An Evaluation of Entity Component Systems to Create Spring/Mass Soft Body Dynamics in Real-Time

Mark Craigie
Bsc Honours Computer Games Technology
School of Design and Informatics
2019

Abertay University

Table of Contents

Table of Figures

1. Abstract

Most modern video games base their principles and worlds on real-world physics and properties, even exploring how they can be exploited in creative ways to make interesting mechanics and appeal to a consumer. Our world is built up of destructible, bendable objects that may in fact be computationally expensive to recreate in the digital world using the current technologies at our disposal, particularly when considering that there is an end consumer, as video games are an entertainment form, whose users may not have the latest and most expensive hardware at their disposal. A key distinction between video games and other software applications is the need for video games to be calculated and rendered in real-time, therefore they do not have the luxury of taking an extended period of time to simulate physics before showing an output.

As programming within the context of the games industry advances, other architectures and data structures are being explored for aiding in optimising applications and systems. Data-driven programming and more specifically, Entity Component Systems are methods of accessing large amounts data and doing calculations rapidly due to its ability to run many different calculations in parallel more effectively than the games industry standard of object-orientated programming.

The purpose of the research conducted is to determine if Entity Component Systems can aid in optimising soft-body dynamics so that they can be more applicable within a real-time scenario. The results showed that Entity Component Systems are in fact able to manage larger amounts of data and calculations, increasing performance, but only in certain situations, meaning that further work and optimisations will need to be applied to the systems explored within this application.

2. Introduction

While programming is seen as a science, games are a creative branch that uses computer science as its tool and therefore requires applications to appear and interact in creative and interesting ways to a consumer or player. The use of physics simulations to re-create the world around us within a computer application is necessary in painting the landscape of the game if the player is to feel immersed properly. As the games industry progresses, developers are looking for more interesting mechanics as well as more advanced and realistic simulations. Advancements in technology available to the consumer is allowing for more intensive operations and complex applications. Physics simulations can demand a large amount of computations and therefore are becoming increasingly appealing to use as technology evolves. Rigid body dynamics have been the focus of game engines for a long time as most physics simulations can be replicated fairly accurately using rigid bodies as well as being very performance efficient (Testing suggests around 3000 rigid bodies on a basic PC) (Brackeys 2017). However, as the demand for realism continues, rigid body techniques cannot fully simulate how certain aspects of our real-world interact. A soft-body (also referred to as a 'deformable object') is used to replicate any sponge-like body in the world, such as tissue, muscles or clothing but can also be applied to create fluids.

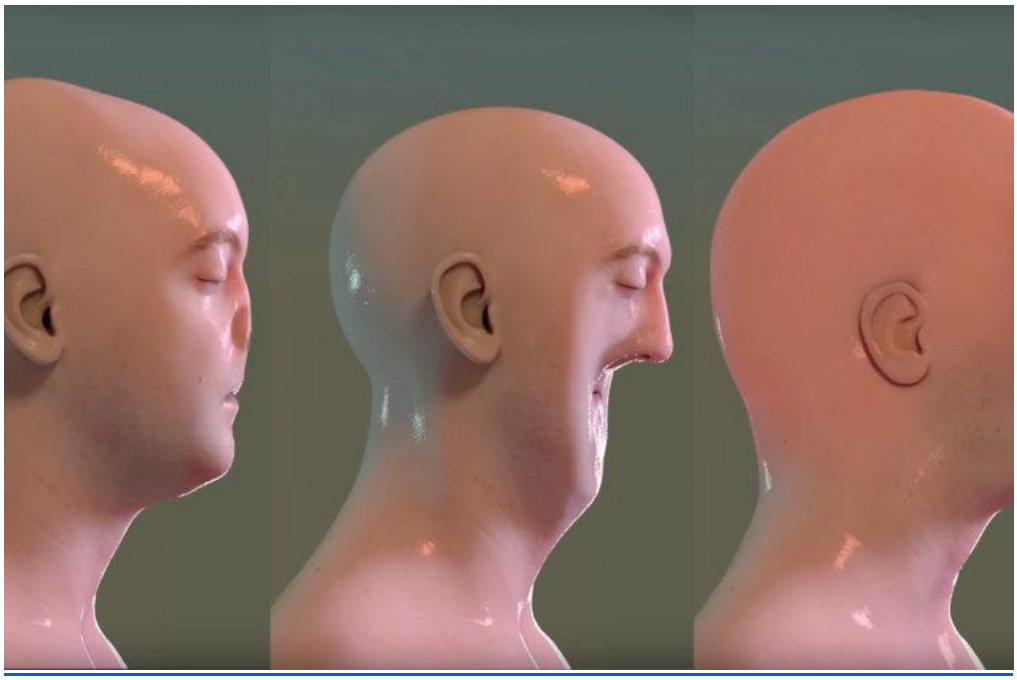

Figure 2-1 - Soft body deformable model face

When considering how best to simulate soft-body dynamics within the context of games, or any real-time application, performance must always be prioritised. Within the games industry a user is always present and therefore any physics simulation must remain consistent but a small amount of accuracy can be sacrificed in order to keep the application efficient.

A form of data-driven programming is an Entity Component System (ECS). Entity Component Systems are methods that allow for large amount of “entities” which are handled by the system as data points, whose components can be easily modified at runtime and therefore are far less complex, allowing for improved compute times. Tests have shown 24,000 more objects executing the same script at the same fps using ECS than a traditional object-based method within Unity (Geig 2018). Each ‘System’ only requires the data from each entity that it will be reading/writing, allowing the systems to work independently from each other and preventing large amounts of information from being passed around that isn’t needed to be accessed. The concept of Entity Component Systems are fairly new within programming and as such, the one that will be used within Unity’s (ECS Unity Technologies) engine is an experimental feature which is still in development and has not been tested thoroughly. For this reason, there is not much documentation on ECS or many components so almost every component will be made from scratch, meaning the same will be applied to the Object-orientated version (i.e. not using Unity’s Components) so as to make sure that the two versions being compared are using the same components and systems.

2.1. Aims and Research question

How can the performance of real-time soft body dynamics be improved by using data-driven programming techniques instead of inheritance object-based methods?

The aims of the research and development are primarily to create a realistic soft-body application that simulates within real-time. Understanding the performance restrictions of various methods within the standardised object-based programming along the way so as to then replicate the soft body simulation using data-driven

techniques (ECS) and compare the differences in programming complexity and optimisations between the two methods.

This dissertation will begin by exploring previous methods used for simulating real-time soft-bodies and the different principles used, to gain an understanding of what methods should be explored and tested. It will also look at how other applications simulate soft-bodies such as animation programs to see what methods are currently used by software programs that do not run in real-time. Following this is a description of the methods taken to build the application and the testing of various principles undertaken during the development process to demonstrate what steps were taken and which methods were dismissed for more optimal solutions so as to gain a better understanding of the programming complexity and difference between object-orientated and data-driven methods. The results are outlined afterwards and then discussed in detail, in the context of its application to the games industry to ensure that the aims of the project can be answered.

The hypothesis of this research is that when increasing the number of mass points and springs in order to achieve a more realistic and accurate simulation, the data-driven methods will begin to perform more efficiently than its object-orientated counterpart and therefore prove that it is possible to simulate soft-bodies in real-time without having drastic performance issues.

3. Literature Review

3.1. Current Applications

Since games often rely on creating realistic physics simulations, Rigid body dynamics and their principles have been used in the games industry for a long time for example, the Havok Physics (Havok Physics) used in Half Life 2 (Valve 2004). However, rigid bodies are proving to be limited in their capabilities as the industry demands increased realism in real-time. Using various algorithms - such as Runge-Kutta calculations to solve the body and deform vertices by treating them like particles (Blender Foundation) - and methods to create soft bodies has been the forefront of testing for a number of animation programs such as Maya (Autodesk 2018), which have produced numerous examples of both realistic and pseudo-realistic animations for many different types of media. There are many differing techniques and principles to solve soft-bodies that can all be compared as there are various advantages and disadvantages attributed to each. For example, when using a Spring/Mass principle, the implicit Euler (Mesit 2010) method can be used for calculating cloths as large spring constants are required, which can be an issue when dealing with explicit integration methods. Finite Element method (Mesit 2010) is a way of deforming a body/mesh and is often seen as more realistic but comes with a large amount of complexity in its calculations. Due to the nature of games being needed to be calculated in real-time, a method that is efficient but also stable must be chosen, without the need for too much accuracy. Fluid simulation equations, such as position-based fluids or smooth particle hydrodynamics (Monaghan 1992) have been used, alongside particle systems to create soft bodies that are similar to how a body of fluid would appear (Weaver 2016). This technique has been proved to work, Nvidia's PhysX (PhysX) has been used to create large-scale fluid simulations but this relies on a large amount of particles that do not necessarily communicate between each other as they follow a set algorithm for the properties of the body of fluid as a whole (Barsegyan 2019) as well as having issues with real-time rendering meaning another method is needed if the body is required to be more of a solid as opposed to a liquid and for real-time use.

The principle of Spring/Mass is to connect many 'Mass points', generally of a uniform size and mass by a set number of springs, with standard spring properties (such as

spring strength, rest distance and dampening). To simulate a soft body, each mass point is then essentially pulling and pushing the rest of the mass points in the body through the series of springs connecting the whole body.

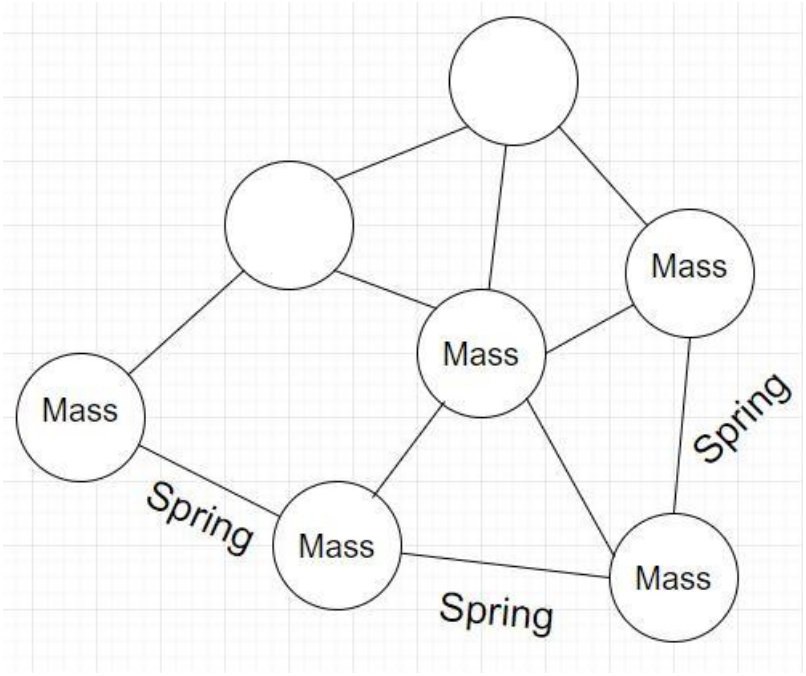

Figure 3-1 - Mass/Spring model example

An example of soft-body dynamics that has been used in animations for numerous media types, such as TV shows, films and adverts is the animating of cloth. Many techniques have been tested when creating these animations (Yalçın 2019), the model can be treated as a number of mass points that build up a mesh of triangles almost how like a real cloth would appear. Various integration techniques are used to solve the algorithms that create the forces which move the cloth during the time of the animation and these sets of equations can be extremely complex, leading to renders taking a long time to complete. The Mass-Spring model has been used to simulate cloth behaviour yet is less accurate as “woven clothes are far from having ideal elastic properties” (Yalçın 2019), and need “small sized time steps to simulate cloth accurately” (Yalçın 2019) when representing cloth. It has been explored to use a particle system to calculate the mass points and then connect each by springs and then also adding a ‘stiffness algorithm’ calculated to maintain the structure of the cloth. While this method may not have been the most accurate for an animation, its reduced complexity in algorithms means it can be investigated for use in real-time applications.

BeamNG (BeamNG 2015) is a realistic real-time car simulation game using soft-bodies for its vehicles, describing their soft bodies as having a node-beam structure, which allows for a ’skeleton’ of an object to be created through a network of nodes and beams. This is the spring/mass principle of having a mass point or ‘node’ that is connected to other mass points by springs or ‘beams’ that have various properties to determine the amount they will move the nodes around them. BeamNG only deals in realistic car simulation and uses only fairly rigid springs to create the components of a car, therefore does not need to have a large number of spring and mass points. If realistic and softer bodies wish to be created, the number of springs and mass points must be increased, which would demand a lot more computations than BeamNG is currently performing. The basis of the spring/mass principle however is proved to be effective in real time.

3.2. Issues with Games

It has been proven that changing the current approach of using very simplified kinematic approaches is not as effective as using more physics-based approaches (Yeh 2018). In games, performance is normally seen as the main concern and therefore efficiency is prioritised over accuracy, leading to simplified models. Because of this, kinematic techniques were favoured, however, as models and environments become more complex, run-time physics calculations are becoming more favourable. (Yeh 2018).

Parallel physics simulations have been seen to improve the performance of many applications and moving onto data-orientated design goes to show how multi-threading can be taken advantage of. (Yeh 2018) GPU parallelisation has been researched as a method of solving soft-body dynamics calculations, using thousands of equations and constraints to deform sets of vertices. One example of research (Fratarcangeli 2016) uses the Gauss-Seidel method to optimise computing a large system of complex graph equations on the GPU. The paper proves that the method is effective in animations, if the number of calculations were to be reduced as an optimisation, approximations would have to be used, causing artefacts and large inaccuracies to occur therefore meaning this method has too many issues to be applicable in real-time due to the number of calculations needed to be done each

frame. This method does however prove that GPU multi-threading is a viable technique to explore for increasing performance. With the way that hardware is advancing and the future of graphics cards, it seems that parallel programming will be used more for all forms of computing, not just within the games industry. Fabian agrees with this sentiment, and even goes as far as to say that it will define how the design and structure of programs will adapt to these advancements in technology as memory management will still be as prolific of an issue but that future hardware will be “battles of the parallel processing beasts” (Fabian 2013). This leads us to look towards alternate designs for programming systems.

One of the main hurdles for real-time physics simulations as a whole tends to be collision detection. This is especially an issue for soft-body dynamics as the collider cannot have fixed dimensions as the body morphs in shape. A common method for solving collision within games is using an axis-aligned (Miligton) bounding box (Each axis has a maximum and minimum value that defines the area in which the ‘box’ collider exists) and comparing each axis-aligned bounding box (AABB) to every other in the scene. Generally when using a single large collider for the whole body the AABB has to be subdivided and modified as the body deforms, which causes more complex collision detection techniques (Mesit 2010). AABB collision is still viable as each mass point can be treated as a separate collider that will not need to be deformed but collision detection may still be a major cause of issues within real-time rendering unless other architectures are explored to aid in the compute time.

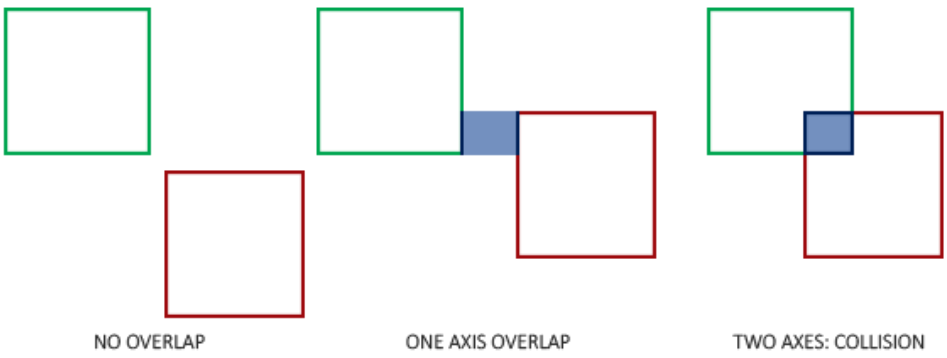

Figure 3-2 Axis-Aligned Bounding Box theory

3.3. Data-orientated Design

Data-driven programming is the base idea behind component systems and uses the principle of manipulating pure data rather than creating classes that inherit and then handle data. (Wallentin 2014)

The comparison of object-orientated systems and data-driven systems has been important in recent years, specifically between entity systems and the benefits behind different structures (Wallentin 2014). An ‘Aggravated’ Component-Based entity system, which attaches components to entities and then refers to each component by an assigned number is not dissimilar to the structure of the Spring/mass principle. The performance of this method is the most effective of Entity Component System types as it iterates through memory so efficiently (Wallentin 2014) but from a programmer’s standpoint can be difficult to implement due to the logical complexity. This may be less of an issue as the soft-body application will be dealing with how efficiently it can iterate through data, rather than having to use the modularity of entities and their components, even though this is one of the major benefits of using such a system.

There are not many examples of Entity Component Systems in use within the games industry but there are some applications that have been developed that demonstrate the principle of these systems and how they work efficiently. Simple flocking algorithms to simulate birds flying or basic ‘AI’ bots that move to a target position. (Boids Demo 2018). Data-driven programming structures can be explored as to how they have been used to improve real-time applications.

A data-driven engine test observed that when handling large amounts of data, particularly data that would be overwritten each frame, data-orientated design was particularly efficient. These sorts of operations are a flaw of large-scale object-orientated structures with the data being essentially obscured through all the hierarchy levels. (Danielsson 2017). Creating Systems for these data-design structures allows for components to be read and written independently and for all the logic to be computed without the need for accessing unnecessary information. A MMOG server middleware application (Martin 2007) has been using entity systems within a games context and proves how Entity Component Systems are entirely feasible for use within the games industry.

Physics simulations such as soft-body dynamics have been important within animation programs and other rendering software for years and many methods have been tested already due to this. The main restriction with real-time is the ability to maintain stable performance and the emergence of data-driven methods within programming systems and architectures can be explored to help with this.

4. Methodology

4.1. Intro process to the method

Before being able to develop the ECS version, the first aim of the investigation was to test soft-body principles and create an application to ensure that the methods researched worked within real-time as well as to create a benchmark for testing and comparing the two programming architecture approaches. During the development process, a number of principles were still tested and modified to end up with an appropriate outcome for the different versions, however, for accurate and equal results, less efficient methods (such as the collision detection) were sometimes used to ensure both the object-based and data-driven versions executed the same methods and calculations. The first steps conducted serve to create the basic spring/mass principle within real-time, before then exploring how to optimise and change the methods used so as to run the calculations for the simulation within both object-orientated and data-driven versions.

4.2. Soft Body Principles

The method being used in this application treats the whole body as if it is built up of many small points and connects them between a series of springs. Each mass point has its own movement velocity, affected by the forces enacted on it by the springs as well as any collision that is detected that may restrict its movement. Hooke's law declares that the distance a spring is compressed or extended is directly connected to the force required to hold it at said distance. (Cocco 2010) The base formula being Force = K x D (where K is a spring constant and D is the distance from its rest point). The springs use the same method in every variation tested, all abiding by the principles of Hooke's law and executing the code as follows. The distance between the current mass point and the target is stored as well as a direction vector pointing towards the target. A compression value is calculated from the distance between the mass points and the rest distance the spring should be at which is multiplied by the spring constant value. This combined value is the base force of the spring but is also subtracted by a relative velocity between the two points and multiplied by a damping value to make the springs pass velocity through

the body. Finally, the force value is applied to the mass points in the direction of the target mass point.

```
//find distance to base spring
float dist = Vector3.Distance(massPoints[comparison].position, massPoints[i].position);

//direction to base spring
Vector3 dir = (massPoints[comparison].position - massPoints[i].position).normalized;

//compression
// distance from other mass point, allowing for rest dist
float compression = dist - springRest;

//create relative vel between two mass points
Vector3 vel = massPoints[i].velocity - massPoints[comparison].velocity;
float relVel = Vector3.Dot(vel, dir);

//f = kx (force of spring) - relative velocity between the two masses * dampener
float force = (springConst * compression) - (relVel * springDamp);

//push force on mass in correct direction
//dir * force
totalForce += dir * force;
```

Figure 4-1 - Spring calculation code

Various collision detection techniques were tested at the beginning of the application, firstly using a single polygon collider to cover the whole body that had its vertices deformed based on the position of each mass point. This method was tested in conjunction with a way of creating the springs upon creation of the body, using the mesh data from a collider mesh and initialising a mass point at each then creating a spring along each mesh triangle's edge. The polygon collision proved to be inefficient and unstable so was replaced by having a collider placed at each mass point, meaning each mass point could also handle its own collision detection directly. Therefore, in both Obj and Ecs versions, mass points have a box collider stored that is used for detecting AABB collisions.

The mass points are tested for collisions against any mass point that is not part of the body it is in as well as any objects in the scene. The normal of the collision is created and used to remove the component of the velocity that is making the two objects collide. The collisions also allow for moving objects to transfer velocity so that if the soft body is pressed against a flat rigid body for example, the soft body will move with the rigid body, effectively pushing the soft body along. Elastic collisions were implemented and tested between two soft bodies, using the mass points'

collisions but the results were not consistent enough and therefore was removed meaning soft-body to soft-body collisions were not tested for realism within the results of this application.

As the pre-calculated springs often had issues with the body collapsing, a method of recalculating springs at runtime was tested. Each mass point's position was compared to every other mass point in the body and a distance was calculated between them. If this distance was less than or equal to a set amount then a spring was created between the mass points. This closest points method allowed for the body to morph shape easily and adjust after collisions but the body quickly deformed too much into a ball shape. In order to make sure that the body would collapse and expand properly this closest points method was adapted so that the more springs that were connected to a mass point, the more the spring constant value was reduced to effectively weaken the strength of each spring, maintaining an average spring force being enacted on a single mass point at any given time.

The structure of the soft body and the number of springs had a large impact on the appearance of the body and so a nearest-neighbours version was tested. The nearest-neighbours system involved testing each mass point for the closest X number of mass points surrounding it and created springs between these points, replacing any existing springs between further mass points. Various neighbour numbers can be tested and show how the body's structure can be changed by allowing for more or fewer springs between mass points. The neighbours system in the Entity Component version goes into this in more detail.

4.3. Object-Orientated Method

As part of the feasibility demo, submitted at the end of the first semester of this process, the soft-body principles were developed into an application, using Unity3D to render the mass points and standard C# scripts to store and complete the calculations to manipulate the mass points.

For the original basic version to test the principles of spring/mass bodies, a single script was used and attached to the body. Each mass point was spawned in and stored as a simple struct (Struct 2015) that stored each mass point as a position, velocity, acceleration, force, mass and colliding normals. During each frame the

mass points were looped through and their forces were calculated based on the springs attached which then was used to calculate the acceleration and velocity of every mass point. Collision normal for each mass points was determined by the collider attached to each mass point initially using OnTriggerEnter and OnTriggerExit, standard Unity3D functions which return when a collider has entered or exited another, returning the collision point. A normal for any collision was then made and stored at each mass point struct and before the position of a mass point was calculated, the velocity component of the mass point was modified by taking the dot product (Scalar Product) of the velocity and colliding normal to remove the direction of the collision. This method was used for stationary colliders to begin with as the collision solving does not force the mass points back in any other manner. This same principle was then compared to separating each mass point stored in the struct to a separate script and passing the nearest neighbours to each script to solve. Once this method of solving the positions of mass points was proved to work effectively, the collision system was replaced with an Axis-Aligned bouncing box method, to ensure that the collision system used was the same as the ECS version created and therefore not using any collider components provided by Unity to avoid unequal differences between the two versions.

4.4. Entities and ECS

Firstly the body needed to be converted into entities, treating each mass point as a single entity and attaching components to it. The application creates the mass points as prefabs (Prefab Unity 2018) in the wanted scene and then loads them into a separate scene asynchronously, before converting them into entities into the first scene and destroying the copy scene. This method was used to make sure the same body was used for both versions of the application and allows for as many mass points as wanted to be spawned in and converted easily. The mass point entities which had the base components after being converted from game objects (Position and MeshRenderer) were then attached with the body components.

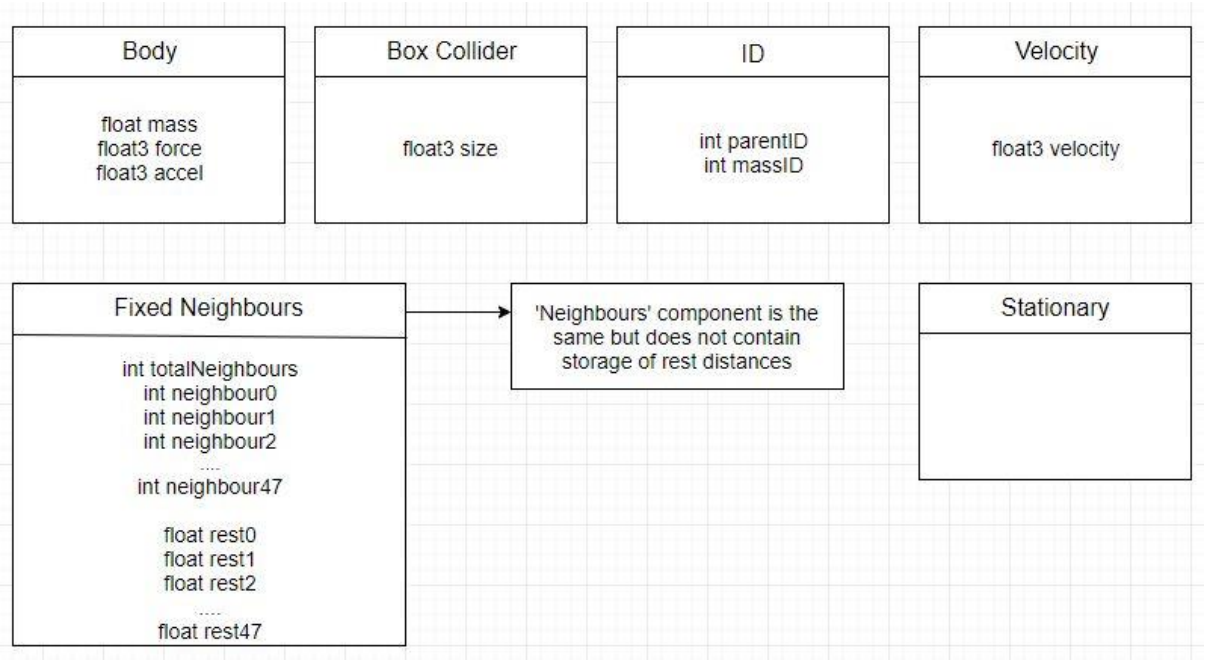

Figure 4-2 - Component architecture and data

Each component generally contains as little data as possible, to make sure that when the systems access the components, unnecessary data is not accessed when it is not needed and also allows for improved parallelisation. An example of this is the velocity data is in a separate component, rather than a part of the Body component even though it would make most sense for it to be contained within it. This is to allow for the velocity to be read and written by the collision job for example, as the collisions job does not need access to the mass of the bodies. One issue with components, however, is not being able to create dynamic storage types. This mean that an array, for example, cannot be used within a component, which meant that for storing the nearest neighbours they had to be initialised as a large amount of integers set to a null value until assigned to a neighbour ID, causing code to appear messy and long.

```
neighbours[33] = objectFixedNeigh[currentMass].neighbour33;
neighbours[34] = objectFixedNeigh[currentMass].neighbour34;
neighbours[35] = objectFixedNeigh[currentMass].neighbour35;
neighbours[36] = objectFixedNeigh[currentMass].neighbour36;
neighbours[37] = objectFixedNeigh[currentMass].neighbour37;
neighbours[38] = objectFixedNeigh[currentMass].neighbour38;
neighbours[39] = objectFixedNeigh[currentMass].neighbour39;
neighbours[40] = objectFixedNeigh[currentMass].neighbour40;
neighbours[41] = objectFixedNeigh[currentMass].neighbour41;
neighbours[42] = objectFixedNeigh[currentMass].neighbour42;
neighbours[43] = objectFixedNeigh[currentMass].neighbour43;
neighbours[44] = objectFixedNeigh[currentMass].neighbour44;
neighbours[45] = objectFixedNeigh[currentMass].neighbour45;
neighbours[46] = objectFixedNeigh[currentMass].neighbour46;
neighbours[47] = objectFixedNeigh[currentMass].neighbour47;

rests[0] = objectFixedNeigh[currentMass].rest0;
rests[1] = objectFixedNeigh[currentMass].rest1;
rests[2] = objectFixedNeigh[currentMass].rest2;
rests[3] = objectFixedNeigh[currentMass].rest3;
rests[4] = objectFixedNeigh[currentMass].rest4;
rests[5] = objectFixedNeigh[currentMass].rest5;
rests[6] = objectFixedNeigh[currentMass].rest6;
rests[7] = objectFixedNeigh[currentMass].rest7;
rests[8] = objectFixedNeigh[currentMass].rest8;
rests[9] = objectFixedNeigh[currentMass].rest9;
rests[10] = objectFixedNeigh[currentMass].rest10;
rests[11] = objectFixedNeigh[currentMass].rest11;
rests[12] = objectFixedNeigh[currentMass].rest12;
rests[13] = objectFixedNeigh[currentMass].rest13;
rests[14] = objectFixedNeigh[currentMass].rest14;
rests[15] = objectFixedNeigh[currentMass].rest15;
rests[16] = objectFixedNeigh[currentMass].rest16;
rests[17] = objectFixedNeigh[currentMass].rest17;
rests[18] = objectFixedNeigh[currentMass].rest18;
rests[19] = objectFixedNeigh[currentMass].rest19;
rests[20] = objectFixedNeigh[currentMass].rest20;
```

Figure 4-3 - Neighbours storage mess

4.5. Systems

A “System” in terms of ECS is similar to the idea of a script or class (Class 2018) in that it executes code line by line, each frame. However, the difference is that a system does not store data locally and instead only uses the data that it is passed in by the components, executing the calculations it needs to and modifying the data. The data passed into these systems can be set to “Read-only” or “Write-only”, so that the system is restricted in how to handle the data in order for the “jobs” – executable tasks sort of like a single iteration of a loop (Johansson 2018) - to work in parallel without attempting to read and write to the same data.

Initially the application was tested by using one system that found the nearest neighbours, created the springs, calculated the forces and collisions before then moving the mass points’ positions. In order to best optimise the application from here each system in the application was designed so that each major computational task could be ran without getting in the way of another, ensuring that only the data

required was accessed and of that data as much of it as possible was made to read or write only to prevent overlap and allow for parallelisation. The code was split into 3 separate systems (NeighboursSystem, ForceSystem and PositionSystem), with 4 jobs being executed in parallel (NeighbourJob, ForceJob, CollisionJob and PositionJob). The Neighbours system accesses 3 chunks of data, the neighbours component attached to each mass point, reading and writing information to it as well as only reading the position components and ID number components of the mass points. The code then iterates through all the mass points and uses its position component to find the distance between mass point, only comparing mass points within the same body. The shortest distances are then chosen and the ID numbers of the closest mass points are stored in the neighbour component.

The Force system contains the same spring force calculations as was used in all previous versions of the soft body application. It accesses the Body and Velocity components, both able to read and write data and Position, Neighbour and ID components are set to read only. The main difference which leads the force system to become harder to read is the accessing of the neighbour elements and their rest distances as the components cannot store arrays, therefore an array is created within the Force system and each element is assigned by getting the equivalent number from the neighbour component, as shown in Figure 3-3.

The Position system is the final step in the process and is responsible for moving the mass points in the scene to where they should be. The system contains two jobs, CollisionJob and PositionJob, executed one after the other in order to ensure that the collisions have been taken into account before moving the mass points. Velocity data for each mass point is set as Read/Write while the Position, Collider and ID data are set to Read only. The collision Job tests each collider in the scene for a collision with every other collider of a different body, using the AABB collision detection system and removes the component from the velocity that moves in the direction of the collided object. The position job then moves the position of each mass point using the velocity of the mass.

4.6. Variations

At this point a fixed neighbours method was created, in which the springs and their rest distances were calculated upon the creation of the body and then never adjusted. A reference to the other mass point that each point is connected to is stored as well as the starting rest distance between the masses so that the shape of the body can be maintained. This means that each mass point does not necessarily have the same number of springs connected to it, as masses on the edges of the body would be closer to fewer mass points. This method is essentially a midpoint between the nearest neighbours method as well as the closest points method previously explained. The fixed neighbours can also be modified to be more similar to the nearest neighbours approach by simply calling the nearest neighbours function once, and then storing the rest distances in order to make sure that each mass point has the same number of neighbours, even if it is on the edge of the body (i.e farther away than a mass in the centre). In order for the body to maintain its shape using this method, the spring constant is increased so that the springs are more rigid. This method saves the application from constantly calculating the distances between points to destroy/create springs and therefore is more efficient but structurally looks different, this is explored in the results and discussion chapters.

5. Results and Discussion

In order to accurately determine how efficient the application performs, how much of an impact the simulation has on the frames per second of the application must be investigated as the major differentiating factor between real-time and pre-rendered is how efficiently the program can run in a measure of outputting frames per second (fps). However, to gain a greater understanding of which processes are the least efficient and therefore causing the greatest issue in the application, the application must be profiled during runtime so as to see the effect on the CPU, GPU and memory as well as how long individual functions take to execute. Multiple simulations must be tested and recorded, varying the number of mass points and springs on the different methods to determine the benefits and disadvantages of them.

Each test was ran on a computer with the specifications below.

Processor (CPU)	Intel Xeon E3-1231 v3 @3.40GHz
GPU	NVIDIA GeForce GTX 1070 8GB
RAM	HyperX 16GB DDR3
Motherboard	MSI Gaming 5
Unity Editor	2019.1.0a14

5.1. Nearest Neighbours Update

A Frames Per Second comparison of the nearest neighbours method is useful even if it is only an overview as it shows how applicable real-time use of this form of soft-body is. Figure 5-1 is generated from tests ran both in the object-orientated version (Obj – lighter lines) and the ECS version (Ecs – darker lines). Three instances of the same test were ran with an increasing number of mass points in each instance. Both bodies began with 48 springs attached to each mass point and the number of springs was decreased to see how the FPS performance varied. As can be seen, the ECS version had better frames per second performance when the number of springs was greater and this difference was particularly noticeable when the number of mass points was also greater. As hypothesised, the ECS version was more efficient when handling larger amounts of data (greater number of springs and masses) even if only

slightly, although the method of solving the soft body could also be improved, especially when considering which portions of the application were causing the largest inefficiencies, the neighbour system.

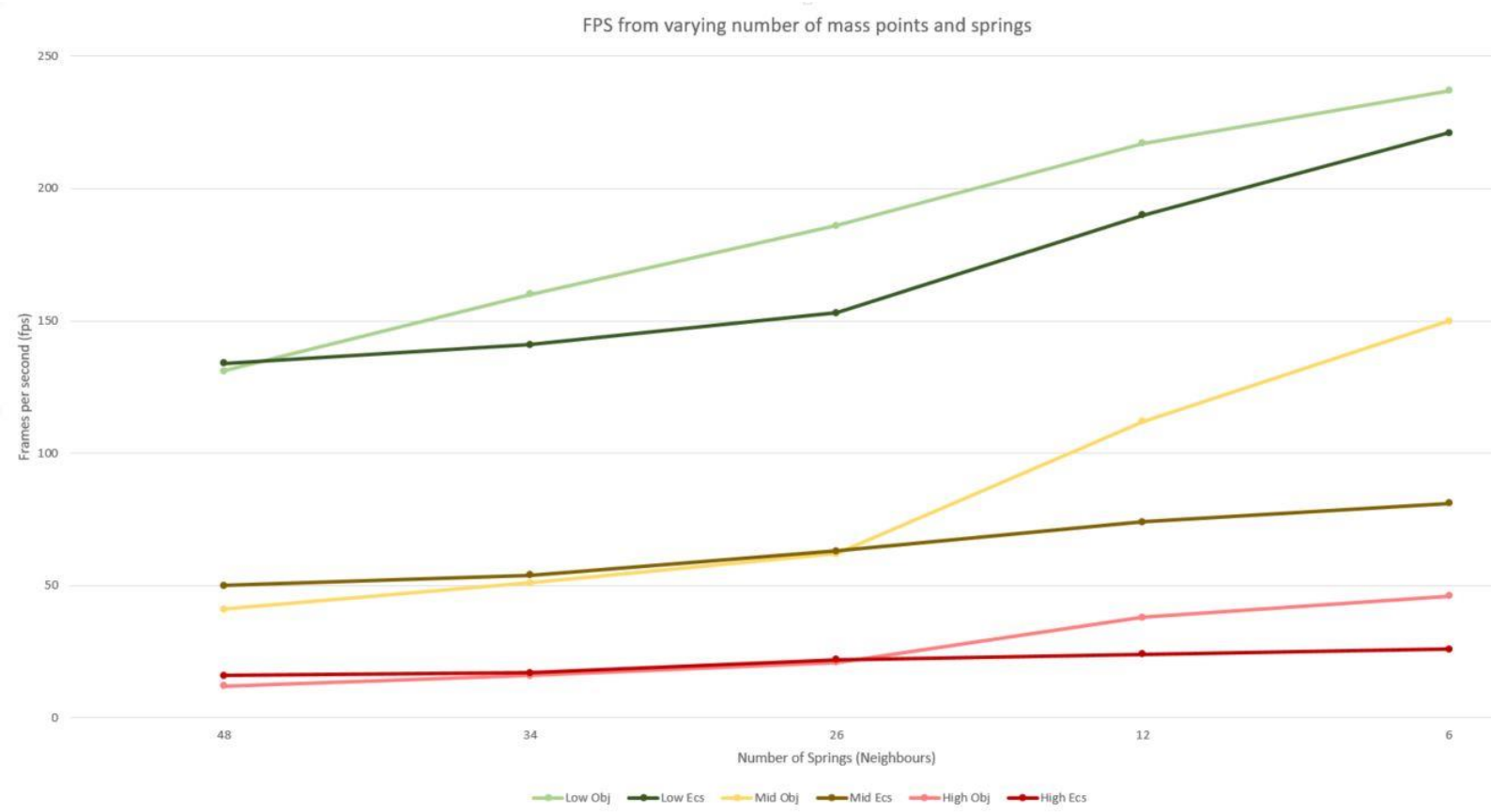

Figure 5-1 - FPS from varying number of mass points and springs

The method of constantly checking for the nearest neighbours clearly created a more realistic type of soft body as the shape was able to morph and was not as restricted. If a more rigid body was required (one that maintains its shape more) then the method of creating springs once and not updating their connections is the better choice. This method of assigning neighbours at the point of creation of the body is also more efficient as fewer calculations are needed to be done - the entire neighbours system can be ran once and then never again. Using a profiler, each thread in the ECS version can be seen as spending the longest amount of time executing the Nearest Neighbours system. (Figure 5-2)

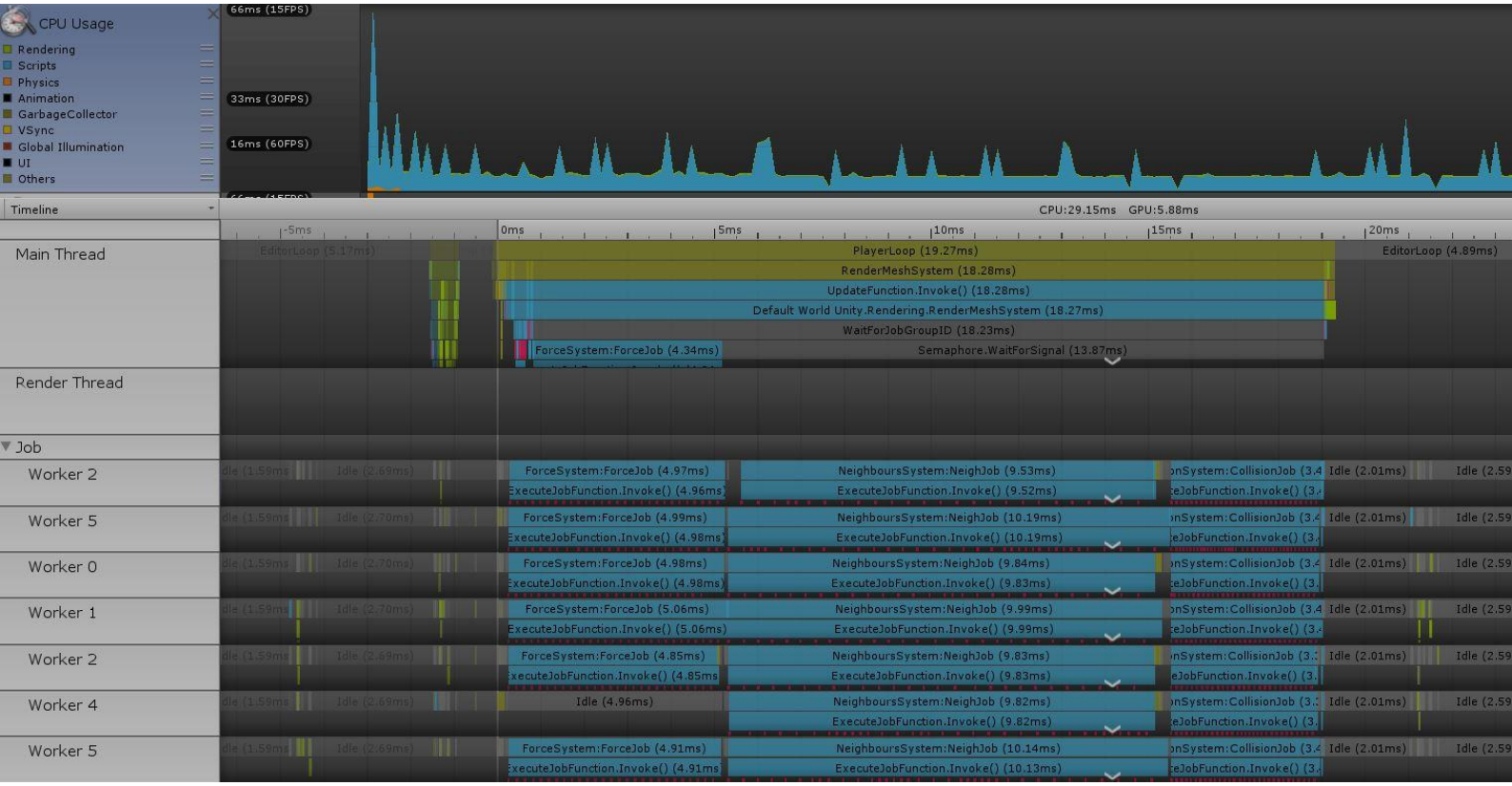

Figure 5-2 - ECS CPU and thread usage updating neighbours

5.2. Fixed Neighbours

In both versions (Obj and ECS) the Nearest Neighbours update method takes a considerable amount of time and is a major cause for the dropping of efficiency of the application. Comparing the same number of mass points and springs using the Fixed Neighbours method, the application performs far more efficiently as shown below.

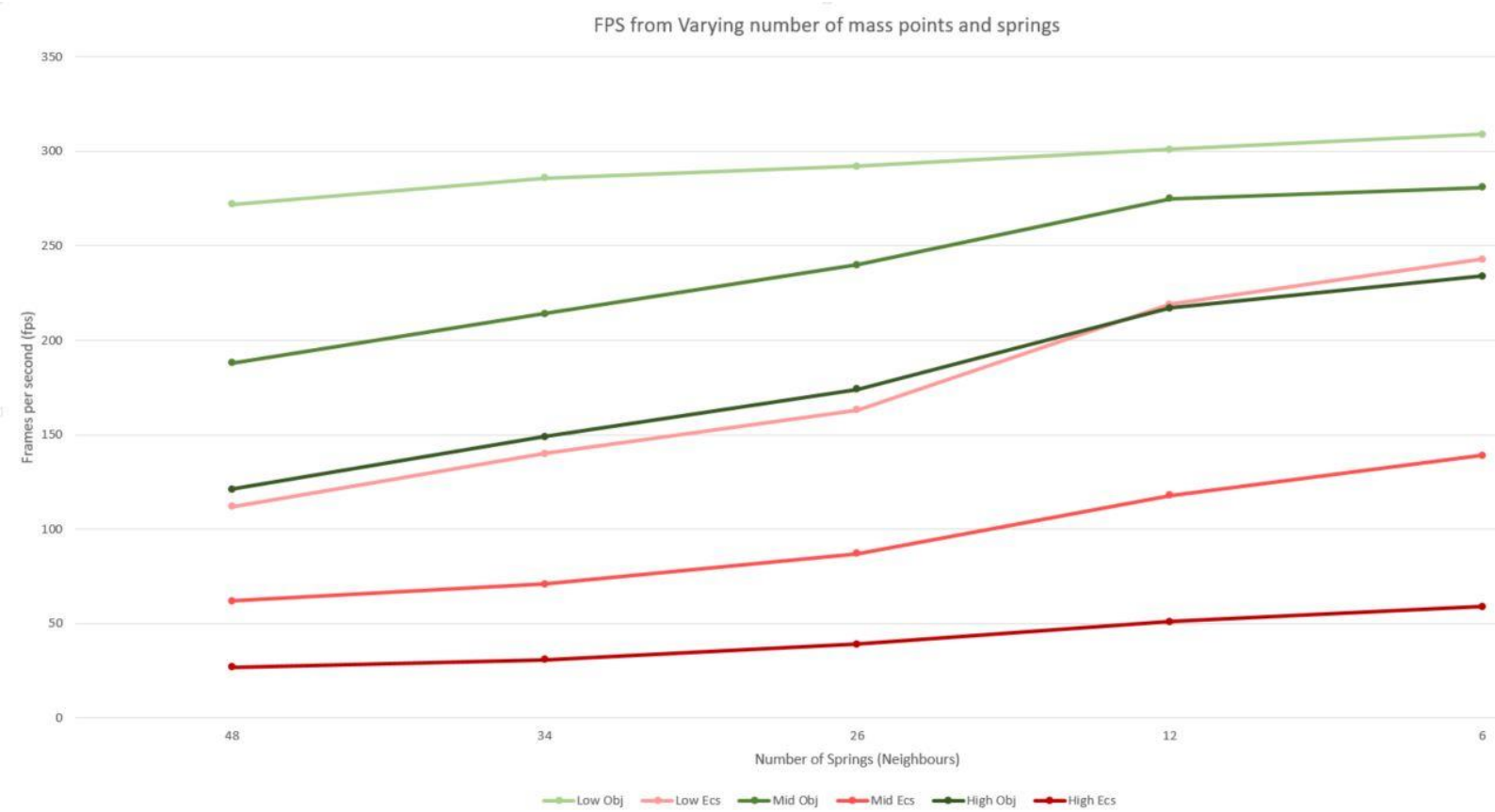

Figure 5-3 - Fps from varying number of mass points and springs

The results from the same FPS test but using fixed springs also demonstrate that the object-orientated version performs far more efficiently when springs are not needed to be recalculated constantly. The Obj version maintains around 120 frames per second more than the ECS version, for every number of mass points or springs. The one factor that the ECS version seems to maintain is the consistency in its values

when changing the number of springs, the FPS changes are not as drastic. The difference between the two versions is further evident when comparing how demanding the script calculations are on the CPU. This can be seen when profiling a smaller cloth object. It is clearly more efficient on the Obj version, which is also further helped by the little number of mass points and fewer calculations due to the fewer number of springs.

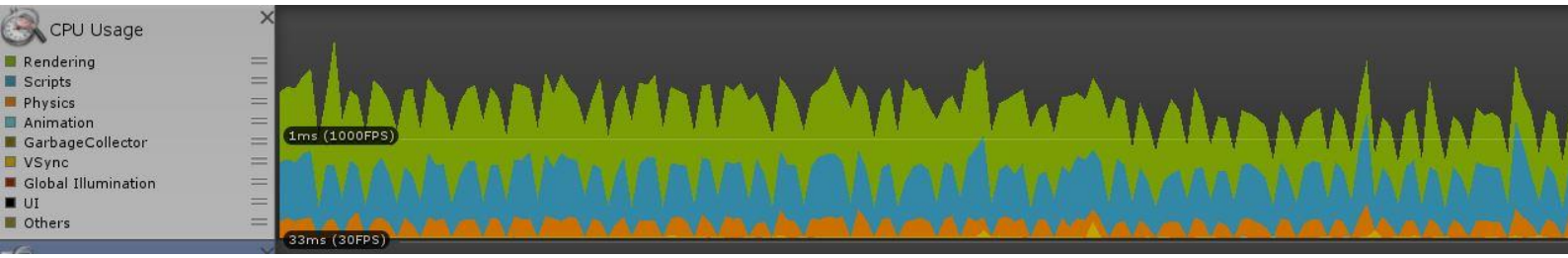

Figure 5-4 - OBJ CPU usage simulating cloth

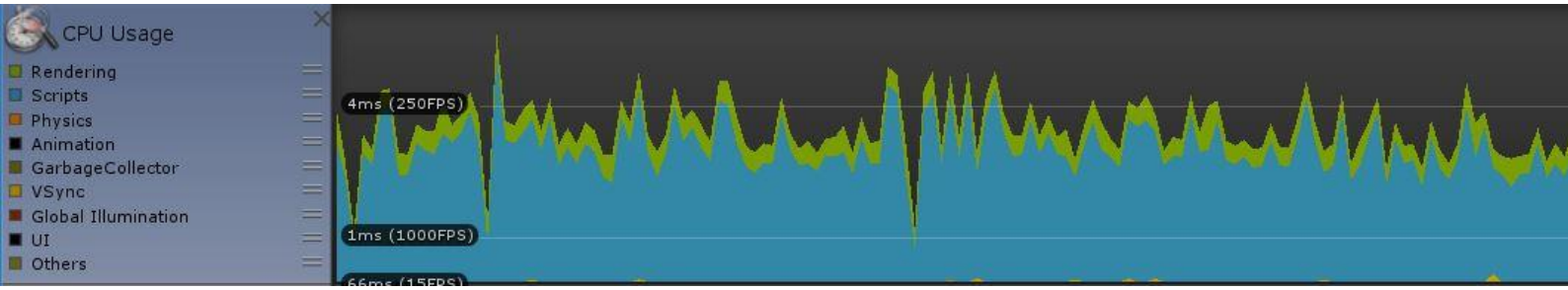

Figure 5-5 - ECS CPU usage simulating cloth

5.3. Rendering

During the comparison of two large bodies simulated in both versions, the GPU usage was monitored using the profiler (Figures 5-6 and 5-7) and shows that the GPU is less efficient in the Obj version than the ECS version, taking longer when rendering. The rendering of the soft body in both versions is possibly less efficient as multiple points have to be rendered, which could be replaced with a method that renders a whole smooth body from the points' positions. The ECS version's "Entity Debugger" (Figure 5-8) shows that the main thread is used almost entirely for the

rendering system, further proving that the rendering of the soft body is an issue for both versions.

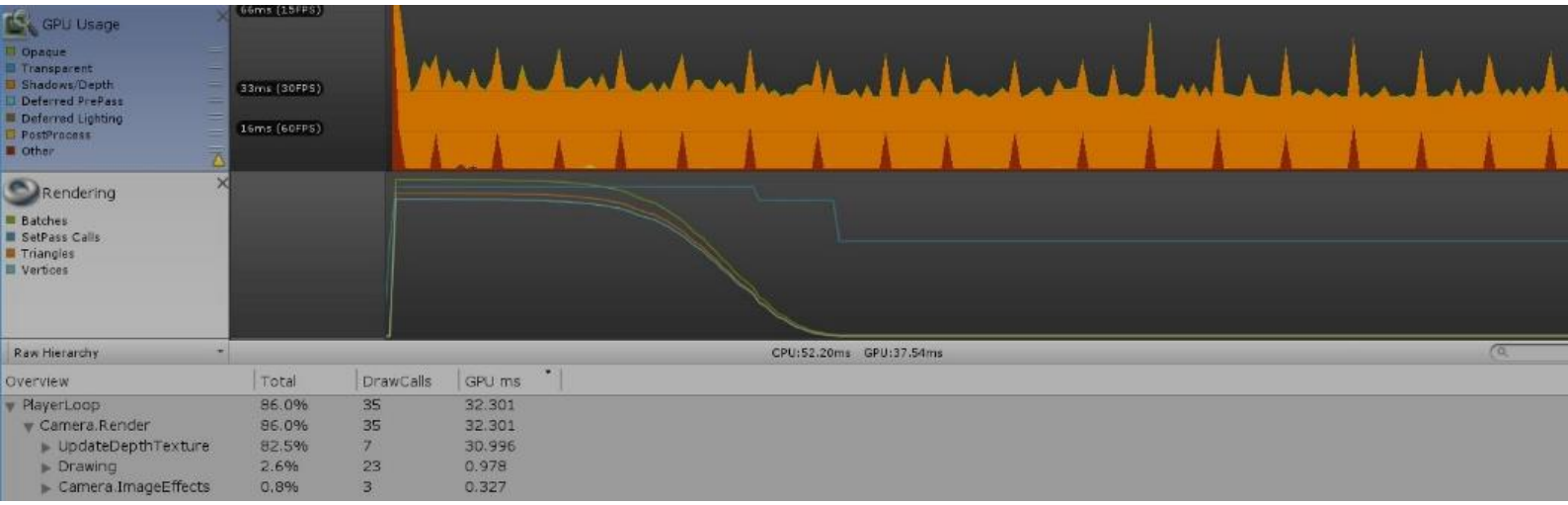

Figure 5-6 - OBJ GPU rendering a larger body

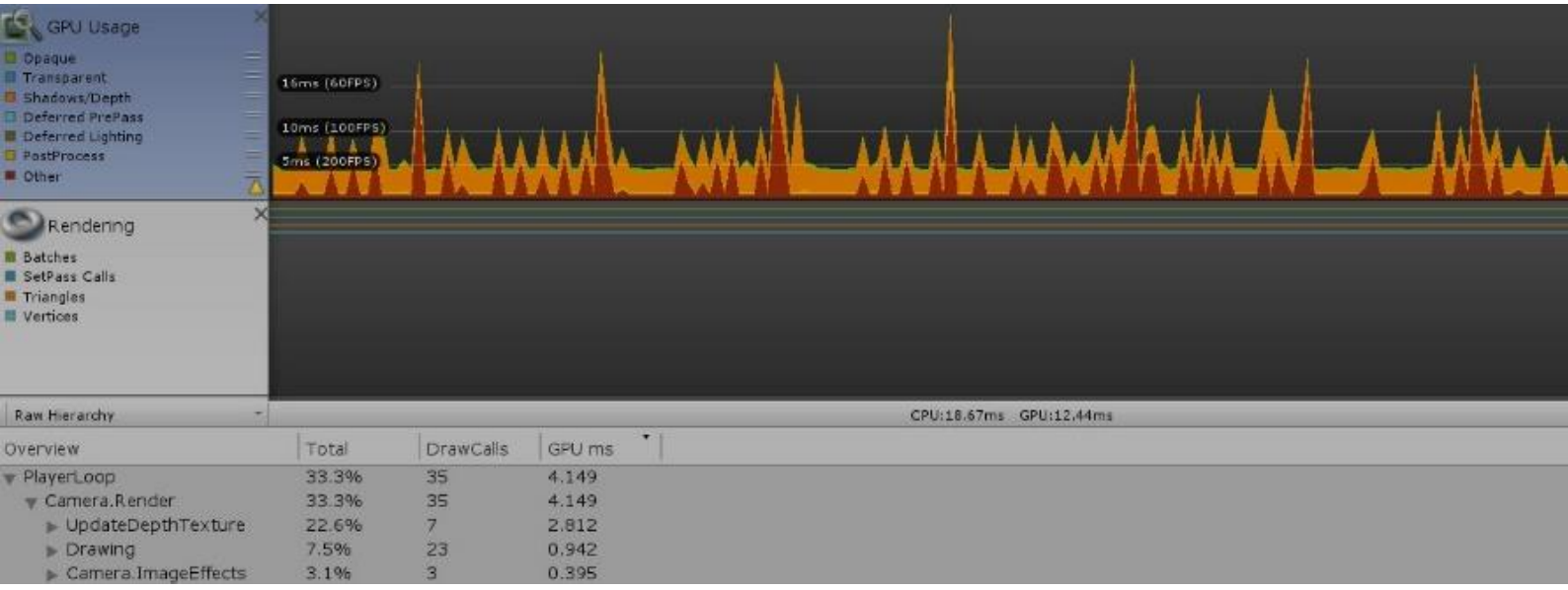

Figure 5-7 - ECS GPU rendering a larger body

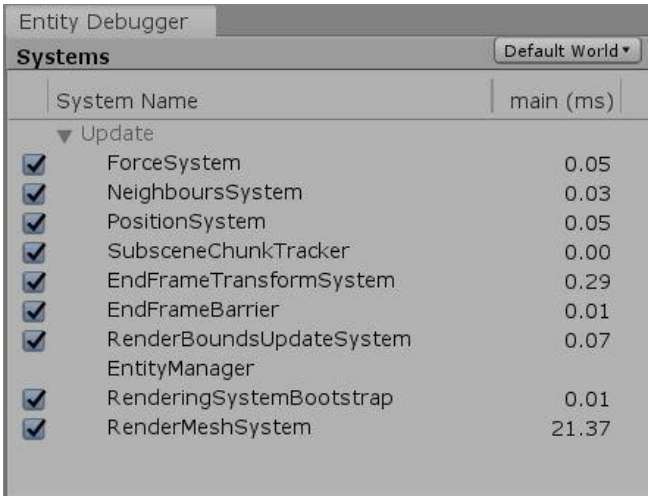

Figure 5-8 - ECS Entity Debugger simulating a large body

5.4. Constraints and Issues

Analysing the time taken within the code using Unity's profiler allows for a greater understanding of which elements of the code took the longest to execute, and therefore had the largest impact on the efficiency of the application. The 'Deep profile' mode times each function call and the output of an example soft body using a moderate amount of mass points can be seen below. However, one issue with the 'Deep Profile' slows down the overall performance of the application as it has to set timers at each function call, so the time displayed in milliseconds(ms) is slightly greater than if the 'Deep Profile' was not enabled, but it still demonstrates which function calls take the longest amount of time compared to others. Within the ECS version, viewing each thread individually shows that the collision system used was clearly a major bottleneck as it is taking up the majority of each thread's time. Within the Object-orientated version, the 'FixedUpdate' function calls are where the collision detection calculations are executed and therefore also show that the Obj version had the same issues with performance within collision detection as the ECS version. The AABB principle is possibly not suited to this number of mass points and research

could be conducted to understand how to reduce the performance issues within collision detection of the soft body.

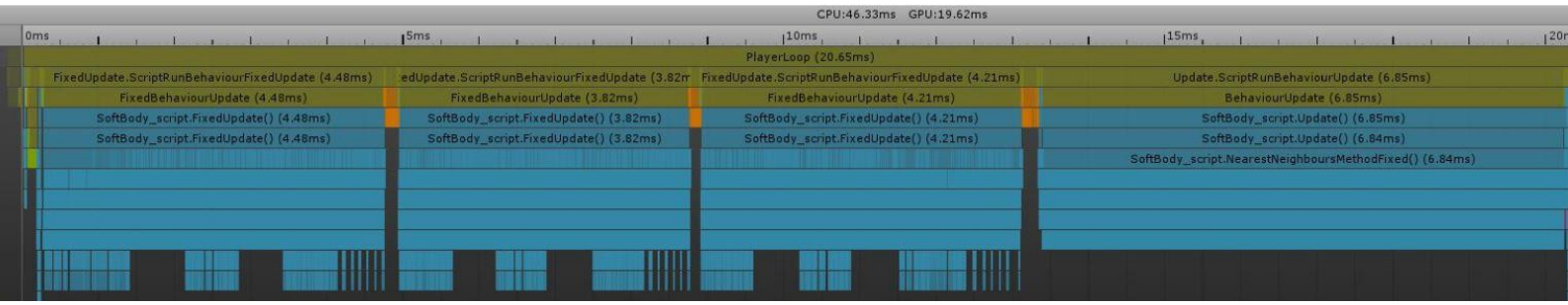

Figure 5-9 - OBJ CPU usage simulating a large body

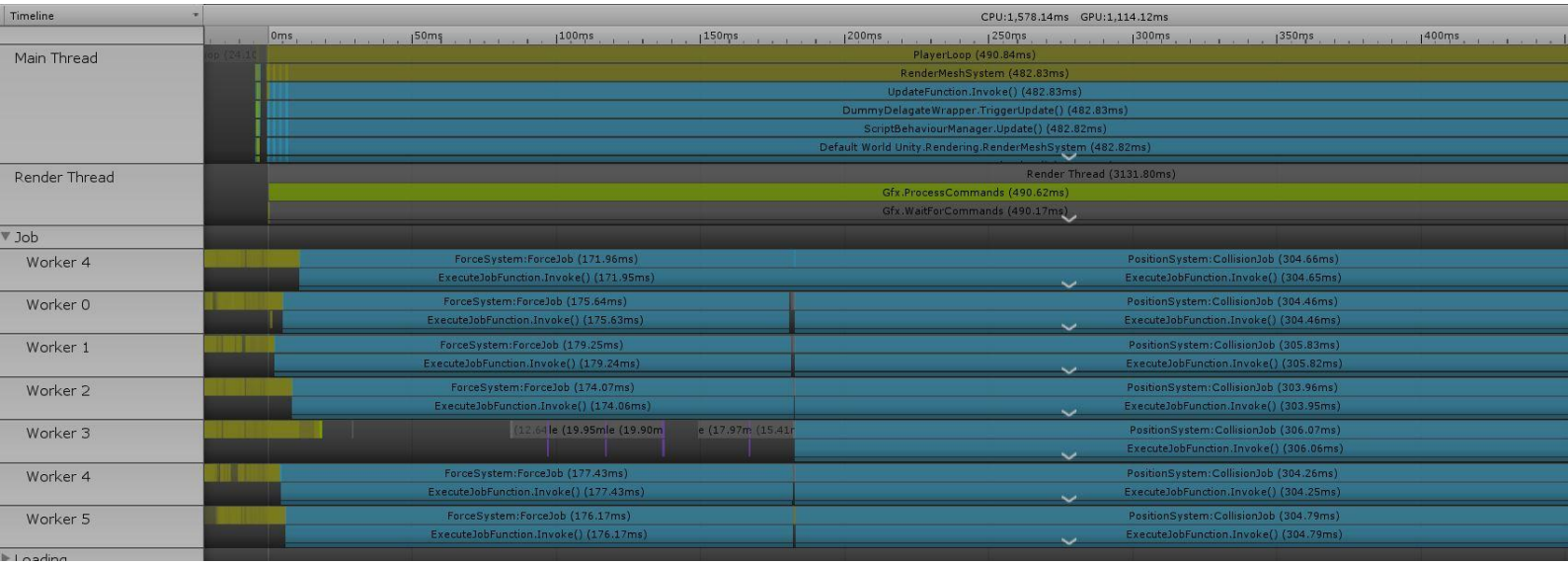

Figure 5-10 – ECS CPU thread usage simulating a large body

5.5. Alternate methods and Architecture

The first test of ‘Closest points’ in which a spring is connected to any mass point within a certain distance has identical performance to the Nearest Neighbours method and therefore needs only to be compared on its visual differences. The body in the closest points method was less realistic as it did not maintain any form of shape or return to its original structure after a collision and was also less realistic when simulating a more liquid body as the mass points would gather together too much into a ball.

In order to get the best performance out of the object-orientated version, rather than using separate scripts for each body, whether it be soft or rigid in the scene, a single ‘master’ script that stores a struct for each type of point was used. This method works the most efficiently out of all the object-orientated approaches that were tested but it is not useful for within a games context as most games require a large amount of objects in the scene and the position and collision of them cannot be done within one script. Therefore, while the method I took was the most beneficial for performance, it would be less efficient if it would be used within a games context as it would have to be separated into different scripts that communicate to each other instead. For this reason, the ECS version is more straightforward to convert into a games development scenario as its architecture is similar to how it would be used within that context. This shows that when analysing the programming complexity of the object-orientated method to the data-driven method, the data-driven method is simpler to separate into using the soft-body principles explored within this application. However, this is mostly due to the fact that each mass point can be treated as a separate data point and this programming complexity benefit may not be as beneficial for data-driven methods when using alternative soft-body principles such as rigid body deformation.

5.6. Parallelisation and GPU

Furthermore, when analysing the programming complexity of the two versions and the benefit of parallelisation, the data-driven approach allows for far simpler integration of multithreading, as was used within the testing of this application. The object-orientated version could be optimised by adding in parallelisation but would require far more programming complexity to ensure that threads were not accessing the same mass point information within the script, which is why it was not added to the application. The ECS version could also possibly have been optimised even further by using either the ‘Burst’ feature that is built into Unity’s Entity Component System that coverts the code into ‘machine code’ designed to run natively on the processor (Amat 2018) or a more common and less constrained method would be to make use of GPU parallelisation. The use of the GPU for parallelisation has been researched and explored for use with soft-bodies already (Fratarcangeli 2016), as

discussed within the literature review section however, if this was paired with a data-driven approach to programming, the programming complexity would not be as great and would also allow for more optimisations with multithreading.

5.7. Evaluation

One of the aims was to recreate a soft-body simulation in real-time that appeared realistic and the application allows for different types of bodies, all of which can be modified to simulate various different types of viscosity and shapes. A major issue with real-time and specifically games that has been discussed is the performance of the simulation. Sacrificing realism in order to obtain increased performance is not uncommon within a games context and the varying methods tested demonstrate the performance restrictions of object-based programming as well as data-driven techniques. The research conducted has compared the differences in the optimisations and programming complexity between the two versions and proved that each have advantages and disadvantages. This means that the choice of programming architecture for an application using soft-body dynamics is dependent on what type of body is required. For example, the closest points method which was dismissed for lack of realism could still be used as a method for simulating a soft-body within games as it still created an interesting body which can be seen as a form of soft-body that, while it may not perfectly represent our world's laws of physics, it could still have a use within the games industry if a less accurate/realistic body was required.

Due to the experimental nature of ECS and the lack of use of data-driven methods in the games industry, this application and analysis of data-driven techniques must also be evaluated for its viability within games development. For most games, a standard of 60 frames per second is the stable minimum. While testing the application, certain measurements were less efficient than 60fps and therefore could not possibly be used in a games scenario. However, less intense soft-bodies that used fewer mass points and springs were still suitable at simulating a soft-body whilst also reaching much more sustainable frames per second (90+ fps) that would allow for them to be used within games. While object-orientated programming is the industry standard within games, a system that used data-driven methods to create soft-body dynamics would not be difficult to integrate within a game engine or application as it could be

self-contained and could still communicate with other parts of the game, even if they were not using data-driven methods.

There may have also been specific parts of Unity's Entity Component System that they have created that could make better use of the Job system and parallelisation, such as the Burst feature previously mentioned which I may have missed. A large amount of this is due to the experimental and undocumented nature of Unity's ECS but it could have been beneficial to explore and through more testing and learning, taken full advantage of all of the features at my disposal. Further on from this, a more experienced programmer may have more knowledge on how else to optimise the ECS version but as I was constrained to research and develop this application during the course of 9 months, it is difficult to determine how much more experience and time would have impacted the development.

6. Conclusion

In conclusion, soft-body dynamics can be used within real-time applications, and will be used more frequently as hardware advances. The application built demonstrated that various techniques can be explored when creating soft-bodies, both in soft-body principles and programming architectures. A real-time simulation was created using the spring/mass principle and was tested for its efficiency within object-orientated and data-driven methods, both of which proved to be successful. However, there are a number of ways in which the application could be improved so that it would be more viable within a games context. The comparison of object-orientated and data-driven methods proved that there are advantages to a data-driven approach when creating soft-bodies in real-time, especially when handling large amounts of mass points or springs and would possibly have increased performance if certain portions of the application were optimised, such as the collision and rendering systems. Greater numbers of Mass points and springs is the best way to gain increased accuracy and realism when simulating a soft-body using the spring/mass method and the ECS version that was tested excelled when dealing with a greater number, proving that it is certainly a viable alternative to object-orientated methods. The main benefit, however, to using data-driven approaches for such physics simulations is the ability to use parallelisation more efficiently and with less programming complexity, a technique which is becoming increasingly common as hardware advances to contain more cores/threads.

There are already a number of different ways that collision detection has been adapted and improved over the lifetime of the games industry but as this was not something explored within the application it would be beneficial for future study to research into how best to optimise the collision detection for use with the spring/mass method. An alternate rendering method for the soft-body would also be useful future research as it was identified as an issue for not only performance reasons but would also make the soft-body appear more realistic if it could cover the entire body, based on the mass points' positions. One method that came to mind during the development of the application was to create a vertex shader that modifies the vertices of a texture to the positions of the mass points and generates triangles between them to give the appearance of a single body.

It is possible that soft bodies may become more of a useful mechanic to explore for players to interact with, using the data-driven methods explored within this application, rather than purely treating soft body simulations as aesthetics for clothing or hair. More work and research would be required to make sure that there is a creative application of soft-bodies and this would likely arise within a smaller indie-game. Possibly if there was more consumer/public interest for mechanics using soft-body simulations, it would be explored more by larger-scale AAA titles.

7. References

Brackeys 2017, How many Rigidbodies can Unity support?, *YouTube*, <https://www.youtube.com/watch?v=8zo5a_QvJtk>.

ECS Unity Technologies, DOTS - Unity's new multithreaded Data-Oriented Technology Stack, *Unity*, viewed 30 March, 2019, <https://unity.com/dots>.

Geig, M 2018, Performance result comparison @mikegeig, *Twitter.com* , <https://twitter.com/mikegeig/status/951260836538003458>.

Havok Physics – Havok, *Havok.com*, viewed 2 April, 2019, <https://www.havok.com/products/havok-physics/>.

List of games using Havok, En.wikipedia.org, viewed 2 April, 2019, <https://en.wikipedia.org/wiki/List_of_games_using_Havok>.

Valve 2004, Half-Life 2 on Steam, *Store.steampowered.com*, viewed 5 April, 2019, <https://store.steampowered.com/app/220/HalfLife_2/>.

Blender Foundation, Blender's SoftBody System, *Download.blender.org*, viewed 5 March, 2019, <https://download.blender.org/documentation/htmlI/ch23s02.html?fbclid=IwAR3Xe5mhzVQtkxrDPVHpc3HwEnzzZccqY_WSG_9736s3VD0OGjHOg1a6O-Y>.

Autodesk 2018, Soft Bodies | Maya 2018 | Autodesk Knowledge Network, *Knowledge.autodesk.com*, viewed 15 March, 2019, <https://knowledge.autodesk.com/support/maya/learn-explore/caas/CloudHelp/cloudhelp/2018/ENU/Maya-SimulationEffects/files/GUID-20790C1E-D32E-457C-A867-BC2345EF707C-

htm.html?fbclid=IwAR1hmPSx7Y8F3IAuH46l74z0wOQYLBGg-tXMgR4R0jI_ABqPTu7xfgvCSeU>.

Monaghan, J 1992, *Smoothed Particle Hydrodynamics*, viewed 24 March, 2019, <http://www.astro.lu.se/~david/teaching/SPH/notes/annurev.aa.30.090192.pdf?fbclid=IwAR3tOmkNmlNyLrwjsMKJ1QY79M05vFcMBzVWP3LxyumRUYlpmET_ol_TKxQ>.

Weaver, T and Xiao, Z 2016, *Fluid Simulation by the Smoothed Particle Hydrodynamics Method: A Survey*, viewed 22 March, 2019, <https://pdfs.semanticscholar.org/8054/1cb2d6951e48fb648f8ce160871e0ab2e3ce.pdf>.

Mesit, J 2010, Modeling And Simulation Of Soft Bodies, *Stars.library.ucf.edu*, viewed 16 November, 2018, <https://stars.library.ucf.edu/cgi/viewcontent.cgi?referer=https://www.google.co.uk/&httpsredir=1&article=2646&context=etd>. (pg 25 - 26)

PhysX, *En.wikipedia.org*, viewed 20 April, 2019, <https://en.wikipedia.org/wiki/PhysX>.

Barsegyan, A 2019, NVIDIA PhysX FleX and other fluid solvers for high-quality fluid simulation, *Cgicoffee.com*, viewed 24 March, 2019, <http://cgicoffee.com/blog/2016/11/nvidia-physx-flex-fluid-simulation-and-other-solvers>.

Yalçın, M and Yıldız, C 2019, *Techniques for Animating Cloth*, viewed 15 November, 2019, <https://pdfs.semanticscholar.org/139f/6df0a07f977a624dd0ed8022688e8431977a.pdf>. (pg 4-5)

BeamNG 2015, JBeam Physics Theory - BeamNG, *Wiki.beamng.com*, viewed 6 October 2018, <https://wiki.beamng.com/JBeam_Physics_Theory>.

Yeh, T, Faloutsos, P and Reinman, G 2018, *Enabling Real-Time Physics Simulation in Future Interactive Entertainment*, viewed 17 November, 2018, <http://www0.cs.ucl.ac.uk/teaching/VE/Papers/p71-yeh.pdf>. (pg 72-79)

Fratarcangeli, M, Tibaldo, V, & Pellacini, F. 2016. Vivace, *A practical gauss-seidel method for stable soft body dynamics*. *ACM Transactions on Graphics (TOG), 35*(6), 1-9.

Fabian, R 2013*, Data-Orientated Design*, <http://www.dataorienteddesign.com/dodmain.pdf>. (pg 201)

Millington, I 2007*, Game Physics Engine Development*, <http://www.r-5.org/files/books/computers/algo-list/realtime-3d/Ian_Millington-Game_Physics_Engine_Development-EN.pdf>. (pg 234)

Wallentin, O 2014, Component-Based Entity Systems, Diva-portal.org, viewed 13 November, 2018, <https://www.diva-portal.org/smash/get/diva2:728755/FULLTEXT01.pdf>. (pg 15-19)

Boids Demo 2018, Unity ECS with Boids, *YouTube*, <https://www.youtube.com/watch?v=SRn1dxStwO0>.

Danielsson, M and Bohlin, G 2017, *A High Performance Data-Driven, Entity-Component Framework For Game Engines With Focus on Data-Orientated Design*, <https://autious.net/files/ECS.pdf>. (pg 13)

Martin, A 2007, *Entity Systems are the future of MMOG development – Part 1* viewed 7 March, 2019, <http://t-machine.org/index.php/2007/09/03/entity-systems-are-the-future-of-mmog-development-part-1/>.

Cocco, A and Masin, S 2010, *The law of elasticity*, <https://www.uv.es/psicologica/articulos3FM.10/13Cocco.pdf>.

Struct 2015, struct - C# Reference, *Docs.microsoft.com*, <https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/struct>.

Scalar Product, Math Centre 2009, *Mathcentre.ac.uk*, <http://www.mathcentre.ac.uk/resources/uploaded/mc-ty-scalarprod-2009-1.pdf>.

Prefab Unity, 2019, Unity - Manual: Prefabs, *Docs.unity3d.com*, viewed 29 April, 2019, <https://docs.unity3d.com/Manual/Prefabs.html>.

Class 2018, Classes - C# Programming Guide, *Docs.microsoft.com*, <https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/classes-and-structs/classes>.

Johansson, T 2018, What is a Job System? – Unity Blog, *Unity Technologies Blog*, <https://blogs.unity3d.com/2018/10/22/what-is-a-job-system/>.

Amat, C 2018, Unity Burst Compiler: Performance Made Easy • Infallible Code, *Infallible Code*, <http://infalliblecode.com/unity-burst-compiler/>.

8. Bibliography

Lembcke, S 2006, *Realtime Rigid Body Simulation Using Impulses*, <https://pdfs.semanticscholar.org/539b/266a96d1654ab4e27c02b547f2cd7c535ab5.pdf>.

Clarke, G2018, *Creating gameplay mechanics with deformable characters*, <https://rke.abertay.ac.uk/ws/portalfiles/portal/15147598/Clarke_CreatingGameplayMechanicsWithDeformableCharacters_Author_2018.pdf>.

Kleppmann, M 2007, *Simulation of colliding constrained rigid bodies*, viewed 30 April, 2019, <https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-683.pdf>.

9. Appendices

Feasibility Demo Example (in OBJ): https://youtu.be/e-H2lSZaFJc

Fixed Neighbours Example (in ECS):
https://www.youtube.com/watch?v=NaVvGqzRaIw&feature=youtu.be

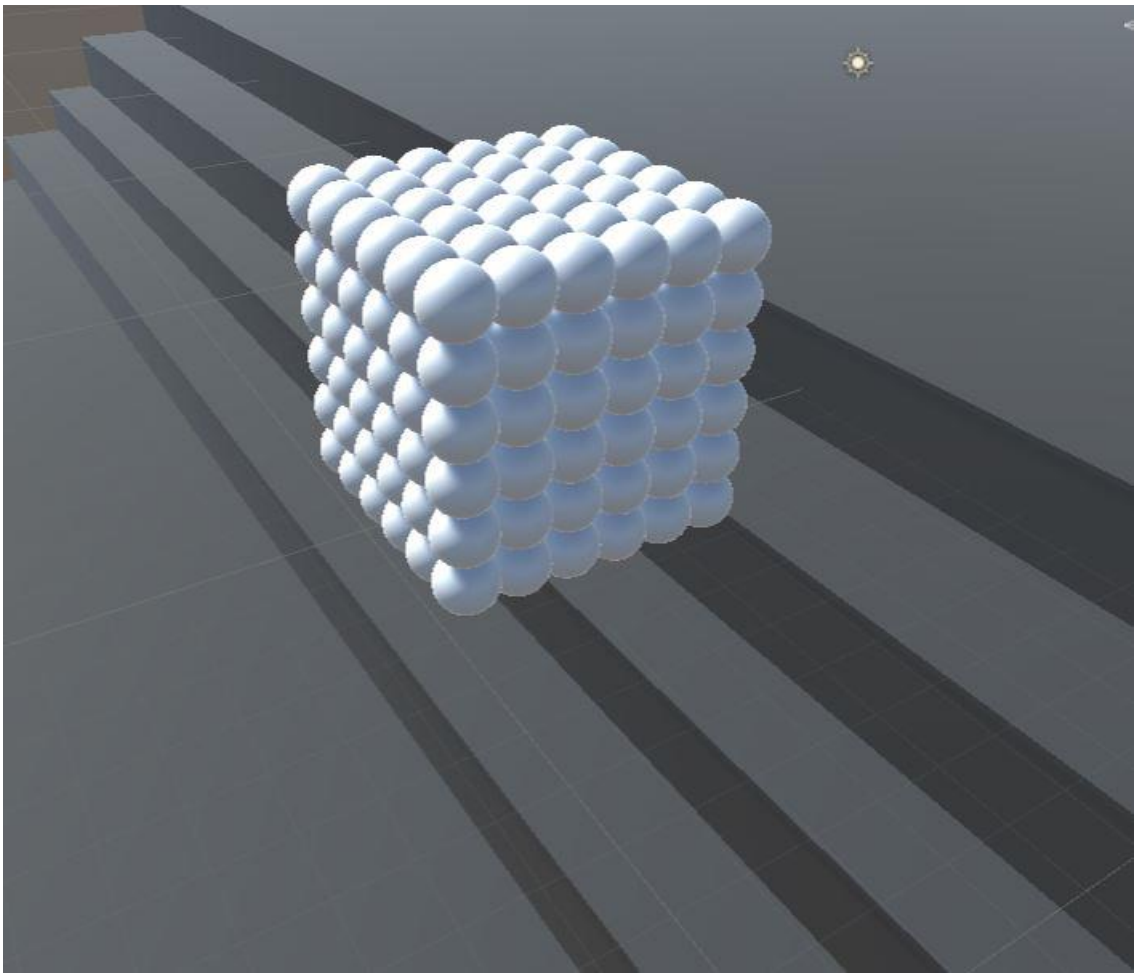

Figure 9-1 - Fixed Neighbour Body

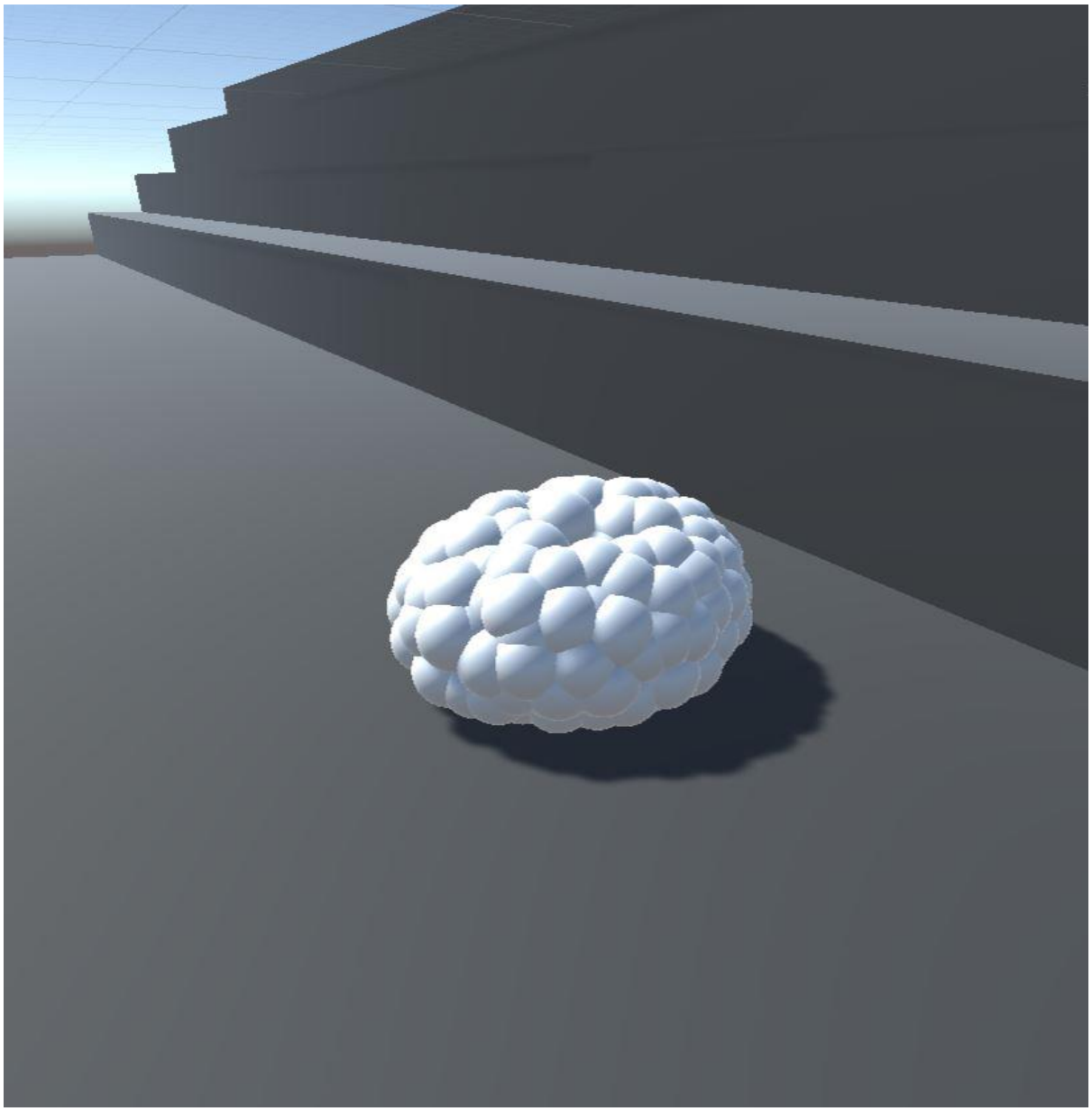

Figure 9-2 - Neighbour update Body

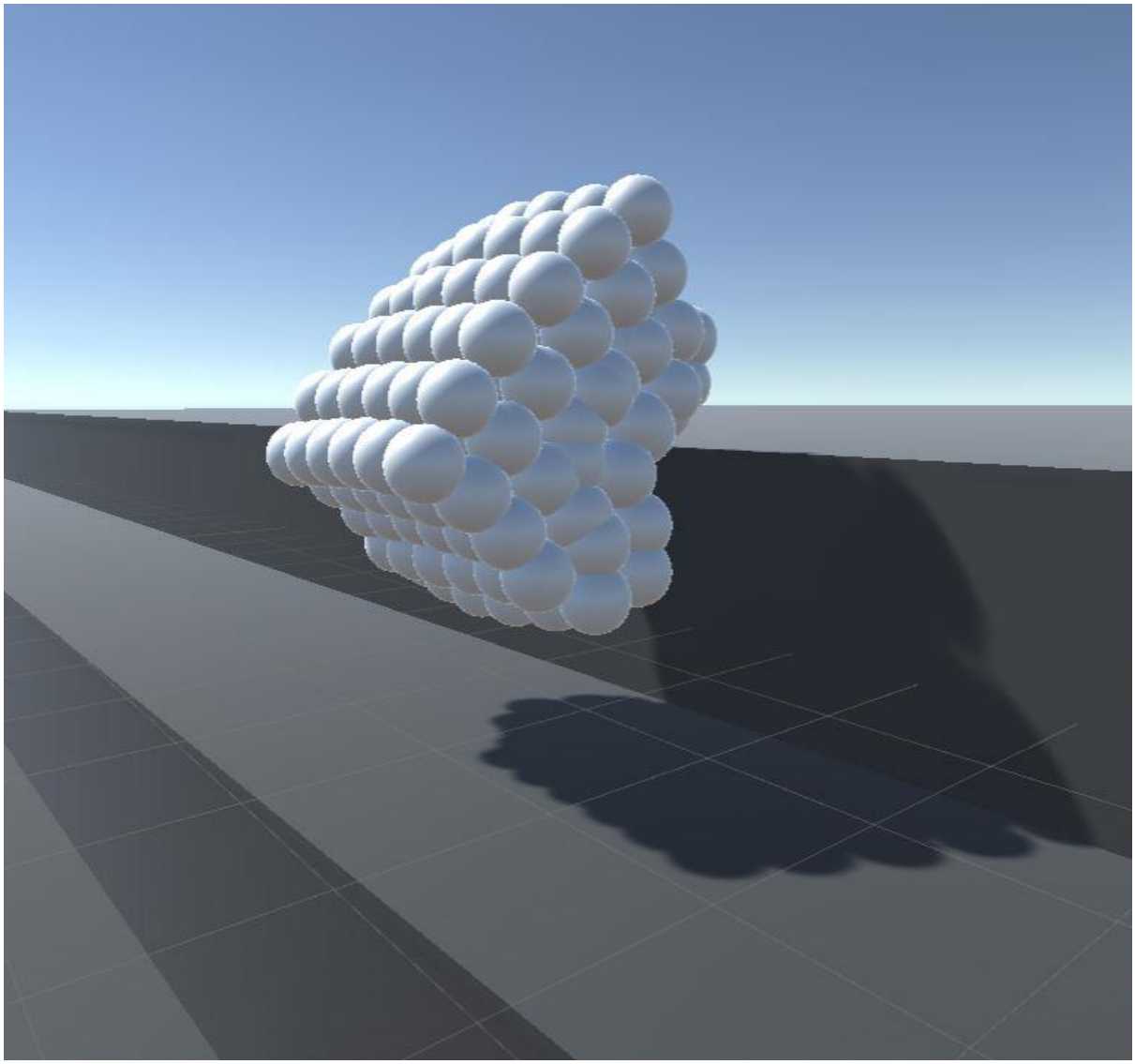

Figure 9-3 - Fixed Neighbour squashing to stairs