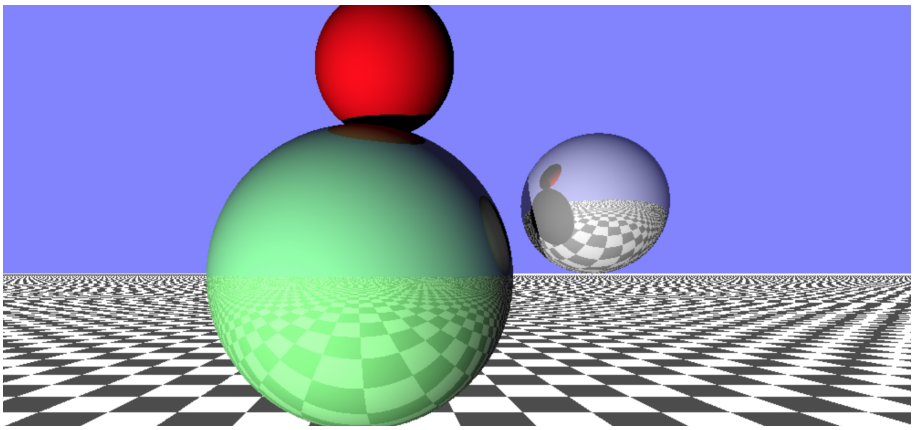
CPSC 314 Assignment 6: Ray Tracing

Out: Wed March 28, 2018
Due: Fri April 6, 2018



[10 marks total; worth 8%]

1. In this coding assignment, you will gain experience with the basics of ray-tracing. See the course web pages for the starting template.

   The code runs in the browser and can thus be run by opening `a6.html`. You will need to enable local file access, as you have done for previous assignments, and as described here: `https://threejs.org/docs/#Manual/Getting_Started/How_to_run_things_locally`.

   The template code comes with keybindings that allow you to move the light source using A,D,S,W.

   You will be implementing a basic ray-tracer in a fragment shader. All your work will be in making changes to `a6.html`, which contains the fragment shader. However, you are also free to make changes to the javascript, `a6.js` if you like.

   Debugging shaders is difficult, as there are no print statements and no standard debugging tools to view the program state. Thus be sure to proceed step-by-step.

   The template code is setup to with a coordinate system that is effectively VCS, i.e., the camera is looking down the negative z-axis. The location of all objects can also be assumed to be in VCS, and so the raytracer does not need to worry about any modeling or viewing transformations. The one exception is the checkerboard plane, which uses `planeMatrix` to transform from VCS back to the plane's local coordinate frame, and is used by `bgColor()` to compute background colors for rays that do not intersect the spheres.

(a) (2 points) As a first step, simply change the `raycast()` function to return the color of the sphere that the ray hits. Note that the provided `rayT()` function already returns the `t` value of the first sphere that the ray hits, as well as setting `nearestSphere` to point to sphere that was hit. Thus `nearestSphere.mtrl.color` contains the color of the sphere that was hit. Test this step.

(b) (2 points) Now implement diffuse shading, i.e., using $N \cdot L$. Assume that the light source is visible, although when $N \cdot L < 0$ you should shade the pixel black, i.e., implement self-shadowing. You should compute the location of the surface point, `P`, the surface normal, `N`, and the incident direction of the ray, `I`. Then call the `localShade()` function in order to compute the local diffuse shading. For extra points, you can later implement the Phong shading model there. Test this step.

(c) (2 points) Now build a shadow-ray in the `localShade` function, and use it to test whether an object exists between the surface point `P` and the light source, which is provided to you via the `lightPosition` uniform variable. You will want to use the `rayT()` function to find the closest intersection. Test this step. Your spheres should now be able to cast shadows on each other.

(d) (2 points) Now implement reflective rays, using steps 5–7 as given in the `raycast()` template code. There are comments there to point you in the right direction. Because GLSL does not allow for recursive function calls, we will simply build a copy of the `raycast()` function called `raycast2()`, and this is what you will use for the second bounce. Test this step. The shiny spheres should now produce visible reflections of the other spheres.

(e) (2 points) Creative component: implement extensions to the ray-tracer of your own choosing. Possible ideas include adding refraction, new types of primitives (planes, triangles, cylinders, cubes, etc.), Phong shading, texture-mapped surfaces, or some animation. The instructors and TAs will not respond to any questions that ask us to define this more precisely. Up to two further bonus marks are available for exceptional creative components.

Submit your code using `handin cs314 a6`.
Include a README.txt file that contains your name, your student number, and any comments and explanations that you wish to include.