Advanced Lighting // Point shadow

-저번에 배운 shadow mapping에서는 directional light에 의해 생성되는 directional shadow mapping을 공부하였음

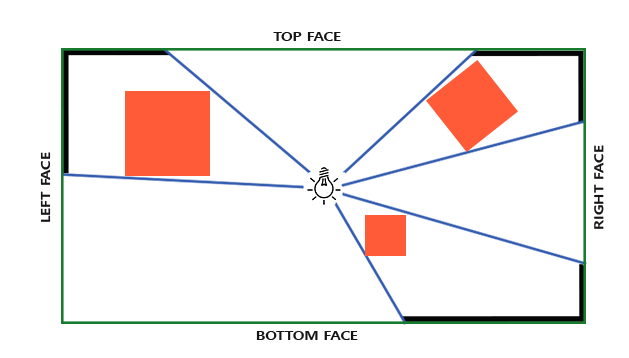
-이번에는 점광원이 있는 point shadows(light) 또는 omnidirectional shadow map을 배움

-알고리즘은 거의 똑같음

-광원의 관점에서 깊이 맵을 생성, fragment 위치를 기반으로 깊이 맵을 샘플링, 각 fragment를 깊이 맵에 저장된 값과 비교해 그림자에 있는가를 판단

-그러나 이번에는 2D 깊이 맵을 사용하는 것이 아닌, 큐브 맵을 사용할 것임

-이전에는 한 방향으로만 바라본 깊이 값을 생성하였으나, 이번에는 모든 방향으로 봐야함

-큐브 맵은 6개의 면을 가지고 있기 때문에, 각각 면에 대해 장면을 렌더링하고, 깊이 값으로 샘플링하는 것이 가능함

Generating the depth cubemap

-대락적인 렌더링 방법 / 6개의 면에 대해서 렌더링 하는 것

|  |
| --- |
| for(unsigned int i = 0; i < 6; i++)  {  GLenum face = GL\_TEXTURE\_CUBE\_MAP\_POSITIVE\_X + i;  glFramebufferTexture2D(GL\_FRAMEBUFFER, GL\_DEPTH\_ATTACHMENT, face,  depthCubemap, 0);  BindViewMatrix(lightViewMatrices[i]);  RenderScene();  } |

-이 방법을 사용하면 많은 렌더 호출이 필요하기에 자원이 많이 들어감

-geometry shader을 사용해서 한 번의 패스로 사용하는 대안적인 접근법을 사용할 것임

-큐브맵 생성

|  |
| --- |
| unsigned int depthCubemap;  glGenTextures(1, &depthCubemap); |

-하나의 큐브맵 평면을 2D 깊이 값 텍스처 이미지로 생성함

|  |
| --- |
| const unsigned int SHADOW\_WIDTH = 1024, SHADOW\_HEIGHT = 1024;  glBindTexture(GL\_TEXTURE\_CUBE\_MAP, depthCubemap);  for (unsigned int i = 0; i < 6; ++i)  glTexImage2D(GL\_TEXTURE\_CUBE\_MAP\_POSITIVE\_X + i, 0, GL\_DEPTH\_COMPONENT,  SHADOW\_WIDTH, SHADOW\_HEIGHT, 0,  GL\_DEPTH\_COMPONENT, GL\_FLOAT, NULL); |

-텍스처 매개변수 설정

|  |
| --- |
| glTexParameteri(GL\_TEXTURE\_CUBE\_MAP, GL\_TEXTURE\_MAG\_FILTER, GL\_NEAREST);  glTexParameteri(GL\_TEXTURE\_CUBE\_MAP, GL\_TEXTURE\_MIN\_FILTER, GL\_NEAREST);  glTexParameteri(GL\_TEXTURE\_CUBE\_MAP, GL\_TEXTURE\_WRAP\_S, GL\_CLAMP\_TO\_EDGE);  glTexParameteri(GL\_TEXTURE\_CUBE\_MAP, GL\_TEXTURE\_WRAP\_T, GL\_CLAMP\_TO\_EDGE);  glTexParameteri(GL\_TEXTURE\_CUBE\_MAP, GL\_TEXTURE\_WRAP\_R, GL\_CLAMP\_TO\_EDGE); |

-일반적으로 큐브맵 텍스처의 한 면을 프레임 버퍼에 첨부, 프레임 버퍼의 깊이 버퍼 대상을 다른 큐브의 면으로 바꿀 때마다 장면을 6번 렌더링 하게 됨, 우리는 한 번의 패스로 모든 면에 렌더할 수 있는 geometry shader을 사용할 것이기 때문에 glFramebufferTexture을 사용해 큐브맵을 프레임 버퍼의 depth attachment로 직접 첨부가 가능함

|  |
| --- |
| glBindFramebuffer(GL\_FRAMEBUFFER, depthMapFBO);  glFramebufferTexture(GL\_FRAMEBUFFER, GL\_DEPTH\_ATTACHMENT, depthCubemap, 0);  glDrawBuffer(GL\_NONE); // 깊이 값만 신경쎠서  glReadBuffer(GL\_NONE); // 사용할 것이라고 명시  glBindFramebuffer(GL\_FRAMEBUFFER, 0); |

-omnidirectional shadow map에는 2개의 렌더링 패스가 있음

-1. 깊이 맵을 생성

-2. 렌더 패스에서 깊이 맵을 사용해 그림자를 만듬

-전체적인 과정

|  |
| --- |
| // 1. first render to depth cubemap  glViewport(0, 0, SHADOW\_WIDTH, SHADOW\_HEIGHT);  glBindFramebuffer(GL\_FRAMEBUFFER, depthMapFBO);  glClear(GL\_DEPTH\_BUFFER\_BIT);  ConfigureShaderAndMatrices();  RenderScene();  glBindFramebuffer(GL\_FRAMEBUFFER, 0);  // 2. then render scene as normal with shadow mapping (using depth cubemap)  glViewport(0, 0, SCR\_WIDTH, SCR\_HEIGHT);  glClear(GL\_COLOR\_BUFFER\_BIT | GL\_DEPTH\_BUFFER\_BIT);  ConfigureShaderAndMatrices();  glBindTexture(GL\_TEXTURE\_CUBE\_MAP, depthCubemap);  RenderScene(); |

-모든 방향에서 장면을 렌더링하기 위해서 적절한 변환 행렬이 필요할 것임

Light space transform

-큐브 맵을 사용하면 모든 장면의 geometry를 모든 방향 6방향에서 빛 공간으로 바꿀 수 있는 방법이 필요함

-이를 위해서 light space 변환 행렬 T를 만들 것임

-변환 행렬은 projection, view를 모두 포함하고 있음

-광원은 공간의 한 point이기에 원근 투영이 적합함

|  |
| --- |
| float aspect = (float)SHADOW\_WIDTH/(float)SHADOW\_HEIGHT;  float near = 1.0f;  float far = 25.0f;  glm::mat4 shadowProj = glm::perspective(glm::radians(90.0f), aspect, near, far); |

-perspective에서의 시야를 90도로 설정하는 것이 중요함, 90도로 설정해야 각 면이 가장자리에서 서로 올바르게 정렬이 됨

-glm::lookAt을 이용해 6개의 view 방향을 설정

|  |
| --- |
| std::vector<glm::mat4> shadowTransforms;  shadowTransforms.push\_back(shadowProj \*  glm::lookAt(lightPos, lightPos + glm::vec3( 1.0, 0.0, 0.0), glm::vec3(0.0,-1.0, 0.0));  shadowTransforms.push\_back(shadowProj \*  glm::lookAt(lightPos, lightPos + glm::vec3(-1.0, 0.0, 0.0), glm::vec3(0.0,-1.0, 0.0));  shadowTransforms.push\_back(shadowProj \*  glm::lookAt(lightPos, lightPos + glm::vec3( 0.0, 1.0, 0.0), glm::vec3(0.0, 0.0, 1.0));  shadowTransforms.push\_back(shadowProj \*  glm::lookAt(lightPos, lightPos + glm::vec3( 0.0,-1.0, 0.0), glm::vec3(0.0, 0.0,-1.0));  shadowTransforms.push\_back(shadowProj \*  glm::lookAt(lightPos, lightPos + glm::vec3( 0.0, 0.0, 1.0), glm::vec3(0.0,-1.0, 0.0));  shadowTransforms.push\_back(shadowProj \*  glm::lookAt(lightPos, lightPos + glm::vec3( 0.0, 0.0,-1.0), glm::vec3(0.0,-1.0, 0.0)) |

-각각을 projection에 곱하여 총 6개의 다른 light space 변환 행렬을 얻을 수 있음

-각 변환행렬 T는 큐브 맵의 깊이를 렌더링하는 shader로 전송됨

Depth shaders

-필요한 쉐이더는 vertex shader, fragment shader, geometry shader가 필요함

-이중에서 geometry shader가 world space vertex들을 6개의 다른 light space로 변환해줌

-vertex shader

|  |
| --- |
| #version 460 core  layout (location = 0) in vec3 aPos;  uniform mat4 model;  void main()  {  gl\_Position = model \* vec4(aPos, 1.0);  } |

-geometry shader은 상각형의 정점과 light space 변환 행렬 T의 배열을 입력으로 받음

-각 shader을 light space로 변환해주는 역할을 함

-geometry shader에는 gl\_Layer라는 변수가 있음, 이것은 primitive 면을 내보낼 큐브 맵 면을 지정함

-이 변수를 업데이트 하면 각 primitive 요소에 대해 렌더링할 큐브맵 면을 제어할 수 잉씀

-cubemap이 연결되었을 때만 사용 가능

|  |
| --- |
| #version 460 core  layout (triangles) in;  layout (triangle\_strip, max\_vertices=18) out;  uniform mat4 shadowMatrices[6];  out vec4 FragPos; // FragPos from GS (output per emitvertex)  void main()  {  for(int face = 0; face < 6; ++face)  {  gl\_Layer = face; // built-in variable that specifies to which face we render.  for(int i = 0; i < 3; ++i) // for each triangle vertex  {  FragPos = gl\_in[i].gl\_Position;  gl\_Position = shadowMatrices[face] \* FragPos;  EmitVertex();  }  EndPrimitive();  }  } |

-삼각형을 입력으로 받아서 총 6개의 삼각형을 출력함(총 18개의 vertex)

-face를 gl\_Layer에 저장해서 각 면을 지정하는 6개의 큐브면을 반복하게 함

-FragPos에 각 변의 light space 변환 행렬 T를 곱해서 world space vertex들을 light space로 변환해서 삼각형을 생성함

-결과 값인 FragPos 변수를 깊이 계산하는데 사용할 fragment shader로 전송함

-fragment shader에서 각 fragment의 위치과 광원 위치 사이의 직선 서리로 깊이를 계산함

|  |
| --- |
| #version 460 core  in vec4 FragPos;  uniform vec3 lightPos;  uniform float far\_plane;  void main()  {  // fragment와 광원 사이의 거리  float lightDistance = length(FragPos.xyz - lightPos);    // 거리를 0~1의 값으로 변경  lightDistance = lightDistance / far\_plane;    // 변환한 거리값을 depth로 사용  gl\_FragDepth = lightDistance;  } |