Raytracer 1

February 13, 2018

Computer Graphics

General note:

In various exercises you will be asked to implement new functionality. In these cases your ray tracer should accept the (syntax of the) example scene files provided. Under no circumstances should your ray tracer be unable to read older scene files (those that do not enable the new functionality), or modify the interpretation of older scene files.

Getting started

In this assignment you will set up the environment for your ray tracer implementation. A note on programming languages: the framework we provide is written in C++.

Tasks:

- Download the source code of the raytracer framework. The code can be found on Nestor. The C++ version includes a CMakeLists.txt file for g++ You should check that your implementation works on the LWP systems in the practical rooms.
- Look at the included README file for a description of the source files and how to compile/run the raytracer.
- Compile it and test whether it works. Using the supplied example scene `scene01.json` the image in Figure 1 should be created:

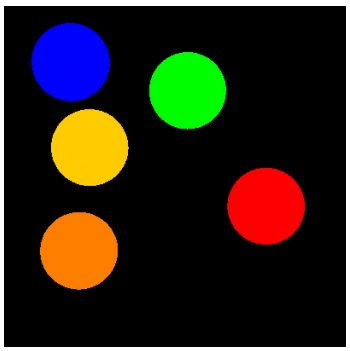


Figure 1: *Spheres*

- Look at the source code of the classes and try to understand the program. Of particular importance is the file `triple.h` which defines mathematic operators on vectors, points, and colors. The actual raytracing algorithm is implemented in `scene.cpp`. The json-based scene files are parsed in `raytracer.cpp`.

1 Raycasting with spheres & Phong illumination (4 points)

In this assignment your program will produce a first image of a 3D scene using a basic ray tracing algorithm. The intersection calculation together with normal calculation will be the groundwork for the illumination.

- For now your raytracer only needs to support spheres. Each sphere is given by its centre, its radius, and its surface parameters (Look at `material.h` and `material.cpp`).
- A white point-shape light source is given by its position (x, y, z) and color. In the example scene a single white light source is defined.
- The viewpoint is given by its position (x, y, z). To keep things simple the other view parameters are static: the image plane is at $z = 0$ and the viewing direction is along the negative z-axis (you might improve this later).
- The scene description is read from a file in `raytracer.cpp`.

Tasks:

1. (1 point) Implement the intersection calculation for the sphere. Extend the function `Sphere::intersect()` in the file sphere.cpp. The resulting image should be similar to Figure 2:

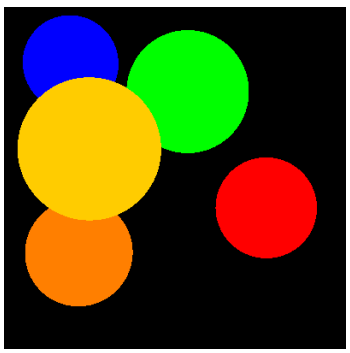
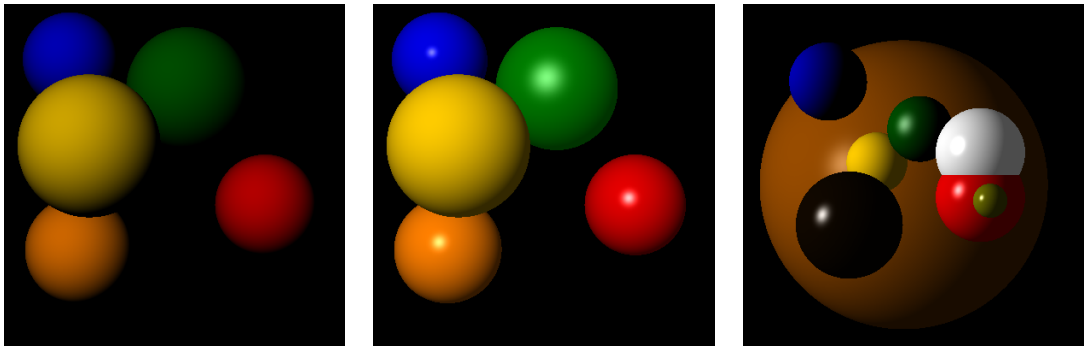


Figure 2: *Spheres after intersection calculation*

2. (1 point) Implement the normal calculation for the sphere. To this end, complete the function `Sphere::intersect()` in the file `sphere.cpp`. Because the normal is not used yet, the resulting image will not change.
3. (0.5 point)Implement the diffuse term of Phong's lighting model to obtain simple shading. Modify the function `Scene::trace(Ray)` in the file `scene.cpp`. This step requires a working normal calculation. The resulting image should be similar to the image in Figure 3a.
4. (1 point) Extend the lighting calculations with the ambient and specular parts of the Phong model. This should yield the result in Figure 3b.

5. (0.5 point) Test your implementation using scene file `scene02.json`. This should yield the result in Figure 3a.



(a) *Phong diffuse* ***(b)*** *Full Phong* ***(c)*** *Scene 2*

Figure 3: *(a) Simple shading and diffuse, (b) Full Phong lighting model applied, (c) The resulting rendering of* `scene02.json`

2 Additional geometry and meshes (4 points)

In this assignment you will implement at least three other geometries beside spheres and will be rendering a mesh obtained from an *.obj* file.

Tasks:

1. Implement the Triangle shape (1 point) and at least two extra geometries (2 points) from the following list (ordered by difficulty):

   - Plane
   - Quad
   - Cylinder
   - Cone
   - Torus

   Only three things need to be added for each new geometry:

   1. Define (`in the .json scene file`) and read the parameters (`raytracer.cpp`)
   2. Intersection calculation (in new shape class)
   3. Normal calculation (in new shape class)

   For this you can take the code files `sphere.cpp` and `sphere.h` as an example. Do not forget to rerun `cmake ..` for the new source files to be build!

   (NOTE: (for now) it is not needed to be able to rotate cylinders, cones or tori and having them fixed along an axis is sufficient.)

2. (1 point) You now should be able to render triangles in the raytracer. From the OpenGL exercises you should understand that most models consist of a collection of triangles. Use the provided OBJLoader class to load all triangles from a scene. Some hints to a possible implementation:

- Create a "mesh" type and a "model" attribute which points to the .obj file (relative to the raytracer executable!) in your scene descriptor.
- The OBJLoader class is the very same class used in the OpenGL assignments, so you will retrieve interleaved data (in a struct). Keep this in mind when creating your triangles.
- You can either add all triangles as single objects to the scene or create a new "Mesh" class which contains a collection of triangles and add that to the scene. The first may be easier to implement, while the latter may allow for more configuration later one. Either is fine.
- **Note!** Raytracing models with many triangles takes a lot of time. Please test on simple models first, such as a rotated cube or another model with less than a thousand triangles.
- **Optional!** If you want, you can easily speed up the ray tracing using multi-threading with OpenMP. Add:
  `-fopenmp`
  to the `CXX_FLAGS` in your `CMakeLists.txt` file and add
  `#pragma omp parallel for`
  just before the outer loop in `Scene::render()`.

3. Possible extensions for the competition:
   - Implement the extra shapes.
   - Experiment with your own scene descriptions. Try to build a composite object using triangles or quads. Even with just spheres you can build some interesting scenes! **Note!** Keep rendering time reasonable, do not use very complex models. Please include the resulting images in a `Screenshots` folder in your submission.

Deadline

See Nestor (*Time Schedule*). Details on how to submit your work can also be found on Nestor (*Lab Assignments*).

Assignment submission

Please use the following format:

- Main directory named `Lastname1_Lastname2_Raytracer_1` , with last names in alphabetical order, containing the following:
- Sub-directory named Code, containing the modified C++ framework (please do **not** include executables or build folders)
- Sub-directory named Screenshots wherein you provide the relevant screenshots/rendered images for this assignment
- README (plain text, short description of the modifications/additions to the framework along with user instructions)

The main directory and its contents should be compressed (resulting in a **zip** or **tar.gz** archive) which is the file that should be submitted (using the *Assignment Dropbox*). An example of a file to be submitted associated with the first raytracer assignment would be: `Catmull_Clark_Raytracer_1.tar.gz`.

Assessment

See Nestor (*Assessment & Rules*).