Dieter Grosswiler

CSCI 441 Program 3

Multi-Threading

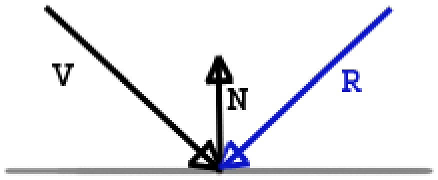
Multi-threading was the first +1 I implemented since it would speed up testing significantly through the development of this program. For this, I used std::async, a kind of high level interface for std::threads that is pretty simple to implement (as long as you don't have to spend hours trying to make the std dependencies work because MinGW sucks). Std::sync takes as argument a callable object, a function for example, and returns a std::future, that will store the result returned by your function or an error message. I store the futures returned by async in a future_vector. And the number of cores on my system is grabbed using the handy std::thread::hardware_concurrency() function. I also use std::atomic to store the count to prevent race conditions (cases where multiple threads are working on the same instance).

Using this information, I create a while loop using the number of cores, then start placing the returned futures in the future_vector. The nested loops running the ray tracing were transformed in to a flat one. Then, using the domain of all the pixels, the x and y values are obtained from the index of the pixel being processed. That way the I can use the atomic count to keep track of the latest pixel to queued for ray tracing. The thread then can communicate by modifying the count since its atomic.

Reflections

The next +1 item I implemented in my Program 3 is reflections. I started with finding the first intersection with a sphere. With this intersection, I perform lighting calculations. In these calculations I use the getReflectiveLighting() function, passing in the intersection information (ray, color, normal). In that function, the reflect vector (R) is calculated using the origin of the ray(known as the velocity vector V), and the normal of the intersection (N). The formula for the reflection vector is as follows:

R = 2*(V dot N)*N – V



This vector is stored in a reflection ray holding the point of intersection as the new origin for the ray as well as the reflect vector for direction. This ray is then cast from the new origin to find the nearest intersection using my castRay() function.

The castRay() function works by running intersection algorithm on all the placed sphere objects. The intersect algorithm stores the distance of each intersection, so then it was as simple as checking for the shortest distance and storing that new Intersection. Then lighting calculations are performed and the color of the intersection is passed back to the main calculateLighting() function and is added to where the first ray intersection hit.

Dieter Grosswiler

Shadows

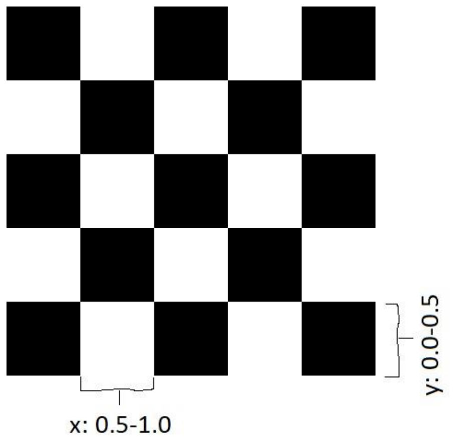
Shadows were implement mostly in the isInShadow() function, returning a boolean. In this function the same calculations are performed as in castRay() to find the nearest intersection of the light ray after the first point of intersection. If it finds an intersection that is greater than the distance from origin to first impact, then it must be a shadow and the returned color is Color(0,0,0).

Procedural Texture: Checkerboard

Checkerboard procedurally generated patter was relatively easy to implement. I added getColorCheckerboard() to my sphere class as an optional "Color" when intersection happens.

This function simply takes in the point of intersection (x,y,z) and casts the values to int. This is important since the values within a range will be even. Then % (mod) 2 == 0 is performed see if any of the 3 values are even (setting them to true if so). Then I check to see if (x xor y xor z), meaning one, and only one of the values are even, then color black, else color white.

The reason the casting to int is important is that it creates ranges of black and white since the z value on the front face will stay relatively constant (say 2 to 2.4) and the x,y values are in an odd range (maybe 1 to 1.5). Then this entire block will be colored black until on of the values becomes even eventually as the rays move across the surface. Below is an example of what I mean in 2 dimensions. The x value will be cast to 1 and y is cast to 0. So over the range [0.5,1.0]x[0.0,0.5], only y is even, so the color will be white.



References

Medium. (2018). Solving multithreaded raytracing issues with C++11 – Foster Brereton – Medium. [online] Available at: https://medium.com/@phostershop/solving-multithreaded-raytracing-issues-with-c-11-7f018ecd76fa [Accessed 26 Apr. 2018].

Scratchapixel.com. (2018). A Minimal Ray-Tracer: Rendering Simple Shapes (Sphere, Cube, Disk, Plane, etc.) (A Minimal Ray-Tracer: Rendering Spheres). [online] Available at: https://www.scratchapixel.com/lessons/3d-basic-rendering/minimal-ray-tracer-rendering-simple-shapes/minimal-ray-tracer-rendering-spheres [Accessed 26 Apr. 2018].

Scratchapixel.com. (2018). Introduction to Shading (Procedural Texturing). [online] Available at: https://www.scratchapixel.com/lessons/3d-basic-rendering/introduction-to-shading/procedural-texturing [Accessed 26 Apr. 2018].