

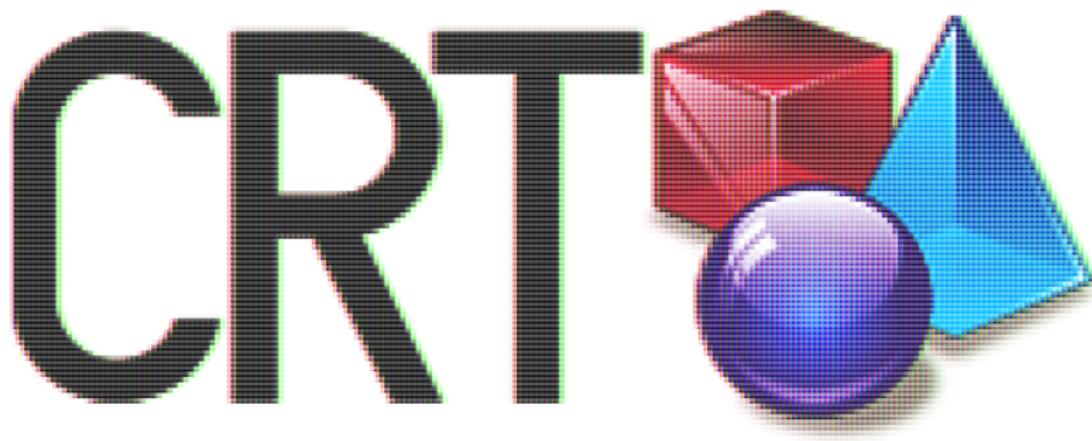
CRT – Thesis Defense

Hamza Haiken

8 September 2015

Introduction

CRT



- Stands for CATHODE RAY TRACER
- Available on GitHub: <https://github.com/Tenchi2xh/CRT>

1 Introduction

2 Ray Tracing

3 CRT Language

4 Interface

5 Demonstration

6 Conclusion

7 Q&A

Goal

- Artistic conception tool
- Realistic 3D images via ray tracing
- Physical light simulation
- Custom language & compiler
- Flexible user interface
- OpenGL live view

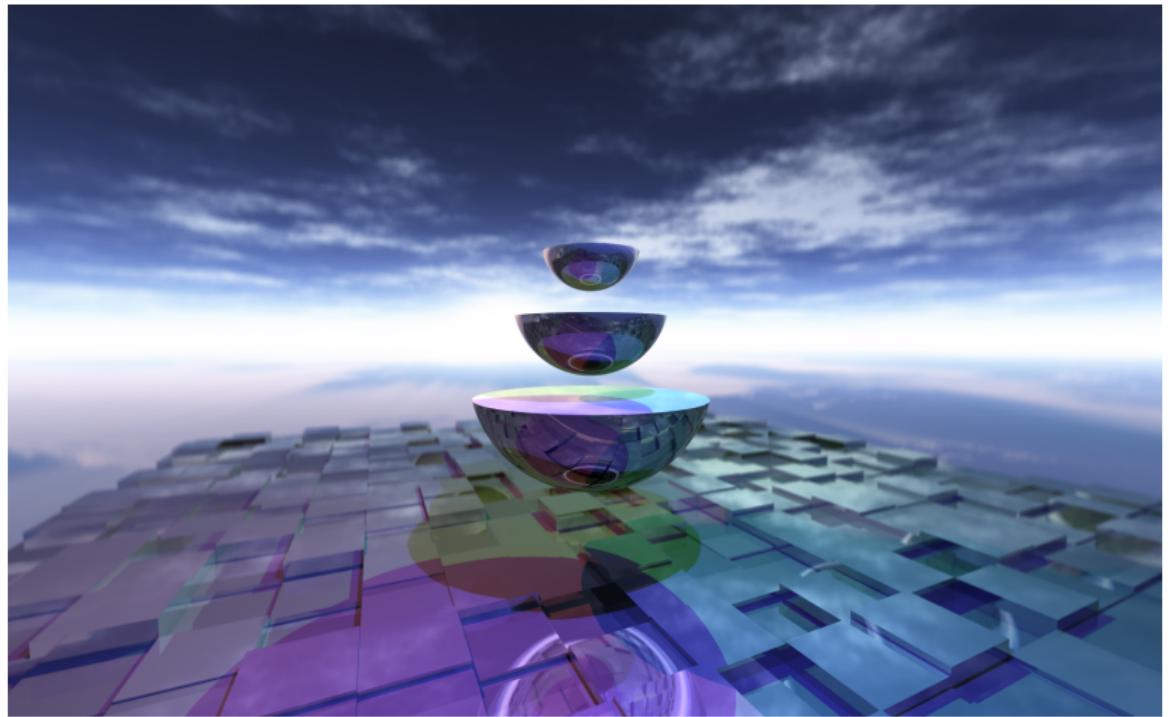


Figure 1: Example of a rendered 3D image



Figure 2: Example of a rendered 3D image

Technologies

- Java 8
- Maths & Physics
- ANTLR4
- OpenGL (jPCT, LWJGL)
- Docking Frames

Ray Tracing

Classical method

- Live 3D uses rasterisation
- World is approximated using triangles
- Project triangles on screen
- Fill the gaps with color
- *Hacks* to produce real-life effects
- Fast, accelerated with hardware

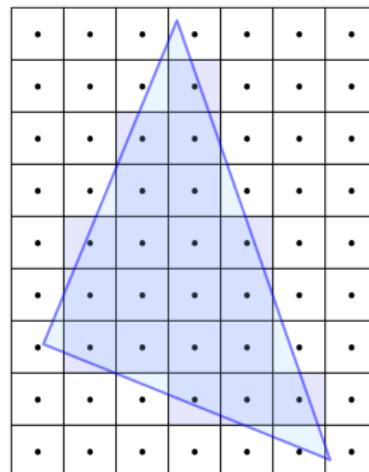


Figure 3: Rasterization

Principle

- Simulate light's path and interactions with environment
- Objects absorb color from rays hitting them, makes colors
- → Slow, but accurate and realistic simulation of light

- Tracing forward like in nature is too costly
- Backward-tracing requires only one ray per pixel
- Rays are traced from the camera and look for light sources
- Rays can bounce off surfaces
- Effects are naturally derived from physics

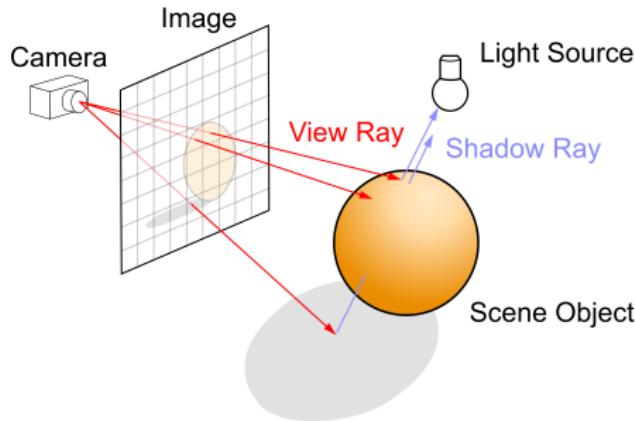


Figure 4: Backward-tracing

Modelization

- Top-level `Scene` object
- Contains `Entities` and `Lights`
- CSG is recursive

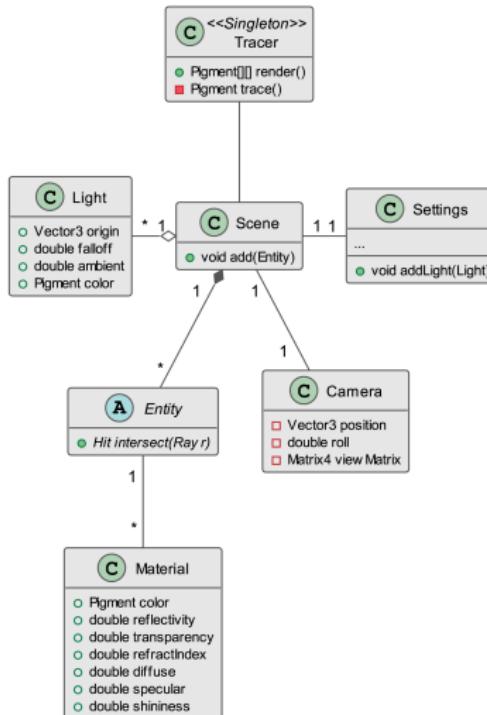


Figure 5: Rendering model

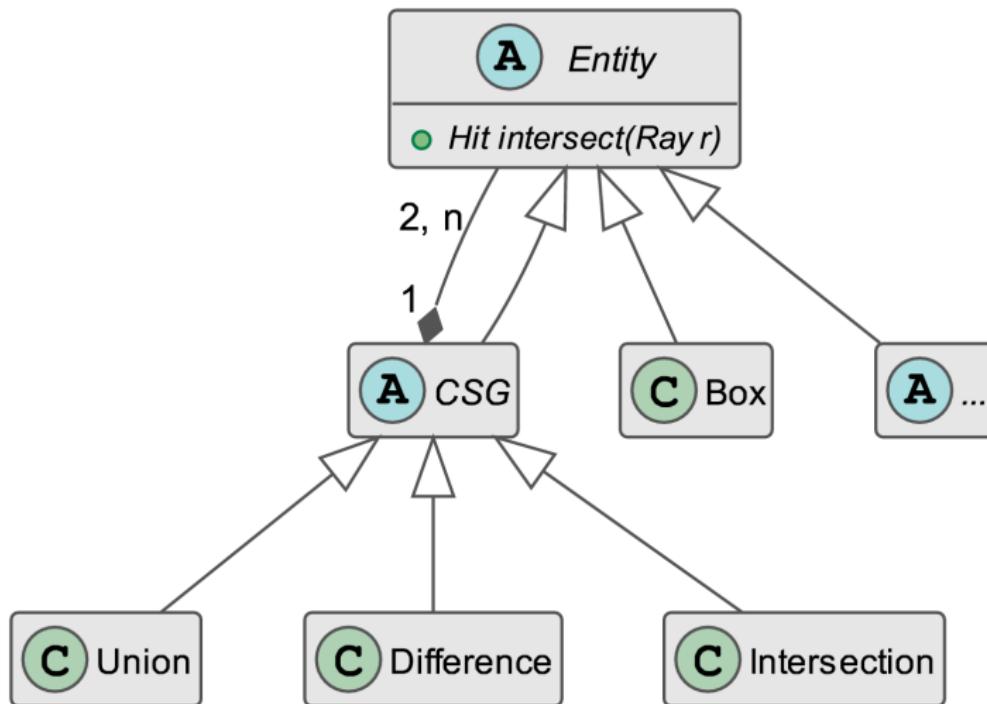


Figure 6: Entity model

Generating rays

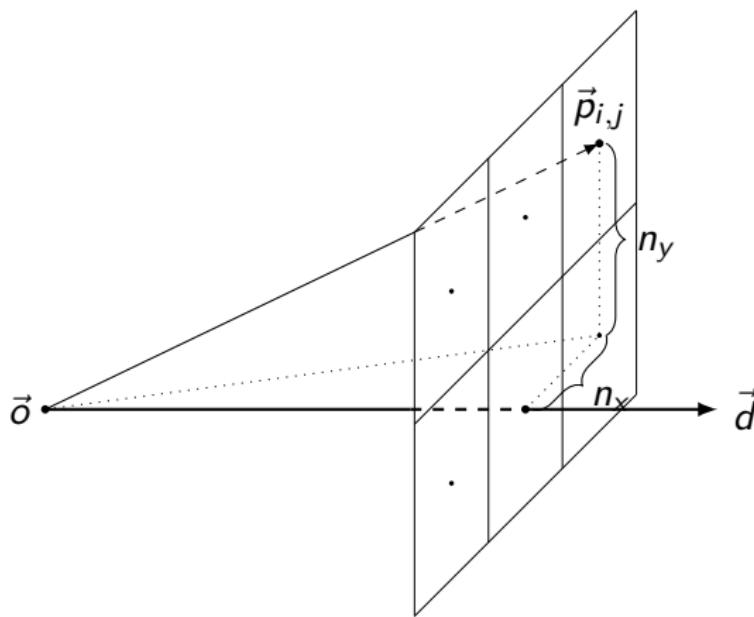


Figure 7: Primary ray generation

Shading

- Color of point depends on how ray touches surface:
 - Angle of incidence
 - Angle of the light source
 - Distance / number of bounces
 - Material of the object

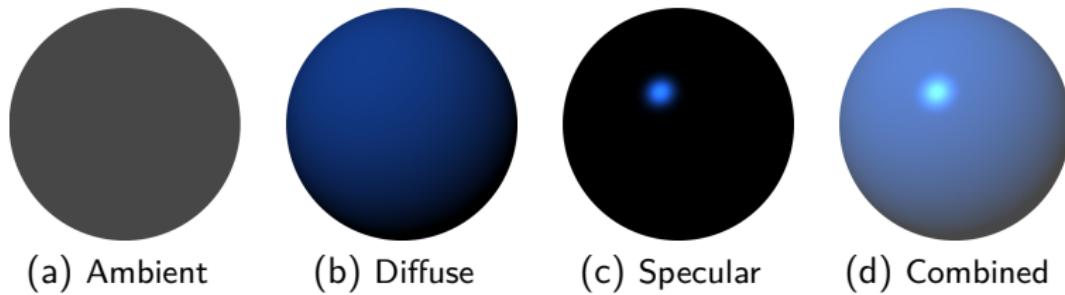
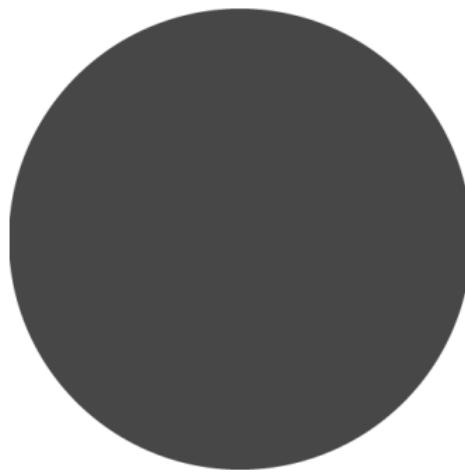


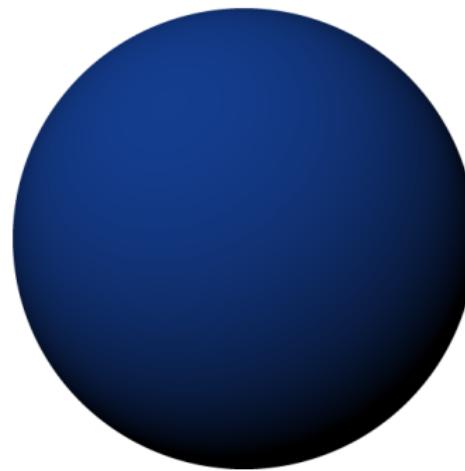
Figure 8: Phong shading model — light components are computed in steps.

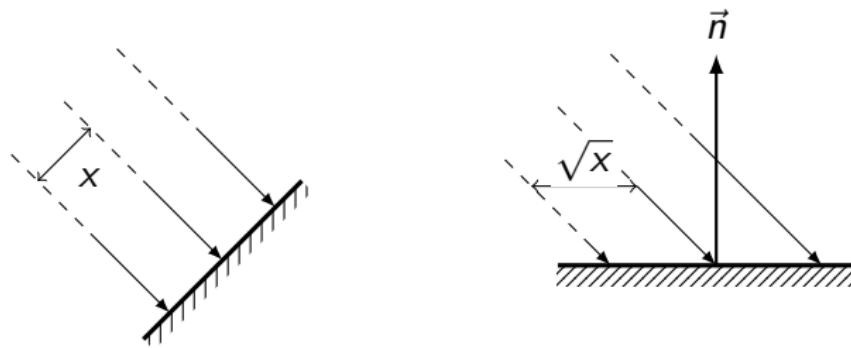
- Ambient lighting hack: $\vec{c}_a = l_a \vec{l}_c$



- Diffuse light, approximation using angle with normal:

$$\vec{c}_d = [\vec{l}_{cisl}(\vec{l}_d \cdot \vec{n})] \cdot (\vec{m}_c m_d)$$

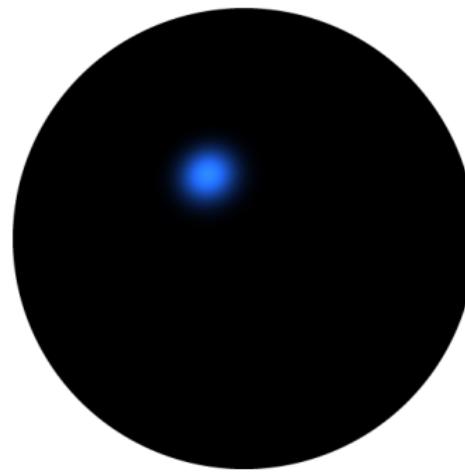




(a) Rays parallel with normal (b) Rays at 45° with normal

Figure 9: Impact of normal angle for diffuse light

- Specular light, using reflected rays: $\vec{c}_s = (\vec{l}' \cdot \vec{r})^{m_{sh}} m_s \text{isl}(\vec{l}_c \cdot \vec{m}_c)$



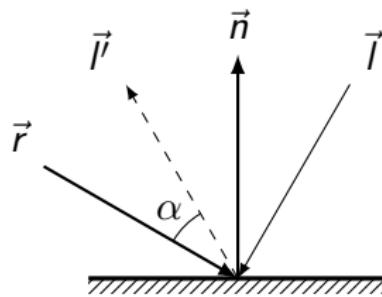
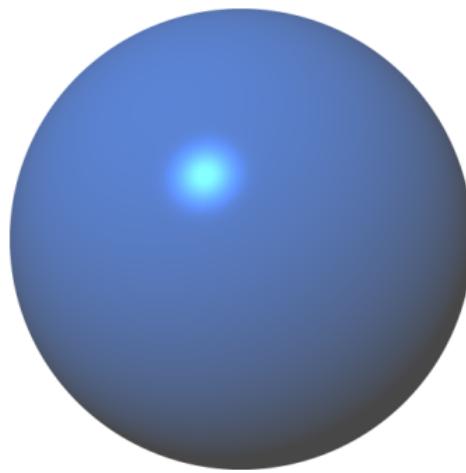


Figure 10: Angle between reflected light ray and primary ray

- Final composite color for a pixel: $\vec{c}_a + \vec{c}_d + \vec{c}_s$



- Inverse square law: $\text{isl} \propto \frac{1}{t^2}$

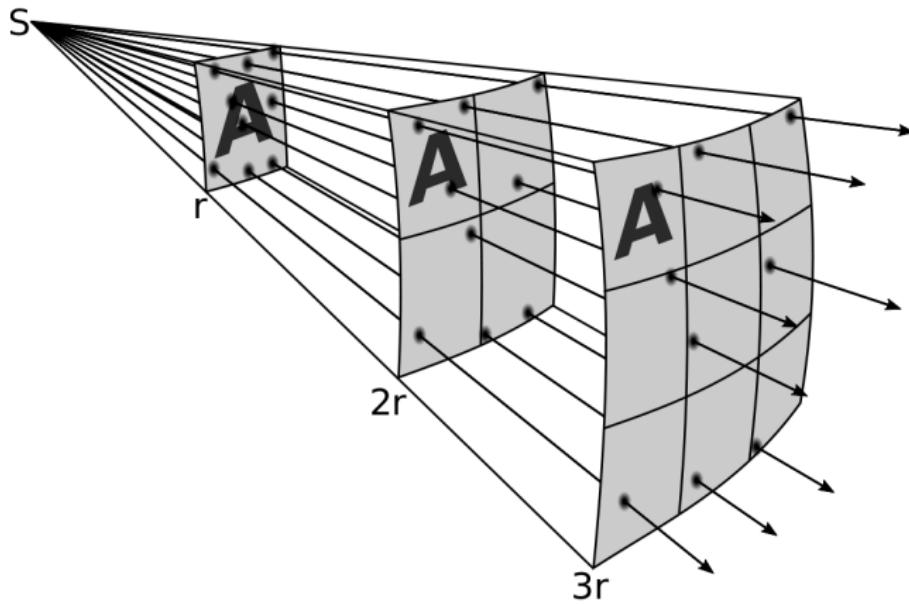


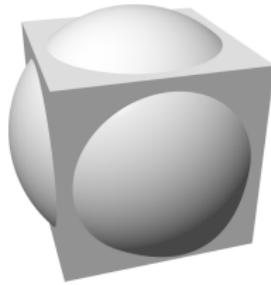
Figure 11: Demonstration of the inverse square law

Primitives

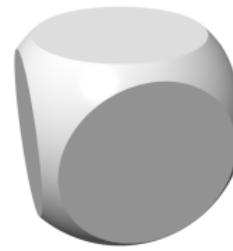
- For each type of volume, compute its intersection with a ray
- Requires solving vector equations

CSG

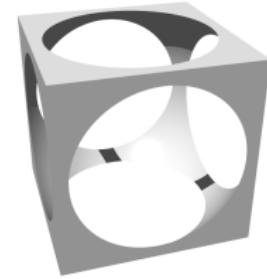
- Primitives are too simple for artistic creativity
- Constructive Solid Geometry combines objects in 3 ways:
 - Union
 - Intersection
 - Difference



(a) Union



(b) Intersection



(c) Difference

Figure 12: All three CSG types

- Recursive implementation
- Compare distances, see what object gets hit first
- Continue to the end of the objects
- Return appropriate normals and distances

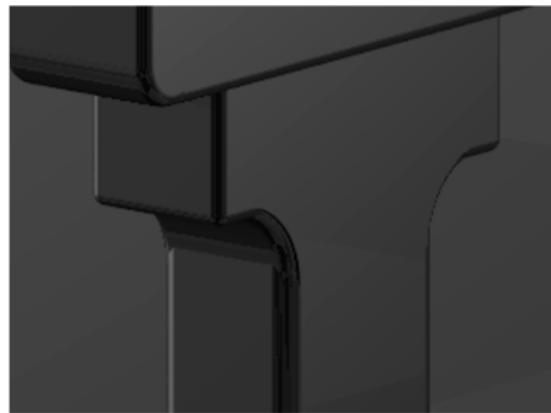
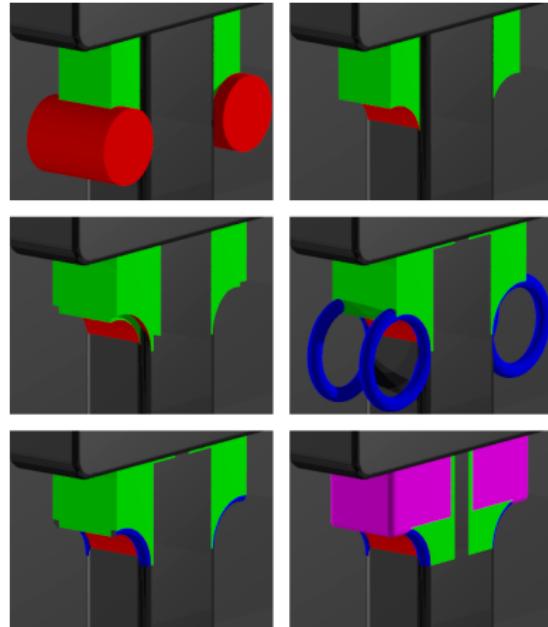


Figure 13: Concrete example of CSG use

Animations

- Simple system
- User can use variable t in code
- User chooses a number of frames and the animation is rendered

Also implemented

Other notable features:

Depth of field

Shifting the ray's origin with a random offset

Supersampling

Split each pixel into subpixel coordinates, trace multiple rays

Backgrounds

Spherical mapping of 360° panoramas

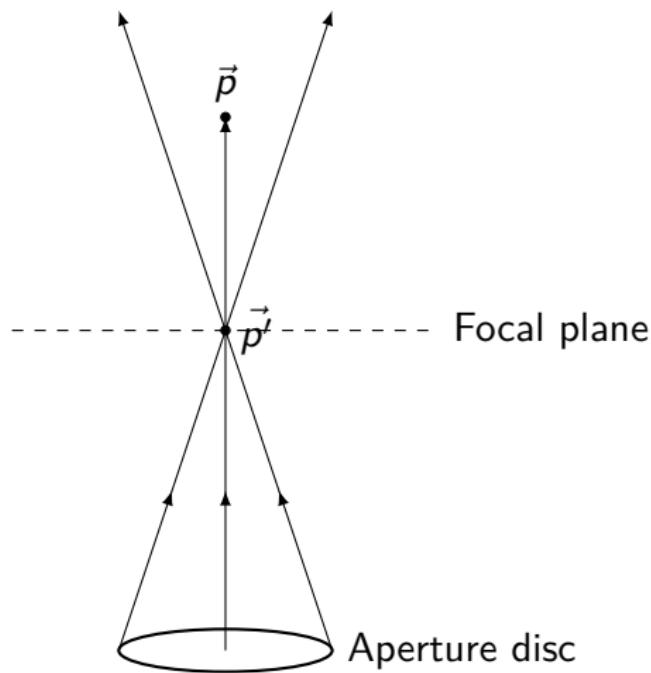


Figure 14: Disc-shaped aperture

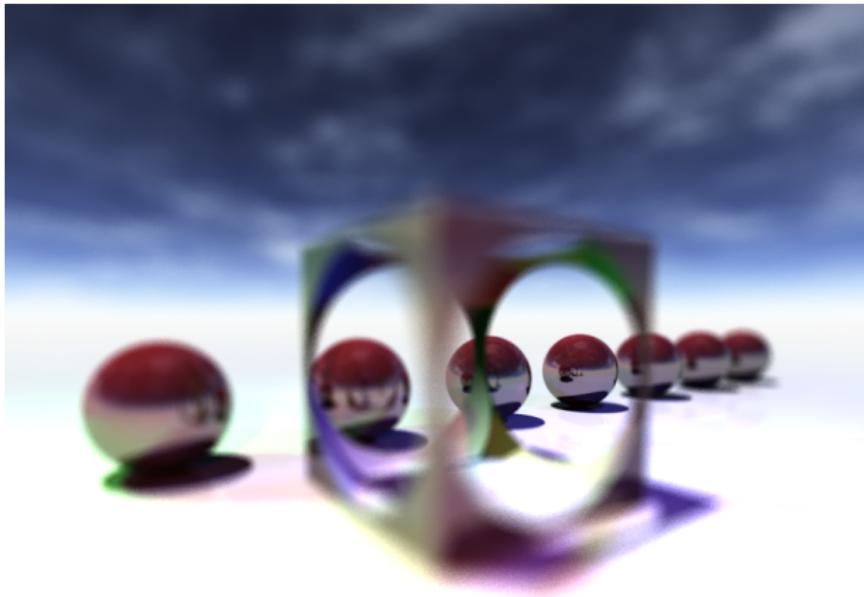
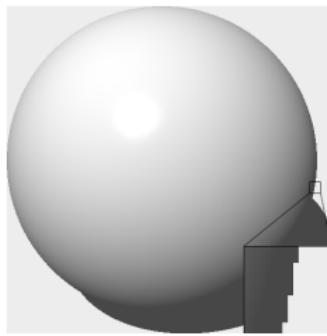
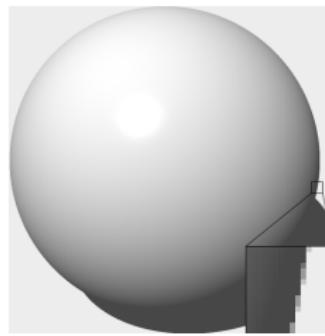


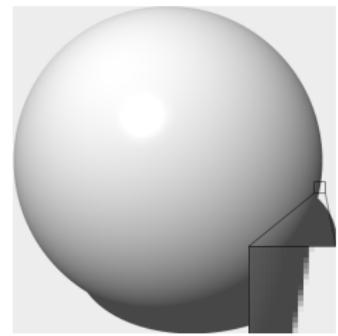
Figure 15: Depth of field example



(a) No supersampling

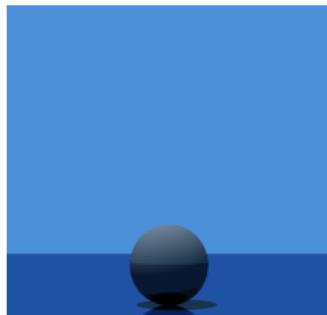


(b) 2×2 SSAA

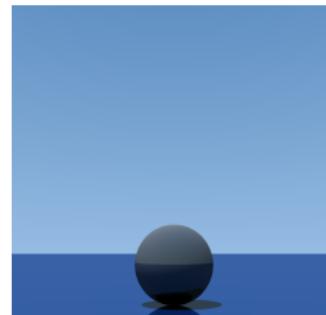


(c) 4×4 SSAA

Figure 16: Different supersampling values



(a) Solid colour



(b) Gradient model



(c) Panorama projection

Figure 17: All 3 different background types

CRT Language

Goal

- Simple language
- Control every aspect of the rendering
- Define user variables, references
- Simple loops and conditions
- Macros

Aspect

```
1 --Entities-----
2 sphere1 = Sphere {
3     center -> vec3(0, 0.5, 0)
4     radius -> 0.5
5 }
6
7 --Constants-----
8 n = 18
9 max = (3 * n) / 4 + 5
10
11 --Macros-----
12 myMacro = Macro (arg1) {
13     i = 0
14     -- Draw sphere1 "max" times on the x axis
15     while (i < max) {
16         sphere1 <translate vec3(i*1.0, 0.0, 0.0)>
17         i = i - 1
18     }
19 }
```

Listing 1: Sample CRT script

ANTLR

- Provides easy syntax for grammar
- Generates all the lexer and parser code
- Also generates base classes for *visitors* and *listeners*

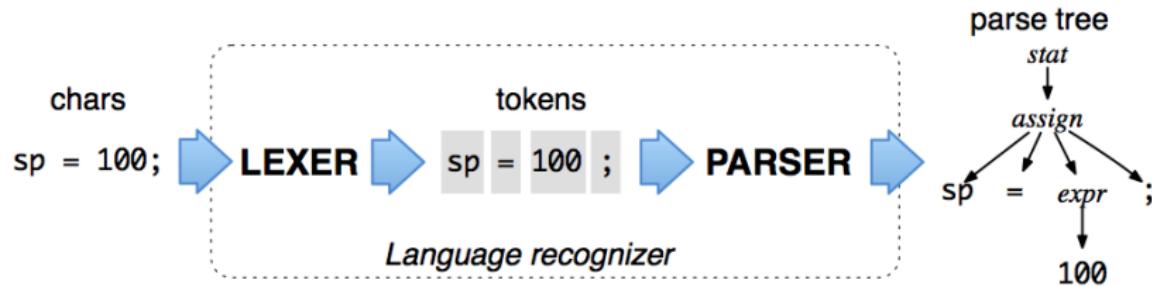


Figure 18: ANTLR4 language recognition process

Grammar

- Non-ambiguous design (context-free)
- Left-recursion for simplicity
- Labelling of alternatives generates a visitor method for each

Compiling

- Top-down process
- Many tests on types, can't use polymorphism
- Accent on exceptions and error handling
- Recursive variable solving
- Nested scopes resolution

Interface

Text editor

- Syntax highlighting using lexer
- Occurrences highlighting

```
    //  
80 r = 0.3  
81 sph2 = Sphere {  
82     center -> vec3(-1.0, r - 0.5, -1.0)  
83     radius -> r  
84     material -> redmat  
85 }  
86 sph3 = Sphere {  
87     center -> vec3(-1.0, r - 0.5, 2.0)  
88     radius -> r  
89     material -> redmat  
90 }  
91 sph4 = Sphere {  
92     center -> vec3(-1.0, r - 0.5, 5.0)  
93     radius -> r
```

Figure 19: Syntax highlighted code editor

Graphs

- AWT
- Multiplatform
- No third party dependency
- Uses obscure Java APIs

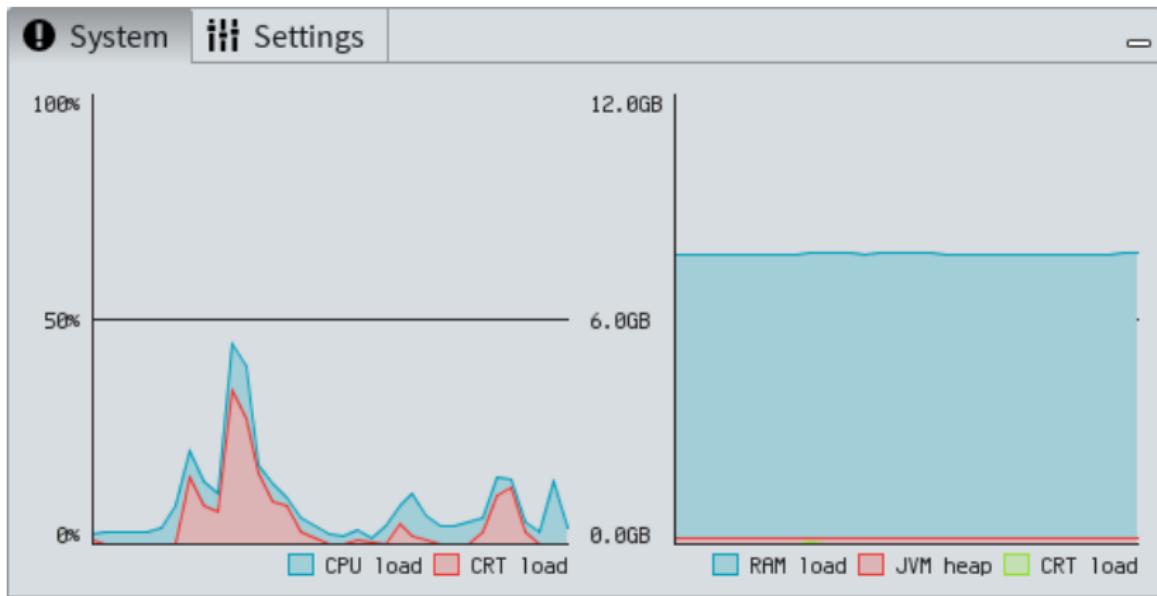


Figure 20: Custom graphs

Console

- Intercepts `stdout` and `stderr`
- Custom `PrintStream`

Property sheet

- NetBeans Open Platform API
- Java Beans
- Custom editors

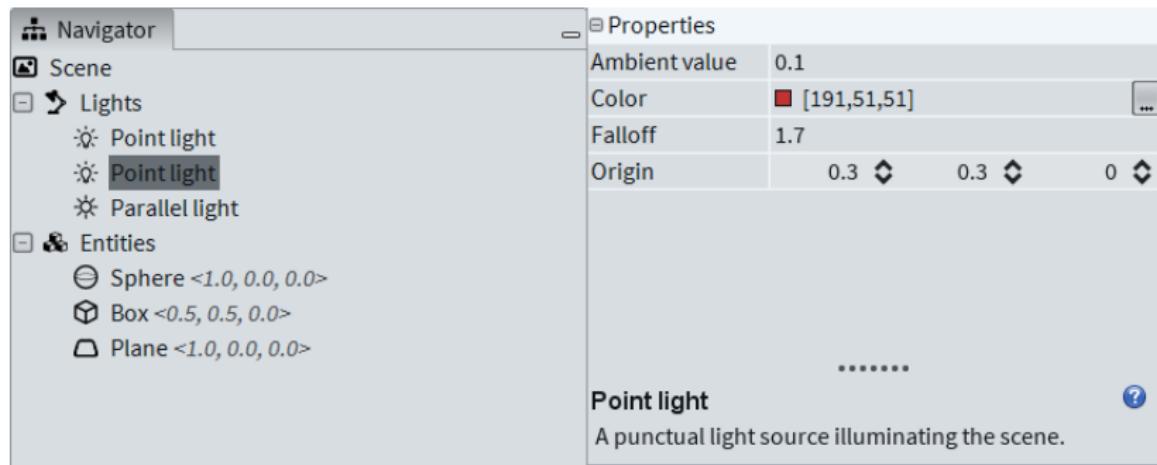


Figure 21: Property tree and editor

Docking frames

- Easy to use
- Robust
- Hard to customize

└ Interface

Figure 22: Example of user-modified layout

Live view

- Doesn't exist in current pure Ray Tracing applications
- Gives the user a good feel of where the objects are placed
- A converter takes a `scene` object and generates corresponding OpenGL constructs

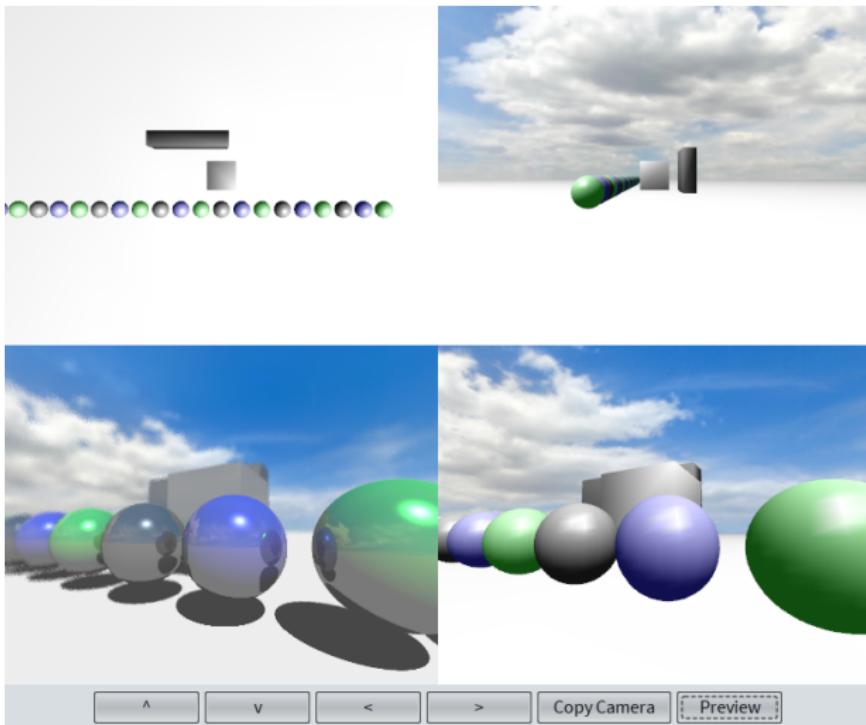


Figure 23: Live view module

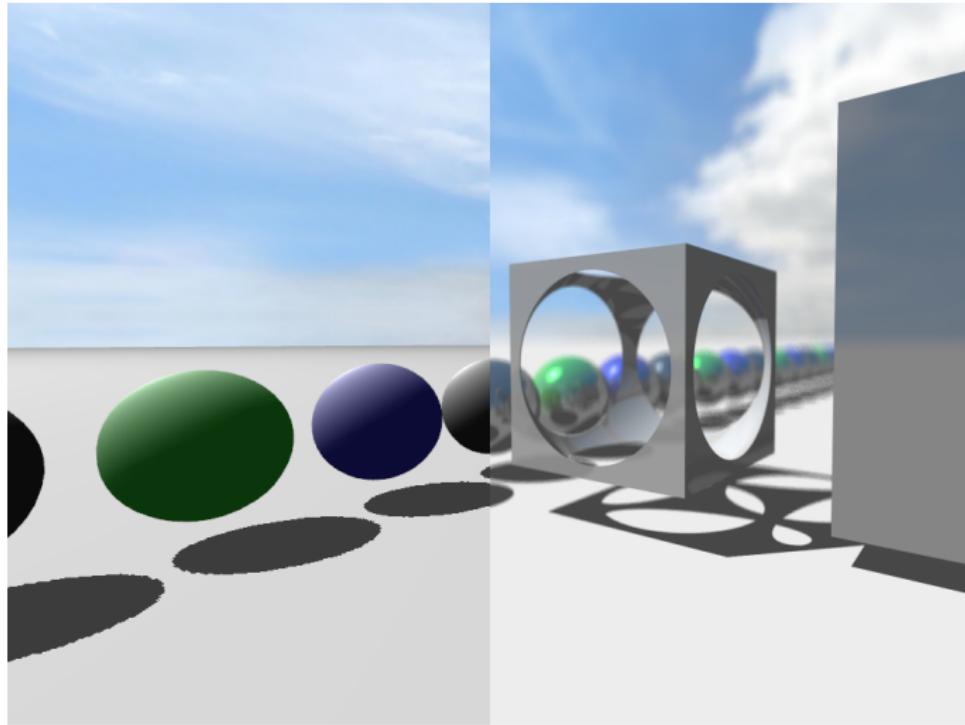


Figure 24: The live view confirms homemade ray-tracing computations

Demonstration

Conclusion

- Very complete project, involved a lot of fields
- Big scope, hard to implement all the ideas
- Good end result, usable as is

Continuation

- Drag-and-drop
- Code generation
 - Property sheets change code
 - Settings tab changes code

Q&A