Ray Tracing with OpenGL Debugging

The Final Project for TI1806

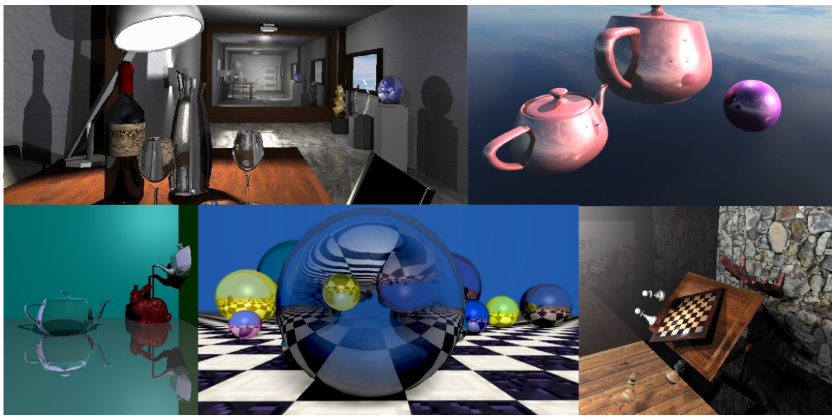

Figure 1: Renders from previous years

Introduction

The final project of the Computer Graphics course revolves around ray tracing. The goal is to develop a *fully capable* ray tracer with your group, 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 in this document. Please read the entire document carefully before getting started. But first, some important facts:

- There are weekly mandatory(!) meetings with your TA. Every week you will show the TA what your group has produced so far (in the form of a demo), and you will inform us of what your (personal) contributions are to the group work.
- You will get access to a GitLab repository. In the README of this repository you should create an overview of the individual contributions of each group member for each week.
- Examination: you will give a 4 minute sharp presentation, followed by 3 minutes of questions. All presentations will take place on Wednesday the 4th and Thursday the 5th of July 2018. More details will follow at a later date, keep an eye on Brightspace.
- Grade: the presentation constitutes 5% of your grade, and the project 35%.
- Hand-in: your Brightspace submission must contain the following:
  1. Your zipped source code folder.
  2. Potentially custom models (in a model.zip file).
  3. A report in the form of a bullet-list (this is important!) listing all implemented features (with illustrations), including a breakdown of the individual contributions of each member to each feature.
  4. Your presentation slides in .pdf format.
  5. One additional .jpeg file, (groupnumber.jpeg) to participate in an image competition.
  6. A list of sources you used.
  7. **Both your slides and report should start with all team members' names and student numbers, and end with an overview of the individual contributions.**

This project is a chance to make something visually stunning, which can earn you extra points. You can see some examples from previous years' student projects in Figure 1. Above all, have fun!

Project Requirements

Minimal requirements

The ray tracer you will be building is quite complicated. There are a number of minimal requirements your ray tracer must at least fulfill. These are:

- Ray intersection with planes, triangles and bounding boxes.
- Computation of shading at the ray's first impact point. (Phong model, consisting of diffuse **and** specular contributions)
- Recursive ray tracing for reflections, to simulate specular materials.
- Hard shadows from a point light, and soft shadows from a spherical light centered at the point light. (You can represent a spherical light by many randomly chosen point lights on this sphere)
- An interactive display in OpenGL of the 3D-scene, as well as a *debug ray tracing*; a ray from a chosen pixel should be shown via OpenGL, illustrating its interactions with the surface.
- A *simple* spatial acceleration structure. Details are discussed below.
- A custom-built scene (export as a Wavefront OBJ and you can load it directly), and screenshots for the image competition. **Up to 1.5 points can be won for the project!**

Optional requirements

Additionally, you can extend your project by adding additional components, which will gain you additional points. For example:

- Extended debugging: show the n^{th} reflection of a ray via the keyboard, trigger a ray highlighting (so, color based on the material) and a command line output of the selected ray's properties.
- Interpolated normals for smooth shading.
- A *less simple* spatial acceleration structure. Details are discussed below.
- A more complex scene hierarchy.
- A selection of triangles whose attributes you modify on the fly.
- Refraction and transparency.
- *Glossy* reflections.
- Soft shadows for other types of light sources.
- Sphere support. (you can draw spheres in OpenGL with `glutSolidSphere()`, the ray traced version should use the proper equation)
- An illustration of interesting test cases.
- A numerical evaluation of the performance of your raytracer.
- Anything else you can think of!

Project Details

Getting Started

Before starting on the project, please read the lab manual carefully. It will explain how to build and run the project code. If you successfully completed the previous lab exercises, building the code should pose no difficulties to you, however.

The base project, which is your starting point, loads a 3D-model including its normals, and displays it in OpenGL. The default light follows the camera. There are many comments throughout the code, to help and guide you while building the ray tracer. Read these carefully.

Additionally, when you press `space`, a single ray is produced at the mouse position. This ray is stored and displayed. When you move the camera, the ray stays visible. Currently, this ray does not interact with any geometry - you will have to implement this.

Finally, when you press `R`, an image is created in your project directory called `result.bmp`, as the result of your ray tracing.

Building a Simple Ray Tracer (Individual work in week 1)

The first step in your project is to code an intersection function between a ray and a triangle. To debug your method, make use of the possibility to render points and lines with the debug drawing functions. In other words: draw only the line segment from the camera to the first intersection point. Such a visualization is also useful while debugging reflections, refractions and even shadows.

You can also collect illustrative screenshots for your report and presentation to show off your debugging.

Building the Spatial Acceleration Structure

In a naive implementation, finding triangles that intersect a ray implies testing that ray against *all* triangles. This process is extremely efficient, as you are bound to find out. Therefore, part of your project consists of building a simple spatial acceleration structure, through use of *axis-aligned bounding boxes*. **If you opt for something more complicated, please still implement this solution as a reference.**

This structure needs to be computed only once for your static scene, and is likely built as follows: you will sort the scene's triangles into small axis-aligned boxes (a few hundred triangles per box, generally). Now, instead of testing all triangles against a given ray, you can test the bounding box first. If a box is not hit by the ray, all the triangles within can be safely ignored. If it is hit, all the triangles within will need to be tested.

For testing rays against boxes, you can reuse your triangle intersection code by building the boxes with 12 triangles. This is just a start of course, as there are many improvements you can make. For example, you can build a quad intersection test to speed up the process.

Here are step-by-step instructions, which may make the process easier for you:

1. Before implementing the structure, start with a small object, and a single bounding box surrounding the entire object. If you now render the object twice, once zoomed in on the object, and once zoomed out so the object takes up a part of your screen, you should notice the differences.
2. Now, start generating many boxes (containing hundreds of triangles, you can make this a variable). To do this, split the single bounding box you have in two, by splitting it along one (the longest) axis at a good position. For example: split along the middle, or split along the average of points in the box. Color the triangles in each side and those intersecting this plane differently to show the result. Then associate triangles to the two sub-boxes and compute their bounding boxes too. Note: both boxes will overlap each

other at the splitting position, as they must both contain the triangles intersecting the plane.

3. Now that you can split a box, do so recursively, thus creating a large collection of bounding boxes. Stop splitting a box when the number of triangles inside reaches a given minimum (as mentioned above, a variable), and try to vary this to see the impact on performance. **Attention: you do not need to implement a hierarchical structure. It is flat for now, so just a list of small bounding boxes.**
4. When you shoot a test ray into your scene, display all intersected boxes.
5. Finally, show all bounding boxes and time your render speed with different values for the minimum number of triangles. You can add your findings as a table to your report.

Here are extensions you can make to this structure:

1. Look at smarter redistributions of triangles in boxes. You want half to be in one side, and half to be in the other.
2. Experiment with a hierarchical (tree) structure by storing all bounding boxes (even those that are split) and then testing against them recursively.

Distributing Work

During the project, you are free to distribute tasks among team members as you see fit, but note that **everyone can receive an individual grade based on their contribution.** An example distribution of tasks is provided below. We advise you to work on tasks in subgroups of 2-3 members.

Hence, in the below example, we have 3 groups: a group of 3 students (called *B*) and two groups of 2 students (called *A1* and *A2*).

1. All groups: start by discussing code, select a plane/triangle intersection implementation.
2. A1: work on bounding box intersection.
3. A2: work on hard shadows.
4. A2: work on soft shadows for sphere lights.
5. B: Work on acceleration structure. (this will take a while)
6. A1: work on diffuse contributions.
7. A2: Work on specular contributions.
8. A1: work on reflection vector and recursive ray tracing (talk to A2 at this point to fit this together with their shadow implementation).
9. A2: work on blender scene.
10. All groups: integrate everything. Reserve at least one full day for this! The lead here should probably be taken by A1, as they implemented recursive ray tracing.

Group Organization

Your student number has been associated to a group automatically, and your group will be assigned a specific TA. Please exchange messages to make sure all your group members are participating. There will **always** be cases where group members might drop out. We take this into account in the final evaluation.

If you encounter problems with your group members (no reaction by the end of the week), please send an email to *cg-cs-ewi@tudelft.nl* by the next Monday with the subject *[TI1806] Group Problem - Group Number XX*. We will take this into consideration for the remainder of the project and will see if we can help out.