Advanced OpenGL (3)

GPU Programming

2022학년도
2학기

ⓒ nvidia

Advanced Data

Advanced Data

- Buffer들을 다루는 좀 더 흥미로운 방법 및
  대량의 데이터를 텍스처 객체로 shader에 전달하는 방법 소개

- OpenGL의 buffer
  - OpenGL은 특정 메모리를 특정 buffer target에 바인딩함으로써 의미를 부여
    (예: GL_ARRAY_BUFFER에 바인딩하면 이 buffer는 vertex array buffer가 됨)
  - OpenGL은 내부적으로 target에 대해 buffer를 저장하고 이 buffer들을 따로 처리

- glBufferData() 함수
  - Buffer 객체에 의해 관리되는 메모리를 할당해주고 여기에 데이터를 삽입
  - 이 함수의 data 파라미터에 NULL 값을 넣으면, 이 함수는 메모리 할당만 해주고 데이터를 채워넣지 않음
    (off-screen rendering시의 glTexture2D() 함수 사용법과 동일)
  - 위 방식은 특정 메모리를 reserve(예약)한 후 나중에 buffer를 채우려 할 때 유용

Advanced Data

- glBufferSubData() 함수
  - 전체 buffer를 채우는 것 대신 buffer의 특정 부분만 채우고자 할 때 사용
  - 파라미터: buffer target, offset, 데이터의 크기, 실제 데이터
  - Offset은 buffer를 어디에서부터 채울지를 지정 → 메모리의 특정 부분에만 삽입/수정을 가능하게 함
  - glBufferSubData() 함수 호출 전 glBufferData() 함수를 꼭 호출하여, 충분한 메모리를 할당해야 함

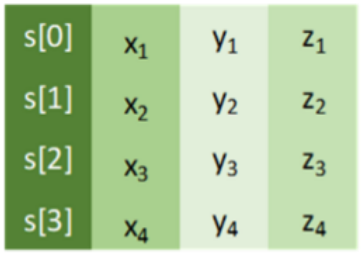
```
glBufferSubData(GL_ARRAY_BUFFER, 24, sizeof(data), &data); // Range: [24, 24 + sizeof(data)]
```

- Pointer에게 buffer의 메모리를 요청하고 데이터를 여기에 직접 복사하는 방법
  - glMapBuffer() 함수는 현재 바인딩된 buffer의 메모리를 가리키는 포인터를 반환
  - glUnmapBuffer() 함수는 이 포인터를 무효화
  - 임시메모리 없이 직접 데이터를 buffer에 매핑시 유용

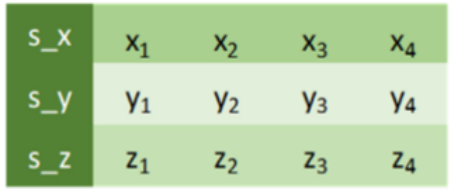
```
float data[] = {
  0.5f, 1.0f, -0.35f
  [...]
};
glBindBuffer(GL_ARRAY_BUFFER, buffer);
// get pointer
void *ptr = glMapBuffer(GL_ARRAY_BUFFER, GL_WRITE_ONLY);
// now copy data into memory
memcpy(ptr, data, sizeof(data));
// make sure to tell OpenGL we're done with the pointer
glUnmapBuffer(GL_ARRAY_BUFFER);
```

Batching Vertex Attributes

- 기존 glVertexAttribPointer() 함수의 interleaving 방법
  - Vertex buffer object (VBO)를 구성하는 속성의 layout을 지정
  - 속성의 유형 - 각 vertex의 위치, normal, 텍스처 좌표 등
  - 각각의 속성은 interleave됨 (123123123)
    → 이러한 저장 방법을 AoS (Array of Structures)라고 함
- Structure of Arrays (SoA)로 저장된 model 파일의 경우?
  - 위 interleaving 방법을 사용한다고 가정한다면,
    각 속성별로 데이터를 읽어 들인 다음,
    이를 다시 AOS 형태로 바꾸어야 함 → 비효율적

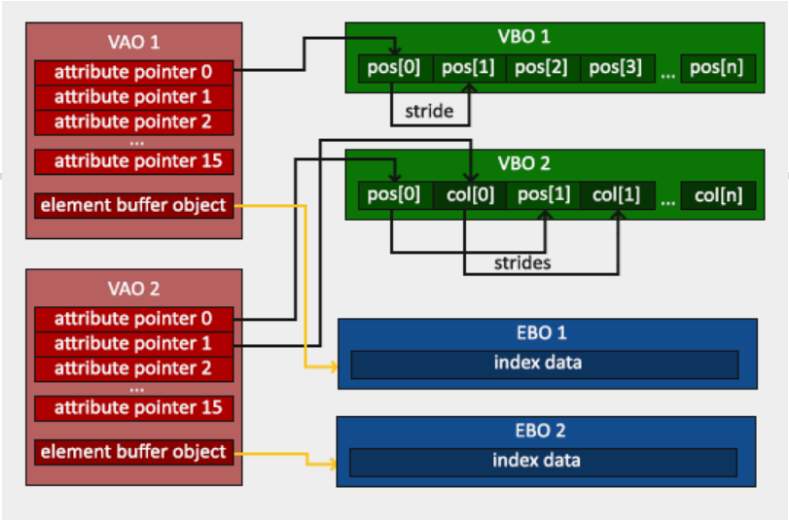


Array of Structures
(AoS)



Structure of Arrays
(SoA)

Machine Learning Frameworks Interoperability, Part 1: Memory Layouts and Memory Pools | NVIDIA Technical Blog




```
4   o cube
5   mtllib cube.mtl
6
7   v -0.500000 -0.500000 0.500000
8   v 0.500000 -0.500000 0.500000
9   v -0.500000 0.500000 0.500000
10  v 0.500000 0.500000 0.500000
11  v -0.500000 0.500000 -0.500000
12  v 0.500000 0.500000 -0.500000
13  v -0.500000 -0.500000 -0.500000
14  v 0.500000 -0.500000 -0.500000
15
16  vt 0.000000 0.000000
17  vt 1.000000 0.000000
18  vt 0.000000 1.000000
19  vt 1.000000 1.000000
20
21  vn 0.000000 0.000000 1.000000
22  vn 0.000000 1.000000 0.000000
23  vn 0.000000 0.000000 -1.000000
24  vn 0.000000 -1.000000 0.000000
25  vn 1.000000 0.000000 0.000000
26  vn -1.000000 0.000000 0.000000
```

Batching Vertex Attributes

- 대안 – Attribute type당 하나의 큰 chunk를 사용
  - 123123123123 → 111122223333
  - 텍스처 좌표의 배열을 읽을 경우 glBufferSubData()를 통해 batching (일괄 처리) 가능

```
float positions[] = { ... };
float normals[] = { ... };
float tex[] = { ... };
// fill buffer
glBufferSubData(GL_ARRAY_BUFFER, 0, sizeof(positions), &positions);
glBufferSubData(GL_ARRAY_BUFFER, sizeof(positions), sizeof(normals), &normals);
glBufferSubData(GL_ARRAY_BUFFER, sizeof(positions) + sizeof(normals), sizeof(tex), &tex);
```

  - 이후 glVertexAttribPointer() 호출시 이러한 SoA 형식을 반영. Stride는 vertex attribute의 크기

```
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 3 * sizeof(float), 0);
glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, 3 * sizeof(float), (void*)(sizeof(positions)));
glVertexAttribPointer(
  2, 2, GL_FLOAT, GL_FALSE, 2 * sizeof(float), (void*)(sizeof(positions) + sizeof(normals)));
```

  - 장점: 데이터의 로딩 시간이 줄어들 수 있음
  - 단점: VS에서 읽어들이는 데이터가 메모리 상에 aligned 되지 않을 수 있음 → 메모리 액세스가 비효율적

Copying Buffers

- glCopyBufferSubData() 함수
  - 한 buffer에서 다른 buffer로 데이터를 복사할 수 있게 해 줌

```
void glCopyBufferSubData(GLenum readtarget, GLenum writetarget, GLintptr readoffset,
                         GLintptr writeoffset, GLsizeiptr size);
```

  - readtarget, writetarget - 각각 읽고 쓸 buffer target을 지정. 현재 바인딩된 버퍼가 영향을 받음 (예: VERTEX_ARRAY_BUFFER에서 읽어서 VERTEX_ELEMENT_ARRAY_BUFFER로 복사)
  - readoffset, writeoffset- 각 버퍼의 offset (시작 지점)
  - GL_COPY_READ_BUFFER 및/또는 GL_COPY_WRITE_BUFFER에 VBO를 바인딩한 후 사용하는 방법도 가능

```
glBindBuffer(GL_COPY_READ_BUFFER, vbo1);
glBindBuffer(GL_COPY_WRITE_BUFFER, vbo2);
glCopyBufferSubData(GL_COPY_READ_BUFFER, GL_COPY_WRITE_BUFFER, 0, 0, 8 * sizeof(float));
```

```
float vertexData[] = { ... };
glBindBuffer(GL_ARRAY_BUFFER, vbo1);
glBindBuffer(GL_COPY_WRITE_BUFFER, vbo2);
glCopyBufferSubData(GL_ARRAY_BUFFER, GL_COPY_WRITE_BUFFER, 0, 0, 8 * sizeof(float));
```

ⓒ nvidia

Advanced GLSL

GLSL's Built-in Variables

- 추가적인 의미를 가지는 gl_ 접두사(prefix)가 붙어 있는 OpenGL 내장 변수들
  - gl_Position : VS의 출력 벡터 (렌더링시 필수)
  - gl_FragCoord : FS의 출력 벡터 (렌더링시 필수)

- 그 외의 다른 흥미로운 내장 입출력 변수들을 다룰 예정

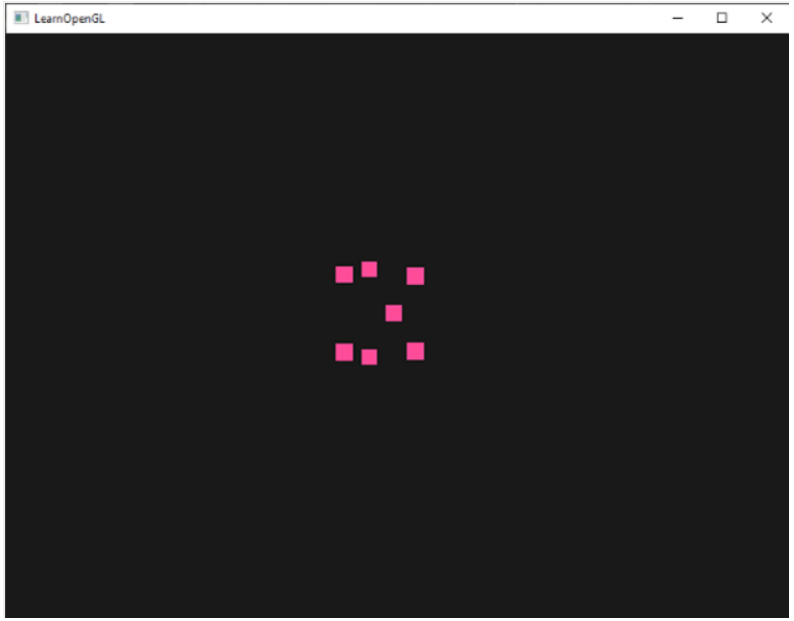
- 모든 GLSL의 내장 변수들은 아래 위키에서 참조
  - Built-in Variable (GLSL) - OpenGL Wiki (khronos.org)

Vertex Shader Variables

- gl_PointSize
  - Primitive로 GL_POINTS를 선택했을 때, vertex마다 point(점)의 너비와 높이를 픽셀 단위로 선택 가능
  - VS에서 점의 크기를 바꾸기 위해서는 glEnable(GL_PROGRAM_POINT_SIZE); 를 먼저 수행해 줘야 함

- 멀리 떨어진 vertex일수록 점의 크기를 더 크게 바꾸는 예제
  - 점의 크기를 viewer와 vertex 사이의 거리인 clip-space 위치의 z값과 동일하게 설정

```
void main()
{
    gl_Position = projection * view * model * vec4(aPos, 1.0);
    gl_PointSize = gl_Position.z;
}
```



- gl_PointSize를 particle 생성에 이용하는 것도 가능

Vertex Shader Variables

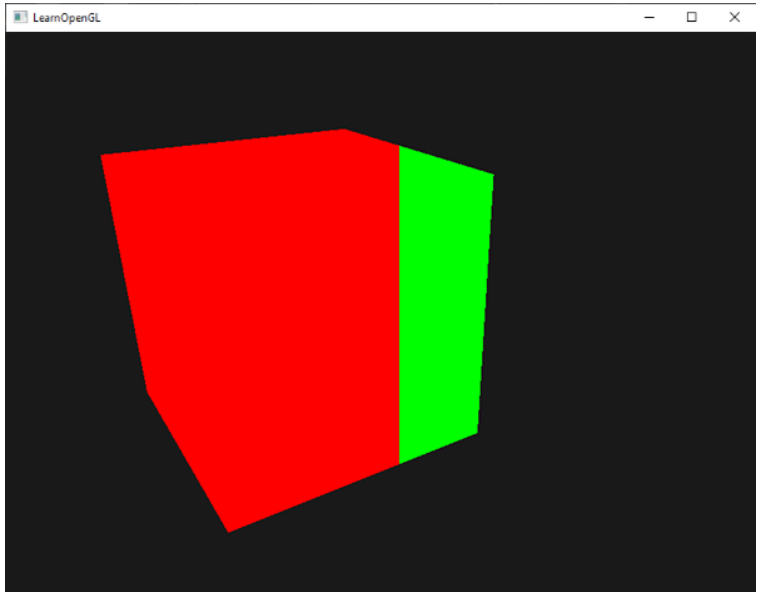
- gl_VertexID
  - 결과에 영향을 주는 출력 변수인 gl_Position, gl_PointSize와 달리, gl_VertexID는 입력 변수임
  - 현재 그리고 있는 vertex의 ID를 가짐
  - glDrawElements()를 사용하여 indexed rendering을 할 때, 그리고 있는 vertex의 현재 index를 가짐
  - 인덱스 없이 glDrawArrays()를 사용할 경우에는 render call의 시작 이후로 현재까지 처리된 vertex의 개수를 가짐

Fragment Shader Variables

- gl_FragCoord
  - Depth testing에서 fragment의 깊이 값을 얻기 위해 gl_FragCoord.z을 사용
  - 이 벡터의 x, y 요소(fragment의 screen-space 좌표)를 사용하여 흥미로운 효과를 낼 수 있음

- Fragment의 윈도우 좌표를 기반으로 다른 컬러 값을 계산하는 예제
  - 서로 다른 두 기법의 시각적 효과를 비교하기 위해 많이 쓰임
    (하나의 출력은 화면 왼쪽에, 하나의 출력은 화면 오른쪽에)
  - Real-time Denoising HW(Ver. 0.5)_San-miguel Scene - YouTube

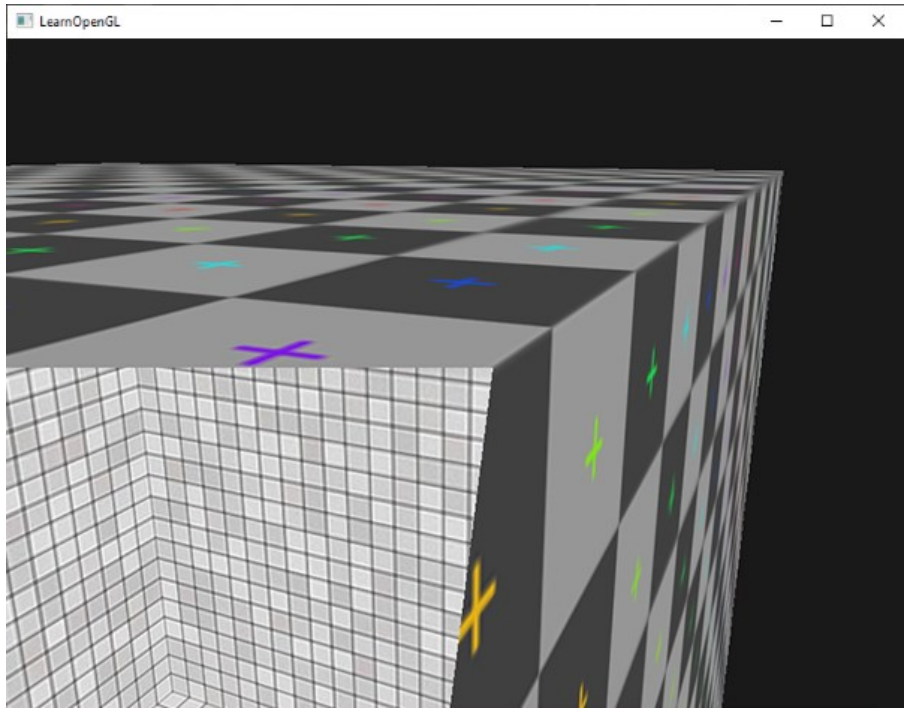
```
void main()
{
    if(gl_FragCoord.x < 400)
        FragColor = vec4(1.0, 0.0, 0.0, 1.0);
    else
        FragColor = vec4(0.0, 1.0, 0.0, 1.0);
}
```



Fragment Shader Variables

- gl_FrontFacing
  - Face culling을 사용하지 않는다면, gl_FrontFacing 변수는 현재 fragment의 면(face)을 알려 줌
  - 전면(front face)이면 true, 후면(back face)이면 false
  - 전면(front face)와 후면(back face)에 각각 다른 텍스처를 입히는 예제 (face culling 활성화시 무의미)

```
#version 330 core
out vec4 FragColor;

in vec2 TexCoords;

uniform sampler2D frontTexture;
uniform sampler2D backTexture;

void main()
{
    if(gl_FrontFacing)
        FragColor = texture(frontTexture, TexCoords);
    else
        FragColor = texture(backTexture, TexCoords);
}
```



Fragment Shader Variables

- gl_FragDepth
  - Read-only인 gl_FragColor와 달리, fragment의 screen-space 좌표의 depth값을 직접 수정 가능한 변수
  - Depth값의 범위는 0.0에서 1.0 사이로 설정 (자동 초기값은 gl_FragCoord.z)

- 직접 depth 값을 설정할 경우의 단점
  - Early depth testing을 강제(OpenGL 4.2) 하지 않았다면 이 기능이 자동으로 비활성화
  - 그 이유는 fragment shader 가 실행되기 전에 이 fragment가 어떠한 depth 값을 가질지 알 수 없기 때문
  - 성능 패널티로 이어질 수 있음

Fragment Shader Variables

- 성능 패널티 최소화를 위해 depth condition과 함께 gl_FragDepth를 재정의 가능 (OpenGL 4.2)

```
layout (depth_<condition>) out float gl_FragDepth;
```

  - 이 조건에 따라 depth값을 변경한다는 선언
  - 부분적으로 early depth testing 적용 가능
  - 사용 예시

```
#version 420 core // note the GLSL version!
out vec4 FragColor;
layout (depth_greater) out float gl_FragDepth;

void main()
{
    FragColor = vec4(1.0);
    gl_FragDepth = gl_FragCoord.z + 0.1;
}
```

Condition	Description
any	The default value. Early depth testing is disabled.
greater	You can only make the depth value larger compared to gl_FragCoord.z.
less	You can only make the depth value smaller compared to gl_FragCoord.z.
unchanged	If you write to gl_FragDepth, you will write exactly gl_FragCoord.z.

- 성능 향상 예시 (11.76 ms → 9.52 ms)
  - Conservative Depth Output (and Other Lesser-Known D3D11 Features) – The Danger Zone (wordpress.com)

Interface Blocks

- 한 shader에서 다른 shader로 데이터를 보낼 때 구조체처럼 변수를 묶어 사용
  - Block name은 셰이더간 같게 명명해야 함 (아래 예에서는 VS_OUT)
  - Instance name은 자유롭게 명명 가능 (아래 예에서는 vs_out 및 fs_in)
    단, 네이밍으로 인해 혼동이 발생되지 않도록 유의

VS

```
#version 330 core
layout (location = 0) in vec3 aPos;
layout (location = 1) in vec2 aTexCoords;

uniform mat4 model;
uniform mat4 view;
uniform mat4 projection;

out VS_OUT
{
    vec2 TexCoords;
} vs_out;

void main()
{
    gl_Position = projection * view * model * vec4(aPos, 1.0);
    vs_out.TexCoords = aTexCoords;
}
```

FS

```
#version 330 core
out vec4 FragColor;

in VS_OUT
{
    vec2 TexCoords;
} fs_in;

uniform sampler2D texture;

void main()
{
    FragColor = texture(texture, fs_in.TexCoords);
}
```

Uniform Buffer Objects

- 여러 shader program에 걸쳐 사용되는 전역 uniform 변수의 모음을 선언 가능
  - 각각의 shader마다 uniform 변수의 값을 따로따로 지정하는 것을 방지할 수 있음
  - Uniform buffer object는 다른 buffer들과 같은 buffer → glGenBuffers() 함수로 생성, GL_UNIFORM_BUFFER에 바인딩 가능, 모든 연관 uniform 데이터들을 buffer에 저장 가능
- Uniform buffer object의 데이터가 저장되는 방법에 대한 특정 규칙 → uniform block layout
- 간단한 VS에서 projection, view 행렬을 uniform block에 저장하는 예제

VS

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

layout (std140) uniform Matrices
{
    mat4 projection;
    mat4 view;
};

uniform mat4 model;

void main()
{
    gl_Position = projection * view * model * vec4(aPos, 1.0);
}
```

Uniform Block Layout

- Uniform block의 내용은 단순 buffer 객체에 저장
  - 어떠한 유형의 데이터를 가지고 있는지에 대한 정보가 없으므로, 메모리의 어떤 부분이 어떠한 uniform 변수들에 대응되는지 OpenGL에 알려줄 필요가 있음
  - OpenGL은 각 요소의 크기는 명시하지만 변수들 사이의 간격은 명시하지 않는데, 이를 지정하는 방법이 uniform block layout으로, shared, std140, packed가 존재
    https://registry.khronos.org/OpenGL/extensions/ARB/ARB_uniform_buffer_object.txt

```
layout (std140) uniform ExampleBlock
{
    float value;
    vec3  vector;
    mat4  matrix;
    float values[3];
    bool  boolean;
    int   integer;
};
```

- shared (기본값)
  - HW에서 offset이 한 번 정의되면 이 layout이 여러 program에서 공유(shared)
  - GLSL은 최적화를 위해 이 변수들의 순서는 유지한 채 uniform 변수들을 재위치시킬 수 있음
  - 결국 각 uniform 변수들이 어떤 offset에 있는지 직관적으로 알 수 없음
    → glGetUniformIndices()와 같은 함수로 uniform block의 index를 검색

Uniform Block Layout

- std140
  - 명시적인 규칙에 따르는 메모리 layout
  - 수동으로 각 변수에 대한 offset을 알아낼 수 있음
  - 많이 쓰이는 규칙들 (아래 표에서 N은 int,float,bool와 같은 4바이트의 각 요소를 의미)

Type	Layout rule
Scalar e.g. `int` or `bool`	Each scalar has a base alignment of N.
Vector	Either 2N or 4N. This means that a `vec3` has a base alignment of 4N.
Array of scalars or vectors	Each element has a base alignment equal to that of a `vec4`.
Matrices	Stored as a large array of column vectors, where each of those vectors has a base alignment of `vec4`.
Struct	Equal to the computed size of its elements according to the previous rules, but padded to a multiple of the size of a `vec4`.

Uniform Block Layout

Type	Layout rule
Scalar e.g. `int` or `bool`	Each scalar has a base alignment of N.
Vector	Either 2N or 4N. This means that a `vec3` has a base alignment of 4N.
Array of scalars or vectors	Each element has a base alignment equal to that of a `vec4`.
Matrices	Stored as a large array of column vectors, where each of those vectors has a base alignment of `vec4`.
Struct	Equal to the computed size of its elements according to the previous rules, but padded to a multiple of the size of a `vec4`.

- std140 (cont.)
  - 예제

```
layout (std140) uniform ExampleBlock
{
                     // base alignment  // aligned offset
    float value;     // 4               // 0
    vec3 vector;     // 16              // 16  (offset must be multiple of 16 so 4->16)
    mat4 matrix;     // 16              // 32  (column 0)
                     // 16              // 48  (column 1)
                     // 16              // 64  (column 2)
                     // 16              // 80  (column 3)
    float values[3]; // 16              // 96  (values[0])
                     // 16              // 112 (values[1])
                     // 16              // 128 (values[2])
    bool boolean;    // 4               // 144
    int integer;     // 4               // 148
};
```

- packed
  - 컴파일러가 uniform 변수들을 uniform block에 상관없이 최적화
  - 한 layout이 서로 다른 program간 같은 값이 유지된다는 보장이 없음

Using Uniform Buffers

- Shader의 uniform block의 선언 및 메모리 layout 지정을 마친 후, 실제로 이를 사용하기 위한 초기 작업
  - 먼저 Uniform block이 저장되는 버퍼를 할당

CPU

```
unsigned int uboExampleBlock;
glGenBuffers(1, &uboExampleBlock);
glBindBuffer(GL_UNIFORM_BUFFER, uboExampleBlock);
glBufferData(GL_UNIFORM_BUFFER, 152, NULL, GL_STATIC_DRAW); // allocate 152 bytes of memory
glBindBuffer(GL_UNIFORM_BUFFER, 0);
```

  - 이후 uboExampleBlock를 바인딩한 후 glBufferSubdata() 함수를 사용하면 메모리를 수정 가능

Using Uniform Buffers

- 다음으로 shader의 uniform block과 OpenGL에서 만든 uniform buffer을 binding해 줌
  - Uniform block과 binding point와의 연결

```
unsigned int lights_index = glGetUniformBlockIndex(shaderA.ID, "Lights");
glUniformBlockBinding(shaderA.ID, lights_index, 2);
```

CPU 또는

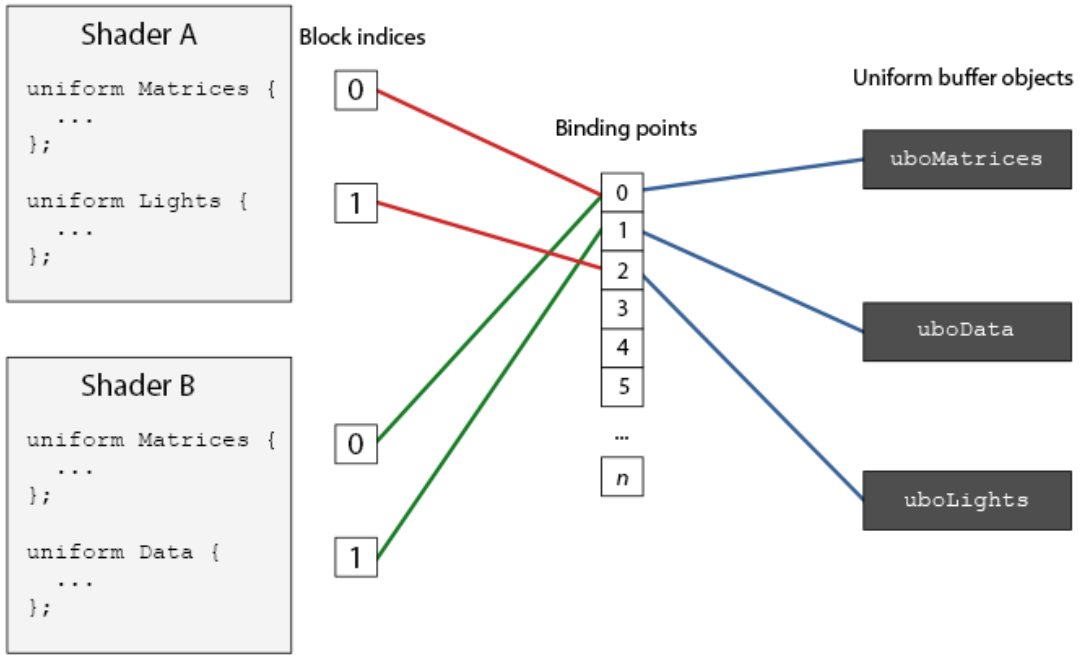
```
layout(std140, binding = 2) uniform Lights { ... };
```

VS FS ← (OpenGL 4.2 이상에서만 가능)

  - Uniform buffer와 binding point와의 연결

```
glBindBufferBase(GL_UNIFORM_BUFFER, 2, uboExampleBlock);
// or
glBindBufferRange(GL_UNIFORM_BUFFER, 2, uboExampleBlock, 0, 152);
```

CPU



Using Uniform Buffers

- 앞서 소개한 glBufferSubData() 함수를 통해 uniform buffer 수정 가능

```
glBindBuffer(GL_UNIFORM_BUFFER, uboExampleBlock);
int b = true; // bools in GLSL are represented as 4 bytes, so we store it in an integer
glBufferSubData(GL_UNIFORM_BUFFER, 144, 4, &b);
glBindBuffer(GL_UNIFORM_BUFFER, 0);
```

CPU

A Simple Example

- Projection, view 행렬을 Matrices라고 불리는 uniform block에 저장하고, 각기 다른 4개 셰이더에서 이를 함께 사용하는 예

VS

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

layout (std140) uniform Matrices
{
    mat4 projection;
    mat4 view;
};
uniform mat4 model;

void main()
{
    gl_Position = projection * view * model * vec4(aPos, 1.0);
}
```

CPU

```
unsigned int uniformBlockIndexRed    = glGetUniformBlockIndex(shaderRed.ID, "Matrices");
unsigned int uniformBlockIndexGreen  = glGetUniformBlockIndex(shaderGreen.ID, "Matrices");
unsigned int uniformBlockIndexBlue   = glGetUniformBlockIndex(shaderBlue.ID, "Matrices");
unsigned int uniformBlockIndexYellow = glGetUniformBlockIndex(shaderYellow.ID, "Matrices");

glUniformBlockBinding(shaderRed.ID,    uniformBlockIndexRed, 0);
glUniformBlockBinding(shaderGreen.ID,  uniformBlockIndexGreen, 0);
glUniformBlockBinding(shaderBlue.ID,   uniformBlockIndexBlue, 0);
glUniformBlockBinding(shaderYellow.ID, uniformBlockIndexYellow, 0);
```

A Simple Example

- Projection, view 행렬을 Matrices라고 불리는 uniform block에 저장하고, 각기 다른 4개 셰이더에서 이를 함께 사용하는 예 (cont.)

CPU

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

glBindBuffer(GL_UNIFORM_BUFFER, uboMatrices);
glBufferData(GL_UNIFORM_BUFFER, 2 * sizeof(glm::mat4), NULL, GL_STATIC_DRAW);
glBindBuffer(GL_UNIFORM_BUFFER, 0);

glBindBufferRange(GL_UNIFORM_BUFFER, 0, uboMatrices, 0, 2 * sizeof(glm::mat4));
```

CPU

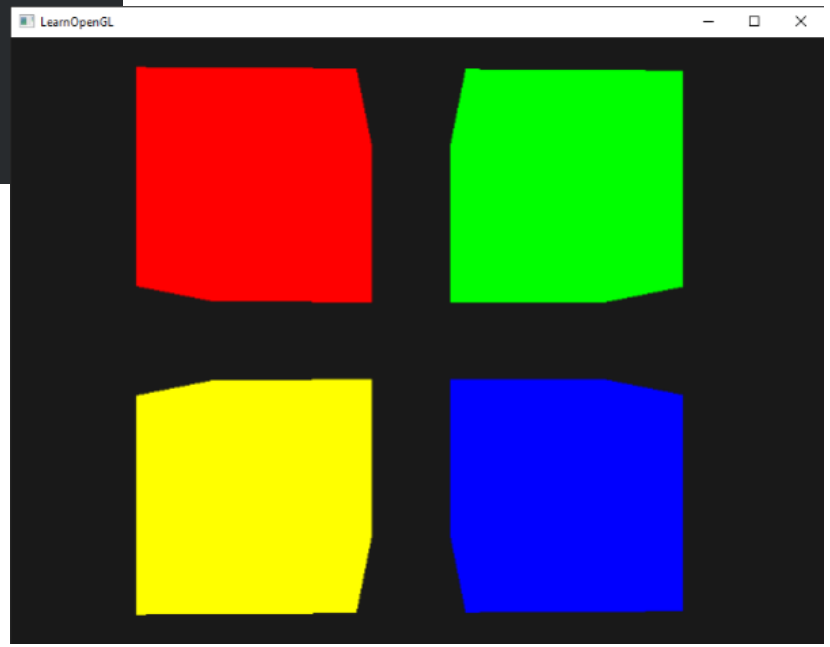
```
glm::mat4 projection = glm::perspective(glm::radians(45.0f), (float)width/(float)height, 0.1f, 100.0f);
glBindBuffer(GL_UNIFORM_BUFFER, uboMatrices);
glBufferSubData(GL_UNIFORM_BUFFER, 0, sizeof(glm::mat4), glm::value_ptr(projection));
glBindBuffer(GL_UNIFORM_BUFFER, 0);
glm::mat4 view = camera.GetViewMatrix();
glBindBuffer(GL_UNIFORM_BUFFER, uboMatrices);
glBufferSubData(GL_UNIFORM_BUFFER, sizeof(glm::mat4), sizeof(glm::mat4), glm::value_ptr(view));
glBindBuffer(GL_UNIFORM_BUFFER, 0);
```

A Simple Example

- Projection, view 행렬을 Matrices라고 불리는 uniform block에 저장하고, 각기 다른 4개 셰이더에서 이를 함께 사용하는 예 (cont.)

CPU

```
glBindVertexArray(cubeVAO);
shaderRed.use();
glm::mat4 model = glm::mat4(1.0f);
model = glm::translate(model, glm::vec3(-0.75f, 0.75f, 0.0f));  // move top-left
shaderRed.setMat4("model", model);
glDrawArrays(GL_TRIANGLES, 0, 36);
// ... draw Green Cube
// ... draw Blue Cube
// ... draw Yellow Cube
```



- Uniform buffer object의 장점
  - 많은 uniform들을 한꺼번에 설정하므로 속도가 빠름
  - 여러 shader에 걸쳐있는 동일한 uniform을 쉽게 수정 가능
  - shader에서 GL_MAX_VERTEX_UNIFORM_COMPONENTS 이상의 아주 많은 uniform들을 사용 가능 (예: 스켈레톤 애니메이션)

ⓒ nvidia

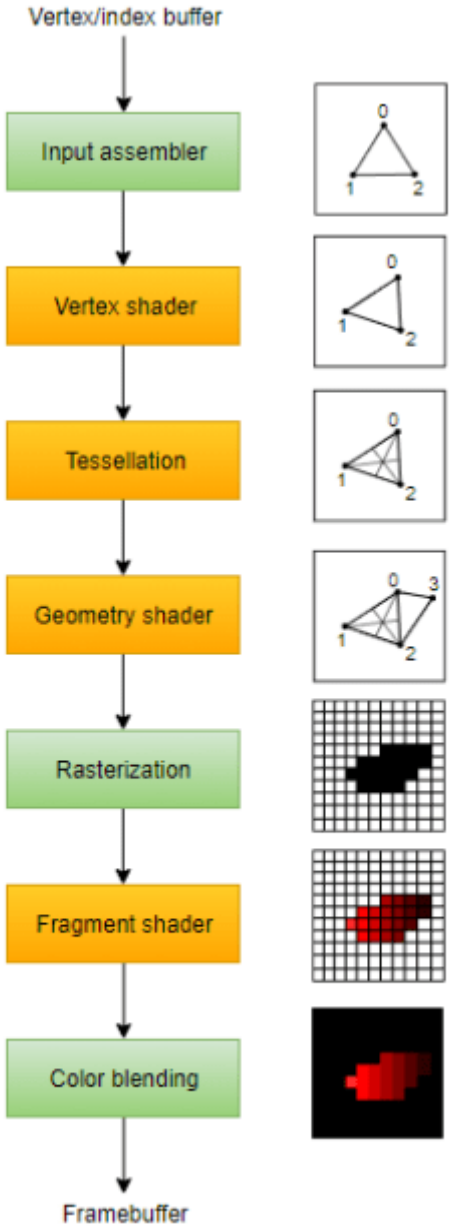
Geometry Shader

Geometry Shader

- Geometry shader
  - VS와 FS 사이에 존재하는 선택적인 shader 단계
  - 입력: 하나의 primitive(점, 삼각형 등)를 이루는 vertex들의 모음
  - 출력: 원래 주어진 vertex들보다 더 많은 vertex들을 생성하는 완전히 다르게 변환된 primitive들
- Geometry shader의 예제
  - 기존 gl_Position의 벡터를 조작한 vertex 두 개를 line strip으로 결합하여 출력

GS

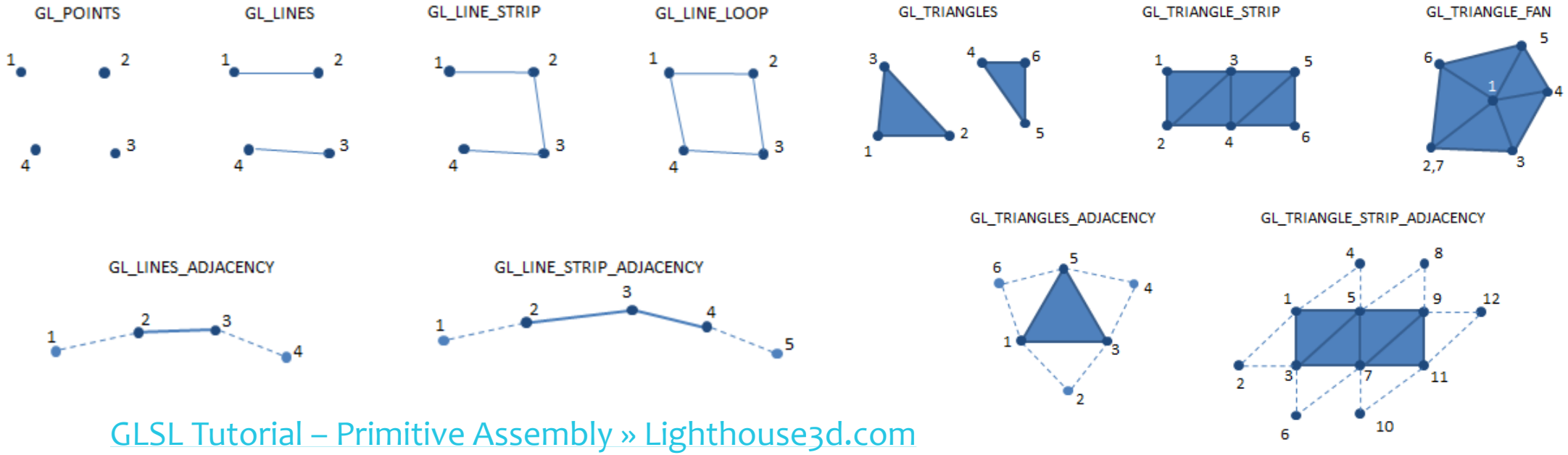
```
#version 330 core
layout (points) in;
layout (line_strip, max_vertices = 2) out;

void main() {
    gl_Position = gl_in[0].gl_Position + vec4(-0.1, 0.0, 0.0, 0.0);
    EmitVertex();

    gl_Position = gl_in[0].gl_Position + vec4( 0.1, 0.0, 0.0, 0.0);
    EmitVertex();

    EndPrimitive();
}
```



Introduction - Vulkan Tutorial (vulkan-tutorial.com)

Geometry Shader

- GS의 input layout qualifier `layout (points) in;`
  - points: when drawing GL_POINTS primitives (1).
  - lines: when drawing GL_LINES or GL_LINE_STRIP (2).
  - lines_adjacency: GL_LINES_ADJACENCY or GL_LINE_STRIP_ADJACENCY (4).
  - triangles: GL_TRIANGLES, GL_TRIANGLE_STRIP or GL_TRIANGLE_FAN (3).
  - triangles_adjacency : GL_TRIANGLES_ADJACENCY or GL_TRIANGLE_STRIP_ADJACENCY (6).



GLSL Tutorial – Primitive Assembly » Lighthouse3d.com

Geometry Shader

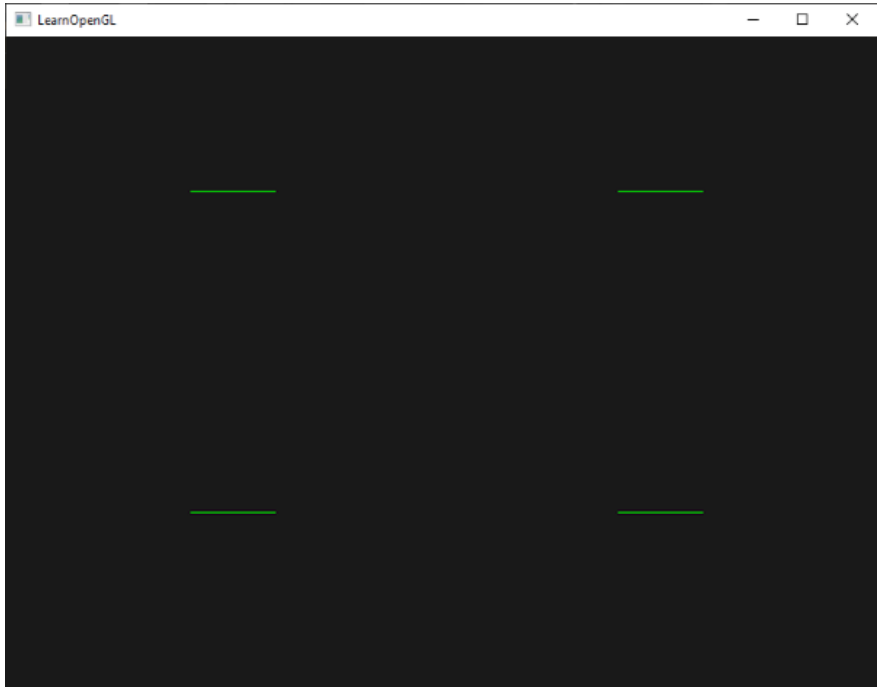
- GS의 output layout qualifier
```
layout (line_strip, max_vertices = 2) out;
```
  - Points, line_strip, triangle_strip
  - 출력될 vertex의 최고 개수도 지정 가능

- GS에서 이전 단계의 기본 출력은 gl_in[]이라는 내장 변수로 얻음
  - Interface block 형태

```
in gl_Vertex
{
    vec4  gl_Position;
    float gl_PointSize;
    float gl_ClipDistance[];
} gl_in[];
```

- p.28의 GS 코드를 이용하여 4개의 점을 입력한 결과
  - 4개의 선으로 변환되어 그려짐

```
glDrawArrays(GL_POINTS, 0, 4);
```
CPU



Using Geometry Shaders

- 점으로 집을 지어보는 예제
  - 먼저 4개 녹색 점을 화면에 그림

```
float points[] = {
        -0.5f,  0.5f, // top-left
         0.5f,  0.5f, // top-right
         0.5f, -0.5f, // bottom-right
        -0.5f, -0.5f  // bottom-left
};
```

```
shader.use();
glBindVertexArray(VAO);
glDrawArrays(GL_POINTS, 0, 4);
```

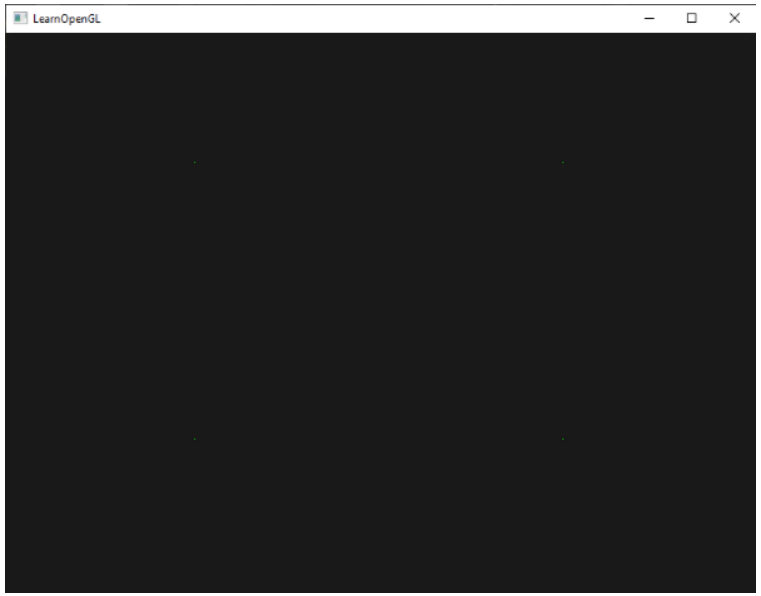
CPU

```
#version 330 core
layout (location = 0) in vec2 aPos;

void main()
{
    gl_Position = vec4(aPos.x, aPos.y, 0.0, 1.0);
}
```

VS

```
#version 330 core
out vec4 FragColor;

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

FS

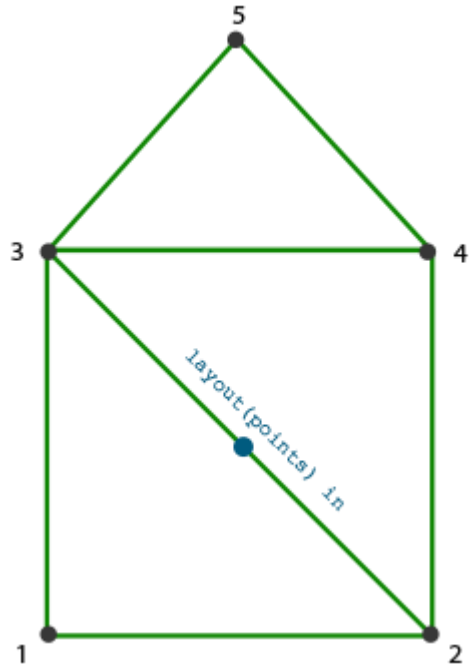


Using Geometry Shaders

- 점으로 집을 지어보는 예제 (cont.)
  - GS를 이용하여 점 1개를 5개로 뻥튀기

```
#version 330 core
layout (points) in;
layout (triangle_strip, max_vertices = 5) out;

void build_house(vec4 position)
{
    gl_Position = position + vec4(-0.2, -0.2, 0.0, 0.0);    // 1:bottom-left
    EmitVertex();
    gl_Position = position + vec4( 0.2, -0.2, 0.0, 0.0);    // 2:bottom-right
    EmitVertex();
    gl_Position = position + vec4(-0.2,  0.2, 0.0, 0.0);    // 3:top-left
    EmitVertex();
    gl_Position = position + vec4( 0.2,  0.2, 0.0, 0.0);    // 4:top-right
    EmitVertex();
    gl_Position = position + vec4( 0.0,  0.4, 0.0, 0.0);    // 5:top
    EmitVertex();
    EndPrimitive();
}

void main() {
    build_house(gl_in[0].gl_Position);
}
```

GS



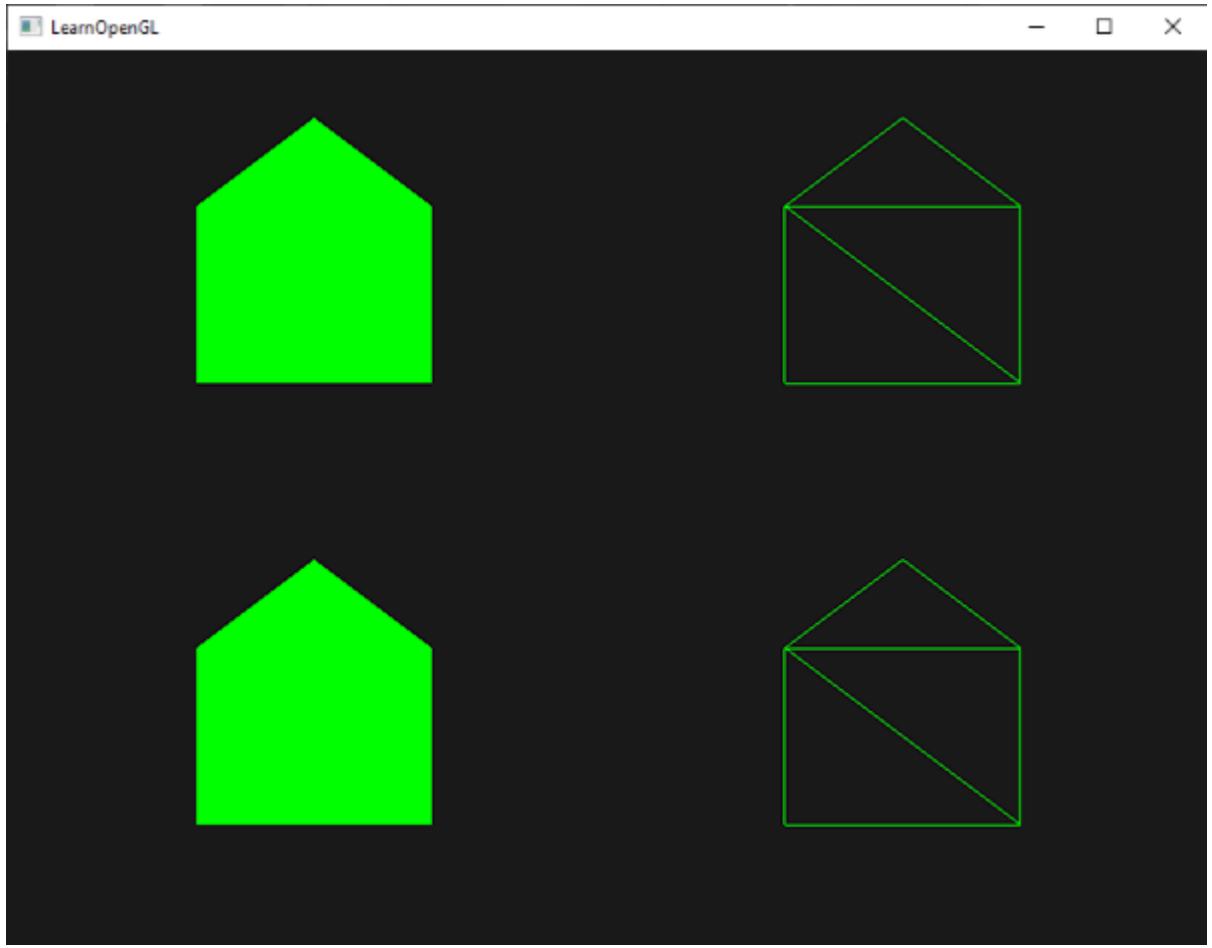
Using Geometry Shaders

- 점으로 집을 지어보는 예제 (cont.)
  - GS는 별도로 컴파일 후 링크해줘야 함

```
geometryShader = glCreateShader(GL_GEOMETRY_SHADER);
glShaderSource(geometryShader, 1, &gShaderCode, NULL);
glCompileShader(geometryShader);
[...]
glAttachShader(program, geometryShader);
glLinkProgram(program);
```

CPU

  - 결과물 (오른쪽은 wireframe)



Using Geometry Shaders

- 점으로 집을 지어보는 예제 (cont.)
  - 집마다 색깔 변경

CPU

```
float points[] = {
    -0.5f,  0.5f, 1.0f, 0.0f, 0.0f, // top-left
     0.5f,  0.5f, 0.0f, 1.0f, 0.0f, // top-right
     0.5f, -0.5f, 0.0f, 0.0f, 1.0f, // bottom-right
    -0.5f, -0.5f, 1.0f, 1.0f, 0.0f  // bottom-left
};
```

VS

```
#version 330 core
layout (location = 0) in vec2 aPos;
layout (location = 1) in vec3 aColor;

out VS_OUT {
    vec3 color;
} vs_out;

void main()
{
    gl_Position = vec4(aPos.x, aPos.y, 0.0, 1.0);
    vs_out.color = aColor;
}
```

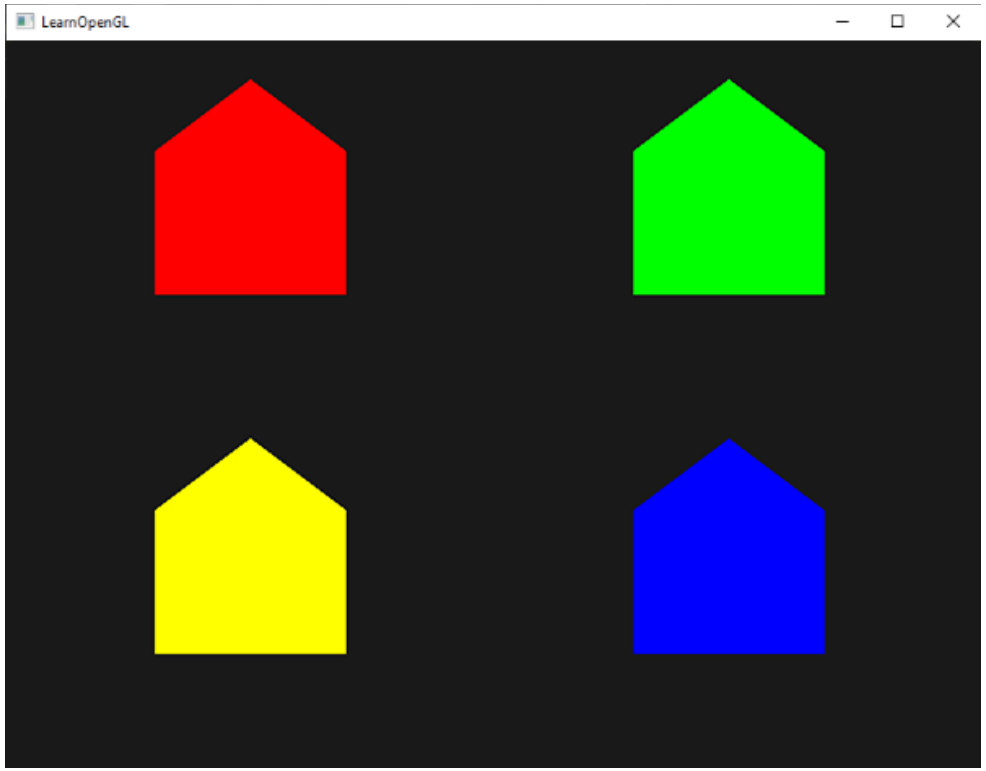
GS

```
in VS_OUT {
    vec3 color;
} gs_in[];

out vec3 fColor;
```

```
fColor = gs_in[0].color; // gs_in[0] since there's only one input vertex
gl_Position = position + vec4(-0.2, -0.2, 0.0, 0.0);    // 1:bottom-left
EmitVertex();
gl_Position = position + vec4( 0.2, -0.2, 0.0, 0.0);    // 2:bottom-right
EmitVertex();
gl_Position = position + vec4(-0.2,  0.2, 0.0, 0.0);    // 3:top-left
EmitVertex();
gl_Position = position + vec4( 0.2,  0.2, 0.0, 0.0);    // 4:top-right
EmitVertex();
gl_Position = position + vec4( 0.0,  0.4, 0.0, 0.0);    // 5:top
EmitVertex();
EndPrimitive();
```

Using Geometry Shaders

- 점으로 집을 지어보는 예제 (cont.)
  - 집마다 색깔 변경한 결과



Using Geometry Shaders

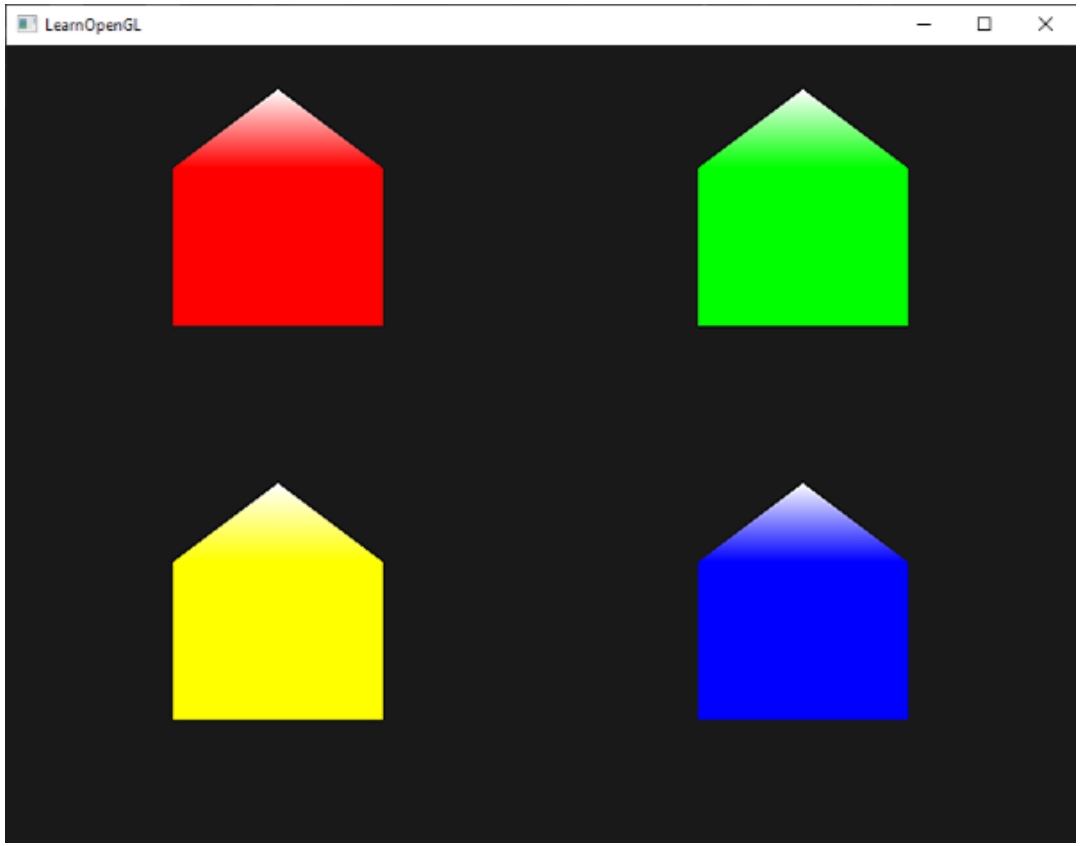
- 점으로 집을 지어보는 예제 (cont.)
  - 지붕의 vertex만 하얗게 지정해 눈이 온 것 같은 효과

```
fColor = gs_in[0].color;
gl_Position = position + vec4(-0.2, -0.2, 0.0, 0.0);    // 1:bottom-left
EmitVertex();
gl_Position = position + vec4( 0.2, -0.2, 0.0, 0.0);    // 2:bottom-right
EmitVertex();
gl_Position = position + vec4(-0.2,  0.2, 0.0, 0.0);    // 3:top-left
EmitVertex();
gl_Position = position + vec4( 0.2,  0.2, 0.0, 0.0);    // 4:top-right
EmitVertex();
gl_Position = position + vec4( 0.0,  0.4, 0.0, 0.0);    // 5:top
fColor = vec3(1.0, 1.0, 1.0);
EmitVertex();
EndPrimitive();
```

GS

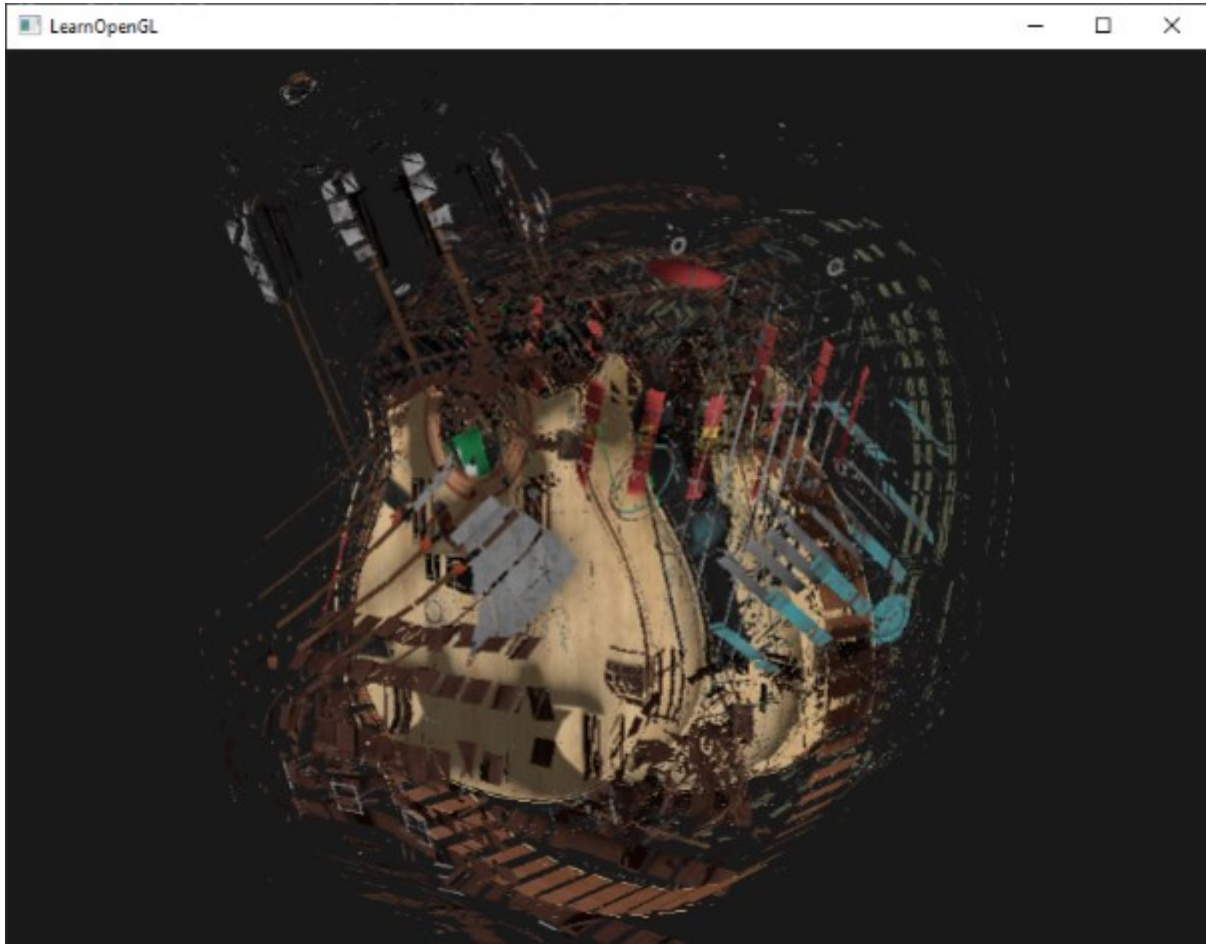
  - 소스 코드

    Code Viewer. Source code: src/4.advanced_opengl/9.1.geometry_shader_houses/geometry_shader_houses.cpp (learnopengl.com)



Exploding Objects

- 각 삼각형들을 normal 벡터 방향으로 시간에 따라 이동시켜, 마치 폭발하는 것처럼 표현
  - 오른쪽은 nanosuit model에 적용한 화면
  - 복잡도의 상관 없이 모든 object에 적용 가능



Exploding Objects

- 삼각형의 면에 수직인 normal 벡터 계산
  - 삼각형의 면에 평행하는 a, b 벡터를 외적하면 가능

```
vec3 GetNormal()
{
   vec3 a = vec3(gl_in[0].gl_Position) - vec3(gl_in[1].gl_Position);
   vec3 b = vec3(gl_in[2].gl_Position) - vec3(gl_in[1].gl_Position);
   return normalize(cross(a, b));
}
```
GS

- 시간에 따라 폭발하는 함수 구현
  - sin함수를 사용하므로 폭발<->원상복구를 반복

```
vec4 explode(vec4 position, vec3 normal)
{
    float magnitude = 2.0;
    vec3 direction = normal * ((sin(time) + 1.0) / 2.0) * magnitude;
    return position + vec4(direction, 0.0);
}
```
GS

Exploding Objects

- 완성된 main 함수 (우측 코드)

- Uniform time 변수를
  OpenGL에서 매 프레임마다 설정해줘야 함

```
shader.setFloat("time", glfwGetTime());
```
CPU

- 소스 코드
  - Code Viewer. Source code: src/4.advanced_opengl/9.2.geometry_shader_exploding/geometry_shader_exploding.cpp (learnopengl.com)

GS

```
#version 330 core
layout (triangles) in;
layout (triangle_strip, max_vertices = 3) out;

in VS_OUT {
    vec2 texCoords;
} gs_in[];

out vec2 TexCoords;

uniform float time;

vec4 explode(vec4 position, vec3 normal) { ... }

vec3 GetNormal() { ... }

void main() {
    vec3 normal = GetNormal();

    gl_Position = explode(gl_in[0].gl_Position, normal);
    TexCoords = gs_in[0].texCoords;
    EmitVertex();
    gl_Position = explode(gl_in[1].gl_Position, normal);
    TexCoords = gs_in[1].texCoords;
    EmitVertex();
    gl_Position = explode(gl_in[2].gl_Position, normal);
    TexCoords = gs_in[2].texCoords;
    EmitVertex();
    EndPrimitive();
}
```

Visualizing Normal Vectors

- 부정확한 normal vector로 인한 lighting 오류를 판별 가능
- 2-pass rendering을 통해 노멀 벡터를 추가로 그림
- NormalDisplayShader 구현
  - VS에서 넘겨준 노멀 벡터를 GS에서 선으로 만들어 줌

CPU

```
shader.use();
DrawScene();
normalDisplayShader.use();
DrawScene();
```

VS

```
#version 330 core
layout (location = 0) in vec3 aPos;
layout (location = 1) in vec3 aNormal;

out VS_OUT {
    vec3 normal;
} vs_out;

uniform mat4 view;
uniform mat4 model;

void main()
{
    gl_Position = view * model * vec4(aPos, 1.0);
    mat3 normalMatrix = mat3(transpose(inverse(view * model)));
    vs_out.normal = normalize(vec3(vec4(normalMatrix * aNormal, 0.0)));
}
```

GS

```
#version 330 core
layout (triangles) in;
layout (line_strip, max_vertices = 6) out;

in VS_OUT {
    vec3 normal;
} gs_in[];

const float MAGNITUDE = 0.4;

uniform mat4 projection;

void GenerateLine(int index)
{
    gl_Position = projection * gl_in[index].gl_Position;
    EmitVertex();
    gl_Position = projection * (gl_in[index].gl_Position +
                                vec4(gs_in[index].normal, 0.0) * MAGNITUDE);
    EmitVertex();
    EndPrimitive();
}

void main()
{
    GenerateLine(0); // first vertex normal
    GenerateLine(1); // second vertex normal
    GenerateLine(2); // third vertex normal
}
```

Visualizing Normal Vectors

- FS에서는 노랗게 normal 벡터의 fragment를 그림

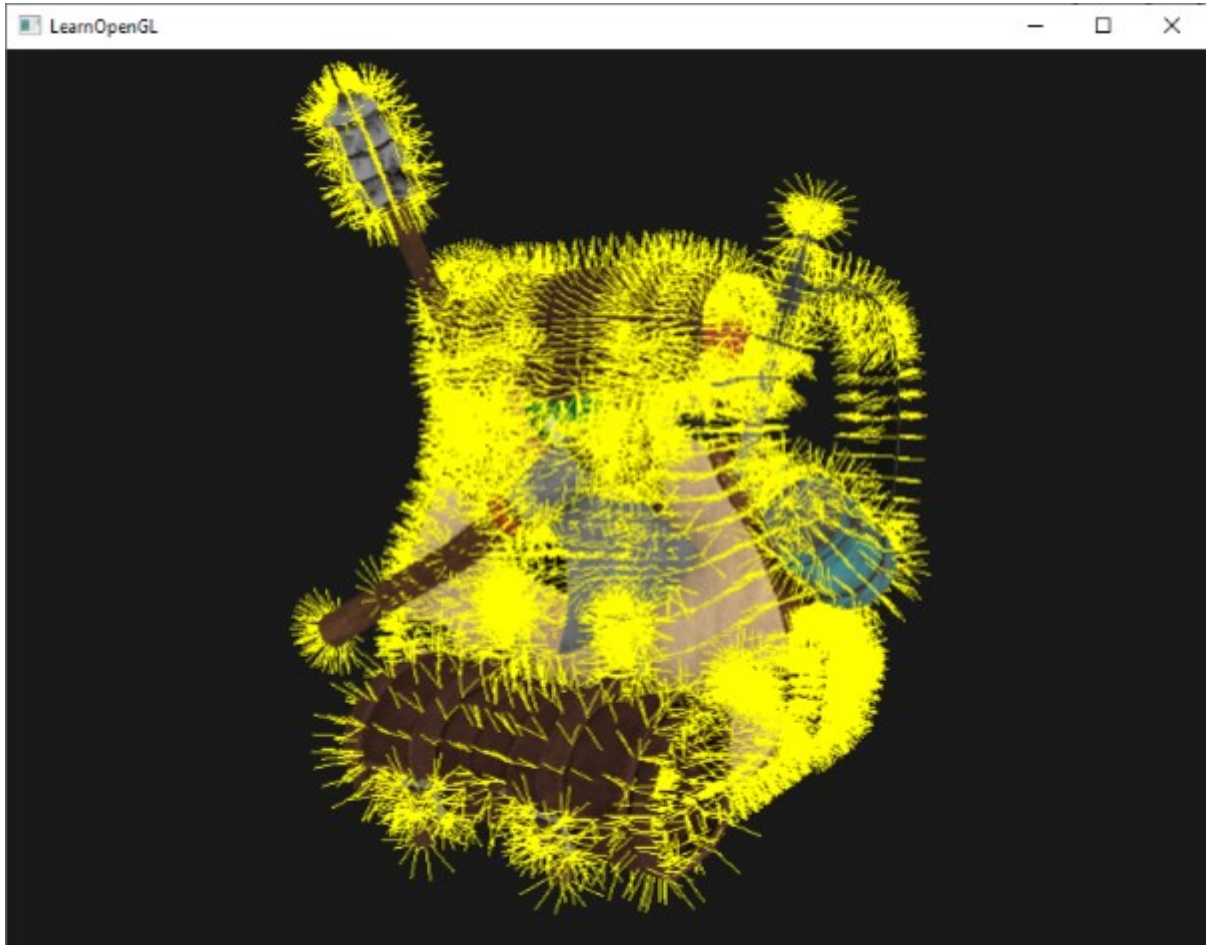
FS

```
#version 330 core
out vec4 FragColor;

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

- 소스 코드
  - Code Viewer. Source code: src/4.advanced_opengl/9.3.geometry_shader_normals/normal_visualization.cpp (learnopengl.com)

- 밤송이 같은 효과도 표현 가능!

- 결과 화면



ⓒ nvidia

마무리

마무리

- Advanced OpenGL의 세 번째 시간으로, 아래와 같은 내용을 살펴보았습니다.
  - Advanced Data
  - Advanced GLSL
  - Geometry Shader

- 다음 시간에는 아래 실습을 수행할 예정입니다.
  - 첫번째 기반 코드(LearnOpenGL 4.8)에서, UBO를 공유하는 object 하나 더 추가
  - 두번째 기반 코드(LearnOpenGL 4.9.1)에서, 집을 육각형의 보석으로 변경
  - 세번째 기반 코드(LearnOpenGL 4.9.2)에서, 지오메트리가 완전히 폭발되어 밖으로 날라가게 변경
  - 네번째 기반 코드(LearnOpenGL 4.9.3)에서, 화면에 보이는 노멀 벡터의 길이를 수정