The Red-Black Physics Engine

A Parallel Framework for Interactive Soft Body Dynamics

Master's thesis in Interaction Design and Technologies

OSKAR NYLÉN
PONTUS PALL

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2017

The Red-Black Physics Engine
A Parallel Framework for Interactive Soft Body Dynamics
Oskar Nylén & Pontus Pall
Department of Computer Science and Engineering
Chalmers University of Technology

Abstract

The simulation of soft bodies has been an ongoing research problem for over 30 years. The desiderata for real-time applications are believable results, while maintaining interactivity. The most popular approach to achieve this has been to use iterative methods, that find an approximate solution to the underlying equation system rapidly. In recent years, efforts have been made to improve the performance of these methods by exploiting the computational capabilities of modern hardware architectures, such as the graphics processing unit.

This thesis introduces a parallel iterative solver that utilizes a Red-Black Gauss-Seidel technique. The solver is implemented within the Projective Dynamics framework, using a quadrangular network of particles and constraints, to simulate soft bodies in real-time. The results show that in this particular case, the Red-Black Gauss-Seidel method outperforms other traditional iterative solvers in terms of convergence speed.

The results were achieved by creating a physics engine prototype, using a Verlet numerical integration scheme, parallel collision handling and three different types of iterative solvers; sequential Gauss-Seidel, parallel Jacobi and parallel Red-Black Gauss-Seidel. These solvers were then compared to each other. The physics engine as a whole was also compared to other contributions in the field. The quadrangular structure of the soft bodies resulted in real-time performance, at the cost of a moderate loss in precision.

Keywords: real-time, computer animation, physics-based animation, simulation, soft body dynamics, deformable bodies, projective dynamics, red-black gauss-seidel

Acknowledgements

We would like to thank Marco Fratarcangeli, our supervisor, for spending a considerable amount of time and effort guiding us. His input, knowledge and enthusiasm has been of tremendous value for the realization of this thesis.

Oskar Nylén and Pontus Pall, Gothenburg, June 2017

Contents

1
Introduction

The need to simulate deformable bodies in real-time can be found in many different areas, ranging from video games to medical simulation to interior design and beyond. While hardware advancements and increasingly efficient algorithms have allowed for impressive results, many problems are yet to be solved. This chapter introduces the topic of soft-body simulation and presents the aim, purpose and limitations of the thesis.

1.1 Background

The simulation of both rigid and soft bodies has been an active area of research for the last 30 years [Terzopoulos et al., 1987]. Traditionally, these simulations have been used in fields such as 3D animation and video games. While still being heavily used in those fields, the need for realistic simulations has grown, as a lot of industries that have not used them in the past are now finding ways to use them to achieve business value. Such areas include interactive surgical simulation [Bender et al., 2015], hair simulation [Rungjiratananon et al., 2010] and motion capture animation [O'Brien et al., 2011].

Along with the increase in interest in physics-based animation, the power of modern personal computers has also increased substantially over time, allowing realistic simulations to be run in real-time. Many attempts have been made to increase the interactivity of the simulations by inventing new, efficient methods.

A popular approach for simulating soft bodies is to model objects as a network of particles and constraints [Liu et al., 2013]. The constraints limit the movement of the particles in ways which plausibly simulates the way real soft bodies move when subjected to external forces. This model produces a large system of constraint equations, which needs to be solved in a few milliseconds in order for the simulation to sustain real-time interactivity. Thus, the manner in which the constraints are solved is a highly important factor for the field of soft body dynamics. In order to further speed up this process, the Graphical Processing Unit (GPU) can be used to parallelize calculations, increasing the number of calculations per second, thus lowering the computation time.

Even though speed is not as critical in offline simulation as in real-time simulation, it is still of great importance for productivity reasons. Thus, the advances in real-time

simulation can also be beneficial for areas which heavily rely on offline simulation, such as the movie industry and virtual prototyping.

1.2 Purpose

This thesis attempts to create a physically based simulation engine, which addresses the problem of creating stable and visually plausible simulations of soft bodies, while allowing for interactivity and stability. This is done by utilizing a constraint solving technique called Red-Black Gauss-Seidel, presented in section *3.2.3 Graph coloring*, with the aim of solving the underlying equation system in just a few milliseconds. The goal is to contribute to the overall knowledge within the area of real-time soft body dynamics.

1.3 Problem statement

As applications of physically-based animation become more common in a wide range of fields, from medical simulations to feature films, the demand for performance increases. At the same time, the required level of believability keeps rising.

The thesis will conclude whether or not a Red-Black Gauss-Seidel solver is viable for real-time physics simulation. This will be determined by answering the following research question:

> *Can a solver which utilizes the Red-Black Gauss-Seidel method be more suitable for real-time graphics applications than other traditional solvers, by enhancing interactivity and producing a more believable visual result?*

If the answer to this question is positive, and a Red-Black Gauss-Seidel solver can be used effectively, this might bring the field closer to achieving realistic looking real-time simulations of soft bodies.

1.4 Limitations

The proposed solver has the potential to accelerate the computational performance of the animation significantly. However, due to the intrinsic limitations of the solver, it can model only objects with a relatively simple structures, such as cloths and ropes. In order to simulate volumetric objects with shape conservation, other constraints, such as bending, would need to be implemented.

The goal of the thesis is primarily to create a solver, but there are various other concepts which needs to be implemented in order to test the solver and create a simulation. For example, collision handling and rendering has to be implemented in order to test the applicability of the solver in a practical environment.

Collision handling always poses a difficulty when creating real-time physics-based simulations, as it is a continuous problem solved in a discrete manner. Furthermore, tens of thousands of particles need to be simulated in real-time, which presents a problem regarding interactivity. Among the many different approaches in the literature, we chose the simple, yet effective in practice, approach presented in [Green, 2010].

Similarly, considerable amounts of time could be spent on achieving realistic rendering. The quality of rendering is limited to a degree which gives the viewer a good understanding of the deformation of the soft objects and how various objects are related to one another.

1.5 Outline

The list below briefly outlines the contents of each chapter of the thesis.

- **2 Previous work** - Presentation of notable research made in the field of soft object simulation.
- **3 Theory** - Review of the theory required for fast soft object simulation.
- **4 Methodology** - High-level descriptions of the methods applied in order to answer the research question and fulfill the purpose of the thesis.
- **5 Process** - In-depth presentation of implementation of the simulation engine.
- **6 Results** - Analysis of the results in terms of both performance and visual appearance.
- **7 Discussion** - Interpretation and explanation of the achieved results, and what they add to the field of soft object simulation.
- **8 Conclusion** - Reflection on the purpose of the thesis and the implications of the results.
- **9 Future work** - Discussion of possible options to expand beyond the limitations of the thesis.

2
Previous work

Within computer graphics, there are several types of simulation techniques for different types of objects. For objects which are not deformed during simulation, methods based on rigid body dynamics can be applied. However, there is also a need to simulate soft objects, which can be deformed by external forces in a physically plausible way. In order to do this, [Terzopoulos et al., 1987] found that the partial differential equations which determine both the shape and the motion of these objects needs to be solved. These equations can be discretized in several ways, producing a system of inter-dependent ordinary differential equations. One way of discretizing continuous shapes is to divide them into particles with constraints between them. The problem is then to satisfy each constraint based on the forces which are applied. This model has been used by [Provot, 1995], to simulate convincing non-elastic cloth, [Faure, 1999] to simulate interactive solid animation. Another notable example is the work of [Baraff and Witkin, 1998], which focused on creating a simulation system which could handle large time-steps for cloth simulation.

The idea has since then been developed to simulate a wide array of deformable bodies, using different types of constraints and constraint solving methods. One of the first examples of this being used in a commercial product, was presented by [Jakobsen, 2001]. This method uses springs which are infinitely stiff, called stick constraints, to simulate ragdolls and cloth. It was used in the game Hitman Codename 47, to great success.

The Position Based Dynamics framework [Müller et al., 2007] presents a model which shows many advantages of purposely disregarding velocities and applying changes over time directly on the positions of the particles making up the shape. In this approach, virtually any type of soft body can be modeled, by using various constraints. For example, one can use bending constraints to simulate cloth more convincingly, volume conservation constraints to keep the volume of tetrahedral meshes [Bender et al., 2015], and density constraints to simulate fluids [Macklin and Müller, 2013]. Position Based Dynamics is fast, easy to implement and controllable. Similarly, Autodesk's Nucleus solver was presented in 2009, which also makes use of constraints to simulate soft bodies [Stam, 2009]. Nucleus was developed independently of Position Based Dynamics, but the two methods have some similarities.

However, Position Based Dynamics has received some critique for not being derived from proper physical principles. In [Bouaziz et al., 2014] the Projective Dynamics framework is presented. The purpose of Projective Dynamics is to fuse together

the simplicity and efficiency of Position Based Dynamics, with physically correct methods such as the Finite Element method. In Projective Dynamics, constraints are defined with energy potentials, which are derived from real physical laws. This work builds upon the work by [Liu et al., 2013], where Hooke's law is applied to a mass-spring system. In [Liu et al., 2013] an alternating minimization technique is used to solve the constraints. This technique is later used in Projective Dynamics, and is generalized for any kind of constraint. The performance of the framework has been significantly increased by using the Chebyshev semi-iterative method [Wang, 2015; Wang and Yang, 2016]. However, this method may introduce artifacts for simulations that require a small number of solver iterations. Recently, Projective Dynamics has been reformulated as a quasi-Newton method, which enables simulation of a large group of hyper-elastic materials with even faster convergence [Liu et al., 2017].

In later years, focus has been made towards solving large systems of equations in the fastest way possible. In [Müller et al., 2007], the constraints are solved in an iterative, sequential Gauss-Seidel fashion. However, with the increased demand for interactive physics and the improvement of graphical processing units (GPUs), effort has been made to parallelize the constraint solving process. In [Macklin and Müller, 2013] a Jacobi solver was used for fluid simulation, a method which is also trivially parallelizable. However, Jacobi solvers have the slowest convergence speed among iterative solvers, so they require a high number of iterations to provide a solution. In case of objects with many constraints, this may severely affect performance. This problem is much less significant with Gauss-Seidel solvers. Research by [Fratarcangeli and Pellacini, 2015] has proven that Gauss-Seidel could also be effectively parallelized, by using a graph coloring technique with the Position Based Dynamics framework. In this approach, the convergence speed per iteration of Gauss-Seidel is preserved, while heavily reducing the computation time spent per iteration. In later work, this technique was also used with Projective Dynamics along with a randomized coloring technique [Fratarcangeli et al., 2016].

3

Theory

Real-time physics-based simulations have two conflicting goals: simulating accurate behavior while remaining interactive. Achieving both goals perfectly is very difficult and not yet achieved, thus the actual objective is finding a satisfactory compromise between accuracy and speed. Implementing a particle based physics engine involves a set of different techniques and methods that all need to relate to these conflicting aspects. The theory behind these methods will be presented in this section. Both methods which were later implemented, and methods that were discarded will be discussed.

3.1 Mass-spring systems

One of the most basic ways of modeling soft bodies is to discretize an object into a network of particles and springs, also known as a mass-spring system. This provides an intuitive, simple, yet powerful way to model deformable objects [Liu et al., 2013]. The springs act as constraints, which limit the motion of the particles by keeping them within a set distance from each other. In [Liu et al., 2013], the spring forces are determined by Hooke's law, while in [Müller et al., 2007] they are mainly geometrical.

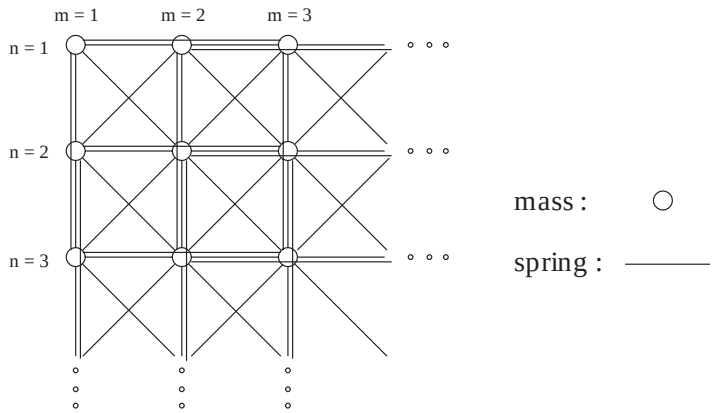

Figure 3.1: *Example of a mass-spring system from [Provot, 1995].*

More complex models consider phenomena such as bending, volume preservation and more to allow for increasingly sophisticated soft body simulations. However, in general, the models still consist of a network of particles and varyingly advanced constraints.

3.2 Linear equation system solving

The constraints can be expressed as a sparse system of linear equations. In each equation, the particle position is the unknown. A set of n equations and n unknowns can be expressed as:

$$\mathbf{Ax} = \mathbf{b} \tag{3.1}$$

where $\mathbf{A}$ is the coefficient matrix, $\mathbf{b}$ is the right-hand side vector of knowns and $\mathbf{x}$ is the vector of unknowns [Saad, 2001].

One way of solving this system is to calculate the inverse of $\mathbf{A}$. However, computing $\mathbf{A}^{-1}$ is very slow, and there is no guarantee that the inverse actually exists. The complexity of calculating the inverse matrix is generally $O(n^3)$, using Gaussian elimination. In the case of soft body dynamics, n usually is far above 10 000, which makes this method unviable [Kim, 2016].

Linear iterative solvers, such as the Jacobi and Gauss-Seidel methods can also be used. For each iteration, these solvers calculate an increasingly better approximation to the correct solution. When the purpose of the simulation is human visual consumption, the correct solution is often not needed, only a good approximation. If the result is close enough to the actual solution, the human eye will not perceive the error. Since iterative solvers usually can produce a good approximation faster than the previously mentioned method, this is most commonly used in real-time physics simulation.

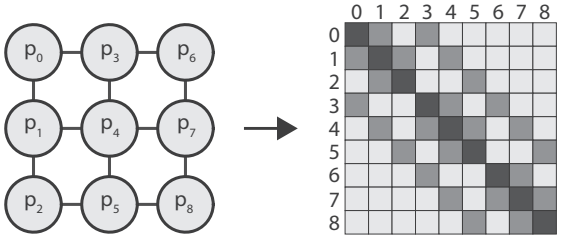

Figure 3.2: *A structure with 9 particles and 12 constraints is represented by the connectivity matrix on the right hand side. Each row constitutes one equation, and each equation is defined in terms of the constraint coefficients.*

In the following two sections, the Jacobi and Gauss-Seidel iterative methods are presented. These are two of the most commonly used iterative methods for solving large systems of equations. The lexicographical definitions will be presented, but there are also many different variations and combinations of the methods [Saad, 2001].

3.2.1 Jacobi iterative method

One method to solve the linear equation system previously presented is by using the Jacobi method. This method calculates an increasingly accurate approximate solu-

tion for the whole system given an initial guess, then updates the system. Thus, the solution of each equation uses the initial values of the previous iteration. Therefore, the equations can be solved independently and in any order. Practically, this means that the system can be solved in parallel.

The following equations describe the Jacobi iterative method [Barrett et al., 1994]:

$$\mathbf{x}_i^{(1)} = \frac{\mathbf{b}_i}{\mathbf{a}_{ii}} \tag{3.2}$$

$$\mathbf{x}_i^{(k)} = \frac{\mathbf{b}_i - \sum_{j \neq i} \mathbf{a}_{ij} \mathbf{x}_j^{(k-1)}}{\mathbf{b}_{ii}} \tag{3.3}$$

In equation 3.1 the initial values are set to a guessed value. In equation 3.2 the next iteration k is calculated, from the known vector b and the sum of equations on which $\mathbf{x}_i$ depend. Subtracting this sum from $\mathbf{b}$ and dividing by the diagonal value of the corresponding row in $\mathbf{A}$ yields the new position of $\mathbf{x}_i$. If the Jacobi method is applied to the structure presented in section *3.1 Mass-spring systems*, the solving could be visualized as follows:

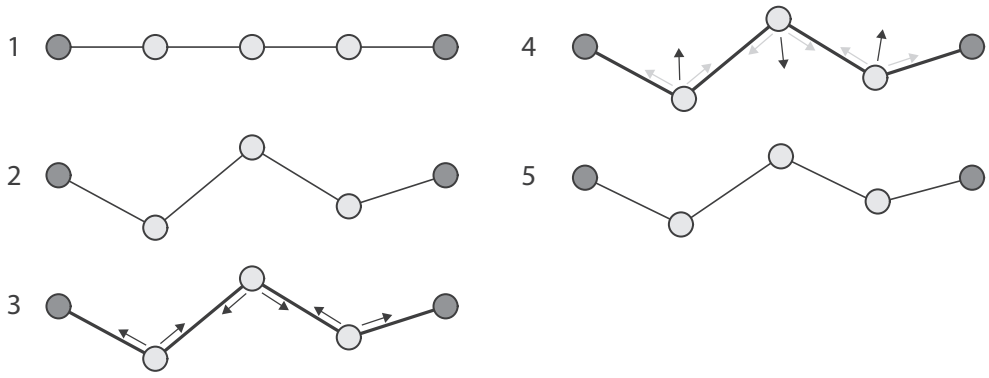

Figure 3.3: *Example of one Jacobi iteration. 1: The rest state of the system. 2: The particles gets displaced by an external force. 3: For each particle, deltas are calculated. 4: The deltas are averaged by the number of constraints. 5: The positions are updated.*

All the positional deltas can be calculated for each constraint in the system in parallel. For each particle, the deltas are accumulated, and averaged to provide one positional change per particle, which also can be applied in parallel. This parallelization means that each iteration is faster than the Gauss-Seidel method. However, the convergence per iteration is not as substantial as with Gauss-Seidel.

3.2.2 Gauss-Seidel iterative method

The Gauss-Seidel method solves each equation sequentially. Each new equation uses the result of the last equation, and directly updates the result. The Gauss-Seidel iterative method can be defined as follows [Barrett et al., 1994]:

$$\mathbf{x}_i^{(1)} = \frac{\mathbf{b}_i}{\mathbf{a}_{ii}} \tag{3.4}$$

$$\mathbf{x}_i^{(k)} = \frac{\mathbf{b}_i - \sum_{j<i} \mathbf{a}_{ij}\mathbf{x}_j^{(k)} - \sum_{j>i} \mathbf{a}_{ij}\mathbf{x}_j^{(k-1)}}{\mathbf{a}_{ii}} \tag{3.5}$$

In equation 3.4, the initial values of the system are calculated. In equation 3.5, the next iteration k is calculated, from the known vector b and the equations on which $\mathbf{x}_i$ depend. The difference between this method and the Jacobi method is that Gauss Seidel considers the previous computations from the current iteration, while Jacobi uses the values of the previous iteration for all computations. Note that the leftmost sum in the numerator uses values of $\mathbf{x}$ from iteration k, while the rightmost sum uses iteration $k-1$. This is in contrast to the Jacobi method which only computes one sum using the values of iteration $k-1$. A visual example of when the Gauss-Seidel method is applied to a mass-spring system can be seen below:

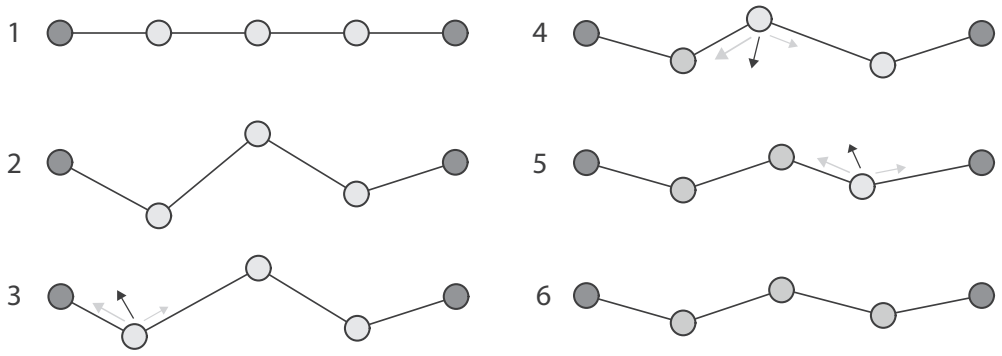

Figure 3.4: *Example of one Gauss-Seidel iteration. 1: The rest state of the system. 2: The particles are displaced by external forces. 3-6: Each equation is sequentially solved.*

The Gauss-Seidel method finds a better approximation in fewer iterations than the Jacobi method, but is less straightforward to parallelize due to the interdependent equations. This results in more time-consuming iterations, and therefore a less efficient solution. This is the reason behind wanting to parallelize Gauss-Seidel.

Successive over-relaxation In order to increase the convergence speed of the Gauss-Seidel method, a technique called successive over-relaxation can be applied. This technique takes the value computed by the standard Gauss-Seidel method, and extrapolates it in the direction given by $\mathbf{x}^{(k)} - \mathbf{x}^{(k-1)}$. The amount of extrapolation depends on the relaxation factor ω.

$$\mathbf{x}^{(k)} = \mathbf{x}^{(k)} + \omega(\mathbf{x}^{(k)} - \mathbf{x}^{(k-1)}) \tag{3.6}$$

If ω is set to 1, the successive over-relaxation method is equivalent to standard Gauss-Seidel iteration. The method is shown to converge for a relaxation factor within the interval $(0, 2)$ [Barrett et al., 1994].

3.2.3 Graph coloring

The structure of particles and constraints can be seen as a graph. As such, methods from graph theory can be applied to streamline various computations. One such method is graph coloring, which aims to mark all elements of a graph with a certain color such that no interconnected elements share the same color. By coloring the particle system, particles can be partitioned into groups which can be solved independently, allowing for parallel solving [Saad, 2001]. The naive way of achieving this is to assign a different color to each particle. However, this would not accomplish much since the number of parallel solving steps is equal to the number of particles, making parallelism pointless. Ideally, we want a small number of similarly sized partitions to allow for maximal parallelization.

Recently, attempts have been made to parallelize Gauss-Seidel without having to use concurrency control techniques which slow down the execution. The concurrency problem occurs when the Gauss-Seidel solver tries to manipulate the same particle position from multiple threads. Multiple threads accessing the same particle at the same time can be avoided by using atomic operations, but this heavily slows down the parallel execution [Bender et al., 2015].

However, these concurrency problems can be avoided by using graph coloring. A graph coloring technique to tackle this problem was proposed by [Fratarcangeli et al., 2016], called Vivace. In this approach, each particle is colored using a parallel graph coloring technique, which allows for a large maximum degree of the constraint graph. The technique utilizes randomization and a heuristic to avoid colors conflicting with each other. Vivace produced impressive results, being able to handle hundreds of thousands of constraints, while keeping interactivity and avoiding any noticeable visual artifacts.

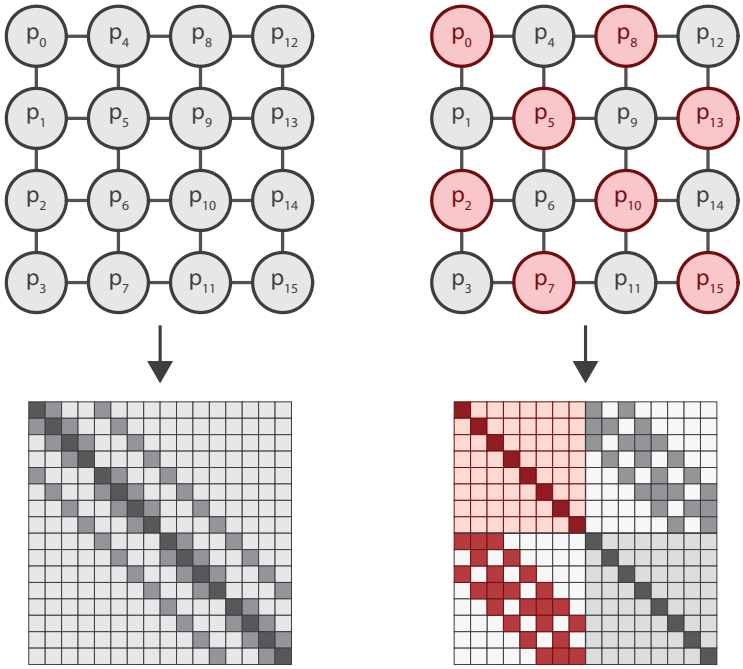

Figure 3.5: *On the left, the unpartitioned system and its corresponding coefficient matrix. On the right is the same system colored red and black.*

A simple graph coloring method is the red-black method, in which the elements of the graph are split into two independent partitions. This approach is described in detail by [Saad, 2001]. A comparison between a unpartitioned graph and a Red-Black partitioned graph can be seen in Figure 3.5. The graphs presented in the figure are so called grid graphs, which belong to the family of bipartite graphs. The chromatic number, i.e. the smallest number of colors needed to color the graph such that no elements of the same color share an edge, of a bipartite graph is 2 [Godsil and Royle, 2001]. When introducing more complex constraints, the chromatic number grows, thus forcing the use of more sophisticated graph coloring techniques such as the one presented by [Fratarcangeli and Pellacini, 2015].

3.3 Numerical integration

For any physics simulation based upon continuum mechanics, a numerical time integration scheme is required in order to update the state of the simulated objects. The integrator takes a time-step value and computes the new positions of the particles based on the velocity and external forces such as gravity, wind and friction. If the desired frame-rate of the simulation is 60 frames per second, the simulation time window is $1/60s \approx 0.016$ seconds, excluding the time consumed by the rendering step. In order to achieve a real-time simulation, the time-step chosen should be dependent on this value. Thus, when choosing a time-step we can simply set it to 0.016 seconds and perform one numerical integration step during each frame. Another option is to select a smaller multiple of 0.016, such as 0.004, and instead perform

several (in this case 4) numerical integration steps during each frame. The size of the time-step gives rise to an important trade-off - precision versus performance. A high time-step allows for fewer numerical integration steps, but also results in lower precision, making collision detection prone to errors. A low time-step gives higher precision, but forces the numerical integration step to be computed several times during each frame, which in turn negatively impacts performance.

Another aspect which needs to be taken into account, for particle-based physics simulations in particular, is that some integration methods can produce unstable behavior [Provot, 1995]. There are methods that produce very accurate and stable results, such as higher order Runge-Kutta methods, but these are generally hard to implement and more computationally expensive [Eberly and Shoemake, 2004]. The general difficulty which gives rise to these problems, is that we are trying to simulate continuous phenomena in a discrete manner. What this means is that the result we get from the numerical integration is only an approximation of an integral [Kim, 2016].

3.3.1 Euler integration

One of the most basic approaches to numerical integration is the explicit Euler method. This method is also called forward Euler, and is a very popular time integration method, due to its simplicity [Eberly and Shoemake, 2004].

Using Euler integration, applying Newton's laws of motion, the new position $\mathbf{x}^{(t+h)}$ and velocity $\mathbf{v}^{(t+h)}$ are computed by applying the following equations:

$$\mathbf{x}^{(t+h)} = \mathbf{x}^{(t)} + \mathbf{v}^{(t)} \cdot h \tag{3.7}$$

$$\mathbf{v}^{(t+h)} = \mathbf{v}^{(t)} + \mathbf{a} \cdot h \tag{3.8}$$

Here, h is the time-step, and $\mathbf{a}$ is the acceleration computed using Newton's second law of motion: $F = m \cdot \mathbf{a}$ (where F is the accumulated force acting on the particle).

This method is trivially implemented, but is inherently unstable. This instability is due to the size of the time-steps. The time-step has to be very small, otherwise, at high velocities and/or forces, there might be inaccuracies which feedback on themselves, making the error bigger and bigger. Eventually, the system becomes too unstable and explodes or breaks apart. Having small time-steps is therefore more or less a requirement for Explicit Euler to give accurate results [Eberly and Shoemake, 2004]. The error is directly proportional to the size of the time-step, meaning that if the time-step is halved, the error is also halved.

3.3.2 Verlet integration

Another approach, which does not require storing the velocity is the Verlet method. Instead of storing the velocity, it is calculated by approximation, which takes place implicitly. This is done by storing the previous position, and keeping the time step constant.

$$\mathbf{x}^{(t+h)} = 2\mathbf{x}^{(t)} - \mathbf{x}^{(t-h)} + \mathbf{a}^{(t)} \cdot h^2 \tag{3.9}$$

$$\mathbf{x}^{(t-h)} = \mathbf{x}^{(t)} \tag{3.10}$$

The current velocity can be written by approximation by rearranging the formula the following way:

$$2\mathbf{x}^{(t)} - \mathbf{x}^{(t-h)} = \mathbf{x} + (\mathbf{x}^{(t)} - \mathbf{x}^{(t-h)}) \tag{3.11}$$

where $\mathbf{x}^{(t)} - \mathbf{x}^{(t-h)}$ is the distance traveled the last time-step. Verlet is both more accurate, and stable than explicit Euler. With Verlet integration, the error is kept at $O(h^4)$ [Eberly and Shoemake, 2004]. This integration technique is also easily implementable, and is not computationally expensive [Jakobsen, 2001]. Another benefit from Verlet integration is that the size of the time-steps can be larger than when using explicit Euler.

3.4 Collision handling

To make objects interact with each other in a natural manner, collisions need to be detected and handled accurately. This involves finding any intersections between objects, calculating collision displacements and applying them to the objects. As always in real-time rendering, the time budget is limited, making a parallel solution more appropriate for the task.

In order to achieve fast collision detection, it is important to only examine objects that are near each other and therefore susceptible to collision. Normally, this is done by employing two phases: one for determining which objects are likely to collide (broad-phase) and another for detecting the exact collisions (narrow-phase) [Le Grand, 2007]. This also allows for the treatment of different groups of particles independently, making parallelization possible. To efficiently determine which particles are within collision distance of each other, a uniform grid can be used to discretize the simulation scene during the broad-phase.

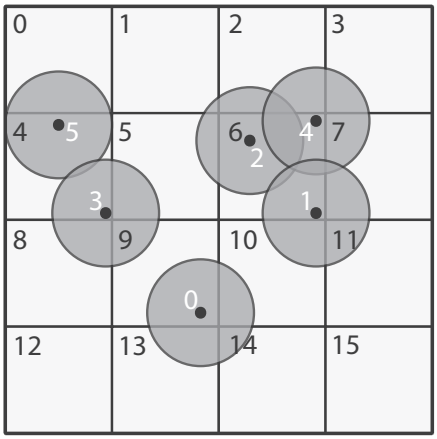

Figure 3.6: *The spacial subdivision into a uniform grid. The indices of the cells and particles are specified with black and white text respectively. (Courtesy of NVIDIA)*

A list of all the object-cell pairs is stored, sorted by the cell id. Then, in the narrow-phase, all potential collisions for each cell are checked, and a response is calculated and applied if the objects are within a set distance from each other. This distance is usually equal to the sum of the particles radii.

Another problem that needs to be addressed is that the collision detection only works with the positions provided by the numerical integration and the solver. Since this is done in a discrete manner, there are risks of missing collisions. With larger time-steps, the problems increase. There are methods that try to address this problem, but they are not in the scope of this thesis.

3.5 Position Based Dynamics

Position based dynamics is a popular framework for physics-based animation [Müller et al., 2007]. Many different physics engines utilizes position based dynamics, including PhysX, Havok Cloth and Bullet [Bouaziz et al., 2014]. There are many factors that contribute to this popularity. Position Based Dynamics provides good stability, which is usually one of the biggest problems with physics-based animation. Another aspect that helps to explain the popularity of the framework is that it is easy to understand and implement [Müller et al., 2007].

In Position Based Dynamics, a deformable object is represented by n particles and m constraints [Müller et al., 2007]. Each particle has a position, velocity and mass. Each constraint has a cardinality n_j, which states the connectivity of the constraint. A constraint also has a function C_j, which describes the manner in which the constraint will need to be solved. The constraint has a set of particle indices $i_1 - i_n$, which states which particles are connected with the constraint. The stiffness is also described by a stiffness parameter. The solver, in the original implementation, uses a Gauss-Seidel type iterative solver. In later implementations a Jacobi solver has also been used [Macklin and Müller, 2013].

While Position Based Dynamics is generally a popular approach, it has some drawbacks. It is not rigorously built upon mechanical principles, which makes it hard or even impossible to correctly combine with parameters derived from physical measurements [Bouaziz et al., 2014]. Because of this, parameter tuning is an inherent problem when working with position based dynamics. Additionally, since the stiffness of the simulated model is not independent of the time-step and number of solver iterations, modifying these parameters without affecting the stiffness is impossible [Bender et al., 2017].

3.6 Projective Dynamics

The Projective Dynamics framework was presented by [Bouaziz et al., 2014] as a way to bridge the gap between Position Based Dynamics and finite element methods, thus providing a more accurate model for simulating soft bodies. In projective dynamics, a constraint is described with a potential energy, rather than the simplified geometric distance, as in Position Based Dynamics. The potential is defined by an energy function derived from physics. In the case of the spring constraint, Hooke's law is used. This energy needs to be minimized for the constraint to be satisfied [Bouaziz et al., 2014]. In Projective Dynamics, the minimization technique from [Liu et al., 2013] is used, which finds optimal spring directions in a local step, then finds the correct node positions in a global step.

All these features combined produces a result which is superior to Position Based Dynamics in terms of visual believability, while still maintaining good efficiency. To summarize, the main difference between Position Based Dynamics and Projective Dynamics is the way the constraints are defined, and that Projective Dynamics implements the alternate optimization technique used by [Liu et al., 2013].

3.7 Parallel computing

Physical barriers such as power usage, heat dissipation and transistor size have in recent years forced CPU manufacturers to focus less on increasing clock speeds and more on putting multiple cores on their processors [Deng, 2013]. Following this development, programs need to be parallelized in order to fully utilize the processor's capabilities. While parallel computing has not always been an important matter on the CPU, the graphics processing unit (GPU) is parallel by design due to the rendering pipeline. Even though the GPU and its API's were originally tailored towards generating bitmaps for the display, this did not stop programmers from performing other, more general purpose programming tasks on it. As the interest in general purpose programming on the GPU increased, software and hardware was developed to make these endeavors more convenient [Sanders and Kandrot, 2010].

When processing large amounts of independent objects, such as vertices, pixels or particles, the speed benefits of parallel computing can prove to be immense. The

GPU provides hardware multi-threading, enabling the execution of many threads in parallel. The GPU is also provided with a higher number of transistors than the CPU, in order to increase the data processing capabilities [Nvidia, 2017].

CUDA NVIDIA provides an interface for programming their GPUs. This interface, called CUDA (short for Compute Unified Device Architecture), is a C/C++ API for general-purpose computing on graphics processing units (GPGPU), and was presented in November 2006 [Nvidia, 2017]. CUDA enables the programmer to write both host code, which runs on the CPU and device code which runs on the GPU [Kim et al., 2012]. The device code is written in CUDA C functions, called kernels. When a kernel is called, CUDA launches a programmer-defined number of threads on the GPU, which all execute the kernel in parallel. This is in contrast to regular functions in languages such as C or Java, which only run once per call. CUDA provides the possibility to use the GPU not only for graphics, but also other computationally heavy tasks.

For example, two arrays can be trivially added and saved to a third array, in parallel, using CUDA. This is done by using the unique indices of the threads as access indices for the arrays.

Listing 3.1: An example of a CUDA kernel adding two arrays (A and B) together to a third array (C)

```
__global__ void
vectorAdd(const float *A, const float *B, float *C,
          int numElements)
{
    int index = blockDim.x * blockIdx.x + threadIdx.x;

    if (index < numElements)
    {
        C[index] = A[index] + B[index];
    }
}
```

Here, a unique thread index is calculated based on the size of a block, the index of that block and the block-local thread index. A block, also called thread block, is a collection of threads that reside on the same core of the GPU, and have access to the same shared memory [Nvidia, 2017]. This provides a way of assigning each array index to an individual thread. The function call to the kernel can be seen in Listing 3.2. Furthermore, the blocks are organized in grids. The grid, block and thread hierarchy are illustrated below:

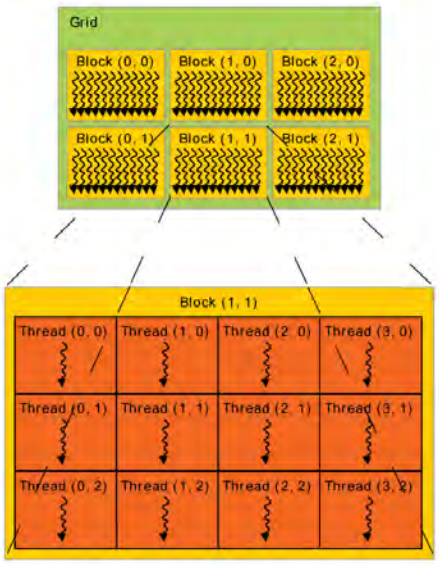

Figure 3.7: *Visualization of the thread hierarchy; with grids, blocks and threads. (Courtesy of NVIDIA)*

Listing 3.2: Code to call the CUDA kernel vectorAdd

```
int threadsPerBlock = 256;
int blocksPerGrid =
        (numElements + threadsPerBlock - 1)/ threadsPerBlock;

vectorAdd<<<blocksPerGrid, threadsPerBlock>>>
        (d_A, d_B, d_C, numElements);
```

In addition to this, memory management is a vital part of programming on the graphics card. CUDA provides several functions to allocate memory on both the CPU (host) and GPU (device), along with methods to easily transfer this data back and forth. However, since the memory transfer is one of the biggest bottlenecks, it is crucial to only use the GPU when the benefits outweigh the cost of the communication between the CPU and GPU [Kim et al., 2012].

4
Methodology

This chapter will present how the process of answering the research question and how the purpose of the thesis was fulfilled. The chapter will treat both specific aspects of the development and more general aspects of the study. First, the workflow of the research is described in detail. After that, the manner in which the research question was answered will be presented. Finally, the method used to evaluate the generated data is described.

4.1 Research workflow

The research was divided into three phases. First, a literature study was conducted. This provided the knowledge needed to develop the first iteration of the prototype, thus creating a platform to build upon. The findings of the literature study can be seen in chapters *2 Previous work* and *3 Theory*.

After this initial phase, the development began. The attention was focused towards building a prototype, and improving it until a point of satisfaction was reached. First, a version of Position Based Dynamics using a parallel Jacobi solver was built. Since Position Based Dynamics basically is a simplified special case of Projective Dynamics, this was a way to create a proper platform for further development. This also gave good insight in how GPGPU computing works with CUDA. After this, Projective Dynamics with the Red-Black Gauss-Seidel solver was implemented. To be able to compare the RBGS solver to others, two additional solvers were implemented and tested:

- Parallel GPU Jacobi solver
- Sequential single-core CPU Gauss-Seidel solver

The entirety of the work was evaluated by conducting several experiments in a structured environment. These experiments provided the data needed in order to draw conclusions about the solver and answer the research question. The research workflow can be viewed in the figure below:

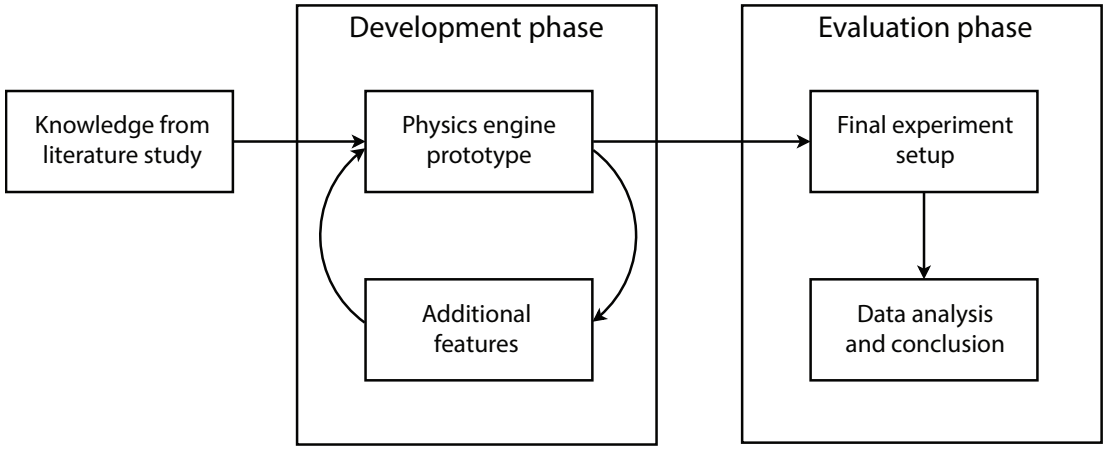

Figure 4.1: *Visualization of the three phases of research.*

The experiments were conducted by comparing the performance of the RBGS solver to the other solvers. The main variables of this comparison were the squared residual error over iterations and squared residual error over time. This will be further explained in section *4.3 Result evaluation*

4.2 Prototype

In order to answer the research question, a high-fidelity software prototype was developed. While a solver could certainly have been developed and tested from a performance-centered perspective without a visual representation, the second part of the research question necessitated a relatively sophisticated rendering of the solved constraints. It was of great importance that the tools and methods used to create the prototype were the most suitable for the context, real-time computer animation.

There were several techniques that could have been utilized to realize the prototype. The most important aspect for this particular context was performance. This excludes many computer languages such as hosted, interpreted or garbage-collecting languages. We chose to use the industry standard language for developing high performance applications such as video games and real-time simulations, which is C/C++.

To further increase performance, it was essential to maximize the number of floating-point operations per second (FLOPS). One method to do this is to utilize multiple cores of the CPU. This can be achieved with tools such as OpenMP.

OpenMP is a cross-platform API which allows programmers to use shared memory multiprocessing by annotating programs in C, C++ and Fortran [Chapman et al., 2008]. By using multiple threads in parallel, the number of FLOPS performed can be increased significantly. However, compared to the GPU, the number of physical parallel processing units is small. Thus, in order to increase performance, the highly

parallel solver of the prototype was implemented on the GPU.

There are many different frameworks available for GPGPU programming. OpenCL is a open framework developed by Khronos Group. It provides a standard for parallel computing, and each implementation of is provided by the manufacturers of the GPUs.

NVIDIA provides their own API for GPGPU programming, called CUDA. This framework is presented in depth in section *3.7 Parallel computing*. Since NVIDIA tailor their API to the hardware they manufacture, CUDA is generally more performant than OpenCL. OpenCL has also been shown to produce significantly more run-time overhead [Demidov et al., 2013].

Apart from performance related considerations, factors such as extensibility and modularity also had to be taken into account. Since a number of solvers were to be developed and compared, the ability to quickly switch between solvers was required. This was handled via a configuration file framework, described in section *5.1 Setup*, which allowed for the creation of scenes with different parameters and contents.

4.3 Result evaluation

There were two types of result evaluation conducted in order to analyze the physics engine. The first being a performance evaluation, where the evaluated data was easily quantifiable as the squared error. The other aspect which had to be analyzed was the visual results. When doing this result evaluation, the experimental design is crucial, as well as addressing the validity threats correctly.

The goal of the performance evaluation was to analyze how the proposed Red-Black Gauss-Seidel solver would perform compared to other iterative solvers, with respect to the convergence speed. This was done by comparing the total internal squared residual error over both iterations and time, between the different solvers. The error was measured using the following equation:

$$totalSquaredError = \sum_{N}(|\mathbf{q}_a - \mathbf{q}_b| - L)^2 \tag{4.1}$$

In the equation, the error is calculated as the distance between the positions, $\mathbf{q}_a$ and $\mathbf{q}_b$, of the connected particles minus the rest length L. The resulting values are squared to give the total internal squared error of the system. Two widely different scenarios were used when conducting these experiments. A more detailed description and the results are presented in section *6.1 Performance*.

When evaluating the visual result, there is no easily quantifiable metric. Therefore the results produced by our physics engine were compared to other examples in literature and online. These results are presented in section *6.2 Test cases*.

Validity threats One concern with the experiments was how valid the results are. There are typically four different types of validity threats that need to be addressed. These are conclusion, internal, construct and external validity threats [Wohlin et al., 2000].

An internal validity threat was the time measurement. The time consumed when writing to disk had to be compensated for correctly, when logging the performance. It is also essential to always run the experiments on the same device, with as few interfering background processes as possible. All experiments were thus executed on a computer with a Intel Xeon E5 CPU, and a NVIDIA GTX970 GPU.

A conclusional validity threat which had to be addressed, was that even though the physics engine might produce less total error, other methods might have better results. Other methods can have a larger amount, or other types of constraints, which produce a larger internal error, although the visual error is small.

When analyzing the visual result, an obvious construction validity threat, which we had to be aware of, was that we would be biased in our analysis. There is not much that can be done about this, other than be aware of it and try to prevent it.

5

Process

In this chapter the implementation of the prototype is presented. This includes a thorough explanation of the creation of the physics engine and its components, along with motivations for the decisions made during the process. There were many different sub-projects within the main project, some of which are not included in the final result. However, they will all be described here, as everything that was developed directly or indirectly helped answer the research question.

5.1 Setup

Before applying the theoretical knowledge described in chapter *3 Theory*, a number of optional steps are taken in order to allow for modular development, rapid prototyping and performance measurement. The steps taken to achieve this are described below.

Setting up a physics engine involves the storage, manipulation and usage of many parameters. To avoid a recompilation between each parameter change, a configuration file framework was developed. The framework was set up using RapidJSON, a fast JSON parser and generator, enabling us to easily write and read configuration files with parameters of all primitive data types.

Listing 5.1: Example of a configuration file

```
{
    "softObjects": [
        {
            "src": "100cloth",
            "position": [0, 0, 0],
            "color": [255, 94, 85],
            "fixed": [0, 99]
        }
    ],
    "solver": "PD_RBGS",
    "innerIterations": 4,
    "outerIterations": 4,
    "timestep": 0.002,
    "stiffness": 1000000,
    "gravity": 9.82,
    "cameraPos": [0, 0, -3],
    "tearingThreshold": 3,
    "modelPath": "model/sitting/sitting"
}
```

In the example above, we simulate a cloth represented as a signed distance field in the file called 32cloth. The position and color of the cloth are also provided. The list of integers named "fixed" determines which particles should remain unaffected by any internal or external forces, effectively making them stationary during simulation. Various constants and properties describing the environment are also defined, such as the gravitational constant. Additionally, here we define if and in that case which model animation should be present in the scene.

Logging was implemented using Loguru, a header-only logging library. This tool was used mainly for outputting various performance characteristics of the different solvers. Since the performance is one of the most important aspects to consider to answer the research question, it is crucial to the project to have a reliable logging tool.

5.2 Data structures

Data layout is critical when pursuing high performance. Generally, the more compact and ordered the data is, the more efficiently it can be accessed by the processor. Therefore, it is important to choose the right model when building real-time systems. At a fundamental level, there are mainly two different layouts one can choose from, namely structure of arrays (SoA) and arrays of structures (AoS). Both of the layouts have their advantages and disadvantages. There are also various modifications and combinations of the two [Strzodka, 2012]. Below, the essentials of the two concepts are described.

Structure of arrays When working with Single Instruction, Multiple Data (SIMD) units such as the GPU, SoA is generally preferable [Strzodka, 2012]. In the figure below, an example of SoA is visualized.

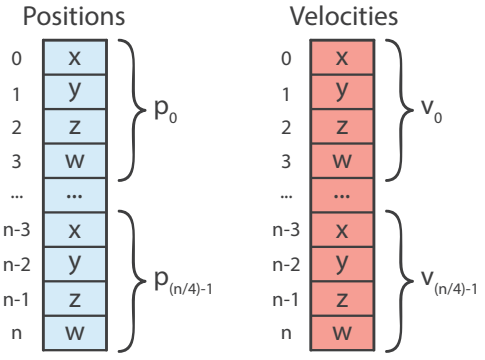

Figure 5.1: *Positions and velocities using the SoA data layout.*

Although the usage of SoA results in superior performance when it comes to GPGPU programming, it carries some difficulties. For example, the indexes of the arrays

needs to be synchronized correctly, and this way of modeling real world objects is sometimes counter-intuitive.

Arrays of structures Most computer languages have good support for the AoS layout. The manner in which objects in a object oriented language are defined could be viewed as a case of the AoS structure. An example of this would be to have a structure called *particle*, which contains all information needed to represent an individual particle. Visualizations of this layout can be seen in figures 5.2 and 5.3.

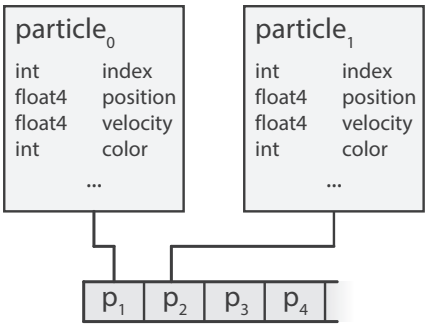

Figure 5.2: *An array containing particle structures. Each structure contains its own position, velocity and color data.*

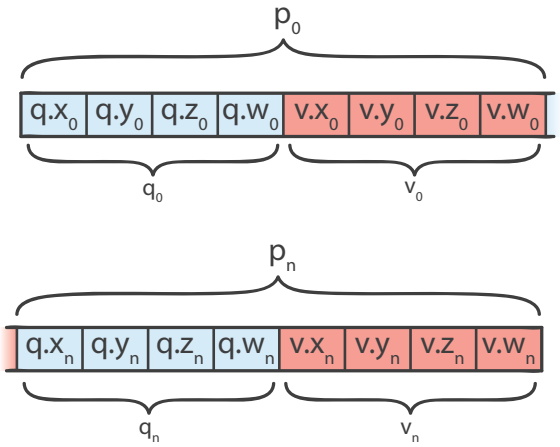

Figure 5.3: *Actual memory layout for the example in Figure 5.2.*

AoS is often seen as more intuitive and easier to implement than SoA, at the cost of performance. This is due to the data being scattered across different locations.

Implementation Due to the improved performance of SoA, the data structuring of the engine was implemented in a SoA fashion. Structures such as positions, previous positions and velocities were all stored as arrays of floats. This was done instead of defining a specific particle data structure containing this information for one particle. The difference can be seen in figure 5.4.

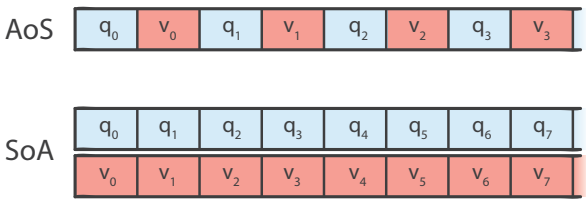

Figure 5.4: *SoA memory usage compared to AoS.*

Similarly, the constraints themselves are stored in an array. For each particle, the constraint array stores 6 particle indexes, which corresponds to the particles which it is connected to. A particle can at maximum be connected to 6 other particles, one in each positive and negative direction. If there is no constraint in a direction, the index value is replaced by a negative value.

The ambition with using this layout for all properties, instead of storing them in structures, was to improve performance. While no explicit performance tests between SoA and AoS were made, it is safe to assume that the SoA approach makes for more efficient memory access.

5.3 Simulation

This section describes and motivates the implementation of all the simulation components. First an overview is presented, and each component of the simulation overview is then explained in detail in the subsequent sections.

Algorithm 1: Simulation overview

```
1  scene initialization
2  particle system initialization
3  while running do
4      for k iterations do
5          numerical integration
6          constraint solving
7          collision handling
8      end
9      user input
10     rendering
11 end
```

The first step of the simulation is to read the configuration file, and load resources into memory. Based on the parameters in the file, different components of the simulation system are activated. When the initialization phase is finished, the simulation loop starts. In it, k is the number of physics simulation steps to perform between each rendered frame. Inside each physics step, numerical integration, constraint solving and collision handling is performed. Between the physics simulation and the rendering of the frame, user input is taken into account. If we want our system to

run in real-time at 60 frames per second, everything within the outermost loop has to be computed in approximately 16 milliseconds.

5.3.1 Initialization

The simulation initialization step involves allocating memory for simulated objects, such that the data can be efficiently transferred to the graphics card. By using information gathered from the configuration file we can determine the number of particles and constraints needed for the simulation. We also need to create particle indices and distance constraints between adjacent particles. This process, including prerequisite steps, is detailed below.

Voxelization A deformable body in the physics engine is represented by a number of particles. However, the most common way of representing objects in computer graphics is with a triangulated mesh. For example, a cube is modeled using 12 triangles (2 triangles per face), and the vertex order is described by a set of indices.

Since this is the standard way of representing an object in computer graphics, an effective method to convert these types of meshes into a set of particles, while still maintaining a reasonably high resolution, is needed. As explained in s ection *3.2.3 Graph coloring*, the structure of particles and constraints must form a grid graph, so that only two colors are needed in the coloring step. A common way to achieve this is by discretizing the mesh into volumetric pixels, or voxels. This is done by sampling the mesh in a uniform grid. One of the most useful structures that can be used for this purpose, is a signed distance field.

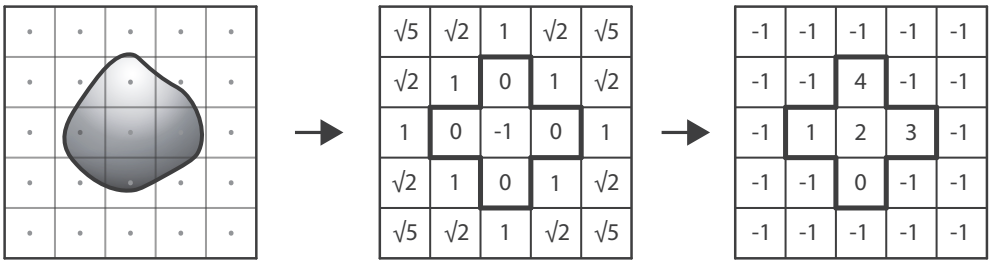

Figure 5.5: *To the left: A triangular mesh. In the middle: The signed distance values. To the right: The particle indices.*

A signed distance field subdivides the space surrounding a mesh into a uniform grid, and for each cell in the grid, stores the distance from the center of the cell to the surface of the mesh. The signed distance value is positive if the cell is outside the mesh, zero if it is on the surface and negative if it is inside the mesh. The SDFGen utility, made by Christopher Batty[1], was used to generate the signed distance field data. In our case, the actual distance of the signed distance value is not considered,

[1]https://github.com/christopherbatty/SDFGen

only the sign. This allows us to determine whether to place a particle or not. The voxelized mesh is stored in a 3 dimensional array, containing a negative value if there is no particle in the voxel, otherwise the index of the particle stored in that position.

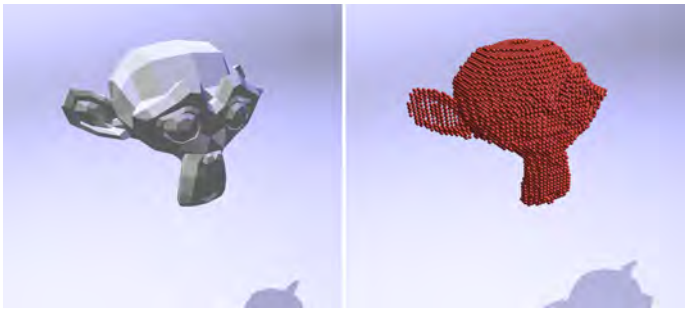

Figure 5.6: *To the left: A monkey head defined as a triangular mesh. To the right: The corresponding particle structure. Monkey model courtesy of Blender Foundation.*

When placing a particle, the color of that particle is also determined. The color is decided depending on the position of the voxel, using the following assignment:

$$color \Leftarrow (x + y + z) \,\%\, 2 == 0 \tag{5.1}$$

This ensures that particles are colored correctly in an alternating fashion. The color is stored as a boolean value.

Constraint creation Once the particle system has been initialized, constraints are created between each adjacent particle in the system. This is done by iteratively examining the neighboring positions of each particle in the voxelized mesh, to determine whether a constraint should be created or not. If the neighboring position contains a non-negative value, i.e. a particle index, a constraint between the particle being processed and the neighbor is created. During this process, the number of constraints per particle and the total amount of constraints for the entire system are also stored for later use in constraint solving.

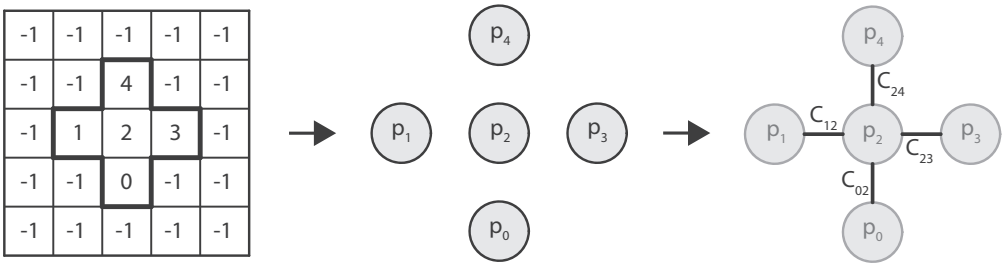

Figure 5.7: *To the left: The particle indices. In the middle: The particle placements. To the right: The generated constraints.*

5.3.2 Verlet integration

In all physics-based animation, the positions and velocities of the simulated objects are updated each time-step. This step is called time integration, and is further explained in section *3.3 Numerical integration*. The integration method chosen for this project was the Verlet method. This is due to Verlet enabling us to more or less discard the velocities out of the equation. It is also very accurate, stable, and easy to implement.

The Verlet method was implemented, by using the equations presented in section *3.3.2 Verlet integration*. Below is the implementation in pseudo code.

Algorithm 2: Verlet integration

```
1 for particles p do in parallel
2 |   if p is fixed then
3 |   |   return
4 |   end
5 |   f_ext ⇐ f_g − v f_damping
6 |   q ⇐ 2 q − q_prev + h^2 f_ext
7 |   q_prev ⇐ q
8 end
```

In steps 2-4 particles which have been marked as fixed in the configuration file are ignored. We calculate external forces in step 5, considering both gravity and damping coefficients. In step 6 the actual integration takes place. This is directly derived from the equation presented in section *3.3.2 Verlet integration*. Finally, we update the value of the old position in step 7.

5.3.3 Position based dynamics

Position based dynamics is a popular framework for soft-body simulation presented by [Müller et al., 2007]. In Position Based Dynamics, the velocity is implicitly updated by manipulating the positions of the particles. The framework provides good stability, at the cost of simplifying physical principles, thus producing a less convincing result. More details about the underlying theory of this framework is presented in section *3.5 Position Based Dynamics*.

Jacobi solver Position based dynamics was implemented using a Jacobi solver implemented on the GPU. The iterative Jacobi method is presented in section *3.2.1 Jacobi iterative method*. Pseudo-code for the implementation is presented in Algorithm 3.

Beginning at step 1 the solver runs i iterations. For each constraint, the displacement delta, Δ_c, is calculated from the two particles of the constraint, $\mathbf{q}_c^a$ and $\mathbf{q}_c^b$. This is done for all constraints in parallel (steps 3-4). The delta is then divided by two (the number of particles in the constraint), and stored in an array of deltas, indexed

Algorithm 3: Position Based Dynamics - Jacobi solver

1 **loop** *iterations* **times**
2 **for** *constraints c* **do in parallel**
3 $\mathbf{d} \Leftarrow \mathbf{q}_c^a - \mathbf{q}_c^b$
4 $\Delta_c \Leftarrow (|\mathbf{d}| - L)\,\hat{\mathbf{d}}$
5 $\Delta\mathbf{q}_c^a \Leftarrow \mathbf{q}_c^a + \Delta_c/2$
6 $\Delta\mathbf{q}_c^b \Leftarrow \mathbf{q}_c^b - \Delta_c/2$
7 **end**
8 **for** *particles p* **do in parallel**
9 $\mathbf{q}_p \Leftarrow \Delta\mathbf{q}_p/n_p$
10 **end**
11 **end**

on particles (steps 5-6). Since each particle may be part of several constraints, updating its accumulated delta directly could lead to concurrency issues. In order to resolve this issue, the built-in atomic addition function was used. This prevents the threads from accessing the same array index simultaneously, but also briefly diminishes the benefits of parallelization. After each constraint has accumulated its delta displacement, the positions for each particle are updated in parallel (steps 8-10). Each positional change is divided by n_p, the number of constraints of that particular particle.

5.3.4 Projective dynamics

Projective Dynamics is a framework presented by [Bouaziz et al., 2014]. It uses energy potentials to define different constraints, then applies an alternating minimization technique to find the optimal solution. A detailed description of Projective Dynamics is presented in section *3.6 Projective Dynamics*. In this section, our implementation is described in general terms, followed by the specific implementations of each solver. Three different solvers were implemented using Projective Dynamics:

- Sequential CPU Gauss-Seidel
- Parallel GPU Jacobi
- Parallel GPU Red-Black Gauss-Seidel

To end this section, the implementation of successive over-relaxation (SOR) for the two Gauss-Seidel solvers is described.

General algorithm Algorithm 4 describes how the alternating optimization method is implemented. Beginning at step 1, the particle positions computed in the numerical integration step are stored in $\mathbf{y}$. Then, i outer iterations are executed. In [Liu et al., 2013], the normalized vector between two particles of a constraint, called $\mathbf{d}$, is calculated in the outer loop (local step). In step 3, we instead store the result calculated by the solver, $\mathbf{q}$, in **temp** and calculate $\mathbf{d}$ on-the-fly based on **temp** inside the solver. The first time the outer loop is executed, the value of $\mathbf{q}$ is equal to $\mathbf{y}$. In

step 4, the inner loop is executed j iterations. Inside each inner iteration (step 5), the solver processes all simulated particles, sequentially or in parallel depending on the specific solver. The three solver algorithms described below are all encapsulated by this general algorithm.

Algorithm 4: Projective Dynamics - General algorithm

```
1 y ⇐ verletIntegration
2 loop outerIterations times
3 |   temp ⇐ q
4 |   loop innerIterations times
5 |   |   solve linear equation system
6 |   end
7 end
```

Sequential Gauss-Seidel solver The sequential Gauss-Seidel solver was implemented on the CPU. The theory behind the Gauss-Seidel method is described in section *3.2.2 Gauss-Seidel iterative method*. Pseudo-code for the implementation is presented in algorithm 5.

Algorithm 5: Projective Dynamics - sequential Gauss-Seidel solver

```
1 forall particles p do
2 |   accum ⇐ y_p (mass/h^2)
3 |   forall connected particles cp do
4 |   |   d ⇐ temp_p − temp_cp
5 |   |   accum ⇐ accum + q_cp + d̂ L k
6 |   end
7 |   a ⇐ n k + mass/h^2
8 |   q_p ⇐ accum/a
9 end
```

An accumulated value **accum** is used to calculate the new position of each particle. In step 5, this value is initiated to $\mathbf{y}$, multiplied by the mass divided by the squared time-step. Considering the linear equation system solving equation $\mathbf{Ax} = \mathbf{b}$, this value corresponds to $\mathbf{b}$ for the particle p. Then, for all particles connected to p, the results of the Gauss-Seidel equations are accumulated. In step 11, when all connected particles have been processed, the value a, corresponding to the diagonal of the matrix $\mathbf{A}$ is computed. As in normal Gauss-Seidel solving (see section *3.2.2 Gauss-Seidel iterative method*), the equations are divided by this diagonal value to compute the new position of particle p, denoted as $\mathbf{q}_p$.

Parallel Jacobi solver The parallel Jacobi solver was implemented on the GPU. For an introduction to the Jacobi method, see section section *3.2.1 Jacobi iterative method*. Pseudo-code for the implementation is presented in algorithm 6.

Algorithm 6: Projective Dynamics - parallel Jacobi solver

```
1  for particles p do in parallel
2  |  accum = y_p (mass/h^2)
3  |  forall connected particles cp do
4  |  |  d ⇐ temp_p − temp_cp
5  |  |  accum ⇐ accum + q_cp + d̂ L k
6  |  end
7  |  a ⇐ n k + mass/h^2
8  |  Δ_p ⇐ accum/a − pos
9  |  Δ_p ⇐ Δ_p/n_p
10 end
11 for particles p do in parallel
12 |  q_p ⇐ Δ_p
13 end
```

The parallel Jacobi algorithm naturally has a lot in common with the sequential Gauss-Seidel method, since both are implemented using Projective Dynamics. However, there are some notable differences inside the inner loop. In steps 4-11 of the Jacobi solver, displacement deltas are computed. Since $\mathbf{q}_p$ is updated afterwards in a separate parallel step, the calculation of deltas is independent of any other calculations, and thus can also be executed in parallel.

Parallel Red-Black Gauss-Seidel solver The parallel Red-Black Gauss-Seidel solver was implemented on the GPU. The differences in respect to the previously mentioned methods are presented below. Pseudo-code for the implementation is presented in algorithm 7.
The Red-Black Gauss Seidel method divides the solving process into two steps, 1-8 and 9-16, in which the two colors are solved. The color partitioning ensures that no particles of the same color share a constraint, thereby allowing the entire system to be solved in only two parallel steps. Inside each parallel step, the solving is equivalent to the standard Gauss-Seidel method described above.

Successive over-relaxation In order to increase the convergence speed of the two Gauss-Seidel-based solvers, successive over-relaxation (SOR) is applied. The theory behind this method can be found in section *3.2.2 Gauss-Seidel iterative method*.

$$\mathbf{q} \Leftarrow \mathbf{q} + \omega\,(\mathbf{accum}/a - \mathbf{q}) \tag{5.2}$$

In the final step where the particle position $\mathbf{q}$ is updated, SOR is trivially implemented. The relaxation factor ω is chosen by hand for each solver. In general, it was set to a value which increased the convergence speed without introducing instabilities.

Algorithm 7: Projective Dynamics - parallel Red-Black Gauss-Seidel solver

for *red particles p* **do in parallel**
  accum $\Leftarrow \mathbf{y}_p\,(mass/h^2)$
  forall *connected particles cp* **do**
    $\mathbf{d} \Leftarrow \mathbf{temp}_p - \mathbf{temp}_{cp}$
    $\mathbf{accum} \Leftarrow \mathbf{accum} + \mathbf{q}_{cp} + \hat{\mathbf{d}}\,L\,k$
  end
  $a \Leftarrow n\,k + mass/h^2$
  $\mathbf{q}_p \Leftarrow \mathbf{accum}/a$
end
for *black particles p* **do in parallel**
  accum $\Leftarrow y_p\,(mass/h^2)$
  forall *connected particles cp* **do**
    $\mathbf{d} \Leftarrow \mathbf{temp}_p - \mathbf{temp}_{cp}$
    $\mathbf{accum} \Leftarrow \mathbf{accum} + \mathbf{q}_{cp} + \hat{\mathbf{d}}\,L\,k$
  end
  $a \Leftarrow n\,k + mass/h^2$
  $\mathbf{q}_p \Leftarrow \mathbf{accum}/a$
end

5.3.5 Collision handling

The collision handling is essential to any physics engine. The collisions needs to be detected and responded to correctly, while still maintaining high performance. In our simulation, we have tens of thousands of particles that all should interact with each other in real-time. Because of this, optimization is essential.

One of the most common and intuitive ways of optimizing the collision handling, is to use spatial subdivision. It is extremely time consuming to collision check each individual particle with all other particles. This gives a complexity of O(n^2), which definitely is not acceptable when n reaches tens of thousands of particles. If spatial subdivision is used, each particle only needs to be checked with its closest neighbors.

A uniform grid was used, where the size of each cell is set to fit one particle diameter. This means that each particle will only collision check with particles that potentially can collide with it, heavily reducing computational time. More information about this method can be found in section *3.4 Collision handling*.

In the following sections we will explain how the collision detection and response between particles and geometrical shapes are calculated. All of these collision scenarios are presented in section *6.2 Test cases*.

Sphere-Sphere The sphere-sphere collision check is trivial. The distance between the center of the two spheres needs to be computed and compared with the sum of the radii. If the distance is less than the sum, the spheres have collided.

In order to generate a proper collision response, a displacement delta needs to be calculated. It is calculated using the normalized direction vector between the spheres centers, with the magnitude of the overlap. In our implementation the sphere-sphere collision algorithm handles both colliding spheres at the same time. This means that we need to displace both spheres simultaneously. Our implementation also considers the general case, with spheres of different size. Therefore, the displacement must be spread across both spheres depending on the size of the spheres. If the spheres are the same size, we simply assign half the displacement to each particle.

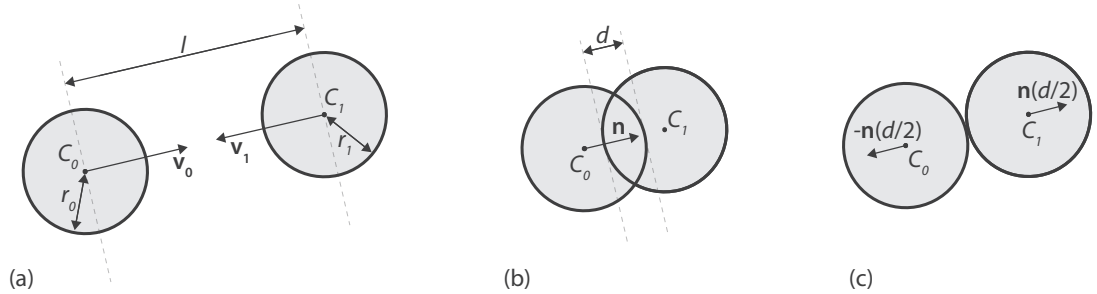

Figure 5.8: *Example of a sphere-sphere collision. C is the center point of a sphere, and r is the radius. This is considering the case of equal sized particles.*

In (a) the two spheres move toward each other. The distance between the spheres is defined by $l = |C_0 - C_1|$. The overlap of the spheres happens in the next time-step (b). The collision is detected when l is less than one rest length (L). The overlap $d = l - L$ is then computed. The response occurs in (c), where each particle gets displaced by $\mathbf{n}(d/2)$.

Sphere-Plane The intersection test between a sphere and an infinite plane is quite straight forward. When a collision is detected, the displacement direction is perpendicular to the plane. This is visualized in the example below:

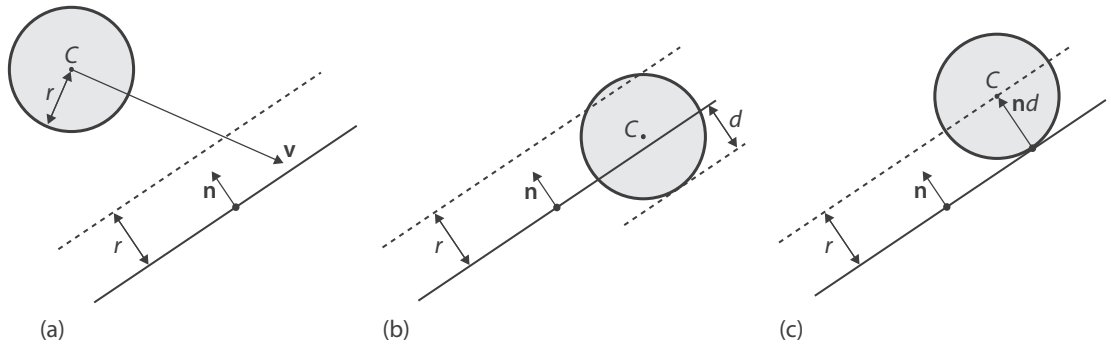

Figure 5.9: *Example of a sphere-plane collision. C is the center point of the sphere, r is the radii and* $\boldsymbol{n}$ *is the normal of the plane.*

The plane is defined by the general equation:

$$ax + by + cz + d = 0 \tag{5.3}$$

The distance l between the sphere center C is determined with the following equation:

$$l = \frac{\mathbf{n}_x C_x + \mathbf{n}_y C_y + \mathbf{n}_z C_z + d}{|\mathbf{n}|} \tag{5.4}$$

If this distance is less than the sphere radii, there is an intersection. The displacement is then simply defined as $\mathbf{n}d$.

Sphere-Box To collide a sphere with a box correctly is a non-trivial task. We define a box as an axis-aligned bounding box (AABB) between two points. After an intersection is found, the sphere is displaced to the nearest point of the plane of the box. This can be seen in the figure below:

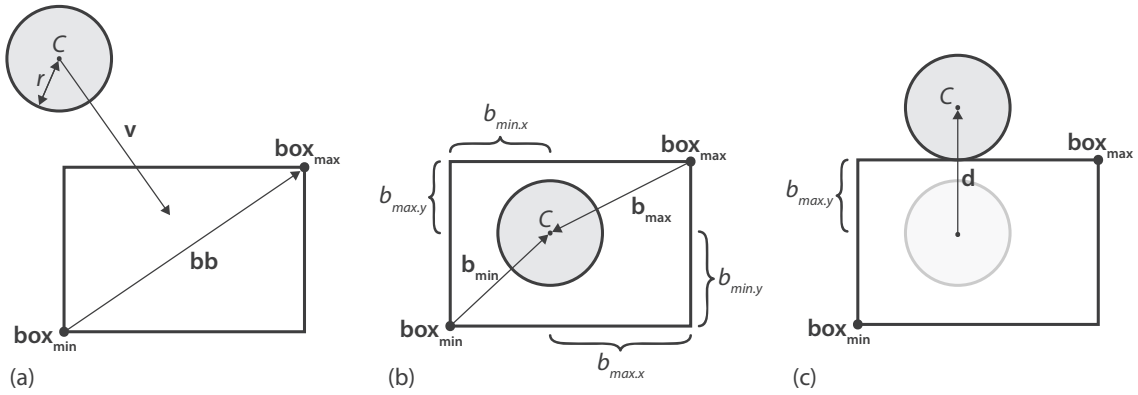

Figure 5.10: *Example of a sphere-box collision. C is the sphere center point, and r is the sphere radius.* $\boldsymbol{box}_{min}$ *and* $\boldsymbol{box}_{min}$ *are the two points that define the box.* $\boldsymbol{b}_{min}$ *and* $\boldsymbol{b}_{max}$ *are vectors going from* $\boldsymbol{box}_{min}$ *and* $\boldsymbol{box}_{min}$ *to the sphere center. The vector* $\boldsymbol{d}$ *is the computed displacement.*

In this implementation, we redefine the origin to be placed at the minimum point of the box. All particles whose position that are less than zero, and all particle positions which are greater than the maximum value of the bounding box are discarded. If this is not the case, the sphere is inside the box, and there has been a collision.

For the collision response, the nearest point on the surface of the box needs to be calculated. For this, we find the minimum value between $\mathbf{b}_{min.x}$, $\mathbf{b}_{min.y}$, $\mathbf{b}_{max.x}$ and $\mathbf{b}_{max.y}$. The lowest value indicates which side of the box the sphere should be displaced to. In the scenario of Figure 5.10, $\mathbf{b}_{max.y}$ is found to be the minimum value. For the displacement, the sphere keeps its x coordinate, but the y coordinate is set

to be $\mathbf{box}_{max.y} + r$. This example is in 2D but the method works equivalently in 3D.

Sphere-Funnel To create the funnel showcase presented in section *6.2 Test cases*, collision detection and response for a funnel was implemented. The intersection test is quite trivial, while the response proved to be a challenge.

The following parametric equations describe a funnel:

$$x = u \cdot cos(v)$$
$$y = a \cdot ln(u)$$
$$z = u \cdot sin(v)$$

In our implementation, the collision between a particle and the funnel is simplified by only using an intersection of the funnel at the current y-value. This intersection gives a circle, whose radius depends on the relation $r_f = e^{\frac{C_y}{a}}$.

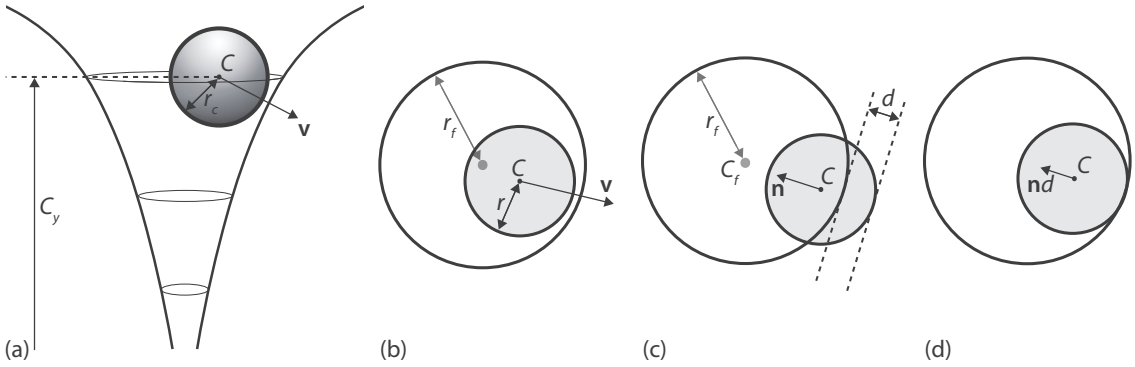

Figure 5.11: *Example of a sphere-funnel collision. In (a), the funnel is seen from the side. In (b)-(d) the funnel segment circle and the particle, is visualized from above.*

This means that the radius of the funnel decreases logarithmically in relation to y. The intersection test is trivial, as it is essentially an inverted circle intersection test. If the particle is within the height of the funnel, and partly or fully outside the cross section of the funnel, a collision has occurred. The difficult part is to calculate the correct displacement of the particle. In our case we simply calculate the 2 dimensional displacement, and add a high friction, simply to be able to demonstrate the funnel. If this were to be implemented properly, the displacement would have to be calculated in three dimensions. For our purposes however, this implementation is sufficient.

5.3.6 Friction

The friction is handled as a subsequent step after the collision response. Instead of damping velocities or calculating external forces which are applied in the integration step, we chose to manipulate the positions directly after the collision response. This

is essentially the same thing, since we are using non-velocity Verlet integration. This is essentially a simplified version of [Bridson et al., 2002].

From the collision handling two values are required, namely the displacement δ and the distance between the two colliding objects. We want to calculate a delta which states how much of the displacement we want to remove due to friction.

First we need to calculate the velocity v_i for particle i. This is done by subtracting the position returned by the integration, with the current position. This basically gives the average velocity.

$$\mathbf{v}_i = \mathbf{q} - \mathbf{q}_{prev}$$

We also need to calculate the positional delta in the tangential direction:

$$\Delta_{tan} = \frac{\mathbf{v} - \mathbf{n}(\mathbf{vn})}{2}$$

This delta needs to depend on the friction coefficient μ and how much the objects collide. If the collision displacement between two objects is large, the impact of the friction needs to be large as well.

$$coef = \frac{\mu \cdot d}{|\Delta_{tan}|}$$

Now, the delta needs to be updated with the coefficient we just calculated. No value above 1 is accepted.

$$\Delta_{tan} = \Delta_{tan} \cdot min(coef, 1)$$

Finally, the total displacement can be calculated, and then applied to the particle.

$$\mathbf{displacement} = \mathbf{displacement} - \Delta_{tan}$$

5.3.7 Tearing

For realistic animation of deformable bodies such as ropes and cloth, behaviors like tearing needs to be reproduced when the body interacts with other objects or is affected by external forces, such as the input from a user [Rungjiratananon et al., 2010].

This was implemented by adding a tearing parameter which stated how far apart the particles can be before the constraint is removed. Once the distance threshold is exceeded, the indexes in the constraint is set to a negative value and thus are not considered any longer. Similarly, the triangles between the particles previously sharing constraints are no longer generated.

5.3.8 User interaction

A goal of the project was to be able to simulate soft objects in real-time. Real-time applications allow the user to view or manipulate various objects directly. To showcase the potential of the engine, some sort of user interaction was required, such as picking objects and dragging them around.

The mouse picking was implemented using ray casting. Whenever the user clicks the mouse, a ray is shot through the three dimensional environment, and each particle checks if it collides with the ray, in parallel. To detect collisions between a ray and a sphere, the algorithm provided by [Haines and Akenine-Möller, 2002] was used.

A sphere can be represented by a point and a radius. A formula for the sphere can then be expressed in the following way:

$$f(\mathbf{p}) = |\mathbf{p} - \mathbf{c}| - r = 0$$

Similarly, a ray can be defined as having an origin and a direction:

$$r(t) = \mathbf{o} + t\mathbf{d}$$

To solve the intersection between a ray and a sphere, replace $\mathbf{p}$ in the equation for a sphere with the equation for a ray:

$$f(r(p)) = |\mathbf{o} + t\mathbf{d} - \mathbf{c}| = r$$

These equations do not consider which particle is closest to the camera. This produces the behavior that all the particles which intersects will be picked. This poses a problem when several objects lie on top of each other, and the user wants to pick the top one. Therefore, a check was implemented which enables the picking of only the closest particle.

The detection is more or less trivial in implementing. However, the picked particles need to move according to the mouse cursor. In order to do this, the screen space coordinates need to be transformed into world space in a correct manner. This requires projecting from the closest particle to the screen in order to find its depth value. The mouse motion in X and Y is then projected back at this depth value in order to keep the picked particle tied to the mouse pointer, independent of the world-space position of the particle.

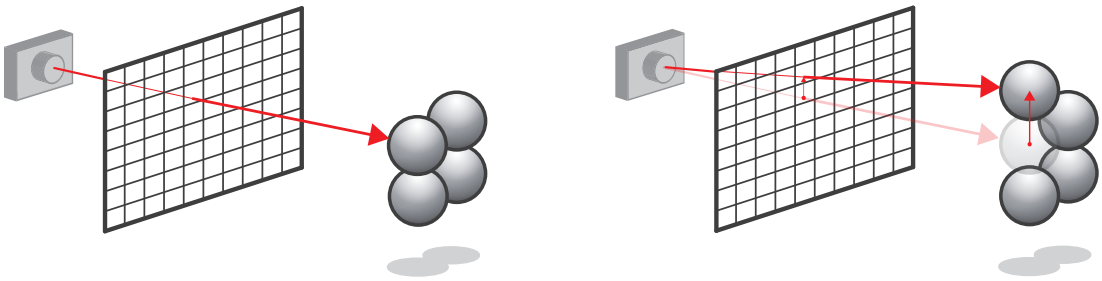

Figure 5.12: *Example of mouse picking with rays. The grid is the screen space and the particles are in world space. The displacement in screen space is converted into motion in world space.*

5.4 Rendering

The main objective of this project was to create a physics engine, not a render engine. However, for the purpose of developing and demonstrating the physics engine, the simulated world has to be conveyed in a comprehensive manner.

In our case, the physics simulation is modeled with particles, therefore the simplest way of visualizing the underlying dynamics is by rendering each particle position as a shaded sphere. This is essentially the only entity that needs to be rendered. However, this is not sufficient to provide a sense of depth and spacial perspective.

The rendering was implemented mainly by using OpenGL, with some calculations done in CUDA, such as generating primitives and normals for the cloth rendering. The lighting, material and shadows were implemented using GLSL, which is the shader language that OpenGL uses. In this model a shader program usually contains two shaders, a vertex shader which makes calculations upon the vertices and a fragment shader which specifies the color of each pixel [John Kessenich, 2016].

5.4.1 Mesh generation

The rendering of a mesh is essential to conveying the underlying dynamics and presenting a realistic looking result. In order to create the mesh, triangles and vertex normals given the current particle positions need to be generated. These are then sent to the rendering pipeline for vertex and fragment shading.

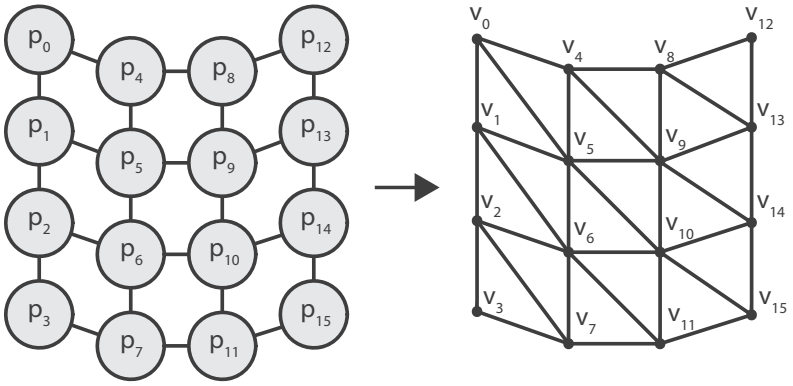

Figure 5.13: *The triangles (right) are generated by the underlying particle system (left).*

The triangles are updated based on the current positions of the particles. For a 2-dimensional cloth c, the maximum number of triangles is equal to $2(c_x - 1)(c_y - 1)$ where c_x and c_y are the dimensions of the cloth. In order to determine if a triangle should be generated, we examine the constraints of each particle in parallel. For example, if a particle p_0 has a constraint in the positive y axis, and the other particle of that constraint, p_1, itself has a constraint in the positive x axis with particle p_2, a triangle is generated. This triangle has its vertices equal to the positions of the three constrained particles, and the vertex normals are computed as the average of the surface normals of the surrounding triangles. By making the triangles depend on the constraints between particles, we can trivially reflect updates in the structure such as tearing.

To visualize the underlying structure of triangles on the smoothly shaded cloth, a wireframe can be rendered on top of it by calling OpenGL's glPolygonMode function with the GL_LINE rasterization mode. In order to prevent z-fighting when rendering the wireframe, we use a shader which applies a minor positional displacement in the direction of the camera.

5.4.2 Shading

Multiple light sources are essential when aiming to produce a realistic image. They also give the viewer a better spatial comprehension [Sunden and Ropinski, 2015]. Since we do not have any global illumination and are only using point lights, all of our scenes contain two light sources.

In many cases, the appearance of rendered images seeks to achieve photo-realism [Haines and Akenine-Möller, 2002]. Generally, the speed of the rendering and the realism of the result are in conflict with each other. To obtain physically correct shading, techniques based on ray-tracing are employed. These methods correctly model how light travels and bounces on surfaces, but are comparatively slow to compute. In this thesis, the speed and simplicity of the rendering is more important than photo-realism, and therefore a simplified model, presented in [Haines and

Akenine-Möller, 2002], is used.

In this model, we calculate the outgoing radiance i_{tot} of the material, in the view direction. The model contains ambient, diffuse and specular light, and sometimes also emissive. Each component is calculated separately in the fragment shader, and is finally accumulated to produce the color of an individual pixel.

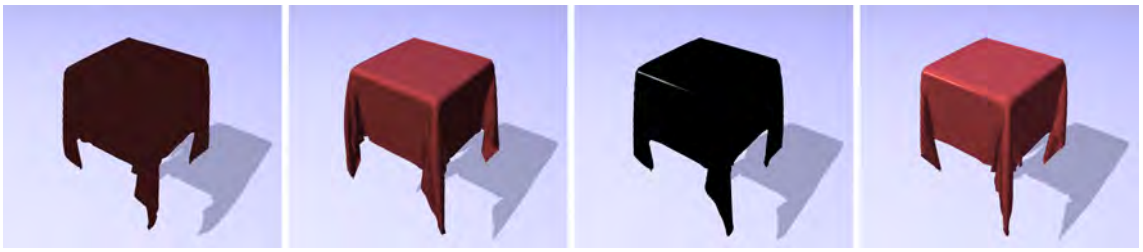

Figure 5.14: *From left to right: cloth with only ambient shading, diffuse shading, specular shading, all three components added together*

The ambient term accounts for the global illumination, and is simply a constant color that depends on the material color and the light colors. The $\otimes$ symbol used in the equations below signifies component-wise multiplication.

$$\mathbf{i}_{amb} = \mathbf{m}_{color} \otimes \mathbf{s}_{color}$$

The diffuse component depends on the material color, light colors and the light positions. It is built upon Lambert's Law, which describes that a totally matte surface will distribute light according to the cosine between the surface normal and the light vector. Another way of expressing this is with the following equation:

$$\mathbf{i}_{diff} = (\mathbf{n} \cdot \mathbf{l})\mathbf{m}_{color} \otimes \mathbf{s}_{color}$$

In order to account for highlights, the specular term needs to be calculated. There are many different models which are used to do this. One of the most famous and common ones is Phong's model. Here m_{shi} is the shininess parameter.

$$\mathbf{i}_{spec} = max(0, (\mathbf{r} \cdot \mathbf{v}))^{m_{shi}}\mathbf{m}_{color} \otimes \mathbf{s}_{color}$$

All of the components added together provide the correct outgoing radiance L_0:

$$\mathbf{i}_{tot} = \mathbf{i}_{amb} + \mathbf{i}_{diff} + \mathbf{i}_{spec}$$

5.4.3 Shadows

Without shadows in a scene, there is no sense of depth. It is very difficult for the human eye to locate various objects in relation to each other when looking at a 2D projection of the 3D world [Sunden and Ropinski, 2015]. We choose to implement a very basic shadow mapping algorithm due to this reason.

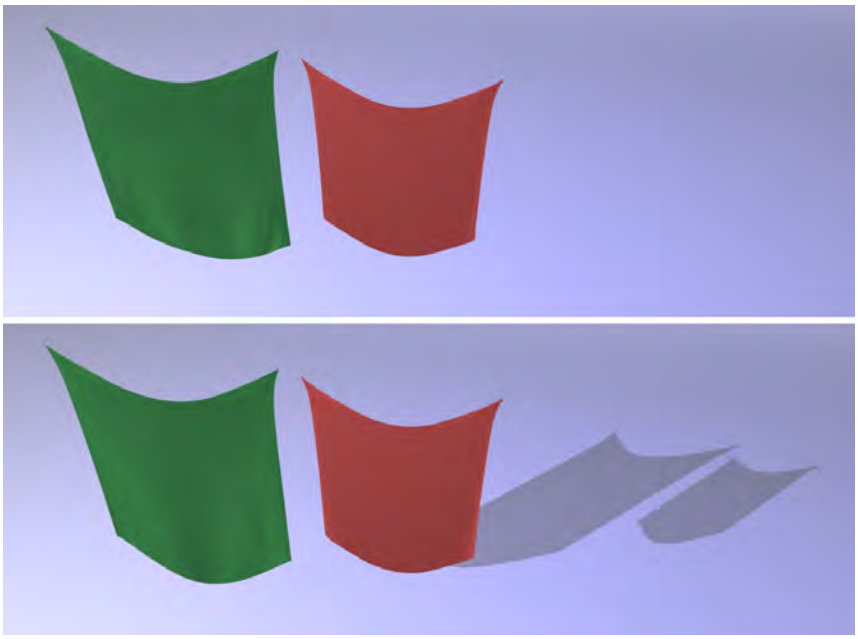

Figure 5.15: *Rendering without and with shadows. A small cloth is hanging close to the camera and a bigger cloth hanging further away.*

Shadow mapping is one of the fastest ways to produce shadows in computer graphics. However, it is quite difficult to implement good looking shadow mapping. The method is not entirely correct in terms of physical accuracy either, but produces a reasonably convincing result. For our purpose, it was the most sensible choice due to performance considerations.

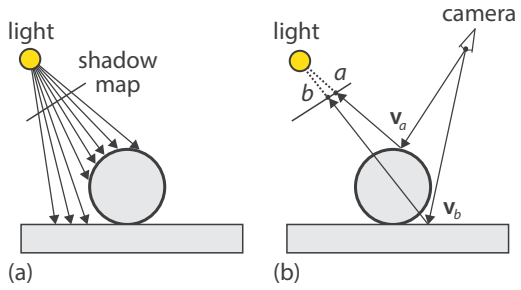

Figure 5.16: *In (a), the depth values are rendered to a texture called the shadow map, from the light's perspective. In (b), each fragment from the camera's perspective is compared to the corresponding value in the shadow map. If the depth value of the shadow map is smaller, the fragment is occluded. For example, point* $\boldsymbol{v}_b$ *is further away from the light than the corresponding depth value at texel b. Image courtesy of [Haines and Akenine-Möller, 2002]*

When using shadow mapping the scene needs to be rendered in two passes. In the first pass, the scene is rendered from the lights perspective, onto a 2D texture, called the depth map. The texture only contains the z-depth values.

When the scene is rendered a second time, from the cameras perspective, each fragments z-value is compared to the shadow map. If the fragment is farther away from the source of light than the corresponding value in the shadow map, it is considered to be occluded [Haines and Akenine-Möller, 2002].

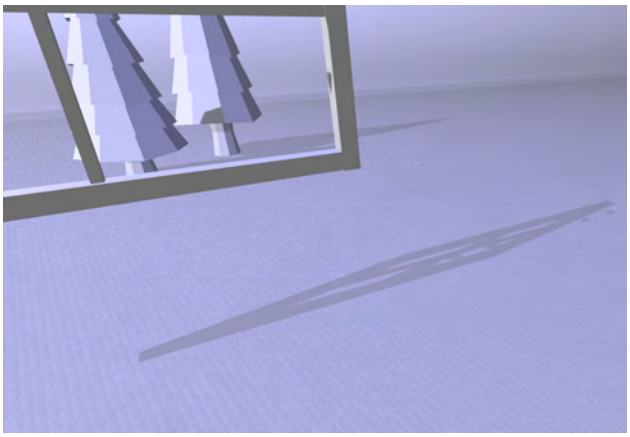

Figure 5.17: *Example of surface acne due to self shadowing.*

There are various problems connected to shadow mapping, such as surface acne and light leaking. Because of this, we choose to only cast shadows on the floor and models of the scene, not the actual cloth. Self-shadowing cloth with shadow mapping simply poses to many problems. The shading of the cloth was considered enough to give the viewer a sense of depth.

The problem of surface acne can be seen in figure 5.17. This problem occurs due to the surface shadowing itself. This problem can be removed by introducing a bias, which shifts the floor to be considered above the shadow depth [Haines and Akenine-Möller, 2002].

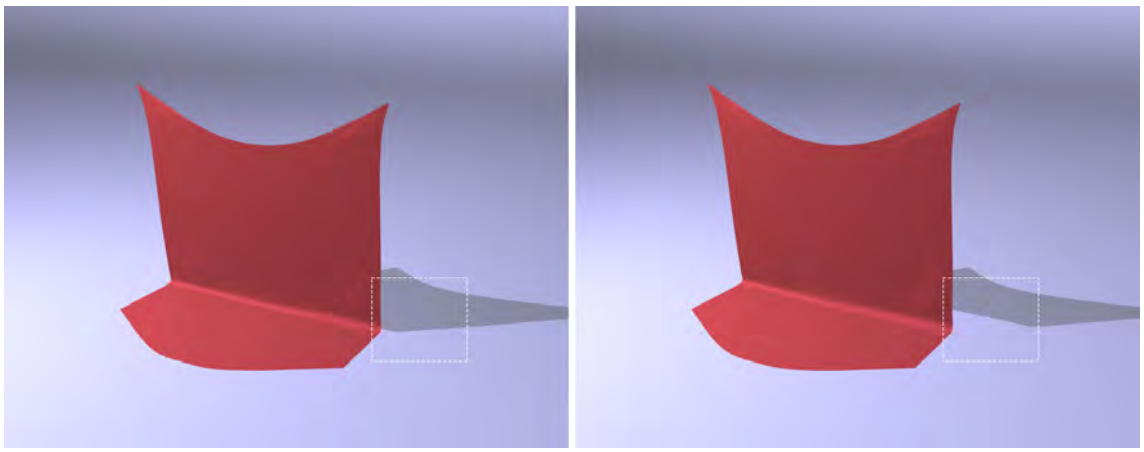

Figure 5.18: *An example demonstrating the problem of light leakage.*

However, when introducing a bias, another problem called light leakage might occur.

This problem produces an illusion that the object is floating above the surface, when it is not [Haines and Akenine-Möller, 2002]. Therefore, a bias needs to be chosen carefully in order to not produce either of these two problems.

5.5 Animation of kinematic bodies

Physically-based simulations can be applied in many different areas. Since the engine, in its current state, mainly can produce cloth simulations, a clear example of such an area is garment simulation. To be able to convincingly showcase this scenario, a number of different aspects need to be considered. In the following sections, the loading of the animation models, the collisions with the models and how the garments are attached, will be explained.

5.5.1 Model sequence

The model animations were implemented with a quick-and-dirty approach. The purpose of the animations was only to showcase the potential of the physics engine, not to implement an efficient animation system.

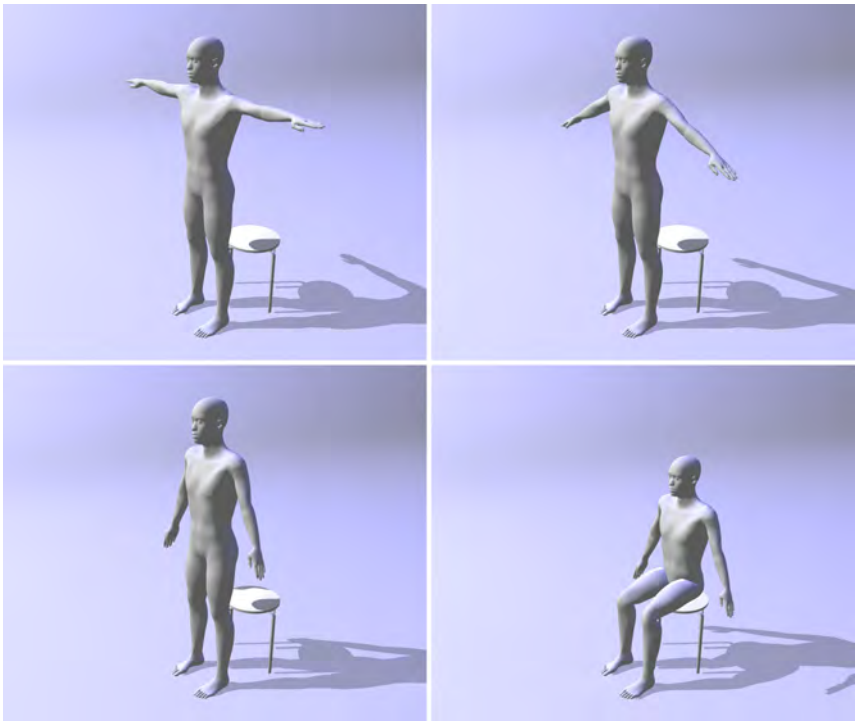

Figure 5.19: *A model sequence, showing four still frames from the generated animation.*

Instead of using more sophisticated techniques such as skinning, a number of 3D models are simply loaded into memory, using the ASSIMP library[2]. Then the rendered model is changed each frame to create an animation. This is generally not

[2]http://assimp.sourceforge.net

an advised approach, due to the amount of memory used, and the static nature of having predefined models, but for showcasing it was considered acceptable.

5.5.2 Collision handling

The main reason behind having an animated model is for it to interact in a convincing manner with the soft objects of the physics engine. This can be achieved in a number of ways.

Signed distance fields (SDFs) could be used to produce very exact collisions with the models. This was not viable, because of the very large amounts of memory it would require. Each frame would need a precomputed SDF, which would need to be stored in memory. In our case, an animation sometimes contains over 600 frames. Given that an SDF at a reasonably high resolution consumes several megabytes of memory, several hundred SDFs would simply take up to much memory to make SDFs a feasible solution. Another approach which could be used is to try to fit mathematically defined bounding volumes to each frame. This however would be too difficult to implement.

In our case, we chose to implement the collisions by distributing particles over the surface of the model. This is done by creating a subdivided model, and placing a particle at each vertex position, in an iterative fashion. If the current particle collides with any other previously placed particle, it is not placed. These particle distributions are then stored for each frame. In contrast to the SDF solution, the total amount of memory needed to store collision primitives for a whole animation is quite small. For example, for the sitting man showcase in section *6.2 Test cases*, the amount of memory needed for storing animation collision particles is approximately 48 MB. The same animation would take up almost 10 GB if SDFs were to be used instead.

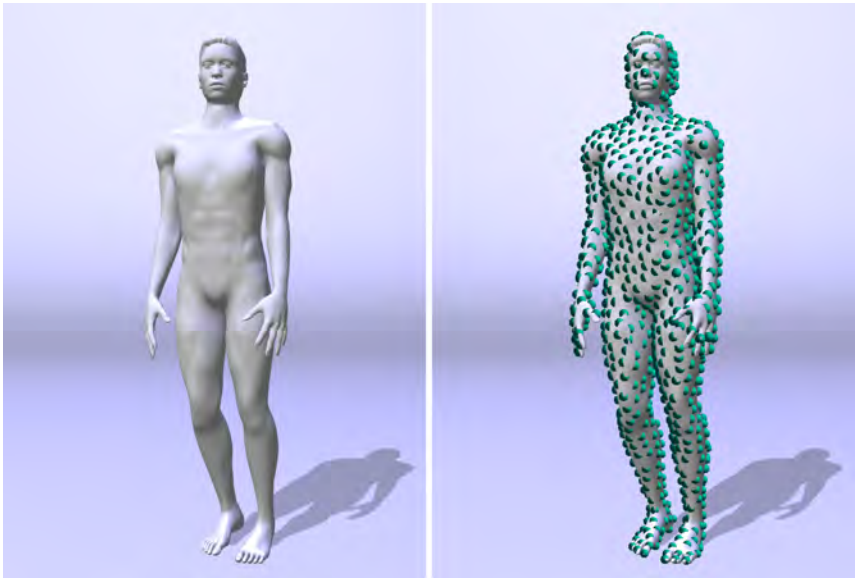

Figure 5.20: *To the left: The model as rendered to the user. To the right: The underlying particle distribution, to account for collisions with the model.*

This will fill the surface of the model with a reasonably even distribution of particles. These particles are not rendered, in order to produce the illusion that the soft objects actually interact with the model.

5.5.3 Attachment points

A method to attach soft bodies to models was implemented. This feature can be seen in the rotating ropes and curtain showcases in section *6.2 Test cases*.

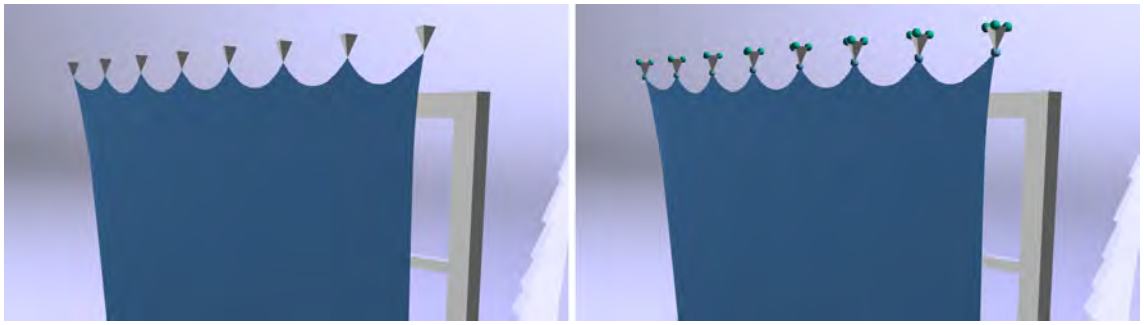

Figure 5.21: *To the left: The rendered image, as visible to the user. To the right: The attachment points. The blue particle is part of the cloth, while the cyan particles are part of the model. When attached, the attachment model particles and the attached cloth particles share the same position.*

This was implemented by giving the option to attach an arbitrary amount of particles in the soft object, by specifying their index, to a preferred location on the model. The closest model vertex to the preferred position is then found, and the index is stored. The attached particles positions are updated to the position of the vertex at each frame.

6
Results

This chapter presents the results of the physics engine, in terms of both performance and visual appearance. First, the experiment setup is described. Then the results of the two experiments are presented and discussed from a data-centric perspective. To end the chapter, the visual results achieved by the physics engine are presented and analyzed. These are showcases which aim to demonstrate what can be achieved with the engine.

6.1 Performance

The main reason for conducting the two experiments was to produce enough data to confidently draw conclusions about the research question. This section includes a presentation of the setup of the experiments, along with the produced results.

Convergence As stated in the research question, this thesis sought to answer if the Red-Black Gauss-Seidel method would be viable in real-time simulations. To draw any conclusions about the efficiency of the solver, the convergence speed would have to be compared to other commonly used methods. In order to compare convergence speed, the error of each solver had to be computed. This computation is described in section *4.3 Result evaluation*.

The first experiment was set up using an enlarged cloth, with the rendering, numerical integration and collision handling disabled. The Gauss-Seidel and Jacobi solvers were implemented based on the lexicographical definitions. The implementations are described in chapter *5 Process*.

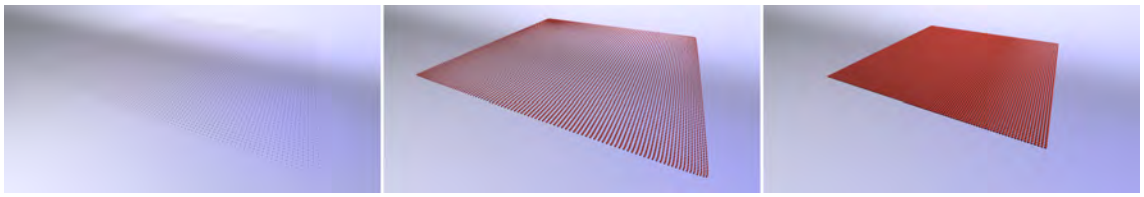

Figure 6.1: *The experiment setup for the enlarged cloth. On the left: the initial configuration of the cloth, being 5 times enlarged. In the middle: the cloth after a few iterations. To the right: the cloth at rest position.*

The second experiment uses a cloth hanging from two points. The cloth is dropped

from a vertical position, and the convergence is measured. In this experiment, the numerical integration is enabled, but collision handling and rendering remain disabled.

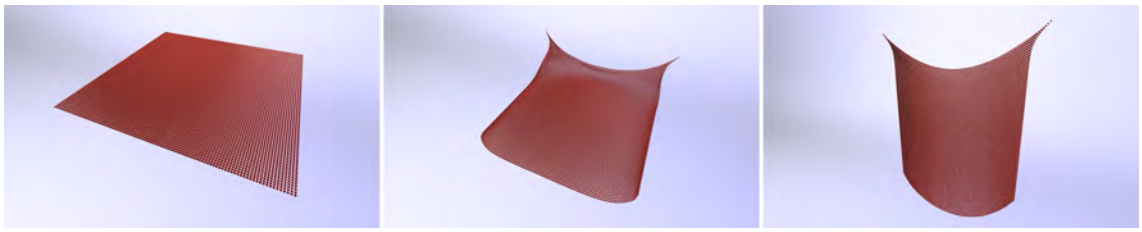

Figure 6.2: *The experiment setup for the hanging cloth. On the left: the start configuration of the cloth, being in a horizontal position. In the middle: the cloth after a few iterations. To the right: the cloth at rest position.*

The hanging cloth experiment can be seen in the supplemental video. In the video, the real frame rate can be viewed as well as the corresponding graphs.

Experiment 1 - Enlarged cloth The setup used for the enlarged experiments can be seen in figure 6.1. In this experiment, the numerical integration was disabled, and the distance between each particle was set to be 5 times the rest length. With this setup, only the solver is manipulating the positions of the cloth. A cloth with 10 000 particles and 19 800 constraints was used for all experiments. For each frame 8 physics steps were used.

The following graph shows the convergence speed per iteration of the different solvers. The x-axis limit is set to 17 000 iterations so that RBGS and GS is still visible.

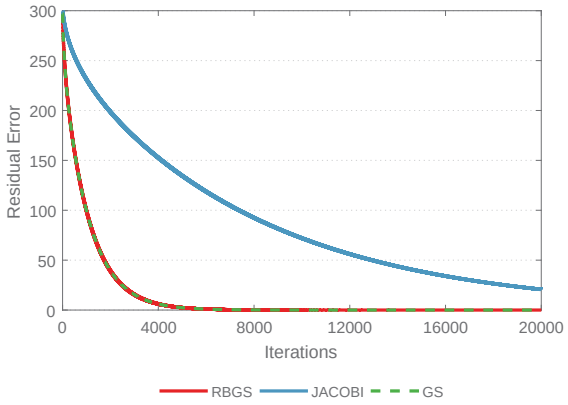

Figure 6.3: *Residual error vs. iterations for the enlarged cloth, using 10 000 particles and 19 800 constraints. The convergence speed of our method (red) is compared with Gauss-Seidel (green) and Jacobi (blue). Note that Gauss-Seidel is a dashed line due to overlapping curves.*

In figure 6.5 the convergence speed of GS and RBGS is identical. This is to be expected, since it is essentially the same solver, only that RBGS is implemented in

parallel. The Jacobi solver shows a slow convergence rate per iteration in comparison to the Gauss-Seidel methods.

In the following graphs two experiments are presented. These two experiments use a different time budget. Next to each curve, the number of iterations used to fulfill the time budget is presented for each solver.

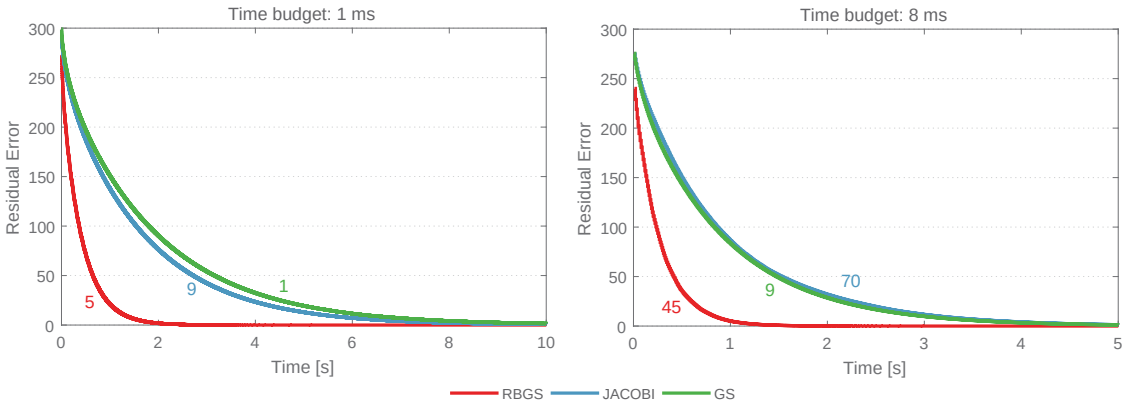

Figure 6.4: *Residual error vs. time for the enlarged cloth, using 10 000 particles and 19 800 constraints. The convergence speed of our method (red) is compared with Gauss-Seidel (green) and Jacobi (blue). On the left hand side, a time budget of 1 ms for the solver is used. On the right hand side a time budget of 8 ms is used. The corresponding number of iterations can be seen next to the curves.*

In figure 6.6, it is clear that the Jacobi solver has better convergence than the GS solver for small time budgets, but that GS performs slightly better at larger time budgets. However, it is also visible that RBGS outperforms both other solvers massively with respect to time.

The following two graphs contain an extra curve which is the RBGS method with SOR, using $\omega = 1.7$. The graph on the left shows the convergence over iterations, while the right shows the convergence over time, with a time budget of 2 ms.

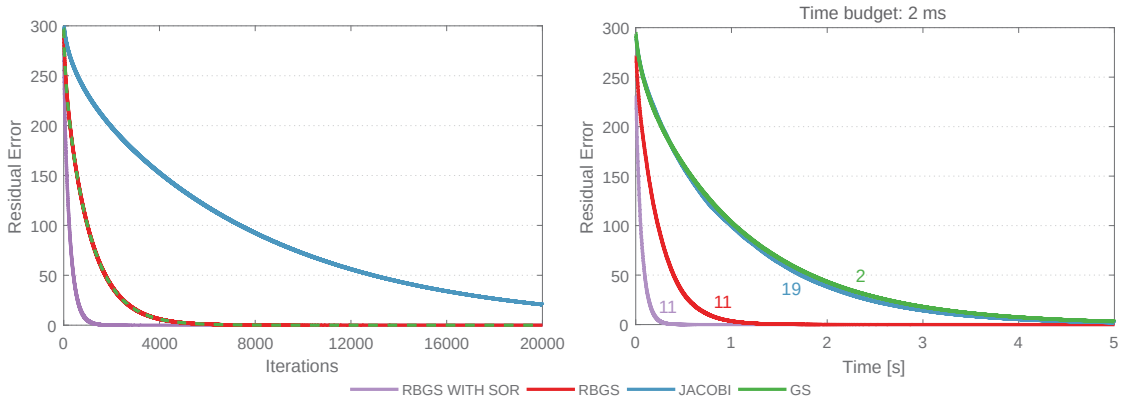

Figure 6.5: *Comparison of our method with and without successive over-relaxation for the enlarged cloth. Left hand side: Residual error vs iterations. Right hand side: Residual error vs time. Using a time budget of 8 ms for the solver. The number of iterations for each solver can be seen next to the corresponding curve.*

From figure 6.7 we can draw the conclusion that SOR heavily increases the convergence rate of the RBGS solver. Note that the experiment for this scenario was a simplified one, which only considers the solver, without any external input.

Experiment 2 - Hanging cloth The setup used for the hanging cloth experiments can be seen in figure 6.2. The cloth is attached in two corner particles and dropped from a horizontal position. This experiment tests the elasticity and convergence speed of the solver, under the stress of external forces. A cloth with 10 000 particles and 19 800 constraints was used for all experiments. For each frame 8 physics steps were used.

The first experiment presented shows the convergence over iteration. The first 60 000 iterations are visible.

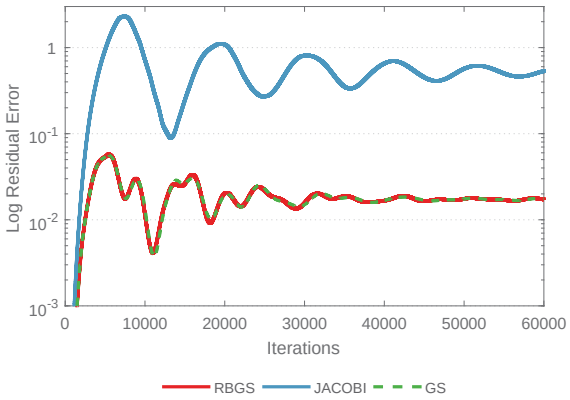

Figure 6.6: *Residual error vs. iterations for the hanging cloth, using 10 000 particles and 19 800 constraints. The convergence speed of our method (red) is compared with Gauss-Seidel (green) and Jacobi (blue). Note that Gauss-Seidel is a dashed line due to overlapping curves.*

In figure 6.8 we can see that the convergence rate of both GS and RBGS is identical.

As with the previous experiment, with an enlarged cloth, this is to be expected. The convergence rate of the Jacobi solver is worse than the Gauss-Seidel solvers, even after 60 000 iterations.

In the following two graphs, the error over time is presented with different time budgets. For the left graph, a time budget of 2 ms was used, and 8 ms for the right.

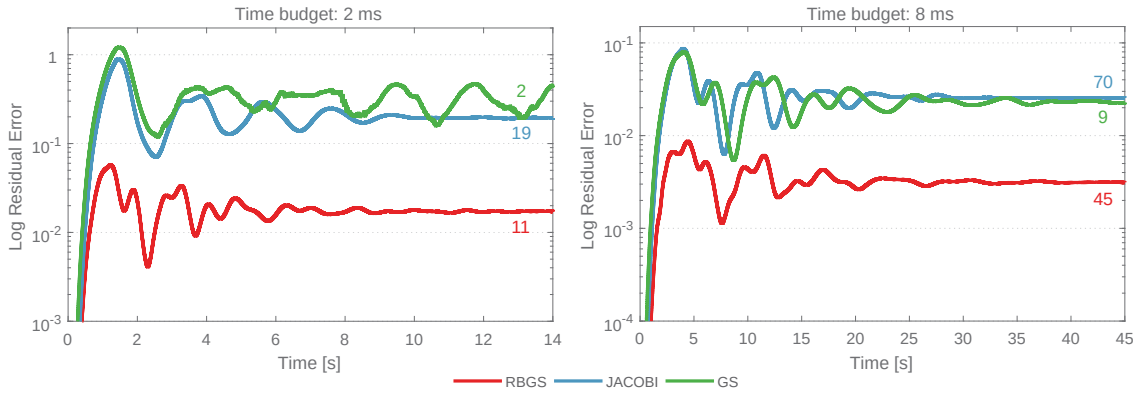

Figure 6.7: *Residual error vs. time for the hanging cloth, using 10 000 particles and 19 800 constraints. The convergence speed of our method (red) is compared with Gauss-Seidel (green) and Jacobi (blue). On the left hand side, a time budget of 2 ms for the solver is used. On the right hand side a time budget of 8 ms is used. The corresponding number of iterations can be seen next to the curves.*

The results in figure 6.9 indicate that the larger time budget, the better GS performs in relation to Jacobi. At small time budgets the GS solver is unstable. This is due to the GS solver requiring very few iterations in order not to exceed the time budget. The instability can be seen in figure 6.10, where the green cloth exhibits strange behavior. At a time budget of 1 ms the problem gets even more visible, which can be seen in appendix A, figure A:10, where all the graphs are included. The RBGS solver is outperforming both Jacobi and GS, regardless of time budget.

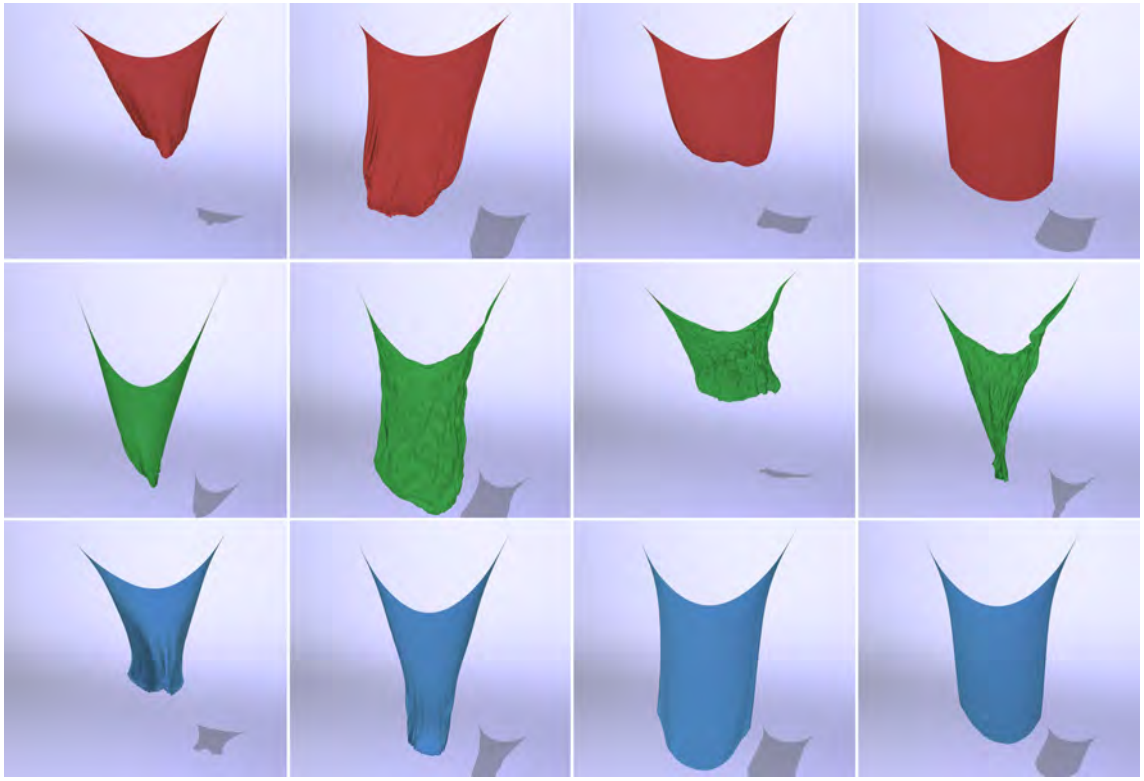

Figure 6.8: *Still frames corresponding to the left graph in figure 6.9, where a time budget of 2 ms is used. From top to bottom: Red-Black Gauss-Seidel, Gauss-Seidel, Jacobi. This sequence can also be seen in the accompanying video*[1].

In the following figure, the results of the hanging cloth experiment, including RBGS with SOR are presented. The successive over-relaxation is using an ω of 1.7. Note that the error over iterations is presented on the left and the error over time, using a time budget of 2 ms, is presented on the right.

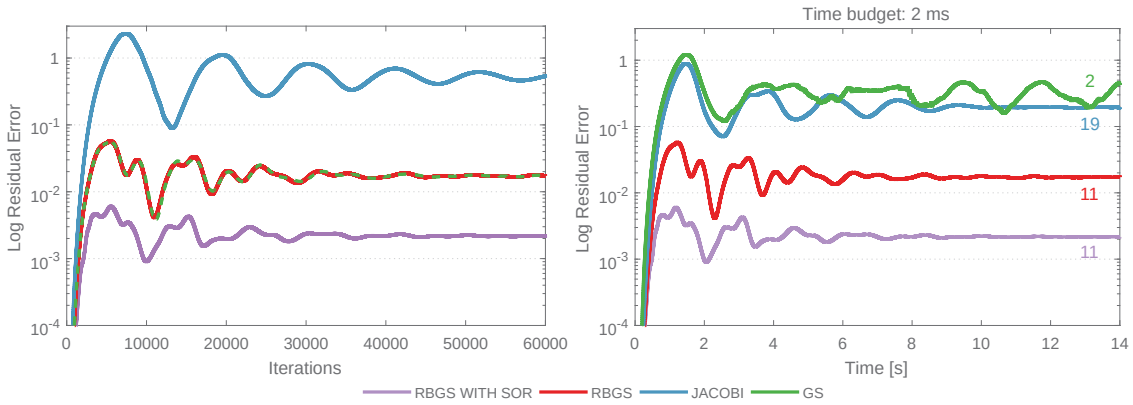

Figure 6.9: *Comparison of our method with and without successive over-relaxation for the hanging cloth. Left hand side: Residual error vs iterations. Right hand side: Residual error vs time. Using a time budget of 8 ms for the solver. The number of iterations for each solver can be seen next to the corresponding curve.*

[1]https://youtu.be/RPDkt6THJ4k

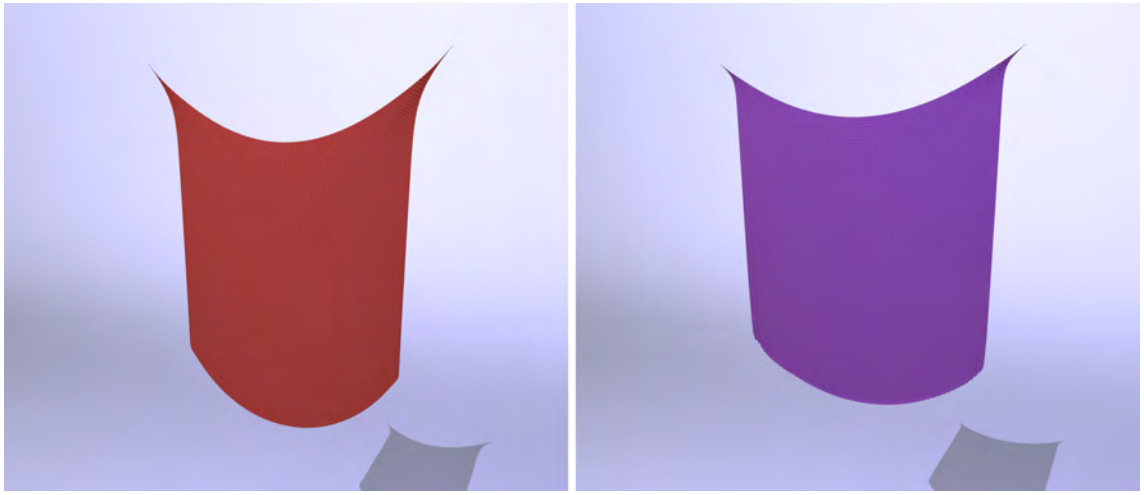

Figure 6.10: *Comparison of our method with and without successive over-relaxation for the hanging cloth after it has stabilized. This corresponds to the graphs in figure 6.11. RBGS to the left, and RBGS with SOR using $\omega = 1.7$, to the right.*

It is clear that RBGS with SOR outperforms all the other solvers, from the data presented in figure 6.10. This performance gain can be seen both when error over iterations and error over time is considered. The resulting cloth, after more than 50 000 iterations can be seen in figure 6.12.

6.2 Test cases

All the showcases can be viewed in the accompanying video[1]. The purpose of the showcases is to demonstrate the potential of the physics engine. In most of these scenarios, a time-step of 2 ms was used, compensated by 8 physical steps per frame. All the showcases are in real-time, run at approximately 60 frames per second, unless explicitly noted otherwise. In some cases, the scenario used is inspired by videos accompanying other papers. In these cases, the other videos are referenced in a footnote. Our showcase is then compared visually using the reference video.

Collisions with geometrical shapes There are several scenes showcasing collisions between cloth and a geometrical shape. All the shapes are defined mathematically, and the collision handling is explained in section *5.3.5 Collision handling*. The purpose of these showcases is to demonstrate that the cloth can interact with its environment.

The scene where the cloths collide with a box demonstrates that the cloths can collide with objects with sharp corners. However, for rendering purposes the box had a scaling bias, otherwise z-fighting between the edges and the cloth would occur.

[1] https://youtu.be/RPDkt6THJ4k

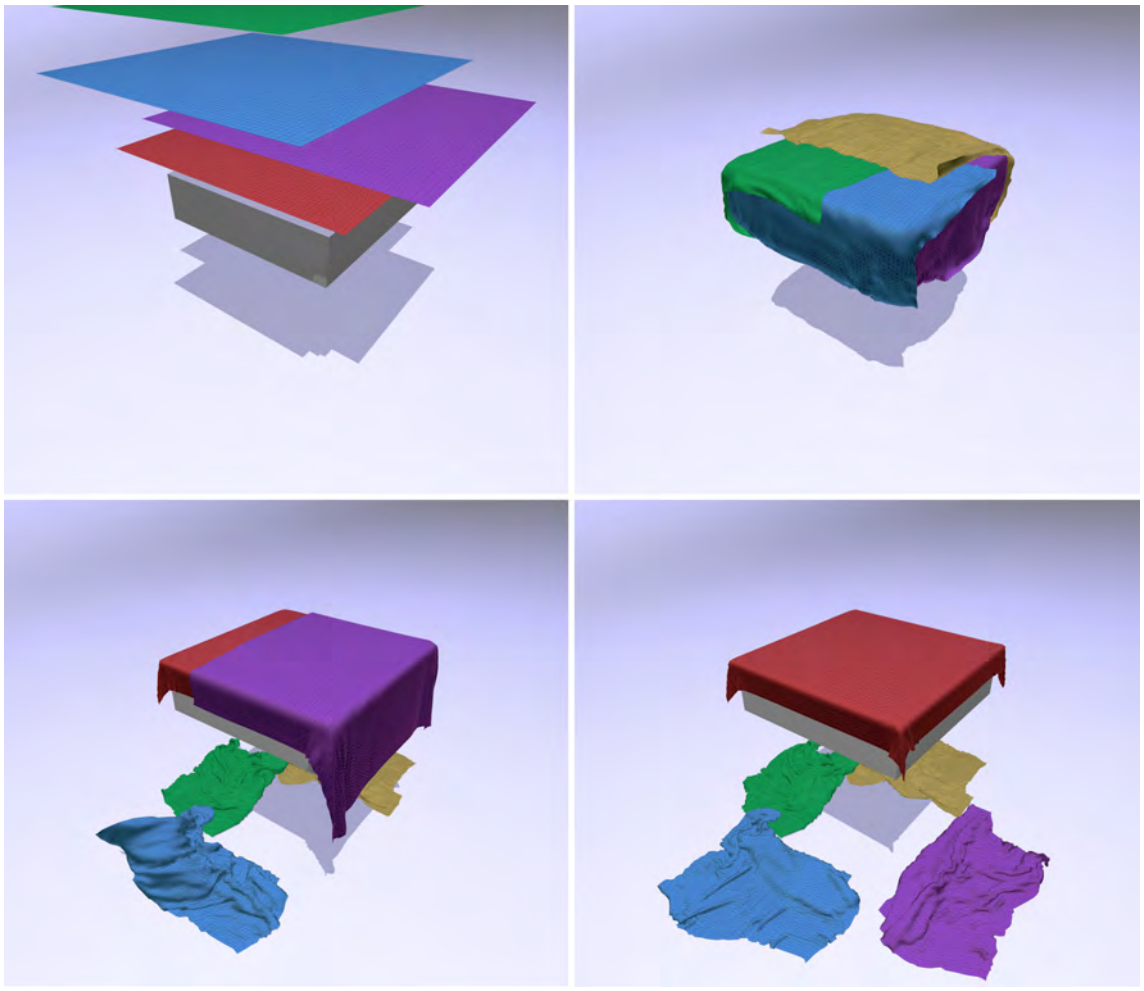

Figure 6.11: *Five cloths of different colors are falling towards the ground. The cloths collide with a box, and slide off to the ground.*

There are some problems with friction between the clothes, which makes the top clothes slide off easily. However, the showcase successfully demonstrates that no inter-cloth penetration or tunneling occurs.

Snapshots from the scene with the four hanging cloths which collide with the moving sphere can be seen in the figure below. It was inspired by the video[2] accompanying [Tang et al., 2016].

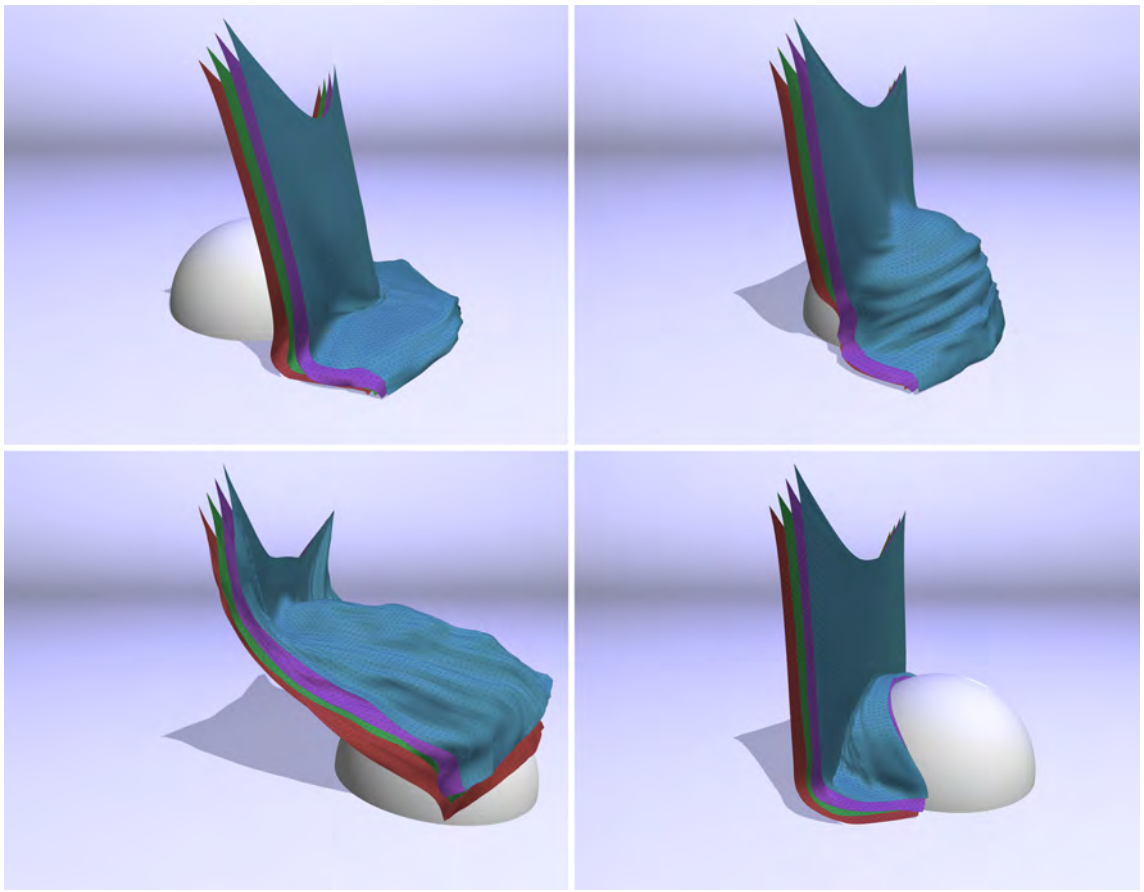

Figure 6.12: *Four cloths hanging from two points each, colliding with a sphere moving at high velocity.*

The main differences between the results of [Tang et al., 2016] and our results, is that our cloth is more elastic and produces more wrinkles. However, in our case, the results are computed in real-time, while they are not in [Tang et al., 2016].

[2]https://youtu.be/ZVyCfrAu7hY?t=1m52s

In the following example, four vertical cloths are thrown against a vertical plane. This showcase demonstrates the collision with a plane, self collision and that a starting velocity can be given to the soft bodies. This is achieved by setting different values for the previous position and the starting position, implicitly creating velocity. The inspiration for this showcase can be seen in the Advanced Molecular & Particle Physics Simulations video[3], which uses the Molecular addon for Blender.

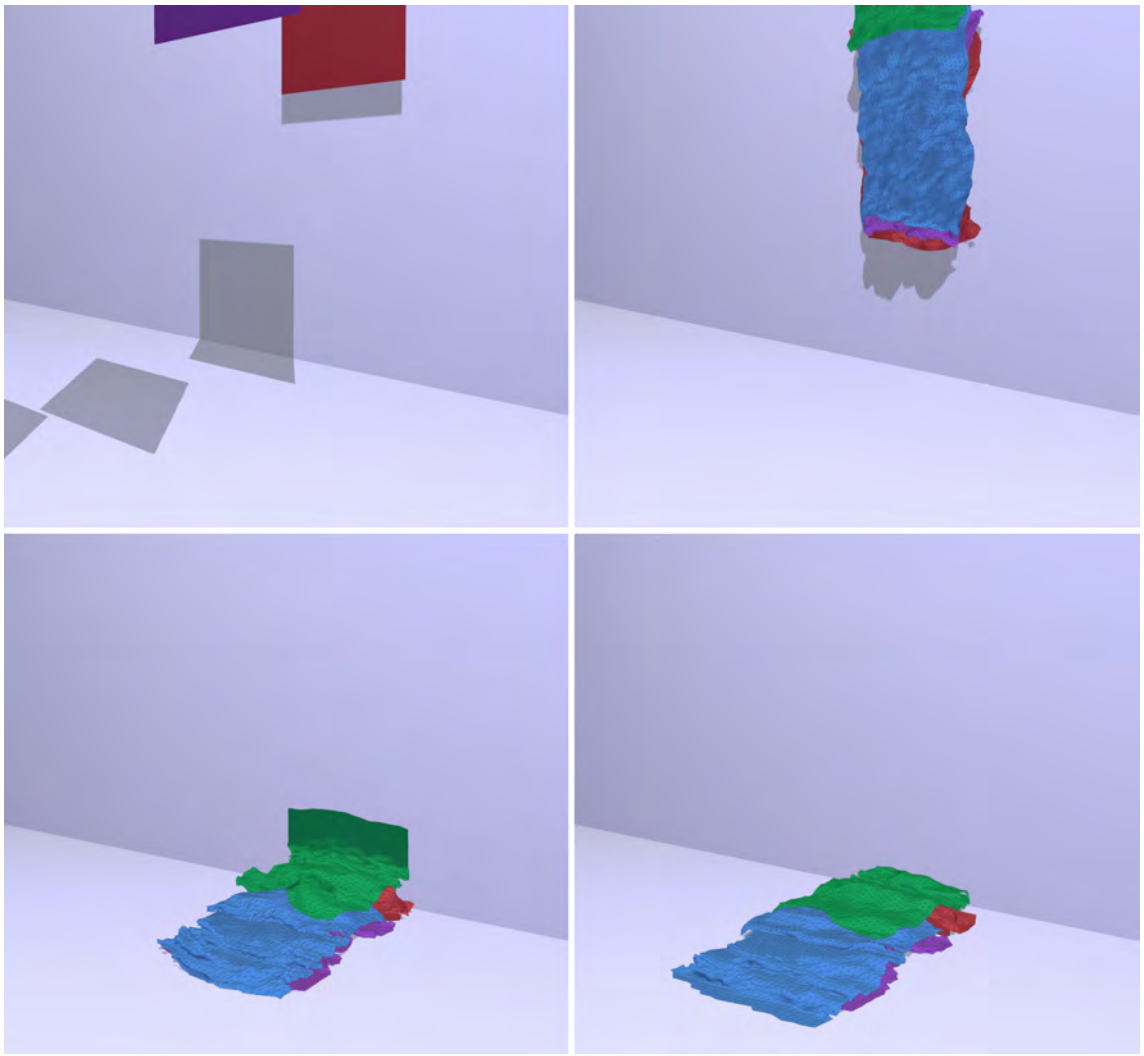

Figure 6.13: *Four cloths of different colors are thrown against a vertical plane.*

The main difference here is that our cloth has been given a much lower velocity than the reference[3]. A higher velocity became infeasible with a time-step of 2 milliseconds, due to inter-cloth penetration. However, our showcase is computed in real-time, while the reference is not.

[3]https://youtu.be/x8Fo2slT2WA?t=1m6s

The following showcase demonstrates three cloths colliding with a mathematically defined funnel. The inspiration for this showcase comes from the video[4] accompanying [Tang et al., 2016].

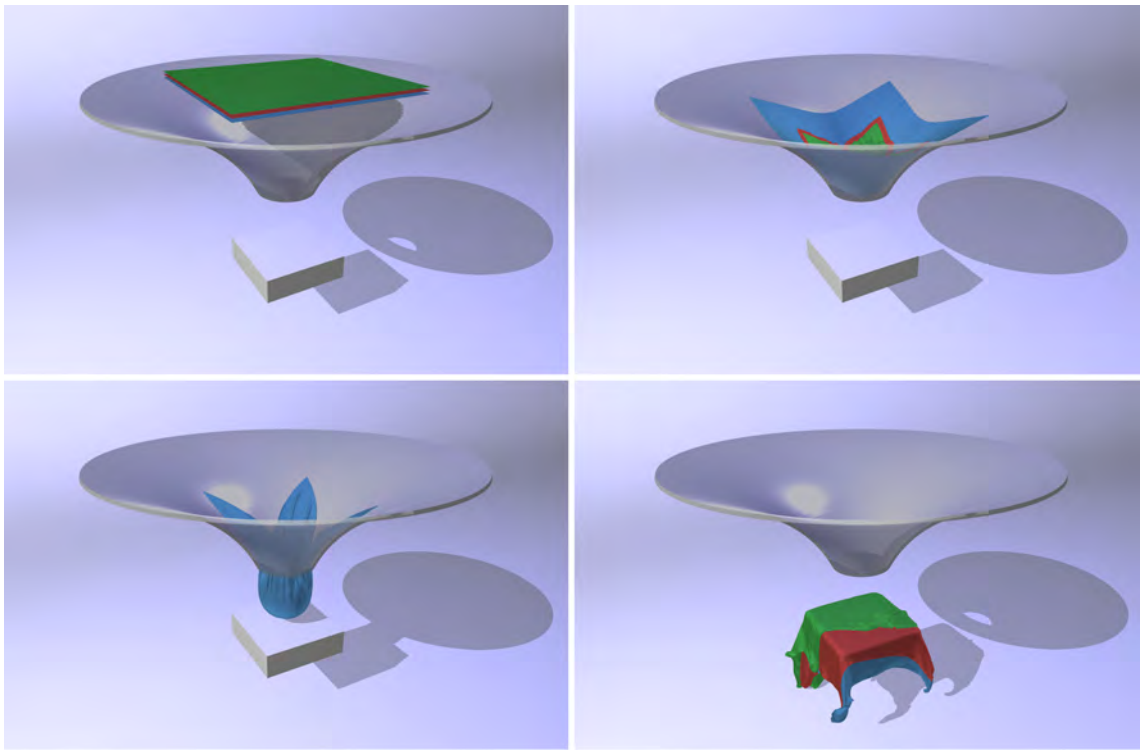

Figure 6.14: *Three cloths colliding with a funnel. After exiting the end of the funnel, they collide with a box.*

In our showcase, some visual artifacts can be seen when the cloths are exiting the end of the funnel. This is due to some compression of the particles, creating a large collision displacement. In the video[4] of [Tang et al., 2016], this problem is not visible. Also, in their example, the wrinkles and the friction between the cloths are more convincing.

User interactions In the following examples, user interaction is demonstrated. Since the purpose of the thesis was to create an interactive physics simulation, showcasing user interaction is of great importance.

In the following scene, mouse picking, dragging and tearing is demonstrated. The entire top row of particles were fixated, otherwise the tearing would be difficult to produce, since the tearing will happen at the weakest point. The inspiration behind this showcase was the tearing example presented in the video[5] accompanying [Müller, 2008].

[4]https://youtu.be/ZVyCfrAu7hY?t=46s
[5]https://youtu.be/FeKkn-Na_Gs?t=1m22s

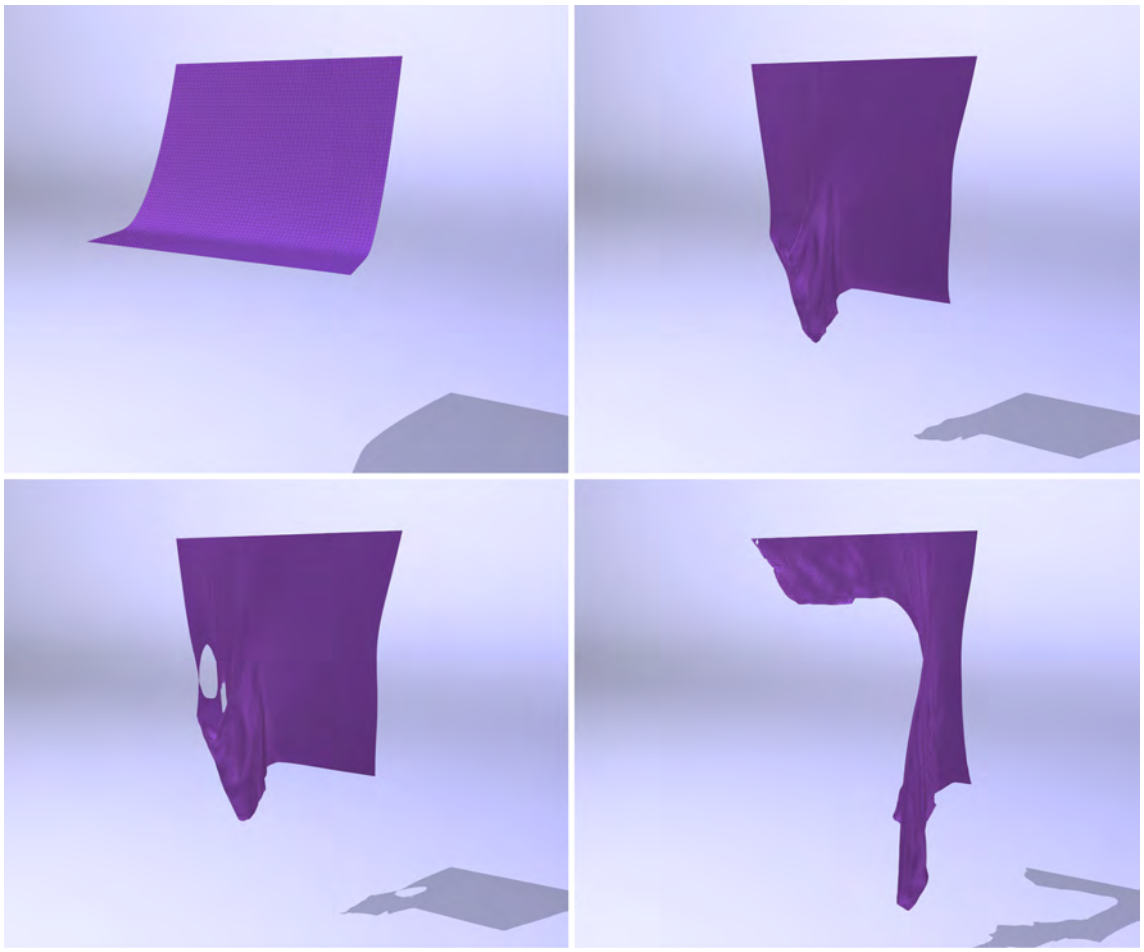

Figure 6.15: *A cloth being torn apart by user interaction. The user picks a point on the cloth and drags it until the cloth tear.*

A difference between the results presented in [Müller, 2008] and ours, is that their cloth does not tear symmetrically, due to their seemingly random structure. In our case, the rest lengths are defined uniformly. This produces a more symmetrical behavior.

The user can also shoot cannonballs at the cloth, which can be seen in the following showcase. This scenario was inspired by the video[6] accompanying [Müller et al., 2007].

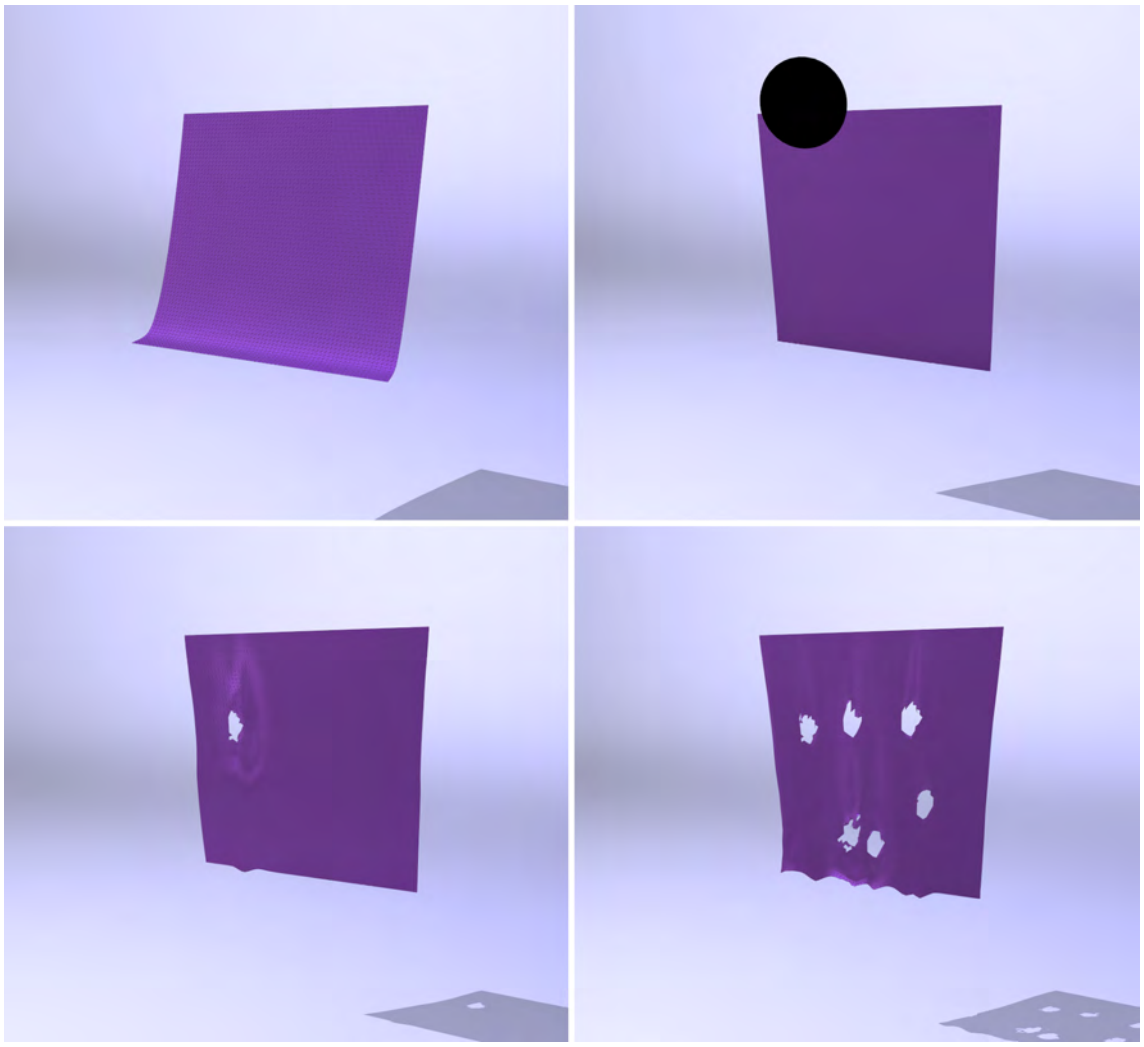

Figure 6.16: *A cannonball being shot at a hanging cloth. The impact produces holes in the cloth.*

In the video of [Müller et al., 2007], the top row is also completely fixated. This is probably due to the same reasons as we fixate our top row. If the cloth was fixated in only a few points, it would break at those points, not being able to show the hole of the cannonball.

Models and animations An important feature to demonstrate is cloth interacting with both static and animated models. For example, this could be used for character clothing in games.

In the rope scene, three ropes are attached to two boxes which are rotating. The

[6] https://youtu.be/j5igW5-h4ZM?t=2m9s

ropes consist of 400 particles each. This example also demonstrates the particle-particle collisions and a collision with a wedge. The scene sought to reproduce the rope animation from the video Advanced Molecular & Particle Physics Simulations[7].

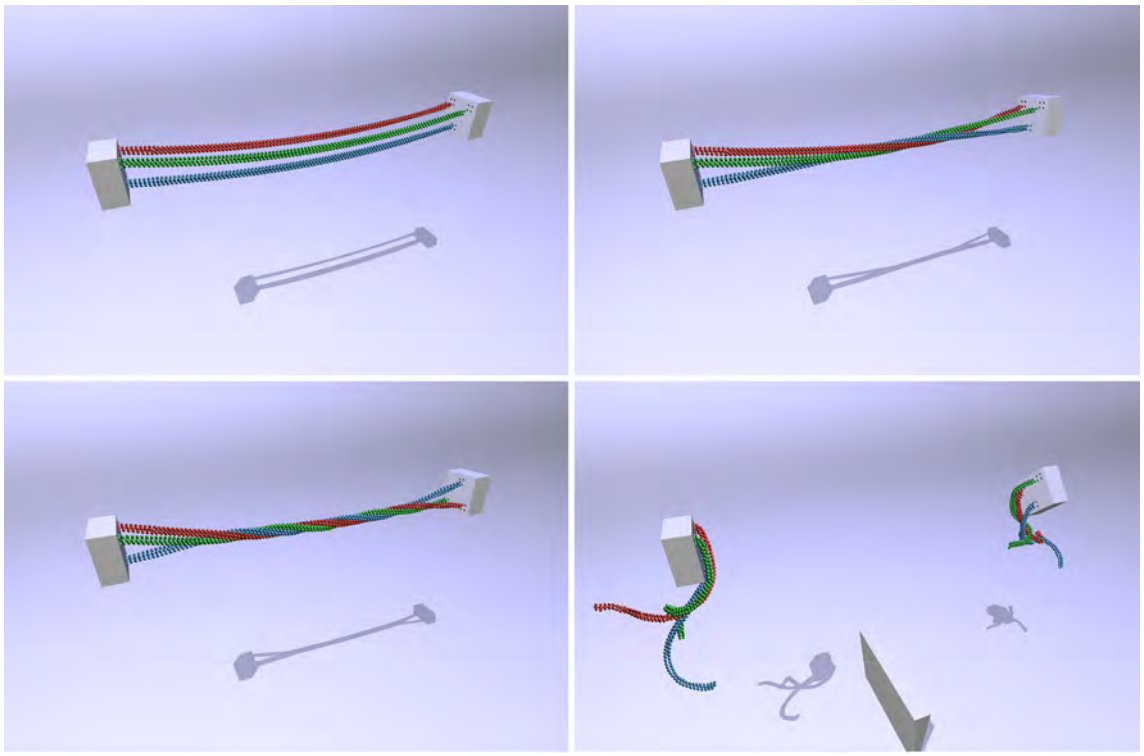

Figure 6.17: *Three ropes attached to one static cuboid and one rotating cuboid. After the cloths are entwined, a wedge falls down and cuts them in half.*

In the Advanced Molecular & Particle Physics Simulation video[7], the animation is not in real-time, while it is run in 60 frames per second in our case. However, in their case, the ropes consist of more particles, and the rotation is faster. This could not be achieved due to the insufficiently precise real-time collision handling.

[7]https://youtu.be/x8Fo2slT2WA?t=17s

The following showcase demonstrates a cloth colliding with a rotating chair. The collisions are handled as described in section *5.5.2 Collision handling*. This showcase was inspired by the video[8] accompanying [Fratarcangeli et al., 2016].

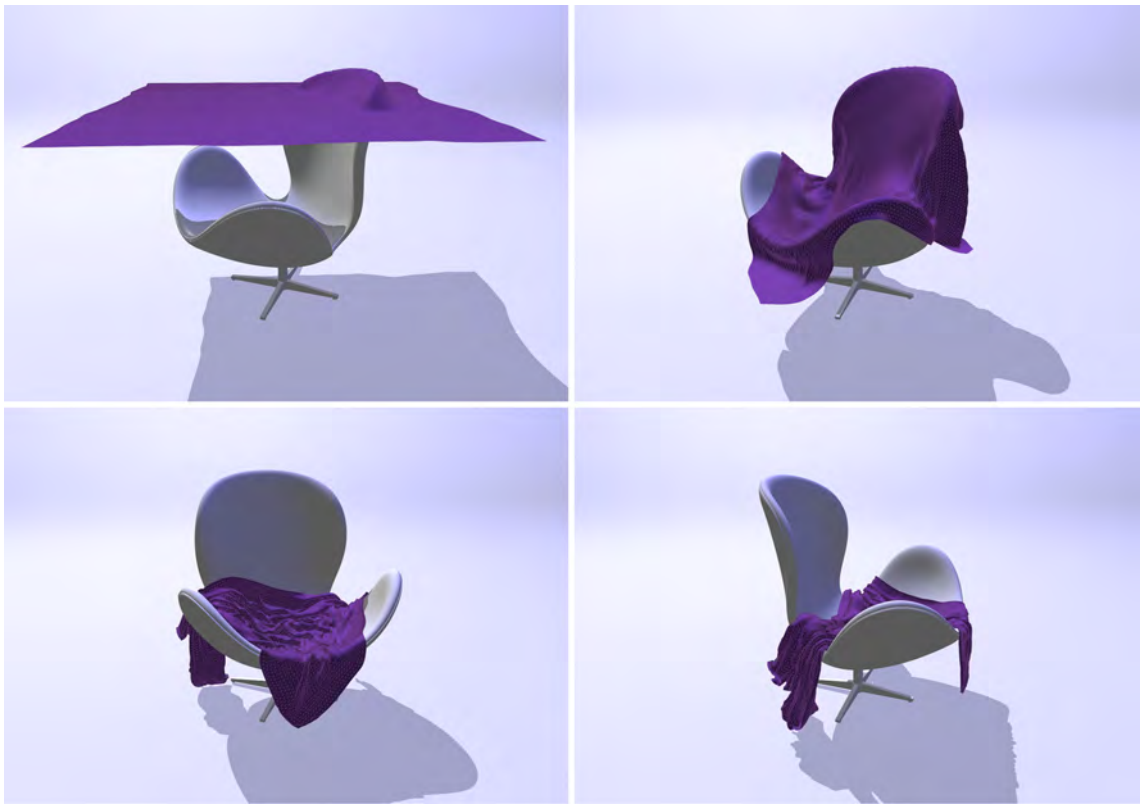

Figure 6.18: *A cloth colliding with a rotating chair, where the kinematic body consists of particles distributed on the surface of the chair.*

There are several disadvantages to be seen with our approach in the comparison. In [Fratarcangeli et al., 2016], the chair is defined by a signed distance field. This makes the collision much more accurate. The friction is also superior, along with better wrinkles. In both cases, the simulation is interactive.

[8] https://youtu.be/xIfuplNTjHc?t=15s

A showcase demonstrating the interaction between cloth and a human model was also created. This can be seen in the figure below. In this example, a poncho is falling over a human character, who moves his arms and sits down on a stool.

Figure 6.19: *A poncho, consisting of 66x66 particles with a hole in the middle, falling on top of an animated character.*

The showcase is played at twice the speed, using a time-step of 1 ms. This is due to the collisions not being detected perfectly at larger time-steps. Thus, a better collision handling needs to be implemented in order for this to be viable in e.g. games.

In the showcase with the hanging curtain, attachment points are demonstrated. The attachment points of the model are also animated, making the curtain move in the desired way.

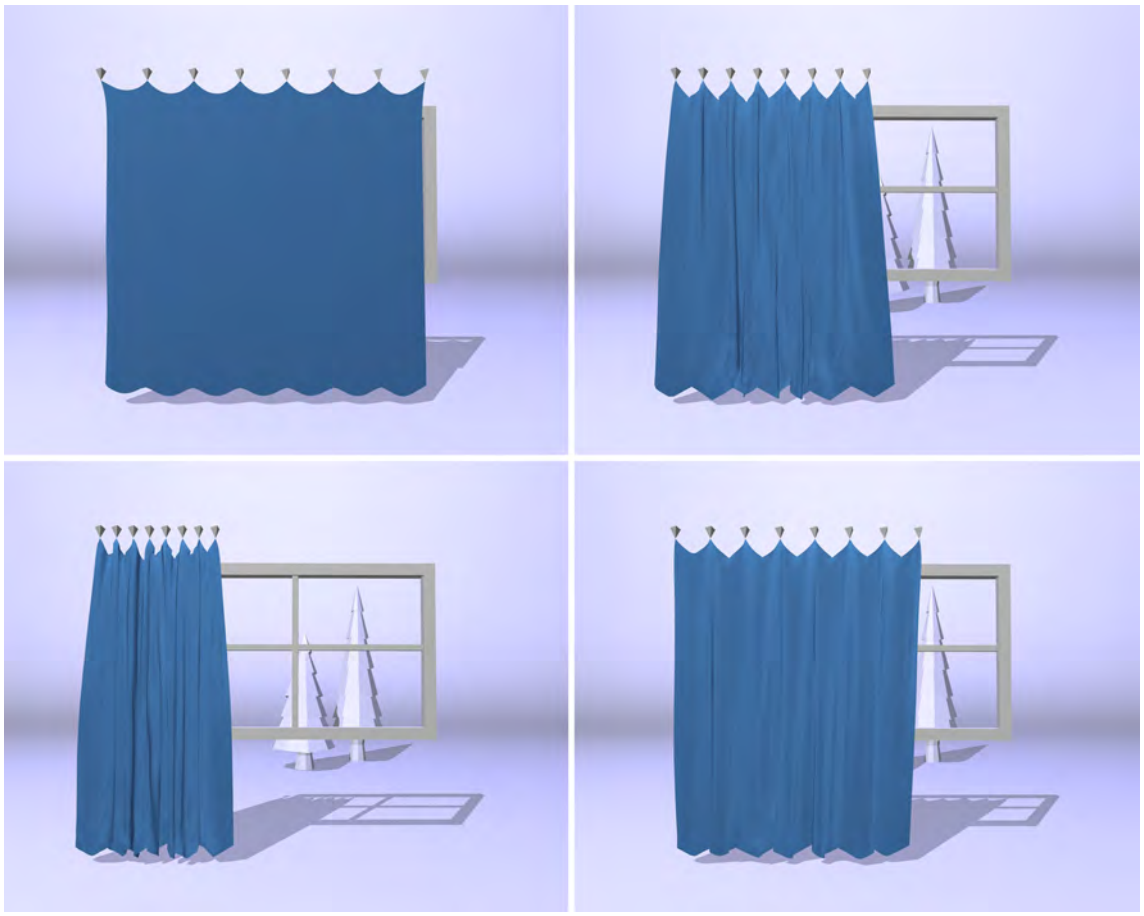

Figure 6.20: *A sliding curtain, made by a cloth attached to several moving points.*

The attachment of the cloth produces some unwanted behavior. This is due to the simplified structure of not having diagonal constraints. The curvature of the cloth in the top, translates to the bottom in an unrealistic way. The curtain should ideally be straight in the bottom.

7
Discussion

This chapter contains a discussion of the results and execution of the thesis. This includes the actual outcome in relation to the expected result, and an analysis of the causes that produced these results. A discussion about how the thesis relates to the previous work is also included.

The results in terms of performance relate well to the theory. A parallel Gauss-Seidel solver should exhibit the same convergence rate per iteration as the standard Gauss-Seidel method, while significantly increasing efficiency, leading to faster solving. The time consumed by one Red-Black Gauss-Seidel solver iteration should be closer to the Jacobi method, since they are both parallel. By combining the advantages of the serial Gauss-Seidel method and the parallel Jacobi method, the Red-Black Gauss-Seidel method should in theory outperform both of these solvers. This theoretical behavior exactly matches the results provided by the experiments.

Due to the quadrangular structure and the absence of constraints other than distance constraints, we suspected that simulating volumetric bodies would be difficult. Furthermore, we also suspected that the cloth would exhibit more elastic behavior than if there were additional diagonal constraints. Both of our suspicions turned out to be true. Another result of the simplified structure that we did not anticipate was the noticeable wrinkles. This is mainly due to there only being distance constraints and no bending constraints. When comparing the visual results to the results of others, some of these drawbacks can be seen. Still, our method is usually faster than the examples with which we compare, making it possible to run the simulations in real-time. The results are presented in section *6.2 Test cases* and in the accompanying video[1].

Our method is a special case of the Vivace solver by [Fratarcangeli et al., 2016], using a predefined coloring scheme. This is possible due to the simplified structure of particles and constraints, allowing for the use of just two colors. The use of a simplified structure means that our physics engine is currently not as widely applicable as other methods presented in chapter *2 Previous work*. Even though our method has a lot of limitations, the idea could be further refined and combined with other methods, thus contributing to the overall knowledge within the field.

In order to counter the internal validity threat of inconsistent performance measur-

[1] https://youtu.be/RPDkt6THJ4k

ing, all experiments were carried out on the same computer. Since the full state of the simulation is known at each point in time, the measuring is quite straightforward. Due to this, no major problems were encountered.

It is difficult to compare our results with the previous work made in soft body simulation, since there is no clearly defined standard for how error measurement should be defined. Two metrics of interest are the number of particles and constraints. For the same amount of particles, different methods might have a different amount of constraints. This leads to differences in the total error, even though the resulting particle positions might be identical.

To account for the conclusional validity threat of contingent bias when conducting the visual analysis, the reference videos were included as footnotes. This gives the readers an opportunity to draw conclusions by themselves, in addition to reading our conclusions.

8
Conclusion

Creating a physics engine for soft body simulation is a complex task. Not only are there many interconnected components which all need to work together, they also need to be computed within a small time budget. This chapter explains what conclusions can be drawn from the results, and in which ways the physics engine can be useful for future purposes.

The research question that this thesis aimed to answer was if a Red-Black Gauss-Seidel solver would be more suitable for real-time simulation than other traditional solvers. This was tested through the Projective Dynamics framework, comparing three different solvers; sequential Gauss-Seidel, parallel Jacobi and parallel Red-Black Gauss Seidel. The results from the comparison can be examined in section *6.1 Performance*.

Given the results achieved, and the analysis of those results, we can be certain that the Red-Black Gauss-Seidel solver performs better than both the Gauss-Seidel solver and the Jacobi solver. The Red-Black Gauss-Seidel solver has the same convergence per iteration as the Gauss-Seidel solver, which is to be expected. But it is much faster, since it is able to solve the system in parallel. Similarly, while both Jacobi and Red-Black Gauss-Seidel solves in parallel, the Red-Black Gauss-Seidel solver has a higher convergence rate per iteration, which results in a better performance overall.

Furthermore, the visual difference between the different solvers suggests that Red-Black Gauss-Seidel is superior. The Jacobi solver, due to its slow convergence, produces a more elastic behavior of the cloth. The sequential Gauss-Seidel solver produces a lower frame rate, since each iteration is very time consuming. This implies that the Red-Black Gauss-Seidel solver is more suitable for real-time soft body simulation than the other two solvers, producing more convincing cloth behavior while keeping a high frame rate. There were no other visual artifacts that differed between the solvers. Thus, we can confidently determine that this conclusion is valid.

However, these results were obtained using a simplified structure of particles and constraints, thus the conclusion can only be drawn under these particular conditions. For a more complex structure, the Red-Black Gauss-Seidel method is not applicable in the way presented in this thesis. The visual implications of this simplified structure can be seen in section *6.2 Test cases*, where our results are compared to other physics engines. Here, it is visible that our method produces more elasticity and less convincing wrinkles. However, our method is significantly faster than most

other methods. This means that our method could be used for applications where a highly approximate solution is sufficient. This could include fields such as games and offline animation prototyping tools. For applications where physical accuracy is important, improvements might have to be implemented.

Despite the shortcomings of our method, it shows the potential of using a simplified structure. The results could be used as a starting point for further research, where this method is combined with other methods in order to increase the applicability of the Red-Black Gauss-Seidel solver. This could be achieved by solving distance constraints with Red-Black Gauss-Seidel, and other constraints with other methods. The method could also be further generalized by introducing several sets of distance constraints, each with a red-black graph coloring.

9
Future work

There are several improvements and extensions that could be implemented to improve the physics engine. Some of these were not implemented due to time limitations, and some were not within the scope of the project. These possible improvements will be presented in this chapter.

One of the most notable limitations with the current state of the engine is the simplified structure of constraints. There are two main problems with this structure. Firstly, the deformation of the cloth is not entirely satisfactory. Due to the lack of support from the constraints, it exhibits a more elastic behavior than if diagonal distance constraints or bending constraints would be included. Additionally, cloths are quite wrinkly.

Secondly, the amount of soft object types that the current state of the engine can simulate is very limited. It can only simulate very simple objects such as cloth and ropes. Trying to simulate volumetric bodies produces poor results. The bodies preserve the volume quite well, due to collisions and constraints interacting with each other. However, the shape is not well preserved, due to the lack of support internally in the body. This is an effect from the grid graph structure used.

Many of these issues could be dealt with by introducing bending constraints. However, this type of constraint considers 4 particles, which means that more colors would have to be added. It would produce more convincing wrinkles on the cloth, as well as shape conservation for volumetric bodies.

Another way of making the cloths less wrinkly would be to apply a low-pass filter on the velocities. Compared to introducing a completely new constraint type, this is a much smaller task. However, its inclusion could still be of great significance for the smoothness of soft objects in the engine.

Although the engine requires quadrangular structures of particles and constraints, this does not limit the range of meshes which can be simulated. Any triangular mesh can be transformed into a quadrangular mesh, using the approach described by [Jakob et al., 2015]. This work was generalized in the recent work by [Gao et al., 2017], where a tetrahedral mesh is changed into a hexahedral-dominant mesh. The output mesh can then be partitioned into red and black particles. In theory, this makes it possible to use any mesh with our engine. This has yet to be tried.

The engine could also be further optimized in terms of memory allocation, to allow for more efficient GPU kernel calls. For example, instead of storing particle positions in one array, the positions of the red and black particles could be divided into separate arrays, further streamlining the solving process on the GPU. Several such optimizations could be implemented in order to improve the performance of the engine.

A major problem is that the collision handling only works well for small time steps. With a large time step, which is desired for games, for example, a more sophisticated collision handling would need to be implemented. One way this can be achieved is by projecting the particles in their movement direction, to see if they will collide in the next time step. This could be thought of as capsules extending into the next position of the particle. In this way, the collision detection would be able to handle larger time steps.

There are several improvements that could be done to the rendering. The computed fragment colors are currently applied to both sides, which means that the side facing away from the light source still looks lit up. Another desirable feature would be to have soft shadows. This could be achieved by using for example variance shadow mapping or Poisson sampling with percentage closed filtering.

These are some of the most impactful improvements, that would enhance the physics engine significantly. All of these improvements could be seen as extensions to the current state of the engine. No substantial alterations would have to be made to the underlying architecture in order to realize these extensions.

Bibliography

Baraff, D. and Witkin, A. 1998. Large steps in cloth simulation. ACM, 43–54.

Barrett, R., Berry, M., Chan, T. F., Demmel, J., Donato, J., Dongarra, J., Eijkhout, V., Pozo, R., Romine, C., and der Vorst, H. V. 1994. *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods, 2nd Edition*. SIAM, Philadelphia, PA.

Bender, J., Müller, M., and Macklin, M. 2015. Tutorial: Position-Based Simulation Methods in Computer Graphics.

Bender, J., Müller, M., and Macklin, M. 2017. A survey on position based dynamics, 2017. In *EUROGRAPHICS 2017 Tutorials*. Eurographics Association.

Bouaziz, S., Martin, S., Liu, T., Kavan, L., and Pauly, M. 2014. Projective Dynamics : Fusing Constraint Projections for Fast Simulation. *ACM Transactions on Graphics 33,* 4, 154.

Bridson, R., Fedkiw, R., and Anderson, J. 2002. Robust treatment of collisions, contact and friction for cloth animation. Vol. 21. ACM, NEW YORK, 594–603.

Chapman, B., Jost, G., and Pas, R. 2008. *Using OpenMP: portable shared memory parallel programming*. MIT Press, Cambridge, Mass.

Demidov, D., Ahnert, K., Rupp, K., and Gottschling, P. 2013. Programming cuda and opencl: A case study using modern c++ libraries. *SIAM Journal on Scientific Computing 35,* 5, C.453–C472.

Deng, Y. 2013. *Applied parallel computing*. World Scientific, Hackensack, NJ;Singapore;.

Eberly, D. H. and Shoemake, K. 2004. *Game physics*. Elsevier Science, Amsterdam.

Faure, F. 1999. *Interactive Solid Animation Using Linearized Displacement Constraints*. Springer Vienna, Vienna, 61–72.

Fratarcangeli, M. and Pellacini, F. 2015. Scalable partitioning for parallel position based dynamics. *Computer Graphics Forum 34,* 2, 405–413.

Fratarcangeli, M., Tibaldo, V., and Pellacini, F. 2016. Vivace: A practical gauss-seidel method for stable soft body dynamics. *ACM Trans. Graph. 35,* 6 (Nov.), 214:1–214:9.

Gao, X., Jakob, W., Tarini, M., and Panozzo, D. 2017. Robust hex-dominant mesh generation using field-guided polyhedral agglomeration. *ACM Transactions on Graphics (Proceedings of SIGGRAPH) 36,* 4 (July).

Godsil, C. D. and Royle, G. 2001. *Algebraic graph theory*. Vol. 207. Springer, New York.

Green, S. 2010. Particle simulation using cuda. *NVIDIA whitepaper 6*, 121–128.

Haines, E. and Akenine-Möller, T. 2002. Real-time rendering.

Jakob, W., Tarini, M., Panozzo, D., and Sorkine-Hornung, O. 2015. Instant field-aligned meshes. *ACM Transactions on Graphics (Proceedings of SIGGRAPH Asia) 34,* 6 (Nov.), 189:1–189:15.

Jakobsen, T. 2001. Advanced character physics. In *Game Developers Conference*. Vol. 3. Citeseer.

John Kessenich, Dave Baldwin, R. R. 2016. The opengl shading language.

Kim, D. 2016. *Fluid Engine Development*. A K Peters/CRC Press.

Kim, H., Vuduc, R., Baghsorkhi, S., and Choi, J. 2012. *Performance Analysis and Tuning for General Purpose Graphics Processing Units (GPGPU)*. Morgan Claypool.

Le Grand, S. 2007. Broad-phase collision detection with cuda. *GPU gems 3*, 697–721.

Liu, T., Bargteil, A. W., O'Brien, J. F., and Kavan, L. 2013. Fast simulation of mass-spring systems. *ACM Transactions on Graphics 32,* 6.

Liu, T., Bouaziz, S., and Kavan, L. 2017. Quasi-newton methods for real-time simulation of hyperelastic materials. *ACM Trans. Graph. 36,* 3 (May), 23:1–23:16.

Macklin, M. and Müller, M. 2013. Position based fluids. *ACM Trans. Graph. 32,* 4 (July), 104:1–104:12.

Müller, M. 2008. Hierarchical position based dynamics. *Proceedings of Virtual Reality Interactions and Physical Simulations*, 13–22.

Müller, M., Heidelberger, B., Hennix, M., and Ratcliff, J. 2007. Position based dynamics. *J. Vis. Comun. Image Represent. 18,* 2 (Apr.), 109–118.

Nvidia. 2017. Cuda c programming guide.

O'Brien, C., Dingliana, J., and Collins, S. 2011. Spacetime vertex constraints for dynamically-based adaptation of motion-captured animation. In *Proceedings of the 2011 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*. SCA '11. ACM, New York, NY, USA, 277–286.

Provot, X. 1995. Deformation Constraints in a Mass Spring Model to Describe Rigid Cloth Behavior. *Integr. Vlsi J.*, 147–154.

Rungjiratananon, W., Kanamori, Y., and Nishita, T. 2010. Chain shape matching for simulating complex hairstyles. *Computer Graphics Forum 29,* 8, 2438–2446.

Saad, Y. 2001. *Parallel iterative methods for sparse linear systems*. Vol. 8. Chapter C, 423–440.

Sanders, J. and Kandrot, E. 2010. *CUDA by Example: An Introduction to General-Purpose GPU Programming*, 1st ed. Addison-Wesley Professional.

Stam, J. 2009. Nucleus: Towards a unified dynamics solver for computer graphics. In *2009 11th IEEE International Conference on Computer-Aided Design and Computer Graphics*. 1–11.

Strzodka, R. 2012. Data layout optimization for multi-valued containers in opencl. *Journal of Parallel and Distributed Computing 72,* 9, 1073–1082.

Sunden, E. and Ropinski, T. 2015. Efficient volume illumination with multiple light sources through selective light updates. *IEEE Pacific Visualization Symposium 2015-July*, 231–238.

Tang, M., Wang, H., Tang, L., Tong, R., and Manocha, D. 2016. Cama: Contact-aware matrix assembly with unified collision handling for gpu-based cloth simulation. *Computer Graphics Forum 35,* 2, 511–521.

Terzopoulos, D., Platt, J., Barr, A., and Fleischer, K. 1987. Elastically deformable models. *ACM SIGGRAPH Comput. Graph. 21,* 4, 205–214.

Wang, H. 2015. A Chebyshev semi-iterative approach for accelerating projective and position-based dynamics. *ACM Trans. Graph. 34,* 6 (Oct.), 246:1–246:9.

Wang, H. and Yang, Y. 2016. Descent methods for elastic body simulation on the gpu. *ACM Trans. Graph. 35,* 6 (Nov.), 212:1–212:10.

Wohlin, C., Runeson, P., Höst, M., Ohlsson, M. C., Regnell, B., and Wesslén, A. 2000. *Experimentation in Software Engineering: An Introduction*, 1 ed. Vol. 6. Springer US, Boston, MA.

A
Appendix 1

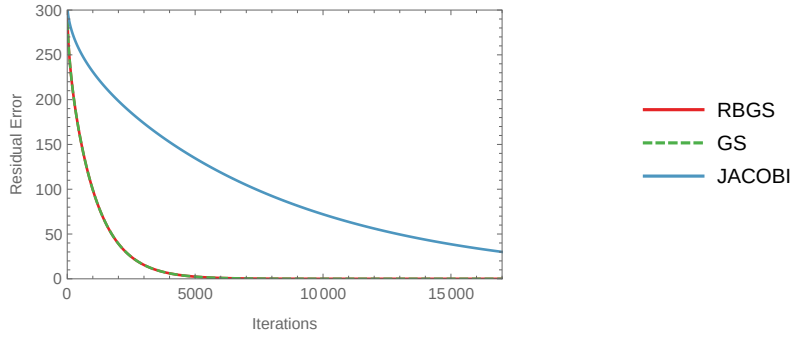

Figure A.1: *Residual error vs. iterations for the enlarged cloth, using 10 000 particles and 19 800 constraints. The convergence speed of our method (red) is compared with Gauss-Seidel (green) and Jacobi (blue).*

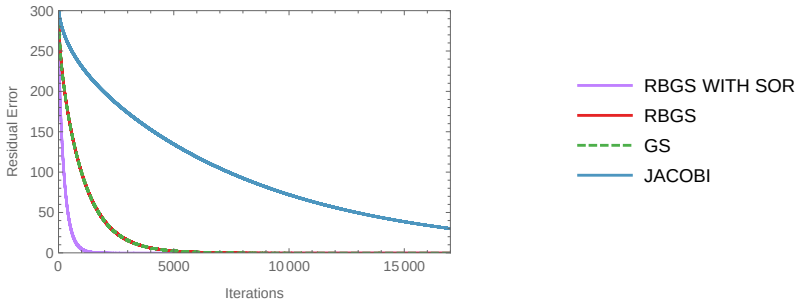

Figure A.2: *Residual error vs. iterations for the enlarged cloth, using 10 000 particles and 19 800 constraints. The convergence speed of our method (red) is compared with Gauss-Seidel (green) and Jacobi (blue). In this graph, RBGS with SOR using an ω value of 1.7 is included.*

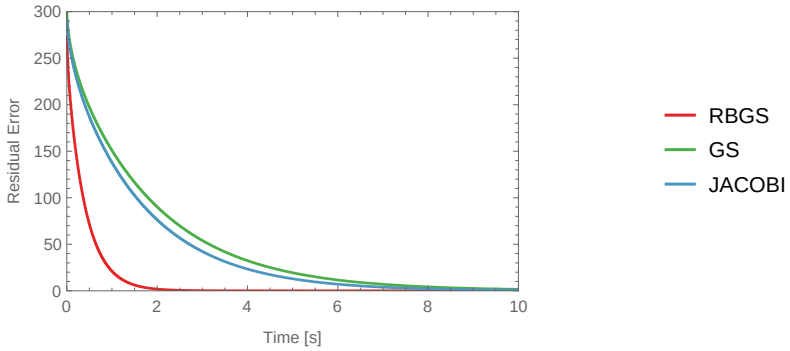

Figure A.3: *Residual error vs. time for the enlarged cloth, using 10 000 particles and 19 800 constraints and a time budget of 1 ms. The convergence speed of our method (5 iterations) is compared with Gauss-Seidel (1 iteration) and Jacobi (9 iterations).*

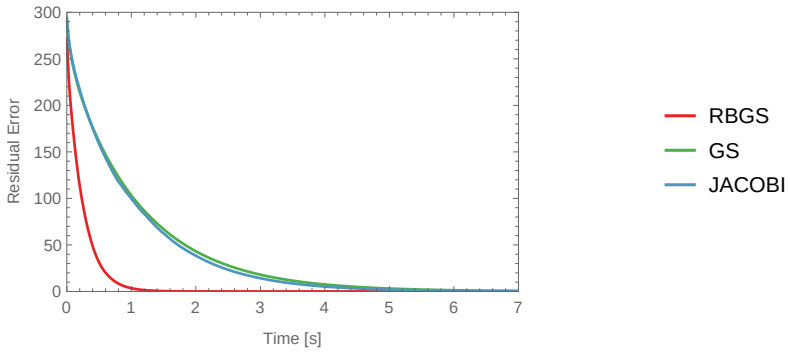

Figure A.4: *Residual error vs. time for the enlarged cloth, using 10 000 particles and 19 800 constraints and a time budget of 2 ms. The convergence speed of our method (11 iterations) is compared with Gauss-Seidel (2 iterations) and Jacobi (19 iterations).*

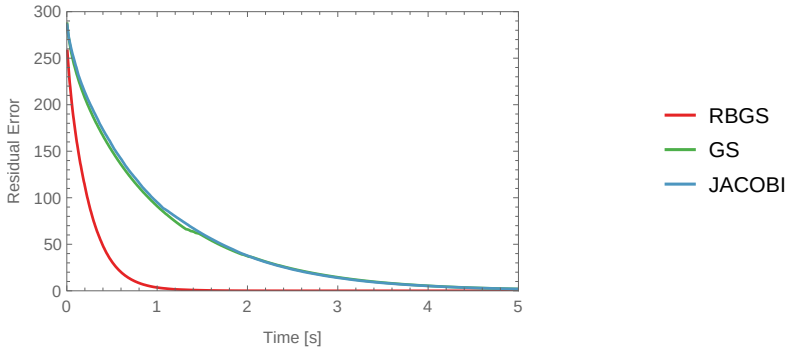

Figure A.5: *Residual error vs. time for the enlarged cloth, using 10 000 particles and 19 800 constraints and a time budget of 4 ms. The convergence speed of our method (22 iterations) is compared with Gauss-Seidel (4 iterations) and Jacobi (32 iterations).*

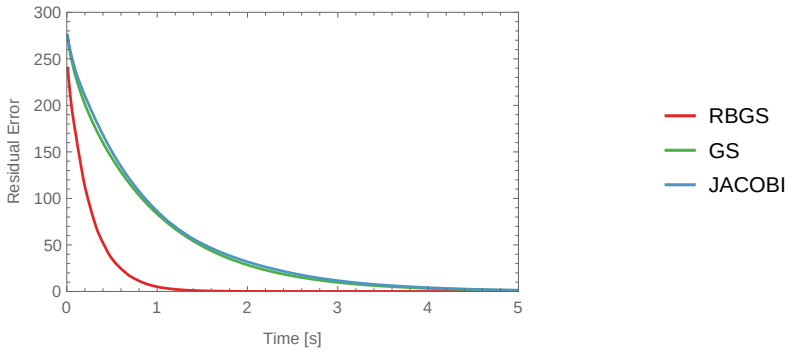

Figure A.6: *Residual error vs. time for the enlarged cloth, using 10 000 particles and 19 800 constraints and a time budget of 8 ms. The convergence speed of our method (45 iterations) is compared with Gauss-Seidel (9 iterations) and Jacobi (70 iterations).*

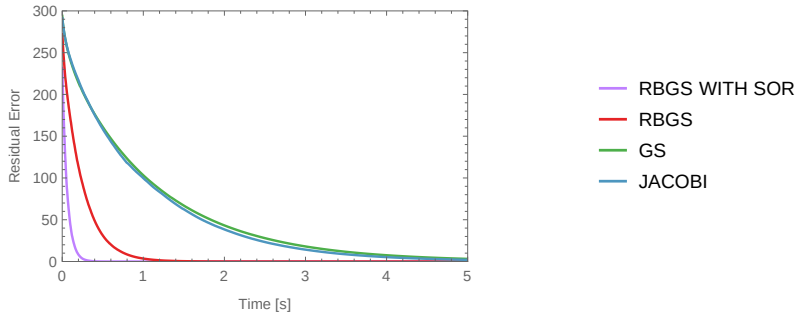

Figure A.7: *Residual error vs. time for the enlarged cloth, using 10 000 particles and 19 800 constraints and a time budget of 2 ms. The convergence speed of our method (11 iterations) is compared with Gauss-Seidel (2 iterations) and Jacobi (19 iterations). In this graph, RBGS with SOR (11 iterations) using an ω value of 1.7 is included.*

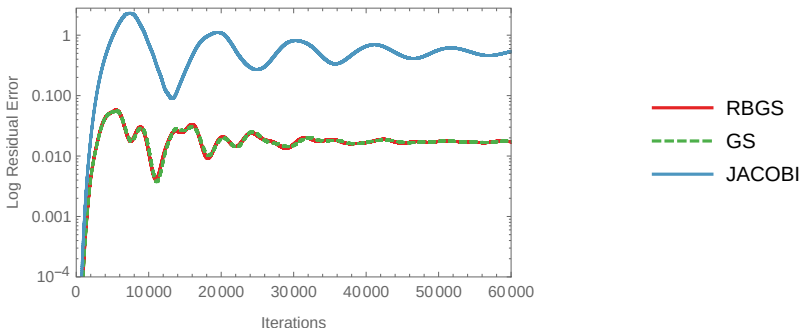

Figure A.8: *Residual error vs. iterations for the hanging cloth, using 10 000 particles and 19 800 constraints. The convergence speed of our method (red) is compared with Gauss-Seidel (green) and Jacobi (blue).*

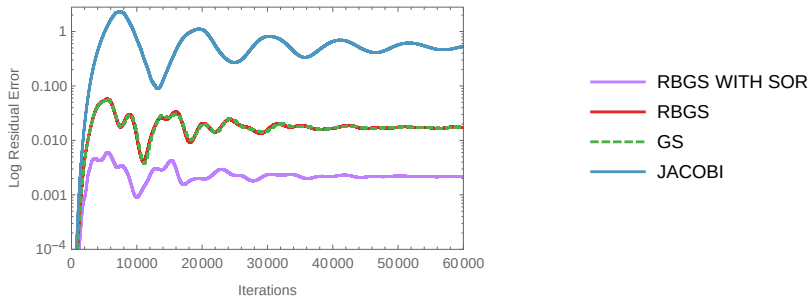

Figure A.9: *Residual error vs. iterations for the hanging cloth, using 10 000 particles and 19 800 constraints. The convergence speed of our method (red) is compared with Gauss-Seidel (green) and Jacobi (blue). In this graph, RBGS with SOR using an ω value of 1.7 is included.*

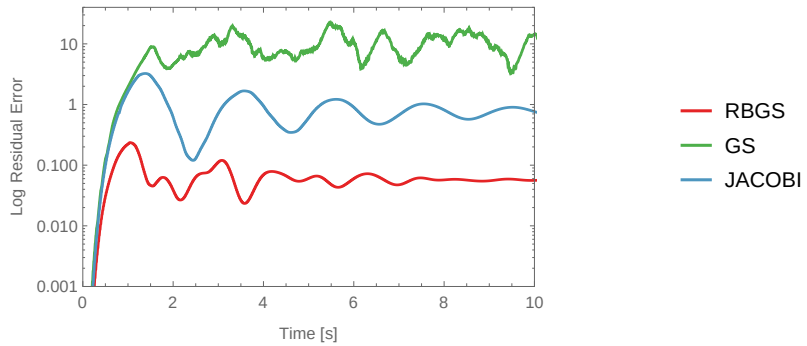

Figure A.10: *Residual error vs. time for the hanging cloth, using 10 000 particles and 19 800 constraints and a time budget of 1 ms. The convergence speed of our method (5 iterations) is compared with Gauss-Seidel (1 iterations) and Jacobi (9 iterations).*

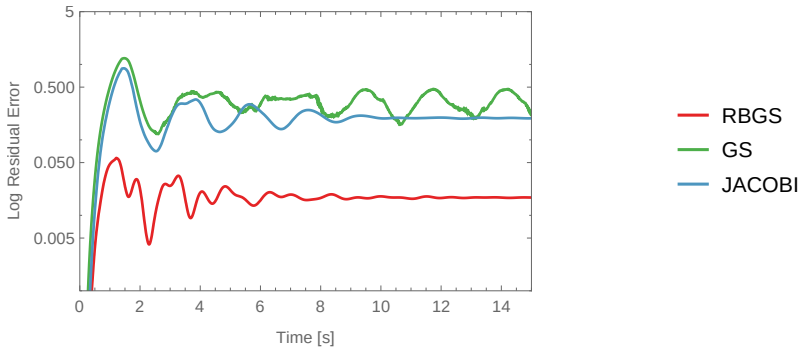

Figure A.11: *Residual error vs. time for the hanging cloth, using 10 000 particles and 19 800 constraints and a time budget of 2 ms. The convergence speed of our method (11 iterations) is compared with Gauss-Seidel (2 iterations) and Jacobi (19 iterations).*

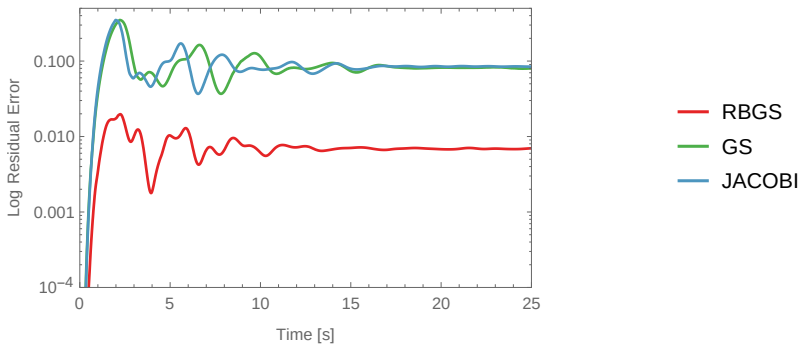

Figure A.12: *Residual error vs. time for the hanging cloth, using 10 000 particles and 19 800 constraints and a time budget of 4 ms. The convergence speed of our method (22 iterations) is compared with Gauss-Seidel (4 iterations) and Jacobi (32 iterations).*

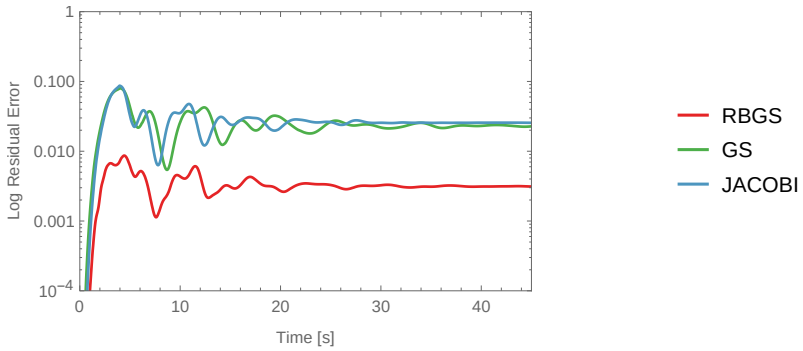

Figure A.13: *Residual error vs. time for the hanging cloth, using 10 000 particles and 19 800 constraints and a time budget of 8 ms. The convergence speed of our method (45 iterations) is compared with Gauss-Seidel (9 iterations) and Jacobi (70 iterations).*

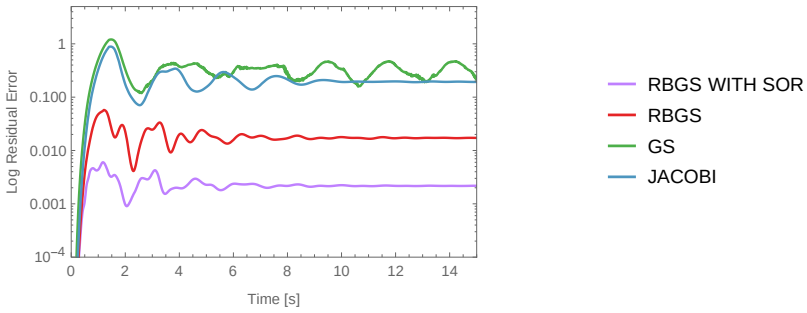

Figure A.14: *Residual error vs. time for the hanging cloth, using 10 000 particles and 19 800 constraints and a time budget of 2 ms. The convergence speed of our method (11 iterations) is compared with Gauss-Seidel (2 iterations) and Jacobi (19 iterations). In this graph, RBGS with SOR (11 iterations) using an ω value of 1.7 is included.*