Advanced OpenGL // Advanced GLSL

-GLSL, OpenGL을 사용해 응용 프로그램을 만들 때, 좋은 방법들에 대해서 설명함

-내장 변수, 쉐이더의 입출력 구성, 균일 버퍼 객체를 설명함

GLSL's built-in variables 내장변수

-shader은 데이터가 필요한 경우, 외부에서 전달해 줘야함

-vertex attributes, uniforms, samplers을 사용하였었음

-gl\_로 시작하는 내장변수를 사용해서 데이터 수집 및 쓰기를 위한 수단을 제공함

-gl\_position : vertex shader의 출력벡터, gl\_FragCoord : fragment shader의 출력벡터

-<https://www.khronos.org/opengl/wiki/Built-in_Variable_(GLSL>)

-Vertex Shader // gl\_PointSize

-지금까지 삼각형 프리미티브를 사용하였음, 이와 마찬가지로 프리미티브 중 GL\_POINTS가 있음

-이 경우에는 단일 꼭짓점이고, 점으로 렌더링 됨

-gl\_PointSize를 사용해서 렌더링되는 점의 크기를 설정할 수 있음 , 이 값을 vertex shader에 전달할 수 있음

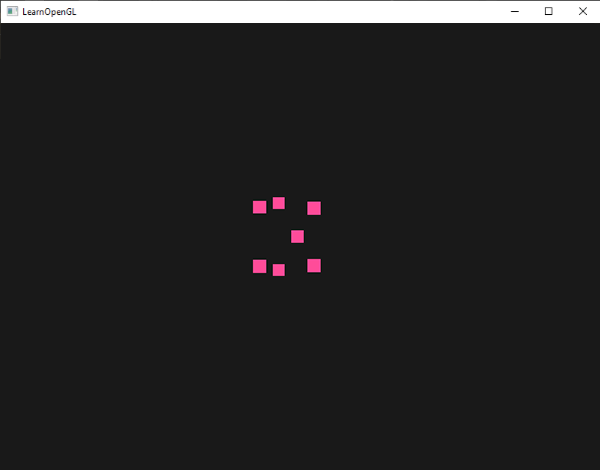
-이것은 기본적으로 비활성화 되어있고, 활성화 하면 사용할 수 있음

|  |
| --- |
| glEnable(GL\_PROGRAM\_POINT\_SIZE); |

-사용 예시

|  |
| --- |
| void main()  {  gl\_Position = projection \* view \* model \* vec4(aPos, 1.0);  gl\_PointSize = gl\_Position.z;  } |

-gl\_Position.z만큼 gl\_PointSize를 설정하기에 카메라에서, 관점에서 멀어질수록 point의 크기가 점점 커질 것임



-Vertex Shader // gl\_VertexID

-vertex shader에 대한 입력 변수임, 위에 있는 gl\_Positionm gl\_PointSize는 출력 변수임

-정점 변수 gl\_VertexID는 그리는 vertex의 현재 ID를 유지함

-glDrawElements로 index 렌더링 할 때, 우리가 그리는 정점의 index를 유지함

-glDrawArray로 index없이 렌더링 할 때, 렌더 호출의 시작 이후 처리된 정점의 번호를 유지함

-Fragment Shader // gl\_FragCoord

-지금까지 gl\_FragCoord벡터의 z 구성요소를 이용해 특정 fragment의 깊이 값을 가지고 이를 깊이 테스트에 사용하였음

-gl\_FragCoord의 x, y 구성요소를 이용해 새로운 효과를 가져올 수 있음

-x, y값은 왼쪽 하단에서부터 0, 우측 하단이 윈도우 창의 크기가 됨

-x좌표의 일정 화면 좌표를 기반으로 다른 색상을 출력하는 예제

|  |
| --- |
| 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);  } |



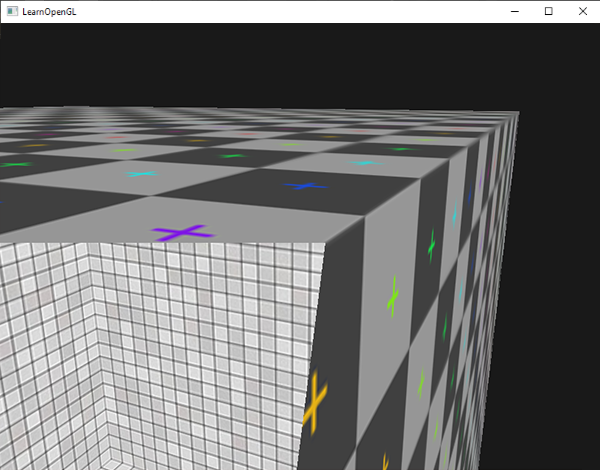
-Fragment Shader // gl\_FrontFacing

-OpenGL이 정점의 winding order을 통해 이 면이 앞면, 뒷면인가 구분할 수 있음

-GL\_FACE\_CULL을 사용해서 컬링을 사용하지 않을 경우 gl\_FrontFacing 변수는 현재 fragment가 앞면인지 뒷면인지 알려주게 됨

-사용 예시

|  |
| --- |
| #version 460 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);  } |

-Face Culling을 사용하게 되면, 컨테이너 내부의 face를 볼 수 없어서 의미가 없어짐

-Fragment Shader // gl\_FragDepth

-gl\_FragCoord는 공간의 좌표를 얻을 수 있는 입력 변수이나, 읽기 전용 변수임

-각 fragment들의 깊이 값을 조절할 수 없었지만 gl\_FragDepth를 이용해 shader 내부에서 fragment의 깊이 값을 설정이 가능함

|  |
| --- |
| gl\_FragDepth = 0.0; |

-깊이 값은 0.0~1.0의 float값을 사용함

-gl\_FragDepth를 쓰는 즉시 깊이 테스트를 비활성화 함

-fragment shader을 실행하기 전에, fragment가 가질 깊이 값을 OpenGL이 알 수 없어서 비활성화 됨

-OpenGL 4,2 ( 420 )부터 변수를 다시 선언함으로 성능 저하를 줄일 수 있음

|  |
| --- |
| layout (depth\_<condition>) out float gl\_FragDepth; |

-<condition>부분에 들어갈 수 있는 값

-any : 기본값, 깊이 테스트는 비활성화

-greater : gl\_FragCoord.z보다 큰 값만 만들 수 있음

-less : gl\_FragCoord.z보다 작은 값만 만들 수 있음

-unchanged : gl\_FragCoord.z랑 다른 값만 gl\_FragDepth를 써야함

-depth조건을 greater로 하면 OpenGL은 fragment의 깊이 값보다 큰 값만 작성해야 함

|  |
| --- |
| #version 460 core // 420보다 커야 쓸 수 있음  out vec4 FragColor;  layout (depth\_greater) out float gl\_FragDepth;  void main()  {  FragColor = vec4(1.0);  gl\_FragDepth = gl\_FragCoord.z + 0.1;  } |

-Interface Blocks

-Vertex shader, Fragment shader사이에서 데이터를 보낼 때 마다 in/out 변수들을 사용했었음

-적은 양일 경우에는 좋은 방법이지만 응용 프로그램이 커지게 되면 배열 또는 구조체를 포함할 수 있는 여러 변수들을 보내는 것이 필요해 질 것임

-Inferface block을 사용해 변수들을 그룹화 할 수 있음

-Vertex shader

|  |
| --- |
| #version 460 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;  } |

-Fragment shader

|  |
| --- |
| #version 460core  out vec4 FragColor;  in VS\_OUT  {  vec2 TexCoords;  } fs\_in;  uniform sampler2D texture;  void main()  {  FragColor = texture(texture, fs\_in.TexCoords);  } |

-vertex shader에서 out VS\_OUT 이라는 블록을 선언해서 그룹화 하고, 이름을 vs\_out으로 선언

-fragment shader에서 in VS\_OUT이라는 블록을 선언해서 그룹화, 이름을 fs\_in으로 선언함

-Uniform buffer object

-shader을 두개 이상 사용할 때, shader마다 똑같은 유니폼 변수를 설정했어야 했음

-uniform buffer object를 사용해서 여러 쉐이더 프로그램에서 동일하게 유지되는 전역 균일 변수 세트를 선언할 수 있음

-glGenBuffers를 통해 버퍼를 만들고 GL\_UNIFORM\_BUFFER타겟으로 바인딩해 모든 관련 유니폼 데이터를 버퍼에 저장할 수 있음

-저장하는 방법은 규칙이 있으며, 나중에 알아보도록 함

-vertex shader

|  |
| --- |
| #version 460 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);  } |

-projection, view를 저장하였음

-4x4 행렬을 저장하는 Matrices라는 Uniform 블록을 선언하였고, 이는 접두사를 쓰지 않고 바로 접근이 가능함

-layout (std140)은 현재 정의된 uniform블록이 content를 위해 특정 메모리 레이아웃을 사용한다는 것을 의미한다.

Uniform Block Layout

-어떤 종류의 데이터를 보유하고 있는지에 대한 정보가 없어서 메모리의 어떤 부분이 쉐이더의 어떤 uniform 변수에 해당하는지 OpenGL에게 알려줘야 함

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

-각 변수의 크기와 offset을 알아야 함

-각 변수의 크기는 C++데이터 유형에 해당되고 OpenGL에 명시되어 있음

-위에 코드는 명확하게 변수 사이의 간격에 대해 명시 되어 있지 않음

-하드웨어는 vec3를 float 근처에 배치할 수 있음, 모든 하드웨어가 이를 처리할 수 있는 것은 아니지만 float를 추가하기 전 vec3를 4개의 float배열에 채움

-좋은 기능이지만, 우리에겐 불편한 기능임

-GLSL의 shared layout 이라는 uniform 메모리 레이아웃을 사용함

-offset은 하드웨어에 의해 정의되고, 여러 프로그램 사이에 똑같이 공유되기 때문

-glGenUniformIndices와 같은 함수로 유니폼 버퍼를 정확하게 정의할 수 있으나, 많은 작업이 필요함

-그러나 일반적인 방법으로 std140 레이아웃을 사용함으로 많은 작업을 줄일 수 있음

-레이아웃 규칙은 OpenGL의 uniform 버퍼 사양에서 찾을 수 있음

<https://www.khronos.org/registry/OpenGL/extensions/ARB/ARB_uniform_buffer_object.txt>

-가장 일반적인 규칙, int, float, bool같은 GLSL 유형은 4바이트의 entity를 N으로 표시함

-Scalar(int or bool) : 각 Scalar값은 N의 정렬을 가짐

-Vector : 2N이나 4N, vec3는 4N의 정렬을 가짐

-Array of Scalar or Vector : 각 요소는 vec4의 정렬을 가짐

-Matrices : column 벡터들을 저장하는데 각 column마다 vec4의 정렬을 가짐

-Struct : 위의 규칙대로 계산되어짐, vec4의 배수로 채워짐

|  |
| --- |
| 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  }; |

-가장 효율적인 방법은 아니지만, std140 레이아웃을 사용함으로 각 이 블록을 선언한 각 프로그램에서 메모리 레이아웃이 동일하게 유지됨

-Using Uniform buffers

-Uniform Buffers를 사용하는 방법

|  |
| --- |
| 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); |

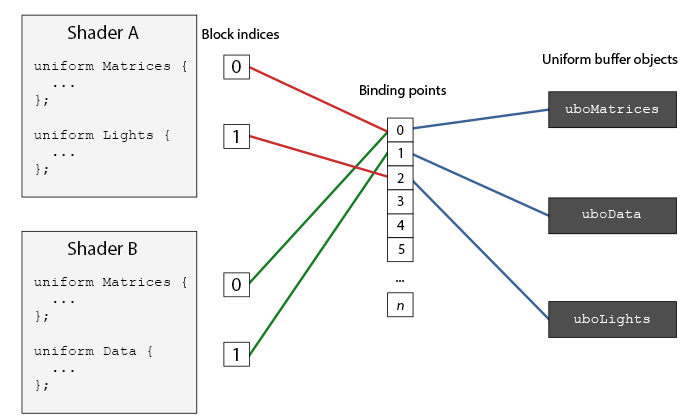
-이제 버퍼를 업데이트 하거나 insert할 때 마다 uboExampleBlock에 바인딩,

glBufferSubData로 메모리를 업데이트 하면 됨

-OpenGL은 uniform buffer가 어떤 uniform block에 해당되는지 아는 방법

-uniform 버퍼를 연결할 수 있는 여러개의 바인딩 포인트가 정의되어 있음

-uniform buffer를 만들면 해당 바인딩 포인트 중 하나에 링크 함



-shader A, B 둘다 동일한 포인트 0에 연결한 블록을 가지고 있어서 uboMatrices에서 발견되는 uniform 데이터를 공유하게 됨, 둘 다 동일한 uniform 을 정의 해야함

-uniform 블록을 특정 바인딩 포인트로 설정하는 방법

|  |
| --- |
| unsigned int lights\_index = glGetUniformBlockIndex(shaderA.ID, "Lights");  glUniformBlockBinding(shaderA.ID, lights\_index, 2); |

-바인딩 포인트 2에 light uniform block을 설정할 수 있음

|  |  |
| --- | --- |
| -OpenGL 4.2부터 glGetUniformBlockIndex, glUniformBlockBinding에 대한 호출을 저장해서 다른 레이아웃 지정자를 추가해 uniform block의 바인딩 포인트를 shader에 명시적으로 저장할 수 도 있음   |  | | --- | | layout(std140, binding = 2) uniform Lights { ... }; | |

-이후 uniform buffer object를 동일한 바인딩 포인트에 바인딩함

|  |
| --- |
| glBindBufferBase(GL\_UNIFORM\_BUFFER, 2, uboExampleBlock);  // or  glBindBufferRange(GL\_UNIFORM\_BUFFER, 2, uboExampleBlock, 0, 152); |

-Base는 타겟, 바인딩 포인트 인덱스, uniform 버퍼 객체를 인자로 받음

-Range는 추가로 여분의 오프셋, 크기 매개 변수를 예상하는 함수, 특정 범위에만 바인딩 포인트에 바인딩 할 수 있음

-이후 glBufferSubData로 데이터를 단일 바이트 배열로 추가, 버퍼의 일부를 업데이트 할 수 있음

|  |
| --- |
| glBindBuffer(GL\_UNIFORM\_BUFFER, uboExampleBlock);  int b = true; // GLSL에서 bool은 4바이트이기 때문에, integer을 만들어서 저장함  glBufferSubData(GL\_UNIFORM\_BUFFER, 144, 4, &b);  glBindBuffer(GL\_UNIFORM\_BUFFER, 0); |

-이전 장에서 가져온 glBufferSubData 설명

|  |
| --- |
| glBufferSubData( Buffer 타겟 , offset, sizeof(data), &data); // data는 들어갈 데이터  // 범위 : [offset, offset + sizeof(data)] |

-Simple Example

-uniform buffer object를 사용한 예시

-projection, view, model이라는 3개의 행렬을 계속해서 사용하고 있음, 이중 model만 자주 변경이 됨

|  |
| --- |
| #version 460 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);  } |

-자주 변경되는 model은 넣지 않았음

|  |
| --- |
| 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); |

-각 색상을 나타내는 다른 shader 4개를 사용함

-각 바인딩 포인트를 0으로 설정하였음

|  |
| --- |
| 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)); |

-uniform buffer object를 만들고 바인딩 포인트 0에 바인딩함

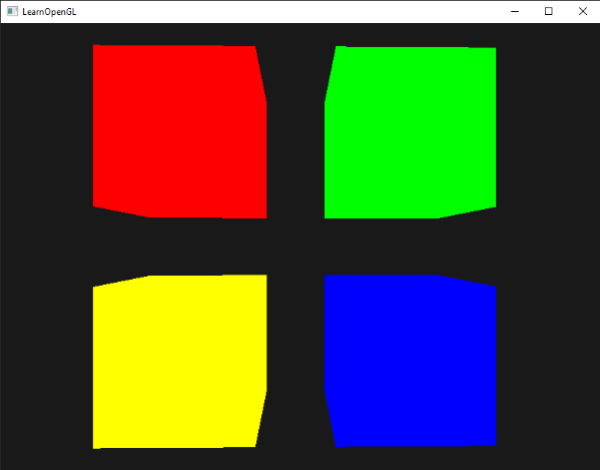
-glm::mat4의 2배 크기(projection, view)의 버퍼를 이해 충분한 메모리를 할당함

-실제로 버퍼를 데이터로 채우기, 루프에 들어가기 전 저장할 수 있음

|  |
| --- |
| 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); |

-이제 4개의 쉐이더를 이용해 4개의 큐브를 그릴 때 projection이랑 view는 똑같아야함

|  |
| --- |
| 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

-1) 한 번에 많은 유니폼을 설정할 수 있어서 하나씩 여러 개 설정하는 것 보다 빠름

-2) 동일한 쉐이더를 여러 쉐이더에서 변경하려는 경우에 uniform buffer에서 변경하는게 쉬움

-3) uniform buffer object를 사용하는 쉐이더에서 더 많은 유니폼을 사용할 수 있음

-OpenGL은 GL\_MAX\_VERTEX\_UNIFORM\_COMPONENTS로 최대 uniform 데이터 크기를 제한함