Project 5 Report

By Fedor Ivachev 费杰 2019280373

Raytracing (adding code to framework)

Complete the //NEED TO IMPLEMENT functions in 3 classes: Raytracer, Primitive, Light.

For more instructions, see the PPT in attachment.

The required materials and goal effect refer to the attachment.

Please submit the source code, report and the screen recording when you run the program

In this report I will briefly talk about the work I have done. For math equations explanation or for the ray tracing principles please refer to internet sources. In my project I used stackoverflow, unity and other forums, as well as books such as Peter Shiley's "Ray tracing in one weekend"[0] and Kevin Suffern's "Ray Tracing from the Ground Up"[1]. The main problem was lack of information provided. Usually only Square and Circle primitives are considered, while all the other objects are rendered as triangular meshes. Though I had to find and count the formulas for the Cylinder and Bezier by myself (and definitely not correct).

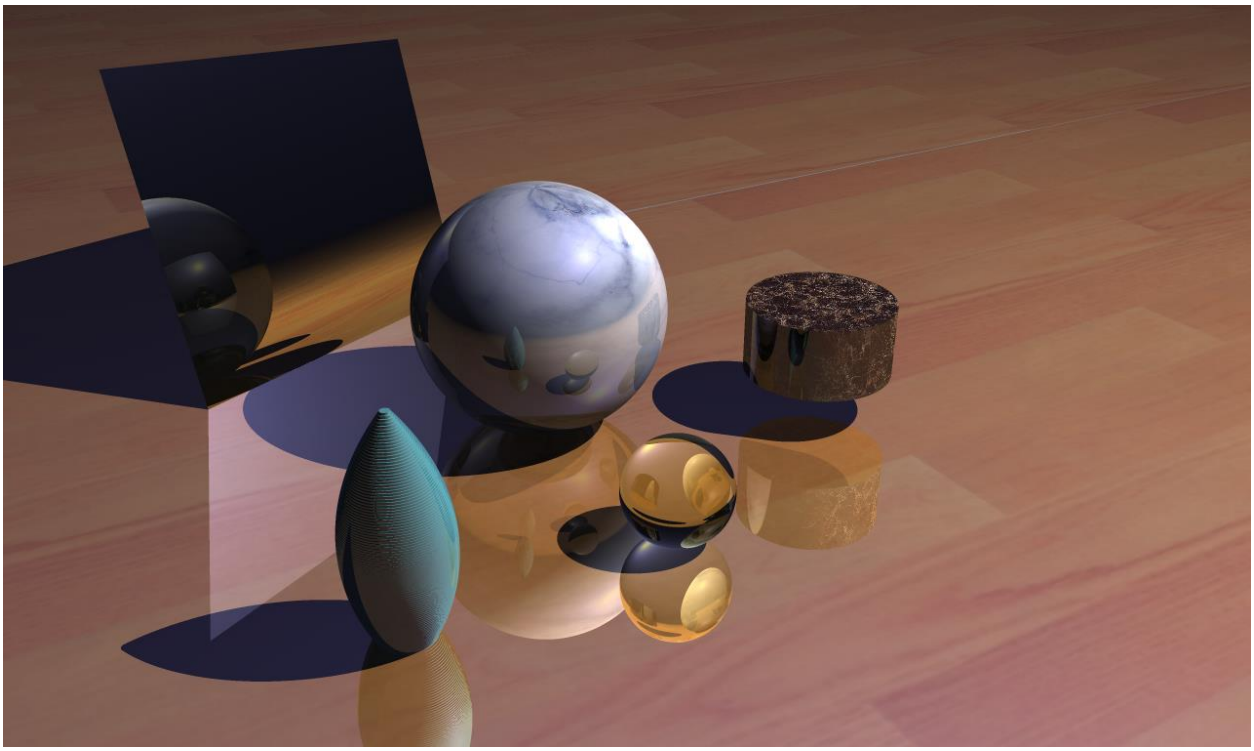


Figure 1 Calculating bezier spinar with step 0.01

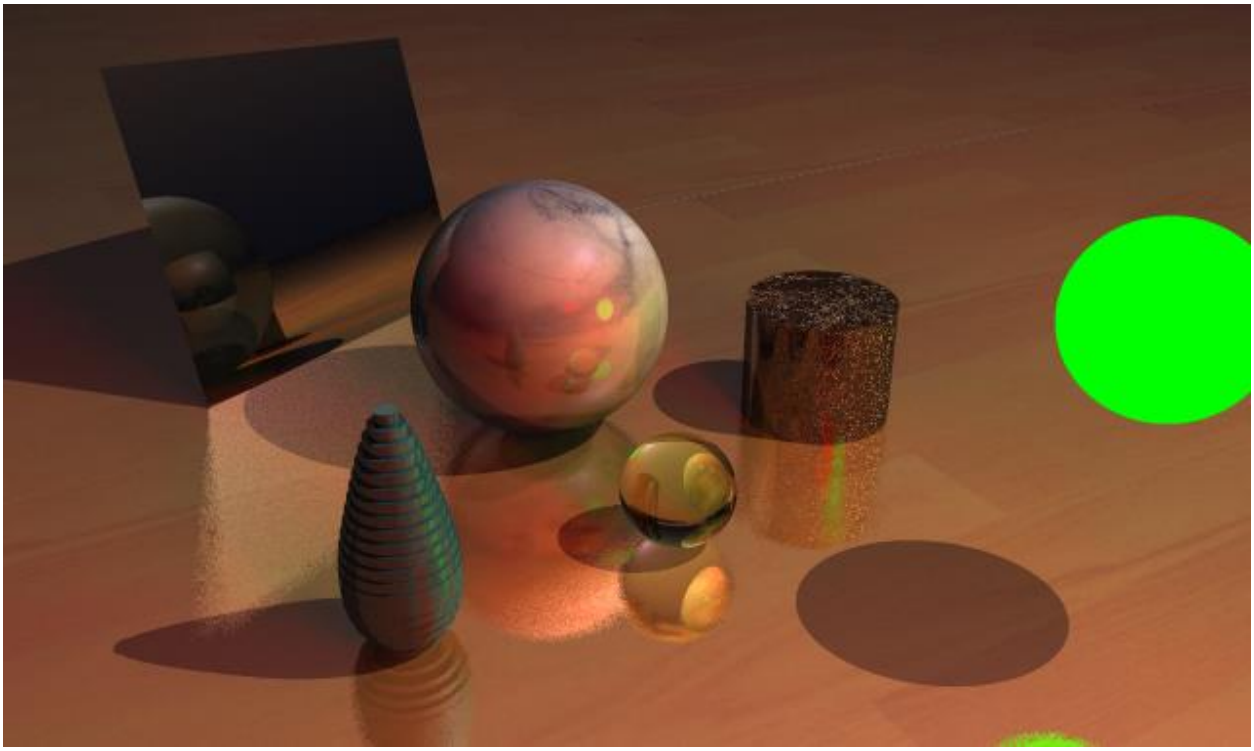


Figure 2 Another bezier spinar with step 0.05

Main part

Raytracer.cpp

```
Color Raytracer::CalnReflection(CollidePrimitive collide_primitive , Vector3 ray_V , int dep , int* hash )
```

In this function I had to implement something new – a new method how the light collides the primitive.

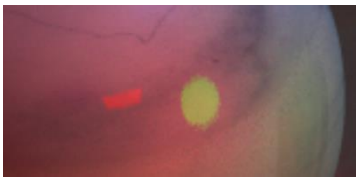
To achieve the new method, I decided to add exponential blur glossy reflection, based on the [1].

```
ray_V = ray_V.GetUnitVector();
Vector3 u = ray_V * collide_primitive.N * primitive->GetMaterial()->drefl;
Vector3 v = ray_V * u * primitive->GetMaterial()->drefl;
Vector3 Random_point_in_unit_sphere = primitive->GetMaterial()->blur->GetXYZ() *
primitive->GetMaterial()->drefl;
Color ret;
for (int i = 0; i < camera->GetDreflQuality(); i++) {
        ret += RayTracing(collide_primitive.C, ray_V + u * Random_point_in_unit_sphere.x + v *
        Random_point_in_unit_sphere.y, dep + 1, hash);
}

return ret * primitive->GetMaterial()->color * primitive->GetMaterial()->refl / camera->GetDreflQuality();
```

We define a random point in unit sphere and add it to the direction of the reflected vector. The Drefl quality defines the number of this number of points.

Light.cpp



You can see green sphere and red square light primitives. Notice that right part of the metal ball is more green (since green light is closer to it), while left part is more red.

```
double SquareLight::CalnShade( Vector3 C , Primitive* primitive_head , int shade_quality ) {
	int shade = 0;

	for (int i = -2; i < 2; i++) {
		for (int j = -2; j < 2; j++) {
			for (int k = 0; k < shade_quality; k++) {
				Vector3 V = O - C + Dx * ((ran() + i) / 2) + Dy * ((ran() + j) / 2);
				double dist = V.Module();

				for (Primitive* now = primitive_head; now != NULL; now = now->GetNext())
{

					CollidePrimitive tmp = now->Collide(C, V);

					if (EPS < (dist - tmp.dist)) {
						shade++;
						break;
					}
				}
			}
		}
	}

	return 1 - ((double)shade) / (16.0 * shade_quality);
}
```

To calculate the square light, we divide the light square primitive to 16 sectors and use one random point from each sector as a point light primitive.

```
double SphereLight::CalnShade( Vector3 C , Primitive* primitive_head , int shade_quality ) {
	int shade = 0;

	for (int i = -2; i < 2; i++) {
		for (int j = -2; j < 2; j++) {
			for (int m = -2; m < 2; m++) {
				for (int k = 0; k < shade_quality; k++) {
					Vector3 P((ran() + i) / 2, (ran() + j) / 2, (ran() + m) / 2);
					P = P.GetUnitVector();
					Vector3 V = O - C + P * R;
					double dist = V.Module();
					for (Primitive* now = primitive_head; now != NULL; now =
now->GetNext()) {

						CollidePrimitive tmp = now->Collide(C, V);

						if (EPS < (dist - tmp.dist)) {
							shade++;
							break;
						}
```

```
                                    }
                                }
                            }
                        }
                    }

                    return 1 - ((double)shade) / (64.0 * shade_quality);
}
```

Same is for the sphere light. We get 64 random points on each sector and consider each of them as a light primitive.

Primitive.cpp

In this part, square collision, Cylinder collision and texture mapping, Bezier collion and texture mapping had to be implemented.

For the square collision, it counts very simple:

```
Vector3 N = (Dx - O) * (Dy - O);
N = N.GetUnitVector();
double d = N.Dot(ray_V);
if (fabs(d) < EPS) return ret;
double l = N.Dot(O - ray_O) / d;
```

First count intersection with the plane: get the Norm of the square, dot product by vector between ray origin and center of the square and divide by Dot product of N by ray direction. Then don't forget to check if the point is inside the square.

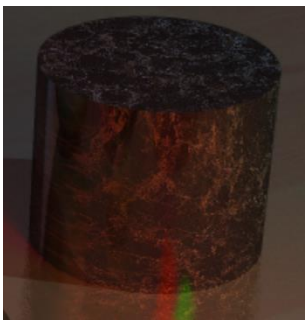
For the cylinder, it is more complex:

The collision point can lie either on the cylinder top, bottom or on the side.

Firstly, the quadratic equation needs to be solved:

```
double A = (ray_V - L * ray_V.Dot(L)).Module2();
double B = 2 * (ray_V - L * ray_V.Dot(L)).Dot(P - L * P.Dot(L));
double C = (P - L * P.Dot(L)).Module2() - R * R;
```

This equation defines endless cylinder, so we need to check whether the intersection lies between the defined cylinder bases. Also, the intersection of the cylinder top and bottom bases needs to be counted. Then the collision closest to the ray origin needs to be taken. Please refer to the source code for more equations. Also it should be noted, that apart from most of the tutorials, in my program a cylinder doesn't have to be parallel to the xy plane.



I'm not sure about the code for getting the texture of the cylinder (and you can see that on Figure 1 and Figure 2 the textures are different). For counting the texture for the bases and for the side part, different formulas are used. For the side part, the position depends on the angle of collision on the current slice of the cylinder (here I used atan2 to count it).

Bezier spinar

Unfortunately, I could not find any information or theory about how to collide with Bezier spinar surfaces. Probably, the collision can be counted, but the value will be approximated. Mainly because counting the roots of the equation is very cost-computing task. But I'm sure it's possible to solve it analytically (otherwise Bezier spinar won't be considered as a primitive in this project task).

I came up with several other solutions to find the collision with Bezier spinar (each of them includes using cylinder bounding box to save the computing time):

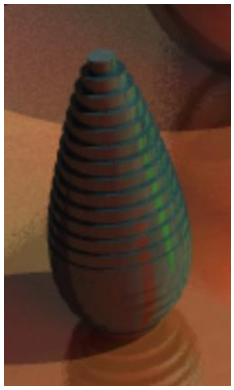
1. Make a triangular mesh of the Bezier spinar and count the intersection for each triangle, then get the closest intersection to the ray origin. After that use texture to smooth the edges.
2. Divide Bezier spinar into cylinders. Count intersection with each cylinder and then get the closest to the ray origin. To smooth the edges, change cylinders to the frustums.

I decided to go with the second option I came up with, but not using frustums.

```
for (double u = 0; u - (1 - step_height) < EPS;) {
	z_cur = O1 + (O2 - O1) * P(u, Z);
	r_cur = P(u, R);
	u += step_height;
	z_next = O1 + (O2 - O1) * P(u, Z);
	Cylinder* curCylinder = new Cylinder(z_cur, z_next, r_cur);
	tmp_dist = curCylinder->Collide(ray_O, ray_V).dist;
	if (tmp_dist < min_dist) {
		min_dist = tmp_dist;
		min_u = u - step_height;
	}
	delete curCylinder;
}
```

Note that each of the cylinders should be deleted, or too much RAM will be used. Also, in the source code provided I had to add the deleting of the textures stored for the material, after what the performance of the raytracer increased dramatically.

With the method I used, it does not matter, what is the degree of the Bezier spinar, because computing time of the Bezier curve is << then computing time of intersection with each of the cylinders. The computing time is linearly dependent to 1 / step_height.

The source code can also be found at https://github.com/FedorIvachev/GraphicsCourse