Final Lab Report

Feiyu Liu

1. Introduction for my final assignment

I really have a good time in this class! It is hard to believe that after 2 mouths' study I can gain so much useful information in computer graphics. In this report, I will show my accomplishment in the final assignment and discuss my "journey" in this process.

The first step is intersection. In triangle situation, I created a three-variable linear equation (using gamma, t, and beta as variables):

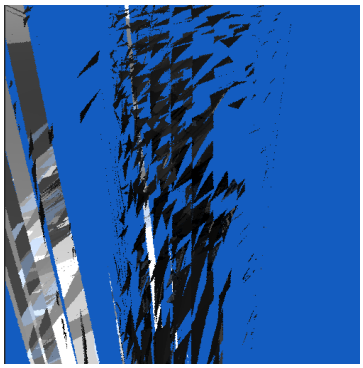
```
d0 * t + (v20 - v10) * beta + (v30 - v10) *gamma = a0 - v10
d1 * t + (v21 - v11) * beta + (v31 - v11) *gamma = a1 - v11
d2 * t + (v22 - v12) * beta + (v32 - v12) *gamma = a2 - v12
```

My solution is by calculation, I can get the value of gamma and t using beta to represent. Then put these two variables back to one of the equations so that equation will only contain one unknown variable (beta). When beta is calculated, the value of gamma and t will soon be solved, too. However, I forgot to use det() method to simplify the equation. The result variable function looks a bit long.

```
double gamma;
double beta;
double t;
double o0 = r.d[1]*(c[0]-a[0])-r.d[0]*(c[1]-a[1]);
double o1 = (c[2]-a[2])*r.d[1]-(c[1]-a[1])*r.d[2];

beta = (r.e[0] - a[0] - (c[0]-a[0])*(r.d[1]* (r.e[0]-a[0])-r.d[0]*(r.e[1]-a[1]))/o0 - r.d[0] * (r.e[1]-a[1])*(c[1]-a[1])-(r.e[2]-a[2])*(c[2]-a[2])/o1)*o0*o1/
((b[0]-a[0])*o0*o1 + o1* (r.d[1]*(b[0]-a[0])-r.d[0]*(b[1]-a[1]))* (c[0]-a[0]) + o0* ((c[2]-a[2])*(b[1]-a[1])-(c[1]-a[1])*(b[2]-a[2]))*r.d[0]);
gamma = (r.d[1]* (r.e[0]-a[0])-r.d[0]*(r.e[1]-a[1])-(r.d[1]*(b[0]-a[0])-r.d[0]*(b[1]-a[1]))*beta)/(r.d[1]*(c[0]-a[0])-r.d[0]*(c[1]-a[1]));
t = ((r.e[1]-a[1])*(c[1]-a[1])-(r.e[2]-a[2])*(c[2]-a[2])-((c[2]-a[2])*(b[1]-a[1])-(c[1]-a[1])*(b[2]-a[2]))*beta)/((c[2]-a[2])*r.d[1]-(c[1]-a[1])*r.d[2]);
```

Of course, bugs happen:



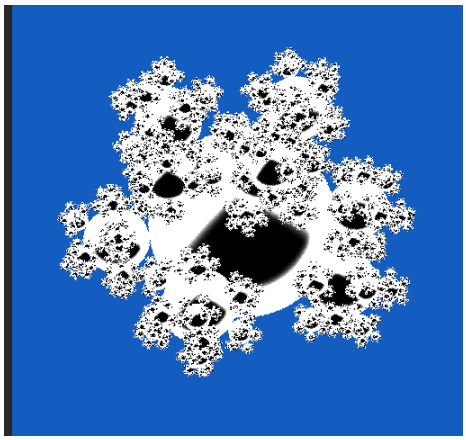
So, after I understand the code from step1 midpoint version, I quoted the code of triangle intersection. It works well this time.

Sphere intersection is easier than triangle intersection, but it contains three situations: ray goes across the sphere (two points), ray is tangent to sphere (one point), and ray does not intersect with the sphere (no point). I used delta to identify the suitable situation and give different solutions:

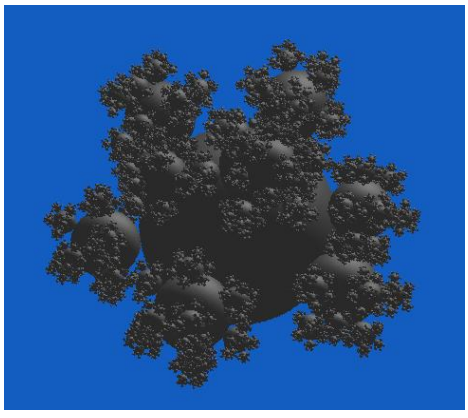
```
double delta = B*B-A*C;
if(delta>=0){
    a1 = (-B+sqrt(delta))/A;
    a2 = (-B-sqrt(delta))/A;
    a3 = a1;
    if (a1 < 0 || (a2 > 0 && a2 < a1)){
        a3 = a2;
    }
```

Another part is writing the code for trace. By using for-loop to check every surface, it allows intersection code to work.

The second step is familiar to me, because we have already done it in assignment 1. The principle is the same as the previous one: calculating diffuse, specular, and normal color separately, then adding them up. I followed these instructions and generated a picture, but this picture looks very sharp:



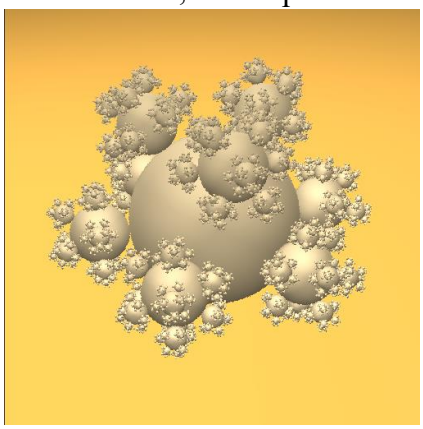
Then I realized that I didn't normalize which causes the value of each pixel too large. After normalization, the generated picture looks smoother, but still a little weird:



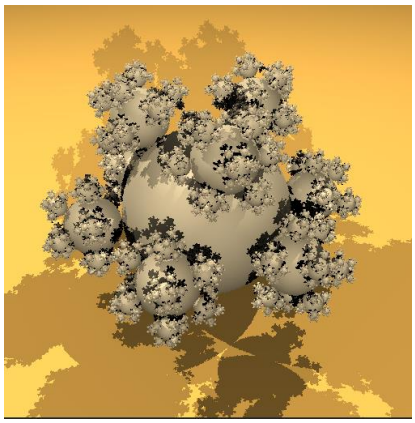
This bug really confused me for a long time, but I cannot tell what went wrong. Thinking back to light again, I found what was wrong with this picture: it only received one direction light. I double checked my code and added "color" for color part so that it can collect information from all lights:

```
color = (diffuse + specular + Normalcolor) * Totalcolor + color;
```

And this time, the output shader works very well:



The third step is adding shadows for the picture, which is checking if there is anything blocking in front of the target. If the answer is yes, this area should be covered with shadows and no need to run shade code. This part is like what I did in trace part: use for-loop to check each surface, then use intersection to decide whether this surface is under shadow or not. After this step, the generated image became more three-dimensional.

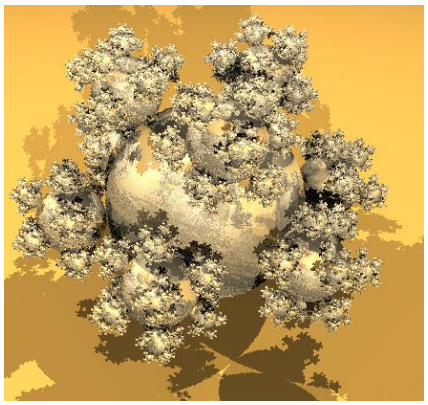


Step four is computing reflection color. My solution is checking if reflection ray hits any surface. If yes, adding reflection color to total color multiply by ks. However, when I finished my code and tested it, it didn't have any picture generated. Then I used std::cout to check if there is any problem with my methods and found the problem: there is a infinite loop between "trace" and "shade", because they called each other and I didn't set a limit for them. Therefore, I set a MAX for hr.raydepth so that it would only run limited times:

```
Ray reflectionRay = Ray(hr.p, reflection);
reflectionRay.depth = hr.raydepth + 1;
normalize(reflectionRay.d);
```

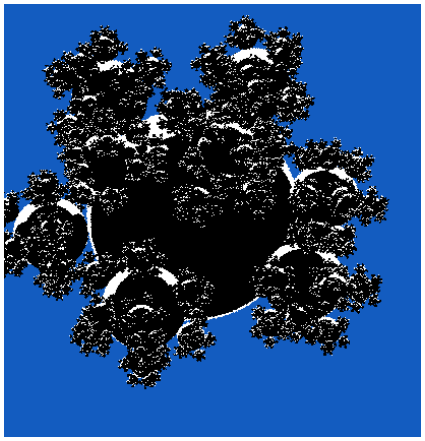
```
if (hit && hr.raydepth < 3) {
    color = shade(hr);
}
```

And here is my result. It is much more amazing than I thought, but seems not that metallic:



Maybe there are still some problems existing…

I still feel surprised when I look at my pictures from single-color painting to such realistic and three-dimensional work. There are also some funny bugs! Like during step 2, I implemented wrong codes for specular part and the result looks very interesting:



This picture is made by not normalized variables and wrong function of specular. The comparison of edge light and black color gave me a sense of “eclipse”, and I though it was a little cool!

2. Reflections

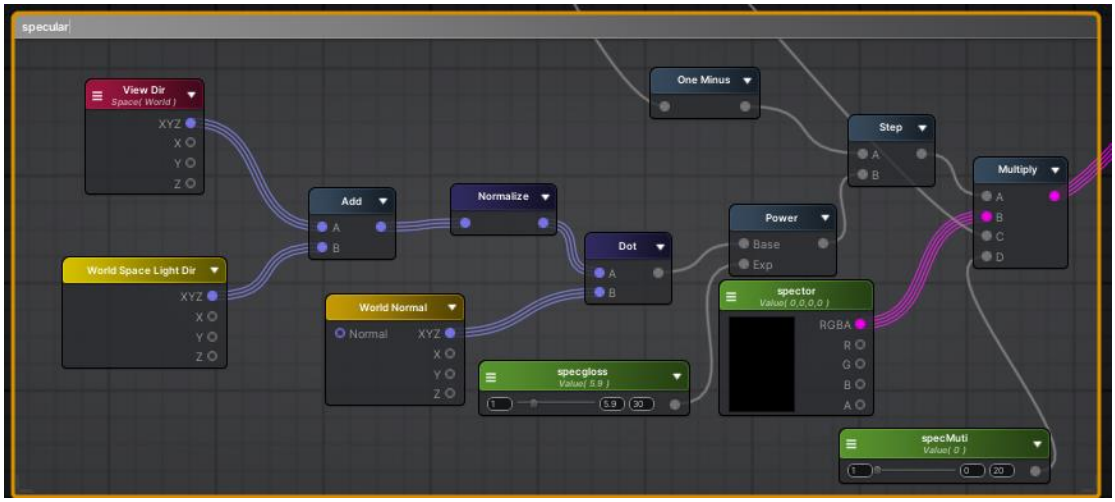
I have learned much from this course! From the most basic structure of a material ball to the trajectory and reflection of rays, this course dissects the knowledge step by step, making them easy to understand with various examples and discussions. I thought computer graphics was a difficult area of study for beginners, and this course tells me it is true but also filled with fun! Now I feel good with shading, color, and raytracing knowledges, and I will continue my interest in computer graphics area in the future!

3.Extension

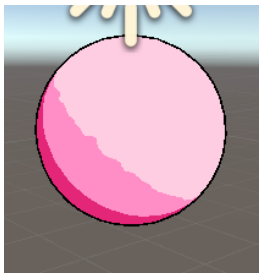
The principle of shading is most impressive for me. It tells me that a basic material ball is made of specular, normal color and diffuse. So, I tried to use same principle to create a simple cel-shading ball. Here is my function:

Specular + first-level shadow + second-level shadow + normal + edge light = cel-shading

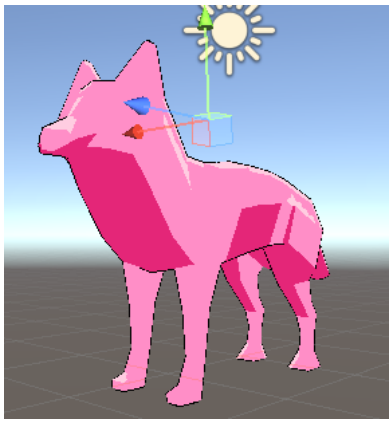
I used Amplify Unity Shader to implement my function, here is what it looks:



It is much harder than I thought: Because I added another layer of shadow, dividing these two types of shadows is a big problem, and there should be almost no transition between two different color types. With the knowledge from the class and internet, I found out how to use the functions from class in this Amplify Unity Shader and add simple ranges for each color type. After adding them step by step, I finally got my cel-shading ball:



And here is my outcome with a low-polygon model dog. Although I couldn't make the range of shadow change with the intensity of light, but the shading can still detect the general direction of light:



In the future, I will try to use C# codes to write a cel-shading using my function. By adding one layer of shadow and edge light, I think I can create a cel-shading ball with codes (maybe range detection will still be a problem for me, but I will try to overcome it!). Thanks for this class!