모던 OpenGL 프로그래밍 기초

김준호

Abstract

쉐이더(shader) 기반의 모던 OpenGL 프로그래밍 기초를 학습한다. 모던 OpenGL의 서버-클라이언트 구조를 이해하고, 가장 기본적인 정점 쉐이더(vertex shader)와 프래그먼트 쉐이더(fragment shader) 코드를 이용한 OpenGL 2.x 기반 프로그래밍 가능한 렌더링 파이프라인(programmable rendering pipeline)을 구축한다.

Index Terms

Modern OpenGL, OpenGL 2.x, OpenGL shader, I/O storage qualifier, uniform, attribute, varying

1 모던 OpenGL 프로그래밍 기본

OpenGL 2.x 이후의 모던 OpenGL(modern OpenGL)은 기존 OpenGL 1.x가 사용하는 고정 렌더링 파이프라인(fixed rendering pipeline)을 버리고 프로그래밍 가능한 렌더링 파이프라인(programmable rendering pipeline)을 지원한다. 여기서는 가장 기본적인 모던 OpenGL에 해당하는 OpenGL 2.x에서 프로그래밍 가능한 렌더링 파이프라인을 설계하는 법을 살펴본다.

프로그래밍 가능한 렌더링 파이프라인이란 쉐이더(shader)를 이용하여 각 파이프라인 단계가 하드웨어적으로 고정된 방식이 아닌 프로그래머가 원하는 방식대로 돌아가도록 하는 그래픽스 파이프라인이다. OpenGL 2.x에서는 렌더링 파이프라인의 정점 프로세서(vertex processor)와 프래그먼트 프로세서(fragment processor)의 동작방식을 각각 정점 쉐이더(vertex shader)와 프래그먼트 쉐이더(fragment shader) 프로그래밍으로 제어할 수 있다.

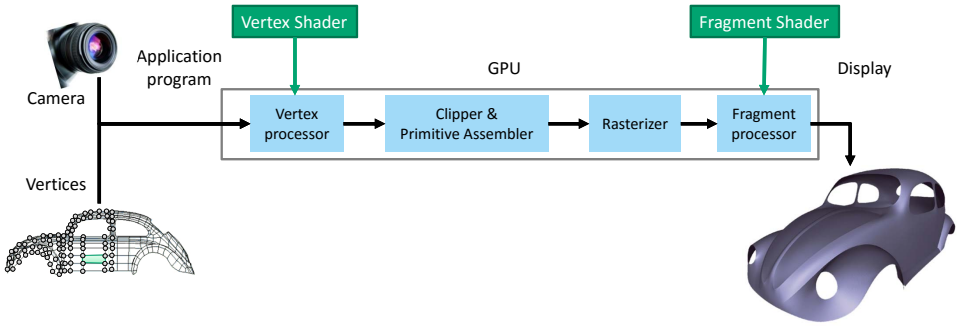


Fig. 1. OpenGL 2.x에서 쉐이더를 이용한 렌더링 파이프라인의 제어

1.1 클라이언트-서버 모델

모던 OpenGL에서는 프로그래밍 가능한 렌더링 파이프라인을 지원하기 위해 클라이언트-서버 모델을 따르는 프로그래밍 기법을 도입하고 있다. 여기서 클라이언트는 CPU와 메인메모리로 이루어지는 일반적인 컴퓨팅 환경을 의미하며, 서버는 GPU와 GPU 메모리로 이루어지는 그래픽카드 상의 컴퓨팅 환경을 의미한다(Fig. 2 참고). 일반적으로 클라이언트 측에서 돌아가는 코드는 C/C++로 작성된 OpenGL 코드이며, 서버 측에서 돌아가는 코드는 OpenGL 쉐이딩 언어(OpenGL Shading Language, 이하 GLSL)로 작성된 쉐이더 코드이다. GLSL 언어는 OpenGL 버전에 따라 조금씩 다른데, 여기서는 OpenGL 2.x에서 사용하는 GLSL 1.2 버전을 사용하기로 한다.

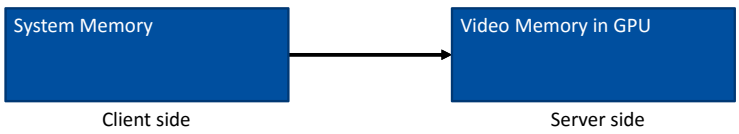


Fig. 2. OpenGL 클라이언트-서버 모델

국민대학교 소프트웨어학부, junho@kookmin.ac.kr

```
#version 120                // GLSL 1.2

uniform mat4 u_PVM;         // Proj * View * Model (global input)
attribute vec4 a_position;  // per-vertex position (per-vertex input)
attribute vec4 a_color;     // per-vertex color (per-vertex input)
varying vec4 v_color;       // per-vertex color (per-vertex output)

void main()
{
        v_color = a_color;
        gl_Position = u_PVM * a_position;
}
```

Fig. 3. 간단한 정점 쉐이더 소스파일: vertex.glsl

```
#version 120                    // GLSL 1.2

//uniform mat4 u_PVM;           // Proj * View * Model (global input)
varying vec4 v_color;           // per-fragment color (per-fragment input)

void main()
{
        gl_FragColor = v_color;
}
```

Fig. 4. 간단한 프래그먼트 쉐이더 소스파일: fragment.glsl

2 서버 측 프로그래밍: GLSL을 이용한 쉐이더 코딩

Fig. 3과 Fig. 4은 가장 간단한 형태의 정점 쉐이더와 프래그먼트 쉐이더 소스파일이다. Fig. 3의 vertex.glsl는 정점 프로세서를 작동시킬 방식을 기술하고 있으며, Fig. 4의 fragment.glsl는 프래그먼트 프로세서를 작동시킬 방식을 기술하고 있다.

Fig. 3과 Fig. 4에서 볼 수 있듯이, 쉐이더는 main 함수 내에서 전체적인 동작방식을 기술하고, main 함수 밖에서 글로벌 변수를 선언할 수 있다는 점에서 C/C++ 언어와 매우 유사한 프로그래밍 스타일을 띄고 있다. 그러나 일반적인 C/C++ 언어와 다른점이 몇가지 있다. 우선, 전처리기 #version으로 사용하고자 하는 GLSL의 버전을 지정해야 한다. 또한, 글로벌 변수 선언 부분을 보면 uniform, attribute, varying과 같이 일반적인 C/C++ 프로그래밍에서는 볼 수 없는 변수 선언 방식이 존재한다는 점을 주목하자. 마지막으로, 선언되지 않은 변수인 gl_Position과 gl_FragColor 등이 문제없이 사용되고 있다는 점도 주목할 필요가 있다.

2.1 쉐이더 입출력 저장소 식별자

Fig. 3과 Fig. 4에서 등장하는 uniform, attribute, varying는 쉐이더 언어에서 특별히 지원하는 3가지 입출력 저장소 식별자(I/O storage qualifier)이다. uniform 식별자는 해당 변수가 모든 쉐이더 단계에서 읽을 수 있는 글로벌 변수임을 지칭한다. attribute 식별자는 해당 변수가 각 정점의 특성값을 저장하고 있는 변수임을 지칭한다. 마지막으로 varying 식별자는 해당 변수가 정점 쉐이더로부터 프래그먼트 쉐이더로 전달되는 정보를 담고 있는 변수임을 지칭한다.

Fig. 3의 vertex.glsl 경우를 보면서, 정점 쉐이더에서 uniform, attribute, varying 식별자 기능을 자세히 살펴보도록 하자.

2.1.1 uniform 식별자

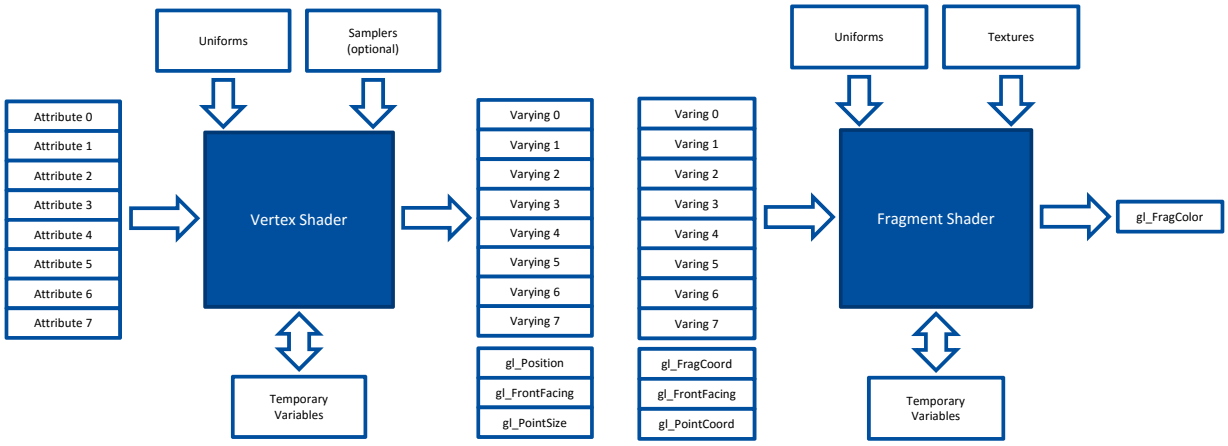
Fig. 3의 3번째 줄에 선언된 u_pvm_matrix 변수는 uniform 식별자로 선언되었기 때문에 그래픽스 파이프라인 전 단계에 등장하는 쉐이더에서 모두 읽을 수 있는 글로벌 변수가 된다. 예를 들어, Fig. 4에 기술된 프래그먼트 쉐이더 fragment.glsl의 3번째 줄이 코멘트 처리되어 있지 않았다면, 정점 쉐이더 vertex.glsl과 프래그먼트 쉐이더 fragment.glsl 모두 u_pvm_matrix로부터 동일한 글로벌 값을 읽을 수 있다는 의미이다.

요약하자면 uniform 식별자로 지칭된 변수는 다음의 성질을 가진다.

1) 전역성: 그래픽스 파이프라인의 모든 단계에서 동일한 변수값을 읽을 수 있다.
2) 읽기전용: 각 쉐이더에서 읽을 수만 있고 그 내용을 업데이트 할 수 없다.
3) 정점 무관성: 각 정점 데이터와는 무관한 데이터가 저장된다.

2.1.2 attribute 식별자

Fig. 3의 4-5번째 줄에서 attribute 식별자로 선언된 a_vertex, a_color 변수들은 각 정점별 데이터(per-vertex data)를 가진 변수이다. 여기서는 a_vertex는 각 정점의 삼차원 위치를 주기 위해 선언되었고, a_color는 각 정점의 색깔을 주기 위해 선언되었다. 이와 같이 attribute 식별자로 선언된 변수는 각 정점에 관한 데이터를 담고 있기 때문에 반드시 정점 쉐이더에만 등장해야 하고, 레스터화된 데이터를 다루는 프래그먼트 쉐이더에서는 attribute 식별자가 등장할 수 없다.



(a) 정점 쉐이더의 입출력 (b) 프래그먼트 쉐이더의 입출력

Fig. 5. 쉐이더 입출력

요약하자면 attribute 식별자로 지칭된 변수는 다음의 성질을 가진다.

1) 정점별 데이터: 각 정점에 연관된 데이터가 저장된다.
2) 정점 쉐이더 전용: 각 정점별 데이터이기 때문에 정점 쉐이더에서만 등장하고 프래그먼트 쉐이더에서는 등장할 수 없다.
3) 읽기전용: 정점 쉐이더에서는 해당 변수 값을 읽을 수만 있고 업데이트 할 수 없다.

2.1.3 varying 식별자

쉐이더 프로그래밍에서 varying 식별자로 선언되는 변수는 정점 쉐이더로부터 나온 데이터를 프래그먼트 쉐이더로 넘겨주는 역할을 한다. Fig. 3의 6번째 줄과 Fig. 4의 4번째 줄에서 볼 수 있듯이, 정점 쉐이더와 프래그먼트 쉐이더 모두 동일한 varysing 변수 v_color를 선언하고 있는데 이는 우연이 아니다.

이 같은 방식으로 동일한 varying 변수를 두 쉐이더 코드 모두에 선언함으로써 정점 쉐이더의 출력을 프래그먼트 쉐이더의 입력으로 넘겨줄 수 있다. 즉, Fig. 3의 10번째 줄과 같이 정점 쉐이더에서 정점별 출력값을 v_color에 저장하게 되면, 그 값을 렌더링 파이프라인의 래스터화(rasterization) 단계 거치면서 각 프래그먼트(per-fragment)의 값으로 변환된다. 프래그먼트 쉐이더에서는 입력으로 들어온 각 프래그먼트의 값을 이용하여 Fig. 3의 8번째 줄과 같이 최종 픽셀의 색깔을 결정한다.

요약하자면 varying 식별자로 지칭된 변수는 다음의 성질을 가진다.

1) 정점/프래그먼트 쉐이더 모두 등장: 동일한 변수명으로 두 쉐이더에서 모두 등장한다.
2) 정점 쉐이더에서는 정점별 출력용: 정점 쉐이더에서는 varying 식별자로 지칭된 변수는 정점별 출력값으로 활용된다.
3) 프래그먼트 쉐이더에서는 프래그먼트별 입력용: 프래그먼트 쉐이더에서는 varying 식별자로 지정된 변수는 프래그먼트별 입력값으로 활용된다.

2.2 정점 쉐어더 요약

요약하자면, 정점 쉐이더는 uniform과 attribute 식별자를 가진 변수들을 입력으로 사용하고 varying 식별자를 가진 변수를 출력으로 사용하는 프로그램으로 볼 수 있다 (Fig. 5(a) 참고).

- uniform: 그래픽스 파이프라인 전단계에 걸쳐 동일한 입력값이 들어오는 읽기전용 변수 지칭 (예: 카메라 정보)
- attribute: 정점별로 다른 입력값이 들어오는 읽기전용 변수 지칭 (예: 정점별 삼차원 위치)
- varying: 정점별 입력에 대해 계산한 출력값이 씌여질 변수 지칭 (예: 카메라 좌표계에서의 각 정점의 삼차원 위치)

2.2.1 정점 쉐이더의 빌트인 출력

화면이 구성되기 위해 정점 쉐이더에서 출력되어야 하는 값은 빌트인 출력변수 (built-in output variable)로 미리 정의되어 있다. Fig. 3의 11번째 줄에 있는 gl_Position 변수는 정점 프로세서를 거친 각 정점들이 정규화된 뷰 볼륨(canonical view volume)에서 어디에 위치하는지 출력하기 위한 빌트인 출력변수이다. GLSL 1.2에서 제공하는 정점 쉐이더에 대한 빌트인 출력변수는 gl_Position, gl_FrontFacing, gl_PointSize 등이 있으며, 이들 중 gl_Position는 반드시 출력값으로 설정되어야 한다. 자세한 사항은 GLSL 1.2 문서를 참고하도록 한다.

2.3 프래그먼트 쉐어더 요약

요약하자면, 프래그먼트 쉐이더는 uniform과 varying 식별자를 가진 변수들을 입력으로 사용하고 각 픽셀의 최종 색깔을 출력으로 설정하는 프로그램으로 볼 수 있다.

- uniform: 그래픽스 파이프라인 전단계에 걸쳐 동일한 입력값이 들어오는 읽기전용 변수 지칭 (예: 카메라 정보)
- varying: 프래그먼트별로 다른 입력값이 들어오는 읽기전용 변수 지칭(예: 프래그먼트별 색깔)

2.3.1 프래그먼트 쉐이더의 빌트인 입출력

화면이 구성되기 위해 프래그먼트 쉐이더에서 입출력되어야 하는 값은 빌트인 출력변수 (built-in input/output variable)로 미리 정의되어 있다. Fig. 4의 8번째 줄에 있는 gl_FragColor 변수는 프래그먼트 프로세서를 거친 각 프래그먼트가 어떤 색깔값을 가지는지 출력하기 위한 빌트인 출력변수이다. GLSL 1.2에서 제공하는 프래그먼트 쉐이더에 대한 빌트인 입출력변수는 gl_FragCoord, gl_FrontFacing, gl_PointCoord, gl_FragColor 등이 있으며, 이들 중 gl_FragColor는 반드시 출력값으로 설정되어야 한다. 자세한 사항은 GLSL 1.2 문서를 참고하도록 한다.

3 클라이언트 측 프로그래밍: OpenGL 함수를 이용한 C/C++ 코딩

모던 OpenGL에서 클라이언트 프로그밍에서는 OpenGL 함수를 이용한 C/C++ 코드로 구성되어 있다. 모던 OpenGL의 클라이언트 프로그램에서는 일반적으로 다음과 같은 일을 한다.

1) 쉐이더 프로그램 생성: 정점 쉐이더 코드와 프래그먼트 쉐이더 코드를 컴파일/링크하여 쉐이더 프로그램 생성
2) 쉐이더 프로그램의 입력변수 위치 얻기: 쉐이더 프로그램에서 uniform/attribute 식별자로 선언한 입력변수의 위치를 얻어옴
3) 클라이언트에서 서버로 데이터 전송: 메인메모리에 있는 데이터를 GPU 비디오메모리로 전송하여 폴리곤을 그림

3.1 쉐이더 프로그램 생성

정점 쉐이더 코드와 프래그먼트 쉐이더 코드를 컴파일하고 링크하여 쉐이더 프로그램 생성한다. 일반적으로 이 과정은 애플리케이션이 초기화될 때 한번 수행한다.

이 단계에서 눈여겨 봐야하는 OpenGL 함수는 다음과 같다.

- glCreateShader
- glShaderSource
- glCompileShader
- glAttachShader
- glLinkProgram

```
1  GLuint  program;        // 쉐이더 프로그램 객체의 레퍼런스 값
2
3  void init()
4  {
5    // ...
6
7    // GLEW 초기화를 통해 OpenGL 2.x 이후 버전 사용 가능
8    glewInit();
9
10   // 정점 쉐이더 객체를 파일로부터 생성
11   GLuint  vertex_shader
12     = create_shader_from_file("./shader/vertex.glsl", GL_VERTEX_SHADER);
13
14   // 프래그먼트 쉐이더 객체를 파일로부터 생성
15   GLuint fragment_shader
16     = create_shader_from_file("./shader/fragment.glsl", GL_FRAGMENT_SHADER);
17
18   // 쉐이더 프로그램 생성 및 컴파일
19   program = glCreateProgram();
20   glAttachShader(program, vertex_shader);
21   glAttachShader(program, fragment_shader);
22   glLinkProgram(program);
23
24   // ...
25 }
26
27 // GLSL 파일을 읽어서 컴파일한 후 쉐이더 객체를 생성하는 함수
28 GLuint create_shader_from_file(const std::string& filename, GLuint shader_type)
29 {
30   GLuint  shader = 0;
31   shader = glCreateShader(shader_type);
32
33   std::ifstream shader_file(filename.c_str());
```

```
  std::string   shader_string;

  shader_string.assign(
    (std::istreambuf_iterator<char>(shader_file)),
    std::istreambuf_iterator<char>());

  const GLchar* shader_src = shader_string.c_str();
  glShaderSource(shader, 1, (const GLchar**)&shader_src, NULL);
  glCompileShader(shader);

  return  shader;
}
```

3.2 쉐이더 프로그램의 입력변수 위치 얻기

쉐이더 프로그램에서 uniform 식별자나 attribute 식별자로 선언한 입력변수의 위치를 얻어온다. 일반적으로 이 과정은 애플리케이션이 초기화될 때 한번 수행한다.

이 단계에서 눈여겨 봐야하는 OpenGL 함수는 다음과 같다.

- glGetUniformLocation
- glGetAttribLocation

```
GLint   loc_u_PVM;      // uniform 변수 u_PVM 위치
GLint   loc_a_position; // attribute 변수 a_position 위치
GLint   loc_a_color;    // attribute 변수 a_color 위치

void init()
{
  // ...
  // 쉐이더 프로그램 생성 및 컴파일 이후

  // 쉐이더 코드에서 uniform 식별자나 attribute 식별자로 선언한 입력변수의 위치를 얻어옴
  loc_u_PVM = glGetUniformLocation(program,  "u_PVM");
  loc_a_position = glGetAttribLocation(program, "a_position");
  loc_a_color = glGetAttribLocation(program, "a_color");
}
```

3.3 클라이언트에서 서버로 데이터 전송

메인메모리에 있는 데이터를 GPU 비디오메모리로 전송하여 폴리곤을 그린다. 일반적으로 이 과정은 애플리케이션의 디스플레이 함수가 불릴 때마다 수행한다.

이 단계에서 눈여겨 봐야하는 OpenGL 함수는 다음과 같다.

- glUseProgram
- glUniform
- glVertexAttribPointer
- glEnableVertexAttribArray
- glDrawArrays
- glDisableVertexAttribArray

```
GLuint  program;        // 쉐이더 프로그램 객체의 레퍼런스 값
GLint   loc_u_PVM;      // uniform 변수 u_PVM 위치
GLint   loc_a_position; // attribute 변수 a_position 위치
GLint   loc_a_color;    // attribute 변수 a_color 위치

GLfloat PVM[] = { /* ... */ };          // Proj * View * Model
GLfloat position[] = { /* ... */ };     // per-vertex 3D homogeneous positions
GLfloat color[] = { /* ... */ };        // per-vertex RGBA color

void my_display()
{
```

```
    // ...

    // 특정 쉐이더 프로그램 사용
    glUseProgram(program);

    // uniform 변수 u_PVM 값 채우기
    glUniformMatrix4fv(loc_u_PVM, 1, false, PVM);
    // 서버측 attribute 변수 a_position, a_color 값을 채워줄 클라이언트 측 데이터 지정
    glVertexAttribPointer(loc_a_position, 4, GL_FLOAT, GL_FALSE, 0, position);
    glVertexAttribPointer(loc_a_color, 4, GL_FLOAT, GL_FALSE, 0, color);

    // 정점 attribute 배열 활성화
    glEnableVertexAttribArray(loc_a_position);
    glEnableVertexAttribArray(loc_a_color);
    // 폴리곤 그리기
    glDrawArrays(GL_TRIANGLES, 0, 6 /* 정점개수 */);
    // 정점 attribute 배열 비활성화
    glDisableVertexAttribArray(loc_a_position);
    glDisableVertexAttribArray(loc_a_color);

    // 쉐이더 프로그램 사용해제
    glUseProgram(0);

    // ...
}
```