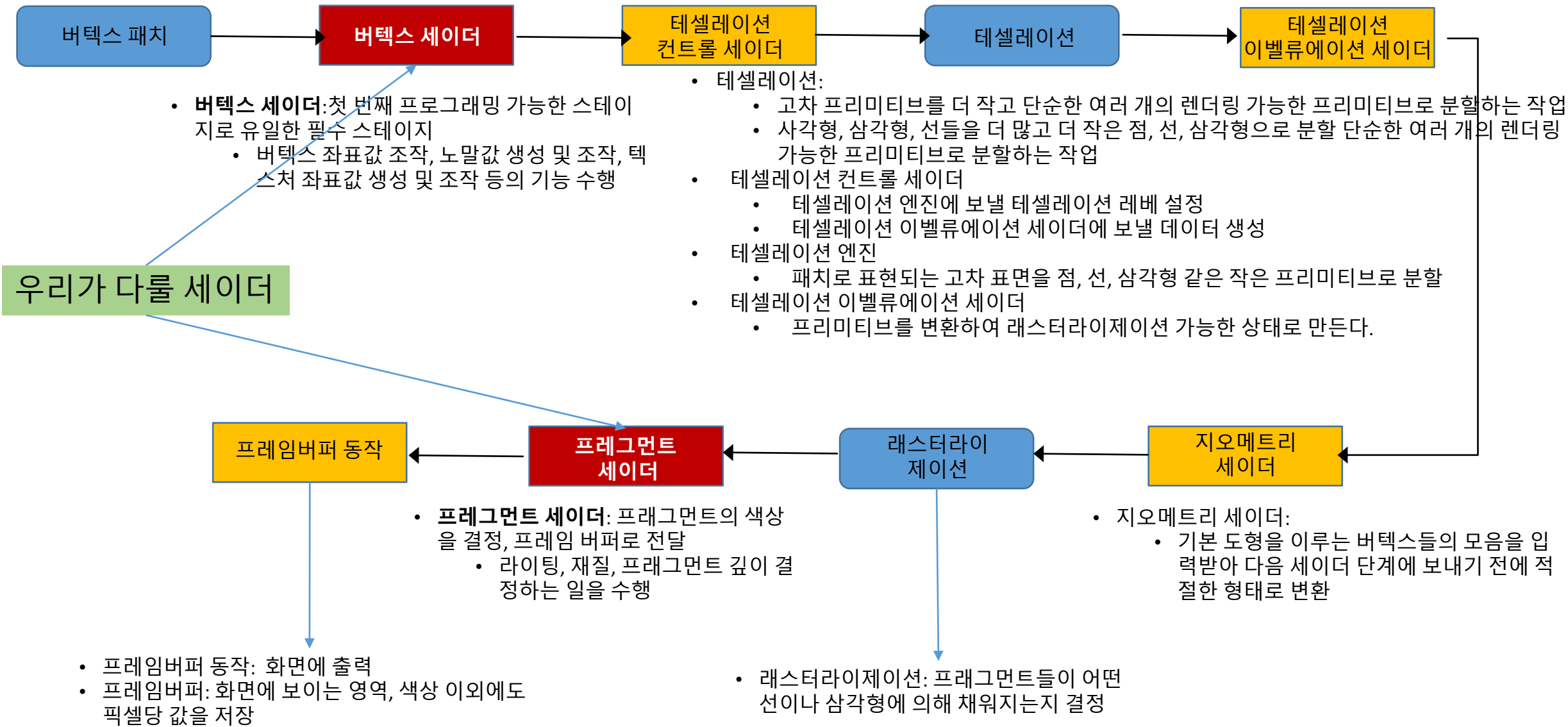


OpenGL 셰이더 만들기

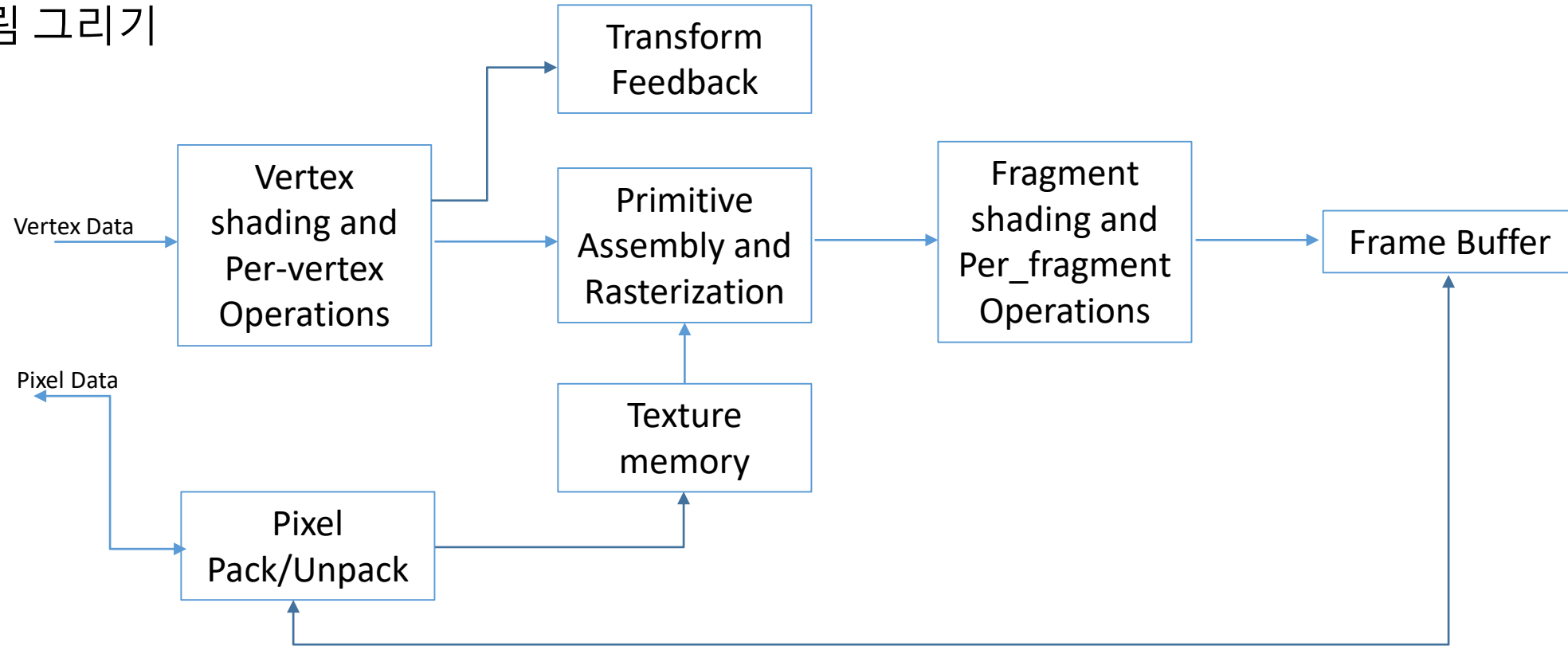
2019-2 컴퓨터 그래픽스
그래픽스 파이프라인
GLSL 사용하기

OpenGL 그래픽스 파이프라인



OpenGL 프로그램 작성하기

- 그림 그리기



- 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.

셰이더 기본 형태

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

```
#version 330 core
```

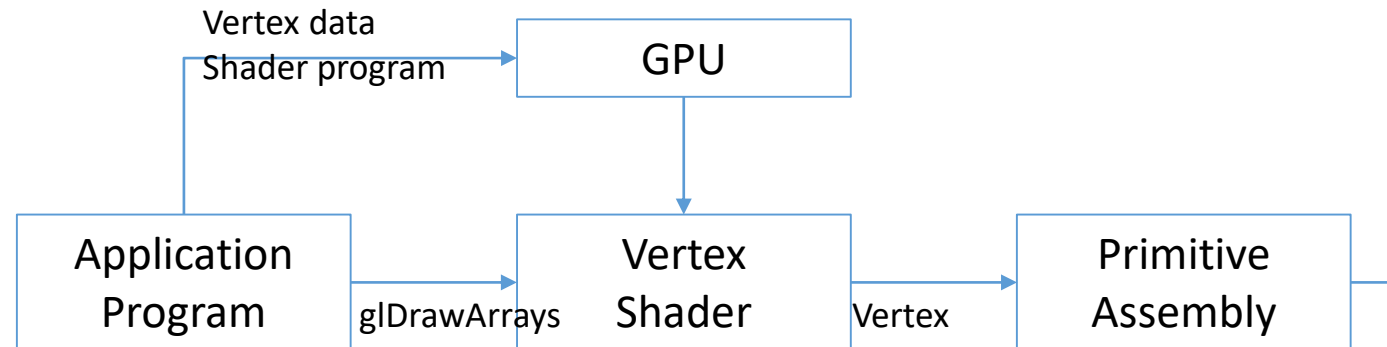
```
in vec3 vPos;
```

```
void main ()
```

```
{
```

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

```
}
```



셰이더 기본 형태

- 프래그먼트 셰이더
 - 래스터라이제이션에서 생성된 프래그먼트를 일련의 색상과 깊이 값으로 처리
 - 색상 설정
 - 조명 설정

```
#version 330 core
```

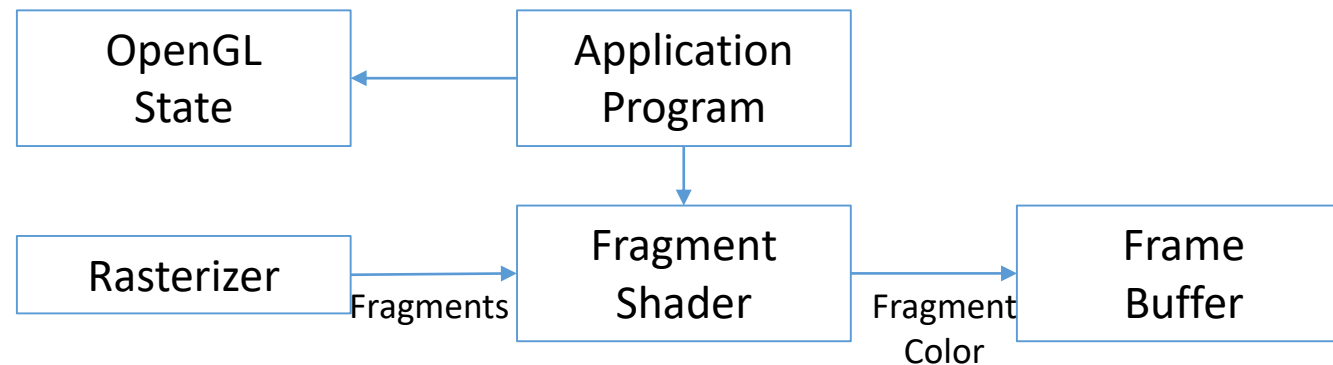
```
out vec4 FragColor;
```

```
void main ()
```

```
{
```

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

```
}
```



GLSL 기본 문법

- 문법의 특징

- C와 비슷한 기본 문법으로 셰이더를 작성할 때 사용한다.
- GLSL에는 포인터 개념이 없음
 - 동적 할당이 없음
- C언어의 구조체 사용 가능
- Matrices나 Vectors는 기본 형으로 함수에서 파라미터 입력이나 반환형 출력으로 사용 가능

- 프로그램 구조

```
#version version_number
```

```
//--- 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;
```

```
}
```

GLSL 기본 문법

- 데이터 타입
 - 기본 C언어 타입
 - **int, float, double, uint, bool**
 - 벡터 타입 ($2 \leq n \leq 4$)
 - **vec n / dvec n / bvec n / ivec n / uvec n**
 - 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
 - 매트릭스 타입 ($2 \leq n \leq 4, 2 \leq m \leq 4$)
 - **mat n / mat $m \times n$ / dmat n / dmat $m \times n$**
 - 열 우선으로 구성
 - 텍스처 샘플러
 - 텍스처 값 (texel)에 접근이 가능한 샘플러 타입
 - **sampler1D, sampler2D, sampler3D**

GLSL 기본 문법

- 한정자 (qualifier)
 - 한정자는 변수나 함수의 형식 인자의 앞에 붙여 해당 변수나 인자의 특성을 결정짓는다.
 - Storage 한정자
 - const, in, out (varying), uniform
 - Layout 한정자
 - layout
 - 버텍스 셰이더: layout(location = attribute index) in vec3 position;
 - 프래그먼트 셰이더: layout(location = output index) out vec4 outColor;
 - 셰이더로 정보를 주고 받을 때는 변수를 사용:
 - Built-in 변수
 - 사용자 정의 변수
 - 선언할 때 가장 앞에 한정자를 붙여 변수의 종류를 결정한다.
- 한정자 (qualifier)
 - const
 - 상수
 - 셰이더에 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)

- 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
 - location 크기: 일반적으로 1 location 사용, double, dvec2 1 location 사용, dvec3, dvec4 2 location 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
 - 내장형 vertex attribute: gl_Position
 - 사용자 정의 vertex attribute: in vec3 velocity

GLSL 기본 문법

- 한정자 (qualifier)

- in/out (varying)

- vertex shader에서 fragment shader로 전달되는 변수
 - vertex shader에서는 쓰기 허용, fragment shader에서는 읽기 전용
 - 3.0 이상의 버전에서는 varying 대신 vertex shader에서는 out으로, fragment shader에서는 in으로 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`

```
const vec4 red = vec4 (1.0, 0.0, 0.0, 0.0);  
out vec3 color;           // 프래그먼트 셰이더로 보내기  
void main ()  
{  
    color = red;  
}
```

- `//--- fragment shader`

```
in vec3 color;             // 버텍스 셰이더에서 받기  
out vec4 outColor;  
void main ()  
{  
    outColor = vec4 (color, 1.0);  
}
```

GLSL 기본 문법

- 한정자 (qualifier)
 - uniform: CPU 위의 응용 프로그램에서 GPU 위의 셰이더로 데이터를 전달하는 한 방법
 - 모든 단계의 모든 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 방법:
 - 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 index를 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
 - vo, 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

- 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)
 - `mat2 m = mat2 (1.0, 2.0, 3.0, 4.0);`
 - `mat2 m = mat2 (1.0);`

GLSL 기본 문법

- 연산자

- 대부분의 연산자는 C언어의 우선순위 규칙과 동일
- 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
 - `vec3 a = vec3 (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

- GLSL에 이미 지정되어 있는 built-in 변수들
 - 버텍스 셰이더:
 - `int gl_VertexID`: 현재 프로세스되고 있는 버텍스의 인덱스값
 - `vec4 gl_Position`: 현재 버텍스의 위치
 - `float gl_PointSize`: 점의 크기값
 - 프래그먼트 셰이더
 - `vec4 gl_FragCoord`: 윈도우 공간에서 프래그먼트의 좌표값
 - `vec2 gl_PointCoord`: 화면의 좌표값
 - `float gl_FragDepth`: 프래그먼트 깊이 값

GLSL 기본 문법

- GLSL에 저장되어 있는 built-in 함수들
 - 삼각함수
 - sin, cos, tan
 - 역삼각함수
 - 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
- 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
 - 입력과 출력
 - in: 함수 내부에서 읽기만 가능, 기본 한정자
 - out: 함수 시작 시점에는 정의되지 않은 값을 가지며, 함수 종료시점에 가진 값을 호출자에게 전달함

셰이더 프로그램 만들기

- OpenGL 셰이딩 언어 GLSL을 사용하여 셰이더 작성
 - GLSL의 컴파일러는 OpenGL에 내장되어 있음
 - 셰이더 프로그램을 만들려면
 - 셰이더 코드 작성 ➔ 셰이더 객체로 바뀌어 컴파일 ➔ 여러 셰이더 객체들이 하나의 프로그램 객체로 링크
- 셰이더 코드 작성 ➔ 버텍스 셰이더, 프래그먼트 셰이더
- 셰이더 객체 만들기 ➔ glCreateShader
- 셰이더 객체에 셰이더 코드 붙이기 ➔ glShaderSource
- 셰이더 객체 컴파일하기 ➔ glCompileShader
- 셰이더 프로그램 만들기 ➔ glCreateProgram
- 셰이더 프로그램에 셰이더 객체들을 붙이기 ➔ glAttachShader
- 셰이더 프로그램 링크 ➔ glLinkProgram
- 셰이더 프로그램 사용하기 ➔ glUseProgram

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
 - 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: string과 length 배열의 개수
 - String: 소스코드가 저장되어 객체 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
 - 프로그램 활성화
 - void `glUseProgram` (GLuint program);
 - 현재 렌더링 상태에 프로그램 객체를 활성화한다.
 - Program: 실행 할 프로그램

셰이더 코드

- 셰이더 코드 작성하기

- 버텍스 셰이더

```
#version 330 core
void main ()
{
    gl_Position = vec4 (0.0, 0.0, 0.5, 1.0);
}
```

- 프래그먼트 셰이더

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

셰이더 객체만들기

- Vertex shader

```
GLuint compleie_shaders ()
```

```
{  
    const GLchar* vertexShaderSource =  
        "#version 330 core\n"  
        "void main()  
        "{  
        "gl_Position = vec4(0.0, 0.0, 0.5, 1.0);"  
        "}";
```

//--- 버텍스 셰이더 읽어 저장하고 컴파일 하기

```
GLuint vertexShader = glCreateShader (GL_VERTEX_SHADER);  
glShaderSource (vertexShader, 1, &vertexShaderSource, NULL);  
glCompileShader (vertexShader);  
  
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 false;  
}
```

- Fragment shader

```
const GLchar* fragmentShaderSource =
```

```
    "#version 330 core\n"  
    "out vec4 fragmentColor;"  
    "void main()"  
    "{  
    "fragmentColor = vec4(1.0, 0.0, 0.0, 1.0); //Red color"  
    "}";
```

//--- 프래그먼트 셰이더 읽어 저장하고 컴파일하기

```
GLuint fragmentShader = glCreateShader (GL_FRAGMENT_SHADER);  
glShaderSource (fragmentShader, 1, &fragmentShaderSource, NULL);  
glCompileShader (fragmentShader);  
  
glGetShaderiv (fragmentShader, GL_COMPILE_STATUS, &result);  
if (!result)  
{  
    glGetShaderInfoLog (fragmentShader, 512, NULL, errorLog);  
    cerr << "ERROR: fragment shader 컴파일 실패\n" << errorLog << endl;  
    return false;  
}
```

셰이더 프로그램 만들기

- 셰이더 프로그램 만들기

```
GLuint ShaderProgramID = glCreateProgram();
```

```
//--- 셰이더 프로그램 만들기
```

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

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

```
glLinkProgram (ShaderProgramID);
```

```
// 셰이더 프로그램 링크하기
```

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

```
// 셰이더 프로그램에 링크하여 셰이더 객체 자체는 삭제 가능
```

```
glGetProgramiv (ShaderProgramID, GL_LINK_STATUS, &result);  
if (!result) {  
    glGetProgramInfoLog (triangleShaderProgramID, 512, NULL, errorLog);  
    cerr << "ERROR: shader program 연결 실패\n" << errorLog << endl;  
    return false;  
}
```

```
// 셰이더가 잘 연결되었는지 체크하기
```

```
glUseProgram (ShaderProgramID);
```

```
//--- 만들어진 셰이더 프로그램 사용하기
```

```
// 여러 개의 프로그램 만들 수 있고, 특정 프로그램을 사용하려면  
// glUseProgram 함수를 호출하여 사용 할 특정 프로그램을 지정한다.  
// 사용하기 직전에 호출할 수 있다.
```

```
return ShaderProgramID;
```

```
}
```

함수 프로토타입

- 셰이더 및 프로그램 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

- 셰이더 및 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 삼각형 그리기 (1)

- 버텍스 셰이더에 삼각형의 좌표값을 직접 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(0.25, 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 인자값부터 시작해서 세 번째 인자인 count만큼의 버텍스까지 한 버텍스에 대해 한 번에 하나씩 증가한다.
 - gl_VertexID의 값에 기반하여 각 버텍스에 다른 위치를 할당할 수 있다.
- gl_Position: 버텍스의 출력위치를 나타낸다.
 - vec4 (0.0, 0.0, 0.5, 1.0)을 할당하면 화면의 중앙에 위치한다.

셰이더 사용하여 삼각형 그리기

- 응용 프로그램의 삼각형 그리기 함수

```
void Display()
{
    glClearColor (1.0f, 1.0f, 1.0f, 1.0f);
    glClear (GL_COLOR_BUFFER_BIT);

    glUseProgram (ShaderProgramID);

    glDrawArrays(GL_TRIANGLES, 0, 3);

    glutSwapBuffers();

}
```

함수 프로토타입

- 배열 데이터로부터 프리미티브 렌더링
 - void **glDrawArrays** (GLenum mode, GLint first, GLsizei count);
 - 배열 데이터로부터 프리미티브 렌더링 하기
 - Mode: 렌더링 할 도형 종류
 - GL_POINTS, GL_LINE_STRIP, GL_LINE_LOOP, GL_LINES, GL_TRIANGLE_STRIP, GL_TRIANGLE_FAN, GL_TRIANGLES,
 - First: 배열에서 도형의 시작 인덱스
 - Count: 렌더링 할 인덱스 개수
 - 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_LINE_STRIP_ADJACENCY, GL_LINES_ADJACENCY, GL_TRIANGLE_STRIP, GL_TRIANGLE_FAN, GL_TRIANGLES, GL_TRIANGLE_STRIP_ADJACENCY, GL_TRIANGLES_ADJACENCY and GL_PATCHES
 - count: 렌더링할 요소의 개수
 - type: indices 값의 타입
 - GL_UNSIGNED_BYTE, GL_UNSIGNED_SHORT, or GL_UNSIGNED_INT.
 - indices: indices 가르키는 포인터

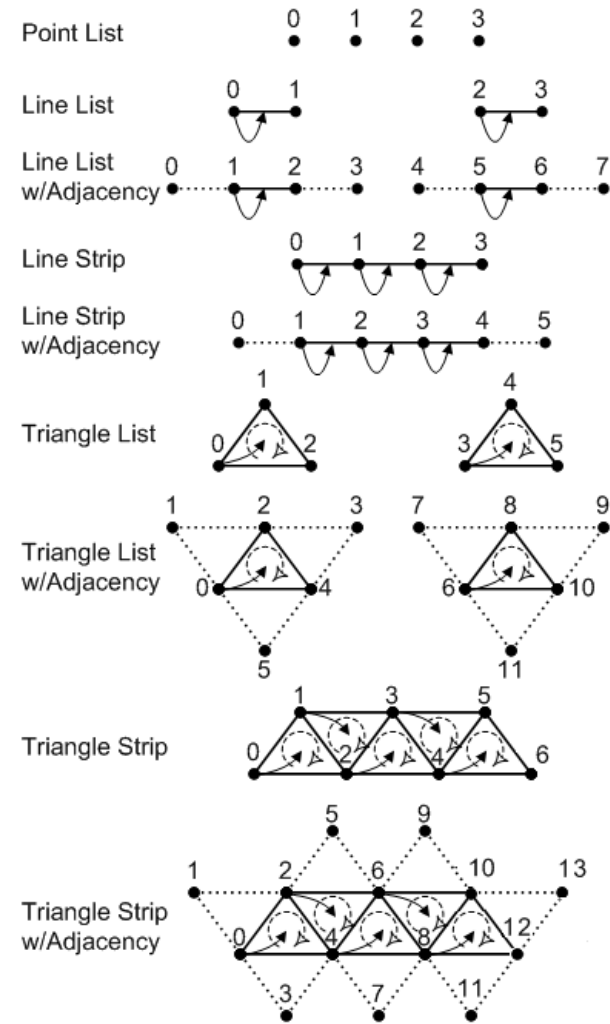
Primitive types

- GL에서 그리기

- `glDrawArrays`, `glDrawElements` 함수 등에서 수행됨
 - Points, lines, polygons들이 이 함수에 의해 그려진다.

- Primitive types

- Points:
- Line strips
- Line loops
- Lines
- Triangle strips
- Triangle fans
- Triangles
- Geometry shader:
 - Adjacency:
 - Geometry shader에서 사용됨
 - 각 끝점이 대응하는 adjacent 버텍스를 가지고 있는 독립적인 조각
 - Lines with adjacency
 - Line strips with adjacency
 - Triangles with adjacency
 - Triangle strips with adjacency



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

삼각형 그리기

- 메인 프로그램에서는
 - 윈도우 띄우기
 - freeGLUT 사용하여 윈도우 설정
 - 삼각형을 그릴 좌표값 정하기
 - 좌표값을 설정하고 그 좌표값을 사용하여 VBO, VAO 값 정하기
 - 삼각형의 속성 정하기
 - 버텍스 셰이더에 필요한 속성 설정하기 (예를 들어 좌표값)
 - 삼각형의 색상 정하기
 - 프래그먼트 셰이더에 색상 설정하기
 - 셰이더 읽고 프로그램 만들기
 - 셰이더 읽기, 컴파일, 링크, 프로그램 만들기
 - 화면에 그리기
 - 프로그램 사용하여 화면에 삼각형 그리기
- 버텍스 셰이더와 프래그먼트 셰이더를 이용하여 삼각형을 그려본다.

셰이더 사용하여 삼각형 그리기 (2)

- 도형 그리기
 - 좌표값을 받아 색이 입혀진 화면의 픽셀로 변환한다.
 - 그래픽스 파이프라인의 첫 단계로 삼각형을 그릴 수 있는 정점 데이터를 입력 받아 삼각형을 구성하는 위치값, 색상 등 정점 속성을 설정하고 화면의 픽셀 값으로 변환하여 그릴 수 있다.
 - 정점 속성: 위치값, 색상값, 노멀값, 텍스처값 등
- 버텍스 셰이더:
 - 정점 데이터가 정의되면 버텍스 셰이더로 전달한 후, GPU에 정점 데이터를 저장할 공간의 메모리를 할당
 - 저장한 데이터의 속성 설정
 - Vertex Buffer Object
 - Vertex Array Object
- 프래그먼트 셰이더:
 - 삼각형의 색상 값을 설정하여 화면에 그려질 수 있도록 한다.
- 메인 프로그램
 - 버텍스 셰이더와 프래그먼트 셰이더의 코드를 OpenGL 메인 프로그램에서 입력받아 화면에 그리도록 한다.
 - 셰이더 코드 컴파일, 셰이더 프로그램 생성하고 셰이더 코드 붙이기
 - 화면에 그리기

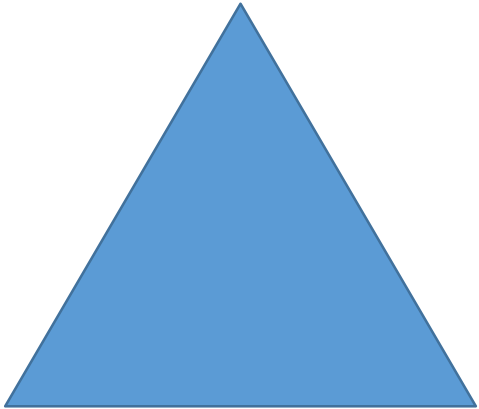
Vertex Specification

- 버텍스 세이더
 - 4요소를 가지고 있는 vertex attributes의 배열을 사용한다.
 - 배열의 인덱스: 0부터 시작, 최대값: GL_MAX_VERTEX_ATTRIBS
 - GL_MAX_VERTEX_ATTRIBS: 하드웨어에 따라 다름
 - glGetIntegerv (GL_MAX_VERTEX_ATTRIBS, &n) 함수로 확인할 수 있음
 - 버텍스 속성 (vertex attributes) 변경하기
 - void **glVertexAttrib{1234}{sfd}** (GLuint index, T values);
 - void **glVertexAttrib{123}{sfd}v** (GLuint index, const T values);
 - Index: 버텍스 속성 배열의 인덱스
 - Values: 버텍스 속성 값
- Vertex Array
 - 버텍스 데이터가 배열에 저장된다.
 - 이 배열의 데이터들은 GL명령어가 수행될 동안 다수의 geometric primitives를 사용할 수 있게 한다.
- Buffer Objects
 - 버텍스 배열이 메모리에 저장된다.
 - GL buffer object는 이 버텍스 배열을 저장, 초기화, 렌더링을 할 수 있게 한다.

Vertex Specifications

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

좌표값(0.5, 1.0, 0.0, 1.0)
색상값(1.0, 0.0, 0.0, 1.0)



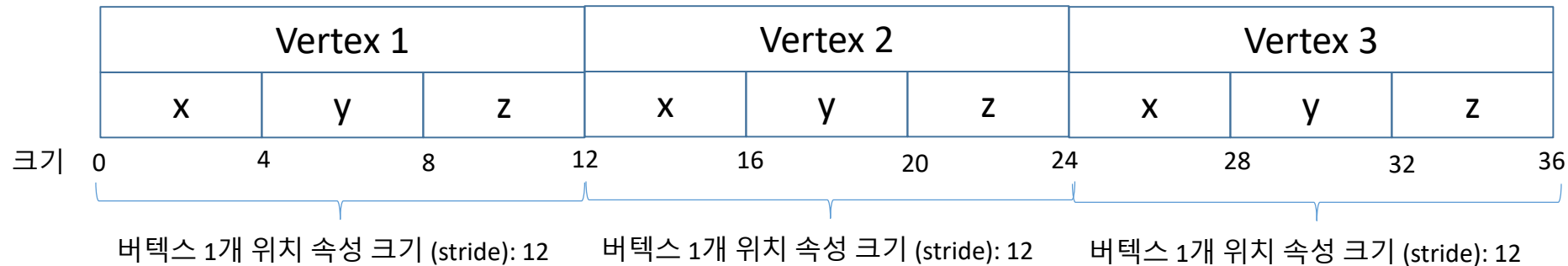
좌표값(0.0, 0.0, 0.0, 1.0)
색상값(0.0, 1.0, 0.0, 1.0)

좌표값(1.0, 0.0, 0.0, 1.0)
색상값(0.0, 0.0, 1.0, 1.0)

```
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  
};
```

Vertex Attribute

- 정점 속성 연결
 - 입력 데이터의 어느 부분이 vertex shader의 어떠한 정점 속성과 맞는지 직접 지정해야 함
- 예) 삼각형의 위치 값
 - 한 개의 버텍스는 3개의 실수 값으로 구성
 - 각 실수값은 4바이트로 구성
 - 즉, 한 개의 버텍스 속성 값(위치)은 12 바이트로 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 개의 속성을 저장, 또는 하나의 VBO에 여러가지 속성 저장도 가능

- 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, Vertex Buffer Object 생성 및 바인딩 함수
 - void **glGenBuffers** (GLsizei n, GLuint *buffers);
 - 버퍼 오브젝트 이름 생성
 - N: 생성할 이름 개수
 - Buffers: 버퍼 오브젝트 이름을 가리키는 배열
 - void **glBindBuffer** (GLenum target, GLuint buffer);
 - 버퍼 오브젝트 이름을 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
 - 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 데이터 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 속성을 지정. (layout (location = 0) 를 사용하여 속성의 위치를 0번째로 설정)
 - 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 채워져있다면 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 가질수도 있음)는 glVertexAttribPointer 함수를 호출할 때 GL_ARRAY_BUFFER 에 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);

Vertex Attribute

- 한 개의 VBO와 한 개의 VAO 사용하기 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
```

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

```
void main()
```

```
{
```

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

```
}
```

Vertex Attribute

- 버텍스 위치 속성 설정하기 응용 프로그램 예)

//--- 변수 선언

GLuint VAO, VBO_position;

//--- 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 shader program과 연결: 0번째 attribute

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,
```

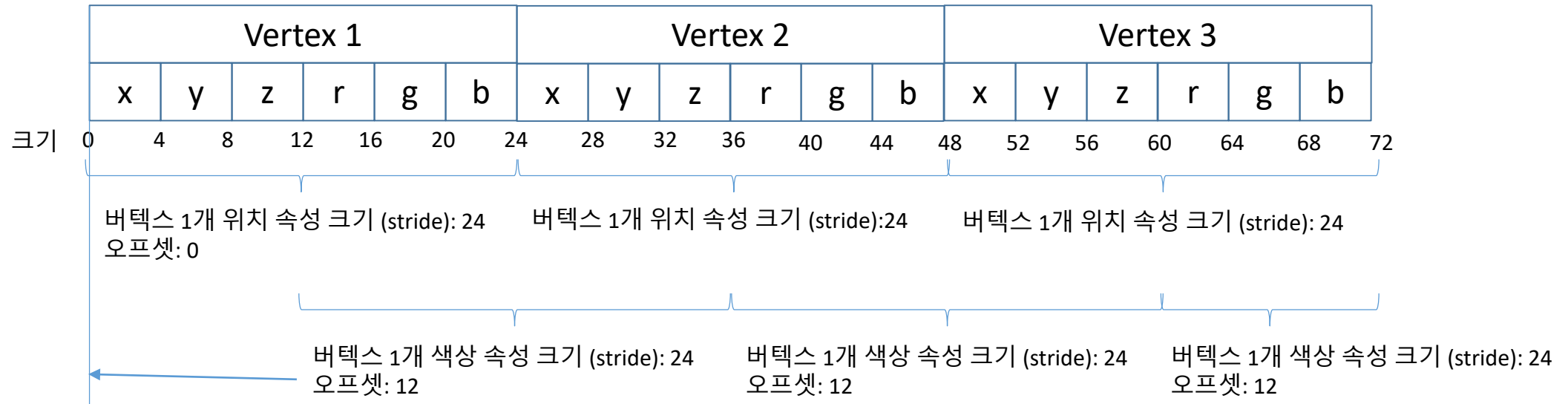
```
    0.0, 0.0, 0.0,          0.0, 1.0, 0.0,
```

```
    1.0, 0.0, 0.0,          0.0, 0.0, 1.0
```

```
};
```

Vertex Attribute: 2개

- 버텍스 포맷에 속성 추가
 - 좌표값 외에 색상 값을 속성으로 추가하는 경우



Vertex Attribute: 2개 --- 속성을 따로 저장할 때

- 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 입력받음
out vec4 FragColor;     // 프레임 버퍼로 출력
```

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

Vertex Attribute: 2개 --- 속성을 따로 저장할 때

- 응용 프로그램

//--- 변수 선언

```
Guint VAO, VBO_position, VBO_color;
```

//--- 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(vertex_list), vertex_list, GL_STATIC_DRAW);
```

//--- 색상 속성

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

```
glBindBuffer (GL_ARRAY_BUFFER, VBO_color);
```

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

//--- aPos 속성 변수에 값을 저장

```
GLint pAttribute = glGetAttribLocation (s_program_id, "vPos");
```

```
glBindBuffer (GL_ARRAY_BUFFER, VBO_position);
```

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

```
glEnableVertexAttribArray (pAttribute);
```

//--- aColor 속성 변수에 값을 저장

```
GLint cAttribute = glGetAttribLocation (s_program_id, "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

Vertex Attribute: 2개 --- 속성을 한 변수로 저장

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

- 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 outColor에 설정
}
```

- Fragment Shader

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

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

Vertex Attribute: 2개 --- 속성을 한 변수로 저장

- 응용 프로그램

//--- 변수 선언

GLuint VAO, VBO;

//--- 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 인덱스 0

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

glEnableVertexAttribArray (0);

//--- 색상 속성: 속성 인덱스 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
};
```


파일에서 코드 읽어오기 샘플

```
#include <stdlib.h>
```

```
#include <stdio.h>
```

```
char* filetobuf (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
```

```
}
```

최종 예) vertex shader와 fragment shader 만들기

//--- vertex shader: vertex.glvs

#version 330

// in_Position was bound to attribute index 0 and

// in_Color was bound to attribute index 1

in vec3 in_Position; // 위치 속성

in vec3 in_Color; // 색상 속성

out vec3 ex_Color; // 프래그먼트 셰이더에게 전달

void main(void)

{

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

ex_Color = in_Color;

}

//--- fragment shader: fragment.glfs

#version 330

// ex_Color: 버텍스 셰이더에서 입력받는 색상 값

// gl_FragColor: 출력할 색상의 값으로 응용 프로그램으로 전달 됨.

in vec3 ex_Color; // 버텍스 셰이더에게서 전달 받음

out vec4 gl_FragColor; // 색상 출력

void main(void)

{

gl_FragColor = vec4(ex_Color,1.0);

}

최종 예) VBO와 VAO 사용하여 데이터 저장하기

//--- 그리기 함수

```
void drawScene ()  
{
```

```
    const GLfloat triShape[3][3] = {  
        { 0.0, 1.0, 0.0 },      // top  
        { -1.0, 0.0, 0.0 },     // left  
        { 1.0, 0.0, 0.0 } };    // right
```

```
    const GLfloat colors[3][3] = {  
        { 1.0, 0.0, 0.0 },      // Red  
        { 0.0, 1.0, 0.0 },     // Green  
        { 0.0, 0.0, 1.0 } };   // Blue
```

```
    GLuint vao, vbo[2];
```

```
    GLchar *vertexsource, *fragmentsource; // 소스코드 저장 변수
```

```
    GLuint vertexshader, fragmentshader;    // 셰이더
```

```
    GLuint shaderprogram;    // 셰이더 프로그램
```

```
    // VAO 를 지정하고 할당하기
```

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

```
    // VAO를 바인드하기
```

```
    glBindVertexArray (vao);
```

```
    // 2개의 VBO를 지정하고 할당하기
```

```
    glGenBuffers (2, vbo);
```

```
//--- 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);
```

최종 예) vertex shader, fragment shader 프로그램 만들기

//--- 셰이더 프로그램 만들기

```
vertexsource = filetobuf ("vertex.glvs");  
fragmentsource = filetobuf ("fragment.glfs");
```

//--- 버텍스 셰이더 객체 만들기

```
vertexshader = glCreateShader (GL_VERTEX_SHADER);
```

// 셰이더 코드를 셰이더 객체에 넣기: GL로 보내진다. (소스코드: 문자열)

```
glShaderSource(vertexshader, 1, (const GLchar**)&vertexsource, 0);
```

// 버텍스 셰이더 컴파일하기

```
glCompileShader(vertexshader);
```

// 컴파일이 제대로 되지 않은 경우: 에러 체크

```
glGetShaderiv(vertexshader, GL_COMPILE_STATUS, &IsCompiled_VS);
```

```
if(IsCompiled_VS == FALSE)
```

```
{  
    glGetShaderiv(vertexshader, GL_INFO_LOG_LENGTH, &maxLength);  
    vertexInfoLog = (char *)malloc(maxLength);  
    glGetShaderInfoLog (vertexshader, maxLength, &maxLength, vertexInfoLog);  
    free (vertexInfoLog);  
    return;  
}
```

//--- 프래그먼트 셰이더 객체 만들기

```
fragmentshader = glCreateShader(GL_FRAGMENT_SHADER);
```

// 셰이더 코드를 셰이더 객체에 넣기: GL로 보내진다.

```
glShaderSource(fragmentshader, 1, (const GLchar**)&fragmentsource, 0);
```

// 프래그먼트 셰이더 컴파일

```
glCompileShader(fragmentshader);
```

// 컴파일이 제대로 되지 않은 경우: 컴파일 에러 체크

```
glGetShaderiv(fragmentshader, GL_COMPILE_STATUS, &IsCompiled_FS);
```

```
if(IsCompiled_FS == FALSE)
```

```
{  
    glGetShaderiv(fragmentshader, GL_INFO_LOG_LENGTH, &maxLength);  
    fragmentInfoLog = (char *)malloc(maxLength);  
    glGetShaderInfoLog(fragmentshader, maxLength, &maxLength, fragmentInfoLog);  
    free(fragmentInfoLog);  
    return;  
}
```

최종 예) 셰이더 객체 만들기

```
// 버텍스 셰이더와 프래그먼트 셰이더가 컴파일 됐고, 에러가 없는 경우
// GL 셰이더 객체를 만들어 두 셰이더를 링크한다.
// 셰이더 프로그램 객체 만들기
shaderprogram = glCreateProgram();
```

```
// 셰이더를 셰이더 프로그램 객체에 붙인다.
glAttachShader(shaderprogram, vertexshader);
glAttachShader(shaderprogram, fragmentshader);
```

```
// in_Position: 인덱스 0,
// in_Color: 인덱스 1 으로 속성 인덱스를 바인드한다.
// 속성 위치는 프로그램 링크 전에 수행한다.
glBindAttribLocation(shaderprogram, 0, "in_Position");
glBindAttribLocation(shaderprogram, 1, "in_Color");
```

```
// 프로그램 링크
// 이 때, 셰이더 프로그램은 에러 없이 바이너리 코드가 셰이더를 위하여
// 생성되고 그 코드가 GPU에 업로드 됨 (에러가 없다면)
glLinkProgram(shaderprogram);
```

```
// 링크가 되었는지 체크하기
glGetProgramiv(shaderprogram, GL_LINK_STATUS, (int *)&IsLinked);
if(IsLinked == FALSE)
{
    glGetProgramiv(shaderprogram, GL_INFO_LOG_LENGTH, &maxLength);
    shaderProgramInfoLog = (char *)malloc(maxLength);
    glGetProgramInfoLog(shaderprogram, maxLength, &maxLength,
                        shaderProgramInfoLog);

    free(shaderProgramInfoLog);
    return;
}
```

```
// 렌더링 파이프라인에 셰이더 불러오기
glUseProgram(shaderprogram);
// 사용할 VAO 불러오기
glBindVertexArray(vao);
```

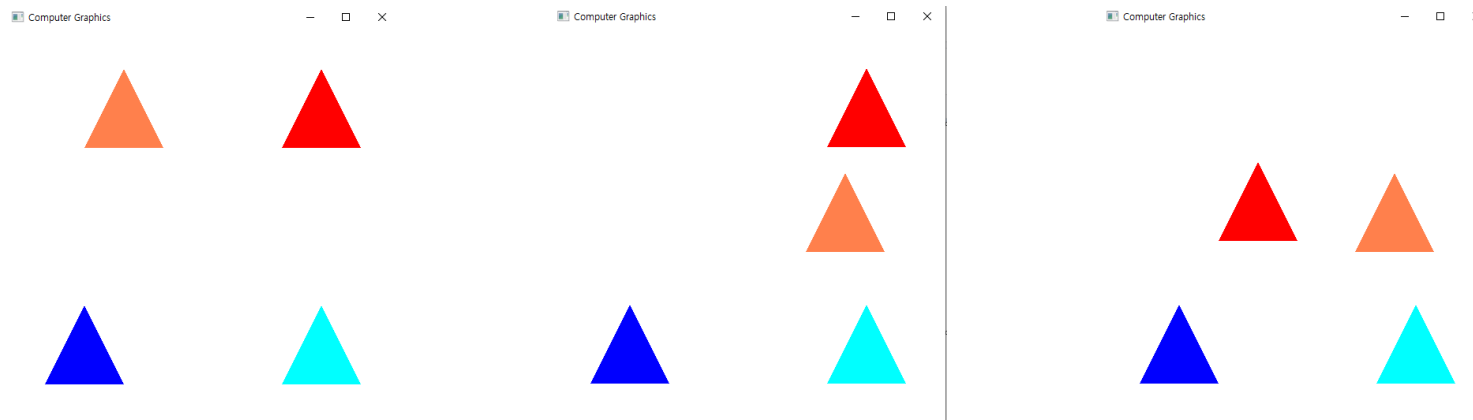
```
// 삼각형 그리기
glDrawArrays(GL_TRIANGLES, 0, 3);
```

프로그램 구조

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

실습 3

- 화면에 삼각형 그리기
 - 화면에 삼각형 사사분면에 4개 그리기
 - 삼각형 혹은 사각형을 그리기
 - 삼각형은 각각 다른 색상 설정
- 마우스를 누르면 그 위치에 새로운 삼각형 을 그린다.
 - 순서대로 이전에 그린 사각형을 삭제된다.
 - 마우스 클릭 -> 첫번째 삭제되고 삭제되고 마우스 위치에 삼각형
 - 마우스 클릭 -> 두번째 삼각형 삭제되고 마우스 위치에 삼각형
 - ...
 - 화면에는 항상 4개의 삼각형 만이 그려진다.

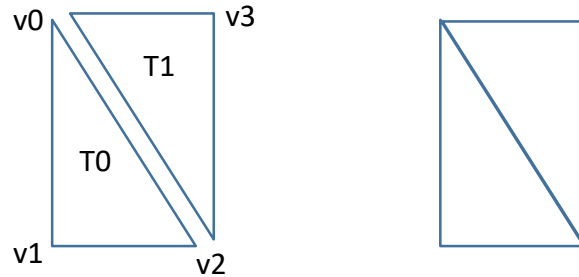


실습 4

Element Buffer Object

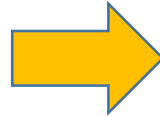
- Element Buffer Object (EBO)

- 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.

```
unsigned int EBO;
```

```
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);
```

```
glDrawElements (GL_TRIANGLES, 6, GL_UNSIGNED_INT, 0);    // glDrawArrays 함수 대신 그리기
```

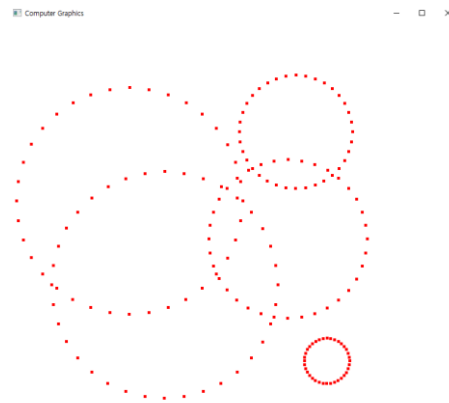
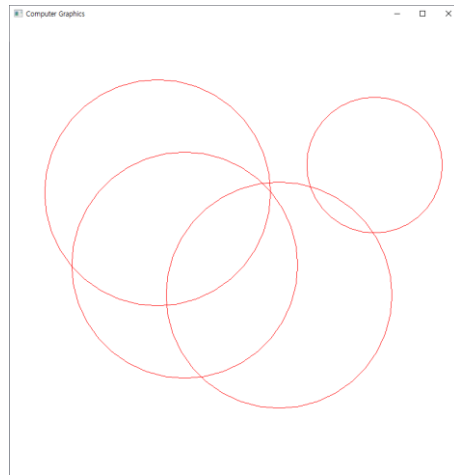
- 함수 프로토타입

- void **glDrawElements** (GLenum mode, GLsizei count, GLenum type, const GLvoid *indices);

- Mode: GL_POINTS, GL_LINE_STRIP, GL_LINE_LOOP, GL_LINES, GL_LINE_STRIP_ADJACENCY, GL_LINES_ADJACENCY, GL_TRIANGLE_STRIP, GL_TRIANGLE_FAN, GL_TRIANGLES, GL_TRIANGLE_STRIP_ADJACENCY, GL_TRIANGLES_ADJACENCY
- Count: 렌더링할 요소의 개수
- Type: indices 값의 타입, GL_UNSIGNED_BYTE, GL_UNSIGNED_SHORT, or GL_UNSIGNED_INT 중 1개
- Indices: 인덱스가 저장된 값

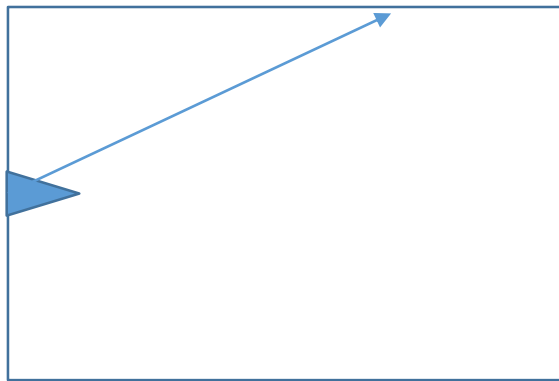
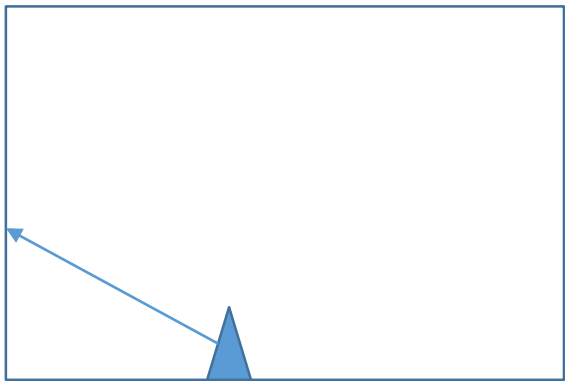
실습 5

- 원형 애니메이션 만들기
 - 마우스를 클릭하면 그 위치를 중심으로 원형으로 도형이 그려지고, 점점 밖으로 퍼져나가고, 특정 반지름이 되면 다시 시작 점부터 밖으로 퍼져나가는 애니메이션이 진행된다. 1개 이상의 원을 그릴 수 있도록 한다.
 - 최대 10개의 원을 그릴 수 있도록 한다.
 - 키보드 명령:
 - 1: 점으로 그리기
 - 2: 선으로 그리기
 - 여러 개의 원 중 임의의 원은 계속 밖으로 퍼져나가며 사라지도록 한다.



실습 6

- 삼각형 튀기기
 - 최대 10개의 삼각형이 임의의 위치에서 시작하여 각각 다른 방향 또는 속도로 이동한다.
 - 삼각형은 정삼각형이 아니다.
 - 벽을 만나면 다른 방향으로 이동한다.
 - 벽을 만나게 되면 삼각형의 방향이 바뀌게 된다.
 - 바뀐채로 계속 이동한다.



실습 7

실습 8

- KPU G-star 참석하기
 - 10월 16일 진행하는 KPU G-star에 참석하기 (대체 수업 진행)
 - 특강 1개, 게임대회 1종목, 과제전을 참석하고 보고서 작성하기
 - 위의 3 항목 별 참가한 후 소감과 인증샷을 추가하여 다음 시간에 출력하여 제출하기
 - 보고서는 3페이지 이상 작성하기

