Project 1: RayCasting

Name: Minsuk Kim(m.kim)
Instructor: Dr. Gary Herron
Class: CS500
Semester: Spring 2022
Date: 1/31/2022

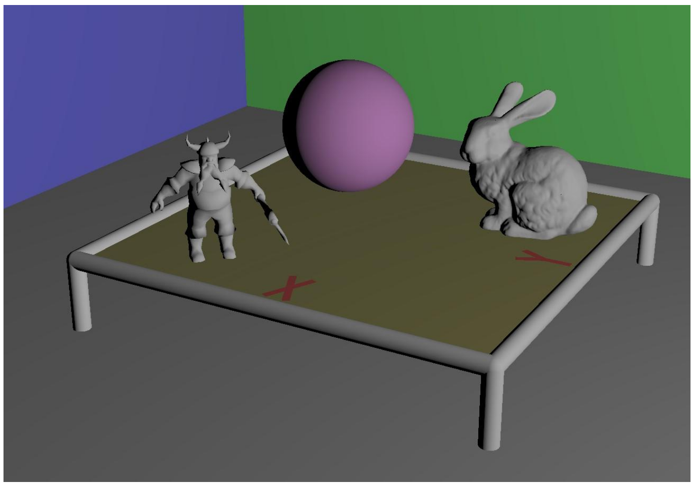


Table of Contents

Introduction

Overview

The main purpose of the project is to implement the ray casting system. The ray casting will be pre-calculated and then stored in hdr image files. It made a container of shapes for each object. After making all shapes, calculate the bounding box of each object and store it in the BVH library. I create the ray for each pixel in windows, query the ray to find the colliding objects. The BVH library detects the object that collides the bounding box with ray. Then, I determine the front-most object that the ray collides with. I stored each colliding object's image value corresponding to each pixel on the screen. I stored the appropriate value on the hdr image file. I started the project with the provided framework from the class. I also used an external library Assimp that read the object files and BVH for bounding volume hierarchy.

Implementation

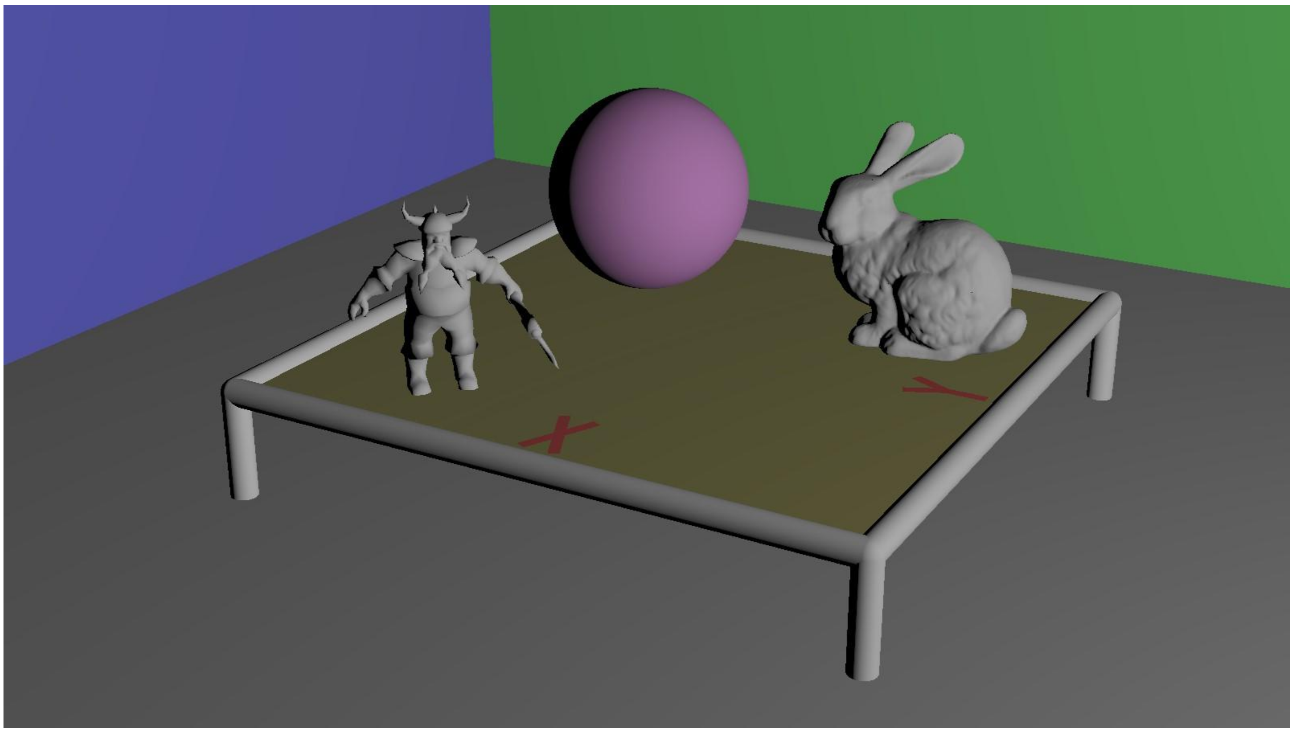
First, the program loaded the scene file and created the shapes for each object file. For this project, the scene reader interface will handle 4 types of objects: Box, Cylinder, Triangle(mesh), and Sphere. For the Triangle type of object, the program will generate many triangles from the object files. It will generate each triangle that forms the mesh file. For example, the bunny mesh file will generate 69451 triangles. Before casting the rays, the program calculates each bounding box of shapes. Then, the program sends the calculated bounding box to the BVH library. The program calculates the ray direction that casts from the camera to each screen pixel. With ray, the BVH library finds the shapes that have a bounding box that collides with the ray. For each colliding bounding box, the program queries the ray casting and determines whether the shapes are colliding with the ray or not. The program finds the front-most shape that collides with rays. Afterward, the program stores the front-most shape, intersecting point, normal of shape in intersecting point, texture coordinate of shape in intersecting point to each pixel. With the stored data, the program will calculate the light and save it into an hdr format image file.

Result Images

all images all drawn with given image testscene.scn with size 1600, 900

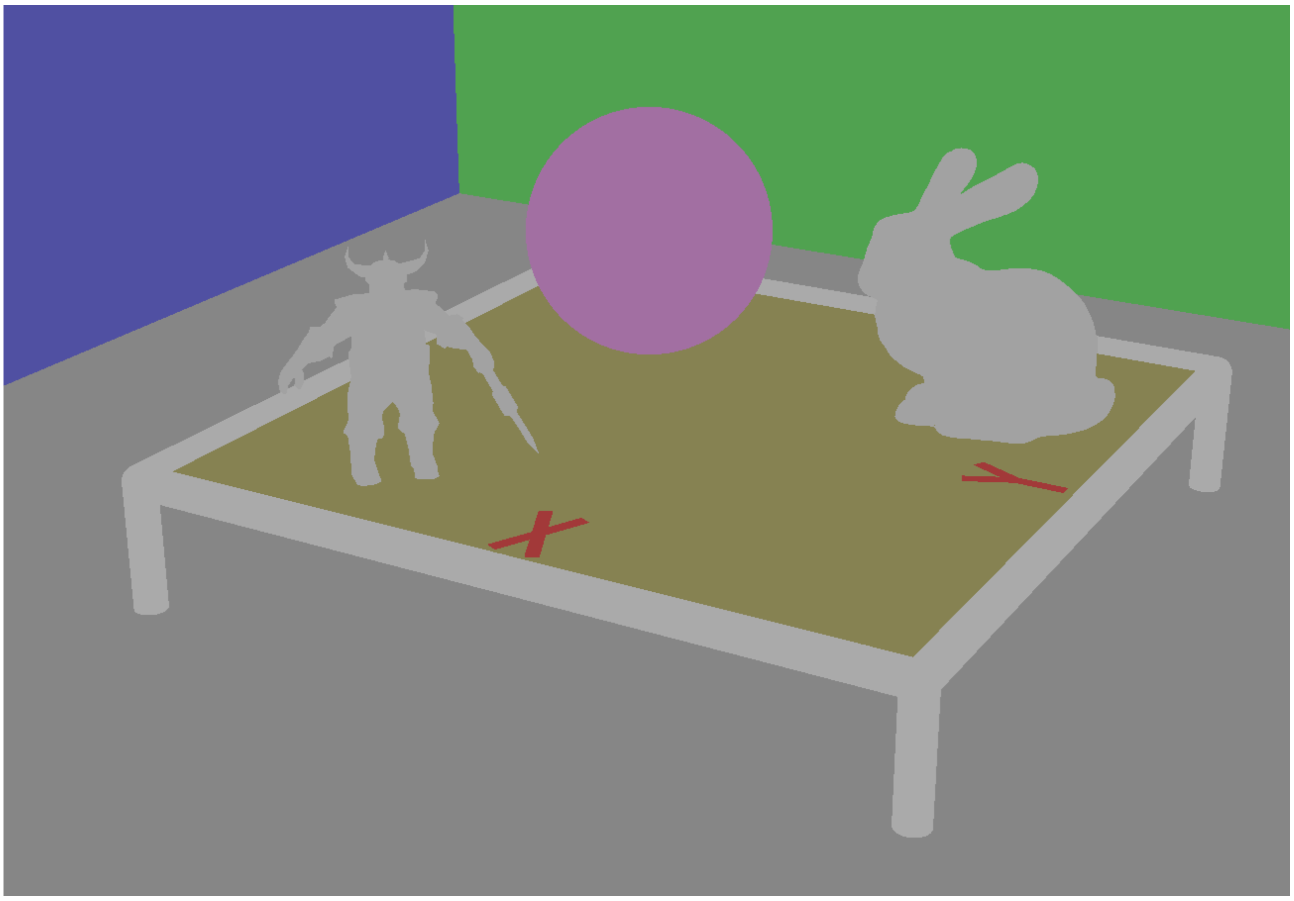
Final image

light calculation done with kd*(dot(N, L))

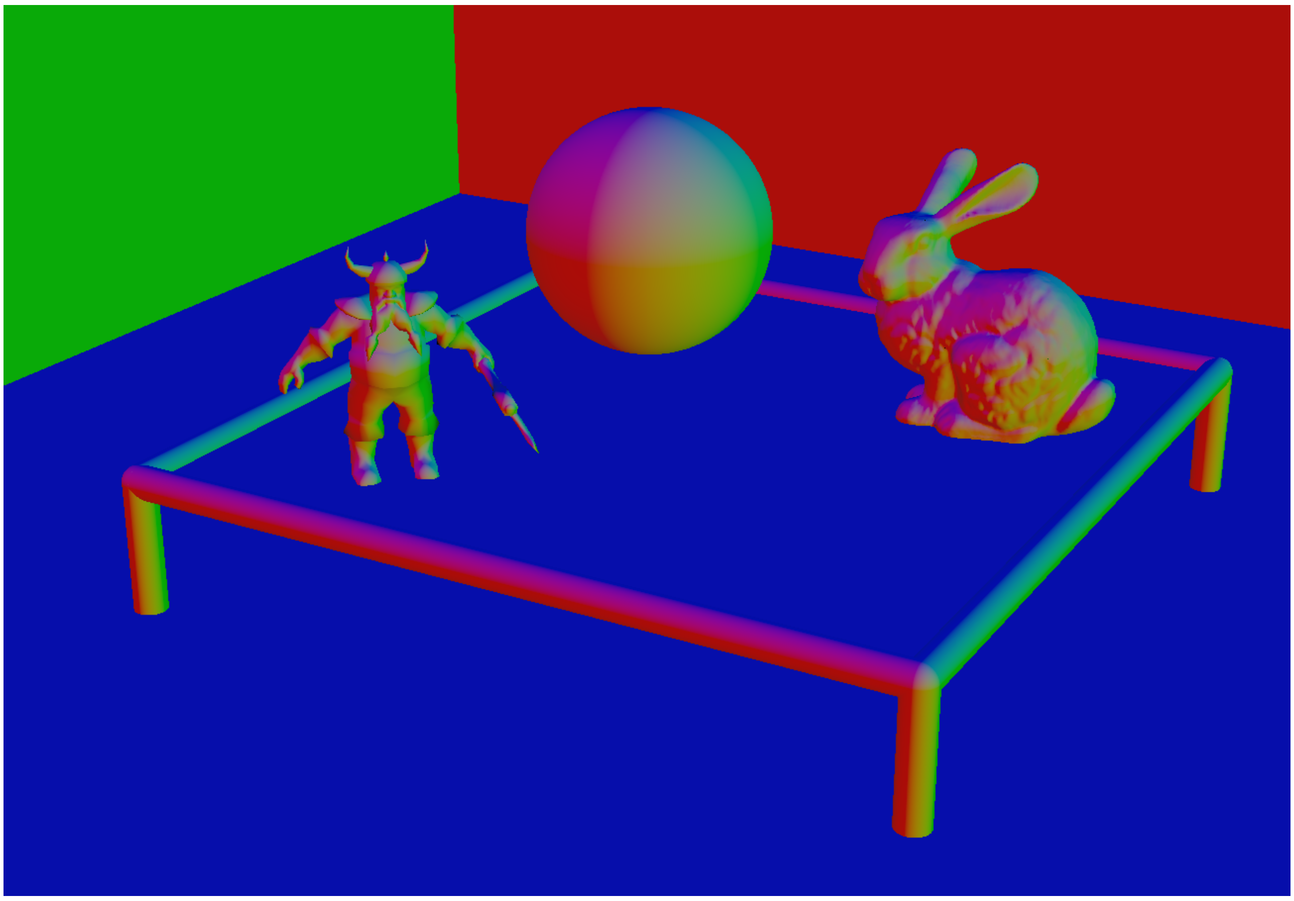


Images different render target

albedo image of scene

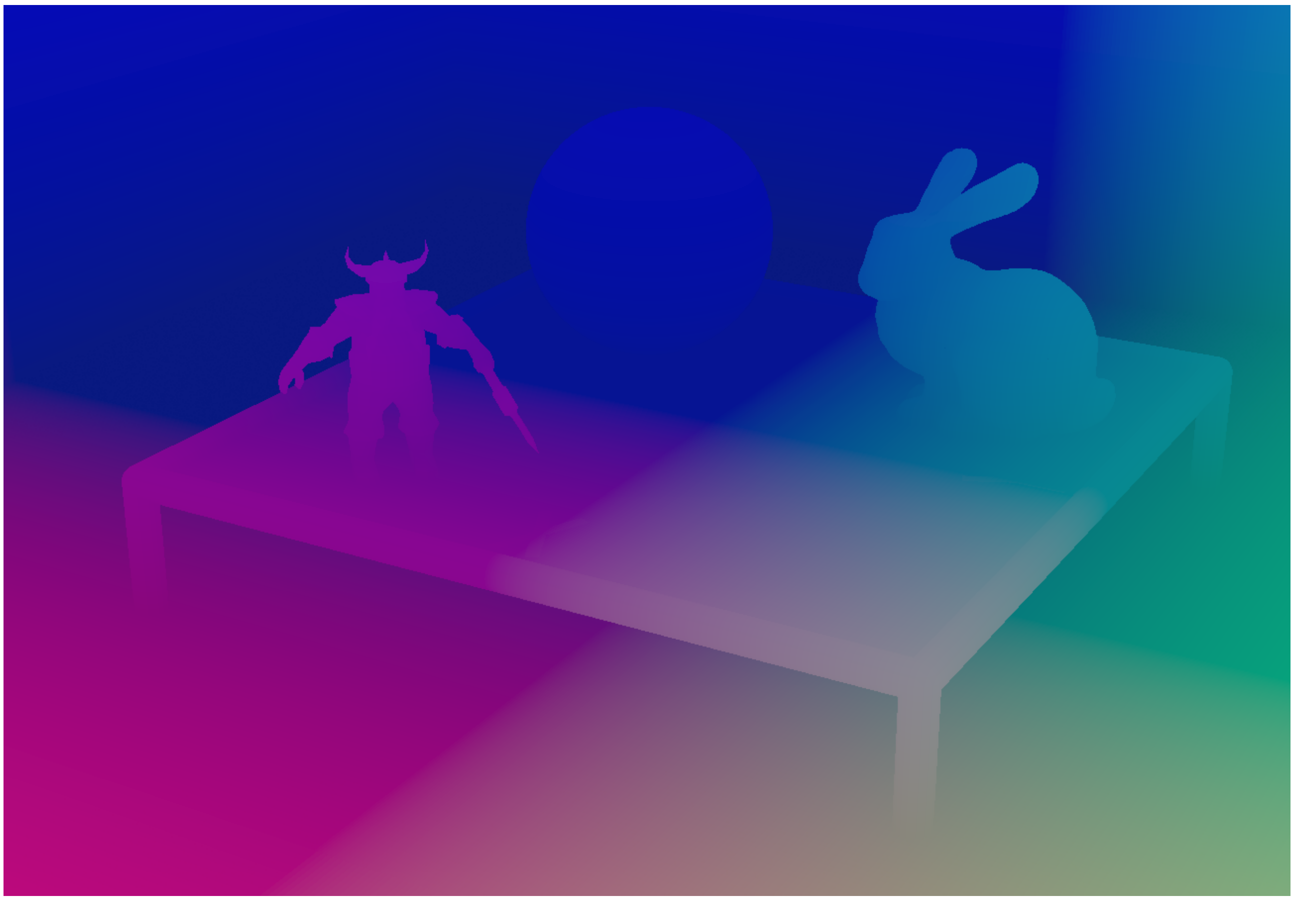


normal image display

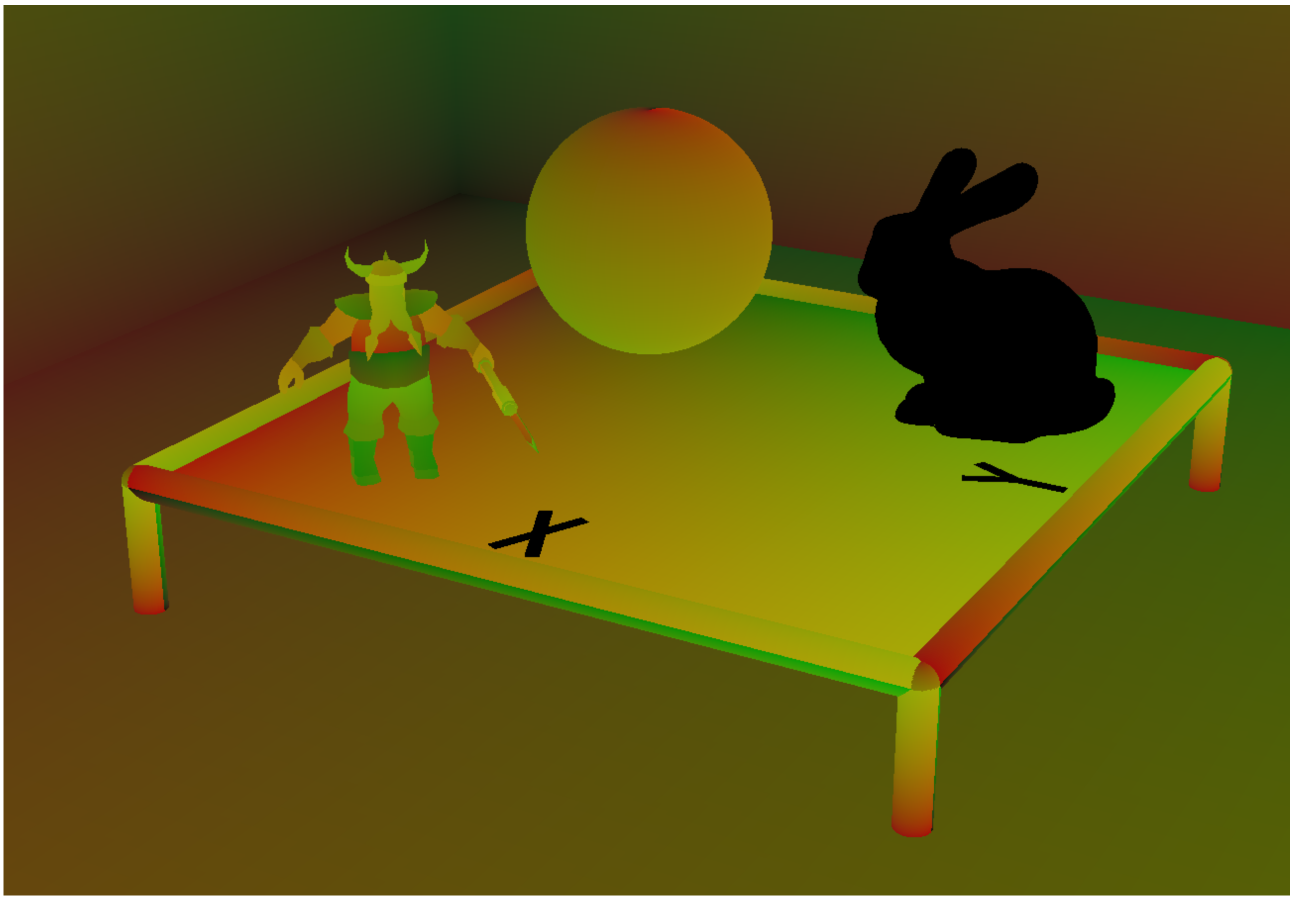


position image display

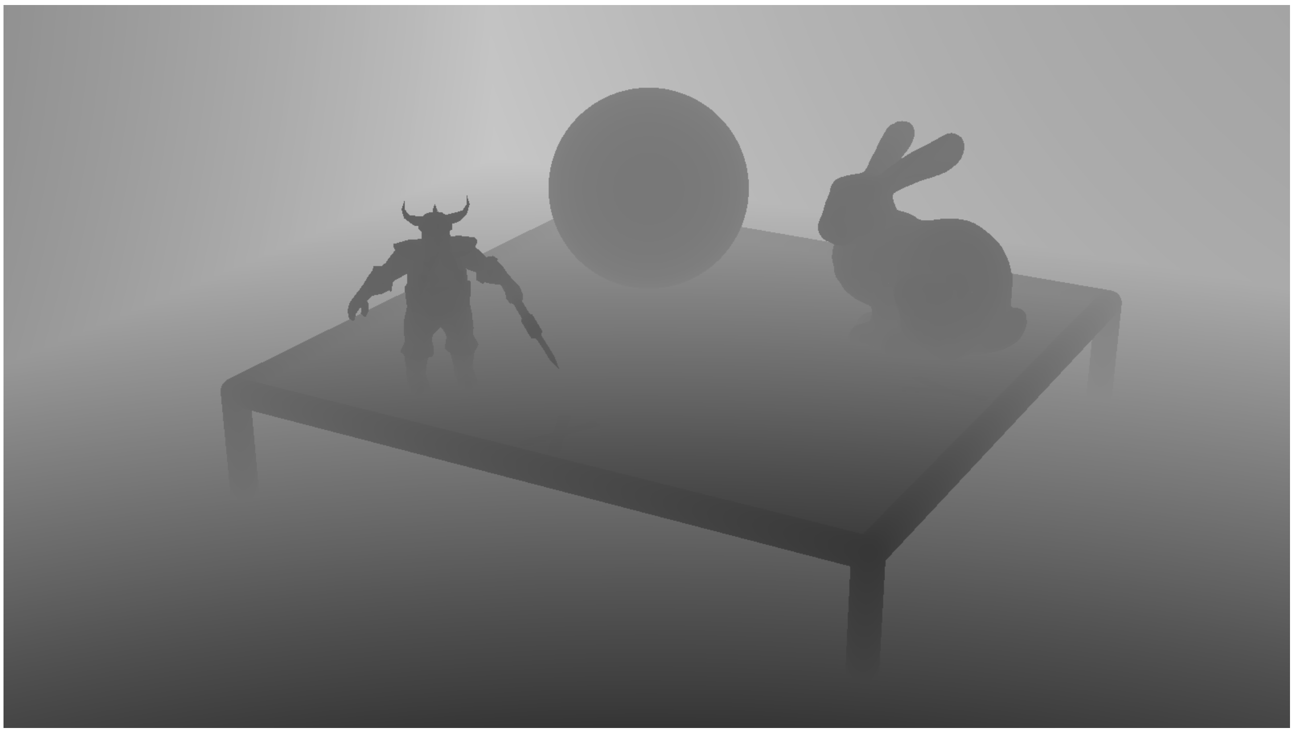
the color is ((position + (0,0,1))/2) for better visual display



texture coordinate image display (red=u, green=v, blue=0)



tvalue display
the color will be rendered by ((tvalue - 5) / 4) for better visual readability



Notes

Intersection

Sphere Intersection

The sphere intersection will be calculated by the ray and sphere function F(P)

$$Ray = Q + tD$$

$$F(P) = (P - C) \cdot (P - C) - r^2 = 0$$

$$(Q + tD - C) \cdot (Q + tD - C) - r^2 = 0$$

$$(D \cdot D)t^2 + 2((Q - C) \cdot D)t + (Q - C) \cdot (Q - C) - r^2 = 0$$

$$t_{\pm} = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

$$= - ((Q - C) \cdot D) \pm \sqrt{((Q - C) \cdot D)^2 - (Q - C) \cdot (Q - C) + r^2}$$

when t has no two real value($b^2 - 4ac < 0$), there will be no intersection.
the intersecting point $Q + tD$ will be calculated with t value of smallest positive in $t_{\pm}$. If two values in$t_{\pm}$ are both negative, there is no intersection. After computing the intersection point, the normal of the sphere is the point - center of the sphere. Therefore, the normal is

$$Normal = normalize(P - C)$$

the texture coordinate of the sphere in the intersection point can be calculated as

$$\theta = atan2(Normal.Y, Normal.X), \quad -\pi \leq \theta \leq \pi$$

$$\Phi = acos(Normal.Z), \quad 0 \leq \Phi \leq \pi$$

$$texcoord = (\frac{\theta}{2*\pi} + 0.5, \frac{\Phi}{\pi}), \ 0 \leq texcoord \leq 1$$

The program sets the texture coordinate range into [0, 1].
It is implemented on line 62 in acceleration.cpp

Axis-Aligned Box Intersection

Generate three axis-aligned slabs with

N=(1,0,0), d0 = -corner.x, d1 = -corner.x - diagonal.x
N=(0,1,0), d0 = -corner.y, d1 = -corner.y - diagonal.y
N=(0,0,1), d0 = -corner.z, d1 = -corner.z - diagonal.z

for each slab, query the intersection between ray and slab

$$t_0 = -\frac{d_0 + N \cdot Q}{N \cdot D}$$

$$t_1 = -\frac{d_1 + N \cdot Q}{N \cdot D}$$

order value t0 and t1 so that to<t1. the normal value at the t0 and t1 goes -N, N
when $N \cdot D$ goes 0, calculate s0 and s1

$$s_0 = N \cdot Q + d_0$$

$$s_1 = N \cdot Q + d_1$$

when s0 and s1 sign is different, the t goes [0, Infinite], or t goes empty.
compute each t0 and t1 value with slab and get the maximum t0 and minimum t1.
compare the t0 and t1 values and set the positive smallest value as t.
The intersection point is Q + tD.
The normal of the intersection point is attached N that came from slab intersection check.

For the UV, the program calculates (the intersection point - the center of the intersecting plane) / half of intersecting plane size(with range [-1, 1]). Adjust the range by multiplying by 0.5 and adding 0.5(with range [0, 1]).
It is implemented on line 94 in acceleration.cpp

Cylinder Intersection

Transform the ray into the cylinder space.

$$A = normalize(Axis_{cylinder})$$

$B = normalize(VXA)$, where V is (1,0,0) or (0,0,1) that is not parallel to A

$$C = A \times B$$

$$Rot_{cylinder} = transpose([B\ C\ A])$$

$$\overline{Q} = Rot_{Cylinder}(Q - Base)$$

$$\overline{D} = Rot_{Cylinder}(D)$$

Do intersection check with slab N=(0,0,1), d0=0, d1=-length(axis) => a0, a1. Intersection with z-aligned cylinder

$$(Q_x, Q_y, Q_z) = \overline{Q}$$

$$(D_x, D_y, D_z) = \overline{D}$$

$$x^2 + y^2 - r^2 = 0$$

$$t^2(D_xD_x + D_yD_y) + 2t(D_xQ_x + D_yQ_y) + (Q_xQ_x + Q_yQ_y - r^2) = 0$$

$$t_{\pm} = \frac{-(D_xQ_x+D_yQ_y)\pm\sqrt{(D_xQ_x+D_yQ_y)^2-(D_xD_x+D_yD_y)\cdot(Q_xQ_x+Q_yQ_y-r^2)}}{D_xD_x+D_yD_y}$$

$N = (Q_x + tD_x, Q_y + tD_y, 0)$, where N is normal that is attached in t

then, get the t0 = max(a0, t-), t1=min(a1,t+). the smallest positive value between t0 and t1 will be t.
The intersection point is Q + tD.
The normals are attached normal(in cylinder space) with t then calculate the world space normal with transpose(R) * normal
The UV is calculated by normal in cylinder space

$$\theta = atan2(Normal.Y,\ Normal.X),\ -\pi \le \theta \le \pi$$

$height = (P - Base) \cdot normalize(axis)$, where P is intersection point

$$(u, v) = (\frac{\theta}{2\pi} + 0.5, \frac{height}{length(axis)})$$

It is implemented on line 181 in acceleration.cpp

Triangle Intersection

let

$$E_1 = V_1 - V_0$$
$$E_2 = V_2 - V_0$$
$$S = Q - V_0$$

set triangle as

$$V_0 + \mu E_1 + \nu E_2,\ \mu \geq 0, \nu \geq 0, 1 - \nu - \mu \geq 0$$
$$Q + tD = V_0 + \mu E_1 + \nu E_2$$
$$- tD + \mu E_1 + \nu E_2 = S$$
$$d = det(- D, E_1, E_2) = (D \times E_2) \cdot E_1$$
$$\mu = det(- D, S, E_2) = (D \times E_2) \cdot S$$
$$\nu = det(- D, E_1, S) = (S \times E_1) \cdot D$$
$$t = det(S, E_1, E_2) = (S \times E_1) \cdot E_2$$

The intersection point is Q + tD.
The normals

$$N = (1 - \nu - \mu)N_0 + \mu N_1 + \nu N_2$$

or

$$N = E_2 \times E_1$$

The UV is calculated by normal in cylinder space

$$UV = (1 - \nu - \mu)T_0 + \mu T_1 + \nu T_2$$

or set UV (0,0)

It is implemented on line 288 in acceleration.cpp

Ray

The program generates rays that cast from eye to pixel on the screen. The center of each pixel is generated with ((2 * X + 1) / width - 1, (2 * Y + 1) / height - 1) where X and Y is pixel index. Since this position is in screen space, the program transforms the position in world space.

```
float rx = ry * width / (float)(height);
vec3 X = rx * transformVector(orient,(1,0,0)); (orient is rotation matrix of camera)
vec3 Y = ry * transformVector(orient, (0,1,0));
vec3 Z = transformVector(orient, (0,0,1));
vec3 D = normalize(dx * X + dy * Y - Z);
```

Then, cast 2D ray with Q + tD where Q is the position of the camera.
It is implemented on line 288 in raytrace.cpp

BVH

The BVH library is a bounding volume hierarchy library that is useful for ray tracing algorithms. After a program registers the axis-aligned bounding box to the library, the library will call a pre-registered callback function when the user inputs a certain ray. In this project, the pre-registered callback function is intersect(), which gets the intersection data between ray and shape.

AABB(axis-aligned Bounding Box)

The AABB is generated before the main loop(generating rays and casting rays). The AABB is generated with the following

Sphere : (C + (r, r, r)), (C - (r, r, r)) is min and max where C is center of sphere and r is the radius of sphere

Cylinder: B is base of the bottom, r is the radius of bottom plane, A is the axis. With the points ((B + (r, r, r), (B - (r, r, r), (B + A + (r, r, r)), (B + A - (r, r, r))), the minimum x, minimum y, minimum z is min and maximum x, maximum y, maximum z is max.

Box: same with AABB

Triangle: With the points V0, V1, V2, the minimum x, minimum y, minimum z is min and maximum x, maximum y, maximum z is max.

Performance

The following table shows the time for querying ray with BVH or not. The time is managed with the Chrono library. This time management only measures elapsed time that takes generating rays and casting rays.

	1 attempt	2 attempt	3 attempt	4 attempt	Average
bunny, no BVH, >70000 triangles	> 3 min				
dwarf, no BVH, >2000 triangles	32.816 sec	32.618 sec	32.721 sec	32.663 sec	32.705 sec
no mesh, no BVH, 10 triangles	0.865 sec	0.770 sec	0.799 sec	0.771 sec	0.801 sec
all mesh, BVH, > 70000 triangles	0.285 sec	0.283 sec	0.284 sec	0.286 sec	0.285 sec
no bunny, BVH, 10 triangles	0.252 sec	0.248 sec	0.254 sec	0.249 sec	0.251 sec

We can show that the 4th option and 5th option do not have much difference. In a simple scene, there will be no big difference between a large number of triangles and a small number of triangles when using the BVH.