

**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.

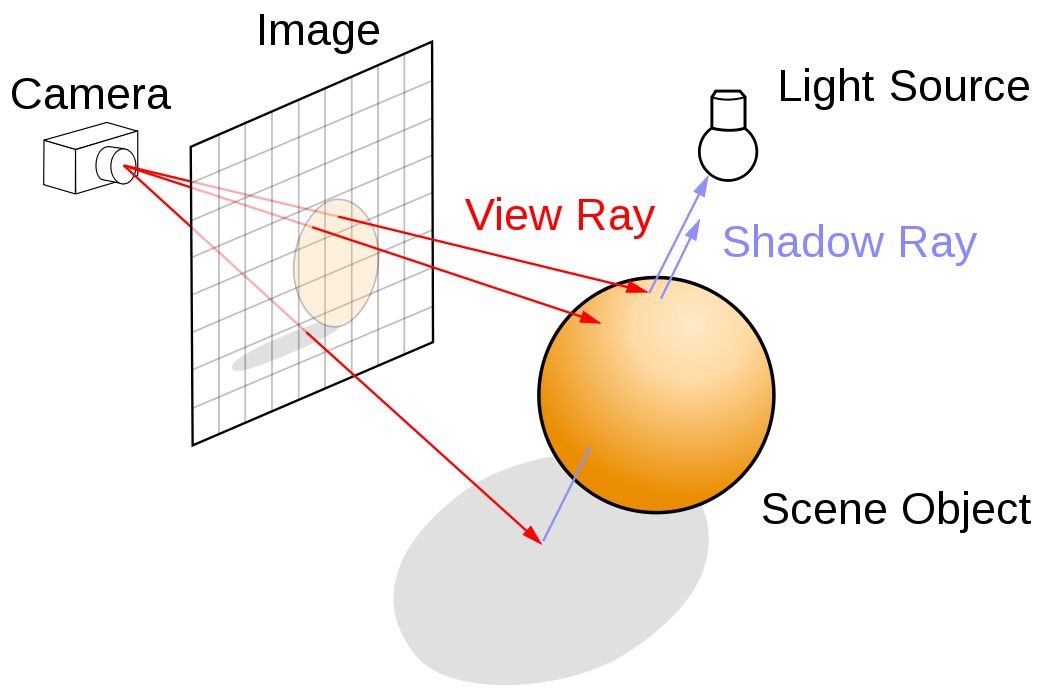
**Principle of Direction:** The Principle of Direction dictates that the scattered illumination emanating from surfaces must follow a specific trajectory. This trajectory is determined by a function of the incoming direction of the incident illumination and the outgoing direction, which is sampled using 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.

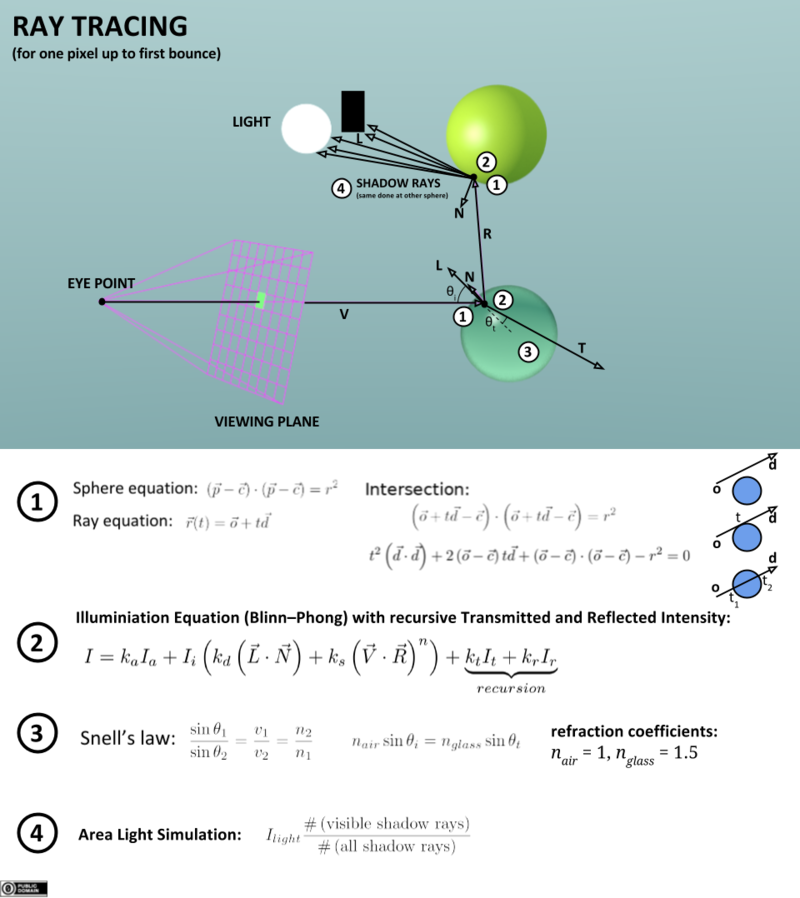
**Ray Tracing:**



**(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 algorith****m**

**(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, ω ) = L +ʃf(P, ω i , ω o )L (P, ω )(ω ∙ n )dω**

**brdfLambertian = (diffuseReflectance / π) \* 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.).



**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 is code is used to generate large samples of rays into the scene and also use these generated rays to compute the ray properties.

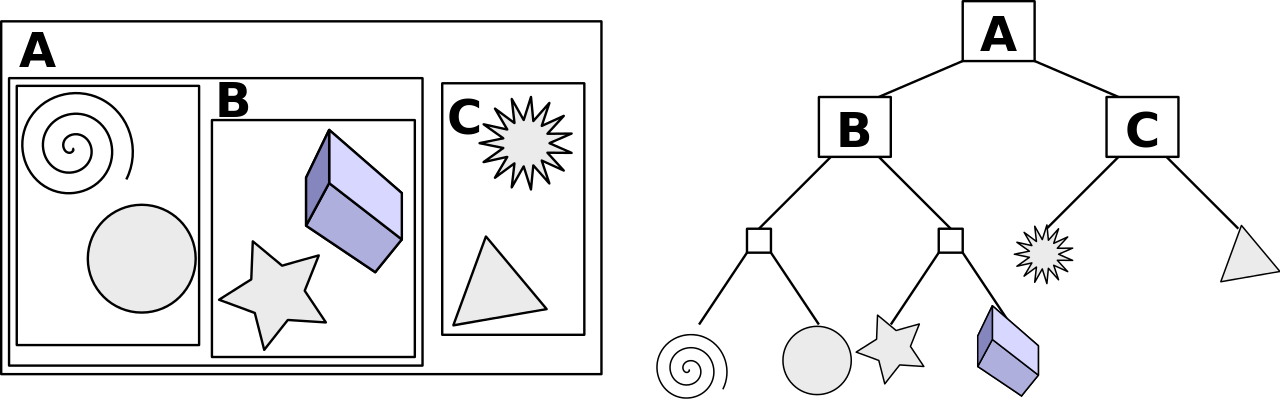


**Bounding Volume Hierarchy (BVH):**

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.

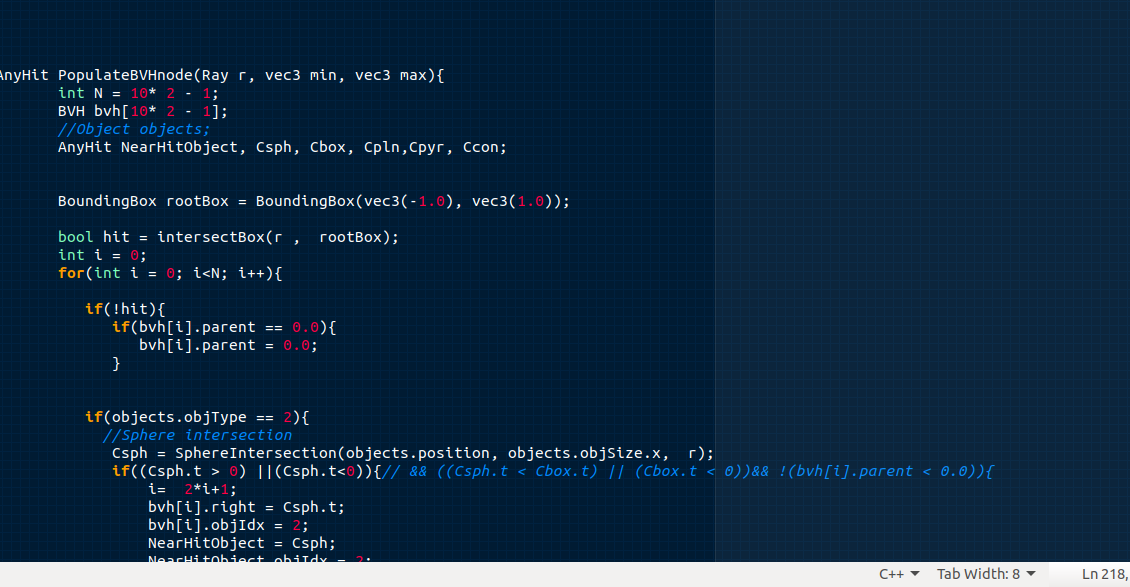
**(NVIDIA DEVELOPER (2018))**

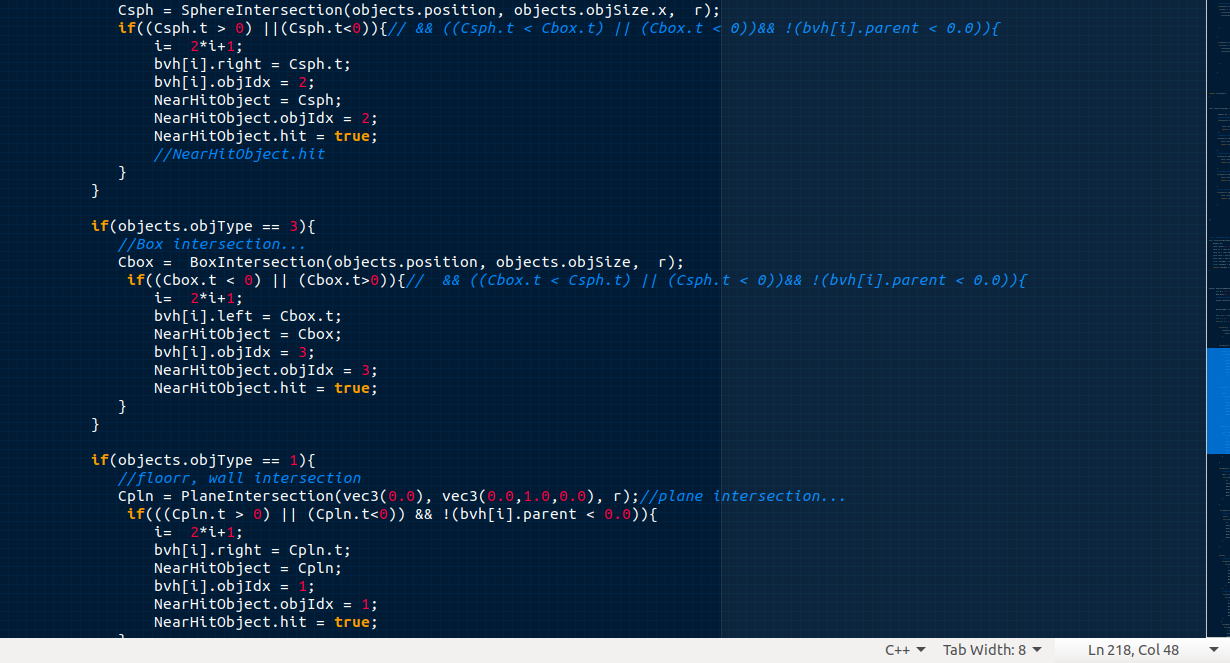


**(By Schreiberx - Own work, CC BY-SA 3.0, https://commons.wikimedia.org/w/index.php?curid=17853864)**

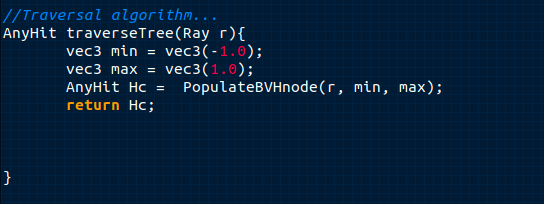
**BVH code snippet from the pathtracer\_fs.fs:**

**the BVH code contains all the intersection algorithm :**

****

****

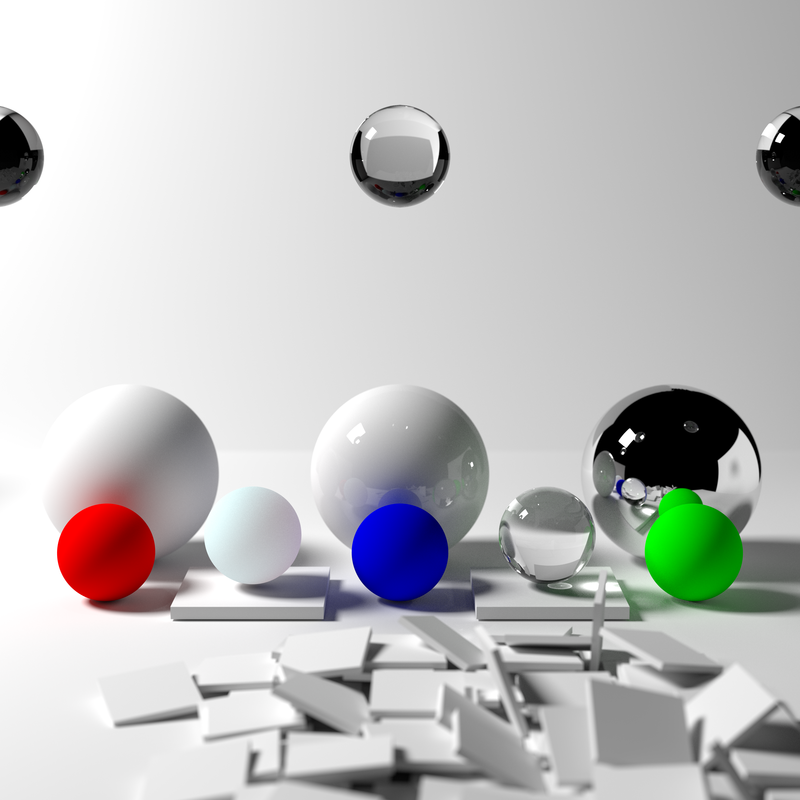
**Below is the traversal snippet:**

****

**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**](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);

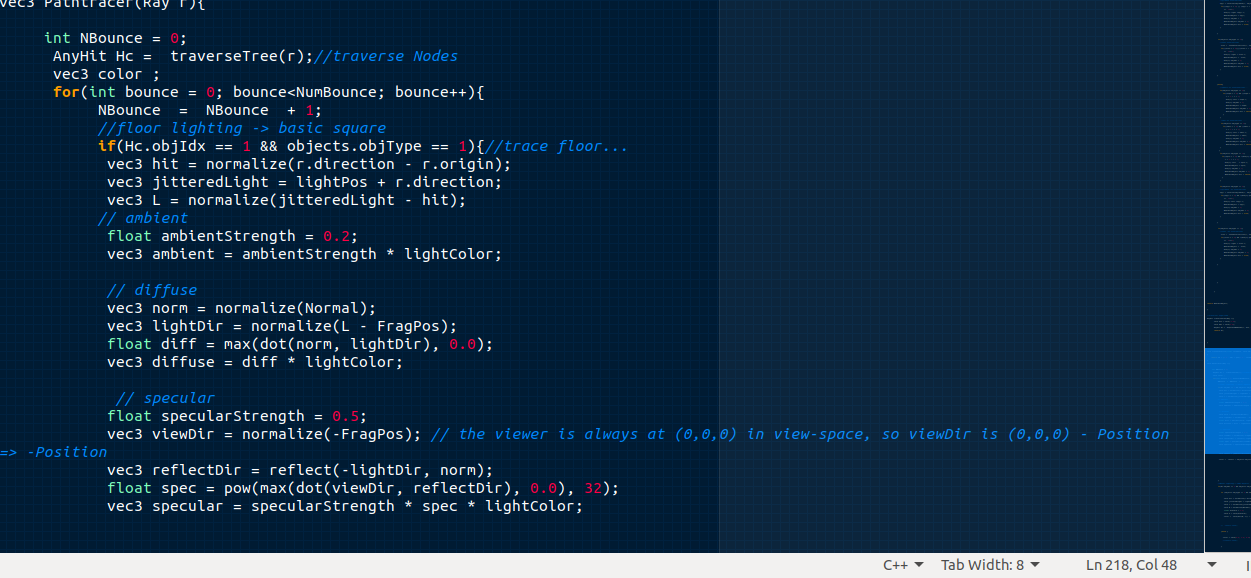
*// Apply the Rendering Equation here.*

**return** emittance + (BRDF \* incoming \* cos\_theta / p);

}

(WIKIPEDIA (2023))

**PATHTRACING CODE SNIPPET FROM PATHTRACER\_MAIN\_FS.FS**

****

**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;

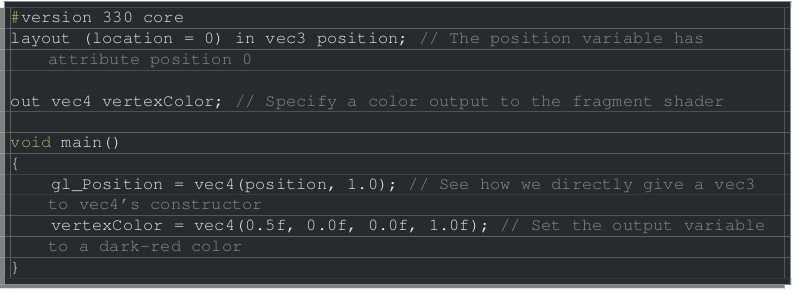
out type out\_variable\_name;

**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:**

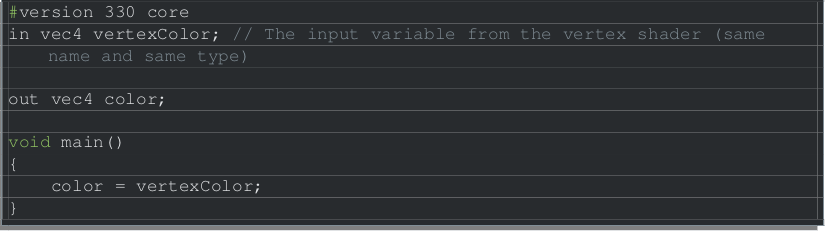
****

**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.**

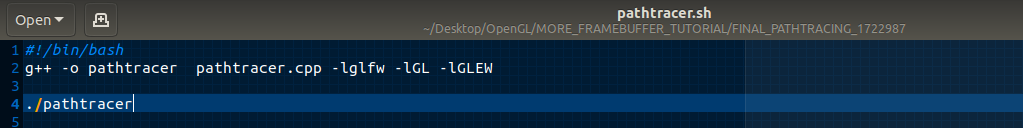
**(Khronos Group (2020))**

**structure of the Fragment shader:**



**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).**



**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.

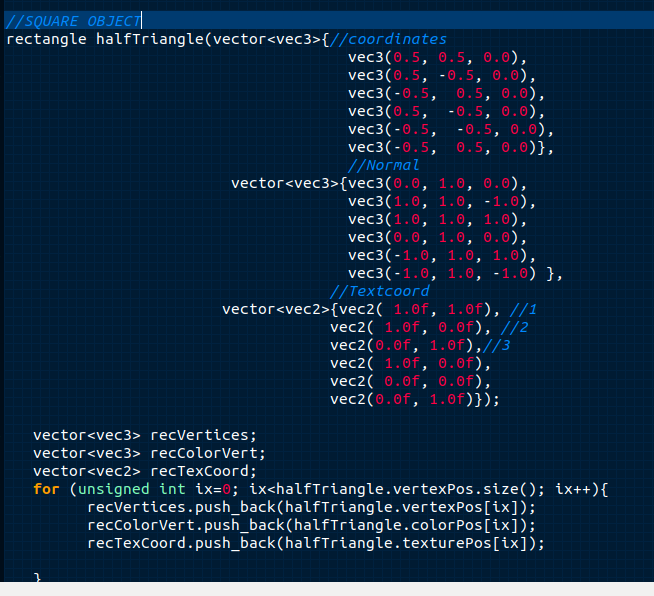
Floor, the walls and roof are scaled by 20.

**SQUARE PRIMITIVES (1):**

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

**First code is the code reponsible 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.)**

****

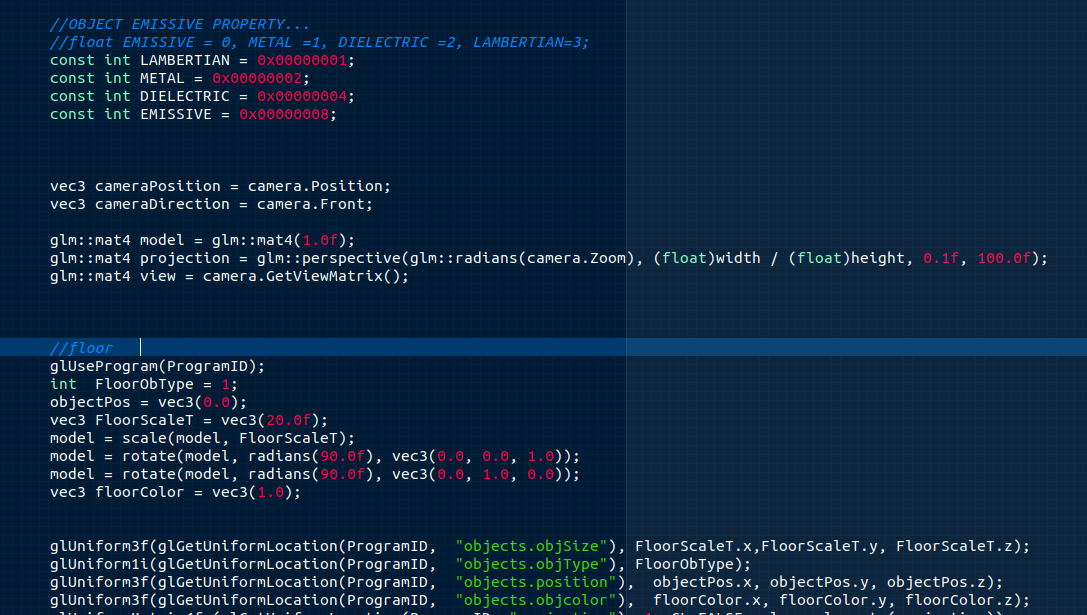
**Second code is the that helps to render these objects through the shader program and we have all the properties need to render, light, colors and also the model, camera view etc to help position these objects in place.**

**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.



**Wall Code (Similar to the floor code)**

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



**Light Object:**

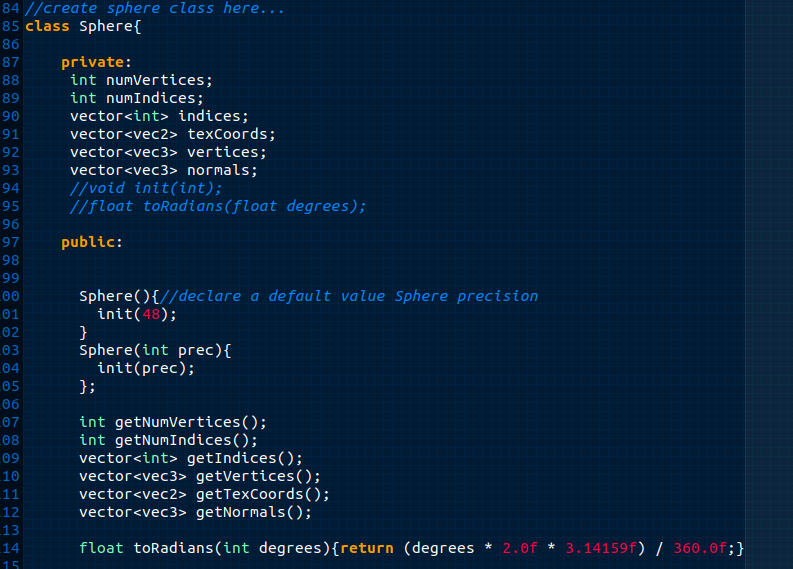
The light object is similar to the floor but the different in the size and also in positioning and the color of the object.



**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:

****

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



**CUBE PRIMITIVES(3).**

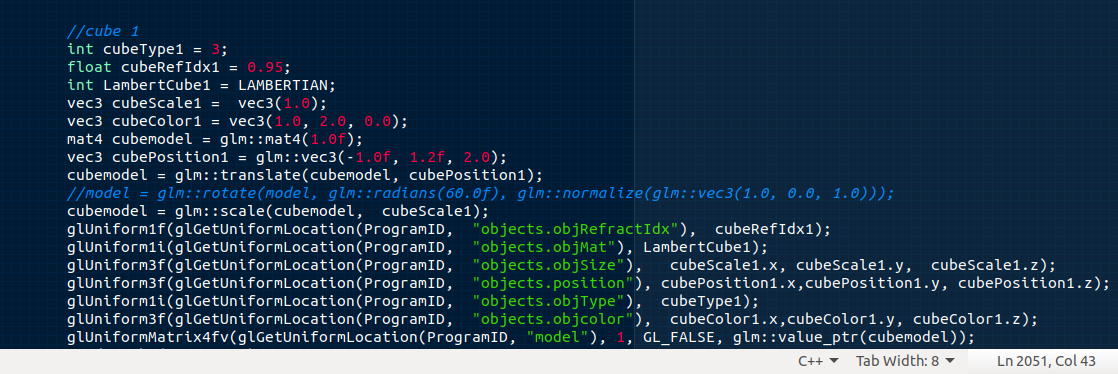
**Cube Vertex**

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 SHADER PROGRAM:**

****

**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.**

****

**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.**

**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.

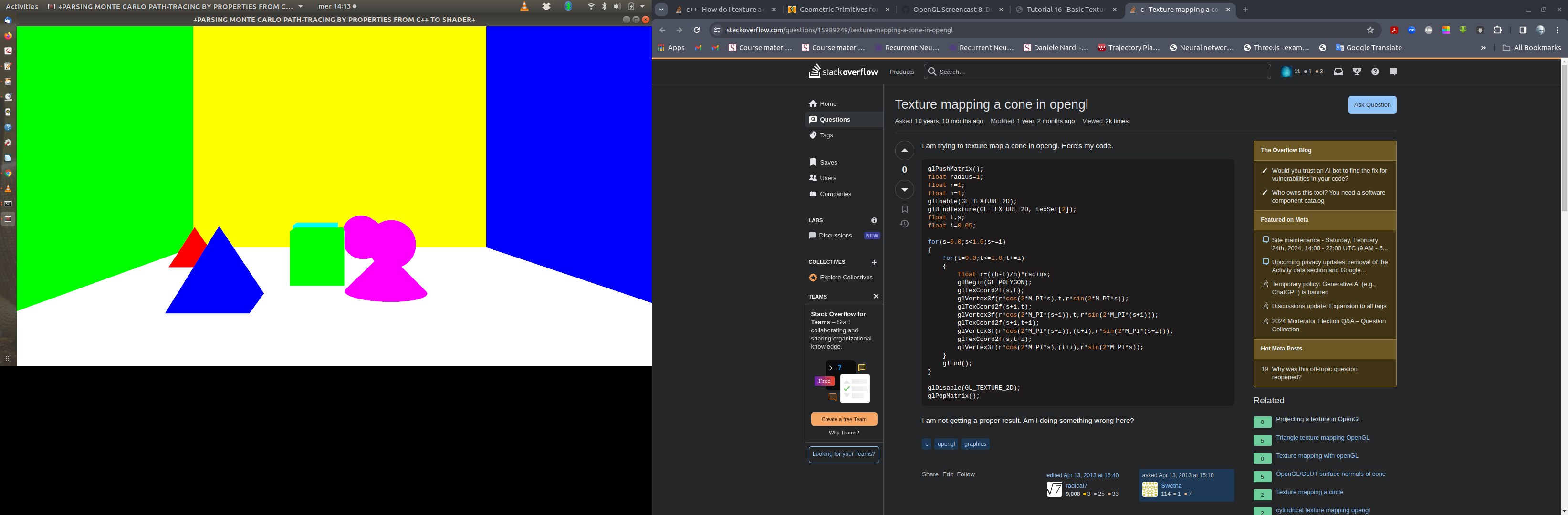


**Cone Shader:**

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

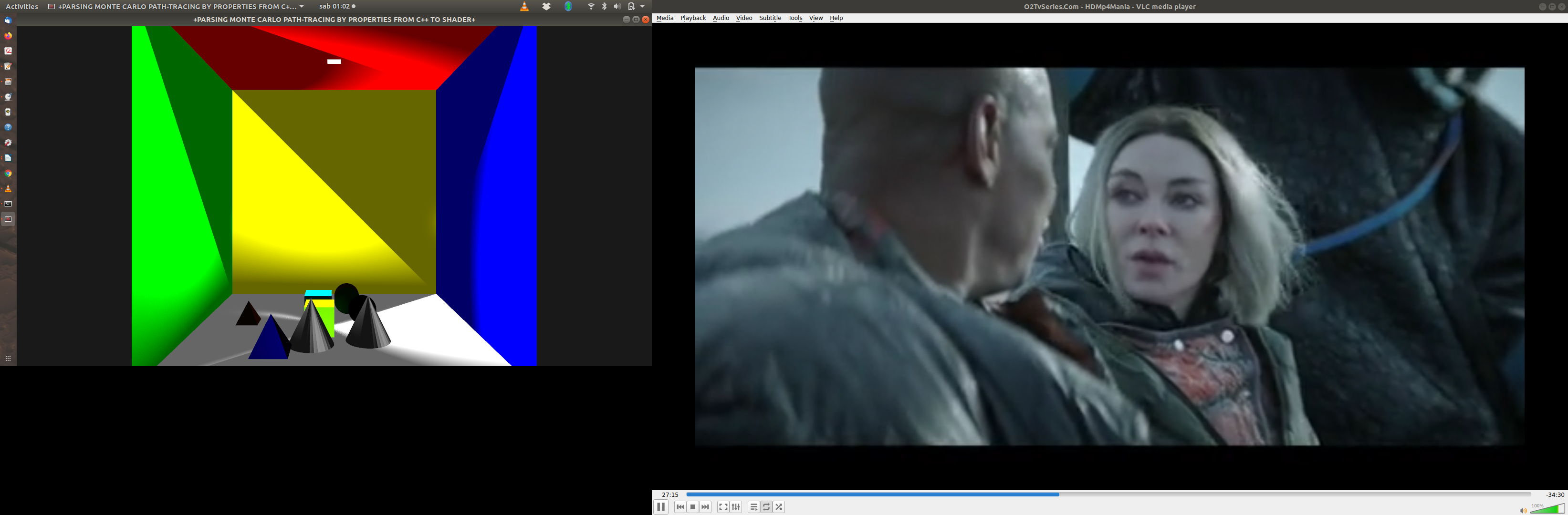


**SCENE IMAGE 1.(Scene with out intersection)**

****

**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:**

**P(t)=O+tD**,where **O** is the origin of the ray, **D** is the direction vector of the ray, and t is a parameter, and a plane defined by its normal vector **N** and a point **P0​** on the plane, the intersection point **P(t)** satisfies the equation: **(P(t)−P0​)**

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**



**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****)****⋅****(****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).(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)**

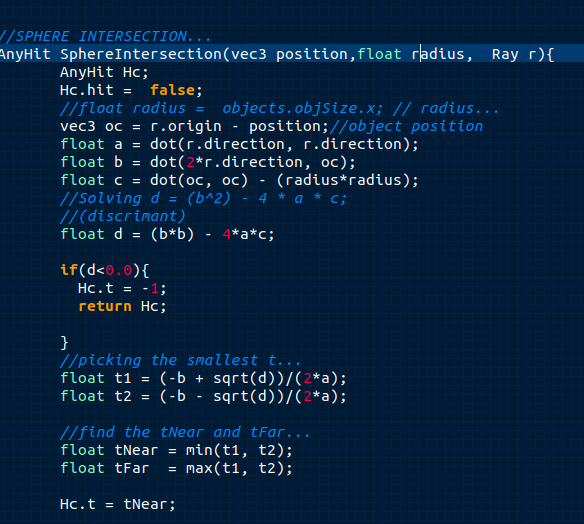
**computing solution: t +/- = b(+/-)d / 2a.**

**b2 – 4ac < 0 ⇒ No intersection**

**b2 – 4ac > 0 ⇒ Two solutions (enter and exit)**

**b2 – 4ac = 0 ⇒ One solution (ray grazes sphere)**

**BELOW IS THE SPHERE INTERSECTION SNIPPET FROM PATHTRACER\_MAIN\_FS.FS:**

****

**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 = x1, x = x2**

**y = y1, y = y2**

**z = z1, z = z2**

**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:**

t**xmin =(x1 - ex )/Dx //assume Dx >0**

**txmax =(x2 - ex )/Dx**

**tymin = (y1 - ey )/Dy**

**tymax = (y2 - ey )/Dy //assume Dx >0**

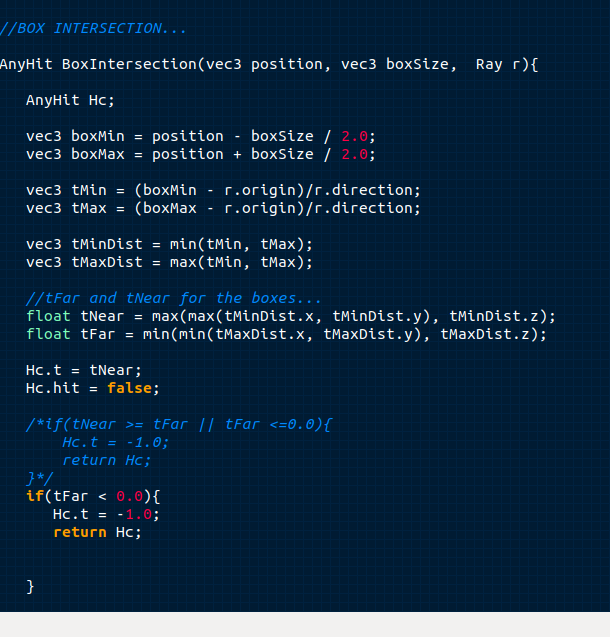
**if (txmin > tymax) or (tymin > txmax)**

**return false**

**else**

**return true**

**BOX-INTERSECTION ALGORITHM, BELOW IS THE CODE SNIPPET**

****

**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:**

****

**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**

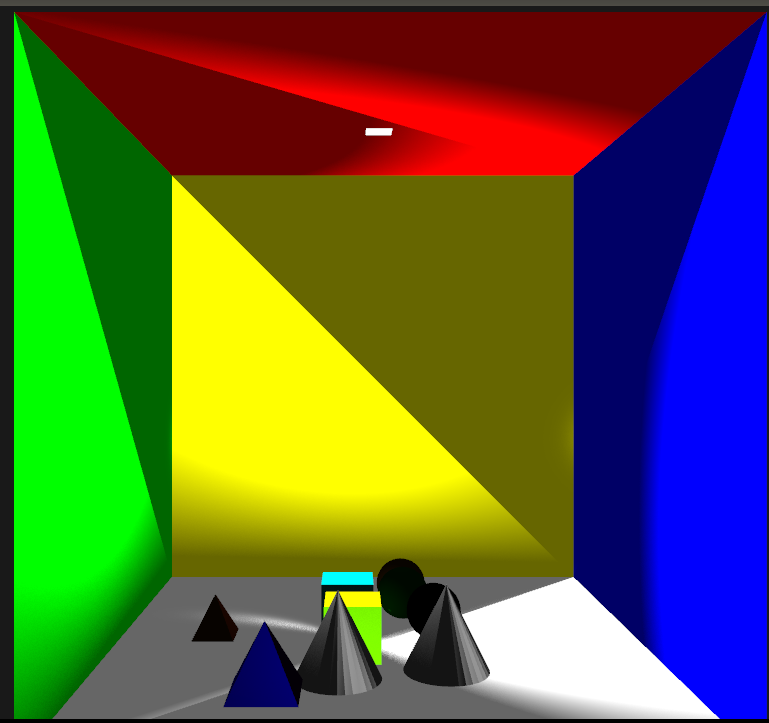
**# Implement this according to your pyramid representation**

**return point\_inside\_base(point, pyramid) and point\_inside\_sides(point, pyramid)**

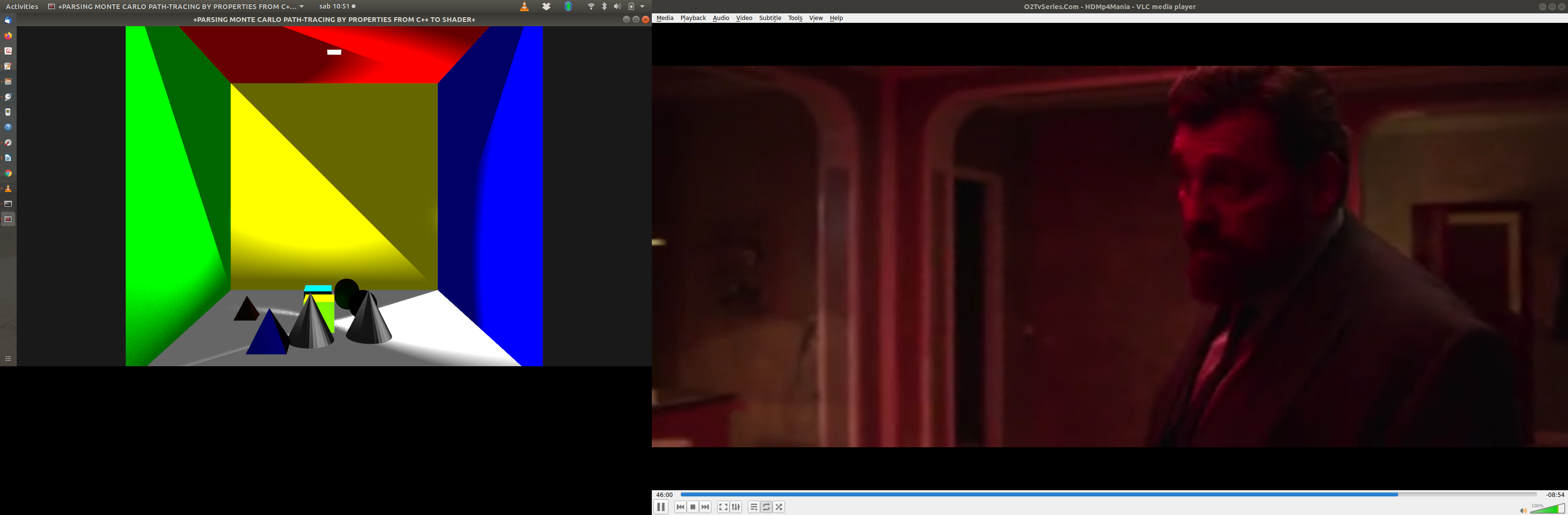
**Below is the pyramid code snippet from pathtracer\_main\_fs.fs:**

****

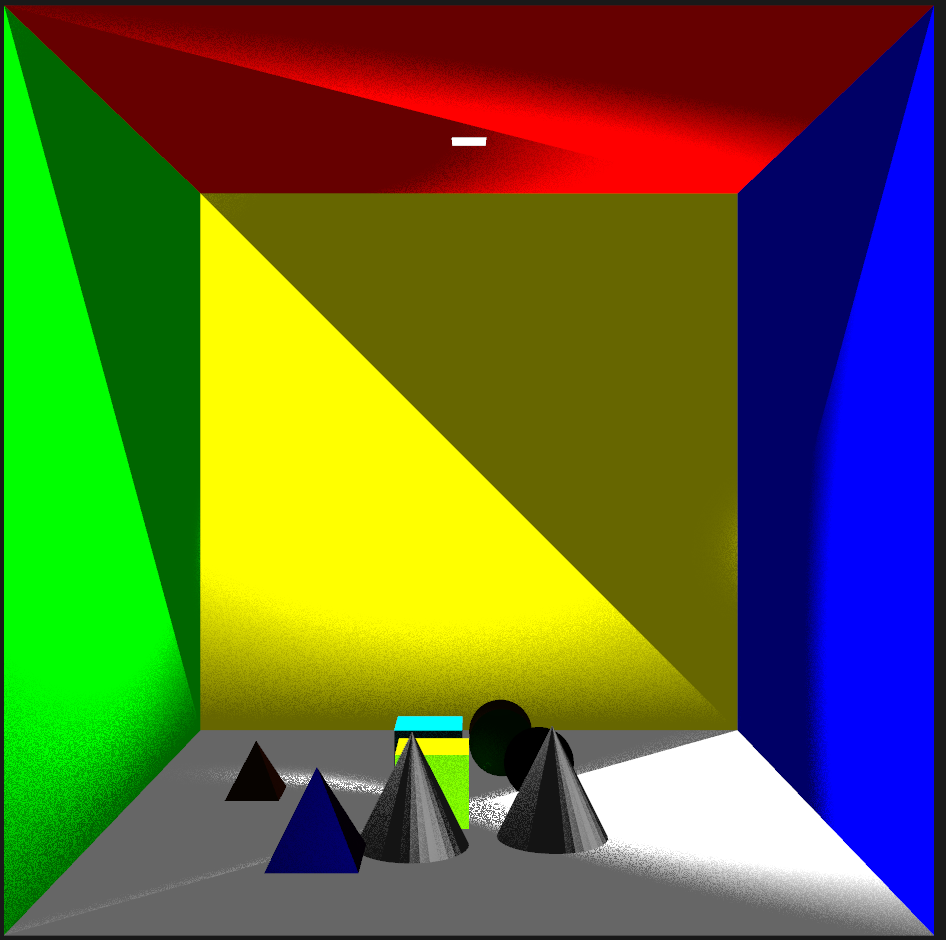
**RESULT OF PATH TRACING TEST:**

**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 [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]

‌