Epitech project RayTracer in C++

Programmer's guide

Version 1.0

May 2023

RayeTaSoeur group

Table of contents

Annex 1

Class Diagram

See Class_Diagram.pdf

Annex 2

User's manual

See this

Annex 3

Project details

See B-OOP-400_raytracer.pdf

Introduction

This chapter introduce the programmer's guide of the RayTracer Project. Read this chapter for an overview of the information provided in this manual and for an understanding of what does and how do work the project

About this manual

The Epitech project RayTracer in C++ Programmer's manual describe how the RayTracer work and how he is coded. This manual provides enough details so you can add your own plugins or for a better understanding of the code of this project.

If you just want to know how does the program works, refer to the user manual in ANNEX 2 or the Chapter 2 just for know how to start the program.

About this project

This project is an project form Epitech, made by four persons in one month. If you want better understanding of what Epitech wanted for this project, please go check the project details in the ANNEX 3

About this Group

This project was made by four Epitech Student :

Théotime SCHMELTZ

Robin DENNI

Frédéric LECHEVALIER

Luca HAUMESSER

The project

Chapter

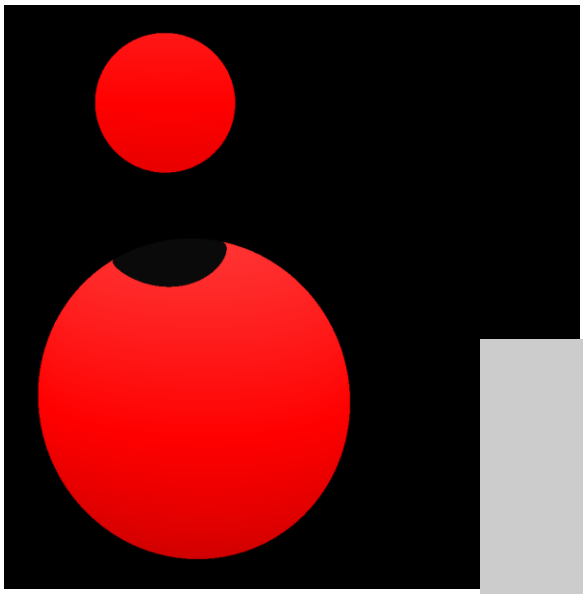
1

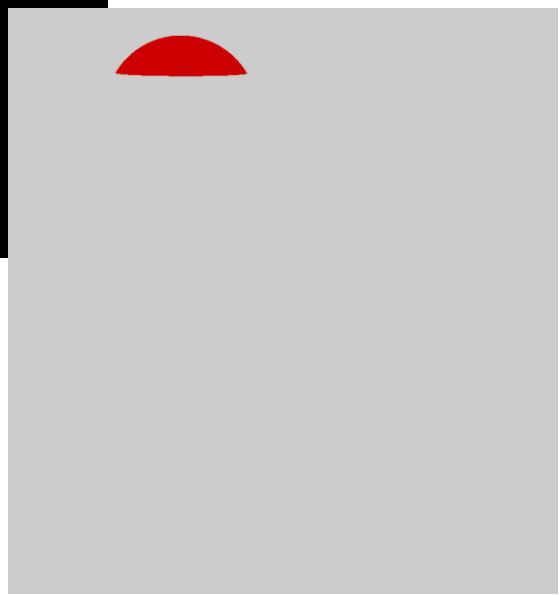
This project is the RayTracer and he is from the object-oriented programming module of the second year of Epitech's grande école program

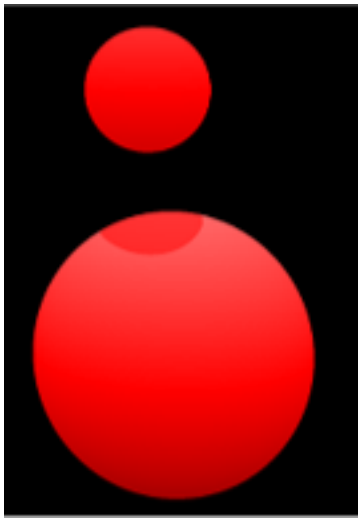
What is a RayTracer

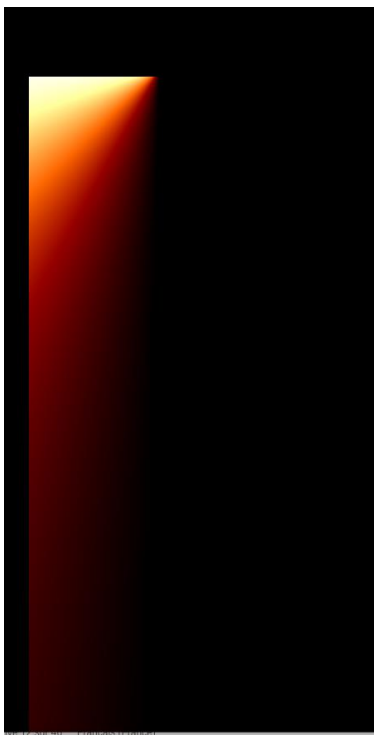
A RayTracer is a program who aims to use the ray tracing for modeling light transport for create realistic images. For this the raytracing can simulate all the things the light can do, like the reflection, refraction, optical effects and can also display various primitives, such as sphere, planes, cube.... If you want to know more about this, please check this link.

Examples pictures









The program

Chapter

2

This project is the RayTracer and he is from the object-oriented programming module of the second year of Epitech's grande école program

The prerequistes

For start this program ensure you have

The libdl

The LibConfig++

The SFML

How to build the program

This program is built with a root Makefile. So just simply run "make" at the root of the repository and the project will be build. You can also run make fclean for remove all the plugins and all the object files.

The arguments

This program takes up to two arguments :

The first argument is the config file that you want to read for build the image

The second parameter is optional, if you don't put a second argument the image will be write to the standard output.

You can also give a file name to the second parameter and the image will be write in this file (Warning, you give the file name without the .ppm extension, the program will add it by himself)

And finally, if you want to have a graphics window with your result, just pass write "sfml" as the second argument and the image will be write on a separated window.

Core Class

Chapter

3

The core class is the main class of this project, it's the main class that you will be using when it comes to build actual images

How does it work

This class allows images to be build. It is the main class link all the other in this program. This class is built by his coreBuilder builder class, for more information about the builder see the Chapter 11 and 12. This class will send the rays and display pixels of the asked colors when a pixel touches a Primitives. It's also the only class linked to the real images in this project.

What does it control

This class controls the primitives and the lights, it allows them to be showed on the screen. This class, when the methods writePPM, writeCout or WriteSFML is called, sends ray depending on the position and fov of his camera(member _camera) and ask to all his primitives (member _primitives) one by one if they is touched, and display the color depending on the light (by using the _lights member) and black otherwise.

His public methods

Add methods :

```
void addLight(std::unique_ptr<Lights::ILights> &)
```

```
void addPrimitives(std::unique_ptr<Primitives::IPrimitives> &)
```

```
void addCamera(std::shared_ptr<Camera>)
```

Add a light or a primitives or a camera to this core by passing a pointer to it, this method is only used by the builder.

Write methods :

```
void writePPM(const std::string &);
```

```
void writeCout();
```

```
void writeSFML(void);
```

This methods are used for write the image that came from the parameters inside of the class in an image(with the writePPM and by passing the the name of the file without the .ppm). You can also write the image in the cout with the writeCout method or write it inside of an SFML window with the writeSFML method.

Other methods :

```
void getNumberOfThreads(int);
```

This method is only used by the builder and tells the core how many threads the user wants to use, and the calculation will be shared between this threads.

Math Namespace

Chapter

4

The Math namespace contains all the classes that are referred to the mathematics point or vector in this project

About this Namespace

This namespace contains two classes, each one who represent coordinates in a 3d space, this classes are used for locate the objects in the 3d space used by the RayTracer, this classes are also used for making calculation in 3d space.

Point3d class

This class represent a point in a 3d space with his coordinates, the members x, y and z represent this coordinates.

It contains 3 constructors, an empty and a copy constructor ad contains also a constructor with x, y and z parameters.

Operators :

It contains an equal(=) operator for assign value from another point3d to him.

It contains a minus(-) operator for subtract a point3d to another point3d

And In last it contains a cast operator for the Math::Vector3d class who casts the point3d parameters to a vector3d with the same x, y and z parameters.

Methods :

```
double dot(Vector3d &vector)
```

```
double dot(Point3d &point)
```

This methods do the scalar product of the point with either a vector or another point.

Vector3d class

This class represent a vector in a 3d space with his coordinates, the members x, y and z represent this coordinates. This class is very similar to the point3d class but does not represent the same mathematical object.

It contains 3 constructors, an empty and a copy constructor ad contains also a constructor with x, y and z parameters.

Operators :

It contains all the base operator (+, +=, -, -=, *, *=, /, /=, =) for do this operations with the x, y and z values of the vector3d.

And In last it contains a cast operator for the Math::Vector3d class who casts the point3d parameters to a vector3d with the same x, y and z parameters.

Methods :

```
double dot(Vector3d &vector)
```

```
double dot(const Vector3d &vector)
```

This methods do the scalar product of the point with another vector.

```
double lenght(void)
```

Return the length of the vector by using the mathematical function for get the length of a vector.

```
const Vector3d normalize()
```

It return a normalized version of this vector, by using the mathematical function for normalize a vector.

Plugins system

Chapter

5

In this program an plugins system is in place for an easier way to add and remove lights and primitives

How does it work

All the lights and primitives are built in shared library (.so files). This .so files are located inside of the plugins folder, the primitives in the primitive's subfolder and the lights in the lights subfolder. When you put the name of a primitives or the name of a light inside of a config file, the program search the corresponding plugin and load him inside of him so you can use it.

Why to use them in the project

We used plugins in this project because we think it was the simplest way to add a lot of different things and load them the simplest way. This also provides some evolution, if a user wants to modify the program and add his own plugins, he just must understand the interfaces (Chapter 6 and 7), the Abstract factory (Chapter 8) and the Parser(Chapter 13) for add his own custom plugins.

How to add a plugin

To add a plugin, your plugin MUST respect four rules :

Inherit from one of the two interfaces, ILight(Chapter 7) if you want to make a light and IPrimitives(Chapter 6) if you want to make a primitive. They must also overload all the methods of these two interfaces.

Be built in a .so shared library with the name of the plugin that you want to make and be available in the good folder (primitive for the primitives and lights for the lights).

Be included with the good parameters to build it inside of the parser(Chapter 13), you may need to create a new method for build your plugin.

Have a factory who inherit from the abstract factory (Chapter 8)

IPrimitives Interface

Chapter

6

The IPrimitives interface is the interface for handle all the primitives in the code and have a generic way to call a primitive

About this interface

This interface handle the primitives in this program so if you want to make your own custom primitives you MUST inherit from this interface. She contains all the useful methods that the core (Chapter 3) use for display some primitives.

His methods

```
bool hits(Ray &ray);
```

Return true if the ray (Chapter 14) intersect this primitive

```
Math::Vector3d getColor(void);
```

Return a vector3d (Chapter 4) who contains the color of the primitive

```
Math::Point3d getOrigin(void);
```

Return a Point3d (Chapter 4) with the origin of the primitive

```
double getRoot(Ray &ray)
```

Return the root obtained with the quadratic equation of the intersection.

```
Math::Vector3d getNormal(const Math::Point3d &point)
```

Return a Vector3d (Chapter 4) with the normal of the primitive

```
void translate(const Math::Vector3d &vector)
```

Translate the primitive around and Vector3d (Chapter 4) vector

```
void rotate(double rotate)
```

Rotate the primitive with a rotate radius

```
void scale(double scale)
```

Scale the primitive with a scale multiplier

How to add new primitives

For add new primitives they MUST in first inherit from the IPrimitives interface and overload all the functions. Then you MUST add his own method inside of the parser(Chapter 13). You must also include an extern "C" part with a function getFactory who gives a pointer (std::unique_ptr) to a class who inherit from the IFactoryPrimitives interface (chapter 9). And for all the other details you can check the chapter 5.

Actual classes that inherit from it

All the classes in the folder primitives inherit from this class. The details of the actual class is :

Cone class, class for handle the cone primitive,

Cylinder class, class for handle the cylinder primitive,

Planes class, class for handle the plane primitive,

Sphere class, class for handle the sphere primitive

Torus class, class for handle the Torus.

Feel free to add your own primitives !

ILight Interface

Chapter

7

The ILight interface is the interface for handle all the lights in the code and have a generic way to call a light

About this interface

This interface handle the lights in this program so if you want to make your own custom lights you MUST inherit from this interface. She contains all the useful methods that the core (Chapter 3) use for use the lights.

His methods

Math::Vector3d getDirection(void)

Return a vector3d (Chapter 4) who contains the direction of the light

Math::Vector3d getColor(void)

Return a vector3d (Chapter 4) who contains the rgb color of the light

Math::Point3d getOrigin(void)

Return a Point3d (Chapter 4) who contains the origin point of the light

How to add new lights

For add new lights they MUST in first inherit from the ILight interface and overload all the functions. Then you MUST add his own method inside of the parser(Chapter 13). And for all the other details you can check the chapter 5.

Actual classes that inherit from it

All the classes in the folder lights inherit from this class. The details of the actual class is :

Ambient class, class for handle the ambient light,

Directional class, class for handle the directionnal light.

Feel free to add your own primitives !

Abstract factory

Chapter

8

An abstract factory is an c++ design pattern used to build families of related objects without specifying their concrete classes

What is an abstract factory

An abstract factory is an interface of factory who can produce different object from a family without have to specify their concreate class and so be more generic. For example, you can have a table abstract factory and derivate classes from this abstract factory for build things like a round chair, or another class that inherit from the factory for get a three-legged table. As long they are tables they can inherit from this abstract factory. For more details go check this.

Why we use it inside of the project

We used two abstract factories because we need to build some generic classes for the primitives(see Chapter 6) and for the lights (see Chapter 7) so an abstract factory will do the job. We just need to have a factory who inherit from the correspondent abstract factory in each of your primitive or light to get a good pointer (the abstract factory create and std::unique_ptr) to the right light or primitive. So two abstract factory exist : the IFactoryPrimitives (see chapter 9) and the IFactoryLights (see chapter 10).

IFactoryPrimitive Interface

Chapter

9

The IFactory Primitive is the interface for the abstract factory (chapter 8) of the primitive and allows them to be built with it.

About this interface

The aim of this interface is to create pointers (std::unique_ptr) to IPrimitives object. And this is an interface so each of of the primitives can built themselves by using concreate class that inherit from this interface and it make the code more generic. This factory is only used inside of the coreBuilder builder class.

His methods

```
std::unique_ptr<IPrimitives> create(ConfigPrimitive);
```

Create and get a pointer to an IPrimitive object built with all this parameters detailed in a configPrimitive object, if you want an object ready and fill with all the good parameters, it's your go to method

```
std::unique_ptr<IPrimitives> create();
```

Create and get a pointer to an IPrimitive object built with the default constructor and with zero arguments. If you want to build your object step by step from scratch, it's your go to method.

How to create a new factoryPrimitives

For create a new factoryPrimitive it must build an object with the two methods described earlier, so you must override these two and your class must inherit from the IFactoryPrimitive interface.

The configPrimtive class

The configPrimtive class, as he's name suggest itis a class who contains all the config parameters, mainly extract from the config file (see chapter 13) when built by this coreBuilder class. This class only use is in the IFactoryPrimitives and it used for retrieve setting more easily. It counts various members :

```
ConfigPrimitive(const std::string &name)
```

A constructor with the name of the primitive that you wants to build

```
Math::Point3d point3d;
```

A point3d (see Chapter 4)

```
double a;
```

A double variable

```
Math::Vector3d vector3d;
```

A Vector 3d (see Chapter 4)

```
Math::Vector3d color;
```

A vector3d(see chapter 4) for the color

```
double x;
```

A double for the x position in 3d space

```
double y;
```

A double for the y position in 3d space

```
double z;
```

A double for the z position in 3d space

```
double height;
```

A double for the height of the primitive

```
std::string axis;
```

A string for indicate on which axis (X,Y,Z) the primitive is

```
std::string _name;
```

A string for indicate the name of the primitive

IFactoryLight Interface

Chapter

10

The IFactoryLight is the interface for the abstract factory (chapter 8) of the light and allows them to be built with it.

About this interface

The aim of this interface is to create pointers (std::unique_ptr) to ILight objects. And this is an interface so each of of the lights can built themselves by using concreate class that inherit from this interface and it make the code more generic. This factory is only used inside of the coreBuilder builder class.

His methods

```
std::unique_ptr<ILights> create(ConfigLights);
```

Create and get a pointer to an ILight object built with all this parameters detailed in a configLight object, if you want an object ready and fill with all the good parameters, it's your go to method

```
std::unique_ptr<ILights> create();
```

Create and get a pointer to an ILight object built with the default constructor and with zero arguments. If you want to build your object step by step from scratch, it's your go to method.

How to create a new factoryLight

For create a new factoryLight it must build an object with the two methods described earlier, so you must override these two and your class must inherit from the IFactoryLight interface.

The configLights class

The configLights class, as he's name suggest it is a class who contains all the config parameters, mainly extract from the config file (see chapter 13) when built by this coreBuilder class. This class only use is in the IFactoryLights and it used for retrieve setting more easily. It counts various members :

ConfigLights(const std::string &name)

Create a new config Lights object with the name name

Math::Vector3d vector;

A vector3d (see Chapter 4) for get a vector for the lights

Math::Vector3d color;

A vector3d (see Chapter 4) for get the color for the lights

Math::Point3d origin;

A point3d (see Chapter 4) for get the origin point of the lights

std::string _name;

A string for the name of the light

Builder

Chapter

11

In this project we will use the builder design pattern for build the core class (chapter 3) this design pattern is useful when you need to create a complex class.

What is a builder

A builder is a class that allows to build complex classes in an easier way and if this object got a lot of different configuration options. The builder is use for create a complex object step by step. Let's say you have a class car; a builder will have some methods like build tires, build motor, build doors... So, you can build your complex car from scratch easier and with a lot of different configurations step by step. For more details, please check this.

Why to use it in this project

In this project, since we have got a complex class as the core class and since this class can have a lot a different configurations (Chapter configuration file of the annex 2), we choose to use a builder for build more easily each part of this class. We want also not rely on one constructor that can lacks to do right his job with too many things to handle.

CoreBuilder class

Chapter

12

The coreBuilder class is the builder class (chapter 11) used for build a core object (chapter 3) from scratch.

About this builder

The coreBuiler is mainly use in the main program for construct a core class object (chapter 3) and for get a pointer (std::unique_ptr) to his. And it's an easier way to construct your core class, you can construct the core class without it, but you will have hard time when you need to add some primitives or lights and for follow a config file.

His methods

```
coreBuilder();
```

Build a coreBuilder object without parameters

```
coreBuilder(const std::string &);
```

Build a coreBuilder object with the path to a config file. This constructor use the parser class (chapter 13)

```
void buildLight(const std::string &, Lights::ConfigLights);
```

Add a ILight (chpter 7) pointer (std::unique_ptr) to object to the core object with the path to the plugin file (chapter 5)

```
void buildPrimitive(const std::string &, Primitives::ConfigPrimitive);
```

Add a IPrimitive (chapter 6) pointer (std::unique_ptr) to the core object with the path to the plugin file (chapter 5)

```
void buildCamera(RayTracer::Camera &);
```

Add a pointer (std::shared_ptr) of the camera class (chapter 15) to the core object.

```
void reset();
```

Reset the core pointer inside of the coreBuilder class

```
std::unique_ptr<core> getCore();
```

Get the pointer (std::unique_ptr) to the core built by the coreBuilder class to a core object.

Parser

Chapter

13

The parser class is the class that change config file (see Annex 2) into real object inside of the code

How does it work

This classe is mainly use by the coreBuilder (chapter 12) . It reads the config file and use the libconfig++ to parse it in real instruction of creation of primitives and lights usable inside of the coreBuilder. If you want to made new Lights (chapter 7) and Primitives (chapter 6) you need to modify this class.

His methods

```
Parser(std::string configFile);
```

Construct a new parser object with a path to a file to parse.

```
void readConfig(std::string configFile);
```

Open and read the config file

```
T getSetting(std::string settingsName);
```

Read a setting with his name and return it

```
T getSetting(std::string settingsName, const libconfig::Setting
    &setting, const std::string errorSupp = "");
```

Read a sub setting with his name and with his main setting and with eventually an error code

```
std::vector<RayTracer::Primitives::ConfigPrimitive>
    &getPrimitives();
```

Get a vector who contains all the configurations details

```
std::vector<RayTracer::Lights::ConfigLights> &getLights();
```

Get a vector who contains all the configurations details for the lights

```
Camera &getCamera();
```

Get the camera (chapter 15) object

```
int getThreads();
```

Get the number of thread to use

Ray class

Chapter

14

Ray class is the class who handle the parameters for a ray inside of the raytracer.

How does it work

This class is mainly use by the core class (chapter 3) for sends ray and for make this raytracer a real raytracer. This class handle the origin point and the direction of a ray.

His methods

`Ray()`

Empty constructor construct the object with default values

`Ray(const Math::Point3d &point, const Math::Vector3d &vector)`

Construct the object with a point and with a vector

`Ray(const Ray &ray)`

Copy constructor

`void operator=(const Ray &ray)`

Make a copy of the ray passed as parameter inside of the targeted object

`Math::Point3d point;`

The origin point of the ray

`Math::Vector3d vector;`

The director vector of the ray

Camera class

Chapter

15

Camera class is used for handle all the parameters about the camera in the raytracer, she is only use by the core class (chapter 3)

About this class

With this class you can modify everything about the camera, his position, field of view... This class is built in the Parser (chapter 13) and then used by the coreBuilder (chapter 12) to construct a core (chapter 3) object for handle the camera parameters when displaying the image.

His publics members

```
Camera();
```

Default empty constructor

```
Camera(Math::Point3d origin, Rectangle3d screen);
```

Constructor with an origin point as a point3d (chapter 4) and a Rectangle representing the screen

```
Camera(const Camera &cam);
```

Copy constructor

```
void operator=(const Camera &cam);
```

Equal operator for copy another camera

```
Ray ray(double u, double v);
```

Method used to generate a ray (chapter 14) based on the camera params and on the actual position on the screen of the user

```
Math::Point3d origin;
```

Position of the camera (origin point) as a Point3d (chapter 4)

```
Rectangle3d screen;
```

A rectangle that represent the screen where the image are displayed

The rectangle 3d class

This class is mainly use by the camera class for represent the screen. This class represent a rectangle in three dimensions. His members are :

```
Rectangle3d()
```

Default empty constructor

```
Rectangle3d(Math::Point3d point, Math::Vector3d botVector,
    Math::Vector3d leftVector)
```

Constructor with the parameters of the rectangle

```
Rectangle3d(const Rectangle3d &og)
```

Copy constructor

```
Math::Point3d pointAt(double u, double v)
```

Return the coordinates in 3 dimensions as Point3d (chapter 4) with 2 dimensions given as parameters

```
Math::Point3d origin;
```

Position of the rectangle (origin point) as a Point3d (chapter 4)

```
Math::Vector3d bottom_side;
```

An Vector3d (chapter 4) representing the bottom side of the rectangle

```
Math::Vector3d left_side;
```

An Vector3d (chapter 4) representing the left side of the rectangle