Kyle Rathman
CSCI 441
Project 2

To demonstrate the following features, other features first needed to be added to the program. The ability to load in an object from a .OBJ file as a vector of triangles[1] was added. The resulting vector had smoothed normal calculated and was placed inside a bounding box.

Additionally, the ability to calculate texture coordinates was added to Spheres using the given point on the sphere. This allowed the textures generated with Perlin noise to be displayed.

The method of returning a background color in the Camera class was also changed. To create an 'outer space' effect, the background is a very dark grey (to make reflective objects easier to see), and has a small chance to return white instead, representing a distant star.

Finally, the method for calculating a triangle's hit point was modified slightly using the Möller–Trumbore intersection algorithm[2], marginally decreasing runtime as the number of triangles in the imported model increases.

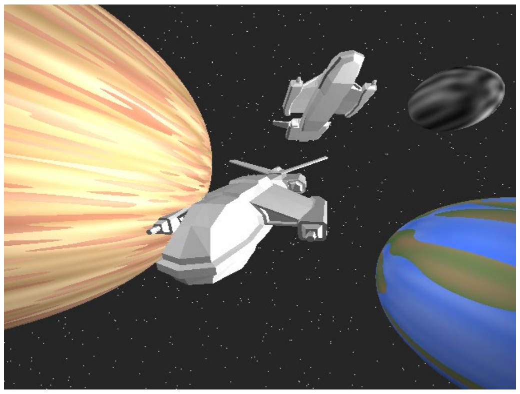
Reflections

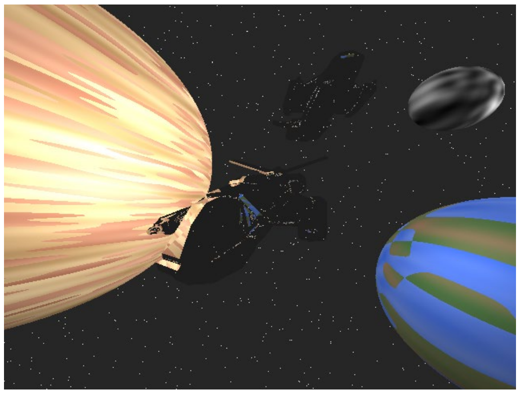
Reflections were added primarily to the Intersector, with functionality added as necessary to the Shape, Hit, and Renderer classes, as well. If the Intersector finds that the nearest detected surface is marked as 'reflective', it calculates a new ray and repeats the process of finding a new surface. The position of the new ray is the calculated position of the hit. The direction is mathematically calculated as being reflected across the normal of the point hit by the original ray[3]. Given R_d as the direction of the incoming ray and N as the direction of the normal at the hit point, the direction of the new ray is calculated as follows:

$$R_d - 2 * (N \cdot R_d) * N$$

Theoretically, if left to its own devices, this could lead to an infinite loop as the ray bounces between two reflective surfaces. To prevent this, the Intersector function was given a 'time to live' argument that decrements on each 'bounce'. The reflection can bounce 20 times before it forces the ray to simply return the color of the reflective surface.

If the reflective color where completely identical to the detected color, it would be incredibly difficult to see the object against the background. Instead, the returned colors of any reflected ray are multiplied by 0.8 (after shading) to make the reflected object marginally darker.



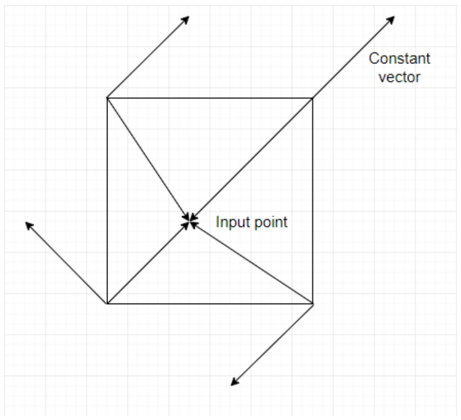


Perlin Noise

To procedurally generate textures for the planets, Perlin noise was implemented[4]. When a texture is created, an array of 512 integers is created, where the first 256 elements are the numbers 0-255 randomly shuffled, and the last 256 elements are in the same order as the first half of the array. A similar array of 8 elements is shown below.

[3, 1, 4, 2, 3, 1, 4, 2]

This array is created to represent a grid of constant vectors. Any possible point is presumed to represent a point on this grid. From a point's position on the grid, certain vectors can be calculated. Four vectors are the constant vectors found at the corners of the grid square the point is in. Given a node on the grid, an integer from the calculated array is computed to % 4, and that integer is used to return one of four possible vectors. The other four vectors are the vectors from the corners of the grid square to the point within the square.



This figure shows the four constant vectors and the four vectors directed towards the point[4].

For each corner, the dot product between the constant vector and the directed vector is found, producing four floating point values. The four values are then interpolated with each other using the following equation, given *a* and *b* as the variables to be interpolated, and *t* as a 'faded' variant of the points x- or y-position within the grid square.

$$a + t * (b - a)$$

The *t* is calculated using the following 'fading' equation, where *p* is the point's x- or y-position.

$$((6 * t - 15) * t + 10) * t * t * t$$

The final value is modified to be between 0 and 1, as is the value of the Perlin noise at that point. Using the texture coordinates calculated in the Shape, that value can then be used to vary how bright or dark a color appears.

Using this method, three different types of texture were created. The first type, OneColorTexture, takes in a single color. Perlin noise is used to vary the darkness of this color, leading to splotching as see on the Moon Sphere in the rendered image.

The second type, TwoColorTexture, takes in two colors as arguments. The Perlin noise is used as a multiplier to vary between the two colors, leading to both colors appearing in a random, yet gradual pattern. This can be most easily seen on the ‘land’ of the Earth Sphere, which varies between a grassy green and a mountainous brown.

The final type, MixedTexture, takes in two Textures of any type, as well as a threshold floating point value. This type determines if the calculated Perlin noise value is above or below the threshold; if it is above the threshold, return the color from the first texture, and the second texture if it is below the threshold. This is how the sharp variance is calculated for the Jupiter and Earth Spheres.

Sources

[1] http://www.opengl-tutorial.org/beginners-tutorials/tutorial-7-model-loading/#reading-the-file
[2] https://en.wikipedia.org/wiki/M%C3%B6ller%E2%80%93Trumbore_intersection_algorithm
[3] https://www.scratchapixel.com/lessons/3d-basic-rendering/introduction-to-shading/reflection-refraction-fresnel
[4] https://rtouti.github.io/graphics/perlin-noise-algorithm