

CRT – Intermediary Presentation

Hamza Haiken

24 June 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

Goal

- Artistic conception tool
- Realistic 3D images via ray tracing
- Physical light simulation
- Custom language

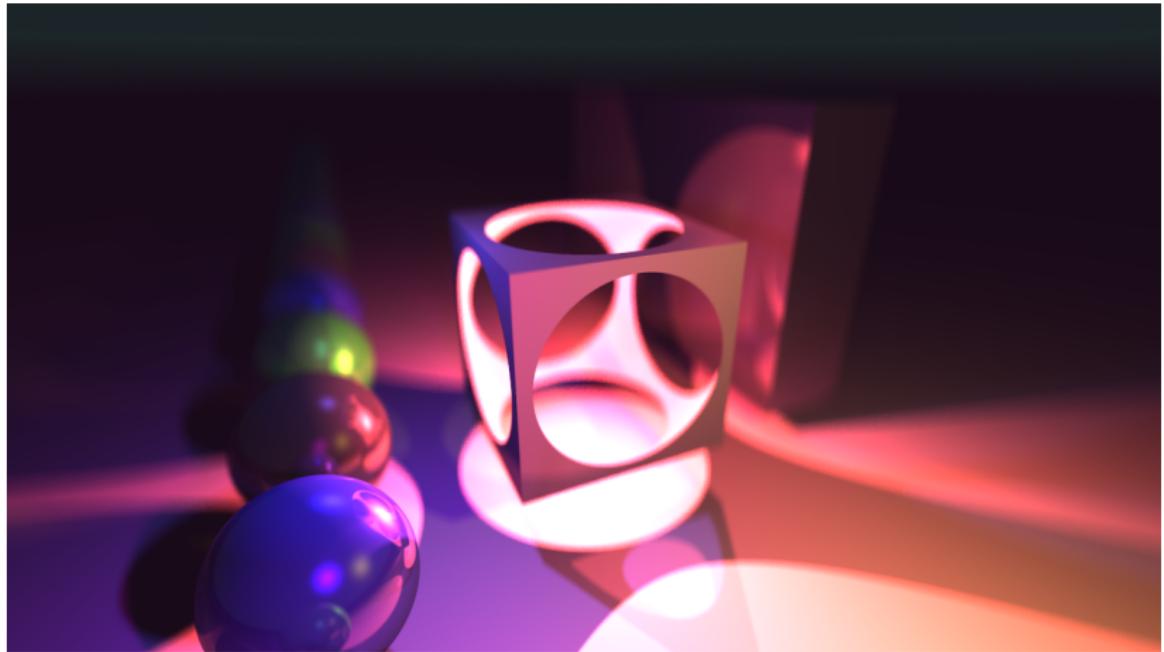


Figure 1: Example of a rendered 3D image

Technologies

- Java 8
- ANTLR4
- OpenGL (jPCT, LWJGL)

Ray Tracing

Classical method

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

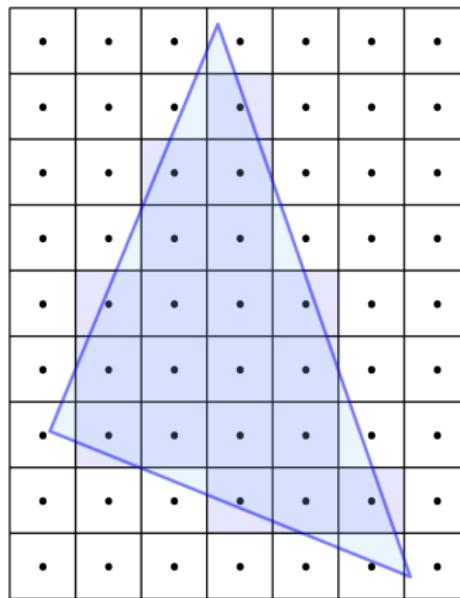


Figure 2: Rasterisation

Principle

- Simulate light's path and interactions with environment
- Objects absorb color from rays hitting them, makes colors
- 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

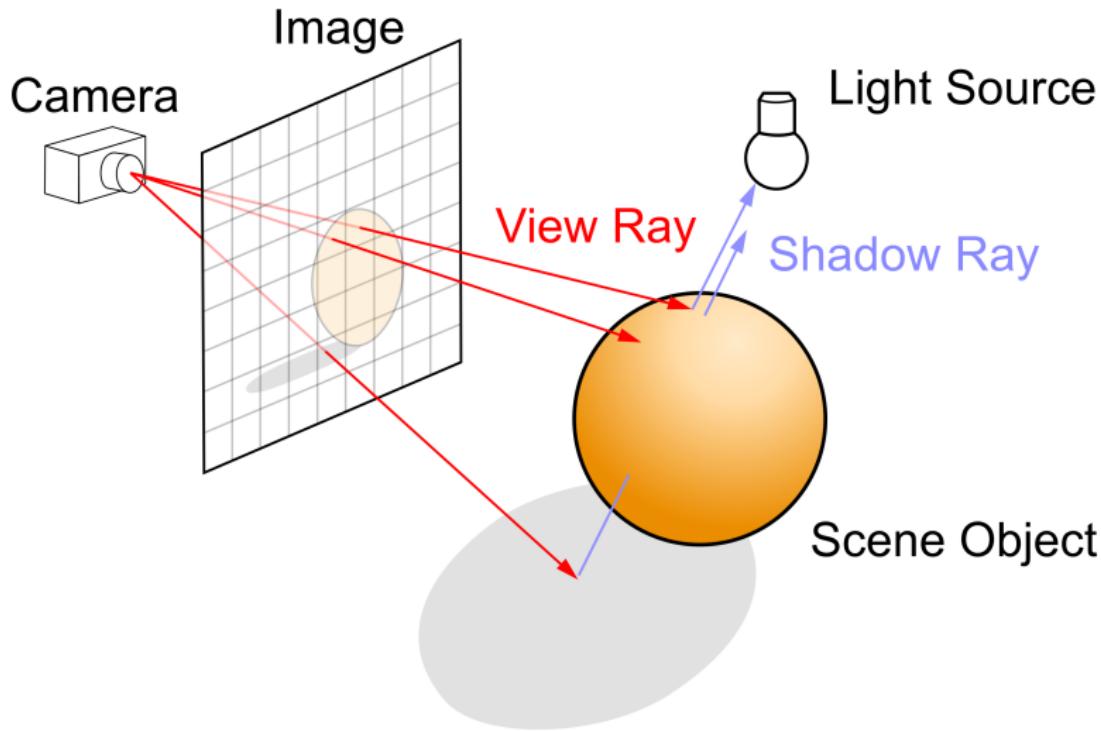


Figure 3: Backward-tracing

Modelization

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

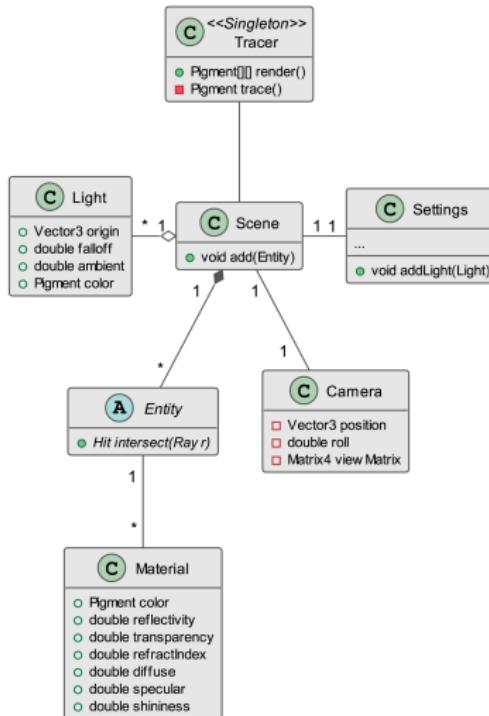


Figure 4: Rendering model

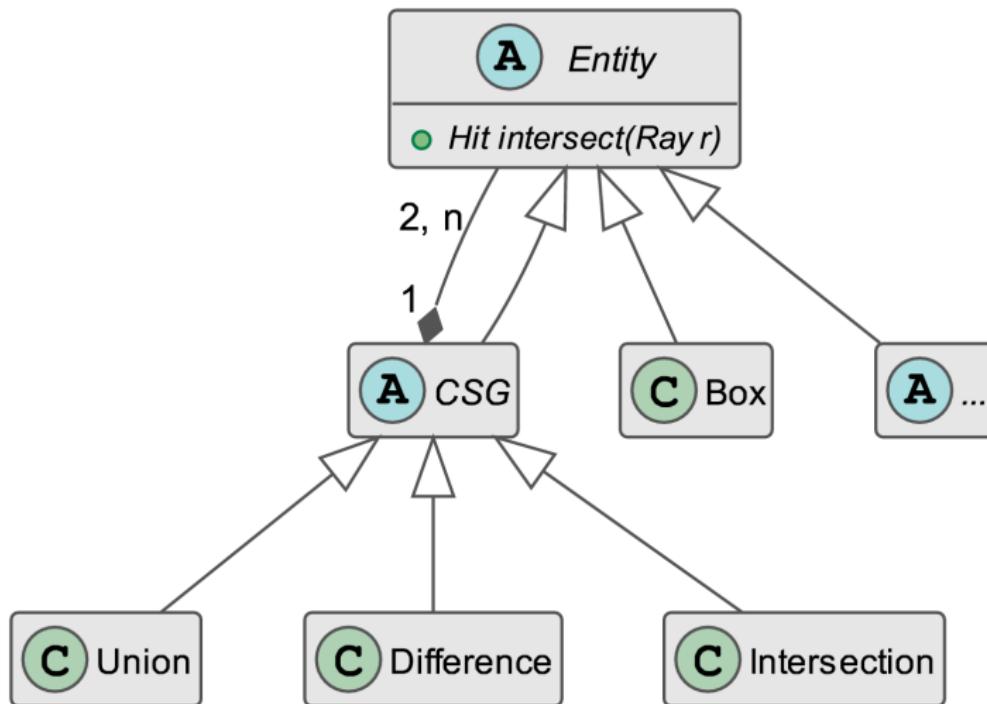


Figure 5: Entity model

Generating rays

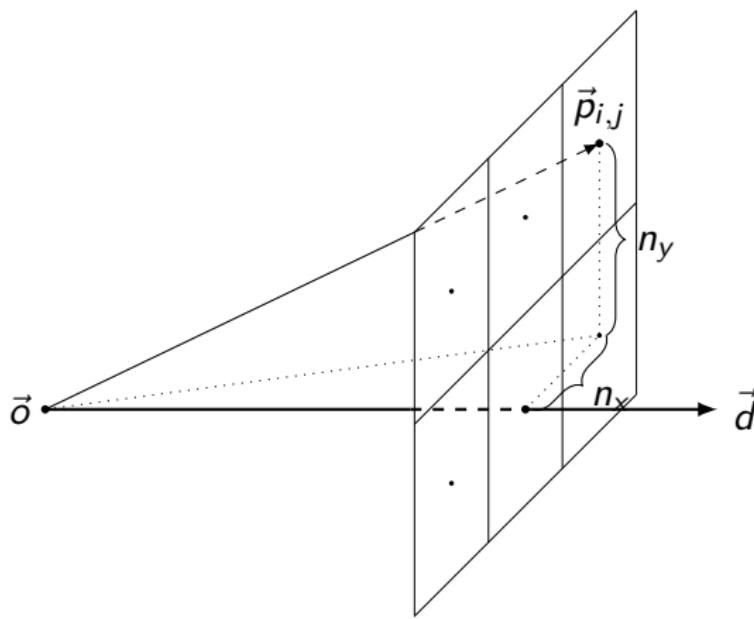


Figure 6: Primary ray generation

Shading

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

- Inverse square law: $I \propto \frac{1}{t^2}$
- Ambient lighting hack: $\vec{c}_a = I_a \vec{l}_c$
- Diffuse light, approximation using angle with normal:
$$\vec{c}_d = [\vec{l}_c I(\vec{l}_d \cdot \vec{n})] \cdot (\vec{m}_c m_d)$$
- Specular light, using reflected rays: $\vec{c}_s = (\vec{l}' \cdot \vec{r})^{m_{sh}} m_s I(\vec{l}_c \cdot \vec{m}_c)$
- Final composite color for a pixel: $\vec{c}_a + \vec{c}_d + \vec{c}_s$

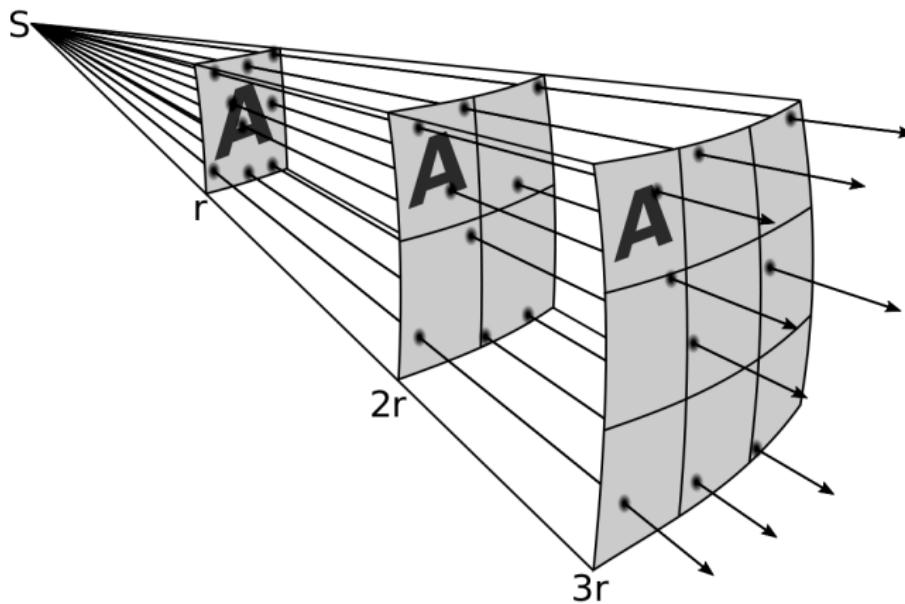
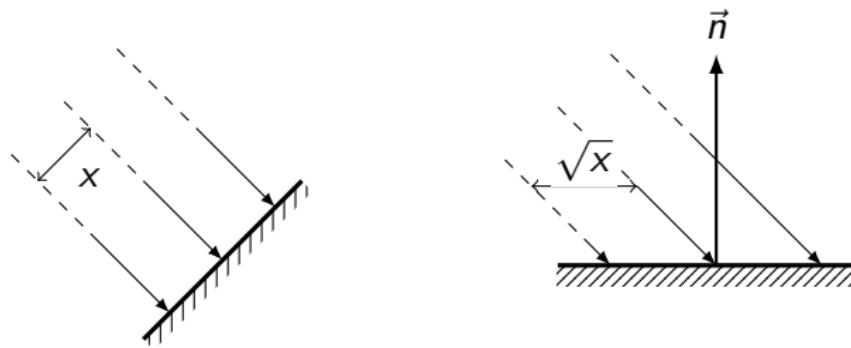


Figure 7: Demonstration of the inverse square law



(a) Rays parallel with normal

(b) Rays at 45° with normal

Figure 8: Impact of normal angle for diffuse light

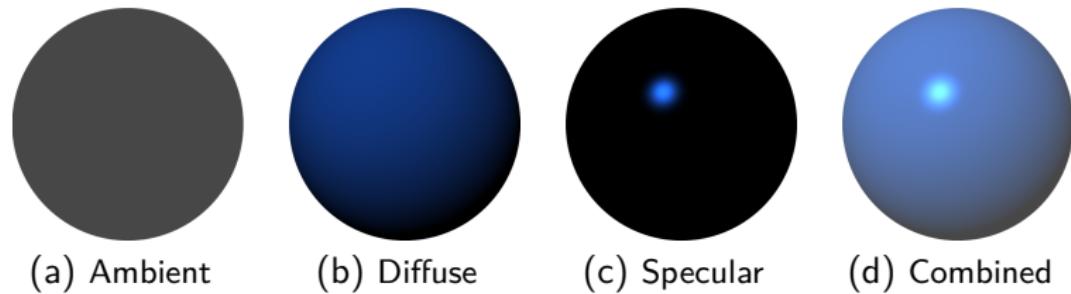


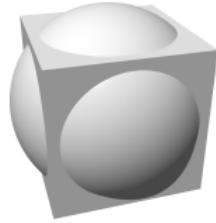
Figure 9: Phong shading model — light components are computed in steps.

Primitives

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

CSG

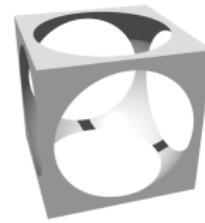
- Primitives are too simple for artistic creativity
- Constructive Solid Geometry combines objects in 3 ways:
 - Union
 - Intersection
 - Difference



(a) Union



(b) Intersection



(c) Difference

Figure 10: 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

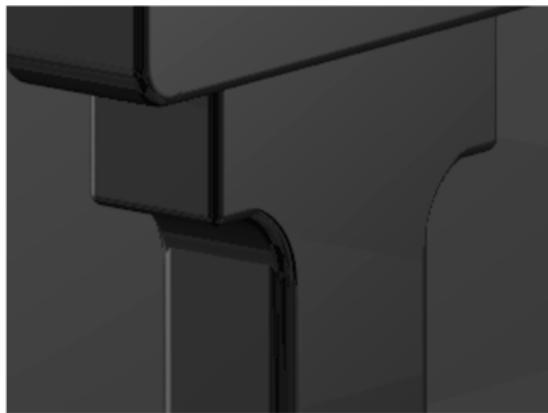
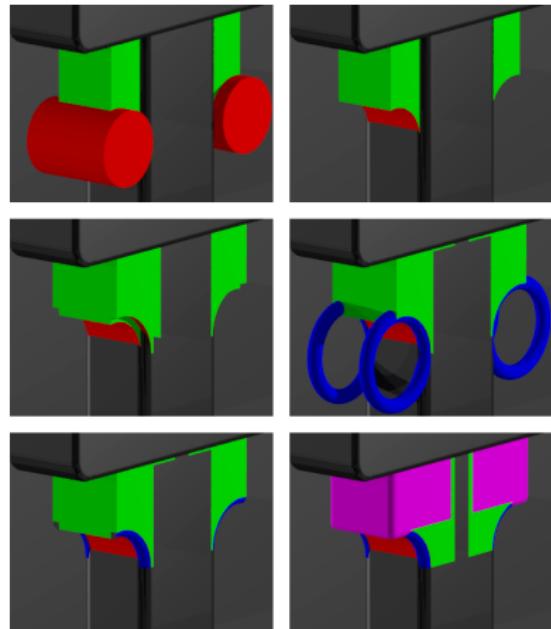


Figure 11: Concrete example of CSG use

Also implemented

Other notable features:

Depth of field

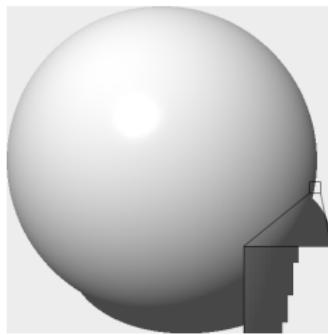
Shifting the ray's origin with a random offset

Supersampling

Split each pixel into subpixel coordinates, trace multiple rays

Backgrounds

Spherical mapping of 360° panoramas



(a) No supersampling

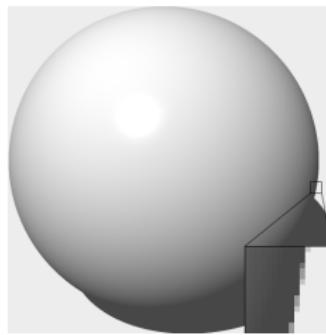
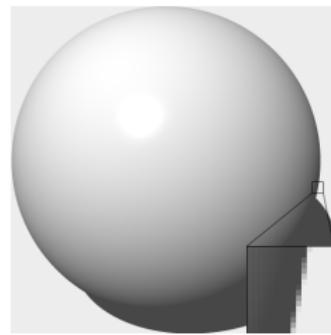
(b) 2×2 SSAA(c) 4×4 SSAA

Figure 12: Different supersampling values

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*

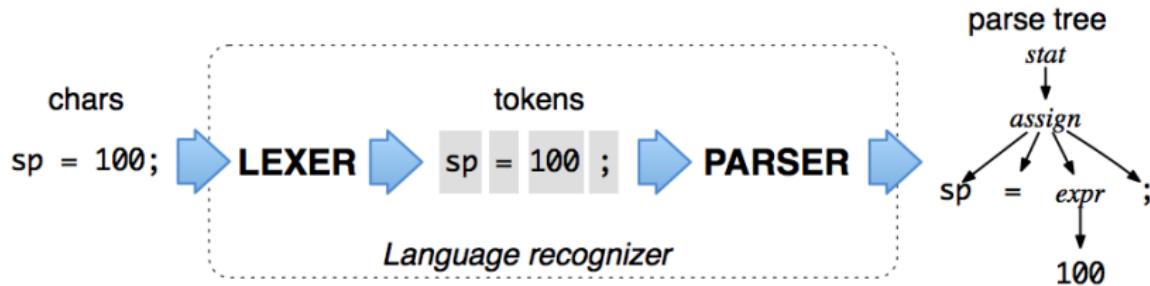


Figure 13: ANTLR4 language recognition process

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 14: Syntax highlighted code editor

Graphs

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

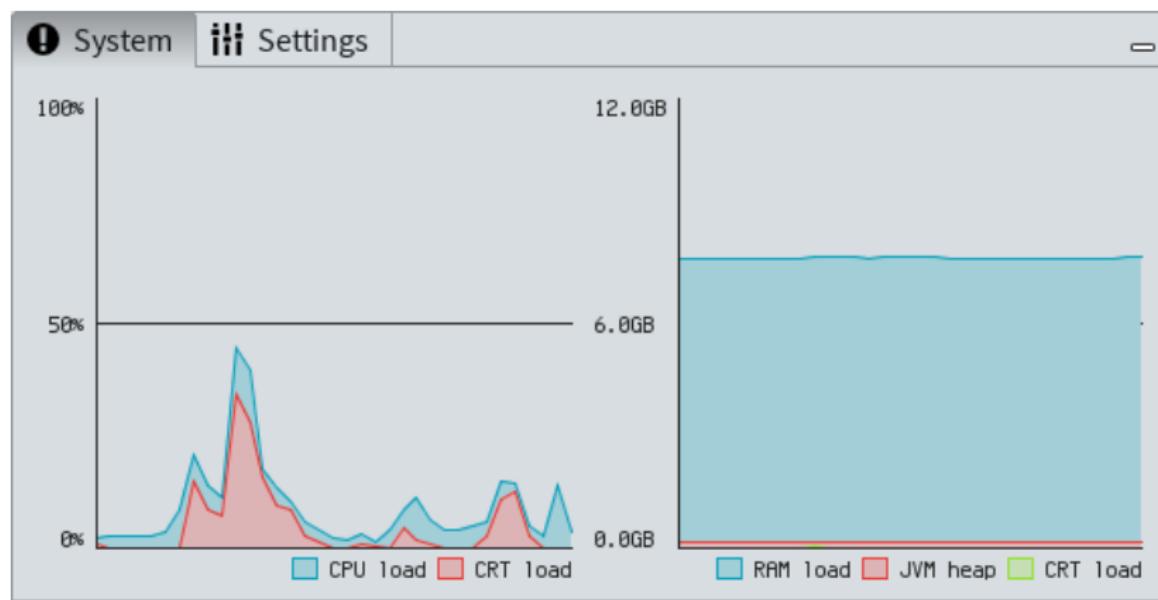


Figure 15: Custom graphs

Console

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

Property sheet

- NetBeans Open Platform API
- Java Beans
- Custom editors

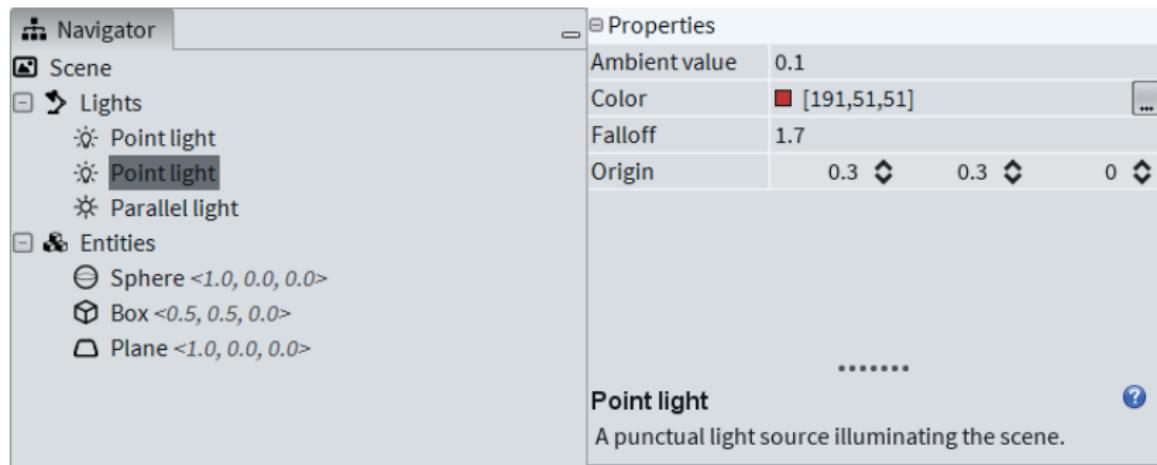


Figure 16: Property tree and editor

Docking frames

- Easy to use
- Robust
- Hard to customize

Live view

Demonstration

Conclusion

What remains ?

- Animations
- Drag-and-drop
- Code generation
 - Property sheets change code
 - Settings tab changes code
- User interface
 - All menus and buttons
 - Configuration window
 - Perspectives
 - Undo / Redo