NTNU
Norwegian University of
Science and Technology

Mesh Deformation Through Mass-Spring Systems in Unity

Per-Morten Straume

16-11-2018

IMT4888 - Specialization in Game Technology
Master of Science in Applied Computer Science
7.5 ECTS
Department of Computer Science
Norwegian University of Science and Technology,

Contents

1 Introduction

Mesh deformation is a broad area and is done for many different reasons. It can be used for collision deformation, animation, physics simulation, and the likes. However, mesh deformation can be quite intimidating to approach, due to the mathematical intensity of the field and lack of good resources in layman's terms.

This paper will present different approaches to mesh deformation as visual effects in interactive applications, and two implementations for mesh deformation using the mass-spring model. One is a relatively simple stress ball like deformation, using semi-implicit Euler integration, the other being a cloth simulation using Verlet integration. This will be followed by an overview of uses for the implementations, and where the technology might be going in the future. Following this is a discussion on useful resources and tutorials that can be helpful for people who consider getting into the field.

2 Problem Description

2.1 Description

Video Games are becoming more and more complicated with each new release, supporting different visual effects and physical simulations to enhance the player's immersion, among these effects are deformation. Mesh deformation can happen in a lot of different cases, for example when cars crash into each other in a car game they might end up with a deformed model due to the collision. Other examples are materials that are deformed upon interaction, like how the snow deforms in God of War[1], flags that are waving in the wind, or plants moving as characters collide with them, only to bounce back upon leaving the collision.

Realistic mesh deformations have long existed in mediums such as animated films. These mediums have the advantage that they can be rendered offline and rendering for a couple of hours per frame is not necessarily a problem. Because of this, films can make use of analytical real-life physics models to create realistic behavior. In the case where they need to use approximations and computational models they can afford to spend a lot of time and resources on the simulation process, to end up with a satisfying result. However, this is not feasible within an interactive application, where high frame rates are a necessity for a functioning, comfortable and immersive experience. As a result, games and other interactive mediums where total realism is not a necessity have looked at other solutions that create plausible results given the constraints of the environment.

2.2 Approaches

Several different approaches exist for mesh deformation. For example, it is possible to do offline pre-computations, i.e. create common deformations for collisions offline, and apply the deformation or switch to the model with the deformation upon collision. This can be a good strategy in situations where the deformation will happen so fast the switch is unnoticeable to the user. An example here could be in the case of a high-speed car-crash, upon impact the user will probably not notice that the car they are sitting in instantly got deformed, rather than it happening by continuous collision with the other object. This is especially true if the switch is covered a bit with some particle effects as a result of the collision. Ideally, if you pre-compute models you should try to ensure that there are enough variations so that the user feel that the different deformations are varied enough. Hauser et al.[1] discusses modeling these deformations can be done through Modal Analysis.

If pre-computations is not a viable option, real-time methods also exist. One such approach is the Finite Element Method [2] which will yield quite realistic results, this method is quite sophisticated, but computationally and memory expensive [3], however, hardware has evolved a lot since these

[1]God of War Snow Deformation Video: https://www.youtube.com/watch?v=BEgY9k89s3o

claims were made, meaning that it might be more viable on modern hardware. Free Form Deformation is also a way to model deformation, while it is more fitting for interactive object modelling [3] it has been extended in different ways to allow for other types of deformation [4].

Rodrigues et al.[3] proposes D4MD, a hybrid model of physically-based deformations (such as the finite element method) and geometrical methods (like the free form deformation) to use in vehicle simulation games.

This article will focus on the mass-spring model, a classic approach proposed by Xavier Provot [5] to implement mesh deformations.

2.2.1 What is the Mass-Spring Model?

The mass-spring model is usually presented in a quite intuitive manner. Each vertex within a mesh is considered a particle with its own mass. Between the vertices are springs which try to hold the whole mesh together. When no forces are applied to the vertices within the model, the lengths of the springs are in their desired "resting" state. When forces are applied to the vertices and they start moving, the springs holding the vertices together will become stretched and will try to pull the particles back to return to their resting state[6, 7, 5].

3 Implementation

3.1 Background Theory

When implementing a real-time mesh deformation system certain decisions must be made. Firstly one must decide upon the model to use for the mesh deformation, the more brute-force mass-spring system or the algorithmic approach using proper physics. As previously indicated we are working with a mass-spring system for this implementation.

Following the decision of models we need to decide on an integration scheme, how should we model the changes in acceleration, position, and velocity of our vertices? Many different integration schemes exits, such as explicit Euler integration, Runge Kutta, or Verlet integration, all with different properties and difficulty of implementation. The two used in the following implementations will be semi-implicit Euler integration and Verlet integration.

Lastly, we need to decide if we go for a kinematic system where we model springs as constraints, or a dynamic system where we model the springs as proper springs working with Hooke's law[8], both of which are presented within this chapter.

3.1.1 What is Semi-Implicit Euler Integration?

Semi-Implicit Euler integration is perhaps the integration scheme that is easiest to follow and comes most natural to programmers when they implement movement in a real-time system for the first time. It is also a scheme used within a lot of physics engines [9].

In Semi-Implicit Euler integration we keep the total accumulated force, velocity and position of each game object. Upon each time-stepped update of the system we divide all the force that has been applied to an object by the mass of the object to get the acceleration of the object. Multiplying this acceleration with the timestep yields the change in velocity of the object, which added on the previous velocity becomes the new velocity. This new velocity is then multiplied with the timestep and added to the position, reflecting the change in position over time. The code for this can be seen in Listing 3.1.

```
private void SemiImplicitEuler(float dt, GameObject go)
{
    var acceleration = go.force / go.mass;
    go.velocity += acceleration * dt;
    go.position += go.velocity * dt;
}
```

Listing 3.1: Semi-Implicit Euler Integration

```
private void Verlet(float dt, GameObject go)
{
    var acceleration = go.force / go.mass;
    var curr = go.pos + (go.pos - go.prevPos) + acceleration * dt * dt;
    go.prevPos = go.pos;
    go.pos = curr;
}
```

Listing 3.2: Verlet Integration

Advantages & Disadvantages

The largest advantage of the semi-implicit Euler integration is its ease of implementation, as well as the generally low computational complexity. However, it is not entirely stable, meaning that you can get into situations where your numbers start exploding.

3.1.2 What is Verlet Integration?

Rather than storing velocities and positions of each game object like the Euler integration, Verlet integration stores the current and previous positions of each game object. Per timestep, the current velocity is calculated implicitly by subtracting the current position from the previous one. Changes to the current velocity are also modeled via positional data by integrating the game object's acceleration twice over the timestep.

The new position of a game object following a Verlet integration is then: The current position added with the calculated velocity and the positional change due to acceleration. Code for this can be seen in Listing 3.2.

Advantages & Disadvantages

Just like semi-implicit Euler integration this integration model is relatively easy to implement, and relatively fast. Due to the velocity being calculated based on the positions, it is very hard to end up in a situation where the velocities and positions come out of sync, meaning that the model is quite stable. I would, however, argue that while it is easy to implement it is not as easy to understand as the Semi-Implicit Euler integration, at least upon the first contact with the model. I needed a proper intuitive explanation from Mosegaard[7] before I understood what was really going on. Additionally, Verlet integration, at least with this implementation requires a fixed time step, otherwise, the velocity calculation won't be correct [10].

3.1.3 Kinematic or Dynamic approach?

When working with mesh deformation one needs to decide upon a kinematic or a dynamic approach for controlling the springs.

Within a kinematic approach, the springs between the vertices are thought of as constraints that you need to solve for. Essentially this means that if two points are too far apart from each other, you move them a bit closer together, potentially over several iterations to ensure that you are not creating new inconsistencies in the model. The kinematic approach is the easiest to implement of

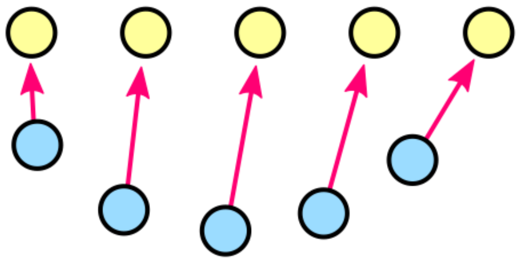


Figure 1: Modified vertices(yellow) are pulled towards their original position(blue) by the springs (pink)[6]

the two and can never explode, however, the extra iterations needed to solve the constraints can have performance implications.

The dynamic approach on the other hand models the springs as actual springs that correct themselves through forces according to the laws of physics, such as Hooke's Law. This approach is much harder to implement and can lead to situations where the model spirals out of control (i.e. explodes). Additionally, while implementing soft springs in with this approach is not that difficult, harder springs require more complicated integration schemes[8].

While both of these approaches imply different terminology for the connections/springs within a system I will refer to the connections between vertices as springs for the rest of the article, as the name of the model is "mass-spring system".

3.2 Implementation

Once all the decisions of the deformation model, integration scheme and kinematic/dynamic approach has been made it is possible to start the implementation of mesh deformation.

3.2.1 Dynamic Approach - Ball Deformation

Jasper Flick[6] shows an implementation[1] where the springs are implied based on the original position of each vertex and their new position. The implementation takes a dynamic approach to deformation and follows a semi-implicit Euler integration scheme.

In this implementation, we keep a copy of the initial vertex positions, as well as a collection of the modified vertices. Additionally, we keep a collection of the velocity of each vertex, which we update each time force is added to a vertex.

The springs come into play when updating the positions of all the vertices based on their velocities. Each spring within this system is the vector between the initial positions of the vertices in the mesh and their current positions, as seen in Figure 1. Upon update of the vertices, the force of a spring is applied to the velocity of a vertex in the direction of the spring, moving the vertex back towards its initial position. The code for this can be seen in Listing 3.3.

[1]Video of this implementation in action: `https://www.youtube.com/watch?v=f-5RzfN8kjw`

```
private void UpdateVertex(int i)
{
    Vector3 velocity = mVertexVelocities[i];
    Vector3 spring = mDisplacedVertices[i] - mOriginalVertices[i];
    spring *= UniformScale;
    velocity -= spring * SpringForce * Time.deltaTime;
    velocity *= 1f - Damping * Time.deltaTime;
    mVertexVelocities[i] = velocity;
    mDisplacedVertices[i] += velocity * (Time.deltaTime / UniformScale);
}
```

Listing 3.3: Catlike coding mesh deformation vertex update

Usage

As seen in the video, although simple, the implementation provides quite a lot of flexibility, toying with the different variables can lead to some interesting visual effects. For example, applying negative force to the sphere can be used to create a visual effect similar to that of a star being swallowed by a black hole[2]. Applying outward force from the inside of the model can create a relatively convincing effect of something moving around inside the object[3]. Giving the sphere a high SpringForce will lead to it being harder to deform, and more bouncy when trying to return to its original shape.

Limitations

As mentioned by Jasper Flick [6] this implementation is not a physics simulation, while the mesh is deformed, the colliders and physical representation of the object stay the same.

Additionally, none of the vertices are connected to each other through springs, and therefore move completely independent from each other. This means that if we were to for instance pin two of the vertices to their initial position, and then apply force to the mesh none of the other vertices would respect that two of them were locked in place, leading to less than satisfying results.

3.2.2 Kinematic Approach - Cloth Simulation

Cloth simulation is a place where the springs are more apparent than the previous implementation[4]. The implementation is based on that of Jesper Mosegaard[7] and Thomas Jakobsen[11]. Unlike the previous implementation, the springs here are explicit and ensure that vertices are not handled in isolation but rather are affected by the other vertices in the system. This gives greater control in that it allows you to apply force to a specific vertex and the whole system responds, rather than having to apply the force to all the vertices.

Spring Types

There are three types of springs within this implementation, all serving the same purpose of holding the model together, but in different ways. The different types can be seen in Figure 2.

[2]See: https://youtu.be/f-5RzfN8kjw?t=62
[3]See: https://youtu.be/f-5RzfN8kjw?t=28
[4]Video of cloth simulation: https://www.youtube.com/watch?v=SY9PPE18Kmo

Figure 2: The different spring types (here labeled constraints). Image taken from Jesper Mosegaard[7]

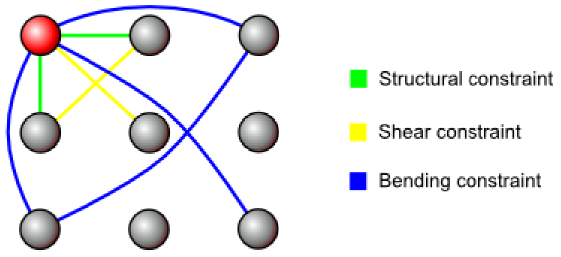
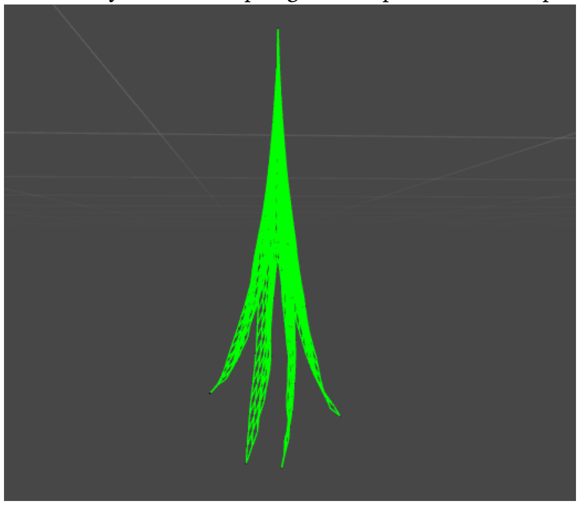


Figure 3: Cloth of only structural springs in the process of collapsing into itself



Structural Springs

The structural springs are the vertical and horizontal connections between the vertices, which ensures that the model stays in one piece. However, they alone are not enough, as the model can collapse into itself in a two dimensional space[12] this can be seen in Figure 3.

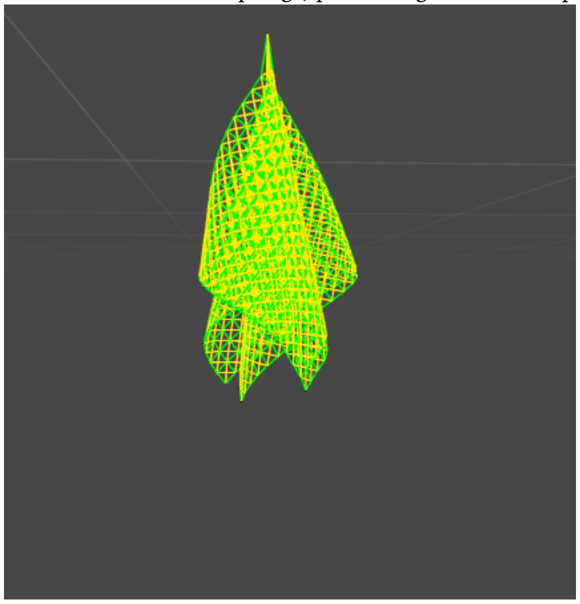
Shear Springs

Shear springs fix the issue mentioned above by connecting each vertex to their diagonal neighbors. When the vertices are about to start collapsing in on themselves the shear springs will push them apart again[12]. With this, the model also acts in a more three-dimensional space, as the vertices need to make use of the third dimension to satisfy the shear springs, as seen in Figure 4.

Bending Springs

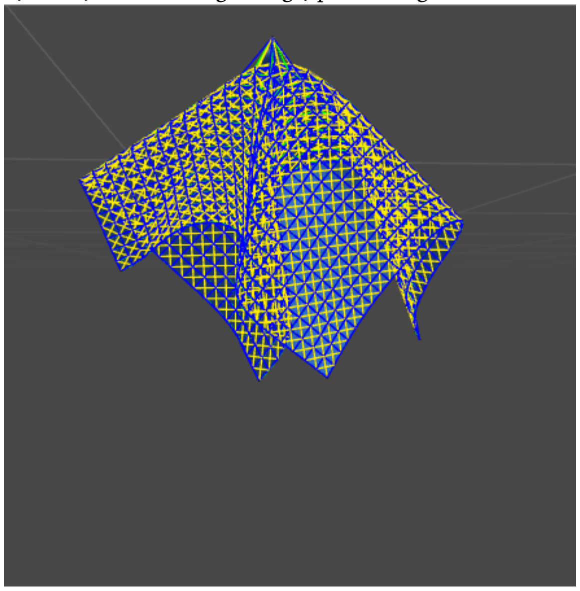
Technically you only need these structural and shear springs to create an ok looking cloth simulation, however, a third type of spring called a bending spring can be added. These are connected between each vertex and their second neighbor, which can help fix cases where the second neighbor

Figure 4: Cloth of structural and shear springs, preventing it from collapsing into 2D space



of a vertex is unrealistically placed in the system. This, in general, leads to the cloth becoming more flat, as extreme differences between the vertices are corrected for through the second neighbouring vertices[12]. While not as obvious, this can be seen in Figure 5.

Figure 5: Cloth of structural, shear, and bending strings, preventing it from bending unrealistically into itself.



```
private void FixedUpdate()
{
    for (int step = 0; step < ConstraintIterations; step++)
    {
        for (int i = 0; i < mSprings.Count; i++)
        {
            var spring = mSprings[i];
            var diff = mPositions[spring.V2Idx] - mPositions[spring.V1Idx];
            var dist = diff.magnitude;
            var correction = (diff * (1 - spring.RestDistance / dist)) * 0.5f;

            mPositions[spring.V1Idx] += correction;
            mPositions[spring.V2Idx] -= correction;
        }
    }
}
```

Listing 3.4: Semi-Implicit Euler Integration

Implementation

In this implementation we keep all the springs within a collection, each spring knowing which two vertices they are connected to and the desired rest length of the spring. Upon update, we iterate through all the vertices and calculate their new positions based on the Verlet integration scheme.

Following the update is fixing the constraints. This is done by iterating through all the springs and correcting the distance between the vertices it is connected to. This needs to be done several times as to avoid the model seeming to elastic. The code for this can be seen in Listing 3.4.

Usage

Due to its more sophisticated nature, this implementation offers more flexibility than the previous one. Simulations of entire pieces of clothes like curtains or flags are obvious ways of using this model. However, it is not a requirement of the model that every vertex in a mesh are simulated and part of the mass-spring system, you can also just simulate parts of it. An example here is if you have a character wearing a dress, you probably do not need that every part of the dress has realistic cloth physics, it is probably enough that just the bottom of the dress reacts to the environment to create a believable appearance.

Another use case with this model could be to have a cloth that tears at a certain point because it is stretched too much, or that parts of the cloth are cut off. This should not be too difficult, you would need to remove the springs between the vertices that have been disconnected from the rest of the model. Additionally, you would have to ensure that all the cuts were in line with the triangles of the mesh.

Issues

Although there are many advantages of this model, like how it is relatively easy to implement, gives quite good results and cannot "explode", it does suffer from a few issues. A problem with this implementation is that it can get into unsolvable states. I.e. when you apply forces to model continuously it can get into a situation where it never rests. Such a situation can be seen in the video of the implementation after around 25 seconds[5]. Here the forces of gravity are continuously being applied to the vertices, in the real world the bottom of this cloth would stop moving after a while. However, in this implementation, the cloth never stops moving because the springs never manage to get into a situation where they all are "satisfied". This is a weakness of the kinematic approach which was chosen for this implementation[8].

The number of times you need to iterate over the constraints to satisfy them is also an issue. Too few iterations and the cloth will look too elastic, too many iterations and it can seem too hard, additionally, the more iterations you add, the more processing time is needed, which can become a performance problem.

Extensions

Matthew Fisher[13] suggests some extensions to the model, such as quilting. Currently, the cloth looks like infinitely thin silk, which is not particularly realistic, quilting would add thickness to the model, allowing it to look and act more like wool or cotton. Fisher suggests doing this with a marching cubes algorithm. However, that would potentially be more costly to simulate and to draw.

Another extension could be self-collision or collision with other soft bodies, however, this is very challenging to implement and quickly ends up being really expensive in terms of performance as well.

Moving these calculations to the GPU is also a possibility, as a lot of the calculations are data parallel and would probably benefit quite a lot for being executed in a parallel environment.

This also implementation does not make any distinction between the different types of springs, extending the implementation to react differently based on the different spring types could lead to more realistic behavior.

[5]See: https://youtu.be/SY9PPE18Kmo?t=25

4 Uses

While cloth simulation and simple deformation effects are obvious applications of the mass-spring system, it has a variety of interesting applications.

One example is to model rope using a mass-spring system. The rope can be modeled as a collection of particles connected together through springs which can stretch and bend to a specified degree. The same line of thinking could then be extended to hair simulation.

Mass-spring systems are also used outside of the video game world. For example, Moosegaard [7] shows how a GPU accelerated mass-spring system is suitable for surgical simulations. Moving the simulation over to the GPU allowed them to reach much faster convergence rates for larger and more detailed organ models, leading to more realistic simulations of organs being cut and morphed. Similarly, Nedel [14] shows simulations of muscle deformations using a mass-spring system, introducing angular constraints to avoid situations where a model can completely change shape.

Müller[2] also brings up how simulators like these could be valuable for animators on projects where rendering takes a long time, as it would allow them to iterate on their ideas before sending them off to hour-long rendering processes.

Moving back to the video game world Jakobsen [11] proposes using a mass-spring system for modeling "skeletons" of in-game characters. He describes how the corpses within the game Hitman: Codename 47 consists of just particles and springs. These springs have varying degrees of stiffness to ensure that the body does not bend or rotate in unnatural ways. For example, a very stiff spring between the legs can ensure that legs never cross over. Additionally, Jakobsen uses angular springs to ensure that the body does not move in ways that the human body would not allow. This approach allows for intuitive physical responses as well, an example given by Jakobsen is that if a character is hit in the shoulder reacting to that hit can be done simply by moving the shoulder particle based on the force from the hit. Moving the particle will impact the other springs attached to it, which will drag the shoulder and its connected particles closer to each other, leading to realistic looking physical behavior.

5 Future

Predicting the future is always a difficult challenge, especially in fields such as technology and graphics. However, I would say that it seems like we have settled down on some of the classic conceptual techniques such as spring mass systems, finite element methods and meshless deformations [15], at least for interactive applications such as video games.

I base this on how the mass spring model, although quite old, is still brought up in many of the slides I have seen on cloth simulation from different universities; such as Trinity College in Dublin[16], and its continuous use in the games industry. For example the clothing in Alan Wake[1] was done through a mass spring model[10], and from my impression this is also the case in Assassin's Creed Unity[2] and Far Cry 4[3][17]. This might change in the future though as other models are proposed. Weidner et al. [18] proposes a way to deal with the situation where a piece of cloth collides with a straight edges that are not exactly on the vertices leading to sub-realistic results. However, it is not entirely clear to me how this implementation actually works and whether or not it is for real-time applications such as games, or would require too much processing power. If that is the case, maybe it will be possible in the future.

While other methods do exist I do not think we will move away from mass spring systems very soon, their relative easy of implementation, believably, and relatively good performance still make them attractive. It is important to remember that it is not always about having the newest shiniest tool, but rather to use the tool that best fit the situation at hand.

[1]http://www.alanwake.com/
[2]https://www.ubisoft.com/en-us/game/assassins-creed-unity/
[3]https://www.ubisoft.com/en-us/game/far-cry-4/

6 Resources

As mentioned in the introduction getting into this field can be quite intimidating, while there are numerous papers on the topics they are often quite heavy on the math side, at the same time, I find that there is a lack of good tutorials, and the good ones are really hidden away. The assumption seems to be that if you are interested in implementing this, you already know enough math and physics to understand the papers. A lot of this stuff is explained through mathematical models which aren't necessarily the most intuitive, such as those found in Provot's paper[5] on calculating the internal force of tensions between springs. Now that I have implemented a mass-spring cloth model I understand more of what they are trying to say here, but initially, most of that stuff was just scary.

Similarly, a lot of important aspects simply were not mentioned in a lot of the different papers and tutorials, like how it is preferable that the timestep in the integration methods is constant. This is probably obvious to a lot of the people working in this field, but to newcomers, it might not be, and it is something I randomly figured out myself (only to later get it confirmed in a tutorial).

It can also be very confusing to get into the field if you do not know exactly what you are looking for. I did not know that modeling of objects for animations etc, also could be called deformation. Now it makes a lot of sense, but when I was initially looking for deformation stuff I was looking at it from a more visual effect viewpoint. This made it really hard to discern what papers on deformation were relevant and not.

6.1 Tutorials

Luckily some really good tutorials and resources exist for this stuff, however, they can be a bit hard to find. Below are some that helped me with the implementations, or that I found after completion that probably would have helped me.

Jasper Flick

Jasper Flick's tutorials on Catlike Coding[1] are a really good starting point. His mesh deformation tutorial gives a good introduction to mass-spring systems and deals with the problems of scale and dampening. Additionally, I would advise anyone who is getting into mesh manipulation to go through his procedural meshes tutorial, it allows you to brush up on how meshes actually work and got me into the mindset of mesh manipulation. Jasper Flick also has several other great tutorials on manipulating meshes for visual effects, such as creating good-looking waves[2], which is something I will give a shot when I get the time.

[1] `https://catlikecoding.com/unity/tutorials/`
[2] `https://catlikecoding.com/unity/tutorials/flow/waves/`

Jeff Lander

Jeff Lander's article was also a good "tutorial", however, I struggled when trying to adapt the code from his article[12] to Unity, I ended up with a model that exploded really quickly, which I believe might be due to the semi-implicit Euler integration and that I was not using fixed time steps, as I did not know of it at the time. I would like to go back to that implementation in the future to give it another spin. Unfortunately, the links to his code on Gamasutra are dead, however, the code is available online[3].

Jesper Mosegaard

Jesper Mosegaards tutorial[7] was the one that helped me the most in understanding how cloth simulation through mass-spring systems worked. He highlights the importance of the Verlet integration and also explains it in a quite intuitive manner, additionally his code is relatively easy to follow, however, I wish he mentioned in his article that the timestep had to be a constant because that was something I figured out after I followed his implementation but got inconsistent results.

Thomas Jakobsen

Reading Thomas Jakobsen's[11] article is also recommended, as it goes deeper into Verlet integration and handling constraints, and other places where mass-spring systems can be used. However, this one gets a bit more advanced.

Henrik Enqvist

The article by Henrik Enqvist[10] on cloth in Alan Wake is something I came upon just recently, but I wish I had found it sooner. It gives a good overview of mass-spring systems and Verlet integration, and the one who confirmed to me that fixed timestep was actually needed. It also comes up with solutions to some of the problems I have in my own model, such as too much stretching and rubbery feel.

Math as Code

While not a tutorial Math as code[4] is a great resource to have when approaching math heavy topics like this from a programmers perspective. Essentially it covers a lot of common mathematical symbols and expresses them in Javascript. Having this resource gave me a lot of confidence as it allowed me to take complicated looking math and move it to a domain I was more familiar with. For instance, it was when I used math as code to "translate" the net force calculation shown by Matthew Fisher[13] that I got the courage to give implementing cloth physics a shot.

6.2 Implementations

The code for this project is on GitHub[5], under the folder named Unity.

[3] `http://www.darwin3d.com/gamedev.htm`
[4] `https://github.com/Jam3/math-as-code`
[5] `https://github.com/Per-Morten/imt4888_advanced_tech`

7 Reflection

Working on this project has been an enlightening experience, but it has not been without issues. Originally I wanted to look at shaders in Unity to brush up on my shader knowledge, with the final goal being to end up in the area of terrain deformation. I spent the beginning of the project toying with Unity's shader graph[1], creating different visual effects[2] to verify that my assumptions of how shaders actually worked were correct.

After getting my feet wet in shader graph I moved over to writing the shaders by hand while looking for tutorials on terrain deformation. It was hard to find good tutorials on this, especially on the GPU. However, I did stumble across a seemingly good tutorial from Jasper Flick[3] on surface displacement which seemed quite close to what I wanted to do. This tutorial assumed that you were already quite familiar with rendering concepts and built on his previous tutorials, so I started going through those. While working through these tutorials I felt that Unity's shaders were too different from those I had written before in OpenGL, the learning curve was a bit too steep, debugging tools were quite lacking, and that I was not making good progress towards the end goal.

I looked around for alternative solutions. Writing shaders in Unreal was a possibility, but that requires building the entire engine from source when you add your own shading models[19], and the compilation times would quickly hurt my iteration process. Another alternative was to implement my own rendering program in C++, but that would have taken too much time.

Due to this I decided to move back to the CPU, and work on more general mesh deformation there. The original intention was to port the logic over to vertex shaders after I had implemented them on the CPU. I soon moved away from that idea. Implementing stuff on the GPU probably would be more of a chore than a learning experience, as I already have dealt with data layout and CPU-GPU communication in other courses. The challenge would be the logic of mesh deformation, not how to do it in shaders.

With the shift over to the CPU, I started to work on procedural mesh generation, as all the different tutorials I looked at on mesh deformation was working on procedurally generated meshes. Working through the procedural mesh generation tutorials of Jasper Flick was quite intuitive and gave results, and in the end, I finished his mesh deformation tutorial. I played around with the implementation for a while, without a clear goal of where to next.

Looking through papers on the topic of mesh deformation most of it seemed to focus on deformation for manipulating meshes more related to creating animations, which was not really what I was after. It was first when I learned that the mesh deformation that Jasper Flick[6] presented was

[1] https://blogs.unity3d.com/2018/02/27/introduction-to-shader-graph-build-your-shaders-with-a-visual-editor/
[2] Video of some of two of the visual effects in shader graph: https://youtu.be/1DBmfpjmMds
[3] https://catlikecoding.com/unity/tutorials/advanced-rendering/surface-displacement/

a sort of mass-spring system that I figured out that deformation through mass-spring systems would be what I wanted to focus on, and its use in cloth simulation was quite enticing. Unfortunately, this was quite late in the course, so I did not get to spend as much time on the background of the technology, alternatives and the future of the technology as I originally wanted. I would have loved to get more into the other approaches like the finite element methods or free form deformations or try the mass-spring system within other contexts or with other integrators, moving more into the area of physics simulations. Figuring out the topic so late also meant that I ran into the problem that you do not really know what to search for until you have been in an area for a while, which meant that looking around for more background papers and tutorials was quite difficult.

However, while the path I took throughout this project was full of twists and turns I do believe I have come out of it with a greater and wider understanding of shaders & rendering, mesh manipulation, and physically based deformation.

The code from the early projects are located in the "projects_for_reflection" folder on the GitHub[4] repository.

[4]https://github.com/Per-Morten/imt4888_advanced_tech

Bibliography

[1] Hauser, K. K., Shen, C., & O'Brien, J. F. 2003. Interactive deformation using modal analysis with constraints. In *Graphics Interface*, volume 3, 16–17.

[2] Müller, M., Dorsey, J., McMillan, L., Jagnow, R., & Cutler, B. 2002. Stable real-time deformations. In *Proceedings of the 2002 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, SCA '02, 49–54, New York, NY, USA. ACM. URL: http://doi.acm.org/10.1145/545261.545269, doi:10.1145/545261.545269.

[3] Rodrigues, T., Pires, R., & Dias, J. M. S. 2005. D4md: deformation system for a vehicle simulation game. In *Proceedings of the 2005 ACM SIGCHI International Conference on Advances in computer entertainment technology*, 330–333. ACM.

[4] Coquillart, S. September 1990. Extended free-form deformation: A sculpturing tool for 3d geometric modeling. *SIGGRAPH Comput. Graph.*, 24(4), 187–196. URL: http://doi.acm.org/10.1145/97880.97900, doi:10.1145/97880.97900.

[5] Provot, X. et al. 1995. Deformation constraints in a mass-spring model to describe rigid cloth behaviour. In *Graphics interface*, 147–147. Canadian Information Processing Society.

[6] Flick, J. Mesh deformation. catlikecoding.com. Retrieved: 28.10.2018. URL: https://catlikecoding.com/unity/tutorials/mesh-deformation/.

[7] Mosegaard, J. Mosegaards cloth simulation coding tutorial. viscomp.alexandra.dk. Retrieved: 30.10.2018. URL: https://viscomp.alexandra.dk/?p=147.

[8] Melax, S. 2013. Math for game programmers: Interaction with 3d geometry. Presented at GDC(Game Developers Conference). Retrieved: 13.11.2018. URL: https://www.youtube.com/watch?v=GpsKrAipXm8.

[9] Fiedler, G. June 2004. Integration basics. gafferongames.com. Retrieved: 12.11.2018. URL: https://gafferongames.com/post/integration_basics/.

[10] Enqvist, H. April 2010. The secrets of cloth simulation in alan wake. Gamasutra.com. Retrieved: 15.11.2018. URL: http://www.gamasutra.com/view/feature/132771/the_secrets_of_cloth_simulation_in_.php.

[11] Jakobsen, T. 2001. Advanced character physics. In *Game developers conference*, volume 3. IO Interactive, Copenhagen Denmark.

[12] Lander, J. March 2000. Devil in the blue faceted dress: Real time cloth animation. www.gamasutra.com. Retrieved: 13.11.2018. URL: https://www.gamasutra.com/view/feature/131851/devil_in_the_blue_faceted_dress_.php?page=2.

[13] Fisher, M. Cloth. graphics.stanford.edu. Retrieved: 13.11.2018. URL: https://graphics.stanford.edu/~mdfisher/cloth.html.

[14] Nedel, L. P. & Thalmann, D. 1998. Real time muscle deformations using mass-spring systems. In *Computer Graphics International, 1998. Proceedings*, 156–165. IEEE.

[15] Müller, M., Heidelberger, B., Teschner, M., & Gross, M. 2005. Meshless deformations based on shape matching. In *ACM transactions on graphics (TOG)*, volume 24, 471–478. ACM.

[16] Dingliana, J. March 2016. mass-spring systems & cloth. www.scss.tcd.ie. Retrieved: 15.11.2018. URL: https://www.scss.tcd.ie/~manzkem/CS7057/cs7057-1516-14-MassSpringSystems-mm.pdf.

[17] Vaisse, A. 2015. Ubisoft cloth simulation: Performance postmortem and journey from c++ to compute shaders. Presented at GDC(Game Developers Conference). Retrieved: 15.11.2018. URL: https://www.gdcvault.com/play/1022350/Ubisoft-Cloth-Simulation-Performance-Postmortem.

[18] Weidner, N. J., Piddington, K., Levin, D. I. W., & Sueda, S. July 2018. Eulerian-on-lagrangian cloth simulation. *ACM Trans. Graph.*, 37(4), 50:1–50:11. URL: http://doi.acm.org/10.1145/3197517.3201281, doi:10.1145/3197517.3201281.

[19] Hoffman, M. December 2017. Unreal engine 4 rendering part 1: Introduction. medium.com. Retrieved: 16.11.2018. URL: https://medium.com/@lordned/unreal-engine-4-rendering-overview-part-1-c47f2da65346.