Ray Tracing with OpenGL Debugging

Final Project CSE2215

Alderliesten, Marroquim, 2019-2020 (Version 1)

October 16, 2019

Deliverables: To be handed-in on Brightspace.

- Zipped Source Code (.cpp and .h/.hpp files)
- The Model(s) created in a model.zip folder.
- Rendered Image, to be used for the competition, in JPG format.
- Bullet-list report, with all features, in PDF format.
- Presentation, in PDF format.

Examination: Your work is evaluated through a presentation that takes place either 31 October or 1 November.

Presentation: A presentation of 4-minutes and a 3-minute question session. These are maximum time limits.

Grading: 35% of your final grade is determined by your project, and 5% is determined by your presentation and questioning session. This together forms 40% of your final grade.

Please read this whole document carefully before getting started.

Introduction

Ray Tracing is one of the oldest techniques for representing three-dimensional graphics. Due to its relatively high computational requirements, it has been ignored in favor of rasterization in many real-time applications of ray tracing. Recently, with the new interest in real-time ray tracing due to hardware vendor marketing such as *NVIDIA's RTX*-series video cards, real-time ray tracing may be an achieveable goal in the coming years.

Beyond real-time ray tracing, many animation studios and production houses have utilized ray-tracing for static images for a long time. A single image is then rendered, and a movie can be created by stitching these images together. Numerous other examples of applications of ray tracing exist.

Project Description

The final project of the course will revolve around ray tracing. The goal is to develop a *full* raytracer that generates static images from a virtual scene containing lights, objects and a camera. To reach this goal, several components have to be created, which are outlined below.

Minimum Requirements for the Implementation

Your ray tracer is expected to at least be able to do the following tasks:

- Perform ray intersections with planes, triangles, and bounding boxes.
- Computate shading at the first impact point (diffuse and specular).
- Perform recursive raytracing for reflections to simulate specular materials.
- Calculate hard shadows from a point light.
- Calculate soft shadows from a spherical light centered at a point light.
- Show an interactive display in *OpenGL* of the 3D scene and a debug ray tracer. A ray from a chosen pixel should be shown via *OpenGL*, illustrating the interactions with the surfaces.
- Implement a (simple) acceleration structure.
- Show a scene created by the group, exported as a wavefront object (OBJ) and directly loaded into the application.

> You can find more information about the acceleration structure later in this document. This includes the requirements for the component.

Possible Extensions for the Implementation

Additionally, you are welcome to extend your ray tracer with extra features that seem interesting or special to you. Some examples of such extensions include the following tasks:

- Implementing a more complex scene hierarchy.
- Utilizing interpolated normals to smooth objects.
- Extending the debugger to show the n^{th} reflection of a ray via the keyboard, or triggering a ray highlighting and showing command line output of the selected ray's properties.
- Allowing modification of triangles within the ray tracer.
- Supporting refraction and the display of transparent objects.
- Supporting soft shadows and other types of light sources.
- Adding support for spherical objects (without using the *OpenGL* provided functions).
- An illustration of interesting test or corner cases.
- A numerical evaluation of the performance of your ray tracer.
- Multicore support of the ray tracer (implementing additional threads).

Project Organization

The project has numerous organizational aspects related to it, including grouping, lab assistance, and permitted inter-group communication.

Grouping

The project is organized as a group project. Groups have been generated for you based on your performance during the labwork for the course. Please be sure to contact all your group members as soon as possible and to form a channel of communication. If you have not heard from your group members on Wednesday, 16 October 2019, or your group has a size smaller than five active students, please send a message to j.w.d.alderliesten@student.tudelft.nl with your group number and names. Groups of diminished sizes will be graded differently to full groups.

> Do you need help in your group, either due to members dropping out or personal issues? Or do you have group members that are colorblind or have a visual impairment? Please send a message to: j.w.d.alderliesten@student.tudelft.nl.

Getting Help

If you require assistance, you are allowed to use the Brightspace forums for the project, or you can discuss amongst your own group. Each group has access to the general project Brightspace forums for general questions (such as about deadlines or organization), and each group gets access to a private Brightspace forum that only the course staff and teaching assistants can see (for group related issues or code related issues). Note that sharing code in the public forums **will be seen as fraud** and should not be done.

Lab Sessions

The lab sessions every Thursday will continue until the day of the deadline. During these lab sessions you are able to get help related to the project. Each week, a day before the lab begins at 13:00, you will be able to choose a slot for your group. Each slot is 20 minutes, and will provide you with a single teaching assistant who will help you with any trouble your group might have. Making use of these lab sessions is **optional** and exists for your benefit.

When you have booked a slot, please be at the lab on time. If you are late, we will not grant you extra time. Please also plan your lab meeting slot yourself, the TA will not make a plan for you or guide you through a procedure. This is your responsibility.

All lab related bookings go through the *Queue*[1] application. Only one person per group may book a slot. If a group books multiple slots, you will be denied all help during that lab session.

Presentations

The project is concluded with a presentation, which each group must give to the course staff (consisting of the two professors and either a teaching assistant or a researcher from the Computer Graphics group). Attendance for all group members at the presentation is mandatory. The presentation should consist of any features and implementation aspects you deem interesting, and should take no longer than four minutes. A question session will take place after the presentation, which will take three minutes.

All presentations take place on 31 October 2019 and 1 November 2019. The presentation order is randomized. We will announce the location of the presentations and exact schedule a few days ahead of time on Brightspace.

It is not permitted to attend another group's presentation.

Permitted Communication

You are allowed to share code amongst members of your own team and discuss anything about the project with members of your own team. It is not allowed to share any code, irregardless of size or form, with a member of another group. It is also forbidden to share any solutions, share presentations, or help another group with any issues. If you need help, you can contact the course team and use the provided channels of assistance. If any aforementioned activity does occur, all members of all involved groups will fail the project.

Using Code from Previous Years

Although code from previous years likely won't work due to the change in framework, it is forbidden to use any content or code from previous years.

[1] queue.tudelft.nl

Getting Started

To get started with the project, a guide to the distribution of work is provided along with a guide to getting started with the project in *Tucano* and some tips and advice regarding the acceleration structure.

Distribution of Work

Although you are free to distribute the tasks as you see fit, we do expect you to write down which group members have been assigned which tasks. This document needs to only be a high-level overview, but will be utilized in the case that there are any disputes among the group. **Grades will be adjusted for individual performance if deemed necessary**, meaning you can fail the project if you do not contribute sufficiently to your group's work. You do not need to hand-in this work distribution sheet, but we do expect you to clearly state which group member did what in the final report.

Working within *Tucano*

We have built a framework in *Tucano* to act as a preview of your scene, and to appropriately store the geometry and appearance of the scene. You will be able to load an *OBJ* file with materials *MTL*, and fly around your scene using the keyboard and mouse. You can also shoot a ray from your current mouse position and see it rendered in your scene. Note that for this debug ray no intersection is calculated, much less reflections (this is part of the assignment).

The code is structured in such a way that you only need to work with the **Flyscene** class. Nevertheless, you are allowed to alter other files and methods if you like.

The method **raytraceScene** loops over all pixels of the screen and generates a ray in world space for each one. It then calls the method **traceray** where you should write the appropriate code for tracing the ray in the scene, and return an RGB value (in range [0,1] for each channel). Your result will be written to a default *result.ppm* image.

The Tucano **mesh** class is not only responsible for maintaining the *OpenGL* buffers to render the preview, but also holds all the geometric information that you will need for your ray tracer. The basic structure is: list of vertices, list of normals, and list of faces with vertices ids, material id, and face normal.

All the necessary methods and their relevant data have been listed below for your convenience.

You can easily access the individual vertices and normals with following methods:

```
// return the number of vertices
    int num_verts = mesh.getNumberOfVertices();
// return the ith vertex and associated normal
    Eigen::Vector4f v = mesh.getVertex(i);
    Eigen::Vector3f n = mesh.getNormal(i);
```

You will also need to access the faces (triangles) of your models:

```
// returns the number of faces
    int num_faces = mesh.getNumberOfFaces();
// returns the ith face of the mesh
    Tucano::Face face = mesh.getFace(i);

// returns the id of the jth vertex of a face
    int vertex_id = face.vertex_ids[j];

// returns the actual jth vertex (xyzw) of the face
    Eigen::Vector4f v = mesh.getVertex(face.vertex_ids[j]);
// returns the normal (xyz) of the jth vertex of the face
    Eigen::Vector3f n = mesh.getNormal(face.vertex_ids[j]);

// returns the material id (from mtl file) of the face
    int mat_id = face.material_id;

// returns the normal of the face
    Eigen::Vector3f facenormal = face.normal;
```

Note that the vertex normals were computed by averaging all faces around the vertex, and are accesed with **mesh.getNormal(i)**. However, when you call *face.normal* you are actually getting the normal per face (or the normal of the plane containing the triangle).

You can then utilize code such as the sample found below to print the first 10 faces of the model you have loaded:

```
for (int i = 0; i < 10; ++i)
{
        Tucano::Face face = mesh.getFace(i);
        std::cout << "face " << i << std::endl;
        for (int j =0; j < face.vertex_ids.size(); ++j)
        {
                std::cout<<"vid "<<j<< " " << face.vertex_ids[j]
                        << std::endl;
                std::cout<<"vertex "<<
                        mesh.getVertex(face.vertex_ids[j]).transpose()
                        <<std::endl;
                std::cout<< "normal "<<
                        mesh.getNormal(face.vertex_ids[j]).transpose()
                        <<std::endl;
        }
        std::cout<<"mat id "<<face.material_id<<std::endl<<std::endl;
        std::cout<<"face normal "<<face.normal.transpose()<<std::endl;
}
```

When you load an *OBJ* file with an associated *mtl* file, *Tucano* will import the materials from the *mtl* file. For the preview, these materials are transformed into Phong Models (ka, ks, kd and shininess). But the complete *mtl* format contains addition illumination parameters that you can use to enhance your ray tracer.

The materials are stored in the **vector¡Tucano::Material::Mtl¿ materials** in the **Flyscene** class. You can access the material properties by using the following code:

```
// return the ith material
    Tucano::Material mat = materials[i];

// return the Ambient component
    Eigen::Vector3f ka = material.getAmbient();

// return the Diffuse component
    Eigen::Vector3f kd = material.getDiffuse();

// return the Specular component
    Eigen::Vector3f ks = material.getSpecular();

// return the Shininess coefficient
    float shininess = material.getShininess();

// return the index of refraction
    float refraction_index = material.getOpticalDensity();

// return the transparency (1.0 = fully opaque, 0.0 = fullly transparent)
    float transparency = material.getDissolveFactor();
```

Thus, to sum up, you now have a list of materials imported from the **mtl** file, they are stored in the **materials** vector in the **Flyscene** class. You can access the faces by querying the mesh. Each face has an index to its materials that you can retrieve from the **materials** vector. If a ray hits a face you can compute the appropriate illumination using the provided geometry and the material information.

Note that the OBJ importer expects only triangles, even though the OBJ format also works for other primitives. So make sure when you export your scene to OBJ, it contains only triangles.

Finally, the previewer will start with one point light source (represented as a yellow sphere in the scene). You can add more light sources by pressing 'L', and use then in your raytracing. The **vector<Eigen::Vector3f> lights** in the Flyscene class contains all the positions of the light sources. Just note that the previewer will only render using the last light source you set, not all.

Implementing the Acceleration Structure

To find the triangles that intersect with a given ray in a naive implementation, all triangles within the model need to be tested. This process is very inefficient & slow, as you will find out during your initial work. Therefore, your task is to implement a simple spatial acceleration structure using axis-aligned bounding

boxes. **If you choose to implement something more complicated, we request that you still implement this solution as a reference and store it somewhere in your repository**.

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 a start, 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, such as a lookup table to determine the box sides that need checking depending on the ray direction. Or a special quad intersection test.

To help you understand this process a bit more, the following step-by-step instructions are provided:

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 (such that the object appears small on the image) and one where you are zoomed in (such that 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 (such as the 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.

Deliverables

At the end of the project, you are expected to hand-in numerous documents and files, and you have the option to participate in an image competition.

Report

Once you are finished programming, you are expect to hand-in a report containing only a bullet-list of all the features you have implemented. For each feature, you must provide an image showing that feature and state the name of the person or people who worked on that feature. If you relied on any sources for possible tips or additional implementations, we expect you to cite these at the end of the report.

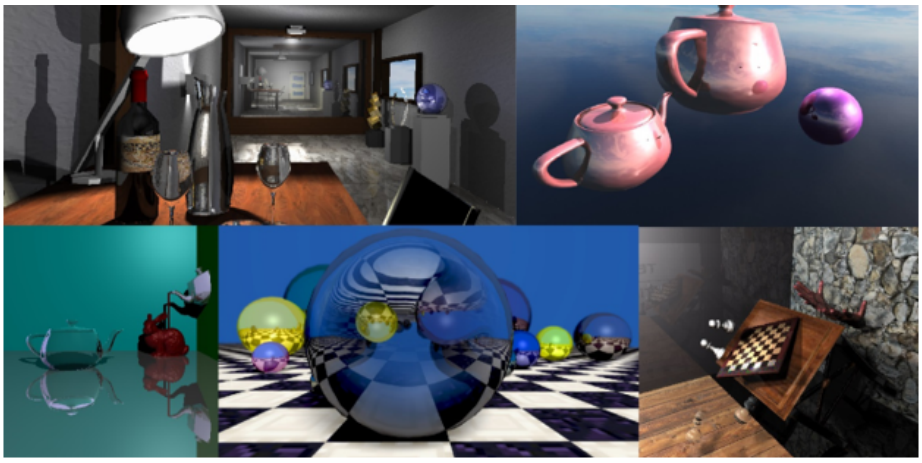
Files to Submit

Your Brightspace submission needs to be done within the group assignment found within the course. You may upload multiple files at once into the submission, but we request you use the following submission format:

- Zip the source code for your ray tracer, containing only the .cpp and .h/.hpp files, and name it groupnumber.zip.
- Place the model(s) you have created into a zip called model.zip.
- Upload the report you have created and call it groupnumber.pdf.
- Upload your presentation as a PDF and call it groupnumber-presentation.pdf.

Image Competition

An image competition will be held in which the course staff will look at individual images produced by your ray tracers. If you make a cool model and have a cool resulting image, you may upload this image alongside the rest of your files on Brightspace as groupnumber.jpg. If you upload an image, you can earn up to 1.5 points bonus (read: 15% bonus on your overall project grade). Participation is voluntary.



Entries for the image competition in previous years.

Deadline

Although there are multiple presentation dates, we expect the final submission of your project work to have taken place on Brightspace **before 31 October 2019, 08:45**. Late submissions will not be accepted under any circumstances.

> Do you have any questions not answered in this report? Send a message to: j.w.d.alderliesten@student.tudelft.nl.