



**DIPARTIMENTO DI INGEGNERIA INFORMATICA, AUTOMATICA E
GESTIONALE “ANTONIO RUBERTI”**

FOUNDATION OF COMPUTER GRAPHICS

PATHTRACING ALGORITHM

BY ALEX UGOCHUKWU GBENIMACHOR

MATRICOLA 1722987

ABSTRACT

Path tracing is a sophisticated photo-realistic algorithm that leverages on Monte Carlo integration to render three-dimensional images, incorporating global illumination to replicate real-world image and graphic properties.

It stands as one of the most advanced algorithms developed for elevating the quality of image rendering in movies and cartoons. However, it demands substantial computational power to execute efficiently.

Path tracing utilizes a diverse range of algorithms to achieve lifelike photos and images in real-time applications. It integrates ray tracing techniques to simulate and enhance various visual effects, including soft shadows, depth of field, motion blur, and ambient occlusion.

This report explores the intricacies of the path tracing algorithm, its implementation, and its implications in the realm of computer graphics.

Introduction

The primary objective of this project is to design and implement a path tracing algorithm employing Monte Carlo integration to accurately simulate the intricate interplay of light with various object materials.

The inception of the path tracing algorithm dates back to 1979 when it was first introduced by Whitted Turner. Nevertheless, the watershed moment in its development occurred in 1986 when Jim Kajiya proposed a groundbreaking algorithm for light transport, utilizing Monte Carlo integration to tackle more complex lighting integral equations.

The overarching goal of this project is to construct a path tracer from the ground up, employing C++ and OpenGL GLSL, with the purpose of rendering scenes that authentically replicate real-life lighting effects on objects through the utilization of light rays.

There are several fundamental principles that underpin the successful implementation of path tracing:

Principle of Global Illumination: This principle entails the comprehensive assessment of enclosed scenes, where the behavior of each object concerning the incident illumination within the environment is meticulously observed.

Principle of Equivalence: This second principle posits that there exists no inherent distinction between emitted illumination originating from a light source and the light rays that are reflected from the surfaces within the scene.

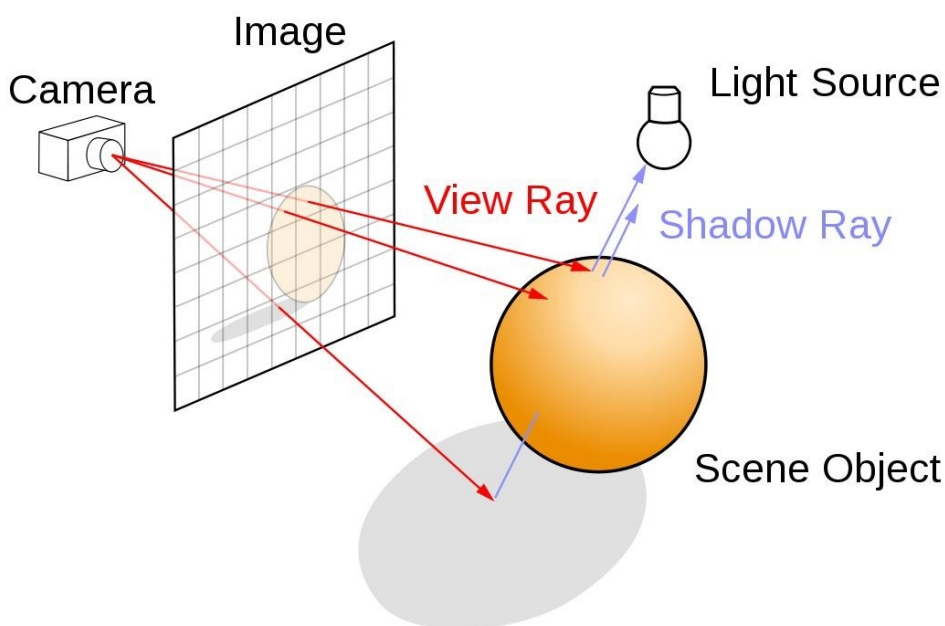
Principle of Direction: The Principle of Direction dictates that the scattered illumination emanating from surfaces must follow a specific trajectory. This trajectory is determined by a function of the incoming direction of the incident illumination and the outgoing direction, which is sampled using the Bidirectional Reflectance Distribution Function (BRDF) algorithm.

This report delves into the intricacies of the path tracing algorithm, its underlying principles, and the methodologies used in its implementation. In addition, it explores the algorithm's capacity to replicate real-world lighting phenomena and its relevance in the realm of computer graphics.

Method

In this section, we will provide an overview of the methods employed in this project, including ray tracing, Blinn-Phong shading, the Bidirectional Reflectance Distribution Function (BRDF), Monte Carlo integration, and the path tracing algorithm.

Ray Tracing:

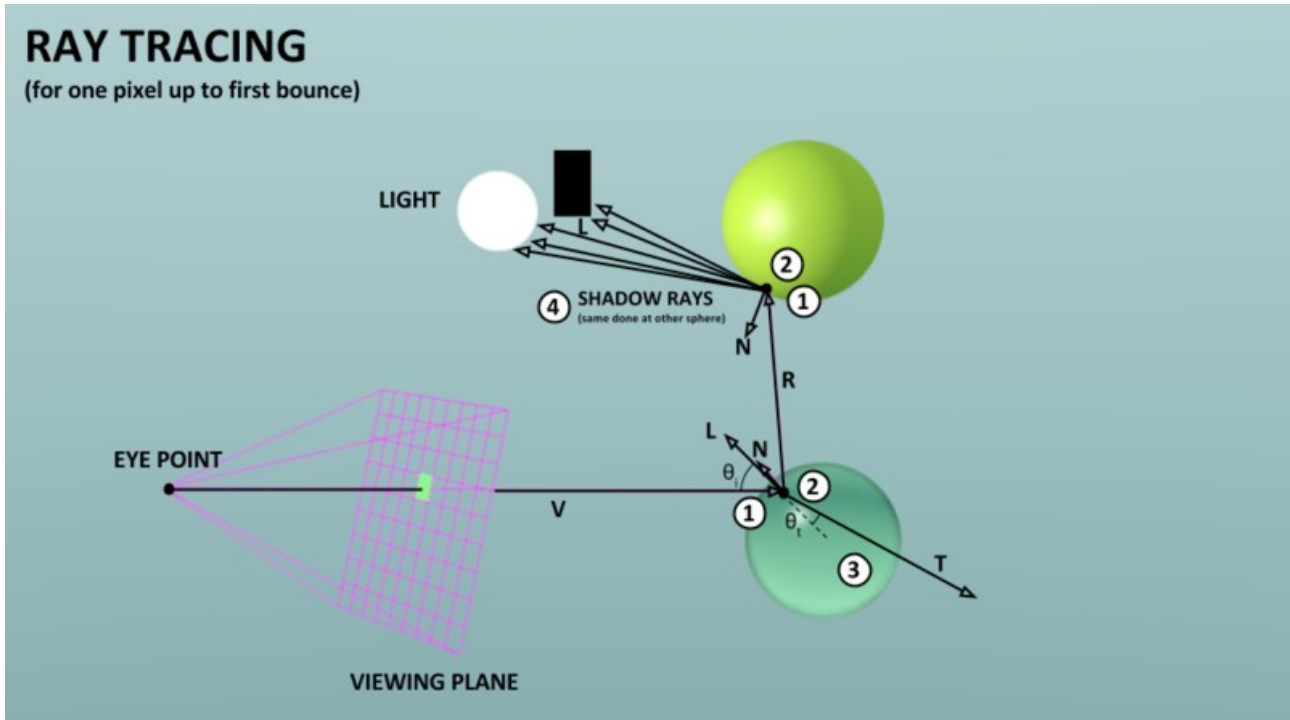


(NVIDIA DEVELOPER (2018))

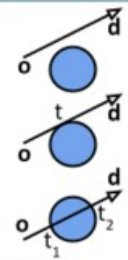
Ray tracing is a prominent graphic rendering algorithm used to simulate various lighting effects on scenes and objects, including accurate reflections, shadows, and refractions.

It involves generating computer-generated images by tracing the paths of light from the camera's viewpoint, assessing the lighting effects within a 3D scene, and determining their impact on the 2D viewing plane (pixel plane) and then back to the light source (NVIDIA DEVELOPER, 2018). The key components of the ray tracing algorithm include:

Ray tracing algorithm



① Sphere equation: $(\vec{p} - \vec{c}) \cdot (\vec{p} - \vec{c}) = r^2$ Intersection: $(\vec{o} + t\vec{d} - \vec{c}) \cdot (\vec{o} + t\vec{d} - \vec{c}) = r^2$
 Ray equation: $\vec{r}(t) = \vec{o} + t\vec{d}$
 $t^2 (\vec{d} \cdot \vec{d}) + 2(\vec{o} - \vec{c}) \cdot \vec{d} + (\vec{o} - \vec{c}) \cdot (\vec{o} - \vec{c}) - r^2 = 0$



② Illumination Equation (Blinn–Phong) with recursive Transmitted and Reflected Intensity:

$$I = k_a I_a + I_i \left(k_d (\vec{L} \cdot \vec{N}) + k_s (\vec{V} \cdot \vec{R})^n \right) + \underbrace{k_t I_t + k_r I_r}_{\text{recursion}}$$

③ Snell's law: $\frac{\sin \theta_1}{\sin \theta_2} = \frac{v_1}{v_2} = \frac{n_2}{n_1}$ $n_{air} \sin \theta_i = n_{glass} \sin \theta_t$ refraction coefficients:
 $n_{air} = 1, n_{glass} = 1.5$

④ Area Light Simulation: $I_{light} \frac{\#(\text{visible shadow rays})}{\#(\text{all shadow rays})}$



(By Nikolaus Leopold (Mangostaniko) - Own work, CC0, <https://commons.wikimedia.org/w/index.php?curid=58447039>)

BRDF (Bidirectional Reflectance Distribution Function):

The BRDF algorithm defines the reflectance of a surface for a given combination of incoming and outgoing light directions. It quantifies how much light is reflected in a specific direction when a certain amount of light is incident from another direction, considering the properties of the object's surface. The BRDF Rendering Equation is used to calculate the reflected light (L) at a point (P) with a particular direction (ω) (Pellacini, F.). A simplified version for a Lambertian surface is as follows:

$$L(P, \omega) = L_i + \int f_r(P, \omega_i, \omega_o) L(P, \omega_o) (\omega_i \cdot n) d\omega_i$$

$$\text{brdfLambertian} = (\text{diffuseReflectance} / \pi) * \text{dot}(N, L);$$

Monte Carlo Integration:

Monte Carlo integration is a technique that employs sampling to estimate the values of integrals. It offers a flexible approach for approximating integral values by evaluating the integrand at arbitrary points, making it versatile and applicable to various problems (Pellacini, F.).

$$I = \int_a^b f(x) dx$$

Pellacini, F. (n.d.)

MONTE CARLO RAY GENERATION

The **MONTE CARLO RAY GENERATION**, is used to generate random amount of ray direction samples, used to perform path-tracing on objects in the scene. This code is used to generate large samples of rays into the scene and also use these generated rays to compute the ray properties.

```
//second generated sample...
// fourth generated sample...
vec3 GenerateRayDirection(vec2 Pixel, int NumSample){
    float px =0.0, py =0.0;
    float scale = tan(fov / 2 * M_PI / 180);
    float Nsqr = 1/sqrt(NumSample);
    float lowerbound = -1.0, upperbound =1.0;
    float randx = lowerbound + random(Pixel) * (upperbound-lowerbound);//creating random X
    float randy = lowerbound + random(Pixel) * (upperbound-lowerbound);//creating random Y
    //float h = tan(theta/2);
    float viewport_height = 2.0 * scale;
    float viewport_width = Iresolution.x / Iresolution.y * viewport_height;
    float focal_length = 1.0;

    vec3 horizontal = vec3( viewport_width, 0, 0);
    vec3 vertical = vec3(0, -viewport_height, 0);

    vec3 origin = cameraPos.zxy * vec3(-1, 1, 1);
    vec3 lower_left_corner = origin - horizontal/2 - vertical/2 - vec3(0, 0, focal_length);

    vec2 uv = (Pixel.xy) / Iresolution.xy;

    for( int i = 0; i<Iresolution.x; i++){
        for(int j = 0; j< Iresolution.y; j++){
            vec3 pixel_center = lower_left_corner + (i * uv.x) + (j*uv.y);//the pixel centre...
            for(int ii = 0; ii< NumSample; ii++){
                for(int jj = 0; ii<NumSample; jj++){
                    px = -0.5 + sqrt(NumSample) * (ii + randx);
                    py = -0.5 + sqrt(NumSample) * (jj + randy);

                    vec3 direction = pixel_center + vec3(px , py, -1.0);

                    return direction;
                }
            }
        }
    }
```

C ▾ Tab Width: 8 ▾

Ln 143, Col 4

Bounding Volume Hierarchy (BVH):

Bounding Volume Hierarchy (BVH) serves as a widely adopted ray tracing acceleration technique. BVH employs a tree-based acceleration structure containing hierarchically organized bounding boxes that encompass different portions of the scene geometry or primitives. This approach significantly improves efficiency by enabling a depth-first tree traversal process for ray intersection testing instead of checking every primitive in the scene. BVH structures are constructed from source geometry before rendering a scene for the first time.

Subsequent frames may require either a new BVH built operation or BVH refitting to account for scene changes.

(NVIDIA DEVELOPER (2018))

(By Schreiberx - Own work, CC BY-SA 3.0, <https://commons.wikimedia.org/w/index.php?curid=17853864>)

BVH code snippet from the pathtracer_fs.fs:

the BVH code contains all the intersection algorithm :

```
AnyHit PopulateBVHnode(Ray r, vec3 min, vec3 max){
    int N = 10* 2 - 1;
    BVH bvh[10* 2 - 1];
    //Object objects;
    AnyHit NearHitObject, Csph, Cbox, Cpln, Cpyr, Ccon;

    BoundingBox rootBox = BoundingBox(vec3(-1.0), vec3(1.0));

    bool hit = intersectBox(r, rootBox);
    int i = 0;
    for(int l = 0; l < N; l++){

        if(!hit){
            if(bvh[l].parent == 0.0){
                bvh[l].parent = 0.0;
            }

            if(objects.objType == 2){
                //Sphere intersection
                Csph = SphereIntersection(objects.position, objects.objSize.x, r);
                if((Csph.t > 0) || (Csph.t < 0)){ // && ((Csph.t < Cbox.t) || (Cbox.t < 0)) && !(bvh[l].parent < 0.0)){
                    i = 2*i+1;
                    bvh[l].right = Csph.t;
                    bvh[l].objIdx = 2;
                    NearHitObject = Csph;
                    NearHitObject.objIdx = 2;
                }
            }

            if(objects.objType == 3){
                //Box intersection...
                Cbox = BoxIntersection(objects.position, objects.objSize, r);
                if((Cbox.t < 0) || (Cbox.t > 0)){ // && ((Cbox.t < Csph.t) || (Csph.t < 0)) && !(bvh[l].parent < 0.0)){
                    i = 2*i+1;
                    bvh[l].left = Cbox.t;
                    NearHitObject = Cbox;
                    bvh[l].objIdx = 3;
                    NearHitObject.objIdx = 3;
                    NearHitObject.hit = true;
                }
            }

            if(objects.objType == 1){
                //floor, wall intersection
                Cpln = PlaneIntersection(vec3(0.0), vec3(0.0, 1.0, 0.0), r); //plane intersection...
                if(((Cpln.t > 0) || (Cpln.t < 0)) && !(bvh[l].parent < 0.0)){
                    i = 2*i+1;
                    bvh[l].right = Cpln.t;
                    NearHitObject = Cpln;
                    bvh[l].objIdx = 1;
                    NearHitObject.objIdx = 1;
                    NearHitObject.hit = true;
                }
            }
        }
    }
}
```

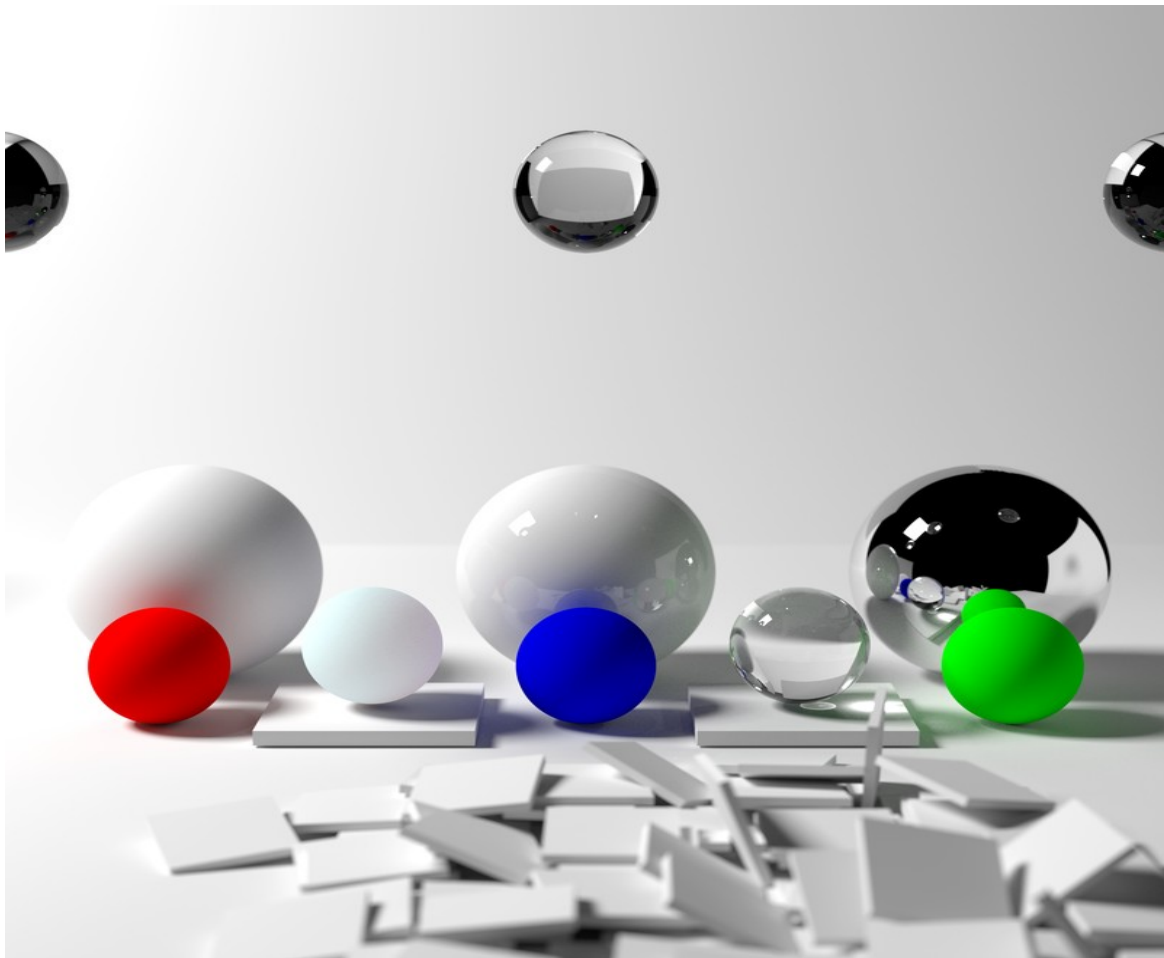
Below is the traversal snippet:

```
//Traversal algorithm...
AnyHit traverseTree(Ray r){
    vec3 min = vec3(-1.0);
    vec3 max = vec3(1.0);
    AnyHit Hc = PopulateBVHnode(r, min, max);
    return Hc;
}
```

Path Tracing Algorithm:

The path tracing algorithm is a Monte Carlo-based rendering technique frequently employed in computer graphics to simulate various lighting effects. This algorithm traces the path of light from objects back to the light source, allowing for the rendering of 3D scenes with global illumination. Path tracing fundamentally relies on Monte Carlo integration to compute the total illuminance arriving at a single point on an object's surface. It combines these elements with the previously mentioned BRDF algorithm to determine how much of the surface reflectance contributes to the camera viewpoint. The path tracing algorithm effectively simulates numerous effects, including soft shadows, reflections, refraction, and more.

In the subsequent sections of this report, we will delve deeper into the practical implementation of these methods and their implications in the context of computer graphics and photo-realistic rendering.



(By Qutorial-Own work, CC BY-SA 4.0,
<https://commons.wikimedia.org/w/index.php?curid=49847721>)

Pseudocode of the Algorithm

```
Color TracePath(Ray ray, count depth) {  
    if (depth >= MaxDepth) {  
        return Black; // Bounced enough times.  
    }  
  
    ray.FindNearestObject();  
    if (ray.hitSomething == false) {  
        return Black; // Nothing was hit.  
    }  
  
    Material material = ray.thingHit->material;  
    Color emittance = material.emittance;  
  
    // Pick a random direction from here and keep going.  
    Ray newRay;  
    newRay.origin = ray.pointWhereObjWasHit;  
  
    // This is NOT a cosine-weighted distribution!  
    newRay.direction = RandomUnitVectorInHemisphereOf(ray.normalWhereObjWasHit);  
  
    // Probability of the newRay  
    const float p = 1 / (2 * PI);  
  
    // Compute the BRDF for this ray (assuming Lambertian reflection)  
    float cos_theta = DotProduct(newRay.direction, ray.normalWhereObjWasHit);  
    Color BRDF = material.reflectance / PI;  
  
    // Recursively trace reflected light sources.  
    Color incoming = TracePath(newRay, depth + 1);  
  
    // Apply the Rendering Equation here.  
    return emittance + (BRDF * incoming * cos_theta / p);  
}
```

(WIKIPEDIA (2023))

PATHTRACING CODE SNIPPET FROM PATHTRACER_MAIN_FS.FS

```
vec3 PathTracer(Ray r){
    int NBounce = 0;
    AnyHit Hc = traverseTree(r);//traverse Nodes
    vec3 color ;
    for(int bounce = 0; bounce<NumBounce; bounce++){
        NBounce = NBounce + 1;
        //floor lighting -> basic square
        if(Hc.objIdx == 1 && objects.objType == 1){//trace floor...
            vec3 hit = normalize(r.direction - r.origin);
            vec3 jitteredLight = lightPos + r.direction;
            vec3 L = normalize(jitteredLight - hit);
            // ambient
            float ambientStrength = 0.2;
            vec3 ambient = ambientStrength * lightColor;

            // diffuse
            vec3 norm = normalize(Normal);
            vec3 lightDir = normalize(L - FragPos);
            float diff = max(dot(norm, lightDir), 0.0);
            vec3 diffuse = diff * lightColor;

            // specular
            float specularStrength = 0.5;
            vec3 viewDir = normalize(-FragPos); // the viewer is always at (0,0,0) in view-space, so viewDir is (0,0,0) - Position
            vec3 reflectDir = reflect(-lightDir, norm);
            float spec = pow(max(dot(viewDir, reflectDir), 0.0), 32);
            vec3 specular = specularStrength * spec * lightColor;
        }
    }
}
```

OPENGL APPLICATION

OpenGL (Open Graphics Library) is a cross-language, cross-platform application programming interface (API) for rendering 2D and 3D vector graphics. It was developed and maintained by Khronos Group.

It uses a file called **Shader**, this file is written in C language format GLSL. The shader is divided into two: we have **vertex shader** and the **fragment shader**.

A **shader** typically has the following structure:

```
#version version_number
in type in_variable_name;
in type in_variable_name;
out type out_variable_name;
```

Vertex shader:

The Vertex Shader is the programmable Shader stage in the rendering pipeline that handles the processing of individual vertices. Vertex shaders are fed Vertex Attribute data, as specified from a vertex array object by a drawing command. The vertex shader is where we compute the position of the , texture and color/normal of these objects found the scene. With these, we can control the position of the object.

(khronos (2017))

structure of the vertex shader:

```
#version 330 core
layout (location = 0) in vec3 position; // The position variable has
    attribute position 0

out vec4 vertexColor; // Specify a color output to the fragment shader

void main()
{
    gl_Position = vec4(position, 1.0); // See how we directly give a vec3
    to vec4's constructor
    vertexColor = vec4(0.5f, 0.0f, 0.0f, 1.0f); // Set the output variable
    to a dark-red color
}
```

Fragment shader

A Fragment Shader is the Shader stage that will process a Fragment generated by the Rasterization into a set of colors and a single depth value. Most of the Path tracing algorithm, goes into these, that is where the texturing, coloring and main computation codes is done. That the objects are presentable and the texture and color are well applied is done in this program, called the **fragment shader**.

(Khronos Group (2020))

structure of the Fragment shader:

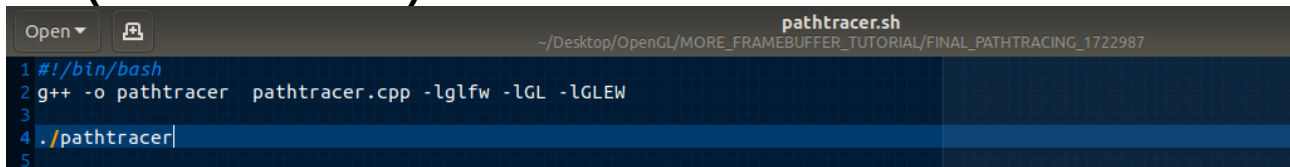
```
#version 330 core
in vec4 vertexColor; // The input variable from the vertex shader (same
    name and same type)

out vec4 color;

void main()
{
    color = vertexColor;
}
```

HOW TO RUN PROGRAM

To run the program, you will need to have **opengl 4 or later** installed on the machine, you will need to have the following **api or program** installed on the computer - they are (-lglfw -lGL -lGLEW) to run the **.sh files(executable files)**.

A screenshot of a terminal window. The title bar shows 'pathtracer.sh' and the path '~/Desktop/OpenGL/MORE_FRAMEBUFFER_TUTORIAL/FINAL_PATHTRACING_1722987'. The terminal content shows a sequence of commands: line 1: '#!/bin/bash', line 2: 'g++ -o pathtracer pathtracer.cpp -lglfw -lGL -lGLEW', line 3: (empty), line 4: './pathtracer|', and line 5: (empty).

```
1 #!/bin/bash
2 g++ -o pathtracer pathtracer.cpp -lglfw -lGL -lGLEW
3
4 ./pathtracer|
5
```

RESULT OF PATH TRACING

FOR THE PATH TRACING PROGRAM, to execute the program, one needs to make bash executable-file using **chmod +x** , to this file bash file: **./pathtracer.sh**

CREATING PRIMITIVES OBJECTS IN THE SCENE

The path tracing algorithm was performed on several primitives objects and the object vertices are generated from pathtracer.cpp and also the shader programer is also written in pathtracer.cpp: They Squares, Cubes, Spheres, Pyramids and cones.

The Floor Object, Light Object at the roof of the box, the walls: These objects are made from squares of different sizes, which are presented as scales. Floor, the walls and roof are scaled by 20.

SQUARE PRIMITIVES (1):

The Floor/Wall/Roof/light snippet Code is given Below.

First code is the code responsible for generating the vertices of the code:

In the code Below, we can see the vertices, the color/normal and textCoord(Texture Coordinate for apply texture to the squares.)

```
//SQUARE OBJECT
rectangle halfTriangle(vector<vec3>{//coordinates
    vec3(0.5, 0.5, 0.0),
    vec3(0.5, -0.5, 0.0),
    vec3(-0.5, 0.5, 0.0),
    vec3(0.5, -0.5, 0.0),
    vec3(-0.5, -0.5, 0.0),
    vec3(-0.5, 0.5, 0.0)},
    //Normal
    vector<vec3>{vec3(0.0, 1.0, 0.0),
    vec3(1.0, 1.0, -1.0),
    vec3(1.0, 1.0, 1.0),
    vec3(0.0, 1.0, 0.0),
    vec3(-1.0, 1.0, 1.0),
    vec3(-1.0, 1.0, -1.0) },
    //Textcoord
    vector<vec2>{vec2( 1.0f, 1.0f), //1
    vec2( 1.0f, 0.0f), //2
    vec2(0.0f, 1.0f), //3
    vec2( 1.0f, 0.0f),
    vec2( 0.0f, 0.0f),
    vec2(0.0f, 1.0f)}));

vector<vec3> recVertices;
vector<vec3> recColorVert;
vector<vec2> recTexCoord;
for (unsigned int ix=0; ix<halfTriangle.vertexPos.size(); ix++){
    recVertices.push_back(halfTriangle.vertexPos[ix]);
    recColorVert.push_back(halfTriangle.colorPos[ix]);
    recTexCoord.push_back(halfTriangle.texturePos[ix]);
}
```


Second code is the that helps to render these objects through the shader program and we have all the properties need to render, light, colors and also the model, camera view etc to help position these objects in place.

Floor - object code:

Floor object pos(Object position) which is set at Vec3(0.0), which means in front of the camera.

FloorScaleT – Scale or the Size of the floor set to vec3(20).

Model matrix is performs, the scaling, rotaton and the translation of the floor.

```
//OBJECT EMISSIVE PROPERTY...
//float EMISSIVE = 0, METAL =1, DIELECTRIC =2, LAMBERTIAN=3;
const int LAMBERTIAN = 0x00000001;
const int METAL = 0x00000002;
const int DIELECTRIC = 0x00000004;
const int EMISSIVE = 0x00000008;

vec3 cameraPosition = camera.Position;
vec3 cameraDirection = camera.Front;

glm::mat4 model = glm::mat4(1.0f);
glm::mat4 projection = glm::perspective(glm::radians(camera.Zoom), (float)width / (float)height, 0.1f, 100.0f);
glm::mat4 view = camera.GetViewMatrix();

//floor
glUseProgram(ProgramID);
int FloorObType = 1;
objectPos = vec3(0.0);
vec3 FloorScaleT = vec3(20.0f);
model = scale(model, FloorScaleT);
model = rotate(model, radians(90.0f), vec3(0.0, 0.0, 1.0));
model = rotate(model, radians(90.0f), vec3(0.0, 1.0, 0.0));
vec3 floorColor = vec3(1.0);

glUniform3f(glGetUniformLocation(ProgramID, "objects.objSize"), FloorScaleT.x, FloorScaleT.y, FloorScaleT.z);
glUniform1i(glGetUniformLocation(ProgramID, "objects.objType"), FloorObType);
glUniform3f(glGetUniformLocation(ProgramID, "objects.position"), objectPos.x, objectPos.y, objectPos.z);
glUniform3f(glGetUniformLocation(ProgramID, "objects.objcolor"), floorColor.x, floorColor.y, floorColor.z);
```

Wall Code (Similar to the floor code)

This is the Left Wall code, for rendering the left wall, same code for the others, the difference is the positioning of the objects.

```
//left side
int leftSideObjType = 1;
mat4 leftmodel = glm::mat4(1.0f);
vec3 leftSideScale = vec3(20.0f);
vec3 leftSidePosition = glm::vec3(-0.5f, 0.5f, 0.0);
leftmodel = scale( leftmodel, leftSideScale);
leftmodel = glm::translate( leftmodel, leftSidePosition);
leftmodel = rotate( leftmodel, radians(-90.0f), vec3(1.0, 0.0, 0.0));
leftmodel = rotate( leftmodel, radians(90.0f), vec3(0.0, -0.5, 0.0));

vec3 leftSideColor = vec3(0.0,1.0,0.0);
//glUseProgram(ProgramID);
glUniform3f(glGetUniformLocation(ProgramID, "objects.objSize"), leftSideScale.x, leftSideScale.y,
leftSideScale.z);
glUniform1i(glGetUniformLocation(ProgramID, "objects.objType"), leftSideObjType);
glUniform3f(glGetUniformLocation(ProgramID, "objects.position"), leftSidePosition.x,
leftSidePosition.y, leftSidePosition.z);
//glUniform3f(glGetUniformLocation(ProgramID, "objects.objcolor"), topLightColor.x, topLightColor.y,
topLightColor.z);
glUniform3f(glGetUniformLocation(ProgramID, "objects.objcolor"), leftSideColor.x, leftSideColor.y, leftSideColor.z);
glUniformMatrix4fv(glGetUniformLocation(ProgramID, "projection"), 1, GL_FALSE, glm::value_ptr(projection));
glUniformMatrix4fv(glGetUniformLocation(ProgramID, "view"), 1, GL_FALSE, glm::value_ptr(view));
glUniformMatrix4fv(glGetUniformLocation(ProgramID, "model"), 1, GL_FALSE, glm::value_ptr(leftmodel));
glBindVertexArray(BaseVAO);
glDrawArrays(GL_TRIANGLES, 0, 6);
```

Light Object:

The light object is similar to the floor but the difference is in the size and also in positioning and the color of the object.

```
//=====TOP LIGHT=====//

glUseProgram(LightProgramID);
//top light
mat4 topmodelLight = glm::mat4(1.0f);
//topmodelLight = scale( topmodelLight, vec3(1.0f));
topmodelLight = glm::translate( topmodelLight, glm::vec3(0.0f, 19.5f, 0.0));
topmodelLight = rotate( topmodelLight, radians(90.0f), vec3(0.0, 0.0, 1.0));
topmodelLight = rotate( topmodelLight, radians(90.0f), vec3(0.0, 1.0, 0.0));
topmodelLight = scale( topmodelLight, vec3(1.0f));
vec3 topSideColor = vec3(1.0f);

//glUseProgram(ProgramID);
glUniform3f(glGetUniformLocation(LightProgramID, "color"), topSideColor.x, topSideColor.y, topSideColor.z);
glUniformMatrix4fv(glGetUniformLocation(LightProgramID, "projection"), 1, GL_FALSE, glm::value_ptr(projection));
glUniformMatrix4fv(glGetUniformLocation(LightProgramID, "view"), 1, GL_FALSE, glm::value_ptr(view));
glUniformMatrix4fv(glGetUniformLocation(LightProgramID, "model"), 1, GL_FALSE, glm::value_ptr(topmodelLight));
glBindVertexArray(BaseVAO);
glDrawArrays(GL_TRIANGLES, 0, 6);
```

SPHERE PRIMITIVES(2).

the sphere primitive object is constructed using the radius and radian of 360 degrees to generate the vertices, the normal and the textCoord(texture coordinate).

This generate in the sphere class:

```
84 //create sphere class here...
85 class Sphere{
86
87     private:
88         int numVertices;
89         int numIndices;
90         vector<int> indices;
91         vector<vec2> texCoords;
92         vector<vec3> vertices;
93         vector<vec3> normals;
94         //void init(int);
95         //float toRadians(float degrees);
96
97     public:
98
99
100     Sphere(){//declare a default value Sphere precision
101         init(48);
102     }
103     Sphere(int prec){
104         init(prec);
105     };
106
107     int getNumVertices();
108     int getNumIndices();
109     vector<int> getIndices();
110     vector<vec3> getVertices();
111     vector<vec2> getTexCoords();
112     vector<vec3> getNormals();
113
114     float toRadians(int degrees){return (degrees * 2.0f * 3.14159f) / 360.0f;}
115 }
```

Below is the code, responsible for Fragment and vertex shader program in the main code pathtracer.cpp:

```
//spheres

//render sphere 0
int sphereType0 =2;
vec3 sphereColor0 = vec3(1.0, 0.0, 1.0);
mat4 spheremodel = glm::mat4(1.0f);
vec3 spherePosition0 = glm::vec3(2.0f, 1.5f, 1.0);
vec3 sphereScale0 = vec3(1.0);
spheremodel = scale(spheremodel, sphereScale0 );
spheremodel = glm::translate( spheremodel, spherePosition0);
spheremodel = glm::scale( spheremodel, sphereScale0);
glUniform3f(glGetUniformLocation(ProgramID, "objects.position"), spherePosition0.x,spherePosition0.y,
spherePosition0.z);
glUniform3f(glGetUniformLocation(ProgramID, "objects.objSize"), sphereScale0.x, sphereScale0.y, sphereScale0.z);
glUniform1i(glGetUniformLocation(ProgramID, "objects.objType"), sphereType0);
glUniform3f(glGetUniformLocation(ProgramID, "objects.objcolor"), sphereColor0.x,sphereColor0.y, sphereColor0.z);
glUniformMatrix4fv(glGetUniformLocation(ProgramID, "model"), 1, GL_FALSE, glm::value_ptr( spheremodel));
renderSphere(ProgramID);

//render sphere 1
```

CUBE PRIMITIVES(3).

Cube Vertex

The **Cube** is primitive is made of six squares, which is equal to 36 vertices. The Cube has normal and textCoord(texture Coordinate). Below is the code for generating for primitive:

Below is the code for position and texturing of the cube through the shader program

```
//CUBE OBJECT
void renderCube(int programID){

    //cube vertex
    vector<vec3> cubeVertices;
    vector<vec3> cubeColorVertices;
    vector<vec2> textureCoord;
    rectangle cubeFaces(vector<vec3>{//TOP SIDE 1
        vec3(-1.0f, -1.0f, -1.0f),
        vec3(1.0f, 1.0f, -1.0f),
        vec3(1.0f, -1.0f, -1.0f),
        vec3(1.0f, 1.0f, -1.0f),
        vec3(-1.0f, -1.0f, -1.0f),
        vec3(-1.0f, 1.0f, -1.0f),

        //RIGHT SIDE 2
        vec3(-1.0f, -1.0f, 1.0f),
        vec3(1.0f, -1.0f, 1.0f),
        vec3(1.0f, 1.0f, 1.0f),
        vec3(1.0f, 1.0f, 1.0f),
        vec3(-1.0f, 1.0f, 1.0f),
        vec3(-1.0f, -1.0f, 1.0f),

        //BOTTOM SIDE 3
        vec3(-1.0f, 1.0f, 1.0f),
        vec3(-1.0f, 1.0f, -1.0f),
        vec3(-1.0f, -1.0f, -1.0f),
        vec3(-1.0f, -1.0f, -1.0f),
        vec3(-1.0f, -1.0f, 1.0f),
        vec3(-1.0f, 1.0f, 1.0f),

        //LEFT SIDE 4
        vec3(1.0f, 1.0f, 1.0f),
```

CUBE SHADER PROGRAM:

```
//cube 1
int cubeType1 = 3;
float cubeRefIdx1 = 0.95;
int LambertCube1 = LAMBERTIAN;
vec3 cubeScale1 = vec3(1.0);
vec3 cubeColor1 = vec3(1.0, 2.0, 0.0);
mat4 cubemodel = glm::mat4(1.0f);
vec3 cubePosition1 = glm::vec3(-1.0f, 1.2f, 2.0);
cubemodel = glm::translate(cubemodel, cubePosition1);
//model = glm::rotate(model, glm::radians(60.0f), glm::normalize(glm::vec3(1.0, 0.0, 1.0)));
cubemodel = glm::scale(cubemodel, cubeScale1);
glUniform1f(glGetUniformLocation(ProgramID, "objects.objRefractIdx"), cubeRefIdx1);
glUniform1i(glGetUniformLocation(ProgramID, "objects.objMat"), LambertCube1);
glUniform3f(glGetUniformLocation(ProgramID, "objects.objSize"), cubeScale1.x, cubeScale1.y, cubeScale1.z);
glUniform3f(glGetUniformLocation(ProgramID, "objects.position"), cubePosition1.x, cubePosition1.y, cubePosition1.z);
glUniform1i(glGetUniformLocation(ProgramID, "objects.objType"), cubeType1);
glUniform3f(glGetUniformLocation(ProgramID, "objects.objcolor"), cubeColor1.x, cubeColor1.y, cubeColor1.z);
glUniformMatrix4fv(glGetUniformLocation(ProgramID, "model"), 1, GL_FALSE, glm::value_ptr(cubemodel));
```

PYRAMID PRIMITIVES(4).

The Primitives of the Pyramids is made of 18 vertices, it also contains textCoord(texture coordinate) and Normal coordinate. Below is the code for generating the pyramid primitive.

```
//Pyramid vertices...
void renderPyramid(int programID){

    // pyramid vertex
    triangleFaces triangleFace1(vector<vec3>{
        // FRONT FACE
        vec3(-0.5, -0.5, -0.5), // left
        vec3(0.5, -0.5, -0.5), // right
        vec3(0.0, 0.5, 0.0), // top
        // RIGHT FACE
        vec3(-0.5, -0.5, 0.5), // LEFT
        vec3(0.5, -0.5, 0.5), // RIGHT
        vec3(0.0, 0.5, 0.0), // TOP
        // BACK FACE
        vec3(-0.5, -0.5, -0.5), // LEFT
        vec3(-0.5, -0.5, 0.5), // RIGHT
        vec3(0.0, 0.5, 0.0), // TOP
        // LEFT FACE
        vec3(0.5, -0.5, -0.5), // LEFT
        vec3(0.5, -0.5, 0.5), // RIGHT
        vec3(0.0, 0.5, 0.0), // TOP
        // BASE LEFT
        vec3(-0.5f, -0.5f, -0.5f),
        vec3(0.5f, -0.5f, 0.5f),
        vec3(-0.5f, -0.5f, 0.5f),
        // BASE RIGHT
        vec3(0.5f, -0.5f, 0.5f),
        vec3(-0.5f, -0.5f, -0.5f),
        vec3(0.5f, -0.5f, -0.5f)
    }), // top
```

C++ Tab Width: 8 Ln 1094, Col 24

Pyramids Shader:

This code is in the Pathtracer cpp which is used as the others mentioned above for position, texturing and for performing intersection of the pyramid etc.

```
//pyramid
//spheres

//render pyramid 1
int pyramidType1 = 4;
vec3 pyramidColor = vec3(1.0, 0.25, 0.0);
float pyramidRefIdx = 0.0;
mat4 pyramidmodel = glm::mat4(1.0f);
vec3 pyramidScale1 = vec3(1.5f);
vec3 pyramidPosition1 = glm::vec3(-4.0f, 1.0f, 1.0);
pyramidmodel = scale(pyramidmodel, pyramidScale1);
pyramidmodel = glm::translate(pyramidmodel, pyramidPosition1);
//pyramidmodel = glm::scale(pyramidmodel, glm::vec3(1.0f));
glUniform1f(glGetUniformLocation(ProgramID, "objects.objRefractIdx"), pyramidRefIdx);
glUniform1i(glGetUniformLocation(ProgramID, "objects.objMat"), DIELECTRIC);
glUniform1i(glGetUniformLocation(ProgramID, "objects.objType"), pyramidType1);
glUniform3f(glGetUniformLocation(ProgramID, "objects.objSize"), pyramidScale1.x, pyramidScale1.y,
pyramidScale1.z);
glUniform3f(glGetUniformLocation(ProgramID, "objects.position"), pyramidPosition1.x, pyramidPosition1.y,
pyramidPosition1.z);
glUniform3f(glGetUniformLocation(ProgramID, "objects.objcolor"), pyramidColor.x, pyramidColor.y, pyramidColor.z);
glUniformMatrix4fv(glGetUniformLocation(ProgramID, "model"), 1, GL_FALSE, glm::value_ptr(pyramidmodel));
glBindVertexArray(PYRVAO);
glDrawArrays(GL_TRIANGLES, 0, 18);
//renderSphere(ProgramID);
renderPyramid(ProgramID);

//render pyramid2
int pyramidType2 = 4;
```


CONE PRIMITIVES(5).

The cone primitive is similar to a cylinder except that its sides are not parallel, it contains the TexCoord(Texture coordinate), 36 vertices and including the circle and vertices and Normal Coordinate.

```
//renders the cone object
void renderCone(float ConHeight,int ProgramID){
    vector<vec3> pts; //vertices
    vector<vec3> nts; // normals
    vector<vec2> tex; //text coordinates

    vec3 a = vec3(0.0, ConHeight, 0.0);
    vec3 b, c;
    b.y = c.y = 0.0;
    float t,s;
    float i = 0.05;
    int iSectorCount = 0;
    float fReciprocalPrecision = (float)(10 / 2*360);
    for (double angle = 0; angle<=360; angle+=10){
        c.x = b.x;
        c.z = b.z;
        b.x = cos(radians(angle));
        b.z = sin(radians(angle));

        // relative texture coordinates
        float fTextureOffsetS1 = iSectorCount * fReciprocalPrecision;
        float fTextureOffsetS2 = (iSectorCount + 1) * fReciprocalPrecision;
        float sa = a.x/ (2*360);
        float ta = a.z/ (2*360);

        float sb = b.x/ (2*360);
        float tb = b.z/ (2*360);
        //cout<<"fTextureOffsetS1:|t"<<fTextureOffsetS1/2*360<<"fTextureOffsetS2:|t"<<fTextureOffsetS2/2*360<<endl;
        float sc = c.x/ (2*360);
        float tc = c.z/ (2*360);

        if(angle!=0){
            //compute the Normal of the cone...
```

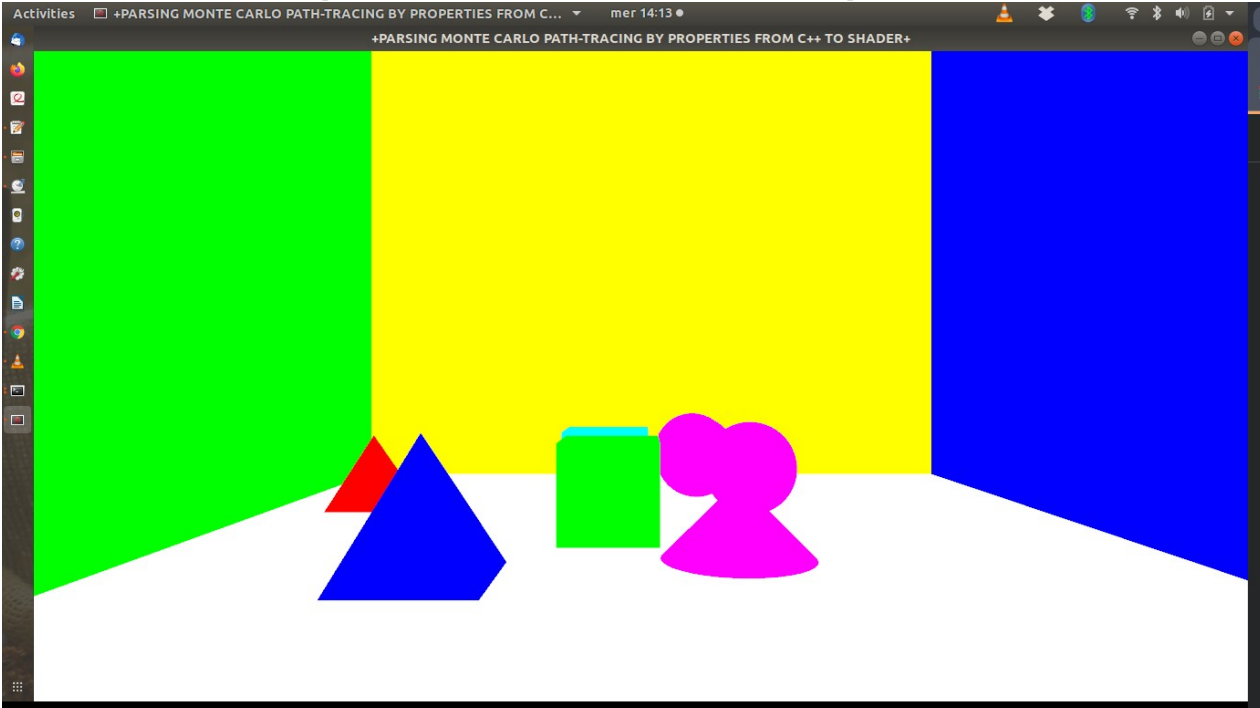
C++ ▾ Tab Width: 8 ▾ Ln 1094, Col 24

Cone Shader:

the Cone shader is used for texturing and positioning the cone and it's also used of the intersection algorithm of the cone.

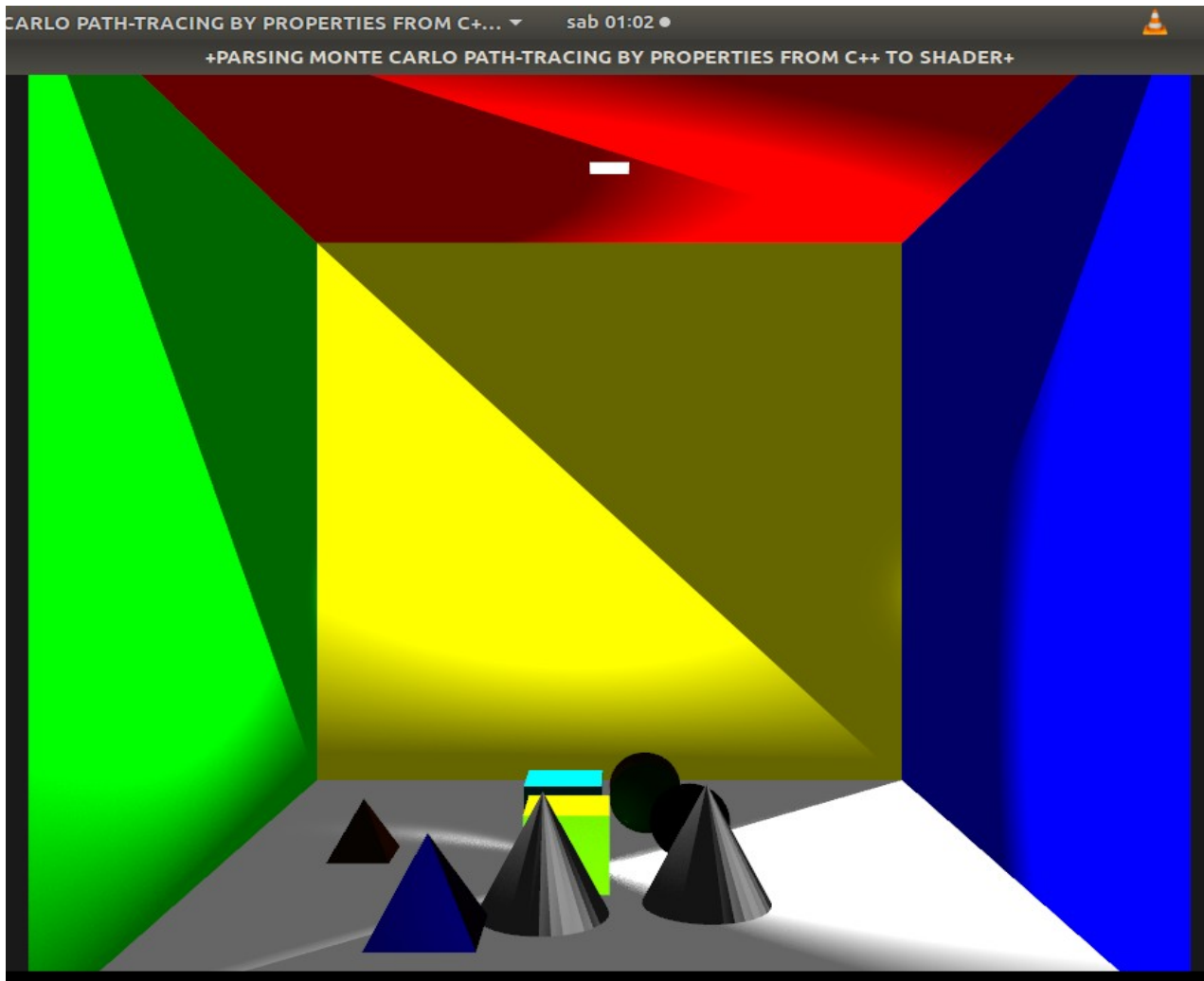
```
//cone 1
int coneType1 = 5;
float coneRefIdx1 = 0.5;
vec3 coneScale1 = vec3(1.5f);
vec3 coneColor1 = vec3(1.0, 0.0, 0.25);
mat4 Conemodel = mat4(1.0);
float ConHeight = 2.0;
vec3 conePosition1 = glm::vec3(1.5f, 0.0f, 2.0);
Conemodel = scale(Conemodel, coneScale1);
Conemodel = glm::translate( Conemodel, conePosition1);
//Conemodel = glm::scale(Conemodel, glm::vec3(1.0f));
glUniform1f(glGetUniformLocation(ProgramID, "objects.objRefractIdx"), coneRefIdx1);
glUniform1i(glGetUniformLocation(ProgramID, "objects.objMat"), METAL);
glUniform1i(glGetUniformLocation(ProgramID, "objects.objType"), coneType1);
glUniform3f(glGetUniformLocation(ProgramID, "objects.position"), conePosition1.x, conePosition1.y,
n1.z);
glUniform1f(glGetUniformLocation(ProgramID, "objects.objectHeight"),ConHeight);
glUniform3f(glGetUniformLocation(ProgramID, "objects.objSize"), coneScale1.x, coneScale1.y, coneScale1.z);
glUniform3f(glGetUniformLocation(ProgramID, "objects.objcolor"), coneColor1.x,coneColor1.y,coneColor1.z);
glUniformMatrix4fv(glGetUniformLocation(ProgramID, "model"), 1, GL_FALSE, glm::value_ptr(Conemodel));
glBindVertexArray(ConeVAO);
renderCone(ConHeight, ProgramID);
```


SCENE IMAGE 1.(Scene with out intersection)



SCENE IMAGE 2.(scene with intersection)

Scene is made of 2 pyramid, two cones , 2 cubes, 3 spheres, five square(wall, top, bottom, light, two sides).



WHAT IS RAY INTERSECTION?

(NB: the intersection algorithm, can be found in the pathtracer_main_fs.fs)

Ray intersection in ray tracing refers to the process of determining whether a ray emitted from the camera or light source intersects with any objects in the scene being rendered. In ray tracing, each pixel on the screen corresponds to a ray that is traced into the scene. These rays are typically cast from the camera's viewpoint through each pixel into the scene.

We have different type of ray intersection:

1. Ray Plane intersection, refers to the process of determining whether a ray, defined by its origin and direction, intersects with a plane in three-dimensional space, and if so, where it intersects.

A plane is defined by a normal vector, which indicates the direction perpendicular to the plane, and a point on the plane. Given this representation, the goal is to find the point where the ray intersects with the plane.

Mathematical equation is given as follows:

$\mathbf{P}(t) = \mathbf{O} + t\mathbf{D}$, where \mathbf{O} is the origin of the ray, \mathbf{D} is the direction vector of the ray, and t is a parameter, and a plane defined by its normal vector \mathbf{N} and a point \mathbf{P}_0 on the plane, the intersection point $\mathbf{P}(t)$ satisfies the equation: $(\mathbf{P}(t) - \mathbf{P}_0) \cdot \mathbf{N} = 0$

The is plane ray intersection, is used to intersect the floor, the wall etc.

BELOW IS CODE SNIPPET FOR PLANE INTERSECTION:

the Plane intersection code can be found in the Pathtracer_main_fs.fs, which is the fragment

```
//PLANE INTERSECTION ...
AnyHit PlaneIntersection(vec3 position, vec3 Normal, Ray r){

    AnyHit Hc;
    float denom = dot(Normal, r.direction);
    if(abs(denom)>0.0001f){

        vec3 oc = position - r.origin;

        Hc.t = dot(oc, Normal)/denom;
        Hc.t = 1.0;
        Hc.worldCoord = r.origin + r.direction * Hc.t;
        Hc.hit = true;
        Hc.Normal =normalize(Hc.worldCoord - oc);
        Hc.objIdx = 3;
        return Hc;

    }else{

        Hc.t = -1.0;
        Hc.hit = false;
        return Hc;
    }

};
```

Sphere intersection,

sphere intersection refers to the process of determining whether a ray intersects with a sphere-shaped object or volume within the scene being rendered. This involves finding the point or points where the ray intersects with the surface of the sphere.

$P(t) = O + tD$, this given as the ray algorithm, O (origin of the Ray), t (distance of the ray), ray direction D .

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

P is defined as the Point on the sphere and C is defined as the centre of the sphere, the dot product of the two, will give us the squared radius.

$p(t) = O + t D$ this is ray intersection point, which was defined earlier.

$$(O+tD-C) \cdot (O+tD-C) - R^2 = 0 .$$

**Ray formular for intersection after further expansion using the quadratic eqn below, we can use this to solve the discriminant:
 $at^2 + bt + c = 0$**

compute the discriminant:

$$d = \sqrt{b^2 - 4ac}$$

computing solution: $t = \frac{-b \pm d}{2a}$.

$b^2 - 4ac < 0 \Rightarrow$ No intersection

$b^2 - 4ac > 0 \Rightarrow$ Two solutions (enter and exit)

$b^2 - 4ac = 0 \Rightarrow$ One solution (ray grazes sphere)

BELOW IS THE SPHERE INTERSECTION SNIPPET FROM
PATHTRACER_MAIN_FS.FS:

```
//SPHERE INTERSECTION...
AnyHit SphereIntersection(vec3 position, float radius, Ray r){
    AnyHit Hc;
    Hc.hit = false;
    //float radius = objects.objSize.x; // radius...
    vec3 oc = r.origin - position; //object position
    float a = dot(r.direction, r.direction);
    float b = dot(2*r.direction, oc);
    float c = dot(oc, oc) - (radius*radius);
    //Solving  $d = (b^2) - 4 * a * c$ ;
    //(discriminant)
    float d = (b*b) - 4*a*c;

    if(d<0.0){
        Hc.t = -1;
        return Hc;
    }
    //picking the smallest t...
    float t1 = (-b + sqrt(d))/(2*a);
    float t2 = (-b - sqrt(d))/(2*a);

    //find the tNear and tFar...
    float tNear = min(t1, t2);
    float tFar = max(t1, t2);

    Hc.t = tNear;
```

BOX INTERSECTION ALGORITHM:

**(BOX INTERSECTION IS IMPLEMENTED IN THE
PATHTRACER_MAIN_FS.FS)**

In ray tracing, box intersection refers to the process of determining whether a ray intersects with a box-shaped object or volume within the scene being rendered. This involves finding the point or points where the ray intersects with the surfaces of the box.

We can re-iterate that the Intersect ray with each plane–Box is the union of 6 planes. (basic algorithm , 2012)

$x = x_1, x = x_2$

$y = y_1, y = y_2$

$z = z_1, z = z_2$

RAY-BOX INTERSECTION:

- 1. Intersect the ray with each plane**
- 2. Sort the intersections**
- 3. Choose intersection with the smallest $t > 0$ that is within the range of the box**

pseudo-code for box-intersection:

$txmin = (x_1 - ex) / Dx$ //assume $Dx > 0$

$txmax = (x_2 - ex) / Dx$

$tymin = (y_1 - ey) / Dy$

$tymax = (y_2 - ey) / Dy$ //assume $Dx > 0$

if $(txmin > tymax)$ or $(tymin > txmax)$

 return false

else

 return true

BOX-INTERSECTION ALGORITHM, BELOW IS THE CODE SNIPPET

```
//BOX INTERSECTION...

AnyHit BoxIntersection(vec3 position, vec3 boxSize, Ray r){

    AnyHit Hc;

    vec3 boxMin = position - boxSize / 2.0;
    vec3 boxMax = position + boxSize / 2.0;

    vec3 tMin = (boxMin - r.origin)/r.direction;
    vec3 tMax = (boxMax - r.origin)/r.direction;

    vec3 tMinDist = min(tMin, tMax);
    vec3 tMaxDist = max(tMin, tMax);

    //tFar and tNear for the boxes...
    float tNear = max(max(tMinDist.x, tMinDist.y), tMinDist.z);
    float tFar = min(min(tMaxDist.x, tMaxDist.y), tMaxDist.z);

    Hc.t = tNear;
    Hc.hit = false;

    /*if(tNear >= tFar || tFar <=0.0){
        Hc.t = -1.0;
        return Hc;
    }*/
    if(tFar < 0.0){
        Hc.t = -1.0;
        return Hc;
    }

}
```


CONE INTERSECTION:

In ray tracing, cone intersection refers to the process of determining whether a ray intersects with a cone-shaped object or volume within the scene being rendered. This involves finding the point or points where the ray intersects with the surface of the cone.

Here's a general overview of how cone intersection can be implemented in a ray tracing algorithm:

1. **Define the Cone Geometry**: First, you need to represent the cone geometry in your scene. This typically involves defining the apex, the direction of the axis of the cone, the radius of the base, and possibly the height of the cone.
2. **Ray-Cone Intersection Test**: Implement the algorithm to check whether a given ray intersects with the cone. This usually involves solving the quadratic equation derived from the intersection of the ray with the infinite surface of the cone.
3. **Calculate Intersection Point(s)**: If the ray intersects with the cone, calculate the intersection point(s). Depending on whether the ray intersects with the cone's side or its base, you might have one or two intersection points.
4. **Shading and Lighting Calculations**: Once you have the intersection point(s), you can calculate the shading and lighting effects at those points using techniques like Phong shading, ray-surface intersection tests with other objects in the scene, or global illumination methods.
5. **Iterate Over Rays**: Repeat the intersection tests for all rays cast from the camera into the scene. Combine the results to generate the final rendered image.

BELOW IS THE PSEUDO-CODE:

```
def ray_cone_intersection(ray, cone):
```

```
    # Compute coefficients of the quadratic equation for ray-cone intersection
```

```
    a = dot(ray.direction, ray.direction) - (1 + cone.radius ** 2) * dot(ray.direction, cone.axis) ** 2
```

```
    b = 2 * (dot(ray.origin - cone.apex, ray.direction) - (1 + cone.radius ** 2) * dot(ray.direction, cone.axis) * dot(ray.origin - cone.apex, cone.axis))
```

```
    c = dot(ray.origin - cone.apex, ray.origin - cone.apex) - (1 + cone.radius ** 2) * dot(ray.origin - cone.apex, cone.axis) ** 2
```

```
    # Solve the quadratic equation
```

```
    discriminant = b ** 2 - 4 * a * c
```

```
    if discriminant < 0:
```

```
        return None # No intersection
```

```
    else:
```

```
        t1 = (-b + sqrt(discriminant)) / (2 * a)
```

```
        t2 = (-b - sqrt(discriminant)) / (2 * a)
```

```
        intersection_points = []
```

```
        if t1 > 0:
```

```
            intersection_points.append(ray.origin + t1 * ray.direction)
```

```
        if t2 > 0:
```

```
            intersection_points.append(ray.origin + t2 * ray.direction)
```

```
        return intersection_points
```

BELOW IS THE CONE INTERSECTION SNIPPET FROM
PATHTRACER_MAIN_FS.FS:

```
//cone intersection ...

AnyHit ConeIntersection(Ray r, vec3 Position){
    //define cone parameters ...
    AnyHit Hc;
    float ConeHeight = 2.0;
    float radius = 1.0;

    float halfHeight = ConeHeight* 0.5;
    float tanTheta = radius / halfHeight;

    //ray parameter
    vec3 oc = Position - r.origin;
    //cout<<"|toc.x:|t"<<oc.x<<"|toc.y:|t"<<oc.y<<"|toc.z:|t"<<oc.z<<endl;
    float k = tanTheta * tanTheta + 1.0;

    //cout<<"|tK parameter:|t"<<k<<endl;

    vec3 oc2 = cross(oc , oc);
    //cout<<"|toc2.x:|t"<<oc2.x<<"|toc2.y:|t"<<oc2.y<<"|toc2.z:|t"<<oc2.z<<endl;
    float c = oc2.x + oc2.y - tanTheta * tanTheta * oc2.z;
    //cout<<"|tc:|t"<<c<<endl;
    float discriminant = k * r.direction.z * r.direction.z - c;
    //cout<<"|tdiscriminant:|t"<<discriminant<<endl;

    if (discriminant < 0.0) {
        Hc.t = -1.0;
        return Hc;
    }
    float sqrtDiscriminant = sqrt(discriminant);
    //cout<<"|tsqrtDiscriminant:|t"<<sqrtDiscriminant<<endl;
    float t1 = (-k * r.direction.z + sqrtDiscriminant) / (1.0 + k);
    float t2 = (-k * r.direction.z - sqrtDiscriminant) / (1.0 + k);
    //cout<<"|tt1-distance:|t"<<t1<<"|tt2-distance:|t"<<t2<<endl;
```

PYRAMID INTERSECTION ALGORITHM

pyramid intersection is defined to a process of determining whether a ray intersects with a pyramid-shaped object or volume in the scene being rendered, this can be similar to the cone intersection. This involves finding the point where the ray intersects with the pyramid's surfaces.

Here's a general outline of how you might implement pyramid intersection in a ray tracing algorithm:

1. **Define the Pyramid Geometry**: First, you need to represent the pyramid geometry in your scene. This typically involves defining the vertices of the base polygon and the apex of the pyramid.
2. **Ray-Pyramid Intersection Test**: Implement the algorithm to check whether a given ray intersects with the pyramid. This usually involves finding the intersection point between the ray and the planes defining the pyramid's sides.
3. **Calculate Intersection Point**: If the ray intersects with the pyramid, calculate the intersection point. This point will be used to determine the color and shading of the pixel in the final rendered image.
4. **Shading and Lighting Calculations**: Once you have the intersection point, you can calculate the shading and lighting effects at that point using techniques like Phong shading, ray-surface intersection tests with other objects in the scene, or global illumination methods.
5. **Iterate Over Rays**: Repeat the intersection tests for all rays cast from the camera into the scene. Combine the results to generate the final rendered image.

Below is pseudo-code for pyramid intersection:

```
def ray_pyramid_intersection(ray, pyramid):  
    # Perform intersection test with the planes defining the pyramid  
    for each plane in pyramid_planes:  
        intersection_point = ray_plane_intersection(ray, plane)  
        if intersection_point is not None and  
is_inside_pyramid(intersection_point, pyramid):  
            return intersection_point  
    return None  
  
def is_inside_pyramid(point, pyramid):  
    # Check if the intersection point lies within the pyramid's base  
and sides  
    # Implement this according to your pyramid representation  
    return point_inside_base(point, pyramid) and  
point_inside_sides(point, pyramid)
```

Below is the pyramid code snippet from pathtracer_main_fs.fs:

```
//Pyramid intersection
AnyHit intersectPyramid(Ray r, vec3 position){
    AnyHit Hc;
    Triangle Tri;
    vec3 oc = position - r.origin;
    float dotProduct;
    vec3 worldCoord;
    Triangle pyramidCoord[]={
        // FRONT FACE
        {{0.5f, -0.5f, -0.5f},{0.5f, -0.5, -0.5}, {0.0, 0.5, 0.0}},
        {{-0.5f, -0.5f, 0.5f},{0.5f, -0.5, 0.5}, {0.0, 0.5, 0.0}}, // TOP
        {{-0.5f, -0.5f, -0.5f},{-0.5f, -0.5f, 0.5f}, {0.0, 0.5, 0.0}}, // TOP
        // LEFT FACE
        {{0.5f, -0.5f, -0.5f},{0.5f, -0.5f, 0.5f}, {0.0f, 0.5f, 0.0f}},

        // BASE LEFT
        {{-0.5f, -0.5f, -0.5f},{0.5f, -0.5f, 0.5f},{-0.5f, -0.5f, 0.5f}},
        // BASE RIGHT
        {{0.5f, -0.5f, 0.5f},{-0.5f, -0.5f, -0.5f},{0.5f, -0.5f, -0.5f}}

    };

    int sizeP = pyramidCoord.length();
    for(int i = 0; i<sizeP; i++){

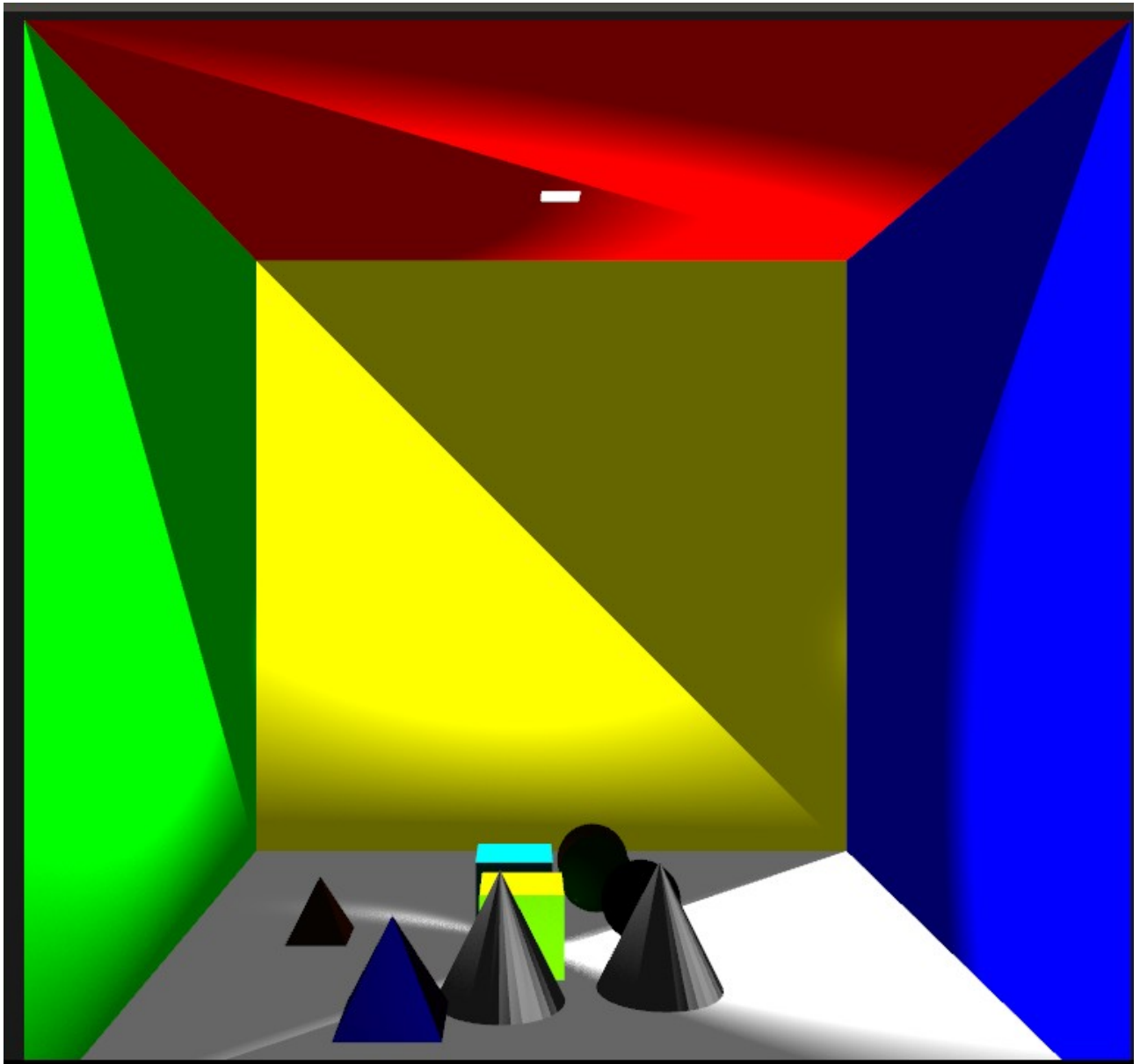
        dotProduct = dot(dot((pyramidCoord[i].V0.x * r.direction.x +pyramidCoord[i].V0.y * r.direction.y+pyramidCoord[i].V0.z *
r.direction.z), (pyramidCoord[i].V1.x * r.direction.x +pyramidCoord[i].V1.y * r.direction.y+pyramidCoord[i].V1.z *
r.direction.z)), (pyramidCoord[i].V2.x * r.direction.x +pyramidCoord[i].V2.y * r.direction.y+pyramidCoord[i].V2.z * r.direction.z));

        if(dotProduct == 0.0 ){
            continue;
        }
        float t = dot(dot((pyramidCoord[i].V0.x * oc.x +pyramidCoord[i].V0.y * oc.y+pyramidCoord[i].V0.z * oc.z),
(pyramidCoord[i].V1.x * oc.x +pyramidCoord[i].V1.y * oc.y+pyramidCoord[i].V1.z * oc.z)), (pyramidCoord[i].V2.x * oc.x
```

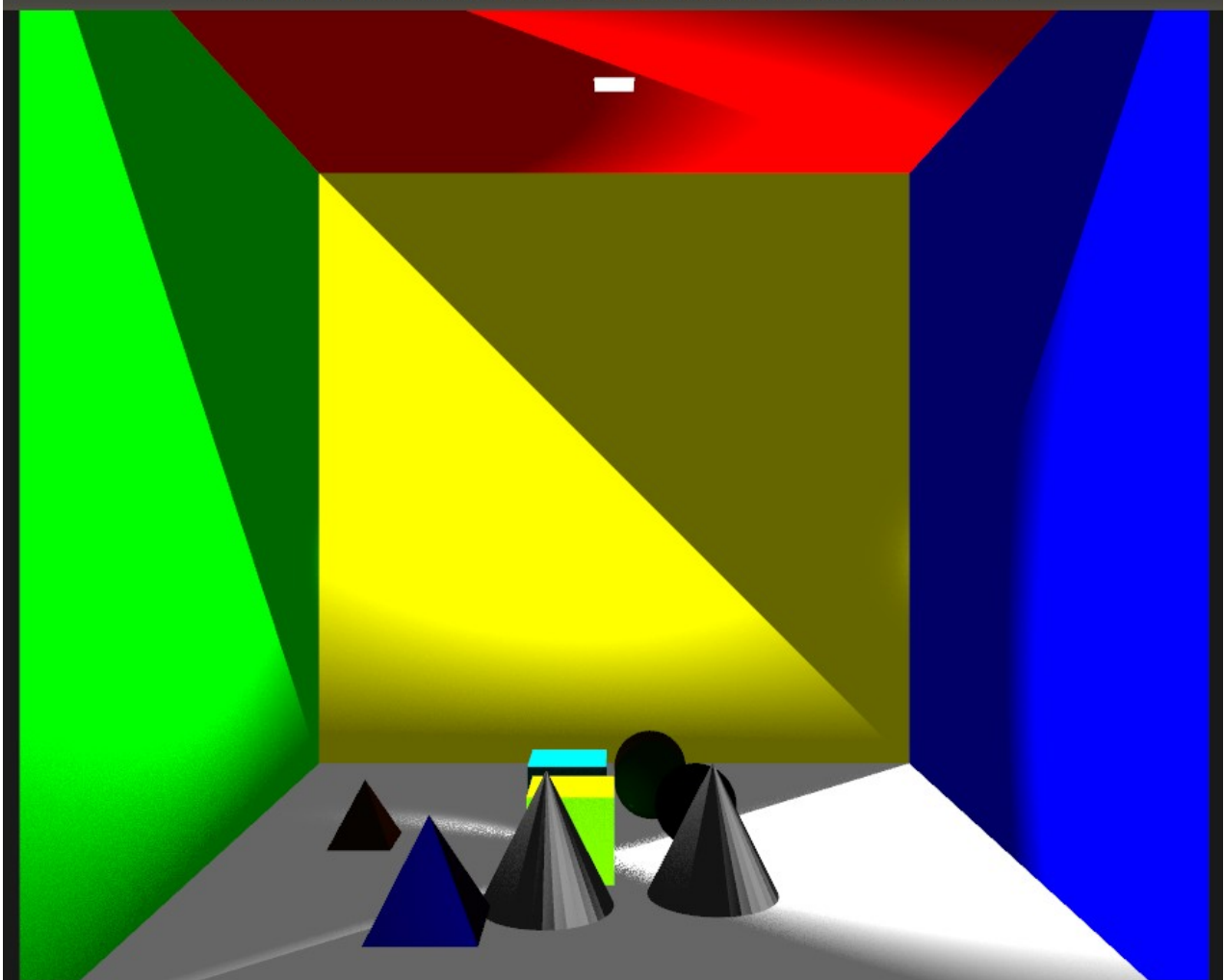
C++ ▾ Tab Width: 8 ▾ Ln 218, Col 48

RESULT OF PATH TRACING TEST:

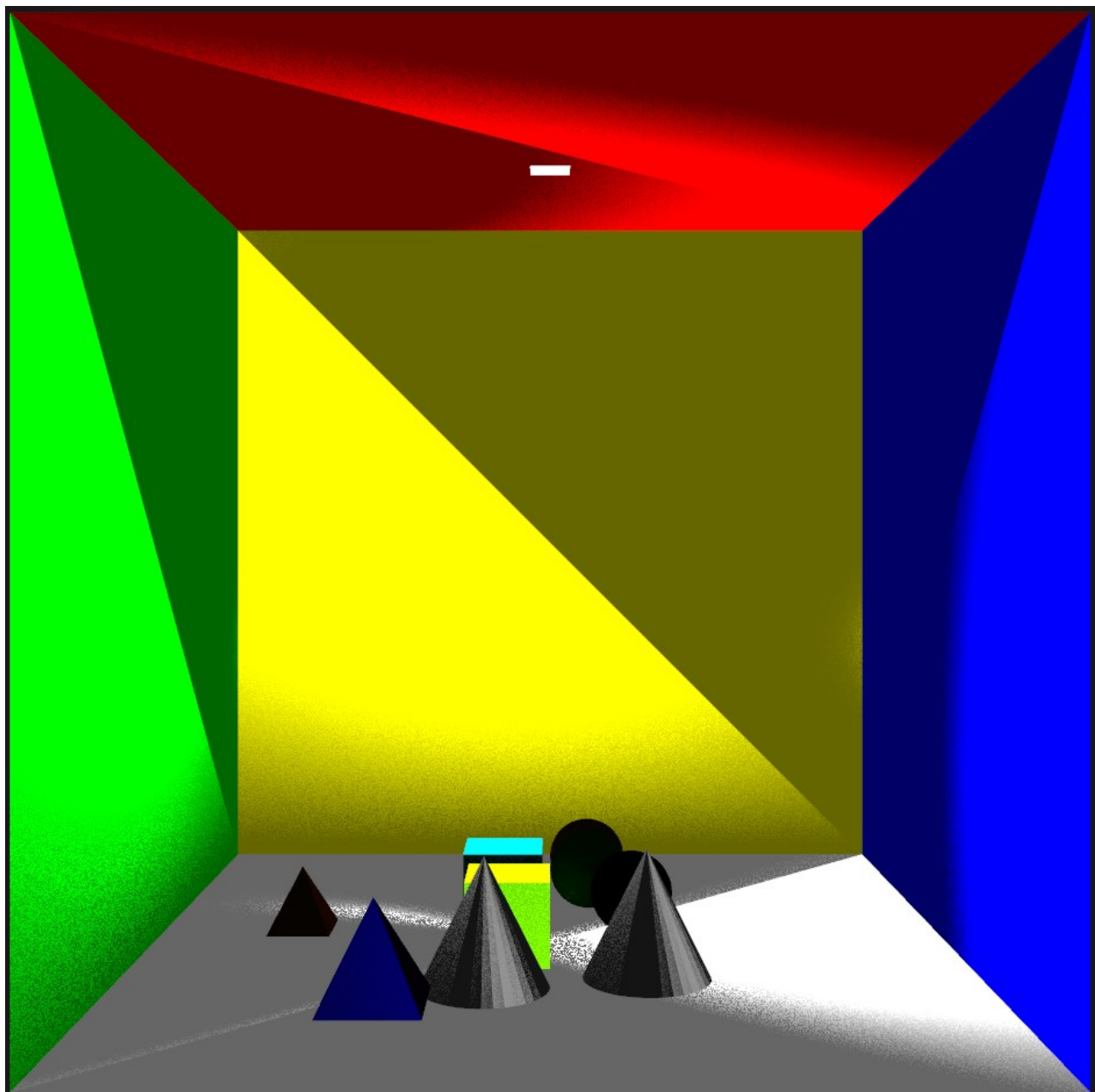
1000 SAMPLES PER PIXEL, THE ENVIRONMENT IS WELL ILLUMINATED NO NOISE IN THE SCENE.



100 SAMPLES PER PIXEL, THE ENVIRONMENT IS ILLUMINATED BUT HAS LITTLE NOISE IN THE SCENE, ESPECIALLY IN THE OBJECT AND FLOOR OF THE SCENE.



RESULT of 10 SAMPLES PER PIXEL, THE ENVIRONMENT IS ILLUMINATED BUT HAS LOT OF NOISE IN THE SCENE, ESPECIALLY IN THE OBJECT AND FLOOR AND THE CEILING.



CONCLUSION

The project is aimed at implementing the path tracing using different primitive objects and using some environment maps to simulate some of the light effects, shadow, reflections and refractions effects.

The hope in the future is to perform the path tracing in a more trivial environment, like a gaming or animation environment.

Reference list

Boksansky, J. (2021). *Crash Course in BRDF Implementation*. [online] Available at: <https://boksajak.github.io/files/CrashCourseBRDF.pdf> [Accessed 7 Sep. 2023].

khronos (2017). *Vertex Shader - OpenGL Wiki*. [online] www.khronos.org. Available at: https://www.khronos.org/opengl/wiki/Vertex_Shader.

Khronos Group (2020). *Fragment Shader - OpenGL Wiki*. [online] www.khronos.org. Available at: https://www.khronos.org/opengl/wiki/Fragment_Shader.

Leopold (Mangostaniko) , N. (2023). *Ray tracing (graphics)*. [online] Wikipedia. Available at: [https://en.wikipedia.org/wiki/Ray_tracing_\(graphics\)#/media/File:Ray_Tracing_Illustration_First_Bounce.png](https://en.wikipedia.org/wiki/Ray_tracing_(graphics)#/media/File:Ray_Tracing_Illustration_First_Bounce.png) [Accessed 7 Sep. 2023].

NVIDIA DEVELOPER (2018). *Ray Tracing*. [online] NVIDIA Developer. Available at: <https://developer.nvidia.com/discover/ray-tracing#:~:text=Ray%20tracing%20is%20a%20rendering>.

Pellacini, F. (n.d.). *monte carlo integration*. [online] <https://pellacini.di.uniroma1.it>. Available at: https://pellacini.di.uniroma1.it/teaching/graphics16/lectures/17_montecarlo.pdf.

Shirley, P.S., David Black, T. and Hollasch, S. (2023). *Ray Tracing: The Rest of Your Life*. [online] Github.io. Available at: <https://raytracing.github.io/books/RayTracingTheRestOfYourLife.html#overview> [Accessed 7 Sep. 2023].

WIKIPEDIA (2023a). *Path tracing*. [online] Wikipedia. Available at: https://en.wikipedia.org/wiki/Path_tracing#:~:text=Path%20tracing%20is%20a%20computer [Accessed 7 Sep. 2023].

WIKIPEDIA (2023b). *Path tracing*. [online] Wikipedia. Available at: https://en.wikipedia.org/wiki/Path_tracing#/media/File:Path_tracing_001.png [Accessed 7 Sep. 2023].

Ray Tracing Basics, CSE 681 Autumn 11 Han-Wei Shen, Available at: https://web.cse.ohio-state.edu/~shen.94/681/Site/Slides_files/basic_algo.pdf [Accessed 2 Mar. 2024]

