# 버텍스 프로세싱

그래픽스 라이브러리의 기본 드로잉 과정은 크게 “버텍스 버퍼 & 인덱스 버퍼 생성 -> 버텍스 속성을 셰이더에 연결 -> 셰이더 상수 연결 -> 드로잉 과정”으로 이루어진다.

## 버텍스 버퍼 생성

* 이름 예약(버퍼를 구별하기 위한 핸들).
* 이름을 GL\_ARRAY\_BUFFER 바인딩 포인트에 바인딩.
* 메모리 할당 및 버텍스 데이터 저장.

*static const float data[] =* ***// CPU시스템 메모리 데이터.***

*{*

*0.25, -0.25, 0.5, 1.0,*

*-0.25, -0.25, 0.5, 1.0,*

*0.25, 0.25, 0.5, 1.0*

*};*

***// 이름 예약.***

*GLuint buffer;*

***// GPU버퍼에 대한 이름 생성. 이하 버퍼로 명명함.***

*glGenBuffers(1, &buffer);*

***// 바인딩 포인트에 바인딩.***

*glBindBuffer(GL\_ARRAY\_BUFFER, buffer);*

***// 1024\*1024 사이즈 버퍼 메모리 할당.***

*glBufferData(GL\_ARRAY\_BUFFER, 1024\*1024, NULL, GL\_STATIC\_DRAW);*

***// 데이터 저장 1 : 메모리 할당과 동시에.***

*// glBufferData(GL\_ARRAY\_BUFFER, 1024\*1024, data, GL\_STATIC\_DRAW);*

***// 데이터 저장 2 : 버퍼 메모리 0 오프셋 위치에 데이터 저장.***

*glBufferSubData(GL\_ARRAY\_BUFFER, 0, sizeof(data), data);*

***// 데이터 저장 3 : 포인터를 얻어서 데이터 복사하는 방법.***

*void\* ptr = glMapBuffer(GL\_ARRAY\_BUFFER, GL\_WRITE\_ONLY);*

*memcpy(ptr, data, sizeof(data));*

*glUnmapBuffer(GL\_ARRAY\_BUFFER);* ***// 포인터 사용 마침을 통보.***

*glClearBufferSubData();*

*glCopyBufferSubData();*

## 인덱스 버퍼 생성

* 이름 예약(버퍼를 구별하기 위한 핸들).
* 이름을 GL\_ELEMENT\_ARRAY\_BUFFER 바인딩 포인트에 바인딩.
* 메모리 할당 및 인덱스 데이터 저장.

*static const GLfloat vertex\_positions[] =*

*{*

*-0.25f, -0.25f, -0.25f,*

*-0.25f, 0.25f, -0.25f,*

*0.25f, -0.25f, -0.25f,*

*0.25f, 0.25f, -0.25f,*

*0.25f, -0.25f, 0.25f,*

*0.25f, 0.25f, 0.25f,*

*-0.25f, -0.25f, 0.25f,*

*-0.25f, 0.25f, 0.25f,*

*};*

*static const GLushort vertex\_indices[] =*

*{*

*0, 1, 2,* ***// 0, 1, 2 위치의 버텍스를 순서대로 삼각형으로 구성.***

*2, 1, 3,*

*2, 3, 4,*

*4, 3, 5,*

*4, 5, 6,*

*6, 5, 7,*

*6, 7, 0,*

*0, 7, 1,*

*6, 0, 2,*

*2, 4, 6,*

*7, 5, 3,*

*7, 3, 1*

*};*

***// 버텍스 버퍼 생성.***

*…*

***// 인덱스 버퍼 생성.***

*glGenBuffers(1, &index\_buffer);*

*glBindBuffer(GL\_ELEMENT\_ARRAY\_BUFFER, index\_buffer);*

*glBufferData(GL\_ELEMENT\_ARRAY\_BUFFER, sizeof(vertex\_indices),*

*vertex\_indices, GL\_STATIC\_DRAW);*

***// 각각 3개의 버텍스를 가진 2개의 삼각형으로 이루어진 6개의 면을 그린다.***

*glDrawElements(GL\_TRIANGLES, 36, GL\_UNSIGNED\_SHORT, 0);*

## 버텍스 속성 셰이더 연결

* glVertexAttribPointer() 함수 : 가장 간단하고 편리함.
* glVertexAttribFormat(), glVertexAttribBinding(), glBindVertexBuffer() : 저수준으로

사용.

***// 1. glVertexAttribPoint 함수 사용 방법.***

***// 1-1 : 촘촘히 패킹된 독립적인 데이터 배열(배열-구조체) 사용 방법.***

***// c++ 에서 버텍스 속성 설정.***

*glBindBuffer(GL\_ARRAY\_BUFFER, buffer);*

*glVertexAttribPointer(0,* ***// 속성 0.***

*4,* ***// 4 컴포넌트.***

*GL\_FLOAT,* ***// 부동소수점 데이터.***

*GL\_FALSE,* ***// 정규화 안 됨.***

*0,* ***// 촘촘히 패킹되어 있음.***

*NULL);* ***// 오프셋은 0.***

*glEnableVertexAttribArray(0);* ***// 활성화.***

*GLuint color\_location = glGetAttribLocation(program, “color”);* ***// location 구하기.***

*glVertexAttribPointer(color\_location,* ***// -1 이 아니면 정상 연결됨.***

*3,*

*GL\_FLOAT,*

*GL\_FALSE,*

*0,*

*NULL);*

*glEnableVertexAttribArray(color\_location); // 활성화.*

***// 셰이더에서 버텍스 입력 속성 사용.***

*#version 430 core*

*layout (location = 0) in vec4 position;*

*in vec4 color;* ***// layout 사용안함.***

*void main(void)*

*{*

*gl\_Position = position;*

*};*

***// 1-2 : 구조체-배열 사용 방법.***

*struct Vertex*

*{*

*// 위치*

*float x;*

*float y;*

*float z;*

*// 색상.*

*float r;*

*float g;*

*float b;*

*};*

*… 버퍼 생성 생략.*

*glVertexAttribPointer(0, 3, GL\_FLOAT, GL\_FALSE,*

*sizeof(Vertex), (void\*)offsetof(Vertex, x));*

*glEnableVertexAttribArray(0);*

*glVertexAttribPointer(1, 3, GL\_FLOAT, GL\_FALSE,*

*sizeof(Vertex), (void\*)offsetof(Vertex, r));*

*glEnableVertexAttribArray(1);*

***// 2. glVertexAttribFormat(), glVertexAttribBinding(), glBindVertexBuffer() 사용 방법.***

***// 셰이더 버텍스 입력 선언이 아래와 같다면.***

*#version 430 core*

*layout (location = 0) in vec4 position;*

*layout (location = 1) in vec3 normal;*

*layout (location = 2) in vec2 tex\_coord;*

*layout (location = 3) in vec4 color;*

*layout (location = 4) in int material\_id;*

*…*

***// C++ 에서 버텍스 속성 입력 연결***

*typedef struct VERTEX\_t*

*{*

*vmath::vec4 position;*

*vmath::vec3 normal;*

*vmath::vec2 tex\_coord;*

*GLubyte color[3];*

*int material\_id;*

*} VERTEX;*

*glVertexAttribFormat(0, 4, GL\_FLOAT, GL\_FALSE, offsetof(VERTEX, position));*

*glVertexAttribFormat(1, 3, GL\_FLOAT, GL\_FALSE, offsetof(VERTEX, normal));*

*glVertexAttribFormat(2, 2, GL\_FLOAT, GL\_FALSE, offsetof(VERTEX, texcoord));*

*glVertexAttribFormat(3, 3, GL\_UNSIGNED\_BYTE, GL\_TRUE,*

*offsetof(VERTEX, color));*

*glVertexAttribFormat(4, 1, GL\_INT, offsetof(VERTEX, material\_id));*

***// 어떤 버퍼에 매핑할지를 설정. 만들어놓은 버퍼의 인덱스가 0이라면.***

*glVertexAttribBinding(0, 0);*

*glVertexAttribBinding(1, 0);*

*glVertexAttribBinding(2, 0);*

*glVertexAttribBinding(3, 0);*

*glVertexAttribBinding(4, 0);*

***// 버퍼 바인딩 포인트에 바인딩.***

*glBindVertexBuffer(0, buffer, offsetof(VERTEX, position), sizeof(VERTEX));*

## 셰이더 상수 전달

셰이더 프로그램에서 사용하는 변환행렬이나 텍스처 등의 데이터는 보통 애플리케이션에서 직접 전달한다. 이 데이터를 OpenGL 에서는 유니폼(uniform) 으로 DirectX 에서는 셰이더 상수(Shader Constant) 로 불리운다.

* 디폴트 블록 유니폼
  + 기본 블록을 사용. 별도 블록을 사용하기 위해 버퍼 생성 등의 번거로운 일이 없음.
  + 유니폼 하나 하나 세팅하는 방법으로 사용이 매우 간단.
  + 유니폼이 많을수록 애플리케이션에서 세팅하는 단계를 거쳐야 하기 때문에 느림.

*uniform float time;* ***// 실수형***

*uniform int index;* ***// 정수형***

*uniform vec4 color;* ***// 벡터형***

*uniform bool flag;* ***// bool.***

*uniform vec4 lightPos;* ***// 벡터형.***

*uniform mat4 mvp;* ***// 행렬***

*layout (location = 4) uniform vec4 loc4Uniform;* ***// 위치 인덱스를 4로 명시적으로 선언.***

*GLint locTime, locIndex, locColor, locMVP, locFlag;*

***// 리턴값이 -1 이 아니면 위치 인덱스 유효.***

*locTime = glGetUniformLocation(shaderProgramIdx, “time”);*

*locIndex = glGetUniformLocation(shaderProgramIdx, “index”);*

*locColor = glGetUniformLocation(shaderProgramIdx, “color”);*

*locFlag = glGetUniformLocation(shaderProgramIdx, “flag”);*

*locMVP = glGetUniformLocation(shaderProgramIdx, “mvp”);*

*…*

*glUseProgram(shaderProgramIdx);*

***// glUniform\*() 함수 계열 사용.***

*glUniform1f(locTime, 45.2f);* ***// time = 45.2f;***

*glUniform1i(locIndex, 42);* ***// index = 42;***

*glUniform4f(locColor, 1.0f, 0.0f, 0.0f, 1.0f);* ***// color = {1.0f, 0.0f, 0.0f, 1.0f};***

*glUniform1i(locFlag, GL\_FALSE);* ***// flag = false;***

***// glUniform\*() 배열 형태 함수 사용.***

*GLfloat fTime = 45.2f;*

*glUniform1fv(locTime, 1, &fTime);* ***// time = 45.2f;***

*GLint iIndex = 42;*

*glUniform1iv(locIndex, 1, &iIndex);* ***// index = 42;***

*GLfloat vColor = {1.0f, 1.0f, 1.0f, 1.0f};*

*glUniform4f(locColor, 1, &vColor);* ***// color = {1.0f, 0.0f, 0.0f, 1.0f};***

*glUniform1i(locFlag, GL\_FALSE);* ***// flag = false;***

* 유니폼 블록
  + 구조체 형식으로 많은 유니폼들을 한 데 모아놓고 선언.
  + 버퍼에(유니폼 버퍼 객체) 유니폼 데이터 저장.
  + 한 번에 데이터를 업데이트 하기 때문에 성능상 이점이 있음.

*layout(std140) uniform TransformBlock* ***// 표준 패킹 처리. std140.***

*{*

***// 멤버 기본정렬 오프셋 정렬된 오프셋***

*float scale;* ***// 4 0 0***

*vec3 translation;* ***// 16 4 16***

*float rotation[3];* ***// 16 28 32(rotation[0])***

***// 48(rotation[1])***

***// 64(rotation[2])***

*mat4 projection\_matrix;* ***// 16 80 80(column 0)***

***// 96(column 1)***

***// 112(column 2)***

***// 128(column 3)***

*} transform;*

***// std140 표준으로 선언하면 정렬된 메모리 오프셋을 쉽게 알 수 있지만 별도 패킹***

***// 처리 시 오프셋을 알 수 없어 별도로 유니폼 블록의 유니폼 정보를 확인해야 한다.***

*static const GLchar\* uniformNames[4] =*

*{*

*“TransformBlock.scale”,*

*“TransformBlock.translation”,*

*“TransformBlock.rotation”,*

*“TransformBlock.projection\_matrix”*

*};*

***// 유니폼 블록 멤버 인덱스 얻어오기.***

*GLuint uniformIndices[4];*

*glGetUniformIndices(program, 4, uniformNames, uniformIndices);*

***// 유니폼 블록 멤버 정보 얻어오기.***

*GLint uniformOffsets[4];*

*GLint arrayStrides[4];*

*GLint matrixStrides[4];*

***// GL\_UNIFORM\_TYPE 으로 해당 정보를 얻어올 수 있음.***

*glGetActiveUniformsiv(program, 4, uniformIndices, GL\_UNIFORM\_OFFSET, uniformOffsets);*

*glGetActiveUniformsiv(program, 4, uniformIndices, GL\_UNIFORM\_ARRAY\_STRIDE, arrayStrides);*

*glGetActiveUniformsiv(program, 4, uniformIndices, GL\_UNIFORM\_MATRIX\_STRIDE, matrixStrides);*

*unsigned char\* buffer = (unsigned char\*)malloc(4096);*

***// scale 값 저장.***

*\*((float\*)(buffer + uniformOffsets[0])) = 3.0f;*

***// translation 값 저장.***

*((float\*)(buffer + uniformOffsets[1]))[0] = 1.0f;*

*((float\*)(buffer + uniformOffsets[1]))[1] = 2.0f;*

*((float\*)(buffer + uniformOffsets[1]))[2] = 3.0f;*

***// rotation 값 저장.***

***// Transform.rotation[0] 은 버퍼 내 uniformOffsets[2] 바이트에 위치.***

***// rotation 의 유니폼 블록 내 인덱스는 2이므로 arrayStrides[2] 를 사용.***

***// arrayStrides[0] 은 scale 멤버로 배열이 아니므로 0.***

***// 배열의 각 요소는 arrayStrides[2] 바이트의 배수마다 위치.***

*const GLfloat rotations[] = {30.0f, 40.0f, 60.0f};*

*unsigned int offset = uniformOffsets[2];*

*for (int n = 0; n < 3; n++)*

*{*

*\*((float\*)(buffer + offset)) = rotations[n];*

*offset += arrayStrides[2];*

*}*

***// projection\_matrix 값 설정.***

***// TransformBlock.projection\_matrix 의 첫 번째 열은 버퍼의 uniformOffsets[3]***

***// 위치에 있다. 각 열은 서로 matrixStride[3] 만큼 떨어져 있고, 내부적으로는***

***// vec4 로 이루어져 있다.***

*const GLfloat matrix[] =*

*{*

*1.0f, 2.0f, 3.0f, 4.0f,*

*9.0f, 8.0f, 7.0f, 6.0f,*

*2.0f, 4.0f, 6.0f, 8.0f,*

*1.0f, 3.0f, 5.0f, 7.0f*

*};*

*for (int i = 0; i < 4; i++)*

*{*

*GLuint offset = uniformOffsets[3] + matrixStride[3] \* i;*

*for (int j = 0; j < 4; j++)*

*{*

*\*((float\*)(buffer + offset)) = matrix[i \* 4 + j];*

*offset += sizeof(GLfloat);*

*}*

*}*

***// 유니폼 블록을 버퍼에 할당.***

***// 1. 일단 버퍼 생성.***

*… buffer* ***// 생략***

***// 2. TransformBlock 유니폼 블록의 인덱스를 얻는다.***

*GLuint transformBlock\_idx = glGetUniformBlockIndex(program, “TransformBlock”);*

***// 3. 버퍼 바인딩일 유니폼 블록에 할당한다.***

*// transformBlock 은 버퍼 0 으로 바인딩.*

*glUniformBlockBinding(program, transformBlock\_idx, 0);*

***// 4. 버퍼를 바인딩 포인트에 할당한다.***

*glBindBufferBase(GL\_UNIFORM\_BUFFER, 0, buffer);*

* 셰이더 스토리지 블록
  + 셰이더에서 쓰기도 가능하다(유니폼 버퍼는 읽기만 가능).
  + std430 패킹 레이아웃 지시어 사용 가능(좀 더 촘촘한 패킹 가능).
  + 어토믹 연산으로 셰이더 간 메모리 동기화 가능.
  + 블록 크기에 대한 명시적인 상한선이 없음(유니폼 블록에 비해 훨씬 더 큼)

*#version 430 core*

*struct Vertex*

*{*

*vec4 position;*

*vec3 color;*

*};*

***// buffer 키워드를 사용 스토리지 버퍼임을 명시. 버퍼 바인딩 번호는 0.***

*layout (binding = 0, std430) buffer Vertices*

*{*

*Vertex vertices[];*

*};*

*uniform mat4 transform\_matrix;*

*out VS\_OUT*

*{*

*vec3 color;*

*} vs\_out;*

*void main()*

*{*

*gl\_Position = transform\_matrix \* vertices[gl\_VertexID].position;*

*vs\_out.color = vertices[gl\_VertexID].color;*

*}*

***// cpp 에서 연결 세팅.***

*GLuint storage\_buffer;*

*glGenBuffers(1, storage\_buffer);*

*glBindBuffer(GL\_SHADER\_STORAGE\_BUFFER, storage\_buffer);*

*glBufferData(GL\_SHADER\_STORAGE\_BUFFER, 1024 \* 1024, NULL, GL\_DYNAMIC\_COPY);*

## 드로잉 커맨드

* 직접 드로잉
  + glDrawArrays()
    - 버텍스 버퍼에 저장된 버텍스 순서대로 처리.

*const GLfloat color[] = { (float)sin(tm) \* 0.5f + 0.5f,*

*(float)cos(tm) \* 0.5f + 0.5f,*

*0.0f, 1.0f };*

*glClearBufferfv(GL\_COLOR, 0, color);*

*glUseProgram(program);*

*Glfloat attrib[] = { (float)sin(tm) \* 0.5f, (float)cos(tm) \* 0.6f, 0.0f, 0.0f };*

***// 버텍스 입력 속성 0의 값을 갱신한다.***

*glVertexAttrib4fv(0, attrib);*

***// 삼각형을 그린다.***

*glDrawArrays(GL\_TRIANGLES, 0, 3);*

* + glDrawElements()
    - 인덱스 버퍼에 저장된 인덱스 순서에 따라 드로잉.
    - 인덱스 버퍼는 GL\_ELEMENT\_ARRAY\_BUFFER 타깃에 바인딩.
    - 버텍스 중복 없앰.

*static const GLfloat vertex\_positions[] =*

*{*

*-0.25f, -0.25f, -0.25f,*

*-0.25f, 0.25f, -0.25f,*

*0.25f, -0.25f, -0.25f,*

*0.25f, 0.25f, -0.25f,*

*0.25f, -0.25f, 0.25f,*

*0.25f, 0.25f, 0.25f,*

*-0.25f, -0.25f, 0.25f,*

*-0.25f, 0.25f, 0.25f,*

*};*

*static const GLushort vertex\_indices[] =*

*{*

*0, 1, 2,*

*2, 1, 3,*

*2, 3, 4,*

*4, 3, 5,*

*4, 5, 6,*

*6, 5, 7,*

*6, 7, 0,*

*0, 7, 1,*

*6, 0, 2,*

*2, 4, 6,*

*7, 5, 3,*

*7, 3, 1*

*};*

*glGenBuffers(1, &position\_buffer);*

*glBindBuffer(GL\_ARRAY\_BUFFER, position\_buffer);*

*glBufferData(GL\_ARRAY\_BUFFER, sizeof(vertex\_positions),*

*vertex\_positions, GL\_STATIC\_DRAW);*

*glVertexAttribPointer(0, 3, GL\_FLOAT, GL\_FALSE, 0, NULL);*

*glEnableVertexAttribArray(0);*

*glGenBuffers(1, &index\_buffer);*

*glBindBuffer(GL\_ELEMENT\_ARRAY\_BUFFER, index\_buffer);*

*glBufferData(GL\_ELEMENT\_ARRAY\_BUFFER, sizeof(vertex\_indices),*

*vertex\_indices, GL\_STATIC\_DRAW);*

***// 각각 3개의 버텍스를 가진 2개의 삼각형으로 이루어진 6개의 면을 그린다.***

*glDrawElements(GL\_TRIANGLES, 36, GL\_UNSIGNED\_SHORT, 0);*

* + glDrawElementsBaseVertex()
    - 특정 위치의 버텍스 오프셋을 넘겨주어 해당 위치부터 드로잉.
    - 한 개의 퍼버에 두 개의 오브젝트 버텍스들이 담겨있을 때 두 번째 오브젝트를 드로잉 하고 싶다면 두 번째 오브젝트의 버텍스 시작 지점을 넘겨주어 드로잉하게 한다.
    - void glDrawElementsBaseVertex(GLenum mode,

GLsizei cound,

GLenum type,

GLvoid\* indices,

Glint basevertex)

* 인스턴싱(직접 드로잉)

동일한 객체를 여러 번 반복해서 그려야 할 경우(잔디나 경기장의 관중들) 동일한 데이터를 매 번 세팅하고(유니폼 설정 등) 드로잉 커맨드를 호출(glDrawArrays() 등)하는 과정을 몇 천 번 또는 몇 백만 번 반복해야 한다면 GPU 는 대부분의 시간을 실제로 렌더링하는 대신 OpenGL에 커맨드를 보내는 데 소비할 것이므로 매우 느릴 것이다. 이 경우 인스턴싱 기법을 사용한다. 커맨드는 한 번만 호출하고 GPU단에서 반복적으로 드로잉 한다.

* + 함수
    - void glDrawArraysInstanced(GLenum mode,

GLint first,

GLsizei count,

GLsizei instancecount);

* + - void glDrawElementsInstanced(GLenum ode,

GLsizei count,

GLenum type,

const void\* indices,

GLsizei instancecount);

* + - 인자의 instancecount 를 100000로 설정하면 지오메트리의 인스턴스를 십만 번 그린다. instancecount를 1로 설정하면 지오메트리의 인스턴스를 한 번만 호출하므로 glDrawArrays() 나 glDrawElements() 를 호출하는 것과 동일하다.
    - 인스턴스 위치 색상 등의 정보 관리
      * gl\_InstanceID : OpenGL 에서 관리되는 GPU 변수로서 GPU 에서 지오메트리 인스턴스가 렌더링 될 때마다 카운트 된다. 100 번째 렌더링 시 gl\_InstanceID 값은 99 가 될 것이다.
      * 텍스처에 정보를 담기 : 2D 배열 텍스처에 R,G,B,A 값을 정보 속성으로 담아서 읽기(회전, 스케일, 색상 등의 값 등 사용자가 정의하여 임의 사용).
      * 인스턴스 배열(버퍼) 사용 : 버퍼에 정보를 담고 렌더링 시 마다 버퍼의 내용을 참조하는 방식. 참조 사이클은 glVertexAttribDivisor() 함수를 사용한다.

glVertexAttribDivisor(GLuint index, // 버퍼 인덱스.

GLuint divisor); // 참조사이클, 1이면 매번.

*static const GLfloat square\_vertices[] =*

*{*

*-1.0f, -1.0f, 0.0f, 1.0f,*

*1.0f, -1.0f, 0.0f, 1.0f,*

*1.0f, 1.0f, 0.0f, 1.0f,*

*-1.0f, 1.0f, 0.0f, 1.0f*

*};*

*static const GLfloat instance\_colors[] =*

*{*

*1.0f, 0.0f, 0.0f, 1.0f,*

*0.0f, 1.0f, 0.0f, 1.0f,*

*0.0f, 0.0f, 1.0f, 1.0f,*

*1.0f, 1.0f, 0.0f, 1.0f*

*};*

*static const GLfloat instance\_positions[] =*

*{*

*-2.0f, -2.0f, 0.0f, 0.0f,*

*2.0f, -2.0f, 0.0f, 0.0f,*

*2.0f, 2.0f, 0.0f, 0.0f,*

*-2.0f, 2.0f, 0.0f, 0.0f*

*};*

*GLuint offset = 0;*

*glGenVertexArrays(1, &square\_vao);*

*glGenBuffers(1, &square\_vbo);*

*glBindVertexArray(square\_vao);*

*glBindBuffer(GL\_ARRAY\_BUFFER, square\_vbo);*

*glBufferData(GL\_ARRAY\_BUFFER,*

*sizeof(square\_vertices) +*

*sizeof(instance\_colors) +*

*sizeof(instance\_positions), NULL, GL\_STATIC\_DRAW);*

*glBufferSubData(GL\_ARRAY\_BUFFER, offset, sizeof(square\_vertices), square\_vertices);*

*offset += sizeof(square\_vertices);*

*glBufferSubData(GL\_ARRAY\_BUFFER, offset, sizeof(instance\_colors), instance\_colors);*

*offset += sizeof(instance\_colors);*

*glBufferSubData(GL\_ARRAY\_BUFFER, offset, sizeof(instance\_positions), instance\_positions);*

*offset += sizeof(instance\_positions);*

*glVertexAttribPointer(0, 4, GL\_FLOAT, GL\_FALSE, 0, 0);*

*glVertexAttribPointer(1, 4, GL\_FLOAT, GL\_FALSE, 0, (GLvoid\*)sizeof(square\_vertices));*

*glVertexAttribPointer(2, 4, GL\_FLOAT, GL\_FALSE, 0, (GLvoid\*)sizeof(square\_vertices) + sizeof(instance\_colors));*

*glEnableVertexAttribArray(0);*

*glEnableVertexAttribArray(1);*

*glEnableVertexAttribArray(2);*

*glVertexAttribDivisor(1, 1);*

*glVertexAttribDivisor(2, 1);*

*static const GLfloat black[] = {0.0f, 0.0f, 0.0f, 0.0f};*

*glClearBufferfv(GL\_COLOR, 0, black);*

*glUseProgram(program);*

*glDrawArraysInstanced(GL\_TRIANGLE\_FAN, 0, 4, 4);*

* 인스턴싱(간접 드로잉)
  + 직접 드로잉 방법은 지오메트리 인스턴스를 한 번 렌더링 할 때마다 드로잉 커맨드를 한 번씩 호출해야만 한다.
  + 간접 드로잉 방법은 렌더링 할 지오메트리 인스턴스의 커맨드 정보를 버퍼에 일괄 저장하여 이를 이용 드로잉 커맨드 한 번 호출로 렌더링한다.
  + glMultiDrawArraysIndirect(GLenum mode,

const void\* indirect,

GLsizei drawCount, // 지오메트리 인스턴스 개수.

GLsizei stride); // 0이면 촘촘히 패킹되어 있음.

glMultiDrawElementsIndirect(GLenum mode,

GLenum type,

const void\* indirect,

GLsizei drawCount,

GLsizei stride);

\* 위 커맨드는 여러 개를 일괄로 처리하지만 개별로 처리하는 방법도 있다.

glDrawArraysIndirect(GLenum mode, const void\* indirect);

glDrawElementsIndirect(GLenum mode, GLenum type, const void\* indirect);

* glDrawArraysIndirect() == glDrawArraysInstancedBaseInstance()
* glDrawElementsIndirect() == glDrawElementsInstancedBaseVertexBaseInstance()

*typedef struct*

*{*

*GLuint vertexCount;*

*GLuint instanceCount;*

*GLuint firstVertex;*

*GLuint baseInstance;*

*} DrawArraysIndirectCommand;*

***// OpenGL 에서 별도 구조체를 제공하지 않는다. 따라서 아래 요소를***

***// 직접 구성해서 사용해야 한다.***

*DrawArraysIndirectCommand draws[] =*

*{*

*{*

*42,* ***// 버텍스 수***

*1,* ***// 인스턴스 수***

*0,* ***// 첫 번째 버텍스***

*0,* ***// 베이스 인스턴스***

*},*

*{192, 1, 327, 0, },*

*{99, 1, 901, 0 }*

*};*

***// draws[] 를 버퍼 객체에 넣는다.***

*GLuint buffer;*

*glGenBuffers(1, &buffer);*

*glBindBuffer(GL\_DRAW\_INDIRECT\_BUFFER, buffer);*

*glBufferData(GL\_DRAW\_INDIRECT\_BUFFER, sizeof(draws), draws, GL\_STATIC\_DRAW);*

*glMultiDrawArraysIndirect(GL\_TRIGNGLES, NULL, sizeof(draws)/sizeof(draws[0]), 0);*

# 프리미티브 프로세싱

## 테셀레이션

버텍스 셰이더 단계에서 처리된 버텍스들을 패치 단위로(삼각형, 사각형, 선 등) 입력 받는다. 그리고 패치를 좀 더 세분화 시켜 디테일을 높일 수 있다.

테셀레이션 컨트롤 셰이더 -> 테셀레이션 엔진 -> 테셀레이션 이벨류에이션 셰이더

* 테셀레이션 컨트롤 셰이더
  + 테셀레이션의 레벨을 결정(세분화 정도 설정).
* 테셀레이션 엔진.
  + 설정된 레벨에 따라 점, 선, 삼각형 같은 작은 프리미티브로 분할.
* 테셀레이션 이벨류에이션 셰이더.
  + 버텍스 생성 옵션 설정.
  + 버텍스 쉐이더처럼 gl\_Position 에 값을 할당.

**사각형을 사용하여 테셀레이션 하기.**

***// 사각형 테셀레이션 컨트롤 셰이더.***

*#version 430 core.*

*layout (vertices = 4) out;* ***// 점 4개. 사각형.***

*void main()*

*{*

*if (gl\_InvocationID == 0)* ***// 첫 번째 버텍스 입력 시에만 레벨 설정.***

*{*

*gl\_TessLevelInner[0] = 9.0f;* ***// 안쪽 사각형 위, 아래 수평***

*gl\_TessLevelInner[1] = 7.0f;* ***// 안쪽 사각형 좌, 우 수직.***

*gl\_TessLevelOuter[0] = 3.0f;* ***// 바깥쪽 사각형 좌 수직.***

*gl\_TessLevelOuter[1] = 5.0f;* ***// 바깥쪽 사각형 아래 수평.***

*gl\_TessLevelOuter[2] = 3.0f;* ***// 바깥쪽 사각형 우 수직.***

*gl\_TessLevelOuter[3] = 5.0f;* ***// 바깥쪽 사각형 위 수평.***

*}*

*gl\_out[gl\_InvocationID].gl\_Position = gl\_in[gl\_Invocation].gl\_Position;*

*}*

***// 사각형 테셀레이션 이벨류에이션 셰이더.***

*#version 430 core*

*layout (quads) in;*

*void main()*

*{*

***// 테셀레이션 좌표의 x 요소를 사용하여 아래쪽 가장자리를 보간한다.***

*vec4 p1 = mix(gl\_in[0].gl\_Position, gl\_in[1].gl\_Position, gl\_TessCoord.x);*

***// 테셀레이션 좌표의 x 요소를 사용하여 위쪽 가장자리를 보간한다.***

*vec4 p2 = mix(gl\_in[2].gl\_Position, gl\_in[3].gl\_Position, gl\_TessCoord.x);*

***// 테셀레이션 좌표의 y 요소를 사용하여 두 결과를 보간한다.***

*gl\_Position = mix(p1, p2, gl\_TessCoord.y);*

*}*

*간단한 삼각형 테셀레이션 셰이더 예제*

***// 테셀레이션 컨트롤 셰이더.***

*#version 430 core*

*layout (vertices = 3) out;*

*void main()*

*{*

*if (gl\_Invocation == 0)*

*{*

*gl\_TessLevelInner[0] = 5.0;* ***// 안쪽 삼각형 세 변.***

*gl\_TessLevelOuter[0] = 8.0;* ***// 바깥쪽 삼각형 좌변.***

*gl\_TessLevelOuter[1] = 8.0;* ***// 바깥쪽 삼각형 아랫변.***

*gl\_TessLevelOuter[2] = 8.0;* ***// 바깥쪽 삼각형 우변.***

*}*

*gl\_out[gl\_invocationID].gl\_Position = gl\_in[gl\_InvocationID].gl\_Position;*

*}*

***// 테셀레이션 이벨류에이션 셰이더.***

*#version 430 core*

*layout (triangles) in;*

*void main()*

*{*

*gl\_Position = (gl\_TessCoord.x \* gl\_in[0].gl\_Position) +*

*(gl\_TessCoord.y \* gl\_in[1].gl\_Position) +*

*(gl\_TessCoord.z \* gl\_in[2].gl\_Position);*

*}*

## 지오메트리 셰이더

* EmitVertex() 함수를 통해 레스터라이제이션으로 보내는 버텍스를 명시적으로 생성할 수 있음.
* 프리미티브 모드 변경 가능.

**삼각형 프리미티브를 점으로 렌더링하는 간단한 지오메트리 셰이더.**

*#version 430 core*

*layout (triangles) in;* ***// 입력 : 삼각형.***

*layout (points, max\_vertices = 3) out;* ***// 출력 : 세 개 점.***

*void main()*

*{*

*for (int i = 0; i < gl\_in.length(); i++)* ***// 입력이 삼각형이므로 length 는 3.***

*{*

*gl\_Position = gl\_in[i].gl\_Position;* ***// 위치값 설정. 버텍스 셰이더와 유사.***

*EmitVertex();* ***// 지오메트리 셰이더 출력으로 점을 생성한다.***

*}*

*}*

* 입력 모드
  + 지오메트리 셰이더의 입력 프리미티브 모드가 points 라면 glDrawArrays() 를 호출할 때 반드시 GL\_POINTS 를 사용해야 한다.

|  |  |
| --- | --- |
| 지오메트리 셰이더 입력모드 | 허용되는 드로우 모드 |
| points | GL\_POINTS |
| lines | GL\_LINES, GL\_LINE\_LOOP, GL\_LINE\_STRIP |
| triangles | GL\_TRIANGLES, GL\_TRIANGLE\_FAN, GL\_TRIANGLE\_STRIP |
| lines\_adjacency | GL\_LINES\_ADJACENCY |
| triangles\_adjacency | GL\_TRIANGLES\_ADJACENCY |

* + 지오메트리 셰이더는 프리미티브 당 한 번씩 수행된다. GL\_POINTS 에 대해서는 점당 한 번씩, GL\_LINES, GL\_LINE\_STRIP, GL\_LINE\_LOOP 에 대해서는 선당 한 번씩, GL\_TRIANGLES, GL\_TRIANGLE\_STRIP, GL\_TRIANGLE\_FAN 에 대해서는 삼각형당 한 번씩 수행된다.
* gl\_in 구조체
  + 아래와 같이 기본으로 정의되어 있다.  
    in gl\_PerVertex  
    {  
     vec4 gl\_Position;  
     float gl\_PointSize;  
     float gl\_ClipDistance[];  
    } gl\_in[];  
    버텍스 셰이더의 output 블록으로 선언된 구조체와 동일한 요소를 가진다. 다른 점은 배열 형태인데 이는 지오메트리 셰이더는 프리미티브가 기본 처리 단위이기 때문이다.
* 지오메트리 조작.
  + EmitVertex() 함수를 호출할 때마다 새로운 버텍스를 생성한다. 호출하지 않으면 그 정점은 출력되지 않는다. 이를 이용하여 새로운 버텍스를 생성할 수도 폐기할 수도 있다.
  + 지오메트리 폐기하기 : 간단히 EmitVertex() 함수를 호출하지 않으면 된다.

**지오메트리 셰이더에서 지오메트리 폐기하기**

*#version 330*

*layout (triangles) in;*

*layout (triangle\_strip, max\_vertices = 3) out;*

*uniform vec3 viewpoint; // 카메라 시점.*

*uniform mat4 mv\_matrix; // 월드-뷰 행렬.*

*void main()*

*{*

*vec3 ab = gl\_in[1].gl\_Position.xyz – gl\_in[0].gl\_Position.xyz;*

*vec3 ac = gl\_in[2].gl\_Position.xyz – gl\_in[0].gl\_Position.xyz;*

*vec3 normal = normalize(cross(ab, ac)); // 삼각형 평명의 노말 벡터.*

*// 뷰공간 노말, 뷰 방향 벡터.*

*vec3 transformed\_normal = (vec4()normal, 0.0) \* mv\_matrix).xyz;*

*vec3 vt = normalize(gl\_in[0].gl\_Position.xyz - viewpoint);*

*float d = dot(vt, normal); // 내적.*

*// 내적 부호가 양수인 경우에만 프리미티브를 출력.*

*if (d > 0.0f)*

*{*

*for (inti = 0; i < 3; i++)*

*{*

*gl\_Position = gl\_in[i].gl\_Position;*

*EmitVertex();*

*}*

*}*

*EndPrimitive();*

*}*

* + 지오메트리 수정하기.  
    버텍스 셰이더와 달리 지오메트리 셰이더는 처리 단위가 프리미티브이기 때문에 공유되는 버텍스가 있다. 따라서 만약 삼각형 단위 처리 모드에서 버텍스 3개를 위치 이동시키면 삼각형이 이동되는 결과를 볼 수 있다. 삼각형들이 분리될 수 있는 것이다.

**지오메트리 셰이더에서 지오메트리 수정하기.**

*#version 330*

*layout (triangles) in;*

*layout (triangle\_strip, max\_vertices = 3) out;*

*void main()*

*{*

*for (inti = 0; i < 3; i++)*

*{*

*gl\_Position = gl\_in[i].gl\_Position + vec4(explode\_factor \* normal, 0.0);*

*EmitVertex();*

*}*

*}*

* + 지오메트리 생성하기.  
    EmitVertex() 를 여러 번 호출하면 지오메트리를 더 생성할 수 있다.

# 프래그먼트 프로세싱

## 프래그먼트(픽셀) 셰이더

## 가위 테스트

레스터라이제이션으로 생성되는 프래그먼트(픽셀) 들에 대해 특정 사각 영역에서만 출력되도록 한다.

* 클리핑 함수
  + glScissorIndexed(GLuint index // 대상 뷰포트 영역 내에서.

GLint left, // 해당 영역만 출력.

GLint bottom,

GLint width,

GLint height);

* + glScissorIndexedv(GLuint index,

const GLint\* v); // 영역을 배열로 전달.

* 활성화 함수
  + glEnable(GL\_SCISSOR\_TEST), glDisable(SL\_SCISSOR\_TEST).

## 스텐실 테스트

화면에 지오메트리가 출력되기 위해서는 프레임 버퍼에 해당 색상의 값이 채워져야 한다. 그런데 조건이 하나 더 있다. 스텐실 테스트가 활성화 되어 있을 경우, 프레임 버퍼의 특정 위치와 동일한 위치에 있는 스텐실 버퍼값과의 특정 연산 결과가 통과해야만 한다.

* 스텐실 버퍼 사용.
* 활성화 함수 : glEnable(GL\_STENCIL\_TEST).
* 비활성화 함수 :glDisable(GL\_STENCIL\_TEST).
* 스텐실 테스트가 성공하는지 실패하는지에 대한 조건 제어 함수.
  + void glStencilFuncSeparate(GLenum face, // 전면, 후면.

GLenum func, // 통과 조건.

GLint ref, // 비교값(참조값)

GLuint mask); // 버퍼값(스텐실 버퍼) 마스크.

* + - face : 지오메트리의 방향을 지정하여(GL\_FRONT, GL\_BACK, GL\_FRONT\_AND\_BACK) 어떤 지오메트리가 적용될 지 알려준다.
    - func

GL\_NEVER : 테스트를 절대 통과 못함.

GL\_ALWAYS : 테스트를 항상 통과함.

GL\_LESS : 비교값이 버퍼값보다 작은 경우 통과.

GL\_LEQUAL : 비교값이 버퍼값보다 작거나 같은 경우 통과.

GL\_EQUAL : 비교값이 버퍼값과 같은 경우 통과.

GL\_GEQUAL : 비교값이 버퍼값보다 크거나 같은 경우 통과.

GL\_GREATER : 비교값이 버퍼값보다 큰 경우 통과.

GL\_NOTEQUAL : 비교값이 버퍼값과 다른 경우 통과.

* + - ref, mask  
      다음과 같은 의사코드를 살펴보면 ref 와 mask 의 역할을 쉽게 알 수 있다.

GLuint currStencilVal = GetCurrentStencilValue(x, y);

if (compare(currStencilVal & mask,

ref & mask,

front\_facing ? front\_op : back\_op))

{

passed = true;

}

else

{

passed = false;

}

* 스텐실 테스트 결과(통과 또는 실패) 후 수행되는 작업.
  + glStencilOpSeparate(GLenum face, // 전면, 후면, 양면.

GLenum sfail, // 스텐실 버퍼 실패 시 작업.

GLenum dppass, // 깊이테스트 실패 시 작업.

GLenum dppass); // 깊이테스트 성공 시 작업.

* + - 두 번째 ~ 네 번째 연산.  
      GL\_KEEP : 스텐실 버퍼를 수정하지 않는다.

GL\_ZERO : 스텐실 버퍼값을 0으로 설정한다.

GL\_REPLACE : 스텐실값을 참조값으로 교체한다.

GL\_INCR : 스텐실값을 한계 조정하여 증가시킨다.

GL\_DECR : 스텐실값을 한계 조정하여 감소시킨다.

GL\_INVERT : 스텐실값을 비트 단위로 반전시킨다

GL\_INCR\_WRAP : 스텐실값을 한계 조정 없이 증가시킨다.

GL\_DECR\_WRAP : 스텐실값을 한계 조정 없이 감소시킨다.

***// 스텐실 버퍼를 0으로 지운다.***

*const GLint zero;*

*glClearBufferiv(GL\_STENCIL, 0, &zero);*

***// 경계 렌더링을 위해 스텐실 상태를 설정.***

*glStencilFunSeparate(GL\_FRONT, GL\_ALWAYS, 1, 0xff); // ref 값 1, 항상 통과.*

***// 깊이테스트를 통과하면 ref 값 1을 스텐실 버퍼에 씌운다. 통과 실패 시에는 0을 스텐실***

***// 버퍼에 씌운다.***

*glStencilOpSeparate(GL\_FRONT, GL\_KEEP, GL\_ZERO, GL\_REPLACE);*

***// 경계 이미지를 그린다.***

*…*

***// 이제 경계 이미지 픽셀들은 스텐실값 1을 갖고,***

***// 다른 모든 픽셀을 스텐실값 0을 갖는다.***

***// 일반 렌더링을 위한 스텐실 상태를 설정.***

***// 픽셀이 경계를 덮어쓰는 경우에는 실패한다.***

***// 1보다 작은 스텐실 값을 가지는 픽셀 위치는 통과. 스텐실 버퍼값이 0일 경우 출력.***

*glStencilFuncSeparate(GL\_FRONT\_AND\_BACK, GL\_LESS, 1, 0xff);*

*glStencilOpSeparate(GL\_FRONT, GL\_KEEP, GL\_KEEP, GL\_KEEP);*

***// 장면의 나머지를 렌더링한다. 스텐실 경계 위에는 렌더링 되지 않을 것이다.***

## 깊이 테스트

깊이 테스트가 활성화되면 프래그먼트의 깊이 값을 기존 깊이버퍼 값과 비교한다. 만약 깊이값 쓰기 기능이 활성화되어 있고, 프래그먼트가 깊이 테스트를 통과하면, 깊이 버퍼는 프래그먼트의 깊이값으로 갱신된다. 만약 깊이 테스트가 실패하면 프래그먼트는 폐기되고 다음 프래그먼트 연산으로 전달되지 않는다.

* 깊이 비교 함수

GL\_ALWAYS : 깊이 테스트가 항상 통과한다.

GL\_NEVER : 깊이 테스트가 절대 통과하지 않는다.

GL\_LESS : 통과(깊이값 < 기존값)

GL\_LEQUAL : 통과(깊이값 <= 기존값)

GL\_EQUAL : 통과(깊이값 == 기존값).

GL\_NOTEQUAL : 통과(깊이값 != 기존값)

GL\_GREATER : 통과(깊이값 > 기존값)

GL\_GEQUAL : 통과(깊이값 => 기존값).

* 활성화 함수 : glEnable(GL\_DEPTH\_TEST).
* 비활성화 함수 : glDisable(GL\_DEPTH\_TEST)

\* 비활성화 하면 깊이 테스트가 항상 통과하는 것(깊이 함수를 GL\_ALWAYS 로 설정한 경우)과 같다.

* 깊이 버퍼 쓰기 여부 설정
  + 활성화 : glDepthMask(GL\_TRUE).
  + 비활성화 : glDepthMask(GL\_FALSE).
  + 기본 설정 : 깊이 테스트를 활성화 하면 깊이 버퍼 쓰기가 기본으로 활성화되어 있다.

## 블렌딩

프레임 버퍼에 이미 그려져 있는 프래그먼트와 입력으로 들어오는 프래그먼트의 블렌딩 비율을 지정하여 색상을 혼합하여 출력한다. 투명도 효과가 핵심이다.

* 블렌딩 함수
  + glBlendFunc(GLenum src, // 깊이테스트까지 통과한 프래그먼트.

GLenum dst); // 이미 그려진 프레임 버퍼 프래그먼트.

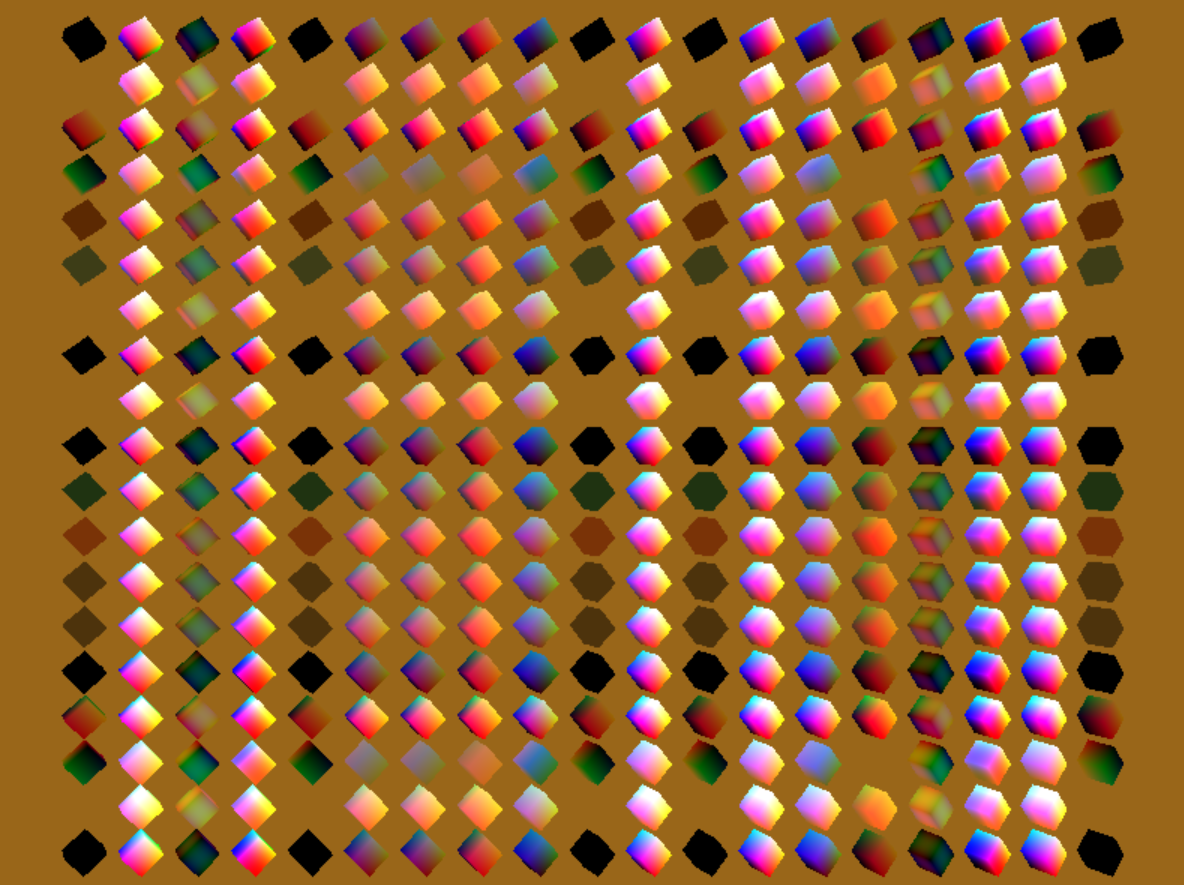
결과 색상 = 입력 프래그먼트 \* src비율 + 프레임버퍼 프래그먼트 \* dst 비율

* + 반투명 예시

glBlendFunc(GL\_SRC\_ALPHA, GL\_ONE\_MINUS\_SRC\_ALPHA);

입력 색상은 입력 프래그먼트의 알파값을 적용한다. 만약 알파값이 0.5 이면 R,G,B 는 각각 50%만 적용된다. 이미 그려진 프레임 버퍼의 프래그먼트 색상값은 1 – 입력 알파값의 비율로 적용된다. 각각 계산된 색상을 최종적으로 더한다.

* + 모든 블렌딩 함수를 사용하여 렌더링하기.



*static const GLfloat orange[] = {0.6f, 0.4f, 0.1f, 1.0f};*

*glClearBufferv(GL\_COLOR, 0, orange);*

***// 블렌딩 함수들.***

*static const GLenum blend\_func[] =*

*{*

*GL\_ZERO,*

*GL\_ONE,*

*GL\_SRC\_COLOR,* ***// 첫 번째 원본 색상.***

*GL\_ONE\_MINUS\_SRC\_COLOR,*

*GL\_DST\_COLOR,* ***// 대상 색상.***

*GL\_ONE\_MINUS\_DST\_COLOR,*

*GL\_SRC\_ALPHA,*

*GL\_ONE\_MINUS\_SRC\_ALPHA,*

*GL\_DST\_ALPHA,*

*GL\_ONE\_MINUS\_DST\_ALPHA,*

*GL\_CONSTANT\_COLOR,* ***// 상수 블렌딩 색상. glBlendColor() 로 설정.***

*GL\_ONE\_MINUS\_CONSTANT\_COLOR,*

*GL\_CONSTANT\_ALPHA,*

*GL\_ONE\_MINUS\_CONSTANT\_ALPHA,*

*GL\_SRC\_ALPHA\_SATURATE,*

*GL\_SRC1\_COLOR,* ***// 두 번째 원본 색상.***

*GL\_ONE\_MINUS\_SRC1\_COLOR,*

*GL\_SRC1\_ALPHA,*

*GL\_ONE\_MINUS\_SRC1\_ALPHA*

*};*

*static const int num\_blend\_funcs = sizeof(blend\_func) / sizeof(blend\_func[0]);*

*static const float x\_scale = 20.0f / float(num\_blend\_funcs);*

*static const float y\_scale = 16.0f / float(num\_blend\_funcs);*

*const float t = (float)currentTime;*

*glEnable(GL\_BLEND);*

*glBlendColor(0.2f, 0.5f, 0.7f, 0.5f);* ***// 상수 블렌딩 색상을 연한 파란색으로.***

*for (j = 0; j < num\_blend\_funcs; j++)*

*{*

*for (i = 0; i < num\_blend\_funcs; i++)*

*{*

*vmath::mat4 mv\_matrix =*

*vmath::translate(9.5f – x\_scale \* float(i),*

*7.5f – y\_scale \* float(j)*

*- 50.0f) \**

*vmath::rotate(t \* -45.0f, 0.0f, 1.0f, 0.0f) \**

*vmath::rotate(t \* -21.0f, 1.0f, 0.0f, 0.0f);*

*glUniformMatrix4fv(mv\_location, 1, GL\_FALSE, mv\_matrix);*

*glBlendFunc(blend\_func[i], blend\_func[j]);*

*glDrawElements(GL\_TRIANGLES, 36, GL\_UNSIGNED\_SHORT, 0);*

*}*

*}*

* 블렌딩 공식
  + 위 블렌딩 함수 호출 예제에서는 입력 색상과 대상 색상의 블렌딩 결과값은 덧셈으로 도출되었다. 하지만 덧셈 외에도 다른 공식들을 적용할 수 있다.
  + glBlendEquation(GLenum mode);

GL\_FUNC\_ADD : 입력 + 대상.

GL\_FUNC\_SUBTRACT : 입력 – 대상.

GL\_FUNC\_REVERSE\_SUBTRACT : 대상 – 입력.

GL\_MIN : min(입력, 대상).

GL\_MAX : max(입력, 대상).

* 논리 연산 : 일반적으로 사용하지 않음.
* 색상 마스킹 : 일단 패스.

## 오프스크린 렌더링 & 렌더타겟

일반적으로 프래그먼트 셰이더의 출력은 백버퍼가 대상이 되고 최종적으로 사용자에게 보여진다. 이와는 별도로 프래그먼트 셰이더의 출력을 텍스처에 직접 출력하게 할 수도 있다. 깊이 버퍼를 이용한 그림자 출력이나 포스트 효과 등에 유용하게 사용될 수 있다.

* 프레임 버퍼 객체(fbo, frame buffer object).
  + 텍스처를 어태치시켜 렌더링용 저장 공간으로 사용하기. 아래 순서를 따른다.
    - 프레임버퍼 객체 생성 : glGenFramebuffers();
    - 프레임버퍼 바인딩 : glBindFramebuffer();
    - 컨텍스트에 어태치

glFramebufferTexture(GLenum target,

GLenum attachment,

GLenum texture,

GLenum level);

. target : 텍스처에 어태치시킬 프레임버퍼 객체의 바인딩 포인트. GL\_FRAMEBUFFER(= GLDRAW\_FRAMEBUFFER), GL\_READ\_FRAMEBUFFER.

. attachment : 텍스처를 어떤 용도로 사용할 것인가(색상, 깊이, 스텐실). GL\_COLOR\_ATTACHMENT(GL\_COLOR\_ATTACHMENT0, GL\_COLOR\_ATTACHMENT1 …), GL\_DEPTH\_ATTACHMENT, GL\_STENCIL\_ATTACHMENT.

. level : 텍스처 밉맵 레벨.

* 1개 텍스처에 렌더링.
  + 아래 코드는 깊이 버퍼를 갖는 프레임버퍼 객체와 렌더링할 텍스처를 설정하는 초기화 코드이다.

// 프레임버퍼 객체를 생성하고 바인딩.

glGenFramebuffers(1, &fbo);

glBindFramebuffer(GL\_FRAMEBUFFER, fbo);

// 색상 버퍼에 대한 텍스처를 설정.

glGenTextures(1, &color\_texture);

glBindTexture(GL\_TEXTURE\_2D, color\_texture);

glTexStorage2D(GL\_TEXTURE\_2D, 1, GL\_RGBA8, 512, 512);

// 밉맵 비활성화.

glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_MIN\_FILTER, GL\_LINEAR);

glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_MAG\_FILTER, GL\_LINEAR);

// FBO 의 깊이 버퍼가 될 텍스처를 생성.

glGenTextures(j1, depth\_texture);

glBindTexture(GL\_TEXTURE\_2D, depth\_texture);

glTexStorage2D(GL\_TEXTURE\_2D, 1, GL\_DEPTH\_COMPONENT32F, 512, 512);

// 색상 및 깊이 텍스처를 FBO 에 어태치.

glFramebufferTexture(GL\_FRAMEBUFFER, GL\_COLOR\_ATTACHMENT0, color\_texture, 0);

glFramebufferTexture(GL\_FRAMEBUFFER, GL\_DEPTH\_ATTACHMENT, depth\_texture, 0);

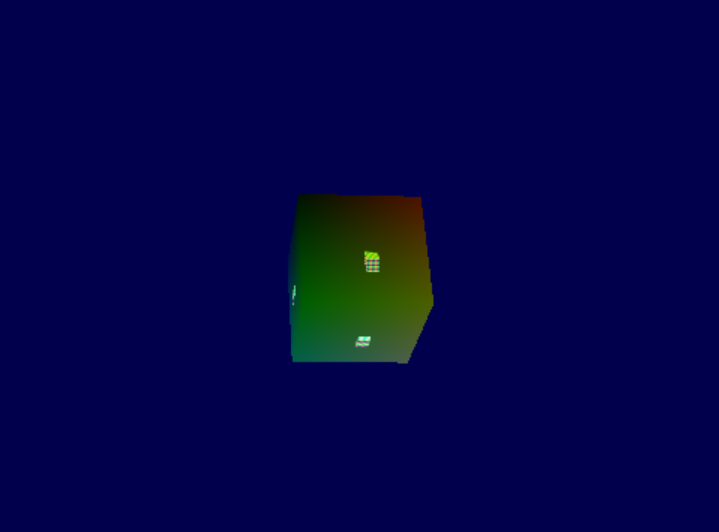
// 프레임버퍼의 색상 어태치먼트에 그릴 것이라고 OpenGL 에 알린다.

static const GLenum draw\_buffers[] = { GL\_COLOR\_ATTACHMENT0 };

glDrawBuffers(1, draw\_buffers);

* + 실제로 텍스처에 렌더링 해보자. 아래는 큐브를 텍스처에 렌더링하고 그

결과를 프레임버퍼의 큐브 각 면에 렌더링하는 예이다.



***// 사용자 정의 fbo 를 바인딩.***

*glBindFramebuffer(GL\_FRAMEBUFFER, fbo);*

***// 뷰포트를 설정하고 깊이 및 색상 버퍼를 지운다.***

*glViewport(0, 0, 512, 512);*

*glClearBufferfv(GL\_COLOR, 0, green);*

*glClearBufferfv(GL\_DEPTH, 0, &one);*

***// 첫 번째 비텍스처 프로그램을 활성화.***

*glUseProgram(program1);*

***// 유니폼을 설정하고 큐브를 텍스처에 그린다.***

*glUniformMatrix4fv(proj\_location, 1, GL\_FALSE, proj\_matrix);*

*glUniformMatrix4fv(mv\_location, 1, GL\_FALSE, mv\_matrix);*

*glDrawArrays(GL\_TRIANGLES, 0, 36);*

***// 기본 프레임 버퍼로 돌아간다.***

*glBindFramebuffer(GL\_FRAMEBUFFER, 0);*

***// 뷰포트를 윈도우 넓이와 높이로 재설정하고 깊이 및 색상 버퍼를 지운다.***

*glViewport(0, 0, info.windowWidth, info.windowHeight);*

*glClearBufferfv(GL\_COLOR, 0, blue);*

*glClearBufferfv(GL\_DEPTH, 0, &one);*

***// 렌더링한 텍스처를 셰이더에 읽기용으로 바인딩.***

*glBindTexture(GL\_TEXTURE\_2D, color\_texture);*

***// 텍스처를 읽어 큐브 면에 출력할 셰이더 프로그램을 활성화.***

*glUseProgram(program2);*

***// 유니폼을 설정하고 그린다.***

*glUniformMatrix4fv(proj\_location2, 1, GL\_FALSE, proj\_matrix);*

*glUniformMatrix4fv(mv\_location2, 1, GL\_FALSE, mv\_matrix);*

*glDrawArrays(GL\_TRIANGLES, 0, 36);*

***// 텍스처를 언바인드.***

*glBindTexture(GL\_TEXTURE\_2D, 0);*

* 여러 개의 단일 텍스처에 렌더링.

여러 개의 텍스처를 프레임 버퍼로 렌더링하게 설정할 수도 있다. GL\_COLOR\_ATTACHMENT0, GL\_COLOR\_ATTACHMENT1 등 attachment 인자로 간단히 설정할 수 있다.

*static const GLenum draw\_buffers[] =*

*{*

*GL\_COLOR\_ATTACHMENT0,*

*GL\_COLOR\_ATTACHMENT1,*

*GL\_COLOR\_ATTACHMENT2*

*};*

***// 프레임버퍼 객체를 생성하고 바인딩.***

*glGenFramebuffers(1, &fbo);*

*glBindFramebuffer(GL\_FRAMEBUFFER, fbo);*

***// 세 개의 텍스처 생성.***

*glGenTextures(3, &color\_texture[0]);*

*for (int i = 0; i < 3; i++)*

*{*

***// 바인딩하고 저장 공간을 할당.***

*glBindTexture(GL\_TEXTURE\_2D, color\_texture[i]);*

*glTexStorage2D(GL\_TEXTURE\_2D, 9, GL\_RGBA8, 512, 512);*

***// 기본 필터 인자들을 설정.***

*glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_MIN\_FILTER, GL\_LINEAR);*

*glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_MAG\_FILTER, GL\_LINEAR);*

***// 프레임버퍼 객체에 색상 어태치먼트로 어태치.***

*glFramebufferTexture(GL\_FRAMEBUFFER, draw\_buffers[i], color\_texture[i], 0);*

*}*

***// 깊이 텍스처를 생성한다.***

*glBindTexture(GL\_TEXTURE\_2D, depth\_texture);*

*glTexStorage2D(GL\_TEXTURE\_2D, 9, GL\_DEPTH\_COMPONENT32F, 512, 512);*

***// 깊이 텍스처를 프레임버퍼에 어태치.***

*glFramebufferTexture(GL\_FRAMEBUFFER, GL\_DEPTH\_ATTACHMENT, depth\_texture, 0);*

***// fbo 에 대한 드로우 버퍼를 색상 어태치먼트에 대한 포인터로 설정.***

*glDrawBuffers(3, draw\_buffers);*

* 2D 배열 텍스처에 렌더링.
  + 기본적인 설정은 1 개 텍스처에 렌더링할 때와 비슷하다. 다른 점은 텍스처 바인딩 시 GL\_TEXTURE\_2D\_ARRAY 를 사용한다는 점이다.

***// 색상 어태치먼트에 대한 텍스처를 생성하고, 바인딩하고, 저장 공간을 할당.***

***// 512 \* 512 size 16개.***

*GLuint color\_attachment;*

*glGenTextures(1, &color\_attachment);*

*glBindTexture(GL\_TEXTURE\_2D\_ARRAY, color\_attachment);*

*glTexStorage3D(GL\_TEXTURE\_2D\_ARRAY, 1, GL\_RGBA8, 512, 512, 16);*

***// 동일한 작업을 깊이 버퍼 어태치먼트에 대해 수행.***

*GLuint depth\_attachment;*

*glGenTextures(1, &depth\_attachment);*

*glBindTexture(GL\_TEXTURE\_2D\_ARRAY, depth\_attachment);*

*glTexStorage3D(GL\_TEXTURE\_2D\_ARRAY, 1, GL\_DEPTH\_COMPONENT, 512, 512, 16);*

***// 프레임버퍼 객체를 생성하고 그곳에 텍스처를 바인딩.***

*GLuint fbo;*

*glGenFramebuffers(1, &fbo);*

*glBindFramebuffer(GL\_FRAMEBUFFER, fbo);*

*glFramebufferTexture(GL\_FRAMEBUFFER, GL\_COLOR\_ATTACHMENT0, color\_attachment, 0);*

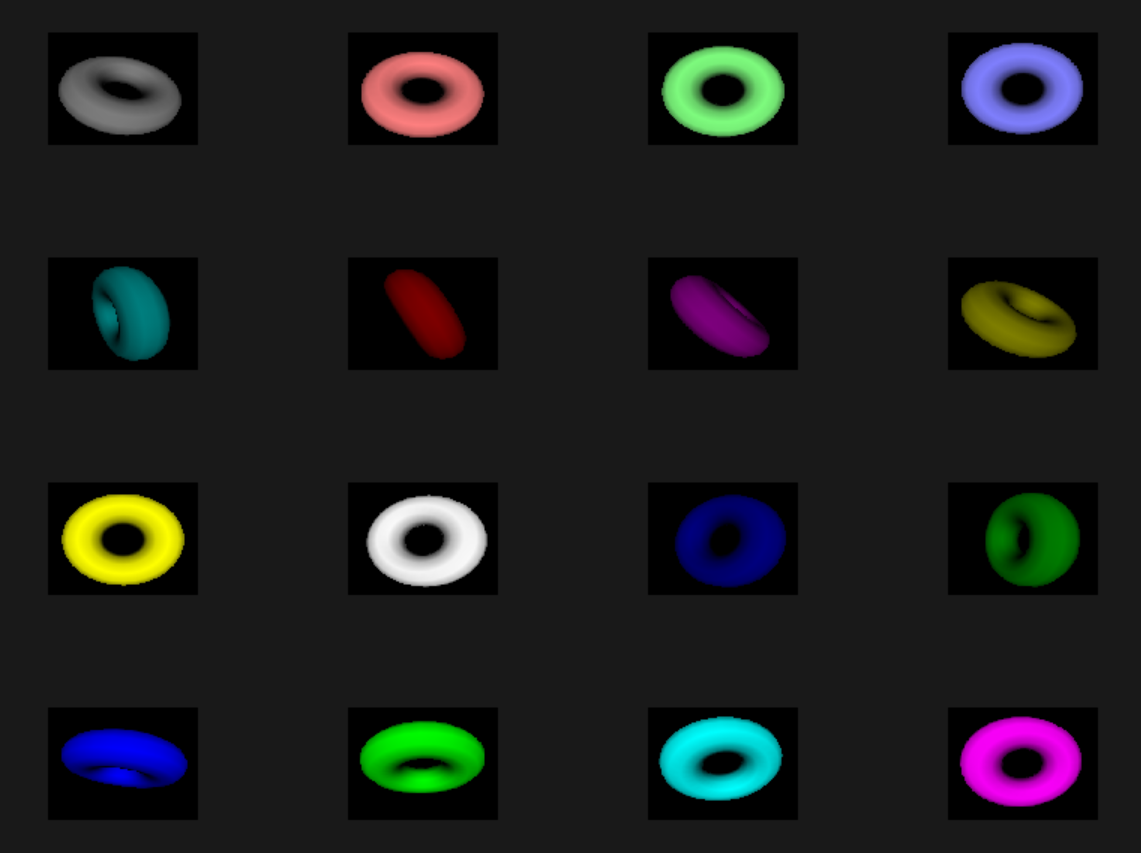
*glFramebufferTexture(GL\_FRAMEBUFFER, GL\_DEPTH\_ATTACHMENT, depth\_attachment, 0);*

***// 색상 어태치먼트로 렌더링할 것이라고 OpenGL 에 알린다.***

*static const GLuint draw\_buffers[] = { GL\_COLOR\_ATTACHMENT0 };*

*glDrawBuffers(1, draw\_buffers);*

* + 레이어 렌더링.



위와 같이 16개의 텍스처에 각각 렌더링하기 위해서는 지오메트리 셰이더를 이용해야 한다. 지오메트리 셰이더의 gl\_Layer 변수에 인덱스를 할당하면 배열 텍스처 중 해당 인덱스의 텍스처에 렌더링 할 수 있다. 만약 지오메트리 셰이더를 이용하지 않으면 0번 텍스처에만 항상 렌더링된다. 다음은 지오메트리 셰이더의 일부 내용이다.

*#version 430 core*

***// 지오메트리 셰이더 16번 호출.***

***// 삼각형 입력과 삼각형 출력 사용.***

*layout (invocations = 16, triangles) in;*

*layout (triangle\_strip, max\_vertices = 3) out;*

*in VS\_OUT*

*{*

*vec4 color;*

*vec3 normal;*

*} gs\_in[];*

*out GS\_OUT*

*{*

*vec4 color;*

*vec3 normal;*

*} gs\_out;*

*layout (binding = 0) uniform BLOCK*

*{*

*mat4 proj\_matrix;*

*mat4 mv\_matrix[16];*

*};*

*void main()*

*{*

*for (int i = 0; i < gl\_in.length(); i++)*

*{*

***// 모든 지오메트리를 그대로 통과시킨다.***

*gs\_out.color = colors[gl\_InvoationID];*

*gs\_out.normal = mat3(mv\_matrix[gl\_InvocationID]) \* gs\_in[i].normal;*

*gl\_Position = proj\_matrix \* mv\_matrix[gl\_InvocationID] \* gl\_in[i].gl\_Position;*

***// 직접 렌더링한다.***

*gl\_Layer = gl\_InvocationID; // 0 ~ 15. 해당 번호 레이어로 렌더링.*

*EmitVertex();*

*}*

*EndPrimitive();*

*}*

glFramebufferTextureLayer() 함수를 사용하면 지오메트리 셰이더를 사용하지 않고도 배열 텍스처에 렌더링할 수 있다. 3D 텍스처도 렌더링 가능하다.

*GLuint tex;*

*glGenTextures(1, &tex);*

*glBindTexture(GL\_TEXTURE\_2D\_ARRAY, tex);*

*glTexStorage3D(GL\_TEXTURE\_2D\_ARRAY, 1, GL\_RGBA8, 256, 256, 8);*

*GLuint fbo;*

*glGenFramebuffers(1, &fbo);*

*glBindFramebuffer(GL\_FRAMEBUFFER, fbo);*

*for (int i = 0; i < 8; i++)*

*{*

*glFramebufferTextureLayer(GL\_FRAMEBUFFER,*

*GL\_COLOR-ATTACHMENT0 + i,*

*tex,*

*0,*

*i);*

*}*

*static const GLenum draw\_buffers[] =*

*{*

*GL\_COLOR\_ATTACHMENT0, GL\_COLOR\_ATTACHMENT1,*

*GL\_COLOR\_ATTACHMENT2, GL\_COLOR\_ATTACHMENT3,*

*GL\_COLOR\_ATTACHMENT4, GL\_COLOR\_ATTACHMENT5,*

*GL\_COLOR\_ATTACHMENT6, GL\_COLOR\_ATTACHMENT7*

*};*

***// 프래그먼트 셰이더는 8 개까지 출력을 가질 수 있다.***

*glDrawBuffers(8, &draw\_buffers[0]);*

* 큐브 맵에 렌더링.

6개의 면을 갖는 큐브 맵 텍스처를 생성하여 각각의 면에 렌더링한다. 렌더링 방식은 다음 두 가지가 있다.

* + 지오메트리 셰이더에서 렌더링.

gl\_Layer 에 인덱스를 쓰면 큐브 맵의 해당 면에 렌더링할 수 있다.

gl\_Layer Index 렌더링 면

0 양의 x 면

1 음의 x면

2 양의 y 면

3 음의 y면

4 양의 z면

5 음의 z면

* + glFramebufferTexture2D() 함수 사용.

glFramebufferTexture2D(GLenum target,

GLenum attachment,

GLenum textarget, // 어떤 면에 렌더링할지 결정.

GLuint texture,

GLint level);

textarget 에 GL\_CUBE\_MAP\_POSITIVE\_X 로 설정하면 큐브맵의 양의 X면에 렌더링한다. GL\_CUBE\_MAP\_NEGATIVE\_Y 로 설정하면 양의 Y면에 렌더링한다.

## 프레임 버퍼

* 부동소수점 프레임버퍼
* 정수 프레임버퍼
* sRGB 색상 공간

## 포인트 스프라이트

# 컴퓨트 셰이더