IM420 Advanced Computer Graphics – WS 2020
Assignment 6 - Raytraced Scene

Egle Sabaliauskaite, Sahabaj Barbhuiya

22nd January, 2021

1.1 Task

This assignment report and code snippets aims to explain the implementation of our own ray tracing scene with one primitive object and a different procedural texture (surface) which has not been covered during the lecture. This assignment is based on the lectures and project template of *raytracing_solution*. In this assignment report we will discuss about the concepts of ray tracing, where we explain the ray intersection calculations and the calculations for the normals for our ray tracing scene. We will also explain implementing a cone and a simple procedural texture which will be used as the surface for our scene instead of using the checkerboard.

1.2 Raytracing

A basic ray tracing algorithm offers the calculations and computations in a 3D scene, to simulate the path of light. For each pixel, the algorithm computes a viewing ray, find the first object hit by the ray and set the pixel color based on the hit point and the location of the lights.

In a ray tracing scene, using the light source we create light rays. We intersect these light rays, with the scene which has the objects (cylinder, cone etc) and hence we intersect the light rays with the objects. So, if a light ray hits an object, we have to compute a reflection (a reflected ray on the surface) and refraction (refracted rays). Based on these two rays we create new rays, and we shoot these new rays on the scene again and intersect with the rest of the scene until the light rays hits the eye or the camera. There are two types of ray tracing: forward ray tracing and backward ray tracing. We will elaborate both types of ray tracing now.

- Forward ray tracing: In forward ray tracing, the rays that interact with the scene with the objects comes directly from the light source. In forward ray tracing many rays never hit the eye or the camera because they are reflected from the objects. Hence, light ray computations are lost which is a disadvantage of forward ray tracing.
- Backward ray tracing: In backward ray tracing, we start from the eye or the camera position and trace the rays through the 3D scene. At the end, we only have to compute every intersection of light rays with the objects in our 3D scene.

1.2.1 Ray-object intersections

Each ray is defined by the following equation:

$$\mathbf{p} = \mathbf{o} + t\mathbf{d}$$

where point **p** lies on a ray with the origin **o** and the direction **d**. Let us take an example of a sphere with the center c and the radius r in the scene. We use backward ray tracing to send rays into the scene. The ray can miss the sphere object and have one intersection point or pass through it and have two intersection points. Moreover, we also have to calculate the normals on the position (normal of the hit point) to perform shading on the sphere object. Figure 1.1 shows the intersection calculation between the ray and the sphere object. The following equation is used to calculate the normal on the hit point:

$$\boldsymbol{n} = (x - c)/r$$

where x is the hit point, c is the sphere radius and r is the radius of the sphere.

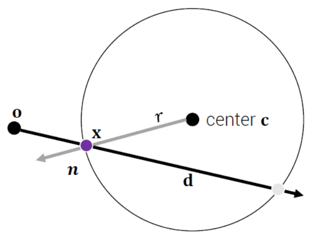

Figure 1.1: Ray intersection with the sphere object

1.2.2 Ray-plane intersections

To compute the ray intersection with the plane, we use the following ray equation for the plane:

$$\boldsymbol{n}.x + d = 0$$

The orientation of the plane is defined by the normal vector n, d is the distance from the origin for the normal and x is any hit point on the plane. Figure 1.2 shows the normal, hit point and ray direction **d**.

Now, we substitute x with the ray equation to calculate the parameter t, which is the distance between the ray direction **o** and any hit point on the plane. t can be calculated with the following equation:

$$t = \frac{-d - \mathbf{n} \cdot \mathbf{o}}{\mathbf{n} \cdot \mathbf{d}}$$

Moreover, if ray is parallel to the plane, we will not get an intersection and parameter t has to be $t{>}0$ so that we have a hit point on the plane.

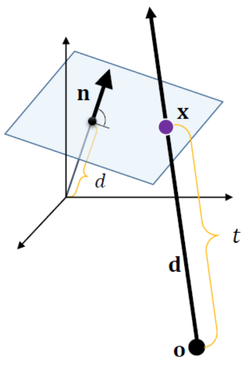

Figure 1.2: Ray intersection with the plane

1.3 Approach to the Solution

To implement a ray traced scene with new primitive object and a new procedural texture, we have to calculate the intersection of an object similar to the sphere previously discussed. In our case we will be using a cone object as a very basic colored lines as a new procedural texture in our ray tracing scene. In the following sections, we will discuss the concepts of ray intersections for the cone and also implementing the colored lines as the surface texture.

1.3.1 Ray tracing a cone

Calculating the intersections for a cone is similar to calculating the intersections of the sphere. We first add the cone with the *addCone* function. The following code in the fragment shader is used to add the cone in our ray tracing scene.

```
void addCone( vec3 ro, vec3 rd, vec3 center, float radius, float height, vec3 color,
     inout float hitDist, inout vec3 hitColor, inout vec3 hitNormal ){

    float dist = intersectCone(ro, rd, center, radius, height);

    if (dist < hitDist) {
        vec3 yMin = center-height;
        vec3 hit = ro + dist * rd;
        hitNormal = normalForCone(hit, center, radius, height);
        hitColor = color;
        hitDist = dist;
      }
}
```

The intersectCone function we uesd is slightly different from the intersectSphere discussed in the lecture, as the cone has a radius as well as a height. Hence, we have to define a height variable *float coneHeight* in the intersectCone function which will be used in the intersection equations for the cone as a parameter. The code snippet of ray intersection for the cone is given below, which will be explained later.

```
float intersectCone(vec3 origin, vec3 ray, vec3 coneCenter, float coneRadius, float
    coneHeight) {
  float yMin = 0.0;
  float yMax = coneHeight;
  vec3 toCone = origin - coneCenter;

  float a = ray.x* ray.x + ray.z * ray.z - ray.y * ray.y;
  float b = 2.0 * toCone.x * ray.x + 2.0 * toCone.z * ray.z - 2.0 *  toCone.y * ray.
    y ;
  float c = toCone.x * toCone.x + toCone.z * toCone.z - toCone.y * toCone.y ;

  float discriminant = b*b - 4.0 * (a*c);

  if(discriminant > 0.0) {
    float t1 = (-b + sqrt(discriminant)) / (2.0 * a);
    float t2 = (-b - sqrt(discriminant)) / (2.0 * a);
    float t;
    float y1 = toCone.y + t1 * ray.y;
    float y2 = toCone.y + t2 * ray.y;
     //front of cone
    if(y2 < yMax && y2 > yMin && t2 > 0.0) {
      return t2;
    }
    // back of cone
    if(y1 < yMax && y1 > yMin && t1 > 0.0) {
      return t1;
    }
  }
  return INFINITY ;
  }
```

In the intersectCone function, *yMin* is the bottom part of the cone which is the starting point of the cone. Changing the value of *yMin* from 0.0 to 2.0 will change the bottom of the cone from sharp bottom to plane bottom and *yMax* is height of the cone or top part of the cone.

One crucial concept of coordinate systems is used to calculate the values of *a, b,* and *c*, where we are adding x and z coordinates and subtracting y coordinate in equation, since we are using x and z coordinates for the plane rather than using x and y for the plane which is usually used in the mathematical equations.

We have to calculate parameter *t* also the values a,b and c similar to the calculation of the sphere. Here, *float discriminant* has to be defined calculated as well for the quadratic equation, which is later used to calculate the values t1 and t2.

In the next step we have to determine if the discriminant, if it is greater than 0, the value infinity is returned and we have no intersection. Using the y values and height of the cone, it is determined whether the rays are on the cone surface, above the cone surface or below the cone surface. After determining all the values, we can return t1 for the back of the cone or t2 for the front of the cone.

So far we have calculated the ray intersection for the cone and now we can calculate the normals and the hit points on the cone. We define a unit vector that points to the height of the cone. We calculate the bottom part of the cone surface. We also define and calculate the normal hitting the slant parts of the cone with the angles, as our cone is slant and not straight unlike a cylinder, and normalize the normal. Using the previously discussed normal equations normals for the cone are finally calculated. The following code snippet shows the calculations involved to determine the normals of the cone.

```
vec3 normalForCone(vec3 hit, vec3 coneCenter, float coneRadius,float coneHeight) {
vec3 unitVector = vec3(0.0, -1.0 , 0.0);
vec3 bottomCylinder = vec3(coneCenter.x, coneCenter.y-coneHeight/2.0, coneCenter.z
  );
vec3 normal = hit- bottomCylinder - (unitVector * (hit-bottomCylinder)) *
  unitVector ;
normal = normal * cos(0.8) + unitVector * sin(0.8);
normal = normalize(normal);
return normal;
}
```

The last step is to add the cone with suitable radius, height and position in the rayTraceScene funtion with the help of the following code snippet.

```
float rayTraceScene(vec3 ro /*rayStart*/, vec3 rd /*rayDirection*/, out vec3
  hitNormal, out vec3 hitColor)
  {
  ...
  ...
  addCone( ro, rd, vec3(3.0, 1.8, 2.4), 0.2, 3.0, vec3(1.0, 0.0, 1.0), hitDist,
  hitColor, hitNormal );
  }
```

1.3.2 New procedural texture

To implement a new surface, we made some changes in the rayPlaneIntersection function using the equations previously discussed in the ray-plane intersections subsection. We use *fract* which returns the fractional part of the hit point only considering x instead of using *mod* as discussed in the lecture. We also changed the hitColor so that surface color looks brown.

```
vec3 hitPos = rayStart + rayDir * t; //t is the distance
float b = fract(1.0*hitPos.x);
hitColor = (b + 0.3) * vec3(0.8, 0.5, 0.2);
```

1.4 Summary and Results

The task of this assignment was to implement a ray tracing scene with a new primitive object for example a cone, cylinder or a torus. The most challenging part of this assignment was the complex calculations of intersections with cone as the cone is slant. Using the reference code for the sphere intersection helped us to understand the complex concept of equations used to calculate and implement the cone intersections. Another challenging part during the cone intersections was to understand the coordinate system and using the correct coordinate systems for the plane. For the new surface, we try to minimize the complexity of the code and implemented a simple surface based on the implementation discussed during the lecture. Figure 1.3 shows the close view of cone with reflected cube, sphere and new surface, whereas Figure 1.4 shows the complete ray tracing scene with a cone primitive and new surface.

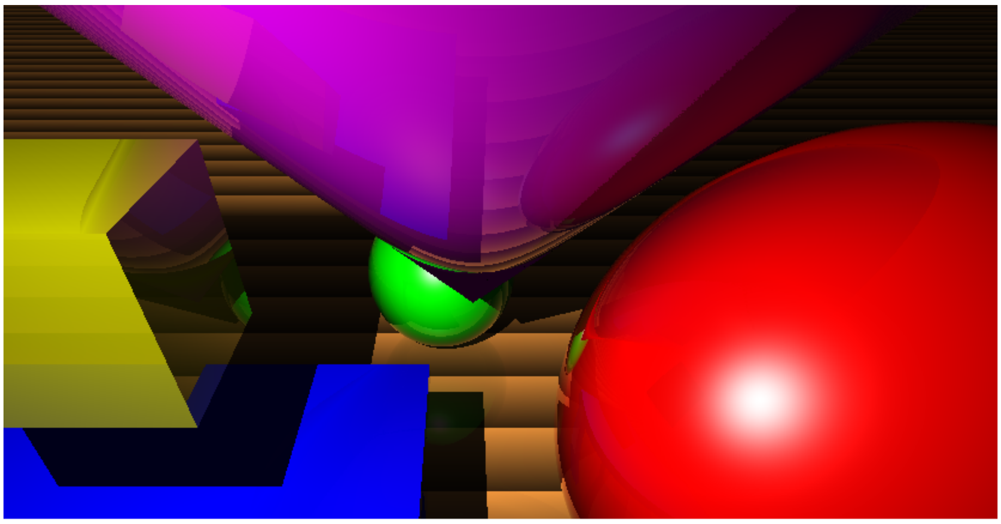

Figure 1.3: Close view of cone with reflected objects and surface

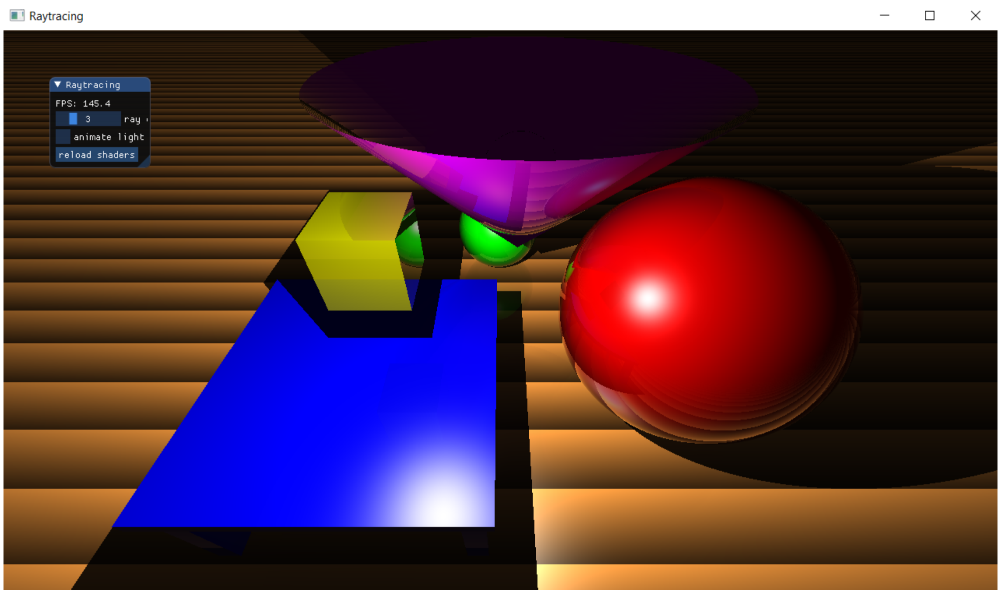

Figure 1.4: Complete ray tracing scene with cone and surface