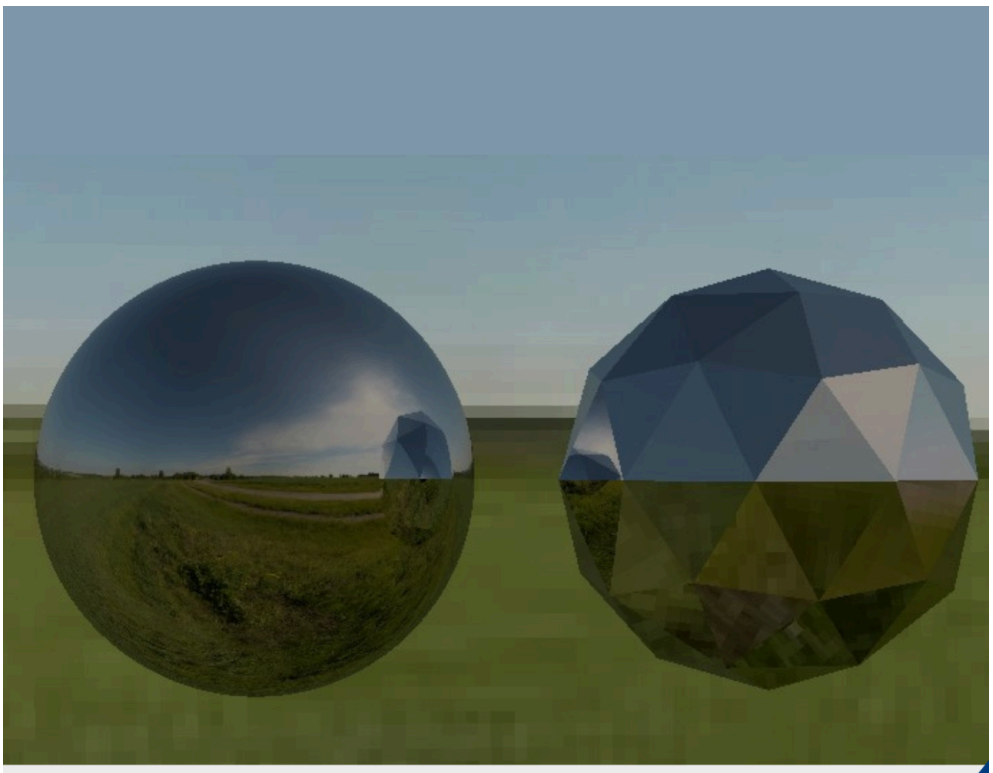


REAL-TIME CPU RAY TRACING

PABLO TRIK MARÍN
Tutor: Elisabet Quintana,
Centre: Escola Pia d'Igualada,
Year: 2024/25, Class: 1st A.

Abstract

English:

The project, entitled “Real-time CPU ray tracing”, is aimed at, starting from scratch, giving anyone a deep understanding of the ray tracing (the 3D rendering technique); and, additionally, the capability of building a simple path tracing application.

The survey is divided into three main parts. It starts giving essential computer-science concepts in order to give the reader a fair understanding of computers and their workflow, as well as explaining the ways of communication between humans and machines to achieve results… Secondly, it deals with the nature of ray tracing itself: states that it is a technique that aims to get close to reality, analyse reality, gives an understanding of how a computer can mirror reality, and explains the maths behind tracing. Regarding the third part, a small path tracing application was developed, and the research work explains it completely file by file.

As a result, the most important information, concepts, and features of any ray tracer in the world were gathered and compiled one by one in the project. The program was developed to be capable of Rendering a three dimensional scene saved in a .json file, as well as .obj models and switching between them on runtime; Running at more than 30 frames per second on low resolutions; Changing its point of view on runtime modifyingthe field of view; and stating a number of bounces for the ray…

To sum up, real-time CPU ray tracing provides the reader with the necessary information to understand ray tracing from a computer-science perspective and it develops an application which runs ray tracing.

Català:

El treball es titula: "Real-time CPU ray tracing", que, en català, significa: traçat de raigs a temps real a la unitat central de processament de l'ordinador. El projecte busca explicar, des de zero, com funciona el traçat de raigs (la tècnica de renderitzat d'escenaris tridimensionals i virtuals a imatges en dues dimensions) per a qualsevol persona interessada sense importar el seu nivell d'experiència. Paral·lelament, pretén explicar com funciona un exemple real d'aplicació de renderitzat per traçat de raigs a temps real que ha sigut creada pel projecte.

El treball conté tres parts principals: Primer, una recopilació de coneixements essencial perquè es pugui entendre el treball independentment del seu coneixement previ de la matèria. En la segona part, hi ha una explicació exhaustiva del funcionament i les matemàtiques darrere del traçat de raigs, que no aprofundeix en els aspectes físics de la llum ni dels materials. La part final conté una explicació sobre l'aplicació que va ser programada durant el desenvolupament del treball.

El projecte ha sigut realitzat amb el següent flux de treball: Primer, definir les funcionalitats que ha de tenir el producte final (l'aplicació); Segon, ordenar els objectius establerts; Tercer, recopilar a la memòria informació sobre cada punt utilitzant els llibres que apareixen a la bibliografia, i altres projectes de codi obert (que tothom pot veure distribuir i contribuir) com el meu; Quart i últim, implementar les característiques a l'aplicació.

Per finalitzar: Aquest treball compleix els aspectes mencionats anteriorment. Té una explicació sobre la tecnologia del traçat de raigs i un exemple pràctic, el que el converteix en una bona guia per entendre la matèria i posar-la en pràctica.

Contents

Introduction

Real-time CPU ray tracing is a project which intends to synthesise, from scratch, the maths behind ray tracing (which is a technique employed to render, using computers, 2D images out of virtual 3D by calculating how light would interact with the objects at the scene casting rays from a virtual camera) for the untrained reader. The project is not just intended to get deeply on physical aspects of light rather but to simplify them into simple mathematical expressions. For example: This project does not state whether light is a particle or a wave, or the way that field of particles or waves interact with materials; It just presents the way light seems to behave from a human perspective and imitates it using linear equations (rays), and applying maths to colours. Apart from that, the survey indeed contains an explanation of a computer application which runs ray tracing in real-time on the core processing unit (CPU), and was created by me during the development of the research, in order to give a better understanding to the reader.

The knowledge shared throughout this project was accumulated over several years of self-study during high school, where I built applications independently, read the books listed in the bibliography, and watched instructional YouTube videos. This knowledge was then organised, synthesised, and compiled into the final document. Most of the work was completed during the summer, with progress tracked using GitHub (a platform and version control system that helps developers manage projects safely). I followed a structured schedule, dedicating at least four hours per day to the project. In summary, the development process followed these steps: first, defining the final goal; second, breaking it down into smaller, manageable tasks; and finally, researching each task individually to gain the understanding needed to complete it.

The contents of this project are divided into three main parts which achieve the objectives described before:

The first part introduces essential computer science concepts to give readers a foundational understanding of how computers work and how humans interact with them to performspecific tasks. It explains how computers execute and store instructions using electrical signals, and how software enables users to communicate with the hardware. The distinction between high-level and low-level programming languages is covered, with examples demonstrating why C++ (a low-level language) is ideal for maximising the potential of ray tracing. Additionally, the concept of shaders (programs that run on the graphic processing unit (GPU) and execute single operations thousands of times in parallel) is introduced, as it is crucial for understanding later topics.

The second part gathers the mathematical and computational principles of ray tracing (as opposed to the physical ones). Although there is an advanced form of ray tracing that incorporates physics (explained in the book Physically Based Rendering, which is listed in the bibliography), this is beyond the scope of this project. Instead, this section focuses on how light interacts with objects to produce colour, supported by real-life images I took to illustrate the concepts. It also explains how cameras work (with a brief mention of analog cameras) and how the relationship between light and a camera can be described mathematically using linear equations and rays. This part includes a workflow diagram I created, along with images to help visualise the process. It further explains how different 3D shapes, such as triangles and spheres, can be mathematically defined and detected, and how this leads to colour rendering. The section embracesrelated technologies like texture mapping (which aligns a flat image's surface coordinates with a 3D object's surface), normal mapping (which is the same as texture mapping but instead of relating colours it relates normal directions for the surface), and anti-aliasing (an algorithm that smooths pixel edges by averaging colours with neighbouring pixels).

When it comes to the third part, a relatively small path tracing (the branch of ray tracing explained along all the project) application was developed and each is file is analysed thoroughly. The section begins by discussing the architecture of the program, explaining the decisions behind using Linux, minimal external libraries, and excluding the GPU and 3D graphics engines. It then dives into the main file, where the core loop of the application is defined and runs repeatedly. The rest of the section carefully explains how each file relates to the main loop's hierarchy and function. This part demonstrates how things are displayed on a window on screen using SFML (a C++ open source library), how objects and scenes are stored and readed, as well as all the ray tracing related features such as the camera, rays and *shader*. It has some charts which visually support the content.

Enjoy your reading!!

1. Important computer-science concepts for ray tracing

1.1. Hardware

Computers are powerful machines that enable humans to perform vast quantities of mathematical operations with remarkable speed. This capability is invaluable in the context of analysing extensive data structures or predicting complex physical phenomena, as intended in this project.

At the core of computer functionality is a sophisticated interplay between electrical circuits and logical operations. Fundamentally, computers operate using binary logic, which is represented by two states: 1 (on) and 0 (off). This binary system forms the foundation of all computer operations, allowing for the efficient processing of large amounts of data and the simultaneous execution of millions of calculations.

Electricity flows through the circuits of a computer, passing through logic gates that make decisions based on the presence or absence of electrical signals. These logic gates, such as AND, OR, and NOT gates, determine whether signals should continue along a particular path or be halted. The decisions made by these gates result in signals being interpreted as either 1 (on) or 0 (off). These digital signals are the fundamental building blocks of all computations and data storage within a computer.

The circuits are organised into different units to achieve specific objectives: processing units and memory units. Processing units utilise physical resistances to create logical gates and perform calculations, while memory units are simple circuits capable of storing large arrays of bits.
In a computer, there are two primary types of processing units:

1. **The Central Processing Unit (CPU)**: The CPU typically contains between 4 and 64 cores, with 8 cores being the most common in contemporary systems. These cores are individual processing units designed to handle separate programs independently. The amount of tasks they can handle in a second is measured in GHz, there are processors between 2.5GHz and 6GHz, but the most common is 4GHz, which means four million tasks per second. When a computer runs out of available cores without fully utilising the CPU's capacity, these cores can be subdivided into virtual threads to perform additional tasks using asynchrony. The CPU excels at general-purpose processing and is essential for executing a wide range of tasks.

2. **The Graphics Processing Unit (GPU)**: The GPU comprises thousands of cores that, although individually less powerful than CPU cores, are highly efficient at performing the simplest mathematical operations. The GPU is specifically designed for tasks that require parallel processing, such as rendering graphics and performing complex calculations in scientific simulations. Unlike the CPU, which is optimised for a variety of tasks, the GPU is tailored to execute many small, independent, programs simultaneously, making it ideal for tasks that involve large-scale data manipulation and repetitive calculations.

Memory units in a computer serve the crucial function of data storage. These units can be broadly categorised into primary and secondary memory. Primary memory, or RAM (Random Access Memory), provides fast, temporary storage that the CPU[1] can access quickly. Secondary memory, such as hard drives and SSDs (Solid State Drives), offers larger, persistent storage capacity but at slower access speeds.

[1] GPUs have their own random access memory called VRAM. When the GPU is inside a CPU then the RAM is virtually divided into RAM and VRAM. For example: 16GB of RAM could lead to 12 to CPU and 4 to GPU. Usually GPUs have that extra VRAM integrated.

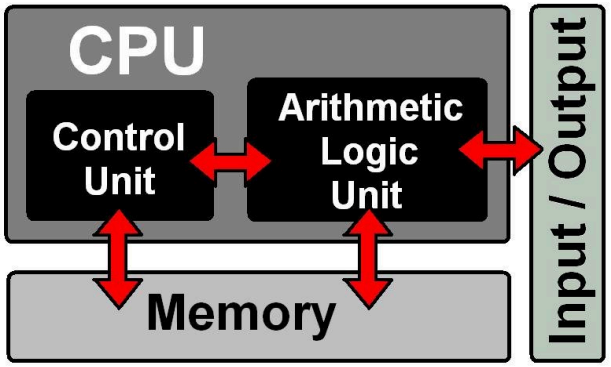


Fig. 1: Simple infographic of the logic from a processing unit taken from Pablo García-Risueño at the ResearchGate website.

1.2. Software

1.2.1. Languages

To take profit from computers, programmers develop applications. These programs manage the circuits and act as intermediaries between humans and machines.

While computers require instructions in binary format, writing millions of numbers to perform even a simple task is impractical for humans. Consequently, programming languages were developed. Each programming language has its own compiler or interpreter, which translates human-readable instructions into machine code, or bits.

Programming languages serve different purposes and operate at varying levels of abstraction. Languages closer to the computer's hardware, known as low-level languages, are highly specific and are typically used for developing operating systems and tasks requiring precise performance. Examples include Rust and C. In contrast, high-level languages are further removed from the hardware and are used

for simpler tasks, such as fetching data from databases or updating software packages. Examples include Python, JavaScript, and ShellScript.

To illustrate the differences between programming languages, consider the following example of a web server. This server hosts an “/upload” endpoint at 127.0.0.1:3000 using Node.js with the Express framework:

```
 1 import express from "express";
 2 import bodyParser from "body-parser";
 3
 4 let app = express();
 5 const port = 3000;
 6
 7 app.use(bodyParser.urlencoded({ extended: true }));
 8
 9 app.post('/upload', (req, res) => {
10    const fileContent = req.body.file;
11    console.log('File content received:');
12    console.log(fileContent);
13
14    res.send('File uploaded successfully');
15 });
16
17 app.listen(port, () => {
18    console.log(`Server running on http://localhost:${port}`);
19 });
```

Code 1: Example of JS[2] server

Then, these are two examples of how an application can use this port to upload a file in two different languages made for two different purposes:

```
1 import requests
2
3 def upload_file(file_path, url):
4    with open(file_path, 'r') as file:
5        file_content: str = file.read()
```

[2] Scripting language for the browser.

```
    payload: dict = {'file': file_content}
    requests.post(url, data=payload)

file_path: str = './file.txt'
url: str = 'http://localhost:3000/upload'
upload_file(file_path, url)
```

Code 2: Example of Python3[3] script (High level)

```
#include <iostream>
#include <fstream>
#include <sstream>
#include <curl/curl.h>

void upload_file(const std::string &file_path, const std::string &url) {
    std::ifstream file(file_path);
    if (!file.is_open()) {
        std::cerr << "Could not open the file: " << file_path << std::endl;
        return;
    }

    std::stringstream buffer;
    buffer << file.rdbuf();
    std::string file_content = buffer.str();

    CURL *curl;
    CURLcode res;

    curl_global_init(CURL_GLOBAL_DEFAULT);
    curl = curl_easy_init();
    if(curl) {
        curl_easy_setopt(curl, CURLOPT_URL, url.c_str());
        std::string post_fields = "file=" + file_content;
        curl_easy_setopt(curl, CURLOPT_POSTFIELDS, post_fields.c_str());
        res = curl_easy_perform(curl);
        if(res != CURLE_OK)
            std::cerr << curl_easy_strerror(res) << std::endl;
        curl_easy_cleanup(curl);
    }
    curl_global_cleanup();
}

int main() {
    std::string file_path = "./file.txt";
    std::string url = "http://localhost:3000/upload";
    upload_file(file_path, url);
}
```

[3] Scripting language used on AI tools.

Code 3: Example of C++[4] app (Low level)

From a preliminary comparison, it might appear that Python offers a superior solution for certain problems due to its simplicity and ease of use. However, while both programs may achieve the same end result, the Python interpreter performs numerous background operations to simplify coding, which can slow down execution and obscure the programmer's understanding of the underlying processes. Although the final output in Python is clearer, the computational path is more convoluted.

In scripting languages like Python, declaring, defining, and using variables follow a uniform syntax, which simplifies the coding process but can be inefficient for execution. In contrast, low-level languages such as C++ allow the programmer to directly access memory addresses using the “&” symbol and manipulate them directly. This approach is more challenging to grasp but results in faster execution. C++ provides the programmer with precise control and understanding of the computer's operations. Additionally, it enables the programmer to choose whether variables are stored in a fixed, constant stack local scope, defined at compile time and automatically managed during runtime, or in the dynamic heap, where memory can be allocated and deallocated manually using this * symbol.

For these reasons, the examples in this project are written in C++. Using C++ ensures that the programmer has a comprehensive understanding of the computational processes and optimises performance by leveraging direct memory manipulation and explicit memory management.

Key concepts in programming languages enable the construction of virtually any software application. These foundational elements include variables, functions,

[4] Low level language designed to give freedom to the programmer about how they manage the memory.

operators, conditions, if statements, and loops. Understanding these concepts is essential for any programmer.:

Variables: Variables are memory addresses that store a value, allowing the value to be remembered and modified. Every stored datum must have a specified type to prevent errors. Common data types include 32-bit integers, decimals, text, and more. Variables are often declared by specifying the type (e.g., int, char, float, string) followed by the variable name, as in float natural_number;. They can be defined by assigning a value to the variable, for example, natural_number = 0;. Most programming languages also allow declaration and definition to occur simultaneously, such as float natural_number = 0;.

In functional programming, the value of a variable can depend on an initial input or various actions. This can be expressed using syntax like: int variable = int parameter => parameter + 1;. Here, the variable itself does not have a static value; instead, variable(1) would yield 2, and variable(2) would yield 3, illustrating the concept of functionally dependent variables.

Arrays: In computer memory, each variable is allocated a specific sequence of bits that represent its value. An array is a data structure that allows the storage of multiple instances of values of the same type, organised contiguously in memory. The total memory consumed by an array depends on both the size of each element (determined by its data type) and the number of elements it holds.

For example, an array of 8-bit integers could be: {255, 64, 32}, which in memory should look like: 11111111 01000000 00100000 (binary), or 0xFF 0x40 0x20

(hexadecimal[5]). Look that each value is stored using 8 digits of memory, and, since it holds three values, the array is stored using 24 bits.

Functions: Functions take some arguments and return a result after executing a series of instructions. They are declared by specifying the return type, the function name, and the arguments, as in: bool isEven(int n);. Functions are defined by providing the return type, function name, arguments, body of the function (actions to be performed), and the return value, for example: bool isEven(int n) { return n % 2 == 0; }.

Inline functions are explained at the *b* part from *Variables.*

Operators: They are symbols which call functions (* calls a product, and - calls a subtraction; for example).

Conditions: They are variables or functions that return true or false. These functions are usually called with operators, such as ==, for equal; >, for greater than; <, for less than…

If statements: They are void functions which take a condition as an argument and another function called callback as another; if the condition is true, they execute the callback.

Loops: They take a condition and execute a callback repeatedly until the condition is false.

[5] There are 16 digits from 0 to 9 plus A to F, A representing 10. FF is the maximum value and represents 255 in decimal, the maximum value that can be stored in 8 bits: 11111111. Hexadecimal is used to make looking at binaries less tough.

Classes and JObjects: An object is a variable that contains name-indexed variables, referred to as attributes. These attributes can sometimes be functions, in which case they are termed methods, as they can interact with and manipulate other attributes within the same object. Attributes can be categorised as either public or private. Public attributes are accessible to any entity within the scope, while private attributes are restricted, meaning they can only be accessed by other attributes within the same object…

Classes serve as blueprints for objects, providing a template from which multiple instances of the object can be created using constructor functions. Unlike regular attributes, static attributes belong to the class itself rather than to any particular instance, making them shared across all instances of that class.

```
 1 #include <iostream>
 2 #include <string>
 3
 4 //integer function
 5 int promt_n(std::string question) {
 6     int output;
 7     std::cout << question; //consol out
 8     std::cin >> output; //consol in
 9     return output;
10 }
11
12 //inline boolean function
13 const auto is_even = [](const int& n) { return n%2?false:true; };
                                    //% is the residue of the relation
14
15 int main() {
16     int number = promt_n("Say a number: ");
17
18     //condition
19     if (is_even(number)) {
20         std::cout << number << " is even :)" << std::endl;
21         return 0;
22     }
23
24     std::cout << number << " isn't even :(" << std::endl;
25     return -1;
26 }
```

Code 4: This super simple code here provides most of the elements explained

before. Any person who has read the project memory until here should be able to firmly know that this is going to behave like this:

```
Say a number: 6
6 is even :)
```

Or this:

```
Say a number: 3
3 isn't even :(
```

In the unlikely case where the reader couldn't guess the result of the code, they must return to the beginning and read again for the sake of understanding the rest of the survey.

1.2.2. Shaders

In computer graphics, there are primarily two types of shaders: vertex shaders and fragment shaders. Vertex shaders handle the geometry of models in a scene, determining the position of vertices in 3D space. Fragment shaders, on the other hand, manage the colour of the pixels displayed on the screen. This project focuses on the latter type, fragment shaders.

Shaders are crucial in graphics programming because applications frequently need to communicate with the computer to render visuals on the screen. This is especially true in 3D environments, where the colour and properties of each pixel must be calculated independently of its neighbours. This characteristic makes graphics rendering an ideal task for the GPU, which excels at performing millions of simple calculations simultaneously. The mathematical operations required for these calculations are specified in the fragment shader.

Fragment shaders typically take the pixel coordinates on the window as input and transform them into UV coordinates. UV coordinates are normalized values that range from 0 to 1, representing the relative position of a pixel within a texture or

image. This transformation allows for consistent and efficient manipulation of pixel data across different resolutions and screen sizes.

In practical terms, fragment shaders execute a series of operations for each pixel, determining its final colour based on various factors such as lighting, texture, and material properties. These operations can include complex mathematical functions, such as interpolation, blending, and procedural generation, all of which contribute to the final rendered image.

The ability of fragment shaders to perform highly parallelized computations makes them indispensable in modern graphics rendering. By leveraging the parallel processing power of the GPU, fragment shaders enable real-time rendering of detailed and dynamic scenes, which is essential for applications ranging from video games to simulations and virtual reality.

```
1 void mainImage( out vec4 fragColor, in vec2 fragCoord )
2 {
3    vec2 uv = (fragCoord/iResolution.xy)*vec2(2)-vec2(1);
4
5    float radius = 1.f;
6    vec2 centre = vec2(0,0);
7
8    vec3 col = vec3(0);
9
10   if (radius < sqrt(dot(uv - centre, uv - centre))) {
11       col = vec3(255);
12   }
13
14   fragColor = vec4(col,1.0);
15 }
```

Code 5: Example of shader written in GLSL[6] which detects if a pixel is inside a circle or not

[6] GLSL is a C based language that runs interpreted by applications on the gpu, it can run on any app which already runs OpenGL as a graphics application programming interface (API).

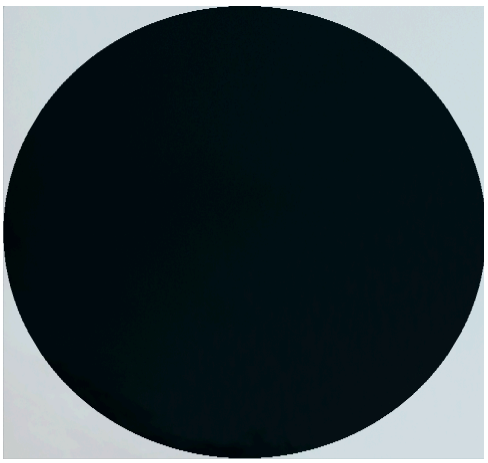


Fig. 2: Output[7] from the shader

[7] I rendered the image from the website Shadertoy which lets the user experiment with fragment shaders easily.

2. Ray tracing

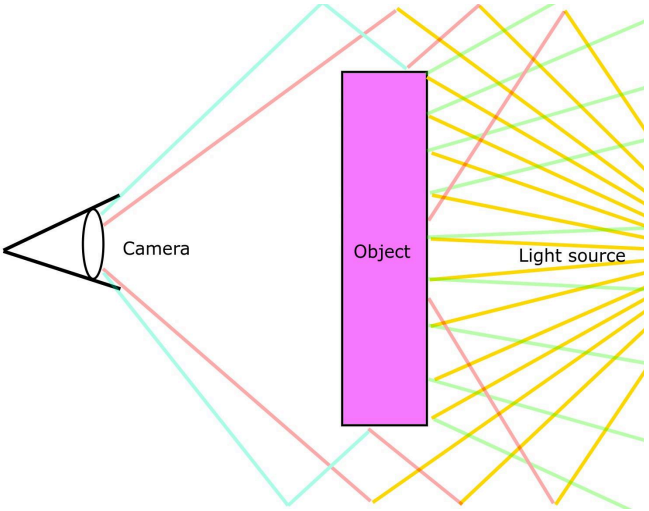
2.1. Definition

A ray tracer is a sophisticated application that creates a virtual environment and renders it using the ray tracing technique. Ray tracing involves simulating the path of light rays as they travel through a scene, determining the colours that a camera would perceive from its perspective based on the screen information and surrounding light sources.

While ray tracing does not delve into the fundamental nature of light, it is considered a physically based rendering method because it aims to accurately simulate the behaviour of light rays. By calculating the paths and interactions of light rays, ray tracing produces images with a high degree of realism, mimicking how light bounces off surfaces and interacts with objects to create realistic shading, reflections, and refractions.

The principle of ray tracing can be illustrated by considering a person in a dark room. Without light, the person cannot discern the objects in front of them. However, when the room is illuminated, the person can see and identify the objects based on the light that reflects off the surfaces and reaches their eyes. This analogy underscores the importance of light in visual perception and forms the basis of ray tracing in computer graphics.

Below are two drawings that illustrate this concept:



Flig. 3: This picture illustrates a situation: where there is a spectator in front of an object which has a light source behind.

If the viewer touches any yellow ray: they can see the light source; if they touch the red ones: they can see the walls; if they touch the green ones: they can see the object; if they touch the blue: they can see something which light is distorsioned by the object's colour.

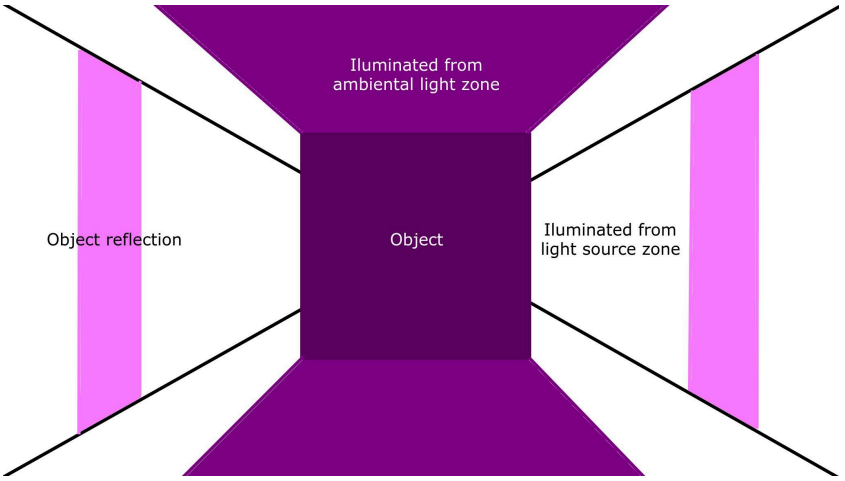


Fig. 4: The viewer's perspective following the rules of the picture 2.1.1

However, it is important to highlight that casting infinite rays from each light source to determine if they reach the camera is computationally expensive. Therefore, the process is typically reversed: the camera casts rays to determine if they intersect any light sources. On their journey, these rays may collide with various objects in the scene.

Ray tracing algorithms mimic this process by tracing the paths of light rays from the camera into the scene. For each pixel on the screen, a ray is cast from the camera, and its interactions with objects and light sources are calculated. The resulting colours and intensities are then used to render the final image.

2.1.1. Real life examples

If fig. 4 seemed unrealistic, here are some real-life examples of how ambient light affects shadows:

Fig. 5: Picture inside a bus which illustrates how the colour of the sky affects shadows while the colour of the sun affects bright zones.

An untrained eye which tried to paint that door could just use grey for the shadow and white for the light because the door is supposed to be white, however, the colours shown at the right of the picture are the real colours of the door. The values are so different.

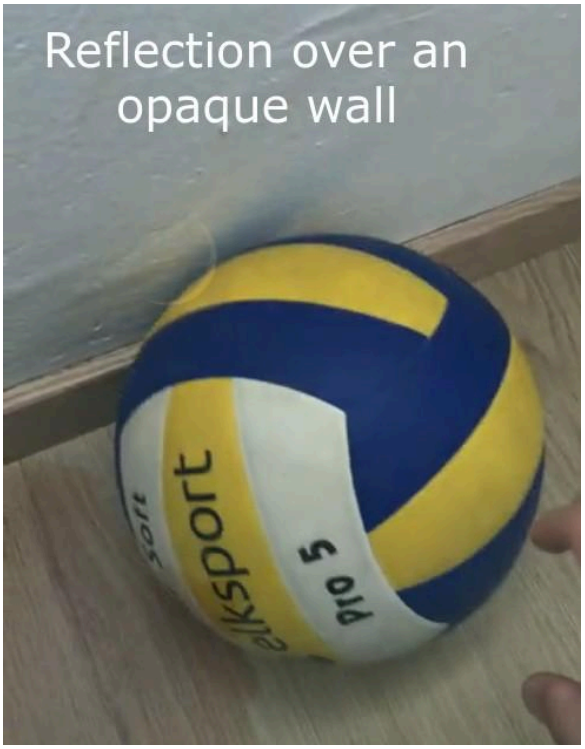


Fig. 6: This picture shows how a volleyball is able to cast a reflection onto an opaque wall. Its yellow shadow is stronger than the ambient light so it is seen as a reflection into the white wall. This video is more illustrative.

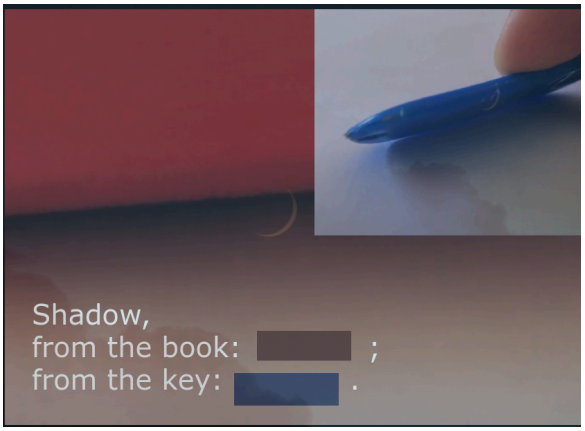


Fig. 7: The shadow of two different objects over the same white paper.

They have different colours due to the fact that the shadows are not illuminated by the lightsource. They are illuminated by the ambiental light which bounces with the objects that cast the shadows. For this reason the shadows adopt the colours of the objects which cast them.

Note that the shadows have the same brightness value, meaning the average intensity of the red, green, and blue channels is equal. As a result, in black and white, they would appear as the same shade of grey.

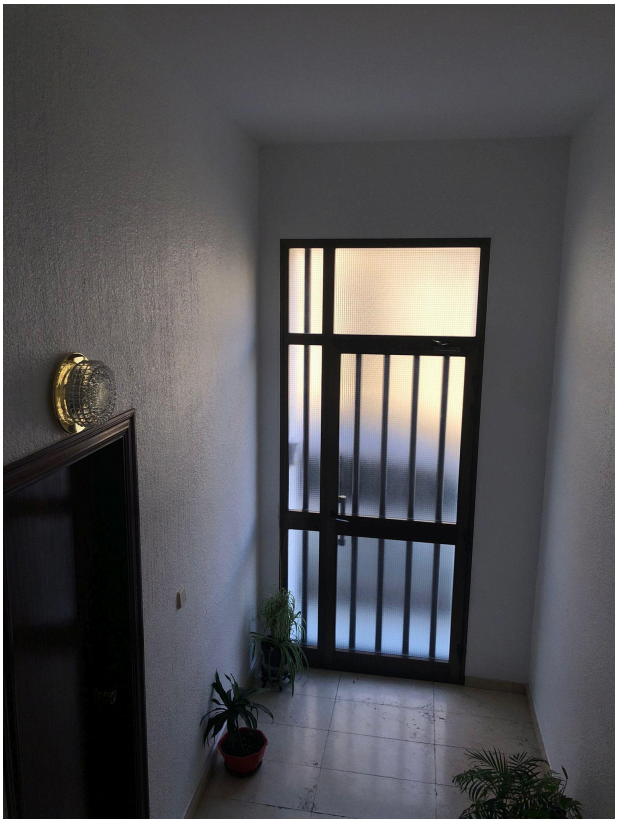


Fig. 7.5: Support to the meaning of figures 5 to 7

2.2. Logic

In ray tracing, each pixel on the screen is related to at least a ray which travels from the camera to a light source, memorising colour information to decide the colour of the pixel at the shader:

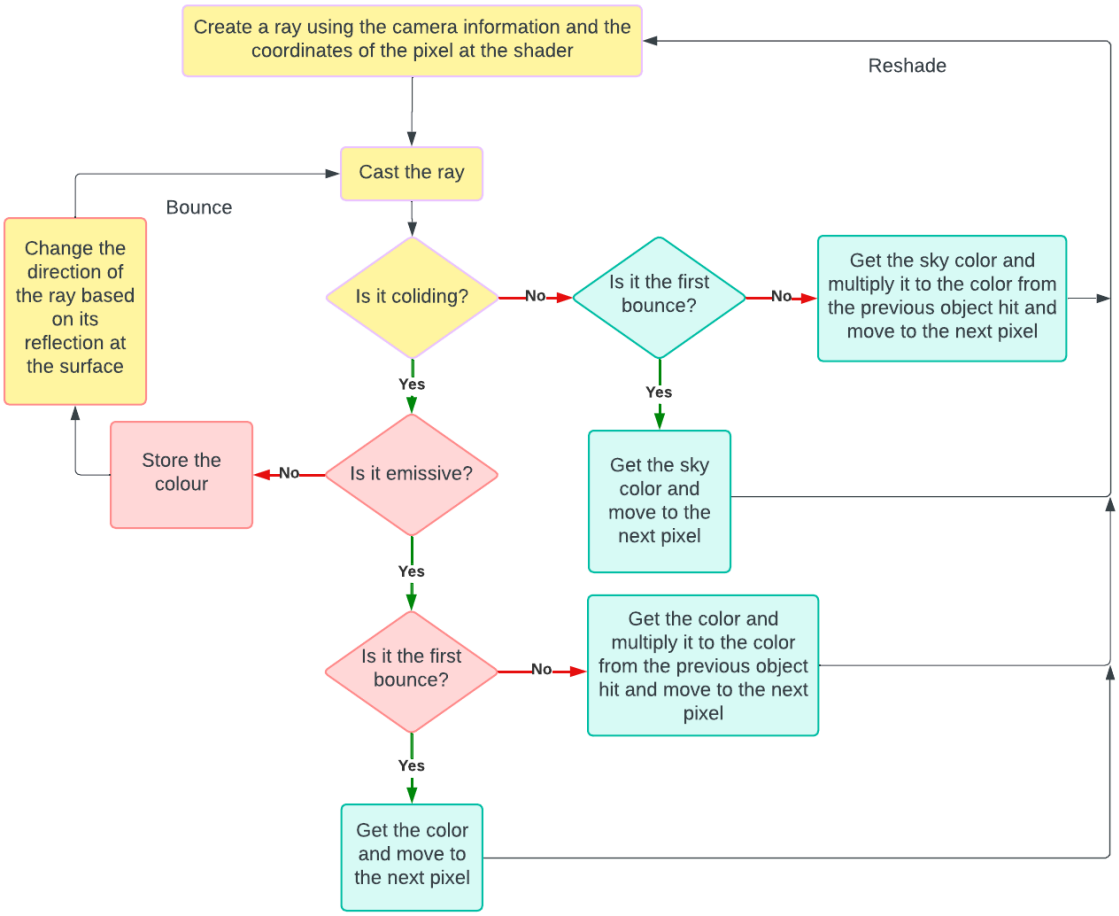


Fig. 8: Ray tracing's flow chart where:
Yellow: start, Red: calculate, Blue: end.

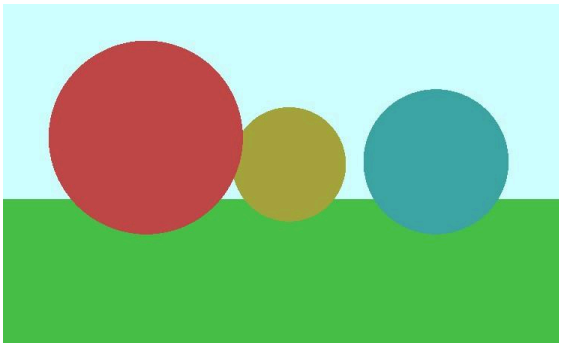


Fig. 9: First bounce's image

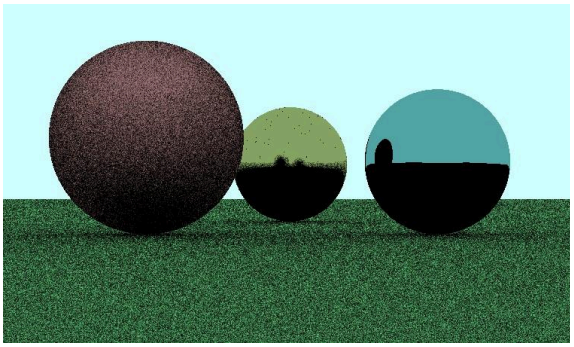


Fig. 10: Second bounce's image

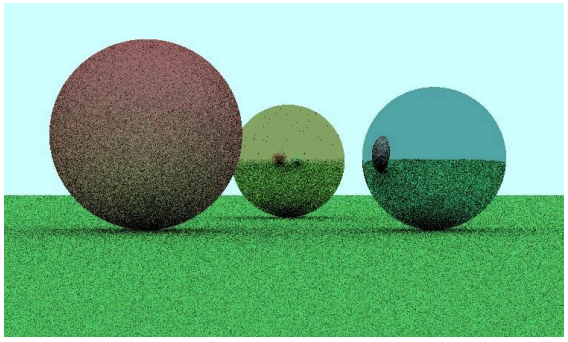


Fig. 11: Third bounce's image

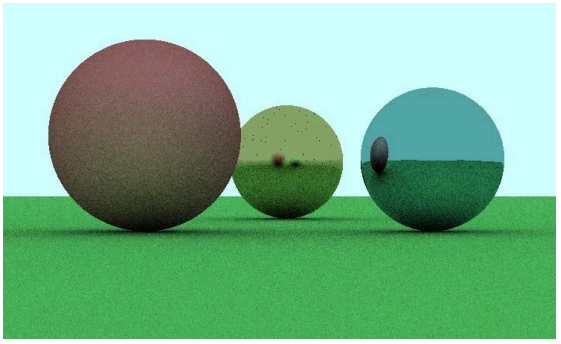


Fig. 12: Image with accumulation

As seen in the images above, the last bounce defines the colour on screen.

In ray tracing, each object in the scene is characterised by an attribute known as material. This material attribute encompasses various properties that significantly influence the appearance and interaction of the object with light. Key aspects defined by the material attribute include emissiveness, which dictates the amount of light the object emits; roughness, which affects the scattering of reflected rays; metallicness, determining the metallic quality and reflective behaviour; and colour, which provides the inherent hue of the object.

These material properties are crucial for determining the final colour of the pixels on the rendered image and for calculating the directions of reflected rays. For instance, the reflection on idealised mirrors, which are objects characterised by zero roughness, is entirely governed by the ray direction and the surface normal. This scenario is described by the specular reflection formula:

$$r: ReflectedDirection,\ i: IncomingDirection,\ n: Normal$$
$$r = i - 2 \cdot (i \cdot n) \cdot n$$

This formula can be explained by the behavior of mirrors. When a person looks at themselves in a mirror, they always see an image that appears to be half their actual size, regardless of the distance from the mirror. This occurs because looking into a mirror is similar to looking through a window, with the difference being that in a mirror, the viewer sees the opposite face of the image they would see through a window. In other words, the reflected image is the incoming light direction inverted across the normal.

This illustration demonstrates the phenomenon:

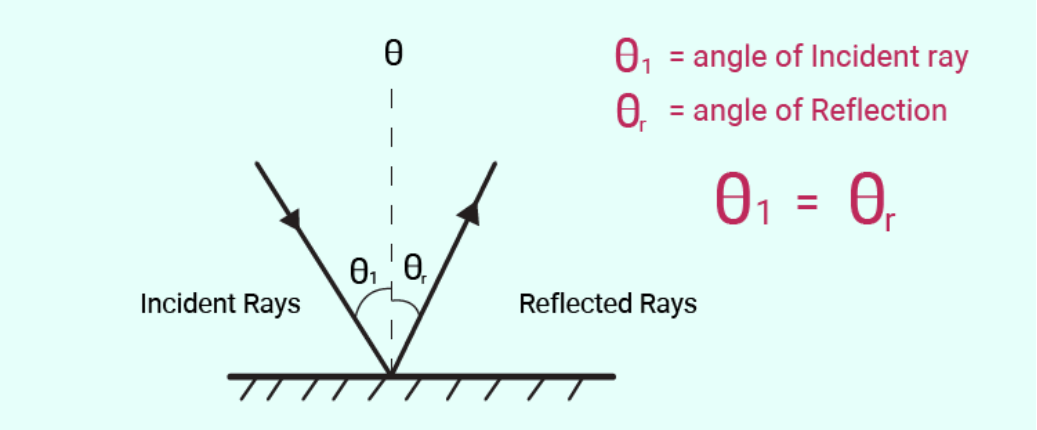


Fig. 13: Illustration of the ray reflection taken from My Tutor Source on an article wrote by Chloe Daniel

However, objects that are not perfectly flat, while still reflecting light in the same basic manner, produce a more distorted result. This occurs because the rays of light bounce in various directions in a pattern that can appear random.

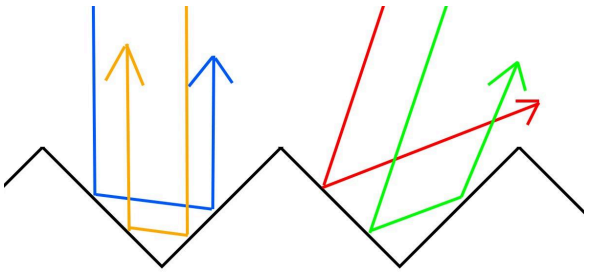


Fig. 14: Specular bounces on a rough surface

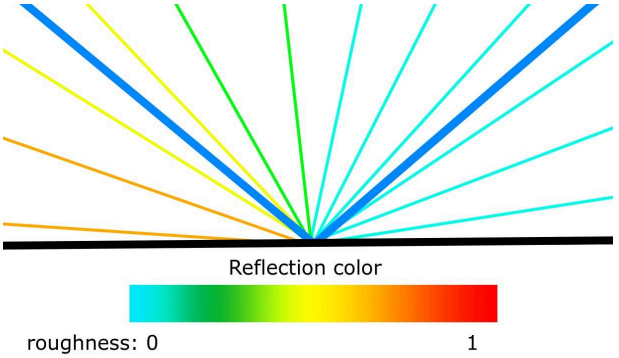


Fig. 15: Random bounces on a flat surface

Where the colour spectrum shows how random can get the bounce in relation to the roughness.

The most efficient way to simulate this roughness is by adding a random vector, within a range of 0 to the desired roughness, to the pre-calculated reflected direction on a flat surface with a single normal. This approach is preferable to calculating reflections across an extensive, infinitely rough mesh with millions of normals.

Owing to its simplicity, this is the main method I used in this project, however, this only applies to materials that have the same roughness along all their surfaces… Sometimes, a material shaded this way can look very flat. For this reason normal maps were invented. They are images which contain *rgb* information from the *xyz* coordinates from the normals of an object. This way, a ray tracer can create more realistic depth effects on flat materials.

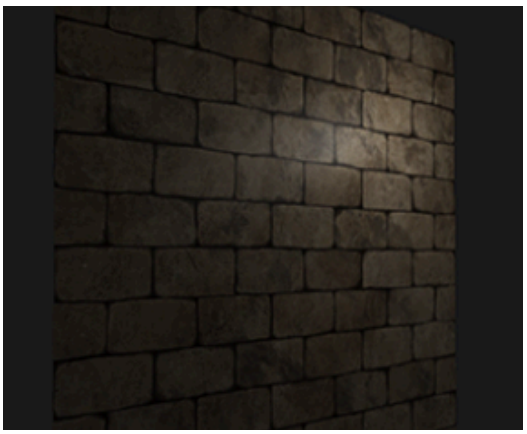


Fig. 16: Wall illuminated with random normals. Src: https://learnopengl.com

Fig. 17: Wall illuminated with normal mapping. Src: https://learnopengl.com

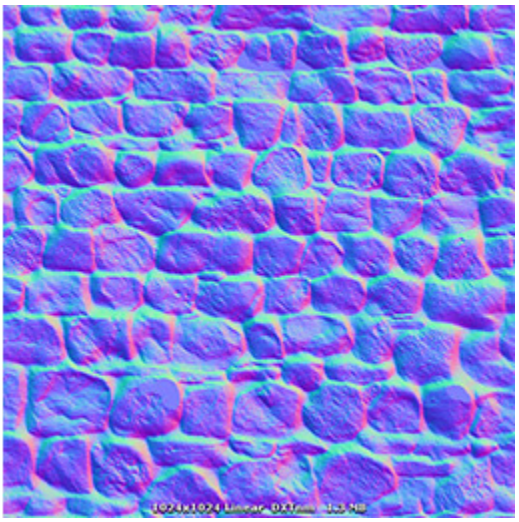


Fig. 18: Normal map texture example. Src: https://docs.unity3d.com

There are also other types of textures called bump maps which are used by vertex shaders to actually modify the geometry of meshes as it is shown in the following images:

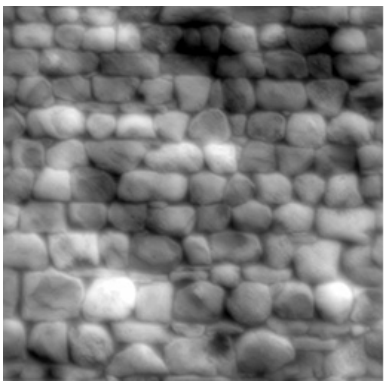


Fig. 19: Bump map texture example. Src: https://docs.unity3d.com

Fig. 20: A flat texture, a normalised texture, and a bumped texture in that order. Src: https://joannelaidlow.wordpress.com

2.3. Rendering spheres

Spheres are the easiest shape to render, which is why they are often the first objects implemented in ray tracers to test out if all is working just fine.

2.3.1. Intersection

The easiest shape to calculate intersections with is a sphere, using this equation:

$$r = radius,\ sphere\colon\ x^2 + y^2 + z^2 = r^2$$

If a point (x, y, z) is at the sphere: $x^2 + y^2 + z^2 = r^2$. Is inside if $x^2 + y^2 + z^2 < r^2$. And is outside if $x^2 + y^2 + z^2 > r^2$.

However this might not apply for a sphere which is not placed at the centre of the space. The for solving this issue is the following one:

$$C = Sphere\ Centre,\ sphere\colon (x - C_x)^2 + (y - C_y)^2 + (z - C_z)^2 = r^2$$

Applying the dot product formula[8] leads to a more clean equation which let the application know the relation between the intersection point of the ray and the sphere:

$$P = SpherePoint_n,$$

$$(P - C) \cdot (P - C) = (x - C_x)^2 + (y - C_y)^2 + (z - C_z)^2$$

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

The following equation takes that concept a bit further adding the ray equation, thus the time:

$$P(t) = rayOrigin + t * rayDirection,\ (P(t) - C) \cdot (P(t) - C) = r^2,$$

$$o\colon RayOrigin,\ d\colon RayDirection,\ ((o + td) - C) \cdot ((o + td) - C) = r^2$$

$$(td + o - C) \cdot (td + o - C) = r^2$$

This is indeed a second grade equation which can be solved with the following formula where:

[8] v · u = vx * ux + vy * uy = ||v|| * ||u|| * cos(φ)

$$a = d \cdot d,\ b = 2d \cdot (o - C),\ c = (o - C) \cdot (o - C) - r^2$$

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

```
struct Sphere {
    float radius;
    vec3 center;
};

bool collision(vec3 rayOrigin, vec3 rayDir, Sphere sphere) {
    vec3 oc = rayOrigin - sphere.center;
    float a = dot(rayDir, rayDir);
    float b = 2.f * dot(oc, rayDir);
    float c = dot(oc, oc) - sphere.radius * sphere.radius;
    float discriminant = b * b - 4.f * a * c;

    return (discriminant > 0.f);
}

void mainImage(out vec4 fragColor, in vec2 fragCoord) {
    vec2 uv = (fragCoord / iResolution.xy) * 2.f - 1.f;
    uv.x *= iResolution.x / iResolution.y;

    vec3 rayOrigin = vec3(0);
    vec3 rayDir = normalize(vec3(uv, 1));

    Sphere sphere;
    sphere.radius = 10.f;
    sphere.center = vec3(0,0,20);

    vec3 col = vec3(0);

    if (collision(rayOrigin, rayDir, sphere)) {
        col = vec3(1);
    }

    fragColor = vec4(col, 1);
}
```

Code 6: Example of shader that detects a sphere

Fig. 21: Output from the shader

2.3.2. Normal

Then it is necessary to know where the rays will bounce after colliding with the sphere, so it's necessary to know the normal of the hit point. This normal can be calculated, obviously, by subtracting the sphere centre to the hitpoint, and visualised with colour applying some changes to the shader:

So the changes can be:

```
13    if (discriminant < 0.0) {
14        return false;
15    }
16
17    float t = (-b - sqrt(discriminant)) / (2.0 * a);
18    hitPoint = rayOrigin + t * rayDir;
19    return true;
```

Code 7: Replace line 13 with this. It tells where the rays collide.

```
36    if (collision(rayOrigin, rayDir, sphere, hitPoint)) {
```

```
37          vec3 normal = normalize(hitPoint - sphere.center);
38          col = (normal + vec3(1.0)) * 0.5;
39      }
```

Code 8: Replace line 29's if statement with this one. Using the normal as a colour translates its xyz values onto rgb.

Fig. 22: Output (front and back side) where the X value of the normal is represented with red; the Y, with green; and the Z, with blue.

If there was a light source infinitely away from the sphere in some direction, the dot product between the normal and that direction results in a 0 to 1 value which can be used as brightness to guess the illumination which that light produces:

```
38          col = dot(normal, vec3(1,2,-1))*vec3(1);
```

Code 9: Replace line 38 with this.

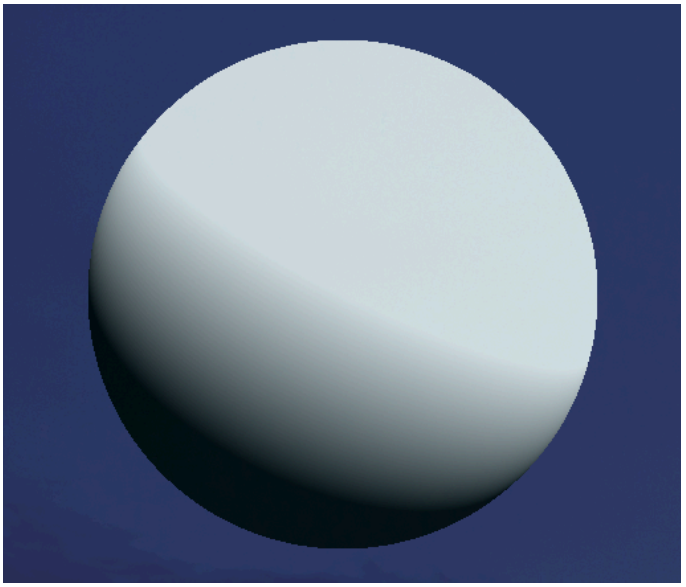


Fig. 23: Output.

2.3.3. Reflection

Now, to calculate a reflection there is the formula which was explained at point 2.2:

$$ReflectedDirection = r,\ IncomingDirection = i,\ Normal = n,$$
$$r = i - 2 \cdot (i \cdot n) \cdot n$$

```
22 void mainImage(out vec4 fragColor, in vec2 fragCoord) {
23    vec2 uv = (fragCoord / iResolution.xy) * 2.0 - 1.0;
24    uv.x *= iResolution.x / iResolution.y;
25
26    vec3 rayOrigin = vec3(0.0);
27    vec3 rayDir = normalize(vec3(uv, 1.0));
28
29    Sphere sphere0;
30    sphere0.radius = 13.0;
31    sphere0.center = vec3(-15.0, 0.0, 25.0);
32
33    Sphere sphere1;
34    sphere1.radius = 13.0;
35    sphere1.center = vec3(15.0, 0.0, 25.0);
36
37    vec3 col = vec3(0.2, 0.2, 0.4);
38    vec3 hitPoint;
```

```
39      vec3 reflectionDir = rayDir;
40      vec3 reflectionOrigin = rayOrigin;
41
42      for (int i = 0; i < 4; i++) {
43          if (collision(reflectionOrigin, reflectionDir, sphere0, hitPoint)) {
44              vec3 normal = (hitPoint - sphere0.center);
45              reflectionDir = reflectionDir - 2.0 * dot(reflectionDir, normal) * normal;
46              reflectionOrigin = hitPoint+normal*0.01;
47              col = vec3(1, 0, 0);
48          }
49          if (collision(reflectionOrigin, reflectionDir, sphere1, hitPoint)) {
50              vec3 normal = (hitPoint - sphere1.center);
51              reflectionDir = reflectionDir - 2.0 * dot(reflectionDir, normal) * normal;
52              reflectionOrigin = hitPoint+normal*0.01;
53              col = vec3(0, 0, 1);
54          }
55      }
56
57      fragColor = vec4(col, 1.0);
58  }
```

Code 10: New main function.

At the code shown there is a loop which iterates through bounces. Each one, the ray origin and direction are set to the hit point and reflection of the last bounce. Now, each bounce takes as colour the sphere that hits, which gives this mirroring effect:

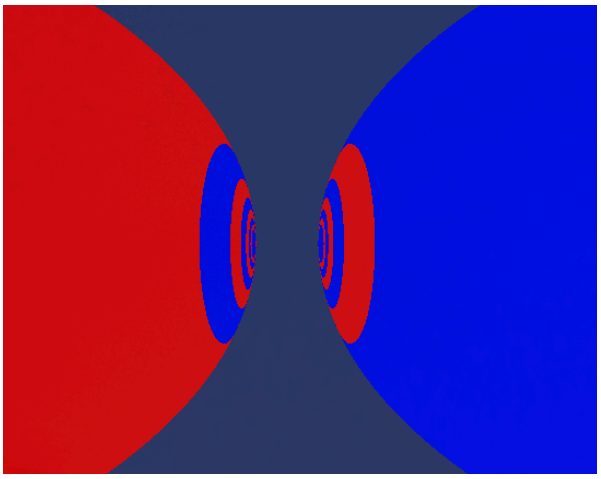


Fig. 24: Mirroring spheres.

If the sphere surface wasn't perfect, the rays would not bounce straight in their reflections. In order to achieve rough materials there has to be a roughness variable which multiplies to a random vector which is later added to the reflected direction.

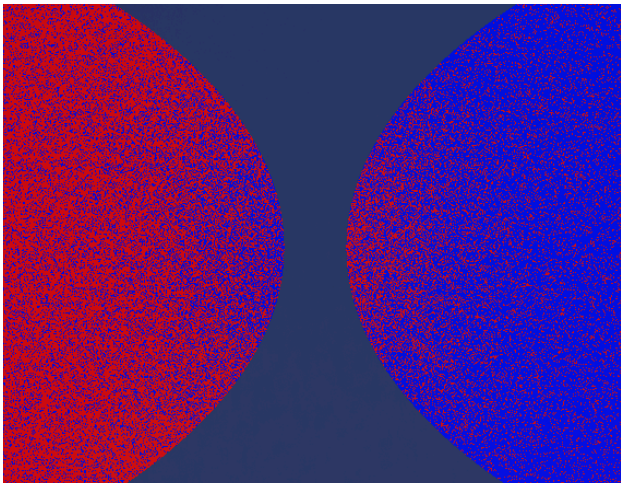


Fig. 25: Output with roughness.

2.3.4. Texturing

In order to visualise an image over the surface of a sphere it is necessary to relate the relative coordinates of the hitpoint between the ray and the sphere at the sphere, and its equivalent in an image file.

This is made by calculating the angles of the normal vector which points to that hitpoint and transforming them to UV coordinates. Which means a vector of 0 to 1 values which represents a position in the image.

$$\theta = acos(N_y),\ \varphi = atan(\frac{N_z}{N_x})$$

$$U = \frac{\varphi+\pi}{2\pi},\ V = \frac{\theta}{\pi}$$

These UV coordinates can be converted to real coordinates by multiplying with the image width and height.

$$width\colon w,\ height\colon h,\ X = Uw,\ Y = Vh$$

```
29    const sf::Color txtr(const glm::vec3 normal) const {
30        float theta = std::acos(normal.y);
31        float phi = std::atan2(normal.z, normal.x);
32
33        float u = (phi + M_PI) / (2 * M_PI);
34        float v = theta / M_PI;
35
36        int texWidth = texture.getSize().x;
37        int texHeight = texture.getSize().y;
38        int x = static_cast<int>(u * texWidth) % texWidth;
39        int y = static_cast<int>((texHeight - v)*texHeight)%texHeight;
40
41        sf::Color color = texture.getPixel(x, y);
42        return sf::Color(color.r*this->color.r/255.f,
                           color.g*this->color.g/255.f,
                           color.b*this->color.b/255.f);
43    }
```

Code 11: C++ implementation which was taken from the actual source code[9] from the final project.

[9] Means all the textfiles meant to be compiled. Mine is uploaded to GitHub.

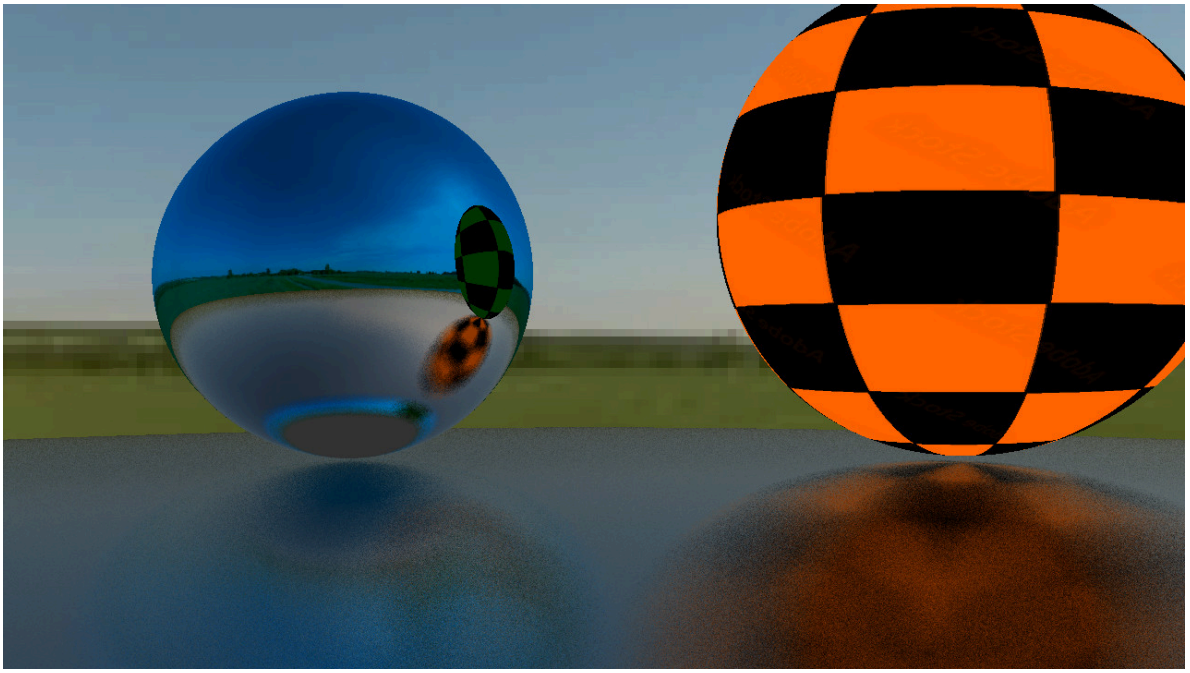


Fig. 26: Resulting output from convincing all the texturing and reflection techniques for spheres and the overall workflow form fig. 8.

2.4. Rendering triangles

Rendering triangles is the most natural next step because they are the simplest shapes that can exist in a plane. This is why all modern 3D meshes are based on triangles. Therefore, a renderer capable of rendering triangles can also render 3D models, which is the ultimate goal of the project.

2.4.1. Intersection

2.4.1.1. Casting planes

A triangle in 3D space can be defined by specifying a center point and two direction vectors representing its sides. Every triangle lies in a plane, which can be described using one of the triangle's points and its normal vector. The normal vector can be calculated easily by taking the cross product of the two direction vectors:

$$P_0\text{: centre on the plane, } P_f\text{: any point on the plane, } n\text{: plane's normal,}$$
$$n \cdot \Delta P = 0$$

All the points which are simultaneously at the plane and at the ray can be described at the ray-plane equation by using the ray equation explained at the point 2.3.1, and the plane equation.

$$o\text{: ray origin, } d\text{: ray direction, } ray\text{: } P(t) = o + td,$$
$$RayPlane\text{: } n \cdot (P(t) - P_0) = 0,$$

For the ray tracer what is important to calculate is the moment where the collision occurs: t. Then, this formula can be rearranged to fit that necessity as follows:

$$RayPlane\text{: } n \cdot (o + td - P_0) = 0,$$
$$RayPlane\text{: } n \cdot o + t \cdot (n \cdot d) - n \cdot P_0 = 0,$$
$$RayPlane\text{: } t = \frac{n \cdot (P_0 - o)}{n \cdot d}$$

```
1 struct Triangle {
2     vec3 normal;
3     vec3 center;
4     vec3 d[2];
5 };
6
7 bool collision(vec3 rayOrigin, vec3 rayDir, Triangle plane, out float t) {
8     float denominator = dot(rayDir, plane.normal);
9
10    if (abs(denominator) > 1e-6) {
11        vec3 diff = plane.center - rayOrigin;
```

```
12          t = dot(diff, plane.normal) / denominator;
13
14          if (t >= 0.0f) {
15              return true;
16          }
17      }
18
19      return false;
20  }
21
22  void mainImage(out vec4 fragColor, in vec2 fragCoord) {
23      vec2 uv = (fragCoord / iResolution.xy) * 2.f - 1.f;
24      uv.x *= iResolution.x / iResolution.y;
25
26      vec3 rayOrigin = vec3(0);
27      vec3 rayDir = normalize(vec3(uv, 1));
28
29      Triangle plane;
30      plane.d[0] = vec3(0,1,1);
31      plane.d[1] = vec3(1,1,0);
32      plane.normal = cross(plane.d[0], plane.d[1]);
33      plane.center = vec3(0, 0, 1);
34
35      vec3 col = vec3(0);
36      float t;
37
38      if (collision(rayOrigin, rayDir, plane, t)) {
39          col = vec3(1);
40      }
41
42      fragColor = vec4(col, 1);
43  }
```

Code 11: Example of shader that detects a plane

Fig.27: Output.

2.4.1.2. Casting triangles

Barycentric coordinates provide us with a way to express the position of a point within a triangle in terms of the triangle's vertices. For a point *P* inside a triangle with vertices *A*, *B*, and *C*, the point can be represented as:

$$triangle: \{\ T \mid P = uA + vB + wC,\ u + v + w = 1\}$$

Which indeed means exactly the same as the plane formula from point 2.4.1.1:

$$triangle:\ n \cdot \Delta P = 0$$

The coordinates *u*, *v*, and *w* are the areas opposite the vertices *A*, *B*, and *C* respectively, normalised by the area of the triangle.

Then to start simplifying, this definitions will be useful:

$$v_0 = B - A,\ v_1 = C - A,\ v_2 = P - A$$

The rearrangement starts making a system of equations between the two formulas:

$$P = uA + vB + wC \text{ and } u + v + w = 1$$

The first step would be to take w out from the main equation:

$$\begin{aligned} P &= uA + vB + (1 - u - v) \cdot C, \\ P &= uA + vB + C - uC - vC, \\ P &= u(A - C) + v(B - C) + C, \\ P - C &= u(A - C) + v(B - C), \\ v_2 &= u(-v_1) + v(v_0 - v_1) \end{aligned}$$

Then it is needed an other rearrangement:

$$v_2 \cdot v_0 = [u(-v_1 \cdot v_0) + v(v_0 \cdot v_0 - v_1 \cdot v_0)],$$
$$v_2 \cdot v_1 = [u(-v_1 \cdot v_1) + v(v_0 \cdot v_1 - v_1 \cdot v_1)],$$

After this, it is useful to proceed with the following definitions:

$$d00 = v_0 \cdot v_0,\ d01 = v_0 \cdot v_1 \cdots$$

Resulting into: $d20 = [u(-d10) + v(d00 - d10)]$, and

$$d21 = [u(-d11) + v(d01 - d11)]$$

This can be expressed with two linear equations:

$$d20 = -u \cdot d01 + v \cdot (d00 - d01), \text{ and } d21 = -u \cdot d11 + v \cdot (d01 - d11)$$

The next step must be to separate u, v and w from the equations using a system:

$$v = \frac{d11 \cdot d20 - d01 \cdot d21}{d00 \cdot d11 - d01 \cdot d01},$$
$$u = \frac{d00 \cdot d21 - d01 \cdot d20}{d00 \cdot d11 - d01 \cdot d01},$$
$$w = 1 - v - u$$

This way: only when u+v+w = 1, the hit point will be at the triangle.

```
 7 bool collision(vec3 rayOrigin, vec3 rayDir, Triangle plane, out float t) {
 8     float denominator = dot(rayDir, plane.normal);
 9
10     if (abs(denominator) > 1e-6) {
11         vec3 diff = plane.center - rayOrigin;
12         t = dot(diff, plane.normal) / denominator;
13
```

```
14          if (t >= 0.0f) {
15              vec3 hitp = rayOrigin + t * rayDir;
16
17              vec3 v0 = plane.d[0];
18              vec3 v1 = plane.d[1];
19              vec3 v2 = hitp - plane.center;
20
21              float d00 = dot(v0, v0);
22              float d01 = dot(v0, v1);
23              float d11 = dot(v1, v1);
24              float d20 = dot(v2, v0);
25              float d21 = dot(v2, v1);
26              float denom = d00 * d11 - d01 * d01;
27
28              float v = (d11 * d20 - d01 * d21) / denom;
29              float w = (d00 * d21 - d01 * d20) / denom;
30              float u = 1.0f - v - w;
31
32              bool cond = (u >= 0.f) && (v >= 0.f) && (w >= 0.f);
33              if (cond) {
34                  return true;
35              }
36          }
37      }
38
39      return false;
40  }
```

Code 12: New collision function

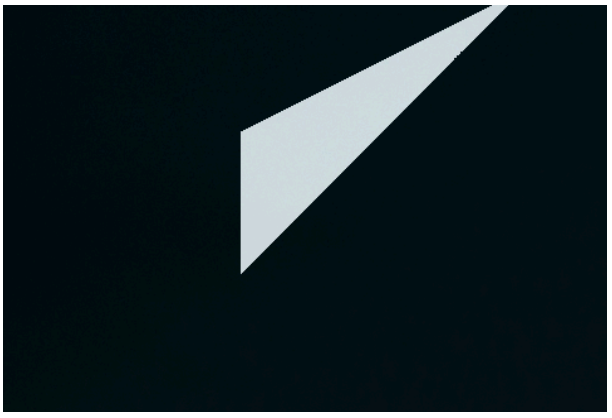


Fig. 28: Output.

2.4.2. Normal

The normal of the triangle is given by the cross product between two sides of itself and it can be visualised the following way:

```
39          col = normalize(plane.normal);
```

Code 13: Replacement for line 39 from code 12.

Also, it is nice to have multiple triangles to compare their normals:

```
 1 Triangle t0;
 2 t0.d[0] = vec3(0, 1, 1);
 3 t0.d[1] = vec3(1, 1, 0);
 4 t0.normal = cross(t0.d[0], t0.d[1]);
 5 t0.center = vec3(0, 0, 1);
 6
 7 Triangle t1;
 8 t1.d[0] = vec3(1, 1, 0);
 9 t1.d[1] = vec3(1, -1, 1);
10 t1.normal = cross(t1.d[0], t1.d[1]);
11 t1.center = vec3(0, 0, 1);
12
13 vec3 col = vec3(0);
14 float t;
15
16 bool col0 = collision(rayOrigin, rayDir, t0, t);
17 bool col1 = collision(rayOrigin, rayDir, t1, t);
18
19 if (col0) {
20    col = t0.normal;
21 }
22 if (col1) {
23    col = t1.normal;
24 }
```

Code 14: Code to be inserted at the main function

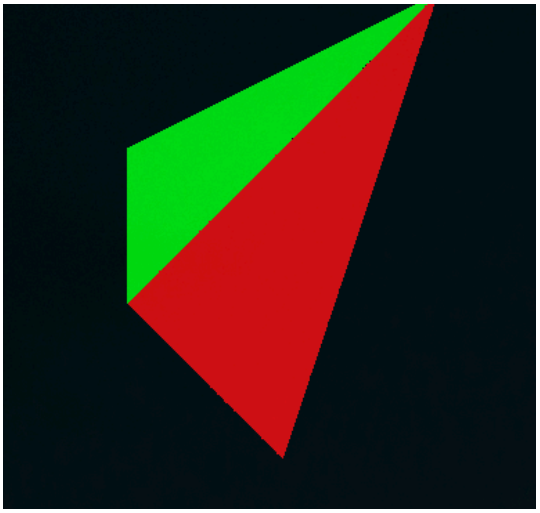


Fig. 29: Output

As the faces are flat, normals are seen as flat colours: they are constant along all their faces.

2.4.3. Reflection

Now, to calculate a reflection there is the formula which was explained at point 2.2:

$$ReflectedDirection = r,\ IncomingDirection = i,\ Normal = n,$$
$$r = i - 2 \cdot (i \cdot n) \cdot n$$

Where n is the normal of the triangle, the cross product from two of its sites.

```
1 for (int i = 0; i < 4; i++) {
2    float t;
3    bool col0 = collision(rayOrigin, rayDir, t0, t);
4    bool col1 = collision(rayOrigin, rayDir, t1, t);
5
6    if (col0) {
7        vec3 normal = t0.normal;
```

```
 8         rayDir = rayDir - 2.0 * dot(rayDir, normal) * normal;
 9         rayOrigin = rayDir*t+rayOrigin;
10         col = normal;//vec3(1, 0, 0);
11     }
12     if (col1) {
13         vec3 normal = t1.normal;
14         rayDir = rayDir - 2.0 * dot(rayDir, normal) * normal;
15         rayOrigin = rayDir*t+rayOrigin;
16         col = normal;//vec3(0, 1, 0);
17     }
18 }
```

Code 15: Bounce loop

At the code shown there is a loop which iterates through bounces. Each one, the ray origin and direction are set to the hit point and reflection of the last bounce. Now, each bounce takes as colour the sphere that hits, which gives this mirroring effect:

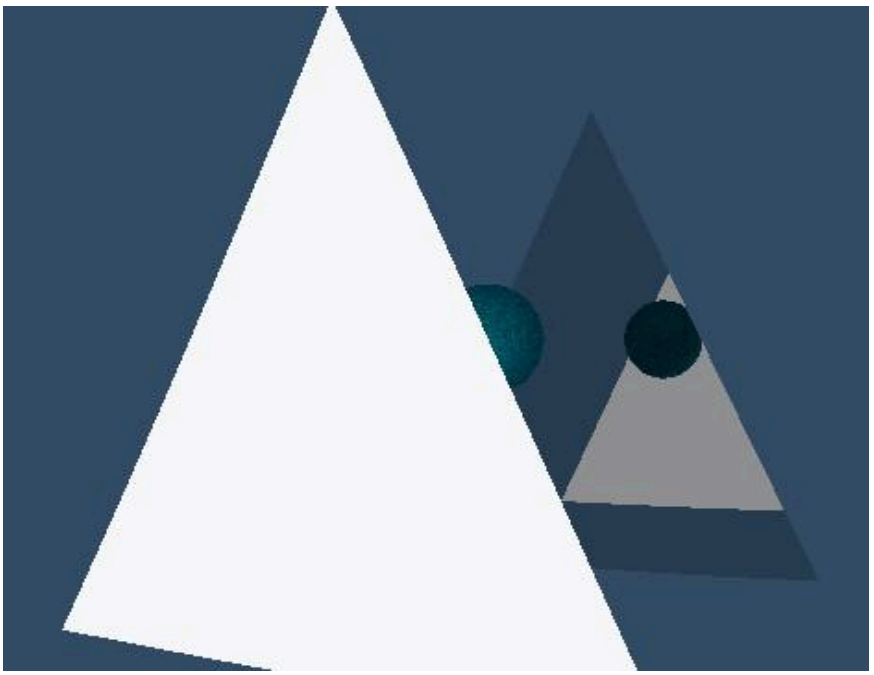


Fig. 30: Output

If the sphere surface wasn't perfect, the rays would not bounce straight in their reflections. In order to achieve rough materials there has to be a roughness variable which multiplies to a random vector which is later added to the reflected direction.

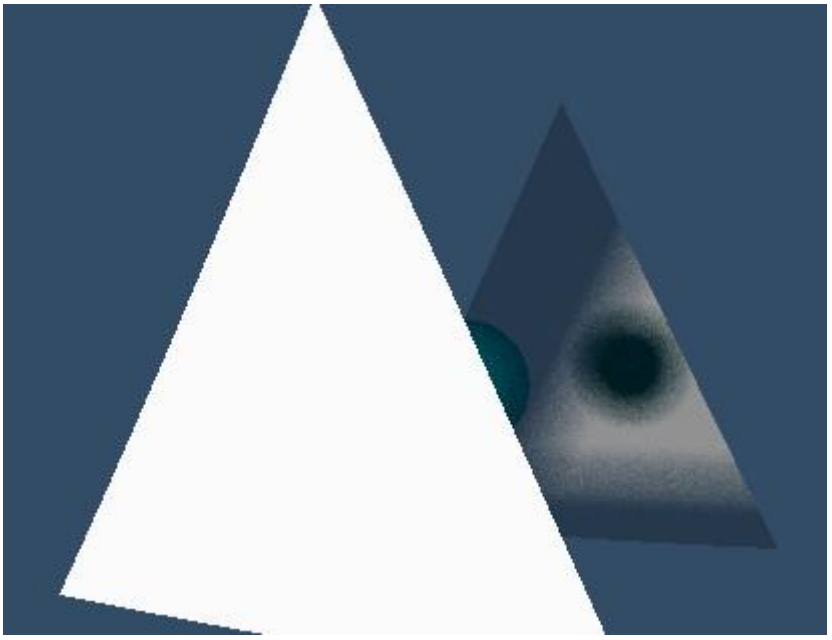


Fig. 31: Output with roughness.

2.4.4. Texturing

To determine which part of the texture a hitpoint on a triangle corresponds to, several parameters must be considered: the triangle's size, its normal, its centroid, and the hitpoint itself.

To achieve this, the following method involves establishing two vectors that define the plane of the triangle: the tangent vector *T* and the bitangent vector *B*. Here's a structured approach:

To calculate the tangent vector based on the normal, there is this function:

$$T\colon f(N) = \left\{ (N_z, 0, -N_x)\ if\ |N_x| > |N_y|,\ (0, N_z, -N_y)\ if\ |N_x| \leq |N_y| \right\}$$

To determine the bitangent vector, as it is perpendicular to both the normal and the tangent, it can be solved using the cross product:

$$B = N \times T$$

To assign texture coordinates, once *T* and *B* are established, the texture coordinates UV for the hitpoint can be computed using:

$$P\colon hitpoint,\ C\colon triangle\ centre,$$
$$u = (P - C) \cdot T,$$
$$v = (P - C) \cdot B$$

By following these steps, texture coordinates are determined corresponding to any hitpoint on the triangle's surface, ensuring clarity and accuracy in the mapping process.

```
47    sf::Color txtr(glm::vec3 hitp, glm::vec3 center, glm::vec3 normal) const {
48        float planeWidth = 100.0f;
49        float planeHeight = 100.0f;
50
51        glm::vec3 tangent, bitangent;
52        if (fabs(normal.x) > fabs(normal.y)) {
53            tangent = glm::vec3(normal.z, 0, -normal.x);
54        } else {
55            tangent = glm::vec3(0, -normal.z, normal.y);
56        }
57        tangent = glm::normalize(tangent);
58        bitangent = glm::normalize(glm::cross(normal, tangent));
59
```

```
60          glm::vec3 localHitPoint = hitp - center;
61          float u = glm::dot(localHitPoint, tangent) / planeWidth;
62          float v = glm::dot(localHitPoint, bitangent) / planeHeight;
63
64          u = fmod(u, 1.0f);
65          v = fmod(v, 1.0f);
66          if (u < 0) u += 1.0f;
67          if (v < 0) v += 1.0f;
68
69          unsigned int texWidth = texture.getSize().x;
70          unsigned int texHeight = texture.getSize().y;
71          unsigned int texX = static_cast<unsigned int>(u * texWidth);
72          unsigned int texY = static_cast<unsigned int>(v * texHeight);
73
74          return texture.getPixel(texX, texY);
75      }
```

Code 16: Code which decides the colour from the texture at the hitpoint in C++ taken from the final project.

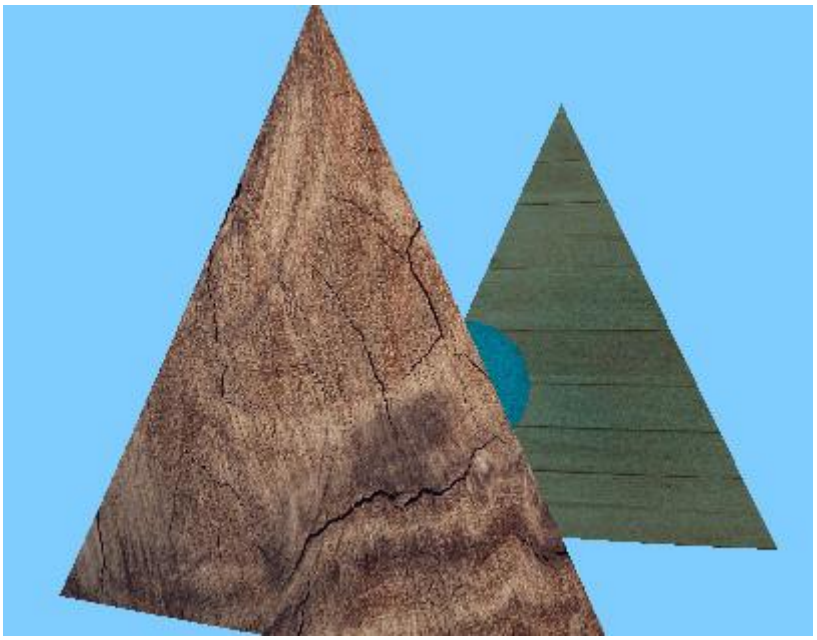


Fig. 32 Output.

2.5. Rendering meshes

A mesh is a series of triangles. In order to render it, it is necessary to create a way that converts any 3D model file format into whatever the program uses to place objects in the scene.

In the application made, explained at point 3, the scene is created by reading a .json[10] file which contains the information necessary to create triangles and spheres at the project.

In other words, it is needed a parallel application which converts models 3D (for the sick of simplicity .obj[11]) to a .json file with all the information for the ray tracer.

```
 1 {
 2     "sky": [0.2,0.3,0.4],
 3     "sphere": [
 4         {
 5             "center": [0,5,0],
 6             "radius": 8,
 7             "color": [255,0,0],
 8             "emission": 1,
 9             "roughness": 5000,
10             "texture": "./bin/texture/wall.jpg"
11         }
           [...]
20     ],
21     "triangle": [
22         {
23             "center": [-20,20,-20],
24             "s0": [0,0,40], "s1": [0,-40,20],
25             "color": [255,255,255],
26             "emission": 0,
27             "roughness": 0,
```

[10] JavaScript Object Notation: uses brackets and commas to define objects and attributes. Example: {“a”:[0,1,2], “b”:”012”}.
[11] Saves without compression the coordinates in 3D space of each vertex.

```
28              "texture": ""
29          }
            [...]
54      ]
55  }
```

Code 16: This is a real example of how all the information of the scenes is stored at the project.

Several file formats are available for exporting 3D models, but the .obj format is the easiest to understand. Therefore, it will be used to demonstrate how to convert a 3D mesh file into a .json file that my program can read.

The .obj file format defines mesh vertices using lines beginning with "v" followed by coordinates like "v 0 1 2". Faces are defined by lines starting with "f" followed by vertex indices, such as "f 3 4 5", specifying triangles.

Taking this into account, this is a little python script which converts .obj meshes into .json triangles:

```
 1 import json
 2
 3 def parse_obj_file(file_path):
 4     vertices = []
 5     faces = []
 6
 7     with open(file_path, 'r') as file:
 8         for line in file:
 9             if line.startswith('#') or line.strip() == '':
10                 continue
11
12             parts = line.split()
13
14             if parts[0] == 'v':
15                 vertex = list(map(float, parts[1:4]))
16                 vertices.append(vertex)
17             elif parts[0] == 'f':
18                 face = [int(index.split('/')[0]) - 1 for index in parts[1:]]
19                 faces.append(face)
20
21     return vertices, faces
22
23 class Triangle:
```

```
    def __init__(self, c, s0, s1):
        self.center = c
        self.s0 = s0
        self.s1 = s1
        self.color = [255, 200, 100]
        self.emission = 0
        self.roughness = 9999999
        self.texture = ""

    def to_dict(self):
        return {
            "center": self.center,
            "s0": self.s0,
            "s1": self.s1,
            "color": self.color,
            "emission": self.emission,
            "roughness": self.roughness,
            "texture": self.texture
        }

obj_file_path = './bin/models/sphere.obj'
vertices, faces = parse_obj_file(obj_file_path)

triangles = []

for face in faces:
    v1 = vertices[face[0]]
    v2 = vertices[face[1]]
    v3 = vertices[face[2]]

    center = v1
    s1 = [v2[i] - v1[i] for i in range(3)]
    s2 = [v3[i] - v1[i] for i in range(3)]

    triangle = Triangle(center, s1, s2)
    triangles.append(triangle.to_dict())

json_content = {
    "res": [16*3, 9*3],
    "sky": [0.8, 0.9, 1],
    "sphere": [],
    "triangle": triangles
}

json_file_path = './bin/scene.json'

with open(json_file_path, 'w') as json_file:
    json.dump(json_content, json_file, indent=4)

print(f"Triangles saved to {json_file_path}")
```

Code 17: This script takes arrays of vectors and faces from the .obj file, converts them into multiple triangle objects, and stores them in my scene.json file.

The script first stores all the vertices and faces from the .obj file in two separate arrays. Then, it creates a list of triangles using a class. It adds the triangles to the content that is supposed to be at *scene.json* and saves it.

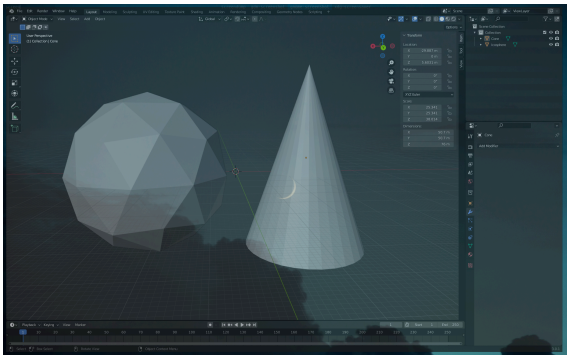


Fig. 33: A 3D model being visualised at Blender[12].

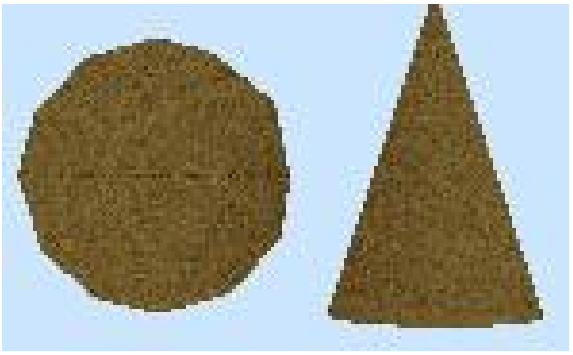


Fig. 34: A 3D model being visualised at this project's ray tracer.

Fig. 35: Comparison between a perfect sphere and a low-poly one.

[12] This software is used to create 3D meshes, commonly employed in movies and video games. It is specifically designed for artistic purposes.

2.6. Antialiasing

Antialiasing is meant to smooth the edges on a pixelated image. A diagonal in a grid of pixels is not straight, it has a stair shape. Antialiasing smooths that stair by blurring all the pixels together, as if the spectator needed to wear glasses.

This technique works well in high resolutions as seen in these pictures:

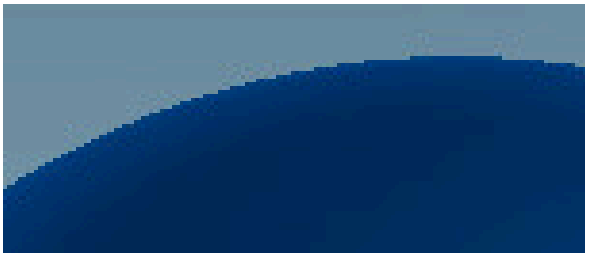


Fig. 36: Edge from fig. 26 with no antialiasing.

Fig. 37: Edge from fig. 26 with antialiasing.

However, it does not work as intended when the resolution is too low because what antialiasing does, in fact, is lower the resolution even more by fading multiple pixels together…

Fig. 38: Screenshot from the final

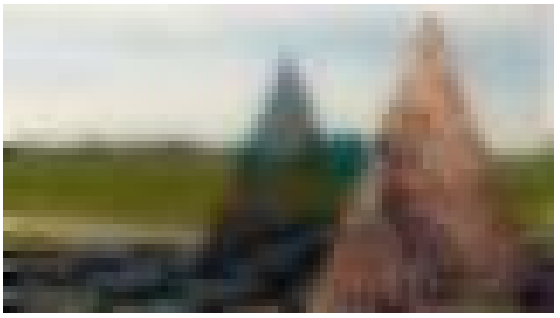


Fig. 39: Screenshot from the final

program with no antialiasing (80x45 px).	*program with antialiasing (80x45 px).*

There are mainly[13] two algorithms for antialiasing: FXAA, which just blurs the whole image; and TAA which uses passed frame information to detect edges and blur them. For the sake of maintaining performance, this project will focus on FXAA. An image can be blurred evenly or weighted. An example of weighted blurring could be gaussian blur, which uses a gaussian function to define the importance of each neighbour's pixel colour over the pixel that is being modified.

```
106 if (x < 1 || x > buff_v.x-1 || y < 1 || y > buff_v.y-1) {
107   return;
108 }
109
110 float r = 0.f, g = 0.f, b = 0.f;
111 for (int offx = -1; offx <= 1; offx += 1) {
112   for (int offy = -1; offy <= 1; offy += 1) {
113     if(offx == 0 && offy == 0) {
114       r += this->m_buffer.getPixel(x + offx, y + offy).r * 2;
115       g += this->m_buffer.getPixel(x + offx, y + offy).g * 2;
116       b += this->m_buffer.getPixel(x + offx, y + offy).b * 2;
117     } else {
118       r += this->m_buffer.getPixel(x + offx, y + offy).r;
119       g += this->m_buffer.getPixel(x + offx, y + offy).g;
120       b += this->m_buffer.getPixel(x + offx, y + offy).b;
121     }
122   }
123 }
124 r /= 10.0f;
125 g /= 10.0f;
126 b /= 10.0f;
127 fcolor = sf::Color(static_cast<sf::Uint8>(r),
128                    static_cast<sf::Uint8>(g),
129                    static_cast<sf::Uint8>(b));
```

Code 18: Example code which adds to a pixel their neighbours colours counting itself by two and then making the average dividing by 10. This creates blur.

[13] You might want to look deeper on each one by yourself by searching for SSAA, SMAA, FXAA, or TAA.

3. The application

The purpose of this point is solely to explain the logic of how the program operates. It serves as a guided walkthrough of the source code files, detailing how each part contributes to the overall functionality. Allso, contains some useful features for ray tracing and computer graphics that weren't included on the ray tracing explanation like how a 3D camera works in general.

This application uses github as a manager for version control, which means, all the updates made sins it was started until now are public in the following repositories: The actual ray tracer[14], an initial descartes ray tracer, and a 3d renderer which i worked on earlier. For the moment, the project has only linux support, which means, if someone wants to run on any other os they need to compile by themselves. In spite of that, everyone is free to download the files and do with them whatever they want.

Developing applications that run on the GPU is a challenging task that demands a deep understanding of computer architecture and parallel programming, knowledge that I currently lack. While there are numerous libraries available that can simplify the process, the complexity of GPU programming remains daunting. Consequently, the application I created for this project runs entirely on the CPU. Translating this application to run on the GPU using CUDA[15] could be a valuable direction for future work on this project.

Running the application on an eight-core CPU yields suboptimal performance, with an average frame rate of just 15 frames per second (fps) at a resolution of 240 by 135 pixels and with only three objects in the scene. This performance limitation has prevented me from implementing support for rendering meshes into the scene

[14] If you, reader, are reading this printed on paper, which you are not meant to, this is the link to my github profile: https://github.com/trmaa which you can copy manually on any browser.
[15] C++ based programming language made to run functions directly on NVIDIA GPUs.

(although they can be imported), as meshes typically consist of thousands of individual shapes. Although there are optimization techniques discussed in section 2.5 of the project, even reducing the number of polygons to a few hundred does not achieve satisfactory performance.

However, eventually I did not manage to achieve non metallic[16] materials nor many polygon optimization techniques.

Despite these challenges, the project lays a solid foundation for further development, and transitioning to GPU-based computation represents a promising future direction for improving both performance and scalability.

3.1. Set up the app

3.1.1. Compiler

This application was developed on an Ubuntu 22 x86 operating system (Linux) with fish (a bash based terminal with autocompletions), utilising the DWM window manager, and Neo Vim as main text editor. The code is written in C++, and I created a custom tool to streamline the use of the g++ compiler and the GitHub version control system.

The manager file I developed allows users to execute a variety of commands that automate repetitive tasks, such as compiling the code, committing changes to the repository, or initializing a new project. This tool simplifies the development process, making it more efficient and manageable.

[16] Imagine it as a shiny quartz surface.

```
#!/bin/bash

error() {
    msg="$1"
    echo -e "\e[31m$msg\e[0m"
}

commit() {
    msg="$1"
    git add .
    git commit -m "$msg"
}

#---------------------------------------------- end of Libs

EXECUTABLE="./bin/app"
SRC="./src/*.cpp"
INCLUDE="./include/"
LIB="./lib/"
FLAGS="-lsfml-graphics -lsfml-window -lsfml-system"

run() {
    if [ ! -f "$EXECUTABLE" ]; then
        error "Error: $EXECUTABLE not found."
        return
    fi
    $EXECUTABLE
}

clean() {
    if [ ! -f "$EXECUTABLE" ]; then
        return
    fi
    rm $EXECUTABLE
}

build() {
    g++ $SRC -o $EXECUTABLE -I$INCLUDE -I$LIB -L$LIB $FLAGS
}

init() {
    find "./" -mindepth 1 ! -name "manager.sh" -exec rm -rf {} +
    mkdir bin
    mkdir src
    mkdir include
    mkdir lib
    touch src/main.cpp
    echo "#include<iostream>" > src/main.cpp
    echo 'int main() {std::cout<<"hola :)";}' >> src/main.cpp
    git init
    git branch -M main
    git add .
    git commit -m "first :)"
```

```
54 }
55
56 main() {
57     for callback in "$@"; do
58         if declare -f "$callback" > /dev/null; then
59             "$callback"
60         else
61             error "Error: Function '$callback' not found."
62         fi
63     done
64 }
65
66 main "$@"
```

Code 19: Code from the manager bash file

3.1.2. File structure

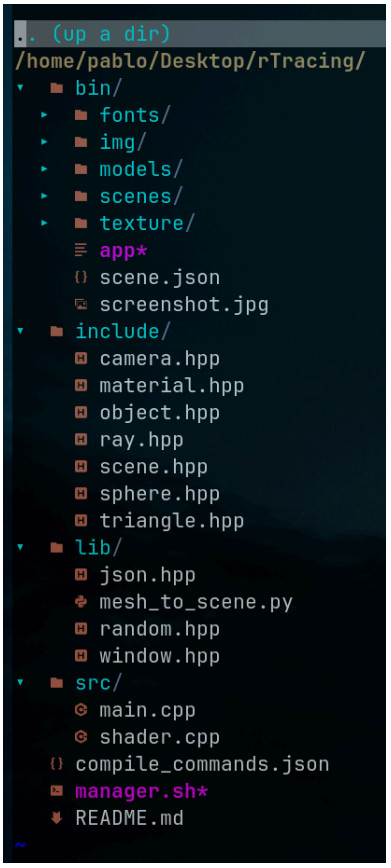


Fig. 40: Image of the files contained at the project

In the system architecture of the project, there are four principal directories each serving distinct organisational purposes:

Bin Directory: This directory is designated for storing binary executables, images, fonts, models, scenes, and other data files that are integral to the operation of the software application. These files are not intended for manual modification and serve as essential resources utilised during program execution.

Include Directory: The Include directory houses header files that provide the compiler with necessary class and function definitions. These headers serve the critical role of predefining the interfaces that will be utilised by the program before their actual execution.

Lib Directory: Within the Lib directory, there are header files that are unrelated to ray tracing and are interchangeable, such as those responsible for window display or random number generation. Additionally, this directory contains scripts that function as essential tools for the project, enhancing its development environment and operational capabilities.

Src Directory: The Src directory is dedicated to storing source code files that contain the actual implementations and definitions of functions utilised within the project. These files provide the detailed programming logic and functionality that the compiler translates into executable binaries.

Outside of these directories, there exists a compiler file, crucial for the final compilation process of the project. This file plays a pivotal role in orchestrating the

integration of the source code, header files, and external libraries into the executable form of the software application.

3.1.3. External libraries

3.1.3.1. Interface

Many applications need to display graphical contents at the screen using windows. There are lods of libraries[17] for C++ that allows the programmer to do so with different features. This project uses SFML though, which gives loads of useful classes to use such as: sf::RenderWindow, sf::Color, sf::Sprite, sf::Vector2, and so on.

Setting up SFML on windows turns out to be very painful even to experienced users, however on linux it is super simple. Let's resume a little bit of the official website's tutorial:

1. Install the library using the main debian[18] package manager with the command: "sudo apt install libsfml-dev". This will automatically create a path and let the clang compiler know where this library exists.

2. Compile using g++ and the SFML flags with the command: "g++ ./src/*.cpp -o ./bin/sfml_app -I./include/*.hpp -lsfml-graphics -lsfml-window -lsfml-system".

After this very simple two steps, the process may be completed.

At this project the window is used the following way: there is an image called buffer which is printed to the window on each frame, each pixel of this image is modified

[17] Pieces of code designed to plug into many projects and still work with little implementation.
[18] Linux distribution mother of Ubuntu and Linux Mint. It can install things with files of .deb

each frame with a *fragment shader*[19] that takes information from the project to determine the colour.

Also, it applies antialiasing to the buffer with the method from point 2.6.

```
#ifndef WINDOW_HPP
#define WINDOW_HPP

#define ANTIALIASING 0
#define MAP 0

#include "camera.hpp"
#include "scene.hpp"
#include <SFML/Graphics/RenderWindow.hpp>
#include <SFML/Graphics.hpp>
#include <SFML/Graphics/Color.hpp>
#include <SFML/Window/Keyboard.hpp>
#include <cmath>
#include <glm/ext/vector_int2.hpp>
#include <glm/geometric.hpp>
#include <glm/glm.hpp>
#include <string>
#include <vector>
#include <algorithm>
#include <execution>

sf::Color shader(int& x, int& y, glm::ivec2& buff_v, Camera& cam, Scene& scn, sf::Color& lastCol);

class Window: public sf::RenderWindow {
private:
    glm::ivec2 m_viewport;
    sf::Image m_buffer;
    int m_frames;
    std::vector<glm::vec3> m_acumulation;

    sf::Font m_font;
    sf::Text m_fpsText;
    uint16_t m_fps;
    uint8_t m_fpsLimit = 45;

    sf::Texture m_texture;
    sf::Sprite m_sprite;

    std::vector<int> x_values;
    std::vector<int> y_values;

public:
    const sf::Image& buffer() { return this->m_buffer; }

    Window(const int& w, const int& h, std::string text)
        : m_viewport(w, h), m_frames(0), m_acumulation(w * h, glm::vec3(0.f, 0.f, 0.f)), x_values(w,0), y_values(h,0) {
        this->create(sf::VideoMode(w, h), text);
        this->m_buffer.create(w, h);

        if (!this->m_font.loadFromFile("./bin/fonts/pixelmix.ttf")) {
            std::cerr << ":) no font" << std::endl;
        }

        this->m_fpsText.setFont(this->m_font);
        this->m_fpsText.setCharacterSize(24);
        this->m_fpsText.setFillColor(sf::Color(0xff00ffff));
        this->m_fpsText.setPosition(10.f, 10.f);

        this->setFramerateLimit(this->m_fpsLimit);

        std::iota(y_values.begin(), y_values.end(), 0);
        std::iota(x_values.begin(), x_values.end(), 0);
    }
    ~Window() = default;

public:
    void repaint(float& dt, Camera& cam, Scene& scn, sf::Event& ev) {
        this->m_fps = static_cast<uint16_t>(1.f / dt);
        this->m_fpsText.setString("FPS: " + std::to_string(this->m_fps));

        glm::ivec2 buff_v(this->m_buffer.getSize().x, this->m_buffer.getSize().y);

        bool resetAccumulation = false;
        if (ev.type == sf::Event::KeyPressed) {
            resetAccumulation = true;
            if (ev.key.code == sf::Keyboard::Tab) {
                this->m_buffer.saveToFile("./bin/screenshot.jpg");
            }
        }

        if (resetAccumulation) {
            this->m_frames = 0;
            this->m_acumulation = std::vector<glm::vec3>(buff_v.x * buff_v.y, glm::vec3(0.f, 0.f, 0.f));
        }
```

[19] It does the work of a shader, but it runs on the CPU so it can't be called shader.

```
85          this->m_frames += 1;
86
87          sf::Color lastCol;
88          std::for_each(std::execution::par, y_values.begin(), y_values.end(), [&](int y) {
89              std::for_each(std::execution::par, x_values.begin(), x_values.end(), [&](int x) {
90                  const int index = buff_v.x * y + x;
91                  sf::Color col = shader(x, y, buff_v, cam, scn, lastCol);
92                  sf::Color fcolor;
93
94                  //acumulation
95                  this->m_acumulation[index] += glm::vec3(col.r, col.g, col.b);
96
97                  fcolor = sf::Color(
98                          this->m_acumulation[index].r/this->m_frames,
99                          this->m_acumulation[index].g/this->m_frames,
100                         this->m_acumulation[index].b/this->m_frames);
101 #if ANTIALIASING
102                 this->m_buffer.setPixel(x, y, fcolor);
103
104                 // antialiasing
105                 if (x < 1 || x > buff_v.x-1 || y < 1 || y > buff_v.y-1) {
106                     return;
107                 }
108
109                 float r = 0.f, g = 0.f, b = 0.f;
110                 for (int offx = -1; offx <= 1; offx += 1) {
111                     for (int offy = -1; offy <= 1; offy += 1) {
112                         if(offx == 0 && offy == 0) {
113                             r += this->m_buffer.getPixel(x + offx, y + offy).r * 2;
114                             g += this->m_buffer.getPixel(x + offx, y + offy).g * 2;
115                             b += this->m_buffer.getPixel(x + offx, y + offy).b * 2;
116                         } else {
117                             r += this->m_buffer.getPixel(x + offx, y + offy).r;
118                             g += this->m_buffer.getPixel(x + offx, y + offy).g;
119                             b += this->m_buffer.getPixel(x + offx, y + offy).b;
120                         }
121                     }
122                 }
123                 r /= 10.0f;
124                 g /= 10.0f;
125                 b /= 10.0f;
126                 fcolor = sf::Color(static_cast<sf::Uint8>(r),
127                                    static_cast<sf::Uint8>(g),
128                                    static_cast<sf::Uint8>(b));
129 #endif
130                 this->m_buffer.setPixel(x, y, resetAccumulation ? col : fcolor);
131             });
132         });
133
134         this->m_texture.loadFromImage(this->m_buffer);
135         this->m_sprite.setTexture(this->m_texture);
136         float scale = static_cast<float>(this->getSize().x) / buff_v.x;
137         this->m_sprite.setScale(scale, scale);
138
139         this->clear();
140
141         this->draw(this->m_sprite);
142         this->draw(this->m_fpsText);
143
144 #if MAP
145         sf::CircleShape camera;
146         camera.setPosition(0.1f*cam.position().x+this->getSize().x/2,
147                 0.1f*cam.position().z+this->getSize().y/2);
148         camera.setFillColor(sf::Color(255, 255, 0));
149         camera.setRadius(10);
150         this->draw(camera);
151
152         for (int i = 0; i < scn.triangle().size()+scn.sphere().size()-1; i++) {
153             sf::CircleShape plane;
154             plane.setPosition(0.1f*scn.object(i).center.x+this->getSize().x/2,
155                     0.1f*scn.object(i).center.z+this->getSize().y/2);
156             plane.setFillColor(sf::Color(255,0,255));
157             plane.setRadius(10);
158
159             this->draw(plane);
160         }
161 #endif
162
163         this->display();
164
165         if (ev.key.code == sf::Keyboard::Tab) {
166             this->m_buffer.saveToFile("./bin/screenshot.jpg");
167         }
168     }
169 };
170
171 #endif
```

Code 20: Window's class code

3.1.3.2. Maths

To create random values without needing to use a seed, this project uses the standard random library from clang. It contains a function called “rand()” which returns a random int which can be used as seed. Then, I manually created a function that uses the seed to create a random float in a range from 0 to 1.

```
1 #ifndef RANDOM_HPP
2 #define RANDOM_HPP
3
4 #include <cstdlib>
5
6 inline float random_float() {
7    return rand()/(RAND_MAX+1.f);
8 }
9
10 #endif
```

Code 21: Random_float function implementation

Most languages don't have libraries to manage vectors: when I worked with C# I had to create all the dot and cross product functions alongside with the different type of vector classes. However, C++ has GLM, an amazing library for vectors which has saved me a lot of time with functions like: length, dot, cross, normalize…

3.1.3.3. File management

The project needed a way to store on memory scenes without having to recompile or write new code. For this reason I decided to use .json file structure to save and load scene contents. This project uses Niels Lhmann’s library for json.

3.1.4. Flow chart

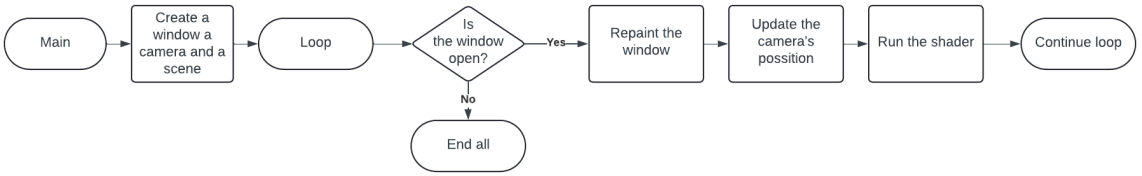


Fig. 41: Main file's abstraction of all the processes that are made each frame.

There is a chart that explains the shader at point 2.2.

3.2. Main component

The program commences by initialising the dimensions of the window and proceeds to instantiate a scene, a window, and a camera using constructors that will be elaborated upon later. The main function establishes an infinite loop which terminates upon the closure of the window. Within each iteration of the loop, the window is refreshed, and the camera updates its position.

```
#include <SFML/System.hpp>
#include <SFML/Window.hpp>
#include <cstdlib>
#include <iostream>
#include <string>
#include "window.hpp"
#include "camera.hpp"
#include "scene.hpp"

static int w = static_cast<int>(16*15);
static int h = static_cast<int>(9*15);
Scene scn("./bin/scene.json", w, h);
Window win(w, h, "rtx");
Camera cam(w, h);

void loop(float& dt, sf::Event& ev) {
    win.repaint(dt, cam, scn, ev);
    cam.move(dt, ev);

```

```
20      if (ev.key.code == sf::Keyboard::Tab) {
21          scn = Scene("./bin/scene.json", w, h);
22      }
23  }
24
25  int main(int argc, char* argv[]) {
26      std::cout << "starting..." << std::endl;
27
28      sf::Clock clck;
29      sf::Event ev;
30      sf::Time elapsed;
31      float dt;
32      while (win.display().isOpen()) {
33          while (win.display().pollEvent(ev)) {
34              if (ev.type == sf::Event::Closed) {
35                  win.display().close();
36              } else if (ev.type == sf::Event::Resized) {
37                  sf::FloatRect visibleArea(0, 0, ev.size.width, ev.size.height);
38                  win.display().setView(sf::View(visibleArea));
39              }
40          }
41          elapsed = clck.restart();
42          dt = elapsed.asSeconds();
43
44          loop(dt, ev);
45      }
46
47      std::system("clear");
48  }
```

Code 22: Main file's code

3.3. Scene component

The following code defines the *Scene class*, which is designed to store information about the sky colour, a list of spheres, and a list of triangles.

It includes a method that accepts an integer identifier as an argument and returns either a triangle or a sphere under a common *object* type.

Additionally, the class provides a function that takes the path to a .json file as an argument and returns its content in an nlohmann::json object.

Within the constructor, the class utilises its static function *read_json* to retrieve all the scene information as a single object. It then iterates through this data, populating the lists of triangles and spheres with the relevant information. Consequently, each instance of a Scene is fully configured and ready to use immediately upon creation with a specified scene file path.

```
#ifndef SCENE_HPP
#define SCENE_HPP

#include "json.hpp"
#include "object.hpp"
#include "triangle.hpp"
#include "sphere.hpp"
#include <fstream>
#include <future>
#include <glm/detail/qualifier.hpp>
#include <glm/ext/vector_float3.hpp>
#include <iostream>
#include <string>
#include <glm/glm.hpp>
#include <vector>

class Scene {
private:
    glm::vec3 m_sky_color;
    std::vector<Sphere> m_sphere;
    std::vector<Triangle> m_triangle;

private:
    static nl::json read_json(const std::string& fpath) {
        std::ifstream file(fpath);
        if (!file.is_open()) {
            std::cerr<<"Json corrupt"<<std::endl;
            return "";
        }
        nl::json data;
        file >> data;
        return data;
    }

public:
    const std::vector<Sphere>& sphere() const { return this->m_sphere; }
    const Sphere& sphere(const int index) const { return this->m_sphere[index]; }

    const std::vector<Triangle>& triangle() const { return this->m_triangle; }
    const Triangle& triangle(const int index) const { return this->m_triangle[index]; }

    const Object& object(const int index) const {
        if (index < this->m_triangle.size()) {
            return this->m_triangle[index];
        } else if (index < this->m_triangle.size() + this->m_sphere.size()) {
            return this->m_sphere[index - this->m_triangle.size()];
        } else {
            throw std::out_of_range("Index out of bounds in Scene::object()");
        }
    }

    glm::vec3& sky_color() { return this->m_sky_color; }

    Scene(const std::string& fpath, int& w, int& h) {
        std::future<nl::json> raw = std::async(std::launch::async, Scene::read_json, fpath);
        nl::json data = raw.get();
        if (data.contains("res")) {
            w = data["res"][0];
            h = data["res"][1];
        }
```

```
61          if (!data.contains("sphere") || !data.contains("sky") || !data.contains("triangle"))
62              return;
63          for (const auto& obj : data["sphere"]) {
64              glm::vec3 center(obj["center"][0], obj["center"][1], obj["center"][2]);
65              float radius = obj["radius"];
66              glm::vec3 color(obj["color"][0], obj["color"][1], obj["color"][2]);
67              float emission = obj["emission"];
68              float roughness = static_cast<float>(obj["roughness"]);
69              std::string path = obj["texture"];
70              this->m_sphere.push_back(Sphere(center, radius, color, emission, roughness, path));
71          }
72          for (const auto& obj : data["triangle"]) {
73              glm::vec3 center(obj["center"][0], obj["center"][1], obj["center"][2]);
74              glm::vec3 sides[2];
75              sides[0] = glm::vec3(obj["s0"][0], obj["s0"][1], obj["s0"][2]);
76              sides[1] = glm::vec3(obj["s1"][0], obj["s1"][1], obj["s1"][2]);
77              glm::vec3 color(obj["color"][0], obj["color"][1], obj["color"][2]);
78              float emission = obj["emission"];
79              float roughness = static_cast<float>(obj["roughness"]);
80              std::string path = obj["texture"];
81              this->m_triangle.push_back(Triangle(center, sides, color, emission, roughness, path));
82          }
83          this->m_sky_color = glm::vec3(data["sky"][0],data["sky"][1],data["sky"][2]);
84      }
85      ~Scene() = default;
86  };
87
88  #endif
```

Code 23: Scene's class code

3.3.1. Object component

Sins C++ is a strongly typed language, and I needed a single stroke in memory which contained both: triangles, and spheres (which are different classes); The program needed a common class which inherited both.

```
 1  #ifndef OBJECT_HPP
 2  #define OBJECT_HPP
 3
 4  #include "material.hpp"
 5  #include "ray.hpp"
 6  #include <glm/ext/vector_float3.hpp>
 7  #include <string>
 8
 9  struct Object {
10      Material material;
11      glm::vec3 center;
12      float radius;
13      glm::vec3 sides[2];
14      glm::vec3 normal;
15
16      Object(const glm::vec3& cen, float rad, glm::vec3 c, float e, float r, const std::string &pa)
17          : material(c,e,r,pa) {
18          radius = rad;
19          center = cen;
20      }
21
22      Object(const glm::vec3& cen ,const glm::vec3 verts[2], glm::vec3 c, float e, float r, const std::string &pa)
23          : material(c,e,r,pa), center(cen) {
24          this->normal = glm::normalize(glm::cross(verts[0], verts[1]));
25          std::copy(verts, verts + 2, sides);
26      }
27
28      virtual ~Object() = default;
```

```
29
30      virtual const std::string type() const {
31          return "Object";
32      }
33
34      virtual const float checkCollision(const Ray& ray) const {
35          return 0.f;
36      }
37  };
38
39  #endif
```

Code 24: Object's class code

This class has two constructors, one which defines the triangle attributes and other which defines the sphere attributes. Allso, it has redefineable functions which stand for collision detection and type handling.

All of these aspects were discussed earlier at points 2.3 and 2.4.

3.3.2. Triangle component

Triangles extend from the object class mentioned earlier, overriding the collision function. This function accepts a ray as an argument and calculates the precise distance to the intersection point, providing the exact step where the intersection occurred. This capability allows the program to accurately determine the location of collisions within the scene.

The code below illustrates it:

```
 1 #ifndef TRIANGLE_HPP
 2 #define TRIANGLE_HPP
 3
 4 #include "object.hpp"
 5 #include "ray.hpp"
 6 #include <glm/ext/vector_float3.hpp>
 7 #include <glm/geometric.hpp>
 8 #include <glm/glm.hpp>
 9 #include <string>
10
11 struct Triangle: public Object{
12     Triangle(glm::vec3 ce, glm::vec3 v[2], glm::vec3 c, float e, float r, std::string pa)
13         : Object(ce, v, c, e, r, pa) {}
14     ~Triangle() = default;
15
```

```
16      const float checkCollision(const Ray& ray) const override {
17          float denominator = glm::dot(ray.direction, this->normal);
18
19          if (glm::abs(denominator) > 1e-6) {
20              glm::vec3 diff = this->center - ray.origin;
21              float t = glm::dot(diff, this->normal) / denominator;
22
23              glm::vec3 hitp = ray.f(t);
24
25              glm::vec3 v0 = this->sides[0];
26              glm::vec3 v1 = this->sides[1];
27              glm::vec3 v2 = hitp - this->center;
28
29              float d00 = glm::dot(v0, v0);
30              float d01 = glm::dot(v0, v1);
31              float d11 = glm::dot(v1, v1);
32              float d20 = glm::dot(v2, v0);
33              float d21 = glm::dot(v2, v1);
34              float denom = d00 * d11 - d01 * d01;
35
36              float v = (d11 * d20 - d01 * d21) / denom;
37              float w = (d00 * d21 - d01 * d20) / denom;
38              float u = 1.0f - v - w;
39
40              bool cond = (u >= 0.f) && (v >= 0.f) && (w >= 0.f);
41              if (cond && t >= 0.f) {
42                  return t;
43              }
44          }
45
46          return -1.f;
47      }
48
49      const std::string type() const override {
50          return "Triangle";
51      }
52  };
53
54  #endif
```

Code 25: Triangle's class code

3.3.3. Sphere component

Spheres extend from the object class mentioned earlier as well, overriding the collision function. This function, again, accepts a ray as an argument and calculates the precise distance to the intersection point, providing the exact step where the intersection occurred. This capability allows the program to accurately determine the location of collisions within the¡ scene.

The code below illustrates it:

```
#ifndef SPHERE_HPP
#define SPHERE_HPP

#include <cmath>
#include <glm/ext/vector_float3.hpp>
#include <glm/glm.hpp>
#include "ray.hpp"
#include "object.hpp"

struct Sphere: public Object {
    Sphere(const glm::vec3& cen, float rad, const glm::vec3& c, float e, float ro, const std::string&
p)
        : Object(cen, rad, c, e, ro, p) {}
    ~Sphere() = default;

    const float checkCollision(const Ray& ray) const override {
        glm::vec3 oc = ray.origin - this->center;
        float a = glm::dot(ray.direction, ray.direction);
        float b = 2.0f * glm::dot(oc, ray.direction);
        float c = glm::dot(oc, oc) - this->radius * this->radius;
        float discriminant = b * b - 4 * a * c;

        if (discriminant < 0) {
            return -1.0f;
        }

        float sqrtDiscriminant = std::sqrt(discriminant);
        float t1 = (-b - sqrtDiscriminant) / (2.0f * a);
        float t2 = (-b + sqrtDiscriminant) / (2.0f * a);

        if (t1 > 0.0f && t2 > 0.0f) {
            return std::min(t1, t2);
        } else if (t1 > 0.0f) {
            return t1;
        } else if (t2 > 0.0f) {
            return t2;
        } else {
            return -1.0f;
        }
    }

    const std::string type() const override {
        return "Sphere";
    }
};

#endif
```

Code 26: Sphere's class code

3.3.4. Material component

The material component contains values that represent various properties of an object, such as colour, light emission, surface roughness, and texture (a flat image that visually represents the object's appearance). Each material instance is associated with a specific object in the scene, providing the fragment shader with the necessary information to render the object accurately.

The function *txtr* takes a normal and a hit point as arguments and returns the corresponding colour from the material's texture. This allows the shader to determine which part of the texture should be rendered based on the specific point on the object that was hit. The way that this works is explained at points 2.3.4 and 2.4.4.

```
#ifndef MATERIAL_HPP
#define MATERIAL_HPP

#include <SFML/Graphics.hpp>
#include <SFML/Graphics/Color.hpp>
#include <glm/detail/qualifier.hpp>
#include <glm/ext/vector_float3.hpp>
#include <glm/geometric.hpp>
#include <glm/glm.hpp>
#include <cmath>

struct Material {
    glm::vec3 color;
    float emission;
    float roughness;
    sf::Image texture;

    Material(glm::vec3 c, float e, float r, const std::string& txtrPath)
        : color(c), emission(e), roughness(r) {
        if (!texture.loadFromFile(txtrPath)) {
            texture.create(1,1);
            texture.setPixel(0, 0, sf::Color(c.r, c.g, c.b));
        }
    }

    ~Material() = default;

    const sf::Color col() const {
        return sf::Color(static_cast<sf::Uint8>(this->color.r * 255),
                static_cast<sf::Uint8>(this->color.g * 255),
                static_cast<sf::Uint8>(this->color.b * 255));
    }

    const sf::Color txtr(const glm::vec3 normal) const {
        float theta = std::acos(-normal.y);
        float phi = std::atan2(normal.z, normal.x);

        float u = (phi + M_PI) / (2 * M_PI);
        float v = theta / M_PI;

        int texWidth = texture.getSize().x;
        int texHeight = texture.getSize().y;
        int x = static_cast<int>(u * texWidth) % texWidth;
        int y = static_cast<int>((texHeight - v) * texHeight) % texHeight;

        sf::Color color = texture.getPixel(x, y);
        return sf::Color(color.r*this->color.r/255.f,
                color.g*this->color.g/255.f, color.b*this->color.b/255.f);
    }
```

```
50
51      sf::Color txtr(glm::vec3 hitp, glm::vec3 center, glm::vec3 normal) const {
52          float planeWidth = 100.0f;
53          float planeHeight = 100.0f;
54
55          glm::vec3 tangent, bitangent;
56          if (fabs(normal.x) > fabs(normal.y)) {
57              tangent = glm::vec3(normal.z, 0, -normal.x);
58          } else {
59              tangent = glm::vec3(0, -normal.z, normal.y);
60          }
61          tangent = glm::normalize(tangent);
62          bitangent = glm::normalize(glm::cross(normal, tangent));
63
64          glm::vec3 localHitPoint = hitp - center;
65          float u = glm::dot(localHitPoint, tangent) / planeWidth;
66          float v = glm::dot(localHitPoint, bitangent) / planeHeight;
67
68          u = fmod(u, 1.0f);
69          v = fmod(v, 1.0f);
70          if (u < 0) u += 1.0f;
71          if (v < 0) v += 1.0f;
72
73          unsigned int texWidth = texture.getSize().x;
74          unsigned int texHeight = texture.getSize().y;
75          unsigned int texX = static_cast<unsigned int>(u * texWidth);
76          unsigned int texY = static_cast<unsigned int>(v * texHeight);
77
78          sf::Color color = texture.getPixel(texX, texY);
79          return sf::Color(color.r*this->color.r/255.f,
80                  color.g*this->color.g/255.f, color.b*this->color.b/255.f);
81      }
82  };
83
84  #endif
```

Code 27: Sphere's class code

3.3.5. Save file component

When scenes are relatively simple, such as a few spheres positioned over a flat surface, they can be manually defined without much difficulty. However, converting complex meshes into triangles is a challenging task that necessitates automation, as detailed in point 2.5. This project addresses this challenge by employing a Python script, as shown in that section, to automate the process of mesh-to-triangle conversion.

3.4. Camera component

Three-dimensional cameras use various techniques to convert 3D scene data into 2D representations on a screen. Among these techniques, projection matrices play a crucial role in 3D rendering methods other than ray tracing, such as rasterization.

A projection matrix transforms 3D coordinates into 2D screen coordinates, effectively mapping a 3D scene onto a 2D plane. One common example is the orthographic projection matrix, which simplifies this process by eliminating the depth (Z) component of each point in the scene. The orthographic projection matrix is represented as follows:

$$\begin{matrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{matrix}$$

When a 3D coordinate is multiplied by this matrix, the Z values are effectively cancelled, projecting the 3D point onto the XY plane. This type of projection is akin to viewing a scene from an infinite distance, where depth does not influence the final image, resulting in a parallel projection without perspective distortion. It provides a straightforward and computationally efficient method for rendering scenes where the relative positioning of objects along the Z-axis is not a factor in the visualization.

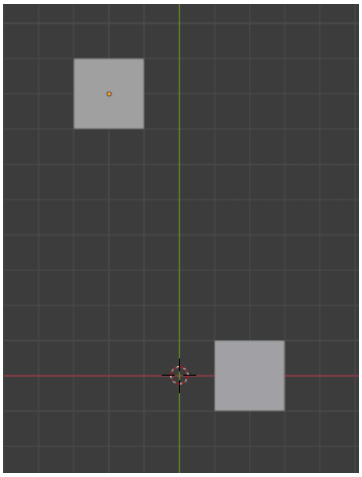


Fig. 42: Orthogonal scene seen from the top

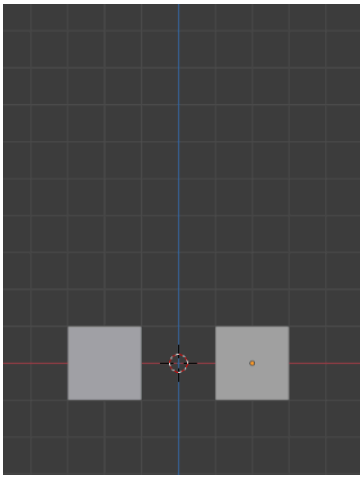


Fig. 43: Orthogonal scene seen from the front

As shown in the images above, using this orthographic projection matrix does not account for perspective. This means that depth information is not conveyed, making it impossible to determine the relative distances of objects from the camera. The orthographic projection simplifies the scene by removing the Z component, resulting in a view that lacks depth cues and perspective distortion.

To solve this, there is the matrix can be changed this way:

$$\begin{matrix} \frac{1}{z} & 0 & 0 \\ 0 & \frac{1}{z} & 0 \\ 0 & 0 & 0 \end{matrix}$$

Using the Z value to approach the points to the centre of the screen provides a nice perspective effect:

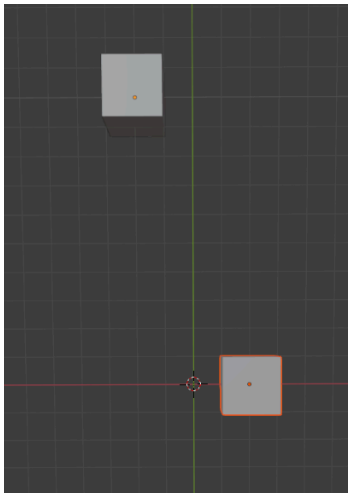


Fig. 44: Scene seen from the top

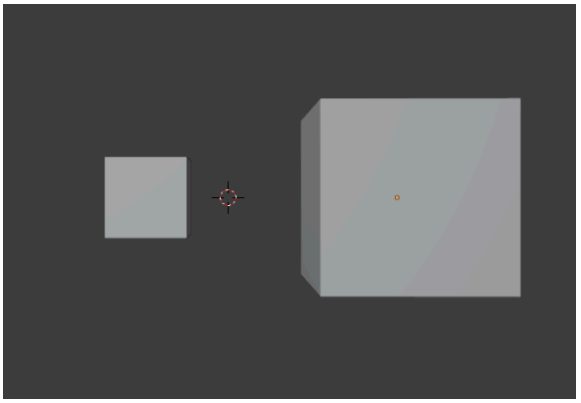


Fig. 45: Scene seen from the front

Then the colour is applied taking into account the normal of the surfaces and the light direction.

This method offers good performance, but it falls short in terms of color accuracy compared to ray tracing. Ray tracing, while computationally more intensive, provides superior results in terms of lighting and color representation, as it simulates the way light interacts with objects in a scene.

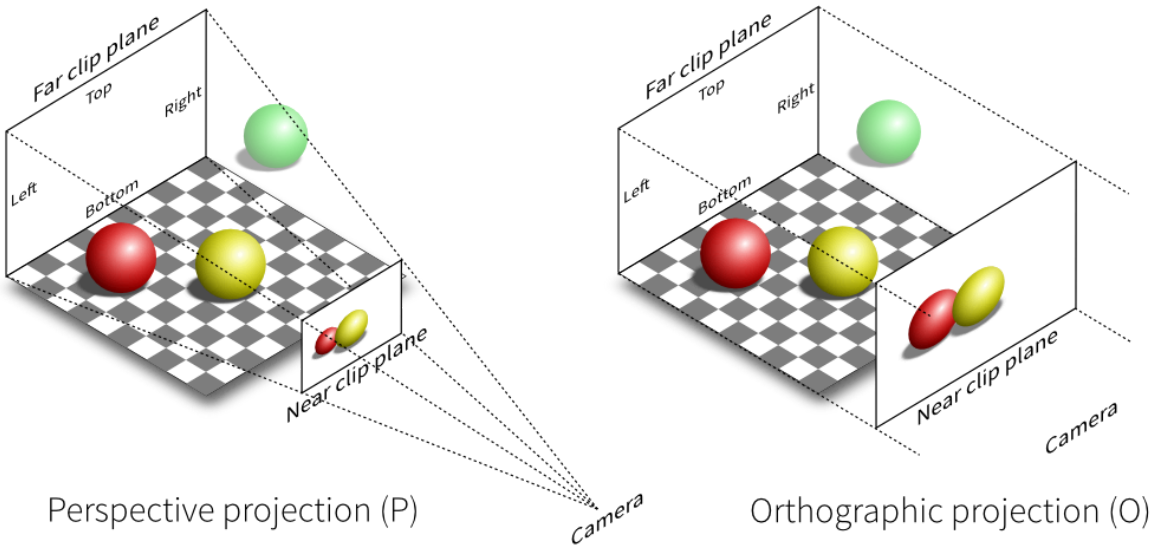


Fig. 46: Graphical difference between the two matrixes by https://dev.opencascade.org/doc/overview/html/occt_user_guides__visualization.html

Fortunately, ray tracing does not require direct manipulation of projection matrices. However, understanding the principles of 3D projection and how cameras work can be extremely beneficial in designing effective ray tracing algorithms. Knowledge of these concepts helps in configuring camera parameters and scene setup, leading to more realistic and visually appealing renderings in ray tracing.

The camera in a 3D rendering system has position and direction parameters, which can be adjusted using the move function. These parameters are crucial in defining the ray direction for a specific pixel on the screen through the cast function.

In the cast function, the process begins by defining the ray's direction if the camera were oriented at (0, 0, 0). This initial ray direction is given by the vector (pixelX, pixelY, 200), where 200 is an arbitrary depth value. From this, the idle ray direction is calculated.

The next step involves computing the new ray direction by applying the camera's current angle to the idle ray. This is done using trigonometric functions to adjust the ray direction according to the camera's orientation. The calculations for the new ray direction are as follows:

$$x = Idle_z \cdot cos(angle_x) \cdot cos(angle_y),$$
$$y = Idle_z \cdot sin(angle_x),$$
$$z = Idle_z \cdot cos(angle_x) \cdot sin(angle_y),$$

```
1  #ifndef CAMERA_HPP
2  #define CAMERA_HPP
3
4  #include <SFML/Graphics.hpp>
5  #include <SFML/Window.hpp>
6  #include <SFML/Window/Keyboard.hpp>
7  #include <cmath>
8  #include <glm/ext/vector_float3.hpp>
9  #include <glm/ext/vector_int2.hpp>
10 #include <glm/glm.hpp>
11 #include <vector>
12 #include "ray.hpp"
13
14 class Camera {
15 private:
16     glm::vec3 _position;
17     glm::vec3 _angle;
18     glm::vec3 _direction;
19
20     float _far = 200;
21
22     float _speed;
23
24 public:
25     std::vector<Ray> ray;
26     const glm::vec3& position() const { return _position; }
27     const glm::vec3& direction() {
28         this->_direction = glm::vec3(
29             -std::sin(this->_angle.x)*std::cos(this->_angle.y),
30             -std::cos(this->_angle.x),
31             std::sin(this->_angle.x)*std::sin(this->_angle.y)
32         );
33         return _direction;
34     }
35
36     Camera(const int& w, const int& h)
37         : _position(-0.0f,2.0f,-200.f), _angle(glm::vec3(0.0f, 3.14159f*0.5f,
0.0f)),
38         _direction(0.0f), _speed(100.0f), _far(400.f*w/192) {
```

```
39          for (int y = 0; y < h; y++) {
40              for (int x = 0; x < w; x++) {
41                  ray.push_back(Ray(
42                      _position,
43                      glm::vec3(x - w / 2, y - h / 2, this->_far)
44                  ));
45              }
46          }
47      }
48      ~Camera() = default;
49
50  public:
51      void cast(int& x, int& y, glm::ivec2& buff_v) {
52          int id = (x) + (y) * buff_v.x;
53
54          glm::vec3 idle((x) - buff_v.x / 2, (y) - buff_v.y / 2, this->_far);
55          glm::vec3 idleA(
56              std::atan2(idle.y, idle.z),
57              std::atan2(idle.x, idle.z),
58              0.0f
59          );
60          glm::vec3 ang = _angle + idleA;
61          glm::vec3 direction(
62              2000.0f * std::cos(ang.x) * std::cos(ang.y),
63              2000.0f * std::sin(ang.x),
64              2000.0f * std::cos(ang.x) * std::sin(ang.y)
65          );
66
67          ray[id] = Ray(_position, direction);
68      }
69
70      void move(float& dt, sf::Event& ev) {
71          float fixedSpeed = _speed * (dt);
72          if (ev.type != sf::Event::KeyPressed) {
73              return;
74          }
75          if (ev.key.code == sf::Keyboard::Up) {
76              _position.x += std::cos(_angle.y) * fixedSpeed;
77              _position.z += std::sin(_angle.y) * fixedSpeed;
78          }
79          if (ev.key.code == sf::Keyboard::Down) {
80              _position.x -= std::cos(_angle.y) * fixedSpeed;
81              _position.z -= std::sin(_angle.y) * fixedSpeed;
82          }
83          if (ev.key.code == sf::Keyboard::Left) {
84              _position.x += std::cos(_angle.y - 3.14159f / 2) * fixedSpeed;
85              _position.z += std::sin(_angle.y - 3.14159f / 2) * fixedSpeed;
86          }
87          if (ev.key.code == sf::Keyboard::Right) {
88              _position.x -= std::cos(_angle.y - 3.14159f / 2) * fixedSpeed;
89              _position.z -= std::sin(_angle.y - 3.14159f / 2) * fixedSpeed;
90          }
91          if (ev.key.code == sf::Keyboard::RControl) {
92              _position.y -= fixedSpeed;
93          }
94          if (ev.key.code == sf::Keyboard::RShift) {
95              _position.y += fixedSpeed;
96          }
97          if (ev.key.code == sf::Keyboard::LAlt) {
```

```
98              _angle.y += 0.1f;
99          }
100         if (ev.key.code == sf::Keyboard::LShift) {
101             _angle.y -= 0.1f;
102         }
103         if (ev.key.code == sf::Keyboard::LControl) {
104             _angle.x += 0.1f;
105         }
106         if (ev.key.code == 40) {
107             _angle.x -= 0.1f;
108         }
109     }
110 };
111
112 #endif
```

Code 28: Camera's class code

3.4.1. Ray component

A ray has an origin and a direction. Each point of the ray can be reached by multiplying a moment in time to the direction and adding it to the origin as explained at point 2.3.1.

```
1  #ifndef RAY_HPP
2  #define RAY_HPP
3
4  #include <glm/glm.hpp>
5
6  struct Ray {
7          glm::vec3 origin;
8          glm::vec3 direction;
9
10         Ray(glm::vec3 o, glm::vec3 d): origin(o), direction(d) {}
11         ~Ray() = default;
12
13         const glm::vec3 f(const float x) const { return this->origin + x*this->direction; }
14 };
15
16 #endif
```

Code 29: Ray's class code

3.5. Shader component

The shader is executed once for each pixel on the screen, with the purpose of determining the final color of that pixel by simulating how light interacts with objects in a 3D scene. It begins by setting the initial color of the pixel to the sky color defined in the scene, which serves as the default color if the ray cast from the camera does not intersect with any objects.

The shader then starts a loop that handles up to four bounces of the ray. This loop simulates the behavior of light bouncing off surfaces, allowing the shader to account for reflections and other complex lighting effects. In each iteration of the loop, the shader checks whether the ray intersects with any objects in the scene. The distances to each potential intersection are calculated and stored in an array called times. The shader then identifies the closest intersection by sorting these distances, ensuring that the nearest object to the camera is the one that affects the pixel color.

If the ray does not intersect with any objects on the first bounce, the shader immediately moves on to the next pixel, as there is no further interaction to compute. However, if no intersection occurs on a subsequent bounce (beyond the first), the shader returns the last colour stored, simulating the effect of the scene's environment light (like the sky) influencing the pixel color.

When a collision occurs, the shader calculates the normal vector at the point of intersection, which is essential for determining how light interacts with the surface.

If the material of the object at the intersection point is emissive, meaning it acts as a light source, the shader returns the color of the emission directly, as this light contributes directly to the pixel color without further reflections.

If the material is not emissive, the shader stores the color of the object and modifies the ray's direction based on the material's roughness. With rougher materials causing more diffusion in the reflected ray. The shader then stores the colour and uses it to calculate the next bounce.

This process repeats up to four times. This way, always peaking the last color hit, scenes are rendered realistically. This logic is explained at point 2.2.

```
#include "window.hpp"
#include "random.hpp"
#include "camera.hpp"
#include "scene.hpp"
#include "object.hpp"
#include <SFML/Graphics/Color.hpp>
#include <SFML/Window/Keyboard.hpp>
#include <algorithm>
#include <cstdio>
#include <glm/common.hpp>
#include <glm/detail/qualifier.hpp>
#include <glm/ext/vector_float3.hpp>
#include <glm/ext/vector_int2.hpp>
#include <glm/geometric.hpp>
#include <vector>

sf::Color shader(int& x, int& y, glm::ivec2& buff_v, Camera& cam, Scene& scn, sf::Color& lastCol)
{

const int index = x + y * buff_v.x;

glm::vec3 sc = scn.sky_color();
sf::Color col = sf::Color(sc.r * 255, sc.g * 255, sc.b * 255);
Ray& ray = cam.ray[index];
cam.cast(x, y, buff_v);

int bounces = 4;
for (int i = 0; i < bounces; i++) {
    float importance = static_cast<float>(bounces - i*(1)) / bounces;
    importance = importance<0?0:importance;

    std::vector<float> times;
    for (int j = 0; j < scn.sphere().size()+scn.triangle().size(); j++) {
        times.push_back(scn.object(j).checkCollision(ray));
    }
    auto t = std::min_element(times.begin(), times.end(), [](float a, float b) {
        return std::abs(a) < std::abs(b);
    }); // t is a pointer
    int id = std::distance(times.begin(), t);
    const Object& object = scn.object(id);

    if (*t < 0) {
        if (i < 1) {
            lastCol = col;
            return col;
        }
        sf::Color lc = lastCol;
        glm::vec3 nc = sc * glm::vec3(lc.r, lc.g, lc.b);
        lastCol = sf::Color(nc.r, nc.g, nc.b);
        nc *= importance;
        lastCol = sf::Color(nc.r, nc.g, nc.b);
        col = sf::Color(nc.r, nc.g, nc.b);
        return col;
    }

    glm::vec3 hitPoint = ray.f(*t);

    glm::vec3 normal;
    if (object.type() == "Triangle") {
        normal = object.normal;
    } else {
        normal = glm::normalize(object.center-hitPoint);
    }
```

```
64    if (glm::dot(ray.direction, normal) > 3.14159f/2) {
65        normal = normal*(-1.f);
66    }
67
68    if (object.material.emission > 0) {
69        sf::Color tc = object.type()=="Triangle"?object.material.txtr(ray.f(*t), object.center, normal):object.material.txtr(normal);
70        if (i < 1) {
71            glm::vec3 nc = object.material.emission*glm::vec3(tc.r, tc.g, tc.b);
72            col = sf::Color(nc.r, nc.g, nc.b);
73            lastCol = col;
74            return col;
75        }
76        sf::Color lc = lastCol;
77        glm::vec3 nc = glm::normalize(glm::vec3(lc.r, lc.g, lc.b))
78            *glm::vec3(tc.r, tc.g, tc.b)*object.material.emission;
79        col = sf::Color(nc.r, nc.g, nc.b);
80        lastCol = col;
81        return col;
82    }
83
84    sf::Color tC = object.type()=="Triangle"?object.material.txtr(ray.f(*t), object.center, normal):object.material.txtr(normal);
85    if (i < 1) {
86        lastCol = tC;
87    } else {
88        glm::vec3 lc = glm::vec3(lastCol.r, lastCol.g, lastCol.b) * glm::normalize(glm::vec3(tC.r, tC.g, tC.b));
89        lastCol = sf::Color(lc.r, lc.g, lc.b);
90    }
91
92    float bright = glm::dot(glm::normalize(normal), glm::normalize(glm::vec3(-1,-1,-1)));
93    bright = bright>1?1:bright;
94    bright = bright<0?0:bright;
95
96    glm::vec3 color = bright * glm::vec3(lastCol.r, lastCol.g, lastCol.b) * glm::normalize(glm::vec3(tC.r, tC.g, tC.b) * importance);
97    col = sf::Color(color.r, color.g, color.b);
98
99    glm::vec3 reflected_direction = ray.direction - 2.0f * glm::dot(ray.direction, normal) * normal;
100
101     glm::vec3 diffusion(0);
102     if (object.material.roughness > 0) {
103         diffusion = glm::normalize(
104                 glm::vec3(rando_float(), rando_float(), rando_float())
105                 * 2.0f - glm::vec3(1.0f)) * object.material.roughness;
106     }
107
108     ray = Ray(hitPoint + normal * 0.001f, (reflected_direction + diffusion));
109 }
110 return col;
111
112 }
```

Code 30: Shader program

4. Graphical bugs and common skill issues

Throughout my research, I realised that making and addressing mistakes is a crucial part of the learning process. Here are a few stories behind some of those errors:

4.1. From JavaScript, to Java, to C#, to C++:

When I mention that I've spent years working on this project, it doesn't mean I've been writing this document or developing the described application for that entire time. Rather, it reflects my long-standing interest in 3D rendering, which began when I first started programming. Early on, when I didn't even fully understand how computers worked, I made several ambitious attempts to build a ray tracer. I used high-level abstraction languages, including scripting languages like JavaScript, but despite my efforts, I was unsuccessful until I had developed a more advanced understanding of mathematics. Along the way, I did create other types of 3D renderers, initially in JavaScript and later rewriting them in Java (an object-oriented, statically-typed programming language similar to C#. Java doesn't compile to binary directly but instead to a .jar file, which is interpreted by the Java JDK).

My first attempt at building a ray tracer involved reusing this Java-based 3D engine, which I later rewrote in C# due to performance limitations. However, as I continued learning and improved my skills, I transitioned to C++. It became clear that my C# code was inefficient and disorganised, so I decided to rewrite everything from scratch in C++. Initially, I struggled with memory management, frequently over-allocating on the heap without proper control, leading to memory leaks and program crashes. Over time, I gained a better understanding of memory handling in C++.

I ended up developing the same ray tracer four times, each iteration better than the last due to my improving skill set. This process of continuous improvement has taught me many of the concepts discussed in this document. The images included in this document reflect my journey: some were rendered using the C# version, while others were generated by the newer C++ application. The distinction between the two is noticeable, with the C# images appearing much clunkier in comparison to the final C++ renders.

4.2. Antialiasing:

In my initial attempts to create an antialiasing algorithm, I inadvertently generated colours that exceeded the 16-bit capacity of each channel. This resulted in some unexpected but visually striking effects:

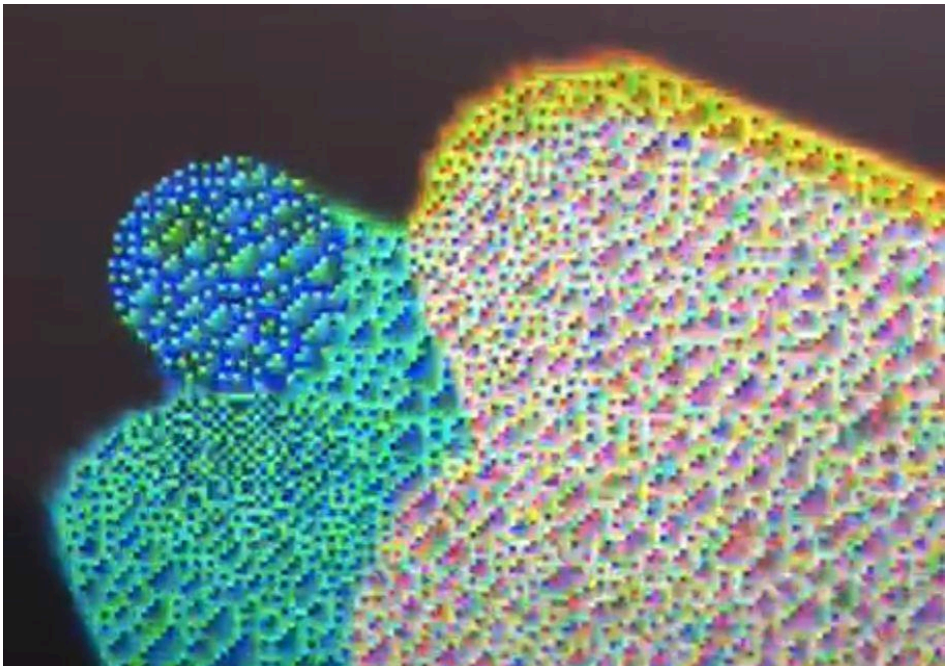


Fig. 47: Graphical bug

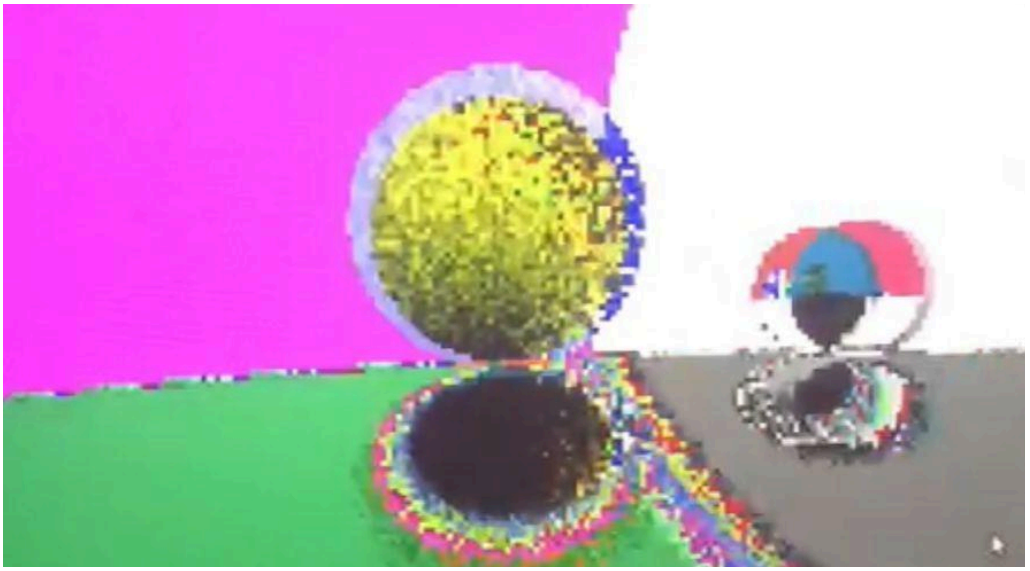


Fig. 48: Graphical bug

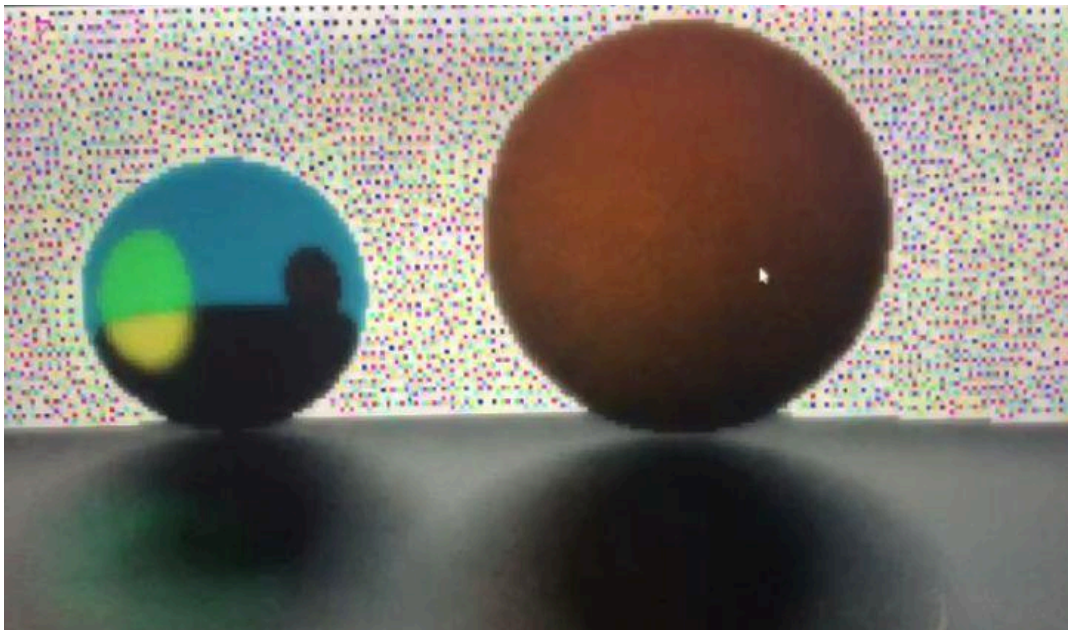


Fig. 49: Graphical bug

4.3. Casting rays:

This is what I saw when I miscalculated the ray directions:

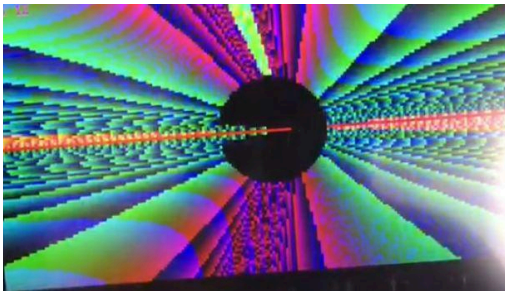


Fig. 50: Graphical bug

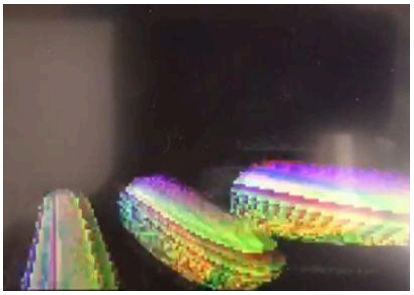


Fig. 51: Graphical bug

The maths are explained at the point 3.4.

4.4. Normals:

This is what I saw when I miscalculated the normals from the spheres:

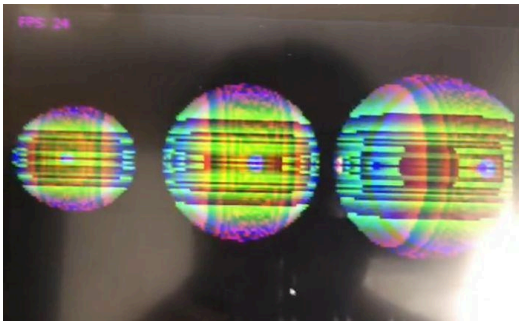


Fig. 52: Graphical bug

Fig. 53: Intended result

The maths are explained at the point 2.3.2.

4.5. From Ubuntu to Arch:

Recently, I switched to a different Linux distribution, and, for some reason, the new operating system handled parallelism differently. As a result, I was unable to compile the project on this distribution for about a week. When I finally managed to compile it, the program ran but exhibited severe graphical bugs, such as the one shown here:

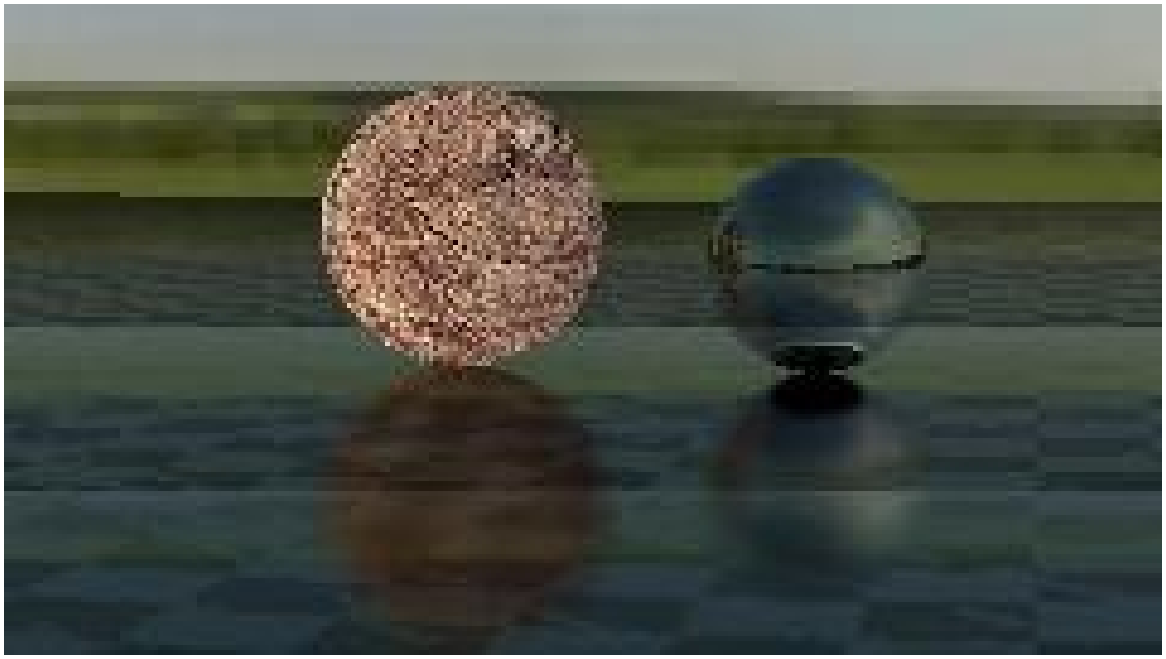


Fig. 54: Graphical bug

Finally, I gave up on standard parallelism and fixed the problem.

5. Final renders

These are some examples of renders made with the application which where never shown onto the project:

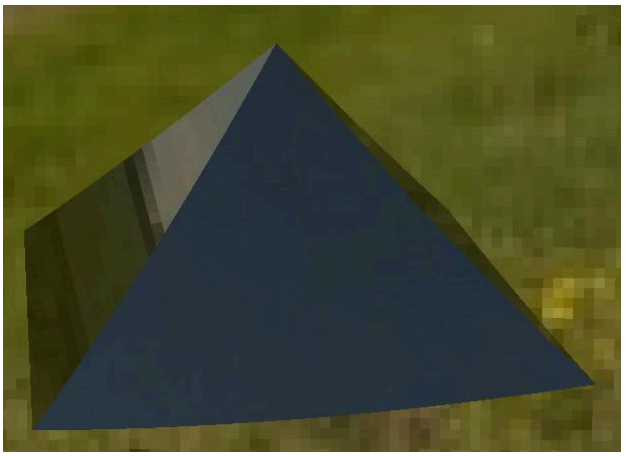


Fig. 55: Some render

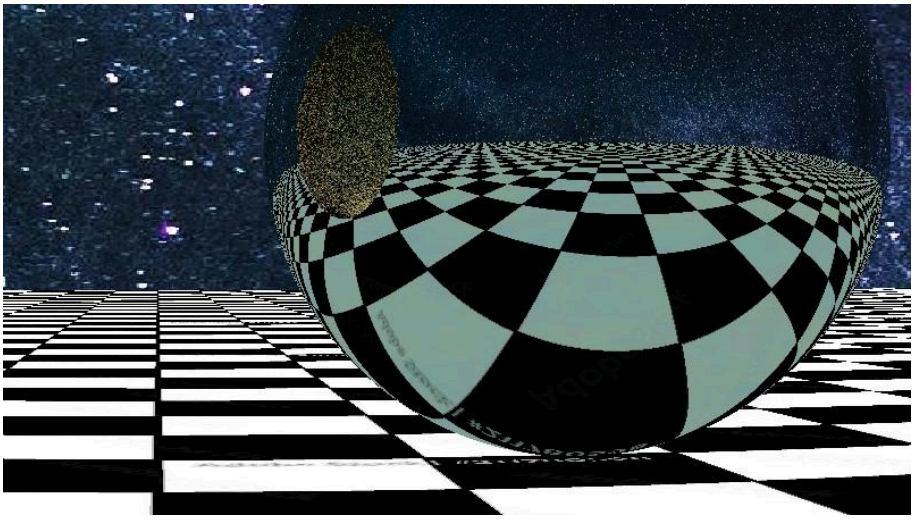


Fig. 56: Some render

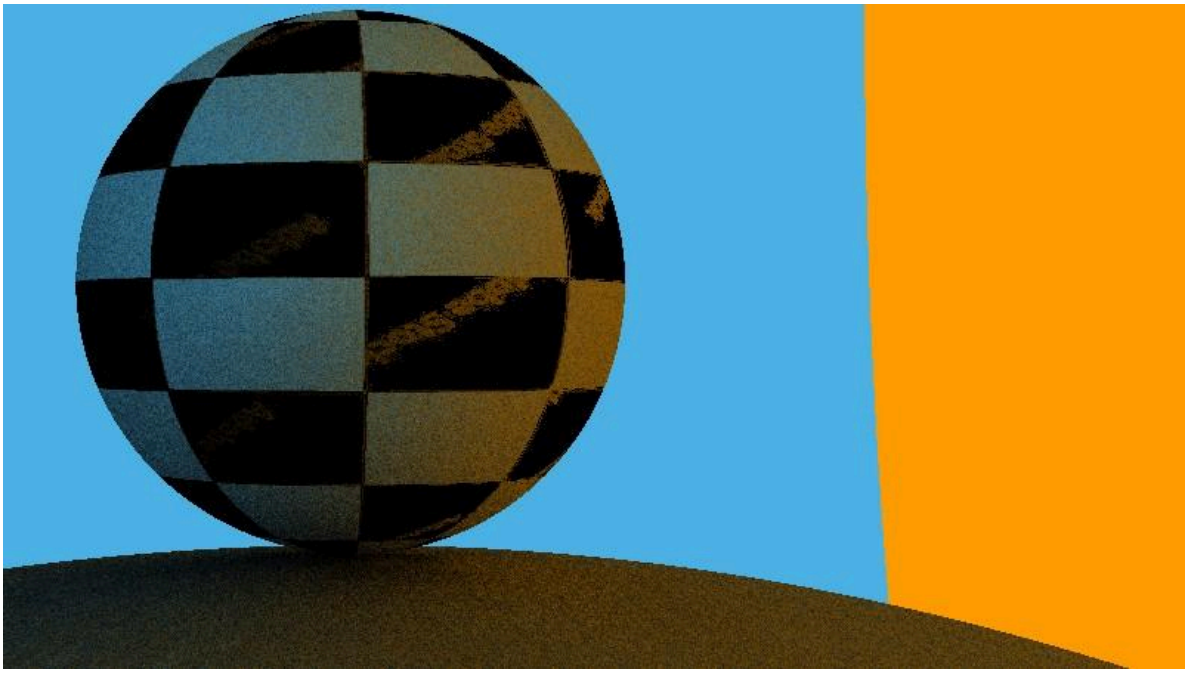


Fig. 57: Some render

Fig. 58: Some render

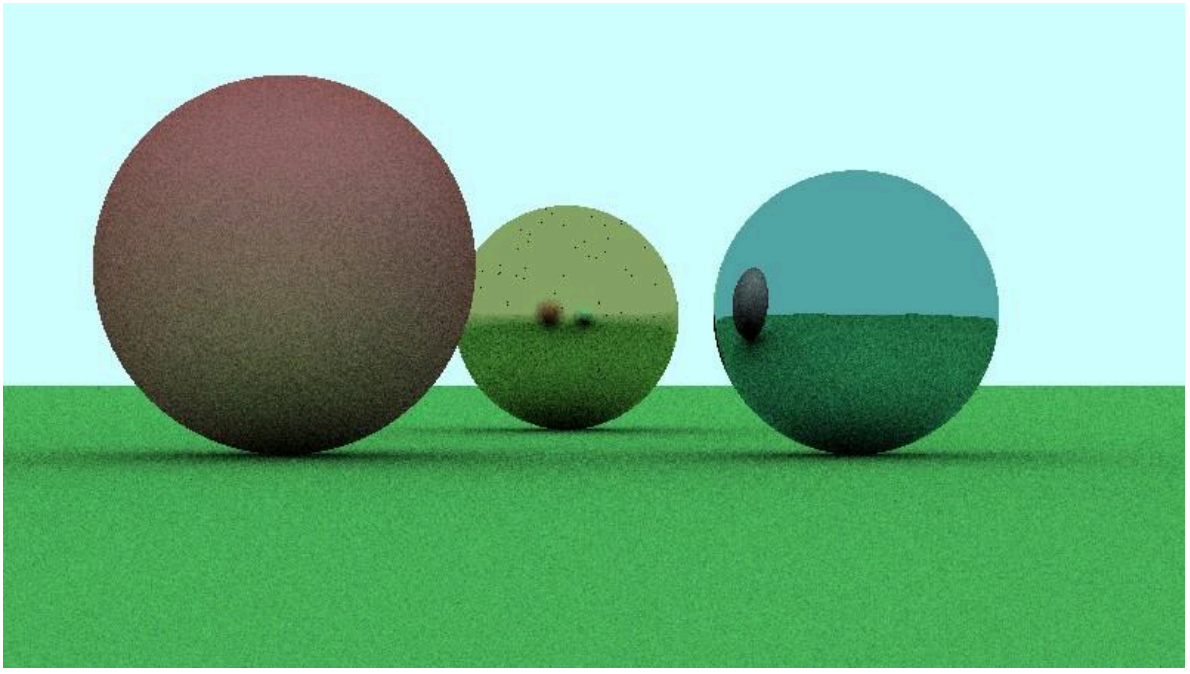


Fig. 59: Some render

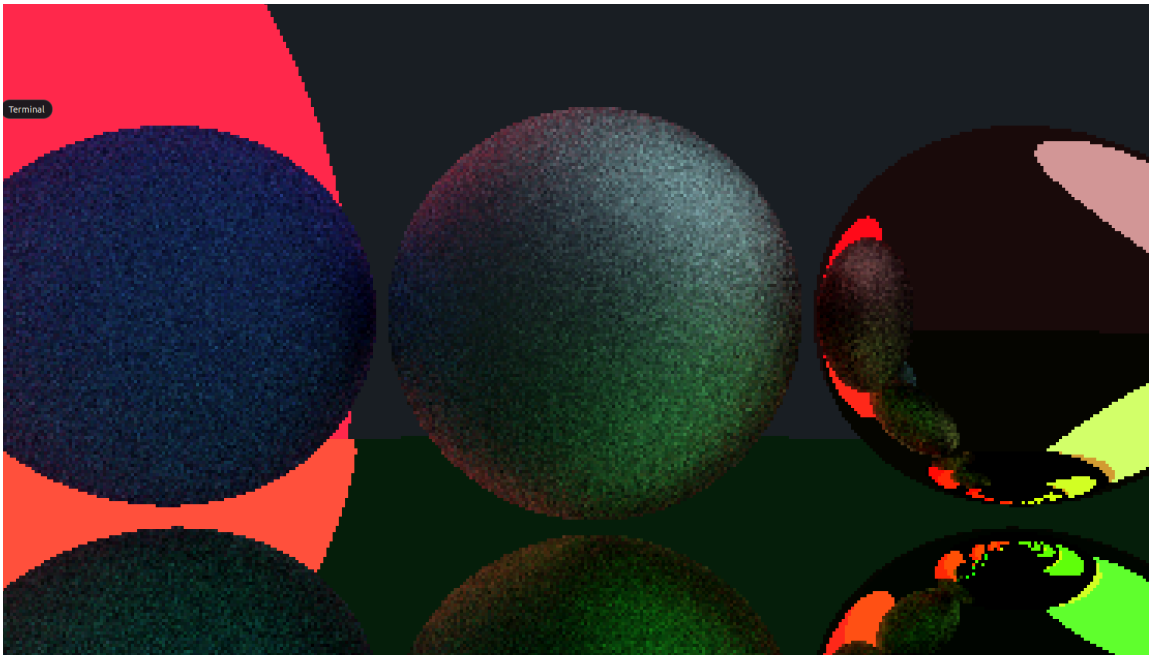


Fig. 60: Some render

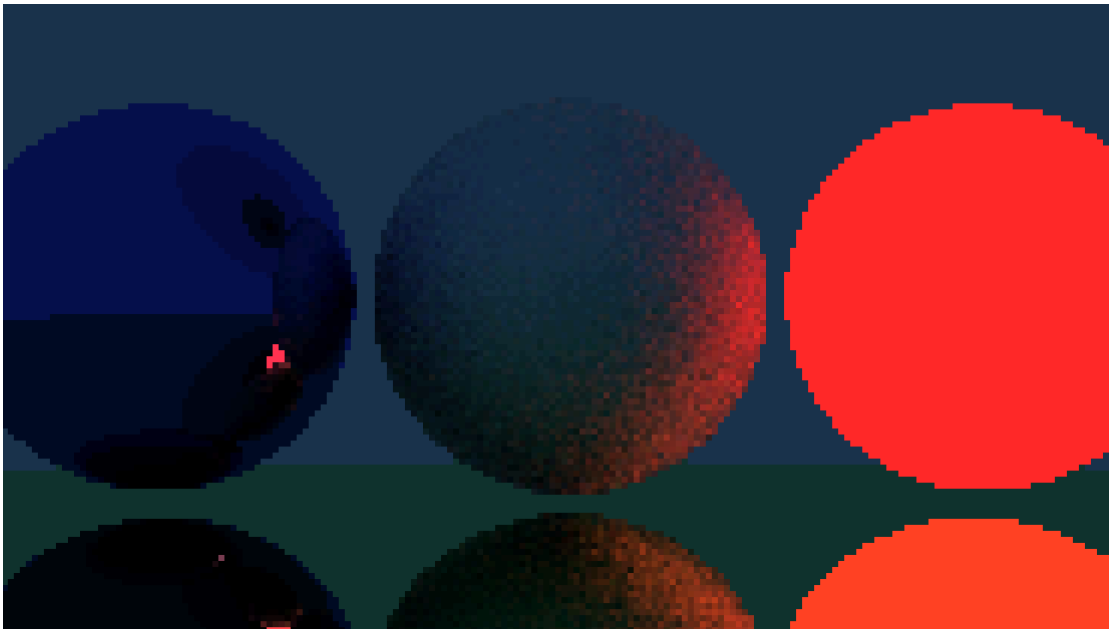


Fig. 61: Some render

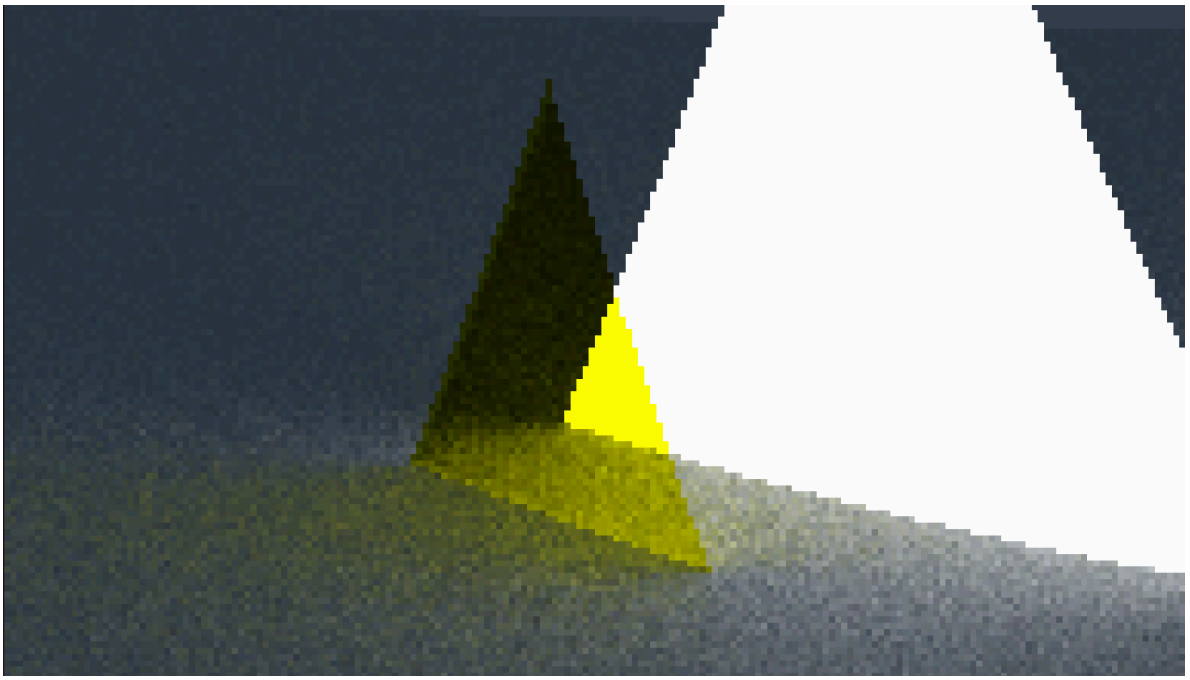


Fig. 62: Some render

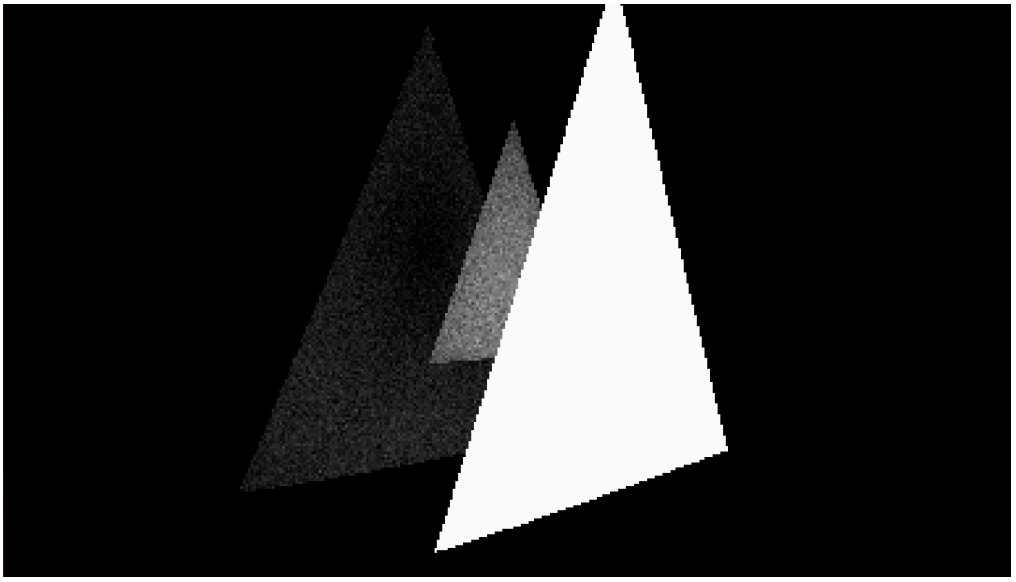


Fig. 63: Some render

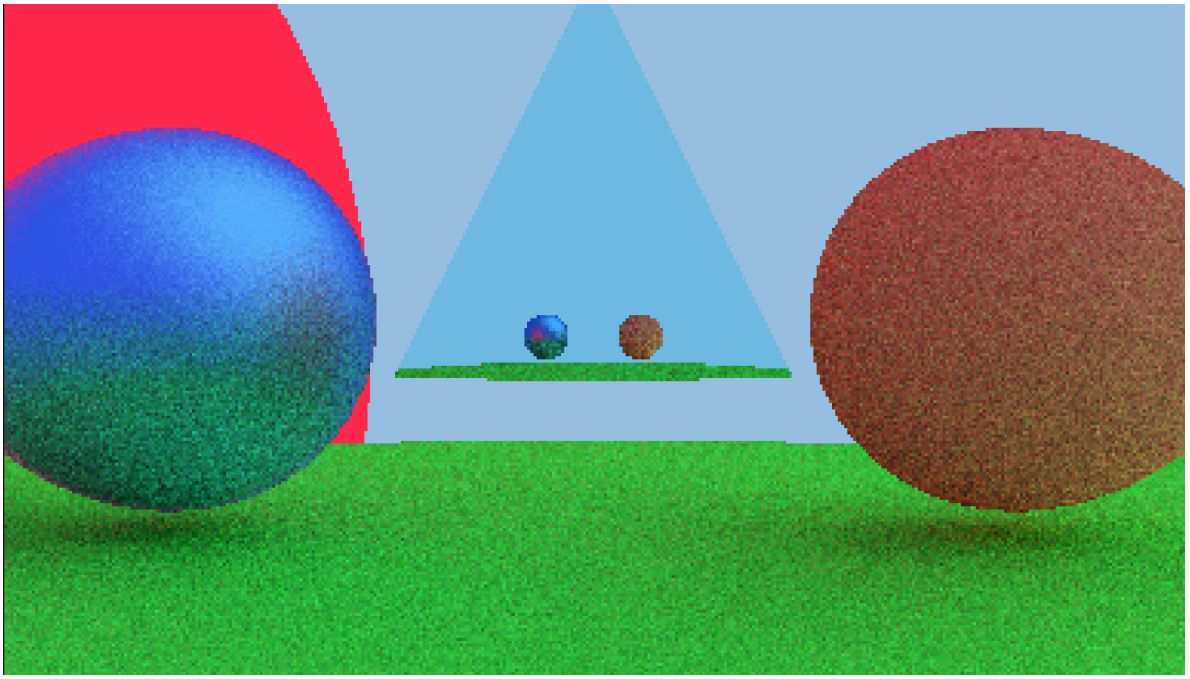


Fig. 64: Some render

Conclusion

Real-time CPU ray tracing successfully synthesises the mathematics behind ray tracing, simplifying the complex physical aspects of light into manageable mathematical expressions for a clearer understanding. It provides a comprehensive yet approachable guide to real-time ray tracing, focusing on how light appears to behave from a human perspective. The project does not delve into the physics of light as particles or waves, but instead, uses linear equations to simulate rays and mathematical operations to render colours. Additionally, it introduces a computer application running real-time ray tracing on the CPU, developed to give readers a practical understanding of the concepts.

This project represents the culmination of years of self-directed learning, combining independent application development, study of related materials, and a disciplined work schedule. The result is a clear, structured guide to ray tracing that breaks down complex topics into understandable segments while providing practical examples and real-time applications to enhance the reader's understanding.

To sum up, in this project, I gathered all the information necessary to acquire the basic understanding of ray tracing and computer graphics while I also created a program able to render 3D scenes in real time using ray tracing. This project represents the end of a chapter in a challenging journey over the past three years, which I believe is not yet complete. Throughout this process, I encountered significant difficulties in gathering and understanding the necessary information, developing the program, and integrating all aspects into this thesis. Even though I definitely succeeded with all the objectives that were set at the middle of last course, I was unable to implement to the program all the features I initially intended years ago due to time constraints: I am satisfied with the final outcome. Documenting this work has greatly improved my understanding of the project and helped me persevere

through the challenges. Overall, this has been a great educational experience and improved my english.

I plan to keep adding new features daily to this project such as supporting a metallic attribute on materials, GPU processed rendering, a window/cuda port, UV maps[20], normal maps, 3D models optimization on runtime…

[20] A file which defines where would be located the vertex of a 3D model along a 2D texture. This makes possible to see detailed 3D models like a car, or a pencil.

Bibliography

Peter Shirley, Ray Tracing in One Weekend, “Version 4.0.1”, num. 4, 31-08-2024, pages 1-41. Available at <https://raytracing.github.io/books/RayTracingInOneWeekend.html>

Joey de Vries, Learn OpenGL, “Graphics programming”, num. 1, 06-2020; pages 42, 55-63,96. Available at <https://learnopengl.com/book/book_pdf.pdf>

Matt Pharr, Physically Based Rendering, “From the theory to implementation”, num. 3, setember 204, pages 1-97 . Available at <https://www.google.es/books/edition/Physically_Based_Rendering/iNMVBQAAQBAJ?hl=es&gbpv=1&dq=physically+based+rendering&printsec=frontcover>

Daniel, C. (2022, 4 august). Laws of Reflection: Types, Formulas, and Examples! Tutorsource Platform. <https://mytutorsource.com/blog/laws-of-reflection/>

Open CASCADE Technology: Visualization. (s. d.). <https://dev.opencascade.org/doc/overview/html/occt_user_guides__visualization.html>