Lisp in Summer Projects Submission

Submission Date	2013-10-24 15:35:47
Full Name	Pius von Däniken
Country	Switzerland
Project Name	Celine - cel-shading raytracer written in clojure
Type of software	command-line/terminal app
General category	art
LISP dialect	Clojure
GitHub URL	https://github.com/31415us/celine
Did you start this project?	Yes, all the code is written by me
Project Description	I want to describe my project in this form.
Purpose	Celine is a simple ray tracing program. Instead of using state-of-the-art realistic shading techniques it implements a very simple cel-shader (see: https://en.wikipedia.org/wiki/Cel_shading)
Function	Celine permits to render a scene defined in the core.clj file and outputs a .png file containing the rendered scene. The scene can be built out of several geometric primitives, at the moment there are spheres, planes, triangles and triangle-meshes implemented.
Motivation	There seem to be plenty of resources on how to implement a ray-tracer in a standard object-oriented and imperative fashion, but only few seem to have considered an implementation in a functional environment. Similarly cel-shading doesn't seem to be as popular as more traditional shading techniques. The main motivation was to explore these two fields without much outside help and at the same time learn a lisp.
Audience	There is no specific target audience. As the code is still relatively raw Celine doesn't yet permit people unfamiliar with the topic to explore ray tracing in an intuitive fashion. People looking for a simple implementation of cel-shading in a ray-tracing context might be interested to look at the code

as there aren't a lot of resources to be found (there are some nice tutorials on how to implement it in OpenGl).

Methodology

Quick overview of ray-tracing and cel-shading:

Ray-tracing:

Lightsource
Image Plane o
Eye/Camera |
/ | _
| ---------------|--------->| |
Ray | |_|
| Object in Scene

In its simplest form a ray-tracer shoots a ray from the eye of the observer through some pixel position in a virtual image plane and tests for intersections in the scene. From the intersection point you get the base color of the pixel. From there you can cast several new ray to calculate reflections and shadows. To see if some intersection point lies in the shadow of an other object you simply cast a new ray in the direction of the light source and test for collision with other objects. Similarly for reflections you calculate the reflection angle after some shading model (e.g. phong-shading). Implementing reflections was never a goal of Celine and they're therefore not present in the current implementation. Shadows we're planned but not yet implemented.

Cel-Shading:

As opposed to more realistic-looking shading models cel-shading gives the scene a comic book style look and feel (and is therefore often also called toon-shading).
To get this look you try to get hard edges between different shades of the same color instead of more soft gradients. This is achieved by looking at the angle between the normal vector at the intersection point and the position of the light source with respect to the intersection point. You then brighten or darken the base color as a function of this angle. Normally you also want to get (black) outlines around rendered objects
(to emulate pen strokes), unfortunately it was hard to find any resources on how to implement this in a ray-tracer and is not implemented (yet).

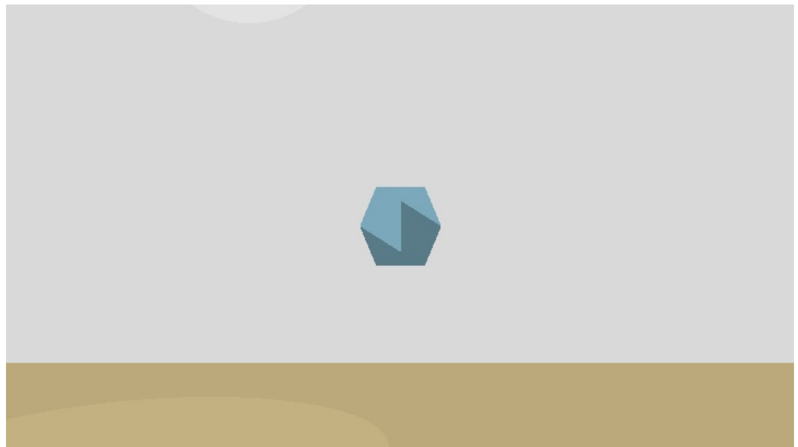
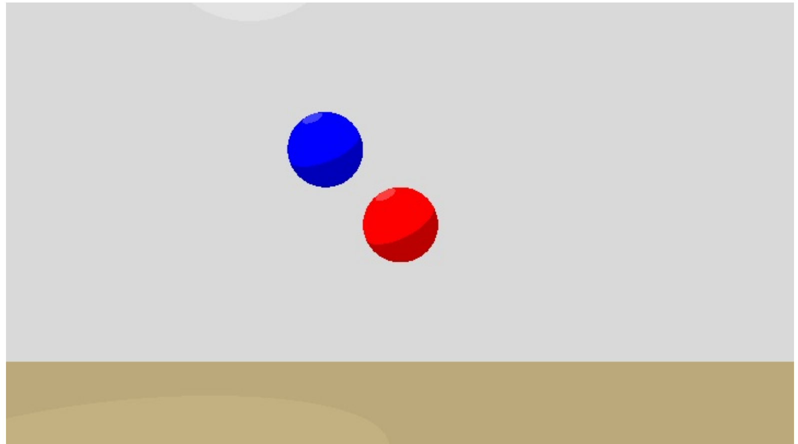
Parallelism:

Knowing that we can compute the color of every pixel independently of the others, ray-tracing can in theory be easily parallelized. Unfortunately this is not implemented yet.

Geometry:

All geometric computations are made in standard euclidean 3D space. The geometry.clj module implements standard vector algebra operations on 3D vectors and offers several geometric primitives such as spheres, planes and triangles. The workhorse of any ray-tracer are its ray-object collision detection functions. In this domain Celine has still a lot of space for improvement such as the use of standard octree space partitioning to avoid unnecessary collision tests.

	Celine does not depend on any external library other than Clojure and Java standard libraries and is entirely built from scratch.
Conclusion	In its current state Celine works as a very simple ray-tracer although much of the planned functionality is not yet implemented. The main drawback at the moment is that people unfamiliar with the code will probably find it very hard to interact with Celine, as there is no way to do so other than manually modifying the core.clj file to change the scene to render. Furthermore the current implementation is very inefficient as it does not yet take advantage of the possibility to parallelize and handles spacial information for ray-object collisions very poorly (at the moment it checks collision between every ray and every object which could be avoided). On the other hand it lends itself well for inspiration for future better implementations as there aren't many resources on the subject of the combination of ray-tracing and cel-shading. Future improvements: - shadows! - parallelize - more efficient space partition and ray object collision detection - better UI
Build Instructions	It's a Leiningen project so to run just do: cd ~/path/to/celine/ lein compile
Test Instructions	There are no unit-tests implemented.
Execution Instructions	To run the code just do: lein run in the project-folder In the version uploaded it should render ./obj/icosahedron.obj in blue on black background
Describe any bugs or caveats	The code is extremely slow so don't try to render anything with a lot of geometric primitives. The .obj parser is very rudimentary and will probably fail on most input. DONT try to render the ./obj/teapot.obj: due to the inefficient implementation it takes forever to render!

Screen shots	 icosahedron.png  two-spheres.png
Official	I have read rules and have abided by them. I am 18 years of age or older. I am not living in Brazil, Quebec, Saudi Arabia, Cuba, Iran, Myanmar (Burma), North Korea, Sudan, or Syria.