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 ii

List of tables ii

List of code listings ii

Foreword iii

1 Rendering process 1

1.1 Scenes . . . 1

1.1.1 Entities . . . 1

1.1.2 Light sources . . . 1

1.1.3 Camera . . . 2

1.1.4 Settings . . . 2

1.1.5 Rendering process summary . . . 2

1.2 Ray tracing . . . 3

1.2.1 Backward tracing . . . 3

1.2.2 Implemented algorithm . . . 4

1.2.3 Bonus features . . . 4

1.2.4 Primitives . . . 4

1.2.5 Constructive solid geometry . . . 5

1.2.6 Background projections . . . 5

1.2.7 Depth of field . . . 5

1.2.8 Materials . . . 5

1.2.8.1 Procedural texturing . . . 5

1.2.8.2 Bump mapping . . . 5

1.2.8.3 UV Mapping . . . 5

1.2.9 Misc . . . 5

2 Language 6

2.1 ANTLR . . . 6

2.2 Grammar . . . 6

2.2.1 Rules . . . 7

2.2.2 Operators . . . 8

2.3 Compiling process . . . 8

A Acknowledgements 9

B Appendix 10

B.1 Mathematical helper classes . . . 10

B.2 GUI . . . 10

C Bibliography **11**

List of figures

1.1 The `Entity` class diagram 1
1.2 Real-life *bokeh* 2
1.3 Rendering process class diagram 2
1.4 Rasterisation of a triangle 3
1.5 Backtracing light rays 4
1.6 Rendering process activity diagram 4
1.7 A piano foot obtained from CSG operations 5

2.1 Family of classes generated by ANTLR4 6
2.2 ANTLR's language recognition process 6

List of tables

2.1 List of CRT operators 8

List of code listings

2.1 Sample CRT script 6
2.2 Generating a parse tree and compiling 6

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 colour, 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.

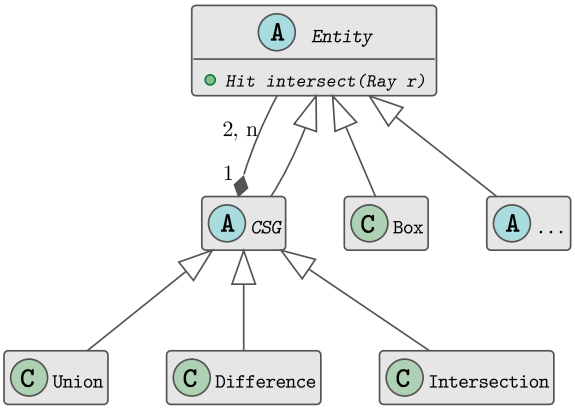

Figure 1.1: The `Entity` class diagram

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.

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

1. Constructive solid geometry

- 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 colour, 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 colour 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** and a **direction** vector, as well as a **field of view** angle.

The **field of view** of a camera *how much* it sees from left to right, or from top to bottom of the image (in photography, this would represent the focal length of the objective). In CRT, the field of view is defined vertically, as an angle in radians. Varying this parameter has a *zoom* effect when lowered, while a big value makes more things visible on the screen.

To further add to the user's creative possibilities, several artistic features which aim to mimic real-life cameras were implemented:

- 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.

1.1.4 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

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.

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.

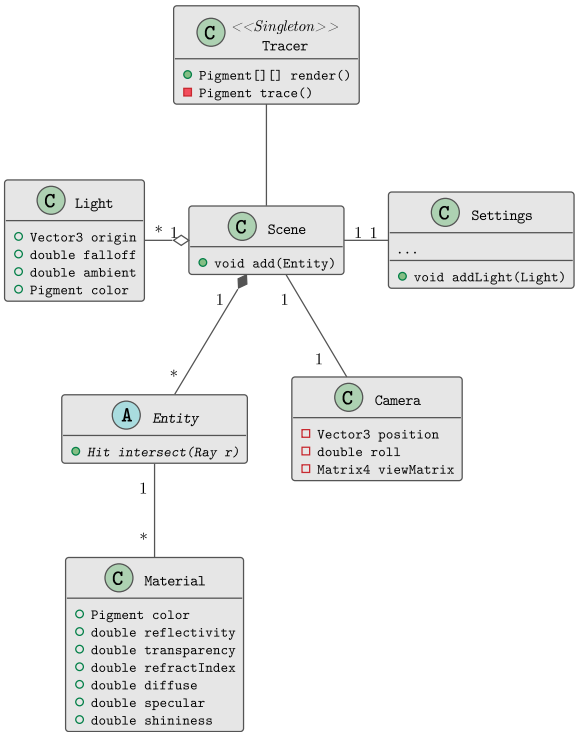

Figure 1.3: Rendering process class diagram

1.2 Ray tracing

A lot of elements were defined thus far — but just what *is* ray tracing? First, a bit of history of computer graphics.

Traditionally, 3D computer graphics are rendered using a technique called **rasterisation**. Compared to ray tracing, rasterisation is extremely fast and is more suited for real-time applications, and takes advantage of years of hardware development dedicated to accelerating it.

In the rasterisation world, a 3D scene is described by a collection of **polygons**, usually triangles, defined by 3 three-dimensional vertices. A rasteriser will take a stream of such vertices, transform them into corresponding two-dimensional points on the viewer's monitor, and fill in the transformed two-dimensional triangles (with either lines, or colours).

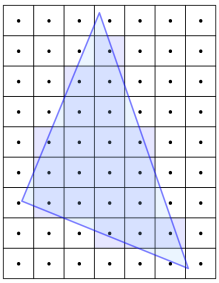

Figure 1.4: Rasterisation of a triangle: once the vertices have been projected on the screen, a discrete pixel is "lit" if its continuous centre is contained within the projected triangle's boundaries

Some effects of *light* observed in real life can be reproduce (or at least *mimicked*) on top of rasterisation. For example, if a polygon is not directly facing the camera (i.e. its *normal vector* is not parallel with the camera's direction), the resulting colour of the rasterised triangle will be darker.

However, the very nature of rasterisation makes it hard to implement other very common effects:

- To reproduce shadows, complicated stencil buffers must be used, along with a depth buffer computed by rendering a sub-scene from the point of view of the light source. This not only is complex but the results look very pixelated
- Refraction is very hard to reproduce. For a long time, raster application went without refraction effects and just have less opaque models. Nowadays, advanced pixel shaders use techniques similar to ray tracing

Ray tracing solves these issues, at the cost of being slower.

Instead of projecting things *from* the scene on the screen like with rasterisation, ray tracing is about *sending* rays *towards* the various elements of the scene.

But why this way? In real life, light sources send protons in all directions at random. Some of them hit objects, which *absorb* some of the energy from the photons (thus changing the perceived colour). The photons are then reflected, bouncing *off* the object with a mirrored angle of incidence[2].

An ideal ray tracer simulating real life would instead send rays *from* the light sources *onto* the subjected surfaces, but this is in reality not practical and one would have to wait a very long time for an image to render; the probability of a light ray coming out of a source in a *random* direction, hitting an object, bouncing off that object in another *random* direction, and finally hitting the camera is *very* small.

In real life, our human eyes still manage to see photons because there is just *too many* of them. Let's count how many photons per second are emitted by a typical 100 W ($100\,\mathrm{J\,s^{-1}}$) lightbulb with an average wavelength of 600 nm:

$$E_{\text{photon}} = hf = \frac{hc}{\lambda} \approx 3 \times 10^{-19}\,\mathrm{J} \tag{1.1}$$

$$\frac{P_{\text{lightbulb}}}{E_{\text{photon}}} = \frac{100\,\mathrm{J\,s^{-1}}}{3 \times 10^{-19}\,\mathrm{J}} \approx 3 \times 10^{20}\,\mathrm{s^{-1}} \tag{1.2}$$

So, just for a normal lightbulb, approximately **300 billion billion** photons are emitted *every second*. In *all possible* directions. And then hit objects, bounce in *all possible* directions again, hit other objects etc., and finally hit the observer's eye. Add to this the fact that because of the *inverse square law*, the further the observer is from a light source, the less photons per square metre he receives, in a quadratic fashion. This makes this model *very* impractical to use.

Just for comparison, a good computer has a power on the order of 10 GFLOPS, that is 10 billion operations per second. To come close to computing as many operations per seconds as photons emitted per second by a light bulb, a good computer would have to be 10^{10} times faster.

1.2.1 Backward tracing

This computational problem has lead computer graphics developers to invent **backward tracing**, where light rays are traced *from* the camera back to the light source. In a best-case scenario, only *one* ray projection is needed per pixel.

The basic idea, demonstrated visually in figure 1.5, is as follows:

- For each pixel of the screen, send a light ray from an imaginary point (the camera position) into the middle of that pixel
- If the ray hits an object, send a new ray (called *shadow* ray) in the direction of all light sources:

2. Note that is angle is generally not exactly the mirrored incident angle and is in fact mostly random. Perfect surfaces like mirrors will indeed bounce off photons with a perfect angle (**specular** reflection), but most surfaces will scatter the protons in all directions (**diffuse** reflection — that is why stones are not reflective like a mirror, their surface is *rough* all incoming photons are dispersed)

  - If the shadow ray hits no object in its way to the light source, then the object is lit by it. The light source's colour and the object's are mixed together and returned to the pixel
  - If the shadow ray hits another object, then the first object must be in shadow (the light source hits the other object first), so no colour is added
- If the hit object is reflective and the recursion limit is not reached, recursively trace a mirrored ray (the mirrored angle is 2 times the dot product between the original ray and the surface normal)

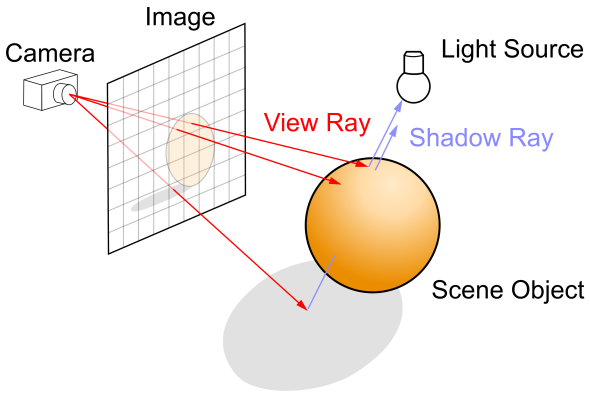

Figure 1.5: Backtracing light rays

1.2.2 Implemented algorithm

- 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

1.2.3 Bonus features

- Supersampling

1.2.4 Primitives

Ray:

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

Sphere: Ray-sphere intersection:

- Diagram

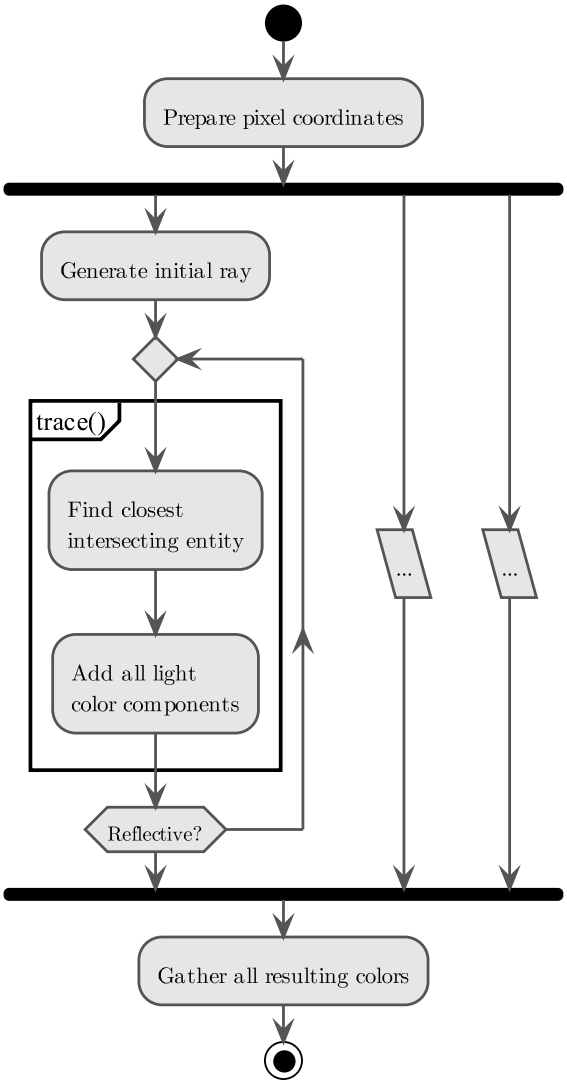

Figure 1.6: Rendering process activity diagram

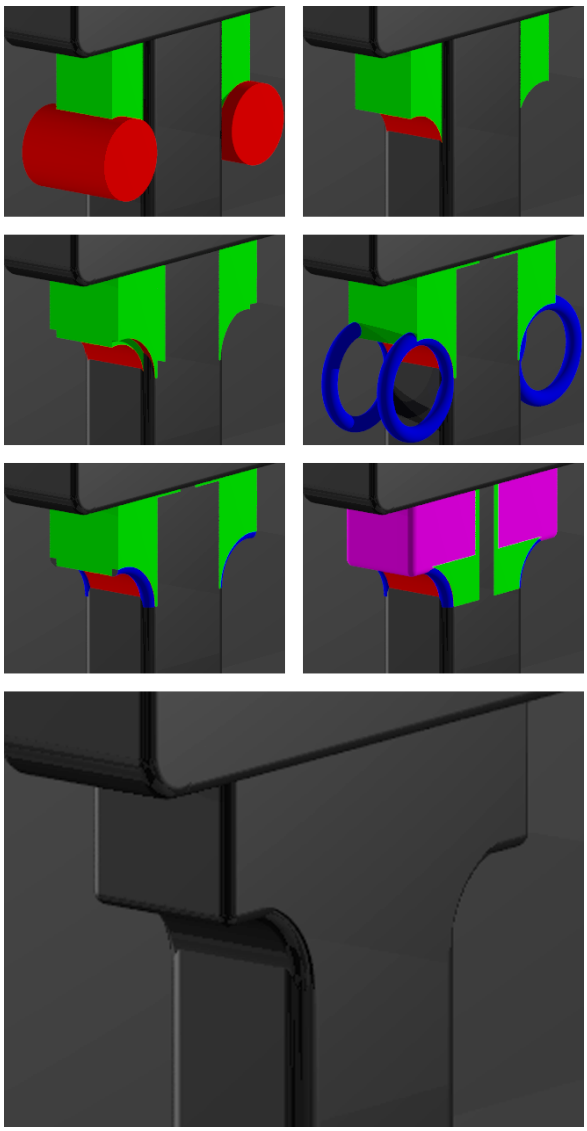

Figure 1.7: A piano foot obtained from CSG operations

1.2.5 Constructive solid geometry

(Union)

(Difference)

(Intersection)

1.2.6 Background projections

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

1.2.7 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

1.2.8 Materials

1.2.8.1 Procedural texturing

1.2.8.2 Bump mapping

1.2.8.3 UV Mapping

1.2.9 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 a EBNF variant, the G4 syntax from **ANTLR**[3], a Java parser generator.

ANTLR will use that grammar specification to automatically generate the code of a lexer, a parser, and base classes useful for implementing tree traversal using design patterns such as *listeners* and *visitors*.

ANTLR works by first lexing the code into *tokens*, defined by their types in the grammar (e.g. names, identifiers, symbols, etc.) then parsing those tokens using the grammar *rules*, producing a parse tree where all the leaf nodes are tokens.

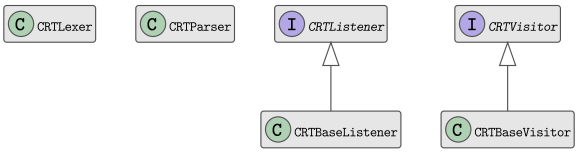

Figure 2.1: Family of classes generated by ANTLR4

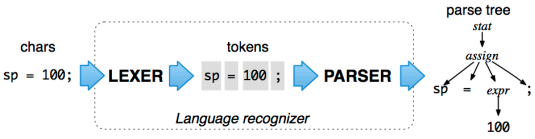

Figure 2.2: ANTLR's language recognition process

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.

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**[4] 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 gram-

3. **AN**other **T**ool for **L**anguage **R**ecognition

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

mar[5], from which much was learned about left-recursion and operator precedence.

Furthermore, the "ANTLR 4 IDE" *Eclipse* plug-in[6] 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.

A similar (and official) plug-in also exists for *Netbeans*, the main IDE used during the development of this project, however it was not compatible with the latest versions of Netbeans.

Because it is important to make a separation between parsing and compiling, the grammar contains no special verifications; they are done 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 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.

⟨*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) Access list element
(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.

5. http://github.com/antlr/grammars-v4/blob/master/java/Java.g4
6. http://github.com/jknack/antlr4ide

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

Because we used *left-recursion* to write the grammar, the operator precedence is visually clear at first sight — however, for the sake of completeness, table 2.1 shows all operators, their level of precedence (lower level is higher precedence), and a short description.

Level	Operator	Description	Associativity
1	`[]`	List access	left-to-right
2	`()`	Macro call	left-to-right
3	`<>`	Entity modifier	left-to-right
4	`+`	Unary plus	right-to-left
	`-`	Unary minus	
5	`!`	Boolean NOT	left-to-right
6	`*`	Multiplication	left-to-right
	`/`	Division	
	`%`	Modulo	
7	`+`	Addition / CSG union	left-to-right
	`-`	Subtraction / CSG difference	
	`^`	CSG intersection	
8	`<=`	Less than or equal	left-to-right
	`>=`	More than or equal	
	`<`	Less than	
	`>`	More than	
	`==`	Equals	
	`!=`	Not equal	
9	`&&`	Boolean AND	left-to-right
10	`||`	Boolean OR	left-to-right
11	`?:`	Ternary operator	right-to-left
12	`=`	Assignment	right-to-left

Table 2.1: List of CRT operators

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...