Introduction to Computer Graphics
Exercise 1 – Raycasting

Handout date: Thursday, 27.09.2012
Submission deadline: Wednesday, 03.10.2012, 14:00 h

Note

Undeclared copying of code or images (either from other students or from external sources) is strictly prohibited! Any violation of this rule will lead to expulsion from the class. Late submission are not accepted.

What to hand in

A .zip compressed file renamed to "Exercise*n*-Name1-Name2.zip" where *n* is the number of the current exercise sheet and the names are the full names of the submitting students. It should contain:

- A zipped folder containing Microsoft Visual Studio 2008/2010 file solution with all source files and project files. In order to avoid sending build files, close your Visual Studio solution and run the file "remove-compiled.bat" located in the top directory of the framework before you submit. Unzip and rebuild your submission to make sure it compiles without error.
- A "readme.pdf" file containing a description on your solution approach (1-2 sentences), rendered images and encountered problems (use the same numbers and titles).
- Submit your solutions to the classes moodle before the submission deadline. The enrollment key is: *icg12*

"Let there be Light!"

In this exercise you will extend a framework to build a minimalistic functional raytracer in order to generate your first raytraced image. You will generate a camera ray for each pixel, intersect the ray with geometric primitives to determine the pixels color. Feel free to play around with the scene description file and send us interesting renderings.

1.1 Get to know the Framework

Download the RayTracer framework from the course home page. The framework should compile without errors or warnings using Microsoft Visual Studio 2008/2010. Try to run it. If you experience a problem with the glut library, copy the file *glut32.dll* from the *lib* folder to your *Framework\Debug* or *Framework\Release* directory. Right click on the black image and choose "Render Image". After rendering you should see a centered black sphere on a slightly blue background. Now try to understand the logic of the framework. The RayTracer consists of the following main components:

RayTracer.cpp: This file contains the `main()` routine and the event functions for the glut window. In the main routine, the scene is created (see **parser**) and passed to the renderer.

Scene.cpp: Main scene access and datastructure class.

Renderer.cpp: Class controlling the main rendering loop.

sceneelements: All the classes modeling elements needed for a simple scene description, i.e. a camera, lights and geometric elements like a sphere or a plane. Most of these classes are derived from a general interface, so that it is easy to replace or extend the existing implementations. The classes *Mesh*, *MeshTriangle* and *MeshVertex* provide a more sophisticated representation of a collection of connected triangles than *Triangle*. For the sake of simplicity, we do not use them in this exercise.

rendererelements: Classes that implement the rendering pipeline like a sampler, which determines the order of pixels to be rendered. And shaders, which compute a final pixel color from information about the intersection point.

utils: A bunch of utility classes to provide convenient data structures for things like Euclidean vectors or rays.

exporter: A simple .bmp image exporter. Feel free to write exporters for other formats by yourself or use the operating system's *print screen*.

parser:

1. A simple XML reader.
2. The `SceneParser` which constructs a scene out of a scene description file. The RayTracer loads the scene description file *SceneDescription.xml* per default. A different file name can be specified at startup as a command line parameter. In Visual Studio, go to `Project` → `RayTracer Properties` and change the Command Arguments under `Configuration Properties` → `Debugging`. Look at the example scene description files in the folder *sceneDescriptions* provided with the framework and read scenedescription_tutorial.html to learn the scene description syntax.
3. The `ConfigParser` which parses the raytracer's config file *Config.xml*. The config file sets the parameters for the renderer.

trianglemeshreader: Contains an importer reading *.obj* files to a *Mesh*. We do not use it in this exercise.

1.2 Constant Shading (2 Points)

The framework provides a `SimpleCamera` and ray-sphere intersection `Sphere::intersect`. In order to render a first scene containing spheres, we only need a shader which computes the final color at an intersection point. A `ConstantShader` is the simplest shader you can think of. It uses the color of the first object hit by camera ray, disregarding normals, material or lighting information. Complete the method `ConstantShader::shade` to return the color of the object at the given intersection point.
Render the scene *Scene_0.xml* provided with the framework. It should look the same as our rendering in figure 2. Provide your rendering in your *readme.pdf*.

1.3 Camera Rays (15 Points)

The method `SimpleCamera::generateRay` currently implements fixed orthographic camera rays. We now want to replace it with a perspective camera. Note that all orthographic camera rays have the same directions but different starting points, while the perspective camera rays will all have the same starting point, but different directions.

A `Sample` represents a square pixel on the camera image and stores its position on the image and its color. For every such sample, the camera has to generate a corresponding ray (primary ray), which can then be cast into the scene. Rewrite `SimpleCamera::generateRay` to follow the camera model of our `SimpleCamera` (see Figure 1) and generate a ray given an image sample.

The camera's parameters needed for this task are: Its position in world coordinates (`m_pos`), half the angle (in degrees) of the aperture in up direction (`m_openingAngle`), the image resolution (`m_resolutionX, m_resolutionY`) and the rotation (no translation) matrix (`m_camToWorld`). To transform a vector v from camera space to world coordinates, multiply v by the rotation matrix. (The caption of Figure 1 provides further information on the camera parameters). Use `sample.getPosX() and getPosY()` for the position of the sample on the image, where $(0,0)$ denotes the center of the bottom left pixel and $($`m_resolutionX-1`,`m_resolutionY-1`$)$ denotes the center of the pixel on the top right.

Again render the scene *Scene_1.xml*, compare it to figure 2, and provide your rendering in your *readme.pdf*.

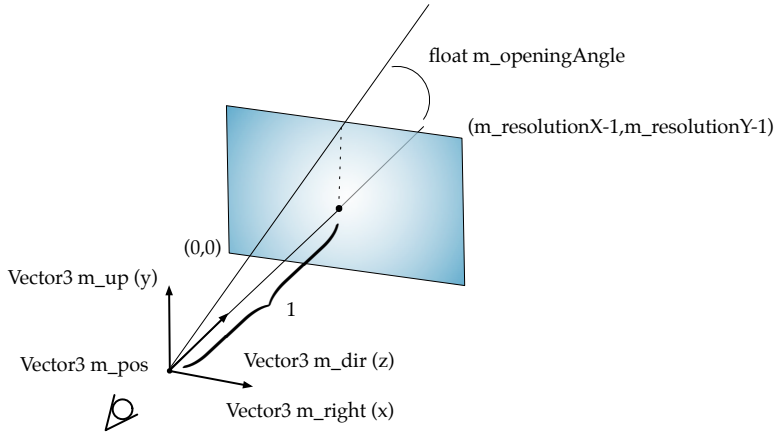

Figure 1: `SimpleCamera` camera model: The camera coordinate system is given by its three orthogonal axes `m_right` (x), `m_up` (y) and `m_dir` (z) in world coordinates. The origin of the camera system is at `m_pos` in world coordinates. The image plane is orthogonal to `m_dir` and intersects `m_dir` at $(0,0,1)$ in camera coordinates. This intersection point defines the center of the image. The height and width of the image in pixels are given by `m_resolutionX` and `m_resolutionY`. `m_openingAngle` defines half the angle (in degrees) of the aperture in up direction (note that it is the angle between the center of the image and its upper border and NOT between the center of the image and the middle of the top most pixelrow!).

1.4 Ray-Surface Intersections

Now change the settings or command-line parameters such that it renders the file *Scene_2.xml* provided with the framework.

1.4.1 Plane (5 Points)

Implement the `intersect` routine for the `Plane`: If an intersection occured, the parameter t of the ray at the intersection is between the ray's `min_t` and `max_t` and t is smaller than the one of the currently nearest intersection (`iData->t`), we call the intersection valid. In this case, fill the instance of `IntersectionData` that was passed as a parameter with information about the intersection's position, the object color (`m_color`) and the parameter t of the ray, such that $position = ray.getPointOnRay(t)$ and return true. You can have a look at `Sphere::intersect` to understand how to fill the intersection data. If no valid intersection occurs return false instead. Render the scene *Scene_2.xml*, compare it to figure 3, and provide your rendering in your *readme.pdf*.

1.4.2 Triangle (15 Points)

Implement the `intersect` routine for the `Triangle` similar to the one of the Plane. Follow one of the chapters in the folder *RayTriangleIntersection* we provide in the moodle. Comment which method you used in the source code and in the *readme.html*.
Important: do not use the three normals of the triangle for the intersection computation. They do not coincide with the triangle normals and are used for shading in the next exercise. Use barycentric coordinates instead, which you will need in later exercises anyway.
Again render the scene *Scene_2.xml*, compare it to figure 3, and provide your rendering in your *readme.pdf*.

1.4.3 Quadrics* (For the passionate)

A surface of the form

$$\frac{(x - center_x)^2}{Real(a^2)} + \frac{(y - center_y)^2}{Real(b^2)} + \frac{(z - center_z)^2}{Real(c^2)} = 1$$

is called a Quadric. a, b and c are complex numbers, however we only need the real values of their squares and can omit the imaginary ones. Depending on these values the equation represents an ellipsoid, a cone, a hyperboloid, etc. (See http://en.wikipedia.org/wiki/Quadric.) Implement the `intersect` routine for the `Quadric`. You can compare your rendering to figure 3.

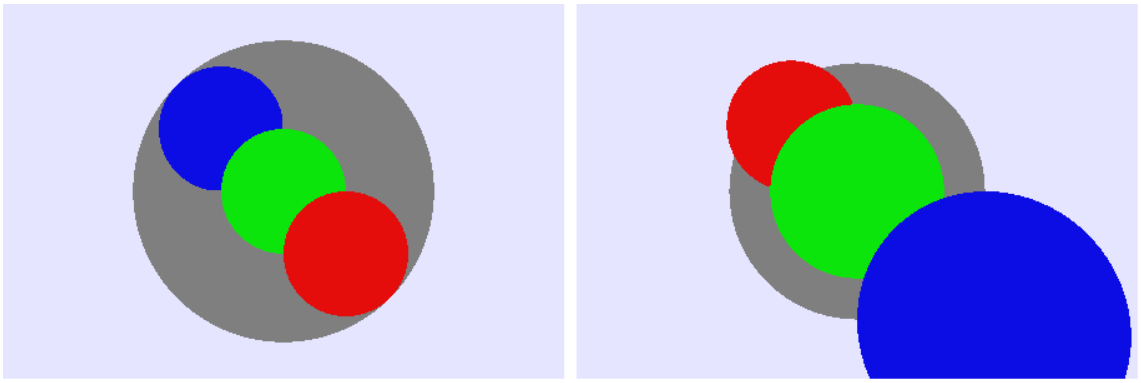

Figure 2: Constant Shading. *Left:* Ex 1.2, *right:* Ex 1.3.

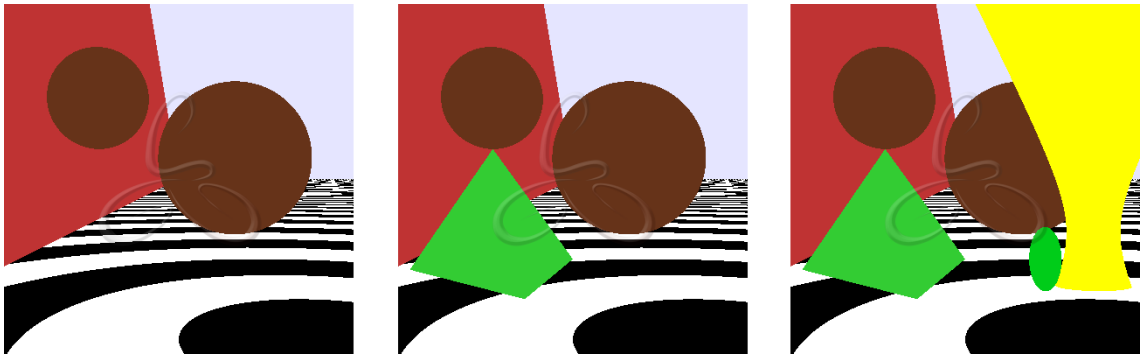

Figure 3: Ray-surface intersections of *Scene_2.xml*. *Left:* Ex 1.4.1, *middle:* Ex 1.4.2, *right:* Ex 1.4.3. Watermarked.