RAYTRACING WITH GPU.JS

CS3211 - Parallel and Concurrent Programming

Andres Galaviz - A0145029J

National University of Singapore

Table of Contents

Implementation:

- **Did you do all the object types?** The only object type that were implemented in the project was a sphere.
- **Did you implement lights and hence shadows?** Yes, both lights and casting shadows were implemented.
- **Did you implement the camera, or do you just have a fixed viewpoint?** A camera with a starting position was implemented. It can be moved with the keyboard.
- **Did you support all modes of reflection (using the ambient/diffuse/specular fields)?** Ambient lighting and Lambert diffuse shading are supported. The grounds to specular reflection were included but not pursued due to the lack of recursion or means to simulate recursion in GPU.js.
- **What type/level of raytracing did you implement? How many bounces?** Ray casting was implemented from the camera to the objects and from the point the ray hit in the object to all the different light sources. There was only one bounce back to the camera in the ray tracing algorithm and one bounce to the light source.
- **Did you do anything special for accuracy? Speedup? Workload reduction?** The scene elements are preprocessed before calling the kernel, this way the amount of repeated computations in each thread is reduced. This results in a better performance and reduced workload. To improve accuracy, the only approach was to reduce the number of divisions as much as possible and due to the preprocessing being carried out only once it was guaranteed that at least every thread will get the same initial parameters.
- **Did you use any other special techniques to improve the system?** Instead of using hardcoded numeric indexes to access the parameter values, a set of passed constants was used to facilitate debugging and escalation of the system. This makes the code much more understandable.
- **Can you adjust the parameters to specify different methods of operation?** Yes, on top of the GPU/CPU mode it also includes sphere rotation mode and light rotation mode.
- **Did you animate your scene?** Yes. In sphere rotation mode the scene shows various spheres rotating and casting shadows against each other and in light rotation mode the scene shows the lights rotating and casting light at the non-intersected objects.
- **Did you provide some sort of camera motion/fly-through technique?** Yes, camera motion and fly through technique was implemented. Use the Arrow keys and Z/X to move the camera X, Y, Z position. To change the normal vector, hold Shift + Arrow keys to change the vector components.
- **Did you provide sample scenes that provide evidence of testing?** The sample scene uses various spheres with different rotation cycles and various light sources of different colors which can also be rotated in order to test the lighting techniques and multiple scenarios.

Ray tracing was achieved in the implementation with only the use of Float 32 variables and certain preprocessing functions in the scene to reduce the workload of each thread. The whole architecture of the program was based on the basis of preprocessing certain aspects of the scene, therefor it was not possible to test the FPS without doing this step as the program would have to be changed. Doing a complexity analysis on this step tells us that in terms of Big O notation the complexity remains the same, however in real life situations the performance will be slightly better as the amount of vector operations is marginally reduced during execution time.

The site it publicly hosted in http://raytracing.azurewebsites.net/ with the source available in GitHub: https://github.com/AndresGalaviz/RayTracing

Results

Several tests were carried out for different screen sizes and object/light count with both GPU and CPU settings. The implementation supports object rotation and light rotation, after testing it can be seen that the results are very similar as no overhead is creating with varying the amount of objects/light sources. One interesting finding was that in CPU mode, the more objects/lights that had to be processed the higher the reported frame rate would be. This might be that as there are more objects in the scene to be triggered and more resources were needed Chrome or the OS will give a higher priority to the rendering.

In the source code and public implementation this is capped to at most 60 FPS, the reason for this is that running at the actual speed limit of the algorithm, Chrome wasn't able to keep up with the rendering process. This might as well be a bug of GPU.js, a bug in Chrome or just my algorithm and load balancing might not be good enough.

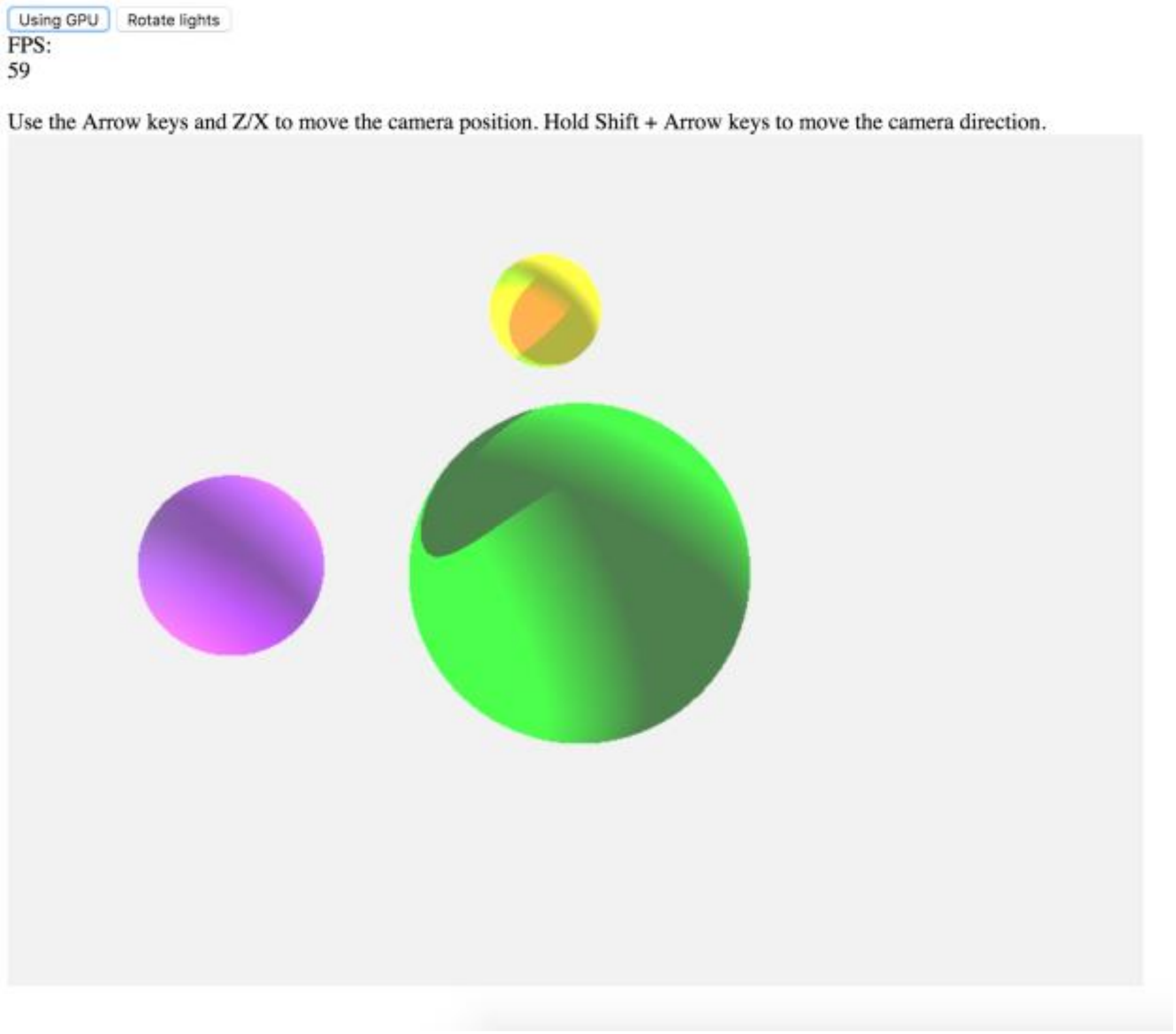
Sphere Mode	**Light#: 1 Object#: 1**		**Light#: 3 Object#: 1**		**Light#: 1 Object#: 3**		**Light#: 3 Object#: 3**	
Canvas Size	**CPU**	**GPU**	**CPU**	**GPU**	**CPU**	**GPU**	**CPU**	**GPU**
800x600	4	190	4	189	11	192	10	198
1024x768	6	193	6	193	6	192	6	193
1280x800	5	195	5	192	5	193	5	194
1280x1024	5	198	5	192	5	192	5	196
1366x768	4	191	3	193	4	194	3	192
1920x1080	1	190	1	186	1	191	1	193

Table 2.1: Tabulated results of Sphere Mode

Light Mode	**Light#: 1 Object#: 1**		**Light#: 3 Object#: 1**		**Light#: 1 Object#: 3**		**Light#: 3 Object#: 3**	
Canvas Size	**CPU**	**GPU**	**CPU**	**GPU**	**CPU**	**GPU**	**CPU**	**GPU**
800x600	4	188	4	188	11	200	10	197
1024x768	6	193	6	196	6	196	6	196
1280x800	5	192	4	197	5	198	4	198
1280x1024	5	194	4	190	4	195	3	196
1366x768	3	191	3	188	2	192	2	197
1920x1080	1	190	1	191	1	189	1	192

Table 2.2: Tabulated results of Light Mode

The results clearly show that the GPU is capable of handling the workload at a very high speed whereas the CPU wasn't able to cope with higher resolutions.



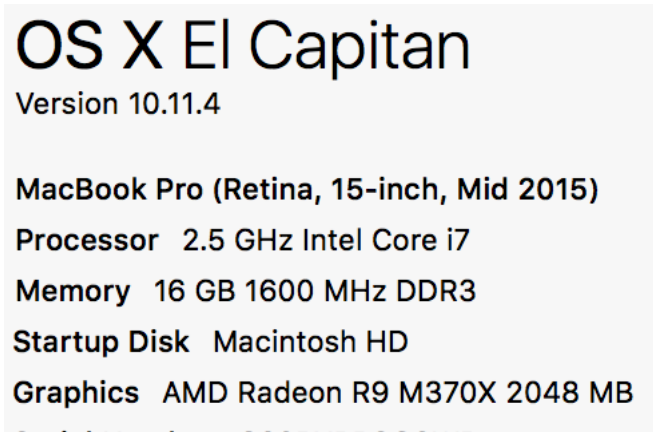
Use the Arrow keys and Z/X to move the camera position. Hold Shift + Arrow keys to move the camera direction.

Figure 2.1: Sample scene from a different angle showing 3 objects with light sources of white, green and red color.

The frame rate will drop in average 1 frame when moving the scene in camera mode, but as the execution mode is capped at the maximum of 60 FPS it is not notable when moving around the world.

Hardware

All the tests (Except when pointed out) were carried out in a MacBook Pro with the following hardware specifications:


OS X El Capitan
Version 10.11.4

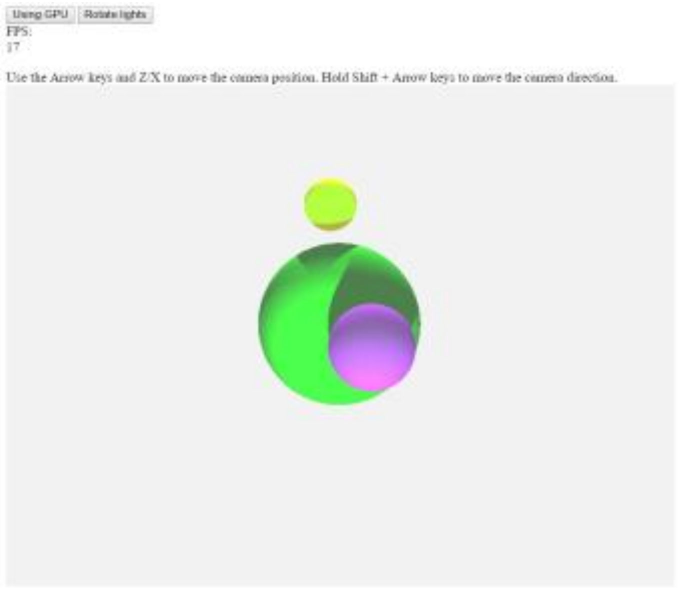
MacBook Pro (Retina, 15-inch, Mid 2015)
Processor 2.5 GHz Intel Core i7
Memory 16 GB 1600 MHz DDR3
Startup Disk Macintosh HD
Graphics AMD Radeon R9 M370X 2048 MB

The CPU is a Quad-core Intel Core i7 with Hyper threading technology running at 2.5 Ghz, which implies up to 8 real threads can be executed at once. The GPU is an AMD graphics card R9 M370x with 10 compute units, 640 stream processors running at 800 Mhz. These specifications contributed significantly to the project speed and behavior. The bigger amount of available processing units available for GPU.js to use in the kernel function resulted in a substantial increase in FPS.

A strange behavior was found when testing in this machine, the canvas would show a slight difference in color representation when changing from CPU to GPU computation mode, this might be because of an internal difference of GPU.js of value types or computation of color.

Other tests were carried out in more systems:

- Windows PC: Quad Core Intel Core i7 running ag 2.4 Ghz with an NVIDIA GeForce GTX 780M and Intel HD 4000 integrated graphics in Windows 10 using Chrome. This test had a particularly unexpected result, when running in GPU the frame count would not go over 20 FPS even though it was a much more powerful computer. After later inspection the reason was found to be that GPU.js was not activating the dedicated graphics card but instead it was using the Intel HD 4000 integrated graphics card. The results are shown in Figure 3.1.

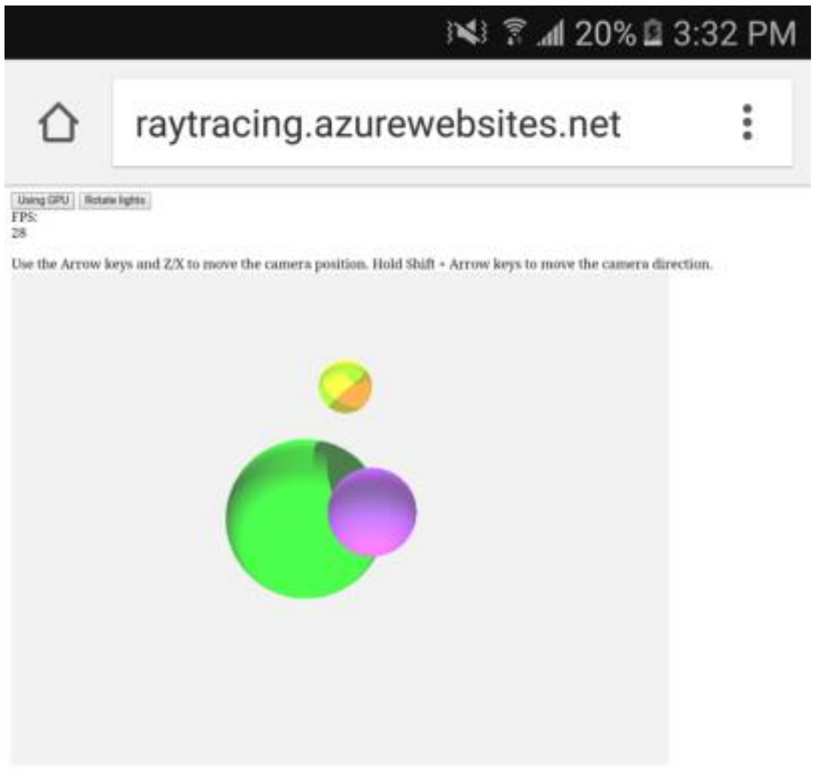


FPS:
17

Use the Arrow keys and Z/X to move the camera position. Hold Shift + Arrow keys to move the camera direction.

Figure 3.1: Sample scene running in Windows PC.

- Galaxy S6: Quad-core 1.5 GHz Cortex-A53 CPU with a Mali-T760MP8 GPU. The test was carried out in the default chrome browser provided in Android 6.0.1. In this hardware test the phone became very slow when using GPU mode yet it still managed to obtain a rather okay frame rate.

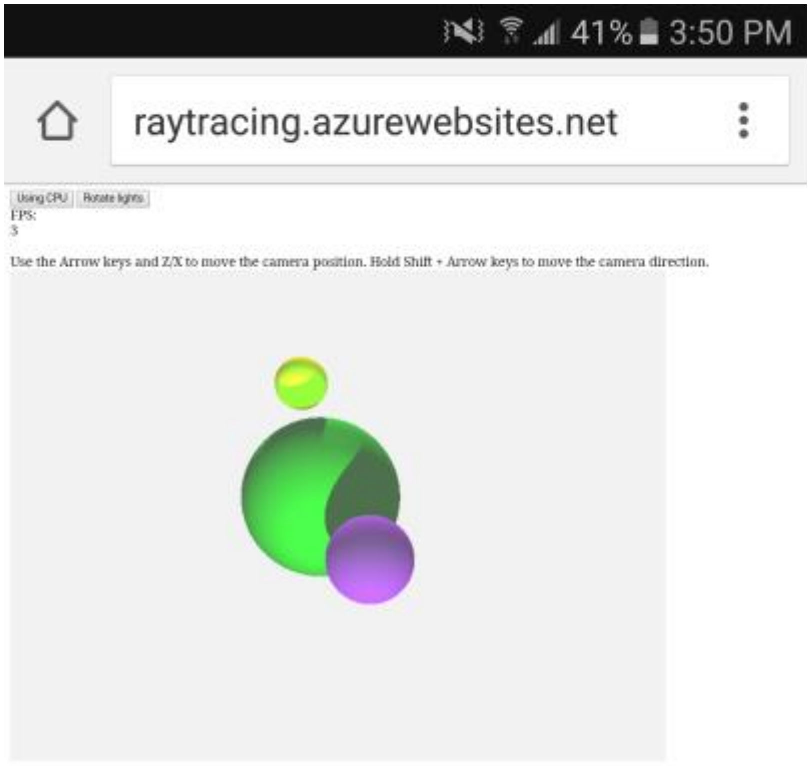


FPS:
28

Use the Arrow keys and Z/X to move the camera position. Hold Shift + Arrow keys to move the camera direction.

Figure 3.1: Sample scene running in Galaxy S6 with GPU Mode

On the other hand, when using CPU mode, the scene became almost still, averaging 2 FPS. This is mainly due to the fact that the phone only has 4 slow CPU cores available and might not be optimized to use them all at the same time for a web application.



FPS:
3

Use the Arrow keys and Z/X to move the camera position. Hold Shift + Arrow keys to move the camera direction.

Figure 3.2: Sample scene running in Galaxy S6 with CPU Mode

Speedup/Accuracy

The speedup from relying on the GPU and not the CPU comes from the fact that the GPU has many more available processors to process the individual pixel's computations. Even though the the CPU processors are faster, the GPU is able to process more operations at once which then translates into a much faster parallel execution than the CPU.

Speedup				
Canvas Size	**Light#: 1 Object#: 1**	**Light#: 3 Object#: 1**	**Light#: 1 Object#: 3**	**Light#: 3 Object#: 3**
800x600	47.5	47.25	17.45454545	19.8
1024x768	32.16666667	32.16666667	32	32.16666667
1280x800	39	38.4	38.6	38.8
1280x1024	39.6	38.4	38.4	39.2

1366x768	47.75	64.33333333	48.5	64
1920x1080	190	186	191	193

Table 4.1: Speedup in sphere rotation mode

Speedup	**Light#: 1 Object#: 1**	**Light#: 3 Object#: 1**	**Light#: 1 Object#: 3**	**Light#: 3 Object#: 3**
Canvas Size				
800x600	47	47	18.18181818	19.7
1024x768	32.16666667	32.66666667	32.66666667	32.66666667
1280x800	38.4	49.25	39.6	49.5
1280x1024	38.8	47.5	48.75	65.33333333
1366x768	63.66666667	62.66666667	96	98.5
1920x1080	190	191	189	192

Table 4.1: Speedup in light rotation mode

The tabulated data clearly shows that speedup steadily increases as program size increases (Both in terms of screen size and object count), with a maximum speedup of 193 while testing the biggest screen size and object count.

As discussed before this behavior is present because as count of objects increases the problem size increases by a square factor. Increasing the screen size also makes the problem behave in a similar way in relation to the increase in size. When the program is sufficiently big, the CPU can't handle the required degree of parallel computations.

Due to the limiting factor of only using Float 32 in the project there were only a few approaches to increase accuracy, some of the ones were to only carry out float divisions when strictly needed and to reduce the operations in where consecutive additions were carried out.