Realtime Soft Body Simulation using a Spring-Mass approach

Charles Hart

School of Computing Science, Newcastle University, UK

Abstract

Soft bodies are physically simulated objects capable of deformation - changing shape in response to forces experienced. Most physical objects in computer simulations are rigid bodies, which do not experience deformation, however soft body physics is required in order to simulate many complex structures in a physically accurate manner, such as cloth, paper and flexible solids. In this project, different approaches to soft body simulation are investigated, before one is chosen and implemented, alongside a system to render the deformable objects in real time. The implemented systems were then tested, and results showed that at least 64 cubic soft bodies were able to be simulated simultaneously, and that the dynamic mesh rendering used to render deforming meshes was able to render these 64 soft bodies in real time at 75.8 frames per second. Further improvements and optimisations proposed suggest that both the quality and efficiency of simulation could be improved even further.

Keywords: Softbodies, Physics, Constraints, Deformation, Springs, Rendering

1 Introduction

1.1 Soft and Rigid Bodies

Soft bodies are simulated deformable objects, used mostly in video games, computer graphics, and mechanical and structural analysis in engineering. [1] Most physically simulated objects in a game world make use of rigid bodies, in which each point on the object remains at a fixed distance from each other, resulting in a fixed shape. While this is practical for many objects, deformable objects are often required, and so soft bodies are used.

A soft body typically consists of a set of points, each with a different position, velocity and forces applied. The distances between points in a soft body can change, driven by the physical properties of the object and interactions with other physics objects, resulting in a shape which can deform over time.

Whilst completely rigid bodies do not exist in reality, due to deformations at a microscopic level and thermal expansion,[3] the deformations that can occur are

[1] Email: C.J.Hart1@newcastle.ac.uk

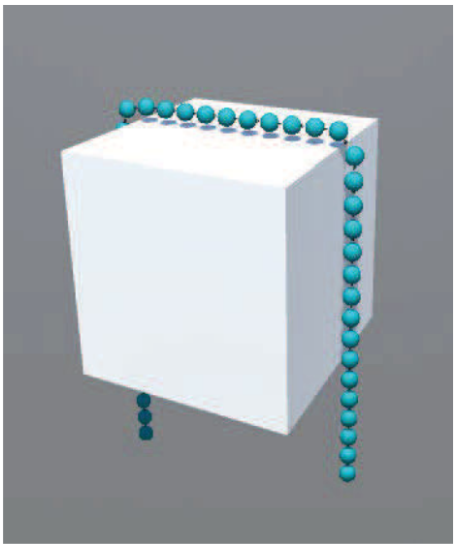

Fig. 1. A simulated rope soft body deformed around a cube, formed by a series of connected particles in a line.[13]

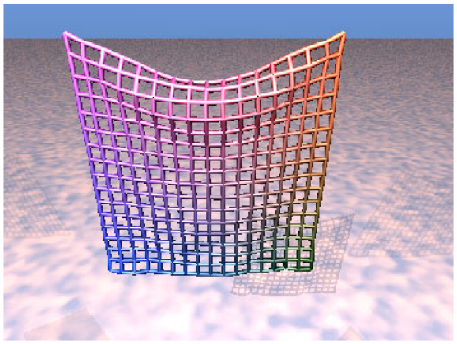

Fig. 2. A cloth soft body deforming due to the effects of gravity.[4]

negligible, and can be ignored in most simulations of common rigid materials in a game world, including metals, wood and stone. Rigid bodies are therefore used when simulating most common objects in a game world, including walls, floors, buildings and terrain.

Though rigid bodies are suitable for many applications, they cannot always be used, as many objects exist in the real world which deform as an essential part of their functionality, and so in order to simulate certain objects in a video game world, this deformation cannot be ignored. These objects include rope,[13] cloth,[4] and rubber objects, as shown in Figures 1 and 2.

Soft bodies are commonly used in cloth simulation, where large, two dimensional meshes are created which must deform when placed on an uneven surface, such as a cape on a character model.[2] One dimensional soft bodies, consisting of a single line of points connected in a certain shape, can be used to simulate objects such as rope or hair. Soft bodies with three dimensions will be the focus of this project, which have a rest shape, and will deform when experiencing forces, before eventually returning to the rest shape. These soft bodies could be used to simulate objects made of less rigid materials such as rubber or organic materials, inflatable objects, and several other objects.

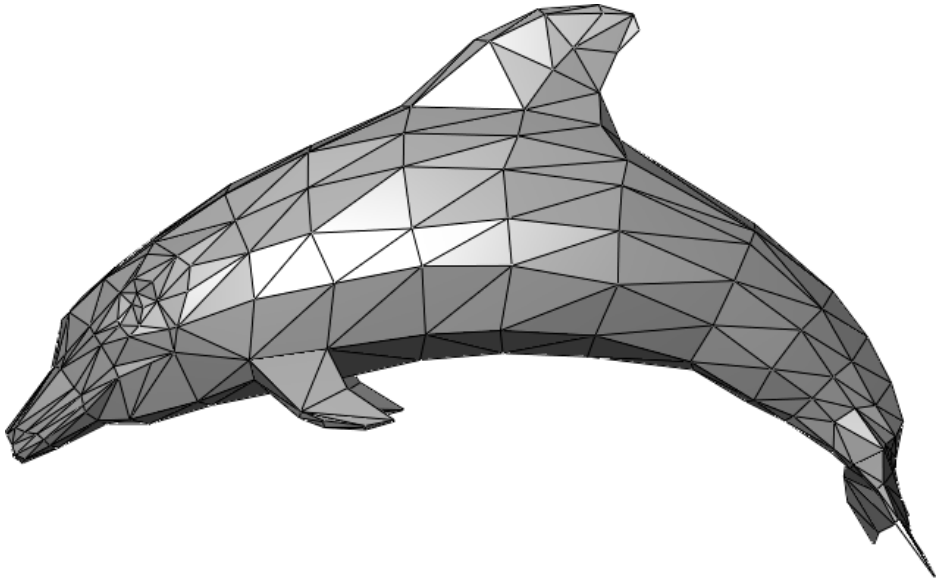

Fig. 3. A simple polygonal mesh of a dolphin, using triangular faces.[6]

1.2 Constraints in a Physics Engine

In a physics engine, constraints are used in order to mathematically control or restrict the position, velocity or acceleration of an object, using a specified formula.[7] One of the most basic examples is a distance constraint, which limits the maximum distance between two objects, and can be used to simulate physically accurate ropes, or ragdolls.[8] Constraints can also indirectly control the velocity, position or acceleration of an object, for example by applying forces to an object, as, due to Newton's second law of motion, a net force F acting on an object with mass M will cause an acceleration A, following the equation $F = MA$.[9]

1.3 Polygon Meshes

In computer graphics, polygonal meshes are used to render three dimensional objects to the screen. A mesh consists of a set of points, or vertices, connected together with edges, to form a set of faces to create the surface shape of an object, as can be seen in Figure 3. Each face can be one of a few shapes, but the most common and generally the most efficient is the triangle.

In order to increase the efficiency of rendering, and to reduce the amount of duplicate data stored and processed, index arrays are often used - an array of vertices stores the position and other data for each vertex, and the index array stores sets of three indices for each triangular face, one for each vertex of the triangle. In meshes containing several faces sharing a common vertex, this can provide a noticeable improvement to performance, as the amount of memory occupied on the GPU by the mesh can often be reduced significantly.[11]

1.4 Overview

This paper will explore approaches to simulating soft bodies in three dimensions, before discussing the design and implementation of a chosen approach, and evaluating

the performance and quality of the approach implemented.

1.5 Aims and Objectives

The overall aim of the project is to investigate and implement efficient three dimensional soft bodies in a game engine.

To achieve this aim, the following objectives should be met:

- Explore and understand the approaches that can be taken to simulate a three dimensional soft body.
- Implement the physical simulation of the chosen type of soft body.
- Implement the rendering of the chosen type of soft body.
- Evaluate the performance of the implemented soft bodies, with varying numbers of soft bodies in a scene.

2 Background Research

Several approaches to soft body simulation exist, and are suitable in different cases. This section covers the most common approaches and proposed solutions, explores how they function, and when each approach is appropriate.

2.1 Spring-mass models

One of the simplest approaches to soft body simulation is the spring-mass model. Spring-mass models can be used in both cloth simulation[4] - the simulation of a deformable two dimensional mesh in three dimensional space - and the simulation of three dimensional soft body objects. In both cases, the models consist of many points, each with unique position, velocity, mass and experienced forces, connected together with spring constraints, to form triangles. Typically, both compression and expansion spring constraints are used, in addition to a damping force,[4] to prevent the energy in a given soft body from increasing due to a lack of energy loss to the environment, such as through air resistance.

Hooke's Law states that the force (F) required to change the length of a spring by a distance (d) is linearly proportional to that distance, where (k) is the spring constant of the spring, and can be expressed as $F = kd$.[5] This is the fundamental law that drives the behaviour of springs in most simulations, including spring-mass soft body simulations, but only applies up to the point at which an object reaches its elastic limit, where permanent deformation begins to occur.

In reality, deformation of most soft objects is not 100% efficient; energy losses such as the heating of the deformed object from deformation forces mean that Hooke's law is not perfectly followed in many cases, though can be viewed as modelling a perfect spring.

Due to the relative simplicity of the spring-mass approach, and minimal computations required to simulate individual spring constraints connecting a small number of points, this method is best suited to perform well in a real time simulation, where speed and efficiency must take priority over perfect physically accurate deformation.

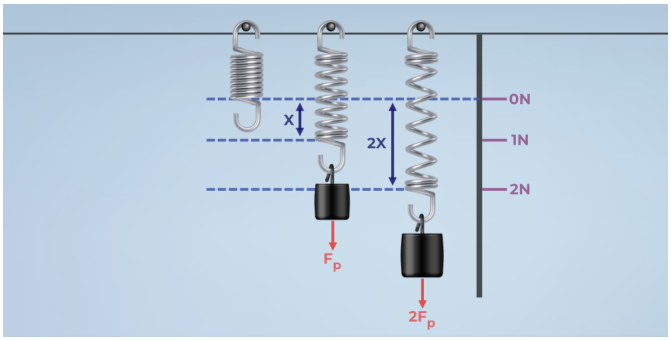

Fig. 4. A diagram demonstrating Hooke's law for three springs with suspended masses. The left spring is at rest, and has a displacement of zero. As the force F_p is applied to the centre spring, it extends by X units. Doubling this force for the right hand spring results in a displacement double that of the centre spring.[19]

2.2 Finite Element Models

Finite element models work by subdividing a body into a large number of simpler elements, such as cubes or tetrahedrons. In order to do this, the original body is subdivided by the discretization of the continuous functions defining the shape of the object.[16] A set of differential equations is then formed, which can then be used to approximate the positions and shapes of each element.

This method is relatively complex, and requires a large number of differential equations to be solved in order to approximate a solution, something which is not reasonable to attempt 50 or more times per second. Finite element models are typically used in engineering and structural analysis, where accuracy and precision are vital, and real time simulation is not a requirement. Therefore, this approach, while producing more physically accurate simulations, is unlikely to be the best choice for a real time simulation inside a game engine.

2.3 Proposed Solutions

In 1987, D. Terzopoulos et al. proposed methods for simulating soft bodies, using models based on elasticity theory,[16][17] as differential equations were produced to model the motion of flexible surfaces and solids over time. In this paper, objects including sheets of paper and rubber were simulated, with separate variables controlling both the resistance of an object to change length, and resistance to bending forces imparted on the object, such as through gravity.

In 1999, M. Desbrun et al. proposed an algorithm for the stable simulation of deformable objects using a spring-mass model.[18] In the paper, an implicit integration method was used to calculate the velocity of point masses, as explicit integration resulted in an unstable soft body for any but the lowest physics time steps, something which cannot always be ensured in a changing simulation. A modification to the spring simulation used in spring-mass systems was also proposed, in which a post-correction phase was added after integration, in order to modify the behaviour of the springs, to mirror the non-linear elasticity experienced by materials displaced beyond their elastic limit.

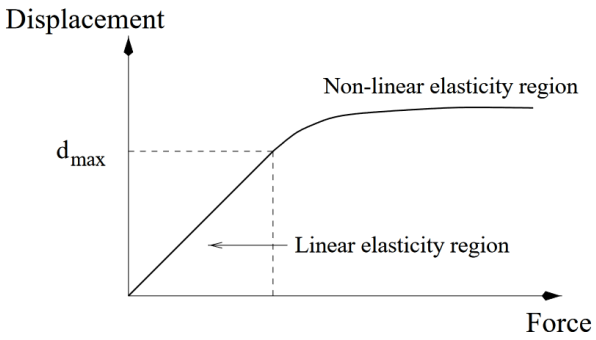

Fig. 5. A graph showing the relationship between displacement and force of a deformable object as it reaches its elastic limit.[18]

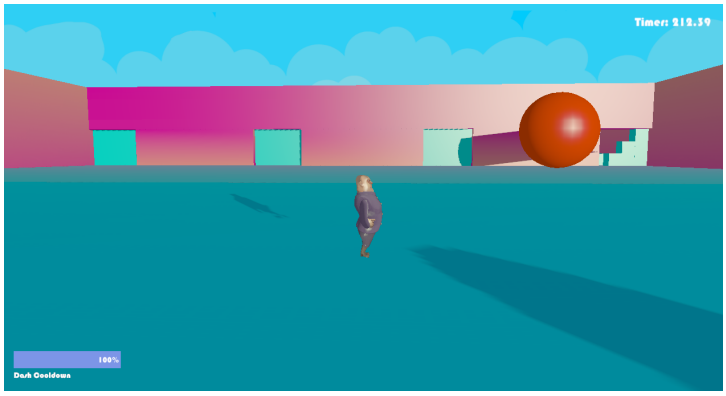

Fig. 6. An existing game, created using the engine used for this project, capable of rendering polygonal meshes with real time lighting and shadows.

3 Design

3.1 Game Engine

This project utilises an existing game engine written in C++ with simple, essential features such as a rendering system for polygonal meshes, lighting and shadows, as seen in Figure 6, and a limited GUI. Notably, the engine only supported the rendering of static meshes, a mesh could only be translated, rotated or scaled. The relative distance between vertices and overall shape of the mesh could not change at runtime, apart from by following preset animations, unable to respond to the current game state.

The existing engine also featured a basic physics system, in which rigid body physics objects each had their own mass, bounding volume or shape, both linear and angular velocities, accelerations and forces applied. The available bounding volumes for physics objects were limited to spheres, planes, and cuboids, both axis aligned and oriented.

3.2 Soft Bodies

It was decided that the approach to soft body simulation to be implemented was the spring-mass approach. This was because the high accuracy of finite element simulation was deemed unnecessary in a real time video game simulation, where

the soft body is unlikely to be the sole focus of the user; the relative simplicity of the spring-mass model was expected to be more performant in a video game scene with many objects, as the number of points could be relatively low, with only points on the surface of most objects being essential to maintain the desired shape at rest.

Therefore, each soft body would take the form of several point masses connected together with both compression and extension spring constraints. Point masses were not supported in the game engine, so a workaround was to instead use spheres with negligible radii, being practically zero. Any convex shape mesh was to be supported in soft body form. Whilst some concave shapes could be simulated by the designed approach, support for all concave shapes introduced more complex geometry and would require some degree of pre-processing before forming a stable mesh of springs and point masses, and so it was decided that convex shapes would be the main focus of the project.

3.2.1 Collision

Collision detection and resolution for soft bodies was designed to be simple, using pre-existing functionality in the game engine. Each of the soft body's vertices would be capable of colliding with other physics objects in the scene and reacting appropriately, conserving momentum according to the respective masses of each object. The edges and faces of the soft body itself however, would not feature collision detection, due to the time constraints of the project, though this could be developed further in the future.

3.2.2 Rendering

In the existing game engine, the rendering of meshes with a variable shape - including soft bodies - was not supported, beyond objects following a preset animation, which was not applicable to real time soft bodies, able to deform and react to the current game state at any given point in time.

Ideally, in order to render an object with variable shape, the positions of all vertices should be sent to the vertex shader each frame, so that a custom vertex shader is able to render the object with the new updated shape. This is the ideal design, however due to the time restrictions of this project, a simpler to implement, but less efficient design was chosen.

This approach was to send an updated mesh to the GPU each frame, with the new vertex positions. It is important to note that the uploaded mesh should take the place of the existing mesh in the GPU's memory, and not be allocated new memory, otherwise the available memory on the GPU would very quickly be exhausted.

It should be noted that this approach will likely scale in a much worse fashion than the ideal approach given, as the number of soft bodies able to be rendered in a scene will be restricted by the available memory on the GPU, as each soft body requires its own mesh, updated every frame. Uploading updated mesh geometry to the GPU every frame for every soft body will likely be slow, as the GPU must wait on the relatively slow transfer between the GPU and CPU to retrieve the mesh data before being able to continue rendering. [10, §28.3.1] The ideal implementation would instead allow each soft body with a common resting shape to share a single mesh on the GPU, and data sent to the vertex shader would vary the rendering of

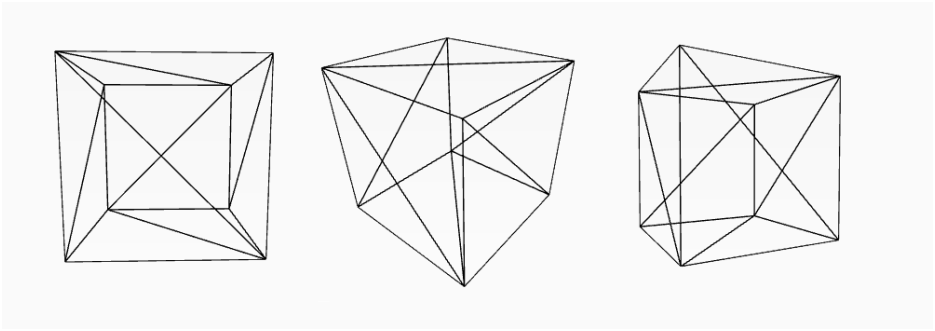

Fig. 7. A wireframe representation of the cube mesh used as the base for the default soft body, showing the vertices and connecting edges, viewed from several angles.

each soft body, much in the same way that different objects in the game scene are rendered by passing different positions, scales and orientations.

Additionally, the normals of each vertex index - that is, the direction orthogonal to the face the index - should be calculated each frame, and be sent to the GPU alongside the vertex positions. The method used to calculate this will be covered in the Implementation section, but it should be noted that performing these calculations in parallel on the GPU, instead of on the CPU, would likely yield better performance as the number of soft bodies increases.

4 Implementation

Unless stated otherwise, all examples of a soft body within this section will be of a cube with an edge length of 1 unit, and square faces consisting of two co-planar triangular faces, as shown in Figure 7.

4.1 Spring Constraints

The first component of the simulation to implement was spring constraints. Both compression springs, springs that oppose a compressing force reducing the distance between each end, and extension springs, those which oppose forces acting to pull the two ends of the spring apart, were implemented. In the case of the spring-mass model soft bodies, the two springs were always combined together between a given pair of points, to form a single spring capable of opposing both compression and extension.

Firstly, a spring constraint is given two physics objects to act upon. It will be the change in distance between these two objects, that will result on an opposing force being imparted on these same objects to mimic the effects of a spring. Then, after the spring is set to be either a compression or extension spring, a rest distance is supplied. This is the distance between the two objects at which the spring system will be at rest, the state it will always attempt to return to by applying forces to each object.

Finally, both a spring constant and a damping factor are required. The spring constant determines the strength of the forces applied by the spring constraint, per meter of deformation from its resting state. The damping factor is used in order to dampen the forces applied, to prevent an excess of energy entering the system,

leading to an unstable system which has erroneous and erratic motion. The damping factor is used as follows:

- The difference in velocity between the two point objects is calculated.
- Next, the dot product of this difference in velocity, and the positional vector from one point object to the other is calculated. This shows how closely the difference in velocity aligns with the line passing between the two objects.
- Then, this value is multiplied by the damping factor, and the force to be applied to the objects is incremented by the normalised direction vector between the objects, scaled by this value.

The produced damping force has an opposing direction to the force applied to each of the spring's point objects, therefore it acts to limit the motion produced by the spring. This is required, because effects such as air resistance, plastic strain limits of spring materials and energy loss through produced heat do not exist in the simulated environment, and springs in reality are not 100% efficient at redirecting the motion of objects due to these energy losses.

Every physics frame, these calculations are completed using the position and velocities of each object, and the respective forces are applied to the point object at either end of the spring constraint, resulting in the simulation of spring-like motion.

4.2 Soft body instantiation

To instantiate a soft body in the game world, several parameters are required. These include a position, scale in the X, Y and Z axis, spring constant value, damping factor, total soft body mass, and a polygonal mesh file from which the soft body is generated. Additionally, a texture and shader can be optionally provided to customise the soft body's appearance.

4.2.1 Position

The position provided is the starting location of the soft body, after it is instantiated. This position may or may not be 'inside' the soft body, the soft body mesh will appear with the origin of the mesh aligned to this provided starting position. Therefore, it is important to know where the origin of a mesh is, in relation the object, before creating a soft body from it.

4.2.2 Scale

The scale is provided in the X, Y and Z directions, and the position of each of the vertices that form the mesh, in each axis, is scaled by the scale value for that axis. Therefore, a negative value for scale in any axis will result in the positions being mirrored about the origin in that axis. The scale is useful, in order to create soft bodies with varying sizes in the game world, while originating from the same mesh file.

4.2.3 Spring Constant

As explained in Section 2.1, the spring constant determines the strength of the force imparted on each point mass at a regular time interval by a spring, proportional to

the distance from rest that the spring has been deformed. This effectively controls the 'stiffness' of the springs that form the soft body, and care must be taken to ensure that this value takes into consideration the mass of each point, as more massive objects will experience less acceleration from a given force, due to Newton's second law of motion.[9] A soft body made of springs with a high spring constant relative to the points' masses would begin to approach the behaviour of a rigid body, as any motion from the points would immediately result in a very large corrective force.

4.2.4 Damping Factor

The damping factor acts to reduce the total energy in the soft body system every frame, as described in section 4.1, to mimic the effects of energy loss in real world physics interactions. With this value too low, forces such as the acceleration due to gravity in a scene can cause the points to gain more and more energy, with nothing acting as an energy sink to limit or control the total energy in the system. For a cubic soft body with a spring constant of 20N/M, a damping factor of 0.2 was found to produce a reliably stable soft body.

4.2.5 Total Soft Body Mass

The total soft body mass, in Kilograms, is the combined mass of all point masses that form the soft body. Each point mass has the same mass as one another, so this total mass value is distributed evenly amongst the soft body's point masses.

4.2.6 Generating springs and masses

Firstly, the array of vertex indices in the mesh file is iterated upon, in order to determine the number of unique vertex positions the soft body mesh should contain. Using the cube mesh as an example, there are eight unique vertex positions, as each triangular face shares two vertices with an adjacent face.

Then, for each unique vertex position in the mesh, a soft body point is created, a physics object with a spherical bounding shape with radius of near zero. This shape was chosen, as the existing game engine requires all physics objects to have a bounding shape, does not support physically simulated objects with a scale of zero, and time restrictions for this project did not allow this to be changed in the engine. The total mass of the soft body is then divided and distributed equally between all points.

In order to identify appropriate locations for spring constraints, the index data of the mesh is used. In groups of three, referring to each of the vertices that make a single triangular face on the mesh, these indices are processed, and spring constraints are placed between each pair, wherever a polygon edge exists in the mesh. Both a compression and an extension spring constraint are generated, with the same spring constant and damping factor supplied for all springs in the soft body.

As many triangles share a common edge with another face in the mesh, it is likely that a vertex pair will already have occurred in the array, and already have spring constraints as the index array is iterated upon. If this is the case, a duplicate set of spring constraints is not generated, each set of two vertices may not have more than one compression and one extension spring directly connecting them together,

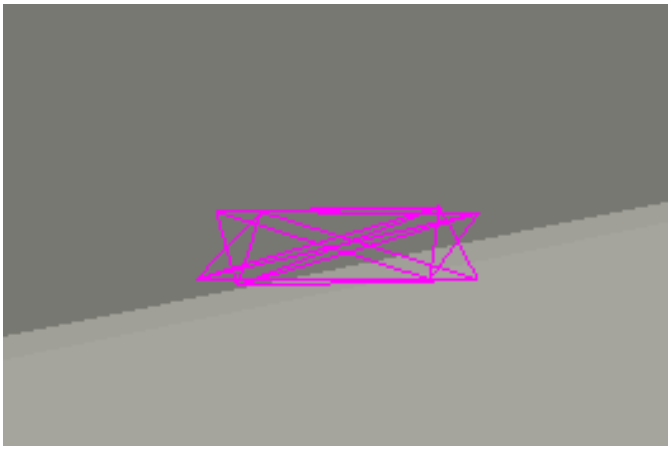

Fig. 8. A cubic soft body without additional supporting springs collapses into a flat set of points when any rotation occurs between two faces.

though many connections may exist indirectly, through mutual neighbour vertices.

4.3 Adding stabilising springs

At this point, some soft bodies would be capable of holding themselves together in a stable manner, returning to their rest shape over time. However, the polygonal meshes used in computer graphics are not designed with this use in mind, and so most soft body meshes require some post processing in order to form a network of springs and masses which will not collapse into a different stable shape.

For example, for the cube shown in Figure 7, the diagonal edges, and therefore springs, on each square face are rotationally symmetrical. This results in a shape that cannot hold a stable form upon experiencing a rotational force, and collapses into a flat set of points, such as in Figure 8.

In order to algorithmically add stabilising springs, such as a second diagonal spring on each face of the example cube, the soft body mesh is processed, and, for every pair of point masses, if another spring of the same length exists somewhere else in the soft body, then a compression and extension spring are added between the two points. Though not appropriate for all convex meshes, this approach successfully works to stabilise the cubic soft body, as shown in Figure 11, where additional diagonal supporting springs are present.

It would be worth investigating algorithms that could more intelligently determine the stability of a given mesh when converted to spring-mass soft body form, and identify the most appropriate places to add additional stabilisation springs.

4.4 Dynamic soft body rendering

To render the soft body object in real time, the existing rendering approach in the game engine cannot be used without modification. This is because the existing method assigned a mesh to an object at the point of object creation, and the position, scale and rotation of the object may only be changed by passing these values into the vertex shader on the GPU, which transforms the positions of all vertices in the mesh at the same time, and by the same amount. Although animation is possible in the engine, this is only possible with preset animation files, which must

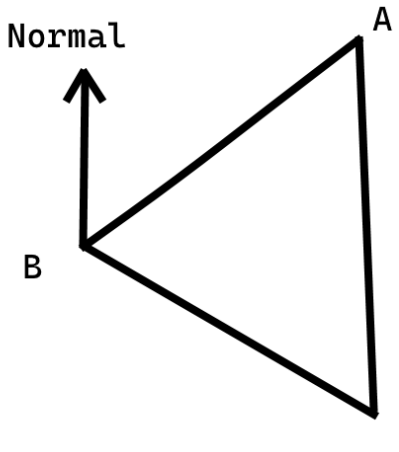

Fig. 9. For the triangle ABC, a normal can be calculated by finding the cross product of AB and CB.

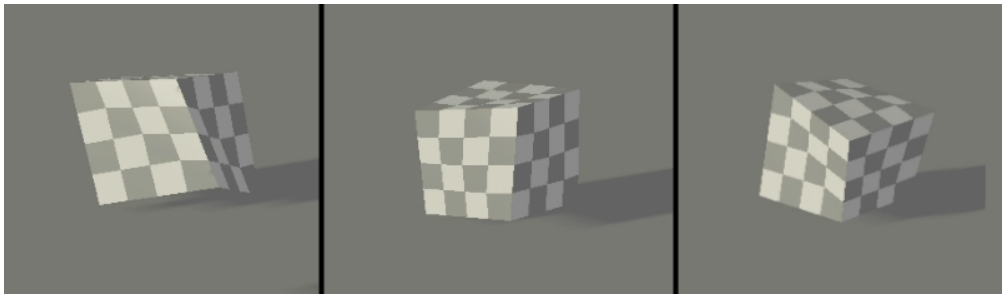

Fig. 10. Several frames taken from a recording of a soft body simulation, showing the mesh deforming over a period of time, and the lighting changing accordingly.

be loaded when the game scene initialises, and cannot dynamically change to reflect a changing game state or environment.

Therefore, an instance of the Mesh class is created alongside each soft body, and is updated every frame with new positions for each soft body point. This is then uploaded to the GPU, alongside the new face normals, calculated as described in section 4.4.1. It is important to note that the uploaded mesh replaces and occupies the same area of memory in the GPU as was allocated for the soft body's mesh upon creation, as to not cause a memory leak on the GPU.

This method is not optimal, as the transfer of data from the CPU to the GPU is typically very slow, and may cause a bottleneck in the rendering loop.[10] It was chosen, however, due to the time restrictions of the project, and section 3.2.2 describes a potentially more effective approach.

4.4.1 Calculating face normals

Each frame, the faces that form the soft body will likely change position, orientation or scale, and so it is also important to update the vertex normals, the direction vector orthogonal to the face, used in common approaches to lighting and shading in the rendering pipeline, such as Phong shading.[12] To calculate the normal vector of a face, the cross product of two edges is used. For example, for the triangle ABC in Figure 9, calculating the cross product of AB and CB results in a vector perpendicular to both vectors.

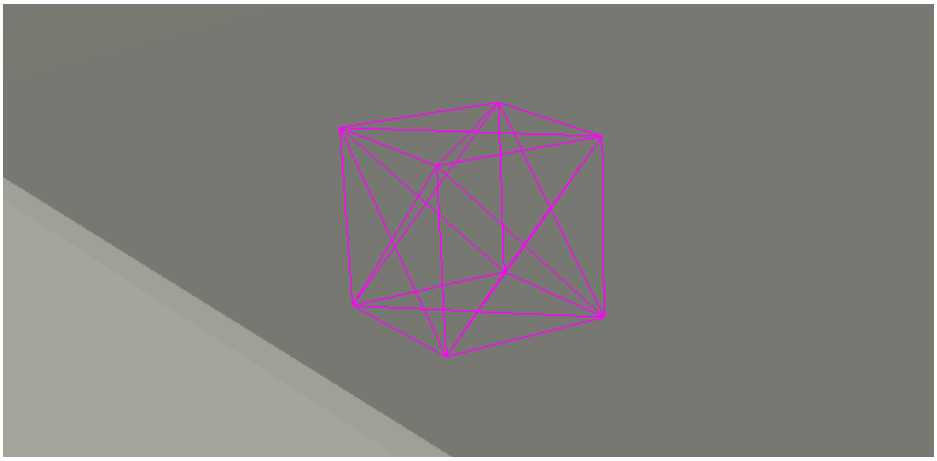

Fig. 11. A real-time wireframe visualisation of the soft body, using debug drawing functionality in the engine, showing each spring that forms the soft body. Note that resampling artifacts may be present in the image, depending on the viewing device.

4.4.2 Wireframe rendering

In order to assist in debugging the rendering code in this project, a second, much simpler rendering method was implemented, in which unshaded lines are drawn to the screen between each pair of points, for every spring constraint in a soft body, as shown in Figure 11. This makes use of existing debugging tools in the engine, and was used to visualise the shape of the soft body before the implementation of the shaded rendering, and when the performance of the physics functionality was being profiled in isolation.

5 Evaluation

This section presents the results of the project, describes how these results were found, before explaining why these results occurred, where possible.

5.1 Evaluation Environment

The soft body simulations described in this section were carried out on a desktop PC, with the following specifications:

- CPU - AMD Ryzen 5 5600G , at 3.9GHz
- GPU - NVIDIA GeForce RTX 3060, with 12GB of dedicated memory
- RAM - 64GB total at 2133MHz
- Monitor - 1920x1080 pixels

All other system specifications were not deemed relevant to the performance of the project, and were omitted for brevity. The system's specifications are generally equivalent to the average system specification of Steam users[15] - one of the world's largest video game distribution services - particularly the GPU and screen resolution.

No CPU or GPU intensive applications were running during the tests, besides the simulation itself, so that the most resources possible were available, and the CPU and GPU usage of other software did not influence the results. The tests were

carried out using the default release configuration for C++ in visual studio, which enables compiler optimisations, improving performance compared to the debug configuration typically used during development. This was done to more accurately reflect a released application, and the results that a real end user would experience.

Additionally, the simulation ran in full screen mode at 1920x1080 pixels, with no objects in the scene besides the floor platform and the soft bodies tested.

5.2 Testing

The soft bodies were simulated both with and without the dynamic rendering enabled, in order to differentiate the impact that the rendering system had on the simulation's performance. This was especially important, as it was expected that the imperfect rendering solution would impact performance somewhat significantly.

For both the wireframe and fully rendered soft body cubes, tests were carried out with an increasing number of soft body objects, from one soft body up until the simulation was no longer successful. The number of soft bodies was increased by 100% for each subsequent test, until a simulation was unsuccessful, and then values in between the last successful and unsuccessful test were used. This method was chosen in order to efficiently determine the limits of the soft body simulation system, with as few tests as possible, much like a binary search.

In all tests, the render time and physics tick period were recorded every 100ms over a period of 10 seconds, and the mean average was taken. GPU memory usage was recorded once at the start of each profiling instance, as the `glGetIntegerV()` call through OpenGL to the GPU could not be called every frame. Due to the time taken to retrieve information from the GPU using this function, calling it every frame would seriously impact the other results recorded. The lack of an averaged GPU memory usage value should not be an issue however, as no new meshes, shaders or textures are uploaded to the GPU once the simulation begins, aside from the dynamic meshes of the soft bodies, which replace existing meshes in the GPU memory, and do not increase their total memory footprint over time.

The render time is inversely proportional to the number of frames per second, and an acceptable render time is 33ms, with a target render time of 16ms or lower desired, as these values correspond to 30 and 60 frames per second, commonly achieved frame rates expected in the videogame industry. [14] The render time typically reflects the load on the GPU to render the graphics for each frame, and more complex graphical features can increase this time.

The physics tick duration represents how long it took the physics system to complete all the calculations required per tick, such as resolving collisions, solving constraints, and integrating acceleration and velocity for physics objects. In general, it is good practice to maintain a physics tick rate similar to the average frame rate, or higher, otherwise the motion of physics objects on screen may appear to move out of sync with the motion of static objects relative to the camera. A tick rate of 50tps, or a physics tick duration of 20ms or lower is the target value.

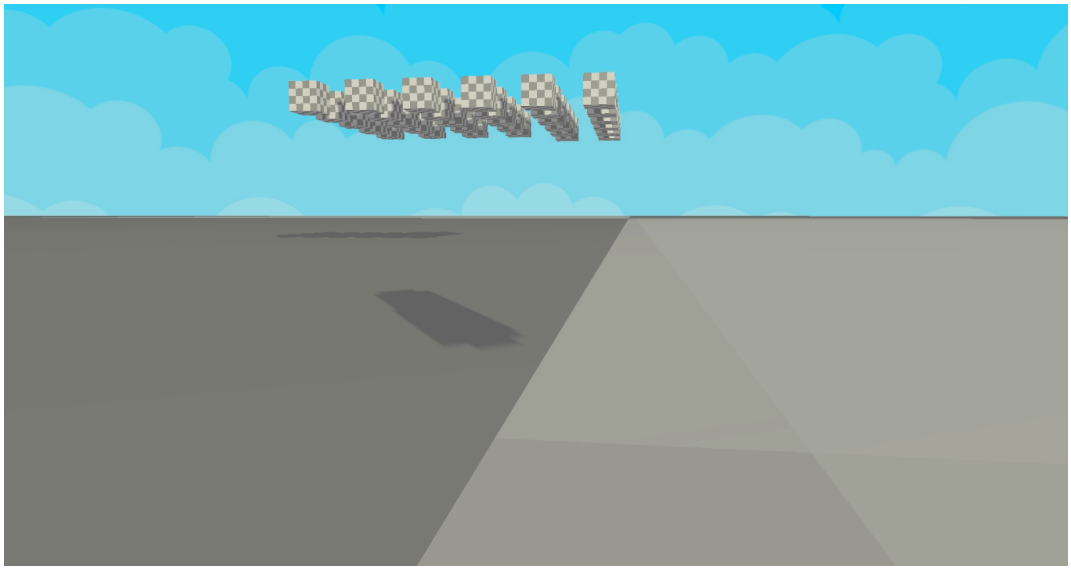

Fig. 12. A frame from a recording of a simulation, in which 32 cubic soft bodies are simulated.

X Value	Successful Simulation	Softbodies Simulated (2^X)	Avg. Render Time (ms)	Avg. Physics Tick time (ms)	GPU memory used (MB)
0	Yes	1	0.292	0.025	2181.7
1	Yes	2	0.304	0.044	2200.1
2	Yes	4	0.288	0.081	2203.4
3	Yes	8	0.31	0.161	2211.5
4	Yes	16	0.329	0.352	2220.4
5	Yes	32	0.411	0.884	2236.3
6	Yes	64	13.274	1.432	2291.9
6.5	Inconsistent	90	10.404	3.669	2294.7
7	No	128	8.004	4.442	2343

Table 1
While rendering dynamic soft body meshes, average render times, physics tick times, and GPU memory usage are shown, as number of soft bodies simulated increases.

5.3 Experimental Results

Results were recorded using the methods described in Section 5.2, and two tables were populated. Table 1 shows the results when rendering the soft bodies fully, using the dynamic mesh rendering system, and Table 2 shows the results recorded from simulations using only the basic wireframe rendering method, described in Section 4.4.2.

Three graphs were formed from these results, and the results they show are

X Value	Successful Simulation	Softbodies Simulated (2^X)	Avg. Render Time (ms)	Avg. Physics Tick time (ms)	GPU memory used (MB)
0	Yes	1	0.327	0.025	2264.9
1	Yes	2	0.337	0.045	2262
2	Yes	4	0.371	0.088	2266.6
3	Yes	8	0.335	0.169	2252
4	Yes	16	0.348	0.405	2255.3
5	Yes	32	0.356	0.935	2242.7
6	Yes	64	0.402	2.524	2258.1
6.5	Inconsistent	90	0.461	4.93	2256.2
7	No	128	0.483	4.526	2257.6

Table 2
While rendering soft bodies using the wireframe method, average render times, physics tick times, and GPU memory usage are shown, as number of soft bodies simulated increases.

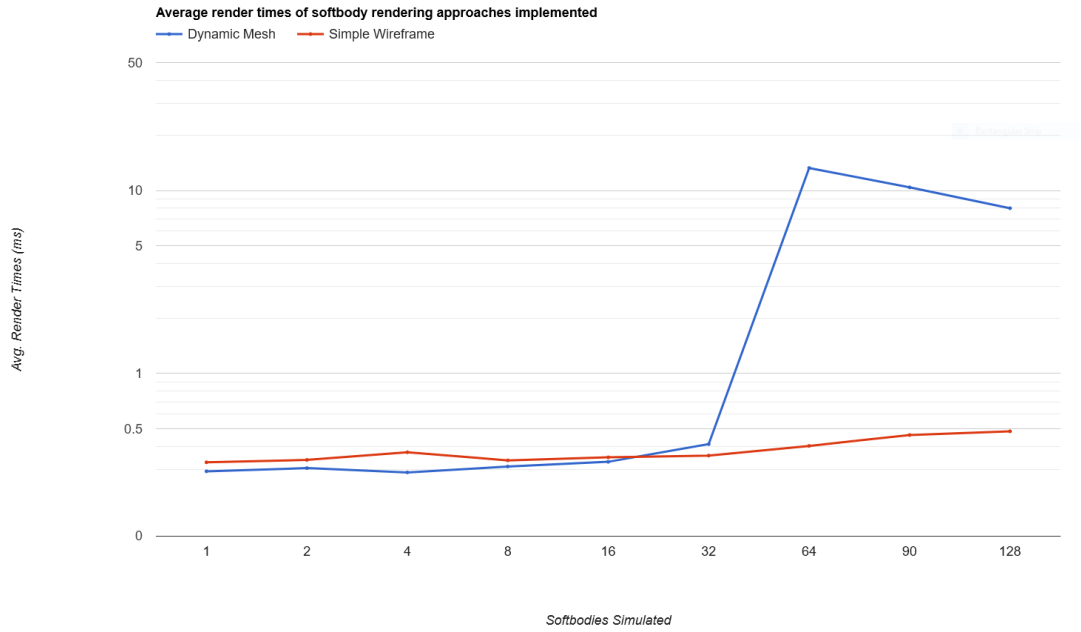

Fig. 13. A graph showing the average render times of implemented soft body rendering approaches. Note that the axes are not linear.

explored below. The simulation of 128 soft bodies consistently failed, as the physics system could not solve every spring constraint in time for the next physics update, and the simulation of 90 soft bodies failed on some occasions, and was successful on others, indicating that 90 soft bodies is approaching the limit of what can be simulated by the system in its current state.

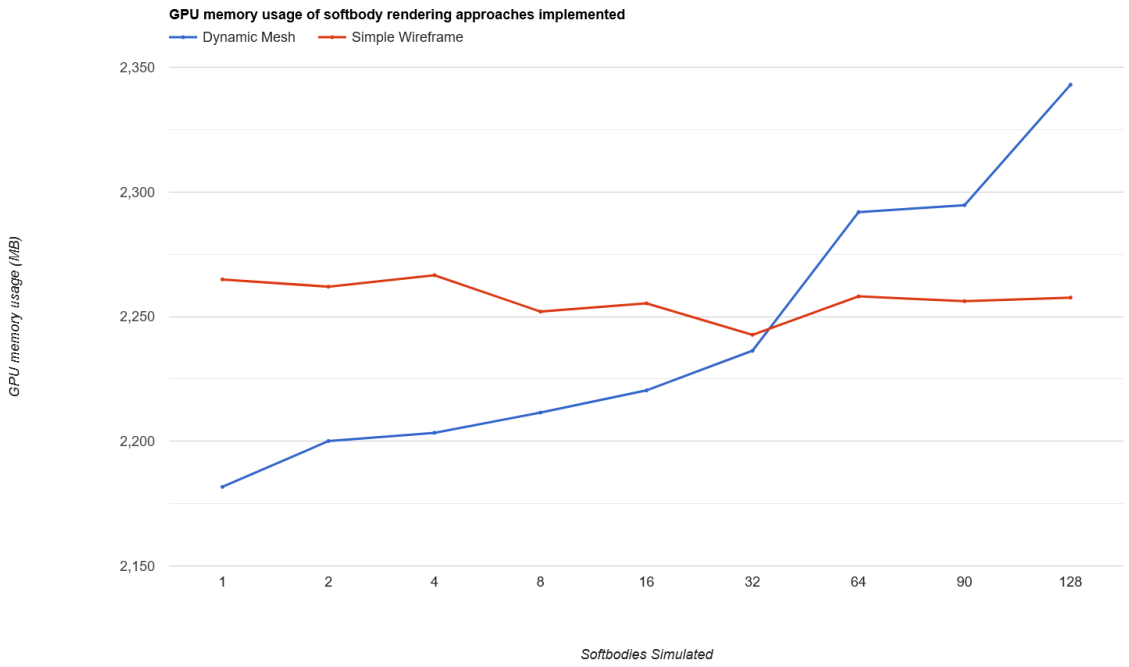

Fig. 14. A graph showing the GPU memory usage of implemented soft body rendering approaches

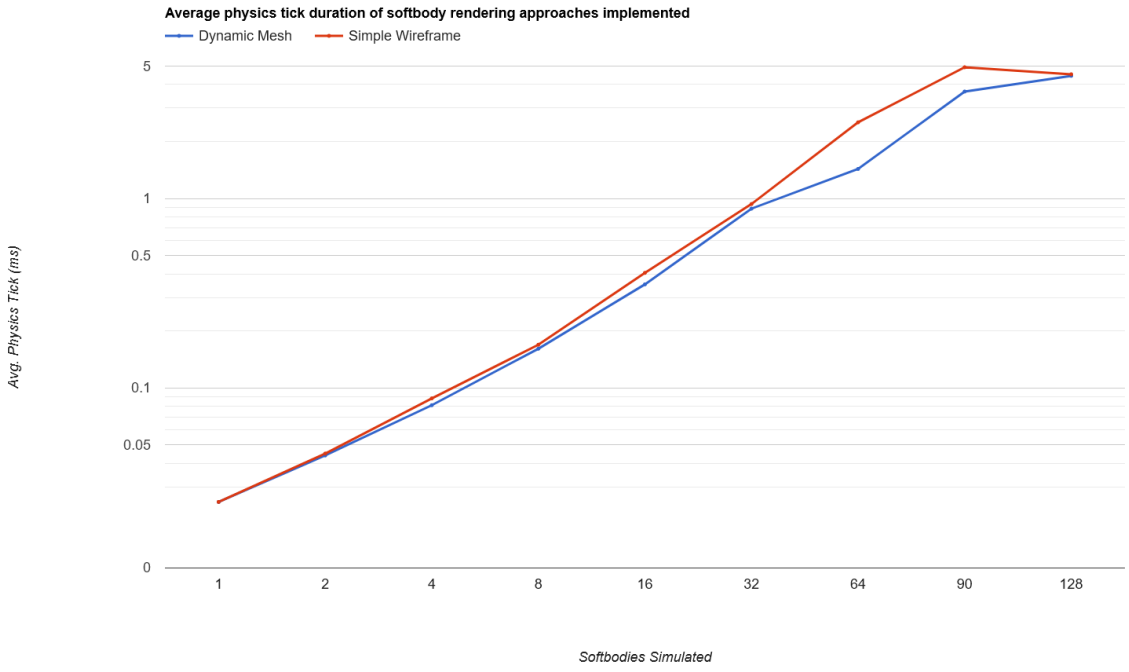

Fig. 15. A graph showing the average physics tick duration of implemented soft body rendering approaches. Note that the axes are not linear.

5.3.1 Rendering Times

The target rendering time was 16ms or lower, and Figure 13 shows that this target was not only met, but that the average rendering times of both rendering approaches were significantly faster than this target time. Whilst the wireframe rendering

approach has a fairly constant rendering time as the number of soft bodies doubles, the dynamic mesh rendering approach appears to take significantly longer to render when 64 soft bodies are in the scene, compared to the previous test of 32. Repeated tests showed this to be the case, but it is not immediately clear why this occurs. Further investigation may be required, to discover if the more potentially more optimal approach described in Section 3.2.2 would experience the same issue.

The reduction in rendering times seen beyond 64 soft bodies does not accurately reflect the performance of the system, as the physics system could not simulate the soft bodies in real time beyond this point, and therefore the load on the physics simulation may affect the final two data points at both 90 and 128 soft bodies.

5.3.2 GPU memory usage

Figure 14 shows the GPU memory usage of each rendering approach. While the wireframe method shows a generally consistent memory usage, the dynamic mesh rendering approach uses more memory as more soft bodies are simulated, with the two methods using a roughly equal amount of memory at around 32 soft bodies simulated. Whilst the memory usage of the dynamic mesh approach is not ideal, it does not use as much memory as expected and the roughly 150 extra megabytes used for the 128 soft body simulation compared to the single soft body simulation is not completely unreasonable, considering the 12,000MB total memory available to the GPU used.

In the tests carried out, the physics system was the bottleneck, and the impact on frame rate and rendering times was not substantial before reaching the number of soft bodies at which the simulation failed. It would still be worth investigating the alternative rendering method described in Section 3.2.2, in order to determine if any further optimisations could be made here.

5.3.3 Physics Tick Duration

Figure 15 displays the average physics tick duration for each rendering approach, as the number of softbodies simulated increases. As expected, the rendering approach used does not appear to have any significant effect on the physics tick rate of the simulation. The physics tick duration appears to scale linearly with the number of soft bodies simulated - note that both axes in Figure 15 are non-linear, but a doubling of the number of soft bodies consistently results in an approximate doubling to physics tick duration.

Beyond 64 soft bodies simulated, the physics system would occasionally drop physics frames, due to the time required, and so the average physics tick duration does not increase substantially beyond this point, but the simulation was not successful here.

6 Conclusions and further work

6.1 Conclusions

Over the course of this project, a soft body simulation system was added to a game engine, including both the physical simulation of deformable objects through the

use of a spring-mass model, and the rendering of meshes deforming in real time.

Through experimental testing, it was found that 64 cubic soft bodies could consistently be simulated simultaneously and successfully in a scene. When rendering the soft bodies, the simple wireframe approach, though lacking in visual fidelity, required a generally consistent amount of system resources and did not suffer significant reductions in frame rate as the number of soft bodies increased. On the other hand, the dynamic mesh rendering approach resulted in far superior visual quality, with a lower impact on frame rate than expected, albeit more than the wireframe method.

6.2 Improvements and Future Work

This section considers both how the project could have been done differently, and potential future improvements or projects that could expand upon the work done in this project.

6.2.1 Soft body rendering

As explained in Section 3.2.2, uploading a new mesh to the GPU every frame for every soft body in a scene is not optimal. Instead, a vertex shader could be written, which could be supplied the new vertex locations as uniform variables, and the calculation of normal vectors could then be completed on the GPU, in parallel. This has the potential to improve render times, as the system currently performs several calculations for each soft body face in series every frame, including trigonometric functions, which may be taking a considerable amount of the rendering time.

6.2.2 Soft body collision

Currently, only the point masses that form the soft body provide collision detection and resolution for soft body objects. This is sufficient for collisions against large, flat geometry, but would not function when colliding with rough terrain, able to enter the gaps between points, for example. In order to improve the collision detection, more point masses could be added to each face, or alternately, collision detection could be added for each triangular plane on the soft body. The latter method would likely be more efficient, but would be unable to deform around smaller obstacles, like the former method could.

6.2.3 Stabilising spring generation

Using polygonal meshes to form spring-mass soft bodies directly typically results in soft bodies which are unstable in one or more axes, linearly or angularly, as explained in Section 4.3. The current method of processing vertices to generate stabilising spring joints works for cubes, but has limited effectiveness for other convex shaped meshes.

It would be worth exploring algorithms that could be used to determine the stability of a given mesh when converted to spring-mass soft body form, and locating the most effective positions to place additional springs for stability. This may be computationally expensive, and so could perhaps take the form of separate application that a developer could use to process polygonal meshes used for computer

graphics into stable spring-mass soft body meshes, in order to not impact load times in a simulation.

6.2.4 Visual-Physical mesh variation

With the soft body simulation system created, the mesh used for both the rendering and physical simulation must match. For the example cube soft body, this is not an issue, but in the case that a game developer wanted to use a higher detail mesh, this would substantially impact performance due to the number of point masses and spring constraints that would be required.

In many cases, the faces on a high quality polygonal mesh can be so small, that the resultant spring constraints in a generated soft body would not make a substantial difference to the physical properties of the soft body. Therefore, it may be worth investigating methods by which vertices of a render mesh can be mapped to points on a simplified soft body mesh, in order to both maintain high quality graphics, and responsive physics simulation.

7 Acknowledgements

Many thanks to Richard Davison for providing the underlying game engine framework from which the engine used was developed, without which this project may not have been possible.

References

[1] "Finite Element Analysis Software," Autodesk, [Online]. https://www.autodesk.co.uk/solutions/finite-element-analysis (accessed May 9, 2024).

[2] P. Volino and N. Magnenat Thalmann, "Implementing fast cloth simulation with collision response," Proceedings Computer Graphics International 2000, Geneva, Switzerland, 2000, pp. 257-266, doi: 10.1109/CGI.2000.852341.

[3] C. Kittel and P. McEuen, Introduction to solid state physics. John Wiley & Sons, 2018, pp. 120-121. [Online]. Available: http://metal.elte.hu/ groma/Anyagtudomany/kittel.pdf (accessed May 10, 2024).

[4] X. Provot, "Deformation Constraints in a Mass Spring Model to Describe Rigid Cloth Behavior," Proceedings of Graphic Interface'95, pp. 147-154, 2005. [Online]. Available: http://171.67.77.70/courses/cs468-02-winter/Papers/Rigidcloth.pdf (accessed May 10, 2024).

[5] R. Hooke, Lectures de Potentia Restitutiva, Or of Spring Explaining the Power of Springing Bodies. John Martyn, 1678, pp 1-5.

[6] Mar. 2007. [Online]. Available: https://en.wikipedia.org/wiki/Polygon_mesh#/media/File:Dolphin_triangle_mesh.png

[7] M. Tamis and G. Maggiore, "Constraint based physics solver", 2015. [Online]. Available: http://mft-spirit.nl/files/MTamis_ConstraintBasedPhysicsSolver.pdf (accessed May 13, 2024)

[8] "Physics constraints and solvers," Newcastle University Games Engineering, [Online]. Available: https //research.ncl.ac.uk/game/mastersdegree/gametechnologies/physicstutorials /8constraintsandsolvers/Physics%20%20Constraints%20and%20Solvers.pdf (accessed May 13, 2024).

[9] "Newton's second law: F = ma," Encyclopædia Britannica, [Online]. Available: https://www.britannica.com/science/Newtons-laws-of-motion/Newtons-second-law-F-ma (accessed May 13, 2024).

[10] C. Cebenoyan, "Chapter 28 graphics pipeline performance," NVIDIA Developer, [Online]. Available: https://developer.nvidia.com/gpugems/gpugems/partvperformanceandpracticalities/chapter28graphicspipelineperformance (accessed May 14, 2024).

[11] “Tutorial
8: Index Buffers & Face Culling,” Newcastle University Games Engineering, [Online]. Available: https://research.ncl.ac.uk/game/mastersdegree/gametechnologies/physicstutorials/8constraintsandsolvers/Physics%20%20Cons (accessed May 13, 2024).

[12] G. Bishop and D. M. Weimer, ‘Fast Phong shading’, in Proceedings of the 13th Annual Conference on Computer Graphics and Interactive Techniques, 1986, pp. 103–106.

[13] X. Jiang, H. Ren, and X. He, ”Simulation of Mooring Lines Based on PositionBased Dynamics Method”, IEEE Access, vol. 7, pp. 4-7, 2019, [Online]. Available: https://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=8850004 (accessed May 14, 2024).

[14] ”Frames per second (fps) in tv, cinema, and gaming”, IONOS SE, Jun. 2022. [Online]. Available: https://www.ionos.co.uk/digitalguide/server/know-how/fps/ (accessed May 13, 2024).

[15] ”Steam Hardware & Software Survey: April 2024”, Valve Corporation, Apr. 2024. [Online]. Available: https://store.steampowered.com/hwsurvey/Steam-Hardware-Software-Survey-Welcome-to-Steam (accessed May 14, 2024).

[16] D. Terzopoulos, J. Platt, A. Barr, and K. Fleischer, ‘Elastically deformable models’, SIGGRAPH Comput. Graph., vol. 21, no. 4, pp. 205–214, Aug. 1987.

[17] S. Timoshenko and J. N. Goodier, ’Theory of Elasticity’. McGraw-Hill, pp. 1-2, 1969.

[18] M. Desbrun, P. Schröder, and A. Barr, ‘Interactive Animation of Structured Deformable Objects’, Jan 1999, pp. 1–8.

[19] Jun. 2023. [Online]. Available: https://media.geeksforgeeks.org/wp-content/uploads/20230227130804/Hookes-Law-Stress-And-Strain-(1).png