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, 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), 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 1.1: 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 Class structure

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 all of the other classes. Also, we can notice that the `CSG` operators follow the *composite* design pattern, being an `Entity` composed of other `Entity`.

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:

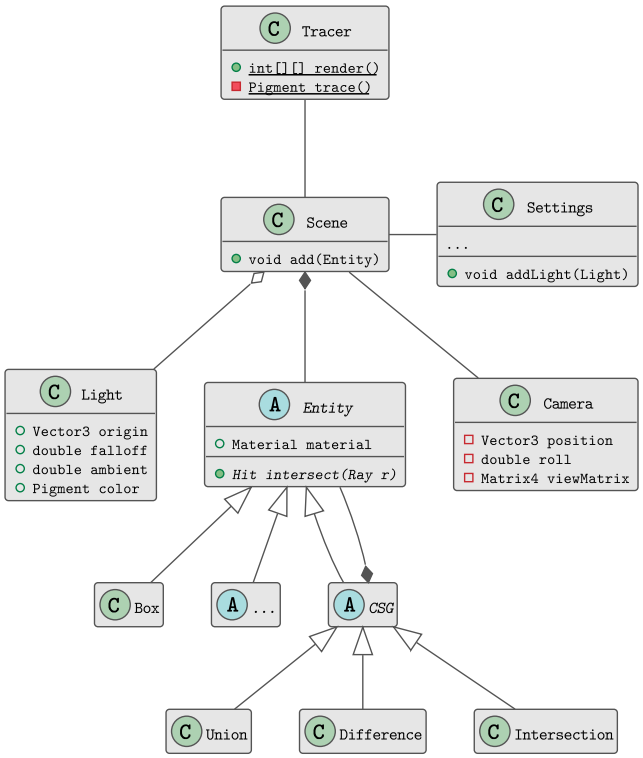

Figure 1.2: Rendering process class diagram

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

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

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

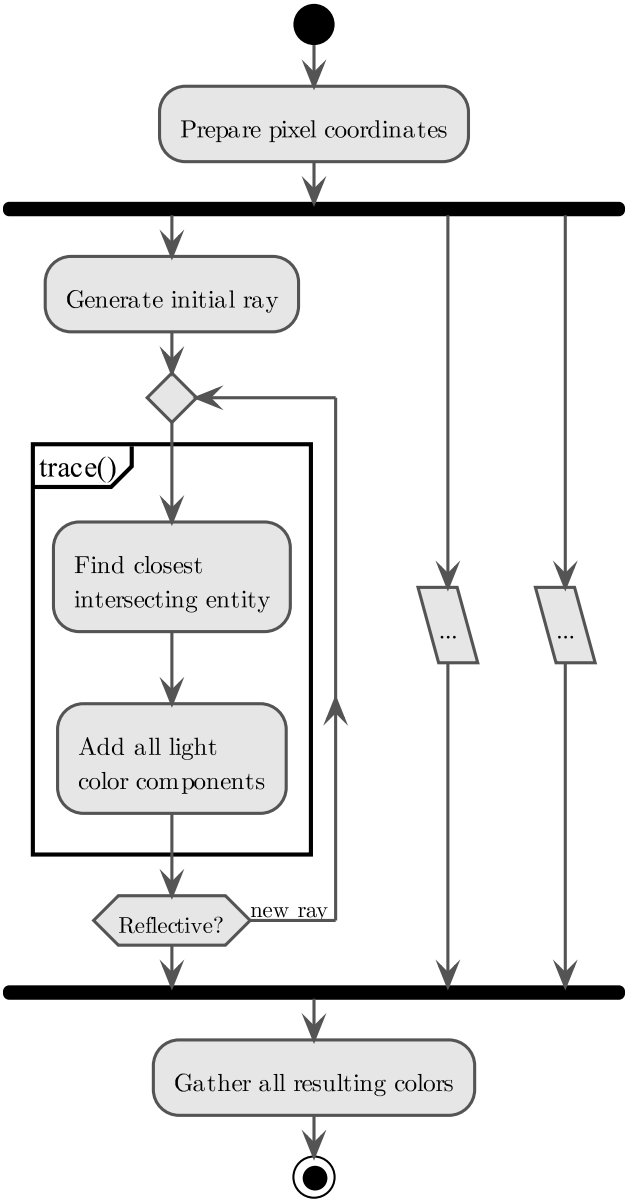

Figure 1.3: Rendering process activity diagram

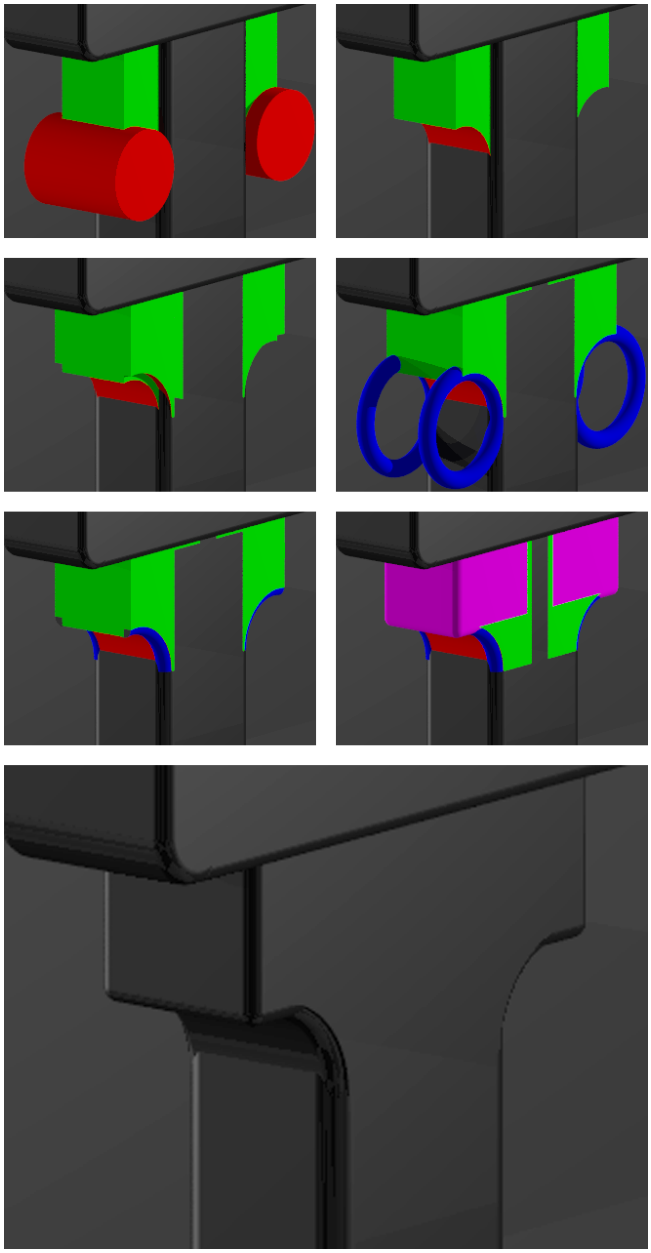

Figure 1.4: A piano foot obtained from CSG operations

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 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.

The CRT scripting language follows an *imperative* paradigm and aims to be simple yet permissive enough to enable creativity. It features two bloc types for describing a scene settings and its content, variables that can store entities and numeric values, parametric procedures with nested scopes (no functions), and entity modifiers for affine transformations.

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 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*.

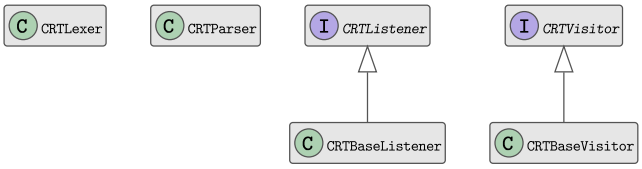

Figure 2.1: Family of classes generated by ANTLR4

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 `Scene` 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.program();
7
8 Scene scene = new CompilerVisitor().visit(tree);
```

Code Listing 2.1: Generating a parse tree and compiling it

2.2 Grammar

2.3 Compiling process

[1] **AN**other **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...