Computer Graphics
Exercise 2 - Simple Raytracer

Handout date: 14.10.2011

Submission deadline: 28.10.2011, 23:00h

You must work on this exercise **individually**. You may submit your solution by:

- Emailing a .zip file of your solution to `introcg@inf.ethz.ch` with the subject and the filename as `cg-ex2-firstname_familyname`

The .zip file should contain:

- An 800x600 final raytraced image named `cg-ex1-firstname_familyname.bmp` of the scene described below
- A folder named `source` containing your source code
- A `README` file inside the `source` containing a description of what you've implemented and compilation instructions

Do not include compiled binaries, external libraries, or the images provided. Even if you need to download GLUT binaries to run your code on Windows, you don't need to include them in your submission.

No points will be awarded unless, your source code can:

- Compile (without any external dependencies besides GLUT)
- Run and produce the same 800x600 .bmp image included in your .zip

Your submission will be graded according to the quality of the image produced by your raytracer, and the conformance of your ray tracer to the expected behavior described below.

Goal

The goal of this exercise is to implement a ray tracer with a basic functionality including

- Shooting a ray from the viewing position to the scene through each pixel of the image plane,
- Intersecting the ray with the scene objects,
- Casting shadow rays from the point of intersection to each light source to determine if the point is illuminated, and
- Shading the point according to Phong lighting model from the material properties of the point and the lighting.

What's given to you already

A small code base has already been written for you. In particular, we provide you with a small header library to handle:

- Basic matrix and vector math: `utils.h`
- Writing to .bmp image file format: `exporter.h`

We also provide the file `main.cpp` which contains the skeleton of a raytracer. As provided, this file opens a single 800x600 GLUT window and hooks up the necessary callbacks: `displayFunction()`, `reshapeFunction()`, etc. Ray tracing is driven by calling `raytraceScene()` through the right-click menu or by a keystroke '`r`'. Currently the `raytraceScene()` contains only two `for` loops, looping over the pixels in the 800x600 to set a dummy color value in the image buffer `g_buffer`. Once raytracing is done, the display callback `displayFunction()` copies the rendered image into the hardware frame buffer instead of clearing the screen black.

`main.cpp` can be easily compiled on various platforms including Mac OS X, Linux, and Windows. For Windows, you may need download GLUT. For more detail, see `COMPILE` contained in the template.

Ray casting (5 points)

Your first task is to shoot a ray into the scene for each pixel of the screen. You're given all necessary parameters to determine the origin and the direction of each ray in the scene section below. What happens to this ray in the scene will determine the color you set in the image buffer while you trace the ray as described below.

Ray-object intersection (5 points)

The second task of your ray tracer is to determine if your ray sent through a pixel on the screen hits any objects in the scene. We do not consider transparency or reflectance at the moment: occlusion in a ray tracer is handled simply by always considering the first intersection of the ray and the objects in the scene and this should be consistent regardless of the order of intersection tests.

Shadows (5 points)

The next task of your ray tracer is to determine if your ray reaches a light source (in a single bounce). If you have determined that your ray intersects an object at a certain point, you must then determine if this point on the object is directly illuminated by any lights in the scene.

Phong lighting model (5 points)

Finally once you have determined that your ray hits an object and that that intersection point is illuminated by the lights, you must use the Phong lighting model and the material properties of the object to determine the final color that the ray will return to the screen pixel.

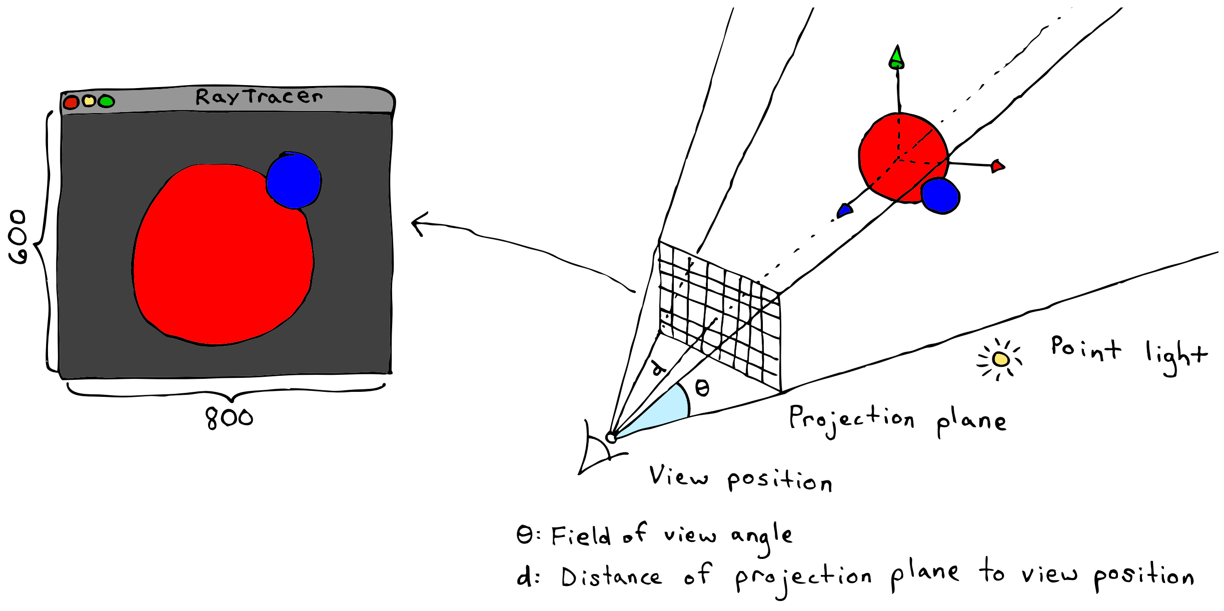

Figure 1: A demonstrative illustration of the scene and viewing arrangement.

The Scene

The scene for this assignment will be very simple. The only objects in your scene are a large, red sphere and a small, blue sphere. Each sphere has a number of properties:

Large, red sphere

Center:	(0,0,0)
Radius:	2
Ambient material color:	rgb(0.75,0,0)
Diffuse material color:	rgb(1,0,0)
Specular material color:	rgb(1,1,1)
Specular exponent:	32.0

Small, blue sphere

Center:	(1.25,1.25,3)
Radius:	0.5
Ambient material color:	rgb(0,0,0.75)
Diffuse material color:	rgb(0,0,1)
Specular material color:	rgb(0.5,0.5,1)
Specular exponent:	16.0

There will be a single, point light illuminating your scene. Its color should be **rgb(1,1,1)** and location **(10,10,10)**. Its ambient, diffuse and specular intensities should be respectively **0, 1 and 1**. In addition, you should add a **global ambient intensity of 0.2**.

The scene should be viewed from position **(0,0,10)** with viewing direction **(0,0,-1)** and up direction **(0,1,0)**

Your final image should be **800 pixels wide and 600 pixels tall**. The center of the projection plane for producing this image should lie a **single scene-space unit** in front of your view position along the viewing direction, i.e., $d = 1$. The field of view θ (angle formed between the top-right corner of your projection plane, the viewing position and the bottom-right corner of your projection plane) should be **40 degrees** (see Fig. 1). Be sure that each ray passes through the center of each pixel of the image.

Final note

Take into account the numerical errors. For example, the computed point of intersection may be off the surface, which may cause self-intersections. Try to avoid unnecessary arithmetic such as repetitive vector normalization.

Example output

Included with the source code starter package is a folder called `example-output` containing four example output images. Three of the images show examples of partial solutions. In particular, the image `no-phong-lighting.bmp` shows a simple solution that correctly determines visibility of objects in the scene but without lighting (see Fig. 2). The image `no-shadows.bmp` shows a solution using the Phong lighting model, but without calculating shadows (see Fig. 3). The image `self-intersection.bmp` shows a full solution with a common mistake arising from numerical problems calculating light visibility (see Fig. 4). Finally, the image `solution.bmp` shows a full solution (see Fig. 5).

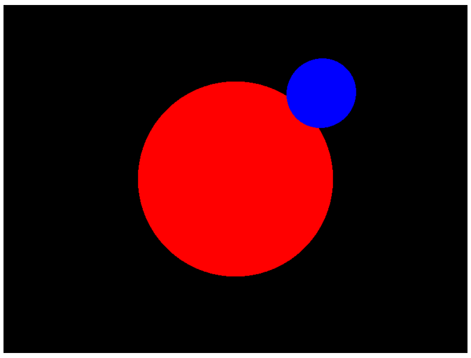
Figure 2: Simple solution with no lighting effects or shadows.

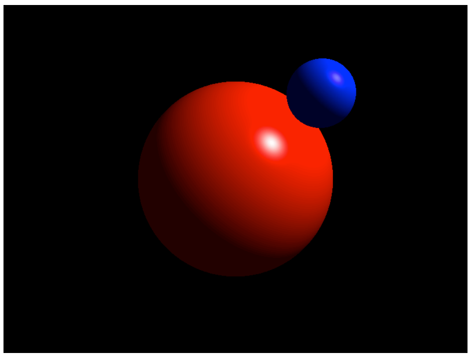
Figure 3: Solution with using the Phong lighting model, but without shadows.

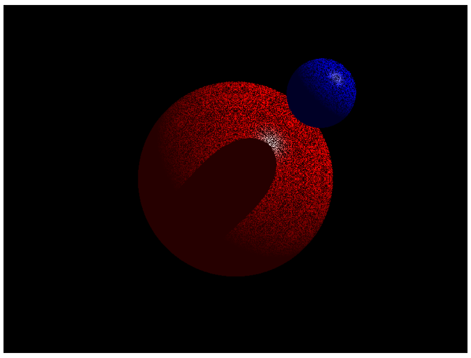
Figure 4: A solution with a common pitfall.

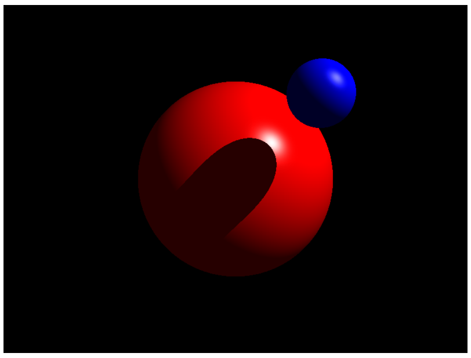
Figure 5: A final solution: visibility, Phong lighting and shadows.