Student Name: ChenYang Yu
ID: 862052273

Check_point_1 (34 points expected)

Camera.cpp

=> calculate the pixel in world (so we can get ray in the world)

vec3 Camera::World_Position(const ivec2& pixel_index)

vec2 PIX = Cell_Center(pixel_index); //

result = film_position + PIX[0]*horizontal_vector + PIX[1]*vertical_vector;

=> film_position is the original of film(the photo) horizontal_vector/vertical_vector are unit vector if the film

Render_world.cpp

=> calculate ray

void Render_World::Render_Pixel(const ivec2& pixel_index)

const vec3 CP = camera.position;

//care with zero vector?

const vec3 DIR = (camera.World_Position(pixel_index)-camera.position);

Ray ray(camera.position, DIR.normalized()); // just finish the declaration of ray

=> With ray, we can check every object in the world and see if there is any intersection

the information of Hit will return through the second input parameter

Object* Render_World::Closest_Intersection(const Ray& ray, Hit& hit)

For (every object o in the world)

if(o.intersection)

We have a hit, return o and save hit information in hit

Nothing intersection for all object in the world: return 0

=> return color depending on the ray is hit or not (the color of object, or the color of background)

vec3 Render_World::Cast_Ray(const Ray& ray,int recursion_depth)

```
Object* objInter = Closest_Intersection(ray,hit);
if( objInter != 0){
   vec3 norm = objInter->Normal(ray.Point(hit.t));
      if(hit.ray_exiting)
            norm *= (-1.0);  => light and object is on the same side, shouldn't see light
      color=objInter->material_shader->Shade_Surface(ray,ray.Point(hit.t),norm,1);
}else{
   vec3 dummy;
   color=background_shader->Shade_Surface(ray,dummy,dummy,1);
}
```

Sphere.cpp

=> for a given ray, check if there is any intersection with this sphere

bool Sphere::Intersection(const Ray& ray, std::vector<Hit>& hits) const

Check D = b^2 - 4ac

if(D<=0) return False;

Else

T1,T2 = two intersection point (T1 < T2)

Plane.cpp

=> for a given ray, check if there is any intersection with this plane

=> (x-ray)dot(normal) = 0, x is the endpoint on this plane, the final t for ray = u+tw

bool Plane::Intersection(const Ray& ray, std::vector<Hit>& hits) const

If T<=0 or numerator ==0 return False

Else

Check if the ray is toward plane, or outward

Phong_shader.cpp

=> return color by phonging model, Ra + Rd + Rs = color

=> We can see in parse.cpp, color_ambient, color_diffuse, color_specular, and specular_power has read from .txt file (usea a map<string, vector> and load the vector)

```
Phong_Shader(Render_World& world_input,
    const vec3& color_ambient,
    const vec3& color_diffuse,
    const vec3& color_specular,
    double specular_power)

vec3 Phong_Shader::Shade_Surface(const Ray& ray,const vec3& intersection_point,
const vec3& same_side_normal,int recursion_depth) const
```

Part 1: Ambient

=> also seen in parse.cpp, world.ambient_color和world.ambient_intensity are given

```
color = world.ambient_color * world.ambient_intensity * color_ambient;
```

Part 2: Diffuse

```
vec3 L = world.lights[i]->position - intersection_point;
color += std::max(0.0, dot( same_side_normal, L.normalized() ) ) *
(world.lights[i]->Emitted_Light(ray)/ dot(L,L) ) * color_diffuse;
```

=> L = lightsource, same_side_normal = normal on object, get theta by dot them

=> Emitted_Light() = color and brightless/ 4pi, divide it by the square of distance

Part 3: Specular

```
L = world.lights[i]->position - intersection_point;
// L = L.normalized();
vec3 reflect = 2*dot(L,same_side_normal)*same_side_normal-L; // the reflect of light
vec3 camera = ray.direction.normalized()*(-1); // vector of camera
color += pow(std::max(0.0, dot(camera, reflect.normalized())), specular_power) *
(world.lights[i]->Emitted_Light(ray)/ dot(L,L) ) * color_specular;
```

=> get pi by dot camera and reflect, cos(pi)^(specular_power=alpha)

=> also use Emitted_Light()

=> the shadow is put at the front of for loop

```
if(world.enable_shadows){
    vec3 P2S_vector = world.lights[i]->position - intersection_point;
    Ray P2S(intersection_point, P2S_vector.normalized());
    Object* objOutput = world.Closest_Intersection(P2S, hit);
    if(objOutput){
```

=> P2S is the ray of intersection point->light source, if we hit anything, there is something between point and light source. So we have a shadow

=> In this case, skip the diffuse and specular and return ambient color

=> if not hit anything, accumulate the diffuse and specular color

```
        Continue;
    }   }
```