Projet INF442
Writing a distributed ray-tracer (*)

Stephane Redon
stephane.redon@inria.fr

X 2014
March 31, 2016

Abstract

The goal of this project is to write a ray-tracer (http://en.wikipedia.org/wiki/Ray_tracing_(graphics)), and make your own computer-generated images. In this project, the objects will be spheres.

1 Basics

1.1 Vectors

Define a class to represent 3D vectors. This class should handle basic operators with vectors (e.g. multiplying by numbers, adding vectors, etc.). You may add more functionality as needed throughout the project.

1.2 Ray data structure

Define a data structure for a ray, which includes both a point (a point passing through the ray) and a vector (to give the direction of a ray)

1.3 Spheres

Define a class to represent spheres (a sphere has a center and a radius). Add a color (three components: red, green and blue).

1.4 Scene

Define a class to represent a scene, i.e. a set of spheres.

1.5 Light

Define a class to represent a point light source. Optionally, the light source could have a non-white color.

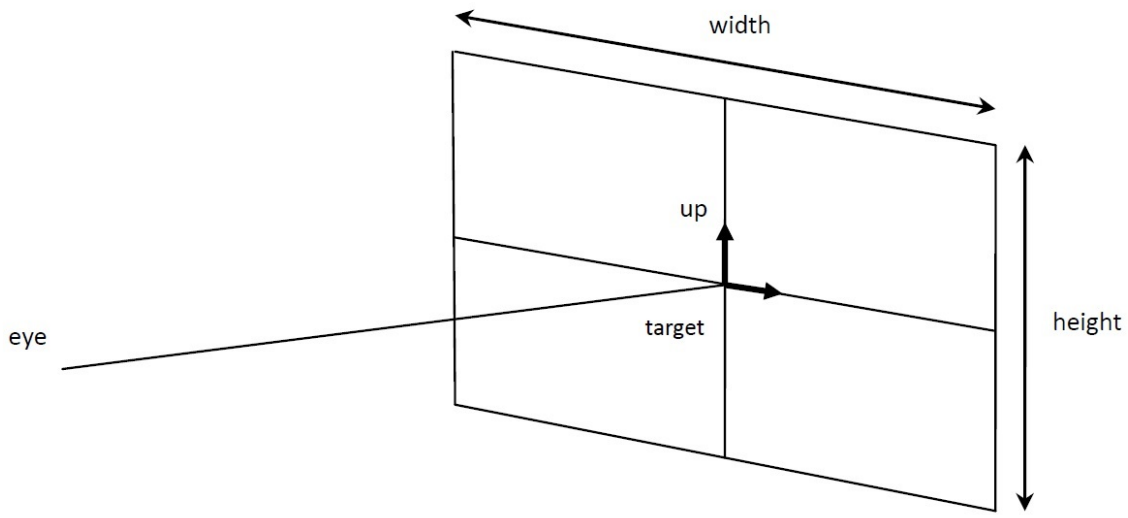

Figure 1: The camera model.

1.6 Camera

Define a class to represent a camera, i.e. a vector representing the "eye", a vector representing the "target" (looked by the camera), a "up vector" representing the orientation of the camera, and "screen dimensions" (see Figure 1).

2 First algorithms

2.1 Ray-sphere intersection

Implement an algorithm to determine whether a ray intersects a sphere. Calling the function should allow to obtain a boolean that represents whether there is an intersection or not, as well as the intersection point (hint: a quadratic equation should be solved).

2.2 Lighting model

Given a point on a sphere, with a given *surface normal* (in the case of a sphere, the vector which goes from the center of the sphere to the point on the surface of the sphere), compute the color of the point using the *Phong reflection model* (`http://en.wikipedia.org/wiki/Phong_reflection_model`).

2.3 Ray-traced image

Implement a first backward ray-tracing algorithm: for each point of the image, cast a ray from the eye of the camera to the point, and detects the intersection between the ray and the spheres in the

scene. If the ray intersects a sphere in the scene, compute the color of the point. Else, the image point has a default color (e.g. white for the background).

3 Shadows

Implement shadows in your ray-tracing algorithm. When a ray hits a sphere in the scene, determine whether a ray from the light to the intersection point hits *another sphere*. If this is the case, the color of the intersection point should be darkened to reflect the fact that it is in the shadow.

4 Reflection

Implement reflections in your ray-tracing algorithm. Add a "reflection coefficient" r to your spheres, and make your ray-tracing algorithm *recursive*: when a ray hits a sphere, generate a new ray starting from the intersection point, in the reflected direction (computed from the initial ray and the surface normal). The final color is $r \times c_r + (1 - r)c_s$, where c_r is the (recursively computed) color seen by the reflection ray, and c_s is the color of the initial intersection point on the sphere.

5 Distributed ray-tracing

Use MPI to compute your image in a distributed fashion. For example, one possibility would be that each node would compute a sub-region of the image, that the main node would collect.