Advanced Lighting // Deferred Shading 2

Combining deferred rendering with forward rendering

-각각의 광원을 3D 큐브로 렌더링 할 것임

-생각나는 첫 번째 방법

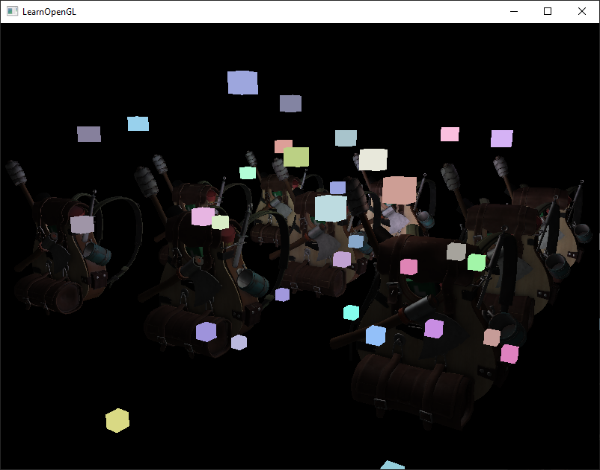
-광원을 deferred shading 파이프라인의 끝에서 모든 광원을 포워드 렌더링 하는 것

-deferred rendering이 끝나고 렌더링 하는 방법

-랜더링 코드

|  |
| --- |
| // deferred lighting pass  [...]  RenderQuad();    // now render all light cubes with forward rendering as we'd normally do  shaderLightBox.use();  shaderLightBox.setMat4("projection", projection);  shaderLightBox.setMat4("view", view);  for (unsigned int i = 0; i < lightPositions.size(); i++)  {  model = glm::mat4(1.0f);  model = glm::translate(model, lightPositions[i]);  model = glm::scale(model, glm::vec3(0.25f));  shaderLightBox.setMat4("model", model);  shaderLightBox.setVec3("lightColor", lightColors[i]);  RenderCube();  } |

-이 방법의 문제점 geometry 깊이를 고려하지 않음

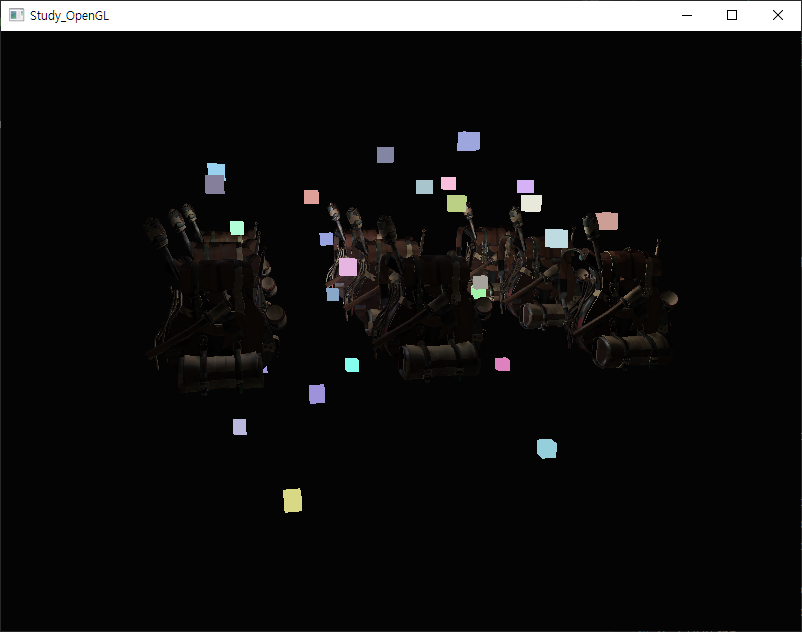


-여기서 해야 할 일은 geometry pass에 저장된 깊이 정보를 기본 프레임 버퍼의 깊이 버퍼에 복사해야함

-그 후 광원을 렌더링에 깊이가 적용이 됨, 그러나 geometry 위에 있을 때만 렌더링이 됨

-glBlitFramebuffer 함수를 사용해서 다른 프레임 버퍼로 내용을 복사

|  |
| --- |
| glBindFramebuffer(GL\_READ\_FRAMEBUFFER, gBuffer);  glBindFramebuffer(GL\_DRAW\_FRAMEBUFFER, 0); // write to default framebuffer  glBlitFramebuffer(  0, 0, SCR\_WIDTH, SCR\_HEIGHT, 0, 0, SCR\_WIDTH, SCR\_HEIGHT, GL\_DEPTH\_BUFFER\_BIT, GL\_NEAREST  );  glBindFramebuffer(GL\_FRAMEBUFFER, 0);  // now render light cubes as before  [...] |

-Blending을 하고 렌더링 할 수도 있음

A large number of lights

-deferred rendering은 큰 비용을 들이지 않고 많은 양의 광원을 렌더링 할 수 있음

-장면의 광원에 대한 각 fragment의 빛 성분을 계산해야하기 때문에, deferred rendering 자체로는 많은 양의 광원을 사용할 수 없음

-많은 양의 광원을 만들 수 있는 것은 deferred rendering pipeline은 매우 좋은 최적화임

-보통 큰 light scene에서 fragment를 렌더링할 때, 거리와의 상관없이 각 광원의 기여도를 계산하는데, 대부분의 fragment는 광원에 영향을 미치지 않는다는 낭비가 있음

-이것을 해결하기 위해서는 광원의 반지름, 즉 light가 fragment에 도달할 수 있는 영역을 계산하면 됨

-대부분은 감쇠를 사용하기 때문에, 계산이 가능함

Calculating a light's volume or radius

-반지름을 구하기 위해서는 감쇠 방정식의 해를 구해야함

-이것은 0.0이거나, 0.03과 같이 매우 어두움

-light의 volume 반지름을 구하는 계산식

-여기서 원하는 값은 Flight가 0.0이 될 때의 거리

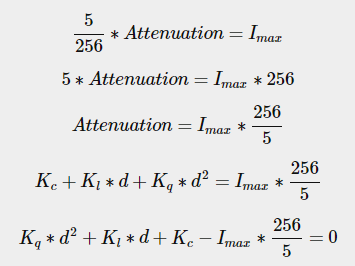
-그러나 이 값은 절대로 0이 될 수가 없기 때문에, 0.0에 가까우면서도 어둡다고 인식되는 값으로 해결함

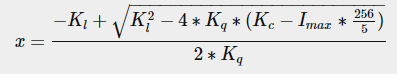
-이 scene에서 허용하는 밝기 값은 5/256이기 때문에 이 값을 사용함 / 기본 8비트 프레임 버퍼가 구성요소 당 여러 개의 강도를 표시할 수 있음으로 그것을 256으로 나눈 값임

-값이 작을수록 광량이 너무 커져서 최적화의 효과가 떨어짐, 값이 너무 크면 경계에서 갑자기 빛이 깨지는 것처럼 보이는 부자연스러움이 생김

-풀어야할 방정식

-여기서 I는 광원의 가장 밝은 색상 요소



-x는 위에 d와 같음

-이제 코드를 만들어보자

|  |
| --- |
| float constant = 1.0;  float linear = 0.7;  float quadratic = 1.8;  float lightMax = std::fmaxf(std::fmaxf(lightColor.r, lightColor.g), lightColor.b);  float radius =  (-linear + std::sqrtf(linear \* linear - 4 \* quadratic \* (constant - (256.0 / 5.0) \* lightMax)))  / (2 \* quadratic); |

-반경은 최대 강도를 기준으로 약 1.0~5.0의 값을 반환함

-lighting pass fragment shader

|  |
| --- |
| struct Light {  [...]  float Radius;  };    void main()  {  [...]  for(int i = 0; i < NR\_LIGHTS; ++i)  {  // calculate distance between light source and current fragment  float distance = length(lights[i].Position - FragPos);  if(distance < lights[i].Radius)  {  // do expensive lighting  [...]  }  }  } |

How we really use light volumes

-위에 fragment shader은 실제로 작동하지 않음, 그러나 volume을 어떻게 사용하는지를 보여줌

-GPU에서의 쉐이더 실행은 대부분의 아키텍처는 스레드를 대량으로 수집하기 위해서 효율적으로 동일한 쉐이더 코드를 실행해야 한다는 요구사항이 있음

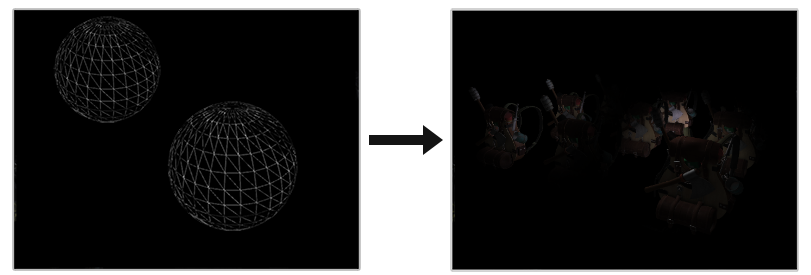
-쉐이더가 실행ㄷ외는 것을 보장하기 위해서 if문의 모든 분기를 실행하는 쉐이더가 실행됨으로 이전에 사용한 if문의 반지름 검사 최적화가 씀로가 없게됨

-결국 모든 광원에 대한 light을 계산하게 될 것임

-Light Volume을 사용하는 적절한 방법은 공처럼 light volume반경으로 축소해서 렌더링 하는 것임

-이 공의 중심에 광원이 존재하게 됨

-광원에 영향을 미치는 부분만 렌더링하게 될 것이고, 나머지 영향을 미치지 않는 픽셀들은 건너뛸 것임

-이것은 nr\_objects \* nr\_objects의 계산수가 nr\_object + nr\_object까지 계산을 줄여줌

-이 방법에 대한 문제점은 face culling 기능을 설정해야 하는 것임

-이것을 사용하면, light volume을 입력한 뒤에 volume이 더 이상 렌더링되지 않음

-그래서 stencil buffer을 이용해 해결이 가능함

-조명 볼륨을 렌더링하면 deferred shading보다 빠르지만, 최고의 최적화 방법은 아님

-가장 많이 사용하고 효율적인 방법은 deferred lighting과 tile-based deferred shading임

-이것은 많은 양의 광원을 렌더링 할 때 매우 효율적이고, 상대적으로 효율적인 MSAA를 허용함 – 이것에 대한 내용은 나중에 할 예정

Deferred rendering vs forward rendering

-light volume을 사용하지 않아도 각 픽셀 당 한번의 shader가 실행되기 때문에 픽셀 당 여러번의 shader을 실행하는 forward rendering보다는 효율적임

-그러나 큰 메모리 오버 해드, MSAA, Blending의 문제점이 있음

-scene이 작고 빛의 개수가 적으면 deferred rendering이 반드시 빠르지 않을 수 있고, 느릴때도 있음

-그러나 여전히 큰 scene과 많은양의 빛은 deferred rendering이 훨씬 빠름

-deferred rendering context에서 렌더링에서 얻는 많은 효과들은 구현이 가능함

-보통 translation 단계만 있으면 구현이 가능함

ex) deferred renderer에서 normal mapping을 하려는 경우, surface normal 대신 world-space에서 추출한 normal vector을 출력하도록 geometry pass shader을 변경하면 됨

ex) parallax mapping을 사용하려면 객체의 확산, 반사, 표준 텍스처를 샘플링하기 전에, geometry pass의 텍스처 좌표를 먼저 이동해야 함