Assignment 1: Basic Ray Tracer

Due date: March 13th, 2014, 12:00

In the assignments of this class you will step-by-step develop a 3D renderer to generate realistic images. The goal of this first assignment is to build a basic ray tracer with the features described below. We are providing a Java code skeleton on GitHub that you are encouraged to start with. Simply import the project into a Eclipse workspace, and you should be ready to go. Make sure to study the JavaDoc documentation before you start. The code skeleton implements an extensible architecture for a ray tracer and it comes with support for multi-threaded rendering. It also includes some basic rendering functionality and examples that we will build on in this first assignment.

If you prefer to build everything from scratch, you are free to do so using the programming language of your choice, such as C++ or C#. Be aware, however, that this will mean a significant amount of extra work, since you will have to implement code infrastructure and some functionality already present in the Java skeleton.

You must turn in your work through Ilias by the deadline. You will also demonstrate your code to the teaching assistant in person during the exercise session on March 13th, 2014.

The assignment includes suggestions for a "hacker's bonus". Here you can earn extra points that will count towards your grade for the assignments, and it allows you to achieve a grade above 6. This grade will be averaged with the exam grade in the end to get the final grade for the course.

Required Features

Camera and image specification (20 points)

The input parameters to specify the camera should be its position, up vector, and look-at point in world space, and the field of view and the aspect ratio of the viewing volume. For the image you should specify its resolution.

Note that the Java base code already provides a basic camera implementation. It also provides functionality to write rendered images to a file.

Intersecting geometry (30 points)

Your ray tracer should handle planes, spheres, and triangle meshes, and it should support instancing. In addition, your ray tracer should support computational solid geometry (CSG)

objects and set operations (add, subtract, intersect). You should support CSG spheres, cones, and cylinders. Your CSG objects should also support instancing.

Note that in the Java base code functionality for reading triangle meshes from obj files is already provided. The code already supports ray tracing planes. Also, a framework supporting CSG operations is already available. You can easily add instancing of CSG objects by extending the `CSGSolid` class.

Shading (20 points)

Your ray tracer should support shading using multiple point light sources. You should implement a shading model including diffuse and glossy components using the Blinn model. You should be able to render shadows by shooting shadow rays towards light sources.

Note that the Java base code already provides code to accumulate the contribution of point light sources, but without shadows. It also provides a shading model for diffuse surfaces, but not the glossy component of the Blinn model.

Reflection and Refraction (30 points)

Implement a recursive ray tracer that can render materials exhibiting reflection and refraction. Use Snell's law to compute the refracted direction, and Schlick's approximation to compute the Fresnel terms. Demonstrate your result by constructing a visually interesting scene using the CSG functionality.

Hacker's bonus

Develop procedural shaders to render interesting materials. Add support for textures and bump maps. Stun us with a nicely designed scene. We will assign points on a case-by-case basis.

Deliverables

You will demonstrate your code during the exercise session of the March 13th, 2014. You should prepare one or several test scenes that show all the features of your code (different types of geometry, instancing, shading models). Save example images beforehand, so that you can show them to us during the exercise session.

Advice

Start programming early. Here is a suggestion for a time plan:

1. week: camera, intersecting geometry (spheres, triangle meshes, CSG objects)
2. week: instancing (including CSG), shading (Blinn model and shadows)
3. week: reflection and refraction, render example scenes, hacker's bonus

You will likely spend more time debugging than coding. Here are some ideas for debugging strategies:

- Build the simplest possible scene that produces your bug, then debug using this scene.
- Render an image exhibiting your bug, identify a pixel that shows the bug, then render only that pixel. Step through the code and examine the values of all relevant variables to check if they are correct.
- Use "debug integrators" as in the Java base code to visualize intermediate values during rendering, like showing normals encoded as RGB values. Often, setting up lightsources to illuminate everything in a scene, avoiding black areas because of shadows, etc. is not 100% trivial. If you are not sure why you are not seeing anything, you can for example simply mark each pixel where something is hit as white to check if the geometry is actually there.
- Render at low resolutions for testing so you do not waste time waiting for results. If it takes more than ten seconds to render your image during debugging, you will spend most of your time waiting.
- Use a single thread for debugging to avoid confusion because of multi-threading.

Test Scenes and Images

The Java base code also includes test scenes and corresponding images that you may use for verifying your results. Note that the test scenes cannot actually be rendered with the skeleton code, since they refer to additional classes that you are supposed to develop in the assignment. The test scenes are just for reference so you have exact scene parameters for comparison.