Hello Triangle

-OpenGL에서 3D 좌표를 2D 좌표로 변환하는 것은 OpenGL의 그래픽 파이프라인(Graphics pipeline)에 의해 관리가 되어짐

Graphics Pipeline

-하는 일

-3D좌표를 2D좌표로 변경

-2D좌표를 실제 색이 들어간 픽셀로 변경

-실제 2D좌표랑 픽셀위치는 다름, 픽셀위치는 근사치임

-파이프라인이 2D픽셀로 변경할 때, 여러 단계로 나누어지는데, 이 단계는 병렬로 실행될 수 있음, 병렬성으로 인해서 그래픽 카드가 GPU위에 작은 프로그램들(Shaders)을 실행해 데이터를 빠르게 처리하기 위해 작은 프로세싱 코어를 가지고 있음

-Graphics Pipeline의 단계

|  |
| --- |
|  |

-파란 부분이 사용자가 작성한 Shaders를 주입할 수 있는 부분임

-Vertex Data[] : 3D좌표에 대한 데이터의 집합

-Vertex Shader : 1개의 정점을 입력으로 받고 3D좌표를 다른 3D좌표로 변환, vertex속성에 대한 기본적인 처리

-Shape assembly : 도형의 모든 점들을 조립하는 vertex쉐이더로부터 입력값(모든 정점들)을 받음, primitive를 구성,

-Geometry Shader : 정저들의 집합을 입력으로 받음, 이 정점들의 집합으로 primitive를 구성, 새로운 primitive를 형성

-Rasterization : 최종 화면의 적절한 픽셀과 매핑

-Fragment Shader : 최종 색을 계산, OpenGL의 모든 고급 효과들이 발생함

Vertex input

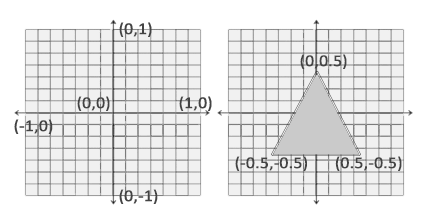
-Vertex는 3D공간의 좌표임(X, Y, Z) 이 값들은 모두 –1.0 , 1.0사이의 값임

-OpenGL에서 무언가를 렌더링 하려면 Vertex Shader, Fragment Shader은 필수

|  |
| --- |
| float vertices[] = {  -0.5f, -0.5f, 0.0f,  0.5f, -0.5f, 0.0f,  0.0f, 0.5f, 0.0f  }; |

-2D공간이기에 Z축은 전부 0.0f

-NDC (Normalized Device Coordinates)

-VBO Vertex Buffer Objects 로 메모리를 관리, 많은 정점들을 GPU메모리에 저장할 수 있음

-VBO도 고유 ID를 가지고 있음

|  |
| --- |
| unsigned int VBO;  glGenBuffers(1, &VBO); |

-VBO는 GL\_ARRAY\_BUFFER 유형을 가지고 있음

|  |
| --- |
| glBindBuffer(GL\_ARRAY\_BUFFER, VBO); |

-정점 데이터를 버퍼의 메모리에 복사하는 함수

|  |
| --- |
| glBufferData(GL\_ARRAY\_BUFFER, sizeof(vertices), vertices, GL\_STATIC\_DRAW); |

-glBufferData(버퍼의 유형, 데이터의 크기(바이트 단위), 실제 데이터, 데이터 관리방법)

-데이터 관리방법

GL\_STATIC\_DRAW: 데이터가 거의 변하지 않음

GL\_DYNAMIC\_DRAW: 데이터가 자주 변경

GL\_STREAM\_DRAW: 데이터가 그려질 때마다 변경

Vertex shader

-OpenGL Shading Languager GLSL로 작성된 기초적인 코드

|  |
| --- |
| #version ~~330~~ 460 core  layout (location = 0) in vec3 aPos;  void main()  {  gl\_Position = vec4(aPos.x, aPos.y, aPos.z, 1.0);  } |

#version으로 버전을 선언해야함, OpenGL버전과 같아야함,

core profile 기능을 사용할 것이라고 명시해 줘야함(현재 사용하는 컴퓨터의 OpenGL버전이 4.6이라 460으로 변경)

-in 을 사용해 모든 입력 정점 속성을 선언, 위치 데이터만 쓸 것이라 하나의 정점 속성만 필요함, aPos라는 vec3타입 입력변수 생성

-(locaton = 0)선언, 나중에 설명

-출력 설정하기 위해서는 미리 선언된 gl\_Position변수에 위치 데이터를 할당해야함

-vec4타입이라 w값(4번째)을 1.0f로 선언

Compiling a shader

ID를 참조해 shader 객체 생성

|  |
| --- |
| unsigned int vertexShader;  vertexShader = glCreateShader(GL\_VERTEX\_SHADER); |

-vertexShader를 GL\_VERTEX\_SHADER로 쉐이더를 생성함

-그 후 shader의 소스코드를 shader객체에 첨부, shader을 컴파일함

|  |
| --- |
| glShaderSource(vertexShader, 1, &vertexShaderSource, NULL);  glCompileShader(vertexShader); |

glShaderSource(shader객체, 소스코드가 몇개의 문자열인가, 실제 VertexShader의 소스코드, NULL);

-오류발생여부 코드

|  |
| --- |
| int success;  char infoLog[512];  glGetShaderiv(vertexShader, GL\_COMPILE\_STATUS, &success); |
| if(!success)  {  glGetShaderInfoLog(vertexShader, 512, NULL, infoLog);  std::cout << "ERROR::SHADER::VERTEX::COMPILATION\_FAILED\n" << infoLog << std::endl;  } |

Fragment shader

-픽셀의 출력 컬러값을 계산하는 쉐이더

-컬러는 RGBA순으로 설정됨

|  |
| --- |
| #version ~~330~~ 460 core  out vec4 FragColor;  void main()  {  FragColor = vec4(1.0f, 0.5f, 0.2f, 1.0f);  } |

-오직 하나의 출력 변수를 가짐(색)

-out으로 출력변수를 선언

-main에 출력변수를 할당함

-Fragment Shader 컴파일(vertex와 비슷함)

|  |
| --- |
| unsigned int fragmentShader;  fragmentShader = glCreateShader(GL\_FRAGMENT\_SHADER);  glShaderSource(fragmentShader, 1, &fragmentShaderSource, NULL);  glCompileShader(fragmentShader); |

Shader Program

-여러 쉐이더를 결합한 마지막 연결된 버전

-컴파일된 쉐이더들을 program객체로 link(연결)해줘야함

-오브젝트를 렌더링할 때, program을 활성화 시키면 됨

-program 객체 생성

|  |
| --- |
| unsigned int shaderProgram;  shaderProgram = glCreateProgram(); |

-객체첨부

|  |
| --- |
| glAttachShader(shaderProgram, vertexShader);  glAttachShader(shaderProgram, fragmentShader);  glLinkProgram(shaderProgram); |

-glAttachShader(program객체, 쉐이더객체) : shader들을 program에 붙힘

-glLinkProgram(shaderProgram) : 연결

-오류검사

|  |
| --- |
| glGetProgramiv(shaderProgram, GL\_LINK\_STATUS, &success);  if(!success) {  glGetProgramInfoLog(shaderProgram, 512, NULL, infoLog);  ...  } |

-프로그램 사용

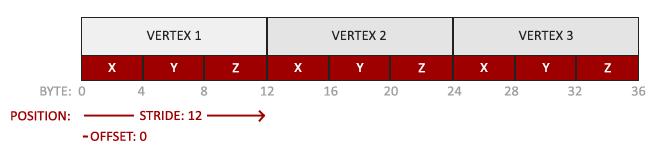
|  |
| --- |
| glUseProgram(shaderProgram); |

-쉐이더를 프로그램에 연결하게 되면, shader객체는 필요가 없음으로 삭제해야함

|  |
| --- |
| glDeleteShader(vertexShader);  glDeleteShader(fragmentShader); |

Linking Vertex Attributes

-vertex buffer데이터의 형식

 -위치 데이터는 4바이트(32비트)의 실수형 점

-각 위치는 3가지 값으로 구성

-각 3개 값 사이에 공백은 없음

-데이터 첫 번째 값은 버퍼의 시작 지점에 있음

-위 내용으로 OpenGL에게 vertex데이터의 해석방법을 알려줄 수 있음

|  |
| --- |
| glVertexAttribPointer(0, 3, GL\_FLOAT, GL\_FALSE, 3 \* sizeof(float), (void\*)0);  glEnableVertexAttribArray(0); |

-glVertexAttribPointer()

-1번 매개변수 : vertex속성, VertexShaderSource에 location = 0으로 설정했기에 vertex속성의 위치를 0으로 지정해놨음, 이를 사용

-2번 매개변수 : vertex 속성의 크기, vec3임으로 3개의 값으로 이루어져있음

-3번 매개변수 : 데이터 타입

-4번 매개변수 : 정규화 여부 (TRUE일 경우 0(부호를 가지면 -1), 1사이에 있지 않은 데이터 값들이 그 사이의 값들로 매핑)

-5번 매개변수 : stride라고 불림, vertex 속성 세트들 사이으 ㅣ공백을 알려줌, 다음 포지션의 vertex데이터 세트는 정확하게 float타입 3개만큼 뒤에 떨어져있음

-6번 매개변수 : void\*로 형 변환이 필요함, 버퍼에서 시작하는 위치의 offset

-OpenGL의 오브젝트를 그리는 형식

|  |
| --- |
| // 0. 정점 배열을 OpenGL에서 사용하기 위해 버퍼에 복사  glBindBuffer(GL\_ARRAY\_BUFFER, VBO);  glBufferData(GL\_ARRAY\_BUFFER, sizeof(vertices), vertices, GL\_STATIC\_DRAW);  // 1. 그런 다음 vertex 속성 포인터를 설정  glVertexAttribPointer(0, 3, GL\_FLOAT, GL\_FALSE, 3 \* sizeof(float), (void\*)0);  glEnableVertexAttribArray(0);  // 2. 오브젝트를 그리고 싶을 때 우리가 생성한 shader program을 사용  glUseProgram(shaderProgram);  // 3. 이제 오브젝트를 그림  someOpenGLFunctionThatDrawsOurTriangle(); |

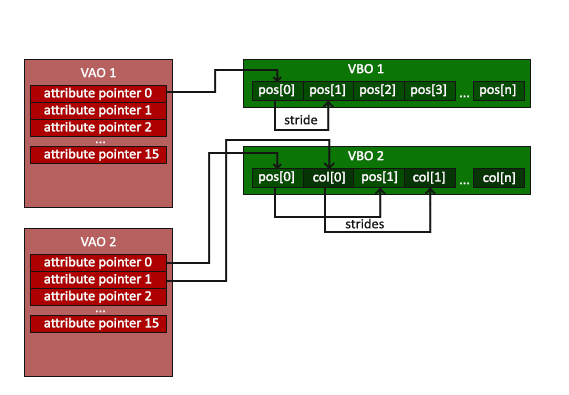
Vertex Array Object VAO

-VBO처럼 바인딩 될 수 있음

-VAO는 속성 포인터를 구성할 떄, 한번만 호출하면 됨

-오브젝트를 그릴 때 마다 해당 VAO를 바인딩 하기만하면 됨

-VAO바인딩 실패 시, OpenGL은 그리지 않음

-VAO생성

|  |
| --- |
| unsigned int VAO;  glGenVertexArrays(1, &VAO); |

-오브젝트를 그리기 전 세팅

|  |
| --- |
| //초기화 코드 (한번만 실행됩니다(오브젝트가 자주 변경되지 않는 한))  // 1. Vertex Array Object 바인딩  glBindVertexArray(VAO);  // 2. OpenGL이 사용하기 위해 vertex 리스트를 복사  glBindBuffer(GL\_ARRAY\_BUFFER, VBO);  glBufferData(GL\_ARRAY\_BUFFER, sizeof(vertices), vertices, GL\_STATIC\_DRAW);  // 3. 그런 다음 vertex 속성 포인터를 세팅  glVertexAttribPointer(0, 3, GL\_FLOAT, GL\_FALSE, 3 \* sizeof(float), (void\*)0);  glEnableVertexAttribArray(0);  [...]  //드로잉 코드 (렌더링 루프 내부)  // 4. 오브젝트를 그립니다.  glUseProgram(shaderProgram);  glBindVertexArray(VAO);  someOpenGLFunctionThatDrawsOurTriangle(); |

The triangle we've all been waiting for

-오브젝트를 그리기 위한 함수 glDrawArrays

|  |
| --- |
| glUseProgram(shaderProgram);  glBindVertexArray(VAO);  glDrawArrays(GL\_TRIANGLES, 0, 3); |

-glDrawArrays(primitive유형, vertex배열의 시작 인덱스, 정점의 수)

Element Buffer Objects EBO

-사각형을 그린다고 생각해보자, 삼각형 2개로 만들 수 있음

우측 상단

좌측 상단

좌측 하단

우측 하단

2

1

|  |  |
| --- | --- |
| float vertices[] = {  // 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  }; |  |

-중복되는 vertex가 있음 (좌측 상단, 우측 하단)

-EBO는 정점의 인덱스들을 저장해서 중복을 없앰

-고유한 정점들을 지정, 이 정점들의 인덱스를 지정해야함

|  |
| --- |
| float vertices[] = {  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 indices[] = { // 0부터 시작한다는 것을 명심  0, 1, 3, // 첫 번째 삼각형  1, 2, 3 // 두 번째 삼각형  }; |

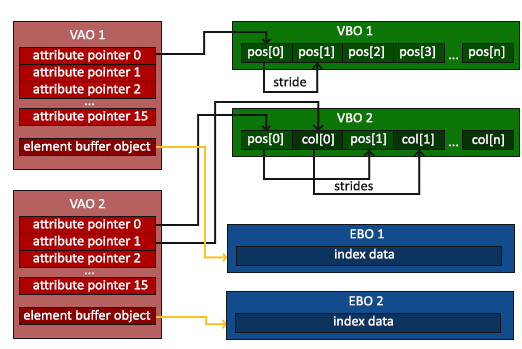
-EBO 생성

|  |
| --- |
| unsigned int EBO;  glGenBuffers(1, &EBO); |
| glBindBuffer(GL\_ELEMENT\_ARRAY\_BUFFER, EBO);  glBufferData(GL\_ELEMENT\_ARRAY\_BUFFER, sizeof(indices), indices, GL\_STATIC\_DRAW); |

-이제 glDrawArrays를 glDrawElements로 변경

|  |
| --- |
| glBindBuffer(GL\_ELEMENT\_ARRAY\_BUFFER, EBO);  glDrawElements(GL\_TRIANGLES, 6, GL\_UNSIGNED\_INT, 0); |

glDrawElements(그리기 모드, vertex갯수, 인덱스 타입, offset(첫번째 index) );

-VAO를 바인딩 하기 전에 EBO를 번저 하면 안됨, 순서가 중요함

-초기화, 드로잉 코드

|  |
| --- |
| //초기화 코드  // 1. Vertex Array Object 바인딩  glBindVertexArray(VAO);  // 2. OpenGL이 사용하기 위해 vertex 리스트를 vertex 버퍼에 복사  glBindBuffer(GL\_ARRAY\_BUFFER, VBO);  glBufferData(GL\_ARRAY\_BUFFER, sizeof(vertices), vertices, GL\_STATIC\_DRAW);  // 3. OpenGL이 사용하기 위해 인덱스 리스트를 element 버퍼에 복사  glBindBuffer(GL\_ELEMENT\_ARRAY\_BUFFER, EBO);  glBufferData(GL\_ELEMENT\_ARRAY\_BUFFER, sizeof(indices), indices, GL\_STATIC\_DRAW);  // 4. 그런 다음 vertex 속성 포인터를 세팅  glVertexAttribPointer(0, 3, GL\_FLOAT, GL\_FALSE, 3 \* sizeof(float), (void\*)0);  glEnableVertexAttribArray(0);  [...]  //드로잉 코드 (렌더링 루프 내부)  glUseProgram(shaderProgram);  glBindVertexArray(VAO);  glDrawElements(GL\_TRIANGLES, 6, GL\_UNSIGNED\_INT, 0)  glBindVertexArray(0); |