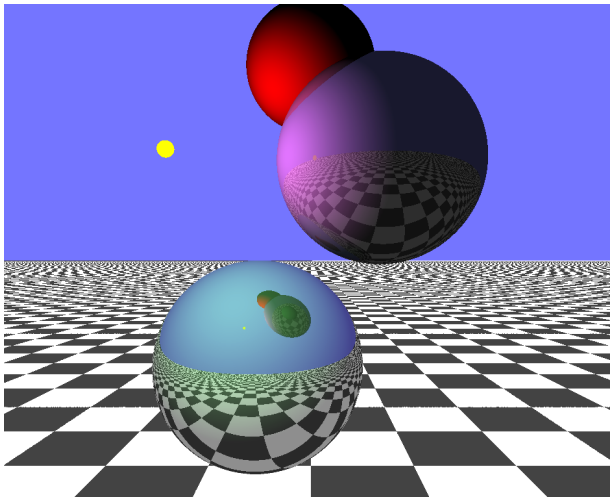
CPSC 314 Assignment 6: Ray Tracing

Out: Mon Nov 18, 2019
Due: Fri Nov 29, 2019



[10 marks total; worth 8%]

1. In this coding assignment, you will implement a basic raytracer in a fragment shader. See the course web pages for the starting template. As usual, you will need to enable local file access, as you have done for previous assignments.

   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. As in previous assignments, debugging shaders can be 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. You can cut-and-paste the full fragment shader into `https://shdr.bkcore.com/` and to use that as your development environment. There are two lines near the beginning that you will need to comment or uncomment when switching between the two environments. When done, do not forget to copy your fragment shader back into `a6.html` when done, to test it there, and comment/uncomment those lines.

   The template code is setup to with a coordinate system that is equivalent to 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.

   The template code comes with keybindings that allow you to move the light source using A,D,S,W, and to stop/start the animation using the spacebar. These bindings will not work in `https://shdr.bkcore.com/`.

(a) (1 point) As a first step, determine the basic call graph used in the fragment shader, i.e., which functions call which other functions. Document this in your README.txt file. Be sure to understand which function loops through all the spheres in the world to be rendered. Also, understand which of the spheres corresponds to the light source by using the ADSW keys to move the sphere that represents the light source.

(b) (1 point) As a first step, simply change the `raycast()` function to return the color of the sphere that the ray hits. Note that the provided `nearestT()` 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.

(c) (3 points) Now implement simple diffuse shading. Note: this is a simplification from the original version of the assignment that asked you to implement Phong shading; you can still do that for the creative component below. Assume that the light source is visible. 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 illumination. Complete the `localShade()` function.

(d) (1 point) 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 `nearestT()` function to find the closest intersection. Test this step. Your spheres should now be able to cast shadows on each other.

(e) (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.

(f) (2 points) Creative component: implement extensions to the ray-tracer of your own choosing. Possible ideas include implementing a third bounce; adding refraction; new types of primitives (planes, triangles, cylinders, cubes, etc.); texture-mapped surfaces; additional animation; and a different procedural texture for the floor. The instructors and TAs will not respond to any questions that ask us to define this more precisely.

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.