NTNU

Innovation and Creativity

Cloth Modelling on the GPU

Kjartan Dencker

Master of Science in Computer Science
Submission date: June 2006
Supervisor: Torbjørn Hallgren, IDI

Norwegian University of Science and Technology
Department of Computer and Information Science

Problem Description

This project explores the possibility to use general purpose programming on the GPU to simlate clothes in 3D. The goal is to implement a faster version of the method given in 'Large Steps in Cloth Modelling' by Baraff et. al. (Implicit Euler).

Assignment given: 19. January 2006
Supervisor: Torbjørn Hallgren, IDI

1 Introduction

Simulating clothes is one of the most important finishing touches of a 3D application, be it a movie or a game. Cloth simulation is also a CPU-intensive task and therefore many games choose not to simulate them. Without moving clothes in an animated movie it would not be as realistic and visually pleasing as it could have been if cloth simulators were used in the construction of the movie. Similarly one could improve on the realism in games as well.

1.1 Motivation

I am studying an implementation of a cloth simulator on the Graphical Processing Unit (GPU), because I want to find out how a GPU implementation would perform compared to the CPU implementation, in order to understand how the GPU compares to the CPU when it comes to a complete general purpose program.

The implicit particle system implementation constructed by Baraff et. al. [2] is a well known cloth simulator in the cloth simulation community, but this method is not a very fast method and real-time performance is only achieved for smaller resolutions of particles. It would be useful to investigate the possibility of making this method perform faster, and for this purpose general purpose programming on the GPU seems appropriate.

Many researchers have found that GPU programs perform 10-20 times faster than the same program written for the CPU. In this rapport we will look at whether this applies for a cloth simulator. The speedup achieved when using a GPU come from the fact that GPU's are highly parallel, and thus can perform many arithmetic operations at the same time. The GPU consist of vertex- and fragment-processors, GeForce 7800 GTX has 24 pixel-pipelines, and 8 vertex-pipelines. A GPU is not intentionally designed to execute general purpose programs, but different techniques have been found that allows us to utilize the massive power of the GPU to create stunning visual effects.

1.2 Objectives and Requirements

This report will guide you through the process of constructing both a CPU and GPU implementation of the method described in [2], to see if we can implement a GPU version that can achieve, if not real-time, then interactive simulations of larger particle systems. Interactive simulations have a frame-rate of 30+ fps and real-time simulations have a frame-rate equivalent with the time-step used in the simulations. If dt=0.01s, then we need 100fps to achieve real-time performance. With interactive simulations, we mean that the user can easily rotate and look at the cloth while it is being simulated, and this is achieved with preferably 30+ fps.

1.3 Approach

This project consists of four stages, information retrieval, understanding, implementing and experimenting/testing.

The first thing to do is to acquire information of the research in the field of interest, namely cloth modelling and general purpose programming on the GPU. This is going to be challenging since cloth modelling is an area where there has been done much research since the mid-1980s [2]. But it helps that I had undertaken a project last autumn that involved three different cloth simulation techniques. The challenge in this project is to understand the underlying nature of the GPU and to find the best way to implement the cloth simulator.

When general purpose GPU is understood, implementation is going to be attempted and the testing and experimenting is conducted. These steps are not linear in time, since problems are going to occur in later stages and more research is going to be needed.

1.4 Structure

Chapter 1 Introduction
This chapter will give an introduction to what is going to occur in this project.

Chapter 2 Background
We will look at clothes in general, as well as the physics required to understand what is happening within the cloth. We will also look at the GPU to understand how it works, and some optimizing techniques for the GPU.

Chapter 3 Cloth Modelling
In this chapter we will look more closely at how we can integrate the differential equation that we end up with when we simulate clothes using a particle system. We will also look at some different methods to solve linear sets of equations and look at what is the state of the art for cloth modelling techniques today.

Chapter 4 Conceptual Design
In this chapter we will look at the implicit particle system from [2], and how we can apply the forces to the particle system to make it act like a cloth. We will also look closer at two different numerical solvers for linear sets of equations.

Chapter 5 Implementation
In this chapter we will look at how the different versions of the cloth simulator were implemented. The implementation of both the CPU version and the GPU version with the Conjugate Gradient method and the Jacobi method will be explained, and we will look at how we implemented the different data-structures.

Chapter 6 Results, Conclusion and Future Work
In this chapter we will look at the results we got from the different methods, both visual and performance results will be given. We will also look at what could be done to improve the GPU version of the cloth simulator to achieve a more realistic, larger and faster cloth simulator on the GPU.

1.5 Summary

Clothes are an important factor of visual realism, both for movies and for games. The goal of this project is to construct a faster version of the cloth simulator given in [2] by implementing the method on the GPU. We hope to achieve interactive simulations for large particle systems.

2 Background

2.1 Clothes

We will start of now by looking at real-life clothes, if you look at a draping, you might think that it is a solid surface, but this is not the case. When you look closer you will see that clothes are in fact knitted yarn or woven threads [20]. We often look at clothes in real life and do not realize the complex interaction of the yarn or threads amongst each other, but as you can see in figure 2.1, the complex structure of the cloth reveals itself. As you now might imagine, physically correct simulation of this is complex and non-trivial. This comes from the fact that there are so many varieties of fabrics that can be used in the production of clothes, and they have different physical behaviour such as friction between the fibres, how much they stretch, bend and so on [3].

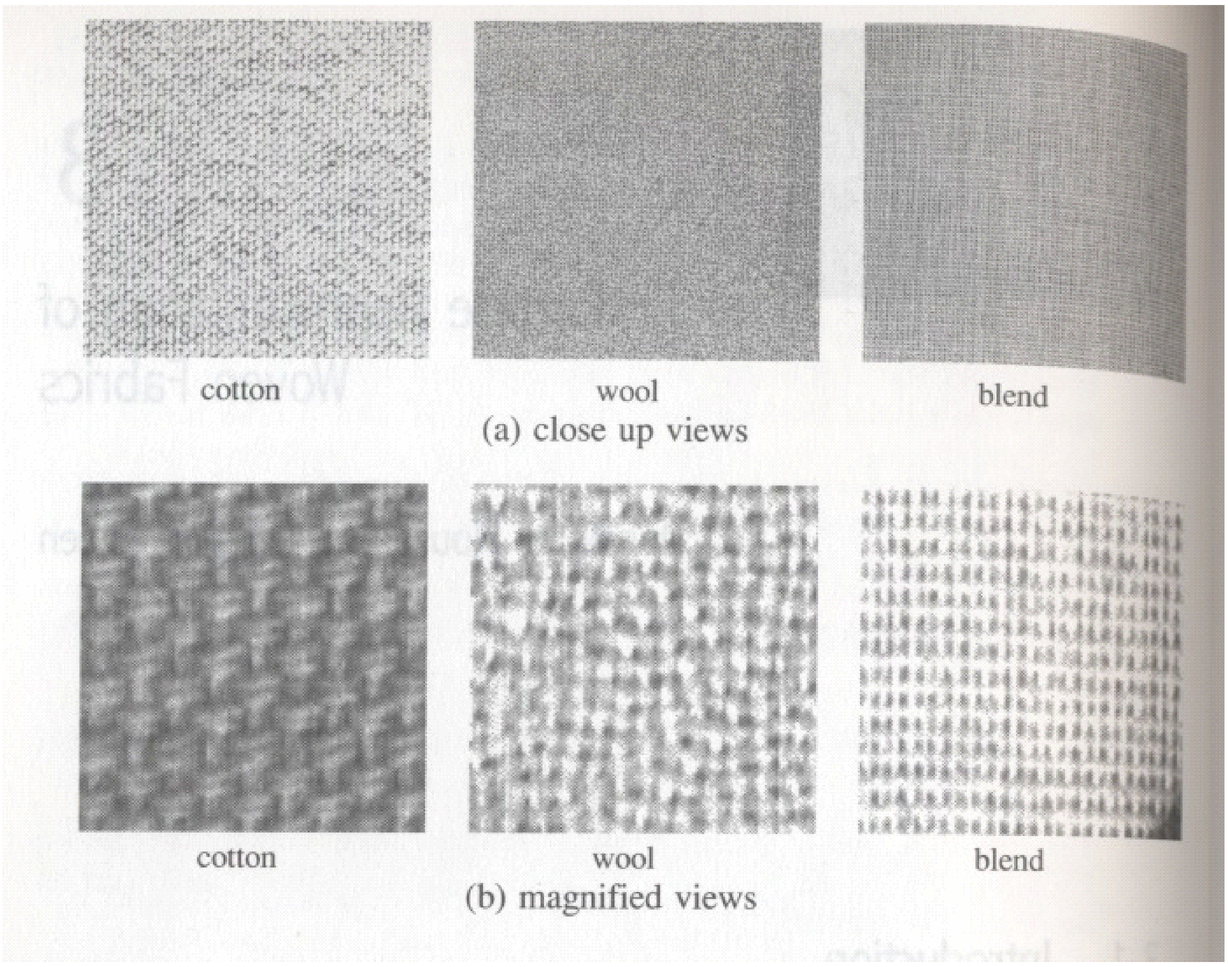


Figure 2.1 – This figure shows different fabrics both close up and magnified view [3].

There are two different ways to make clothes, knitting or weaving, both methods are based on interleaving threads to give the cloth a shape, like a sweater. Each type of cloth has its own characteristics based on the type of thread that has been used and the method that this thread has been woven/knitted. This is why cotton-sweater is different from a wool-sweater, and it is a fact that we normally don't think twice about, but when we attempt to simulate a cloth on the computer we need to look at how a cloth should behave and how we can make it do so.

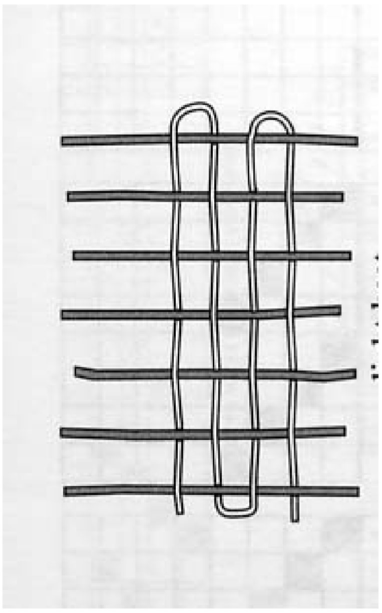


Figure 2.2 - As this picture illustrates, clothes are just interleaved threads.

2.1.1 The Kawabata System

There has been performed many formal studies of the elastic properties of clothes [27], though the Kawabata experiments [30] are the most used as a basis for cloth modelling techniques.

The Kawabata experiments consist of 5 tests that represent the most important properties of the clothes. Tensile (stretch), bend, shear, compression and friction. These tests are an indication of which forces the cloth simulation depends on to behave naturally.

With the Kawabata system we can obtain 21 different parameters to express the physical properties of a certain type of cloth.

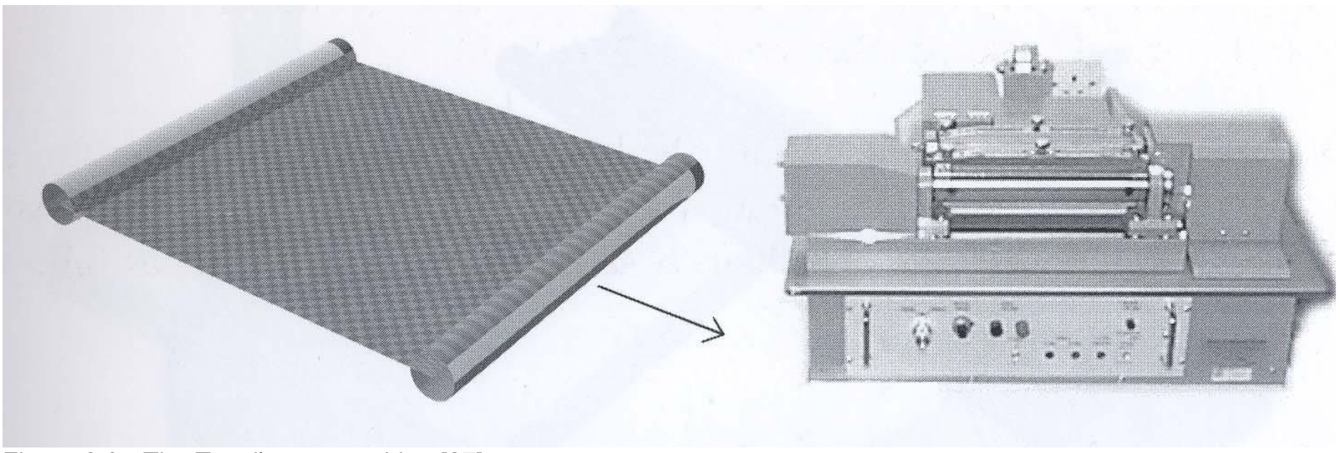


Figure 2.3 - The Tensile test machine [27].

The Tensile test extends a piece of cloth at a constant speed with two opposite edges fixed, when the deformation limit is reached, the process is reversed at the same speed and we get 7 parameters from the test that can be used to express the stretch-forces of the cloth [27].

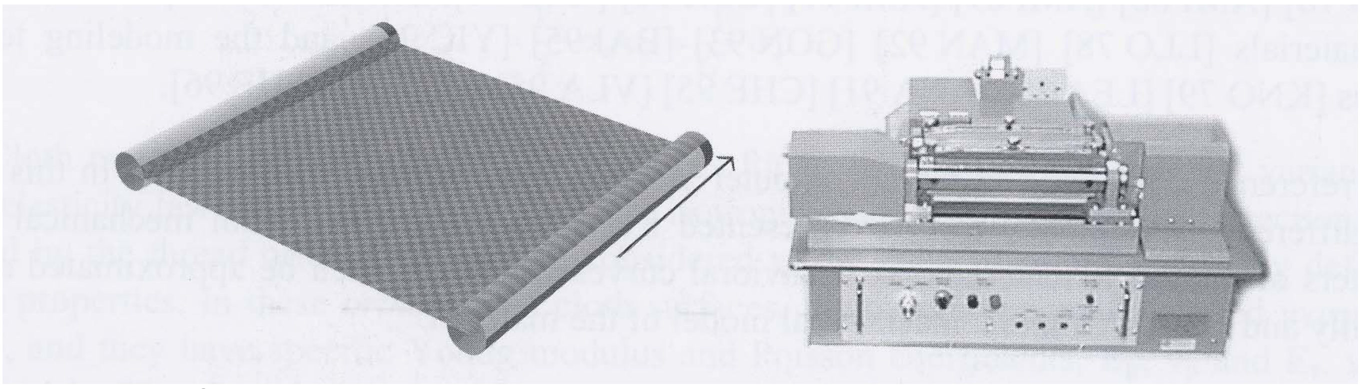


Figure 2.4 - The Shear test machine [27].

The Shearing test is performed in the same manner as the tensile test, but now the fixed edges moves in opposite directions parallel to each other and the shearing stiffness is calculated at three different angles; 2.5, 0.5 and 5.0 degrees [27].

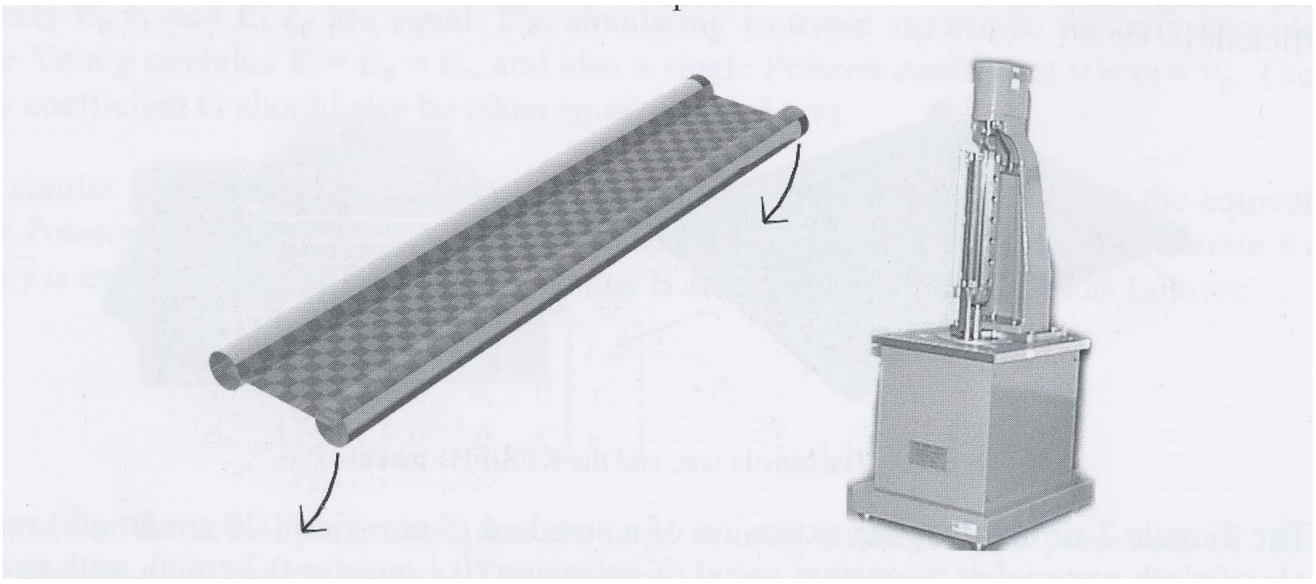


Figure 2.5 - The Bend test machine [27].

The bending test, as the name suggests, finds the bend-forces, this is done by attaching two opposite edges of a cloth and then rotating an edge to obtain a cylindrical form, and the flexion stiffness is calculated [27].

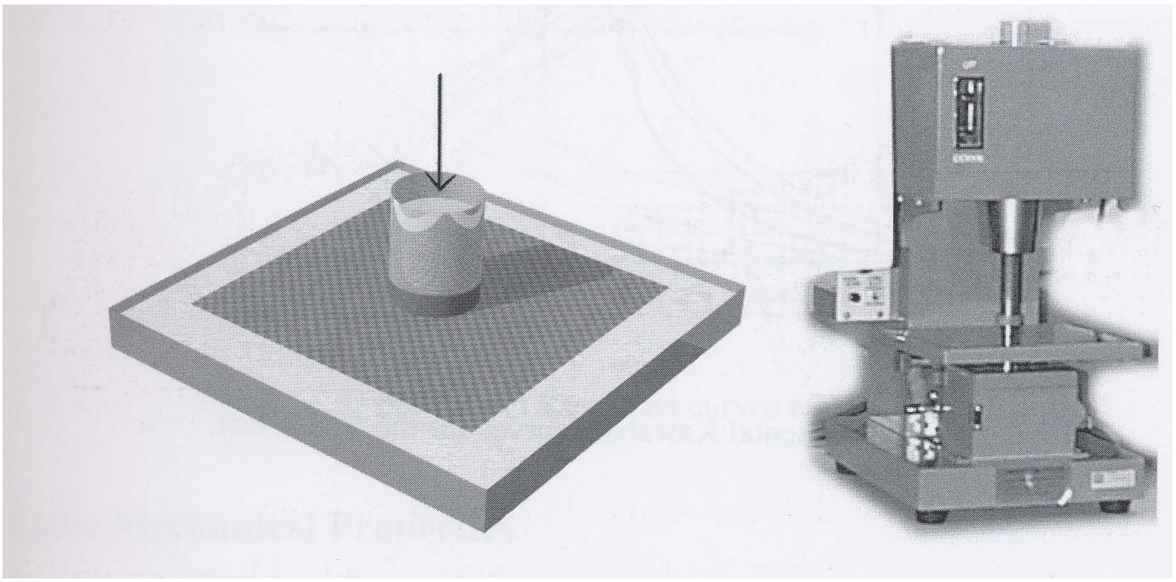


Figure 2.6 - The Compression test machine [27].

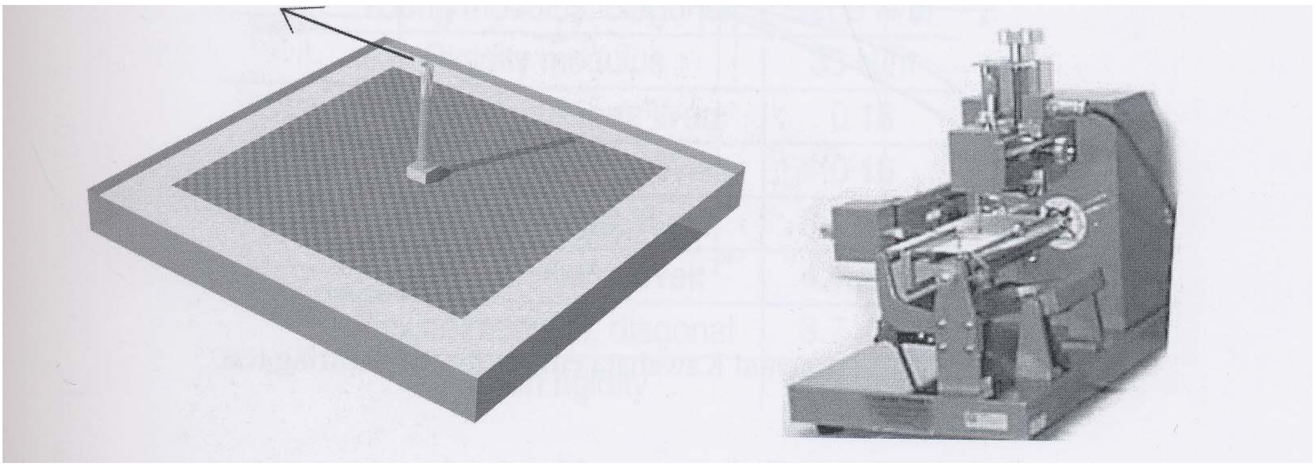


Figure 2.7 - The Friction test machine [27].

These three tests are the most widely used within the fabric industry, and represent the most important forces needed for a cloth simulator. However there are two more tests in the Kawabata system, the compression test and the friction test. The compression test calculates how much the thickness of the cloth compresses when a force is applied, the friction tests finds the frictional forces the cloth has.

2.1.2 Underlying Representation

If you look closer at figure 2.1, you might realize that one way to simplify the problem is to think of it as a grid of particles with springs connecting them, also known as a particle representation, this is illustrated in figure 2.10 [3]. This representation can be used for any substance, if it is applied to the atomic or molecular structure of that substance. This would not be a very efficient representation since there would be too many particles [3]. But as we saw in figure 2.1, the structure of a cloth is indeed very similar to a particle representation at a macroscopic level, compared to molecules or atoms.

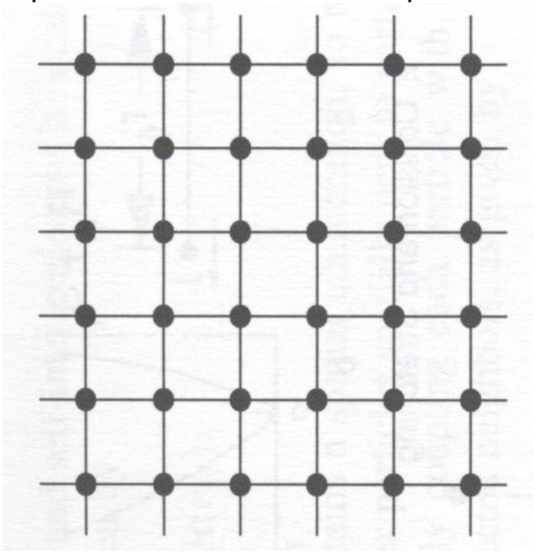


Figure 2.10 – This figure illustrates a particle grid [3].

All simulation techniques have used a mesh to represent the clothes. Two different methods of representing cloth meshes where encountered. Terzopoulos et. al. [12], amongst others, use a rectangular mesh, as seen in figure 2.8 (a), and the other method is to use a triangular mesh like in figure 2.8 (b). When they have used rectangular meshes, they often also use shear-meshes, like Zeller [14]. These meshes are equivalent to rectangular meshes, but they are rotated 45 degrees, as seen in figure 2.8 (c).

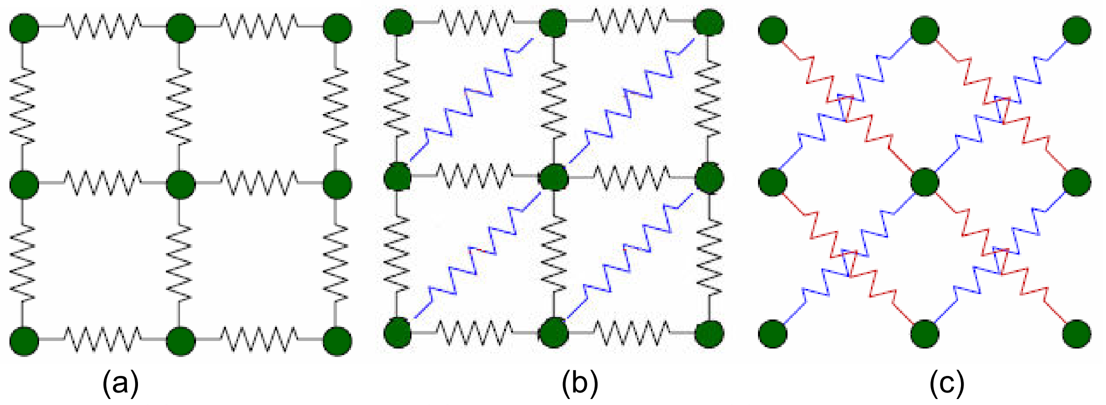


Figure 2.8 - (a) is a rectangular grid, (b) is a triangular grid and (c) is a shear grid.

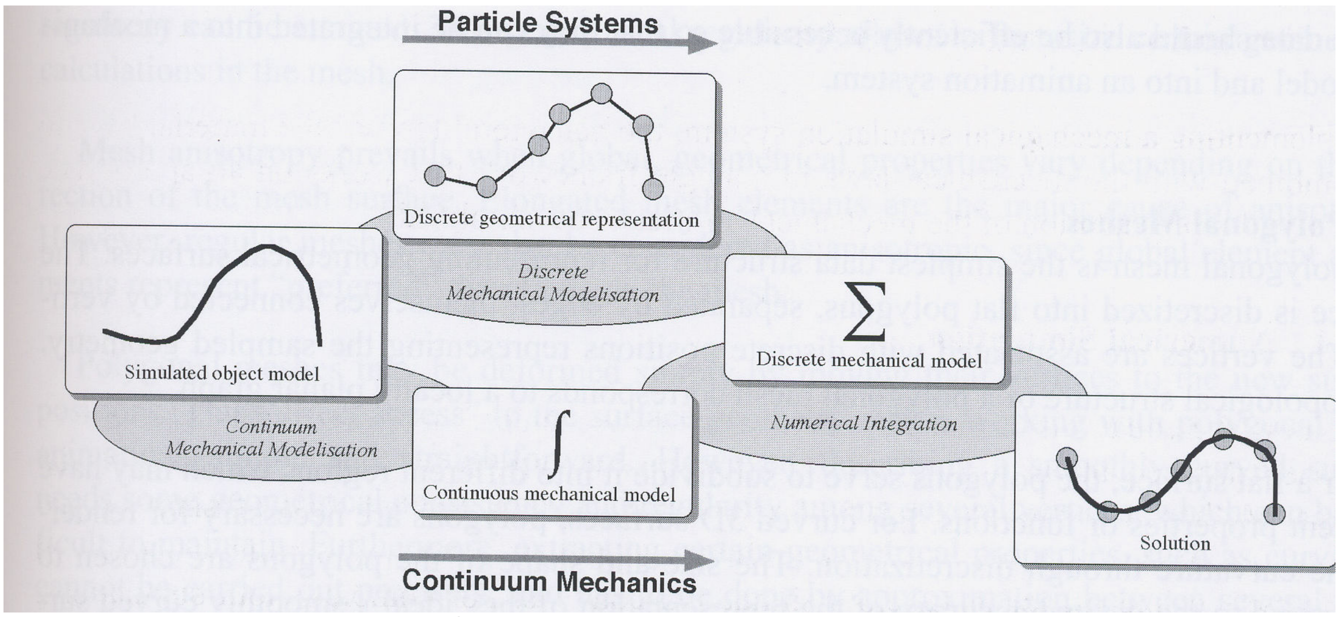


Figure 2.9 - The discretization scheme for the continuum mechanic model and particle systems [28].

2.1.2.1 Particle Systems

There has been done much research with particle systems for cloth modelling, [2, 4, 6, 12, 14], and though there are some differences among the different approaches (force-functions, numerical solution methods, collision detection, etc.) there is a common factor. They all formulate the problem as a time-varying differential equation (2.1).

$$a = \frac{F}{m} \tag{2.1}$$

Where a is acceleration, F is the force and m is the mass of the particle.

When we use this representation, we can formulate forces that act on the particles and the simulation consists of moving these particles, as seen in figure 2.9. We will look closer at how we can formulate forces for particles in the physics sub-chapter.

The differential equation in these methods is to complex to integrate directly, thus they have to solve it using numerical integration techniques. We will look closer at these methods in the next chapter.

2.1.2.2 The Continuum Mechanic Model

The continuum mechanic (CM) model expresses the surfacic deformation energy using differential expressions which are continuous over the entire surface being modelled, unlike the particle systems which apply forces to the particles, as seen in figure 2.9. This is shown in figure 2.9. The big advantage this representation yields, comes from the fact that these differential equations can be directly extrapolated from the mechanical properties the surface has in the real world, using for instance the Kawabata system. The drawback is that it does not handle frequent collisions well, and in cloth modelling collisions occur quite often.

The CM model was first used in cloth modelling by Terzopoulos et. al. [12], he used the Lagrange equation models to integrate the model. The Lagrange equation is the starting point of many current models that models the cloth using continuum mechanics [28].

$$\frac{\partial}{\partial t}\left(\mu(a)\frac{\partial r(a,t)}{\partial t}\right) + \gamma(a)\frac{\partial r(a,t)}{\partial t} + \frac{\partial \varepsilon(r)}{\partial r}(a,t) = f(r(a,t),t) \tag{2.2}$$

The Lagrange form balances the internal forces with the external forces acting on the body. r (a, t) is a position of a, the particle of the body with coordinates α_1 and α_2 at time t. $\mu(a)$is the mass density and $\gamma(a)$ is the dampening force, $\varepsilon(r)$ is a functional which measures the net instantaneous potential energy of the elastic deformation of the body. f (r, t) represents the net externally applied forces [12, 28]

The mechanical behaviour of the body should be expressed as the local deformation energy related to the actual material deformation and expressed locally for any surface point. Most of the time both elongation and curvature deformations will be allowed, and then the elastic surface energy is represented by two components G and B [28].

$$G_{ij}(r(a,t)) = \frac{\partial r}{\partial \alpha_i} \cdot \frac{\partial r}{\partial \alpha_j}$$

$$B_{ij}(r(a,t)) = \frac{\frac{\partial r}{\partial \alpha_1} \times \frac{\partial r}{\partial \alpha_2}}{\left| \frac{\partial r}{\partial \alpha_1} \times \frac{\partial r}{\partial \alpha_2} \right|} \cdot \frac{\partial^2 r}{\partial \alpha_i \partial \alpha_j} \tag{2.3}$$

The internal energy is then derived from these expressions and equation (2.4) expresses how we can find $\varepsilon(r)$ [28].

$$\varepsilon(r) = \int_{\Omega} \sum_{i,j=1}^{2} (\eta_{ij}(G_{ij} - G_{ij}^0)^2 + \xi_{ij}(B_{ij} - B_{ij}^0)^2) d\alpha_1 d\alpha_2 \tag{2.4}$$

Where η_{ij} and ξ_{ij} are the weighting functions which controls the elongation and curvature rigidities of the surface.

The expression is then integrated in the LaGrange equation, equation (2.5) and (2.6), which has turned into a linear differential system that has to be solved for obtaining the evolution of the deformable surface [28].

$$\frac{\partial r}{\partial \alpha_2}(m,n) = \frac{r(m,n+1) - r(m,n-1)}{2\Delta_2}, \frac{\partial^2 r}{\partial \alpha_2^2}(m,n) = \frac{r(m,n+1) + r(m,n-1) - 2r(m,n)}{\Delta_2^2} \tag{2.5}$$

$$\frac{\partial^2 r}{\partial \alpha_1 \partial \alpha_2}(m,n) = \frac{\partial^2 r}{\partial \alpha_2 \partial \alpha_1}(m,n) = \frac{r(m+1,n+1) + r(m-1,n-1) - r(m+1,n-1) - r(m-1,n+1)}{4\Delta_1 \Delta_2} \tag{2.6}$$

These equations cannot be solved analytically, but has to be discretisized and a numerical method applied to solve it. Usually, an interpolation-function is used, these functions can have different degrees of freedom depending on how many parameters are used (bi-linear, tri-linear, etc.). The more degrees of freedom the function has, the more accurately it approximates the surface. [28].

2.1.3 The Winged Edge Data Structure

The WE structure consists of vertices, faces and edges.

- Vertices only have information of the position.
- Edges have information of which vertices it connects, and which faces it is a part of, as well as the other edges it is connected to.
- Faces have information of which edges it consists of, and which vertices define it.

An illustration of the WE structure is given in figure 2.11.

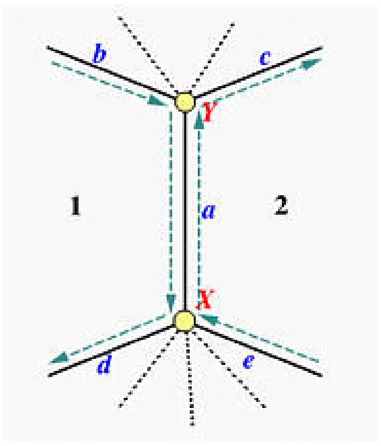


Figure 2.11 - An example of the WE structure: (a, b, c, d, e) are edges (1, 2) are faces and (X, Y) are vertices. E.g. (a) has information of all the elements in the figure [8].

If we look at figure 2.10 and 2.11, we see that the WE structure can easily model the particle grid that represents a cloth. The particles can be implemented as vertices and springs can be implemented as edges.

There are other data-structures that could have been used to represent the clothes. Half edge (HE) [7] is an example of this. The HE structure is almost equivalent to winged edge. The HE structure has edges that connect two vertices directionally, and each face has its own edges. Therefore if two faces share two vertices, there will be two duplicate edges (with opposite directions) in the structure. WE only have unidirectional edges and therefore will not have duplicate edges. Figure 2.12 illustrates the HE structure. HE would have worked just as well as the WE structure, but WE was chosen. It is also worth mentioning that since the WE structure was well known prior to the start of this project, and it was understood that it had easy access to the information of the particle grid which was needed for all the force and derivative calculations. There was no need to find another representation.

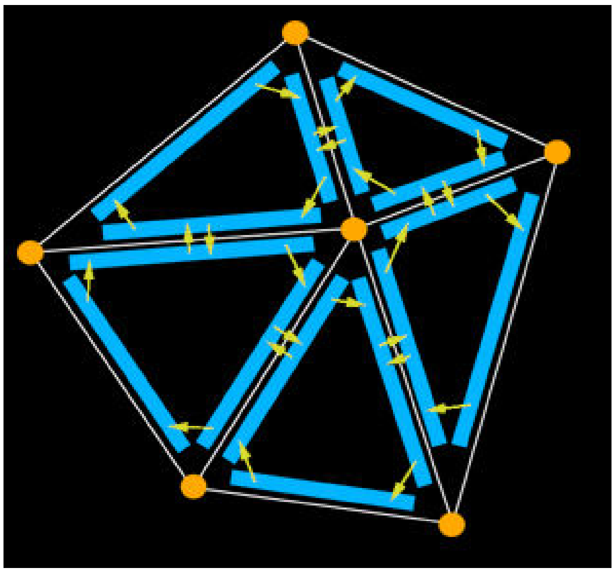


Figure 2.12 - An example of a HE structure. As you can see, the similarities with the WE structure is significant, the difference is however that the HE structure has directional edges [7].

2.2 Physics

We will now look at the physics of clothes, in cloth simulations we often look at only three internal forces, stretch, bend and shear, as mentioned in chapter 2.1. These forces are modelled by energy functions. Energy functions will be briefly explained before we look at the different internal forces of the cloth.

2.2.1 Energy functions

Energy functions are in general not good for representing the internal forces acting on a cloth. This is because sensible damping functions cannot be derived from energy functions as shown by [2]. If we try to construct a damping function directly from the energy function, we will get a damping function that is independent of the velocity that affects the spring. The damping function for a spring should only reduce forces from the spring when the velocity does change the length of the spring. Thus, we should rather construct the energy function as a sum of sparse vector conditions, C(x). Vector conditions should be thought of as functions which we want to satisfy the condition C(x) = 0, this is done by defining the associated energy functions as shown in equation (2.7) [13].

$$E_C(x) = \frac{k}{2} C(x)^T C(x) \qquad (2.7)$$

Where k is the stiffness constant of the expression, C (x) is the vector condition and x is a vertex/particle-position.

The force an energy functions produces can be derived by differentiating equation (2.7) with regards to x_i:

$$F_i = -\frac{\partial E_C}{\partial x_i} = -k \frac{\partial C(x)}{\partial x_i} C(x) \qquad (2.8)$$

When we apply this force to the particles, the particles will try to reach a state that makes the vector condition, C (X), reach zero.

2.2.2 Damping functions

For a particle system one needs damping forces to prevent the system from bouncing forever. If no damping functions were applied, no energy would be lost at all. This would result in a system that would never come to rest.

A good way to describe damping functions is to construct an energy functions based on vector conditions, and then differentiate with regards to time. This way, each damping force is applied only on behalf of their corresponding force, and this is exactly what we want [2]. I.e. damping forces for springs should not involve damping of the force of the gravity. Equation (2.9) shows how the damping function is dependent on the rate of change [13]. Equation (2.9) is derived by differentiating equation (2.8) with regards to time.

$$d = -k_d \frac{\partial C(x)}{\partial x} \dot{C}(x) \tag{2.9}$$

Where kd is the damping constant, $\dot{C}$ is the time derivative of C.

2.2.3 Springs

A spring is a way to connect two masses so that they apply a force towards each other. When specifying a spring we need three constants, the springs stiffness (ks), the springs damping factor (kd) and a rest distance between the masses (rest).

Since a spring tries to hold two masses at rest at a predefined distance, we can construct a vector condition that satisfies the rest-condition.

$$C(x) = |L| - rest \tag{2.10}$$

Where L=xi-xj, (xi, xj) is the position of the particles i and j, and rest is the predefined distance between them. Note that C (x) = 0 when the distance between particle i and j is equal to the rest-distance.

From the vector condition in equation (2.10) we need to derive two other functions, $\dot{C}$ and $\frac{\partial C(x)}{\partial x}$, we obtain $\dot{C}$ by differentiating equation (2.10) with regards to time.

$$\dot{C} = \frac{L \cdot \dot{L}}{|L|} \cdot L \tag{2.11}$$

Where $\dot{L}$=vi-vj, (vi, vj) are the velocities of particle i and j.
This equation can be explained as a vector which holds the combined velocity of the two particles parallel to the vector between them.

To find $\frac{\partial C(x)}{\partial x}$ we only need to derivate equation (2.10), and the result is given in equation (2.12)

$$\frac{\partial C(x)}{\partial x_i} = \frac{L}{|L|} \tag{2.12}$$

We now have all the functions which we need to construct the force-function for a spring, and by inserting equations (2.10), (2.11) and (2.12) into (2.8) and (2.9) we get the following equation

$$F_i = -\left(k_s\left(|L| - rest\right) + k_d \frac{\dot{L} \cdot L}{|L|} \right) \frac{L}{|L|} \qquad (2.13)$$

A more detailed walkthrough is given in [13].

2.2.4 Internal forces

In a cloth there are certain forces acting on the cloth which have are based on the physics of the cloth and are not directly dependent on external factors. There are three forces that are called internal forces in the Kawabata system [27], bend, shear and stretch [19]. We will look closer at the Kawabata system in the next chapter.

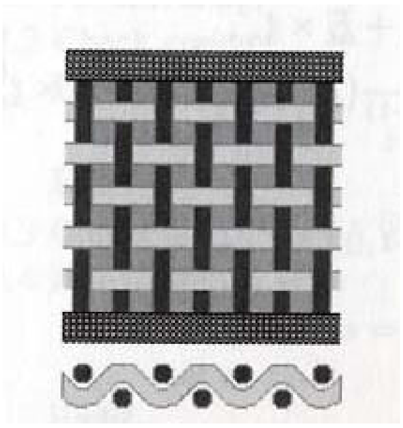


Figure 2.13 – This figure illustrates how the cloths internal forces act [29].

As we all know, threads can be stretched, but they resist this motion with a certain force. We then see that If we try to deform it “in-plane” (shear) the threads will interact and resist this shearing since some threads will be stretched around other threads. The same happens if we try to bend it, some threads will become longer then they are (stretched) and will resist this action as well, the structure of cloth illustrating these forces are shown in figure 2.13.

The three different types of internal forces are:

- **Stretch**-forces try to resist stretching and compression of connected particles. These forces will try to maintain the distance between the particles.
- **Shear**-forces try to resist in-plane deformation of adjacent particles, it will try to maintain the in-plane angle between edges that connect particles.
- **Bend**-forces try to resist out-of-plane deformation of adjacent particles, as shown in figure 2.14, these forces will try to maintain the out-of-plane angle between connected particles and prevent two edges from folding together due to bending of the cloth.

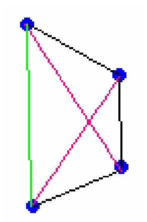


Figure 2.14 - As illustrated in this figure, the green and red bend-springs will prevent the cloth from folding completely flat. Black lines are the stretch-springs, and the blue dots are the particles.

2.2.5 External Forces

The most important external forces are gravity, wind and air-drag.

- The gravity force applies a force towards the center of the earth.
- Wind applies a force to the particle according to the angle between the wind-direction and the triangles that each particle is a part of.
- Air-drag applies forces to the particles according to the angle between the directional velocity particles and the triangles that each particle is a part of.

2.3 Graphical Processing Unit

The graphical processing unit (GPU) was initially designed to speed up 3D processing, but as time has gone by, the previously static programming on the GPU has been replaced by vertex- and fragment-processors that can be programmed much like you can program the CPU. Recently framebuffer-objects were introduced to openGL, this allows the programmer to have much more control over where the rendering occurs, and it speeds up switching between different drawbuffers. Although the GPU is faster than the CPU when it comes to pure arithmetic calculations, there are some aspects we should think about in our GPU-programs.

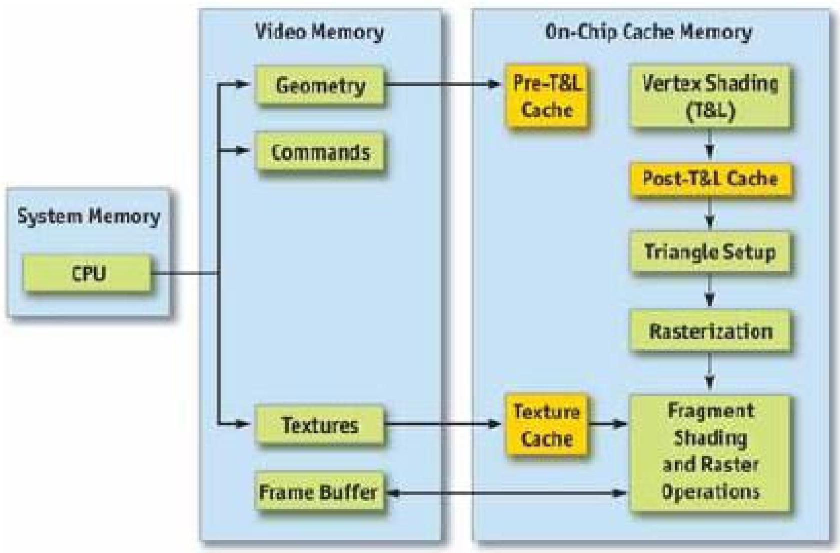


Figure 2.15 – The process of rendering an object on the GPU [35]. In newer versions of GPUs, textures can also be used in the vertex-shader.

The process of rendering an image on the GPU, as illustrated in figure 2.15, is initiated by the CPU, it sends information about the positions/vertices and which geometry these vertices has (triangle, points, quads, etc.). The CPU also tells the GPU which textures to use. When the GPU has received all the information about what it is going to render, the vertices are transformed and lit (T&L) in the vertex-shader. Then the GPU splits up the object to be drawn into triangles, these triangles are rasterized, meaning that the GPU finds out which pixels each triangle covers and thus which pixels are to be drawn to. Finally, the fragment-shaders calculate the final colour and apply the texture on the pixels before they write the colours to the frame buffer.

2.3.1 Vertex-processor

The Vertex-processors (VPs) are small processing units on the GPU which works with the data that is sent to the GPU. The VPs main objective is to transform vertices via the modelview-matrix and calculate the resulting colour from lighting. An illustration of a vertex-program that moves a particle is given in figure 2.16 where a single position is translated. Recently it also became possible to relocate the point by a texture-lookup which was quite convenient when you want to use the result from a calculation, now you can simulate water or particle systems entirely on the GPU and use these positions to relocate the vertices that is sent to the GPU and create astonishing visual effects. VPs however lack the ability to write any values to memory, all the data from the VP is sent, and possibly interpolated, to the fragment-processors.

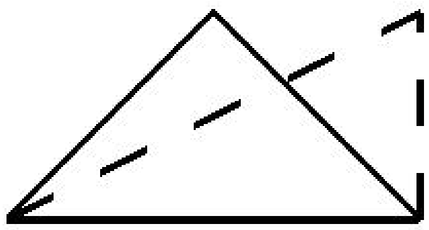
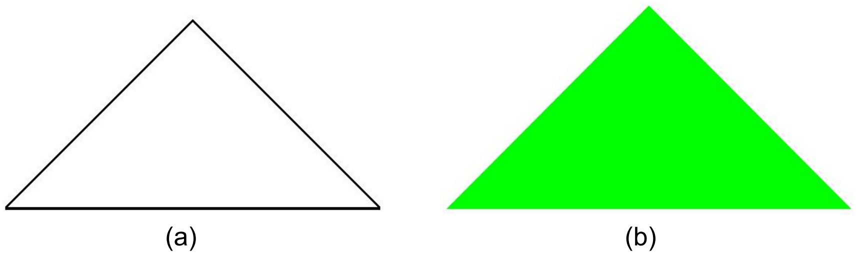


Figure 2.16 - This figure illustrates what would happen if we instructed the vertex-program to change the position of the upper position/vertex. The stippled line shows us the new edges of the triangle, the solid lines shows us the original triangle.

2.3.2 Fragment-processors

The fragment-processors (FPs) are used to calculate the final colour of each pixel that is drawn. The strength of the FPs are that they can write data to memory, however they can only write up to 16 floating point values per pixel, and they can only write these values to their precalculated addresses. The FPs are very fast because of their parallel Single Instruction Multiple Data (SIMD) architecture, and it is here most General Purpose GPU (GPGPU) work is done. Figure 2.17 shows us a triangle before and after a fragment-program has coloured it.



(a) (b)

Figure 2.17 - This figure shows us a triangle (a) without fragment-programs colouring it and (b) with fragment-programs colouring it green.

2.3.3 Framebuffers

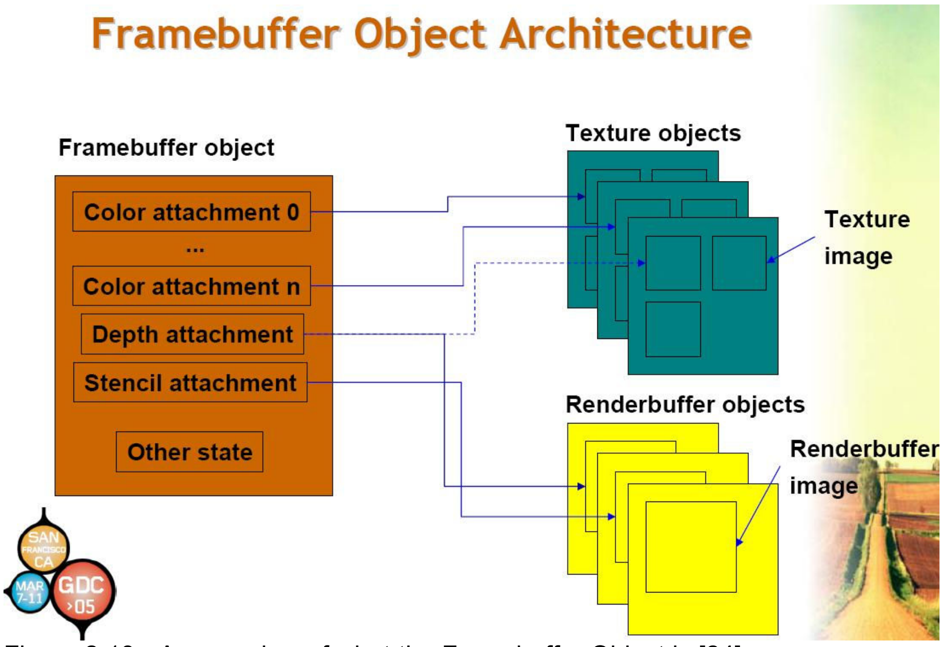


Figure 2.18 - An overview of what the Framebuffer Object is [24].

The Framebuffer Object (FBO) is, as we can see in figure 2.18, a collection of logical buffers (color, depth, stencil, accumulation). A texture can be attached to any of these buffers, when this is done, you can activate rendering to this texture by using glDrawBuffer (TARGET) when its FBO is active. As figure 2.18 also indicates, one can use renderbuffers, but these cannot be attached as textures later on.

Though FBOs are very fast, there are some traps one can fall into, you should not create and destroy FBOs every frame, neither should you draw directly to the buffer using TexImage or CopyTexImage [23]. You should also try to avoid switching the active FBO too much, rather you should “fill up” the FBO. This means that you should attach different textures to the same FBO and switch between them using glDrawBuffer. Currently there is a limit of four colour-buffers per FBO, and all textures must have the same dimension.

2.3.4 Optimization

Even though the GPU is very fast, one also has to consider the algorithm to use, a O(n^3) is still slower than O(n^2). But there are also many traps we can fall into when we try to make a GPU program, we could be wasting the performance of the GPU just because we did not think through how the GPU works. We will now look at some quick hints about what to look for to optimize our GPU programs. Most of these performance tips are given in [23].

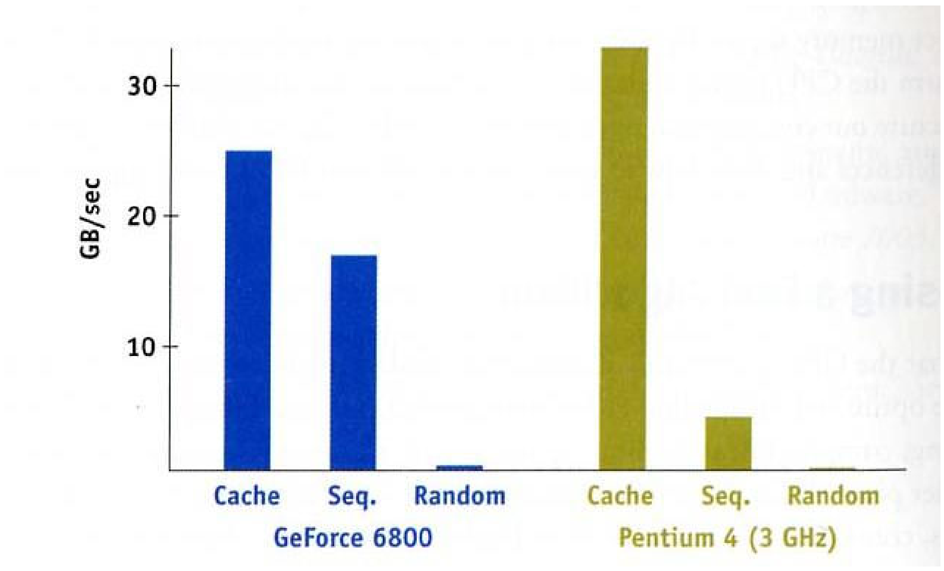


Figure 2.19 - this figure illustrates the differences in memory performance between a graphics card and a CPU [26].

As we see from figure 2.19, the GPU performs better than the CPU when it fetches data sequentially or randomly, though the performance for random look-ups is very poor. This might look as another advantage for the GPU, however the CPUs cache is large enough to significantly reduce the cache-miss ratio, and thus most of the memory-reads on the CPU are done from the cache, which is much faster than cache-reads on the GPU. The cache on the GPU is significantly smaller than the cache on the CPU, typically only a few texels to accelerate texture filtering. This is however not the typical scenario for a GPGPU program, and to optimize our program we need to reorder our data to reduce the cache-miss ratio as much as possible, and if that is not possible, we should at least try to avoid the devastating random-access pattern.

We can further improve the performance of the GPU by using half-precision floating points (FPs) instead of the full-precision FPs, according to [23] this can decrease rendering time by a factor of up to nine.

We should also consider balancing the workload between the fragment- and vertex-programs, there is no need to calculate the same operation e.g. a = width*height for every pixel, such values can be calculated in the vertex-program and passed on to the fragment-program. In general this means that you should not have an empty vertex-program and an overloaded fragment-program. If this is the case, look for something to move to the vertex-program to obtain balance.

Another improvement is to compile each program with the earliest version possible, nVidias C for Graphics (cg) provides different versions of both vertex- and fragment-programs. The lower this version-number is, the more simple operations are used. Thus an earlier version operates faster than a newer version. In addition to this, if you try to change the program to fit into an earlier version, you might find some expensive operations that can be simplified using ordinary algebraic operations. The built-in operations are faster in use than if you try to do the same as the operation.
$length(vek) == \sqrt{vek.x^2 + vek.y^2 + vek.z^2}$,
but if you have knowledge about the length of this vector (for instance it is normalized) you can avoid using it at all, which is much faster than both.

Loops and branches are the final optimization issue that will be addressed here, though it is possible to use such programming-techniques, it is recommended that you do not. A general tip is to use branching when pixels in the same region tend to choose the same route, 30x30 regions or larger with the same branch is preferable. If it is much smaller, then it might be more efficient to use more rendering-passes with different fragment-programs [23].

2.4 Summary

Clothes are complex in nature, but several attempts have been made to simulate them. We looked at the continuum mechanic model which gave visually pleasing results, but this method is not optimal when frequent collisions occur. We also looked at how particle systems can be used to simulate clothes by applying springs between the particles.

The internal forces of clothes can be derived from energy functions and energy functions can be constructed by vector conditions. We saw that when we construct energy functions based on vector conditions, the force we calculate from the energy functions will move the particles so that the vector conditions reach 0. We also introduced damping functions by differentiating the energy functions with regards to time. Since the energy functions are constructed by vector conditions, the damping functions for each energy function will only dampen the forces according to the velocities that affect the energy functions.

Internal forces are forces that represent the internal dynamics of a cloth. These forces are not dependent on external forces directly. External forces include gravity, wind and air-drag, these forces have nothing to do with the internal dynamics of a cloth, but they affect the cloth, and are necessary for a realistic simulation.

The GPU is a powerful tool, it is much faster then the CPU because of its parallel architecture. Recently FBO support was implemented in OpenGL. These FBOs have both drawbuffers and renderbuffers, the renderbuffers can however not be used in later render-passes. A texture can be attached to any drawbuffer, and in turn we can write to these textures. We also looked at how the GPU pipeline renders an image using the vertex- and fragment-shaders. The vertex- and fragment-shaders have evolved to include programmability, this means that we can construct our own programs to manipulate what happens in the GPU pipeline.

3. Cloth Modelling (Previous Work)

There has been done much in the area of cloth modelling for many years. Terzepoulos [12] was the first to use a physically based method to simulate clothes. He used a finite element method to simulate the cloth based on its mechanical properties, but in later years much simpler methods have been proposed. [2, 4, 6, 14, 15, 16, 21] use various methods to simulate clothes. The common factor of these methods is that they define the problem using a particle system and end up with a differential equation that has to be integrated. However, [2] also has to solve the linear system of equations that arise. The particle system methods express the changes using a differential equation. This differential equation has to be integrated numerically, for this purpose there are two different categories of methods to use, explicit and implicit integration techniques.

3.1 Implicit and Explicit Methods

Cloth simulation eventually comes down to a differential equation that has to be solved. For each iteration of the simulation, this differential equation has to be integrated to obtain the velocity changes of the particles. The differential equation that arises is to complex to solve directly and therefore we need to solve it using a numerical approximation. The Lagrange equation model is integrated using the finite element method [28]. To integrate the differential equation from particle systems, there are two fundamentally different numerical methods that are used, explicit and implicit methods.

The explicit Euler method is a simple method and only requires knowledge of the force and velocity of the particles in this state, as seen in equation (3.1) which is the equation for explicit Euler.

$$\frac{d}{dt}\left(\frac{x}{v}\right)=\left(\frac{v}{M^{-1}f(x,v)}\right) \qquad (3.1)$$

Here x is the position of the particles, v is their velocities, M^{-1} is their inverse mass and f(x, v) is a function of their forces with regards to all the particles position and velocities.

Integrating equation (3.1) is easy, as nothing in this formula is dependent on time. Thus, multiplying the right side by Δt is the whole integration. The integration will lead us to: Δx =v Δt and $\Delta v = M^{-1}$f(x, v) Δt. This is why an explicit method often is called a linear method, its complexity is O(n), and the rate of change is linear with regards to the time-step.

The implicit Euler method tries to find the next state by finding a state in which it can go back to the present state if we negated the time step (move backward in time). This can be seen in equation (3.2), which is the equation for implicit Euler.

$$\frac{d}{dt}\begin{pmatrix} x \\ v \end{pmatrix} = \begin{pmatrix} v0 + \Delta v \\ M^{-1} f(x0 + \Delta x, v0 + \Delta v) \end{pmatrix} \tag{3.2}$$

Although this equation yields a better solution, it is important to understand that this equation leads to a nxn set of linear equations that needs to be solved for each iteration. As pointed out by [2], due to its sparse nature and the fact that you can use larger time-steps for each iteration makes this a very reliable system, and the ability to use larger time-steps more than makes up for the complexity. An attempt has been made to avoid solving the linear set of equations and still have the strength of an implicit method, [4] found that if one solved a certain Hessian matrix at the beginning of the simulation and used its inverse later, the result would only have rotational errors. These errors were corrected using the knowledge of them at runtime and the method yielded very good results [4]. We will look closer at how we can integrate the implicit method in chapter 4.

As cloth is very resistant to stretching and compression, one will have to apply strong forces between the particles. As pointed out by [5], explicit methods do not handle strong springs well, but there are some “tricks” one can use so they will handle them. The reason why explicit methods have problems with stiff springs is because the explicit method is linear with regards to the size of the time-steps. Thus, the particles will oscillate to infinity if the forces are strong enough [1], see figure 3.1. This occurs because an explicit method only considers the force and velocity of the particle now when it calculates the changes in positions and velocities. If we increase the time-step the rate of change will only increase accordingly. Implicit methods handle this in a better way, because they also look at the force and velocity for the next state.

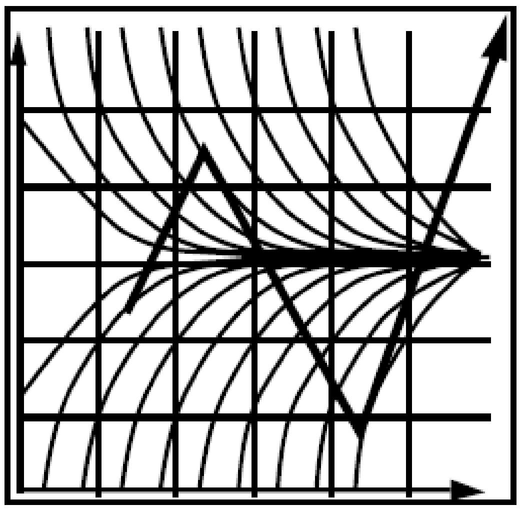


Figure 3.1 – This figure illustrates a particle in a force-field where the force-field tries to drag the particle to the y=0 line. The straight lines show how the explicit method moves the particle, but because of the size of the time-step the particle moves further away from the line each iteration. The curved lines illustrate the force-field [1].

The explicit method is a linear approximation, if you increase the time-step, it will move further along the same line. Distance travelled is only dependent on the force and velocity in this state. The implicit method also depends on its next position, and therefore will include this knowledge and when the time-step increases it will also alter the direction of the changes, and thus the particle will move closer to the solution if we increase the time-step, as seen in figure 3.2. This is however not the case in a cloth simulation, these equations will not allow for infinite time-steps, this is because there are so many particles that change positions and alter the forces in the model.

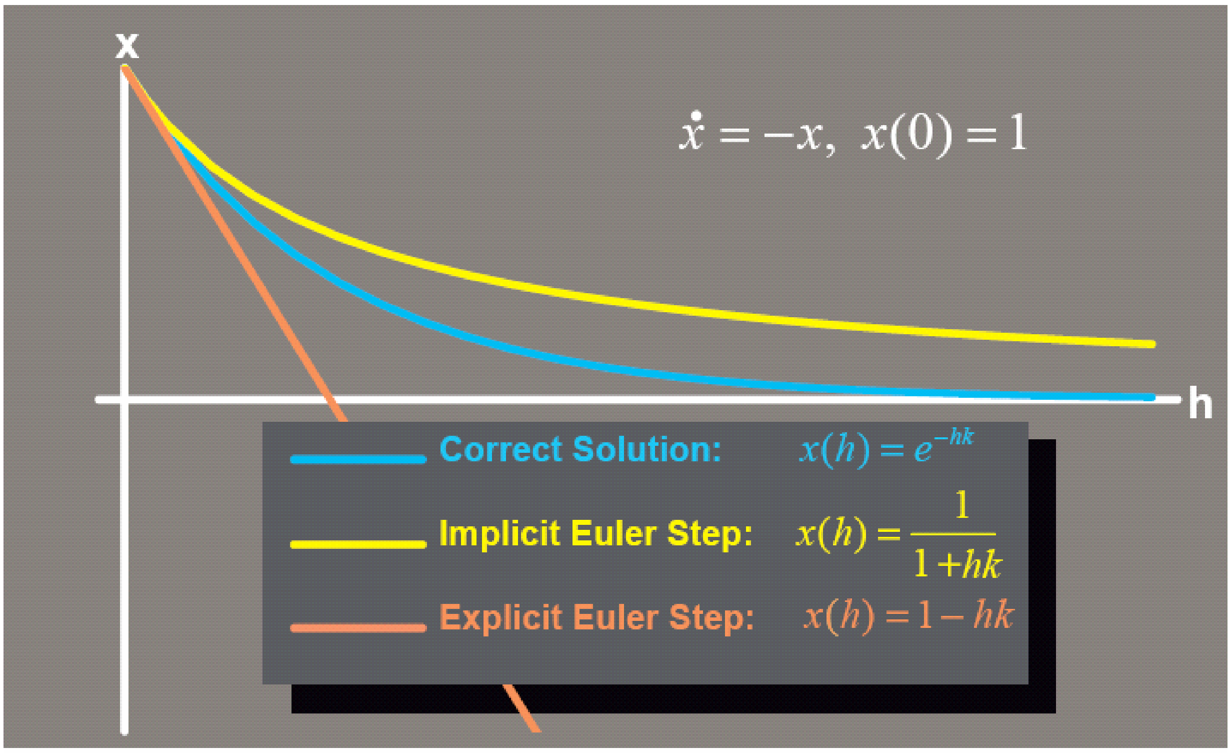


Figure 3.2 – In this figure you see the correct function in blue, the explicit approximation in orange and the implicit in yellow [18].

If one where to use an explicit method to solve the differential equation that contains stiff springs, one could do three things to prevent it from oscillating:

- Reduce gravity, thus one can reduce the spring forces.
- Add strong damping forces on all velocities.
- Reduce time-step.

It is usually not enough to use only one of these alone. The most important is the third, but if one reduces the time-steps, one would have to solve the system many times for each frame. This would reduce its frame-rate and thus its practicality would drop considerably since the best reason for using explicit methods is that it is faster per iteration. Reducing gravity and/or adding strong damping forces to the entire system would make it look unnatural; it might look as if it was on the moon or in a fluid.

3.2 Previous Solution Methods

When we simulate clothes using the continuum mechanic model or the implicit particle systems we get a large sparse linear system of equations. There are many methods to solve this equation.

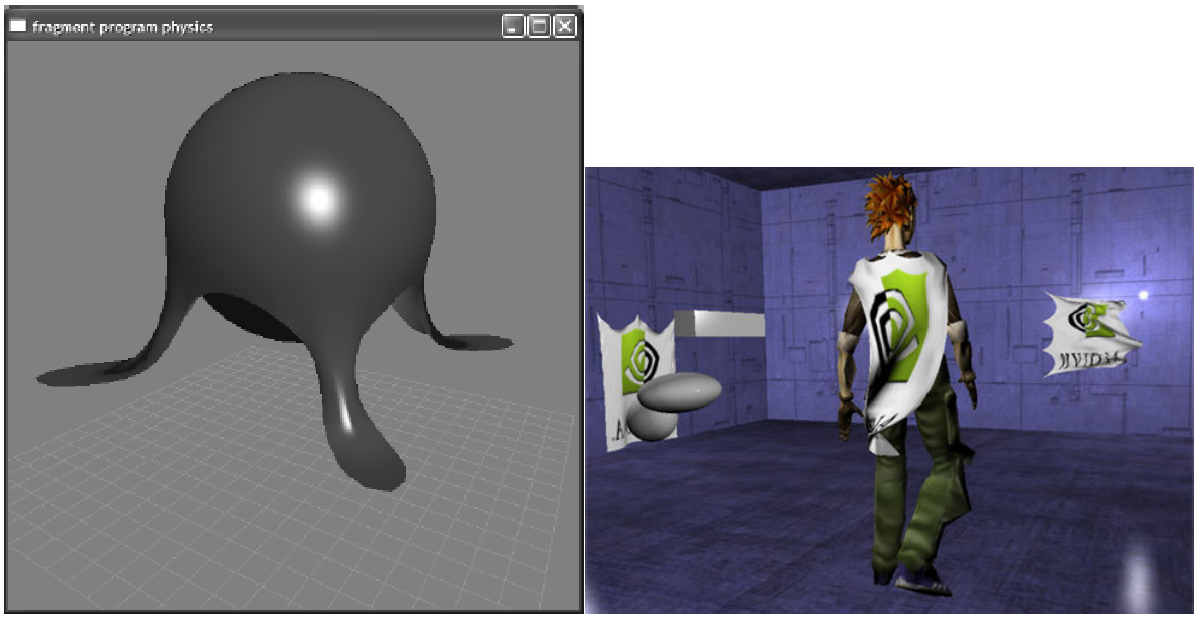
$Ax = b$

This system of linear equations can be solved in two different ways, direct methods, like Cholesky decomposition [17], Gauss elimination [11] and LU decomposition [32], or numerical approximations, like Conugate Gradient[10, 19], Jacobi [33] and Gauss-Seidel relaxation [34].

The direct methods, as the name indicates, solve the linear system of equations directly, using mathematical operations that guarantee that the solution is correct, if one exists. The numerical methods iterate to converge to the correct solution vector. If the input meets all the requirements of the method, it will converge; if not, no result is guaranteed. The Conjugate Gradient method and the Jacobi method will be presented later on in this report.

3.3 State of the Art

Although there has been done a lot of work for many years, some innovative ideas are attempted. Attempting to construct more realistically cloth simulations with fewer particles and moving the calculation to the GPU are some of the fields of interest today. [25] tries to construct realistically wrinkles on small particle systems, NVidia has attempted to construct cloth simulators on the GPU, twice, Green [15] implemented an explicit particle system which proved to be very fast, but it was not very visually pleasing as we can see from figure 3.3 (a). Zeller [14] implemented a cloth simulator based on constraint relaxation, this is not a physically correct method, but it looks very good and performs very well, as seen in figure 3.3 (b). The trend of today seems to be that the graphics community tries to construct visually pleasing clothes that does not require too much of the processing power of the computer so that it can be used in real-time applications.



(a) (b)

Figure 3.3 - (a) shows us the result from Simon Greens cloth simulator, (b) shows us the result from Cyril Zellers cloth simulator. Both these methods are implemented on the GPU.

3.4 Summary

The Kawabata tests are used to find the internal forces in the cloth, how much it stretches, bend and shear. The expressions that are found from using this test can be used directly in the CM model, this is however not the case for particle systems, but it gives a good indication of which forces are needed and how strong they should be.

We looked at the two different underlying representations of a cloth simulator, the CM model and particle systems. They have many similarities, but they differ in their discretization scheme. The CM model discretizise during the intergration process, whereas the particle systems use their particle grid as their discretization.

There are two different ways to solve the differential equation that particle systems comes down to, explicit and implicit methods. We looked at the implicit and explicit Euler methods and found that the implicit method is much more stable, but requires much more work per iteration.

Both the CM model and the implicit particle systems requires us to solve a large linear system of equations. However, in both cases the system is very sparse, and thus efficient numerical solvers can be used to solve them efficiently.

We briefly looked at some of the recent research. It seems that it has its focus on constructing fast cloth simulators, either by making the cloth look more realistically with few particles or by implementing the cloth simulator on the GPU.

4 Conseptual Design

The aim of this diploma is, as previously mentioned, to investigate the possibility of implementing an implicit particle system on the GPU. This diploma is based on the implicit Euler method from [2] in order to see how this method will perform on the GPU compared to the CPU.

The approach used in [2] is based on the Euler method from equation (3.1), but instead of just moving the system forward, a restraint is added into the system. The restraint is that they start with the unknown next position and state that they want to go to the current, known, position. This is called the Backward Euler Method, the same as equation (3.2), and is an implicit method.

Implicit methods are much more stable than explicit methods, which means that much larger time steps can be used. This more than outweighs the nxn linear system of equations that has to be solved for each iteration of the system [2]. The system of equations is very sparse, and because of this we can apply efficient algorithms when solving them.

We will first look at the implicit Euler method to find out which forces and derivatives we need to calculate to solve the system. Then, we will se how these forces and derivatives are approximated in this project. Finally, we will look at a couple of numerical solvers that can be used to solve the linear system of equations.

4.1 Implicit method

As we saw previously, the formula for implicit Euler was

$$\frac{d}{dt}\begin{pmatrix} x \\ v \end{pmatrix} = \begin{pmatrix} \dfrac{v0 + \Delta v}{M^{-1} f(x0 + \Delta x, v0 + \Delta v)} \end{pmatrix} \tag{4.1}$$

Integrating equation (4.1) leads us to equation (4.2).

$$\begin{pmatrix} \dfrac{\Delta x}{\Delta v} \end{pmatrix} = \Delta t \begin{pmatrix} \dfrac{v0 + \Delta v}{M^{-1} f(x0 + \Delta x, v0 + \Delta v)} \end{pmatrix} \tag{4.2}$$

Now, we have to find a value for $f(x0 + \Delta x, v0 + \Delta v)$, this can be found by applying a first order Taylor series to f.

$$f(x0 + \Delta x, v0 + \Delta v) = f0 + \frac{\partial f}{\partial x}\Delta x + \frac{\partial f}{\partial v}\Delta v \tag{4.3}$$

After applying equation (4.3) to equation (4.2) and doing some regrouping we get

$$\left(I - \Delta t M^{-1}\frac{\partial f}{\partial v} - \Delta t^2 M^{-1}\frac{\partial f}{\partial x}\right)\Delta v = \Delta t M^{-1}\left(f0 + \Delta t\frac{\partial f}{\partial x}v0\right) \tag{4.4}$$

This matrix is not symmetrical. Thus, the CG solver cannot be applied to it. However, as explained in [2], we multiply the system by M on both sides to get a symmetric system.

$$\left(M - \Delta t\frac{\partial f}{\partial v} - \Delta t^2\frac{\partial f}{\partial x}\right)\Delta v = \Delta t\left(f0 + \Delta t\frac{\partial f}{\partial x}v0\right) \tag{4.5}$$

We now see from equation (4.5), we need to know $\frac{\partial f}{\partial v}$, $\frac{\partial f}{\partial x}$, $f0$ and $v0$. $v0$ however does not need to be calculated using a special equation as the others, we only need to calculate the new $v0$ using equation (4.2).

4.2 Forces and derivatives

As we saw earlier, there are three different internal forces acting on the cloth; bend, stretch and shear. There are several ways to approximate these forces [2, 12, 13, 16], usually energy functions are used to represent these forces [2, 13]. Baraff et. al. [2] formulated each of these forces by specific functions. However, in this project a much simpler approach was attempted with visual pleasing results, though not as adjustable to the Kawabata system as many other cloth simulators. This simplification was possible since the goal of this project is to compare a CPU-implementation against a GPU-implementation, not to create a highly adjustable cloth simulator. Also, these simplifications are visually pleasing enough to construct a particle system that looks like a cloth and has been used by other researchers, like [16]..

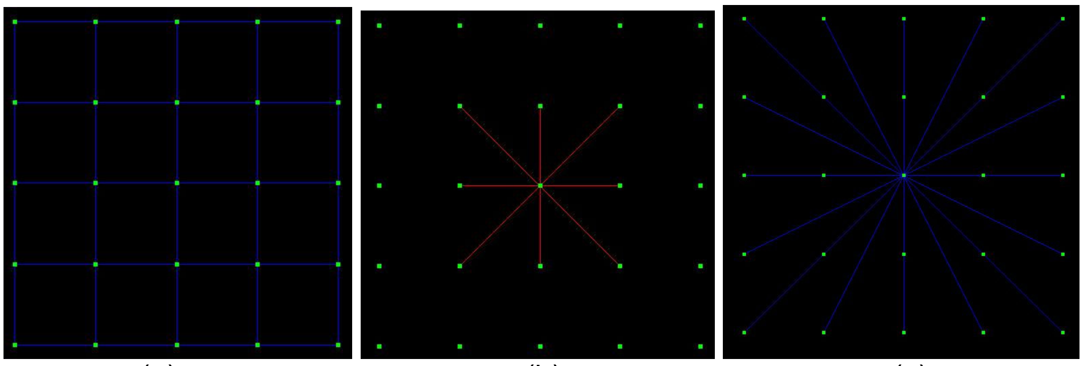

(a) (b) (c)

Figure 4.1 – This figure shows us that each particle is connected to exactly 24 other particles. (a) is a grid of 25 particles, (b) shows us the stretch- and shear-springs that involves the middle particle. (c) shows us the bend-springs involving the middle particle.

Forces are trivially computed using equation (4.6), and as we saw in chapter 4.1, we also need to know about the derivatives, $\frac{\partial F_i}{\partial x_j}$ and $\frac{\partial F_i}{\partial v_j}$, to construct our system of linear equations from equation (4.5). These equations are easily derived from the force function we saw in the chapter 2.

The formula for the spring-force is as follows

$$F_i = -\frac{\partial E}{\partial x_i} = -(k_s C(x) + k_d \dot{C}(X)) \frac{\partial C(X)}{\partial x_i} \tag{4.6}$$

When we now want to find $\frac{\partial F_i}{\partial x_j}$ and $\frac{\partial F_i}{\partial v_j}$, we differentiate equation (4.6), and thus differentiate the vector conditions. The derivative of the damping condition is physically incorrect, as shown in Baraff et. al. [2]. A term in the derivative is not symmetric, and thus is removed, without any observed ill effects.

$$\frac{\partial F_i}{\partial x_j} = -k_s \left(\frac{\partial C(x)}{\partial x_i} \frac{\partial C(x)}{\partial x_j}^T + \frac{\partial^2 C(x)}{\partial x_i \partial x_j} C(x) \right) - k_d \left(\frac{\partial^2 C(x)}{\partial x_i \partial x_j} \dot{C}(x) \right) \tag{4.7}$$

The next derivative is simpler, since the force is not dependent on the velocities, the ks term vanishes in the derivation.

$$\frac{\partial F_i}{\partial v_j} = -k_d \left(\frac{\partial C(x)}{\partial x_i} \frac{\partial C(x)}{\partial x_j}^T \right) \tag{4.8}$$

Now we have all the information we need to insert into equation (4.5) and solving it.

4.3 Constraints

In [2] they also show how one can apply constraints by modifying the mass of each particle. They represent the mass as a 3x3 matrix which contains the inverse mass, this way you can constrain it from travelling in specified directions. It is easy to see that if you want it to only be able to move in y and z direction, you would make its x-component of the mass to be infinitely large, thus making its inverse 0, we then get this matrix:

$$\begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Since this matrix is multiplied by the vertex representing the force of this particle, the force in the x-direction would disappear, and thus it would not move in the x-direction.

This can be made more general if you construct the matrix from equation (4.9) [2].

$$S_i = \begin{cases} I \\ (I - p_i p_i^T) \\ (I - p_i p_i^T - q_i q_i^T) \\ 0 \end{cases} \tag{4.9}$$

Where p and q are mutually orthogonal unit vectors. P and q represent directions that the particle is constrained from moving in. Thus the particle has to move in the plane that p and/or q defines.

When one implements the constraints this way the result would be equation (4.10).

$$\left(I - \Delta t W \frac{\partial f}{\partial v} - \Delta t^2 W \frac{\partial f}{\partial x} \right) \Delta v = \Delta t W \left(f0 + \Delta t \frac{\partial f}{\partial x} v0 \right) + z \tag{4.10}$$

Here W is M^{-1} with constraints and z is the change of velocity of the particles which we want to enforce in the constrained direction.

According to Baraff et. al. [2] this works as intended when solved with Gauss elimination, but this is too slow for larger systems. The real problem comes when they tried to use the CG-solver on it, because the matrix is not guaranteed to be symmetric. Thus various numerical methods for solving a system of linear equations cannot be used. The solution however is to introduce filter-functions in the numerical methods, and then apply the numerical method to equation (4.5). The filtering procedure is performed by multiplying each vertex in the array, vi, with its corresponding constraint matrix, Si, it then filters the vertices to take into account their constraints. Alternatively you could look at it as multiplying S with the array, which is in essence what happens.

4.4 Solvers

Equation (4.5) leads us to a system of linear equations, Ax=b. For larger simulations, this system is to large to be solved efficiently using a direct method, like Gauss elimination. However, we know that A is a very sparse matrix with at most 25 entries per row. Thus we see that various operations on this matrix can be done in O (n) time, like matrix-array multiplication and matrix-matrix sum- and minus-operations. Many numerical methods use only these operations to approximate the solution-vector x, and therefore such methods would solve the system very fast. This project explores the use of both the Conjugate Gradient (CG) method and the Jacobi method.

The filtering matrix S is applied in both methods in the same manner as used in [2].

4.4.1 Conjugate Gradient Method

In this project a preconditioned CG method is used - the same CG method that Baraff et. al. [2] uses. One problem with this method is that it will require many dependant operations per iteration of the system, and the GPU, with its pipelined structure, will have to revert to a serial execution, it has to finish with one operation before the next can start.

Listing (4.1) - Pseudo-code for the Conjugate Gradient method

```
CGSolver(A, x, dv)
     dv = z
     $\delta_0$ = filter(b)$^T$ * P * filter(b)
     r = filter(b)$^T$ - A * dv
     c = filter (P$^{-1}$ * r)
     $\delta_{new}$ = r$^T$ * c
     while ($\delta_{new}$ > $\varepsilon^2$ * $\delta_0$)
          q = filter (A * c)
          $\alpha$ = $\delta_{new}$ / (c$^T$ * q)
          dv = dv - $\alpha$ * c
          r = r - $\alpha$ * q
          s = P$^{-1}$ * r
          $\delta_{old}$ = $\delta_{new}$
          $\delta_{new}$ = r$^T$ * s
          c = filter(s + ($\delta_{new}$ / $\delta_{old}$) * c)
     end while
end CGSolver
```

4.4.2 Jacobi Method

The Jacobi method however can be reduced to one operation per iteration, as will be shown in the implementation chapter, and therefore can perform better than the CG method. On the other hand, one of the requirements for the Jacobi method is that the diagonal elements has to be significantly larger than all the other elements, and a solution is not guaranteed if the diagonal elements are smaller than the sum of all the values in one row. $M_{i,i} >> \sum_{i!=j} M_{i,j}$ must be true if a solution is to be guaranteed, but if the diagonal-elements are slightly larger than the sum of the other elements, a solution is often found [33]. This restraint will however limit the magnitude of the forces and range of the timesteps in the simulation, as these reduces the ratio between the diagonal element and the sum of the other elements in the row.

Listing (4.2) - Pseudo-code for the Jacobi method (Ax=b)

```
Jacobi(A, x, b)
    Choose an initial guess for x→x0.
    Invert the diagonal-elements of A → aInv
    For step = 1 to n or convergence is reached
        Multiply A (without the diagonal-elements) with x0 → x1
        X0 (i) = filter( aInv( i, i ) * ( b( i ) - x1( i ) ) )
    Next step
End Jacobi
```

4.5 Collision handling

To achieve a realistic simulation, collision handling is necessary. For this project only simple objects, like planes, were used as collision-objects.

Each particle is tested for collision during the update-process of the positions. When a collision is detected, the particle is constrained to move in the plane defined by the normal of the object it has collided with. This constraint is removed when the particle moves away from the object. The method from [2], which is the one this report is based on, includes these constraints directly in the equation used to represent the particle system.

Usually in cloth simulators, self-collisions are also handled, but to implement this on the GPU is a project in itself and is thus not implemented.

4.6 Summary

This project is based on the method from [2], using a particle system and integrating the differential equation using the implicit Euler method. We saw that the forces and derivatives have to be calculated and inserted into the linear system of equations. We saw how the internal forces of the cloth can be simplified by using only springs. The method from [2] also includes restraints. We saw how these can be used to restrain a particle from moving at all, and how to only allow it to move along a plane or a line. To avoid the system from becoming unsymmetrical, we introduce filtering functions to the numerical solvers.

5. Implementation

As the goal of this project is to compare a CPU versus a GPU implementation of the implicit Euler method from [2], we will look at both implementations. We will first look at some similarities between the implementations of the two methods. Then we will be looking at how the CPU version was implemented and then finally how the GPU version was implemented.

5.1 Common Grounds

Although, the two methods are implemented differently, we have to have the same matrices and arrays for both methods. If we look at the equations for the force and derivatives for a spring we see that we need two matrices and one array to hold the information of these. To calculate them, we need to know about the positions, the velocities, the weight, their restrictions and their rest-distance.

We will now look at which types of data structures we can use to represent these values:

- The forces can be represented as an array, since each particle has one 3D force.
- The positions can be represented as an array, since each particle has one 3D position.
- The velocities can be represented as an array, since each particle has an 3D velocity.
- Their weight can be represented as a diagonal-matrix, since each particle has one 3x3 matrix to represent their weight. The GPU version has only the diagonal-elements of the 3x3 matrix stored. The CPU version implements it as a sparse-matrix.
- $\frac{\partial F}{\partial x}$- and $\frac{\partial F}{\partial v}$-matrix can be represented as sparse-matrices, since they hold information regarding any two particles that are connected. The row- and column-number tells us which two particles this entry holds information about.
- The restraint-matrix S can be represented as a diagonal-matrix with 3x3 matrices, since it holds one 3x3 restraint-matrix per particle. This is the case for the GPU version. The CPU implements this matrix as a sparse-matrix.
- Their rest-distance are implemented directly. The CPU implements their rest-distance in the WE structure, whereas the GPU version uses their indexes to calculate the rest-distances.

Both versions perform the same task:

1. Calculate Forces and derivatives.
2. Calculate A and b of the linear system of equations.
3. Solve the linear system of equations, using either CG or Jacobi.
4. Update positions and velocities.
5. Go to 1.

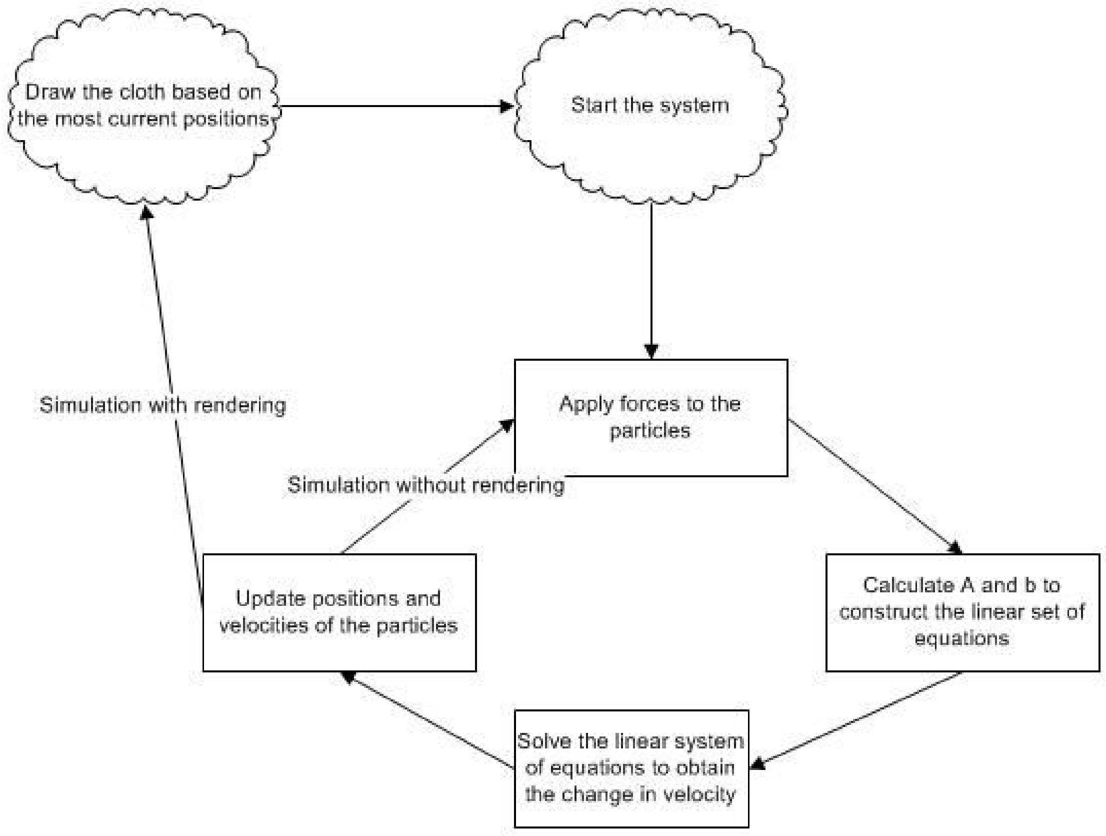

Draw the cloth based on the most current positions
Start the system
Simulation with rendering
Apply forces to the particles
Simulation without rendering
Update positions and velocities of the particles
Calculate A and b to construct the linear set of equations
Solve the linear system of equations to obtain the change in velocity

Figure 5.1 - The flow-diagram for the particle system.

5.2 CPU Version

We will now look at how the CPU version was implemented in this project. We will start off by looking at the winged edge data-structure that was used to represent the clothes before we look at how the other data-structures were implemented. Finally, some pseudo-code for the most important parts of the program will be presented.

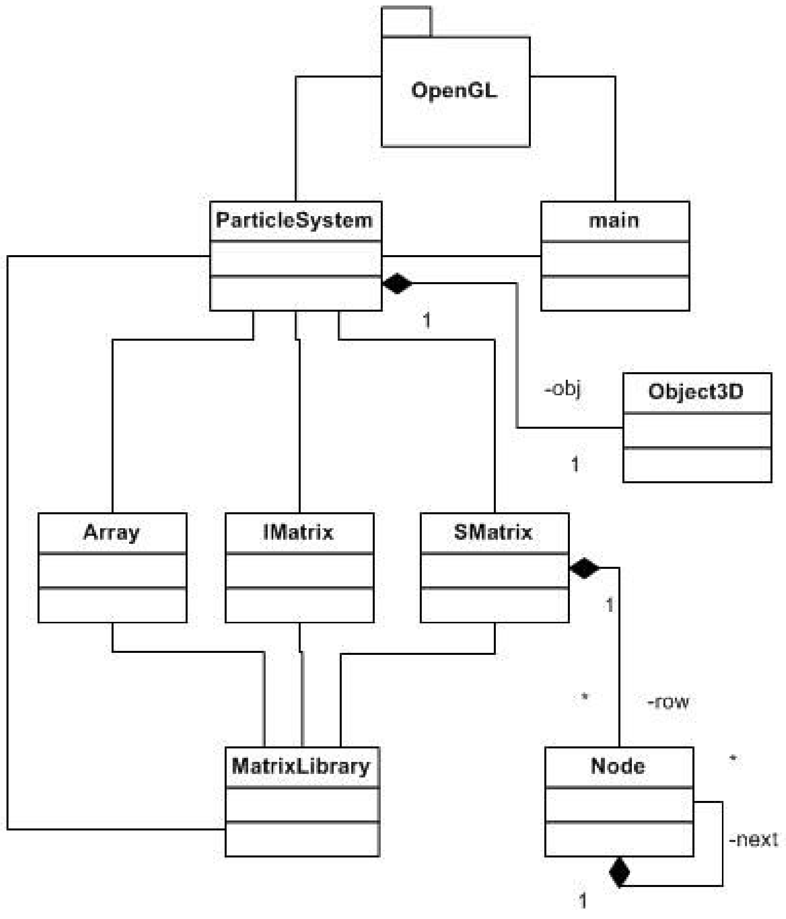


Figure 5.2 - The UML-digram for the CPU version.

5.2.1 Winged Edge Data Structure

We have seen how the Winged Edge (WE) data structure is defined, and it has been used in this project since the structure of cloth is based on particles and springs. The reason that this representation is a good structure for the purpose of representing clothes is that it has easy access to all of the information needed for calculating the external forces acting on the particles, and it can easily be extended to represent all the internal forces.

For this project an automatic winged-edge generator was implemented, and an explanation of how this was achieved is given below. This is a integer-pointer based implementation, meaning that there are three arrays for each object, and the objects (vertex, face, edge) themselves only has integer-values as pointers, these integer-values point to the object in the corresponding array. For a more detailed discussion, see [8].

This generator gets the triangles as input and it goes through seven steps to create a winged edge structure of them:

1. Construct triangles with all WE information, allow duplicate points and edges.
2. Construct the vertex-array from the triangles.
3. Find the duplicate vertices and construct a non-duplicate array of the vertices.
4. Change all the triangles so that their vertex-pointers now point to the right index in the non-duplicate vertex-array.
5. Construct the edge-array from the triangles.
6. Find the duplicate edges and construct a non-duplicate edge-array.
7. Change the edge-pointers in the triangles to point to the right index in the non-duplicate edge-array.

Step 3 and 6 is done by having a pointer-array which holds information for what an old integer-value would correspond to in the non-duplicate array.

The WE data structure used in this project also includes support for edges that are not a part of the triangles. This was done since we use springs to simulate the bend- and shear- forces.

5.2.2 Matrices

The array has been implemented straight-forward using only a simple class that supports *operator (int indeks)* and holds a single array of floating point values. The diagonal-matrix is implemented exactly the same way.

The $\frac{\partial F}{\partial x}$-, S, M and $\frac{\partial F}{\partial v}$-matrices are represented as nxn matrices with 3x3 matrices. Each of these 3x3 matrices holds the information about the derivative between two particles. Position (i, j) is the 3x3 matrix that represents the derivative of particle i differentiated with respect to particle j. Since these matrices are sparse, we can use the sparse-matrix class to represent them.

The sparse-matrix implementation is however not as simple as the array, the sparse-matrix-class holds an array of nodes, one for each row in the matrix. each node in the row-array is the first node of a sorted linked-list. When you insert a new element in a row it is inserted according to its column-value. This method was chosen because there are two operations this matrix has to support, sum/minus with another sparse-matrix and sparse-matrix multiplied by an array. Both these methods can easily be implemented by using the sorted linked lists in each row-entry. Had there been a need for an array multiplied by a sparse-matrix, xA, this structure would not be as efficient since we then would need a sorted list of each column, but this is not the case since xAx can be calculated by splitting the calculation in a certain order, $xAx = x(Ax)$ then it would only involve a matrix-array multiplication and an array-array dot-product.

5.2.3 Forces and derivatives

To calculate the forces and derivatives present in the particle system, we need only to go through the edge-array in the WE structure and add the force from each edge to the force-array, and the derivatives to the derivative-matrices. External forces are also added here.

Listing (5.1) – Calculating The forces and derivatives.

```
CalculateForces
     For all p ϵ P
          Add gravity- force to the particles force
     Next p
     For all t ϵ Faces
          Calculate wind and air- drag
          For all p ϵ t
               Add the calculated force to the particles force
          Next p
     Next t
     For all e ϵ edges
          Xi = e.p0
          Xj = e.p1

          Calculate spring- force and add to the force- array for
          particle i and j.

          Calculate $\frac{\partial F_i}{\partial x_j}$ and $\frac{\partial F_i}{\partial v_j}$ and add it to the derivative-
          matrices for particle i and j. Add their negatives to
                                         the diagonal-element of
                                         their row.
     Next edge
End CalculateForces
```

5.2.4 Solvers

With the implicit Euler method a set of linear equations is constructed for each iteration, luckily this system is sparse, and it can be expressed on the form Ax = b. This set of linear equations has to be solved each iteration to find the change in velocity.

Listing (5.2) – Calculating A and b and solving the system of linear equations.

```
IterateEulersBackward
     // Construct the matrices needed for the CG solver
     A = M – $\Delta$t * $\frac{\partial F}{\partial v}$ – $\Delta$t * $\Delta$t * $\frac{\partial F}{\partial x}$
     b = (F-$\Delta$t*$\frac{\partial F}{\partial x}$*v)

     dv = F    // The change in velocity is somewhat in the
               // region of the forces applied to the particles
     // Solve the equation A*dv = b, and get dv
     CGSolve (dv, A, b, S)
     Or jacobi (dv, A, b, S)
     p += v * $\Delta$t + 0.5 * dv * $\Delta$t * $\Delta$t
     v += dv*$\Delta$t
End IterateEulersBackward
```

Two different numerical solvers have been implemented in the CPU version, the conjugate gradient method and the Jacobi method. For the CPU version it was not needed to implement the Jacobi method since the Conjugate Gradient method performed quite well, but it was needed to compare the CPU version against the GPU version as we will see later on in this report.

The Conjugate Gradient (CG) method used in this project is the same as used in [2], it is the Preconditioned CG method. The implementation of the CG method is rather straightforward, we first calculate the preconditioners P and P1 and the rest of the method consists of using ordinary mathematical operations. We saw earlier in this report that $P_i = \frac{1}{A_{i,i}}, P1 = P_i^{-1} = A_{i,i}$. The rest of the algorithm was given in listing (4.1).

For the jacobi method we first need to calculate the inverse of the diagonal-elements of A, the rest is straight-forward from listing (4.2).

5.3 GPU Version
As we saw in the previous chapter, the first step of cloth simulation is to gather information about the forces and derivatives. When we have all that information we need only apply a numerical solver to the system of linear equations that remains.

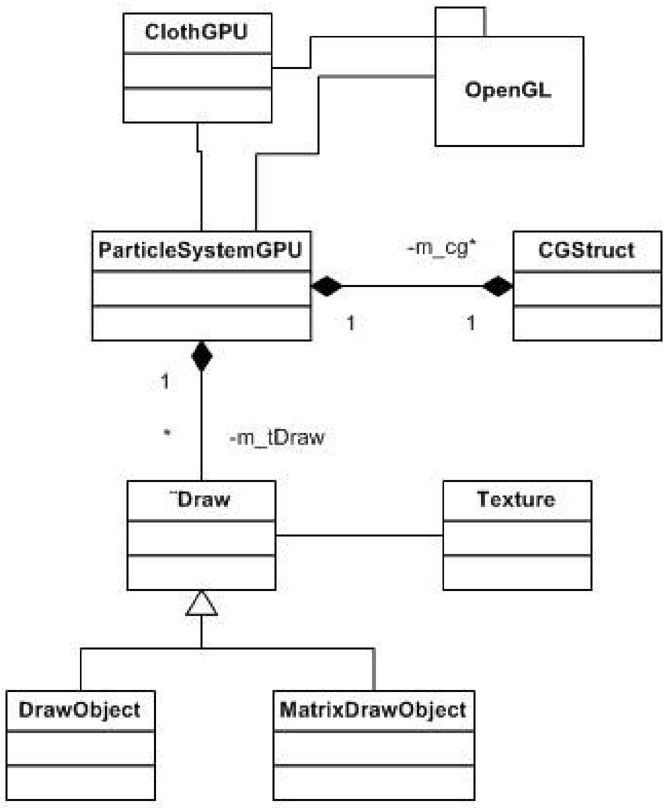


Figure 5.3 – The UML-diagram for the GPU version.

5.3.1 Introduction
We will now look at how a mathematical operation can pe executed on the GPU, first we will look at how the GPU works before we look at how the CPU has to initialize the GPU.

The GPU can perform a mathematical operation by a render-pass, meaning that we render an image over the draw-area we want the results to be written to. As we saw earlier in this report, the FBO has drawbuffers that textures can be attached to. We can then draw to these textures, thus we can perform mathematical operations by drawing to this texture. The GPU has many fragment-shaders, or fragment-processors if you will. These processors can output up to 16 floating point values, depending on the number of drawbuffers that are active.

However, the CPU has to initialize the GPU before it does what we want, the CPU has to tell which fragment- and vertex-program that is to be used, which textures are to be used and pass on different variables the GPU-program needs.

Listing 5.3 - a fragment-program that copies three textures simultaneously
fragment.cg
```
void main( uniform samplerRECT tex_1 : TEXUNIT0,
           uniform samplerRECT tex_2 : TEXUNIT1,
           uniform samplerRECT tex_3 : TEXUNIT2,
           float2 texCoord : TEXCOORD0,
           out float4 color_1 : COLOR0,
           out float4 color_2 : COLOR1,
           out float4 color_3 : COLOR2
     )
{
     color_1 = texRECT(tex_1, texCoord);
     color_2 = texRECT(tex_2, texCoord);
     color_3 = texRECT(tex_3, texCoord);
}
```

Listing 5.4 - a C++ program that uses the fragment program from `listing 5.3.`
`program.cpp, using the program for this project`
```
GLuint fb,
       buf[3] = {
          GL_COLOR_ATTACHMENT0_EXT,
          GL_COLOR_ATTACHMENT1_EXT,
          GL_COLOR_ATTACHMENT2_EXT
       };

Texture original_1(Texture::FLOAT, 10, 10, imaginary_data_1),
        original_2(Texture::FLOAT, 10, 10, imaginary_data_2),
        original_3(Texture::FLOAT, 10, 10, imaginary_data_3);

Texture copy_1,
        copy_2,
        copy_3;

// Get a FBO handle from openGL
glGenFramebuffersEXT(1, &fb);

// Compile and initialize our fragment-program
int fraParNum = 3;
char **fraParName = { "tex_1", "tex_2", "tex_3" };
CGStruct fragment(NULL, "fragment.cg", NULL, 0, fraParName,
                  fraParNum, NULL, "main");

// Initialize the draw-object
DrawObject *draw = new MatrixDrawObject(copy_1, copy_2, copy_3,
                                  fb, buf[0], buf[1], buf[2]);
```

```
// Activate drawing to the three textures
draw->EnableDraw();

// Set up the fragment-program
fragment.loadFragment();
copy_1.BindTexture(GL_TEXTURE0_ARB, fragment.fraPar[0]);
copy_2.BindTexture(GL_TEXTURE0_ARB, fragment.fraPar[1]);
copy_3.BindTexture(GL_TEXTURE0_ARB, fragment.fraPar[2]);

// Draw a quad covering the entire screen
drawQuad();


// Unloading the fragment-program
framgent.unloadFragment();

// Now:
// copy_1 == original_1
// copy_2 == original_2
// copy_3 == original_3
```

This CPU code is very much the same for any GPU program that we want to execute. The code in the fragment-program only has to execute the work that affects the pixel it is assigned in the result-set.

Listing 5.5 - CPU code that does the same as the fragment-program above:

```
for(int i = 0; i < width; i++)
{
     for(int j = 0; j < height; j++)
     {
          copy_1(i, j) = original_1(i, j);
          copy_2(i, j) = original_2(i, j);
          copy_3(i, j) = original_3(i, j);
     }
}
```

Thus, we see that drawing an image to perform a GPU operation is the same as a double for-loop. Each pixel performs the work inside the double-loop that has its coordinates. This can be different in real-life, since we might map an array to a 2D texture, but in essence this is what occurs.

5.3.2 Data Representation

Some simplifications are made in this project, mostly to avoid spending too much time on implementing the underlying structures than on the cloth simulator itself. We could have implemented a Winged Edge structure similar to the one we saw in the CPU-implementation, but for simplicity the particles were represented using rectangular textures, one for positions and one for velocities. In table 5.1 and figure 5.4 we see how a texture for the positions looks like and how the corresponding grid would look like when drawn.

(-1, -1)	(0, -1)	(1, -1)
(-1, 0)	(0, 0)	(1, 0)
(-1, 1)	(0, 1)	(1, 1)

Table 5.1 - An example of a texture with positions, each entry has a xy-coordinate.

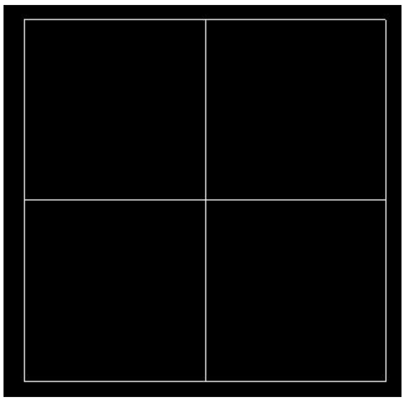


Figure 5.4 - This is the grid the texture from table 5.1 represents.

The rest-distances were implemented by using their indexes, distance between $P_{i,j}$ and $P_{i+1,j+2}$ equals $\sqrt{((i+1)-(i))^2+((j+2)-(j))^2}$.

There are three different types of storage that we need; sparse-matrix, diagonal-matrix and array.

The array is implemented as a rectangular texture where each pixel corresponds to a particle. A particle's velocity has the same coordinate in the velocities-texture as the particle's position has in the position-texture. Though an array is 1-Dimensional (1D), the texture representing it need not be, the texture is 2-dimensional (2D) and each particle is given a 2D coordinate in this texture. A 1D address for the particle, i, is transformed to the 2D coordinate with $\begin{aligned} x &= i\%width \\ y &= \frac{i}{width} \end{aligned}$ using integer division, like we see in figure 5.5.

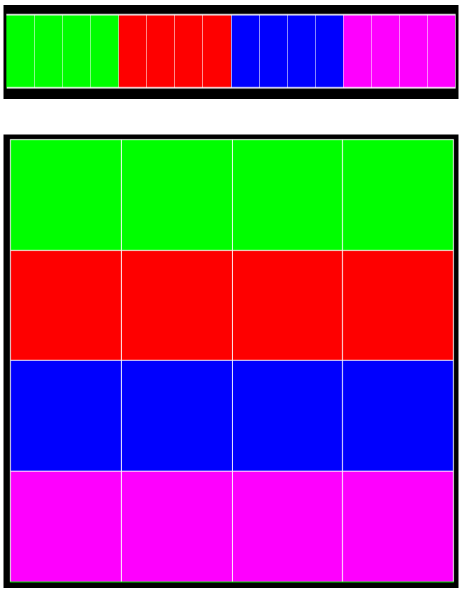


Figure 5.5 - The array is transformed, by using 1D to 2D mapping, into a 2D texture.

The diagonal-matrix is constructed the same way as the array; the diagonal-element corresponding to a vector-element is placed in the same position in its texture as the vector-element is placed. Matrix (i, i) has the same position in its diagonal-texture as vector (i) has in its texture, as shown in figure 5.6 and 5.7.

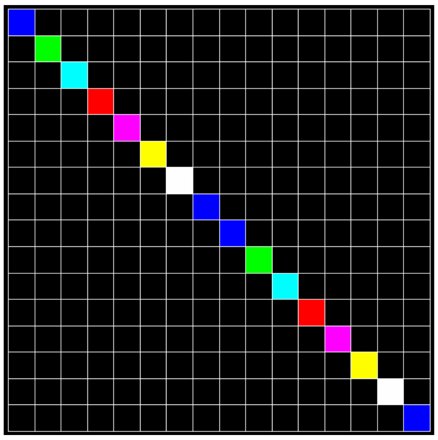


Figure 5.6 - This is an ordinary matrix where the diagonal-elements are given a colour.

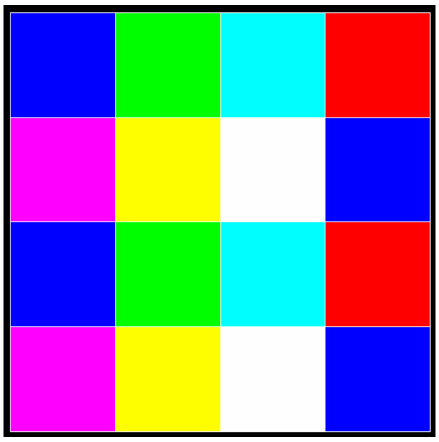


Figure 5.7 - This is the 2D texture representing the diagonal-elements of a matrix, the colour of the pixels corresponds to figure 5.4, and you can see that each diagonal-element is aligned with its corresponding element in the 2D texture representing a vector.

To explain how the sparse-matrix is implemented it will be helpful to understand how an ordinary matrix is represented first. Each element in a matrix, in this project, is a 3x3 matrix, we therefore need to store 9 values for each pixel in our calculations later on. One way to implement this is to use one texture and use three consecutive rows to represent one row with 3x3 matrices, as shown in figure 5.8. This however, means complex programs when one wants to use the matrix since we always have to consider that to fetch one element, we need to fetch data from three different rows. This implementation is also inefficient when we want to construct the matrix, this is because each 3x3 matrix requires the same series of calculations, but as we saw earlier in this report, we can only write to one position at a time, and thus we need to calculate the same series of calculations three times. However if we represent the matrix with three textures, as seen in figure 5.10, we can use each of them to represent their own rows of the 3x3 matrices, e.g. each pixel of TEXTURE0 represents the entries of row0 of their corresponding 3x3 matrices, TEXTURE1 represents row1, TEXTURE2 represents row2. This is illustrated in figure 5.9.

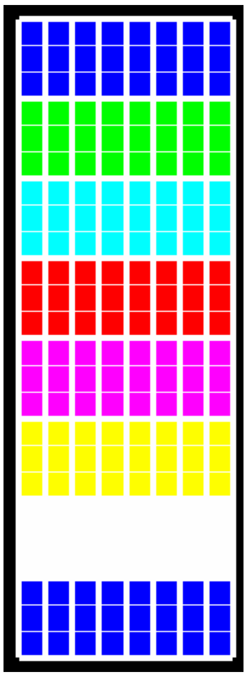


Figure 5.8 - This figure is a 2D texture representing a matrix of 3x3 matrices. Each pixel horizontally contains 3 values. Heavy lines indicate which three pixels that makes one 3x3 matrix.

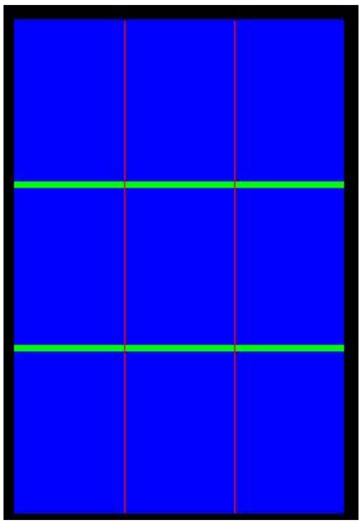


Figure 5.9 - This figure shows us a sub-sample of the texture from figure 5.6, each green line separates rows in the texture, each red line separates the RGB values in one pixel. Each block can be used to represent one floating point value.

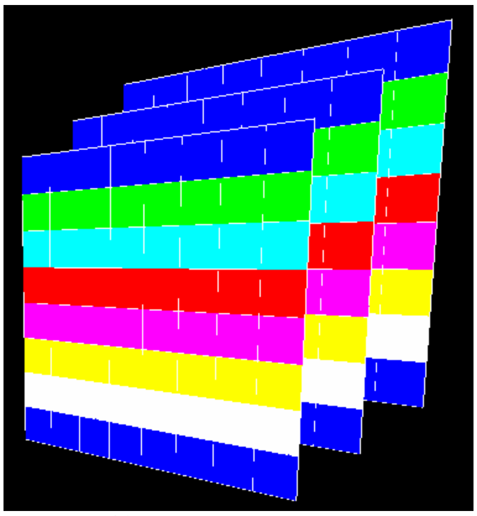


Figure 5.10 - These three textures represents the same 3x3 matrices as figure 5.8, instead of using three lines in the same texture to represent them, three different textures hold information about the different rows in the 3x3 matrices.

The derivative-matrices are sparse-matrices, and hold the derivative between any two particles, $\frac{\partial F}{\partial x}_{i,j}$ is the derivative of particle i differentiated by particle j, but since each particle is connected to only 24 other particles which are a fixed distance from particle i, as we saw in figure 4.1. Thus, we only need 25 columns in this texture as shown in figure 5.12. Each x-coordinate of a pixel in the sparse-matrix is applied equation (5.1) and (5.2) to obtain its x-coordinate if it were an ordinary matrix, i in the equation is derived directly from the y-coordinate since the row number is the same for an ordinary matrix and a sparse-matrix.

E.g. sparse-matrix (y, x), y is used directly as the i-coordinate, x is transformed to the j-coordinate using equation (5.3), and we know that this pixel holds the information of the derivative between particle i and j.

yo\x0	-2	-1	0	1	2
-2	(x-2, y-2)	(x-1, y-2)	(x, y-2)	(x+1, y-2)	(x+2, y-2)
-1	(x-2, y-1)	(x-1, y-1)	(x, y-1)	(x+1, y-1)	(x+2, y-1)
0	(x-2, y)	(x-1, y)	(x, y)	(x+1, y)	(x+2, y)
1	(x-2, y+1)	(x-1, y+1)	(x, y+1)	(x+1, y+1)	(x+2, y+1)
2	(x-2, y+2)	(x-1, y+2)	(x, y+2)	(x+1, y+2)	(x+2, y+2)

Table 5.2 - This table represents which particles each particle (x, y) is connected to, it is not connected to itself, this position is used to represent the diagonal-element in the matrix. In the sparse-matrix, the x-coordinate from the sparse-matrix is transformed by equation (5.1) and (5.2) to simulate this table.

As we saw in figure 4.1, each particle is connected to the 24 closest (relative to coordinates in the table, not real positions) particles, instead of using table 5.2 to find the offsets from the particle, we can use equation (5.1) and (5.2) instead, since this equation will give us the same offset-coordinates.

$$yo = \frac{x}{5} - 2 \tag{5.1}$$
$$xo = x\%5 - 2 \tag{5.2}$$
$$j = i + yo * width + xo \tag{5.3}$$

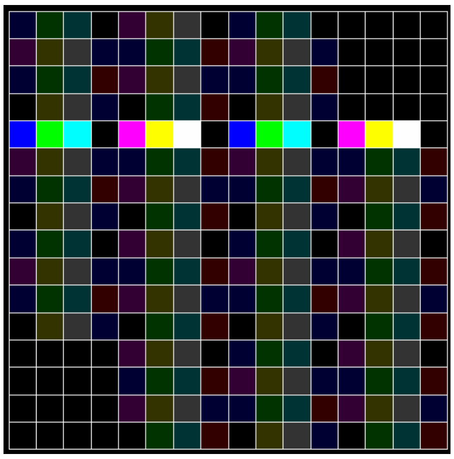


Figure 5.11 - This is an ordinary matrix, line 5 is highlighted.

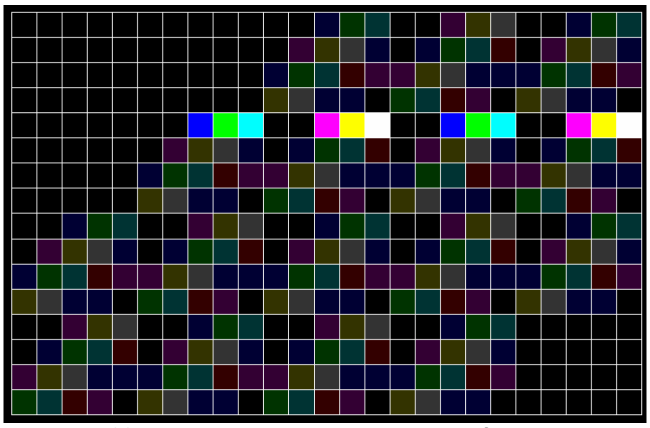


Figure 5.12 - This is my implementation of a sparse matrix, this matrix is obtained using the shifting-formula described, using the inverse of equation (5.3), on figure 5.11. This seams to expand the matrix in this example, but the advantage is that this matrix is always the same width, and thus for larger data-sets it will both save space and perform better than the texture in figure 5.11.

5.3.3 Forces and Derivatives

We will now look at how the GPU can use these data-structures to find F, $\frac{\partial F}{\partial x}$ and $\frac{\partial F}{\partial v}$ in the simulation.

Calculating the forces is done straightforward, each pixel of F represent the force acting on one particle and thus we only sum up the forces contributed by the springs connecting this particle to the others.

Listing 5.6 - Fragment-program for calculating the forces for each particle, one pixel calculates the force of one particle:

```
For each pixel
     P = pixel.position
     Sum up all the forces of the 24 particles around P and
          return answer
End pixel
```

However, the derivatives are slightly more difficult to calculate since we have to use the x- and y-coordinate to calculate j. i is trivially computed from the y-coordinate as we saw earlier. We then have to convert the 1D coordinates i and j to 2D coordinates so that we can fetch the right values from the arrays.

Listing 5.7 - Fragment-program for CalculateDFDX

```
For Each pixel
     P = pixel.coordinate2D

     // Find the 2D-coordinate for the particle that the
     // y-coordinate in the sparse-matrix represents
     xj.x = p.y % width
     xj.y = p.y / width

     // find the 2D-coordinate for the particle that the
     // x-coordinate in the sparse-matrix represents
     xOffset = p.x % 5 - 2
     yOffset = p.x / 5 - 2
     xi.x = xj.x + xOffset
     xi.y = xj.y + yOffset

     if (xi is within the array)
          calculate $\frac{\partial F}{\partial x}$, or $\frac{\partial F}{\partial v}$ for the $\frac{\partial F}{\partial v}$ program, and return
          answer
     end if
end pixel
```

After the forces and derivatives are calculated, we are ready to construct A and b to from the linear system of equations, Ax=b. A was calculated in a very simple manner, since there are no dependencies in equation (5.4), we only need to do one pass. Only a single fragment-program was needed to calculate A.

Although it is possible to reduce the calculation of b to one pass, b was calculated with the matrix-library which will be presented later in this chapter. The calculation of b could have been implemented as a multiplication of $\frac{\partial F}{\partial x}$ and v where F and Δt would have been applied to the result before it was written.

$$A = M - \Delta t * \frac{\partial F}{\partial v} - \Delta t^2 * \frac{\partial F}{\partial x} \tag{5.4}$$

$$b = F + \Delta t * \frac{\partial F}{\partial x} * v \tag{5.5}$$

Listing 5.8 - Fragment-program for calculating A:
```
For each pixel
     Result = 0
     If (this pixel is a diagonal-element)
          Add the mass to the result
     End if
     Subtract ($\Delta$t * $\frac{\partial F}{\partial v}$ + $\Delta$t^2*$\frac{\partial F}{\partial x}$) from the result and return
     answer

End pixel
```

5.3.4 Solvers

We are now ready to start solving the linear system of equations. In this project two different numerical solvers has been implemented, the Conjugate Gradient method and the Jacobi method. We saw how these methods work in chapter 4, now it is time to see how each of these methods has been implemented on the GPU.

5.3.4.1 The Conjugate Gradient Method

For the Conjugate Gradient (CG) method we need a matrix-library, as we can see from listing (4.1) we need 4 different operations, sparse matrix-vector multiplication, diagonal matrix-multiplication, vector-vector multiplication (dot-product) and vector-vector sum and difference.

The diagonal matrix-vector multiplication and vector-vector-sum difference are implemented straightforward, the operation in each pixel uses data from the same position in the two other textures and writes to the same coordinate.

The sparse matrix-vector multiplication is a bit more difficult. The result is a vector, and in our fragment-program we get the position to write to, let us call this 1D-coordinate i. In this pixel we calculate row i from the sparse-matrix multiplied by the entire array. For each element in the row we need to calculate a 1D-coordinate, j, in the array by using the inverse of equation (5.3), then we multiply the 3x3 matrix from the sparse-matrix in position (y, x) with the 3x1 vertex in the array in position (j/w, j%w).

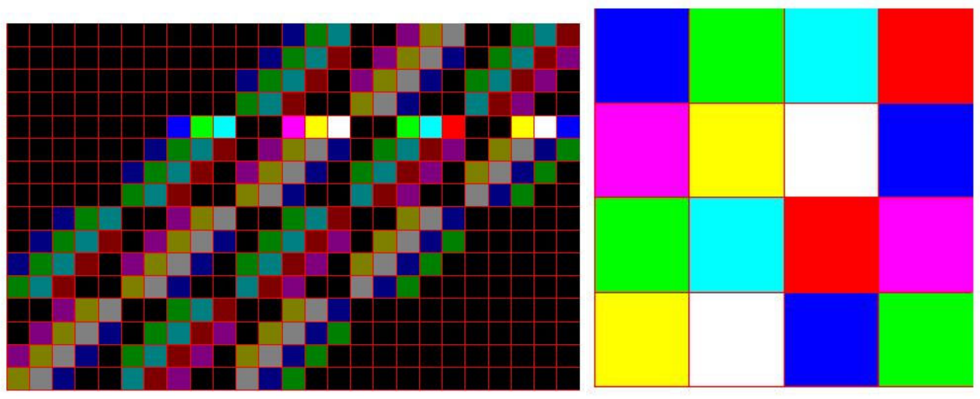


Figure 5.13 - (a) is the sparse-matrix. (b) is the array. This figure shows us the multiplying process between a sparse-matrix and an array. The colour of the pixels corresponds between the two images if you think of the coloured pixels in one row in (a) as an array to be multiplied with the array from (b). Similar colours in (b) do not mean they are the same pixel, the same goes for each row in (a).

We will now illustrate this multiplication process. As we can see from figure 5.13, row 5 is highlighted. This yields i=5 and gives us the 2D coordinate (2, 1) (4 elements per row, so the fifth element is the first in the second row) in the array. So for each entry in row 5 in the sparse-matrix, we have to calculate an offset from (2, 1) before we multiply smatrix(y, x) with array(ax, ay). To find the position (ax, ay), we need to apply equation (5.1) and (5.2) to the x-coordinate of the sparse-matrix, this gives us (ax, ay)=(2+x%5-2, 1+x/5-2).

Listing 5.9 - Fragment-program for the sparse-matrix array multiplication:
```
For each pixel
     P = pixel.position
     // transform the 2D-array-coordinate to find the row in the
     // sparse-matrix to multiply with the array.
     Y = p.y * width + p.x

     Float3 res = (0, 0, 0);

     For x = 0..24
          // calculate the 2D coordinate that
          // sparse-matrix (p.y, x) should be
          // multiplied by
          ySmall = p.y + x / 5 - 2
          xSmall = p.x + x % 5 - 2

          if ( xSmall and ySmall both are
                    within the borders of the array)
               multiply sparse-matrix (y, x)
                    with array (ySmall, xSmall) and add the
                    answer to res
          end if
     end for
end pixel
```

The vector-vector multiplication is simpler, at least in theory. First, a dot-product between corresponding pixels are performed, and the result is stored in an intermediate texture. Then, we need to row-reduce by reducing the width and height of the output by two each iteration until we have a 1x1 texture representing the value. Each pixel-operation sums up four values from the previous result. To avoid the program to be dependent on that the size of the initial array is $2^n x 2^n$, the last row of the output also sums up the last row, if the input for this iteration is not dividable by two. The same goes for the last column.

Listing 5.10 - Example of a dot product between the two textures A and B, P0 and P1 are the intermediate textures. This simple example does not take into account odd-numbered textures, or textures with different width and height.

```
P0(I, j) = A(I, j) * B(I, j)
Until i == 1 and j == 1
     Run fragment-program to calculate
          P1(I, j)) = P0(2*I, 2*j) + P0(2*I+1, 2*j) + P0(2*i+1,
     2*j+1) + P0(2*i, 2*j+1)
     End fragment-program
     I /= 2
     J /= 2
loop
```

5.3.4.2 The Jacobi Method

The Jacobi method is much simpler then the CG method. All the calculations in each iteration can be reduced to one render-pass, and the implementation is straightforward.

Listing 5.11 - Fragment-program for inverting the diagonal-elements of A:
The result-set is a diagonal-matrix, with three textures so that it can hold 9 values (3x3 matrix).

```
For each pixel
     P = pixel.position
     // calculate the row this pixel should
     // calculate the inverse of.
     Y = p.y * width + p.x
     // 13 has the offset (0, 0) and is the diagonal
     Invert the 3x3 matrix from A (Y, 13) And return the answer.
End pixel
```

Listing (5.9) - Fragment-program for one iteration of the jacobi-method:

```
For each pixel
     P = pixel.position
     Y = P.y
     Float3 res = (0, 0, 0)
     For x = 0..24
          // calculate the 2D coordinate that
          // sparse-matrix (p.y, x) should be
          // multiplied by
          ySmall = x / 5 - 2
          xSmall = y % 5 - 2

          if ( xSmall and ySmall both are
                    within the borders of the array)
               multiply sparse-matrix (p.y, x) with
                    array(ySmall, xSmall) and add the
                    answer to res
          end if
     end for

     multiply res with the 3x3 inverse matrix from A(y, 13).

     Filter res by multiplying the corresponding 3x3 filtering
          matrix from S and return Answer.
End pixel
```

5.4 Summary
Both versions of the method needs the same matrices and arrays, they also perform the same task; calculating forces, constructing the linear set of equations, solving it and update positions and velocities. How this is done on the CPU and GPU are however very different. Whereas the CPU version calculates the matrix-operations by using ordinary matrix-library operations, the GPU is set up by the CPU. The CPU instructs the GPU which fragment- and vertex-program are to be used and passes along information about which textures it should use in its calculations. The GPU then renders to a texture using this information and produces a result from an operation. We also saw how the textures are defined with regards to arrays, sparse-matrices and diagonal-matrices for the GPU version, and how these data-structures were defined on the CPU.

6 Results, Conclusion, and Future Work

We have throughout this report seen how the implicit particle system from [2] has been implemented, both on the CPU and the GPU. It is now time to see how the implemented versions perform. We will first look at how fast the Jacobi method and the CG method performed on the CPU and the GPU. We will then look at what might have caused these results. Then, we will look at the visual results from these methods, before we finally look at what could have been done better, as well as possible future work.

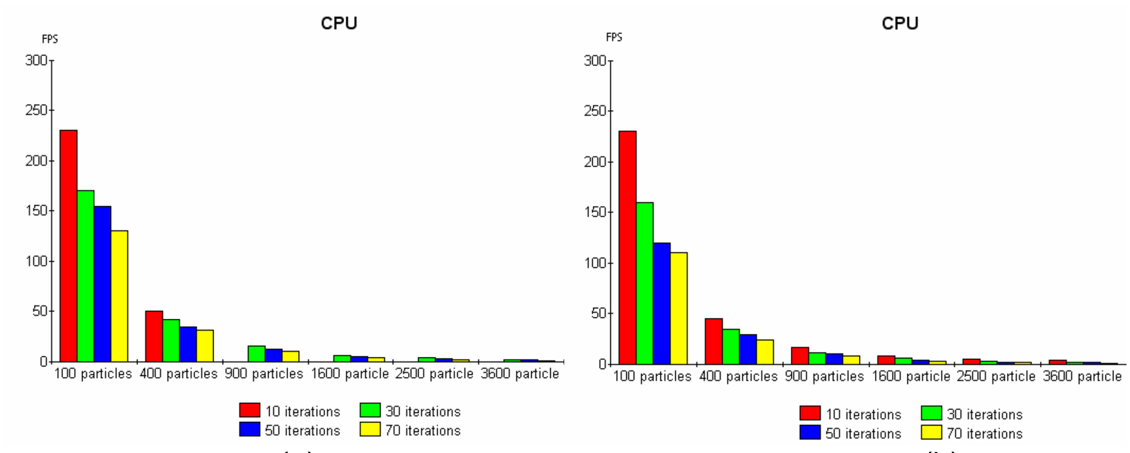
6.1 Results

All tests have been run on the same computer, a laptop with 2.0GHz Pentium-M processor, 1GB RAM and a nVidia GeForce 6800 Go graphics card with 256 MB graphics memory.

There are two basic parameters we can tune in the implemented methods with regards to performance:

1. The number of iterations that we will let the numerical methods use to converge to the solution vector. This value will range from 10 to 70 iterations, although 10 iterations might not be sufficient to let the method converge close enough to the solution vector, it has been included in the tests. 30 iterations usually give us a very nice simulation.
2. The number of particles in the simulation. This value will range from 100 to 3600 particles, 4096 is the maximum number of particles for the GPU version since textures can only be 4096 in any dimension, but this range gives us a good indication of how the methods perform.

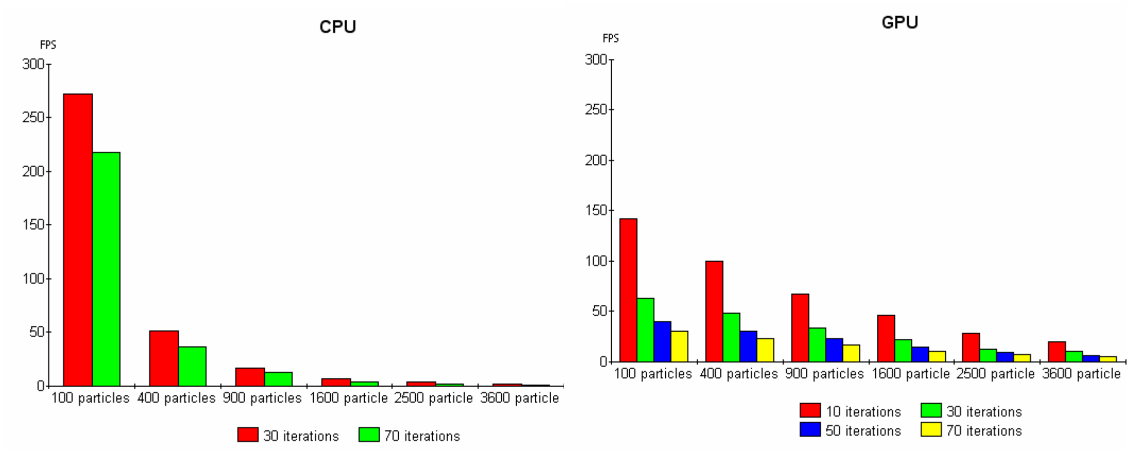
The histograms that will be shown specify the number of particles on the x-axis and the frame-rate on the y-axis, the different blocks shows us different number of iterations for the given number of particles. The tables with numbers for the histograms are given in the appendix. In the CPU version with jacobi, there are no values for 10 iterations for more than 900 particles, this is because the simulation failed with these settings, the GPU version of Jacobi can however simulate these sizes with 10 iterations, although sometimes it will fail too, as we will discuss later in this chapter.



(a) (b)

Figure 6.1 – Both these results are from the CPU version, and are rendered with light and texture. (a) is with Jacobi as the numerical solver. (b) is with Conjugate Gradient as the numerical solver.

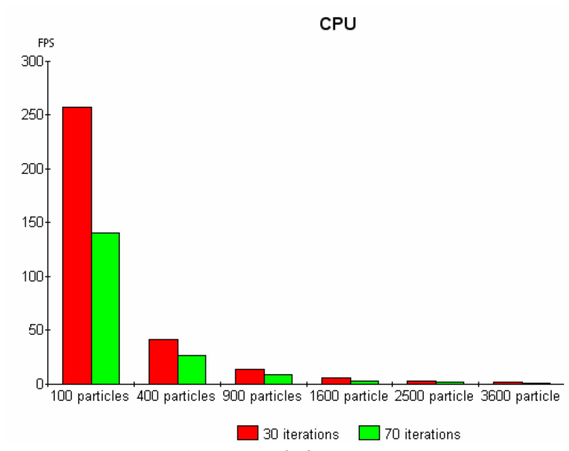
As we can see from figure 6.1, the two different CPU versions perform equally well for a given set of parameters. This means that both methods involve the same amount of work. If we study figure 6.1 closely, we can see that the Jacobi method is slightly faster than the CG method.

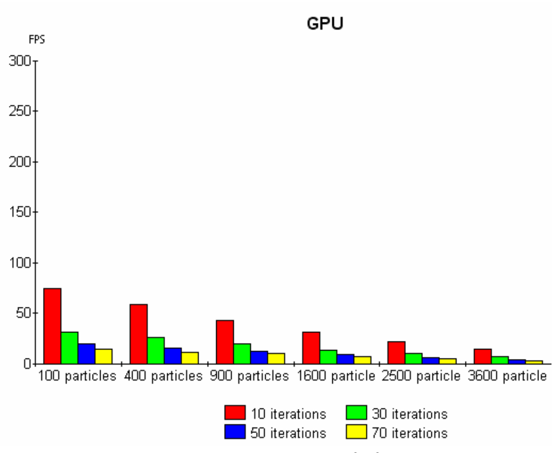


(a) (b)

Figure 6.2 - Both these results use the Jacobi method as their numerical solver, and are tested without rendering. (a) is the results from the CPU version and (b) is the results from the GPU version.

As we can see from figure 6.2, the CPU version greatly outperforms the GPU version for smaller number of particles. When the number of particles increases, the GPU version outperforms the CPU version. Also, we see that the GPU version performs significantly worse for larger number of iterations than the CPU version. There is almost no difference between 30 and 70 iterations for the CPU version, but for the GPU version there is about a 50% reduce in frame-rate just by changing the number of iterations for convergence from 30 to 70.





(a) (b)

Figure 6.3 – Both these results use the Conjugate Gradient method as their numerical solver, and are tested without rendering. (a) is the result from the CPU version and (b) is the result from the GPU version.

As we can see from figure 6.3, we get the same results as between the CPU and GPU version with the CG method as we did with the Jacobi method. For smaller numbers of particles, the CPU version outperforms the GPU version, but for larger numbers of particles the GPU version outperforms the CPU version. As with the last test, the GPU versions frame-rate is reduced significantly by increasing the number of iterations for convergence, whereas the CPU version almost has no decrease in performance. However the penalty of using more iterations for convergence is more noticeable in the CG method, since there are more operations involved in this method compared to the Jacobi method.

If we compare figure 6.1, 6.2 (a) and 6.3 (a) we see that there is a significant increase in frame-rate without rendering compared to with rendering for smaller numbers of particles. For larger numbers of particles the difference is insignificant. The GPU versions were not implemented with lights and texture, therefore there are no comparisons between rendered and not-rendered simulations. Drawing the particle system with wire-frame, as was done for the GPU versions, yielded the same frame-rate as without rendering (typically 2-3 fps difference for smaller systems).

If we compare the two CPU versions from figure 6.2 (a) and 6.3 (a), we see that they perform equally well, this indicates that the two methods involve about the same amount of work. However, when we look at the difference between the Jacobi method and the CG method, we see that they perform rather differently on the GPU than on the CPU. The Jacobi method is now about twice as fast as the CG method for small numbers of particles, for larger numbers of particles it performs about 25-50% better. Although both these methods do the same amount of work they do not perform equally on the GPU. The reason for this is, as mentioned earlier, that the CG method uses several rendering-passes per iteration of the convergence. The Jacobi method however can be reduced to one rendering-pass per iteration. And as we can see if we compare the results from the GPU tests, 1600 particles can be simulated with 30 iterations with the same fps as 400 particles with 70 iterations. If we cross-reference these values with the CPU version, we see that this means that there is a large penalty of iterating with the GPU. It is 4-5 times slower to use 70 iterations compared to 10 iterations with the GPU, and it is under two times slower with the same increase in iterations for the CPU. This hidden workload is in general that the GPU has to initialize the FBO and draw-buffers before we can render to them.

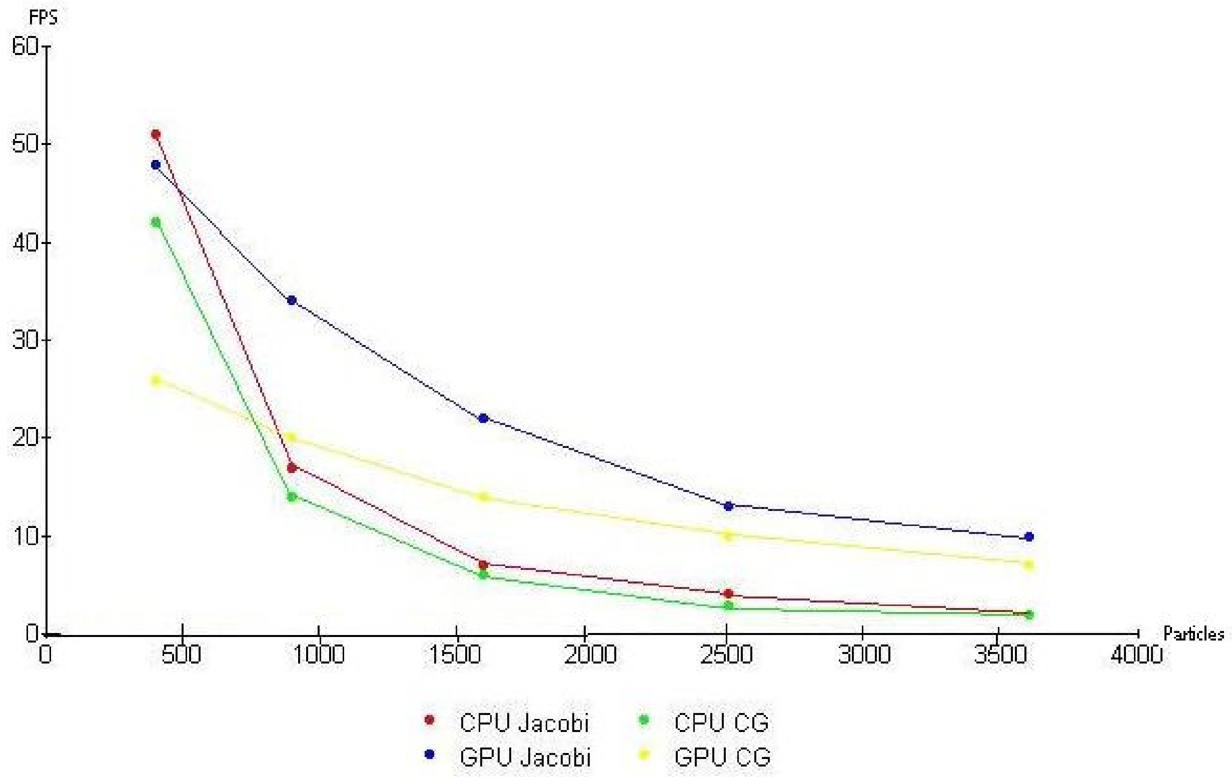


Figure 6.4 - This figure shows us how the different methods compare. The number of particles is shown on the x-axis while the frame-rate is shown on the y-axis. The frame-rate for 100 particles has not been included so that we can see the results for larger numbers of particles better. All the values in this graph have used 30 iterations for convergence.

As we can see from figure 6.2-6.4, the CPU version outperforms the GPU version for smaller number of particles, but as the system gets larger, the overhead in the GPU version gets less significant. Thus, the GPU version outperforms the CPU version when there are more than 400 particles, as seen in figure 6.4. However, the Jacobi method is more unstable than the CG method, and if we try time-steps larger than 0.01s we will have to reduce the spring-constants for it to converge to the correct solution, it does not help to increase the number of iterations for convergence as it does for the CG method. But for reasonable forces and time step of 0.01s the Jacobi method and CG method solves the system equally well and thus the Jacobi method is the best solver to use on the GPU version.

Although the simulation with Jacobi on the GPU performs well, there are some artefacts in the simulation, for simulations of more than 1600 particles, the particles just vanish.

This Houdini-act does not seem to be an oscillation error, which is often the reason why the particles disappear. The particles move realistically for a long time, 2-3s (2-300 iterations); before they in one frame just disappear. This seems to be caused by inversion of the diagonal-elements of A, one or more 3x3 matrices are singular during this frame and their inverses have only elements NaN. Thus, all the particles will get this value and disappear. Often it helps to adjust the number of iterations for convergence, since this will lead to a slightly better solution vector all the way. Thus, the particles will not be in the same position as they had when there were found singular matrices.

We set out on a mission to implement a faster version of the implicit particle system from [2], have we succeeded on this task?

We saw that the GPU version allows us to have 20fps for 3600 particles whereas the CPU version only gave us 2fps. Although, this is not a real-time frame-rate, it is sufficient to give the user an interactive simulation, meaning that there is no problem in exploring the cloth while it is simulated, even for such a large particle system. This is a rather good result for this project, and still there is work to be done that can make the system larger, more adjustable and faster.

6.2 Pictures

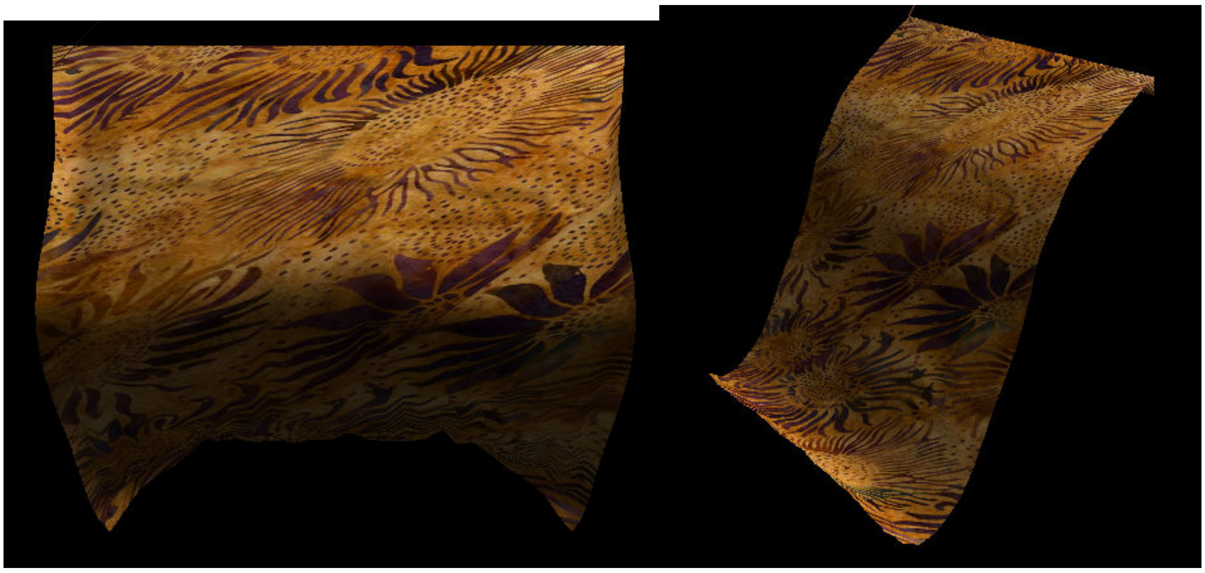


Figure 6.5 – Screenshots from the CPU version. These pictures were taken during simulation of a 30x30 grid of particles.

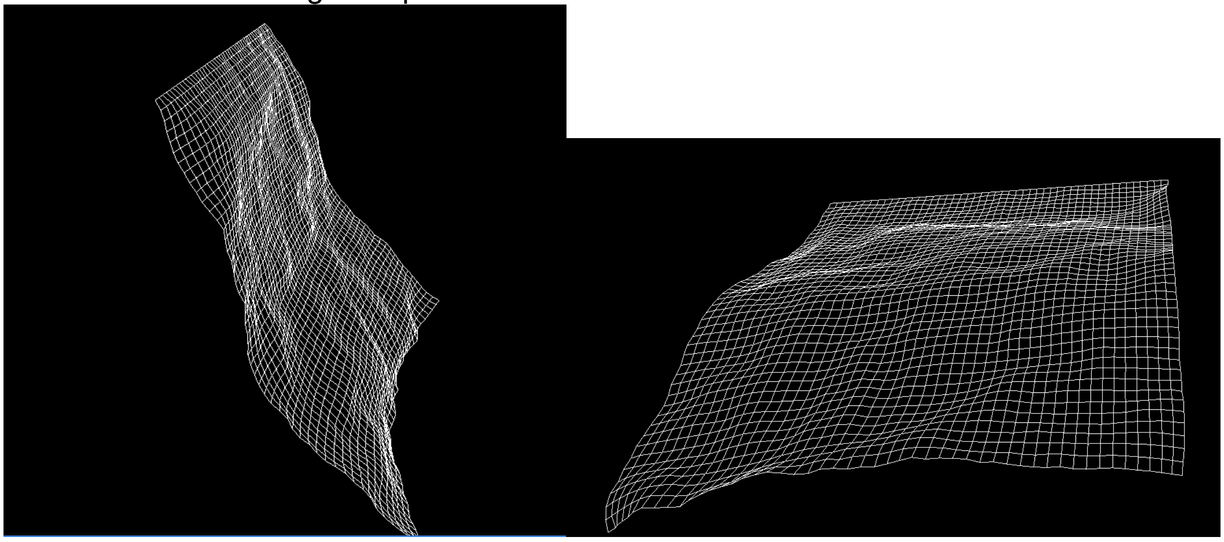


Figure 6.6 – Screenshots from the GPU version. These pictures were taken during simulation of a 50x50 grid of particles.

6.3 Future Work

We have now seen how this method performed, but what could have been done to make it better? As with any project there are certain features that we do not have time to implement, or did not think of before it was too late.

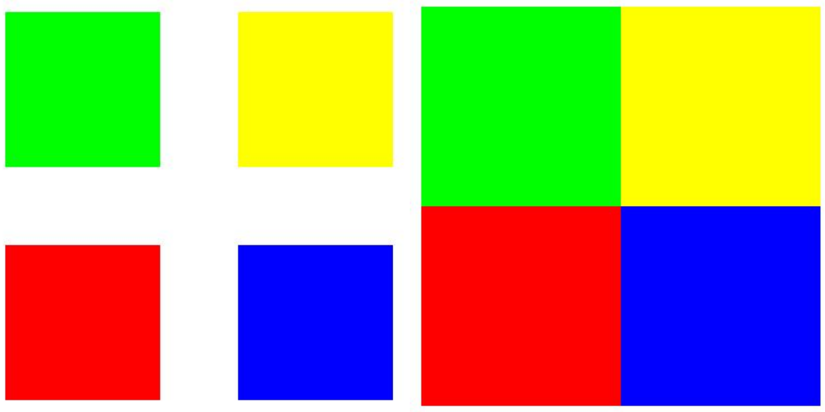
We saw in chapter 6.1 that the GPU version performed better and better, compared against the CPU version, as the number of particles grew in numbers. However, the GPU version is currently limited to 4096 particles due to texture-dimension restrictions on the GPU. The restricting textures are the sparse-matrices, they have the same height as the number of particles, but since they have only 25 entries per particle, thus 4096 particles use texture[0..24, 0..4095]. This means that texture[25..4095, 0..4095] is unused, over 99% of the texture is unused. Thus, if we let particle 4096 through 8191 lie in texture[25..49, 0..4095] and so on, we could have a particle system with support for 667 648 particles, the other textures can already hold this many particles, or can easily be converted to hold them.

We also mentioned the fact that no self-collision was implemented. There is no support for complex collision-objects either. A data-structure like the WE data-structure could have been implemented. This would give us a much better freedom to construct complex clothes to simulate, instead of a rectangular sheet as is the case now. We could also implement a data-structure, like the oct-tree [31] to include collision detection, both self-collisions and collisions with complex objects. Both these structures would have consumed too much of the time scope of this project, but they would make this cloth simulator much more adjustable and allow for more complex and realistically simulations.

It is also possible to make the program more realistic by using the force-functions from Baraff et. Al [2] instead of only using springs. This would also give the user a much more adjustable cloth simulator, and clothes could have been realistically simulated. Thus, instead of just looking like a cloth, the simulated cloth could have the behaviour of for instance silk.

All of these improvements are general improvements to the GPU version that can make the program better. However, there are also some improvements that can optimize the performance of the GPU version described throughout this report.

We saw that the performance of the GPU version with the CG was poor, we also discussed the reason for this, there are too many rendering-passes per iteration of convergence. [36] explains the use of a ATLAS texture, a large texture that simulates many textures, we only have to map the view-port to the correct location in the ATLAS to access a certain texture as seen in figure 7.1. This could have reduced the workload from the iterations somewhat since we do not have to set up a new FBO and drawbuffer, although the glViewport function also has a hidden workload. Simple tests comparing the time to draw 100 times to 10 different textures in a loop and drawing 100 times to 10 different places in the ATLAS texture showed a slight improvement, typically from 80 to 90 fps. This improvement is rather hard to implement, because when we want to read from the ATLAS texture we also have to send coordinates for the texture we want to the GPU program. Thus, every little part of the project that uses one of these ATLAS textures has to be altered to include the coordinates. However, even if this method might increase performance slightly, it would reduce the number of particles we could have in the simulation, if we also implement the new sparse-matrices, but seeing that 3600 particles run at 20fps, there is also a limit to how many particles we do want in the simulation.



(b) (b)

Figure 7.1 - This figure shows us the difference between ordinary textures and ATLAS textures. (a) shows us four different textures. (b) shows us an ATLAS texture, where the four different textures are put together to form one texture.

As we saw earlier in this report, one of the optimization tips given by nVidia [23] tells us that we could improve performance by using half-values and by balancing the workload between the vertex- and fragment-programs. This would lead to minor improvements in this project, the process of optimizing these has been started, but not completed. Some of the fragment-programs are accompanied by vertex-programs that calculate the texture-coordinates from [0..1, 0..1] to [0..w, 0..h]. Some fragment-programs also uses texture-lookups for the same values over the whole texture, this should have been moved to the vertex-programs. This was attempted, but it seems that the test machine has no support for the vertex-texture-fetch even though the OpenGL Extension Layer (GLEW) extension tester says that it has. For instance, it is faster to download the positions and drawing them using OpenGL than it is to draw and altering their positions based on their texture-coordinates in the vertex-program. This leads us to believe that there is no support for texture-fetches in the vertex-processors on this computer, and the driver runs the vertex-program on the CPU.

The optimization guide by nVidia [23] also inform us that we should use the smallest possible version when we compile, if one program can be compiled using version 1.0, it should be, since this version uses lighter operations. However, in this project each GPU program is compiled using version 3.0 since this version includes everything we need, but there might be a slight performance-boost if we compile each program using the smallest possible version.

A. References

[1] – Andrew Witkin and David Baraff.
Physically Based Modelling, Differential Euqation Basics.
Course Notes SIGGRAPH 2001.

[2] – D. Baraff and A. Witkin.
Large steps in cloth simulation.
Computer Graphics (Proc. SIGGRAPH), 1998.

[3] – Donald H. House and David E. Breen.
Particle Representation of Woven Fabrics.
Chapter 3 (pp 55-78) of Cloth Modelling and Animation, ISBN: 1-56881-090-3
edited by Donald H. House and David E. Breen (2000).

[4] – Mathieu Desbrun , Mark Meyer and Alan H. Barr.
Interactive Animation of Cloth-Like Objects for Virual Reality.
Chapter 9 (pp 219-329) of Cloth Modelling and Animation, ISBN: 1-56881-090-3
edited by Donald H. House and David E. Breen.

[5] – David Baraff.
Physically Based Modelling, Implicit Methods for Differential Equations.
Course Notes SIGGRAPH 2001.

[6] – Li Ling.
Aerodynamic Effects.
Chapter 7 of Cloth Modelling and Animation, ISBN: 1-56881-090-3
edited by Donald H. House and David E. Breen.

[7] – Max McGuire.
The Half-Edge Data Structure.
http://www.flipcode.com/articles/article_halfedge.shtml

[8] – Dr. Ching-Kuang Shene,
Associate Professor Department of Computer Science
Michigan Technological University.
http://www.cs.mtu.edu/~shene/COURSES/cs3621/NOTES/model/winged-e.html

[9] – Jonathan Richard Shevchuk.
An Introduction to the Conjugate Gradient Method Without the Agonizing Pain.
Technical Report CMU-CS-TR-94-123,
Carnegie Mellon University, 1994
http://www.cs.cmu.edu/~quake-papers/painless-conjugate-gradient.pdf

[10] – Wikipedia.
Conjugate Gradient Method.
http://en.wikipedia.org/wiki/Conjugate_gradient

[11] – Wikipedia.
Gaussian Elimination.
http://en.wikipedia.org/wiki/Gauss_elimination

[12] – D. Terzepoulos and K. Fleischer.
Deformable Models.
Visual Computer, 4:306-331, 1988.

[13] – Andrew Witkin.
Physically Based Modelling, Particle System Dynamics.
Course notes SIGGRAPH 2001.

[14] – Cyril Zeller.
Cloth Simulation on the GPU.
SIGGRAPH 2005.
[15] – Simon Green.
Stupid OpenGL Shader Tricks.
GDC 2003.
[16] – Xavier Provot.
Deformation Constraints in a Mass-Spring Model to Describe Rigid Cloth Behaviour.
In Graphics Interface (pp 147-155), 1995.
[17] – Wikipedia. Choleski Factorization.
http://en.wikipedia.org/wiki/Choleski_factorization
[18] – David Baraff.
Physically Based Modelling, Slides for Implicit Methods for Differential Equations.
Course Notes SIGGRAPH 2001.
[19] – D.E. Breen, D.H. House and M.J. Wozny.
Predicting the drape of woven cloth using interacting particles.
Computer Graphics (Proc. SIGGRAPH), pages 365-372, 1994.
[20] – Nell Znamierowski.
Woven Fabrics.
Chapter 1 In Cloth Modelling and Animation, ISBN: 1-56881-090-3
edited by Donald H. House and David E. Breen.
[21] – Roberto Bigliani and Jeffery W. Eischen.
Collision Detection in Cloth Modelling.
Chapter 8 in Cloth Modelling and Animation,
edited by Donald H. House and David E. Breen.
[22] – Michael Kass.
Physically Based Modelling,
Cloth and Fur Energy Functions.
Course Notes SIGGRAPH 2001.
[23] – NVIDIA
GPU Programming Guide.
http://developer.nvidia.com/object/gpu_programming_guide.html
[24] – Simon Green.
The OpenGL Framebuffer Framebuffer Object Object Extension
download.nvidia.com/developer/presentations/
2005/GDC/OpenGL_Day/OpenGL_FrameBuffer_Object.pdf
[25] – R. Bridosn, S. Marino and R. Fedkiw.
Simulation of Clothung with Folds and Wrinkles.
SIGGRAPH 2003.
[26] – Ian Buck, Stanford University.
Taking the Plunge into GPU Computing.
Chapter 32, GPU Gems 2 ISBN: 0-321-33559-7
[27] – P. Volino and N. Magnenat-Thalman.
Chapter 2 (pp. 19-24), Virtual Clothing, ISBN: 3-540-67600-7.
[28] – P. Volino and N. Magnenat-Thalman.
Chapter 2 (pp. 30-51), Virtual Clothing, ISBN: 3-540-67600-7.
[29] – Jeffrey W. Eischen and Roberto Bigliani.

Continuum versus Particle Rperesentation.
Chapter 4, Cloth Modelling and Animation, ISBN: 1-56881-090-3
edited by Donald H. House and David E. Breen.
[30] – S. Kawabata, M. Niwa, H. Kawai.
The Finite Deformation Theory of Plain-Weave Fabrics.
Journal of the Textile Institute,
64(1-2), pp 21-85, 1973.
[31] – Sylvain Lefebvre, Samuel Hornus and Fabrice Neyret.
Octree Textures on the GPU.
Chapter 37, GPU Gems 2, ISBN: 0-321-33559-7
[32] – Wikipedia.
LU decomposition.
http://en.wikipedia.org/wiki/LU_decomposition
[33] – Wikipedia.
Jacobi metoden.
http://en.wikipedia.org/wiki/Jacobi_method
[34] – Wikipedia.
Gauss-Seidel relaxation.
http://en.wikipedia.org/wiki/Gauss-Seidel_method
[35] – Cem Cebenoyan
Graphics Pipeline Performance.
Chapter 28, GPU Gems. ISBN: 0-32122-832-4
[36] – Matthias Wloka.
Improved Batching via Texture Atlases.
Chapter 2 (pp 155-167), Shader X3. ISBN: 1-58450-357-2

Andrew Witkin and David Baraff - Course Notes SIGGRAPH 2001 can be found at:
http://www.cs.cmu.edu/~baraff/sigcourse/index.html (14. June 2006).

All hyperlinks were checked and verified 14. June 2006.

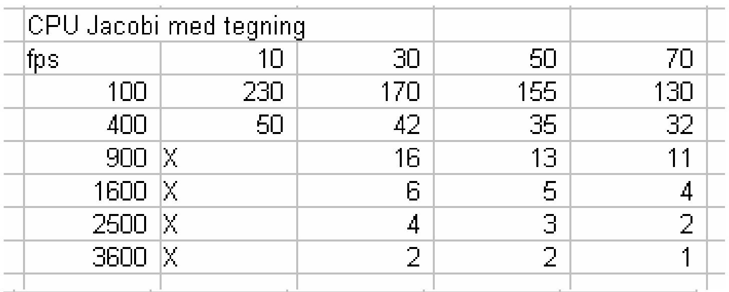
B Acronyms, Abbreviations and Dictionary

cg –	C for Graphics, a programming language similar to c++ for GPUs, made by NVIDIA.
CG –	The Conjugate Gradient method.
CPU –	The Central Processing Unit, this is the heart of every computer.
Draw-Buffer –	A Draw-Buffer is the buffer that the fragmens write their results to.
fps –	The number of frames drawn per second.
Fragment –	A fragment is equivalent with a pixel on the screen, though a fragment-processor processes several pixels at the same time.
Frame-Buffer –	A Frame-Buffer is a logical collection of Draw-Buffers and Render-Buffers
Frame-rate –	See fps.
GB –	Giga-Byte, one billion bytes
GHz –	Giga-Hertz, the clock-speed of a CPU or GPU.
GPGPU -	General Purpose Programming on the GPU.
GPU –	Graphical Processing Unit, a GPU consists of two types of processors, fragment-processors and vertex-processors. When we talk about the GPU, we talk about the entire process of going from 3D points to rendered scenes.
Iterations for convergence –	The number of iterations the numerical solver uses to find the solution vector.
MB –	Mega-Byte, one million bytes
Node –	A Node is an object that has a pointer to another Node, Nodes are used to construct linked lists.
RAM –	Random Access Memory, equvalent with memory.
Rendering-Pass –	Refers to drawing of an entire scene, in GPGPU programs it refers to one calculation which fills up one texture.

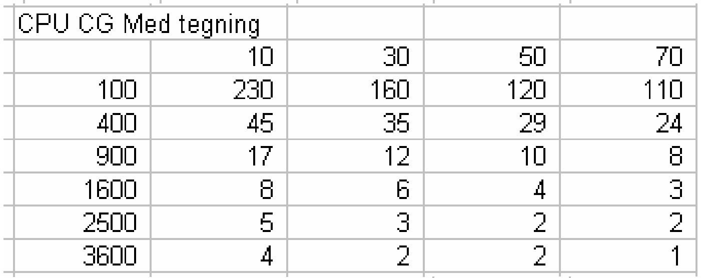
Solution Vector –	When we try to solve a linear set of equations, Ax=b, we are in essence trying to find the solution vector x.
State –	The state of the particle system is equivalent with all the particles positions, velocities and the forces acting on them.
Vertex –	A vertex is equivalent with a 3D point sent from for instance OpenGL, though a vertex-processor processes several points at the same time.

C Appendix

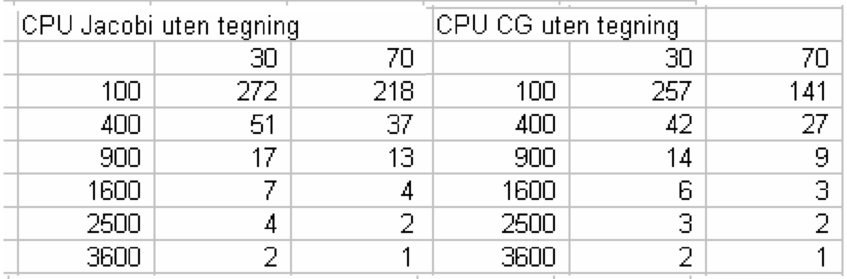
C.1 Tables



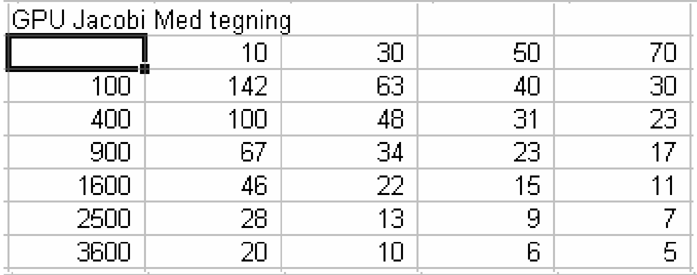
CPU Jacobi med tegning				
fps	10	30	50	70
100	230	170	155	130
400	50	42	35	32
900	X	16	13	11
1600	X	6	5	4
2500	X	4	3	2
3600	X	2	2	1



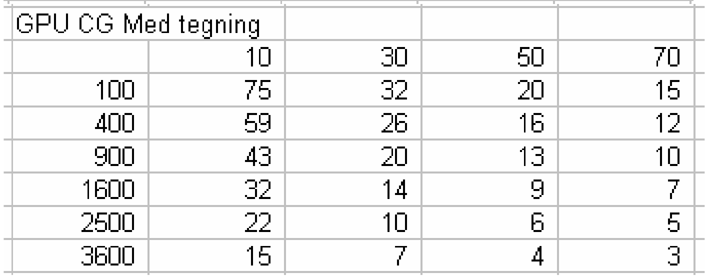
CPU CG Med tegning				
	10	30	50	70
100	230	160	120	110
400	45	35	29	24
900	17	12	10	8
1600	8	6	4	3
2500	5	3	2	2
3600	4	2	2	1



CPU Jacobi uten tegning			CPU CG uten tegning		
	30	70		30	70
100	272	218	100	257	141
400	51	37	400	42	27
900	17	13	900	14	9
1600	7	4	1600	6	3
2500	4	2	2500	3	2
3600	2	1	3600	2	1



GPU Jacobi Med tegning				
	10	30	50	70
100	142	63	40	30
400	100	48	31	23
900	67	34	23	17
1600	46	22	15	11
2500	28	13	9	7
3600	20	10	6	5



GPU CG Med tegning				
	10	30	50	70
100	75	32	20	15
400	59	26	16	12
900	43	20	13	10
1600	32	14	9	7
2500	22	10	6	5
3600	15	7	4	3

C.2 Fragment-programs

```
// fmod fungerer ikke alltid som den skal, derfor egen modulo-funksjon for å
garantere rett resultat på grafikk-kortet mitt
inline float mod(int x, int y)
{
	int temp = x/y;
	return x - y * temp;
}


// Summrerer 2 vektorer piksel for piksel(res = A + B * (c/d)
float4 sumVekVek(
	uniform samplerRECT v0 : TEXUNIT0,  // 1. operand
	uniform samplerRECT v1 : TEXUNIT1,  // 2. operand
	uniform samplerRECT var : TEXUNIT2, // matrisen med variablene
	float2 p : TEXCOORD0,                     // tekstur-koordinaten til
output
	uniform float2 size,                      // Størrelsen [w, h]
	uniform float2 indA,                      // indeks til c i var
	uniform float2 indB                             // indeks til d i var
	) : COLOR
{
	float2 pos = p * size;
	float faktorA;
	if(indA.x == -1) faktorA = 1;
	else faktorA = texRECT(var, indA).x;

	float faktorB;
	if(indB.x == -1) faktorB = 1;
	else faktorB = texRECT(var, indB).x;
	if(faktorB == 0) faktorB = 0.0000000000000001;

	float faktor = faktorA / faktorB;
	return (texRECT(v0, pos) + texRECT(v1, pos) * faktor);
}

// Summerer 2 vektorer piksel for piksel og ganger resultat med faktor
float4 sumVekVekA(
	uniform samplerRECT v0 : TEXUNIT0,  // 1. operand
	uniform samplerRECT v1 : TEXUNIT1,  // 2. operand
	float2 p : TEXCOORD0,                     // tekstur-koordinaten til
output
	uniform float2 size,                      // Størrelsen [w, h]
	uniform float faktor                      // faktor som det skal ganges
med
	) : COLOR
{
	float2 pos = p * size;

	return (texRECT(v0, pos) + texRECT(v1, pos)) * faktor;
}

// Subtraherer 2 vektorer piksel for piksel(res = A - B * (c/d)
float4 minusVekVek(
	uniform samplerRECT v0 : TEXUNIT0,  // 1. operand
	uniform samplerRECT v1 : TEXUNIT1,  // 2. operand
	uniform samplerRECT var : TEXUNIT2, // matrisen med variablene
	float2 p : TEXCOORD0,			// tekstur-koordinaten til
output
	uniform float2 size,			// Størrelsen [w, h]
	uniform float2 indA,			// indeks til c i var
	uniform float2 indB				// indeks til d i var
	) : COLOR
{
	float2 pos = p * size;
	float faktorA;
	if(indA.x == -1) faktorA = 1;
	else faktorA = texRECT(var, indA).x;

	float faktorB;
	if(indB.x == -1) faktorB = 1;
	else faktorB = texRECT(var, indB).x;
	if(faktorB == 0) faktorB = 0.0000000000000001;

	float faktor = faktorA / faktorB;

	return (texRECT(v0, pos) - texRECT(v1, pos) * faktor);
}

// Obsolete, dot-produktet av 2 vektorer i en piksel
float4 mulVekVek(
	 uniform samplerRECT v0 : TEXUNIT0,
	 uniform samplerRECT v1 : TEXUNIT1,
	 uniform samplerRECT var : TEXUNIT2,
	 float2 p : TEXCOORD0,
	 uniform float2 size,
	 uniform float2 indA,
	 uniform float2 indB
			  ) : COLOR
{

	float faktorA;
	if(indA.x == -1) faktorA = 1;
	else faktorA = texRECT(var, indA).x;

	float faktorB;
	if(indB.x == -1) faktorB = 1;
	else faktorB = texRECT(var, indB).x;
	if(faktorB == 0) faktorB = 0.0000000000000001;

	float faktor = faktorA / faktorB;

	float res = 0;
	for(int x = 0; x < size.x; x++)
	{
		for(int y = 0; y < size.y; y++)
		{
			res += dot(texRECT(v0, float2(x, y)).bgr, texRECT(v1,
float2(x, y)).bgr);
		}
	}

	res = res / faktor;
	return float4(res.xxx, faktorB);
}

// Multipliserer en identitetsmatrise(av 3x3 identitetsmatriser) med en vektor
float4 mulIMatVekFil(
				uniform samplerRECT IMat : TEXUINT0,	//
Identitetsmatrisen
				uniform samplerRECT S : TEXUNIT1,	//
Filter-matrisen
				uniform samplerRECT Vek : TEXUNIT2,	//
Vektoren
				float2 pos : TEXCOORD0,
	// tekstur-koodinaten til output
				uniform float2 size,
	// Størrelsen [w, h]
				uniform float faktor
	// Faktor som skal ganges med resultatet
				) : COLOR
{
	float2 p = pos * size;

	// Skal gange matrise fra IMat med vektor fra Vek, blir det samme som å
gange vektorene
	float3 v = texRECT(Vek, p).bgr * faktor;
	float3 temp; temp = texRECT(IMat, p).bgr;

	float3 res = temp*v;
	float3x3 m;
	p.y = ((int)p.y) * 3;
	m = float3x3(texRECT(S, p).bgr, texRECT(S, p + float2(0, 1)).bgr,
texRECT(S, p + float2(0, 2)).bgr);
	return float4(mul(m, res).bgr, 0);
}



// Multipliserer en sparse-matrise med en vektor og filtrerer resultatet
float4 mulSMatVekFil(
	uniform samplerRECT M0 : TEXUNIT0,	// 1. linje av 3x3 matrisene i
M
	uniform samplerRECT M1 : TEXUNIT1,	// 2. linje av 3x3 matrisene i
M
	uniform samplerRECT M2 : TEXUNIT2,	// 3. linje av 3x3 matrisene i
M
	uniform samplerRECT S : TEXUNIT3,	// S-matrisen av 3x3 matriser
som skal ganges inn i res (3hxw)
	uniform samplerRECT Vek : TEXUNIT4,	// Vektoren(wxh)
	float2 texCoord : TEXCOORD0,		// Teksturkoordinaten til
output
	uniform float2 size,				// Størrelsen [w, h]
	uniform float2 sizeBIG,				// Størrelsen [26, w*h]
	uniform float faktor				// Faktor ganges direkte
inn sammen med multiplisering
	) : COLOR
{
	float2 p = texCoord*size;
	p.x = floor(p.x);
	p.y = floor(p.y);

	// Må finne hvilken linje i MatD som skal ganges med Vek
	float y = floor((p.y * size.x + p.x));

	float3x3 m;
	float3 v;

	// ganger diagonalelementet med Vek[p]
	m = float3x3(texRECT(M0, float2(0, y)).bgr, texRECT(M1, float2(0, 
y)).bgr, texRECT(M2, float2(0, y)).bgr);
	v = texRECT(Vek, p).bgr;


	float3 res = mul(m, v);

	// ganger hver 3x3 matrise med tilsvarende element i vektoren
	for(int x = 1; x < 26; x++)
	{

		float xo = (mod(float(x-1), 5) - 2);
		float yo = floor(float(x-1))/5-2;//((x-1) / 5 - 2);

		float x1 = xo + p.x;
		float y1 = yo + p.y;
		if(!(x1 < 0 || size.x <= x1 || y1 < 0 || size.y <= y1 || (xo == 0 
&& yo == 0) || (12.99 <= x && x <= 13.0001)))
		{

			m = float3x3(texRECT(M0, float2(x, y)).bgr, texRECT(M1, 
float2(x, y)).bgr, texRECT(M2, float2(x, y)).bgr);
			v = texRECT(Vek, float2(x1, y1)).bgr;

			res += mul(m, v);
		}
	}

	// Filterer resultatet
	m = float3x3(texRECT(S, float2(p.x, p.y*3)).bgr, texRECT(S, float2(p.x, 
p.y*3+1)).bgr, texRECT(S, float2(p.x, p.y*3+2)).bgr);
	res = mul(m, res);

	return float4(res.bgr, 0);
}


// Summerer M(i, i) = Sum(i!=j, M(i, j)
void sumRekke(
				uniform samplerRECT M0 : TEXUNIT0,  // 1. linje i 3x3 
matrisene i M
				uniform samplerRECT M1 : TEXUNIT1,  // 2. linje i 3x3 
matrisene i M
				uniform samplerRECT M2 : TEXUNIT2,  // 3. linje i 3x3 
matrisene i M
				uniform float2 size,                    // Størrelsen 
[w, h]
```

```
                        float2 texCoord : TEXCOORD0,        // tekstur-
koordinaten til output
                        out float4 res0 : COLOR0,                 // 1. linje
av 3x3 matrisen
                        out float4 res1 : COLOR1,                 // 2. linje
av 3x3 matrisen
                        out float4 res2 : COLOR2                  // 3. linje
av 3x3 matrisen
      )
{
      float2 pos = texCoord * size;

      res0 = float4(0, 0, 0, 1);
      res1 = float4(0, 0, 0, 1);
      res2 = float4(0, 0, 0, 1);



      // Skal summere sammen alle elementer med samme y-verdi, får feil verdi
om jeg prøver å ha løkken fra 0 til 25
      for(int x = 1; x < 13; x++)
      {
            res0.bgra -= texRECT(M0, float2(x, pos.y)).bgra;
            res1.bgr -= texRECT(M1, float2(x, pos.y)).bgr;
            res2.bgr -= texRECT(M2, float2(x, pos.y)).bgr;


      }
      for(int x = 14; x < 26; x++)
      {
            res0.bgra -= texRECT(M0, float2(x, pos.y)).bgra;
            res1.bgr -= texRECT(M1, float2(x, pos.y)).bgr;
            res2.bgr -= texRECT(M2, float2(x, pos.y)).bgr;
      }
}


// tar dot-produktet mellom 2 tilsvarende element og legger resultatet i en
tilsvarende piksel i output
float4 mulPiksel(
                         uniform samplerRECT v0 : TEXUNIT0, // 1. operand
                         uniform samplerRECT v1 : TEXUNIT1, // 2. operand
                         uniform float2 size,                     // Størrelsen
[w, h]
                         float2 texCoord : TEXCOORD0        // tekstur-
koordinaten til output
                         ) : COLOR
{
      float2 pos = texCoord * size;

      return texRECT(v0, pos) * texRECT(v1, pos);
}


// Halverer input og legger summen av 4 elementer fra input i output
float4 halver(
                    uniform samplerRECT v : TEXUNIT0, // Vektoren som skal
halveres
                    uniform float2 forrigeSize,      // Størrelsen [2*w, 2*h]
                    uniform float2 size,                   // Størrelsen [w,
h]
                    float2 texCoord : TEXCOORD0      // tekstur-koordinaten
til output
                    ) : COLOR
{

      int2 pos = texCoord * size;// + float2(0.5, 0.5);
      int2 pos2 = 2 * pos;

      // denne posisjonen skal inneholde de 4 pikslene som tilsvarer den
forrige teksturen
      // Er det den nederste linjen/kolonnen i dette ut-bildet, skal den ta
med seg evt. overskytende verdier
      float3 res = float3(0, 0, 0);
      int sluttX = (pos.x == size.x-1 ? forrigeSize.x : pos2.x+2);
      int sluttY = (pos.y == size.y-1 ? forrigeSize.y : pos2.y+2);
      for(int x = pos2.x; x < sluttX; x++)
      {
            for(int y = pos2.y; y < sluttY; y++)
            {
                  res.bgr += texRECT(v, float2(x, y)).bgr;
            }
      }

      return float4(res, sluttY);


}

// Kopierer en tekstur til en annen, kan gange med 2 variable i teksturen Var
res = v * (c/d)
float4 kopier(
                    uniform samplerRECT v : TEXUNIT0, // Matrise som skal
kopieres
                    float2 texCoord : TEXCOORD0,           // tekstur-
koordinaten til output
                    uniform float2 size,                   // Størrelsen [w,
h]
                    uniform float2 indA,                   // indeksen til c i
var
                    uniform float2 indB,                   // indeksen til d i
var
                    uniform samplerRECT var : TEXUNIT1// Matrisen med variable
                    ) : COLOR
{

      float2 p = texCoord * size;
      float3 temp = texRECT(v, p).bgr;
      float3 res;
      if(size.x == 0)
      {
            float faktorA;
            if(indA.x == -1) faktorA = 1;
            else faktorA = texRECT(var, indA).x;

            float faktorB;
            if(indB.x == -1) faktorB = 1;
            else faktorB = texRECT(var, indB).x;
            if(faktorB == 0) faktorB = 0.0000000000000001;

            float faktor = faktorA / faktorB;
            res.bgr = (temp.x+temp.y+temp.z).xxx*faktor;
      }
      else
      {
            res = temp;
      }
      return float4(res.bgr, 0);
}

//Filterer vektoren
float4 filter(
                    uniform samplerRECT v : TEXUNIT0, // vektoren som skal
filtreres
                    uniform samplerRECT S : TEXUNIT1, // Filterings-matrisen
                    float2 texCoord : TEXCOORD0,           // tekstur-
koordinaten til output
                    uniform float2 size                    // Størrelsen [w,
h]
                    ) : COLOR
{

      float2 p = texCoord * size;

      float2 pS = floor(p); pS.y *= 3;

      float3x3 m = float3x3(texRECT(S, pS).bgr, texRECT(S, pS + float2(0,
1)).bgr, texRECT(S, pS + float2(0, 2)).bgr);
      float3 vek = texRECT(v, p).bgr;

      return float4(mul(m, vek).bgr, 0);

}


// Test-metode for TestDraw-klassen
float4 nullstill(
                         uniform samplerRECT tex0 : TEXUNIT0,
                         uniform samplerRECT tex1 : TEXUNIT1,
                         uniform samplerRECT tex2 : TEXUNIT2,
                         uniform samplerRECT Vek : TEXUNIT3,
                         float2 texCoord : TEXCOORD0,
                         uniform float4 size, //
                         uniform float4 value
                         ) : COLOR
{
      float4 res = float4(0, 0, 0, 0);
      half3x3 temp;
      half3 v;
      float2 pos = texCoord * size.zw;
```

```
        pos.x = 0;
        for(int i = 0; i < 26; i++)
        {
                v = (texRECT(Vek, pos+float2(i, 0)).bgr);
                temp = half3x3(texRECT(tex0, pos+float2(i, 0)).bgr,
                                                texRECT(tex1, pos+float2(i, 0)).bgr,
                                                texRECT(tex2, pos+float2(i, 0)).bgr);
                res.bgr += mul(temp, v);

        }
        if(value.x == 0 && value.y == 0 && value.z == 0 && value.a == 0)
        {

                // res.bgra = size.xyzw;
        }
        else
        {
                res += value;
        }
        return res;
}


// Test-metoden for TestDraw klassen
float4 mainNull() : COLOR
{
        return float4(0, 0, 0, 0);
}

struct OUTTex
{
        float4 pos : POSITION;
        float2 faktor : TEXCOORD1;
        float2 tex : TEXCOORD0;
};

// Endrer tekstur-koordinatet fra [0..1, 0..1] til [0..w, 0..h]
OUTTex mainVertex1(
                        uniform float2 size,
                        float2 texCoord : TEXCOORD0,
                        float4 position : POSITION
                        )
{
        OUTTex ret;
        ret.pos = position;
        ret.faktor = float2(0, 0);
        ret.tex = texCoord * size;
        return ret;
}

// Endrer tekstur-koordinatet fra [0..1, 0..1] til [0..w, 0..h] og laster inn
faktor som kan brukes videre i fragment-programmet
OUTTex mainVertex2(
                        uniform float2 size,
                        float2 texCoord : TEXCOORD0,
                        float2 test : TEXCOORD1,
                        float4 position : POSITION,
                        uniform float2 indA,
                        uniform float2 indB,
                        uniform samplerRECT Var : TEXUNIT5
                        )
{
        OUTTex ret;
        ret.pos = position;
        ret.tex = texCoord * size;

        float faktorA;
        if(indA.x == -1) faktorA = 1;
        else faktorA = texRECT(Var, indA).x;

        float faktorB;
        if(indB.x == -1) faktorB = 1;
        else faktorB = texRECT(Var, indB).x;
        if(faktorB == 0) faktorB = 0.0000000000000001;

        float faktor = faktorA / faktorB;

        ret.faktor = faktor.xx;
        return ret;
}


OUTTex mainVertexNull(
                        uniform float2 size,
                        float2 texCoord : TEXCOORD0,
                        float4 position : POSITION
                        )
{
        OUTTex ret;
        ret.pos = position;
        ret.faktor = float2(0, 0);
        ret.tex = texCoord * size;
        return ret;
}

Jacobi.cg
inline float modulo(int x, int y)
{
        int temp = x/y;
        return x - y * temp;
}
// Jacobi løser ligningen Ax = b
// Hver piksel tar seg av en i-verdi
float4  Jacobi(
                        uniform samplerRECT A0 : TEXUNIT0,        // Er 1. linje av
3x3 matrisene i A
                        uniform samplerRECT A1 : TEXUNIT1,        // Er 2. linje av
3x3 matrisene i A
                        uniform samplerRECT A2 : TEXUNIT2,        // Er 3. linje av
3x3 matrisene i A
                        uniform samplerRECT inv0 : TEXUNIT3,      // ER 1. linje av
invers 3x3 matriser av diagonalen til A
                        uniform samplerRECT inv1 : TEXUNIT4,      // ER 2. linje av
invers 3x3 matriser av diagonalen til A
```

```
                  uniform samplerRECT inv2 : TEXUNIT5,     // ER 3. linje av
invers 3x3 matriser av diagonalen til A
                  uniform samplerRECT dv0 : TEXUNIT6,      // Forrige dv
verdier
                  uniform samplerRECT b : TEXUNIT7,        // Er b vektoren
                  float2 tex : TEXCOORD0,                        // Tekstur-
koordinaten til output
                  uniform float2 size                                  //
Størrelsen [w, h]
                  ) : COLOR
{


      // dv1(i) = A(i, i)^-1 * (b(i) - Sum(i!=j, A(i, j) * dv0(j))
      float2 pos = floor(tex * size);
      float i =  floor((pos.y * size.x + pos.x));
      half3 res = half3(0, 0, 0);
      half3x3 mat;
      half3 vek;
      half3 temp = half3(0, 0, 0);

      // Sum(i != j, A(i, j) * dv0(j))

      for(int j = 1; j < 26; j++)
      {
            half xo = (modulo(half(j-1), 5) - 2);
            half yo = floor(half(j-1))/5-2;

            float x1 = xo + pos.x;
            float y1 = yo + pos.y;
            if(!(x1 < 0 || size.x <= x1 || y1 < 0 || size.y <= y1 || (xo == 0
&& yo == 0) || (12.99 <= j && j <= 13.0001)))
            {
                  // Henter ut matrisen3x3 fra A og vektoren fra dv0
                  mat = half3x3(    texRECT(A0, half2(j, i)).bgr,
                                          texRECT(A1, half2(j, i)).bgr,
                                          texRECT(A2, half2(j, i)).bgr);
                  vek = texRECT(dv0, half2(x1, y1)).bgr;
                  res += mul(mat, vek);
            }
      }

      // b - Sum(...)
      temp = texRECT(b, pos).bgr - res;

      // (b-Sum(...))/A(i, i)
      mat = half3x3(texRECT(inv0, pos).bgr, texRECT(inv1, pos).bgr,
texRECT(inv2, pos).bgr);
      res = mul(mat, temp);


      // TODO: res = mul(S, res) for filtrering

      return float4(res.bgr, 0);

}
```

```
inline half det(half2x2 m)
{
	return m._m00*m._m11 - m._m01*m._m10;
}
inline half det(half3x3 m)
{
	return	m._m00 * det(half2x2(m._m11_m12_m21_m22)) -
		m._m01 * det(half2x2(m._m10_m12_m20_m22)) +
		m._m02 * det(half2x2(m._m10_m11_m20_m21));
}

void Invert(
			uniform samplerRECT A0 : TEXUNIT0,
			uniform samplerRECT A1 : TEXUNIT1,
			uniform samplerRECT A2 : TEXUNIT2,
			float2 tex : TEXCOORD0,
			uniform float2 size,
			out float4 res1 : COLOR0,
			out float4 res2 : COLOR1,
			out float4 res3 : COLOR2,
			out float4 test : COLOR3
			)
{
	float2 pos = floor(tex * size);
	half y = pos.y * size.x + pos.x;
	half3x3 m = half3x3(texRECT(A0, half2(0, y)).bgr, texRECT(A1, half2(0,
y)).bgr, texRECT(A2, half2(0, y)).bgr);
	half3x3 inv = half3x3(det(half2x2(m._m11_m12_m21_m22)), -
det(half2x2(m._m10_m12_m20_m22)), det(half2x2(m._m10_m11_m20_m21)),
					-det(half2x2(m._m01_m02_m21_m22)),
det(half2x2(m._m00_m02_m20_m22)), -det(half2x2(m._m00_m01_m20_m21)),
					det(half2x2(m._m01_m02_m11_m12)), -
det(half2x2(m._m00_m02_m10_m12)), det(half2x2(m._m00_m01_m10_m11)));

	// Skal finne invers av m og legge resultat i inv
	half detA = det(m);

	inv = inv / detA;


	//// legger data i color
	res1 = float4(inv._m02_m01_m00, 0);
	res2 = float4(inv._m12_m11_m10, 0);
	res3 = float4(inv._m22_m21_m20, 0);
}

ItererV0.cg
// Beregner ny fart etter formelen: V(i+1) = V(i) + DV(i) * dt
float4 main(
		uniform samplerRECT V0 : TEXUNIT0,  // Vektoren med den gamle
farten
		uniform samplerRECT DV : TEXUNIT1,  // Vektoren med
akselerasjonen
		uniform float dt,			// Tidssteget
		float2 pos : TEXCOORD0,			// Tekstur-
koordinaten til output
```

```
                  uniform float2 size                            // Størrelsen
[w, h]
                  ) : COLOR
{
      float2 p = pos * size;
      return texRECT(V0, p)*0.999 + texRECT(DV, p) * dt;
}

ItererP0.cg
// Oppdaterer posisjonen etter formelen: P(i+1) = P(i) + V(i) * dt + DV(i) *
0.5 * dt * dt
float4 main(
                  uniform samplerRECT V0 : TEXUNIT0, // Vektoren med farten
                  uniform samplerRECT DV : TEXUNIT1, // Vektoren med
akselerasjonen
                  uniform samplerRECT P0 : TEXUNIT2, // Vektoren med de gamle
posisjonene
                  uniform float dt,                        // tidssteget
                  float2 pos : TEXCOORD0,                  // tekstur-
koordinaten til output
                  uniform float2 size                            // Størrelsen
[w, h]
                  ) : COLOR                                      // Skriver
resultatet til output
{
      float2 p = pos * size;
      return texRECT(P0, p) + texRECT(V0, p) * dt + texRECT(DV, p) * dt * dt *
0.5;
}

genererP.cg

// Genererer identitets-matrisen P, der P(i) = 1/A(i, i)
float4 genP(
                  uniform samplerRECT A0 : TEXUNIT0, // Sparse-matrisen med 1.
linje av 3x3 matrisene i A
                  uniform samplerRECT A1 : TEXUNIT1, // Sparse-matrisen med 2.
linje av 3x3 matrisene i A
                  uniform samplerRECT A2 : TEXUNIT2, // Sparse-matrisen med 3.
linje av 3x3 matrisene i A
                  uniform float2 size,                     // Størrelsen [w,
h]
                  float2 texCoord : TEXCOORD0              // Tekstur-
koordinaten til output
                  ) : COLOR                                      // Returnerer
resultatet
{
      int2 pos = texCoord * size;

      int i = ((int)pos.y)*size.x + pos.x;

      float4 res;

      res.bgra = float4(1/texRECT(A0, float2(0, 3*i)).b, 1/texRECT(A1,
float2(0, 3*i+1)).g, 1/texRECT(A2, float2(0, 3*i+2)).r, 0);

      return res;
```

```
}


// Genererer identitets-matrisen P1, der P1(i) = A(i, i)
float4 genP1(
                    uniform samplerRECT A0 : TEXUNIT0,  // Sparse-matrisen med 1.
linje av 3x3 matrisene i A
                    uniform samplerRECT A1 : TEXUNIT1,  // Sparse-matrisen med 2.
linje av 3x3 matrisene i A
                    uniform samplerRECT A2 : TEXUNIT2,  // Sparse-matrisen med 3.
linje av 3x3 matrisene i A
                    uniform float2 size,                      // Størrelsen [w,
h]
                    float2 texCoord : TEXCOORD0               // Tekstur-
koordinaten til output
                    ) : COLOR                                        // Returnerer
resultatet
{
      int2 pos = texCoord * size;

      int i = ((int)pos.y)*size.x + pos.x;

      float4 res;

      res.bgra = float4(texRECT(A0, float2(0, 3.*i)).b, texRECT(A1, float2(0,
3.*i+1)).g, texRECT(A2, float2(0, 3.*i+2)).r, 3*i);

      return res;
}



genererA.cg

// dette fragment-programmet utfører A = M - dt * dfdv - dt * dt * dfdx
void main(
      uniform samplerRECT M : TEXUNIT0,         // Identitets-Matrisen med
massene
      uniform samplerRECT dfdv0 : TEXUNIT1,     // Sparse-Matrisen med 1. linje
i 3x3 matrisene for dfdv
      uniform samplerRECT dfdv1 : TEXUNIT2,     // Sparse-Matrisen med 2. linje
i 3x3 matrisene for dfdv
      uniform samplerRECT dfdv2 : TEXUNIT3,     // Sparse-Matrisen med 3. linje
i 3x3 matrisene for dfdv
      uniform samplerRECT dfdx0 : TEXUNIT4,     // Sparse-Matrisen med 1. linje
i 3x3 matrisene for dfdx
      uniform samplerRECT dfdx1 : TEXUNIT5,     // Sparse-Matrisen med 2. linje
i 3x3 matrisene for dfdx
      uniform samplerRECT dfdx2 : TEXUNIT6,     // Sparse-Matrisen med 3. linje
i 3x3 matrisene for dfdx
      uniform float dt,                               // Tidssteget
      uniform float2 size,                            // Størrelsen [w, h]
      uniform float2 sizeS,                           // Størrelsen [26, w*h]
      float2 texCoord : TEXCOORD0,              // Tekstur-koordinaten til
output
      out float4 res0 : COLOR0,                       // 1. linje i 3x3
matrisen i resultatet
```

```
	out float4 res1 : COLOR1,				// 2. linje i 3x3 
matrisen i resultatet
	out float4 res2 : COLOR2				// 3. linje i 3x3 
matrisen i resultatet
	)
{

	float2 pos = texCoord * size;
	float dt2 = dt * dt;
	res0 = float4(0, 0, 0, 0);
	res1 = float4(0, 0, 0, 0);
	res2 = float4(0, 0, 0, 0);

	// Dersom pos.x = 0, er dette diagonal-posisjonen, da skal elementet fra 
masse-matrisen også summeres med
	if((int)pos.x == 0)
	{
		int x = fmod(((int)pos.y), sizeS.x);
		int y = ((int)pos.y) / sizeS.x;

		float4 temp = texRECT(M, float2(x, y)).bgra;
		res0.b = temp.x;
		res1.g = temp.x;
		res2.r = temp.x;
		res0.bgra = res0.bgra - dt * texRECT(dfdv0, pos).bgra - dt2 * 
texRECT(dfdx0, pos).bgra;
		res1.bgra = res1.bgra - dt * texRECT(dfdv1, pos).bgra - dt2 * 
texRECT(dfdx1, pos).bgra;
		res2.bgra = res2.bgra - dt * texRECT(dfdv2, pos).bgra - dt2 * 
texRECT(dfdx2, pos).bgra;

	}
	else
	{
		res0.bgra = - dt * texRECT(dfdv0, pos).bgra - dt2 * texRECT(dfdx0, 
pos).bgra;
		res1.bgra = - dt * texRECT(dfdv1, pos).bgra - dt2 * texRECT(dfdx1, 
pos).bgra;
		res2.bgra = - dt * texRECT(dfdv2, pos).bgra - dt2 * texRECT(dfdx2, 
pos).bgra;
	}
}
```

beregnF.cg

```
// Beregner fjærkraften mellom xi og xj
inline float3 beregnKrefter(float3 x0, float3 x1, float3 v0, float3 v1, float 
ks, float kd, float rest)
{
	// Fi = -(ks*(vL - rest) +  kd(L*L'/vL))*v / vL

	float3 xVek = x1 - x0;
	float3 vVek = v1 - v0;

	float xLen = length(xVek); // if(xLen == 0) xLen = 0.0000000000000001;

	return (ks*(xLen-rest)+kd*(dot(xVek, vVek)/xLen))*xVek/xLen;
```

```
}


float4 main(
                  uniform samplerRECT P0 : TEXUNIT0,             // Er en 2D
tekstur som representerer posisjonene til partiklene
                  uniform samplerRECT V0 : TEXUNIT1,             // Er en 2D
tekstur som representerer farten til partiklene
                  uniform samplerRECT rest : TEXUINT2,           // Er en 2D
tekstur som representerer hvileavstanden mellom 2 partikler
                  float2 pos : TEXCOORD0,                              // Er
posisjonen til denne partikkelen i teksturene
                  uniform float ks,                                    // Er
stivheten på fjærene
                  uniform float kd,                                    // Er
dempingsfaktoren på fjærene
                  uniform float2 size
      // Er størrelsen på teksturene
                  ) : COLOR
      // Retur-verdien returnerer vektoren som er resultatet
{

      // Skal hente ut posisjon og fart fra alle partikler denne er koblet til
      // Denne partikkelen ligger i posisjonen (pos.x, pos.y) i teksturene
      // Den skal kobles til de 24 nærmeste partiklene
      int2 ipos = int2(pos);
      int xStart = ipos.x-2;
      int yStart = ipos.y-2;
      int xSlutt = ipos.x+2;
      int ySlutt = ipos.y+2;

      if(xStart < 0) xStart = 0;
      if(xSlutt >= size.x) xSlutt = size.x-1;
      if(yStart < 0) yStart = 0;
      if(ySlutt >= size.y) ySlutt = size.y-1;

      // Initialiserer color og itererer over alle naboene som denne er koblet
til og summerer krefter
      // Nullstiller kraften til å være gravitasjons-vektoren, massen er
antatt å være 1
      float4 color = float4(9.81, 0, 0, 0);

      for(int x = xStart; x <= xSlutt; x++)
      {
            for(int y = yStart; y <= ySlutt; y++)
            {
                  float3 rVek = float3(x-ipos.x, y-ipos.y, 0);
                  float r = length(rVek);
                  // Beregner kreftene mellom (ipos.x, ipos.y) og (x, y)
                  if(x != ipos.x || y != ipos.y) color.bgr += beregnKrefter(
      texRECT(P0, ipos).bgr,

                              texRECT(P0, float2(x, y)).bgr,

                              texRECT(V0, ipos).bgr,

                              texRECT(V0, float2(x, y)).bgr,
                              ks, kd,

                              r);
            }
      }
      return color;
}

beregnDFDX.cg
// fmod fungerer ikke alltid som den skal, derfor egen modulo-funksjon for å
garantere rett resultat på grafikk-kortet mitt
inline int modulo(float x, int y)
{
      int temp = ((int)x)/y;
      x = x - temp * y;
      return (int)x;
}

// Beregner 1. linje i 3x3 matrisen som skal skrives ut
inline float3 beregn1(float3 dcdx, float3 d2, float C, float C_dot, float ks,
float kd)
{
      float3 res;
      res.x = -ks*(-dcdx.x*dcdx.x + d2.x * C) - kd * (d2.x * C_dot);
      res.y = -ks*(-dcdx.x*dcdx.y + d2.x * C);
      res.z = -ks*(-dcdx.x*dcdx.z + d2.x * C);

      return res;
}

// Beregner 2. linje i 3x3 matrisen som skal skrives ut
inline float3 beregn2(float3 dcdx, float3 d2, float C, float C_dot, float ks,
float kd)
{
      float3 res;
      res.x = -ks*(-dcdx.y*dcdx.x + d2.y * C);
      res.y = -ks*(-dcdx.y*dcdx.y + d2.y * C) - kd * (d2.y * C_dot);
      res.z = -ks*(-dcdx.y*dcdx.z + d2.y * C);

      return res;
}

// Beregner 3. linje i 3x3 matrisen som skal skrives ut
inline float3 beregn3(float3 dcdx, float3 d2, float C, float C_dot, float ks,
float kd)
{
      float3 res;
      res.x = -ks*(-dcdx.z*dcdx.x + d2.z * C);
      res.y = -ks*(-dcdx.z*dcdx.y + d2.z * C);
      res.z = -ks*(-dcdx.z*dcdx.z + d2.z * C) - kd * (d2.z * C_dot);

      return res;
}


// Obsolete funksjon for å mappe addresse til rest-matrisen
```

```
inline float2 finnRest(int2 xi, int2 xj, float2 size)
{
      int y = xi.y*size.x+xi.x;
      int dx = xj.x - xi.x;
      int dy = xj.y - xi.y;
      int x = (dy+2) * 5 + (dx+2);
      return float2(x+1, y);
}


// pos.x --> angir hvilken partikkel denne er koblet til, pos.x E [1..25] -->
den er koblet, pos.x == 0 --> dette er diagnoalelementet
void main(
                  uniform samplerRECT P0 : TEXUNIT0,             // Er en 2D
tekstur som representerer posisjonene til partiklene
                  uniform samplerRECT V0 : TEXUNIT1,             // Er en 2D
tekstur som representerer farten til partiklene
                  uniform samplerRECT rest : TEXUNIT2,           //
Hvileavstander mellom partiklene
                  float2 pos : TEXCOORD0,                              // Er
posisjonen (i, j) i den store matrisen, må omforme for å finne rett posisjon i
P0 og V0
                  uniform float ks,                                    // Er
stivheten på fjærene
                  uniform float kd,                                    // Er
dempingsfaktoren på fjærene
                  uniform float2 sizeBIG,                              // Er
størrelsen på skjermen(den store matrisen)
                  uniform float2 size,                                 // Er
størrelsen på teksturene
                  out float4 res0 : COLOR0,                            //
Output for 1. linje i 3x3 matrisen som skal skrives
                  out float4 res1 : COLOR1,                            //
Output for 2. linje i 3x3 matrisen som skal skrives
                  out float4 res2 : COLOR2                             //
Output for 3. linje i 3x3 matrisen som skal skrives
                  )
{
      res0 = float4(0, 0, 0, -1);
      res1 = float4(0, 0, 0, -18);
      res2 = float4(0, 0, 0, -1);
      float xpos, ypos;
      float2 posB = floor(pos);
      if(posB.x != 0)
      {

            // y-verdien sier hvilken partikkel xi er
            xpos = modulo(posB.y, size.x);
            ypos = floor(posB.y / size.x);
            float2 pos2 = float2(xpos, ypos);


            // x-verdien anngir hvilken partikkel den er koblet til
            xpos = modulo(posB.x-1, 5) - 2;
            ypos = floor(posB.x-1.)/5-2;
            float2 pos1 = floor(pos2 + float2(xpos, ypos));
```

```
		float EPS = 0.0000001;

		pos1 = floor(pos1);
		pos2 = floor(pos2);


		// Luker ut de xj, der koordinater er utenfor teksturen
		if(pos1.x >= -0.00000001 && pos1.x < size.x && pos1.y >= -
0.000000001 && pos1.y < size.y && !(-0.0000001<=xpos && xpos <= 0.00000001 &&
-0.0000001<=ypos && ypos <= 0.00000001))
		{
			// Begynner utregningen av deriverte mellom punktene
			float3 rVek = float3(pos1.x-pos2.x, pos1.y-pos2.y, 0);
			float r = length(rVek);


			float3 v = texRECT(P0, pos2).bgr - texRECT(P0, pos1).bgr;
			float3 fv = texRECT(V0, pos2).bgr - texRECT(V0, pos1).bgr;

			float lengde = length(v);
			if(lengde == 0) lengde = 0.00000000000000000001;
			float3 dcdx = v / lengde;

			float oneovernorm = 1/lengde;
			float oneovercubed = oneovernorm*oneovernorm*oneovernorm;

			float3 d2 = (dot(v, v)*oneovercubed - oneovernorm).xxx;
			float C = lengde - r;

			float C_dot = dcdx * fv;


			res0.bgr = beregn1(dcdx, d2, C, C_dot, ks, kd);
			res1.bgr = beregn2(dcdx, d2, C, C_dot, ks, kd);
			res2.bgr = beregn3(dcdx, d2, C, C_dot, ks, kd);
		}
	}
}
```

beregnDFDV.cg

```
// fmod fungerer ikke alltid som den skal, derfor egen modulo-funksjon for å
garantere rett resultat på grafikk-kortet mitt
inline int modulo(float x, int y)
{
	int temp = ((int)x)/y;
	x = x - temp * y;
	return (int)x;
}

// Beregner 1. linje i 3x3 matrisen som skal skrives ut
inline float3 beregn1(float3 dcdx, float kd)
{
	float3 res;
	res.x = kd * dcdx.x * dcdx.x;
	res.y = kd * dcdx.x * dcdx.y;
	res.z = kd * dcdx.x * dcdx.z;

	return res;
}

// Beregner 2. linje i 3x3 matrisen som skal skrives ut
inline float3 beregn2(float3 dcdx, float kd)
{
	float3 res;
	res.x = kd * dcdx.y * dcdx.x;
	res.y = kd * dcdx.y * dcdx.y;
	res.z = kd * dcdx.y * dcdx.z;

	return res;
}

// Beregner 3. linje i 3x3 matrisen som skal skrives ut
inline float3 beregn3(float3 dcdx, float kd)
{
	float3 res;
	res.x = kd * dcdx.z * dcdx.x;
	res.y = kd * dcdx.z * dcdx.y;
	res.z = kd * dcdx.z * dcdx.z;

	return res;
}



// Denne metoden beregner dfdv(xi, xj), 2 partikler med posisjon gitt av
output-koordinat
// Kjører vertex-program for å gange opp teksturkoordinat til [0..w, 0..h]
void main(
			uniform samplerRECT P0 : TEXUNIT0,		// Er en 2D
tekstur som representerer posisjonene til partiklene
			uniform samplerRECT V0 : TEXUNIT1,		// Er en 2D
tekstur som representerer farten til partiklene
			float2 pos : TEXCOORD0,				// Er
posisjonen (i, j) i den store matrisen, må omforme for å finne rett posisjon i
P0 og V0
			uniform float ks,				// Er
stivheten på fjærene
			uniform float kd,				// Er
dempingsfaktoren på fjærene
			uniform float2 sizeBIG,				// Er
størrelsen på skjermen(den store matrisen)
			uniform float2 size,				// Er
størrelsen på teksturene
			out float4 res0 : COLOR0,			//
Output for 1. linje i 3x3 matrisen som skal skrives
			out float4 res1 : COLOR1,			//
Output for 2. linje
			out float4 res2 : COLOR2			//
Output for 3. linje
			)
{
	res0 = float4(0, 0, 0, -1);
	res1 = float4(0, 0, 0, -1);
	res2 = float4(0, 0, 0, -1);

	float xpos, ypos;
	float2 posB = floor(pos);

	if(posB.x != 0)
	{

		// y-verdien sier hvilken partikkel xi er
		xpos = modulo(posB.y, size.x);
		ypos = floor(posB.y / size.x);
		float2 pos2 = float2(xpos, ypos);


		// x-verdien anngir offsettet til xj i forhold til xi
		xpos = modulo(posB.x-1, 5) - 2;
		ypos = floor((posB.x-1)/5-2);
		float2 pos1 = floor(pos2 + float2(xpos, ypos));

		float EPS = 0.0000001;

		pos1 = floor(pos1);
		pos2 = floor(pos2);


		// Luker ut de xj verdiene som har koordinat utenfor teksturen,
pluss nr. 13 som er offset(0, 0)
		if(pos1.x >= -0.00000001 && pos1.x < size.x && pos1.y >= -
0.000000001 && pos1.y < size.y && !(-0.0000001<=xpos && xpos <= 0.00000001 &&
-0.0000001<=ypos && ypos <= 0.00000001))
		{

			// Begynner utregningen av deriverte mellom partiklene
			float3 v = texRECT(P0, pos2).bgr - texRECT(P0, pos1).bgr;
			float3 fv = texRECT(V0, pos2).bgr - texRECT(V0, pos1).bgr;

			float lengde = length(v);
			if(lengde == 0) lengde = 0.0000000000000000001;
			float3 dcdx = v / lengde;

			res0.bgr = beregn1(dcdx, kd);
			res1.bgr = beregn2(dcdx, kd);
			res2.bgr = beregn3(dcdx, kd);

		}
	}
}
```