Computer Graphics
Exercise 2 - Simple Raytracer

Handout date: 4.10.2013

Submission deadline: 25.10.2013, **14:00h**

For the first time this year, you are free to choose between C/C++ and JavaScript for coding the exercises. Specific instructions are written in **green for C/C++ users** and in **blue for JavaScript users**.

You must work on this exercise **individually**. You are required to 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-yourname`.

The .zip file should contain:

- 800x600 final raytraced images named `cg-ex2-yourname-module_id.bmp` or `.png` of the scene described below for each module, where `module_id` should be replaced with the ID of the module (e.g. A1 or B2)
- A folder 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. Even if you need to download GLUT binaries to run your code on Windows, you don't need to include them in your submission.

In any case, do not include the example images that we provided with the source code template.

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

- **Compile and run (without any external dependencies besides GLUT) on student lab PCs at CAB H 56 or 57 running Windows or Linux (specify which OS you used in your `README`)**
- **Run on student lab PCs at CAB H 56 or 57 running Windows or Linux with Firefox**
- Produce the same 800x600 images 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.

This exercise will be graded **during the exercise session on 25.10.2012**. You are required to attend, and those absent will receive 0 points. Grading will be conducted on the lab machines in room **CAB H 56** and/or **CAB H 57**. You are also required to **email your solution beforehand** as described above.

Each feature is encapsulated in a *module* below. All B modules are to be built upon the raytracer and scene of module A1. This means that unless otherwise noted the camera, light and object descriptions should remain the same. Also the basic features from module A1 should remain: Phong shading, shadows, etc.

We encourage you to implement all the modules in a common framework, but the solutions for each module should be independent. This means your solution for the Specular Reflection module should not contain Anti-aliasing. Structure your code so that each feature can be switched on or off.

Use a command-line argument. An example execution to run module B2: ./raytracer B2
Use the GET parameters. An example execution to run module B2: raytracer.html?B2

What's given to you already

We provide a template for both C/C++ and JavaScript. They are equivalent, so pick the one you prefer. You can find specific instructions in the template code. If you don't know which one to choose, we recommend **JavaScript**, for mainly two reasons:

- no compilation required, just open the web page in Firefox or Chrome
- debugging is much easier

A1: Basic features

This module encapsulates some basic features of a ray tracer.

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.

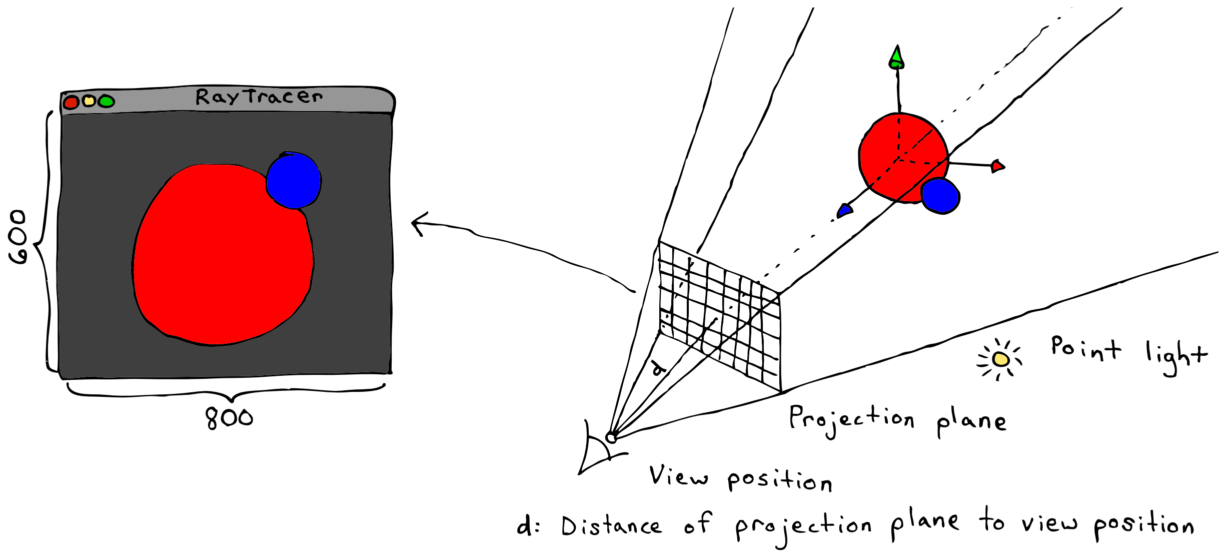

Figure 1: A demonstrative illustration of the scene and viewing arrangement.

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.

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., $d = 1$ (see Fig. 1). The field of view θ (angle formed between the midpoint of the top edge of your projection plane, the viewing position and the midpoint of the bottom edge) should be **40 degrees**. Be sure that each ray passes through the center of each pixel of the image.

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.xxx` shows a simple solution that correctly determines visibility of objects in the scene but without lighting (see Fig. 2). The image `no-shadows.xxx` shows a solution using the Phong lighting model, but without calculating shadows (see Fig. 3). The image `self-intersection.xxx` shows a full solution with a common mistake arising from numerical problems calculating light visibility (see Fig. 4). Finally, the image `solution.xxx` shows a full solution (see Fig. 5).

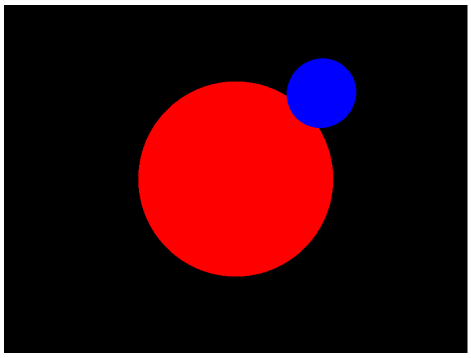

Figure 2: Simple solution with no lighting effects or shadows.

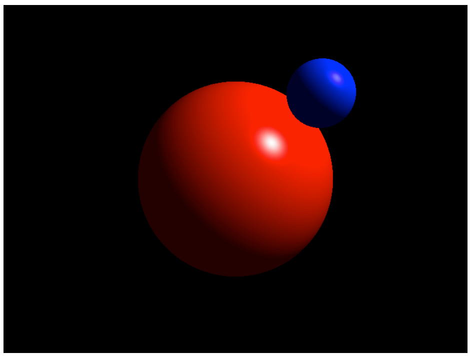

Figure 3: Solution with using the Phong lighting model, but without shadows.

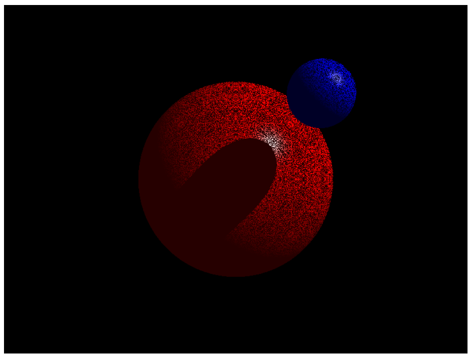

Figure 4: A solution with a common pitfall.

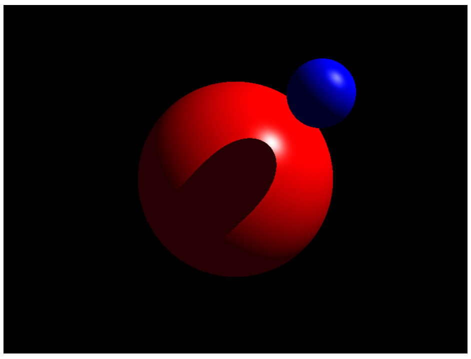

Figure 5: A final solution: visibility, Phong lighting and shadows.

B1: Specular reflection and specular refraction (5 points)

Use the same scene as described in module A1 but now **both spheres** should be *reflective*. The **small, blue sphere** should be *refractive*. Use index of refraction **1.5**. To shade a ray hitting an object at point **x** along vector w use the following recursive formula:

$$L(\mathbf{x}, w) = 0.5E + 0.5 * (L_s + L_t)$$

where E is represents the color computed using the same Phong model in module A1, L_s is the color returned from shooting another ray along the reflected direction and L_t is the color returned from shooting yet another ray along the refracted direction.

Do *not* hard code a single extra ray bounce. You must implement *recursive raytracing tree* as rays may bounce and split several times.

B2: Anti-aliasing (5 points)

Use the same scene as described in module A1, but use *super-sampling* to achieve anti-aliasing. Use **16** samples per pixel.

B3: Quadrics (5 points)

Use the same scene as described in module A1, but replace the two spheres with the following **quadrics**.

Large, red elliptic cylinder

Axis line:	$x = 0, z = 0$
Radii (x,z):	(2,1)
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 ellipsoid

Center:	(1.25,1.25,3)
Radii (x,y,z):	(0.25,0.75,0.5)
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

B4: Boolean operations (5 points)

Use the same scene as described in module A1, but replace the two spheres with the following shapes constructed using boolean operations on implicit surface representations.

Large, red-yellow, open hemi-sphere

Center:	(0,0,0)
Radius:	2
Intersection half-space:	$x >= z$
Exterior ambient material color:	rgb(0.75,0,0)
Exterior diffuse material color:	rgb(1,0,0)
Exterior specular material color:	rgb(1,1,1)
Exterior specular exponent:	32.0
Interior ambient material color:	rgb(0.75,0.75,0)
Interior diffuse material color:	rgb(1,1,0)
Interior specular material color:	rgb(1,1,1)
Interior specular exponent:	32.0

Small, blue sphere-sphere intersection

Center of sphere A:	(1.25,1.25,3)
Radius of sphere A:	0.5
Center of sphere B:	(0.25,1.25,3)
Radius of sphere B:	1
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

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.