FABRIC SIMULATION ON THE GPU

Michaela Rabenius (micra753)

Wednesday 26th December, 2018
20:00

Abstract

Simulating fabric well can be a challenge due to the special properties fabric has. Fabric can be simulated relatively easy by modelling it as particles connected by springs. This report describes the theory behind and the implementation of a fabric simulation using the mass spring model. The calculations are done on the GPU, making use of the GPU's processing power to increase the performnce of the simulation. The final result shows that while the implementation has some faults, it manages to simulate the basic behaviour of falling/hanging fabric.

1 Introduction

Simulating fabrics in computer graphics is a problem with many different solutions, all varying in complexity and realism. Fabric has several interesting properties that can make it a challenge to animate, especially related to how it deforms: hanging fabric typically drapes as it is pulled down by gravity and dynamic fabric has folds that behave and propagate in a certain way as it is affected by external forces such as wind.

The dynamic properties especially can be a challenge to simulate correctly. Earlier works have found that physically-based models work rather well for reproducing the behaviour of several different cloth items, such as flags or clothes, one such model being the mass spring model which represents the fabric with a number of particles connected by springs.

Simulations with computer graphics can easily become very demanding. While complex simulations can be very costly on the CPU, it is possible to make use of the processors on the GPU to perform the costly calculations. GPUs today have vastly improved their performance compared to a couple of years ago and are able to process complicated computations quickly[1], making GPGPU programming a good choice for simulations.

This report describes the theory behind and the implementation of a fabric simulation in real time on the GPU based on the mass spring model. The implementation is done in C++ using OpenGL and GLSL. The more advanced computations are done on the GPU using several shader programs. The report also discusses limitations of the current implementation and future additions and improvements that can be made.

1.1 Aim

The aim of the project and report was to simulate fabric in real time based on the mass spring model using OpenGL and GLSL. The computations of physics were to be done on the GPU in order to gain understanding of how GPGPU programming can be used to accomplish non-graphical tasks in simulations.

1.2 Limitations

This report will only discuss the implementation of the mass spring model on the GPU, hence no comparison to any particular implementation on the CPU will be mentioned. The implementation will also only include basic movement behaviour of fabric and will thus lack several properties present in the real world counter part, such as self-collision and tearing. This will be discussed further in section 5.

2 Background

This section will briefly touch upon related works to fabric simulation as well as explain the basic theory behind the mass spring model used in the implementation.

2.1 Related work

There are several different methods for simulating fabric. Many of them are particle-based systems with some variations between them.

Rodriguez-Navarro and Susin implements a Finite Element Method(FEM) to simulate cloth on the GPU with collision detection[2]. It has the advantage that it can be used for general triangulated cloth meshes. They also found that their method has improved realism compared to other methods.

As explicit integration methods have a large risk of becoming unstable if the time step is too large, using implicit integration provides more stability at the cost of a more complex implementation. Volino and Magnenat Thalmann created a method for simulating fabric based on implicit integration with efficient collision detection[3]. While their method is stable, there are still situations where an explicit integration scheme is more accurate.

2.2 Mass spring model

Fabric can be simulated relatively easy using a mass spring system: a system consisting of particles with some mass connected by springs, see Figure 1.

The particles are arranged in a rectangular grid. Each particle is then connected to several of its neighbouring particles by springs. These springs generate force between the particles and provides constraints for how the fabric can deform. The

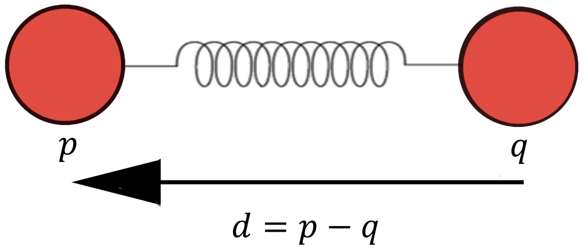

Figure 1: Two particles p and q connected by a spring. d is the distance vector from q to p.

spring forces will pull the particles closer if they are too far away from each other while also pushing them away from each other if they are too close. This way, the particles can be treated as one cohesive object (i.e. a piece of fabric).

To simulate how the fabric moves and deforms it is necessary to look at how each individual particle in the system moves as a result of both internal and external forces. The force on one particle can be derived from the second law of motion: $\mathbf{F} = m \cdot \mathbf{a}$. The force term $\mathbf{F}$ can be separated into internal forces, i.e. the force exerted between particles within the fabric, and external forces such as gravity or wind.

Since all particles are connected by springs, it is possible to derive the internal force on one particle from Hookes law, $\mathbf{F} = k \cdot \mathbf{x}$. As explained by Ragnemalm[1], the force between two particles p and q (see Figure 1) mapped in the direction of the spring can be computed with Equation 1.

$$\mathbf{f}_p = -k_s(|\mathbf{d}| - r)\frac{\mathbf{d}}{|\mathbf{d}|} \tag{1}$$

r is the resting distance between the two particles, $\mathbf{d}$ is the distance vector in the direction of the spring and $|\mathbf{d}|$ is the current distance between the particles. k_s is a spring constant that determines the stiffness of the spring. Both particles enact this force on each other in opposing directions. Equation 1 gives the force for an undampened spring, meaning that the force would not decrease with time as it would realistically. To achieve a more realistic behaviour damping must be added to the spring force. Damping is added by projecting the speed difference v between the particles onto the direction of the spring, which results in the following equation:

$$\mathbf{f}_p = -\left(k_s(|\mathbf{d}| - r) + k_d \frac{(\mathbf{d} \bullet \mathbf{v})}{|\mathbf{d}|} \right) \frac{\mathbf{d}}{|\mathbf{d}|} \qquad (2)$$

k_d is the damping coefficient that determines how much the system should be damped. Thus, two terms in Equation 2 gives the force and damping respectively and their contributions are determined by the constants k_s and k_d.

To compute the total internal force on one particle all forces from its neighbours must be summed up. For other types of mass spring systems, only springs between the closest neighbours may be necessary to achieve the desired behaviour. However, for fabric simulations three different springs are needed to reproduce the properties of a cloth. These three types of springs are also known as *structure springs*, *shearing springs* and *bend springs* [4][5][6].

The structure springs connects one particle to its four closest neighbours horizontally and vertically, shown in blue in Figure 2. As the name suggests, these springs define the structure of the system.

The structure springs are not enough as the particle grid will shear and collapse, as explained by Lander[6]. To prevent this, shearing springs are needed. The shearing springs connects one particle to its diagonal neighbours and prevents the fabric from shearing too much, shown in green in Figure 2. The structure springs together with the shearing springs will help the fabric maintain its shape.

The fabric must also have some constraint to stop it from folding in on itself. This achieved with bend springs that connects on particle with its horizontal and vertical neighbours one step away, shown in yellow in Figure 2. The bend springs constrains the bending of the fabric.

By computing Equation 2 for all 12 neighbours and summing them together the total internal force on the particle is obtained. External forces can also be added to the internal force to get the total force. From the total accumulated force, the acceleration of the particle can be computed using $\mathbf{a} = \mathbf{F}/m$.

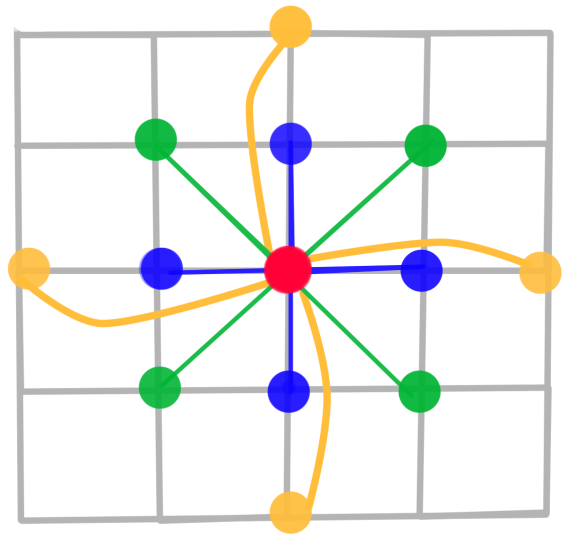

Figure 2: The structure of the mass spring fabric. The red particle is connected to its neighbours with structural springs(blue), shearing springs(green) and bending springs(yellow).

Once the acceleration has been computed, the speed and position of the particle can be updated using integration.

2.3 Integration methods

It is important to choose a good integration method as the mass spring system can easily become unstable, as discussed by Volino and Magnenat Thalmann[3]. The simplest integration method is Euler integration which can be used to update the position by using the old position p_i and the velocity v_i of the particle as well as a small time step dt:

$$\begin{aligned} p_{i+1} &= p_i + v_i \cdot dt \\ v_{i+1} &= v_i + a \cdot dt \end{aligned} \qquad (3)$$

However, Euler is not a very good approximation. Since the fabric consists of many different spring forces, using Euler can easily cause the system to become unstable. In this case it is more appropriate to use other integration methods, such as Verlet integration. Verlet integration can be used to compute the new position using the current position as well as the previous position along with the acceleration:

$$p_{i+1} = 2p_i - p_{i-1} + a \cdot dt \tag{4}$$

2.4 Rendering to textures with framebuffer objects

To compute the physics update on the GPU, data must be passed to a shader program. This can be achieved by rendering to textures using framebuffer objects (FBOs) as explained by Ragnemalm[1]. By rendering to textures it is possible to store the data of each individual particle in a texture, such as the position or velocity. Each pixel will then contain the data for one particle in the fabric. By ping-ponging between textures it is possible to compute the new position based on the old one. The data can be accessed from one texture in the shader using appropriate texture coordinates to perform a texture access. The data texture can the be used to update the actual vertices in the fabric to perform the simulation.

3 Implementation

This section will the describe the implementation of the mass spring model described in section 2.2. The final structure of the implementation consists of a `Fabric` class that generates the data needed to both render the fabric and generate textures used for simulation, and a few shader programs to update the positions of the vertices in the fabric. A class and shaders to draw a sphere and as well as a camera class were also added.

3.1 Libraries

The project was implemented using C++ and the graphics library OpenGL as well as the shading language GLSL to render the fabric. GLFW and GLAD were used set up the project and for window creation and other window related operations. The

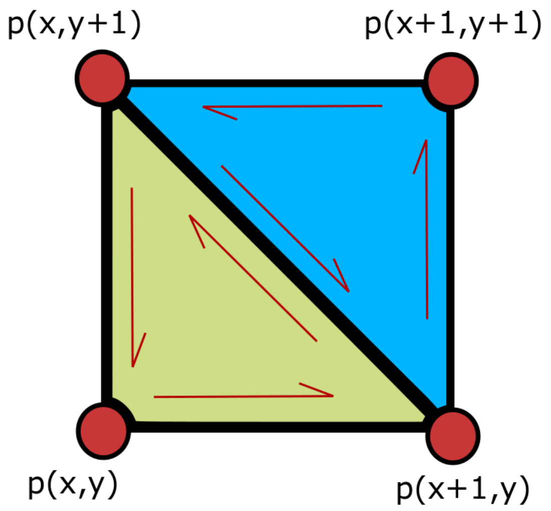

Figure 3: Two triangles drawn between four particles and their orientation.

OpenGL mathematics library GLM was used for mathematics such as vector- and matrix operations. A part of the GL_utilities library by Ingemar Ragnemalm used in a lab series in the course *TSBK03 - advanced game programming* was used to simplify using framebuffer objects and rendering to textures.

3.2 The `Fabric` class

The `Fabric` class takes 4 input values: the width and height of the fabric, as well as the number of particles along the width and height respectively. This makes it possible to adjust the size and resolution of the fabric grid. The `Fabric` class takes these inputs and generates lists containing the vertices(position and texture coordinates) and the drawing index order for each triangle in the fabric. These list are used by OpenGL to draw triangles between vertices that form a rectangular grid with counter-clockwise rotation, see Figure 3.

The `Fabric` class also stores an array containing the position of each vertex represented by a 4-dimensional vector. While only the first 3 coordinates are necessary to represent a position in 3D space, the fourth coordinate is used to differentiate between moving and static particles by giving it the value 0 or 1, storing it in the texture's alpha-channel. Particles with the value 1 are pinned points in the fabric and will thus not move from their original position while points with value 0 will be able to be moved in the shader. This is used to pin parts of the fabric up, such as the sides or corners. This array is then used to generated textures containing the position data of all particles in the fabric. Data for a striped texture is also generated in the `Fabric` class.

3.3 Shaders

To simulate the fabric a number of different shaders were created. This step of the implementation was done in two different variations: one initial variation using Euler integration to start up the project and for testing the algorithm, and one variation used in the final implementation which uses Verlet integration. Both variations has a *fabric shader* which takes the updated position texture as input and uses it to move the vertices in the fabric.

The variant using Euler integration was implemented based on a ping-ponging scheme described by Ragnemalm[1], see Figure 4. This scheme requires two textures containing the positions of the particles and two containing the velocities associated with each particle. Thus, two different shaders were needed to update the positions and velocities respectively: a *position shader* and a *velocity shader*.

The internal forces on the current particle was calculated in the velocity shader by computing Equation 2 for each of the 12 neighbours as described in section 2.2. To access the neighbours in the textures, texture offsets were calculated based on the grid size of the fabric and passed on to the shader.

Since the resting distance varies depending on which neighbour force is currently calculated, different values for r in Equation 2 are needed. These were passed in as a uniform `vec3` to the shader, containing the shortest resting distance along the width and height, as well resting distance between diagonal particles. The resting distance for the bend springs was computed in the shader as double the resting distance for the structure springs.

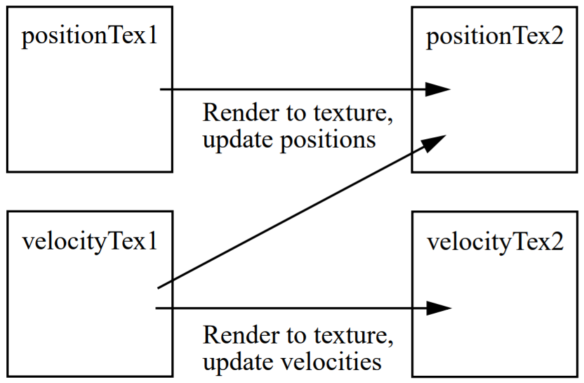
positionTex1
positionTex2
Render to texture, update positions
velocityTex1
velocityTex2
Render to texture, update velocities

Figure 4: The updating scheme by rendering to textures for particle systems described by Ragnemalm[1].

As previously mentioned, the simulation makes use of the alpha-channel of the position texture to store information about static particles. Thus, if the alpha channel of particle p contains the value 1, its position will not change.

The resulting force of summing the force from all springs, force from gravity as well as a default damping force on the momentum of the particle was then used to compute the acceleration of the particle using $\mathbf{a} = \mathbf{F}/m$. The velocity could then be updated using Euler integration (Equation 3). The updated velocity texture and the old position texture were sent to the position shader as input to update the position texture using Euler integration.

As explained in section 2.3, Euler integration does not work well in the case of fabric simulation as the system easily becomes unstable, causing the fabric to explode. Thus, the algorithm was adapted for Verlet integration in the final implementation.

Since velocity is not directly needed for Verlet integration, the velocity textures and velocity shader could be completely removed from the implementation. Instead, only three position textures were needed: one to store the old positions, the current positions as well as the new positions respectively. The algorithm for computing the forces was moved to the position shader. The final position shader

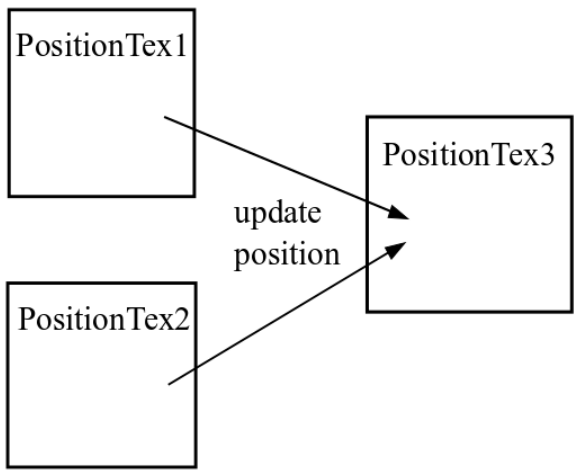

Figure 5: The final updating scheme by rendering to textures using Verlet integration.

takes the old and current position textures as input and uses them to compute the internal forces of the fabric as well as adding a gravitational force directed down, see Figure 5.

The resulting position texture is then sent to the fabric shader that uses the data to update the actual positions of the vertices in the fabric.

3.3.1 Collision detection

Very basic collision detection against a sphere with known, fixed position and size was also implemented. With the centre point and the radius of the sphere, the position shader can prevent the fabric from going through it. This is done by for each particle compute the distance to the centre of the sphere and check if the distance is smaller than or equal to the radius. If it is smaller than or equal to the radius it means the fabric is inside the sphere and that there is a collision. If this is the case, the particle is moved back to its previous position outside of the sphere.

Even if this approach stops the particles themselves from entering the sphere, it does not necessarily stop the triangles drawn between the particles from clipping through the sphere as the fabric stretches. This was solved by adding a small fraction to the radius of the sphere, making the fabric collide with a slightly bigger sphere. While this is technically cheating, the visual results still make it look like the fabric collides with the original sphere, only without clipping.

4 Results

This section will showcase the results of the final implementation of the fabric simulation. The following figures(6 - 11) shows a fabric made up of 50X50 particles.

A fabric with lower particle resolution (20X20) can be seen in Figure 12.

4.1 Controls

Some basic interactive controls were also added to the simulation, mainly related to camera movement. The camera can be moved using both mouse and keys(WASD-keys and arrow-keys). Wireframe-mode can be toggled on and of using the T-key. The sphere can be hidden using the H-key. The fabric can be reset to its starting position using the R-key.

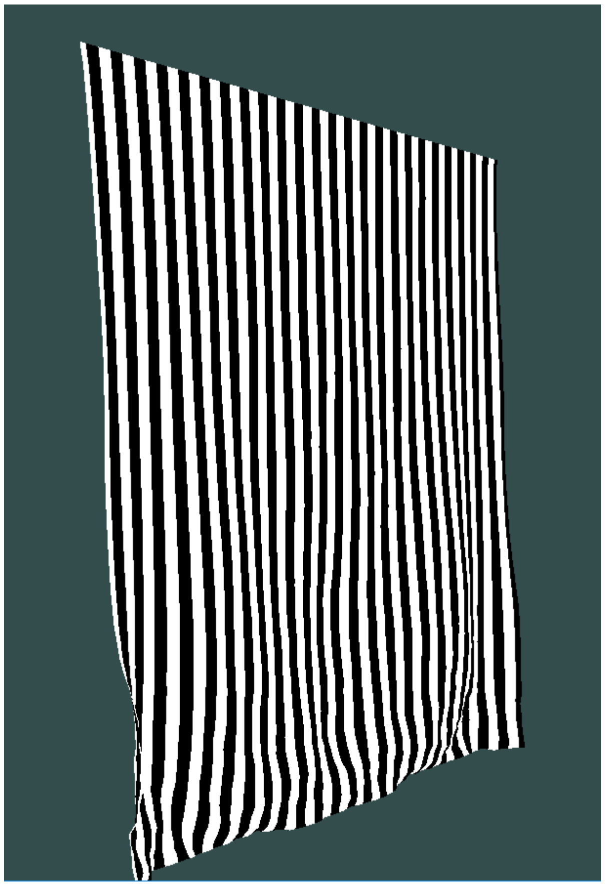

Figure 6: Free hanging fabric pinned along the upper edge.

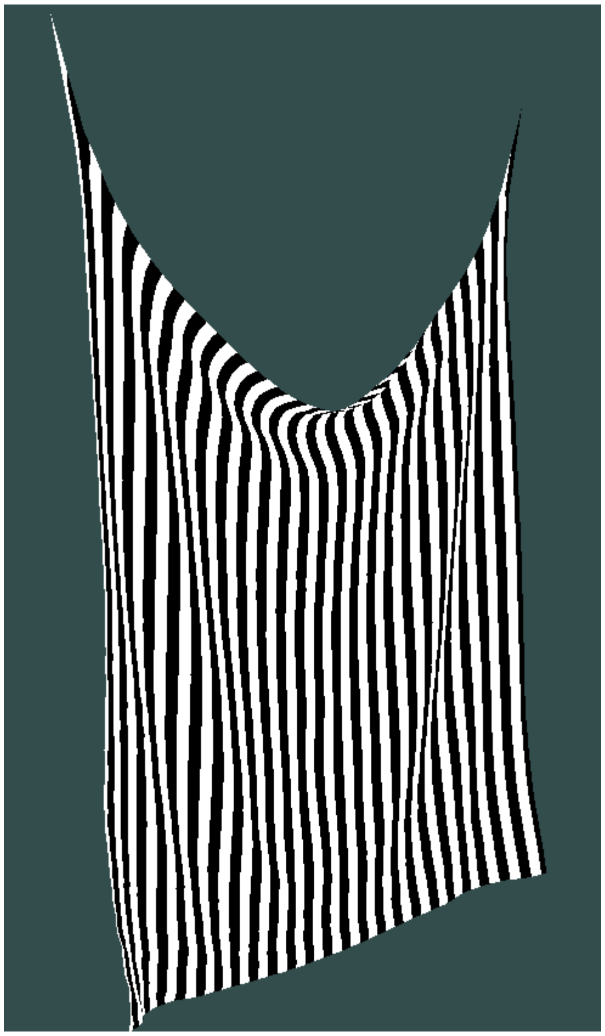

Figure 7: Free hanging fabric pinned in the upper corners.

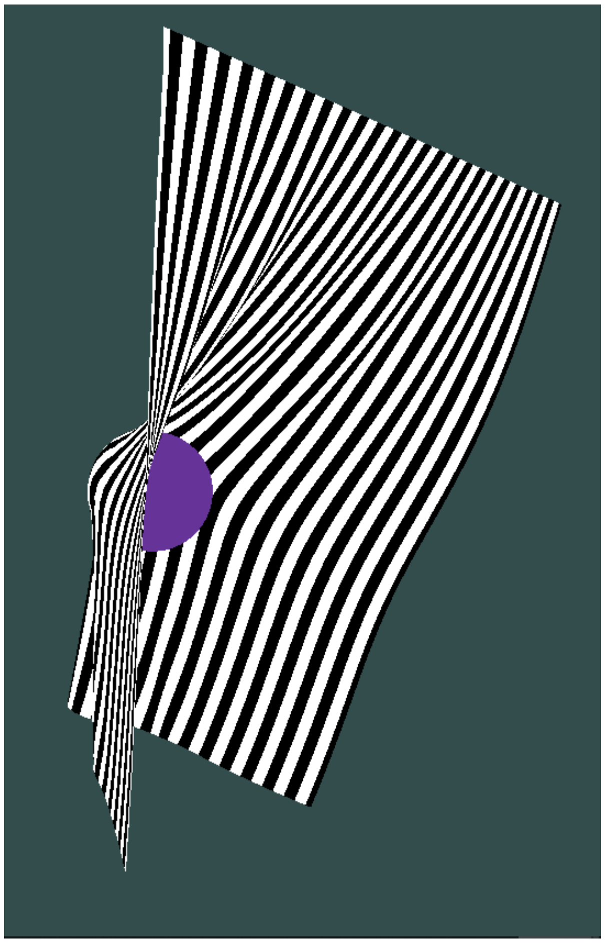

Figure 8: Free hanging fabric colliding with a sphere.

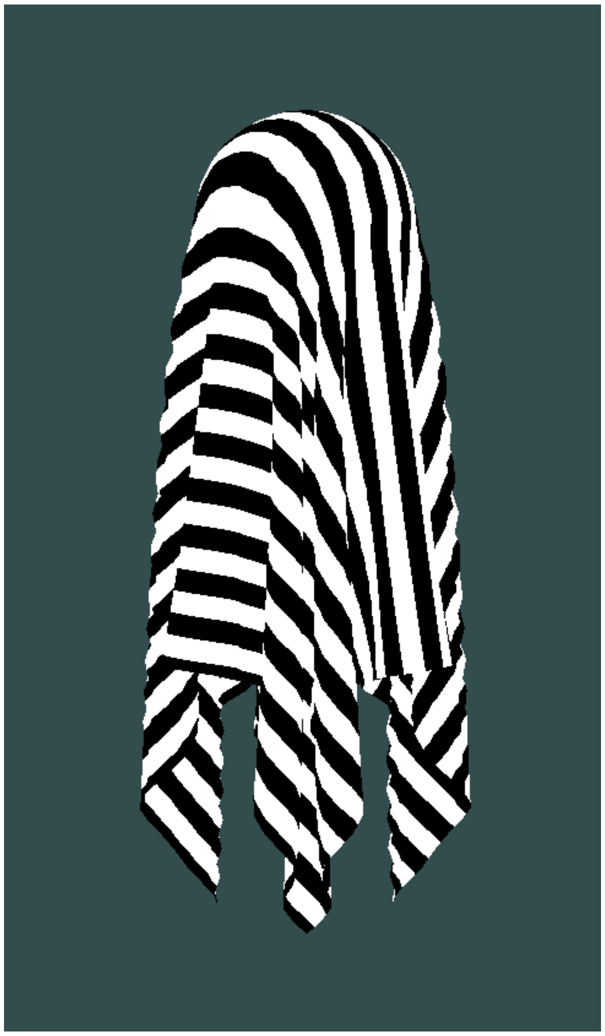

Figure 9: Free falling fabric colliding with a sphere.

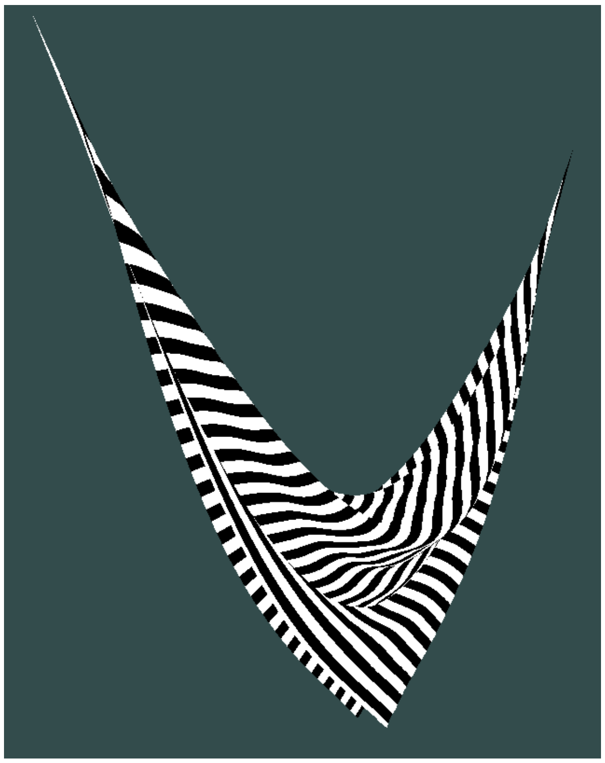

Figure 10: Free hanging fabric pinned in diagonal corners colliding with a sphere. The lack of self-collision can clearly be seen

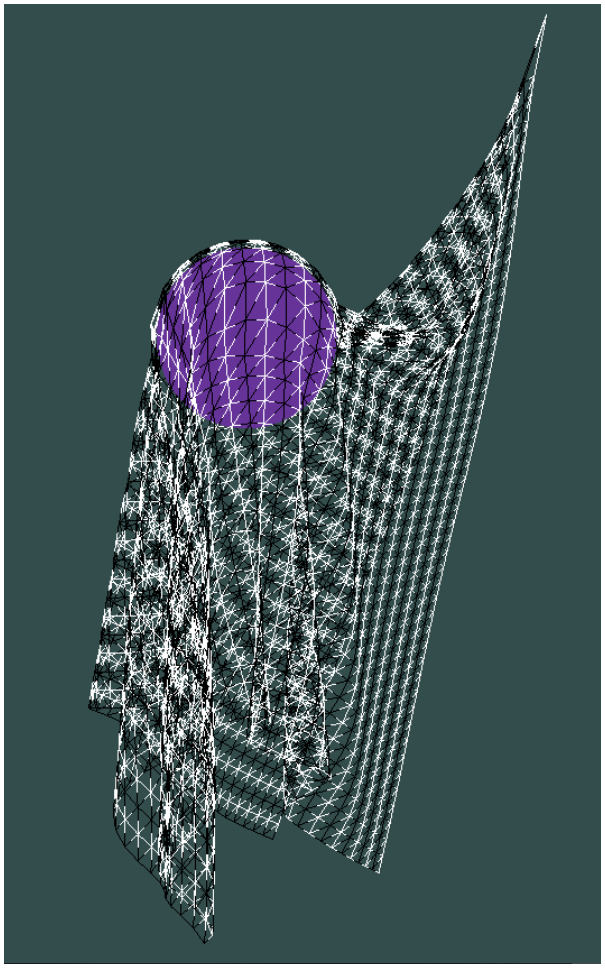

Figure 11: Fabric with wireframe mode on.

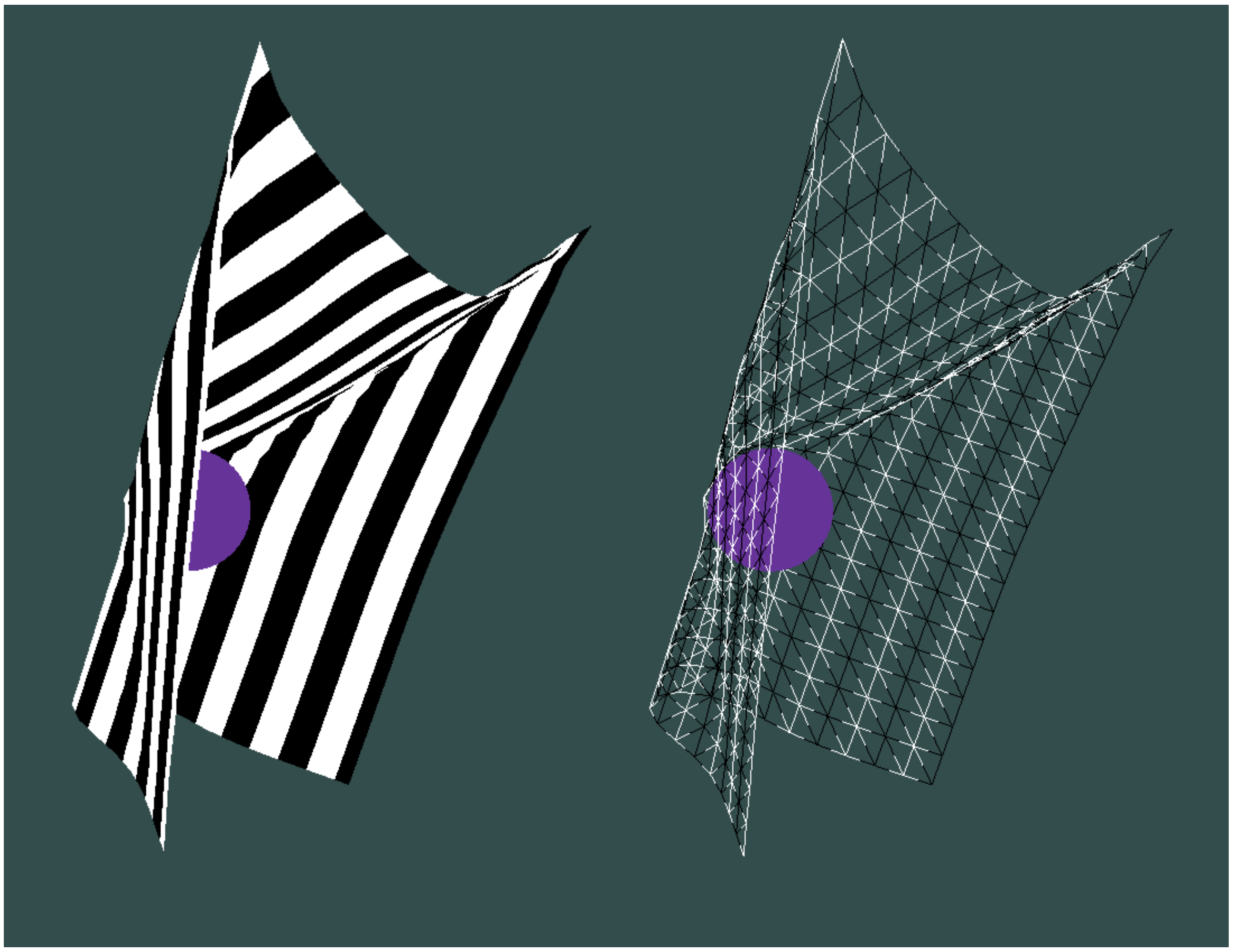

Figure 12: Fabric with fewer particles.

5 Discussion and future work

The final implementation succeeds in simulating the basics of falling/hanging fabric behaviour. The fabric folds naturally at the ends while hanging (Figure 6), and the upper edge is pulled downwards if the fabric is suspended only by the corners(Figure 7). The simple collision detection against a fixed sphere also works, as can be seen in Figure 8. The fabric has multiple realistic looking folds when hanging over the sphere in Figure 9.

Since the fabric is composed by springs, the fabric is slightly elastic. While this is not how fabric works in reality (fabric is generally not elastic) it is still possible to approximate the behaviour with satisfactory results. However, this means that the fabric may stretch under high stress. Stretching is especially prevalent near pinned parts of the fabric, as explained by Provot[4]. Stretching is also very noticeable when the number of particles increase, possible due to the increasing total mass of the fabric. To solve this issue in the future, a limit to how much a spring is allowed to stretch could be added.

The simulation has some difficulties deforming the fabric in the centre of its folds, leading the fabric to looking dented instead of straight. This issue can be made less noticeable by increasing the particle resolution of the fabric.

The fabric is currently only affected by gravity. It would be interesting to add other external forces, such as wind, to the simulation in the future.

While the fabric successfully collides with the sphere, it cannot collide with itself. This becomes especially apparent when pinning two diagonal corners of the fabric, as can be seen in Figure 10. As the gravity pulls the fabric downwards, the corners that are not pinned will fall in an arch towards the middle, making them cross and intersect. In the future, self-collision and collision with other types of objects should be added to the simulation for a more realistic result.

With 50X50 particles the simulation runs smoothly. The number of particles can be increased to 4 times the amount without any larger noticeable slow down, showcasing the advantage of performing the calculations on the GPU. Theoretically the number particles should be able to be even larger, but due to a bug the system appears to become unstable at 106X106 particles and the fabric disappears. This bug could potentially be caused by parameter values, integration method or insufficient constraints on the simulation. Unfortunately, this bug makes it impossible to test the performance limit of the simulation at time of writing. In a continuation of this project, this bug should be analyzed and solved and the performance limit should be tested.

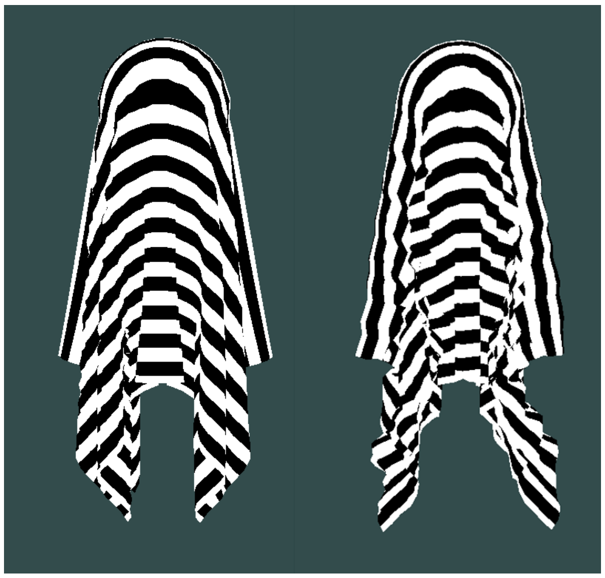
Figure 13: Left: fabric with 50X50 particles. Right: fabric with 51X51 particles. The error in the fabric on the right can clearly be seen.

The simulation also has some difficulties with an odd number of particles along the sides. To access neighbour particles in the position textures, neighbour texture coordinates are computed using offset values along the x- and y-axis. This works best for a power of 2 number, as it will result in texture coordinates at exactly the centre of the neighbour pixel. However, for an odd number of particles, the calculated texture coordinates will have a small error, not being in the exact centre of the pixel. This will lead to errors in the simulation, in the worst case having big consequences on how the fabric deforms, see

Figure 13. In the future it would be worthwhile to use a different strategy to access neighbour values and hopefully minimize the error.

6 Conclusion

In conclusion, while there are some errors, the simulation manages to animate the basic behaviour of falling/hanging fabric with simple collision detection against a sphere. The implementation successfully runs on the GPU, but due to a bug in the code, the number of particles that make up the fabric is limited. In the future it would be interesting to test the performance limit of the simulation. For a more realistic behaviour, other properties should be added such as self-collisions, and also add some kind of limit on how much the fabric can stretch.

References

[1] I. Ragnemalm, *Polygons Feel No Pain So How Can We Make Them Scream?*, Vol. 2, 2017 edition.

[2] J. Rodriguez-Navarro and T. Susin, *Non structured meshes for Cloth GPU simulation using FEM*, Journal, 2006.

[3] P. Volino and N. Magnenat Thalmann, *Implementing Fast Cloth Simulation with Collision Response*, Proceedings of the International Conference on Computer Graphics, 2000.

[4] X. Provot, *Deformation Constraints in a Mass-Spring Model to Describe Rigid Cloth Behaviour*, Proceedings of Graphics Interface '95, 1995.

[5] C. OConnor and K. Stevens, *Modeling Cloth Using Mass Spring Systems*, 2003.

[6] J. Lander, *Devil in the Blue Faceted Dress: Real-time Cloth Animation*, `https://www.gamasutra.com/view/feature/131851/devil_in_the_blue_faceted_dress_.php`, accessed: 2018-11-12.