RayTracer
Technical Documentation

By **Rémi F**, **Pablo Z**, **Romain C** & **Aurélien P**.

Table of contents

Preamble

This project is for educational purposes only.

This project has been developed as a part of our studies at **Epitech Technology** (Lille).

It is part of the **B-OOP-400** (Object Oriented Programming) module which is a part of our second year at **Epitech Technology**.

I - General presentation

What is “ray tracing” ?

Raytracing stands as a fundamental technique in computer graphics, revered for its capacity to render exceptionally realistic images.

At its essence, raytracing emulates the behavior of light within virtual environments. Unlike conventional rendering methods, which rely on approximations, raytracing directly traces the path of individual light rays.

This meticulous approach yields lifelike visuals characterized by accurate lighting, reflections, and shadows.

Raytracing finds broad application across industries such as film, gaming, and architecture, where immersive experiences and precise visualizations are paramount.

Goals of the project

The project focuses on building a Raytracer using C++ and SFML for creating a 2D window to simulate ray tracing effects. The Raytracer computes light ray interactions with geometric objects like circles, cylinders, and cones, defined in a configuration file (.cfg).

Users can specify the .cfg file path at program launch to customize scenes. SFML facilitates real-time visualization of the scene in the interactive 2D window.

The implementation includes advanced features like light reflection, aiming for realistic rendering effects. This project merges computer graphics and mathematical principles, offering a versatile tool for visually captivating scene generation through raytracing.

In addition, users have the flexibility to launch the program with the "-sfml" option to display the rendered scene, or without the option to solely view the computational process without graphical output.

II - Installation

Environment requirements

To make sure this project is able to run on your computer, you will need to install some of the dependencies that we are using

- **libconfig**
- **libconfig-devel**

In addition of these libraries, your computer is required to have a graphic card (in the most of cases, a graphic card is already bundled in your CPU)

To compile this project you will need **Cmake** and **g++** installed on your machine.

Build the project

If all environment requirements are satisfied, you are now able to build the project.

To do so, you will find a file named **build.sh** at the root of the repository, execute the file without arguments and the project should build automatically.

If it doesn’t work, take a look at the requirements and assert that all of them are satisfied.

III - Usage

Using the example scenes

In order to execute the program, you need to specify a “scene configuration file” which ends by **.cfg**

All example scenes that we prepared can be found in the **scenes/** directory. All of these are well-syntaxed and are working with the binary.

Here’s the syntax of the raytracer :

```
./raytracer <path-to-cfg>.cfg -sfml
```

Here’s an example to run an example scene :

```
./raytracer scenes/example1.cfg -sfml
```

If everything is working fine, you’re now able to see a nice raytracing scene in a SFML window !

Feel free to edit the example scenes in order to play with our raytracer by editing objets’ positions / colors.

Running without rendering

Our raytracer allows you to run it without graphical rendering. All you have to do is to execute the binary without the **-sfml** argument.

By doing this, the window won’t show up but all the mathematical and rendering operations will be displayed in the standard output.

Scenes configuration syntax

Configurations files are fully modulable, you can edit whatever you want without breaking the program.

Camera configuration (camera):

- Center (center = { x = **double**, y = **double**, z = **double** })
- Direction (lookAt = { x = **double**, y = **double**, z = **double** })
- Ambient Color (ambientLightColor = **{r, g, b}**)
- Width (width = **number;**
- Samples/Pixel (samplePerPixel = **number**)
- Field of view (fov = **double**)

Spheres configuration (spheres[]):

- Center (center = { x = **double**, y = **double**, z = **double** })
- Radius (radius = **double**)
- Material (material = { name = **string**, color = **{r, g, b}** })

Cylinders configuration (cylinders[]):

- Center (center = { x = **double**, y = **double**, z = **double** })
- Axis (axis = { x = **double**, y = **double**, z = **double** })
- Radius (radius = **double**)
- Height (radius = **double**)
- Material (material = { name = **string**, color = **{r, g, b}** })

Planes configuration (planes[]):

- Center (center = { x = **double**, y = **double**, z = **double** })
- Norm (norm = { x = **double**, y = **double**, z = **double** })
- Material (material = { name = **string**, color = **{r, g, b}** })

Cones configuration (cones[]):

- Center (center = { x = **double**, y = **double**, z = **double** })
- Apex (apex = { x = **double**, y = **double**, z = **double** })
- Radius (radius = **double**)
- Height (radius = **double**)
- Material (material = { name = **string**, color = **{r, g, b}** })

Cubes configuration (cubes[]):

- Center (center = { x = **double**, y = **double**, z = **double** })
- Edge Length (edgeLength = **double**)
- Material (material = { name = **string**, color = **{r, g, b}** })

IV - Technical Specifications

How to create a new object

In order to add an object to the raytracer, you will need to create a new class that implements **IObject3D**.

Here’s the **IObject3D** class structure :

```cpp
namespace Rtx {
    class Ray3D;
    class IObject3D {
    public:
        IObject3D() = default;
        virtual ~IObject3D() = default;
        virtual bool hit(const Rtx::Ray3D &ray, HitData &hitData, Utils::Range<double> range) = 0;
    private:
    };
}
```

As an example, the following class represents a **plane shape** :

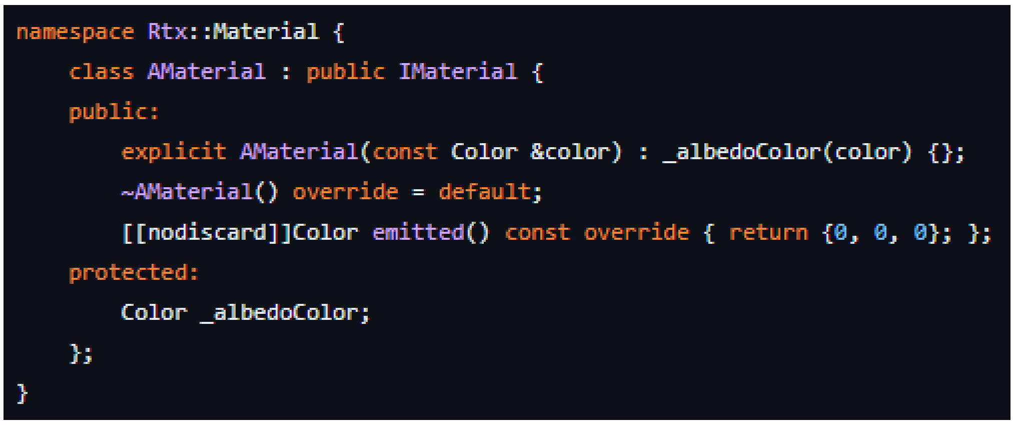
```cpp
namespace Rtx {

    class Plane : public IObject3D {
    public:
        Plane(const Math::Point3D& pos, const Math::Vector3D& norm, const std::shared_ptr<IMaterial> &material)
            : position(pos), normal(norm.unitVector()), _material(material) {}

        virtual bool hit(const Ray3D& ray, HitData& hitData, Utils::Range<double> range) override;

    private:
        Math::Point3D position;
        Math::Vector3D normal;
        std::shared_ptr<IMaterial> _material;
    };

}
```

The **hit** method represents a ray hitting the object.

How to create a new material

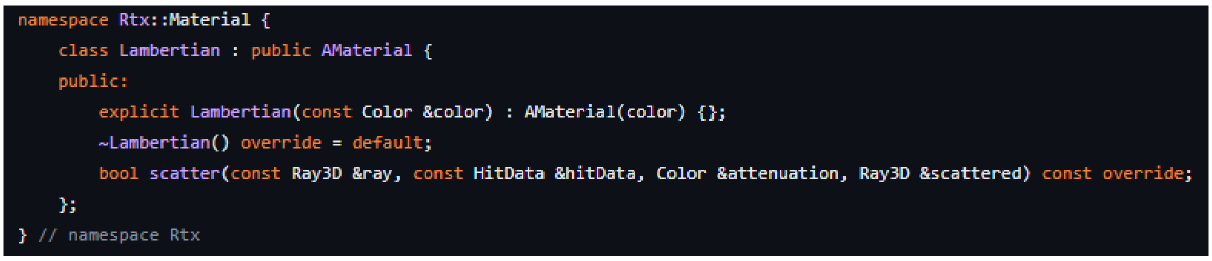
In order to add a new material to the raytracer, you will need to create a new class that implements **AMaterial** which inherits from **IMaterial**.

Here's the **AMaterial** class structure :



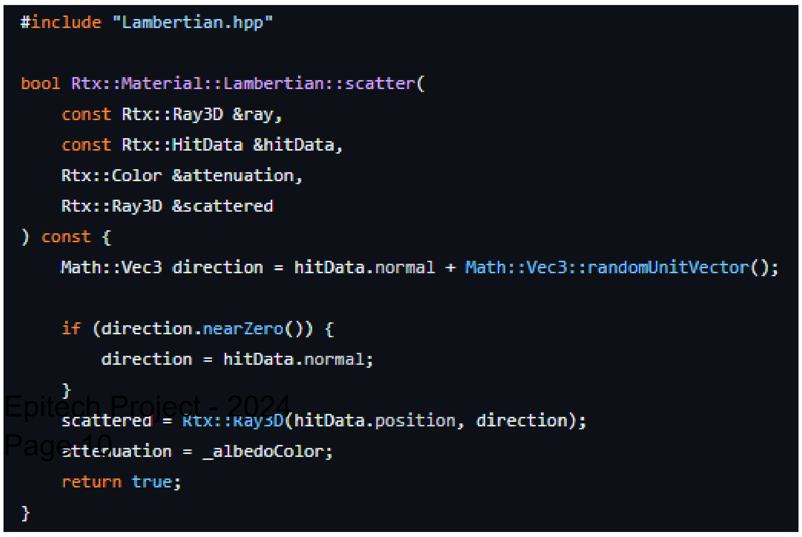
```
namespace Rtx::Material {
    class AMaterial : public IMaterial {
    public:
        explicit AMaterial(const Color &color) : _albedoColor(color) {};
        ~AMaterial() override = default;
        [[nodiscard]]Color emitted() const override { return {0, 0, 0}; };
    protected:
        Color _albedoColor;
    };
}
```

As an example, the following class represents the **Lambertian** material :



```
namespace Rtx::Material {
    class Lambertian : public AMaterial {
    public:
        explicit Lambertian(const Color &color) : AMaterial(color) {};
        ~Lambertian() override = default;
        bool scatter(const Ray3D &ray, const HitData &hitData, Color &attenuation, Ray3D &scattered) const override;
    };
} // namespace Rtx
```

Here's the concrete implementation :



```
#include "Lambertian.hpp"

bool Rtx::Material::Lambertian::scatter(
    const Rtx::Ray3D &ray,
    const Rtx::HitData &hitData,
    Rtx::Color &attenuation,
    Rtx::Ray3D &scattered
) const {
    Math::Vec3 direction = hitData.normal + Math::Vec3::randomUnitVector();

    if (direction.nearZero()) {
        direction = hitData.normal;
    }
    scattered = Rtx::Ray3D(hitData.position, direction);
    attenuation = _albedoColor;
    return true;
}
```