DOD Assignment – John Green

Introduction

For this assignment I have implemented a optimised simulation that allows for tens of thousands of circles/spheres to bounce throughout a scene. Within the project I have used several DOD optimizations as well as worker threads. For the assignment I believe I have met all criteria and below is details on how I attempted them.

Features

Worker threads

Worker threads are used within the project for all performance critical work as well as the scene setup. The way that this was achieved was by creating a array of threads that wait for a notification that a task is ready for them to process. Upon sending a task using the “StartTask” function It will split the task between ‘n’ threads where ‘n’ must be below available workers and is defined in argument list as well as how many spheres each thread will process.

The benefit of using worker threads for this project is that it allowed me to split up a task that would normally be ran on a single thread, into ‘n’ tasks that run on ‘n’ threads and that can be ran in parallel as they work on there own isolated data set so there is no race conditions or deadlocks with mutex’s.

Line to sphere collision

For the project I have implemented a line to sphere collision detection that is used to simulate a laser firing through the scene and hitting the spheres. The way that I achieved this was by using the algorithm described by (Thompson., 2020), where we find the point on the line that is closest to the sphere and determine if the circle/sphere’s radius encompasses it, if it does, we must be colliding with it.

The brief tells us that we must only hit the first sphere within the scene, to do this I first broke the task down into worker threads where each thread processes a number of spheres. Each worker individually figures out which sphere is closest to them and records the data in a result array where the index is there unique worker id. Once the workers have finished, we have a array of closest hit spheres for there respective group, and the recorded distance. Lastly to find the closest sphere to the start of the laser, we loop through the found spheres and get the closest one.

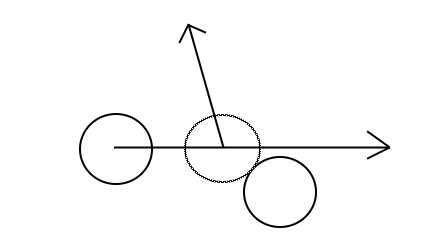
Sphere to sphere collision

Sphere to sphere collision was a issue when first starting this project. At first, I attempted a brute force approach to get the logic down, but soon realised that N\*N collision checks on the main thread is was too slow. Furthermore, I was taking a naïve approach to the collision where I would move the sphere into the other spheres, detect that a collision was happening and move it back out, but this is not accurate.

To start addressing the problem, I needed to modify the data set so that I was only checking spheres that were close to each other, to do this I implemented a sorting algorithm that sorted the whole scene by a axis (2D used X axis, 3D used Z axis, refer to 3D rendering section on why this was chosen). Now that the data was sorted, I needed a way to split the scene up and decided on a grid system where I would have ‘N’ grids side by side that overlapped each other slightly to account for spheres that were between grids (check Wobbly Scene Splitting for details on how it was done). Since the data was now in grids, I would check each grid and it would only have spheres that were relatively close to it on one axis. Next as a further optimization, I used the other two axis as well as the velocity component to quickly calculate that if the sphere was to move along its velocity, could it in theory reach the other sphere, if not, this was a quick break out of the collision.

Now that I had the framework of a quick collision detection, I needed to decide on a algorithm to implemented and settled on the one described by (Joe van den Heuvel, 2020) where we are able to calculate if the sphere was to hit the other sphere, but also where would it land after the collision.

For this implementation we first need to calculate that if we hit the other sphere, where along the velocity vector would we hit them.



Once we know where along the velocity vector we are going to hit the sphere, we can move the sphere along the path so that it is touching the face of the other sphere, but now in the same frame, we need to handle the bounce. To do this, we can use the vector between the spheres and the input velocity vector to calculate a reflection vector. Then finally using this reflection vector, move the sphere the final distance it would have moved this frame.

Sorting the sphere is order as well as using multithreading really helped improve the performance of the project as it broke the task down into very manageable workgroup sizes.

Vulkan Visualiser

Rendering within the project, I chose to use the vulkan api. The reason for this was for its ability to be multithreaded easily as well as its low level data access making it easier to optimize memory management. An example of this that is used within the project is where we create a buffer on the GPU to store the sphere positions. We can get a raw pointer to the memory on the GPU as since we know where our memory is within the array that’s on the GPU, I could split the righting of the memory between multiple threads. Furthermore Vulkan was chosen due to its ability to support indirect drawing where we give the GPU a pre packaged list of models we want to render and instead of us sending a model draw request for every model, we can give it a position array and tell it to render ‘n’ models reducing BUS traffic between the CPU and GPU.

Alpha issue

Since I’m using sprites for my sphere (See Sprites as Spheres) a issue that became prevalent was sprite ordering and out of order sprites being rendered on top of each other, because of this sprites would often have corners missing out of them that was easily noticeable.



To resolve this, I hijacked another optimization I was doing and changed it slightly to help fix this issue. To get the grouping working I had to order the spheres by a axis, for 2D work I did this by the X axis, but it did not make sense doing it by the X axis again for 3D work as if I ordered it by the Z buffer instead, I could render the spheres one at a time in order and fix my alpha rendering issue.

Optimizations Used

DOD Data Packing / Cache misses

To help mitigate agonised cache misses within the program I packaged all commonly used data together but in there own arrays, for example, when doing collision detection, I can normally determine quickly if a collision will happen based on one axis, if that axis is way out of range, I can ignore the collision. Because of this, having all of one axis in the same array really helps to mitigate agonised cache misses and it will increate our chances of not missing as for example a chunk of X positions will be ready in the cache for the next loop.

Furthermore a attempt was made to reduce cache misses in function calls by ether removing simple maths functions all together or inlining them, so that the CPU dose not need to do a new lookup for the code, hopefully as it is processing the current code we are in, the next bit of code is already in memory and waiting.

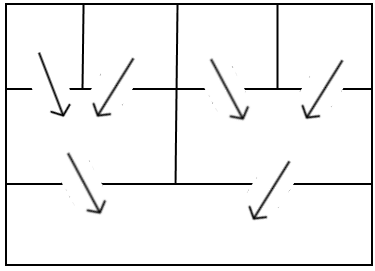
Multithreading

Worker threads as mentioned before have been a big part of the project as they have helped split up a task that normally would be ran on one thread, between multiple threads. To do this we take a task that can be ran on a data set and split it up so that each thread works on a range of data, but we need to make sure that none of the data needs each other otherwise we will need to start adding in mutex’s and that could cause race conditions, but more importantly, slowdowns.

Multithreading - Sort

One part that has seen the most benefit from the multithreading is within the sorting of the spheres along one of there axis. To do this, I settled on splitting up all the spheres between ‘n’ threads where ‘n’ MUST be a power of 2 and run bubble sort on each range. The reason bubble sort was chosen here over quick sort is that bubble sort proved to be faster in this case as the data set is already almost sorted and the complexity of quick sort was not worth it.

Now that the data is sorted in each worker group, the whole array needs to be sorted, to do this I take a pair of worker ranges that are next to each other and I merge there sorted data, since they are already sorted lists, this proves to be trivial. Once this is done, we repeated the process again and again until you are left with two final arrays that you then merge together in a final pass. Since we can not read and write to the same buffer for this merge pass, it was nessesery to double buffer the array storing this data but since we used a lookup table (See Lookup Table) the memory inpact is only “sizeof(unsigned int) \* sphere\_count \* 2”



Shows the process of taking two pre sorted and merging them into a larger sorted array

Loop unravelling

Bit Shifts

Wobbly Scene Splitting

Line Sweep

Lookup Table

Cache miss mitigation

Branch prediction fail mitigation

Early loop breakouts

Unions and Arrays

Inline

Rendering whilst simulating

We simulate the scene whilst rendering

Rendering

Rendering – Indirect Drawing and CMD buffers

Sprites as spheres

Fake lighting

# Bibliography

Joe van den Heuvel, M. J. (2020, 03 30). Retrieved from https://www.gamasutra.com/view/feature/131424/pool\_hall\_lessons\_fast\_accurate\_.php?print=1

Thompson., J. (2020, 3 30). Retrieved from http://www.jeffreythompson.org/collision-detection/line-circle.php