Raytracer 2

March 1, 2019

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. Make sure you set a default value for new functionality if none are given in the scene file.

This time, we learn how to add optical laws, texturing and anti-aliasing into our framework. You can download the `.json` files, which describe added scene elements, from Nestor.

3 Optical laws (3 points)

In this assignment you will implement a global lighting simulation. Using recursive ray tracing the interaction of the lights with the objects is determined. The program should be able to handle multiple colored light sources and shadows.
Tasks:

1. **(1.25 points)** Extend the lighting calculation in `Scene::trace(Ray)` such that it produces shadows. First make it configurable whether shadows should be produced (e.g., "Shadows": true). The general approach for producing shadows is to test whether a ray from the light source to the object intersects other objects. Only when this is not the case, the light source contributes to the lighting. For the example in Figure 1 a large background sphere is added to the scene.

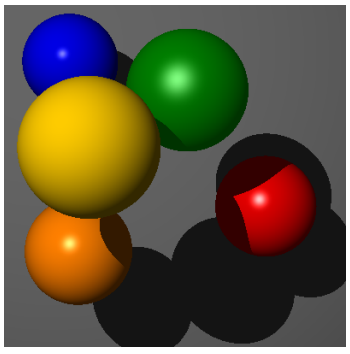


Figure 1: *shadow (`scene01-shadows.json`)*

2. **(0.5 point)** Now loop over all light sources (if you didn't do that already) and use their color in the calculation. For the following result two different lights were used. The output of this scene file is in Figure 2.

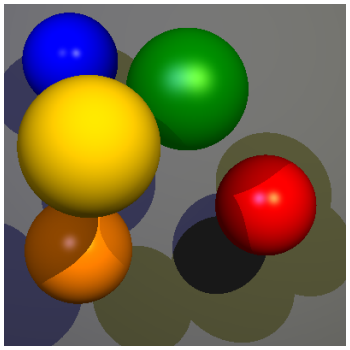


Figure 2: *Shadows caused by multiple lights (`scene01-lights-shadows.json`)*

3. **(1.25 points)** Implement reflections, by recursively continuing rays in the direction of the reflection vector, treating the found values as light sources. Be sure to only compute the specular reflection for these "light sources"(ambient makes no sense at all, and diffuse reflection is better approximated by not taking it into account in this coarse approximation). An example result with a maximum of two reflections is shown in Figure 3.

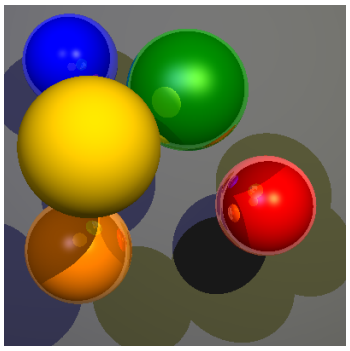


Figure 3: *Reflection (`scene01-reflect-lights-shadows.json`)*

- **Optional extension:** Implement refraction, by recursively continuing rays in the direction of the transmission vector, using the parameters `refrct` and `eta`, where `eta` is the index of refraction of the material.
- **Optional extension:** You'll notice that the specular reflections from the scene are not blurred, like the light sources. One way to do something about this is to sample along multiple rays (around the reflection vector) and average the results. Note that for a correct result you should be careful about selecting your vectors and/or the way you average them. Hint: if you take a normal average you should select more rays in those areas where the specular coefficient is high.

4 Anti-aliasing (1 point)

Anti-aliasing will result in better looking images by sampling multiple rays.

Tasks:

1. **(1 point)** Implement super-sampling (anti-aliasing), i.e., casting multiple rays through a pixel and averaging the resulting colors. This should give your images a less jagged appearance. Note that you should position the (destinations of the) initial rays symmetrically about the center of the pixel, as in Figure 4 (for 1×1 and 2×2 super sampling):

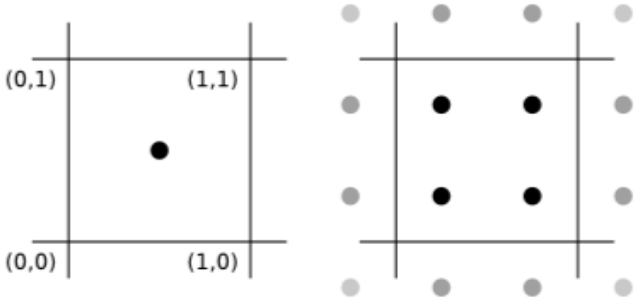


Figure 4: *Super sampling*

   Again, make sure this is configurable in the scene file. The default should be to have no super sampling (or, equivalently, super sampling with a factor of 1). An example of 4×4 super-sampling is in Figure 5 (`scene01-ss.json`).

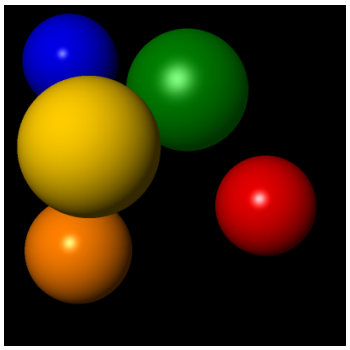


Figure 5: *Super sampling example*

5 Texturing (4 points)

In this assignment you will implement texture mapping and rotation of shapes.

Tasks:

1. With textures it becomes possible to vary the lighting parameters on the surface of objects. For this a mapping from the points of the surface to texture-coordinates is needed. You might want to make a new pure virtual function in Object (and give it a non-trivial implementation in at least Sphere) for computing texture coordinates so that it only has to be done for objects which actually need it. See this link for example textures to use and section 11.2 (2D texture mapping) of your book for how to compute the texture coordinates. For reading the textures you can use the following line: `Image texture("relative/path/to/texture.png")`, and access the

pixel data: `texture.colorAt(float x, float y)`, where x and y are between 0 and 1.

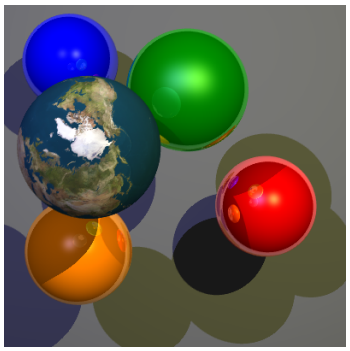


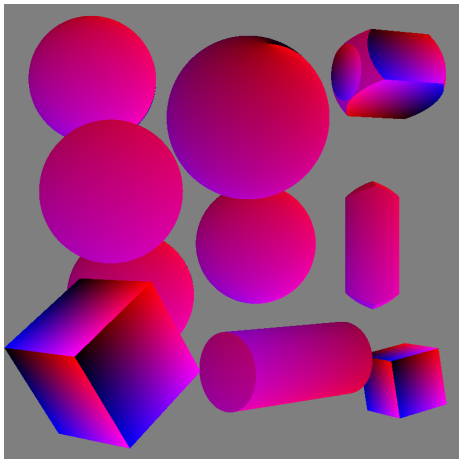
Figure 6: *Scene containing a texture mapped sphere*

For testing your texture mapping code you may want to use `bluegrid.png` instead of a real image.

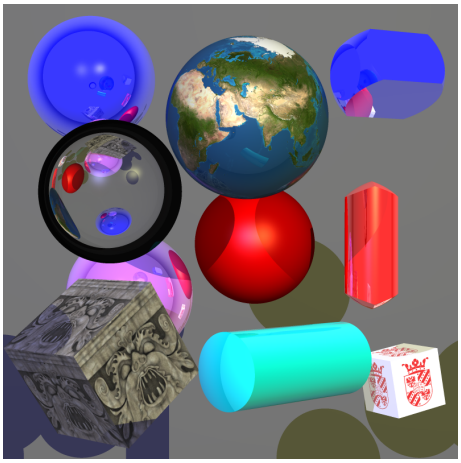
2. Also implement rotation of (at least) spheres (if you haven't already). In this case you are NOT required to exactly follow the syntax given here, but you are required to implement something that gives the same degrees of freedom. Please look at the given json scene file to view the changes to the sphere definition. You are given an axis and an angle. The radius component of the sphere has been augmented with an axis such that the first component `radius[0]` gives the radius and the second component `radius[1]` is a vector. In the example given here the vector defined by `radius[0]*radius[1].normalized()` is mapped to $(0, 0, 1)$. This is done through the simplest possible rotation and then the whole sphere is rotated by angle degrees around the z-axis (note that this corresponds to subtracting the angle in radians from the angle gotten by the arctan). The result should look like this:

Figure 7: *Scene with a rotated sphere (`scene01-texture-ss-reflect-lights-shadows.json`)*

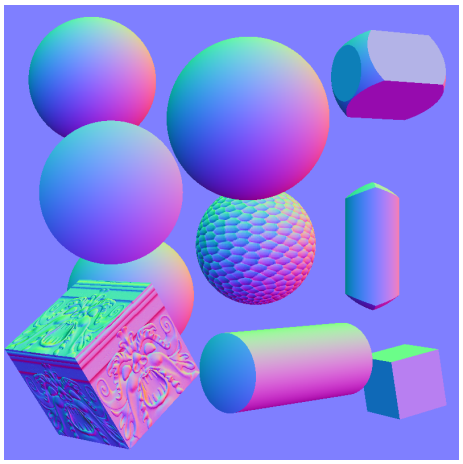
3. **Optional extension** Implement normal/bump-mapping. The results could be something similar to the following:



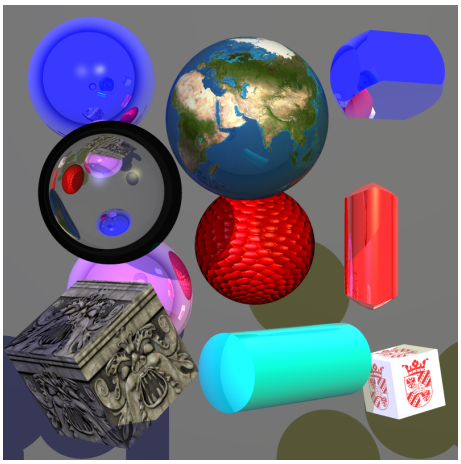
Texture coordinate buffer



Textured objects



Normal buffer



Textured objects

Figure 8: *Normal/bump maps*

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_2` , 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_2.tar.gz`.

Assessment

See Nestor (*Assessment & Rules*).