

컴퓨터그래픽스

과제 5 보고서



세종대학교

제출일	2025.12.23	전공	소프트웨어학과
과목	컴퓨터그래픽스	학번	21011746
담당교수	박상일 교수님	이름	양현석

1. GPU-Based Raytracer

a) 문제 분석

기존 Simplified Graphics Pipeline는 삼각형(Polygon) 단위로 처리되기 때문에, Ray Tracing과 같은 Global Illumination 효과나 반사, 그림자를 표현하는데 한계가 존재한다. 따라서 GPU 상에서 구를 대상으로 하는 Ray-Tracer를 구현한다. 마치 Unity Engine의 Ray-Cast처럼 동작하는 Ray를 활용하여, 구에 대한 교차 판정을 진행하고 이를 바탕으로 다중 반사와 그림자를 처리한다.

b) 문제 풀이 및 학습 내용

1) Intersection

Ray Tracing의 가장 기본이 되는 교차 판정이다. 구의 방정식과 Ray의 방정식을 연립하여 교차 여부를 판단해야 한다. 이 때 Ray의 방정식은 사실 상 벡터로 표현된 직선의 방정식이고, 구의 방정식도 마찬가지이다. 두 식을 연립하면 이차 방정식의 꼴이 도출되는데, 가장 간단한 아이디어로는 이차 방정식의 판별식을 이용하여 근이 존재하는 지를 판단하는 것이다. 근은 즉, 교점을 의미한다. 만일 근이 존재한다면, 카메라에서 가장 가까운 교차점을 찾아야 하므로, 양의 근 중 가장 작은 값을 선택할 수 있도록 한다. 이를 기반으로 HitInfo를 갱신한다. 또한 교차 여부를 반환한다.

구의 방정식과 Ray(직선)의 방정식에 대한 교점을 구하는 것은 구의 방정식에 직선의 방정식을 대입하고, 직선의 방정식의 거리 t 에 대한 식으로 세우면 된다.

$$\text{Ray : } \mathbf{P}(t) = \mathbf{O} + t\mathbf{D}$$

$$\text{Sphere : } |\mathbf{P} - \mathbf{C}|^2 = r^2$$

$$|(\mathbf{O} + t\mathbf{D}) - \mathbf{C}|^2 = r^2$$

$$\text{Let } \mathbf{oc} = \mathbf{O} - \mathbf{C}$$

$$|t\mathbf{D} + \mathbf{oc}|^2 = r^2$$

$$(t\mathbf{D} + \mathbf{oc}) \cdot (t\mathbf{D} + \mathbf{oc}) - r^2 = 0$$

$$t^2(\mathbf{D} \cdot \mathbf{D}) + 2t(\mathbf{D} \cdot \mathbf{oc}) + (\mathbf{oc} \cdot \mathbf{oc}) - r^2 = 0$$

위의 식을 가지고 일차항의 계수와 상수항을 구하여 판별식을 사용하면 다음과 같다.

```
vec3 oc = ray.pos - uSpheres[i].center.xyz;
float b = dot(ray.dir, oc);
float c = dot(oc, oc) - uSpheres[i].radius * uSpheres[i].radius;
```

```
// 판별식
float h = b*b - c;

if (h < 0.0) continue; // 교차하지 않음

// t 구하기 (가까운 점)
float t = -b - sqrt(h);

// 구체 내부 혹은 뒤쪽일 경우 처리
if (t < 0.0) t = -b + sqrt(h);

// 유효한 교차점이고 기존에 찾은 것보다 더 가까우면 갱신
if (t > 0.0 && t < hit.t) {
    hit.t = t;
    hit.position = vec4(ray.pos + t * ray.dir, 1.0);
    hit.normal = normalize(hit.position.xyz - uSpheres[i].center.xyz);
    hit.mtl = uSpheres[i].mtl;
    foundHit = true;
}
```

t 값은 근의 공식을 이용하여 구하고, 가장 가까운 지점을 찾기 위해 구의 개수만큼 반복문을 갱신하며 최저 값을 찾는다.

따라서 본 함수를 통해 HitInfo를 얻고 이를 다른 함수에서 사용할 수 있도록 한다.

2) Shade

이 함수에서는 그림자에 대한 처리를 진행한다. 그림자는 현재 지점에서 광원 방향으로 Ray를 쏘고, 이 Ray가 광원에 맞기 전에 다른 물체에 가려지는 지를 확인하여 구현한다. 만일 Ray가 다른 구와 교차하고, 그 교차 거리가 광원까지의 거리보다 작다면 그림자가 발생한 것이다. 이때 Diffuse와 Specular를 고려하지 않고, 오직 Ambient만 고려하여 색을 정한다. 이는 어렵게 만들기 위함이다. 또한 자기 자신과 교차하는 문제가 발생할 수 있기에 이를 방지하고자 표면에서 아주 살짝 법선 방향으로 시작 위치를 조정한다.

```
// Light Direction & Distance
vec3 lDir = uLPos.xyz - position.xyz;
float lDist = length(lDir);
lDir = normalize(lDir);

// Shadow Check
bool shadowed = false;
Ray shadowRay;
shadowRay.dir = lDir;

// 자기 충돌 방지 위해 표면에서 분리
shadowRay.pos = position.xyz + 0.0001f * normal;

HitInfo shadowHit;
// 레이가 무언가와 충돌했고, 그 충돌 지점이 광원보다 가까우면 그림자 발생
if (IntersectRay(shadowHit, shadowRay)) {
    if (shadowHit.t < lDist) {
```

```

        shadowed = true;
    }
}

```

이후에 만일 그림자가 발생하지 않았다면, Color에 Diffuse와 Specular를 더하여 처리한다.

3) RayTracer

다중 반사 및 Ray Trace를 구현하는 함수이다. GPU-based Ray Tracer는 반복문을 사용하여 BounceLimit 횟수만큼 반사를 추적하도록 구현하였다. 각 단계마다 reflect() 함수를 이용해 반사 벡터를 구하고, 새로운 Ray로 교차 판정을 반복적으로 수행한다. 만일 교차가 발생한다면 획득한 색상에 현재 물체의 k_s (Specular 계수)를 곱하여 누적해주는 방식으로 색상을 혼합할 수 있다. 이는 반사가 진행될수록 색상에 미치는 영향력이 줄어드는 물리적 현상을 모사한 것이다.

반복문 내부에서는 현재 Ray와 충돌 정보를 갱신하며 경로를 추적한다. 만약 새로운 물체와 충돌하면 해당 지점의 셰이딩 결과(Diffuse + Specular)에 현재까지 누적된 반사율(k_s)을 곱하여 최종 색상에 적용한다. 만일, 물체와 충돌하지 않고 나아갈 경우 Environment Map에서의 색상을 반사광으로 추가하고 루프를 바로 종료하도록 하였다. 이는 불필요한 연산을 방지하는 최적화이다.

또 반사율 누적값(k_s)가 0에 가까워지는 경우에는 더 이상 추적하는 의미가 없기에 이도 마찬가지로 break를 통해 반복문을 종료하도록 하였다. (누적값이 0에 가까우면 계산을 해도 그저 의미가 존재하지 않는다.)

```

// Compute reflections
vec4 k_s = hit.mtl.k_s; // 현재까지 누적된 반사율

// 현재 Ray 정보를 갱신하며 추적하기 위한 변수들
Ray currentRay = ray;
HitInfo currentHit = hit;

for ( int bounce=0; bounce<uBounceLimit; bounce++ )
{
    // 반사율이 너무 낮으면 계산 중단 (최적화)
    if ( k_s.r + k_s.g + k_s.b <= 0.0 ) break;

    Ray r;

    // 반사 벡터 계산
    r.dir = reflect(currentRay.dir, currentHit.normal);
    r.pos = currentHit.position.xyz + 0.001 * currentHit.normal; // Bias

    HitInfo h; // reflection hit info

    if ( IntersectRay( h, r ) )
    {
        // TO-DO: Hit found, so shade the hit point
        // TO-DO: Update the loop variables for tracing the next
        reflection ray

        // Hit found

```

```

    vec3 rView = normalize(-r.dir);
    // 반사된 지점의 색상을 계산하고 현재 반사율(current_ks)만큼 감쇠하여 더함
    clr += k_s * Shade(h.mtl, h.position, h.normal, rView);

    // 다음 바운스를 위해 반사율 누적 (곱하기)
    k_s *= h.mtl.k_s;

    // 다음 루프를 위해 현재 상태 업데이트
    currentRay = r;
    currentHit = h;
}

```

c) 시행착오 및 추가적인 학습 사항

사실 구현하는 것 자체는 어렵지 않았지만, 이 아이디어를 생각하기까지의 시간이 상당히 오래 걸렸습니다. 먼저 구의 방정식과 직선의 방정식을 벡터의 관점에서 이해하고 이를 가지고 교점을 찾는 것에 대한 아이디어가 가장 중요한 아이디어였습니다. 사실 교점에 대해서 생각은 해보았지만, 코드를 통해 근을 구해본 적이 없었습니다. 그러나 이차 방정식의 판별식과 근의 공식을 생각하게 되었고, 이는 본 문제를 푸는데에 핵심 포인트가 되었습니다.

또한 Shadow나 다중 반사 처리도 교수님께서 Ray-Tracing을 말씀하셨을 때, 또 이에 대한 중간고사 대비를 진행할 때 유튜브 강의와 다양한 자료를 참고하였고, 이해를 하고 있었기 때문에 특별한 시행착오는 없었습니다.

3학년까지 보내면서, 가장 전공답고 가장 흥미롭게 들은 과목인 것 같습니다. 교수님의 열정있는 강의와 다양한 예제, 영상들을 보면서 Computer Graphics에 대해 많은 지식을 얻을 수 있었습니다. 한 학기동안 정말 감사했습니다.

d) 참고 자료

- [How does Ray Tracing Work in Video Games and Movies? \(Youtube\)](#)
- [벡터를 이용한 원의 방정식 \(무무기행 부동산 동행\)](#)

이상입니다.