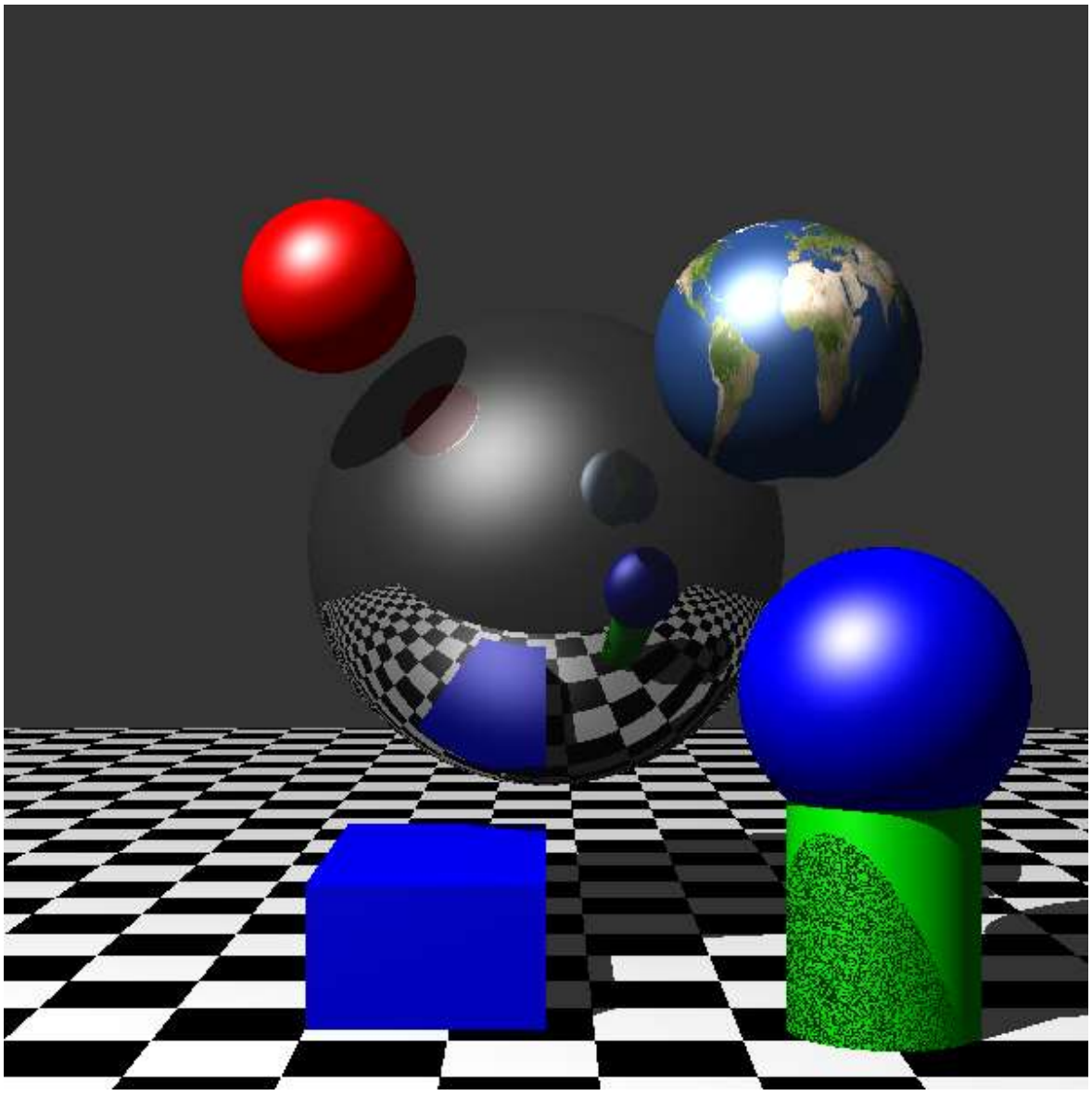
Ben Moskovitz
32520353
bjm213@uclive.ac.nz

Scene Description:



The scene shows three spheres of varying colours, one textured to look like the Earth, as well as a blue cuboid and a green cylinder, all sitting on a checkerboard plane and reflected in another sphere.

Texturing A Sphere

Texturing the sphere on the top right to look like the Earth was initially much more difficult than expected; I used the wrong equation to convert between xyz cartesian coordinates into UV texture coordinates. Initially, the equations

$$0.5 + \frac{atan2(n_z, n_x)}{2\pi}$$

and

$$0.5 - \frac{asin(n_y)}{\pi}$$

Were used to determine U and V, respectively, where the equation

$$0.5 + \frac{asin(x)}{\pi}$$

Should've been used in both cases, substituting n_x for x when determining U and n_y when determining V.

These equations, while possibly correct in some situations, are not suitable for determining UV coordinates when taking the normal vector as an argument. This caused the textured solid to appear black, with no lighting, and a blue border on one edge of the solid. It would follow from the output of these equations that when generating texture coordinates only had valid output (0 < output < 1) at the very edges of the sphere.

Procedural texturing of the "ground" plane

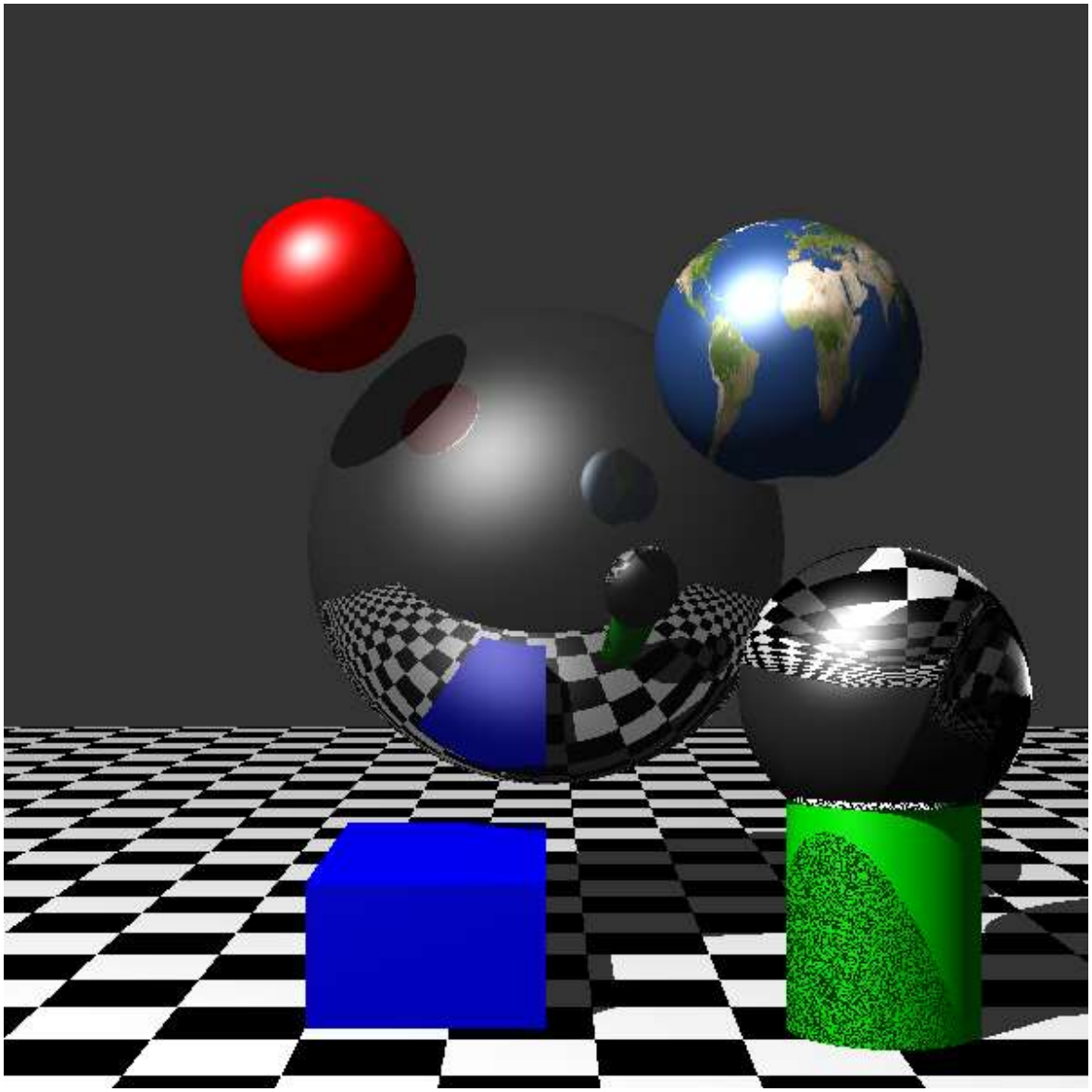
The ground plane was procedurally textured using the following algorithm

```
float u = (int) ((q.point.x - 10) / 3) % 2;
float v = (int) ((q.point.z + 20) / 3) % 2;
```

Both of these lines return -1 in certain ranges and 0 in other ranges. If u and v are equal, then the colour white is given to that pixel. If not, the colour black is given. This results in a checkerboard texture generated on the ground plane, as demonstrated in the scene description above.

Refraction

Although refraction was not included in the final scene description, there is a (broken) implementation of it included, commented out, in the code provided. When uncommented, it produces the following output:



While this implementation of refraction is obviously incorrect, the method followed was this:

1. Find the “entry point” of the ray into the sphere using `q.point`.
2. Find the direction vector of the ray refracted within the sphere using Snell’s Law of Refraction
3. Use the `closestPt()` function to determine the “exit point” of the ray - the location on the other side of the sphere that the refracted ray will intersect
4. Repeat steps 1 and 2 to determine the refracted vector when changing media from the sphere’s medium (in this case diamond) to air
5. Call the `trace()` function using the vector determined to determine the colour of `q.point`

I suspect that the error in this algorithm might be that using the `closestPt()` function while the vector is “inside” an object doesn’t return the point on the other side of the object - that is, in this case, the raytracer doesn’t see spheres as “hollow”.

Adding a Cylinder

Adding a cylinder to the scene also proved difficult at first. The quadratic equation was used to solve the cylinder-vector intersect equation:

$$t^2(d_x^2 + d_z^2) + 2t\{d_x(x_0 - x_c) + d_z(z_0 - z_c)\} + \{(x_0 - x_c)^2 + (z_0 - z_c)^2 - R^2\} = 0$$

This yielded both points of intersection, the nearer of which, and the one that was within the cylinder’s `height` variable was returned.

Initially, there were issues with this quadratic solver where the b term ($\{d_x(x_0 - x_c) + d_z(z_0 - z_c)\}$) wasn’t multiplied by 2, causing the function `Cylinder::intersect()` to fail silently, which caused the `Cylinder::normal()` function to never be called. The impact of this was that the only evidence of a cylinder being rendered was that there was a small shadow wherever it intersected another object.