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. CRT

This paper is about the development of **CRT**, a project done for the requirements of my Bachelor's degree at HEIG-VD, under the guidance of Prof. Pier DONINI.

The goal of this project was to study, design and implement an artistic conception tool for creating realistic three-dimensional images, based on a physical simulation of light and its fundamental properties (diffusion, reflection and refraction), method known as *ray tracing*, along with a custom *scripting language* used to describe scenes, that will be compiled then rendered.

Ray tracing is a technique for computer optical calculations used for rendering an image or for optical studies. This technique involves simulating the inverse path light takes from the camera and compute the interactions of light rays with abstract primitive objects, composing a scene. This technique is used in the entertainment industry by Pixar and Dreamworks for animation, and by 3d artists for concept art.

The project was designed to involve a lot of different computer science and general science fields: mathematics, physics, algorithmics, grammar, compilation, user interfaces and object-oriented design patterns.

1.1 Technologies

CRT was realized in Java 8, with the help of the following libraries:

- ANTLR4
- Apache Commons
- Docking Frames
- jPCT
- LWJGL
- Netbeans OpenIDE
- Substance

1.2 Intermediary report

This version of the paper is an **intermediary report**: it contains every section that has been written so far, as well as this section which will describe the current state of affairs and what remains to be done.

1.2.1 State of affairs

Here are all the features that have been implemented so far:

- The ray tracer can render some primitives and panorama backgrounds. It supports CSG[1] operations (albeit with some bugs), supersampling, depth of field blur, and parallelization.
- The user interface is mostly fully designed but not fully functional. It has a dockable windows system which displays a syntax-highlighted code editor, a property tree, a renderer panel, a toolbar, a console which redirects `stdout` and `stderr`, and some computer statistics graphs. The interface supports two themes: light and dark, and all the icons are inverted at runtime depending on the current theme.
- A live view module converts scene objects to OpenGL displayable elements, with multiple viewports and object picking.
- The language's grammar is almost fully complete.
- The compiler compiles everything the grammar defines so far. It features nestable scopes, recursive variable solving with references, and all the elements needed to describe a scene. It has custom error reporting functionalities and indicates precisely where the errors are.

1.2.2 Remaining features

- The user interface lacks many features: file/project management, updating the property tree and updating the code from it, importing the settings tab from the prototype and giving it functionalities for grabbing settings from the code and writing back to it, the live view perspective, a status bar, a configuration window, an about window.
- The live view has no editing capabilities.
- The grammar lacks rules for loops and conditions, and the CSG blocks are wrong.
- No animation support for now.
- The ray tracer lacks: gamma support, more primitives, affine transformations, procedural texturing, post processing.

1. See section 2.2.6

2. 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.

2.1 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

2.1.1 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*[2] *operation*, which can be either a union, a difference or an intersection.

All of these operations will be explained in further details in section 2.2 about ray tracing.

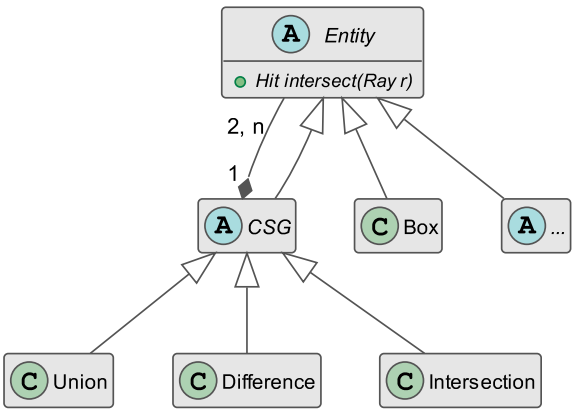


Figure 2.1: The `Entity` class diagram

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

2.1.2 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:

2. 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.

2.1.3 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 2.2).
- A focal distance, defining at which distance objects are sharp.

2.1.4 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 2.2: Real-life *bokeh* — the blurriness of out of focus objects will take the shape of the camera's aperture (pinhole). Here, the *bokeh* is octogonal. Image source: Scott Tucker on Flickr

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

2.1.5 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.

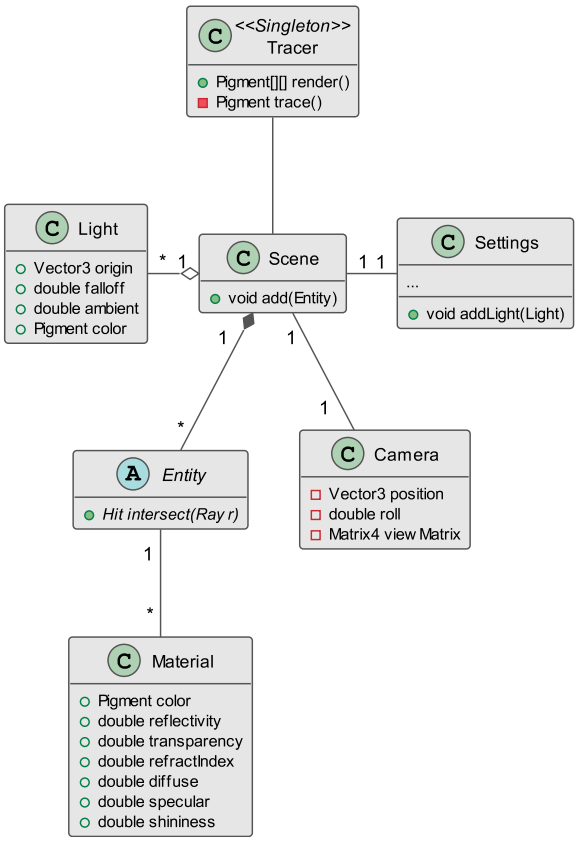


Figure 2.3: Rendering process class diagram

2.2 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).

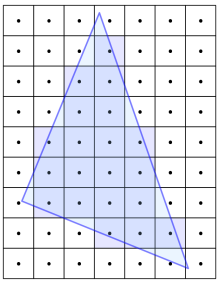


Figure 2.4: Rasterisation of a triangle — once the vertices have been projected on the screen, a discrete pixel is "lit" if its continuous centre is contained within the projected triangle's boundaries. Image source: Wikipedia

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 *from* the screen *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[3].

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:

$$E_{\text{photon}} = hf = \frac{hc}{\lambda} \approx 3 \times 10^{-19}\,\mathrm{J} \tag{2.1}$$

$$\frac{P_{\text{lightbulb}}}{E_{\text{photon}}} = \frac{100\,\mathrm{J\,s^{-1}}}{3 \times 10^{-19}\,\mathrm{J}} \approx 3 \times 10^{20}\,\mathrm{s^{-1}} \tag{2.2}$$

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.

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.

This solve the difficulties of rasterisation previously mentioned in that the very nature of tracing rays makes it possible to apply the exact same formulas used in physics: the law of reflection, the Snell-Descartes law of refraction, Beer-Lambert law, the inverse square law, and so on. Also, shadows don't have to be drawn, they just exist — light just "naturally" never reaches shadowed spots in a scene, so no light comes from it.

The basic idea of backward tracing, explained in details in the next section, is demonstrated visually in figure 2.5

3. 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* so all incoming photons are dispersed)

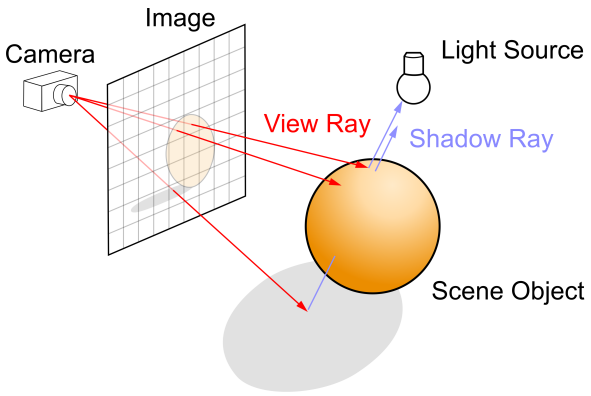


Figure 2.5: Backtracing light rays. Image source: Wikipedia

2.2.1 Backward tracing implementation

For every pixel on the screen, a **primary ray** is generated, then traced. Because each pixel's tracing is **independent** from one another, the process can be parallelized. This is easily done thanks to the new Java 8 API and its *streams*: after splitting the screen into blocks in a Java list `coords`, all we have to do is call:

```
1 coords.parallelStream().forEach(
2     (int[] c) -> processPixel(c, image, scene));
```

Code listing 2.1: Java 8's easy parallelization

The process of tracing a ray takes the following steps:

1. Start with a black colour
2. Search for the entity closest to the ray's origin that intersects with the ray. If nothing is hit, return the background colour
3. For every light in the scene, send a **shadow ray** originated on the intersection point towards the light
   - Add the *ambient* factor of the light
   - If it hits nothing before reaching the light, add a mix of the light's colour and the object's to the current colour. The light's colour contribution is attenuated by two factors:
     - The less parallel the surface normal is with the incoming ray, the darker
     - The further the light had to travel, the darker (inverse square law)
   - If the shadow ray hits an object, it is in its shadow: no colour is added
4. If the surface's material has a reflective component, recursively trace a **reflection ray** in an angle symmetrical to the angle of incidence, and add the resulting colour
5. If the surface's material has a refractive component, recursively trace a **refraction ray** in an angle obtained with the Snell-Descartes law, and add the resulting colour

2.2.2 Coordinate system

Like *POV-Ray*, *OpenGL*, *DirectX*, *Unity* and many others, this project opted for a **left-handed coordinate system** where the z axis points inside the screen and not outwards.

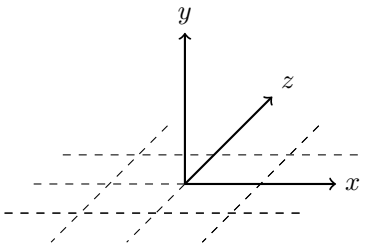


Figure 2.6: Left-handed coordinate system

For graphical composition, having an inverted z axis is (for some people) more intuitive: if the origin is located at the bottom-left of the screen, x goes *right*, y goes *up*, and z goes *inside* the screen.

This system choice incurs a small consideration when doing linear algebra, but can be converted any time between both systems. The only real thing to change is the way the *cross product* behaves and invert some results.

2.2.3 Ray generation

Before tracing the path of a ray, we need to *generate* it. A **ray** has an origin ($\vec{o}$), a length (t) and a direction ($\vec{d}$). Its equation is thus:

$$\vec{r} = \vec{o} + t\vec{d} \tag{2.3}$$

The initial rays we begin with when ray-tracing are called **primary rays**. For a standard *pinhole* projection, they originate (before camera transformation) at $\vec{o} = \vec{0}$ and each ray points towards the centre of a pixel situated on a *virtual screen* the same resolution as the desired output, situated 1 unit away on the z axis.

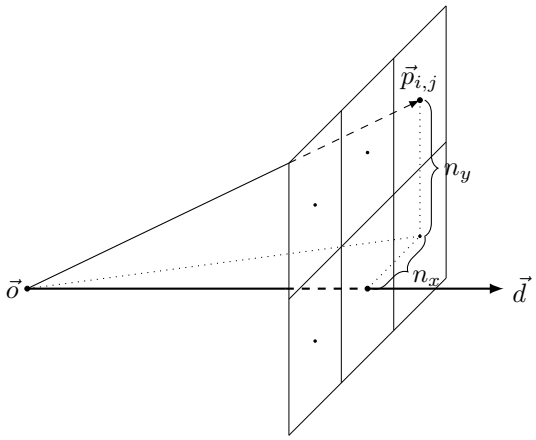


Figure 2.7: Primary ray

The first step is to normalize the coordinates of the targeted pixel to be between -1 and 1. We add 0.5 in both directions so that the ray "aims" towards its centre, if it was 1 unit long and 1 unit high.

```
double nX = (2 * ((x + 0.5) / settings.width) - 1);
double nY = (1 - 2 * ((y + 0.5) / settings.height));
```

Code listing 2.2: Pixel coordinates normalization

The next is to take in account the camera's *field of view* (see section 2.1.3): the wider the angle, the more we will see left and right, up and down. To reflect this, we have to multiply both normalized pixel coordinates by a **FOV factor**.

Because the FOV value corresponds to the vertical FOV, we also need to multiply the x coordinates by the screen's ratio.

The FOV factor is easily calculated: if α is the FOV angle and because $\|\vec{d}\| = 1$, simple trigonometry tells us that the factor we need to multiply our normalized coordinates by is $\tan(\frac{\alpha}{2})$:

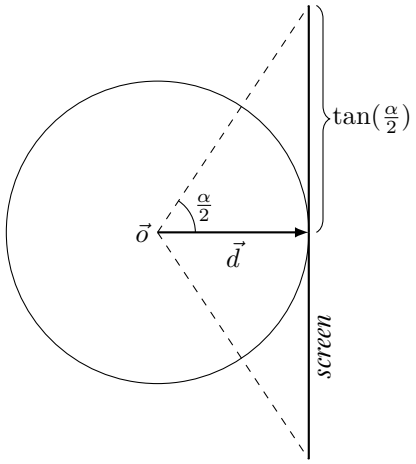


Figure 2.8: Finding the FOV factor

These two operations leave us with primary rays originating from $\vec{0}$, looking at a screen centred at $(0, 0, 1)$. To transform these rays and put them where the camera actually is, we could translate and rotate them using a *transformation matrix*, but it is simpler to just add the camera's *up* and *right* component multiplied by the normalized and FOV-adjusted coordinates we computed earlier.

If we sum up, these are all the steps needed for primary ray generation (code from the `Tracer` class):

```
double nX = (2 * ((x + 0.5) / settings.width) - 1);
double nY = (1 - 2 * ((y + 0.5) / settings.height));

double camX = nX * settings.fovFactor * settings.ratio;
double camY = nY * settings.fovFactor;

Vector3 rightComp = camera.getRight().multiply(camX);
Vector3 upComp = camera.getUp().multiply(camY);
direction =
     direction.add(rightComp).add(upComp).normalize();

Ray primary = new Ray(direction, camera.getPosition());
```

Code listing 2.3: Primary ray generation

2.2.4 Primitives

Next in the pipeline after generating a primary ray is to check whether or not it *intersects* with any of the *primitives* present in the scene, and take the closest one.

Checking if a ray intersects with a primitive amounts to put both their equations in one and *solve it*. In this section we will see how to compute a ray-sphere intersection and find the intersection points if they exist[4].

In vector notation, the equation of a sphere is

$$\|\vec{r} - \vec{c}\|^2 = R^2 \tag{2.4}$$

where $\vec{r}$ is a point on the sphere, $\vec{c}$ the centre of the sphere and R its radius. By substituting $\vec{r}$ with the ray equation (2.3), we get

$$\|\vec{o} + t\vec{d} - \vec{c}\|^2 = R^2 \tag{2.5}$$

which, when expanded and rearranged, gives

$$t^2(\vec{d}\cdot\vec{d}) + 2t(\vec{d}\cdot(\vec{o}-\vec{c})) + (\vec{o}-\vec{c})\cdot(\vec{o}-\vec{c}) - R^2 = 0 \tag{2.6}$$

We can now solve this quadratic equation of the form $at^2 + bt + c = 0$ for t, where

$$a = \vec{d}\cdot\vec{d} \tag{2.7}$$

$$b = 2(\vec{d}\cdot(\vec{o}-\vec{c})) \tag{2.8}$$

$$c = (\vec{o}-\vec{c})\cdot(\vec{o}-\vec{c}) - R^2 \tag{2.9}$$

Using the method of the discriminant, we get:

$$t = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a} \tag{2.10}$$

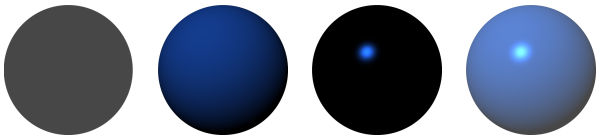
Next, depending on whether or not there is zero, one or two solutions, we can find the intersection points by injecting t back into the ray equation, check whether or not the ray was shot from inside the sphere, compute the normals, etc. These informations will be useful later when we will be doing *CSG operations*.

2.2.5 Light calculations

As quickly described in section 2.2.1, colours are computed in an *additive* fashion, depending on several factors such as light source *angle*, *intensity* and material properties.

There are several colour components derived from the lights present in a scene: ambient, diffuse and specular. This method of computing light components is called the **Blinn–Phong shading model**.

4. In the code, more than that is done because other parts of the ray-tracer need intersection normals



(a) Ambient (b) Diffuse (c) Specular (d) Combined

Figure 2.9: Phong shading model — light components are computed in steps.

First off, let's define the **inverse square law**, which we will use for computing the following components. In physics, the amount of light received from a light source at a given distance is *inversely proportional* to the square of the distance:

$$I \propto \frac{1}{t^2} \tag{2.11}$$

In the following diagram, we can see that effect represented in three dimensions: every square is the same area, but from a greater distance, the same amount of rays hit more squares and thus, each square gets less light.

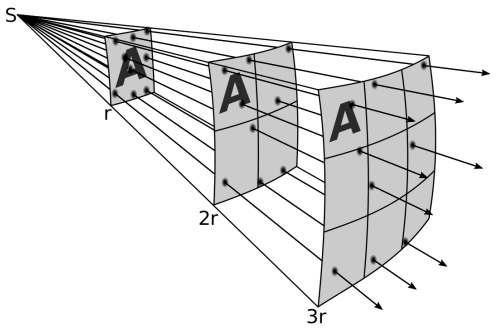


Figure 2.10: Inverse square law Image source: Wikipedia

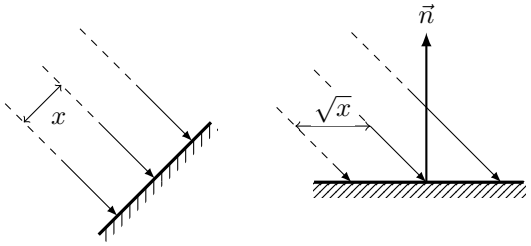
In the code, the recursive tracing functions keeps track of all the distance that the starting primary ray travelled, and computes the ISL factor by dividing the light's *falloff* factor by the square of the total distance squared.

The first component of the Blinn–Phong model is trivial: **ambient** light is just a fraction of the light's colour that is added to any point on the scene, whether or not it is in shadow:

$$\vec{c}_\mathrm{a} = l_\mathrm{a}\vec{l}_\mathrm{c} \tag{2.12}$$

where l_a is the light's ambient factor and l_c its colour. Note that this is a quick and dirty *trick* to simulate global illumination which would otherwise be costly, inherited from more standard rendering techniques like in OpenGL for example.

Diffuse light is the amount of light that is scattered by the material when hit by a light source. It simulates the fact that for a point on a material's surface, the more a light source's *direction* is aligned with its *normal*, the more the point gets illuminated by the light source. Let's observe that in the following diagram:



(a) Rays parallel with normal (b) Rays at 45° with normal

Figure 2.11: Surface angle with incoming rays

As we can see, if our rays are separated by a distance x and hit a surface whose normal is parallel with them, their distance on the surface is still x, whereas if the surface normal is at a 45° angle with the rays, they are separated by a distance of $\sqrt{x}$ — reducing the density of photons per area, and thus the surface is *darker*.

An *attenuation* factor is first computed by taking the *dot product* of the surface normal and the light's direction vector, giving the cosine of the angle. We then proceed to multiply this factor with the ISL and the light's colour, then multiply this result with the surface's material colour:

$$\vec{c}_\mathrm{d} = \left[\vec{l}_\mathrm{c} I(\vec{l}_\mathrm{d} \cdot \vec{n})\right] \cdot (\vec{m}_\mathrm{c} m_\mathrm{d}) \tag{2.13}$$

Lastly, we have the **specular** component which emulates shininess. This is also not physically accurate; it is a *trick* to give light sources more "width" instead of just being single infinitesimally small points, so that their reflections can be seen on diffuse shiny surfaces.

The specular factor of any given point on a surface depends on how much the light source's reflected ray ($\vec{l'}$) on that point is aligned with the viewing direction ($\vec{r}$), i.e. the angle between the reflected light ray and the incoming primary ray:

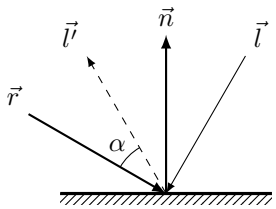


Figure 2.12: Angle between reflected light ray and primary ray

The reflected ray is computed as follows:

$$\vec{l'} = \vec{l} - 2(\vec{l} \cdot \vec{n})\vec{n} \tag{2.14}$$

The specular factor is then computed by taking the cosine of the angle (*aka* dot product) between the reflected ray and the primary ray to the power m_sh, the material's shininess factor. The formula for the specular component is then:

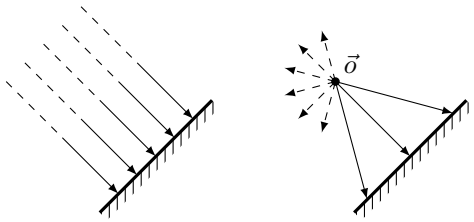
$$\vec{c}_\mathrm{s} = (\vec{l'} \cdot \vec{r})^{m_\mathrm{sh}} m_\mathrm{s} I(\vec{l}_\mathrm{c} \cdot \vec{m}_\mathrm{c}) \tag{2.15}$$

Summing up, the total colour of a given point using the Blinn–Phong model is

$$\vec{c} = \vec{c}_a + \vec{c}_d + \vec{c}_s \tag{2.16}$$

To this, we can add reflectivity and refraction, by recursively tracing the reflected / refracted rays.

In this project, two types of light sources were implemented: *parallel* lights and *point* lights. As explained on the next figure, parallel lights like the sun represent a light source that is infinitely far away rendering its rays virtually parallel. This implies that the ISL is no longer in effect[5] whereas with the point light, we can see on the figure that just by going a bit further down the surface the light vectors are longer and more spread out.



(a) Parallel light (b) Point light

Figure 2.13: Light source types

In the code, this is implemented by making parallel light sources always return the same direction vector, whereas point light sources compute each direction vector by subtracting the surface point to their origin.

2.2.6 Constructive solid geometry

CSG operations are similar to *set operations*, but apply to primitive solids or results of other CSG operations. They are of three kinds: union, intersection and difference. In the code, CSG operations inherit from `Entity` and must provide an `intersect(Ray r)` method.

The first CSG operation is the **union**. It is very trivial: the intersection of a ray and and a union of objects is just the intersection point of all the objects that is *closest* to the ray's origin.

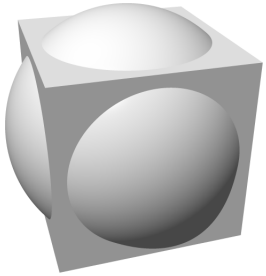


Figure 2.14: CSG union between a cube and a sphere

Next is the **intersection** ...

5. By looking at figure 2.10, we can see that if all rays were indeed parallel, all the squares would receive the same amount of rays

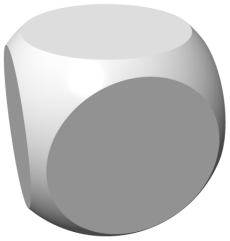


Figure 2.15: CSG intersection between a cube and a sphere

Finally, the **difference** ...

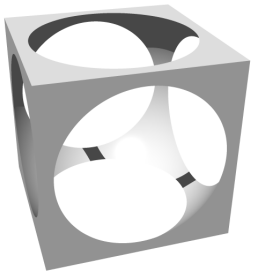


Figure 2.16: CSG difference between a cube and a sphere

Using only those three basic processes, one can combine simple primitives and iteratively produce complex shapes, like following upright piano foot produced in POV-Ray, resulting from multiple CSG operations: cylinders are subtracted twice from boxes and tori quarters are used to fill the gaps between the two resulting boxes.

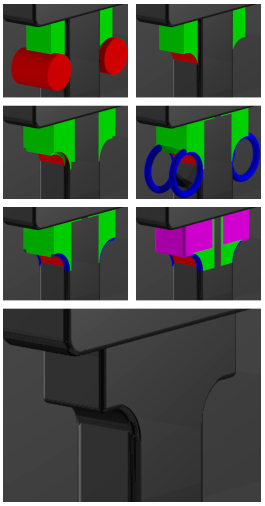


Figure 2.17: A piano foot obtained from CSG operations

2.2.7 Background projections

Rendering purely mathematical primitive volumes is nice and fun but a little dull without an interesting background behind them. To add more depth to the scenes, backgrounds were implemented in the `Background` object. It provides three modes:

- Solid colour
- Horizon and zenith gradient
- Spherical panoramic projection

The interesting one is the latter, which takes a 360° panorama and a direction vector, and returns the colour at which the vector would point if the panorama was wrapped around the scene on an infinitely big sphere.

The way the pixel coordinates are computed on the panorama picture is as follows: using `atan2()`, the angle of the vector projected on the xz plane (*yaw*) is mapped to the x coordinate on the picture, and the angle of the vector projected on the yz plane (*pitch*) is mapped to the y coordinate. The final colour is then bilinearly interpolated and returned to the ray tracer.

TODO picture

2.2.8 Depth of field

To simulate a real camera, the implemented ray tracing model supports depth of field simulation. It provides an additional layer of realism and dynamism to the rendered pictures by making all out of focus objects blurred in a physically realistic way.

In order to achieve this effect, the ray tracing model is modified from the rays all originating from one point to another model where rays can originate from any point on a disc (mimicking the fact that a camera has a disc-shaped aperture hole and not a single point hole).

TODO diagram

As can be seen on the diagram, the primary rays can be generated anywhere on the origin disc, and aim at the desired object as seen from the focal plane. If the object lies on that focal plane, all rays originating from the disc will land on the same spot on the object.

However, if the object is further away from the focal plane, rays originating from different location on the disc will meet on the focal plane and then start to diverge until they reach their destination, which will not always be the object aimed at.

By averaging the colour resulting from these rays, the final colour will either be the object's precise colour if it lies on the focal plane, or a mixture of the object's colour and the background around it. This effectively creates a depth of field blur.

TODO example of DOF

Three factors have an effect on how the depth of field blur will look like: aperture size, focal distance and aperture shape.

Aperture size determines the thickness of the area that will be in focus, and will effectively represent the radius of the aperture disc (or other regular polygon). The smaller this disc is, the closer it will be to a point-source, and thus resembling more the normal model with no depth of field. The bigger it is, the shallower the area where objects are in focus will be.

The focal distance's effect is very straight forward: it controls the distance at which the focal plane will reside, in which objects are in focus and not blurred.

Finally, the aperture shape will determine the shape of the *bokeh* (see section 2.1.3). We need to model shapes, which will provide a method to obtain a uniformly distributed random point from inside their boundaries. A lazy but efficient approach is to take a random point from a square, which only consists of having two random float numbers ranging from 0 to 1 as x and y coordinates.

A perfect model uses a disc, which is the shape cameras try to obtain. Efficient and correct algorithms to generate a random point in a circle already exists and were implemented in the project.

In reality, because real life cameras need their aperture size to change for different exposures, their aperture mechanism is constructed of a number of blades that can change the size of the hole they create:

Figure 2.18: Aperture blades, also called iris, here with a pentagonal shape Image source: Nayu Kim on Flickr

In practice, this means that the resulting shape is never a perfect circle, but is approximated with a regular polygon such as a pentagon for cheap lenses or octagons for luxury lenses. Professor Jean-François HÊCHE devised an efficient algorithm for generating random coordinates inside a regular polygon, by dividing the work in sub steps. First, generate a point inside a square and map it to a triangle. Because all regular polygons can be split into triangles, the last step is to randomly choose which triangle contains the point and rotate the coordinates to that position. This gives us a smooth, shaped *bokeh*:

The depth of field model isn't very efficient performance-wise: because the source of the rays is not a point any more, we need to trace more than one ray per pixel on the screen. The rays origins are to be chosen randomly on the aperture shape, and their number is chosen by the user at render time. If this number is too low, the resulting image will look very noisy, as seen in the following diagram. A number too big will

(a) Generating 1000 points in a pentagon takes less then 10 ms

(b) Pentagonal *bokeh* in a ray-traced scene

Figure 2.19: In-engine *bokeh*

multiply the render time linearly, and by adding supersampling (see section 2.2.10), the render time will become very long. Empirically, a good number of samples for depth of field has been found to be between 32 and 64.

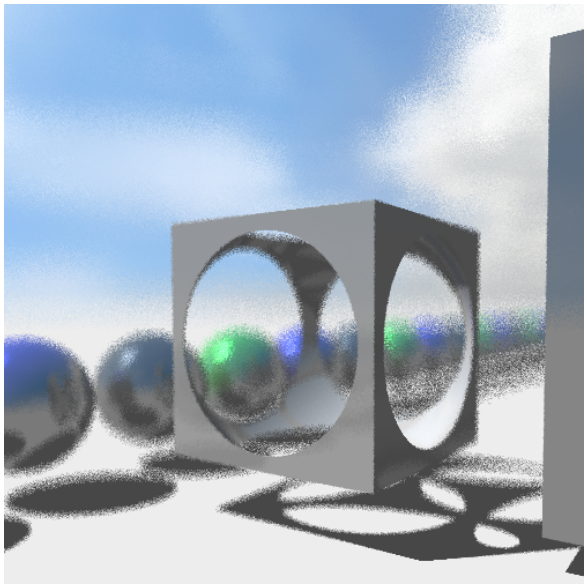


Figure 2.20: Noisy DOF

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

2.2.9 Materials

The following ideas were not implemented in the final project but were studied in detail.

INTRO TEXT

TODO perlin noise picture

- 3D texturing function
- Procedural texturing

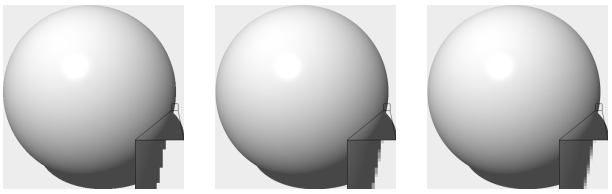
We can also use perlin noise to do bump mapping...

Map pictures to spheres or boxes UV mapping

2.2.10 Supersampling

In order to produce quality pictures, **supersampling** was implemented. It is a spatial anti-aliasing[6] method which works by tracing multiple rays per pixel instead of just one, then *averaging* the resulting colours, providing a much more accurate final colour for a given pixel.

There are multiple ways to select coordinates inside a pixel: one could choose n random points inside each pixels, or choose a more advanced sampling pattern[7], but for the purpose of this project, a simple *grid* pattern was used.



(a) No supersampling (b) 2×2 SSAA (c) 4×4 SSAA

Figure 2.21: Different supersampling values

In figure 2.21, we can see that the bigger grid we use, the more refined the final picture is. This, however, comes at a big cost: with a 4×4 grid, a render will likely take at least 16 times as much time to render.

6. Aliasing is seen in most edges, which appear jagged and pixelated.
7. See http://en.wikipedia.org/wiki/Supersampling#Supersampling_patterns

3. Language

So far, we can compose and render scenes directly by writing them in Java by 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:

```
--Entities---------------------------------------------------
sphere1 = Sphere {
    center -> vec3(0, 0.5, 0)
    radius -> 0.5
}

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

--Macros-----------------------------------------------------
myMacro = Macro (arg1) {
    i = 0
    -- Draw sphere1 "max" times on the x axis
    while (i < max) {
        sphere1 <translate vec3(i*1.0, 0.0, 0.0)>
        i = i - 1
    }
}
```

Code listing 3.1: Sample CRT script

3.1 ANTLR

The language's grammar will be designed in a EBNF variant, the G4 syntax from **ANTLR**[8], 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.

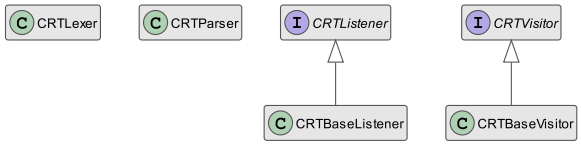


Figure 3.1: Family of classes generated by ANTLR4

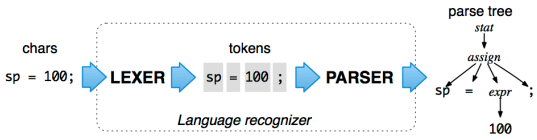


Figure 3.2: Language recognition process. Image source: ANTLR

For our compiler, we will use the *visitor* pattern, which allows for more control over the tree traversal; the listener provided by ANTLR 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):

```
String code = "..."

CRTLexer lexer = new CRTLexer(new ANTLRInputStream(code));
CommonTokenStream tokens = new CommonTokenStream(lexer);
CRTParser parser = new CRTParser(tokens);
ParseTree tree = parser.script();

Compiler compiler = new Compiler(code);
Script script = compiler.visit(tree);
```

Code listing 3.2: Generating a parse tree and compiling

3.2 Grammar

The designed grammar is non-ambiguous (context-free), but uses **left-recursion**[9] for ease of writing *and* reading, which ANTLR supports since version 4.2.

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

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

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

mar[10], from which much was learned about left-recursion and operator precedence.

Furthermore, the "ANTLR 4 IDE" *Eclipse* plug-in[11] proved to be very useful during the development of the grammar. It provides useful tools for debugging such as syntax diagrams and a live parse tree visualisation — just by selecting a grammar rule and typing in code, 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.).

3.2.1 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 2.1.1).

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.

10. http://github.com/antlr/grammars-v4/blob/master/java/Java.g4
11. 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.

3.2.2 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 3.1 shows all operators, their level of precedence (lower level is higher precedence), and a short description.

3.3 Compiling process

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 3.1: List of CRT operators

4. User interface

> "*A user interface is like a joke. If you have to explain it, it's not that good.*"
>
> — Martin LEBLANC, *Iconfinder*

The goal of this project from a GUI point of view was to mimick the professional look, feel and modularity of a standard IDE such as Netbeans or Eclipse. The interface has to be immediately understandable and appealing to the user.

The main window is composed of the following components:

- Code editor
- Renderer
- Property viewer
- Console
- Settings tab

The following sub-sections will present some of the highlights about the user interface and the libraries used to produce them.

4.1 Netbeans API

The property tree and property editor were implemented using the Netbeans Platform API. This API provides many useful components used in the Netbeans IDE and with a little bit of tinkering, allowed for

- Introspection, beans
- Property sheet, editor, hacks

4.2 Live view

To provide a more direct feedback to the user and give him a feeling of where his entities are placed without going through the wait of a full render, a live view module was implemented.

It consists mainly of two components: a `Scene` translator and an 60 frames per second OpenGL view.

The OpenGL view was implemented using jPCT, a library acting as a front-end for LWJGL, the most commonly used OpenGL library for Java. Given the scope of this project, a front-end was used to reduce development time: jPCT provides a complete framework with wrapper methods that do all the low-level OpenGL calls internally, enabling us to use simple methods like `Object3D.getCube(scale)` for creating a cube, instead of the following LWJGL code:

```
GL11.glBegin(GL11.GL_TRIANGLES);
GL11.glVertex3f( 0.0f, 1.0f, 0.0f);
GL11.glVertex3f(-1.0f,-1.0f, 1.0f);
GL11.glVertex3f( 1.0f,-1.0f, 1.0f);
// Repeated 5 times
GL11.glEnd();
```

Code listing 4.1: Creating a cube with LWJGL

And that is without colour or texture management, which jPCT provides easy methods for. Another useful aspect is the easy integration in Swing windows, albeit with some workaround having to be used when adding multiple viewports.

The conversion is pretty straight forward: we recursively go through a `Scene` object and whenever a supported entity or light is encountered, the jPCT equivalent creator method is invoked, and the created object is translated to its correct position.

During that conversion process, some sacrifices have to be made. The CSG objects cannot be easily translated to the polygonal world and computing the positions of the vertices of a CSG operation would be a project of its own right. Also, limitations of jPCT only allow for one light source for use with shadow mapping, which will make the preview even more different than the resulting ray-traced render.

Finishing and testing the live view module was a very exciting moment, because comparing the same `Scene` rendered once with our ray tracing engine and then with OpenGL would validate that all our algorithms are correct. And, indeed, all lines match, the perspective is correct.

TODO: comparison shot

After tweaking the module for a moment, the lack of a background in the previewed scenes was quickly made apparent. Because there are no rays in rasterization, we cannot just assign a colour to rays hitting infinity like we did with ray tracing (see section 2.2.7).

So instead, we invented a clever hack that uses jPCT's implementation of sky boxes[12]: during the conversion process of a `Scene`, a secondary empty `Scene` is created that shares the same background. In its settings, the field of view is set to 90°, which is the angle needed to see only one face of the cube when the camera is placed in its centre.

We then proceed to render 6 separate images of the panorama background and use them for the jPCT sky box,

12. In video games, sky boxes are used for backgrounds and consist of cube much larger in magnitude than the rendered world and 6 textures are mapped on its faces.

which leads to an almost similar result to the regular spherical projection mapping used earlier:

TODO: skybox screenshot

- jPCT, LWJGL
- Conversion
- Sacrifices - only one shadow source, no CSG
- Hacks
- Problems with multiple viewports - no shadows

4.3 Docking Frames

- General
- Difficulties

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`

B.3 Development journal

09.03.2015 - 15.03.2015

- GitHub repository[13] created
  - `.gitignore` file tailored for project
- Specifications document
- Maven project created via NetBeans
  - Some dependencies
  - Default license header set (GPL3)
- Logo design
  - First simple attempt
  - Created CRT effect as a Photoshop action and patterns
  - Multiple sizes

16.03.2015 - 22.03.2015

- LaTeXthesis template
- Python script for building thesis
- UML class diagrams for rendering process, ANTLR4 structure
- Some sequence diagrams
- Paragraphs on the rendering process and grammar parsing
- Reworked the grammar

23.03.2015 - 30.03.2015

- Established basic ray tracing skeleton
- Refactored code from old prototype
- Changed structure according to new decisions made with supervisor
- Added Box primitive

01.04.2015 - 07.04.2015

- Work on CSG operations:
  - Union OK
  - Intersection OK
  - Difference has buggy normals but works
- Change in `Hit` class model to indicate entry and exit points of `Ray`
- Corrections to Sphere primitive intersection and normals
- Rendering classes refactoring
- Rendering UML diagram split in two, added cardinalities and fixed a few aspects

08.04.2015 - 15.04.2015

- Wrote the grammar from scratch
- Wrote test cases
- Started work on compiler:
  - General framework
  - Scope and variables
  - Scoped variable resolution
  - Variable reference pointers
  - Literal variable assignments
  - List variable assignments
  - List access
  - Function calls for color and vector types

13. https://github.com/Tenchi2xh/CRT

16.04.2015 - 22.04.2015

- Corrections
- Typography improvements
- Section about ANTLR, the grammar, BNF listing and operators table
- Section about ray tracing history, workings, process
- Citing image sources
- Rewrote how the camera system works
  - Removed awkward 4×4 matrix system (more appropriate for rasterisation)
  - System with up and right vectors
  - Left-handed coordinate system
  - Corrected camera projections
- Rewrote how lights the work
  - Multiple light types
  - Each light type has to give its own direction vector and distance. This lead to correct a bug where a light source is inside a primitive, because the ray intersection point is exactly on the surface, it tries to go through anyway instead of stopping
- Much work on CSG
- Animation test
- Drew diagrams with TikZ for light explanations
- Experimented with global illumination
- Adapted depth-of-field calculations to new camera system

23.04.2015 - 29.04.2015

- Set up base environment for jPCT + LWJGL in Maven
  - Local repository
  - JVM arguments
- Configured Maven's POM for producing distributable content
- Corrected run classpath
- Some documentation
- Idea for poster
- Set up base test for jPCT loop
- Basic Scene to jPCT converter
- Ported lighting system with basic shadow mapping for first parallel light
- Many scenes from close up would clip and setting a closer near plane would break shadows. So a hack where all distances and values like light falloff are multiplied by a constant when imported and everything is bigger, making the default nearPlane appear small in comparison
- Project background on a cube via ray tracing to export to jPCT skybox
- Corrected ray tracing camera for looking straight up or down
- Live view camera control, + mouse
- Implemented object picking in live view
- Optimized rendering speed
- Shader prototype

30.04.2015 - 06.05.2015

- Studied Docking Frames API
- Java Beans infos and editors
- Code editor with syntax highlighting and some features
- Entity trees
- Entity properties
- Docking system
- Language touch-ups
- Console panel, catching `stdout` and `stderr`, autoscrolling
- Renderer panel
- Dynamic icon system dependent on theme
- Theme singleton
- Multiple resizeable viewports
- System informations

07.05.2015 - 13.05.2015

- Work on the paper

14.05.2015 - 20.05.2015

- Work on the paper

21.05.2015 - 27.05.2015

- Work on the paper
- Bibliography collecting
- Bibliography integration

28.05.2015 - 03.06.2015

- Work on the paper
- Toolbar and menubar
- Icons

04.06.2015 - 10.06.2015

- Work on the paper
- Almost finished compilation — can now fully compile and render scenes without macros and loops

11.06.2015 - 17.06.2015

- Intermediary report

C. Bibliography

[1] *ANTLR4 wiki*.
URL: https://theantlrguy.atlassian.net/wiki/display/ANTLR4/.

[2] *Better Sampling*. CodeItNow. Jan. 7, 2009.
URL: http://www.rorydriscoll.com/2009/01/07/better-sampling/.

[3] Emmanuel Bourg. *Is there a Java function which parses escaped characters?* Oct. 21, 2011.
URL: http://stackoverflow.com/a/7847310.

[4] btilly. *Generate a random point within a circle (uniformly)*. Apr. 30, 2011.
URL: http://stackoverflow.com/a/5838991.

[5] Rob Camick. *Text Component Line Number*. Java Tips Weblog. May 23, 2009.
URL: https://tips4java.wordpress.com/2009/05/23/text-component-line-number/.

[6] *Coding Bilinear Interpolation*. The Supercomputing Blog.
URL: http://supercomputingblog.com/graphics/coding-bilinear-interpolation/.

[7] Benjamin M. Crist. *How to rotate a vector by a given direction*. Jan. 4, 2014.
URL: http://stackoverflow.com/a/20925348.

[8] *Docking Frames documentation*.
URL: http://dock.javaforge.com/dockingFrames_v1.1.1/doc/index.html.

[9] *Docking Frames forums*.
URL: http://forum.byte-welt.net/byte-welt-projekte-projects/dockingframes/.

[10] Elrac. *Determining if a point is inside a regular polygon*. everything2. Mar. 15, 2001.
URL: http://everything2.com/title/Determining+if+a+point+is+inside+a+regular+polygon.

[11] Igal. *How to get percentage of CPU usage of OS from java*. Feb. 22, 2014.
URL: http://stackoverflow.com/a/21962037.

[12] johnny. *How to invert an RGB color in integer form?* Aug. 21, 2014.
URL: http://stackoverflow.com/a/25425107.

[13] *jPCT documentation*.
URL: http://www.jpct.net/doc/index.html.

[14] *Line–sphere intersection*. In: *Wikipedia, the free encyclopedia*. Page Version ID: 645471517. Feb. 3, 2015.
URL: http://en.wikipedia.org/wiki/Line%C3%A2%C2%80%C2%93sphere_intersection.

[15] *NetBeans Platform Learning Trail*.
URL: https://netbeans.org/kb/trails/platform.html.

[16] Simon Nickerson. *What's the most efficient way to test two integer ranges for overlap?* Oct. 15, 2012.
URL: http://stackoverflow.com/a/3269471.

[17] *The jPCT wiki*.
URL: http://www.jpct.net/wiki/.

[18] Mikhail Vladimirov. *Analyze system memory/cpu stats with Java*. Mar. 14, 2013.
URL: http://stackoverflow.com/a/15402012.

[19] Amy Williams et al. *An Efficient and Robust Ray–Box Intersection Algorithm*. June 9, 2005.
URL: http://people.csail.mit.edu/amy/papers/box-jgt.pdf.