CS3211: Parallel Raytracer

Due on April 18, 2016

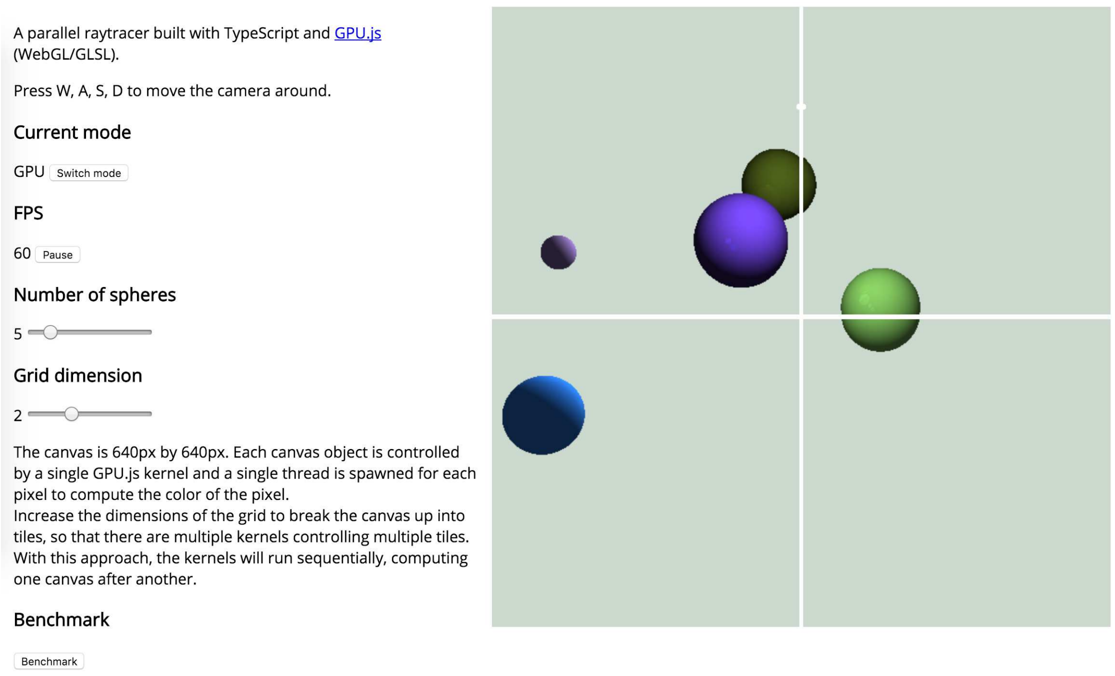
Chen Jingwen A0111764L

Introduction

In this paper, we provide an implementation of an animated and parallel 3D raytracer built with GPU.js, along with evaluation of the performance, rendering results of visuals, data analysis of speedup and accuracy, and summary.

GPU.js is a JavaScript library that provides an API and a compiler into GLSL, the language specification for WebGL. This provides web developers a way to access GPU resources and parallelize computations (e.g. matrix multiplication).

The working demonstration is available online at http://raytracer.crypt.sg, which has been tested on the latest versions of Google Chrome and Safari on OSX. The source code repository is available at http://github.com/jin/raytracer.



A parallel raytracer built with TypeScript and GPU.js (WebGL/GLSL).

Press W, A, S, D to move the camera around.

Current mode

GPU

FPS

60

Number of spheres

5

Grid dimension

2

The canvas is 640px by 640px. Each canvas object is controlled by a single GPU.js kernel and a single thread is spawned for each pixel to compute the color of the pixel.
Increase the dimensions of the grid to break the canvas up into tiles, so that there are multiple kernels controlling multiple tiles. With this approach, the kernels will run sequentially, computing one canvas after another.

Benchmark

Figure 1: Screenshot of the raytracer

Initial survey and thoughts

Having no prior knowledge of raytracer implementations, we looked into the Literate Raytracer guide (MacWright, 2013) that was linked within the assignment document. It's based on the popular backward tracing algorithm, which traces the path of light rays from the camera to entities in the scene. It's also more efficient than the forward tracing algorithm where even though it mirrors how photons work in nature, the majority of rays emitted from entities do not reach the camera and are wasted.

MacWright's algorithm is also sequential. It does a nested for loop to iterate through all pixels, resulting in a $O(n^2)$ time complexity. Here, we aim to parallelize computation as much as possible with the use of GPU.js.

Hardware

Development, testing and benchmarking are conducted on a Early 2015 Macbook Pro 13", with the following hardware specifications:

1. Intel Core i5 clocked at 2.9Ghz, 2 cores
2. Intel Iris Graphics 6100 with 1536MB of VRAM

Being an onboard processor, the Intel Iris Graphics 6100 does not that the hardware capabilities that a dedicated GPU will have. It has 48 execution units as compared to the hundreds of a dedicated GPU, but it is still magnitudes higher than the 2-core Intel core i5. We will see the performance speedup in a later section.

Implementation

We chose to implement the raytracer using TypeScript, a superset language of JavaScript that provides several enhancements to the language and compiles down to standard JavaScript. The most notable enhancement is static typing – this provides compile time type safety for portions of the code where the type has been explicitly annotated. This has proved to be immensely useful during development, when several classes of bugs were caught by the compiler as complexity grew. TypeScript also provides APIs for classes and enums, both of which are features in ES6 but not ES5.

Unit tests were also written for the Vector operation functions to ascertain confidence in the actual raytracing algorithm.
On top of the regular initial raytracing to detect and infer the pixel colour from the closest entity intersecting with the camera ray, we also implemented the additional following features:

1. Lambertian shading

   This provides a lighting effect using light rays shone onto entities, from light points placed in the scene. A corollary effect from this is having entities being able to cast shadows on each other.

2. Specular reflection

   This makes the surface of entities reflective, hence mirroring the shapes and colors of other objects on an entity's surface if the ray intersects. This is implemented with a depth of three bounces.

3. Ambient coloring

   Ambient coloring adds the ability to color shadows, so entities do not look completely black when parts of it receive no light rays.

4. Camera movement

   The keys W, A, S and D have been programmed to move the camera forward, backwards, leftwards and rightwards respectively.

5. Animated spheres

   For visual enhancements, each scene has randomly generated spheres with different sizes and colors. These spheres are animated to bounce within an imaginary cuboid boundary.

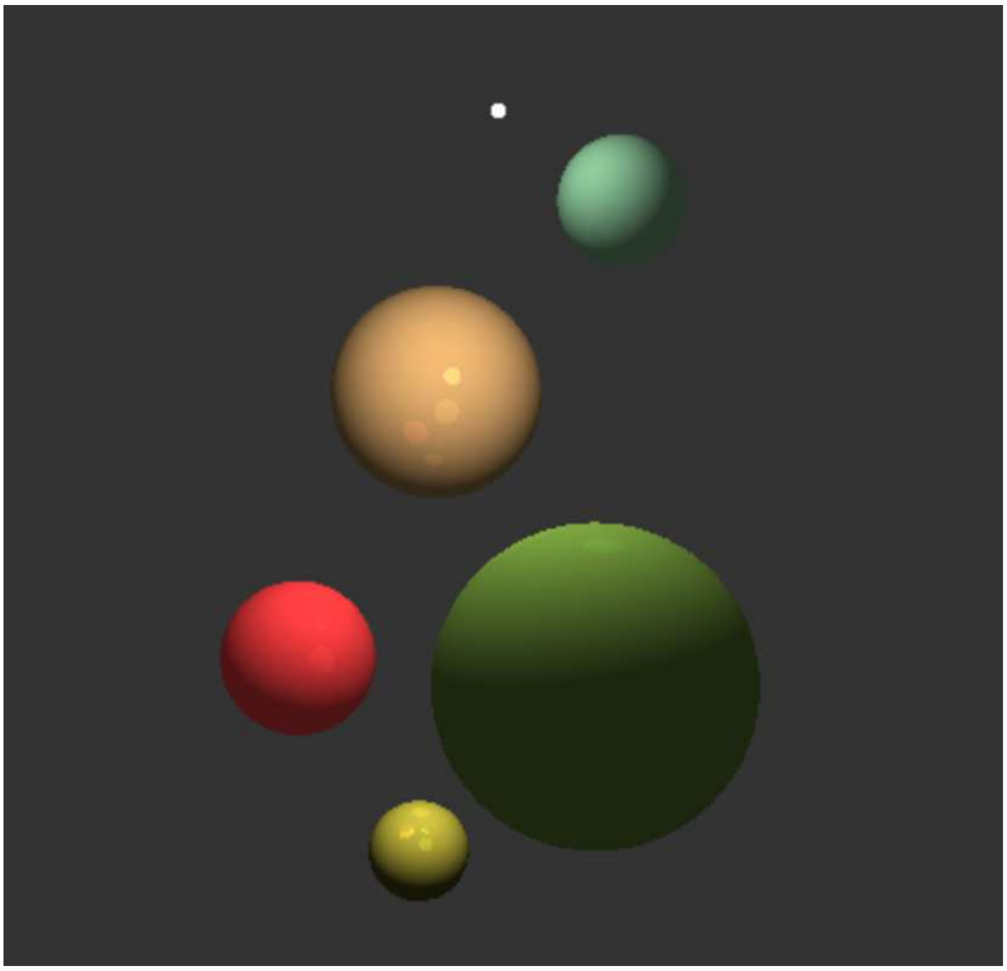


Figure 2: Raytracer scene

6. Collision detection and basic 3D physics

   There is also basic collision detection between two spheres, as well as a function to reflect two spheres after they have collided with each other.

7. Additional LightSphere type as a physical scene entity for lights.

   There was a small visual annoyance as to how the light source was invisible to the user, so we added a LightSphere type that acts as a white light and does not interfere physically with any other entity in the scene.

8. Plane intersection

   Ray-Plane intersection functions were also implemented, but we were unable to make planes aesthetically pleasing enough to include in any demonstration scenes.

9. Slider inputs to tune scene entity count and rendering methods on the fly.

Benchmarking method

Before we can implement optimizations to the raytracer, it is crucial to quantify the performance of it. There are a few features that we can use to determine the speedup obtained with parallelization.

1. Frames rendered per second
2. Minimum, average or median time taken to render a frame
3. Time taken to render N frames

Here, we compare the advantages and disadvantages of each feature.

1. **Frames per second**: This is dependent directly on the time it takes to render a single frame, along with any overhead of the browser and hardware's capability to draw the image on the screen. It can be also seen as the tick rate, which is a good measure for speedup.
2. **Average frame render time**: Fluctuates far too much due to the susceptibility to outliers. If a single frame takes an abnormally long time to render, the average becomes an inaccurate metric.
3. **Minimum frame render time**: Theoretical fastest time it takes to render a frame, good metric to analyse.
4. **Median frame render time**: Not affected by outliers, especially abnormally slow render times. Representative of data.
5. **Time taken to render N frames**: This is just a summation of the total time taken for each frame and may contain outliers. Not as representative as the median.

With these in mind, we have decided to go with average **FPS**, **Minimum** and **Median frame render times** as benchmark features.

Each benchmarking run will be based on the following steps:

1. Benchmark CPU
2. Benchmark GPU with the same scene
3. Compare speedups on the various selected features.

An important thing to note is that calls to the GPU kernel are asynchronous, unlike the blocking calls to the CPU kernel. Therefore, the speedups measured are *perceived* by the user and not the *true* speedup. Unfortunately, due to the implementation of GPU.js, this is the closest we can get to the metal without writing GLSL directly. A feature improvement for GPU.js will be providing a callback method in the kernel to correctly time the execution time.

Accuracy

In both JavaScript and the compiled GLSL code, all number data types are floats. There are no integers at all, which makes calculation results at several corners in the raytracer less accurate than they should be.

Examples where this inaccuracy led some unexpected behaviour is when indexes are computed and used before they are floored (e.g. array[10 / 3]), and during ray intersection detection where the minimum distance limit couldn't be exactly zero (but it should be theoretically) due to the floating-point vector operations, so we had to modify it to become a very small negative number (e.g -0.005).

Another example is that a particular speedup benchmark was computed to be 24.797081926775217, but cases like these do not usually require exact accuracy and can be rounded up or down.

GLSL's Float32-only computations also hasn't presented any serious issues to the raytracer functionalities.

Parallelization and optimization techniques

Motivation

A huge advantage GPU.js has that many WebGL wrappers lack is the ability to fall back to use the CPU for the kernel if WebGL is not available. The user also has the ability to create the kernel to use the CPU mode explicitly.

However, the biggest disadvantage of the CPU kernel is that all parallel code are ran sequentially, which slows the algorithm execution down significantly. A speedup comparison plot of the demo raytracer scene with an increasing number of spheres is shown in Figure 3.

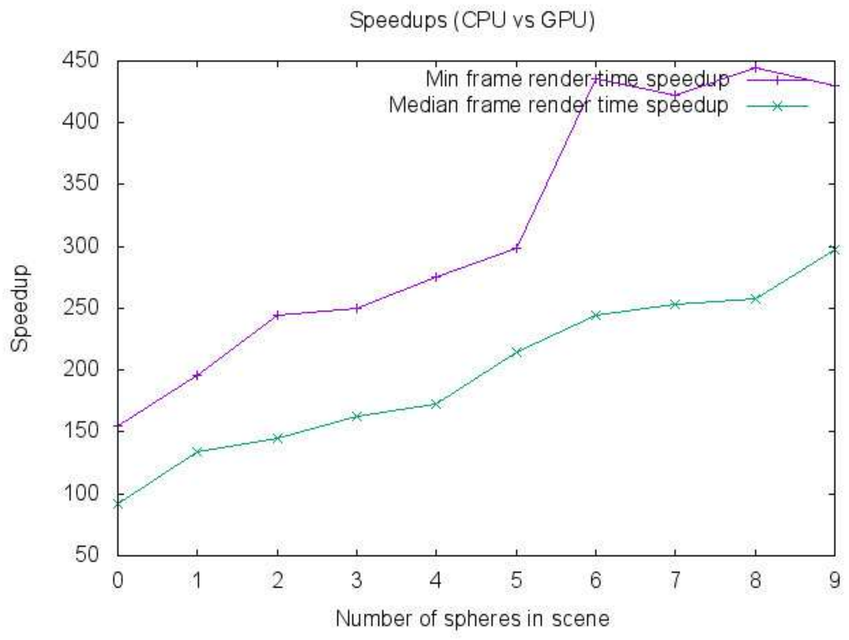


Figure 3: Speedup with an increasing number of spheres in scene

From Figure 3, we can see that with more entities in the scene, the larger the speedup will be. It is therefore crucial to implement parallelization techniques to take advantage of the modern multi-core and multi-processor systems.

Pixel by pixel parallelization

Raytracing has been described as *embarassingly parallel*, where the main task can be broken down into parallelizable and independent subtasks easily. In raytracing, each pixel's computed color is independent from every other pixel's, and thus we can distribute each pixel, or sets of pixel, among processing units. This is also known as **parallel rendering**, which happens to be the only parallelization implementation for coloring a canvas with GPU.js.

For benchmarking purposes, we fix the canvas size to be 640 pixels wide by 640 pixels high, and raytrace a light source and a few spheres bouncing about in a cuboid boundary. The GPU kernel will create one thread per pixel, so the upper bound on speedup is equal to the number of pixels on the screen.

A downside to this method is that the overhead of managing so many threads and may cause a bottleneck or resource starvation in the GPU. This might be counterproductive to the speedup advantages that it's expected to bring, which is why it has inspired us to attempt an optimization technique: tiling.

Tiling of canvas into smaller chunks

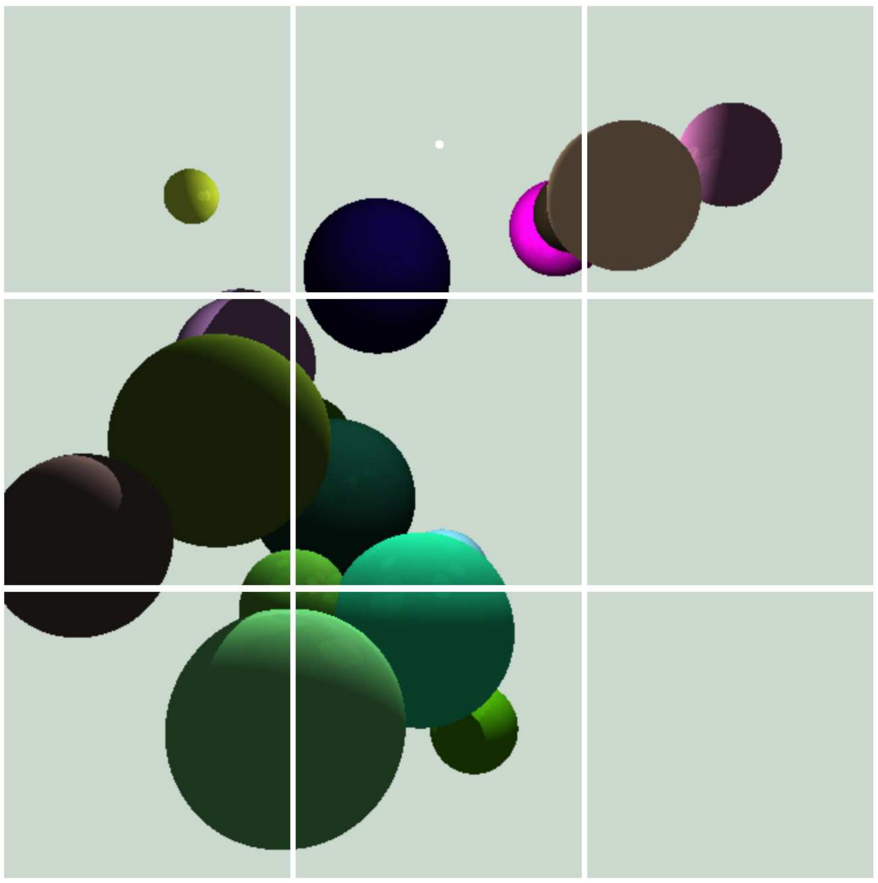


Figure 4: 3 by 3 tiling grid

Instead of assigning the entire canvas to a single kernel and creating N^2 threads, we can reduce the number of threads created at any point in time by slicing the canvas up into small tiles, and then assigning each tile to a GPU.js kernel. The idea here is to have each kernel exploit spatial locality, and the reduction in overhead caused by the enormous number of threads.

In the current implementation, the canvas can be equally divided into 2x2, 3x3 and 4x4 tiles, which represents 4, 8 and 16 kernels created respectively. Each kernel is executed sequentially, so the number of threads created at each point in time is reduced quadratically as the grid dimension of sub-canvases increases.

4x4 tiles is the limit as the browser will show a warning for too many WebGL contexts created with a 5x5 grid.

Next, we benchmark the GPU vs CPU speedups for each tile division setting.

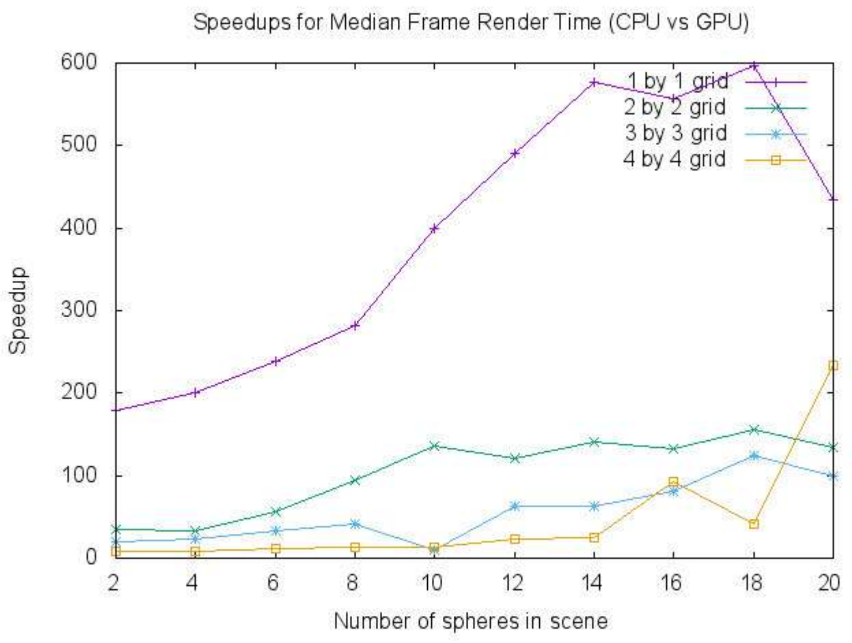


Figure 5: Median frame render time speedup

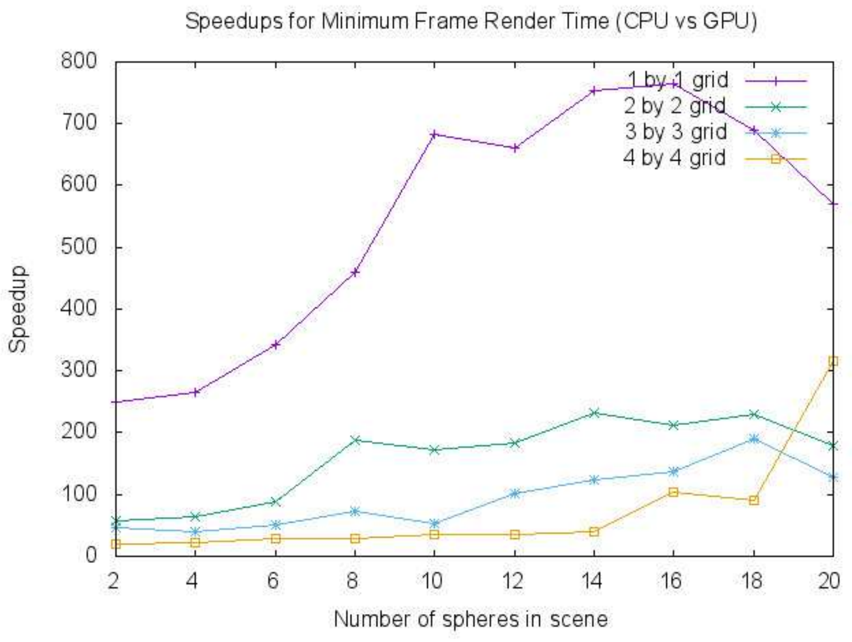


Figure 6: Minimum frame render time speedup

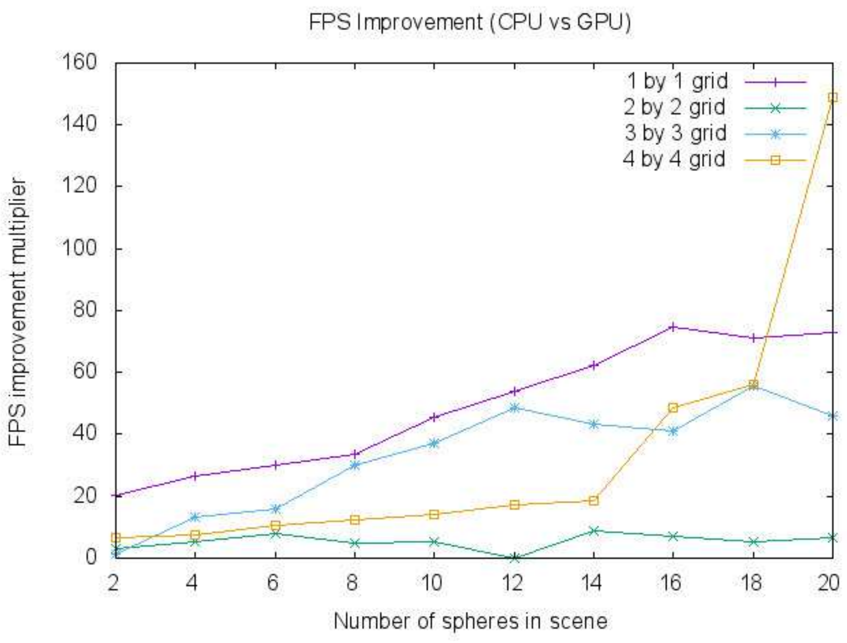


Figure 7: Frames per second improvement

Here, we observe that speedups were observed across all grid settings. We also observe that the speedups obtained with 2x2 grid and above are much lower than the 1x1 setting, and this is likely due to the sequential execution aspect of multiple kernels, which brings the GPU execution model closer to the CPU counterpart.

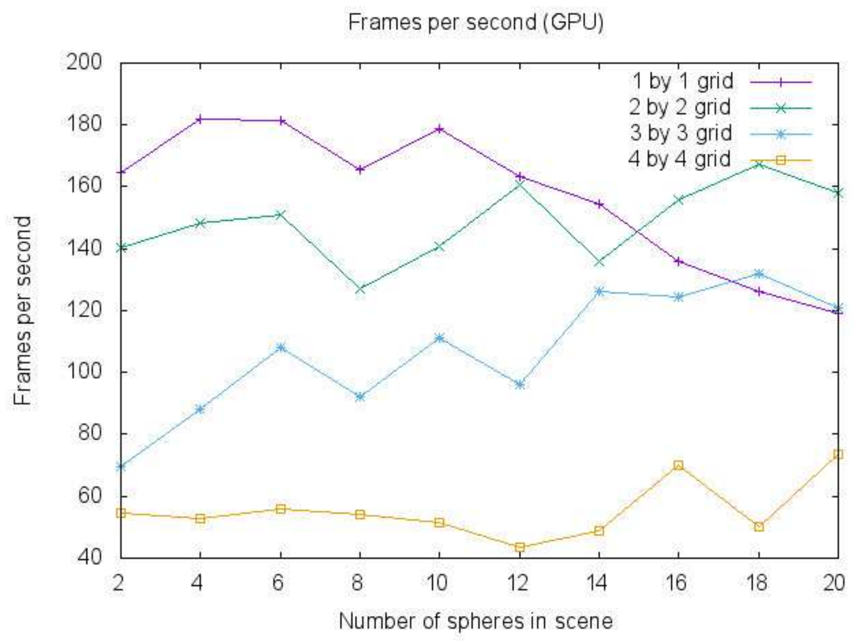


Figure 8: Frames per second benchmark

We observe that at a lower entity count, a 1x1 grid has a better performance, but as the entity count increases, larger grids with more subdivisions had a better performance. This shows that the more complex a scene is, the more performance gain one will get by subdividing the canvas into subcanvases.

However, the 4x4 grid performance stuttered – this is likely to due to massive overhead that comes with having 16 WebGL subcanvases on a browser screen.

Challenges

Parallelization library in alpha stage

Having to work with a library that's still in its alpha stage was incredibly challenging, especially due to the lack of data types like vectors, the lack of debugging methods and different ways to set the default parallel task distribution instead of the default 1-thread-per-pixel option.

The JavaScript subset language that we were able to use in the kernel code was also very limited, leading to very complex and non-DRY (Don't Repeat Yourself) code.

However, this forces us to really understand the inner working of the arithmetics that goes on behind building a raytracer, which we thought was a really good opportunity to learn more about computer graphics.

Future work

Sparse voxel octree and 3D-DDA

We aim to further implement more optimizations and benchmarks on this raytracer, and the next step is to implement a data structure that we came across during literature survey: sparse voxel octree.

A sparse voxel octree (Sung, 1991) is a tree data structure for 3D spaces, where each node has exactly 8 child nodes. The 3D space can be recursively divided into 8 smaller volumes called voxels (Volumetric Pixel). The recursion depth is not bounded, but the base case is usually met when a voxel has at most N entities that intersects it. The raytracer then traverse through this data structure using an algorithm like 3D-DDA (3D Digital Differential Analyzer), and only raytrace voxels that has known entities from the pre-computation.

A sparse scene will result in larger, empty voxels, which can be skipped by the ray instantly. This prevents wasteful computation while still allowing as much detail as possible, due to the recursive nature of voxelisation.

Another advantage of this data structure is that it does not need to be completely loaded into memory – it can be streamed as needed to the raytracer for voxels that it intersects. This greatly lessens the memory pressure for scenes with larger resolutions.

Conclusion

In this paper, we presented an implementation of a parallel raytracer, additional features, and a couple of parallelization methods to divide the raytracer tasks up and distributed to multiple processing units.

We have observed that with a lower entity count in the screen, a single canvas with one thread per pixel works better, but as the entity count increases, subdividing the canvas into subcanvases and executing each subcanvas with a kernel sequentially will actually have a higher frames per second rate. This concludes that subdivision and tiling of the canvas is a viable parallelization optimization of a parallel raytracer.

References

MacWright, T. (2013, November 2). Literate Raytracing. Retrieved April 7, 2016,
from http://www.macwright.org/literate-raytracer/

An Overview of the Ray-Tracing Rendering Technique. (n.d.). Retrieved April 13, 2016,
from http://www.scratchapixel.com/lessons/3d-basic-rendering/ray-tracing-overview

Boulos, S. (n.d.). Retreived April 5, 2016
from https://graphics.stanford.edu/ boulos/papers/efficient_notes.pdf

Sung, K. (1991). A DDA octree traversal algorithm for ray tracing. In F. H. P. and W. Barth (Ed.), Eurographics91. Proceedings of the European Computer Graphics Conference and Exhibition (pp. 73-85). Retrieved April 8, 2016 from http://faculty.washington.edu/ksung/pub/1991.EG91Paper.pdf