

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
List of Code Listings	iv
Foreword	v
1 Rendering process	1
1.1 Scenes	1
1.1.1 Entities	1
1.1.2 Light sources	1
1.1.3 Camera	2
1.1.3.1 Field of view	2
1.1.3.2 Camera matrix	2
1.1.4 Settings	2
1.1.5 Rendering process summary	2
1.2 Ray tracing	2
1.2.1 Process	3
1.2.2 Constructive solid geometry	4
1.2.2.1 Union	4
1.2.2.2 Difference	4
1.2.2.3 Intersection	4
1.2.3 Background projections	4
1.2.4 Depth of field	4
1.2.5 Materials	4
1.2.5.1 Procedural texturing	4
1.2.5.2 Bump mapping	4
1.2.5.3 UV Mapping	4
1.2.6 Misc	4
2 Language	5
2.1 ANTLR	5
2.2 Grammar	5
2.3 Compiling process	5
A Acknowledgements	6
B Appendix	7
B.1 Mathematical helper classes	7
B.2 GUI	7
C Bibliography	8

List of Figures

1.1	The Entity class diagram	1
1.2	Real-life <i>bokeh</i>	2
1.3	Rendering process class diagram	2
1.4	Rendering process activity diagram	3
1.5	A piano foot obtained from CSG operations	3
2.1	Family of classes generated by ANTLR4	5

List of Tables

List of Code Listings

2.1	Sample CRT script	5
2.2	Generating a parse tree and compiling	5

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, this chapter will outline the classes representing a scene, all designed in an object-oriented style, using common design patterns when relevant.

We will then concentrate on how ray tracing – the technique used for rendering – works: the physics and mathematics involved, common light interactions, and CSG operations.

1.1 Scenes

We call “scene” the composition of *elements* and *parameters* that, after the rendering process is finished, define what the final image looks like. In CRT, a scene is represented by the `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, the objects composing the rendered world
- A list of light sources
- A camera
- Other settings, stored in a `Settings` object

1.1.1 Entities

Entities are **primitive volumes** that can easily be described with *mathematical equations*, such as boxes (*parallelepipeds*), spheres, cones, planes and half-planes, tori, etc.

Every entity has a position in space and must provide an `intersect()` method to compute its eventual intersection point or points with any given ray, which we will need later on to do the rendering.

Entities also contain a `Material` property, which defines what material the entity is made out 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.

Thinking about ability to compose creative scenes, one can ask: “Isn’t only having *cubes and spheres* a bit limited?” To remedy this, users can compose groups of entities using the result of a *CSG¹ operation*, which can be either a union, a difference or an intersection.

All of these operations will be explained in further details in the section on ray tracing.

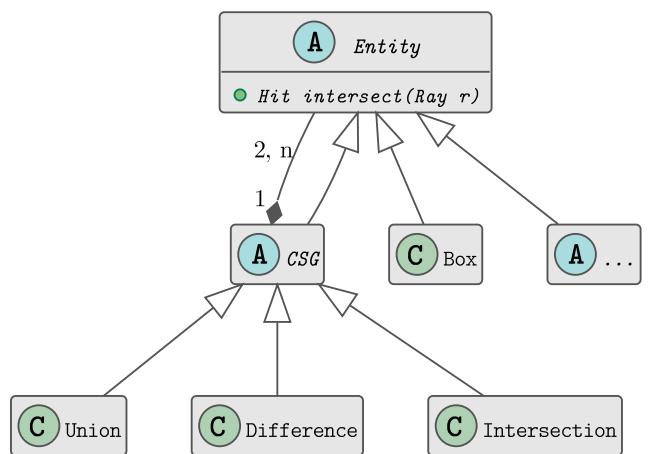


Figure 1.1: The `Entity` class diagram

We can notice that the `CSG` operators follow the *composite* design pattern, being an entity type composed of other entities.

1.1.2 Light sources

Light sources illuminate a scene and give entities a component of their colors. When ray tracing, they are the targets of all the rays we *back-trace* from the camera lens and 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

¹Constructive solid geometry

- 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 off many surfaces) by setting an ambient factor, which will basically add a fraction of the value of its color and intensity.

One has to keep in mind that each additional light source adds up to the amount of rays to bounce and thus linearly increase computation time.

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), effect that creates a plane in which objects are sharp, and blurry outside, akin to a tilt-shift effect in photography.
- 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 1.2: Real-life *bokeh*: the blurriness of out-of-focus objects will take the shape of the camera's aperture (pinhole). Here, the *bokeh* is pentagonal.

1.1.3.1 Field of view

1.1.3.2 Camera matrix

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 Rendering process summary

In the following class diagram are all the main classes involved in the rendering of a scene. The `Tracer` class contains the static methods responsible for the actual ray tracing. They are invoked with a `Scene` object as a parameter, which contains references to all of the other classes.

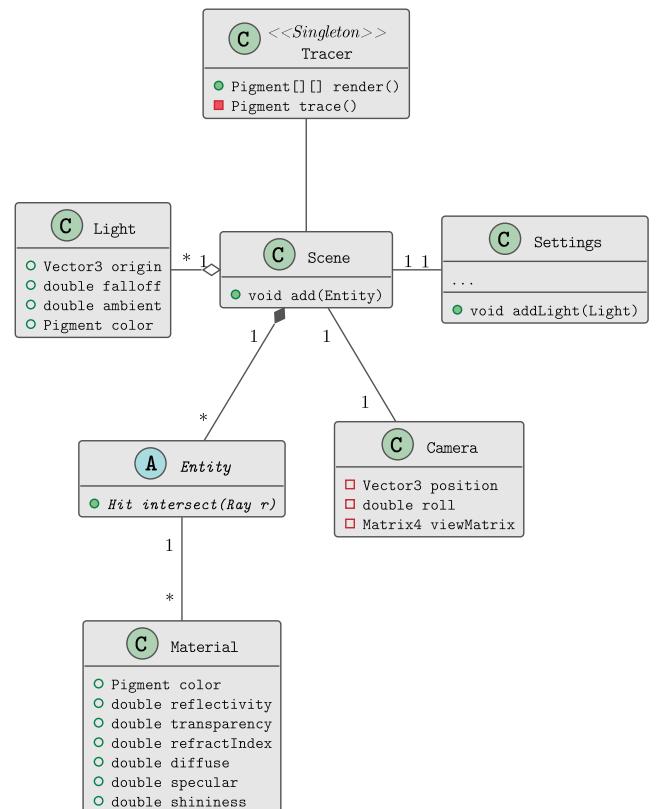


Figure 1.3: Rendering process class diagram

1.2 Ray tracing

- Reverse path of a light ray
- Accumulate all colors along the way
- Recursion
- 3d diagram with virtual screen and pixels

Ray:

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

Sphere: Ray-sphere intersection:

- Diagram

1.2.1 Process

- Parallel via Java 8
- Find closest
- Keep distance in memory for falloff
- For each light
 - Find if intersection point is hit by light
 - Compute color
 - Bounce if reflective, recurse

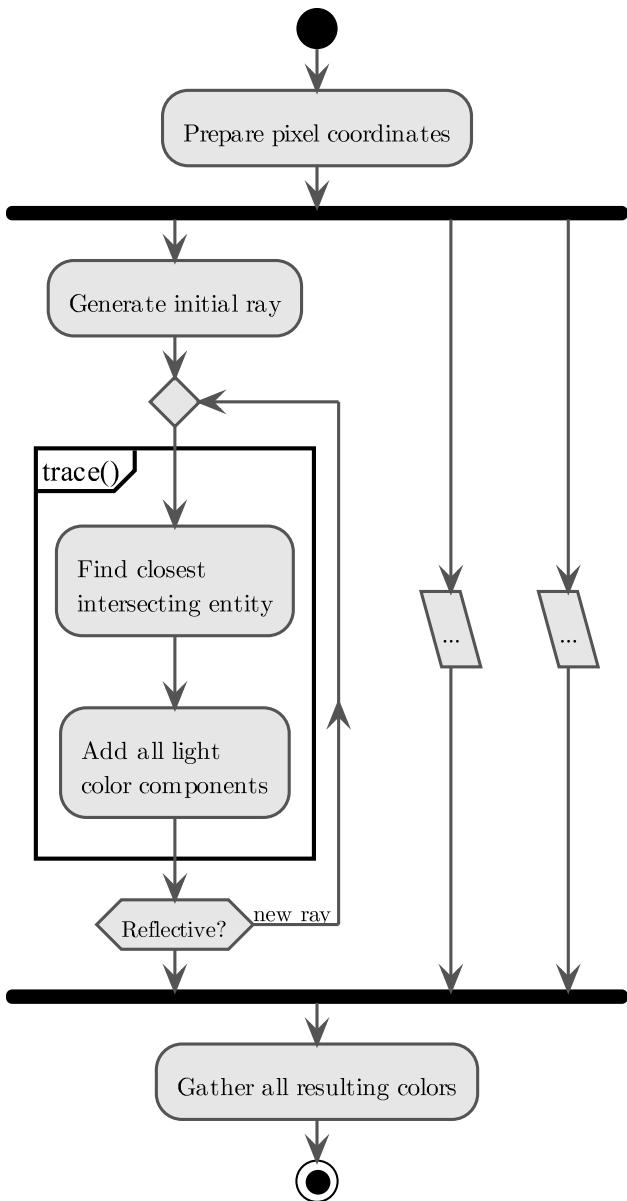


Figure 1.4: Rendering process activity diagram

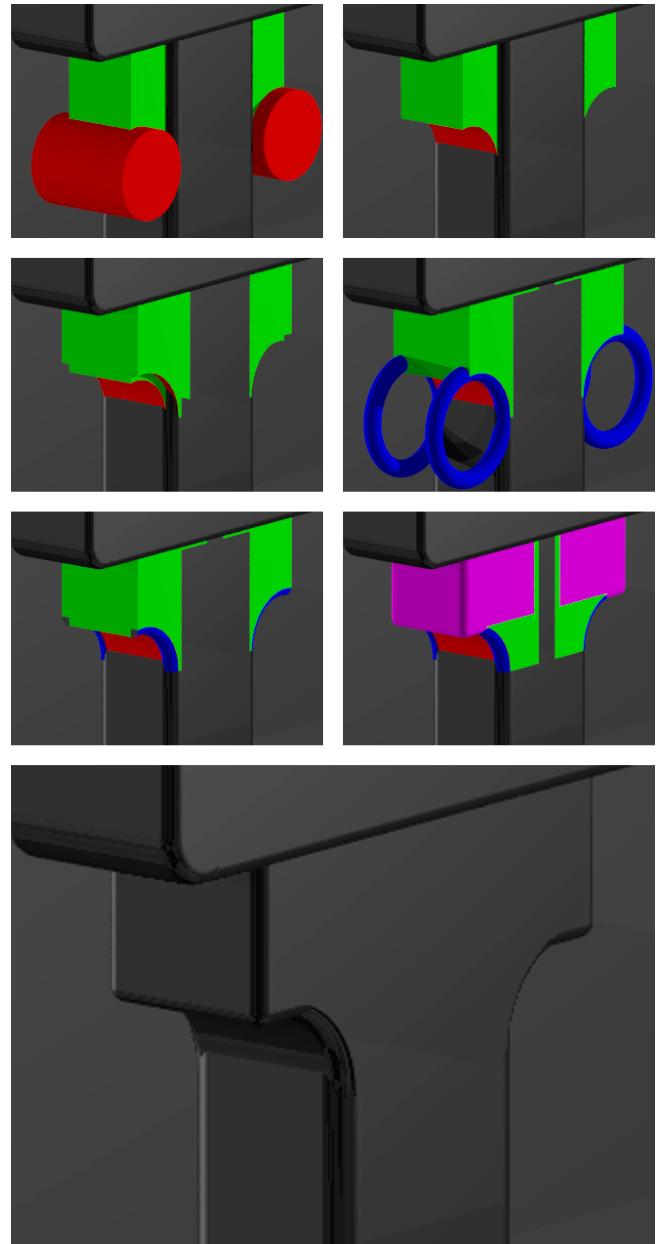


Figure 1.5: A piano foot obtained from CSG operations

1.2.2 Constructive solid geometry

1.2.2.1 Union

1.2.2.2 Difference

1.2.2.3 Intersection

1.2.3 Background projections

- Take other 3d diagram and apply projection
- List projections

1.2.4 Depth of field

- Vector shifting diagram
- Aperture shape diagrams
- Effect of aperture
- Effect of focal distance
- Number of DOF samples
- List shapes with a few pics

1.2.5 Materials

1.2.5.1 Procedural texturing

1.2.5.2 Bump mapping

1.2.5.3 UV Mapping

1.2.6 Misc

- Gamma value
- Super-sampling factor

2. Language

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

The CRT scripting language follows an *imperative* paradigm and aims to be simple yet permissive enough to enable creativity.

It features two block types for describing a scene and its settings, variables that can store entities, literal values, and point to other variables, parametric procedures (hereinafter referred to as “*macros*”) with nested scopes but no return value, and entity modifiers for affine transformations.

Visually as well as syntactically, the language tries to be simple on the eyes, with no end-of-statement terminator. Here is a sample of what it looks like:

```
1 --Example-----
2
3 myObject = Object {
4     attribute1 -> vec3(0.0, 0.5*3, -0.5)
5     attribute2 -> "foobar"
6     attributeList -> [true, true, false]
7 }
8
9 n = 18
10 max = (3 * n) / 4 + 5
11
12 myMacro = Macro (arg1) {
13     i = 0
14     -- Draw myObject "max" times
15     while (i < max) {
16         myObject <translate vec3(i*5.0, 0.0, 0.0)>
17         i = i + 1
18     }
19 }
```

Code Listing 2.1: Sample CRT script

2.1 ANTLR

The language’s grammar will be designed in EBNF using the G4 syntax from **ANTLR**¹, a Java parser generator. ANTLR will then use that grammar specification file to automatically generate a lexer, a parser, and base classes for implementing tree traversal using design patterns such as *listeners* and *visitors*.

For our compiler, we will use the *visitor* pattern, which allows more control over the tree traversal; the listener provided by antler automatically traverses the tree whereas the visitor forces manual traversal implementation.

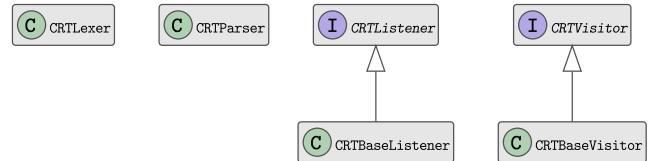


Figure 2.1: Family of classes generated by ANTLR4

Using the generated lexer and parser, we can produce a parse tree (lines 3-6). Then, using custom-made visitors, we can visit each node of the tree to compile the code to a `Script` object (line 8):

```
1 String code = "..."
2
3 CRTLexer lexer = new CRTLexer(new ANTLRInputStream(code));
4 CommonTokenStream tokens = new CommonTokenStream(lexer);
5 CRTParser parser = new CRTParser(tokens);
6 ParseTree tree = parser.script();
7
8 Compiler compiler = new Compiler(code);
9 Script script = compiler.visit(tree);
```

Code Listing 2.2: Generating a parse tree and compiling

2.2 Grammar

The designed grammar is non-ambiguous (context-free), but uses left-recursion² for ease of writing *and* reading, which ANTLR allows since version 4.2.

Some parts were inspired from example grammars provided by the ANTLR team on GitHub, in particular the Java grammar³.

The “ANTLR 4 IDE” Eclipse plug-in⁴ proved very useful during the development of the grammar. It provides useful tools for debugging such as syntax diagrams and live parse tree visualisation.

Syntax diagram screenshot

Priority of operations table

Alternative labelling generates one visitor method per label

No business logic in grammar

2.3 Compiling process

²http://en.wikipedia.org/wiki/Left_recursion

³<http://github.com/antlr/grammars-v4/blob/master/java/Java.g4>

⁴<http://github.com/jknack/antlr4ide>

¹**A**nother **T**ool for **L**anguage **R**ecognition

A. Acknowledgements

Acknowledgements...

B. Appendix

B.1 Mathematical helper classes

- `Matrix4`
- `Vector3`
- Poisson disk distribution
 - Explanation
 - Nice diagrams
 - Explain why it's slow and not so useful
- Uniform Distribution
 - Explanation
 - Nice diagrams
 - Explain why it's nice
 - Credits to J.-F. Hêche

B.2 GUI

- Substance
- `GUIToolkit`

C. Bibliography

Bibliography...