Prof. G. Zachmann
J. Teuber

University of Bremen
School of Computer Science
CGVR Group
April 22, 2015

Summer Semester 2015

Assignment on Advanced Computer Graphics - Sheet 2

Due Date 05. 05. 2015 11:59pm
joern.teuber@cs.uni-bremen.de

Exercise 1 (Raytracing Basics, *10 Credits*)

In this exercise, you should complete our raytracing framework. To do that, you have to implement some functions in this framework.

First, please take some time to become familiar with the structure of the framework. The hints in the tutorial class, but also the comments in the code should help you.

Basically, the framework consists of four essential components:

- **GUI:** The GUI is based on Qt4. In order to compile the framework you will need a version of the Qt-framework to be installed on your PC (`http://qt-project.org/`). In the window on the right side, the GUI shows the resulting image that is generated via raytracing. The window on the left side shows an OpenGL rendering of the scene. You can use it for debugging and for a comparison between the global illumination generated by raytracing and the local OpenGL lighting.

  However, OpenGL and Qt are not necessary for the understanding of the raytracer; therefore, you should simply ignore this part of the code.
- **XML Parser:** The program reads 3D scenes in a simple XML format. We use a freely available XML library for parsing. You will find a pre-compiled version for windows in the zip-file. If you use Linux or Apple, you have to compile the library from the sources. You can find a `Makefile` in the `xmlParser` sub-folder.

  The XML file format should be almost self-explanatory. However, you don't have to create scenes by your own, you can simply use the pre-defined example scenes (`objects.xml`, `glass-spheres.xml`, `metal-spheres.xml` or `steinbach.xml`).
- **Mathematical Helper Classes:** You will find some simple template-classes to simplify mathematical computations:
  - `VectorT`: template for n-dimensional vector computations
  - `MatrixT`: template for quadratic n×n matrices
  - `Matrix33T`: specialisation of `MatrixT` for 3×3 matrices
  - `ColorT`: template for RGB color computations
- **Raytracer:** The basic components of the raytracer are concentrated in the `Raytracer` class:
  - `Raytracer::render()`: generates a ray for each pixel.
  - `Raytracer::traceRay()`: traces the ray through the scene.
  - `Raytracer::shade()`: This function should compute the local (Phong) shading for a point in the scene. Moreover, it should test whether or not the point is in the shadow.

The `PinholeCamera` class implements a simple pinhole camera. The most important function is `PinholeCamera::generateRay()`. It generates a ray from the eye point through the pixel `(x,y)` in the image plane.

The `Ray` class represents such a ray. Basically, rays a defined by their starting point and a direction. Moreover, the class includes functions to compute the reflected and refracted rays.

All classes for entities in the scene (geometry, materials, light sources) are derived from their respective virtual base classes:

- `Surface`: virtual base class for geometric objects. All derived classes implement their own `intersect()` function that computes the intersection between the objects and a ray. The derived classes `Plane`, `Sphere` and `Checkerboard` are fully implemented.
- `Shader`: Virtual base class for materials. The derived class `PhongShader` is fully implemented. The function `PhongShader::shade()` computes the local Phong model.
- `Light`: Virtual base class for light sources. The derived classes `PointLight` and `DirectionalLight` are fully implemented.

Your tasks:

a) Implement the function `Raytracer::shade()` in the file `Raytracer.cpp`. In this function, you should test whether or not the point is in the shadow. If the point is not in the shadow, you have to set its color to the appropriate (Phong) lighting.

   If you implement this function correctly, the resulting image should look almost like the OpenGL rendering on the left side.

b) In order to realize a real raytracer, we have to trace also reflected rays. To do that, you should implement the function `Ray::reflectedRay()` in `Ray.cpp`.

   Moreover, you have to use this function in `Raytracer::traceRay()` to recursively compute the reflected color (that has to be added to the current color value).

c) Finally, we also want to include refractions into our raytracer. Therefore, you have to implement the function `Ray::refractedRay()` in `Ray.cpp`. It should, depending on the refraction parameter of the material, return the refracted ray.

   As in the previous part, you also have to trace the refracted ray recursively through the scene in `Raytracer::traceRay()` to compute the appropriate amount of color.

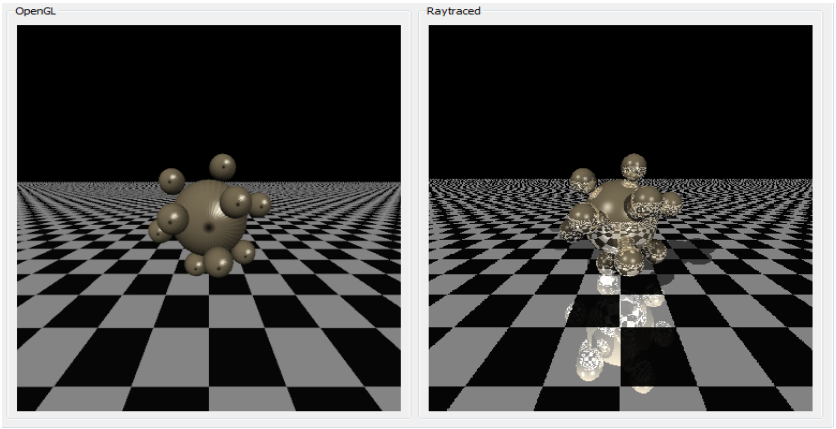

Figure 1: Correct example rendering of the metal-spheres.xml scene (left: OpenGL; right: Raytracer)