2016 - Final Project TI1806 Computer Graphics

Ray Tracing with OpenGL Debugging

Examination: Exact defense date at the end of the quarter will be announced via blackboard (we want to avoid collisions with other courses)

Evaluation: 4 min. sharp (!) for a presentation (Powerpoint & (optional) demo) + 3 min. of questions Presentation and questions constitute 5 % of your grade, the project 35%.

To hand in: Everything should be uploaded via Blackboard one day before the presentations start. Your submission should contain: 1) your zipped source code (only cpp and h files), 2) potentially models (in a model.zip file), 3) a report in form of a bullet list (!) of the implemented features (illustrated with images), including a breakdown of the contribution of each individual member, 4) your presentation slides in PDF format, 5) one additional JPG (Groupnumber.jpg) to participate in an image competition. **The presentation slides and the report (bullet list) should start with all member names and their student number and stop with an overview of individual contributions.**

Project Description

The final project of the course Computer Graphics will revolve around ray tracing. The goal is to develop a „full“ raytracer that generates images from a virtual scene containing lights, objects and a camera.
To reach this goal, several steps have to be taken, which are outlined below. **Read this whole document carefully before getting started!**

The minimal requirements are:

- Ray intersection with planes, and triangles
- Computation of the shading at the first impact point
- Recursive raytracing to simulate specular materials (using reflection)
- An option to move the light sources in the scene
- An interactive display in OpenGL of the 3D scene and a debug ray tracing; a ray from a chosen pixel into the OpenGL scene, showing the interaction with the surfaces
- A (simple) spatial acceleration structure (see below for details)
- Own-built scene(s) and screen shots for competition (up to 1.5 points can be won for the project)

Of course, you can always extend your project by adding additional components. For example:

- A possibility to loop over the rays via the keyboard, triggering a ray highlighting (e.g., change of color) and a command line output of the selected rays properties.
- Interpolated normals and properties for smooth objects
- A more complex scene hierarchy
- A selection of triangles for which to modify attributes on the fly
- Ray intersection with spheres
- Refraction and transparent objects
- Design your own 3D scene in Blender
- Create interesting test cases
- …

Getting started

The starting point of your project is the attached code sample, which opens a 3D model including its normals, and displays it in OpenGL (the light is following the camera).

Additionally, upon pressing on "space", a single ray is produced underneath the mouse position. This ray is stored and then displayed. When you move the camera, the ray stays visible. Currently, this ray does not interact with the geometry – which is for you to implement.

When pressing "r", an image is created in the current working directory called 'result.ppm'.

Talking of implementation, you really only need to take a look at four files:

> **raytracing.h/cpp**: this is the file you mainly work in
> **mesh.h (and Vertex.h)**: the mesh structure, against which you will trace rays
> (**Vec3D.h**: vectors to describe vertex positions --- but you already know this one)

There are many comments in the code, please read through them carefully.

The compilation is the same as for your last exercise. In fact, you can take the last solution file, open it in Visual Studio, remove all files from the solution and insert all new files instead.

ATTENTION: YOU NEED TO CHANGE THE FILE PATH in the "init" function of your raytracing.cpp to point to the right directory containing the mesh model.

A simple ray tracer

The first step is to code an intersection function with a triangle. To debug your method, make use of the possibility to render points and lines with the debug drawing function. E.g., draw only the line segment from the camera to the first intersection point. Such visualization is also very useful when debugging reflection, refraction, or even lighting. Collect illustrative screenshots for your report/presentation to show the debug illustrations.

Spatial acceleration structure

To find the triangles that intersect with a given ray in a naive implementation, all triangles need to be tested. This process is very inefficient (read: slow) as you will find out. Therefore, your task is to implement a *simple spatial acceleration structure using axis-aligned bounding boxes*. **If you opt for something more complicated, still implement this solution as a reference.**

This structure needs to be computed once for your static scene and is built as follows: the scene's triangles are sorted into smaller axis-aligned boxes (a few hundred triangles per box). Then, instead of testing all the triangles for a given ray, the bounding boxes are tested first. If a box is missed by the ray, its contained triangles can be ignored. If the ray does intersect the box, its triangles need to be tested against the ray. This needs to be done for all boxes.

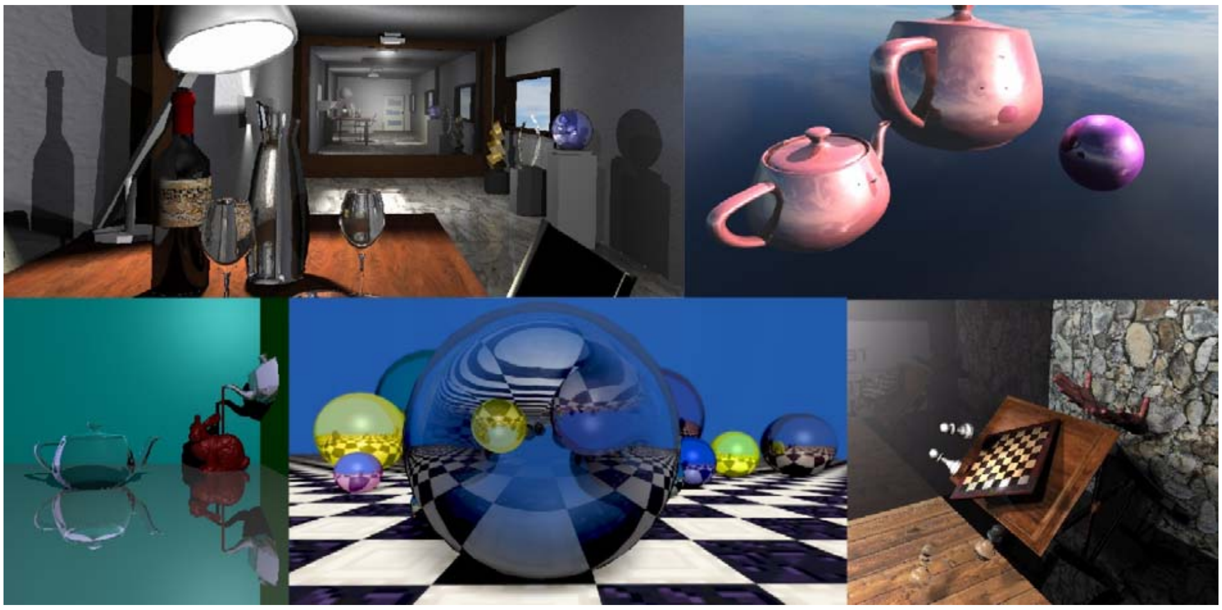
For simplicity, the boxes' sides can be considered being made of triangles (so you can reuse your ray-triangle intersection test). Hence, after you have tested the ray tracer without any structure, implement the box intersection as a test against 12 triangles of a box (don't take a shortcut here!). Once it works, you can think about accelerations, e.g., a lookup table to determine the box sides that need checking depending on the ray direction. Or a special quad intersection test.

Step-by-step instructions to make this process easier:

1. Before you implement the actual structure, start with a small object and a single bounding box for the whole object. Make one rendering where you zoom out (i.e., the object appears small on the image) and one where you are zoomed in (i.e., the object fills a good part of the image) and check your findings.
2. So far, we have just a single box. Now we start generating many boxes, each containing a few hundred triangles (variable parameter). To this extent, split the single bounding box you have so far into two, using a splitting plane along one axis (e.g., longest?) at a good position (for example, the middle or maybe better the position where the average of all points in the bounding box lies). Color the triangles in each side and those intersecting the plane differently and show the result. Then associate the intersecting triangles to both sets on the two sides of the plane and compute a bounding box for each. Attention: both boxes need to completely encompass the triangles contained in them, which usually means that the resulting boxes will overlap each other at the splitting position.
3. Now that we know how to split a single box, we can start splitting them recursively, thus creating many bounding boxes. Stop splitting a box when the number of triangles reaches a given minimum (as mentioned above, a variable parameter, possibly in the hundreds, but try and vary it to see how the performance differs). Attention: you do **not** need to implement a *hierarchical* structure. The structure is flat for now, i.e., you have a list of bounding boxes.
4. When you shoot a test ray into the scene, display all intersected bounding boxes.
5. Finally, show all bounding boxes (possibly with more than one color) and time the rendering speed with different values for the minimum number of triangles. You can add your findings as a table to the report.

EXTENSIONS:

6. Think about a smarter redistribution of the intersected triangles (half to one side, the other to the other side).
7. Experiment with a hierarchical representation by storing all bounding boxes during the splitting process and testing them recursively.



Organization

Your student ID has been associated to a group ID and you can subscribe yourself to corresponding group on Blackboard. Please exchange messages in the first week to make sure that all members are participating. In case that you encounter problems with your group members (e.g., no reaction by the end of the week), please send an email to: e.eisemann@tudelft.nl on **next Monday morning** with the **subject "[TI1806] Group Problem – Group Number XX"**. We will then find a solution for you and will modify groups where many members are missing.

In the next week, you will receive the name of a TA to contact in case of urgent questions. Nonetheless, as you are numerous and have many members in a group (and colleagues), you should first try to solve the issue internally. Please notice that the TAs will not be allowed to write code for you! ☺

Good luck!