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

List of Figures

List of Tables

List of Code Listings

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 *CSG*[1] *operation*, which can be either a union, a difference or an intersection.

All of these operations will be explained in further details in section 1.2 about ray tracing.

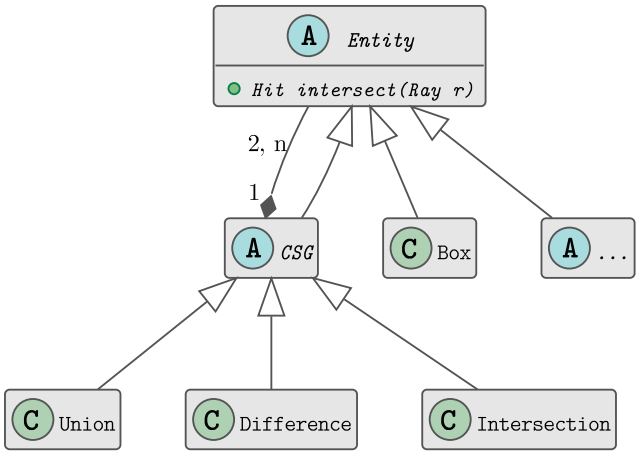

Figure 1.1: The `Entity` class diagram

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.

Several light source types exist: spotlight (cone), cylinder, parallel, and point. For now, only point light sources are implemented.

[1] Constructive solid geometry

A light source is defined by the `Light` class and has the following properties:

- An 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.

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 1.2).
- A focal distance, defining at which distance objects are sharp.

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.

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.

1.1.5 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.

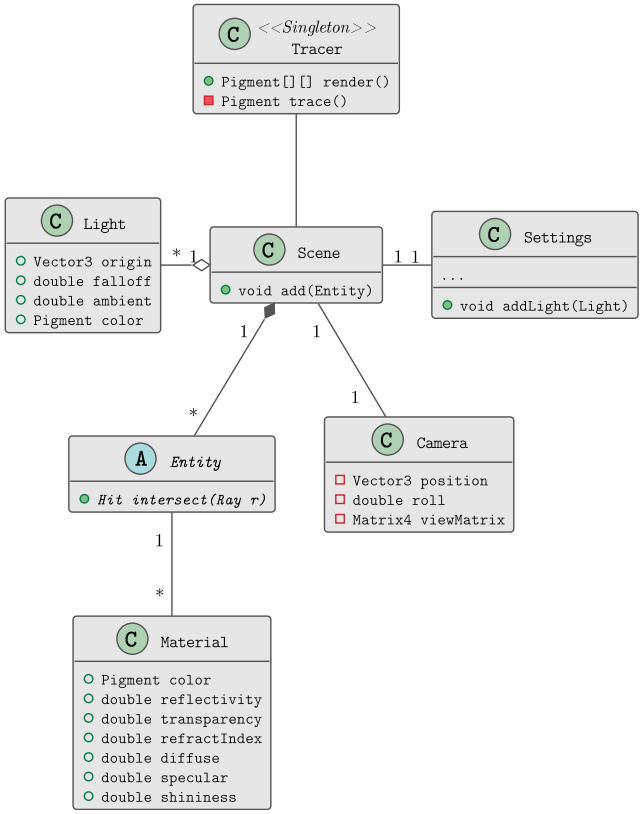

Figure 1.3: Rendering process class diagram

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

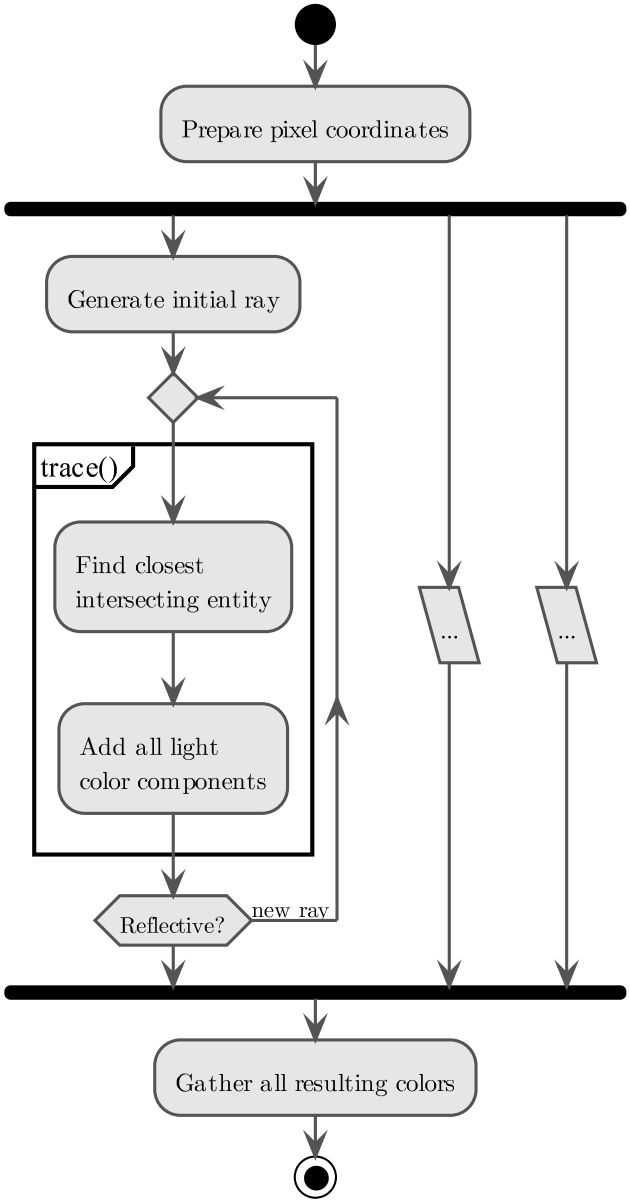

Figure 1.4: Rendering process activity diagram

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

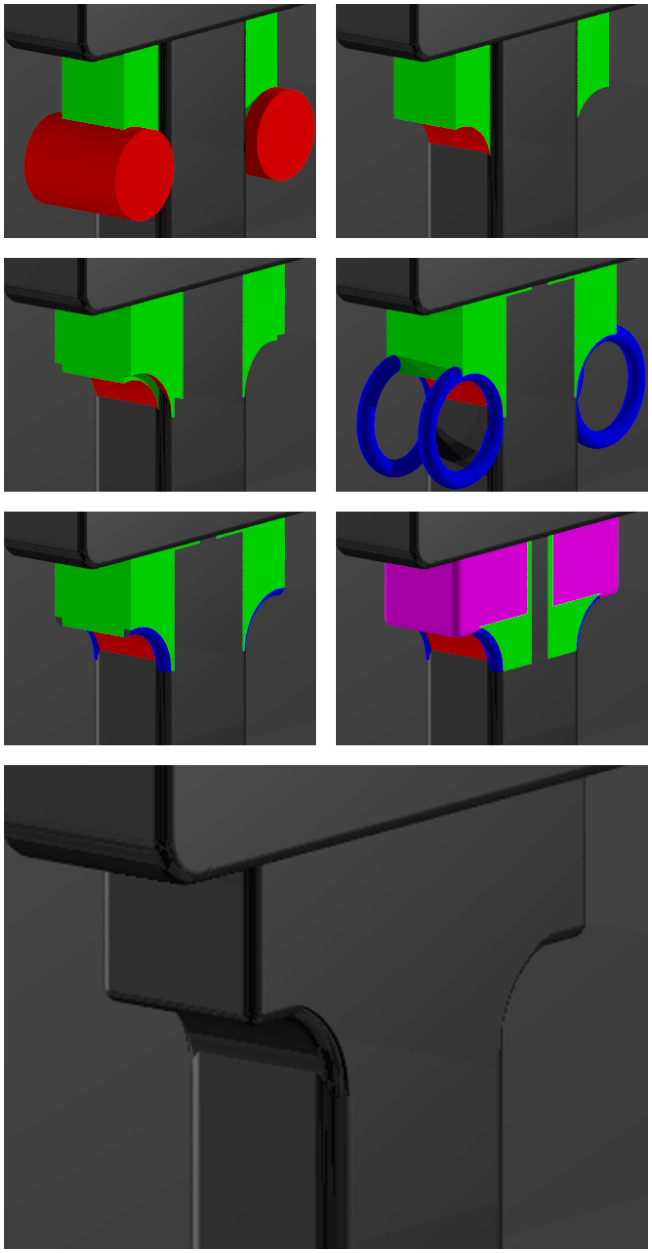

Figure 1.5: A piano foot obtained from CSG operations

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:

```
--Example-------------------------------------------------

myObject = Object {
    attribute1    -> vec3(0.0, 0.5*3, -0.5)
    attribute2    -> "foobar"
    attributeList -> [true,  true, false]
}

n = 18
max = (3 * n) / 4 + 5

myMacro = Macro (arg1) {
    i = 0
    -- Draw myObject "max" times
    while (i < max) {
        myObject <translate vec3(i*5.0, 0.0, 0.0)>
        i = i - 1
    }
}
```

Code Listing 2.1: Sample CRT script

2.1 ANTLR

The language's grammar will be designed in a EBNF variant, the G4 syntax from **ANTLR**[2], 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.

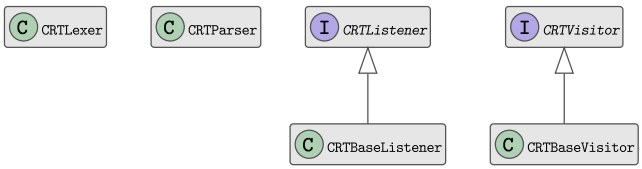

Figure 2.1: Family of classes generated by ANTLR4

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-recursion**[3] 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 grammar[4], from which much was learned about left-recursion and operator precedence.

The "ANTLR 4 IDE" Eclipse plug-in[5] proved to be very useful during the development of the grammar. It provides useful tools for debugging such as syntax diagrams and live parse tree visualisation — just choose a grammar rule, type in code and a corresponding parse tree is updated at every keystroke.

The grammar contains no special verifications, which happen at compile time. This makes the grammar *much* more readable and easy to understand. Also, ANTLR provides a feature for labelling the alternatives of a rule, which it will use

[2] **AN**other **T**ool for **L**anguage **R**ecognition

[3] http://en.wikipedia.org/wiki/Left_recursion

[4] http://github.com/antlr/grammars-v4/blob/master/java/Java.g4

[5] http://github.com/jknack/antlr4ide

for code generation where it will generate one visitor method per label (e.g. instead of having to implement a very extensive `visitExpression()` method, it will be broken down to all its alternatives `visitAddition()`, `visitMultiplication()` etc.).

2.2.1 Rules

This section lists all the grammar rules defined in the `CRT.g4` file, in a **BNF** notation, followed by a quick overview of how they work.

Nonterminal names are enclosed within angled brackets (⟨…⟩). Names starting with a capital are rules, small letter are token types (which are not defined here).

⟨*Script*⟩ ::= ⟨*Statement*⟩* (1)

⟨*Statement*⟩ ::= (⟨*Settings*⟩ | ⟨*Scene*⟩ | ⟨*Expr*⟩) (2)

⟨*Settings*⟩ ::= `Settings {` ⟨*Attribute*⟩* `}` (3)

⟨*Scene*⟩ ::= `Scene {` ⟨*Expr*⟩* `}` (4)

⟨*Expr*⟩ ::= ⟨*Primary*⟩ (5)
| ⟨*Object*⟩ (6)
| ⟨*Macro*⟩ (7)
| `[` ⟨*ExpressionList*⟩? `]` (8)
| ⟨*Expr*⟩ `[` ⟨*Expr*⟩ `]` (9)
| ⟨*Expr*⟩ `(` ⟨*ExprList*⟩? `)` (10)
| ⟨*Expr*⟩ `<` ⟨*Modifier*⟩ (`,` ⟨*Modifier*⟩)* `>` (11)
| (`+` | `-`) ⟨*Expr*⟩ (12)
| `!` ⟨*Expr*⟩ (13)
| ⟨*Expr*⟩ (`*` | `/` | `%`) ⟨*Expr*⟩ (14)
| ⟨*Expr*⟩ (`+` | `-` | `^`) ⟨*Expr*⟩ (15)
| ⟨*Expr*⟩ (`<=` | `>=` | `<` | `>` | `==` | `!=`) ⟨*Expr*⟩ (16)
| ⟨*Expr*⟩ `&&` ⟨*Expr*⟩ (17)
| ⟨*Expr*⟩ `||` ⟨*Expr*⟩ (18)
| ⟨*Expr*⟩ `?` ⟨*Expr*⟩ `:` ⟨*Expr*⟩ (19)
| ⟨*Expr*⟩ `=` ⟨*Expr*⟩ (20)

⟨*ExprList*⟩ ::= ⟨*Expr*⟩ (`,` ⟨*Expr*⟩)* (21)

⟨*Primary*⟩ ::= `(` ⟨*Expr*⟩ `)` (22)
| ⟨*Literal*⟩ (23)
| ⟨*identifier*⟩ (24)

⟨*Object*⟩ ::= ⟨*name*⟩ `{` ⟨*Attribute*⟩* `}` (25)

⟨*Macro*⟩ ::= `Macro (` ⟨*ParamList*⟩? `) {` ⟨*Expr*⟩* `}` (26)

⟨*ParamList*⟩ ::= ⟨*identifier*⟩ (`,` ⟨*identifier*⟩)* (27)

⟨*Literal*⟩ ::= (⟨*integer*⟩ | ⟨*float*⟩ | ⟨*string*⟩ | ⟨*boolean*⟩) (28)

⟨*Attribute*⟩ ::= ⟨*identifier*⟩ `->` ⟨*Expr*⟩ (29)

⟨*Modifier*⟩ ::= `scale` ⟨*Expr*⟩ (30)
| `translate` ⟨*Expr*⟩ (31)
| `rotate` ⟨*Expr*⟩ (32)

A **script** (1) is a set of **statements** (2), which can either be settings blocks, scene blocks, or expressions.

Settings and **scene** blocks (3, 4) are expressed using their names followed by braces containing either a number of attributes, or expressions — this difference existing because settings have defined names to which we can assign values, and a scene renders all contained expressions that resolve to an entity (see section 1.1.1).

An **expression** is either a primary type (5), an object (6), a macro (7), or one of the following:

(8) List of *heterogeneous* expressions (21)
(9) List access
(10) Macro call, which takes an optional list of expressions (21) as formal parameters
(11) Entity modified with an affine transformation
(12) Sign unary operators
(13) Negation boolean unary operator
(14) Multiplication, division and modulo operators
(15) Addition and subtraction operators. If both operands are entities, the operators are instead the CSG union (`+`), difference (`-`) and intersection (`^`)
(16) Boolean comparison operators
(17) Boolean conjunction operator
(18) Boolean disjunction operator
(19) Ternary operator
(20) Assignment operator

A **primary** type is either a parenthesised expression (22), a literal type (23) or an identifier (24) — a token made of alphabetical characters starting with a small letter.

An **object** (25) has a name — a token made of alphabetical characters starting with a capital letter — and is followed by a brace separated block of attributes.

A **macro** (26) starts with the word `Macro` and a list of formal parameters (27), followed by a brace separated block of expressions.

A **literal** type (28) can be one of four token types:

- A whole number
- A decimal number
- A string of characters inside straight double quotes
- A boolean value (the words `true` or `false`)

Attributes (29) are identifier tokens followed by an arrow (`->`) and an expression.

Finally, **modifiers** (which apply an affine transformation to an entity) can either be a scaling operation (30), a translation (31) or a rotation (32).

Without ANTLR's compatibility with left-recursion, most of the rules referencing expressions would have to be written in such a way that the grammar is only read from left to right, involving a *lot* more rules.

2.2.2 Operators

Priority of operations table

2.3 Compiling process

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...