

**This is the work of a student with specific learning difficulties.**

**Please mark in accordance with the guidelines.**

**Student ID No: G20713365**

**Course: Masters Game Development**

Please include this coversheet with any piece of course work handed in for marking. Tutors will refer to the SpLD Marking Guidelines when marking your work. Please contact [inclusivesupport@uclan.ac.uk](mailto:inclusivesupport@uclan.ac.uk) if you would like a copy of the SpLD Marking Guidelines.

Introduction

For this assignment I have implemented an 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.

With the way the scene is split up, during the first minute of the program running, the spheres move away from the centre of the map and move towards the outside, because of this each worker is able to process collisions more effectively and performance boosts of around 400-500% can be seen.

Example Projects

Several example projects have been produced for this project which are located within “Project/Bin/\*.exe”

The projects are built with 38,400 spheres and depending on the file name, defines what they have enabled/disabled.

2D/3D – Is the demo project 2D or 3D

Collision – Is sphere to sphere collision enabled on the project

Visualiser – Is the Vulkan visualiser enabled.

Realtime – Are the spheres moving using the real time clock

Controls

Scroll Wheel: Zooms the camera in and out of the scene

Build Settings

At the top of the file is several hash defines used to configure the project, below is a description of what they do.

THREE\_D: Three D enabled the Z axis throughout the application, this is both for the simulation movement / collision as well as rendering.

VISUALISER: This is used to enable / disable the rendering of the scene as well as enable / disable the window opening.

SPHERE\_TO\_SPHERE\_COLLISION: This toggles sphere to sphere collision on and off, if off, we use a default movement system that only bounces of the walls of the scene and ignores collision.

SPHERE\_DEATH: if enabled, when a sphere hits 0 or less life, a message will appear in the console letting you know of its death. Also the sphere will not be considered for future line to sphere calculations or be rendered.

SPHERE\_COUNT: Sphere count defines how many spheres are allowed in the scene. This value must be;

* (SPHERE\_COUNT / WORKER\_COUNT) % 10 == 0: For loop unravelling
* SPHERE\_COUNT % 8 == 0: For sorting

TARGET\_FPS: Target fps is what fps we want to achieve as a maximum, this stops the renderer rendering every logic update.

WORKER\_COUNT: Defines how many worker threads we want

AREA\_SLICE\_COUNT: How many areas we want to break the world up into. This value must be less then or equal to WORKER\_COUNT

DELTA\_TIME: This makes the spheres move the velocity vector every second rather than every frame.

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.

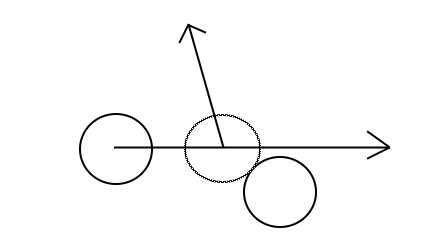
Sphere to sphere collision

Sphere to sphere collision was an 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) an 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.

Another place where CPU cache miss mitigation was used a lot was within the scene setup where we are initializing all the sphere data. I was very cautious about using a function call and worried about its performance penalty, so I made sure that all calls to “Random(float min, float max)” happened in the same place so that the CPU could cache the simple function and reuse the pre cached result.

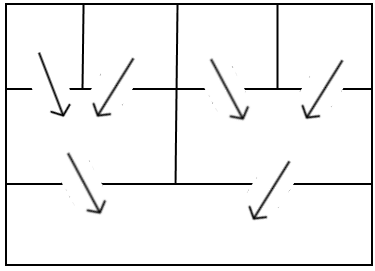
Multithreading

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 necessary to double buffer the array storing this data but since we used a lookup table (See Lookup Table) the memory impact is only “sizeof(unsigned int) \* sphere\_count \* 2". Using the double buffer, we read from one global array and write to the other, then back again like a swap chain within a graphics renderer.



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

Loop unravelling

For parts of the project, a optimization used was loop unravelling where we take the contents of a for loop and preform the same action multiple times for multiple spheres and then increment the counter based on the amount of spheres processed in the loop rather one at a time. This has the added benefit of allowing the next spheres code to be cached as we need to wait for the result of a comparison otherwise to determine if we go back into the loop. Because of this we reduce the amount of comparisons done (the one that sees if we should stay in the loop) from N comparisons to N / Loop unravel amount.

A issue that comes with this is one where if we have loop unravelled for example 10 times, we need to make sure not only is the total sphere count a multiple of 10, but that if a task is split up between threads, that the result worker pool size is a multiple of 10 also. To validate this, staric\_assets have been added into the start of the main function to provide an error on launch if the sphere count is in the wrong range.

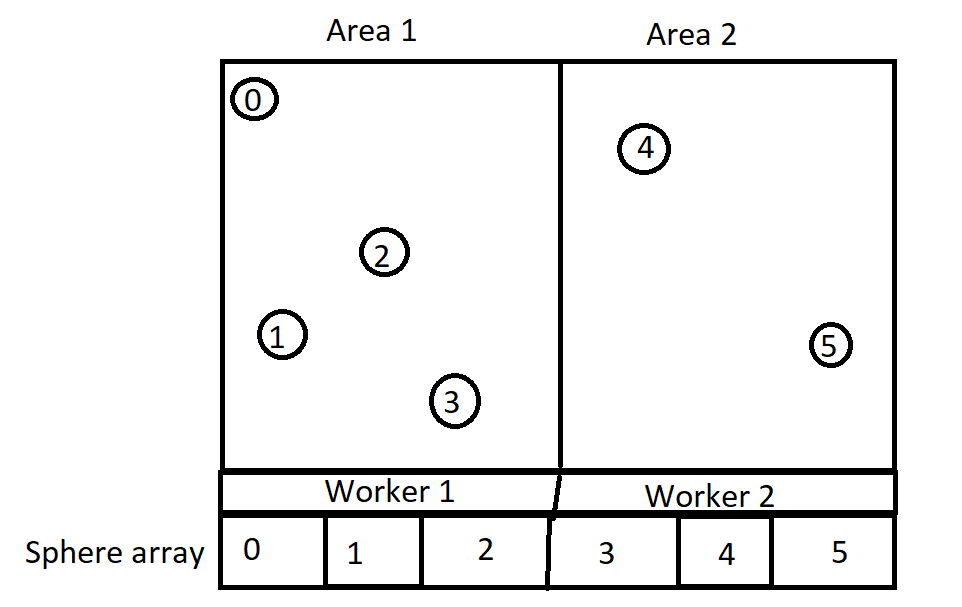
Bit Shifts

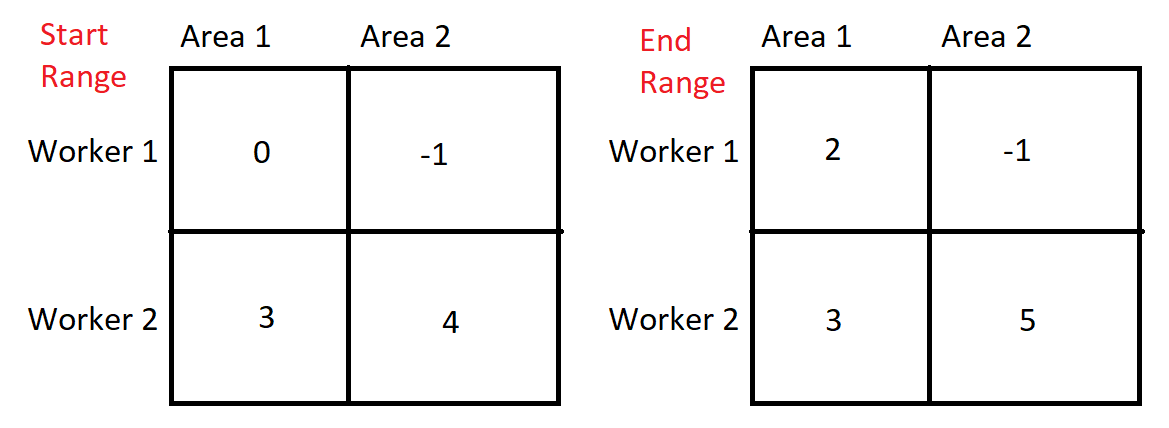
Within C++, multiplication factors that are a power of 2 (2,4,8,16,etc) can be optimised by using a process called bit shifting. This is where instead of multiplying the result, we can instead shift the physical bits in memory to the left or right, doubling or halving the result (if we shift them 1 place left or right respectively). A example place where this is used is within the “WorkerSetupSimulation” where we give all the workers a start index to loop through for the colours. The colour data is stored in RGBA format, so they will come in multiple of 4’s. Because of this, we can use the offset and bit shift it two times to the left and multiply it for 4 with little performance overhead.

Wobbly Scene Splitting

Scene splitting was used to help optimize collision detection within the project. The concept is that we want to group as few spheres together as possible, but make sure we only have spheres together that could collide, then using this data, we can check only spheres that could theoretically touch each other in one go and skip all the others.

To do the scene splitting, we first run workers and get them to find what spheres are in what area boundary. Since the data is sorted, if we find one sphere that has moved onto the next boundary, we can assume that all the spheres following will ether be in that boundary or subsequent ones, so we have found the sphere we think “starts” this boundary. If we find a sphere that is in a new boundary, we take our last sphere result and use that as the sphere that we think “ended” the last boundary.

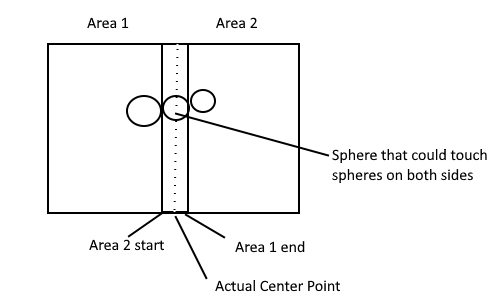
In this example we can see that in area 1 we have 4 spheres, area 2, 2 spheres and each worker will be processing 3 spheres each. From this each worker has a array of ‘n’ elements for the start and end boundary and they put sphere id’s in if they think there sphere fits into the boundary.



Once we have found the results (-1 being used for when a valid solution was not found) we can work out the area boundary’s trivially. For the starting range of the boundary’s, we loop through each worker and the first worker we find with a valid result MUST be the correct starting position. In this case if we loop through worker 1 then 2, we see we get index 0 then 3. From the diagram we can see that indeed index 0 is the first element and that’s why we ignore all subsequent ones. In the case for area 2’s starting index, worker 1 could not find one but worker 2 did find one, so we take its result.

For the end ranges we do things a little differently, we loop through each workers result taking and we take whatever is the last valid result, so in the case of area 1 we have 2 and 3 (from worker 1 and 2). Since we take the last valid result, we see that the value 3 is indeed the last sphere in the area.

This area boundary method is not perfect though so I ended up having to add a little wobbling to the area boundary’s so that they overlap slightly to account for spheres that could be between areas, to do this, I took the start boundary’s and end boundary’s and +/- half the max radius of a sphere from them so that if a sphere was to be in between, we would process it in both groups.



There is one downside to this method that is apparent in the initial program performance and that is when we first start the bounding box surrounding the area is much bigger then the spawn area, so because of the high concentration of spheres in the middle, we have large area groups making collision detection a bit slower, but as time passes and they spread out, this issue subsides.

Line Sweep

Since our data is already pre sorted to help with Sphere to Sphere collision as well as fixing alpha sorting, we can use it to help speed up line to sphere collision also. To do this, we check to see if the sphere we are going to be checking is close enough on the sorted axes (ether X or Z depending on 3D) to touch it. If we have passed the line, we can assume all further spheres are too far away from the line also as the data is sorted, so we can just brake out of the search speeding up the algorithm.

Lookup Table

The lookup table was used in the project as a method to get around having to sort many arrays each frame. Since we sort all spheres along one axis each frame for various optimizations, we would have to sort several arrays in the process, such as the X, Y, Z, Scale, Health, Colour and Name arrays. To get around this, we can use a lookup table, this table contains a list of lookup index’s that are used to find spheres, for example index 0 in the lookup array wont point to sphere 0, but instead the sphere that is furthest along the X or Z axis based on rendering mode. Using lookup tables allow for much faster sorts as well since the problem can be broken up into worker tasks.

Branch prediction fail mitigation

Branch perdition can be a issue and attempts were made to mitigate agonised it. Since we are sorting the scene in order of spheres, when preforming such optimizations like line sweep, since we have the early breakout, it means that we do not need to have many branch fails at all as it will be true for every sphere in theoretical range, then as soon as its impossible for spheres to touch, its breaks out making it ‘n’ successes and 1 fail. This was taken into consideration throughout the program where it was possible, but its hard to fit branch perdition into every check as it will fail on occasion.

Unions and Arrays

Since we know how the data within the system, we could use unions to split one large data array into multiple smaller ones for each components of the spheres position. The reason for choosing to do this rather then have 4 separate arrays for the spheres X, Y, Z and scale is because since it is all in one array, it made it easier to transfer the data over to the GPU in one large chunk, rather than in 4 separate passes. Because of this, we are able to lower memory fragmentation on the CPU and GPU memory as well since the buffer is in one large chunk rather then in 4 smaller ones with potential memory wasted in between. Furthermore since we have the option to change between 2D and 3D, using hash defines we can dynamically insert the Z axis between the Y axis and scale. This helps with GPU side memory processing as well as being a logical format of the memory.

static union

{

struct

{

float X[sphere\_count];

float Y[sphere\_count];

#if THREE\_D

float Z[sphere\_count];

#endif

float Scale[sphere\_count];

}spheres;

float SphereData[sphere\_position\_array\_length]{ 0.0f };

};

Inline

Inline was used where possible to get the system at compile time to instead of creating a function, instead copy the code and directly add it into the calling function. This dose increate overall application memory, but lowers the caching needs of the CPU. In some cases it was found that the system was not inlining as it did not find it necessary, so \_\_forceinline was used in its sted, if this failed (even though it says force...) I removed the function and directly added it to code.

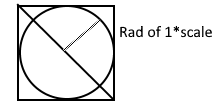
Rendering whilst simulating

Since the rendering of the scene can take a short amount of time, it gives us some time to do further CPU side processing as we wait for the results, because of this, I update the simulation after sending the most recent one to the GPU, then once the new simulation is compiled, we wait on the GPU to finish (but since the GPU is only rendering primitives, it is very fast and is normally done half way through the simulation update).

Rendering

A process within the rendering called indirect drawing was used to help speed up the performance. The way this works is, normally for each model we want to render, we send a separate draw request for over the GPU. Because of this we are left waiting for the GPU response for each GPU draw request as well as waiting on the delay of message transfer over the system BUS. A way to get around this in Vulkan is pre recording the scene drawing and store this on the GPU, then when it comes to rendering, we tell the system we want to render ‘n’ spheres rather then each sphere ‘n’ times. Because of this, we tell the GPU in one message, to render all the spheres in the scene reducing wait times.

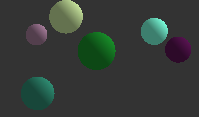
A further optimization used within the renderer was faking 3D spheres. The way this was achieved was because of the unique requirements of the solution. We have a static camera (or a camera on a Z axis dolly in my case) so there is no dynamic lighting. Since lighting/shading on a spheres surface gives it the illusion of 3D, I instead opted to draw a square and fake lighting on its surface to give it a 3D look. To do this, I gave the quad UV coordinates of a range -1 to 1 rather then the traditional 0 – 1 that’s used for normal texturing. Because of this we were able to note that the centre of the quad would be UV 0, 0. Now when we render each pixel, we use the distance from the centre to determine if it’s a part of the sphere, if its length is more then 1, we must be off the spheres surface, so render alpha 0.



Now that we have a circle being drawn, we need to give it 3D shading. to do that I create two const vec2 that tells me the position of the light on the sphere as well as the lights power min and max. Next I multiply the UV with the light position to give me a position relative to the models fake normal. Finally using this we generate a colour power by adding running;

Colorpower = (light\_position.x + light\_position.y + 2.0f) \* 0.25f

Here we get the x and y and 2 and add them together (2 is used because the UV could be -1 – 1 so its to bring the total value between 0 – 4) then we multiply the result by 0.25 to give us a 0-1 range. After this, we clamp the resulting light power by the min, max and we have our final surface shading.



Conclusion

In conclusion I am happy with how the project came out, the only thing I am a little disappointed I could not work out is when you have two spheres collide and they are moving in the same direction, the one that collides from behind dose not enrapt any extra force on it making some of the collisions look strange, as there is no transfer of energy between them, but this was not within the assignment scope, so was left out.

# 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