Computer Graphics Tutorial 4

In this tutorial, we will explore raytracing. The code for this assignment is based on the simple WebGL raytracing project here: http://www.mbroecker.com/project_webgltracer.html . The work will involve making edits to this existing ray-tracer code base.

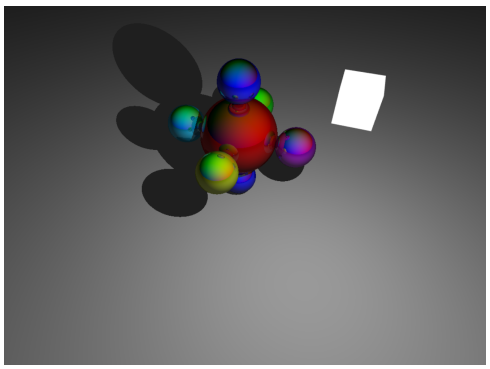
Running the Code

- Extract the given zip file from the course webpage (http://www.inf.ed.ac.uk/teaching/courses/cg/index2017.html#Tutorials).
- If using Google Chrome, you must open it via the command line with `google-chrome -allow-file-access-from-files`, which allows the 3D model file to be loaded.
- If using Google Chrome, helpful debugging tools can be opened by pressing `F12`.
- If using Firefox, debugging is opened via `Ctrl + Shift + K`, or via Firebug.

Questions

1: Exploring the Raytracer

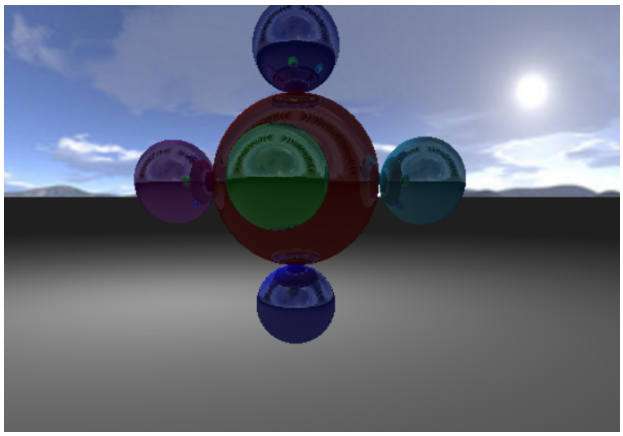
Open `WebGL_Raytracer.html` in a WebGL compatible browser. You should see:



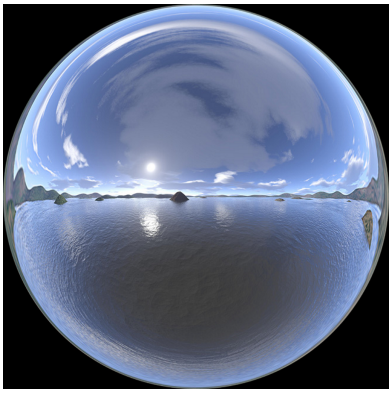
The shaders for this program are located in the `raytracer.webgl/shaders` directory. If you open one of them, you may notice that they use a slightly different syntax from prior assignments, which used WebGL 2.0 shaders. The main difference is that per-vertex input variables are labeled as `attribute` instead of `in`, and the `in`/`out` variables for passing information from the vertex to the fragment shader are labeled `varying`. Some functions such as `transpose` and `inverse` are missing from this version of OpenGL too.
The file we will be editing during this tutorial is `raytracer.webgl/shaders/raytracer.frag`. Other files of note are `shader.js` which loads in all the shaders and textures, and records which uniform variables the shaders use. The `main.js` file is where the central rendering loop occurs, and will set the vaules of uniform values each frame.

2: Sphere-mapped Skybox

Calculating Sphere-Map UVs



We will explore two different methods of drawing a distant static background in our ray tracing environment. The first will be sphere mapping, which acts as if the camera were on the inside of a spherical dome. We will use the below image for our sky sphere:
Open `raytracer.webgl/shaders/raytracer.frag`, and scroll down to the `main` method, where the ray-tracing loop is called. Whenever a ray misses objects in the scene, it will need to return a default colour. For now, it these rays return their XYZ direction as an RGB vector, but we will change this to read from our `sphereMap` texture.



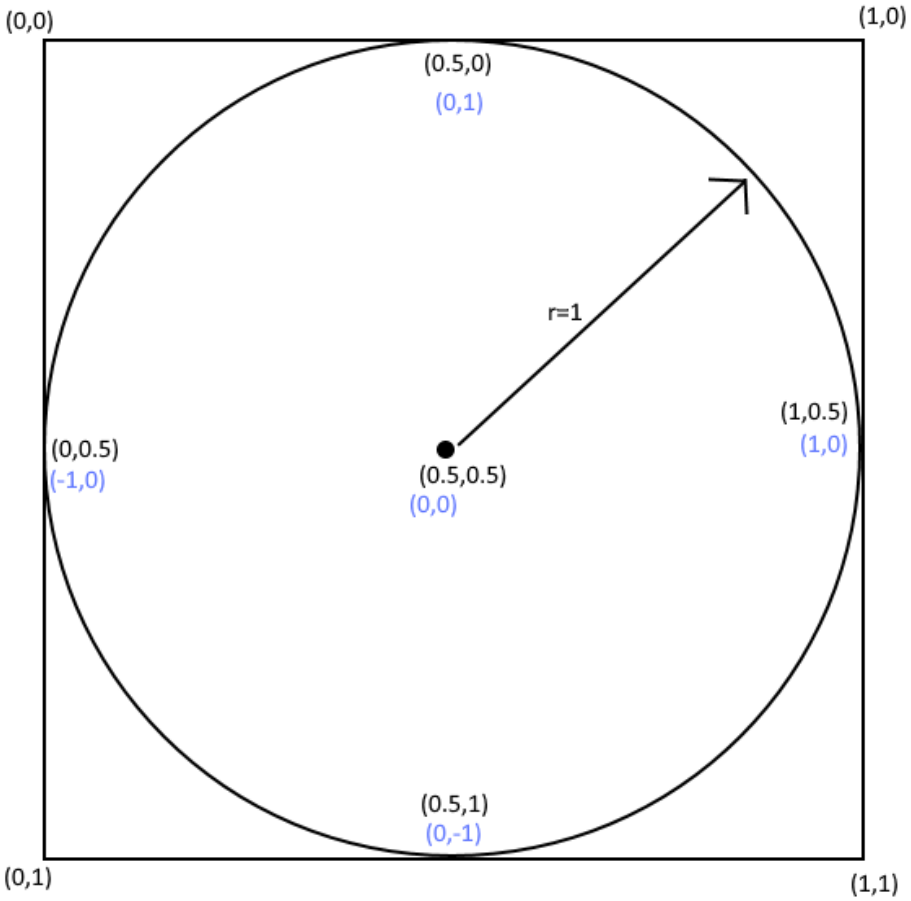
To work out which UV coordinates to use in our sphere map, we take the X and Y coordinates of `ray.direction`, and need to map them on to the 2D texture's coordinates so that they match up with the coordinates shown in the diagram below (black numbers are texture coordinates, and blue numbers are from the ray direction XY coordinates). Because the XY coordinates came from our normalized view vector, they will naturally fall within the $r = 1$ circle on our sphere map texture.
Once you have calculated the UV coordinates for each missed ray, use the `texture2D` function to read the colour from the `sphereMap` texture.

Reflections

Edit the main method of `raytracer.webgl/shaders/raytracer.frag` so that the environment from the sphere map also shows up in the reflections on the spheres. Setting the default `color` value to the sphere mapped texture only causes it to appear in the first cast of a ray, so locate the portion of the code where missed reflection rays are coloured, and make them choose a colour from the sphere map too.

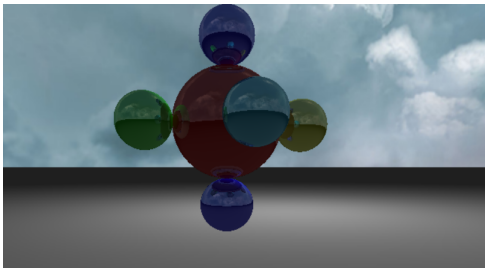
Different Maps

Download some new sphere maps from Google images. Rename them `sphere_map.png`, and text how well they work in this environment, and how visible their seams are.



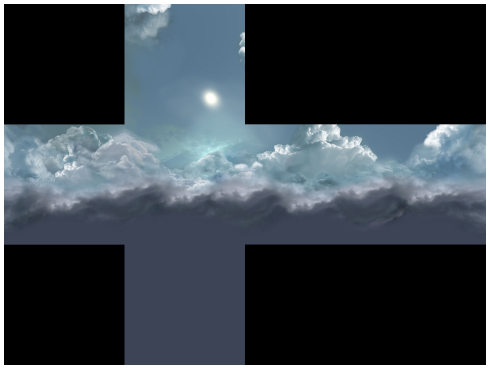
3: Cube-mapped Skybox

Sampling a Cube Map



If we look around in the sphere-mapped world, there are very noticeable seams on the edges, which are caused by distortions around the edges of our sphere-mapped image. Here, we will demonstrate a method for reducing these artefacts via a different environ-

ment mapping technique.



This time, we will sample from a cube map, which has been generated from 6 separate images, one for each face of a cube (posx, posy, posz, negx, negy, negz). In `raytracer.webgl/shaders/raytracer.frag` this has been set up in the `cubeMap` texture variable. In WebGL 1.0, we can sample from a cube map using the `textureCube` function. Instead of 2D UV coordinates, this function will use 3D coordinates to help it select the correct cube face to read from. Sample the environment colour from this cube map using `ray.direction` as the texture coordinates.

Reflections

Make sure the cube mapped sky box shows up in the reflections on the spheres too (as in part 2).

Different Maps

Download a cube mapped image (Google images brings up numerous). Divide it into 6 parts (each part should be a square with a power-of-2 number of pixels). Back up the old cube map images, and rename your new images (posx.png, posy.png, posz.png, negx.png, negy.png, negz.png) corresponding to the correct faces of the cube. Run the ray tracer, and make sure your cube faces connect smoothly with one another, including the top segment visible in the reflections of the spheres. You can rename the files, or change the ordering in the `imageUrls` array at the bottom of the `shader.js` file.

3: Raytracing Cylinders

Edit `raytracer.webgl/shaders/raytracer.frag` to fill in the `intersectRayCylinder` function. You can re-use properties of the spheres such as their positions, and using their radius as the cylinder's height. You may simplify this task by assuming the cylinder is never rotated, and their flat sides are parallel to the ground plane. Their radius and length are equal too.

You may find the intersection equations available http://www.cl.cam.ac.uk/teaching/1999/AGraphHCI/SMAG/node2.html#SECTION00023200000000000000 useful for this excersise, and reading the previous code and equations about ray-sphere intersections may be helpful too.