



Cathode Ray Tracer

*Submitted in partial fulfillment of
the requirements for the award of the degree of*

**Bachelor of Science
in
Computer Science and Engineering**

Author:
Hamza HAIKEN

Supervisor:
Prof. Pier DONINI

heig-vd

HAUTE ECOLE D'INGÉNIERIE ET DE GESTION
Yverdon-les-Bains, Canton de Vaud

Summer Semester 2015

Table of Contents

Table of Contents	i
List of Figures	ii
List of Tables	iii
Foreword	iv
1 Rendering process	1
1.1 Scenes	1
1.1.1 Entities	1
1.1.2 Light sources	1
1.1.3 Camera	1
1.1.4 Settings	2
1.1.5 Class structure	2
1.2 Ray tracing	2
1.2.1 Process	2
1.2.2 Constructive solid geometry	2
2 Language	4
2.1 ANTLR4	4
2.2 Grammar	4
2.3 Compiling process	4
A Acknowledgements	5
B Appendix	6
C Bibliography	7

List of Figures

1.1	Real-life <i>bokeh</i>	2
1.2	Rendering process class diagram	2
1.3	Rendering process activity diagram	3
1.4	A piano foot obtained from CSG operations	3
2.1	Classes generated by ANTLR4	4

List of Tables

Foreword

Foreword...

1. Rendering process

Rendering an image involves several steps. The general thought process is as follows: what objects are placed on the scene? What are they made of and how does *light* interact with them? Where is the camera placed, and where is it pointing to? How many light sources are present in the scene, and which ones have an effect on which objects? What rendering options are enabled?

To answer these questions, we will see what classes represent a scene and how to trace rays in the following sections.

1.1 Scenes

A scene is represented by a `Scene` class which contains all the entities that will be drawn, as well as all important information on how to draw them:

- A list of entities
- A list of light sources
- A camera
- A `Settings` object

1.1.1 Entities

Entities are primitives volumes that can easily be described with mathematical equations, such as boxes (*parallelepipeds*), spheres, cones, planes and half-planes, etc. Each entity must provide an `intersect()` method for computing its intersection points with a given ray, which we will need later on to do the rendering.

Entities also contain a `Material` object, which will describe what the entity is made of. Materials possess several attributes that describe how light interacts with it:

- A color, provided by the `Pigment` class
- Reflectivity, for shiny surfaces
- Transparency, defining how many photons can go through.
- Refractive index, defining how much light is slowed down when passing through the material.
- A diffuse factor, which makes light bounce diffusely.
- Specularity, for harsh highlights (this is a computer graphics trick, it is not physically accurate).
- Shininess, defining how sharp the specular highlight will be.

Only having mathematical primitives is however very limiting for a creative user. To remedy this, an entity can also be the result of a CSG operation, which can be a union, a difference or an intersection. CSG operations will be explained in details in the section on ray tracing.

1.1.2 Light sources

Light sources give color to entities, and is the target of all the rays we bounce off entities. A light source is defined by the `Light` class and has the following properties:

- A point of origin, defining from where the light is shining.
- A *falloff* factor: describes the natural effect observable in nature, where light follows an inverse square law: the intensity of light from a point source is inversely proportional to the square of the distance from the source. We receive only a fourth of the photons from a light source twice as far away.
- A color, given by the `Pigment` class
- An ambient light factor: because simulating global illumination is mathematically difficult and takes a lot of processing, we can simulate ambient light (accumulation of light that bounces of many surfaces) by setting an ambient factor, which will basically add a fraction of the value of its color and intensity.

1.1.3 Camera

A lit and populated scene still needs a window through which we will observe it: the `Camera` class defines the point of view of our rendered scene. It has a position, a direction vector, and a focal length (field of view angle). To further add to the user's creative possibilities, we implemented several features which aim to mimic real-life cameras:

- Depth of field (DOF), creating a plane in which objects are sharp, and blurry beyond.
- An aperture shape, which will be used to physically simulate the shape that *bokeh* will have (see figure below).
- A focal distance, defining at which distance objects are sharp.

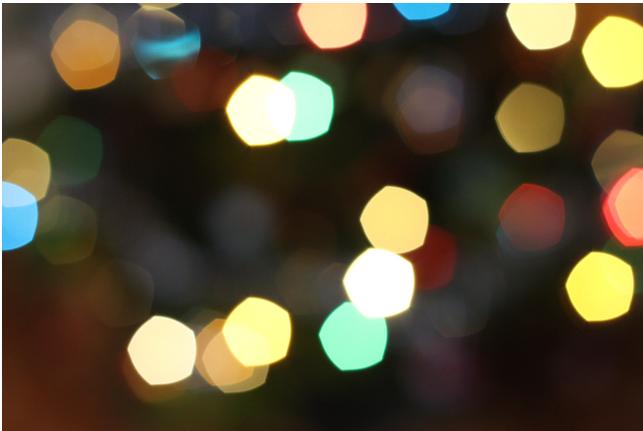


Figure 1.1: Real-life *bokeh*: the blurriness of out-of-focus objects will take the shape of the camera's aperture (pin-hole). Here, the *bokeh* is pentagonal.

1.1.4 Settings

The **Settings** class encapsulates all remaining options for customizing the way we render a scene:

- Picture resolution
- Gamma value
- Super-sampling factor
- Number of DOF samples
- Recursion depth

The meaning of these settings will further be explained in the section regarding ray tracing.

1.1.5 Class structure

1.2 Ray tracing

Ray:

$$\vec{o} + t\vec{r}$$

Sphere:

Ray-sphere intersection:

1.2.1 Process

1.2.2 Constructive solid geometry

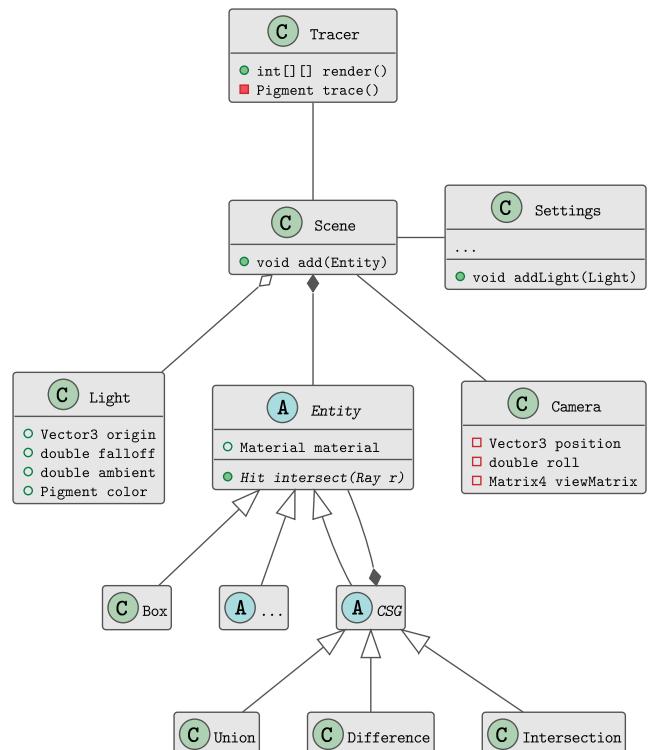


Figure 1.2: Rendering process class diagram

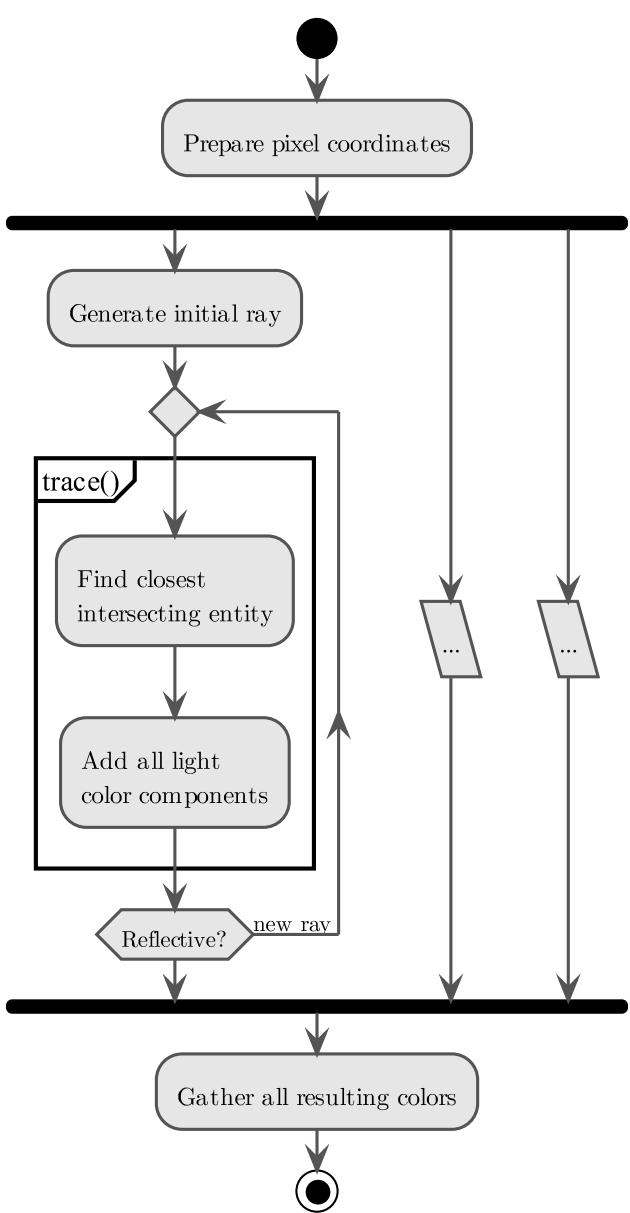


Figure 1.3: Rendering process activity diagram

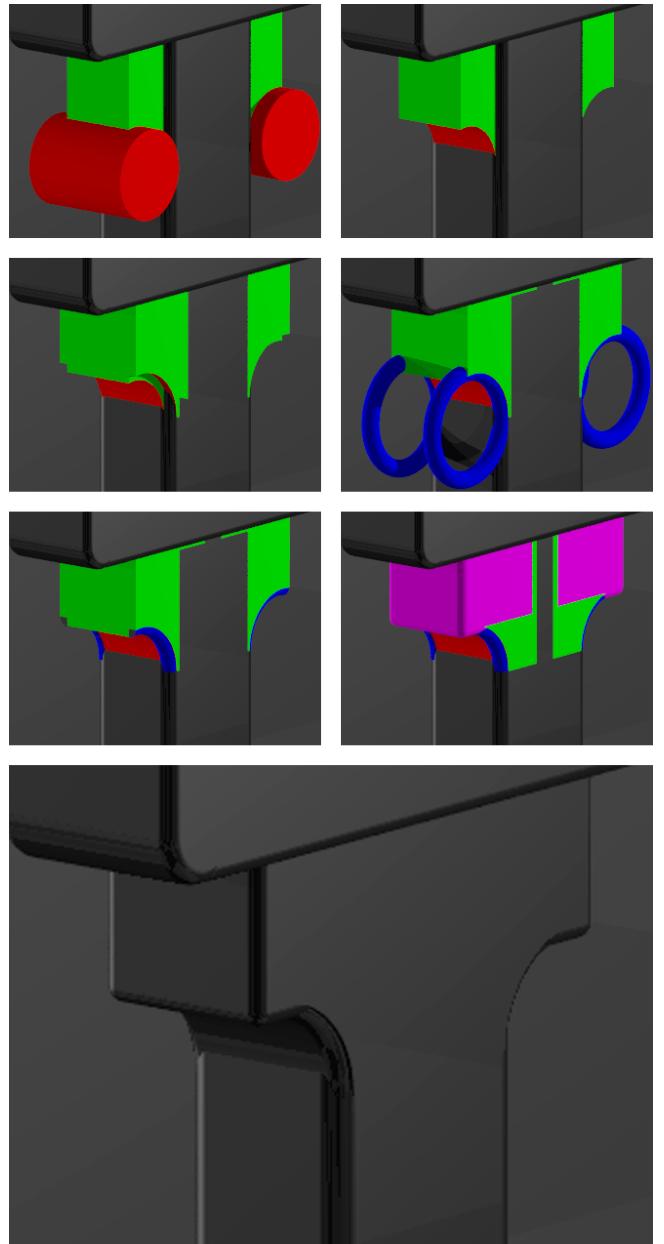


Figure 1.4: A piano foot obtained from CSG operations

2. Language

So far, we can compose and render scenes by directly writing them in Java, instancing `Scene` and `Entity` objects. But for the user to compose his own scenes, we need to define a language: the CRT scripting language.

...

- Imperative, no objects
- Procedures with parameters, no functions
- Variables can store entities
- Nesting scopes

2.1 ANTLR4

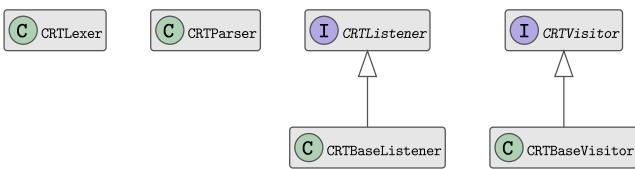


Figure 2.1: Classes generated by ANTLR4

2.2 Grammar

2.3 Compiling process

```
set title = "Example 01"
set author = "Tenchi (tenchi@team2xh.net)"
set date = "08.06.2014"
set notes = "Sample CRT scene in TRC language,"
            "language specification exploration."

# Sample camera
let cam = Camera(position -> vec3 (0.0, 0.5, -0.5),
                  pointing -> vec3 (0.0, 0.0, 0.0))

let redLight = Light
(
    position -> vec3 (1.0, 1.0, 1.0),
    color      -> rgb  (1.0, 0.9, 0.8)
)

let box1 = Box
(
    corner1 -> vec3 (-0.2, 0.0, -0.2),
```

```
        corner2 -> vec3 (0.2, 0.4, 0.2),
        color     -> rgb  (0.5, 0.5, 0.5)
    )

let sphere1 = Sphere
(
    origin -> vec3 (0.0, 0.4, 0.0),
    radius   -> 0.2
)

let tub = box1 - sphere1

# Ground
let plane1 = Plane
(
    normal   -> vec3 (0.0, 1.0, 0.0),
    position -> vec3 (0.0, 0.0, 0.0)
)

set settings = Settings
(
    gamma      -> 1.0,
    background -> rgb (1.0, 1.0, 1.0),
    camera     -> cam,
    lights     -> [redLight, blueLight]
)

scene {
    box1 <scale 3.0> - sphere1 <scale vec3 (1.0, 1.0, 2.0)>
    tub <scale 2.0, translate vec3 (1.0, 0.0, 1.0)>
}
```

A. Acknowledgements

Acknowledgements...

B. Appendix

Appendix...

C. Bibliography

Bibliography...