1 INTRODUCTION

All raytracing implementations for this project make use of GPU.js, a JavaScript library built on top of WebGL for GPGPU processing in the web browser. A notable feature of this library is for the user to specify whether to run the kernel function with a GPU or CPU. While JavaScript is inherently single-threaded, the kernel function allows parallel execution of threads to take place. This makes it suitable for implementing raytracers, which benefits greatly from parallel processing.

2 IMPLEMENTATION

Raytracing is a technique used to render images by firing rays backwards from the camera through the viewing plane and into the scene. The ray is then checked for intersections with objects present in the scene. When the ray intersects an object, the surface color of the object which would be 'observed' by the ray is evaluated.

There are various ways to evaluate the color of an object's surface. For this project, a simple shading algorithm is used. In this approach, the surface color of the object comprises the ambient color, the diffuse color and the specular color, and is evaluated using the equation shown in Figure 2.1.

$$S = k_{amb}C + \sum_{m \in lights} \left(k_{lamb}(L_m \cdot N)\, C + k_{spec}\, S_{reflect}\right)$$

Figure 2.1: *The above equation is used to evaluate the surface color S of an object, where k_{amb} is the ambient reflection constant, k_{lamb} is the Lambertian reflection constant, k_{spec} is the specular reflection constant, L_m is the direction vector from point on surface to light source m, N is the normal at the point on surface, $S_{reflect}$ is the surface color of an reflection object, and C is the color of the object. If there is no reflection object, the specular color will not be considered.*

Shadows are also implemented. Hence, when an object is blocked from the light by another object, only its ambient color would be observed. While it is possible to render complicated images using raytracing, the raytracers for this project are limited to rendering spheres.

Two raytracers have been implemented. One is a naïve parallel raytracer and the other is an optimized version.

A naïve parallel raytracer

The naïve parallel raytracer is a very simple raytracer which makes use of data parallelism.

Our computer screen can be treated as a viewing plane that lies between between the camera and the scene. For the naïve raytracer, this viewing plane is split into atomic partitions, whereby each partition consists of a single pixel. A thread will be created and assigned to each pixel. Each thread will evaluate the color to be displayed by its pixel by considering only the ray which passes through it from the camera.

This coarse-grained approach of data parallelism minimizes any communication overhead that is incurred since the execution of a thread is independent from other threads. Communication is only required for initial passing of data from master thread to slave threads, and sending of results from slave threads back to the master thread.

Such parallelism is ideal if all threads were load balanced. Load balancing refers to the distribution of equal amounts of work amongst the threads, such that the total time threads spend idling is minimized.

However, if some threads were to have a significantly higher load than the rest, the performance of the program suffers. A kernel function can only finish running after all of its threads have finished and the slowest thread will determine the overall performance of the kernel function.

This is the case for the naïve parallel raytracer since it does not consider load balancing. Not all threads will have rays which intersect objects. For such threads, they will remain idle and wait for the other threads to finish running their tracing.

Despite this oversight, we still observe reasonable performance in the naïve parallel raytracer since the scene objects are restricted to spheres, which are computationally easy to render. If more complicated shapes, such as multifaceted objects, were to be rendered, there will be greater load imbalance amongst

threads and this may be enough to impair the performance of the program.

Optimized parallel raytracer

The optimized parallel raytracer seeks to achieve load balancing amongst threads. While perfect load balancing is impossible, the program does its best to minimize the number of idle threads at any point in time. The main differences between the optimized raytracer and the naïve version are the distribution of work amongst threads and preprocessing of data by the optimized raytracer; the raytracing methodology remains mostly the same.

To achieve load balancing, the optimized raytracer only traces rays which has a high probability of intersecting an object. It first splits the viewing plane into view boxes and finds out which are the active view boxes, i.e. view boxes from which objects can be observed. This is computationally light compared to raytracing from individual pixels. The actual raytracing is only limited to pixels inside active view boxes.

Splitting of viewing plane

In the naïve raytracer, the viewing plane is split into pixels, and each pixel is assigned to one thread. For the optimized raytracer, the viewing plane is split into view boxes, as seen in Figure 2.2.

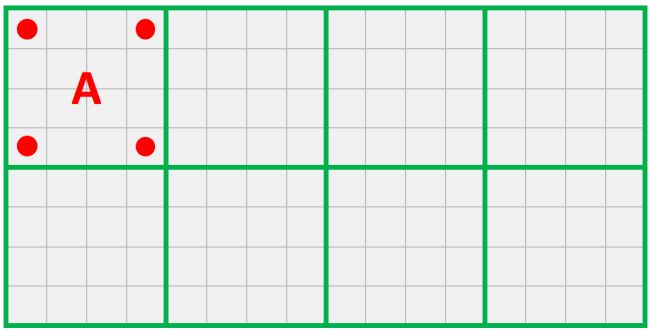

Figure 2.2: *This example shows a 16 x 8 viewing plane split into 8 view boxes, each containing 16 pixels. For view box A, the pixels marked by red circles are the boundary pixels, and rays which pass through them are called boundary rays. The projection of these four boundary rays into the scene will create a trapezoid with infinite depth. Any object that is observable from A must at least overlap with the trapezoid.*

Each view box consists of a two-dimensional array of pixels and is assigned to a single thread. Each thread will determine if the scene that is observable from a view box contains in it any objects. If so, that view box is deemed as active.

Axis-aligned bounding boxes

Knowing the boundary rays of the view box allows us to map the part of the scene which is observed through it into a trapezoid of infinite depth (assuming the depth of the scene is unbounded). We can iterate through the objects and find out if any of them overlaps with the trapezoid. If so, we can conclude the view box to be active.

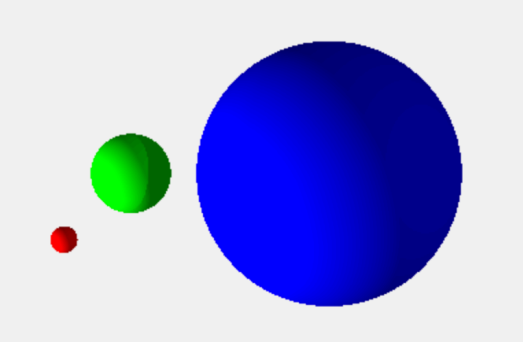

Figure 2.3: *An image frame that was rendered by the optimized parallel raytracer.*

Figure 2.4: *The axis-aligned bounding boxes that were generated from the scene shown in Figure 2.3.*

Without any preprocessing, checking whether an object overlaps the trapezoid in 3D space is computationally expensive, especially in cases where the objects are multifaceted. As such, the optimized raytracer first evaluates the bounding boxes for all objects. The bounding box of an object is a box with the smallest volume within which all points for that object lie. The bounding box is deemed a good enough estimate for the space taken by an object, and is checked for overlap with the trapezoid instead.

All bounding boxes evaluated are axis aligned, i.e. the edges of the box are aligned with the x, y and z coordinate axes of the scene. This enables us to reduce the problem of determining whether the bounding box overlaps with the trapezoid in 3D space into a problem of determining whether two rectangles overlap in two dimensional space. This is much less expensive computationally.

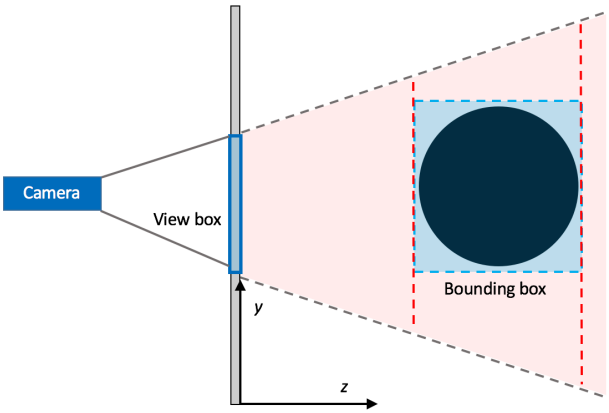

Figure 2.5: *To check if the bounding box (highlight blue) overlaps with the trapezoid (highlighted pink) that is encapsulated by the boundary rays of the view box, we can check for overlaps between the xy-planes of the bounding box with the planes of the trapezoid at similar depth (shown as red dotted lines). If there is at least one overlap, we can conclude that the bounding box will be observed through the view box.*

Limit raytracing to active view boxes

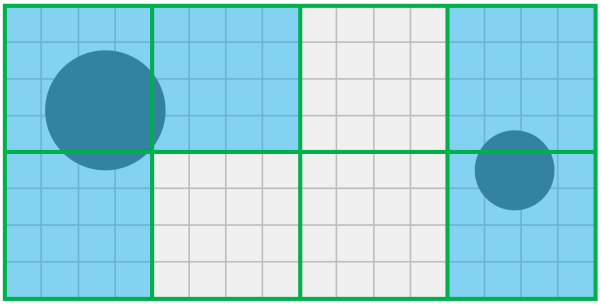

Figure 2.6: *Active view boxes are view boxes from which objects can be observed. They are colored blue in the above diagram.*

In the naïve parallel raytracer, all pixels have rays passing through them traced. This is considered inefficient given that the rays of some pixels may not intersect with any objects, and thus the threads for these pixels have much lighter loads.

To alleviate load imbalance of threads, the optimized parallel raytracer limits raytracing to active view boxes, i.e. view boxes through which at least one object can be observed. This reduces the number of threads that are idle during the execution of the kernel function. For instance, in the example presented in Figure 2.6, the naïve and optimum raytracers will create 128 and 80 threads respectively to render the same image.

Implementation

Naïve parallel raytracer

The source for the naïve parallel raytracer can be found at:
http://www.comp.nus.edu.sg/~wenqiwu/naive_rt.html
You may use the keys W, S, A, D and arrow keys to move/rotate the camera for this raytracer.

Optimized parallel raytracer

The source for the optimized parallel raytracer can be found at:
http://www.comp.nus.edu.sg/~wenqiwu/optimized_rt.html

Naïve parallel raytracer with alternate shading

Another raytracer, which utilizes a different shading algorithm, has also been implemented. There may be some inaccuracies for the specular colors of objects. Its source can be found at:
http://www.comp.nus.edu.sg/~wenqiwu/naive_rt2.html

3 PERFORMANCE

Two performance measures are used to gauge the efficiency of the raytracers for this project. They are the duration it takes to render a single image of the scene, and the frame rate per second of an image that is being rendered repeatedly.

In this project, all experiments are performed using Google Chrome, on a computer with an i5-4258U CPU and an integrated Intel Iris 5100 GPU, unless stated otherwise.

An experiment was carried out to determine whether the number of bounces of a ray takes affects the performance of our raytracers. The results are presented in Figure 3.1 and Figure 3.2. From Figure 3.2, we see that the frame rate for GPU rendering on the optimized raytracer is 0, since the frame rendering took too long.

Hence, another experiment was performed to obtain the time taken by the optimized raytracer to generate a single frame, with the results presented in Figure 3.3.

No of bounces	3	8	16	32	64
GPU frame rate (avg)	0	0	0	0	0
CPU frame rate (avg)	8	6	4	3	2

Figure 3.1: *Average frame rates of naïve parallel raytracer for varying number of bounces of rays.*

No of bounces	3	8	16	32	64
GPU frame rate (avg)	60	47	34	20	10
CPU frame rate (avg)	8	4	5	3	2

Figure 3.2: *Average frame rates of optimized parallel raytracer for varying number of bounces of rays.*

No of bounces	3	8	16	32	64
GPU time taken / ms	1464	1493	1632	1602	1778
CPU time taken / ms	190	208	246	342	515

Figure 3.3: *Average frame rates of naïve parallel raytracer for varying number of bounces of rays.*

We see that for both the naïve and optimized raytracers, time taken to render a frame increases with maximum number of ray bounces. This is probably due to the increased workload for some of the threads with greater number of bounces taken by rays. Since the kernel function can only return once all threads have finished, the increased workload for some threads will lengthen the runtime of the entire kernel function.

Another experiment was carried out to find out whether the number of objects in the scene will affect the performance of our raytracers. The results are shown in Figure 3.4 and Figure 3.5. Again, we see that the frame rate for GPU rendering on the optimized raytracer is 0, since the frame rendering took too long. Another experiment was performed to obtain the time taken by the optimized raytracer to generate a single frame, with the results present in Figure 3.6.

No of spherical objects	2	4	8	16	32
GPU frame rate (avg)	60	60	50	30	15
CPU frame rate (avg)	12	11	9	7	4

Figure 3.3: *Average frame rates of naïve parallel raytracer for varying number of spherical objects in scene.*

No of spherical objects	2	4	8	16	32
GPU frame rate (avg)	0	0	0	0	0
CPU frame rate (avg)	12	12	11	11	10

Figure 3.4: *Average frame rates of optimized parallel raytracer for varying number of spherical objects in scene.*

No of spherical objects	2	4	8	16	32
GPU time taken / ms	1554	1551	1447	1524	1555
CPU time taken / ms	120	128	131	136	142

Figure 3.5: *Time taken for optimized parallel raytracer to render a single frame for varying number of spherical objects in scene.*

For both the naïve and optimized raytracer, we observe that the time taken to render a single frame increases with number of objects to be rendered, for both CPU and GPU. This is expected since an increase in number of objects leads to higher occurrence of hits for rays. This increases the overall load of pixels for the naïve raytracer. For the optimized raytracer, the number of active view boxes to have their pixel colors evaluated also increases.

We see that GPU performance on the optimized parallel raytracer is dismal compared to that on the naïve version, despite its optimization. Benefits of this optimization can only be observed for the rendering of complex objects. However, for this project, all objects are spheres, and are not computationally expensive to render. Calculating whether a bounding box overlaps with a view box scene is not that much computationally lighter than evaluating a ray-sphere intersection.

Another reason for the degraded performance is that the optimized parallel raytracer utilizes multiple kernel functions. This leads to more switching between processors when running the raytracing on GPU, and incurs additional overhead.

4 HARDWARE

Various CPUs and GPUs were used to run the raytracers to determine if the hardware has a significant impact on the performance of the raytracers.

	Frame rate (avg)	Time taken to render single frame / ms
2.4 GHz i5-4258U	7	146
3.4 GHz i3-3240	7	151
Intel Iris 5100	30	35
AMD R9 280X	55	21

Figure 4.1: *Average frame rate and time taken to render a single frame by the naïve parallel raytracer for various hardware. There are 16 spherical objects in the scene and number of ray bounces is capped to 3.*

	Frame rate (avg)	Time taken to render single frame / ms
2.4 GHz i5-4258U	11	136
3.4 GHz i3-3240	3	420
Intel Iris 5100	0	1524
AMD R9 280X	3	341

Figure 4.2: *Average frame rate and time taken to render a single frame by the optimized parallel raytracer for various hardware. There are 16 spherical objects in the scene and number of ray bounces is capped to 3.*

Figures 4.1 and 4.2 show the tabulated results of the raytracers on various hardware. Though the i3-3240 has a faster clock speed, the i5-4258U actually offers a better performance here.

	GFLOPs
Intel Iris 5100	823
AMD R9 280X	3315

Figure 4.3: *FLOP performances of graphics processors*

In terms of FLOPs, the AMD R9 280X is around 4 times faster than the integrated Intel Iris 5100 (see Figure 4.3). We see that the AMD R9 280X offers suboptimal speedup (around 1.5 times faster) for the naïve parallel raytracer, and near-linear speedup (around 5 times faster) for the optimized version (see Figures 4.1 and 4.2). While it might suggest that the optimized raytracer scales better, this is not a good performance measure as it assumes that a GPU's FLOP performance correlates exactly to its raytracing performance. This assumption is unlikely to hold in reality and there are other factors to consider.

Even if more powerful graphics processors are used, linear speedup is hindered by the sequential instructions of the program that have to be executed by the CPU on a single thread, such as JavaScript code executions. There is also the additional overhead of creation of kernel functions and the switching of processers whenever there is a switch between JavaScript and kernel function code executions.

5 ACCURACY

Several accuracy issues have arisen from the implementation of the raytracers. For example, differences are observed when the same frame of a scene is rendered on the CPU and GPU (see Figures 5.1 and 5.2).

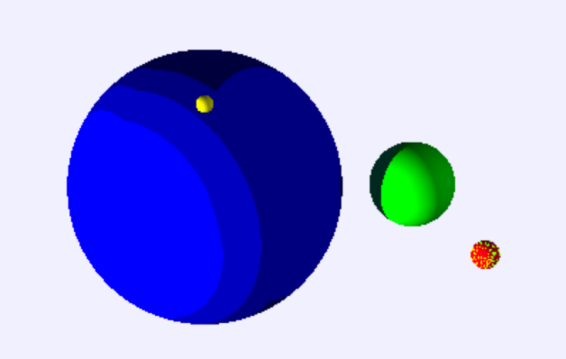

Figure 5.1: *GPU rendering of frame by optimized parallel raytracer. Notice that there are artifacts on the red sphere and the contrast of the blue sphere is too high.*

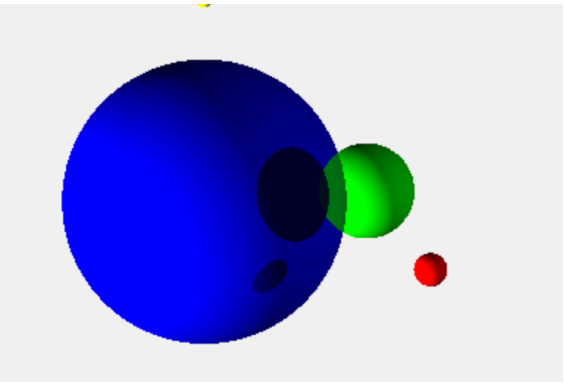

Figure 5.2: *CPU rendering of the sample frame shown in Figure 5.1 by optimized parallel raytracer. Notice that the artifacts on the red sphere are gone. However, the green sphere is now rendered as a translucent object. The background is also of a different color when rendered by the CPU.*

I was unable to arrive at a conclusion for this inconsistency but I suspect that it may be caused by some threads in the kernel function taking too long to finish running. On the naïve parallel raytracer, where the execution speed is a lot faster, no such artifacts were observed.

6 CONCLUSION

This project has provided a solution to load imbalances amongst threads in raytracers. We did not manage to see a performance improvement for the optimized raytracer as the rendering of objects in the scene is too light computationally. However, the benefits of such optimization will be observed if more complex objects are rendered.

The effectiveness of this optimization also depends on the distribution of objects in the scene. If the objects are sparsely and evenly distributed in the scene, then the optimization will not have much benefit since most of the view boxes will be active, i.e. there are observable objects in it. A solution will be to split the viewing plane into smaller view boxes. Though this will increase the number of threads created, there should not be a noticeable degradation of performance if the kernel functions are run using discrete GPUs, since they have many cores available for parallel execution.

7 APPENDIX

The following figures show renderings of scenes from the experiments carried out to measure performance of the raytracers.

Figure 7.1: *2 spheres in scene.*

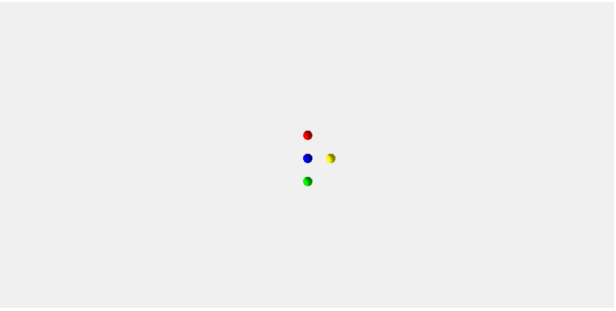

Figure 7.2: *4 spheres in scene.*

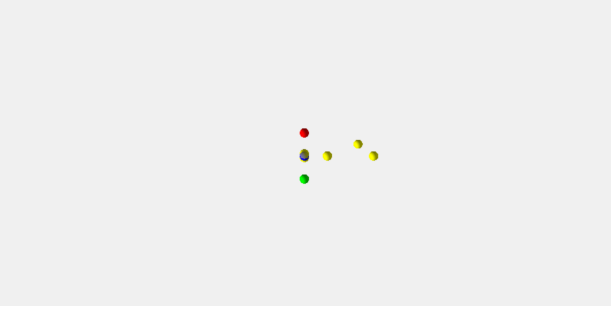

Figure 7.3: *8 spheres in scene.*

Figure 7.4: *16 spheres in scene.*

Figure 7.5: *32 spheres in scene.*