# Computer Programming 4

## “Simple Raytracer”

Final Report

Supervisor: DSc, PhD R. Starosolski

Author: Gracjan Jeżewski

Informatics, II year, 4th semester

23 May 2021

## Topic and analysis

For the implementation of the Raytracer, I have decided to use a simplified version of **Path Tracing algorithm**. Ray tracing is a rendering technique used to create digital rendering image by following (imitating) the rules behind the phenomenon of light. Algorithm behind this rendering method is more of a set of loose rules rather than strict scheme of working, that is why each implementation could support different set of features.

Notable features available in my implementation of the algorithm, are such:

- Program imitates the behaviour of a light ray [Ray] in a way, where ray is interpreted as a **linear function of time**:

$$f(t) = ray_{origin} + ray_{direction} \times t$$

- Each ray follows an **inverse path** (with respect to the real light ray), eg. rays are traced from the point of view [Camera], instead of following the path of each ray coming from the light source. This way, rays which could never reach the observer are discarded.

- The final result follows the rules of **Monte-Carlo simulation** eg. the more samples are gathered, the more accurate the results,

- Ray can be reflected or scattered depending on the surface properties of different objects [Material]. It is **unbiased**, meaning that the reflected ray is generated randomly, not biased towards light source in any way.

To make the project doable in such limited amount of time, **many simplifications** had to be implemented instead of more physically-correct (and significantly harder) concepts. Most notably, the scene is globally illuminated, meaning, that if ray reaches the outer space (“sky”) it is considered to have reached the light source.

### Output & Library usage

Output of the engine, defines colour of each pixel in the image grid of given resolution, which can be interpreted in many ways. For my implementation however, I have decided to use SFML library, which allows me to interpret one dimensional array of RGBA colour partials, as a consecutive pixels displayed as a sf::Sprite on a sf::RenderWindow.

Additionally, sf::RenderWindow is able to handle user interactions such as key presses or manipulation of the window itself, which allowed me to implement few user interaction functionalities.

**SFML library** is only used to create a window which is able to display the result by interpreting the pixel array ( [Renderer] output ) and for handling the user keyboard input.

## External specification

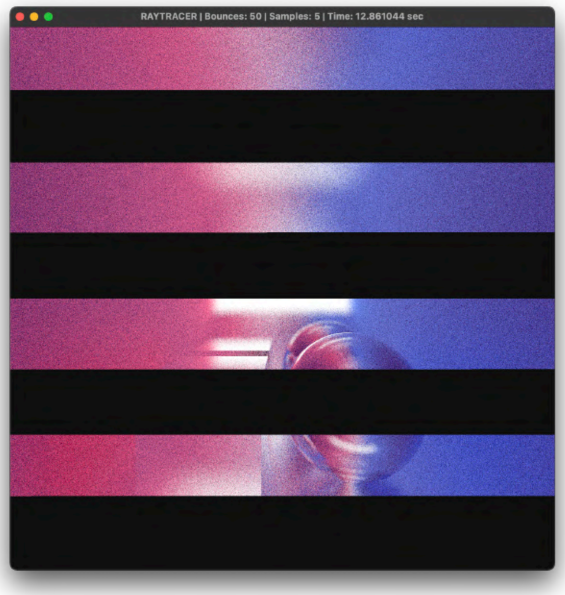
During the execution of the program, user is able to interact with the program in a following way:

- **Left / Right** keyboard arrow key allows to change currently rendered [Scene] to the previous / next accordingly.
- **Up / Down** keyboard arrow key allows to change number of samples taken for each pixel of the rendered image
- **Spacebar** allows to terminate rendering of the current [Scene].

After first two interactions (arrow key), rendering is automatically restarted to match the new settings. To restart the rendering after terminating the current render, it is necessary to change the scene.

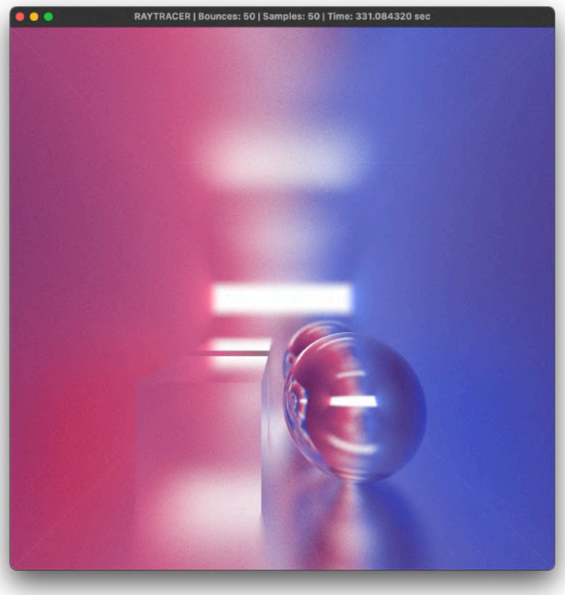
Each scene has its own set of settings. These settings (if altered) are saved for the time of execution of the program.

Additionally, **window title** acts as unobtrusive menu displaying current settings and execution time.



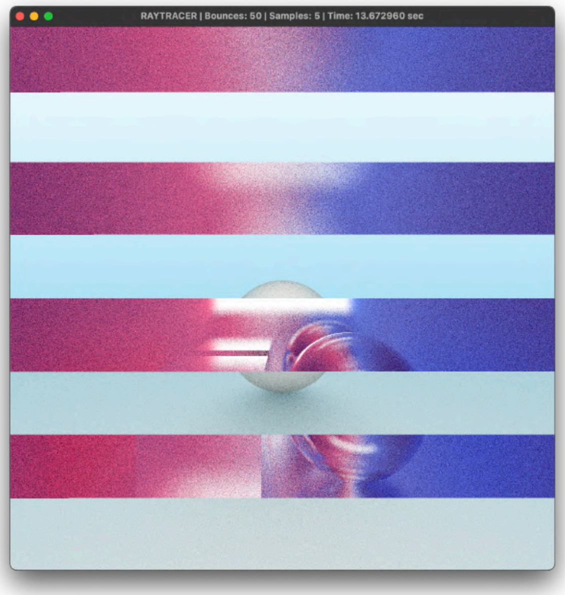
RAYTRACER | Bounces: 50 | Samples: 5 | Time: 12.861044 sec

Predefined scene during rendering with initial low settings. Relatively short time of rendering. Significant noise resulting from small number of samples.



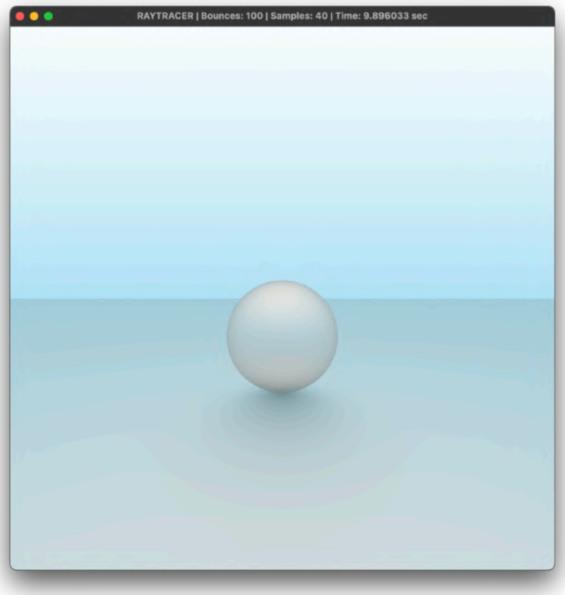
RAYTRACER | Bounces: 50 | Samples: 50 | Time: 331.084320 sec

Identical scene with higher number of samples per pixel. Significant reduction of noise, but  significant increment of rendering time.



RAYTRACER | Bounces: 50 | Samples: 5 | Time: 13.672960 sec

Change of scene, new image overwrites old rendering. New settings are visible on the title of the window, timer counts the time from start to finish of the current rendering.



RAYTRACER | Bounces: 100 | Samples: 40 | Time: 9.896033 sec

Plain scene with diffused materials, small number of objects and significant amount of light substitute to short rendering time even for higher settings.

# Internal specification

Program consists of many classes, however, there exist three classes which coordinate all of the smaller elements.

[App], [Window] and [Renderer] are the main classes of the program, each of them, starts with initialisation of the data necessary to perform the rendering. Most allocations are performed in "Initialise" methods, additionally, because of the nature of these methods, this is where **Exceptions** are caught. If any of these methods fails, execution of the program is stopped and objects are destructed (exceptions handling is not performed inside constructors).

On a side note, [App] class might seem a little bit unnecessary, however it acts as a container for all of the program functionality, and was created with future expansion in mind.

After successful initialisation, sf::RenderWindow from [Window] class is opened and first scene from the predefined scenes is rendered by [Renderer].

Rendering is a more complex process, but below, a short walkthrough is described:

1. [Scene] has a [Camera] object which defines "perspective plane". This plane is an abstraction of the image and camera (eye) in 3D space, it is created with the same aspect ratio as the window. The distance between point of view and this plane is defined as a focal length of [Camera].

2. For each pixel, based on his X and Y coordinates (current_Width / Width, current_Height / Height) rays are shoot through the points of the projection plane (Scene::prepRay method creates ray of adequate origin and direction). Rendering starts from the left upper corner (abbr. **LUC**), which happens to be the first pixel location of Texture object from SFML library (general convention).

3. After preparing the [Ray], it is intersected with the [Scene]. Each scene has a container of objects (**STL Containers**). Objects ([Sphere], [Plane], [Rectangle], [Disc], [Cube]) must be defined in terms of mathematical equations, “intersection” is simply looking for a common solution to both, Ray function and object equation.

4. Using ray equation and obtained solution, position of the intersection in 3D space can be determined. What happens after the collision is defined by the [Material]. There are currently two materials, Metallic (ray reflection) and Diffused (ray scattering).

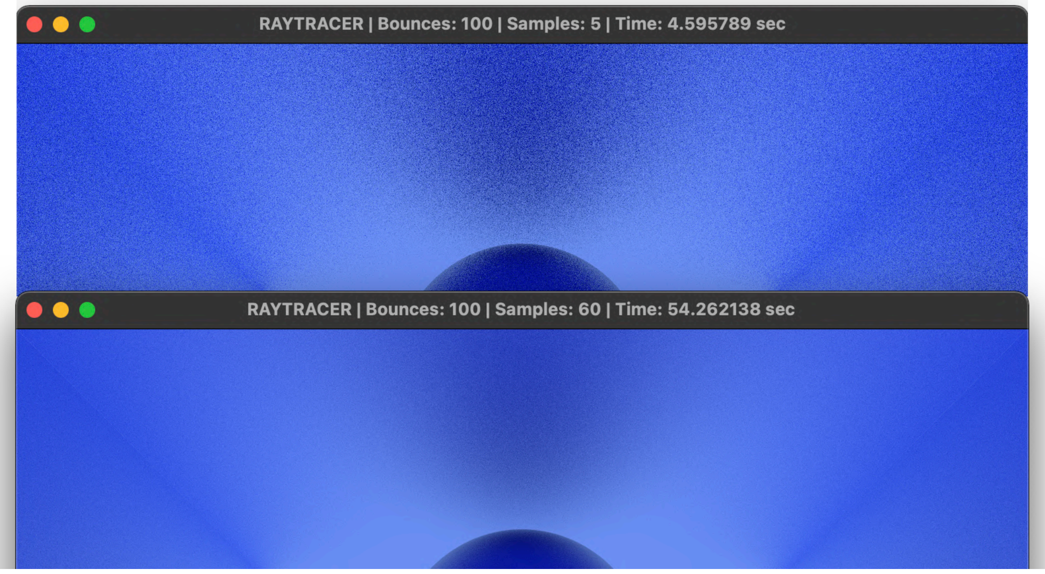
- Metallic material generates new ray (Reflected) which has its origin at the point of intersection and direction set accordingly to be perfectly reflected (angle of incidence = angle of reflection) using [vect3D] vector transformations. To imitate roughness, this direction can be altered by small randomly generated vector (using functionality of randomNumber **Template** methods).
- Diffused material creates a new ray by randomly generating scatter direction at the point of collision. The only requirement for the new random direction, is being on the proper side of the geometry (Compared using dot product with outward normal of the geometry).

*Simplification: Only outward normal vector is taken into account when performing intersections and reflections. What it means, is that using geometry with incorrect normal vector (or using geometry with outward normals as a enclosing geometry eg. Cube as a “room”) leads to incorrect reflection of the ray and could lead to incorrect lighting.*

5. Scene::colourRay is recursive, it returns a [Colour] in such way:

- If reflected ray was created, the whole procedure repeats.
- If reflected ray reached upper limit of bounces plain black colour is returned
- If ray reached the “sky”, sky colour depending on the height (gradient) is returned. (*Simplification, no light sources besides sky*)

| SAMPLES | BOUNCES |
|---|---|
| Define how many rays per pixel are "shoot". Colour of the pixel is an **average of sum** of colour returned by Scene::colourRay. **Reduces amount of noise.** | Defines how many reflected rays are created for each pixel, **"depth of reflections"**. Final returned colour from colourRay method is a **product** of all reflections. |


RAYTRACER | Bounces: 100 | Samples: 5 | Time: 4.595789 sec
RAYTRACER | Bounces: 100 | Samples: 60 | Time: 54.262138 sec

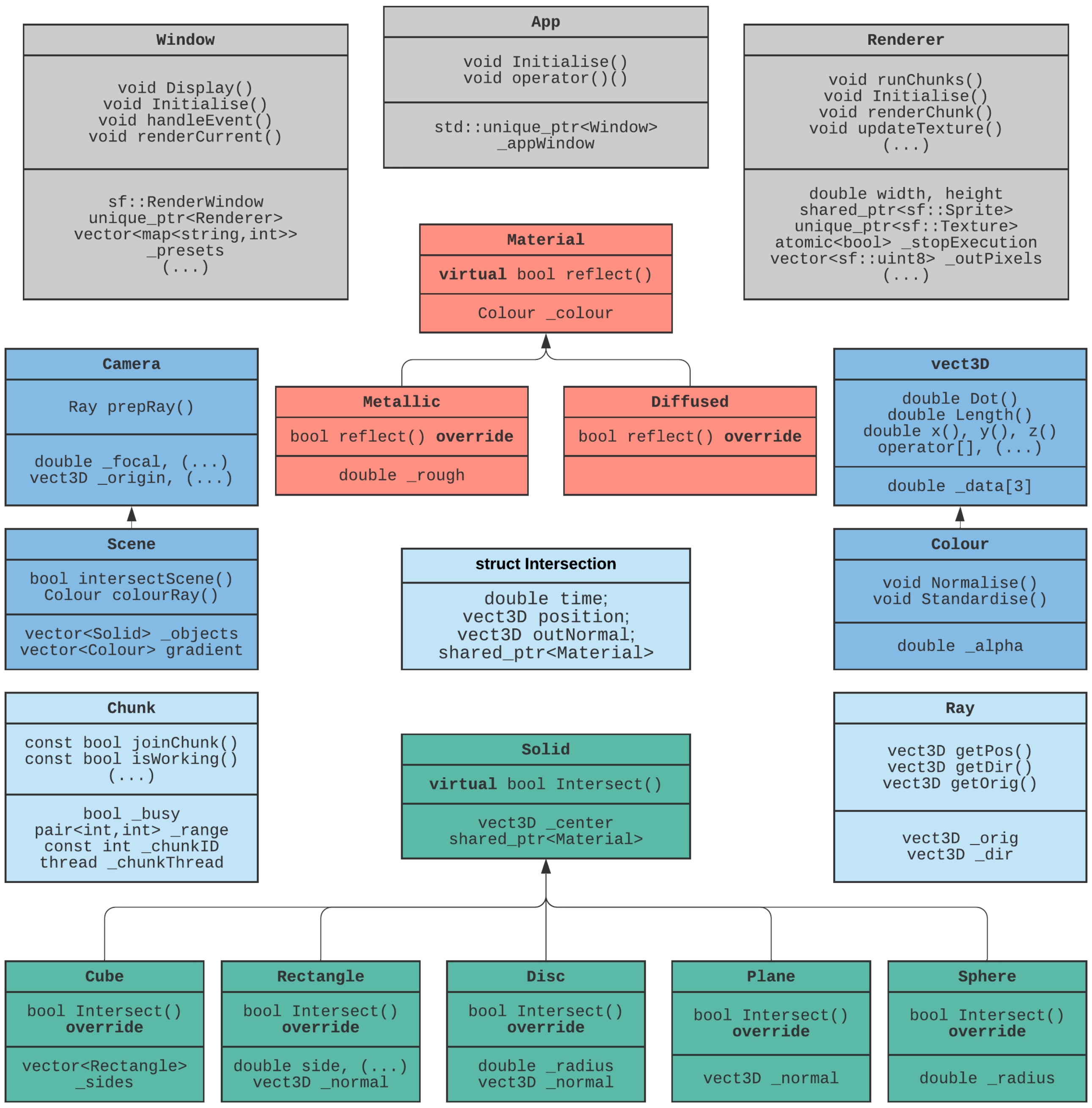
Difference between two differently sampled renders of the same scene.

**Multi-threading** is implemented in a way, where each thread [Chunk] performs the above (points 2-5) for different X and Y pixels (**unique range** defined by height). According to the C++ memory model, writing and reading memory contents from different memory locations of the same container (output pixel std::vector) is thread-safe. Other data is read-only.

**Smart Pointers** are extensively used in place of raw unsafe pointers throughout the whole program. Very minimal usage of **STL Algorithm** is also implemented for Map STL Container.

## Class diagram

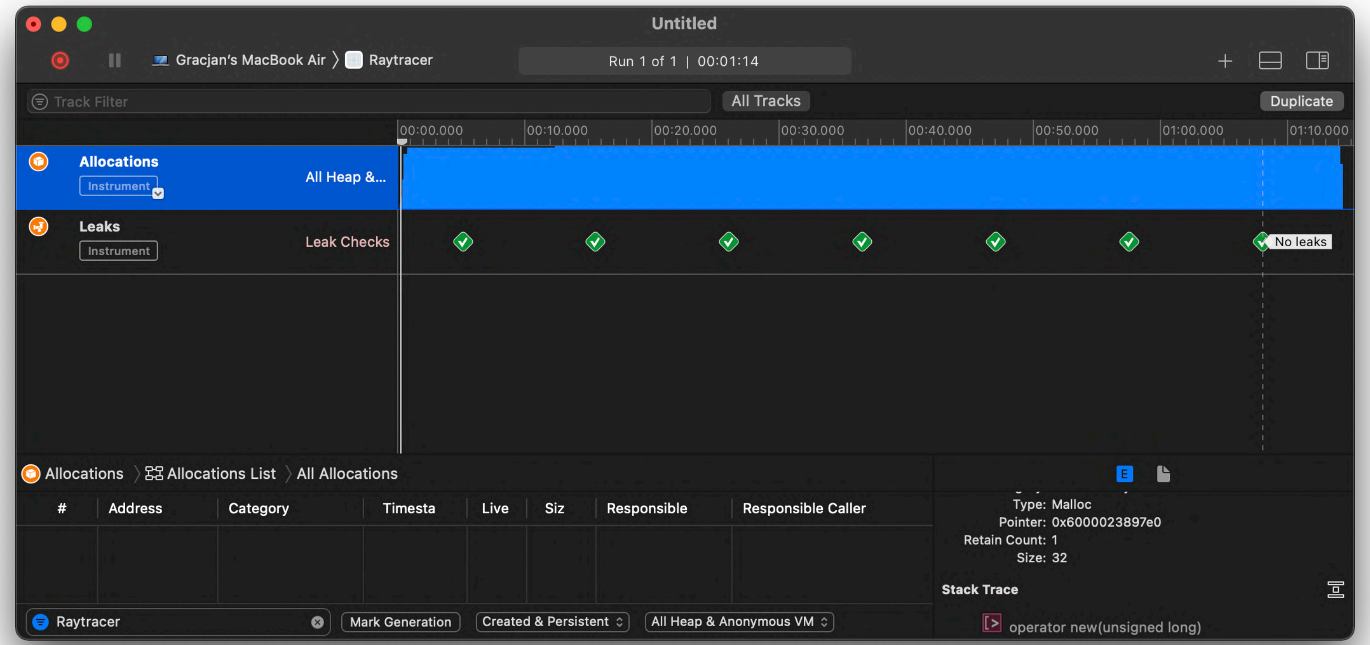
*Diagram presented below contains shortened naming convention and limited number of defined methods. It serves a purpose of showing the general structure of the program. Often, only the most important methods and variables are declared.*



## Testing and Debugging

Program was thoroughly tested using many debugging tools available in Xcode IDE:

- **Thread Sanitizer** was used to check data races. This way, a major, random number generator bottleneck was detected and resolved (static -> thread_local static),
- **Address Sanitizer** was used to determine if buffer overflow occurs (Small mistake where last thread obtained incorrect range was fixed),
- **Time Profiler** which determined where the most amount of time was spent which helped optimise the code immensely,
- **Leaks instrument** used to detect memory leaks. Significant leak was detected - result of not making base destructor of [Solid] virtual. Fortunately after applying the above fix, no other leaks were detected afterwards. Instrument was ran for 1 minute during which all user input methods were used and one predefined scene was fully rendered. At the bottom of the instrument, no persistent (leaked) objects are detected.



| # | Address | Category | Timesta | Live | Siz | Responsible | Responsible Caller |
|---|---|---|---|---|---|---|---|

Type: Malloc
Pointer: 0x6000023897e0
Retain Count: 1
Size: 32

operator new(unsigned long)

## Conclusions and Remarks

In conclusion, I am satisfied with the final program and its functionality. I have experienced many different problems during implementing the logic of the program, which helped me to get a grasp of more complex topics (especially threading) and made me understand some of the basics a little bit better as well.  As a computer graphics enthusiast, I became familiar with underlying logic of basic computer graphics software.

Additionally, topic was broad enough to cover many new concepts from the thematic classes. What is more, it is very well researched due to recent advancements made, books such as Physically Based Rendering or Raytracing series by Peter Shirley were immensely helpful.

I haven't however, planned to use many simplifications such as globally illuminated scene (no user defined light sources), or one-sided geometry. If I am to improve my program, these are the issues which have to be solved to move forward. Additionally, primitive Chunk division and antialiasing need rework, current implementation is straightforward but far from perfect.

Concerning the classes, few specific presentations were great, and provided all necessary information about the topic in a concise way, which speeded up the process of implementing them into the program.