Advanced Lighting // Deferred Shading

-지금까지 사용한 조명은 Forward Rendering, Forward Shading 이라고 함

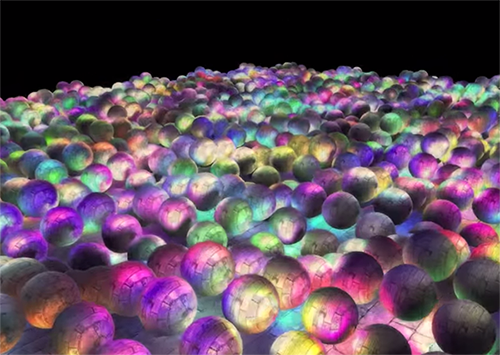
-객체를 렌더링하고 장면의 모든 광원에 따라 조명을 비추는 방법

-렌더링이 모든 fragment에 대해서 각 광원에 대해서 반복해야 해서 자원을 많이 잡아먹음

-복잡한 장면일 때, 가려지는 픽셀에 대한 많은 fragment shader 실행을 낭비함

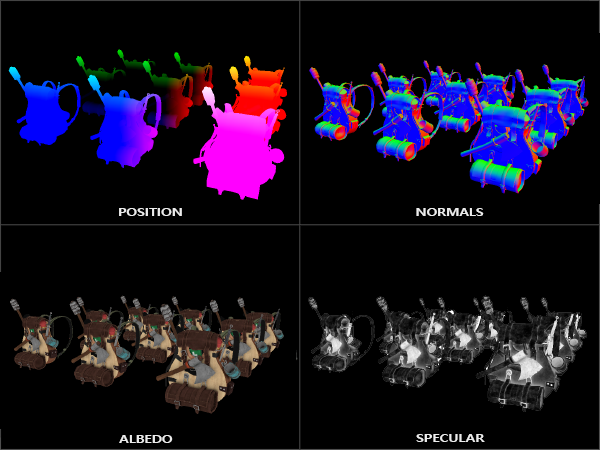
-Deferred Shading, Rendering은 이러한 방법을 해결하기 위해 사용함

-1847개의 point lighting이 있는 장면, Forward Rendering으로는 만드는 것이 불가능 할 것임, 엄청난 버벅임이 있을 것으로 예상됨

-Deferred Shading은 무거운 렌더링의 대부분을 이후 단계로 미루는 것임

-두 번의 단계로 이루어짐

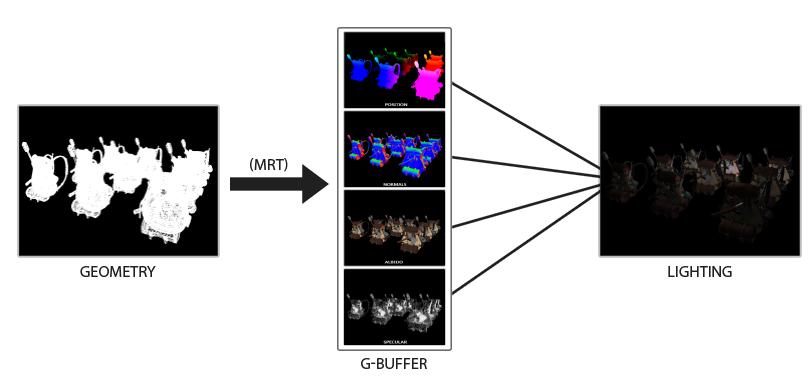
-첫 번째는 형상 패스라고 하며, 장면을 한 번 렌더링 하고 G-Buffer라고 하는 텍스처 모음에 저장한 모든 종류의 기하학적 정보를 검색함



-두 번째는 screen-filled 쿼드를 렌더링하고, G-buffer에 저장된 기하학적 정보를 사용해서 각 조각에 대한 장면의 조명을 계산하는 라이팅 패스라고 함

-G-buffer의 텍스처를 사용하는데, 픽셀 단위로 버퍼를 반복시킴

-Deferred Shading의 전체 과정

-이렇게 하면 조명단계에서 각 픽셀마다 단 한 번의 처리로 이루어짐

-사용하지 않는 많은 렌더링 호출로 부터 절약이 가능해짐

-그러나 단점이 몇 가지 있는데, blending을 지원하지 않고, MSAA가 더 이상 작동하지 않음

-단점에 대한 해결방법도 따로 존재함(이번 장의 맨 뒤에서 배울 예정)

The G-Buffer

-최종 조명 패스에 대한 조명 관련 데이터를 저장하는데 사용되는 모든 텍스처의 통칭

-필요한 데이터

-3D World space 위치 벡터 / lightDir, viewDir사용된 FragPos를 계산

-RGB Diffuse color 벡터 / albedo

-3D Normal 벡터 / 표면의 기울기를 결정

-Specular Intensity Float

-모든 광원의 위치, 색상 벡터

-사용자의 위치 벡터

-이러한 변수들을 통해서 Blinn-Phong 조명을 계산할 수 있음

-OpenGL은 텍스처에 저장할 수 있는 것에 제한이 없음

-G-buffer라는 텍스처에 모든 fragment 데이터를 저장하고, 나중에 조명패스에서 사용하는 것이 좋음

-렌더링 루프의 pesudo 코드

|  |
| --- |
| while(...) // render loop  {  // 1. geometry pass: render all geometric/color data to g-buffer  glBindFramebuffer(GL\_FRAMEBUFFER, gBuffer);  glClearColor(0.0, 0.0, 0.0, 1.0); // keep it black so it doesn't leak into g-buffer  glClear(GL\_COLOR\_BUFFER\_BIT | GL\_DEPTH\_BUFFER\_BIT);  gBufferShader.use();  for(Object obj : Objects)  {  ConfigureShaderTransformsAndUniforms();  obj.Draw();  }  // 2. lighting pass: use g-buffer to calculate the scene's lighting  glBindFramebuffer(GL\_FRAMEBUFFER, 0);  lightingPassShader.use();  BindAllGBufferTextures();  SetLightingUniforms();  RenderQuad();  } |

-각 fragment에서 저장할 데이터는 위치 벡터, 법선 벡터, 색상 벡터, 반사 강도가 있음

-따라서 Geometry pass에서 장면의 모든 객체를 렌더링하고, 데이터 구성요소들을 G-buffer에 저장함, MRT을 사용해서 한 번의 렌더 패스에서 여러 색상 버퍼로 렌더링 함

-Geometry pass의 경우, 색상 버퍼가 연결된 G-buffer와 하나의 깊이 렌더 버퍼 객체를 직관적으로 호출할 프레임 버퍼를 초기화 해야함

-위치, 법선 텍스처의 경우, 고 정밀 텍스처(16, 32비트 부동 소수점)를 사용하는 것이 좋음

-albedo, 반사 정도는 기본 텍스처 정밀도(8비트)로도 충분함

|  |
| --- |
| unsigned int gBuffer;  glGenFramebuffers(1, &gBuffer);  glBindFramebuffer(GL\_FRAMEBUFFER, gBuffer);  unsigned int gPosition, gNormal, gColorSpec;    // - position color buffer  glGenTextures(1, &gPosition);  glBindTexture(GL\_TEXTURE\_2D, gPosition);  glTexImage2D(GL\_TEXTURE\_2D, 0, GL\_RGBA16F, SCR\_WIDTH, SCR\_HEIGHT, 0,  GL\_RGBA, GL\_FLOAT, NULL);  glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_MIN\_FILTER, GL\_NEAREST);  glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_MAG\_FILTER, GL\_NEAREST);  glFramebufferTexture2D(GL\_FRAMEBUFFER, GL\_COLOR\_ATTACHMENT0,  GL\_TEXTURE\_2D, gPosition, 0);    // - normal color buffer  glGenTextures(1, &gNormal);  glBindTexture(GL\_TEXTURE\_2D, gNormal);  glTexImage2D(GL\_TEXTURE\_2D, 0, GL\_RGBA16F, SCR\_WIDTH, SCR\_HEIGHT, 0,  GL\_RGBA, GL\_FLOAT, NULL);  glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_MIN\_FILTER, GL\_NEAREST);  glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_MAG\_FILTER, GL\_NEAREST);  glFramebufferTexture2D(GL\_FRAMEBUFFER, GL\_COLOR\_ATTACHMENT1,  GL\_TEXTURE\_2D, gNormal, 0);    // - color + specular color buffer  glGenTextures(1, &gAlbedoSpec);  glBindTexture(GL\_TEXTURE\_2D, gAlbedoSpec);  glTexImage2D(GL\_TEXTURE\_2D, 0, GL\_RGBA, SCR\_WIDTH, SCR\_HEIGHT, 0,  GL\_RGBA, GL\_UNSIGNED\_BYTE, NULL);  glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_MIN\_FILTER, GL\_NEAREST);  glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_MAG\_FILTER, GL\_NEAREST);  glFramebufferTexture2D(GL\_FRAMEBUFFER, GL\_COLOR\_ATTACHMENT2,  GL\_TEXTURE\_2D, gAlbedoSpec, 0);    // - tell OpenGL which color attachments we'll use (of this framebuffer) for rendering  unsigned int attachments[3] = { GL\_COLOR\_ATTACHMENT0,  GL\_COLOR\_ATTACHMENT1,  GL\_COLOR\_ATTACHMENT2 };  glDrawBuffers(3, attachments);    // then also add render buffer object as depth buffer and check for completeness.  [...] |

-여러 렌더 타겟을 사용하기 때문에, G-Buffer와 관련된 컬러 버퍼 중 어떤 것을 렌더링 할지 OpenGl에게 알려줘야 함

-색상과 반사강도를 RGBA 텍스처로 저장하는데, 이렇게 하면 추가 색상 버퍼 텍스처를 선언할 필요가 없음

-다음으로 G-buffer로 렌더링 하기 위한 쉐이더

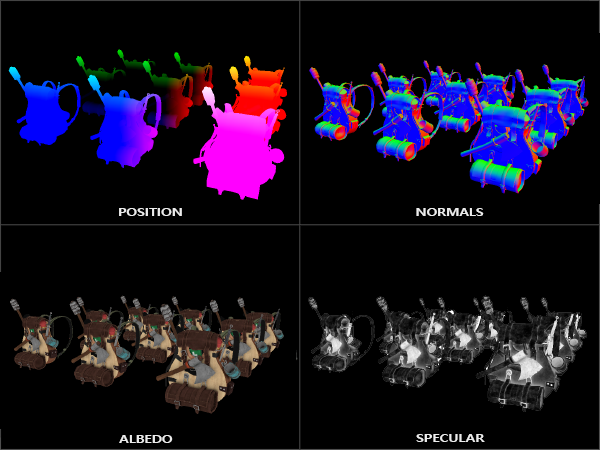
-g-buffer의 Fragment shader

|  |
| --- |
| #version 460 core  layout (location = 0) out vec3 gPosition;  layout (location = 1) out vec3 gNormal;  layout (location = 2) out vec4 gAlbedoSpec;  in vec2 TexCoords;  in vec3 FragPos;  in vec3 Normal;  uniform sampler2D texture\_diffuse1;  uniform sampler2D texture\_specular1;  void main()  {  // store the fragment position vector in the first gbuffer texture  gPosition = FragPos;  // also store the per-fragment normals into the gbuffer  gNormal = normalize(Normal);  // and the diffuse per-fragment color  gAlbedoSpec.rgb = texture(texture\_diffuse1, TexCoords).rgb;  // store specular intensity in gAlbedoSpec's alpha component  gAlbedoSpec.a = texture(texture\_specular1, TexCoords).r;  } |

-여러 개의 렌더 타겟을 사용할 때, 레이아웃 지정자가 OpenGL에게 렌더링 할 현재 활성화된 프레임 버퍼의 컬러 버퍼를 알려줌

-빛 계산을 할 때는, 모든 변수를 동일한 좌표 공간에 유지하는 것이 중요하기 때문에 모든 변수를 World-Space에 저장, 계산함

-각 내용을 시각화 한 장면



The Deferred lighting pass

-G-buffer의 많은 Fragment Data를 이용해서 G-buffer 텍스처를 픽셀 단위로 반복하고, 조명 알고리즘에 대한 입력으로 장면의 최종 색상을 계산할 수 있는 옵션이 있음

-G-buffer의 텍스처 값은 모두 최종 변환된 값이라서 픽셀 당 한 번의 조명 작업만 수행함

-따라서 deferred shading이 효율적인 것임

-lighting pass의 경우 2D screen-filled 쿼드를 렌더링하고, 각 픽셀에 조명 쉐이더를 실행함

|  |
| --- |
| glClear(GL\_COLOR\_BUFFER\_BIT | GL\_DEPTH\_BUFFER\_BIT);  glActiveTexture(GL\_TEXTURE0);  glBindTexture(GL\_TEXTURE\_2D, gPosition);  glActiveTexture(GL\_TEXTURE1);  glBindTexture(GL\_TEXTURE\_2D, gNormal);  glActiveTexture(GL\_TEXTURE2);  glBindTexture(GL\_TEXTURE\_2D, gAlbedoSpec);  // also send light relevant uniforms  shaderLightingPass.use();  SendAllLightUniformsToShader(shaderLightingPass);  shaderLightingPass.setVec3("viewPos", camera.Position);  RenderQuad(); |

-렌더링 전에 G-buffer의 모든 텍스처를 바인딩, 조명 관련 uniform 변수를 쉐이더에 보냄

-조명 pass의 fragment shader

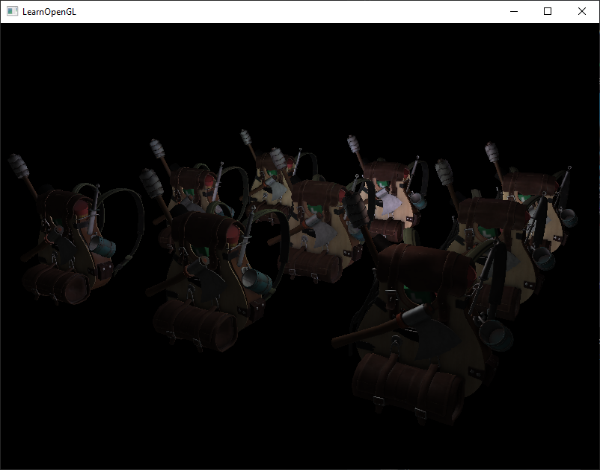
|  |
| --- |
| #version 460 core  out vec4 FragColor;    in vec2 TexCoords;  uniform sampler2D gPosition;  uniform sampler2D gNormal;  uniform sampler2D gAlbedoSpec;  struct Light {  vec3 Position;  vec3 Color;  };  const int NR\_LIGHTS = 32;  uniform Light lights[NR\_LIGHTS];  uniform vec3 viewPos;  void main()  {  // retrieve data from G-buffer  vec3 FragPos = texture(gPosition, TexCoords).rgb;  vec3 Normal = texture(gNormal, TexCoords).rgb;  vec3 Albedo = texture(gAlbedoSpec, TexCoords).rgb;  float Specular = texture(gAlbedoSpec, TexCoords).a;    // then calculate lighting as usual  vec3 lighting = Albedo \* 0.1; // hard-coded ambient component  vec3 viewDir = normalize(viewPos - FragPos);  for(int i = 0; i < NR\_LIGHTS; ++i)  {  // diffuse  vec3 lightDir = normalize(lights[i].Position - FragPos);  vec3 diffuse = max(dot(Normal, lightDir), 0.0) \* Albedo \* lights[i].Color;  lighting += diffuse;  }    FragColor = vec4(lighting, 1.0);  } |

-G-buffer을 나타내는 3개의 uniform 텍스처를 받고, geometry pass에 저장한 모든 데이터를 가지고 있음

-현재 fragment의 텍스처 좌표로 샘플링 한다면, geometry를 직접 렌더링 하는 것과 같은 결과를 얻을 수 있을 것임

-albedo랑 specular값을 gAlbedoSpec에서 가져옴

-간단한 데모를 실행



-G-buffer은 단일 fragment에서 왔기 때문에, 여러 조각의 조합인 Blending은 사용할 수 없음

-Deferred shading의 대부분이 장면 조명에 대해 동일한 조명 알고리즘을 사용해야 한다는 단점도 있음

-이러한 것들은 material-specific을 데이터에 포함함으로 다소 완화는 가능함

-이러한 단점을 극복하기 위해서 렌더러를 2개의 부분으로 나누는 경우가 많음

-1. deferred rendering

-2. 블랜딩을 위한 forward rendering