Wendy Chen
CPSC 424
5/11/16

Final Project: GPU Raytracer

Problem Description

In raytracing, a 2D image is produced as output from a 3D input scene. Light behavior is simulated by shooting rays from the camera to all areas of the scene, and then checking if these rays hit any objects or light sources. The color returned to each pixel is an average of the number of rays used to sample the scene.

A number of bottlenecks on the speed of raytracing exist. First, each ray casted must iterate through the entire list of scene objects to check for intersections. A scene that is heavy geometrically (ie. Ten thousands of triangles) will be highly inefficient in render time if no acceleration data structures are used to help with search through space. Next, another major bottleneck is the number of pixels in the image and the number of rays sampled at each pixel. We want to sample a number of jittered rays at each pixel for anti-aliasing; however, the more rays we sample, the longer the render time. Additionally, for a higher resolution image, we need to render an image with more pixels, which will also lengthen the render time.

In this project, I will address parallelizing the pixel sampling process. In a serial program, the program moves from pixel to pixel to render the image. In this project, I will parallelize the process by assigning one pixel to each thread. Thus, with all threads running at the same time, all pixels should finish rendering in the time it takes a serial code to render one pixel.

I have limited the scope of this project to parallelizing only the rendering of pixels to achieve speedup. Adding an acceleration data structure or parallelizing ray sampling would provide significant additional speedup. I leave those two problems as areas of further work.

Parallelization Approach

As discussed before, my parallelization approach was to assign each pixel to each thread. The serial version of the code naively steps through pixels one at a time, going across each row until completion. For an image of size n by m, the serial code would take $O(nm)$ time to step through all pixels.

In my parallelization approach, I utilize CUDA to access the NVIDIA GPU on my Mac. I initialize the grid to be the same dimensions as the size of the desired image. Then, the call to the kernel assigns each thread to the appropriate pixel, according to which row and column the thread resides on.

In order to render the pixel, each thread also needs to know the scene information. In the serial program, the scene was loaded into global memory once. By contrast, my parallelization approach requires that each thread load the scene and then perform raytracing for just one pixel. Each thread can safely write the resulting color into a globally shared array, which is then sent back to the host and then reassembled as an image.

Code Description

In this project, I wrote both a serial C code and a parallelized C code that runs with CUDA. I refactored the serial code from a JavaScript raytracer project from last semester (http://vverovvero.github.io/CPSC-290-index/). I preserved the code logic

while making the structures compatible with C. The main data structures of the program were structs that held scene information and an image array that held floating values for (R,G,B). The scene struct contained pointers to lists of all camera structs, light structs, material structs, and object structs. Object structs were either sphere structs or triangle structs.

The program logic goes as follows: for each pixel, shoot a sample of rays into the scene. (In my program, I shoot nine rays for anti-aliasing purposes). Each ray casted into the scene will then check for intersection with an object. If no object is hit, then the color black is returned. If an object is hit, then I check if the object is visible from a light source, and determine what color the ray is seeing. Light bounces are bounded at a depth of three. After all nine samples return, the values are compounded into a single pixel. A tone mapping function is called when all pixels have returned, in order to ensure that all (R,G,B) values fall between 0 and 255.

Lastly, the image data is written to a png format using the Cairo library. (https://www.cairographics.org/)

The code structure of the parallelized raytracer remains faithful to the structure of the serial raytracer. The main difference is that the render function does not need to step through each pixel. Instead, the render function determines what the row and column numbers of its thread are, and then the function assigns the pixel of the same row and column numbers to that thread.

Results

I used both my serial and parallel programs to render an image of two spheres sitting in a Cornell box. With only twelve objects total, the scene is not geometrically complex, which is acceptable since I am only looking to test speed up in rendering a pixel, not speedup due to accelerating search through space (by using a data structure like a k-d tree, for example). Thus, the following results show render times returned by the serial and parallel programs for the same image, increasing only the number of pixels.

Table 1. Serial and Parallel Render Times for Static Scene of Increasing Resolution

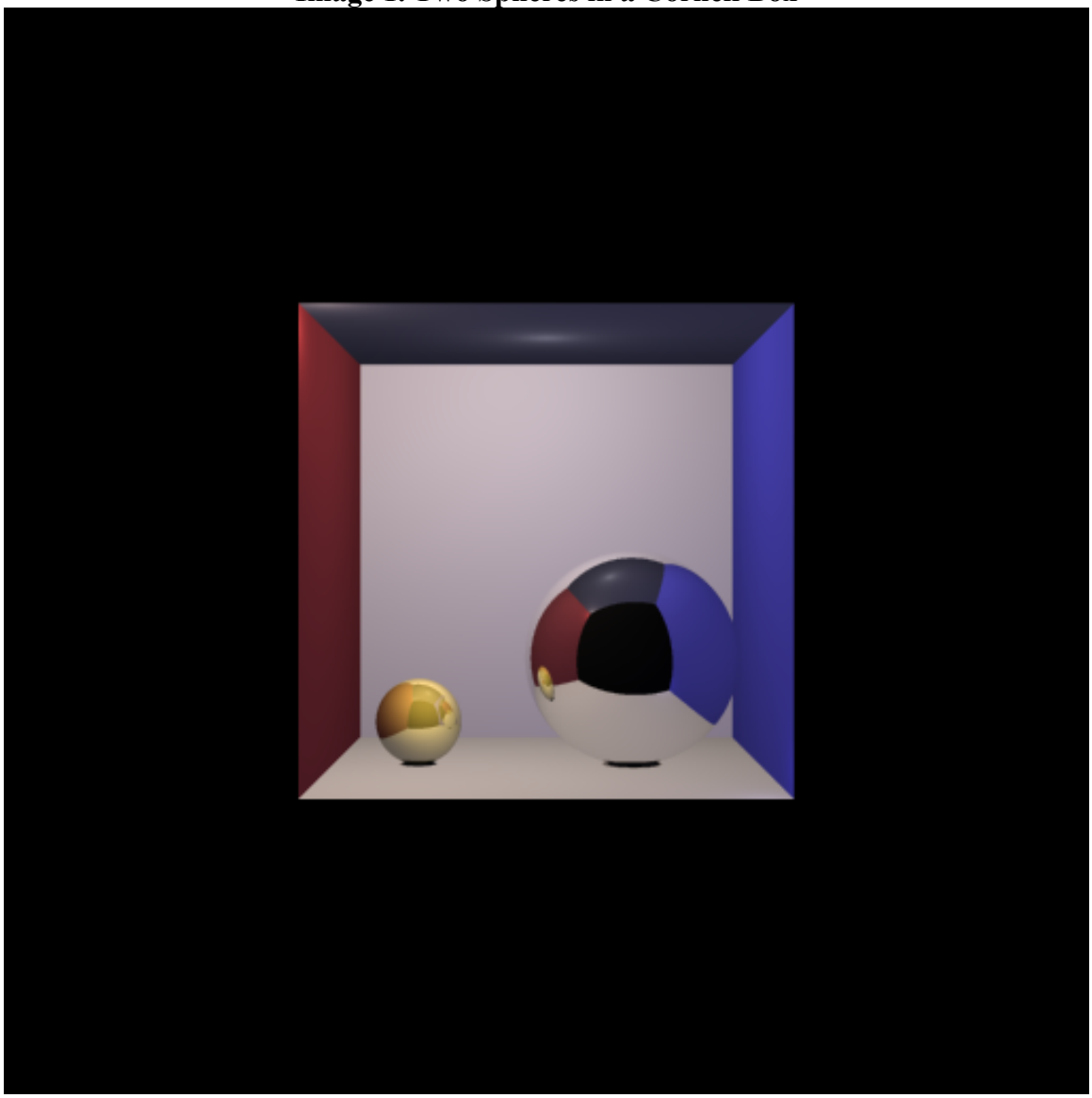
Width (Pixels)	Height (Pixels)	Serial Time (milliseconds)	Parallel Time (milliseconds)	Speedup Factor
256	256	5219.02124	3400.354736	1.53484611
512	512	21529.23389	10011.75977	2.150394575

Table 2. Serial Render Times for Static Scene of Increasing Resolution

Width (Pixels)	Height (Pixels)	Serial Time (milliseconds)
256	256	5219.02124
512	512	21529.23389
1024	1024	86540.99194
2048	2048	342839.0981
4096	4096	1308778.755

Below is the rendered image of the static scene used, shown at a resolution of 512 by 512 pixels:

Image 1. Two Spheres in a Cornell Box



As observed, the parallel code was able to render the image in about half the time that the serial code could. The observed speedup was less than the hypothetical time of rendering all pixels in parallel in the time of rending one pixel serially. The discrepancy is likely due to the overhead of starting up CUDA and transferring data between the CPU and GPU memory.

Additionally, my parallel program was unfortunately unable to render images at resolution 1024 by 1024 and above. This size limitation is likely due to the limit of the thread stack size on my GPU. While I tried changing the stack size, at a certain point, the program became constrained by hardware limitations.

The fact that my parallel code would have required more memory than available on the stack is due to possibly two reasons. One, because of the design of the serial code, the easiest way to modify the code for running in parallel was to have each thread copy the scene into its memory. Each thread making a copy of the scene is highly redundant and wastes memory. Two, because the original JavaScript code was written recursively, my refactored serial and parallel C programs were both recursive as well. CUDA does not seem to support recursion well, as the use of recursion makes CUDA unable to determine how much space to allocate for the stack.

Conclusions & Further Work

As evidenced, while my parallel program achieved speedup in render time, there still remain areas for improvement. The most obvious area would be to reduce the strain on memory. The parallel program should store the scene information in either shared or global memory in order to prevent each thread from needing a copy of the scene on each thread stack. Additionally, the recursion in the program could be changed to iteration. By removing recursion, CUDA would also be able to determine how much space to allocate for the stack.

Another area to parallelize would be ray sampling per pixel. Currently, each thread performs all samples per pixel. In my current implementation, nine ray samples are used to generate the color of a pixel. This process could be parallelized by having each thread simulate the casting of one ray, and then combining the results of nine threads to get the color information for one pixel.

Lastly, an area to improve would be the data structure used to store the scene. Last semester, I implemented a k-d tree that helps make searching through objects in a scene more efficient by grouping objects into appropriate bounding boxes. I did not implement any spatial acceleration data structures in this C program, since spatial acceleration data structures are not related to parallelization of the raytracing problem, although they would improve performance.

Notes: Running the Code

I have attached the code in a zip file. Inside the zip file, there is a serial C program, a parallel C program, and a sample image of what the scene should be rendered as. The code was written to run on Mac. The user must also have the Cairo graphics library installed to run the code. To run the parallel code, the user must have an NVIDIA graphics card.

Inside both codes are #define statements that set the height and width resolution. The user can change these values, so long as they remain factors of 32 (to stay consistent with the block size used in the parallel code).

To make and run the serial program, use the following commands:

```
gcc -std=c99 -o main -L/usr/local/lib/cairo/ -lcairo main.c -
I/usr/local/include/cairo/
./main
```

To make and run the parallel program, use the following commands:

```
make
./main
```