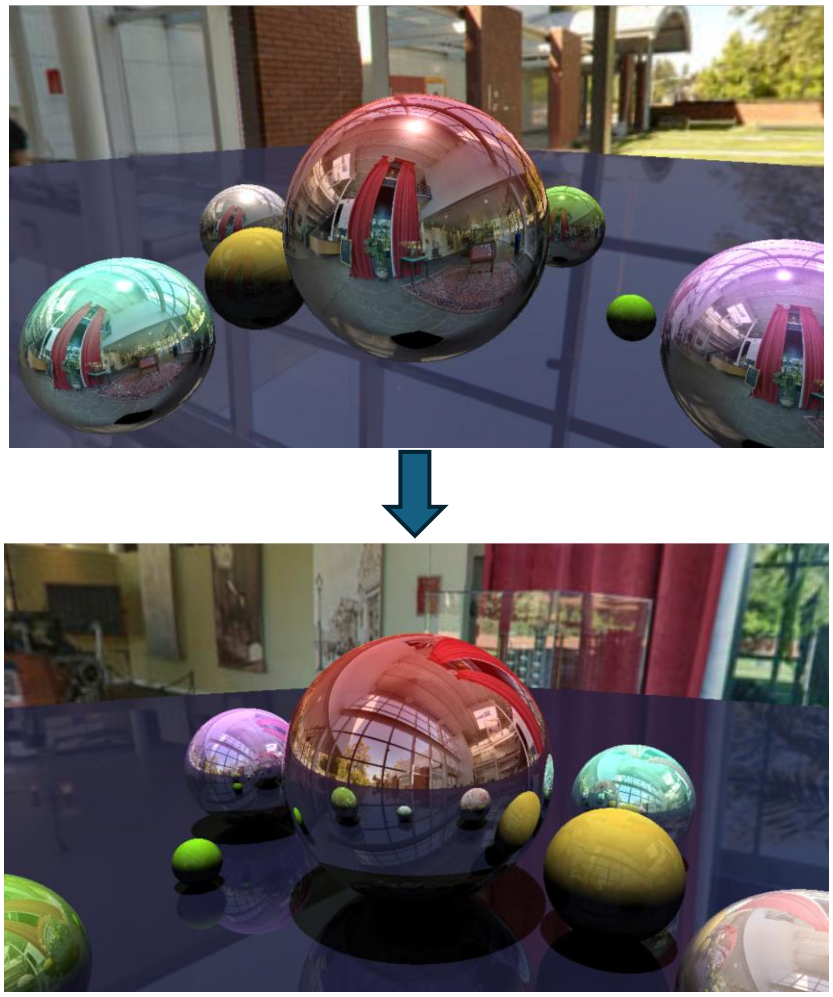
Homework#5: GPU-Based Raytracer만들기에 관한 보고

21011794 황재동(2025.12.22)

1. 개요:

일반적으로 Ray-tracer는 색 결정을 위해 장면 내 모든 도형 정보를 알고 있어야 하므로, 각 삼각형을 독립적으로 처리하는 표준 그래픽 파이프라인(Rasterization)에서는 구현이 어렵다. 그러나 Shader로 도형 정보를 전달할 수 있다면 레이트레이싱이 불가능한 것은 아니다. 본 과제에서는 그려야 할 도형이 구임을 가정하여 이를 GPU의 Fragment Shader 내에서 연산하는 레이트레이서를 구축하는 것을 목표로 한다. 특히 다중 반사(Multiple reflection)와 그림자(Shadow)를 구현하여 사실적인 렌더링 결과물을 생성한다.



2. 구현 방법 구상

2-1. Ray-Sphere Intersection 수학적 모델링

광선의 방정식 P(t) = O + tD와 구의 방정식 |P - C|^2 = R^2을 결합하여 t에 대한 2차 방정식을 도출한다. 판별식을 통해 광선과 구의 충돌 여부를 판단하고 가장 가까운 양의 실근을 찾아 충돌 지점의 좌표와 법선 벡터를 계산해야 한다.

2-2. 그림자 및 조명 연산

충돌 지점에서 광원(uLPos) 방향으로 그림자 광선(Shadow Ray)을 투사한다. 만약 이 광선이 광원에 도달하기 전 다른 구체와 부딪힌다면 해당 지점은 그림자 구역으로 판정하여 Diffuse와 Specular 항을 제외한다.

2-3. 다중 반사(Multiple Reflection) 구조

GLSL은 재귀 호출을 지원하지 않으므로 uBounceLimit만큼 반복문을 돌며 반사를 처리해야 한다. 충돌 지점에서 reflect() 함수를 사용해 반사 광선을 생성하고 반복적으로 색상을 누적한다. 광선이 구체에 부딪히지 않을 경우 큐브 맵을 참조하여 환경색을 반환한다

3. 구현 단계

3-1. IntersectRay 함수 구현

전체 구체 배열(uSpheres)을 순회하며 광선과의 교점을 계산한다. 2차 방정식의 계수를 계산할때 방향 벡터(ray.dir)가 정규화된 상태임을 활용하여 식을 단순화했다. 현재까지 발견된 t값 중 가장 작은 값을 유지하여 카메라와 가장 가까운 구체만 선택되도록 구현했다.

3-2. Shade 함수 및 그림자 처리

충돌 지점의 재질 정보와 법선을 받아 기본 색상을 계산한다. 이때 자신과의 충돌을 방지하기 위해 그림자 광선의 시작점에 미세한 오프셋을 주었다. IntersectRay가 true를 반환하면 그림자 상태로 간주하여 감쇠된 색상을 반환한다.

3-3. RayTracer

메인 루프에서는 최초 광선 투사 후 반사 횟수만큼 반복하며 k_s(반사 계수)를 누적 곱한다. 반사 광선이 구체에 부딪히면 그 지점의 Shade 값을 더하고, 부딪히지 않으면 uCube 환경 텍스처를 뽑아 최종 색상을 결정하도록 구현했다.

4. 시행착오 및 해결과정

4-1. 그림자 이상현상

그림자 광선을 생성할 때 충돌 지점에서 바로 시작하니 수치 오류로 인해 자기 자신과 즉시 충돌하여 표면에 검은 점들이 생기는 문제가 발생했다. 이를 해결하기 위해 광선의 시작 지점을 법선 방향으로 0.001만큼 띄워주니 깨끗한 그림자를 얻을 수 있었다.

4-2. 다중 반사 시 연쇄적 에너지 감쇠 구현 문제

반사가 반복될수록 빛의 에너지가 물리적으로 줄어들어야 하는데, 처음에는 단순히 각 지점의 Shade 값을 더하기만 해서 모델이 비정상적으로 밝아지는 현상이 있었다. 이를 해결하기 위해 루프 외부에서 누적 반사 계수 변수(k_s_accum)를 선언하고, 매 충돌마다 해당 지점에서 계산된 색상에 현재까지의 누적 계수를 곱해준 뒤, 다음 반사 단계를 위해 누적 계수에 새로 부딪힌 구체의 반사 계수인 h.mtl.k_s를 다시 곱하여 업데이트하는 방식으로 로직을 구성하였다. 이러한 연쇄적인 곱셈 처리를 통해 반사가 거듭될수록 빛이 자연스럽게 감쇠되는 효과를 줬다.