Raytracer 1

February 6, 2017

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 Makefile for gcc/MingW (for building on Windows). You should check that your implementation works on the LWP systems in the practical rooms.
- Compile it and test whether it works. Using the supplied example scene `scene01.yaml` the image in Figure 1 should be created:

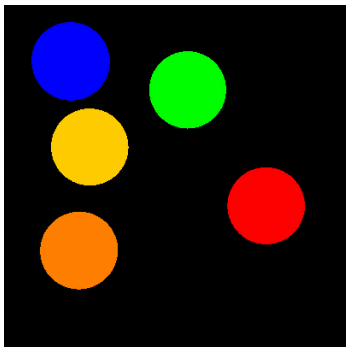


Figure 1: *Spheres*

- 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 YAML-based scene files are parsed in `raytracer.cpp`. An instructive guide on parsing YAML files is available on Nestor. Look at the included README file for a description of the source files.

Raycasting with spheres & Phong illumination

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 will improve this later).
- The scene description is read from a file in `raytracer.cpp`. Additional information about parsing YAML-files is available on Nestor.

Tasks:

1. 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:

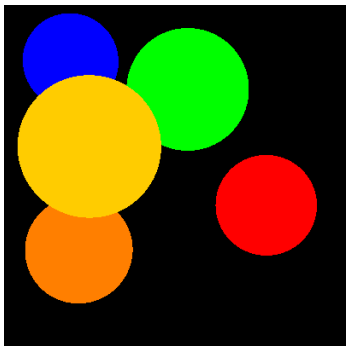
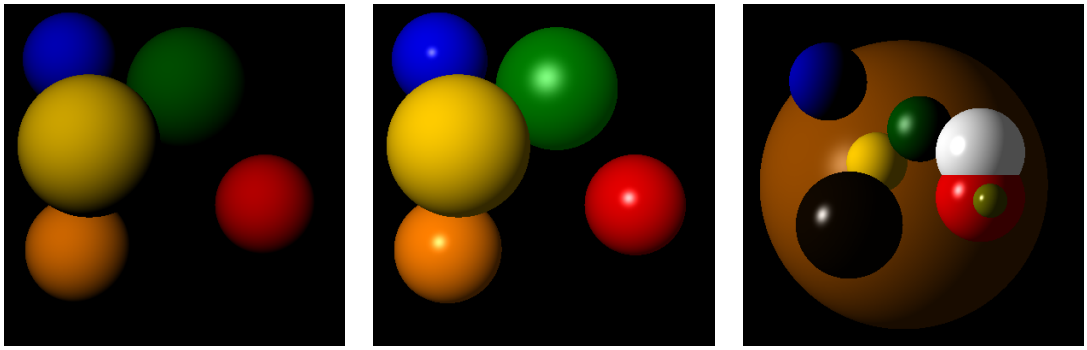


Figure 2: *Spheres after intersection calculation*

2. 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. 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. Extend the lighting calculations with the ambient and specular parts of the Phong model. This should yield the result in Figure 3b.

5. Test your implementation using scene file `scene02.yaml`. This should yield the result in Figure 3c.



(a) ***(b)*** ***(c)***

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

Normal buffer, z-buffer & additional geometry

In raytracing, Hidden Surface Removal (HSR) is basically included in the technique and you are using (without noticing) a z-buffer-like algorithm in your raytracer. In this assignment you will gain a deeper understanding of this process. In addition you will create a normal buffer. Both buffers allow you to gain insight in what is rendered on screen.

Tasks:

1. Adapt your program such that it can also produce a z-buffer image instead of the usual rendering. This should be configurable in the scene file. For this introduce a RenderMode directive which can be set to z-buffer instead of the default Phong. Use gray levels to code distances. Make sure you use the whole range of gray levels available to display the image (scale using min, max). An example z-buffer image can be found in Figure 4.

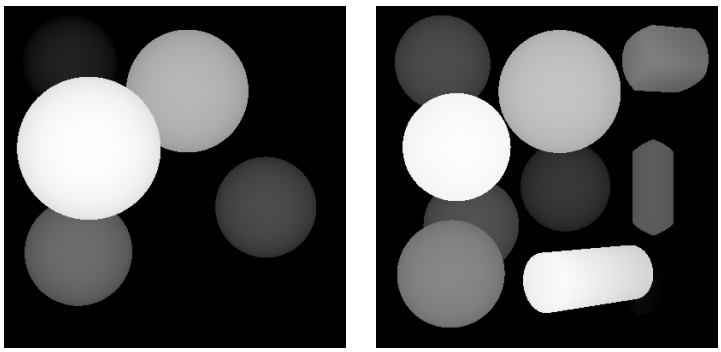
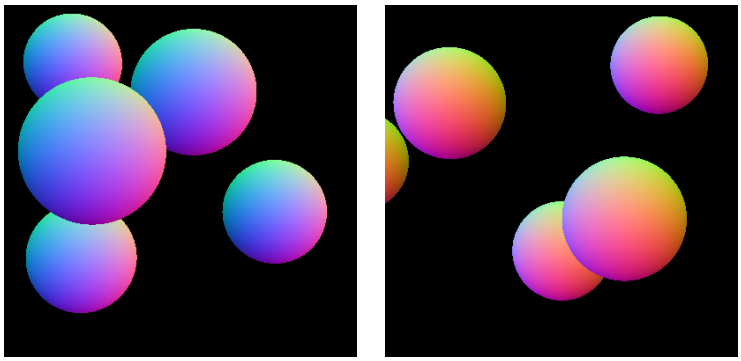


Figure 4: *z-buffered scenes*

2. Adapt your program such that it can also produce a normal buffer image instead of the normal rendering (again, this should be configurable in the scene file, name it

normal). Map the three components of a normal to the three colors and be sure to map the possible range of the components ($-1..1$) to the range of the colors. Figure 5 shows the normal buffer from two different eye positions. Figure 5b is taken from $[1000, 200, 200]$, and needs an adjustment in `scene.cpp` to look in the right direction (NOTE: it is not requested that you implement a different camera position for this assignment):



(a) ***(b)***

Figure 5: *The normal buffered spheres from different camera positions*

3. Implement at least two geometries (more will count towards bonus) from the following list (ordered by difficulty):

   - Plane
   - Triangle
   - Quad
   - Cylinder
   - Cone
   - Torus

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

   1. Reading the parameters (`raytracer.cpp`)
   2. Intersection calculation
   3. Normal calculation

   For this you can take the code files `sphere.cpp` and `sphere.h` as an example. Do not forget to add your new files to the MakeFile!

   (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.)

4. (Bonus) 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!

Use the following additional scene files to test your implementation:

- `scene01-phong`: Explicitly uses Phong shading.

- `scene01-zbuffer`: Shows z values.
- `scene01-normal`: Shows normal buffer.

These files can be found in the scenefile archive found on Nestor and you should be able to reproduce the images included in this document.

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)
- 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*).