B4 - Object-Oriented Programming
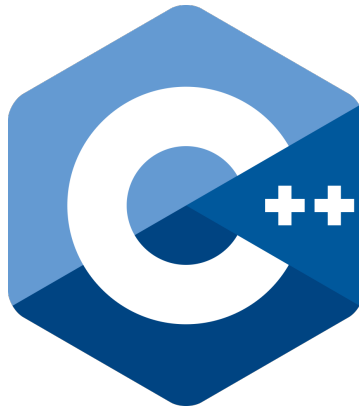
B-OOP-400

Raytracer

Your CPU goes brrrrr!

Raytracer

binary name: raytracer
language: C++
**compilation:** via Makefile (all, clean, fclean, re) or CMake

- The totality of your source files, except all useless files (binary, temp files, obj files,...), must be included in your delivery.

- All the bonus files (including a potential specific Makefile) should be in a directory named *bonus*.

- Error messages have to be written on the error output, and the program should then exit with the 84 error code (0 if there is no error).

Ray tracing is a technique used to generate realistic digital images by simulating the inverse path of light. Your goal is to create a program able to generate an image from a file describing the scene.

```
~/B-OOP-400> ./raytracer --help
USAGE: ./raytracer <SCENE_FILE>
  SCENE_FILE: scene configuration
```

Here are some examples of the expected results:
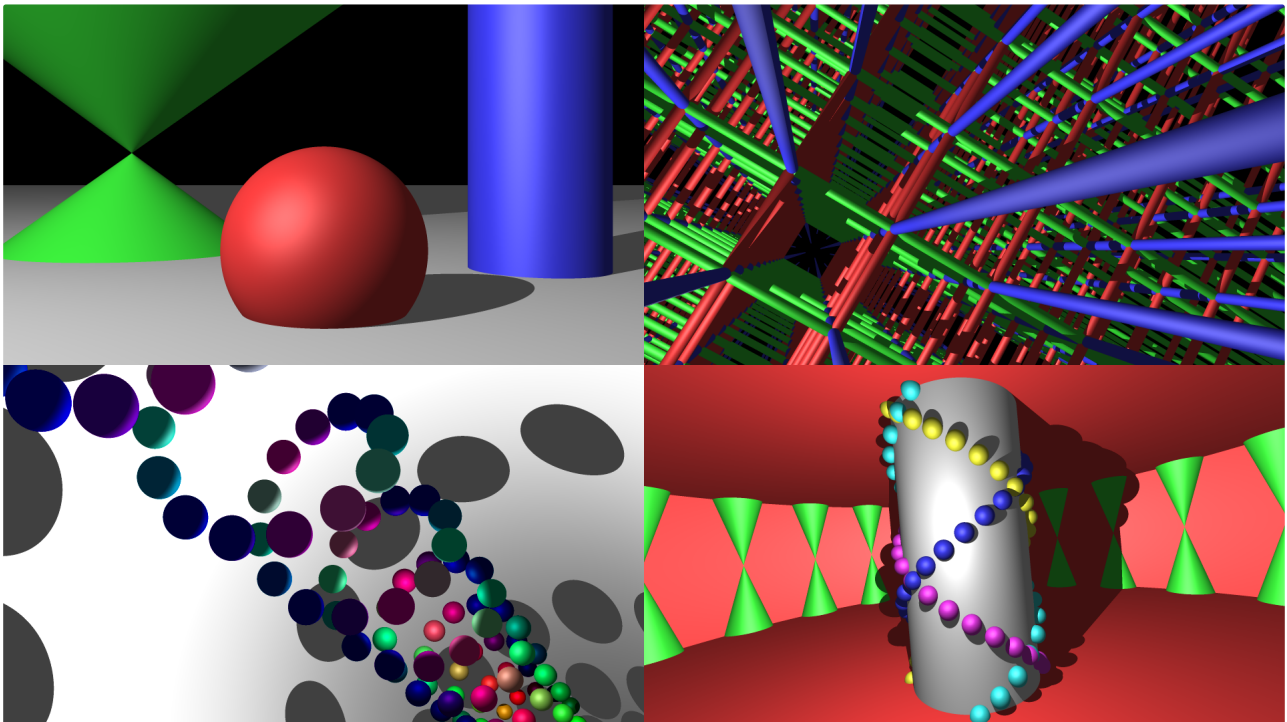
Mandatory features

Must

Your raytracer **must** support the following features:

- Primitives:
  - Sphere
  - Plane
- Transformations:
  - Translation
- Light:
  - Directional light
  - Ambient light
- Material:
  - Flat color
- Scene configuration:
  - Add primitives to scene
  - Set up lighting
  - Set up camera
- Interface:
  - No GUI, output to a PPM file

> This section lists only required **features**. The *Architecture* section lists other mandatory requirements.

Should

Once the previous features are working properly, you **should** add the following ones:

- Primitives:
  - Cylinder
  - Cone
- Transformation:
  - Rotation
- Light:
  - Drop shadows

Could

Now that you've implemented the very basics of your raytracer, implement freely the following features:

- Primitives:

  - Limited cylinder *(0.5)*
  - Limited cone *(0.5)*
  - Torus *(1)*
  - Tanglecube *(1)*
  - Triangles *(1)*
  - .OBJ file *(1)*
  - Fractals *(2)*
  - Möbius strip *(2)*

- Transformations:

  - Scale *(0.5)*
  - Shear *(0.5)*
  - Transformation matrix *(2)*
  - Scene graph *(2)*

- Light:

  - Multiple directional lights *(0.5)*
  - Multiple point lights *(1)*
  - Colored light *(0.5)*
  - Phong reflection model *(2)*
  - Ambient occlusion *(2)*

- Material:

  - Transparency *(0.5)*
  - Refraction *(1)*
  - Reflection *(0.5)*
  - Texturing from file *(1)*
  - Texturing from procedural generation of chessboard *(1)*
  - Texturing from procedural generation of Perlin noise *(1)*
  - Normal mapping *(2)*

- Scene configuration:

  - Import a scene in a scene *(2)*
  - Set up antialiasing through supersampling *(0.5)*
  - Set up antialiasing through adaptative supersampling *(1)*

- Optimizations:

  - Space partitionning *(2)*
  - Multithreading *(1)*
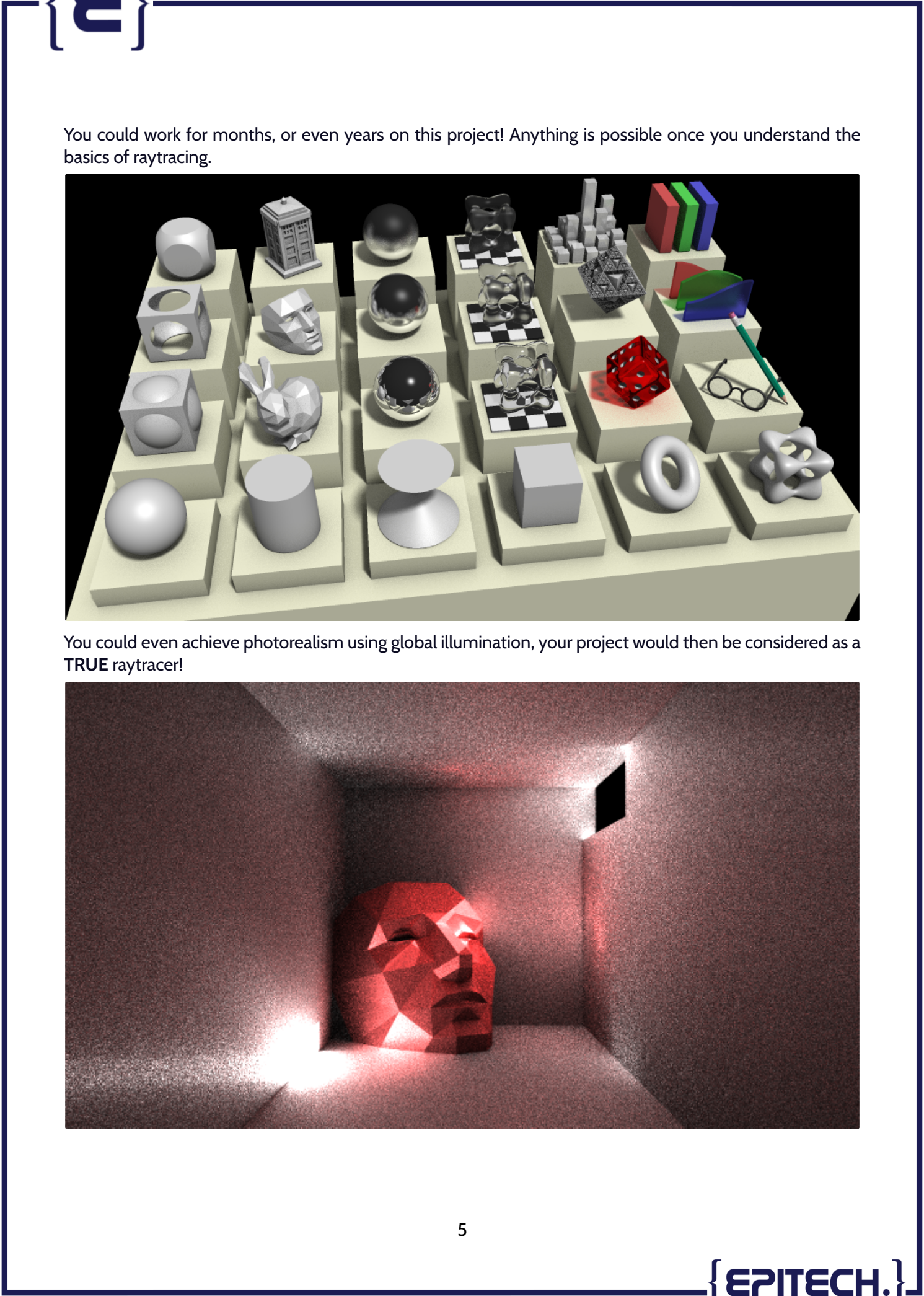 - Clustering *(3)*

{ EPITECH. }

- Interface:
  - Display the image during and after generation *(1)*
  - Exit during or after generation *(0.5)*
  - Scene preview using a basic and fast renderer *(2)*
  - Automatic reload of the scene at file change *(1)*

The number in parentheses is the number of points given in the grading scale for each feature.

> An infinite amount of bonuses are possible. Feel free to add as many features as you want as long as the mandatory (**must** and **should**) features are completed.

You could work for months, or even years on this project! Anything is possible once you understand the basics of raytracing.

You could even achieve photorealism using global illumination, your project would then be considered as a **TRUE** raytracer!

SCENE FILE FORMAT

You **must** set up the rendered scene in an external file. We suggest you use the `libconfig++` library, but you can also implement your own parser and syntax.

Here is an example of a scene file using `libconfig++` file format. Feel free to modify its structure as you like.

```
# Configuration of the camera
camera:
{
    resolution = { width = 1920; height = 1080; };
    position = { x = 0; y = -100; z = 20; };
    rotation = { x = 0; y = 0; z = 0; };
    fieldOfView = 72.0; # In degree
};

# Primitives in the scene
primitives:
{
    # List of spheres
    spheres = (
        { x = 60; y = 5; z = 40; r = 25; color = { r = 255; g = 64; b = 64; }; },
        { x = -40; y = 20; z = -10; r = 35; color = { r = 64; g = 255; b = 64; }; }
    );

    # List of planes
    planes = (
        { axis = "Z"; position = -20; color = { r = 64; g = 64; b = 255; }; }
    );
};

# Light configuration
lights:
{
    ambient = 0.4; # Multiplier of ambient light
    diffuse = 0.6; # Multiplier of diffuse light

    # List of point lights
    point = (
        { x = 400; y = 100; z = 500; };
    );

    # List of directional lights
    directional = ();
};
```
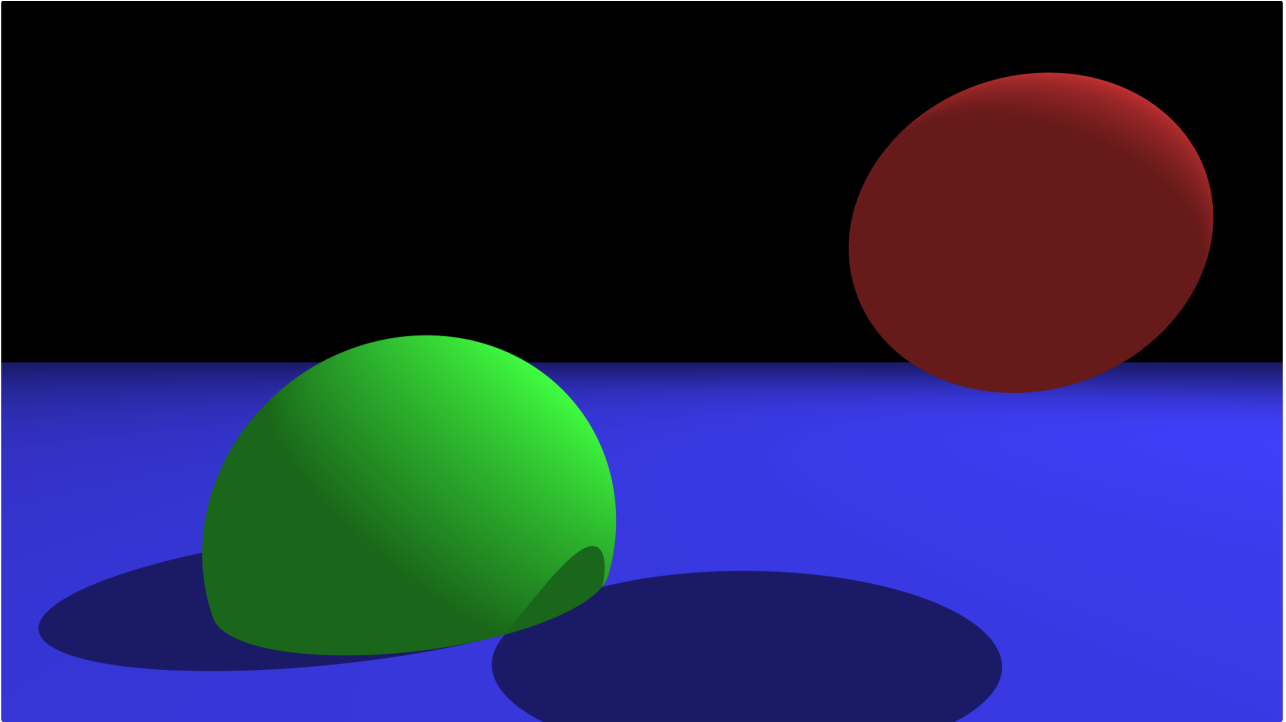
There may be better ways to structure your file, take some time to think about it.

This file would produce a picture looking like this:

Architecture

Interfaces

To allow extensibility, you **must** use interfaces at least for your primitives and lights.

Plugins

A rendering engine should be extensible. One should be able to add new features without completely rewriting the code. You could do it using dynamic libraries (`.so`) as plugins and load them at runtime. This feature is **not mandatory**.

Your executable must not be linked to any of these plugins. You must store your plugins in a `./plugins/` directory.

You **could** use a plugin system for the following features:

- Primitives
- Lights
- Scene loaders
- Graphical User Interface
- Core renderers
- Optical effects
- Etc.

Design patterns

Additionally, you **must** use at least 2 design patterns from the following list in your project:

- Factory
- Builder
- Composite
- Decorator
- Observer
- State
- Mediator

> Your choices of design patterns will be discussed during the defense.

Authorized libraries

The only authorized libraries are:

- `libconfig++` to parse the scene configuration file
- `SFML 2.5.1` for displaying

Ask your pedagogical team if you want to use additional libraries for your bonuses.

> Only the standard C++ library, `SFML` and `libconfig++` are authorized for mandatory features.

Build

You are free to choose either `make` or `cmake` to build your project.

Makefile

Your Makefile **must** have the usual mandatory rules.

The results of running a simple `make` command in your turn in directory must generate the `raytracer` program at the root of the repository and the (optional) plugins in the `./plugins/` directory.

CMake

Your CMakeLists.txt **must** build a program at the root of the repository and the plugins in the `./plugins/` directory at the root of the repository.

```
▽                                      Terminal                              –  +  x
~/B-OOP-400> mkdir ./build/ && cd ./build/
~/B-OOP-400> cmake .. -G "Unix Makefiles" -DCMAKE_BUILD_TYPE=Release
[...]
~/B-OOP-400> cmake --build .
[...]
~/B-OOP-400> cd ..
~/B-OOP-400> ls ./raytracer ./plugins/
./raytracer

./plugins/:
raytracer_cone.so
raytracer_cylinder.so
raytracer_plane.so
raytracer_sphere.so
```

{ EPITECH. }

Defense

You must demonstrate your raytracer's features during the defense. Prepare some scenes and screenshots of your work!

```
▽                              Terminal                          –  +  x
~/B-OOP-400> ls -R ./scenes/ ./screenshots/
./scenes/:
demo_cone.cfg
demo_cylinder.cfg
demo_plane.cfg
demo_sphere.cfg
light_point.cfg
light_directional.cfg
[...]

./screenshots/:
demo_cone.ppm
demo_cylinder.ppm
demo_plane.ppm
demo_sphere.ppm
light_point.ppm
light_directional.ppm
[...]
```