Report:

Kiran Khambhla, Ben Cagan

15418 Final Project: Path Tracing with CUDA

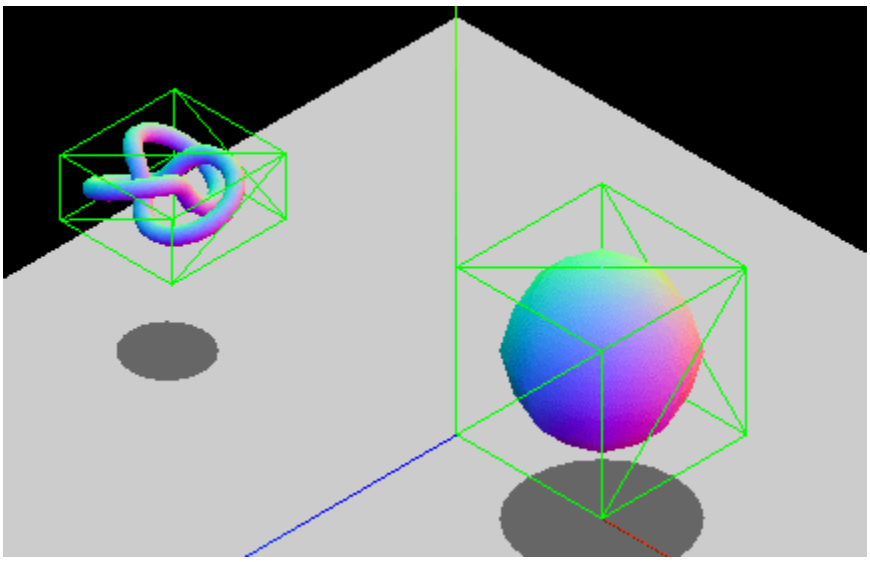
SUMMARY:

We implemented a path tracer for 3D rendering sequentially on the CPU, and using CUDA on the GPU and compared the performance of the two implementations.

BACKGROUND:

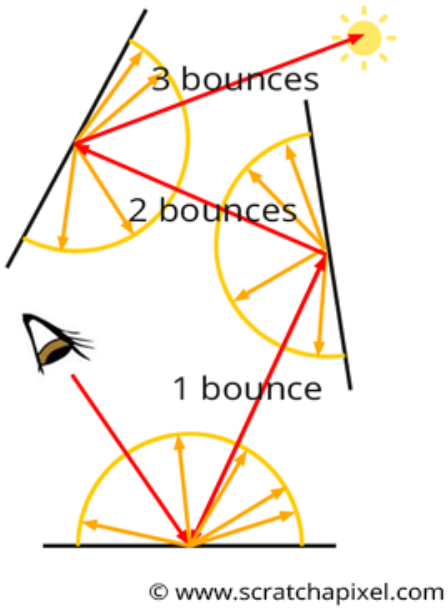
The path tracing algorithm is a fairly simple way of rendering the behavior of light in 3D space. Rather than calculating the behavior of rays directly from a light source, rays are cast from the camera's perspective and travel until they intersect with an object in space and eventually to the light source. When a ray hits an object in space, we draw a new ray from the surface normal of the object with a random angle. This continues until the ray misses or it reaches its maximum bounce depth.

The key data structures are a Scene, which contains a list of the Objects in the scene and performs the render, a Camera, which is in charge of emitting rays, and Hits which store the information about a rays hit for detecting misses and color calculation. An important addition we made for optimization, was using bounding boxes for Objects to detect possible intersections before performing the actual intersection check.



Examples of Bounding Boxes in 3D (From MDN Docs)

The algorithm simply takes in a list of Objects in 3D space, a camera position, and the maximum bounce depth and outputs a render of the scene.
The computationally expensive part is having to take multiple samples of the ray's path, using an average as the actual pixel's color in the render. This has to be done for every pixel on the Camera, so a naive implementation is dreadfully slow. The upside is the rays have no dependencies between each other, so their paths can be calculated in parallel without any communication required. Moreover, each ray corresponds to a pixel in the image, and our image is just a large array of color values, so our rays, hits, and output are completely data-parallel. Since each ray performs almost exactly the same, the algorithm lends itself very well to SIMD execution.



APPROACH:

Our serial implementation follows the same structure as defined above. First, the objects in our Scene are initialized with their coordinates in 3D as well as their Material (emitted and albedo values), stored in sceneObjs. A Scene is rendered by going pixel by pixel across the image, shooting a ray from the camera, detecting if there was an intersection with any Object's bounding box. Then we make sure the intersection actually happens, calculate the direction of the next ray and continue bouncing around the image until we either miss all objects or reach our (predetermined) bounce limit. Then the color is calculated recursively, using this equation:

$$RayColor(t) = Hit.emitted + Hit.albedo\ (RayColor(t-1))$$

(A Hit Object is initialized when an intersection is detected and stores the object's Material values)

We repeat the ray calculations for a certain number of samples, and then average them out, passing the final color value into the image.

Our CUDA version works similarly. The first step is the same. Initialize Objects with their Materials in the Scene, then pass this Scene to our CUDA code which then initializes the Camera and arrays for the Rays, Hits (the index of a ray in the array corresponds to its index in the hit array), Objects, and output image.

Our first step in CUDA is to initialize rays in parallel. We create Ray objects which are then placed in our Ray array and index corresponding to their pixel index. This task is done with (8,8) threads per block and ((cam.resX + blockSize2d.x - 1) / blockSize2d.x, (cam.resY + blockSize2d.y - 1) / blockSize2d.y) blocks. Then, we see if any of our rays have intersections in parallel, then we calculate the color of each array in parallel depending on whether or not each ray had an intersection. These both are done with cam.resX * cam.resY rounded to the nearest multiple of 128 blocks, and 128 threads per block. The mapping was pretty straightforward, each ray/pixel is just mapped to a thread. This is repeated across a bunch of samples, each iteration's color being added to the image array, and then finally we compute the average across all samples, writing it to a final image array, which is copied over to the image in the Scene object (sent back to the CPU). These are both done in parallel as well.

We used the GLM library for rendering (and CUDA obviously), but besides that all our libraries were written ourselves, since the CUDA linker was being really fussy about __host__ functions being called from __device__ functions. Our implementation didn't change that much, the CUDA linker was the biggest hurdle to get over, as it isn't a big fan of OOP so we ended up having to rewrite a lot of functions for our CUDA version to both dodge memory errors and appease the linker.

Our only optimization that we were able to add was using BBoxes for fast intersection calculations, but we tried to add Ray trimming, ignoring Rays that had already missed and dynamically sizing the block size depending on how many Rays were still bouncing, which we weren't able to get in time.

RESULTS:

Final Project Compared to Initial Goals: We were able to successfully create a real-time path tracer that can be run on CPU or on a GPU using CUDA, with custom scenes and physical accurate lighting. While not a part of the original plan, we weren't able to implement a movable camera, however, the path tracer updates the window whenever a new frame is complete. We were able to add implicit shape support, and lambertian materials, but were unable to add mesh, reflection, refraction, or BVH acceleration support, due to time constraints, and unexpected challenges, such as getting OpenGL to work on both of our VisualStudio set ups, and getting VisualStudio to properly compile our CUDA code.

Performance Measurements: We measured speedup by comparing the time to render a full serial frame on a given scene, compared to the same scene on the GPU running using CUDA, with the frame measurement consisting of everything, from initializing the CUDA data, to drawing the scene and refreshing the GPU's frame buffers. We also tested each scene with

5,10,and 15 maximum recursive bounces, in order to judge how the parallelization handles larger differences in recursion length from one ray to another, or in particular, how a given thread running longer than others due to more ray bounces affects the efficiency of a total warp.

In addition to the direct CPU to GPU comparison, we also timed particular parts of the CUDA code to judge where the bottlenecks lied. In particular, we judged the creation of rays at the start of each sample, the test of if and where a ray intersects with the scene, the calculation of the color value at that given intersection, the time spent waiting for all active threads to synchronize, and the time to load data from the GPU to the CPU at the end.

Scene setups: In addition to varying the maximum allowed recursive depth, we also tested the path tracer on 5 different scenes: A cornell box (which helped to test visually apparent bounce lighting, and heavy regions of high recursive depth), a cornell box containing a sphere (which helped at variation to the above scene), a scene consisting of a plane of 169 (13*13) cubes, and a scene consisting of 169 spheres, in either case fully reflective (lambertian material with a white color) and all emitting light, in order to test scenes with a larger number of objects, and a fully random scene consisting of 50 randomly placed (in the area visible to the camera) cubes and spheres, all emitting light, to test a fully random scene with heavy variation across screen regions.

Graphs of execute time are below

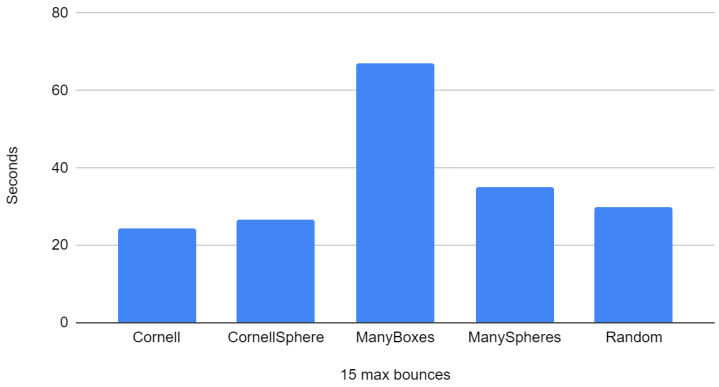
Effects of scene structure and problem size on performance: Beyond just varying problem sizes (number of objects on screen), varying the layout of the objects, and the maximal depth of ray bounces, led to heavy variation between the performance of different scenes. In particular, ManyBoxes and ManySpheres saw the lowest level of speed up compared to the cornell boxes, as the cornell boxes consisted on average of more hits for initial ray casts, so on average a given block of the screen was more likely to have more of its rays active from the start. This leads to more threads in a given warp seeing active calculations in later bounces, leading to a more efficient usage of the available threads, noting that in the serial set up, there is one thread, so it is always active for each ray. The ManyObject scenes also were less likely to see recursive bounces, than the inside of the close cornell box, leading to more chances of rays being terminated earlier. As we were unable to refactor the ray ordering to only calculate active rays in later iterations, we were unable to optimize this behavior, which explains the difference in performance. Limiting the recursive ray depth saw a notable increase in performance, and higher speedups, as not only were less calculations completed, but there was less chance for heavy ray depth variance (1-5 vs 1-15) and as such, more efficient usage of all warps. In particular, speedup for 15 threads vs 5 threads for Cornell was 77 and 142, and for ManyBoxes was 20 to 54 times. This is very similar for either scene, though many boxes sees a slightly higher difference, 2.7x vs 1.8x, and this is likely since the distribution in average ray depth is more uniform over the Cornell box, than the highly varied ManyBoxes scene. In general, increasing ray depth leads to better performance, but worse speedup, across all scenes, while the scene itself leads to a more significant difference in speedup.

CUDA Execution Breakdown, and Bottleneck Analysis: Looking at the specific breakdowns of GPU time across all the different scenes, intersection calculation, and loading consistently was not the bottleneck, with both taking up a small fraction of the total runtime. Loading likely benefited from the small frame size (720p image with 24bit RGB), and high bandwidth of the 3070 GPU, while intersection benefitted from, on average, a small number of objects per scene, consistency across all threads (limited room for the intersection checks for rays to diverge, most divergences lasted only a few instructions), and less memory accesses than the color calculations. Ray creation itself was moderately costly, but certainly not a bottleneck, consistently 10 times longer than intersection. Ray creation is the main instance of matrix multiplication being used in the program, as well as random number generation (also occurs in color calculations, but not in intersection testing). Moreover, ray creation also accesses memory more often than intersection testing. In turn, this likely explains the increased cost, but the two portions of the code share consistency along all threads, so neither are a bottleneck.

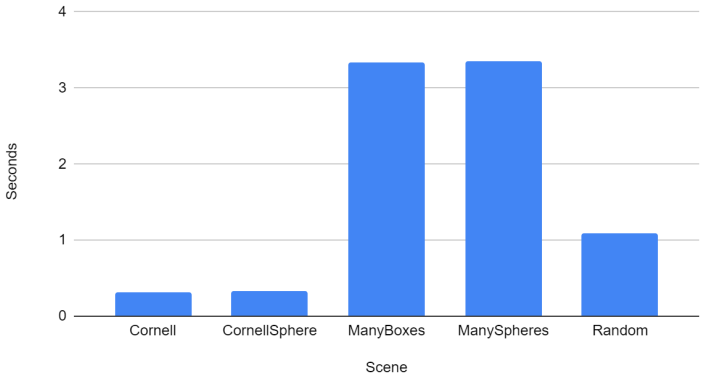
Depending on the scene the bottleneck is either the color calculation, or the synchronization. For all scenes, color calculation has a similar cost, 0.1-0.2 seconds. This likely owes to additional vector math operations, randomization (as mentioned above), and large memory accesses (storing multiple color values per ray). However, while in some scenes this is the bottleneck, taking up slightly more time than synchronization (Cornell Boxes and Random), in others, specifically the ManyObjects scenes, synchronization is the bottleneck, taking far longer,1-3 seconds. This likely owes to the previously discussed difference in divergence caused by a higher variation in maximum ray depth across the screen in the ManyObjects scenes compared to the others. Random acts similar to the cornell box, as even though it has a high degree of variation, many objects are far closer to the camera compared to the ManyObjects scenes, leading to more rays having limited depth in a consistent manner. Had we finish implementing the system which refactors rays into those that are still active, and those that have terminated, the difference in runtime between threads in a given warp would have been far smaller, leading to this bottleneck in the synchronization being less common, and color calculation being the main bottleneck.

Evaluation of Choice of GPU: Even with limited optimizations (No reordering of rays during each iteration, not parallel BVH tree implementation), we still saw anywhere between 40-130 times speedup across different scenes. In addition, the GPU is highly suited to the ray tracing problem, and graphics problems in general, allowing for a large number of threads to be computed in parallel, or in the case of path tracing, a large number of rays from the same frame to be computed at once. As such, we feel that the choice of the GPU and CUDA for this project was highly suited.

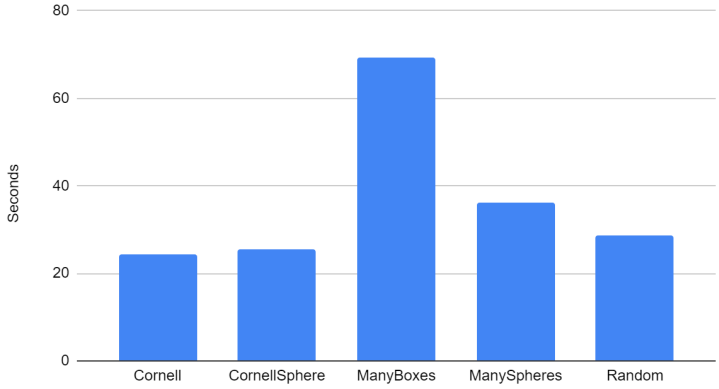
15 max bounces (Serial)



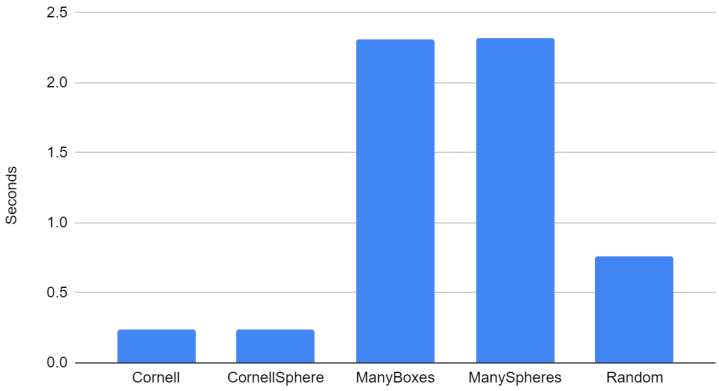
15 max bounces (CUDA)



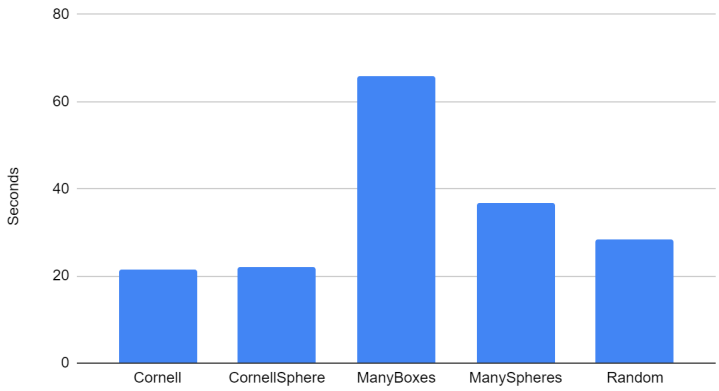
10 max bounces (Serial)



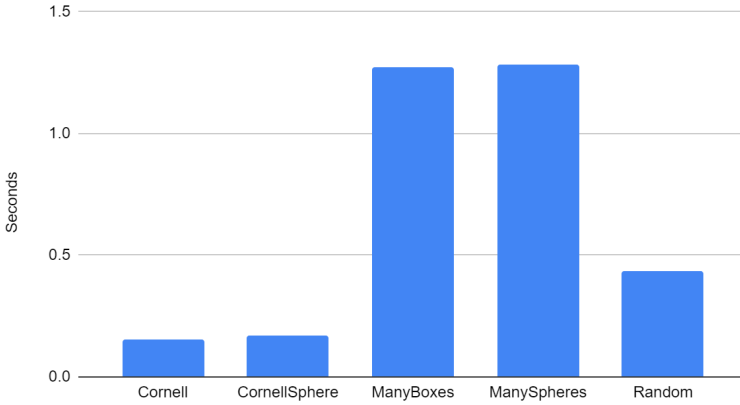
10 max bounces (CUDA)



5 max bounces (Serial)



5 max bounces (CUDA)



Performance: Average of 3 iterations of each of the following (CUDA) and 1 for sequential
All scenes rendered at 720p with 10 samples per pixel
Random scene was 50 random spheres and cubes. While not the same for 15,10,and 5 bounces, the sequential and parallel
runs for a given number of bounces used the same randomized scene
Specifications:

- Intel i7-10870H CPU @ 2.20GHz
- 32 GB RAM
- Windows 10
- RTX 3070 MaxQ (5120 cores, 8 GB,384.0 GB/s bandwidth)
- Visual Studio 2019 Compiled in Performance Mode

15 max bounces

Serial:

Cornell: 24.5332
CornellSphere: 26.5313
ManyBoxes: 67.1994
ManySpheres: 35.0251
Random: 29.804

Parallel:

Cornell (Average)
Ray: 0.0175426667 Intersection: 0.001665 Sync: 0.269562333 Loading: 0.000638 Delta: 0.3173857

CornellSphere (Average)
Ray: 0.008282 Intersection: 0.001854 Sync: 0.0130023333 Color: 0.286132 Loading: 0.00022 Delta: 0.333371

ManyBoxes (Average)
Ray: 0.0219143 Intersection: 0.002136 Color: 0.349326 Sync: 2.955005 Loading: 0.000598 Delta: 3.342763

ManySpheres (Average)
Ray: 0.021529 Intersection: 0.002286 Color: 0.337645 Sync: 2.984938 Loading: 0.000574 Delta: 3.35871333

Random (Average)
Ray 0.018761 Intersection 0.002003 Color 0.283469 Sync 0.765683 loading 0.000597 Delta 1.08386

10 max bounces

Serial:

Cornell: 24.4099
CornellSphere: 25.5279
ManyBoxes: 69.2385
ManySpheres: 36.2354
Random: 28.7541

Parallel:

Cornell (Average)
Ray: 0.043203 Intersection: 0.001164 Color: 0.191679 Sync: 0.012923 Loading: 0.000659 Delta: 0.236976

CornellSphere (Average)
Ray: 0.018356 Intersection: 0.001238 Color: 0.220429 Sync: 0.056958 Loading: 0.000627 Delta: 0.238796

ManyBoxes (Average)
Ray: 0.019905 Intersection: 0.001576 Color: 0.200461 Sync: 2.078025 Loading: 0.000582 Delta: 2.312443

ManySpheres (Average)
Ray: 0.018702 Intersection: 0.001635 Color: 0.199128 Sync: 2.086556 Loading: 0.000589 Delta: 2.319957

Random (Average)
Ray: 0.01891 Intersection: 0.001327 Color: 0.195187 Sync: 0.012315 Loading: 0.000737 Delta: 0.759494

5 max bounces

Serial:

Cornell: 21.5326
CornellSphere: 22.0443
ManyBoxes: 65.8896
ManySpheres: 36.7965
Random: 28.3933

Parallel:

Cornell (Average)
Ray: 0.019055 Intersection: 0.000653 Color: 0.105474 Sync 0.013101 Loading: 0.000649 Delta: 0.151191

CornellSphere (Average)
Ray: 0.02058 Intersection: 0.000791 Color: 0.119082 Sync: 0.012618 Loading: 0.000630 Delta: 0.168873

ManyBoxes (Average)
Ray: 0.019396 Intersection: 0.000830 Color: 0.10816 Sync: 1.134939 Loading: 0.000707 Delta: 1.273317

ManySpheres (Average)
Ray: 0.016246 Intersection: 0.000989 Color: 0.110994 Sync: 1.138806 Loading: 0.000640 Delta: 1.283513

Random (Average)
Ray: 0.019181 Intersection: 0.000777 Color: 0.107329 Sync: 0.293283 Loading: 0.000785 Delta: 0.434038

REFERENCES: Please provide a list of references used in the project.

[Scratchapixel Path Tracing]
[Fast BBox Intersection Calculations]

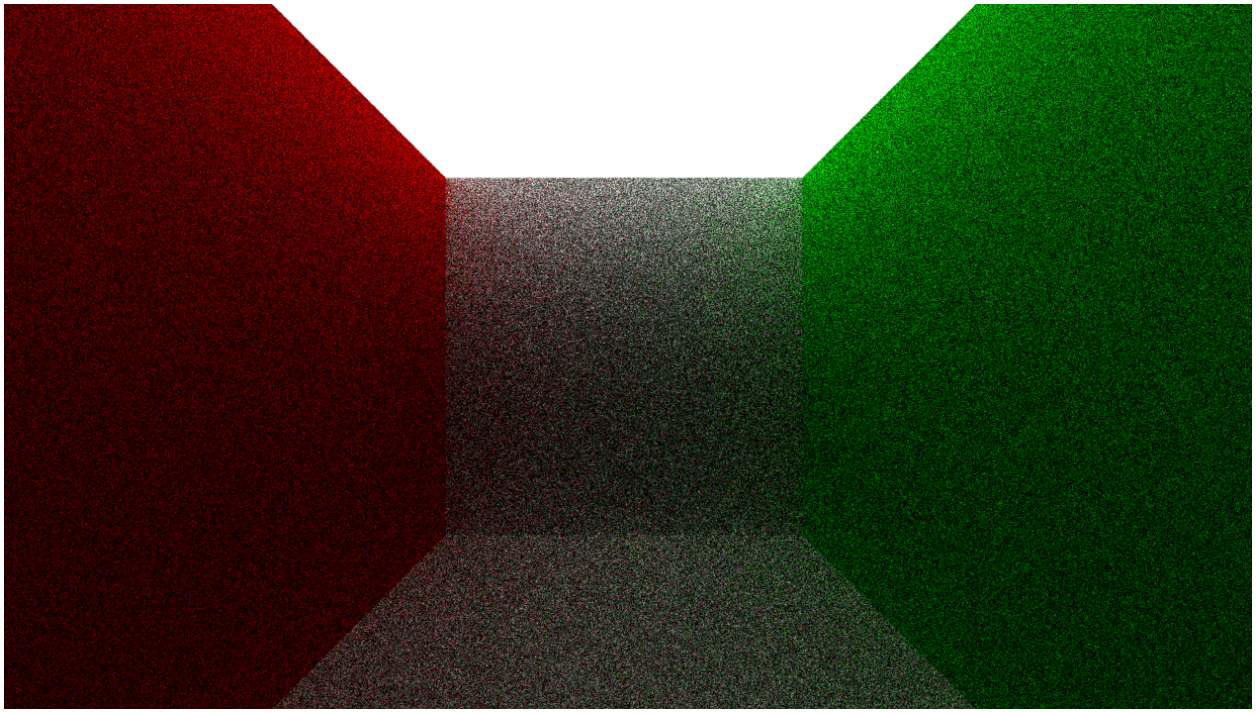
LIST OF WORK BY EACH STUDENT, AND DISTRIBUTION OF TOTAL CREDIT: Please list the work performed by each partner. Given that you worked in a group, how should the total credit for the project be distributed amongst the participants? (e.g., 50%-50%, 60%-40%, etc.) If you do not feel comfortable placing this information on a public web page, it is okay to include it only on the version that you submit via Gradescope.
Your final report should be included in your Git repo as a separate TeX / Word / etc file + a PDF.

GLM integration (creating the actual render): Ben
Camera, Scene, Ray, Hit Definitions: Ben
Vector, Color Operations and Definitions: Ben
Creation of Testing Renders, Data Collection: Ben
Naive Render Method: Ben/Kiran
Object Classes, Intersection Calculations: Kiran
CUDA Ray Emission, Intersection Computation, Color Calculation: Kiran

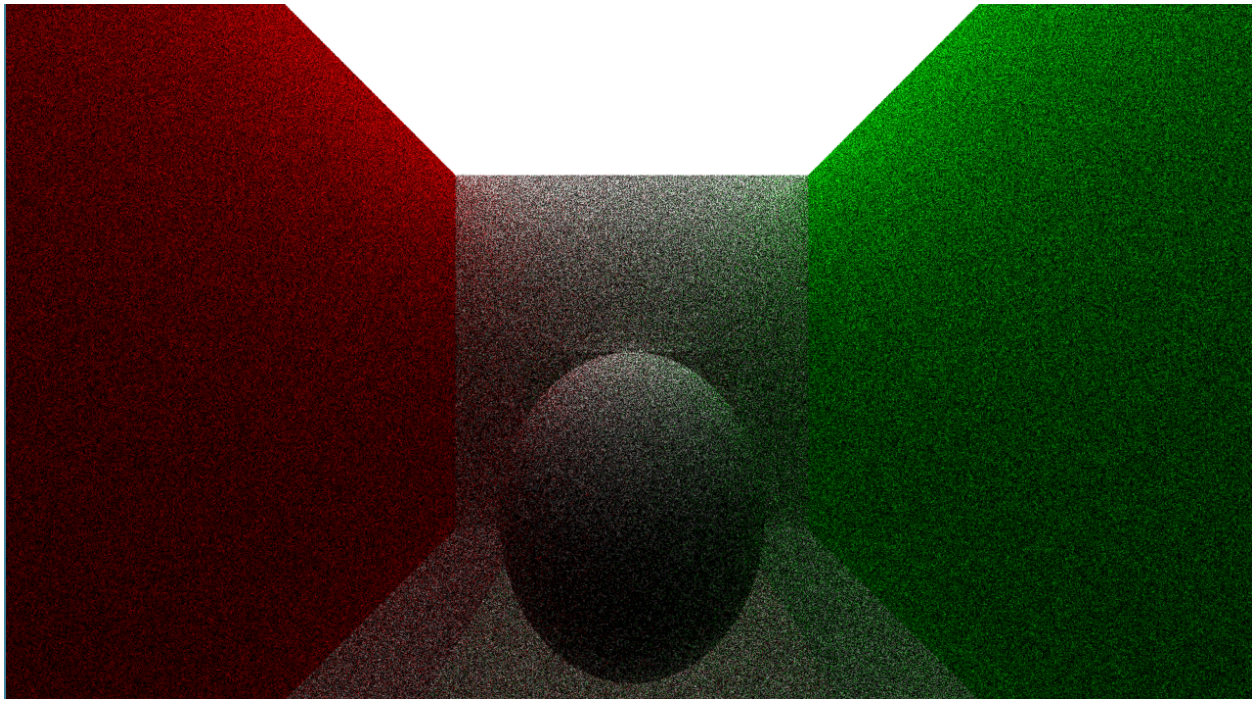
50/50

RENDERS (15 maximum Bounces, 720p render):

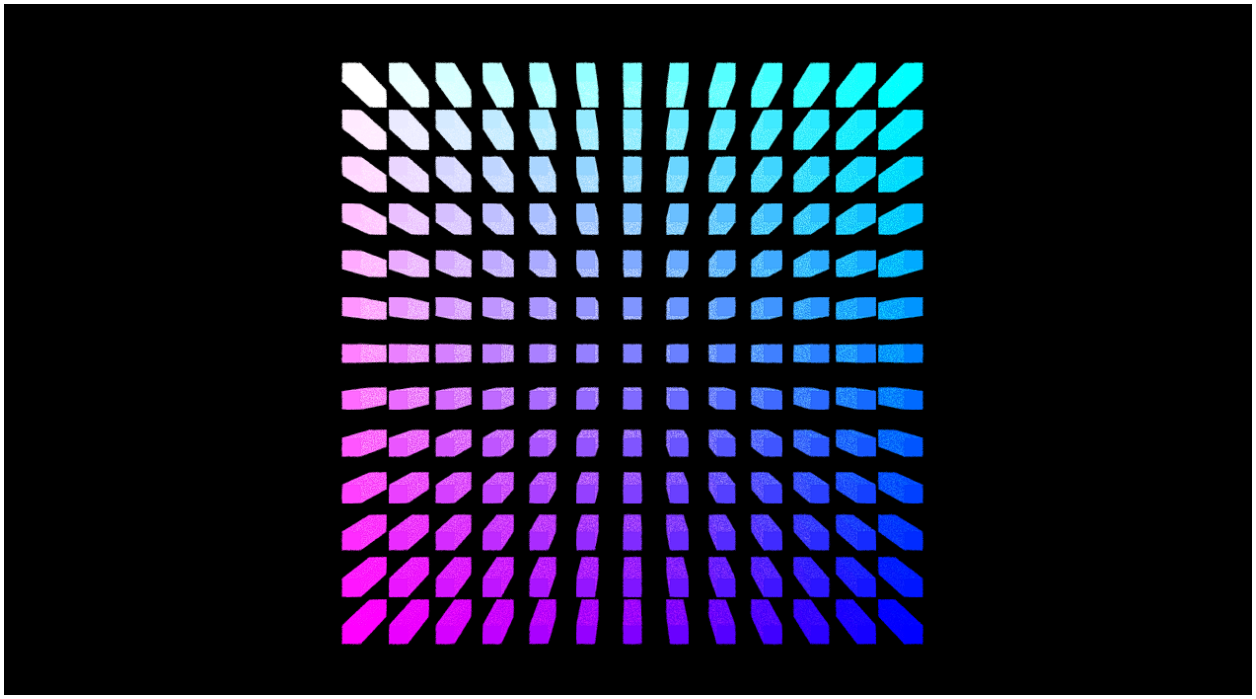
Cornell Box:



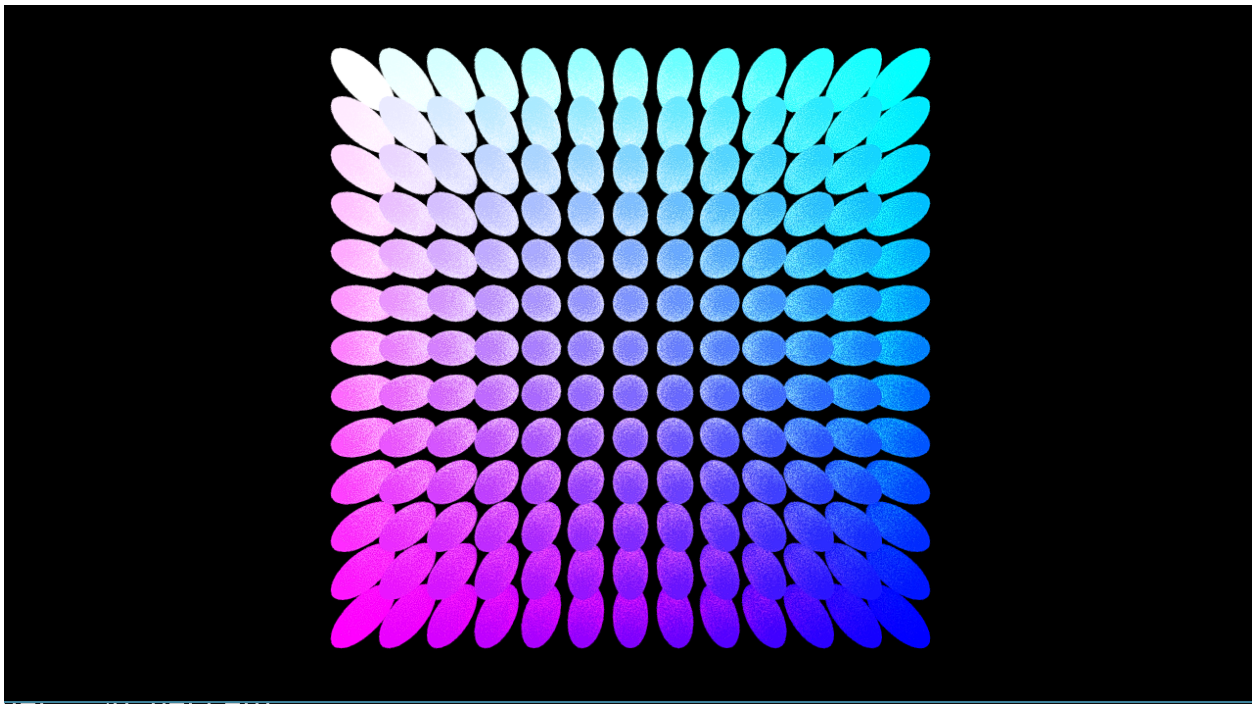
Cornell Box with Sphere:



Many Boxes:



Many Spheres:



Random: