Computer Graphics
Exercise 3 - Advanced Raytracing

Handout date: 25.10.2013

Submission deadline: 08.11.2013, 14:00h

You must work on this exercise **individually**. You are free to choose between C/C++ and JavaScript to code the exercise. Specific instructions are written in **green for C/C++ users** and in **blue for JavaScript users**. 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-ex3-firstname_familyname`

The .zip file should contain:

- An 800x600 final raytraced image named `cg-ex3-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. C2 or D1)
- 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 do not need to include them in your submission.

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 images produced by your raytracer, and the conformance of your raytracer to the expected behavior described below for each module.

This exercise will be graded **during the exercise session on 08.11.2013**. You are required to attend, and those absent will receive 0 points. Grading will be conducted on the lab machines at **CAB H 57**. You are also required to email your solution beforehand as described above.

Goal

The goal of this exercise is to extend your raytracer from Exercise 2, part A with more advanced features.

Each feature is encapsulated in a *module* below. All modules are to be built upon the raytracer and scene from Exercise 2, **part A**. This means that unless otherwise noted the camera, light and object descriptions should remain the same. Also the basic features from Exercise 2 part A should remain: Phong shading, shadows, etc. Do not include features from Exercise 2 part B: anti-aliasing, etc.

While we encourage you to implement all the modules in a common framework, the solutions for each module should be independent. This means your solution for the area lighting module should not contain texture mapping. Structure your code so that each feature can be switched on or off. Your final `raytracer` program should take **a single command line argument** or **a GET parameter** which will be the ID of the module to be rendered.

An example execution to run module C2:

```
./raytracer C2
```

```
raytracer.html?C2
```

Grading

If you have partial solutions in your submission, describe them clearly: in which modules you have them, which part you have implemented for each, and what are working and aren't though you've tried.

Supplementary Code and Data

You're given the textures and mesh needed to complete this exercise as well as simple libraries for loading them. **They are all included in the JavaScript framework you used for the exercise 2.** For security reasons, most web browsers do not allow a web page to download resources from local storage, thus preventing you from loading texture images and meshes from your computer. To work around this, you could change your browser's security settings (**at your own risk albeit not recommended**), or, better, use your ETH web space to host your files and work remotely. **You may download a .zip file containing them from the course website for the C++ framework. See the `SUPPLEMENT` file for more detail about the code.**

C1: Stereoscopic rendering (5 points)

An interesting extension to your raytracer is to render stereoscopic output instead of monoscopic. Humans perceive the depth of an object from many contexts. One of the strongest is the stereoscopic cue. Your eyes see the world from two slight different viewpoints. This causes disparities between the positions of a scene point in two images, which is called *binocular disparity* or *binocular parallax.*

You will render a stereoscopic image pair of the same scene. Instead of keeping two image planes explicitly, place a common image plane at the distance of **8.5 scene-space unit** from the current viewing position. For each pixel, you shoot two rays, the viewing positions of which are respectively translated **half a unit distance** towards left and right with respect to the up direction. These two rays represent two eye points. After determining two intersection points (if there are any) and the color values at those points, you composite those two using the following equation:

$$\begin{pmatrix} r \\ g \\ b \end{pmatrix} = \begin{pmatrix} 0.3 & 0.59 & 0.11 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix} \begin{pmatrix} r_\mathrm{L} \\ g_\mathrm{L} \\ b_\mathrm{L} \end{pmatrix} + \begin{pmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} r_\mathrm{R} \\ g_\mathrm{R} \\ b_\mathrm{R} \end{pmatrix}.$$

Other camera parameters such as the field of view and the image resolution remain to be the same as in Exercise 2. Also note that the position of the projection plane does **not** affect culling.

The conceptual image plane placed in the scene is called a *convergence plane* or a *zero-parallax plane*, and the distance between the two viewpoints is called a *baseline* or *interaxial distance.* The resulting image is called a *half-color anaglyph* image, and can be seen using glasses with red tint in front of your left eye and cyan in front of your right eye. Several anaglyph glasses will be prepared for you on the next exercise session. Send an email to a TA to reserve yours before the session.

If you implemented texture mapping (see below), try to use texture mapped spheres for stereoscopic rendering. Otherwise use **rgb(1, 1, 0)** for the ambient and diffuse properties of the big sphere, and **rgb(0, 1, 1)** for those of the small sphere.

C2: Texture mapping and bump mapping (10 points)

Texture mapping

Textures are images wrapped over the geometry. They provide scene elements with an illusion of fine detail without increasing geometric complexity. In 2D texturing, texture coordinates $(u, v) \in [0, 1]^2$ parameterize the 2D image, or texture, domain. To wrap the texture onto the geometry, you must define a map between each point (x, y, z) on the surface and to some (u, v) on the texture. This map can be defined as a function, or interpolated from a lookup table.

You must implement a simple texture mapping for the two spheres from the scene in Exercise 2. Wrap the two spheres with satellite photos of the earth and the moon. For this you will need to implement:

- A mapping between the coordinates within a surface, which is in our case a sphere, and texture coordinates,
- An *alias-free* texture sampler to compute the color from the texture image given a texture coordinate (you may use *mipmapping* for this), and
- A modified shader to determine the final color of a scene point using the sampled texture color as *all* three material properties in Phong lighting model previously implemented.

Let the north poles of both spheres be in the direction of **(0, 1, 1)** from the center of sphere, and the prime meridians (horizontal mid-points in both textures) pass through the direction of **(-1, 1, -1)**.

Bump mapping

While texture mapping modifies material properties on the geometry, bump mapping modifies the normals on the geometry, giving an illusion of a bumpy surface. In this exercise we use a normal map, an RGB image encoding the normals, to compute the normal for a point in a surface. As for texture mapping, (u, v) coordinates are computed from a surface point, and the normal vector is sampled from the normal map. Each component of normal vectors are encoded in each color channel of the normal map as following:

$$(t, b, n)^T = (2r - 1, 2g - 1, 2b - 1)^T,$$

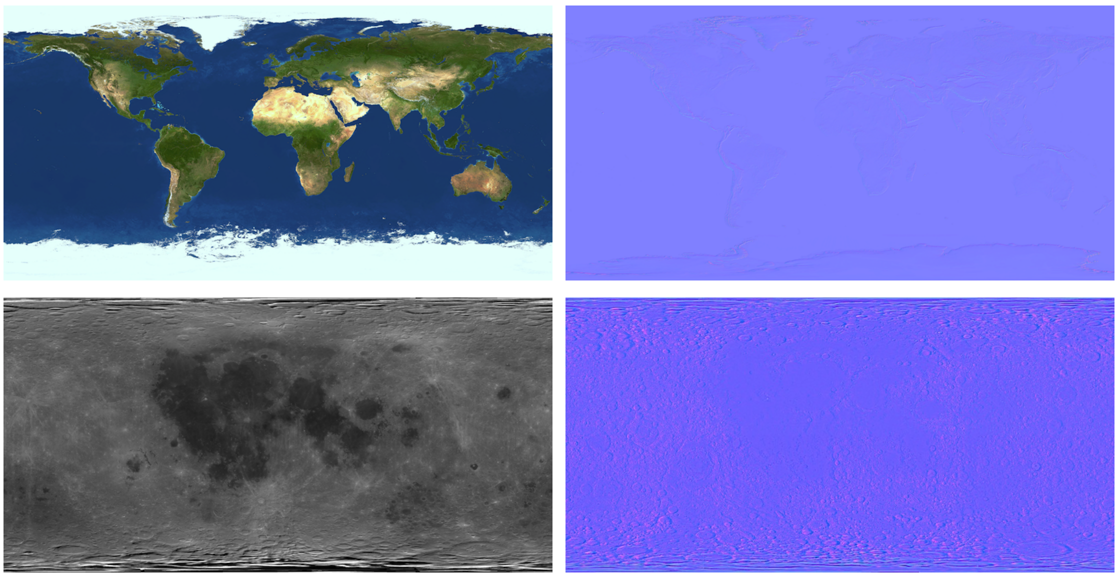

Figure 1: Textures (left column) and normal maps (right) for two spheres. Use the earth texture for the bigger sphere, and the moon texture for the smaller one (or the other way around if you want ;-)).

where t is along the tangential direction, b the cotangential direction, and n the normal direction in the *tangent surface*. As the obtained normal vector is defined on a tangent surface, you will need to first calculate the tangent space at the point, and convert the normal from the tangent space coordinates to the world coordinates. The resulting normal vector is used in lighting instead of the geometric normal at the point.

You are given the textures and normal maps[1] of the earth and the moon (Figure 1) in the `texture` folder of the supplementary data, together with a simple image reader in the `source` folder. Use them for this module.

C3: Triangle meshes (5 points)

Meshes are collections of polygons which share edges to form a larger, more complex scene element. Because of its simplicity, it is widely used in modeling objects. Here we only handle triangle meshes using a vertex list and a face index list representation: for each triangle you are given the indices to its three vertices, and for each vertex its 3D position is given.

Normal computation

When you load a mesh, you need to compute the normal of each vertex. You can uniquely define the normal of a triangle from its three vertices. Then the normal for a vertex is interpolated from its neighboring triangles. Use a weighted average based on the neighboring triangles' areas as the interpolation scheme. Be consistent with the orientation of vertices.

Intersection test

Add triangle as one of primitives your raytracer can render. This means you need to be able to test an intersection between a ray and a triangle.

[1]These images are taken from The Celestia Motherlode (http://www.celestiamotherlode.net). All credits go to the creators of the images.

Phong shading

Given an intersection point inside a triangle, you need to compute lighting to determine the color of the point. Use Phong shading to interpolate the color inside the triangle. In Phong shading model, the normal used for shading the point inside a triangle is computed by linearly interpolating the normals stored at the triangle corners. Use the same Phong lighting model from the last exercise to shade the point after you compute the normal of the point.[2]

A mesh in .OBJ format representing a sphere will be given in the `mesh` folder, and a simple .OBJ mesh loader in the `source` folder. Replace your two spheres with the given mesh, and render the scene with the remaining scene settings from Exercise 2. You may need to scale and translate them appropriately. Use the same material properties for the mesh version spheres. Try other meshes you may find on the web. For complex meshes, however, you may find it more practical to implement the Octree first (see below).

D1: Octree (10 points)

Use the same scene as described in Exercise 2, but replace the two spheres with **1000 small, blue spheres**. Let the spheres be enumerated by subscripts i, j, k with each an integers ranging from 0 to 9. Then we place sphere (i, j, k) centered at $(\mathbf{i-4.5}, \mathbf{j-4.5}, -\mathbf{k^3})$.

The spheres should having the following properties:

Small, blue spheres

Radius:	0.25
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

Use an Octree to accelerate the rendering of these spheres. The root node should be the bounding cube of the scene objects. Rays should no longer test for intersection with all objects in the scene rather they should test in a top down manner with the nodes of the Octree, only finally testing for intersections with spheres when reaching a leaf node.

In your `README` file, justify your choice of **termination criteria** for choosing the number of levels.

D2: Area lights (5 points)

Use the same scene as described in Exercise 2, but replace the single point light with a single area light. Use *Monte Carlo integration* to implement soft shadows. Use **50** samples on the area light, each treated as a point light in a weighted average during shadow and shading computation.

There area light illuminating your scene should have the following properties. Its color should be **rgb(1,1,1)**. Its ambient, diffuse and specular intensities should be respectively **0, 1 and 1**. The area of the light should be represented as a **disk** centered at **(10,10,10)** with radius **1** and oriented to face the origin.

The choice of how to implement Monte-Carlo integration to is left open (for example, how to sample the area light), but should be justified in the `README`.

[2]Please note that Phong *lighting* and Phong *shading* are different. Phong lighting is for *shading* (determining the color of a point), and Phong shading is for *interpolating* the color of a point within a polygon.