타입

- 정점 속성 설정 함수

- void **glVertexAttribPointer** (GLuint index, GLint size, GLenum type, GLboolean normalized, GLsizei stride, const GLvoid *pointer)
 - 버텍스 속성 데이터의 배열을 정의
 - index: 설정할 vertex 속성의 인덱스 값을 지정. (셰이더에서 layout (location = 0) → 속성의 위치가 0번째)
 - size: 버텍스 속성의 크기 (버텍스 속성이 vec3라면 3)
 - type: 데이터 타입 (vec3 라면 GL_FLOAT)
 - normalized: 데이터를 정규화할지 (GL_TRUE: [0, 1] 사이의 값으로 정규화, GL_FALSE: 그대로 사용)
 - stride: 연이은 vertex 속성 세트들 사이의 공백 (값이 공백없이 채워져 있다면 0, 1개 이상의 속성들이 저장되어 있다면 크기를 설정. 예) 버텍스 vec3라면 다음 버텍스 위치는 12바이트)
 - pointer: 데이터가 시작하는 위치의 오프셋 값
 - 각 vertex 속성은 VBO에 의해 관리되는 메모리로부터 데이터를 받는다.
 - 데이터를 받을 VBO (하나가 여러 VBO를 가질 수도 있음)는 glVertexAttribPointer 함수를 호출할 때 GL_ARRAY_BUFFER에 현재 바인딩된 VBO로 결정
 - 사용 예) glVertexAttribPointer (0, 3, GL_FLOAT, GL_FALSE, 3 * sizeof(float), (void*)0);
- void **glEnableVertexAttribArray** (GLuint index);
 - 버텍스 속성 배열을 사용하도록 한다.
 - index: 버텍스 속성 인덱스
 - 사용 예) glEnableVertexAttribArray (0);

1) Vertex Attribute 1개 인 경우

- 한 개의 VBO와 한 개의 VAO 사용하기 예)

- Vertex shader

```
#version 330 core
```

```
layout (location = 0) in vec3 vPos;           //--- 1개의 속성 - attribute로 설정된 위치 속성: 인덱스 0
```

```
void main()
```

```
{
```

```
    gl_Position = vec4 (vPos.x, vPos.y, vPos.z, 1.0);
```

```
}
```

- Fragment shader

```
#version 330 core
```

```
out vec4 FragColor;                          //--- 출력할 객체의 색상
```

```
void main()
```

```
{
```

```
    FragColor = vec4 (1.0f, 0.5f, 0.3f, 1.0f);
```

```
}
```

Vertex Attribute 1개 인 경우

- 버텍스 위치 속성 설정하기 응용 프로그램 예)

//--- 변수 선언

GLuint VAO, VBO_position;

GLvoid InitBuffer ()

{

//--- VAO와 VBO 객체 생성

glGenVertexArrays (1, &VAO);

glGenBuffers (1, &VBO_position);

//--- 사용할 VAO 바인딩

glBindVertexArray (VAO);

//--- vertex positions 저장을 위한 VBO 바인딩.

glBindBuffer (GL_ARRAY_BUFFER, VBO_position);

//--- vertex positions 데이터 입력.

glBufferData (GL_ARRAY_BUFFER, sizeof (vertexPosition), vertexPosition, GL_STATIC_DRAW);

//--- 현재 바인딩되어있는 VBO를 0번째 attribute에 가져오도록 지정하고 그 인덱스의 attribute를 활성화

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

glEnableVertexAttribArray (0);

}

```
const float vertexPosition [] =  
{  
    0.5, 1.0, 0.0,  
    0.0, 0.0, 0.0,  
    1.0, 0.0, 0.0  
};
```

```
const float vertexColor [] =  
{  
    1.0, 0.0, 0.0,  
    0.0, 1.0, 0.0,  
    0.0, 0.0, 1.0  
};
```

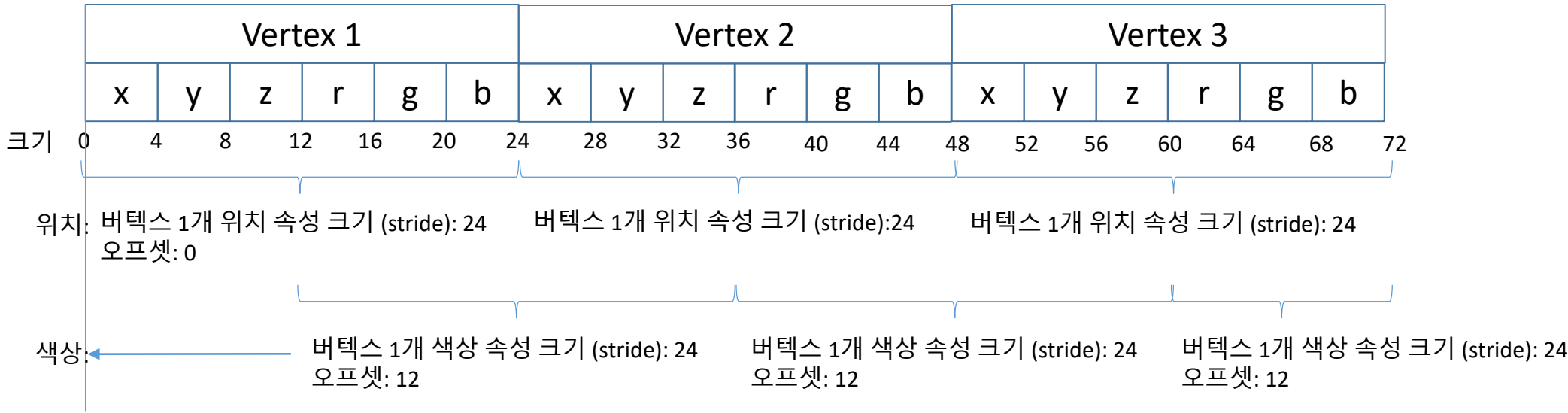
// 위치와 색상을 한 개의 데이터로 저장

```
const float vertexData [] =  
{  
    0.5, 1.0, 0.0,      1.0, 0.0, 0.0,  
    0.0, 0.0, 0.0,      0.0, 1.0, 0.0,  
    1.0, 0.0, 0.0,      0.0, 0.0, 1.0  
};
```

2) Vertex Attribute 2개인 경우

• 버텍스 포맷에 속성 추가

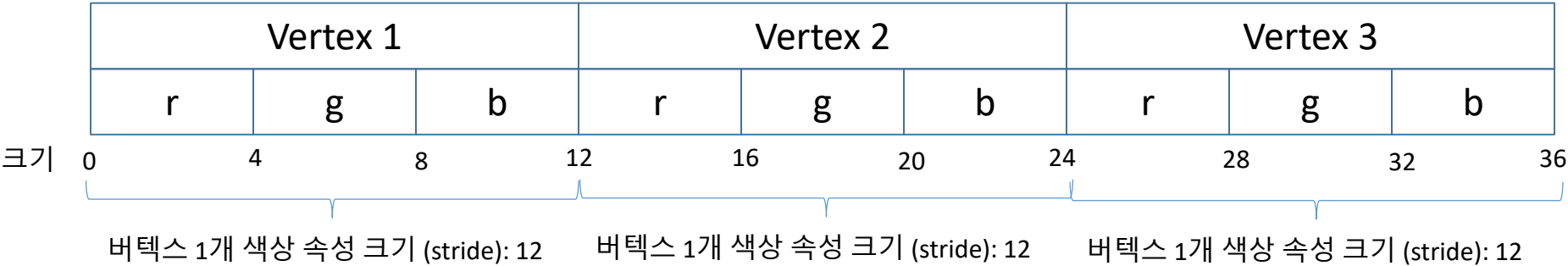
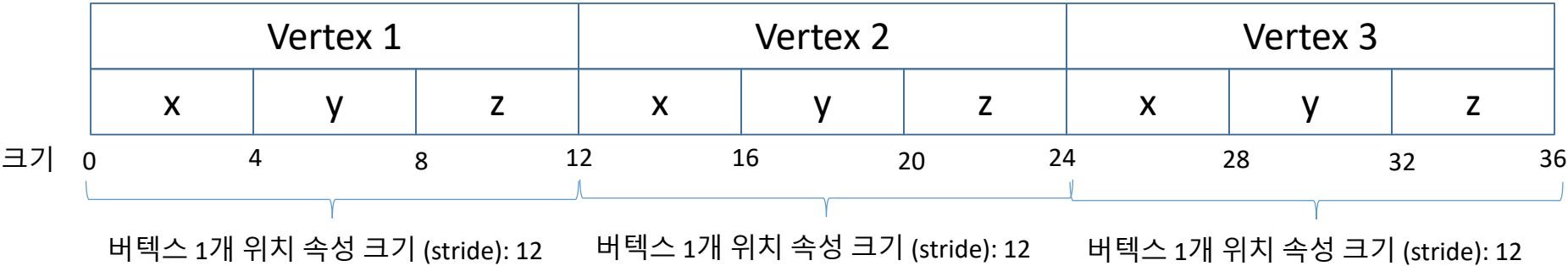
① 좌표값 외에 색상 값을 속성으로 추가하는 경우: 두 개의 속성을 같이 저장하는 경우



2) Vertex Attribute 2개인 경우

- 버텍스 포맷에 속성 추가

② 좌표값 외에 색상값을 속성으로 추가하는 경우: 두 개의 속성을 따로 저장하는 경우



2-1) Vertex Attribute: 2개 속성을 따로 저장할 때 (앞의 2번 경우)

(55페이지)

- VBO를 2개 생성
 - 각 VBO에 속성값 (위치, 색상) 저장
 - 셰이더에 위치와 색상 저장하기
- VAO를 생성
 - VAO에 2개의 속성을 바인드하기

• Vertex Shader

```
#version 330 core
```

```
in vec3 vPos;           //--- 메인 프로그램에서 입력 받음
in vec3 vColor;         //--- 메인 프로그램에서 입력 받음
out vec3 passColor;      //--- fragment shader로 전달
```

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

• Fragment Shader

```
#version 330 core
```

```
in vec3 passColor;      //--- vertex shader에서 입력 받음
out vec4 FragColor;     //--- 프레임 버퍼로 출력
```

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

2-1) Vertex Attribute: 2개 속성을 따로 저장할 때

- 응용 프로그램

//--- 변수 선언

GLuint VAO, VBO_position, VBO_color;

void InitBuffer ()

{

//--- Vertex Array Object 생성

glGenVertexArrays (1, &VAO);

glBindVertexArray (VAO);

//--- 위치 속성

glGenBuffers (1, &VBO_position);

glBindBuffer (GL_ARRAY_BUFFER, VBO_position);

glBufferData (GL_ARRAY_BUFFER, sizeof(vertexPosition), vertexPosition, GL_STATIC_DRAW);

//--- 색상 속성

glGenBuffers (1, &VBO_color);

glBindBuffer (GL_ARRAY_BUFFER, VBO_color);

glBufferData (GL_ARRAY_BUFFER, sizeof(vertexColor), vertexColor, GL_STATIC_DRAW);

//--- vPos 속성 변수에 값을 저장

GLuint pAttribute = glGetAttribLocation (shaderProgramID, "vPos");

glBindBuffer (GL_ARRAY_BUFFER, VBO_position);

glVertexAttribPointer (pAttribute, 3, GL_FLOAT, GL_FALSE, 3 * sizeof(float), 0);

glEnableVertexAttribArray (pAttribute);

//--- vColor 속성 변수에 값을 저장

GLuint cAttribute = glGetAttribLocation (shaderProgramID, "vColor");

glBindBuffer (GL_ARRAY_BUFFER, VBO_color);

glVertexAttribPointer (cAttribute, 3, GL_FLOAT, GL_FALSE, 3 * sizeof(float), 0);

glEnableVertexAttribArray (cAttribute);

}

```
const float vertexPosition [] =
```

```
{  
    0.5, 1.0, 0.0,  
    0.0, 0.0, 0.0,  
    1.0, 0.0, 0.0  
};
```

```
const float vertexColor [] =
```

```
{  
    1.0, 0.0, 0.0,  
    0.0, 1.0, 0.0,  
    0.0, 0.0, 1.0  
};
```

// 위치와 색상을 한 개의 데이터로 저장

```
const float vertexData [] =
```

```
{  
    0.5, 1.0, 0.0,      1.0, 0.0, 0.0,  
    0.0, 0.0, 0.0,      0.0, 1.0, 0.0,  
    1.0, 0.0, 0.0,      0.0, 0.0, 1.0  
};
```


함수 프로토타입

- 위치 가져오기 함수
 - GLint **glGetAttribLocation** (GLuint program, const GLchar *name);
 - Attribute 변수의 위치를 가져온다.
 - 리턴값: 속성 변수의 위치
 - 위치를 찾지 못하면 음수값을 리턴한다.
 - Program: 프로그램 이름
 - Name: 위치를 찾으려는 attribute 변수 이름

2-2) Vertex Attribute: 2개 속성을 한 변수로 저장 (앞의 1번 그림)

- 속성 추가하여 1개 이상의 속성을 사용 하는 경우 (VertexData 변수 사용)

(54페이지)

- Vertex Shader

```
#version 330 core
layout (location = 0) in vec3 vPos;      //--- 위치 변수: attribute position 0
layout (location = 1) in vec3 vColor;    //--- 컬러 변수: attribute position 1

out vec3 outColor;                       //--- 컬러를 fragment shader로 출력

void main()
{
    gl_Position = vec4 (vPos, 1.0);
    outColor = vColor;                   //--- vertex data로부터 가져온 컬러 입력을 ourColor에 설정
}
```

- Fragment Shader

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

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

2-2) Vertex Attribute: 2개 속성을 한 변수로 저장

- 응용 프로그램

//--- 변수 선언

GLuint VAO, VBO;

void InitBuffer ()

{

//--- VAO 객체 생성 및 바인딩

glGenVertexArrays (1, &VAO);

glBindVertexArray (VAO);

//--- vertex data 저장을 위한 VBO 생성 및 바인딩.

glGenBuffers (1, &VBO);

glBindBuffer (GL_ARRAY_BUFFER, VBO);

//--- vertex data 데이터 입력.

glBufferData (GL_ARRAY_BUFFER, sizeof (vertexData), vertexData, GL_STATIC_DRAW);

//--- 위치 속성: 속성 위치 0

glVertexAttribPointer (0, 3, GL_FLOAT, GL_FALSE, 6 * sizeof(float), (void*)0);

glEnableVertexAttribArray (0);

//--- 색상 속성: 속성 위치 1

glVertexAttribPointer (1, 3, GL_FLOAT, GL_FALSE, 6 * sizeof(float), (void*)(3 * sizeof(float)));

glEnableVertexAttribArray (1);

}

```
const float vertexPosition [] =
{
    0.5, 1.0, 0.0,
    0.0, 0.0, 0.0,
    1.0, 0.0, 0.0
};
const float vertexColor [] =
{
    1.0, 0.0, 0.0,
    0.0, 1.0, 0.0,
    0.0, 0.0, 1.0
};
```

// 위치와 색상을 한 개의 데이터로 저장

```
const float vertexData [] =
{
    0.5, 1.0, 0.0,      1.0, 0.0, 0.0,
    0.0, 0.0, 0.0,      0.0, 1.0, 0.0,
    1.0, 0.0, 0.0,      0.0, 0.0, 1.0
};
```

void glVertexAttribPointer (GLuint index, GLint size, GLenum type, GLboolean normalized, GLsizei stride, const GLvoid *pointer)

- Index: 설정할 vertex 속성의 인덱스
- Size: 버텍스 속성의 크기
- Type: 데이터 타입
- Normalized: 데이터를
- Stride: 연이은 vertex 속성 세트들 사이의 공백 (값이 공백없이 채워져 있다면 0, 1개 이상의 속성들이 저장되어 있다면 크기를 설정.
- Pointer: 데이터가 시작하는 위치의 오프셋 값

3) 프래그먼트 셰이더로 직접 데이터 전달하기

- Uniform 한정자를 사용하여 원하는 셰이더로 직접 데이터 전달 가능
 - Uniform: CPU의 응용 프로그램에서 GPU의 셰이더로 데이터를 전달하는 방법
 - Uniform은 global 변수
 - 변수 앞에 uniform 키워드를 붙여 변수 선언
- 유니폼 변수 다루기
 - GLint **glGetUniformLocation** (GLuint program, const GLchar *name);
 - 프로그램에서 uniform 변수의 위치를 가져온다.
 - program: 셰이더 프로그램 이름
 - name: uniform 변수 이름
 - 리턴값: uniform 변수 위치 (-1: 위치를 찾지 못함)
 - void **glUniform{1|2|3|4}{f|i|ui}** (GLuint location, {GLfloat v0, GLfloat v1, GLfloat v2, GLfloat v3});
 - glUniform1f, glUniform2f, glUniform3f, glUniform4f...
 - 현재 프로그램에서 uniform 변수의 값을 명시
 - location: 수정할 uniform 변수의 위치
 - v0, v1, v2, v3: 사용될 uniform 변수 값

프래그먼트 셰이더로 직접 데이터 전달하기

- 응용 프로그램

//--- 그리기 출력 콜백 함수에서 프래그먼트 셰이더로 컬러 값을 보내기

```
void drawScene ()
```

```
{
```

```
    ...
```

```
    //--- uniform 변수의 인덱스 값
```

```
    int vColorLocation = glGetUniformLocation (shaderProgramID, "vColor");
```

```
    //--- uniform 변수가 있는 프로그램 활성화:
```

```
    glUseProgram (shaderProgramID);
```

```
    //--- uniform 변수의 위치에 변수의 값 설정
```

```
    glUniform4f (vColorLocation, 1.0f, 0.0f, 0.0f, 1.0f);
```

```
    //--- 필요한 드로잉 함수 호출
```

```
    glBindVertexArray(VAO);
```

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

```
    glutSwapBuffers();
```

```
}
```

프래그먼트 셰이더로 직접 데이터 전달하기

- 프래그먼트 셰이더에 색상 직접 전달하기 예)

- Vertex shader

```
#version 330 core
```

```
layout (location = 0) in vec3 vPos;  //--- attribute로 설정된 위치 속성: 인덱스 0
```

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

- Fragment shader

```
#version 330 core
```

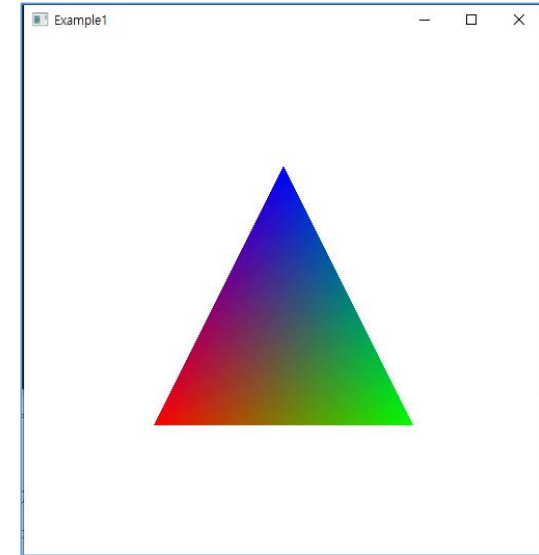
```
uniform vec4 vColor;  //--- 응용 프로그램에서 변수 값 설정
```

```
out vec4 FragColor;  //--- 출력할 객체의 색상
```

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

<최종 예: 위치, 색상 속성을 사용하여 화면 중앙에 삼각형 그리기>

- Main (): 관련 콜백 함수 지정
 - 윈도우 띄우기
 - 필요한 콜백 함수 지정
 - 이벤트 루프 시작
- InitBuffer ():
 - VAO, VBO 만들기
 - 속성 (attributes) 설정하기
- InitShader ():
 - 셰이더 객체 만들기, 컴파일 하기, 셰이더 프로그램 만들고 링크하기
- 콜백 함수들
 - 각종 이벤트 처리하기
 - 출력 함수: display 콜백 함수
 - 화면에 출력하기 (glDrawArrays 또는 glDrawElements)
- 셰이더 만들기
 - Vertex shader, fragment shader 파일 작성



```
const GLfloat triShape[3][3] = {           // 삼각형 꼭지점 좌표값
    {-0.5, -0.5, 0.0},
    { 0.5, -0.5, 0.0 },
    { 0.0,  0.5, 0.0} };

const GLfloat colors[3][3] = {             // 삼각형 꼭지점 색상
    {1.0,  0.0,  0.0},
    {0.0,  1.0,  0.0},
    {0.0,  0.0,  1.0} };
```

최종 예) 위치, 색상 속성을 사용하여 화면 중앙에 삼각형 그리기

//--- vertex shader: vertex.glsl 파일에 저장

```
#version 330 core
```

```
//--- in_Position: attribute index 0
```

```
//--- in_Color: attribute index 1
```

```
layout (location = 0) in vec3 in_Position; //--- 위치 변수: attribute position 0
```

```
layout (location = 1) in vec3 in_Color; //--- 컬러 변수: attribute position 1
```

```
out vec3 out_Color; //--- 프래그먼트 셰이더에게 전달
```

```
void main(void)
```

```
{
```

```
    gl_Position = vec4 (in_Position.x, in_Position.y, in_Position.z, 1.0);
```

```
    out_Color = in_Color;
```

```
}
```

//--- fragment shader: fragment.glsl 파일에 저장

```
#version 330 core
```

```
//--- out_Color: 버텍스 셰이더에서 입력받는 색상 값
```

```
//--- FragColor: 출력할 색상의 값으로 프레임 버퍼로 전달 됨.
```

```
in vec3 out_Color; //--- 버텍스 셰이더에게서 전달 받음
```

```
out vec4 FragColor; //--- 색상 출력
```

```
void main(void)
```

```
{
```

```
    FragColor = vec4 (out_Color, 1.0);
```

```
}
```


최종 예) 위치, 색상 속성을 사용하여 화면 중앙에 삼각형 그리기

//--- 메인 함수

//--- 함수 선언 추가하기

```
const GLfloat triShape[3][3] = {           //--- 삼각형 위치 값
    { -0.5, -0.5, 0.0 }, { 0.5, -0.5, 0.0 }, { 0.0, 0.5, 0.0 } };
const GLfloat colors[3][3] = {             //--- 삼각형 꼭지점 색상
    { 1.0, 0.0, 0.0 }, { 0.0, 1.0, 0.0 }, { 0.0, 0.0, 1.0 } };
GLuint vao, vbo[2];
```

```
void main (int argc, char** argv)           //--- 윈도우 출력하고 콜백함수 설정
{
```

 //--- 윈도우 생성하기

```
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGBA);
    glutInitWindowPosition(100, 100);
    glutInitWindowSize (800, 600);
    glutCreateWindow ("Example1");
```

 //--- GLEW 초기화하기

```
    glewExperimental = GL_TRUE;
    glewInit ();
```

```
    make_shaderProgram();
    InitBuffer();
```

```
    glutDisplayFunc (drawScene);
    glutReshapeFunc (Reshape);
    glutMainLoop();
```

```
}
```

//--- 그리기 콜백 함수

GLvoid **drawScene**()

```
{
```

 //--- 변경된 배경색 설정

```
    glClearColor(rColor, gColor, bColor, 1.0f);
    //glClearColor(1.0, 1.0, 1.0, 1.0f);
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
```

 //--- 렌더링 파이프라인에 셰이더 불러오기

```
    glUseProgram(shaderProgramID);
```

 //--- 사용할 VAO 불러오기

```
    glBindVertexArray(vao);
```

 //--- 삼각형 그리기

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

```
    glutSwapBuffers();       //--- 화면에 출력하기
```

```
}
```

//--- 다시그리기 콜백 함수

GLvoid **Reshape** (int w, int h)

```
{
```

```
    glViewport ( 0, 0, w, h );
```

```
}
```

최종 예) 위치, 색상 속성을 사용하여 화면 중앙에 삼각형 그리기

//--- 버퍼 생성하고 데이터 받아오기

GLuint vao, vbo[2];

void **InitBuffer** ()

```
{
    glGenVertexArrays (1, &vao);           //--- VAO 를 지정하고 할당하기
    glBindVertexArray (vao);               //--- VAO를 바인드하기

    glGenBuffers (2, vbo);                  //--- 2개의 VBO를 지정하고 할당하기

    //--- 1번째 VBO를 활성화하여 바인드하고, 버텍스 속성 (좌표값)을 저장
    glBindBuffer(GL_ARRAY_BUFFER, vbo[0]);

    //--- 변수 diamond 에서 버텍스 데이터 값을 버퍼에 복사한다.
    //--- triShape 배열의 사이즈: 9 * float
    glBufferData(GL_ARRAY_BUFFER, 9 * sizeof(GLfloat), triShape, GL_STATIC_DRAW);

    //--- 좌표값을 attribute 인덱스 0번에 명시한다: 버텍스 당 3 * float
    glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 0, 0);

    //--- attribute 인덱스 0번을 사용가능하게 함
    glEnableVertexAttribArray(0);

    //--- 2번째 VBO를 활성화 하여 바인드 하고, 버텍스 속성 (색상)을 저장
    glBindBuffer(GL_ARRAY_BUFFER, vbo[1]);

    //--- 변수 colors에서 버텍스 색상을 복사한다.
    //--- colors 배열의 사이즈: 9 * float
    glBufferData(GL_ARRAY_BUFFER, 9 * sizeof(GLfloat), colors, GL_STATIC_DRAW);

    //--- 색상값을 attribute 인덱스 1번에 명시한다: 버텍스 당 3*float
    glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, 0, 0);

    //--- attribute 인덱스 1번을 사용 가능하게 함.
    glEnableVertexAttribArray(1);
}
```

//--- 셰이더 프로그램 생성하기

void **make_shaderProgram** ()

```
{
    make_vertexShaders ();                 //--- 버텍스 셰이더 만들기
    make_fragmentShaders ();              //--- 프래그먼트 셰이더 만들기

    //--- shader Program
    shaderProgramID = glCreateProgram();

    glAttachShader (shaderProgramID, vertexShader);
    glAttachShader (shaderProgramID, fragmentShader);
    glLinkProgram (shaderProgramID);

    //--- 셰이더 삭제하기
    glDeleteShader (vertexShader);
    glDeleteShader (fragmentShader);

    //--- Shader Program 사용하기
    glUseProgram (shaderProgramID);
}
```

최종 예) 위치, 색상 속성을 사용하여 화면 중앙에 삼각형 그리기

//--- 버텍스 셰이더 객체 만들기

```
GLchar *vertexSource, *fragmentSource; //--- 소스코드 저장 변수
GLuint vertexShader, fragmentShader; //--- 셰이더 객체
GLuint shaderProgramID; //--- 셰이더 프로그램
```

void **make_vertexShaders** ()

```
{
    vertexSource = filetoBuf ("vertex.glsl");
```

//--- 버텍스 셰이더 객체 만들기

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

//--- 셰이더 코드를 셰이더 객체에 넣기

```
glShaderSource(vertexShader, 1, (const GLchar**)&vertexSource, 0);
```

//--- 버텍스 셰이더 컴파일하기

```
glCompileShader(vertexShader);
```

//--- 컴파일이 제대로 되지 않은 경우: 에러 체크

```
GLint result;
GLchar errorLog[512];
glGetShaderiv (vertexShader, GL_COMPILE_STATUS, &result);
if (!result)
{
    glGetShaderInfoLog (vertexShader, 512, NULL, errorLog);
    cerr << "ERROR: vertex shader 컴파일 실패\n" << errorLog << endl;
    return;
}
}
```

//--- 프래그먼트 셰이더 객체 만들기

void **make_fragmentShaders** ()

```
{
```

```
    fragmentSource = filetoBuf ("fragment.glsl");
```

//--- 프래그먼트 셰이더 객체 만들기

```
fragmentShader = glCreateShader(GL_FRAGMENT_SHADER);
```

//--- 셰이더 코드를 셰이더 객체에 넣기

```
glShaderSource(fragmentShader, 1, (const GLchar**)&fragmentSource, 0);
```

//--- 프래그먼트 셰이더 컴파일

```
glCompileShader(fragmentShader);
```

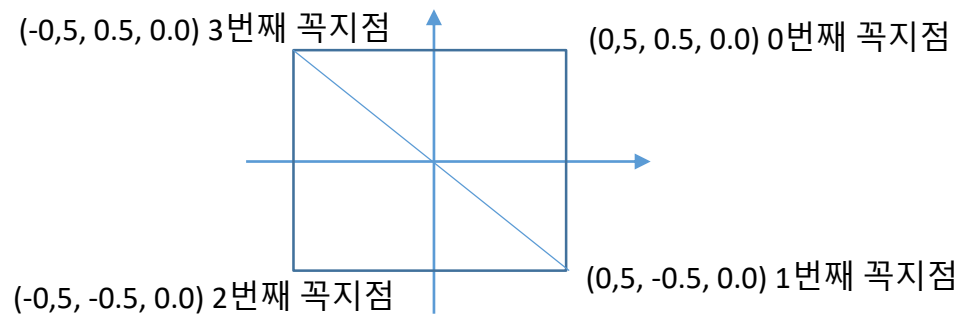
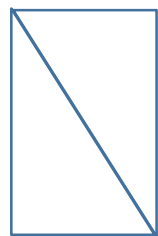
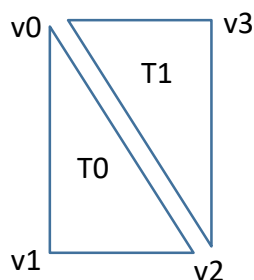
//--- 컴파일이 제대로 되지 않은 경우: 컴파일 에러 체크

```
GLint result;
GLchar errorLog[512];
glGetShaderiv (fragmentShader, GL_COMPILE_STATUS, &result);
if (!result)
{
    glGetShaderInfoLog (fragmentShader, 512, NULL, errorLog);
    cerr << "ERROR: fragment shader 컴파일 실패\n" << errorLog << endl;
    return;
}
}
```

Element Buffer Object

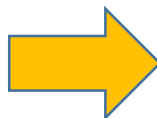
- Element Buffer Object (EBO)

- EBO는 VBO와 같은 버퍼인데, 버텍스 좌표값 대신 인덱스를 저장한다.
- 사각형을 그린다면,
 - 사각형 -> 삼각형 2개 -> 6개의 정점 -> 2개의 정점이 중복, 즉 4개의 정점으로 생성 가능



```
float vPosition[] = {  
    // 첫 번째 삼각형  
    0.5f, 0.5f, 0.0f, // 우측 상단  
    0.5f, -0.5f, 0.0f, // 우측 하단  
    -0.5f, 0.5f, 0.0f, // 좌측 상단  
    // 두 번째 삼각형  
    0.5f, -0.5f, 0.0f, // 우측 하단  
    -0.5f, -0.5f, 0.0f, // 좌측 하단  
    -0.5f, 0.5f, 0.0f // 좌측 상단  
};
```

} 2개가 같은 값



```
float vPositionList[] = {  
    0.5f, 0.5f, 0.0f, // 우측 상단  
    0.5f, -0.5f, 0.0f, // 우측 하단  
    -0.5f, -0.5f, 0.0f, // 좌측 하단  
    -0.5f, 0.5f, 0.0f // 좌측 상단  
};  
unsigned int index[] = {  
    0, 1, 3, // 첫 번째 삼각형  
    1, 2, 3 // 두 번째 삼각형  
};
```

Element Buffer Object

- 정점들을 저장하고, 그 정점을 사용하여 인덱스 리스트를 만든다.

```
void InitBuffer ()
{
    GLuint VAO, VBO_pos, EBO;

    glGenVertexArrays (1, &VAO);
    glGenBuffers (1, &VBO_pos);

    glBindVertexArray (VAO);
    glBindBuffer (GL_ARRAY_BUFFER, VBO_pos);
    glBufferData(GL_ARRAY_BUFFER, sizeof(vPositionList), vPositionList, GL_STATIC_DRAW);

    glGenBuffers (1, &EBO);
    glBindBuffer (GL_ELEMENT_ARRAY_BUFFER, EBO);           //--- GL_ELEMENT_ARRAY_BUFFER 버퍼 유형으로 바인딩
    glBufferData (GL_ELEMENT_ARRAY_BUFFER, sizeof(index), index, GL_STATIC_DRAW);
    glVertexAttribPointer (0, 3, GL_FLOAT, GL_FALSE, 3 * sizeof(float), 0);
    glEnableVertexAttribArray (0);
}

Void drawScene ()
{
    glUseProgram (shaderProgramID);
    glBindVertexArray (VAO);
    glDrawElements (GL_TRIANGLES, 6, GL_UNSIGNED_INT, 0);    //--- 0번부터 6개의 꼭지점을 사용하여 삼각형을 그린다.
}
```

함수 프로토타입

- 함수 프로토타입
 - void **glDrawElements** (GLenum mode, GLsizei count, GLenum type, const GLvoid *indices);
 - mode: GL_POINTS, GL_LINE_STRIP, GL_LINE_LOOP, GL_LINES, GL_TRIANGLE_STRIP, GL_TRIANGLE_FAN, GL_TRIANGLES
 - count: 렌더링할 요소의 개수
 - type: indices 값의 타입 (GL_UNSIGNED_BYTE, GL_UNSIGNED_SHORT, or GL_UNSIGNED_INT 중 1개)
 - indices: 바인딩 되는 버퍼의 데이터 저장소에 있는 배열의 첫 번째 인덱스 오프셋

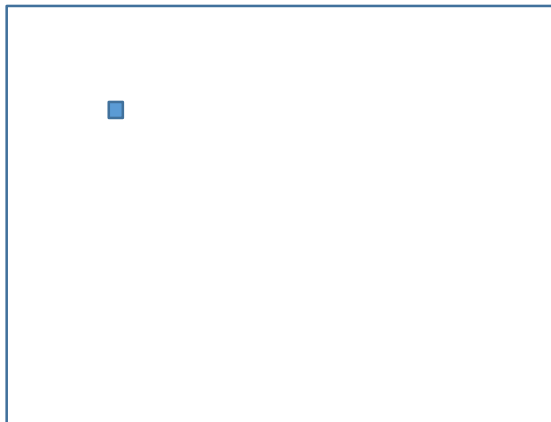
실습 7

- 화면에 기본 도형 그리기

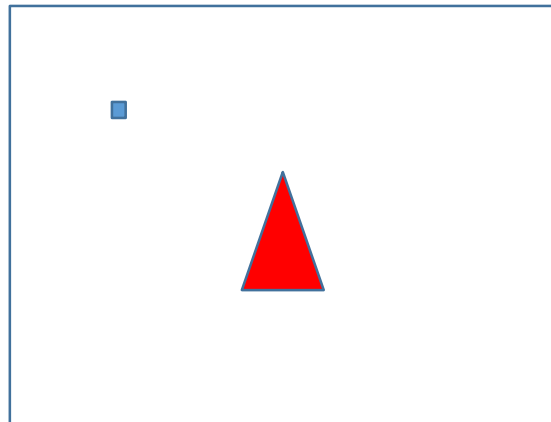
- 키보드 명령에 따라, 마우스를 누르는 위치에 점, 선, 삼각형 또는 사각형 (삼각형 2개 붙이기) 그린다.
- 색상과 크기는 자율적으로 정하고, 최대 10개의 도형을 그린다.

- 키보드 명령

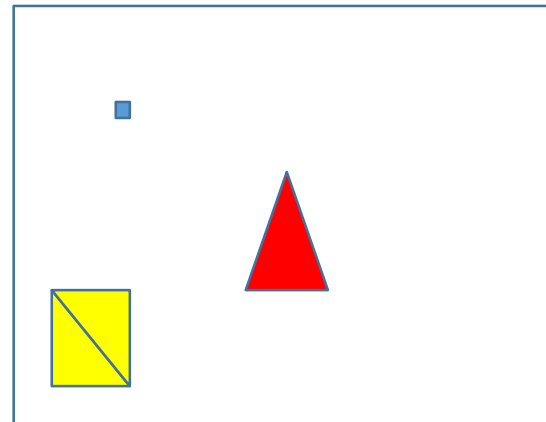
- p: 점 그리기
- l: 선 그리기
- t: 삼각형 그리기
- r: 사각형 그리기
- w/a/s/d: 그린 모든 도형 중 랜덤하게 한 개를 선택한 후 화면에서 위/좌/아래/우측으로 이동한다. 명령어를 선택할 때 마다 다른 도형이 선택되어 이동된다.
- c: 모든 도형을 삭제한다.



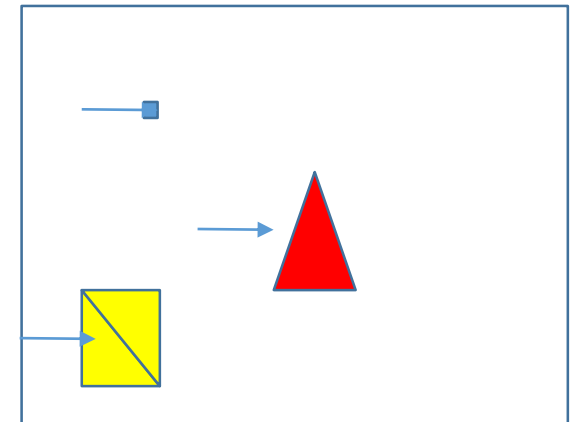
명령어 p



명령어 t



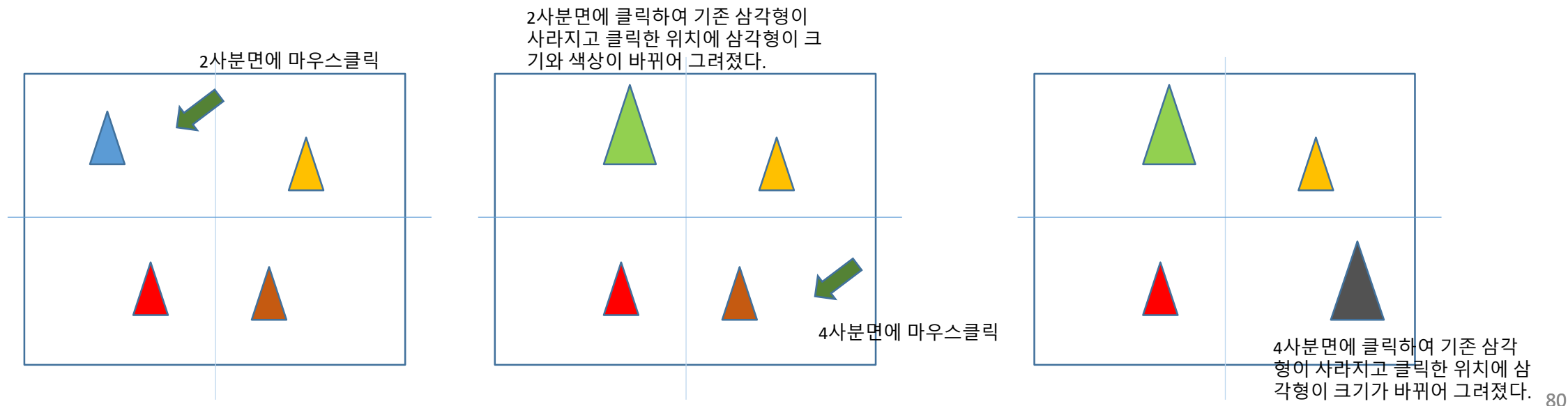
명령어 r



명령어 d (우측으로 이동)

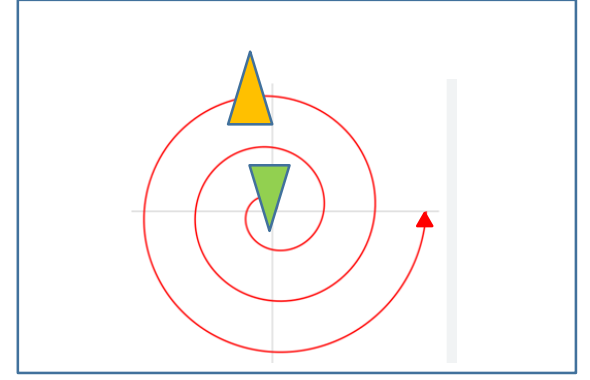
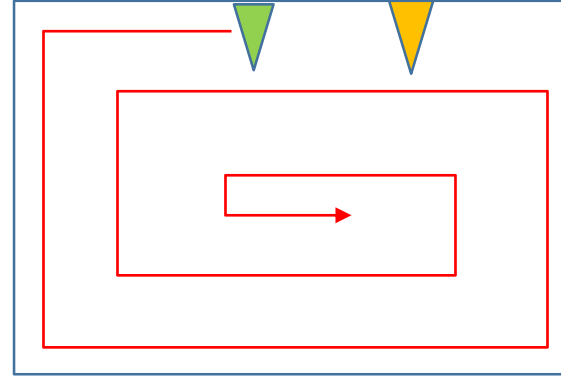
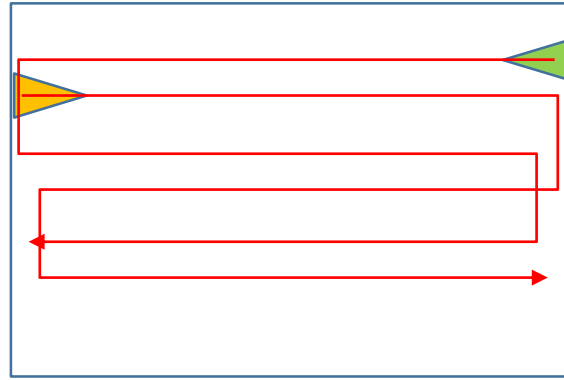
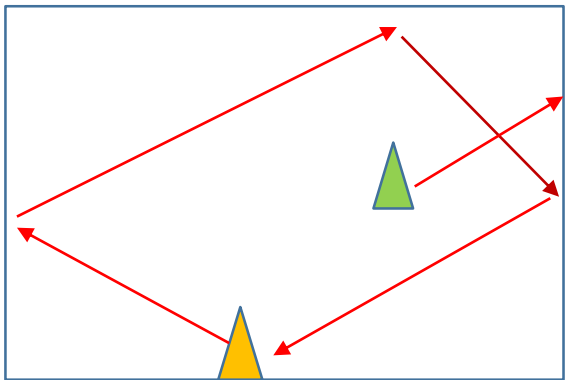
실습 8

- 화면에 삼각형 그리기
 - 화면을 x축, y축에 의해 네 부분으로 나눈 후, 삼각형을 사분면에 각각 1개씩 그리기
 - 삼각형 그리기: 삼각형은 이등변 삼각형이고 각각 다른 색상 설정
 - 왼쪽 마우스를 누르면 그 위치에 새로운 삼각형 을 그린다.
 - 마우스의 위치에 따라 해당 사분면에 있는 사각형이 삭제되고, 마우스 클릭 위치에 새로운 삼각형이 그려진다.
 - 새로운 삼각형을 그릴 때 크기와 색상이 바뀐다.
 - 오른쪽 마우스를 누르면
 - 해당 부분에 새로운 삼각형이 그려진다. 한 부분에는 최대 3개의 삼각형을 그릴 수 있다.
 - 키보드 명령
 - a/b: 도형을 면으로/선으로 그린다



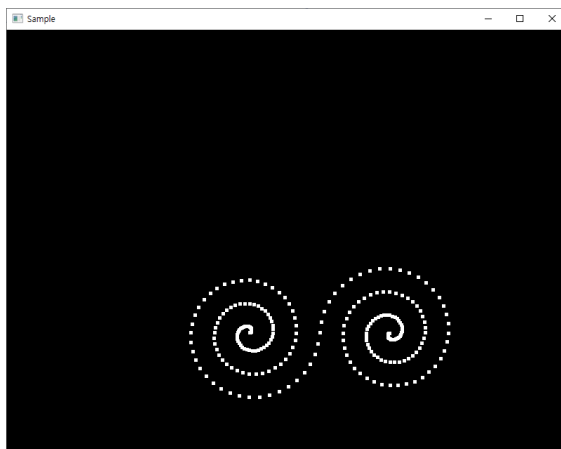
실습 9

- 삼각형 움직이기
 - 실습 8에서 그린 삼각형들이 각각 다른 방향과 다른 속도로 이동한다.
 - 실습 8과 같이 4개의 삼각형을 그릴 수 있다.
 - 삼각형은 이등변 삼각형으로 그린다.
 - 키보드 명령으로 4개의 다른 이동을 한다.
 - 1: 튕기기
 - 2: 좌우로 지그재그
 - 3: 사각 스파이럴
 - 4: 원 스파이럴
 - 가장자리를 만나게 되면 삼각형의 방향이 바뀌게 된다.



실습 10

- 스파이럴 애니메이션 만들기
 - 마우스 클릭 또는 키보드 명령에 의해 스파이럴 모양으로 원이 그려진다.
 - 마우스를 클릭하면 그 위치에 그린다.
 - 1개 이상일 경우에는 랜덤한 위치를 설정하여 그 위치에 그린다.
 - 스파이럴 방향은 시계방향 또는 반시계 방향이 임의로 선택되어 그려진다. 2~3 바퀴 스파이럴을 그린 후 끝나는 지점에서 밖에서 안으로 다시 2~3바퀴 회오리를 그린다.
 - 안에서 밖으로, 밖에서 안으로 들어오는 스파이럴을 연결하여 그린다.
 - 점이 순서대로 그려진다 (애니메이션)
 - 새롭게 그려질 때마다 배경색을 변경한다.
 - **키보드 명령어**
 - p: 점으로 그린다.
 - l: 선으로 그린다.
 - 1~5: 입력된 숫자만큼의 스파이럴이 그려진다.



실습 11

• 여러 도형 그리기

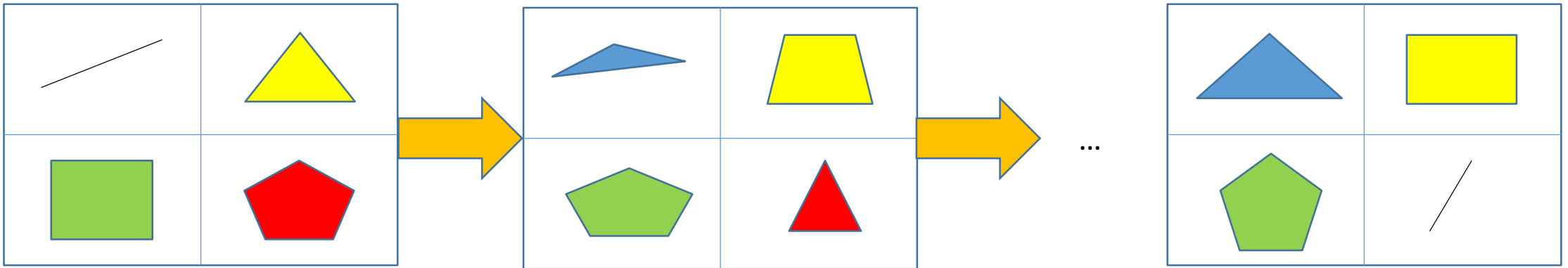
- 화면을 x축, y축에 의해 이등분한 사분면에 각각 선, 삼각형, 사각형(2개의 삼각형 사용), 오각형(2개 이상의 삼각형 사용)을 그린다.
- 네 개의 도형을 단계별로 다른 도형으로 변환시킨다. (애니메이션 적용)

1) 선 → 삼각형

2) 삼각형 → 사각형

3) 사각형 → 오각형

4) 오각형 → 선



- 키보드 명령에 의해 화면에 한 개의 도형 만을 그리고 그 도형의 변화 애니메이션을 그린다.

• l: 선 → 삼각형

t: 삼각형 → 사각형

r: 사각형 → 오각형

p: 오각형 → 선

- a: 네 개의 도형이 다시 모두 그려진다.

이번 주에는

- 셰이더를 위한 GLSL기초
 - 버텍스 셰이더 만들기
 - 프래그먼트 셰이더 만들기
 - 셰이더 프로그램 만들기
 - 버퍼 다루기
- 다음에는
 - 이론: 그래픽스 개요 및 2차원 기본 요소 1
 - 실습: 변환