Name: Curtis Tan Wei Jie
Matriculation Number: A0129529W

CS3211 Programming Assignment 2 Report

General Hardware Specifications

AfterShock Laptop (Personal Laptop):

Core: Intel i7-4710MQ CPU @ 2.50GHz, 4 Hyperthreaded Cores, each with 2 Threads,
4 * 32KB L1 icache, 4 * 32KB L1 dcache, 4 * 256KB L2 cache, 6MB L3 cache

Integrated GPU: Intel HD Graphics 4600 @ 400MHz, 2GB Graphics Video Max Memory (from CPU)

Discrete GPU (**not used**): NVIDIA GeForce GTX 860M @ 797MHz, 4GB GDDR5 Memory (Discrete)

General Hardware Discussions

In GPU.js, our variables are given the value of float32, which are 32 bits in size. Given that the Processor used has a 32KB L1 dcache, it should be able to fit approximately 1000 variables at most. Furthermore, it also has a 32KB L1 icache for storing the instructions for GPU.js.

As the number of variables used is relatively small, compared to the number of variables storable by the L1 cache, CPU mode should have the advantage of not going beyond L1 cache. On the other hand, GPU mode might need much more memory to store the variable for every thread's local data - integrated GPU uses the CPU's memory resources.

Ray Tracing Features Implemented

Objected supported: Spheres
Lights supported: Yes, single source
Shadows supported: Yes
Camera supported: Yes
Camera movement supported: Yes, and always points at center of first object

Ambient shading supported: Yes
Diffuse shading supported: Yes
Specular shading supported: Yes, with shininess (Phong's reflection model)

Ray Tracing Level: 1 bounce (and 1 penetration, if object is translucent)

Antialiasing supported: No

Toggling between CPU and GPU: Yes
Animated scene: Yes

Other features: Able to modify Object properties from the web page itself

Reducing Work

In Physics, the general behavior of a photon is one that is emitted from a source, which travels either by a direct path, or through several reflections and refractions, bounces into the eyes of the observer. The photon, depending on its wavelength, then induces a response in the light-detecting cones of the eye that detects red, green or blue light.

Simulating the above behavior, however, is computationally expensive. Given an evenly lit environment, the surface area of the environment is much larger than the surface area of the eye or camera. This means that a large number of the photons that are absorbed, do not end up in the camera or eye, and hence, is rarely of concern to the observer.

One strategy to reduce the number of expensive computational work, is to reverse the path of the ray tracing simulation. Instead of following all the photons that are emitted from a light source and gathering the information of those that do bounce into the camera, we instead simulate the reversed path of the photon from the camera to the light source. This allows us to only need to compute a small subset of all the photons that were emitted by the light source, to produce a visually similar result.

Consideration of Bounding Box

The bounding box is a useful technique that has been used to reduce the number of computational work for the raytracer.

In the naive approach, each pixel of the canvas is a given a direction vector, for which the (reversed) photon will travel from the camera to some potential objects. Therefore, we need to do some computational work to check if every pixel's photon vector intersects with any of the objects in the scene. Suppose we are also given the opportunity to move our camera, it would be computationally expensive to have to calculate the possibility of the photons intersecting any object, if we had already moved our camera to face away from all the objects in the scene.

Instead, we can use a slightly cleverer approach. Using the same "reversing" approach as we did with the photons, we can calculate the direction vector of some object's center with our camera's position. This allows us to know if the center of the object is within our field of view. By using the coordinates of the camera and the object's center, we can also calculate the distance of the direct vector from the camera to the object. Finally, using the object's radius, we can determine if any part of the object stays in the field of view of the camera.

Using this approach, we have 2 options for reducing the computational complexity of the raytracer. Firstly, if the object is entirely not in the field of view of the camera, we can remove it from the list of objects to consider. This approach has a caveat, however. If the raytracer is interested in tracing contributions from its reflection, then we cannot do so, as an object that is out of view might still be able to contribute some intensity of red, green and blue through reflection. Next, we can opt to only send out rays that we estimate, will intersect with an object.

For pixels where we are certain that they will not intersect with any object, we can avoid tracing those rays, and instead, just paint it with the background color.

While the bounding box approach is very useful for sequential ray tracers, it fails to provide performance boosts when it comes to parallel ray tracers. To be more exact, rays that do not intersect with any object in the parallel version of the ray tracer, tend to have less computational work than rays that do intersect some object. Therefore, saving computational work here do not decrease the length of the critical path in the parallel ray tracing program - rays that end earlier still need to wait for other rays to complete their computation before the frame can be rendered.

Therefore, the bounding box approach was not implemented in the parallel ray tracing program.

Consideration of kd-tree

In most ray tracing programs these days, as bounding boxes don't provide sufficient speedups, the use of the kd-tree works well as an extension of the bounding box solution. In this case, as the bounding box was not implemented, then the kd-tree solution was also foregone.

Choice of Shading Model

There are several shading models that we can consider for our implementation of the ray tracer. In particular, the resource given in the basic ray tracing javascript example, supplied in the project specifications, uses the Lambertian shading model. This particular shading model is useful because only few computation is required to shade any pixel in the canvas. However, the downside to this shading model presents itself through the apparent lack of brightness in most objects that are rendered through this shading model, giving it a rather dark and eerie feel.

Since we do not implement objects as polygon meshes, but as implicit surfaces, it also makes little sense to use the Gourand shading model for this implementation. Thus, I chose to implement the Phong reflection model, giving the ray-traced objects a more realistic appearance, while keeping the number of computational work relatively low, compared to the Lambertian shading model. Furthermore, the Phong reflection model only requires simple arithmetic for the calculation of each component's contribution, which makes it great for the implementation of the ray tracer in GPU.js, which generally accepts functions that uses numerical primitives only.

The Phong reflection model derives the contribution of each Red, Green, and Blue component of the photons using the following formula:

Ambient Contribution: Illuminance * Light Color Intensity * Object's Ambient * Object's Color
Diffuse Contribution: Illuminance * (Vector to Light • Surface Normal Vector) * Light Color Intensity * Object's Diffuse * Object's Color
Specular Contribution: Illuminance * (Reflection Vector • View Vector)$^{\text{Smoothness}}$ * Light Color Intensity * Object's Specular * Object's Color

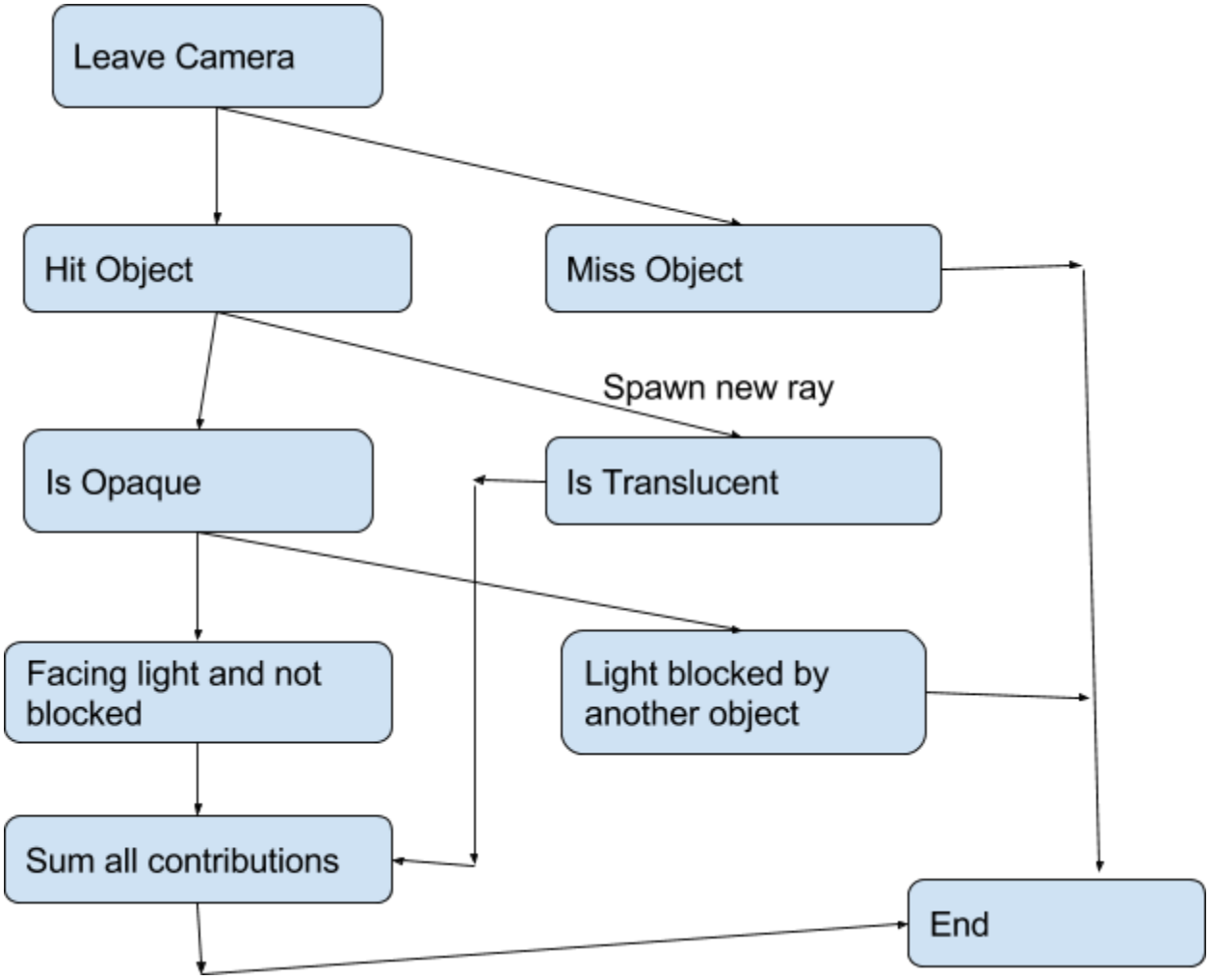
In particular, **Illuminance** refer to the remaining intensity of the light photon that contributes to that component. For a single bounce (and see-through) ray tracer, the illuminance of the photon at some surface, corresponds to the opacity of the object. The **Vector to Light** is the vector from the surface normal to the light source. The **Surface Normal Vector** is the vector of the normal of the surface at the point where the photon hits the surface of the object. The **View Vector** is the vector from the point of intersection at the surface of the object, to the camera. Finally, the **Reflection Vector** is the vector the photon would have taken in the real-life Physics model, when the photon leaves the light source, hits the surface of the object, and bounces out.

Redefining Properties of Object

In order to perform the Phong reflection model for the ray tracer program. We need to add one more property to the objects that are represented in scene.js. Specifically, we need to add the "smoothness" property that finds itself in the calculation for the specular contribution. We find that the recsz variable can be replaced with a single external constant that defines the number of hops it takes to go from an object's property to another, in the array. Hence, we replace the meaning of that variable with the smoothness variable.

Life Cycle of a Ray

In the single bounce and see-through implementation of the ray tracing program, each ray that originates from a pixel has the following critical path:



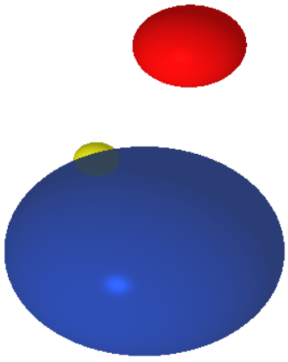
Therefore, the longest path for any ray would be the path where it hits a translucent object.

Measuring Speedup

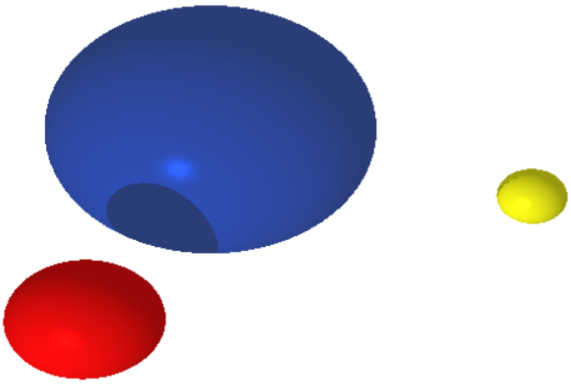
The most direct way to measure speedup would be to calculate the number of frames rendered per second. However, there are some important points that we might want to take note of. In particular, some frames might require more time to compute than other frames. Therefore, as the spheres continues its orbit in the scene, the amount of frames that the GPU can render per second might fluctuate. Furthermore, the frames per second value is generally too small to give us any meaningful computation of speedup. Therefore, we would like to change the measure of speedup to the number of milliseconds it takes to compute 100 frames.

For the case of our parallel ray tracer, we would like to know if there are any difference in the amount of time required to compute the result of some different frames. From the diagram above, illustrating the path of processes that some ray might consider, we hypothesize that there should be no difference in computing the time taken to generate 100 frames, regardless of whether the objects are moving, stationary, or behind some translucent object.

First, we define 3 particular scenarios for our experiment. The objects can either be rotating about some central sphere:



Angled view of rotating sphere t_1



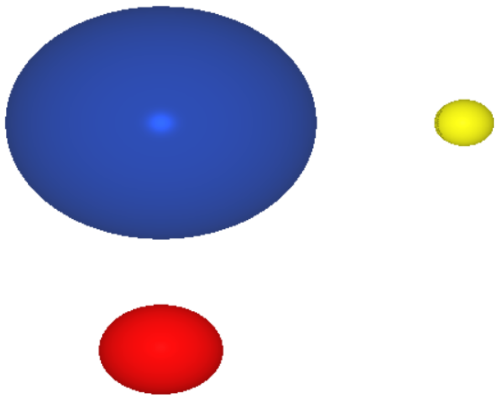
Angled view of rotating sphere at t_2

In order to achieve this, the rotation logic in render_loop is as such:

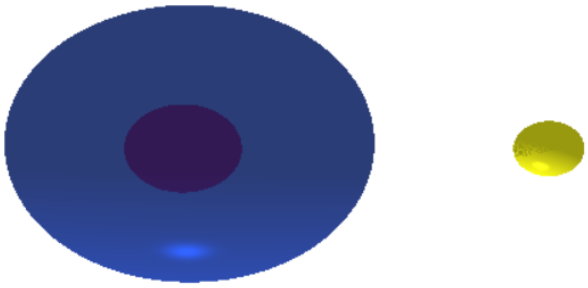
```
objects[1 + 1 * object_skip + 1] = 16 - 2 * Math.cos(loop_y);
objects[1 + 1 * object_skip + 2] = 8.5 + 2 * Math.sin(loop_z);

objects[1 + 2 * object_skip + 0] = 15 + 2 * Math.sin(loop_y);
objects[1 + 2 * object_skip + 2] = 8.5 + 2 * Math.cos(loop_z);
```

The objects can also be fixed to stay at the side of the central sphere:



Front view of spheres fixed to the side



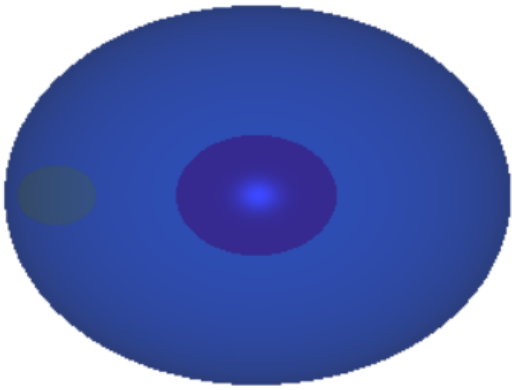
Top view of spheres fixed to the side

Here, we modify the rotational logic such that there is no confusion between whether the speedup comes from the not having the compute the new coordinates of the sphere, or whether there actually is a speedup in rendering such a frame. To do this, we maintain the need to perform the arithmetic:

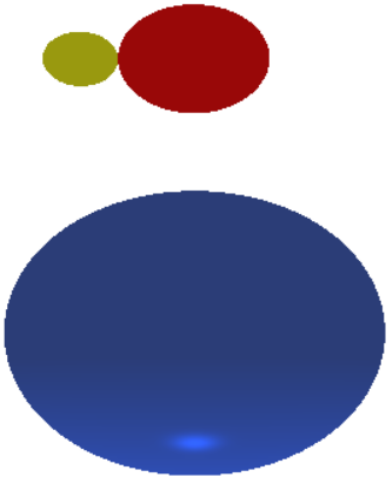
```
objects[1 + 1 * object_skip + 1] = 14 - 0 * Math.cos(loop_y);
objects[1 + 1 * object_skip + 2] = 8.5 + 0 * Math.sin(loop_z);

objects[1 + 2 * object_skip + 0] = 17 + 0 * Math.sin(loop_y);
objects[1 + 2 * object_skip + 2] = 8.5 + 0 * Math.cos(loop_z);
```

Finally, the objects can also be fixed to the back of the central sphere:



Front view of spheres fixed to the back



Top view of spheres fixed to the back

Again, we maintain the need to perform the computation of the new coordinates, even though the coordinates do not, themselves, actually change. This prevents any confusion in whether the speedup comes from the loss of computation of actual less work for the GPU:

```
objects[1 + 1 * object_skip + 1] = 16 - 0 * Math.cos(loop_y);
objects[1 + 1 * object_skip + 2] = 6.5 + 0 * Math.sin(loop_z);

objects[1 + 2 * object_skip + 0] = 14.4 + 0 * Math.sin(loop_y);
objects[1 + 2 * object_skip + 2] = 6.5 + 0 * Math.cos(loop_z);

loop_y = (loop_y + 0.01) % (2 * Math.PI);
loop_z = (loop_z + 0.01) % (2 * Math.PI);
```

We run all three scenarios and compute the number of milliseconds each scenario needed to compute a 100 frames. We do this 3 times, to try to avoid cases where computation take slightly longer due to the scheduler's decision:

GPU

Moving	Stationary (At side)	Stationary (Behind)
4442	4251	4250
4229	4292	4278
4220	4474	4508

The minimum number of time required to render a frame for the three cases are 4.220ms, 4.251ms, and 4.250ms, respectively. We can see that there is almost no discernible difference between the time required for the three cases. Therefore, we can be certain that the number of work required to compute 100 frame is the same, regardless of each object's position relative to another's.

To support the idea that there is actual difference in computational work (that is hidden by the parallel implementation due to no real change in the length of the critical path), we also run the same simulation in a relatively sequential CPU version.

CPU

Moving	Stationary (At side)	Stationary (Behind)
35136	34804	33887
34955	34834	34005
34898	34770	33054

Here, we find that the difference become slightly more significant, with the three timings being 34.898ms, 34.770ms, and 33.054ms, respectively.

Also of slight interest, we find that there is an approximately **8 times speedup** from the CPU version to the GPU version of the algorith.

Space Locality

Attempts were made to increase performance through better space locality (e.g. lumping creation and use of variables as closely as possible). This is because a memory fetch operation is expected to be much more computationally expensive than an arithmetic operation. Therefore, increasing the space locality, such that there is less cache misses, should normally result in a better performance for the algorithm.

Our final results were obtained below:

GPU

Moving (raw)	localising parameters
4442	4249
4229	4461
4220	4287

The timing for the three cases were 4.220ms, 4.249ms, and 4.217ms for the three cases, respectively. As the size of the cache was generally much bigger than the size of the program, as mentioned previously, this might explain why there is no discernible speedup from trying to increase the spatial locality of the algorithm.

Accuracy of Ray Tracer and Float32

The accuracy of the ray tracer, largely, depends on the accuracy of the float32 primitives that are supplied by GPU.js. As the rays heavily rely on floating point arithmetic, it is entirely possible for floating point errors to creep into the calculation, especially as the number of bounces become significant.

Another source of error for the ray tracer comes from the inaccuracy that arises when deciding whether to shade a pixel using only a single light vector. In particular, this problem can be addressed by using anti-aliasing, where each pixel now takes the average contribution of multiple (slightly skewed) light vector that originates from the same pixel.

The larger problem that arises from anti-aliasing is the larger number of computation required, as each pixel must now potentially trace more than a single ray. In general, there are strategies to minimise the increase in computation that is caused by anti-aliasing.

Instead of having every pixel throw a constant number of rays, as is the case for super-sampling anti-aliasing, we can have pixels only throw out more rays when the pixel expects itself to hit an object, as is the case for multi-sampling anti-aliasing. In multi-sampling anti-aliasing, we find that it is computationally wasteful to sample the color of the background more than once. Hence, we can cut the number of extra rays to throw out, if we can define the pixel areas where super-sampling is computationally useful.

Since bounding box was not implemented for this ray tracing program, as it was hypothesized to not improve the time taken for the algorithm to run significantly, multi-sampling anti-aliasing is also not implemented.

The alternative, super-sampling anti-aliasing, was also foregone as it might take significant computation time to reduce a super-sampled canvas obtained from the GPU, using the CPU. In particular, to condense the higher resolution result from the GPU, we need the CPU to iterate through all the pixels again, which runs contradictory to our motivation for keeping the individual pixel calculation in the GPU. On the other hand, super-sampling anti-aliasing might become more feasible when there is some way for each thread in the GPU to communicate with the other, so as to reduce the result in parallel.

On that note, it was also not likely to be feasible to pass the super-sampled result back to another kernelFunction to compute the new smaller sized array, as GPU.js only support one to all communication of data (from CPU to GPU), which could mean that we would need to waste significant computational time sending the entire super-sampled result to each GPU thread.