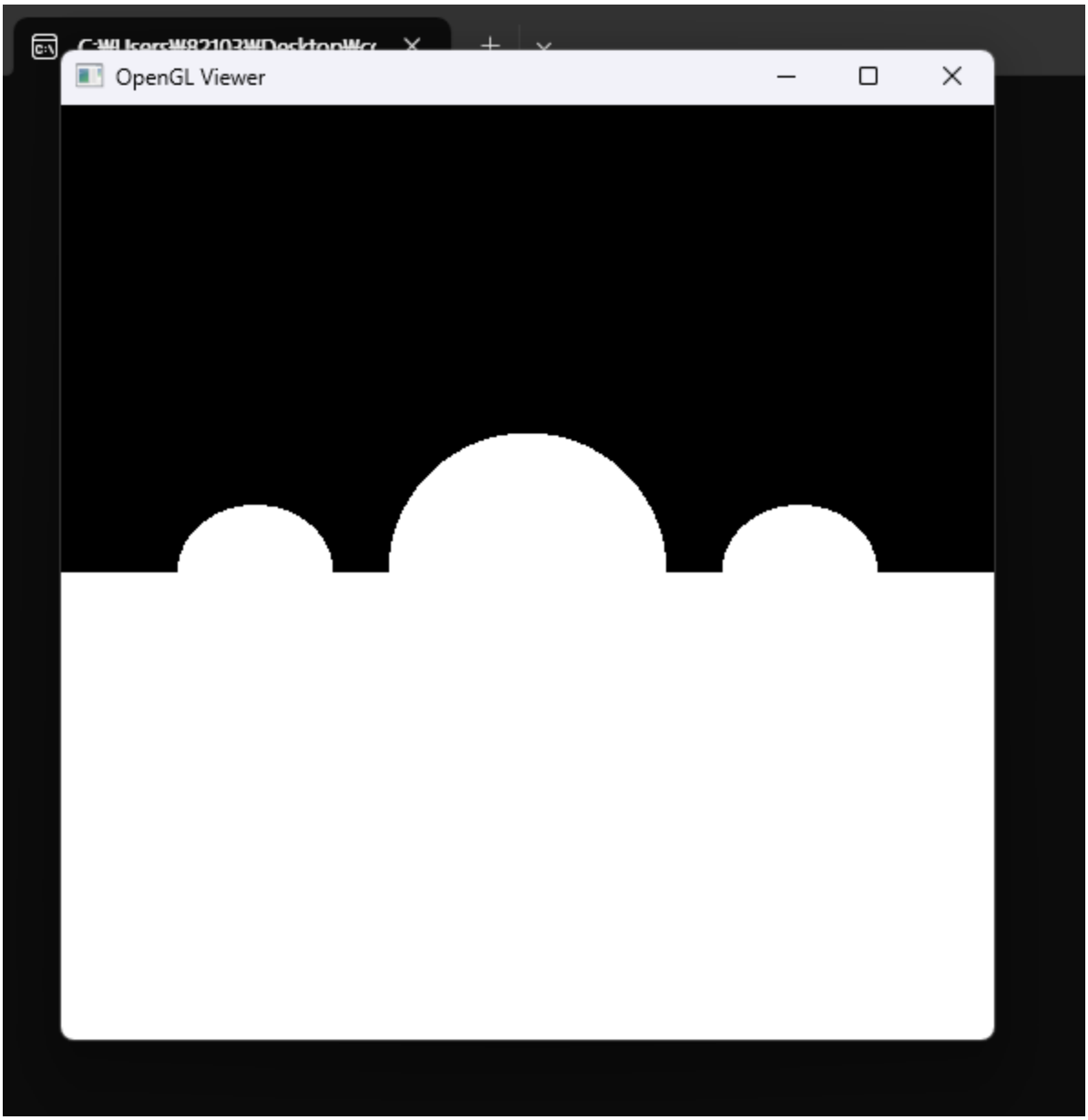
202112336 송수민 CG HW1 결과화면 + README 대용 레포트

결과화면



레포트

주제 : 레이트레이서를 작성하여, 한 평면과 세 개의 구 렌더링

구현 환경 : window 11에서 개발환경은 VS2022을 이용함.

구현 방법 : C++언어 이용. 교수님이 넣어주신 라이브러리를 이용하여 개발. 기존 코드에 github 코파일럿의 도움을 받아 코드 작성함. 작성 후 git LFS를 이용해 깃허브에 업로드. README 파일을 대신해 pdf 레포트 작성. 코드 실행 방법은 교수님이 제공해주신 코드에 추가된 부분, 변경된 부분 위주로 언급.

```
//라이브러리는 기존 코드와 동일
```

요구 조건	– Also assume that the image resolution is 512 × 512 (i.e., nx = ny = 512)
코드 내용	결과화면 전역변수로 고정.

```
int Width = 512;
int Height = 512;
```

요구 조건	• Implement the Following Classes: – Ray
코드 내용	Ray 클래스 작성. 시작점과 방향 선언

```
// class Ray
class Ray {
public:
        vec3 origin;
        vec3 direction;
};
```

요구 조건	Surface s = new Sphere(0.0, 0.0, 0.0), 1.0);
코드 내용	Surface 클래스 작성. 추후에 구와 평면이 해당 클래스를 상속받음. Intersect 함수는 ray객체와 광선의 시작(tMin)과 끝(tMax), 만나는 점을 저장하는 t값으로 구성. 상속받은 객체마다 식이 달라짐.

```
// class Surface and virtual bool intersect
class Surface {
public:
	virtual bool intersect(const Ray& ray, float tMin, float tMax, float& t) const = 0;
};
```

요구 조건	Make sphere class Calculates the intersection between each ray and each object in the scene
코드 내용	Sphere 클래스 작성. Sphere(const vec3& c, float r) : center(c), radius(r) {} 원의 중심과 반지름을 받음. Ray와 구의 방정식을 이용하여 이차방정식의 판별식 계산 (+,- 따로) (판별식) > 0면 t에 근의 공식의 값을 저장하고 intersect가 참 반환. 아니면 안만남으로 false반환.

```
// make Sphere using discriminant
class Sphere : public Surface {
public:
	vec3 center;
	float radius;

	Sphere(const vec3& c, float r) : center(c), radius(r) {}

	bool intersect(const Ray& ray, float tMin, float tMax, float& t) const override {
		vec3 oc = ray.origin - center;
		float a = dot(ray.direction, ray.direction);
		float b = 2.0f * dot(oc, ray.direction);
		float c = dot(oc, oc) - radius * radius;
		float discriminant = b * b - 4 * a * c;
		if (discriminant > 0) {
			float temp = (-b - std::sqrt(discriminant)) / (2.0f * a);
			if (temp < tMax && temp > tMin) {
				t = temp;
				return true;
			}
			temp = (-b + std::sqrt(discriminant)) / (2.0f * a);
			if (temp < tMax && temp > tMin) {
				t = temp;
				return true;
			}
		}
		return false;
	}
};
```

요구 조건	Make plain class Calculates the intersection between each ray and each object in the scene
코드 내용	Plane 클래스 작성. `Plane(const vec3& p, const vec3& n) : point(p), normal(normalize(n)) {}` 평면위의 한점과 정규화 된 법선벡터를 얻음. Ray와 내적한 값이 0보다 크면 (하지만 오류를 줄이기 위해 매우 작은 수 1e-6 설정) true반환. 이때 Ray시작점에서 해당 점까지 거리를 계산 후 t에 저장. (직선과 평면이 만나는 공식 이용) 내적이 0이면 평행하므로 false반환.

```
// class Plane
class Plane : public Surface {
public:
        vec3 point;
        vec3 normal;

        Plane(const vec3& p, const vec3& n) : point(p), normal(normalize(n)) {}

        bool intersect(const Ray& ray, float tMin, float tMax, float& t) const override {
                float denom = dot(normal, ray.direction);
                if (abs(denom) > 1e-6) {
                        vec3 p0l0 = point - ray.origin;
                        t = dot(p0l0, normal) / denom;
                        if (t >= tMin && t <= tMax) {
                                return true;
                        }
                }
                return false;
        }
};
```

아래 이어서..

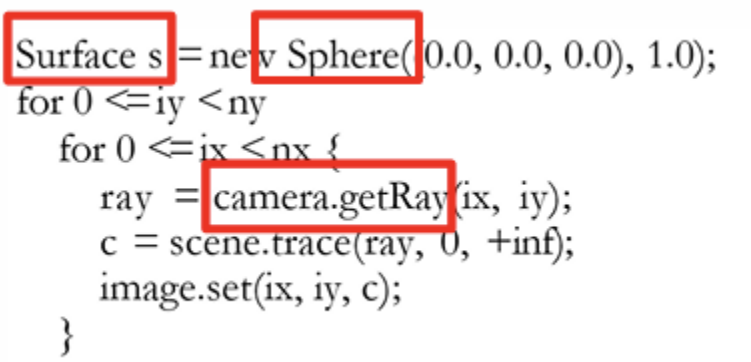
요구 조건	Camera Write a ray tracer which generates eye rays through the center of each pixel – Assume a perspective camera – eye point at e = (0, 0, 0) – orientation given by u = (1, 0, 0), v = (0, 1, 0) and w = (0, 0, 1). – (Note that the camera is looking along the direction −w.) – Assume that the viewing region on the image plane is defined by l = −0.1, r = 0.1, b = −0.1, t = 0.1, and d = 0.1
코드 내용	카메라 위치, 좌표축, 뷰포트 설정. 결과 크기에 맞게 뷰포트 좌표 변환을 하여 ray객체 반환. (ray생성 공식 이용)

```
// set Camera and get Ray
class Camera {
public:
	vec3 eye;
	vec3 u, v, w;
	float l, r, b, t, d;

	Camera() {
		eye = vec3(0.0f, 0.0f, 0.0f);
		u = vec3(1.0f, 0.0f, 0.0f);
		v = vec3(0.0f, 1.0f, 0.0f);
		w = vec3(0.0f, 0.0f, 1.0f);
		l = -0.1f;
		r = 0.1f;
		b = -0.1f;
		t = 0.1f;
		d = 0.1f;
	}

	Ray getRay(float i, float j) const {
		float u_coord = l + (r - l) * (i + 0.5f) / Width;
		float v_coord = b + (t - b) * (j + 0.5f) / Height;
		vec3 direction = normalize(u_coord * u + v_coord * v - d * w);
		return Ray{ eye, direction };
	}
};
```

요구 조건	Scene.trace(ray, tMin, tMax) {
	surface, t = surfs.intersect(ray, tMin, tMax);
	if (surface != null) return white;
	else return black;
	}
	For each pixel, if the corresponding ray intersects an object, set the pixel's color to white; otherwise, set the pixel's color to black.

코드 내용	Scene 클래스를 통해 ray와 물체가 만났다면 흰색, 아니면 검은색 반환. Trace 함수는 Ray가 각 물체마다 만나는 지 계산. 광선이 물체와 만나는 지점 중 가장 가까운 지점을 찾아 closest.t에 저장. 그리고 흰색 반환.

```
// class Scene if hit object return white else return black
class Scene {
public:
        std::vector<Surface*> surfaces;

        vec3 trace(const Ray& ray, float tMin, float tMax) const {
                float closest_t = tMax;
                const Surface* hit_surface = nullptr;
                for (const auto& surface : surfaces) {
                        float t;
                        if (surface->intersect(ray, tMin, closest_t, t)) {
                                closest_t = t;
                                hit_surface = surface;
                        }
                }
                if (hit_surface) {
                        return vec3(1.0f, 1.0f, 1.0f); // white
                }
                return vec3(0.0f, 0.0f, 0.0f); // black
        }
};
```

요구 조건	The scene consists of the following four objects:
	– Plane P , with equation y = –2.
	– Sphere S1, with center at (–4, 0, –7) and radius 1.
	– Sphere S2, with center at (0, 0, –7) and radius 2.
	– Sphere S3, with center at (4, 0, –7) and radiu
	 
	Surface s = new Sphere(0.0, 0.0, 0.0), 1.0);
	for 0 <= iy < ny
	for 0 <= ix < nx {
	ray = camera.getRay(ix, iy);
	c = scene.trace(ray, 0, +inf);
	image.set(ix, iy, c);
	}

코드 내용	카메라와 씬 생성. 클래스를 바탕으로 한 평면과 구 3개를 만듦. 픽셀마다 광선을 생성해 충돌검사를 하여 색깔 저장.

```
void render()
{

	//Create camera and scene
	Camera camera;
	Scene scene;
	scene.surfaces.push_back(new Plane(vec3(0.0f, -2.0f, 0.0f), vec3(0.0f, 1.0f, 0.0f))); //
Plane P
	scene.surfaces.push_back(new Sphere(vec3(-4.0f, 0.0f, -7.0f), 1.0f)); // Sphere S1
	scene.surfaces.push_back(new Sphere(vec3(0.0f, 0.0f, -7.0f), 2.0f)); // Sphere S2
	scene.surfaces.push_back(new Sphere(vec3(4.0f, 0.0f, -7.0f), 1.0f)); // Sphere S3

	OutputImage.clear();
	for (int j = 0; j < Height; ++j)
	{
		for (int i = 0; i < Width; ++i)
		{
			Ray ray = camera.getRay(i, j);
			vec3 color = scene.trace(ray, 0.0f, std::numeric_limits<float>::max());

			OutputImage.push_back(color.x); // R
			OutputImage.push_back(color.y); // G
			OutputImage.push_back(color.z); // B
		}
	}
}
```