Ray Tracer Project Specification v1

DH2323 (DGI17)

KTH Royal Institute of Technology

Mikael Forsberg <miforsb@kth.se>
Robin Gunning <rgunning@kth.se>

May 3, 2017

Idea

The main idea of the project is to explore more features (more than were explored in lab 2) common in ray tracing rendering in an effort to gain an increased understanding of the techniques, strengths and weaknesses of conventional ray tracing in general. Starting from the feature set established by the ray tracer implemented in lab 2 (but not necessarily directly building upon that code) we will be adding a number of features to be implemented in a single executable application.

Project plan

The following section contains two sets of implementation goals. The first set ("Baseline") is what we consider to be the minimum for project completion, while the second set ("Bonus") contains ideas that are not essential but could improve the project should we find the time to implement them.

Baseline features

- **Fast wireframe view**
  The application should default to a fast wireframe renderer for positioning the camera before initiating the potentially time-consuming high-quality render by means of ray tracing. The user should, from the wireframe view, be able to initiate a single-frame high-quality render or simply a switch to continuous ray tracing (along with the ability to switch back to wireframe mode). The code for this will based on existing work done for lab 3.

- **Multithreading**
  Rendering should be parallelized by partitioning the screen space into N parts and rendering each part in a separate thread. The number of threads should be configurable by command line argument and/or at runtime. The well-known `pthreads` library will be used for thread setup and management.

- **Back-face culling**
  The application should at runtime support enabling/disabling back-face culling, which when enabled shall remove from the rendering context those triangles that do not face the camera. This is a simple optimization and the basic code has already been written while experimenting with the labs.

- **Loading of models in the `.STL` format**
  The application should, at runtime and by command line argument, support loading models stored in the `.STL`[1] format. This is a very simple format that is popular within the 3D printing community. The basic format supports nothing more than triangle geometry with face normals, making it trivial to implement.

- **Loading of models in the Wavefront `.OBJ` format**
  The application should, at runtime and by command line argument, support loading triangle geometry stored in the `.OBJ`[2] format. Features supported by the format that are outside the scope of basic triangle geometry (free-form curves, smoothing groups etc.) are not part of this goal and so may or may not be implemented.

  Most of the code required for this has already been written while experimenting with the labs. The `.OBJ` format itself does not include texture information. There is however a bonus goal for supporting the companion `.MTL` format which would allow the application to load models with surface material properties such as textures.

- **Materials**
  Primitives such as triangles should support being assigned a "material" giving properties such as base color equivalent to the color property of the `Triangle` objects used in lab 2. In addition to the base color, the following properties should be implemented:

  - **Phong shading**
    We will implement the well-known combination of Phong interpolation (interpolation of vertex normals over surfaces) and the Phong reflection model to enhance shading of complex surfaces. Each material should therefore have the appropriate properties for specular, diffuse and ambient reflection.

  - **Texture mapping**
    Triangle primitives should support texture mapping by equipping each vertex with a pair of numbers denoting texture coordinates. The basic code for this has already been written while experimenting with lab 3. Different techniques may be required for other primitives (see further down).

[1] https://en.wikipedia.org/wiki/STL_(file_format)
[2] http://www.martinreddy.net/gfx/3d/OBJ.spec

- **Opacity**

  A material should have a configurable opacity property with a range of $[0, 1]$, where a value of 0 would indicate a fully transparent material and a value of 1 would indicate a fully opaque material. A fully transparent material would not reflect any color whereas a fully opaque material would simply reflect the color of the material equivalent to surfaces in lab 2. A semi-transparent material should reflect some combination of the material color and some other color gathered by casting a new ray to simulate light passing through the material and potentially hitting some other object.

- **Specular reflection and reflectivity**

  A material should have configurable properties regarding specular reflection and reflectivity. Reflections should be implemented by casting additional rays according to the law of reflection and mixing the resulting color according to the reflectivity setting, where zero reflectivity would mean to simply return the materials own color and full reflectivity would mean that the resulting color is entirely determined by the material struck by the reflected ray (if any). To avoid infinite loops we will keep track of the number of reflections that have previously occurred when creating a new ray and cancel the new ray if this number grows too large. The limiting number of "generations of reflection" should be configurable at runtime.

- **Refraction**

  A material should have configurable properties regarding refraction. Refraction should be implemented by casting additional rays simulating Snell's law. Similar to reflections, we will need to keep track of the number of "generations of refraction" to avoid potential infinite loops.

- **Non-triangle based primitives**

  - **Spheres**

    The ray tracer should support rendering perfect spheres defined geometrically by position and radius. Spheres should support the same material settings as other primitives and should also be shown appropriately in the wireframe view.

- **Area lights and soft shadows**

  Possibly the most technically complicated feature, we will attempt to implement soft shadows by some method of simulating area lights (as opposed to the point light simulated in the labs). We have not yet decided on the exact method to employ here. Candidates include approximating area lights with a number of point lights, casting additional rays to compute a non-integer shadow value or using some specific algorithm such as soft shadow volumes[3].

[3] https://graphics.ethz.ch/teaching/former/seminar/handouts/Lang_SoftShadowVolumes.pdf

Bonus features

- **Support the `.MTL` format as companion to `.OBJ`**
  Reading `.MTL` companion files when loading `.OBJ` models in order to load textured models (and perhaps other material properties).
- **Anti-aliasing**
  Implement some kind of supersampling to alleviate aliasing (jagged edges). A likely candidate technique should be adaptive supersampling where only pixels at the edges of objects are supersampled.
- **Fast flat-shaded view**
  As an alternative to the wireframe mode, shade (fill) each triangle by simple calculation of color involving the normal of the triangle and the direction of a light source.
- **Normal mapping**
  Use special texture maps to ”fake” details without increasing geometry complexity.

Implementation

The features specified in the previous section will be implemented in a single executable application using C++. We will be borrowing much of the provided/resulting code of lab 2, but will not simply be extending it. For example, we will not be using the `Triangle` class as we will be using individual vertex normals. We will implement the application using the object-oriented facilities of C++, using standard OOP practices for modularity such as encapsulation, separation of concerns and principle of least knowledge. We will be using SDL2, and as such will not be using the SDL helper code provided for the labs. We are not planning to divide the work between individual team members in advance, but leave open the possibility of doing so for individual tasks/features as the project progresses. We are aware of the requirements regarding keeping track of individual contributions and will do our best to accomplish this.

Evaluation

The final application could be evaluated by comparing images created with other renderers to images created with our application. It may be somewhat tricky to find good candidate images along with the data needed to render the same scene, but we believe there are atleast a few commonly used scenes and objects (the Cornell box, the Stanford bunny) available for such comparisons.