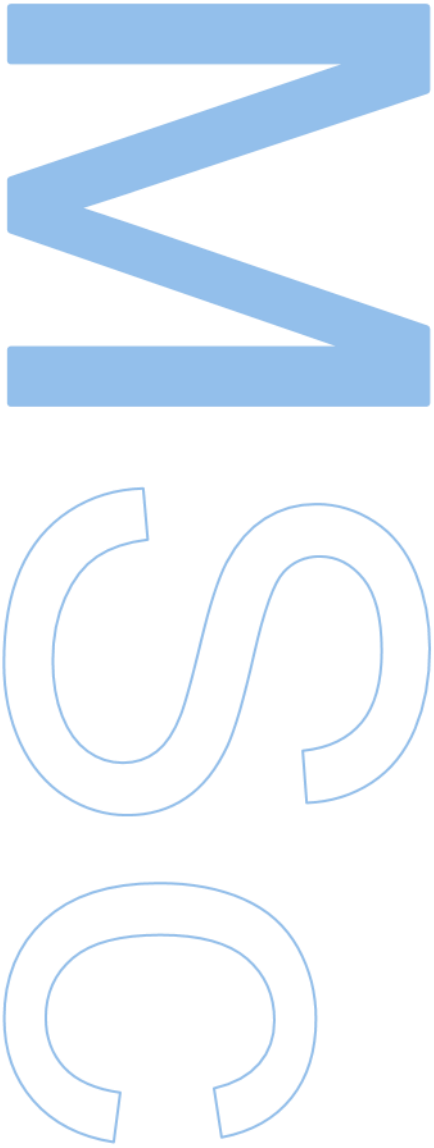
Massively parallel visual simulation of deformable objects on the GPU



Pedro Ferreirinha
Computer Science
Department of Computer Science
2023

Supervisor
Cumhur Ozan Çetinaslan, Junior Researcher, Instituto de Telecomunicações

Co-supervisor
Verónica Costa Pinto Teixeira Orvalho, Professor Auxiliar, Faculdade de Ciências da Universidade do Porto


U.PORTO
FACULDADE DE CIÊNCIAS
UNIVERSIDADE DO PORTO

Abstract

This thesis presents an approach to real-time physically-based simulation of deformable objects in a parallel way, based on Extended Position-based Dynamics (XPBD) with its unique Gauss-Seidel iterative method. The proposed approach uses a graph coloring algorithm based on DSTAUR (degree of saturation) in the pre-computation stage. This algorithm divides the simulation into independent groups of primitives, which avoids race conditions and enables efficient parallelization. The resulting simulations can be executed in a parallel and non-parallel way on both the CPU and GPU.

The proposed approach is evaluated on various deformable objects, including 2D meshes for cloth and 3D meshes for volumetric models. The results show that the proposed approach is able to achieve high performance on both the CPU and GPU while maintaining a high level of stability.

Keywords: framework, physics-based simulation, parallel physics-based simulation, extended position-based dynamics, graph coloring

Resumo

Esta tese apresenta uma abordagem para a simulação baseada em física em tempo real de objetos deformáveis de forma paralela, baseada em Extended Position-based Dynamics (XPBD) com o seu unico metodo iterativo Gauss-Seidel. A abordagem proposta utiliza um algoritmo de coloração de grafos baseado no DSATUR (grau de saturação) na fase de pré-computação para dividir a simulação em grupos independentes de primitivas, o que evita condições de corrida e permite uma paralelização eficiente. As simulações resultantes podem ser executadas de um modo paralelo e não paralelo tanto na CPU quanto na GPU.

A abordagem proposta é avaliada em uma variedade de objetos deformáveis, incluindo geometria 2D para tecidos, corpos moles e geometria 3D para modelos volumétricos. Os resultados mostram que a abordagem proposta é capaz de atingir um alto desempenho na CPU e na GPU, mantendo um alto grau de estabilidade.

Palavras-chave: framework, simulação baseada em física, simulação baseada em física em paralelo, extended position-based dynamics, coloração de grafos

Acknowledgments

I would like to take this opportunity to express my sincere gratitude to all those who have contributed to the completion of this thesis.

First and foremost, I want to thank my family for their unwavering support, encouragement, and belief in my abilities. Your love and encouragement have been my pillars of strength throughout this journey.

I extend my heartfelt appreciation to my friends for their camaraderie and understanding during the challenging moments of my academic pursuit. Your friendship has added warmth and joy to this remarkable journey.

I am profoundly grateful to my co-advisor, Verónica Orvalho, for affording me the opportunity to pursue my dream in the fascinating field of computer graphics. Your guidance, mentorship, and belief in my potential have been invaluable.

My deepest gratitude goes to my advisor, Ozan Çetinaslan, for your unwavering support, patience, and exceptional guidance. Your dedication to excellence, willingness to answer my countless questions, and insistence on the highest standards have been instrumental in shaping this research. I am grateful for your continuous encouragement and for never giving up on me.

Finally, I would like to gratefully acknowledge IT, UIDB/50008/2020 funded by the applicable financial framework (FCT/MCTES, PIDDAC), for hosting this research work and for the strong institutional support.

Contents

List of Tables

List of Figures

Acronyms

FCUP Faculdade de Ciências da Universidade do Porto

DCC Departamento de Ciência de Computadores

PBD Position-based Dynamics

XPBD Extended Position-based Dynamics

GPU Graphics Processing Unit

CPU Centeral Processing Unit

FEM Finite Element Method

PCG Preconditioned Conjugate Gradient

GMRES Generalized Minimal Residual Method

CAD Computer-aided Design

GUI Graphical User Interface

FFD Free-form Deformation

PCG Preconditioned Conjugate Gradient

RK4 Fourth-order Runge-Kutta

DSATUR Degree of Saturation

FPS Frames per Second

Chapter 1

Introduction

1.1 Motivation

Deformable objects, such as cloth or soft bodies, are integral to many virtual environments, simulations, and computer graphics applications. Simulating the behavior of these objects is essential for achieving realistic and immersive virtual experiences. Physically-based simulation has emerged as a compelling and indispensable tool in various fields, including computer graphics, virtual reality, animation, video games, and scientific research. By using the laws of physics, this type of simulation can offer a remarkable level of realism, enabling the visual representation of complex natural phenomena with astounding accuracy.

This thesis will focus on one of the cornerstones of physically-based simulation, Position-based Dynamics (PBD) [40] and its Gauss-Seidel solver [37], a powerful technique that has garnered significant attention and acclaim for its ability to efficiently simulate deformable objects, fluid dynamics, cloth behavior, and more.

The appeal of PBD lies in its simplicity, stability, and versatility, allowing it to handle a wide range of physical interactions with relative ease. However, as the demand for more intricate and realistic simulations increases, the need for improved performance becomes paramount. The scale and complexity of modern simulations, coupled with ever-advancing hardware capabilities, necessitate exploring new avenues for optimizing simulation performance.

Traditionally, common solvers in physics-based simulators have been implemented in a serial manner, leading to performance bottlenecks and limiting the potential for faster computations. Parallelization presents itself as a promising solution to this challenge. However, the direct application of parallelization to the PBD algorithm may lead to unpredictable results. One example is that two primitives (edges or faces) that share the same particle can be processed by two different threads at the same time [15]. Also known as a race condition.

Direct parallel implementations are not successful due to the well-known race conditions. However, by taking advantage of precomputation techniques such as graph coloring [26], it is

possible to convert serial solvers to parallel either on CPU or GPU.

1.2 Problem Statement

To outline the specific challenge we aim to tackle, it's crucial to delve into the intricacies of Position-Based Dynamics (PBD), particularly the pivotal "Constraint Projection" step. In this phase, particles undergo projection to conform to simulation constraints, and the current solvers operate sequentially. Despite the solver traversing all mesh primitives (edges or triangles) during this stage, parallelization is hindered by the inherent non-deterministic behavior that arises when multiple threads process the same particle simultaneously, a phenomenon known as a race condition.

Various attempts have been made to address this issue. For instance, Jan Bender and Daniel Bayer [7] proposed a context-specific precomputation process involving the division of the mesh into horizontal and vertical strips. However, this method's applicability is limited to quadratic meshes, which are not commonly encountered. Marco Fratarcangeli and collaborators [26] introduced a precomputation process featuring a non-deterministic graph coloring algorithm. This algorithm randomly colors each primitive in the graph, followed by conflict resolution for connected primitives sharing the same color. While effective, this introduces an element of randomness, and our aim is to develop a more deterministic process.

Of particular relevance to our thesis is the work by Ozan Cetinaslan [16], who introduced a precomputation process dividing the mesh into a non-adjacent primitive list and further segregating it into six groups, ensuring that within each group, no primitives share particles. However, the drawback lies in the time-intensive nature of this process, rendering it impractical for real-time applications.

The primary objective of our work is to build upon the initial framework incorporating the precomputation process of [16] and devise a novel graph coloring algorithm that solves the race condition problem. This algorithm should significantly reduce precomputation time, enabling its feasibility in a real-time environment.

1.3 Contribution

Current techniques for pre-computing deformable models can be time-consuming [15]. This is a significant bottleneck for physically-based simulation, as it can limit the waiting time of the simulation.

The main contribution of this thesis is to address the aforementioned problem by reducing the pre-computation time significantly. The proposed approach utilizes a graph coloring algorithm based on DSATUR (Degree of Saturation) [12]. The degree of saturation of a particular vertex refers to the count of neighboring vertices that have distinct colors. The primary objective of

this algorithm is to iteratively select vertices in a greedy manner based on the property that their neighbors are more diversely colored. To the best of our knowledge, this is the first time this algorithm has been adapted in the context of computer graphics applications.

The graph coloring algorithm is used for dividing the primitives into groups. This is done in a way that avoids race conditions, which allows the serial solvers to be converted to parallel. The memory management scheme is then used to allocate memory for the solver, ensuring that the solver can be executed efficiently on a variety of hardware platforms.

In the context of this thesis we present how to create a primitive constraint graph out of a mesh inspired from [25] (in Figure 1.1) and then how to convert that into a fully colorized constraint graph using DSATUR for the parallelization (in Figure 1.2).

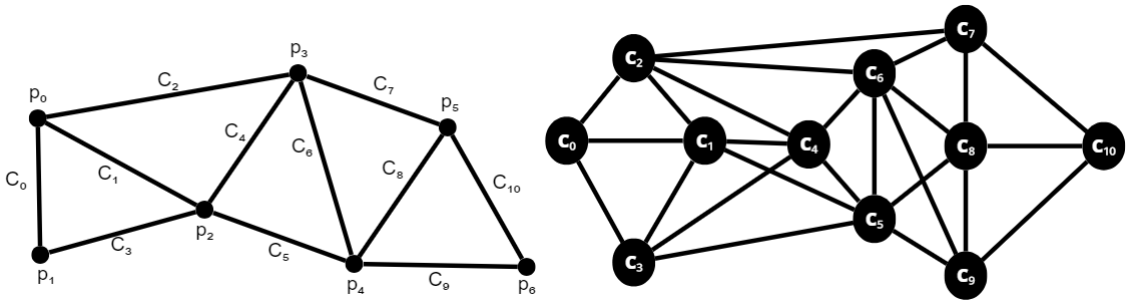


Figure 1.1: Edge constraint graph inspired from [25]

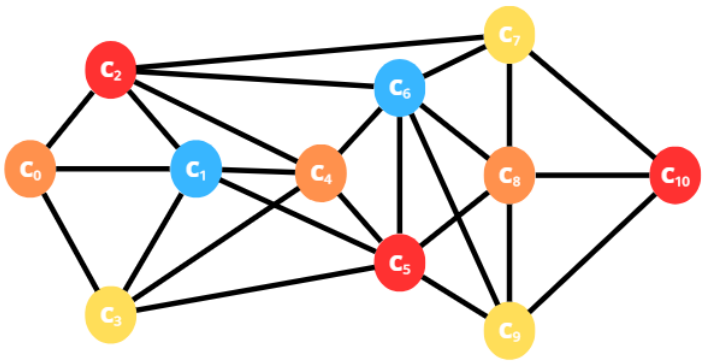


Figure 1.2: Final iteration of the visual representation of the DSATUR algorithm

1.4 Organization

This thesis is composed of the following six chapters:

- Chapter 2: Background
  Setting the stage by presenting fundamental concepts and principles underpinning physics-based simulation. Through a detailed exploration of core ideas, we present the necessary knowledge to comprehend the complexities of the methods discussed further in this thesis.

- Chapter 3: Related Work
  Delving into the prior research conducted in this domain, offers valuable insights into the foundations laid by others in the field. By studying the work of previous researchers, we gain a comprehensive understanding of the developments, challenges, and achievements that have shaped physics-based simulation.

- Chapter 4: Method
  Delving deeper into the intricacies, presenting a comprehensive explanation of a key method - Extended Position-based Dynamics (XPBD). By unraveling their inner workings, including the specific constraints that govern them, we gain a deeper understanding of the methodologies at play. Additionally, we delve into the context of graph coloring and the vital role of DSATUR within the scope of this thesis.

- Chapter 5: Implementation and Results
  Presenting the culmination of our efforts through the display of specific simulation scenarios and their corresponding results. The outcomes offer valuable insights into the effectiveness and performance of the proposed methods. The chapter also includes comparisons that shed light on their relative merits.

- Chapter 6: Conclusion and Future Work
  As we approach the culmination of this thesis, Chapter 6 offers a reflective analysis of the entire research journey. We discuss the significant findings and contributions, drawing together key insights from the preceding chapters. Moreover, we outline potential directions for future research and development, encouraging continuous advancement in the ever-evolving field of physics-based simulation

Chapter 2

Background

Simulating deformable objects is a vast and dynamic field that plays a pivotal role in various applications, including virtual environments, simulations, and computer graphics. The accurate representation and simulation of deformable objects are essential for achieving realism and creating immersive experiences.

In this chapter, we will begin by exploring the fundamental aspects of simulating deformable objects, focusing on two primary categories: non-physical models and physical models. Non-physical models offer flexible and intuitive approaches to simulate deformations, often sacrificing strict adherence to physical laws. Physical models, on the other hand, strive to accurately represent the underlying physics governing deformable objects. Non-physical models, encompass techniques such as splines [6, 49] and free-form deformation [5], while physical models, incorporate methods such as finite element methods [43] and position-based methods [29, 37, 40]. These models provide simplified representations of deformable objects, enabling efficient computations and real-time simulations. We will delve into the principles and concepts behind these models, examining their advantages and limitations.

Mass-spring systems consist of utilizing a network of interconnected springs, enabling efficient approximation of flexible object behaviors, while the finite element method is a versatile numerical technique used to solve complex problems involving diverse occurrences such as structural mechanics or fluid dynamics. While the mass-spring system emphasizes speed and real-time interaction, the finite element method prioritizes precision. Position-based dynamics (PBD), on the other hand, formulate the simulation as a constraint satisfaction problem. We will explore the underlying concepts, formulations and advancements associated with these physical models, highlighting their significance in handling the complex behavior of deformable objects.

Finally, we will explore the topic of parallelization in the context of Position-Based Dynamics (PBD) simulations. Parallelization plays a crucial role in optimizing the performance of deformable object simulations, allowing for faster computations and handling of larger and more complex scenarios. To achieve parallelization in PBD, one promising technique is graph coloring [31]. Graph coloring is a method that assigns colors to vertices of a mesh such that no adjacent

vertices share the same color. In the context of PBD, the graph/mesh represents the connection between particles, and the colors correspond to the processing units or threads used for parallel computation. Although there are already some resources with some different graph coloring algorithms [23], and applications [15, 16], we'll be talking about one, in particular: Vivace [27].

By examining the evolution of non-physical and physical models, we gain insights into the advancements made in the field of deformable object modeling, paving the way for further research and innovation.

Moreover, it is important to note that the field of simulation of deformable objects is extensive and continually evolving. While we have provided an overview of selected methods in this discussion, there are numerous other noteworthy approaches and advancements in the field, some of them such as Finite Differences Method [53], Finite Volume Method [52], Boundary Element Method [30] and Particle Based Approaches [18, 54]. For readers interested in exploring the topic further, we recommend consulting various state-of-the-art (STAR) papers written throughout the years, such as [9, 11, 28, 42].

2.1 Non-Physical Models

Non-physical models, in contrast to their physically-based counterparts, utilize purely geometric techniques. These models are designed to achieve specific visual effects or behaviors without explicitly simulating the physical principles governing the system. Instead, they rely on the expertise and creativity of the designer to craft the desired outcome.

By employing geometric techniques, these non-physical models offer computational efficiency and flexibility in representing complex shapes and behaviors. The designer can manipulate and control the model's geometry, allowing for artistic expression and customization [28].

2.1.1 Splines

Splines are mathematical curves that are widely used in computer graphics and geometric modeling to represent smooth and flexible shapes. They are defined by a set of control points that influence the shape and behavior of the curve. The curve itself is determined through interpolation or approximation techniques.

Let's consider an example of a Bézier curve, which is a type of spline commonly used in computer graphics. A Bézier curve is defined by a set of control points that lie within or on the curve. The positions of these control points determine the shape of the curve. By manipulating the control points, you can modify the curve to create various shapes and curves [22].

Figure 2.1 from [28] demonstrates a Bézier curve with control points:

In this image, you can see a Bézier curve represented by a smooth line. The curve is defined

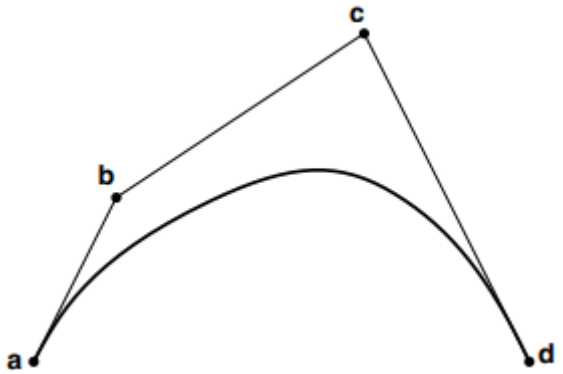


Figure 2.1: Bézier curve with 4 control points from [28]

by four control points: a, b, c and d. The shape of the curve is determined by the positions of these control points. By adjusting their positions, you can change the shape and curvature of the curve.

Note that this is just one example of a spline, specifically a Bézier curve. There are other types of splines, such as B-splines [44], NURBS [47], and T-splines [49], each with their own characteristics and mathematical properties. Nonetheless, the concept of control points influencing the shape of the curve remains consistent across different spline types [28, 42] .

Using splines offers several advantages [28] in various applications, including computer-aided design (CAD), computer animation, and graphical user interfaces (GUIs). Here are some of the advantages:

- Smoothness: Splines allow for the creation of smooth and visually appealing curves and surfaces. They can accurately represent complex shapes, resulting in aesthetically pleasing designs.
- Flexibility: Splines provide a high degree of flexibility in shaping curves. Designers can manipulate control points to customize and adjust the curve's shape, allowing for creative expression.

However, there are also some limitations [28] or considerations when using splines:

- Complexity: Some spline types involve complex mathematics and algorithms, requiring a good understanding of the underlying principles. This can pose challenges for beginners or those without a strong mathematical background.
- Control Point Placement: Achieving the desired curve or surface may require careful placement and adjustment of control points. Finding the optimal positions can be time-consuming and may involve trial and error.

2.1.2 Free-form deformation

Free-form deformation (FFD) provides a powerful technique for manipulating and deforming 3D objects and surfaces. FFD takes a different approach by introducing a lattice or grid structure that surrounds the object or surface of interest. This lattice consists of control points or vertices that can be manipulated to deform the entire shape uniformly or locally [28].

According to [28], the concept of free-form deformation was initially explored by Barr [5], who examined deformations in terms of geometric mappings of three-dimensional space. Sederberg and Parry [48] later generalized Barr's approach by embedding an object in a lattice of grid points, such as a cube or cylinder. Manipulating nodes of the grid induces deformations on the space inside the grid, transforming the underlying graphics primitives that form the object. One advantage of free-form deformation is its ability to provide global and local control simultaneously. The lattice structure allows for uniform transformations of the entire shape, while local adjustments can be made by manipulating specific control points, enabling intricate and detailed deformations.

However, it is important to note that free-form deformation may introduce some challenges and limitations [48]:

- It cannot perform general filleting and blending.
- Local FFD forms a planar boundary with the undeformed portion of the object. To create an arbitrary boundary curve, one would have to begin with an FFD which is already in a deformed orientation, and then deform it some more. this would be quite costly.

Despite these considerations, free-form deformation remains a valuable tool in computer graphics and geometric modeling. It offers a versatile approach for manipulating and deforming 3D objects and surfaces, complementing the precision and flexibility provided by splines.[48]

2.2 Core Concepts Behind Physical Models

Since the pioneering work of Terzopoulos [53] and other researchers in the 1980s, the field of deformable object simulation has witnessed significant advancements. These advancements have led to the development of various techniques for simulating solid objects, fluids, rigid bodies, etc. In these techniques, the principles of physics are incorporated to accurately represent the physical properties and responses of deformable objects when subjected to external forces.[9]

Before delving into the intricacies of these techniques, it is essential to introduce a few core concepts related to physics. Understanding these concepts will lay the foundation for comprehending the underlying principles that govern the behavior of deformable objects.

The most popular approaches for the simulation of dynamic systems in computer graphics are force based. Internal and external velocities are accumulated from which accelerations are

computed based on Newton's second law. A numerical time integration method is then used to update the velocities and finally, the positions of the object [8].

2.2.1 Particle

In a particle-based simulation, each particle i is characterized by its mass m_i, position x_i, and velocity v_i. The simulation process involves examining the state of all particles at a specific time t and utilizing the interactions between particles and external forces to determine the updated state of each particle at a new time $t + \Delta t$.

During each simulation step, the time is advanced by a certain increment Δt, and this incremental approach allows us to view time as a collection of small time steps. By progressing through these time steps, the simulation captures the dynamic behavior of the system, considering the influences of forces and particle interactions. At each time step, the simulation evaluates the forces acting on each particle, which can arise from various sources such as internal interactions between particles or external forces like gravity or user-defined forces. These forces impact the motion and behavior of the particles, resulting in changes to their positions and velocities.

By calculating the effects of these forces and incorporating them into the particle dynamics, the simulation predicts the new positions and velocities of the particles for the next time step. This iterative process continues as the simulation progresses through multiple time steps, accurately tracking the evolution of the system over time. It is important to note that the simulation time step Δt determines the granularity at which the simulation captures the dynamics of the particles. Smaller time steps allow for more precise and accurate simulations but may come at the cost of increased computational requirements [41].

2.2.2 Forces

In a physics-based simulation of deformable objects, the computation of forces acting on each particle is a fundamental aspect. These forces encompass both internal and external components, working together to govern the behaviour of the simulated object.

According to [41], internal forces arise from the interactions between particles within the mesh, playing a crucial role in replicating the behaviour of real-world materials like cloth or flesh, etc. By adjusting the strength of these internal forces, it becomes possible to simulate different types of cloth materials and volumetric models, such as linen, cotton, or polyester. This flexibility allows for the creation of diverse and realistic simulations that accurately capture the unique characteristics of various fabrics. On the other hand, external forces originate from the surrounding environment. Common examples include gravity and wind. While not obligatory, incorporating external forces into the simulation adds an extra layer of realism, as it mimics the influence of real-world factors on the deformable object. These external forces, when combined with internal forces, contribute to shaping the overall behaviour and motion of the simulated

object.

By applying forces to the particles, changes in their velocities occur, which then propagate to alterations in their positions. This propagation of changes in velocity and position forms the basic principle behind a physics-based simulation. It allows for the dynamic updating of the state of each particle over time, reflecting the ongoing interactions and influences of the applied forces.

2.2.3 Simulation Step

In order to translate forces into changes in particle positions within a specific time frame, a process known as simulation is employed.

To numerically solve the equations governing the behavior of the system, a common and straightforward approach is to approximate the derivatives using finite differences. This involves discretizing time into smaller intervals and approximating the rates of change of particle positions and velocities within each interval.

By discretizing time, the simulation progresses in small steps, where the state of the system is updated at each time step. The time integration process calculates the new positions and velocities of the particles based on the forces acting upon them. Finite differences provide a practical means of approximating the derivatives of position and velocity with respect to time. By estimating these derivatives at each time step, the simulation determines how the forces contribute to changes in particle positions over time.

According to [41], this numerical approach allows the simulation to iteratively compute the positions of the particles at successive time steps, providing a means to observe and analyze the system's dynamic behavior over a given time interval. It is important to note that various numerical methods can be employed for time integration, we'll go over explicit Euler integration, Verlet integration, and Runge-Kutta integration. These methods differ in their accuracy, stability, and computational efficiency, and the choice of method depends on the specific requirements and characteristics of the simulation.

By utilizing time integration techniques and approximating derivatives with finite differences, the simulation accurately captures the effects of forces on the particles and enables the tracking of their positions as the system evolves over time.

Newton's second law of motion is the key to getting from the definition of forces to a simulation algorithm

$$F = ma \tag{2.1}$$

where F is the vector sum of all forces and a is the acceleration of a particle. And as such, we can write the velocity and acceleration as

$$v(t) = \dot{x}(t) \tag{2.2}$$

$$a(t) = \dot{v}(t) \tag{2.3}$$

2.2.3.1 Forward Euler Integration

As we already discussed, time will be dealt with as a discrete dimension. To discretize these equations, we can use the forward finite differences [33] approximation to write them as

$$v(t) \approx \frac{x(t+\Delta t) - x(t)}{\Delta t} \tag{2.4}$$

$$a(t) \approx \frac{v(t+\Delta t) - v(t)}{\Delta t} \tag{2.5}$$

Where Δt is a small time step. This approximation only uses the first degree derivative and subsequently assumes that the velocity and forces/acceleration are constant through the entire step. These equations can be rewritten in terms of $x(t+\Delta t)$ and $v(t+\Delta t)$ as

$$x(t+\Delta t) \approx x(t) + \Delta t * v(t) \tag{2.6}$$

$$v(t+\Delta t) \approx v(t) + \Delta t * a(t) \tag{2.7}$$

or using discrete notation for the time

$$x^{t+1} \approx x^t + \Delta t * v^t \tag{2.8}$$

$$v^{t+1} \approx v^t + \Delta t * a^t \tag{2.9}$$

Where t represents the current time and $t+1$ represents the next time step.

We can then combine equation (2.1) and equation (2.9) to find the equation

$$v^{t+1} \approx v^t + \Delta t * \frac{F^t}{m} \tag{2.10}$$

That will be used to update the state of each particle for that time.

According to [42], this method employed for updating the position and velocity in simulations is commonly known as Euler/Explicit Integration or Forward Euler Integration. It assumes that the velocities and forces remain constant between each time step, which is a simplifying approximation made for the finite differences computation. However, this simplification introduces a source of instability, as it fails to capture the true variations in velocities and forces. Consequently, the Forward Euler method tends to repeatedly overshoot the correct values. To mitigate this issue, smaller time steps are typically used, aiming to minimize the impact of velocity and force variations. While this approach helps enhance stability, it comes at the cost of increased computational requirements.

To address the limitations of the Forward Euler Integration, alternative methods and more complex integration techniques have been explored by researchers like Desbrun et al [18].

2.2.3.2 Verlet Integration

In the realm of solving the laws of motion in physics simulation, there are alternatives to Euler Integration that provide more stable results. One commonly used alternative is Verlet Integration.

According to [41], Verlet Integration takes a different approach by using only the current and previous positions of particles to estimate their velocities. This allows for more stable simulations compared to Euler Integration. By relying on position information instead of explicit velocities, it improves the accuracy and reliability of simulation results.

In order to improve stability, we use a more accurate approximation of $x(t + \Delta t)$, by utilizing the second degree derivative of the central finite differences approximation

$$a(t) \approx \frac{\frac{x(t+\Delta t)-x(t)}{\Delta t} - \frac{x(t)-x(t-\Delta t)}{\Delta t}}{\Delta t} = \frac{x(t + \Delta t) - 2x(t) + x(t - \Delta t)}{\Delta t^2} \tag{2.11}$$

which can be rewritten in terms of $x(t + \Delta t)$ as:

$$x(t + \Delta t) \approx 2x(t) - x(t - \Delta t) + (\Delta t)^2 a(t) \tag{2.12}$$

Using Equation (2.1) for the acceleration, and writing the formula using discrete notation we obtain:

$$x^{t+1} \approx 2x^t - x^{t-1} + (\Delta t)^2 * \frac{F}{m} \tag{2.13}$$

which is the basis of this alternative integration method.

2.2.3.3 Runge-Kutta integration

The Runge-Kutta method is another integration technique, it is known for its accuracy and versatility compared to simpler integration methods like Euler and Verlet. In Runge-Kutta integration, the particle's position and velocity are updated based on a more sophisticated calculation that takes into account multiple intermediate steps. The most commonly used variant is the fourth-order Runge-Kutta (RK4) method.

RK4 calculates the position and velocity of a particle over a given time step by considering the slopes at different points within that interval. It involves evaluating the particle's acceleration at various intermediate points and then using weighted averages to obtain the final position and velocity. As seen more detailed in [41].

Compared to Euler and Verlet integration, the Runge-Kutta method offers higher accuracy and stability. However, it's worth noting that the increased accuracy of Runge-Kutta comes at the cost of additional computational complexity. The method requires evaluating the particle's

acceleration multiple times per time step, which can be more computationally demanding than Euler or Verlet integration. Therefore, the choice of integration method should consider the trade-off between accuracy and computational performance based on the specific requirements of the simulation.

2.2.4 Solvers

In the context of physics simulations, objects or particles often have certain constraints imposed on them. These constraints can include geometric constraints, such as maintaining a specific distance or angle between objects, or physical constraints, such as preserving volume. The goal is to ensure that the simulated objects adhere to these constraints throughout the simulation.

In these scenarios, the time integration step provides an initial prediction of the positions and velocities of the objects or particles at the next time step. However, this prediction may not satisfy the imposed constraints. This is where solvers come into play. They help refine the initial prediction by iteratively solving the system of equations that enforce the constraints. In this section, we'll only be referring to the Jacobi and Gauss-Seidel solvers because those are the most important in the context of this thesis, but be aware that more alternatives exist and some good references for more in-depth explanations about other methods are presented in [46, 57].

2.2.4.1 Jacobi and Gauss-Seidel

The Jacobi method and the Gauss-Seidel method are two closely related iterative techniques used to solve systems of linear equations. Both methods belong to the class of stationary iterative methods [57] (for solving a linear system of equations), where the solution vector is updated component-wise in an iterative manner. These methods provide approximate solutions by repeatedly improving the solution estimate until a desired level of accuracy is achieved.

According to [57], the Jacobi method and the Gauss-Seidel method share a similar iterative structure but differ in the way they update the solution vector. In the Jacobi method, each component of the solution vector is updated based on the previous iteration's values, while in the Gauss-Seidel method, the updated values of the solution vector are used immediately within the same iteration.

Now, let's delve into the Jacobi method, we first need to do two assumptions [57]:

1. The system given by $ax = b$ where a is the coefficient matrix, b is the right-hand side vector, and x is the vector of unknowns

$$a_{11}x_1 + a_{12}x_2 + ... + a_{1n}x_n = b_1$$
$$a_{21}x_1 + a_{22}x_2 + ... + a_{2n}x_n = b_2$$
$$\vdots$$
$$a_{n1}x_1 + a_{n2}x_2 + ... + a_{nn}x_n = b_n$$

Has a unique solution.

2. The coefficient matrix A has no zeros on its main diagonal, namely, $a_{11}, a_{22}, ..., a_{nn}$ are nonzeros.

To begin, we need to solve the first equation for x_1, the second equation for x_2 and so on to obtain the rewritten equations:

$$x_1 = \frac{1}{a_{11}}(b_1 - a_{12}x_2 - a_{13}x_3 - ... - a_{1n}x_n)$$
$$x_2 = \frac{1}{a_{22}}(b_2 - a_{21}x_2 - a_{23}x_3 - ... - a_{2n}x_n)$$
$$\vdots$$
$$x_n = \frac{1}{a_{nn}}(b_n - a_{n1}x_1 - a_{n2}x_2 - ... - a_{n,n-1}x_{n-1})$$

Then we make an initial guess of the solution $x^{(0)} = (x_1^{(0)}, x_2^{(0)}, ..., x_n^{(0)})$. Substituting these values in the right-hand side of the rewritten equations makes us obtain the first approximation, $(x_1^{(1)}, x_2^{(1)}, ..., x_n^{(1)})$. With this, we get one iteration.

In the same way, the second approximation $(x_1^{(2)}, x_2^{(2)}, ..., x_n^{(2)})$ is computed by substituting the first approximation's x-values into the right-hand side of the rewritten equations.

By repeated iterations, we form a sequence of approximations $x^{(k)} = (x_1^{(k)}, x_2^{(k)}, ..., x_n^{(k)})$.

So this introduces the Jacobi method itself which consists in, for each $k \geq 1$ and for $i = 1, 2, ..., n$, generating the components $x_i^{(k)}$ of $x^{(k)}$ from $x^{(k-1)}$ by

$$x_i^{(k)} = \frac{1}{a_{ii}}[\sum_{j=1}^{n}(-a_{ij}x_j^{(k-1)}) + b_i]$$

So, for this method, the values of $x_i^{(k)}$ obtained in the kth iteration remain unchanged until the entire (k+1)th iteration has been calculated. With the Gauss-Seidel method, we use the new values $x_i^{(k+1)}$ as soon as they are known. For example, once we have computed $x_1^{(k+1)}$ from the first equation, its value is then used in the second equation to obtain the new $x_2^{(k+1)}$, and so on.

So the Gauss-Seidel method consists in, for each $k \geq 1$ and for $i = 1, 2, ..., n$, generating the components $x_i^{(k)}$ of $x^{(k)}$ from $x^{(k-1)}$ by

$$x_i^{(k)} = \frac{1}{a_{ii}}[-\sum_{j=1}^{i-1}(a_{ij}x_j^{(k)}) - \sum_{j=i+1}^{n}(a_{ij}x_j^{(k-1)}) + b_i]$$

2.3 Physical Models

Having explored the core concepts behind physical models in the previous section, we now venture into the realm of specific physical modeling techniques. In this section, we delve into the world of physical models and their applications in computer graphics and simulation.

Physical models provide a means to simulate the behavior and interactions of objects in a virtual environment by incorporating the principles of physics. They enable the creation of realistic and dynamic simulations, bringing virtual worlds to life with accurate representations of materials, forces, and motions [41].

In this section, we will explore three prominent classes of physical models: mass-spring systems, finite element methods, and position-based methods. Each approach offers distinct advantages and is suitable for different types of simulations and scenarios.

2.3.1 Mass Spring Systems

One of the fundamental approaches used to simulate deformable objects is the mass-spring system. It serves as a simple yet effective framework for utilizing different simulation techniques and time integration methods. The concept behind a mass-spring system involves representing objects as interconnected point masses connected by springs [28]. This straightforward representation provides a compact and concise implementation of a simulator.

According to [41], however, it is important to acknowledge some limitations associated with the mass-spring system approach. The behavior of the simulated object heavily relies on the configuration of the spring network. Finding the optimal spring constants to achieve the desired behavior can sometimes be a challenging task. Additionally, mass-spring systems do not inherently capture volumetric effects such as volume conservation. Despite these limitations, mass-spring systems are often an excellent choice for various applications. Their simplicity and computational efficiency make them highly desirable, especially when accurate physics simulations are not the primary requirement.

However, when more accurate and sophisticated physics modeling is necessary, alternative methods like the Finite Element Methods, as detailed in the next section, should be considered.

A mass-spring system is composed of a set of N particles, each one with masses m_i, positions

x_i and velocities v_i, with $i \in 1...N$ and a set of springs. This system has been used widely and effectively for modeling deformable objects. An object is modeled as a collection of point masses connected by springs in a lattice structure (Figure 2.2 from [28]).

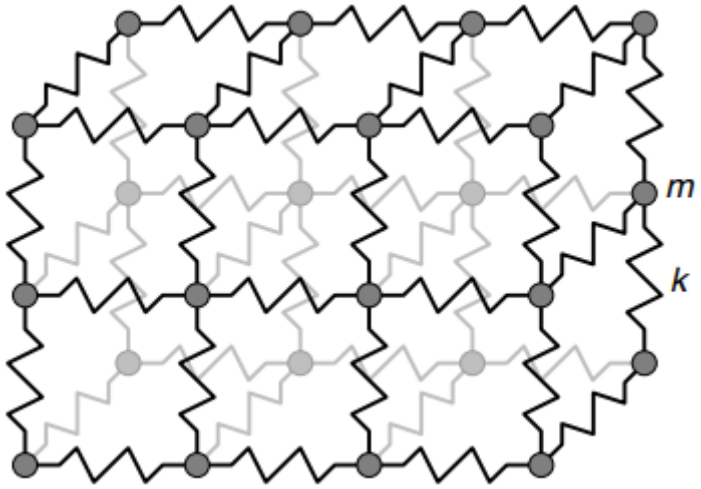


Figure 2.2: Portion of a mass-spring model from [28]

The spring forces are often linear, but nonlinear springs can be used to model tissues such as human skin that exhibit inelastic behavior [28]. In a dynamic system, Newton's second law governs the motion of a single mass point in the lattice (in Figure 2.2):

$$m_i\ddot{x}_i = -\gamma_i\dot{x}_i + \sum_j g_{ij} + f_i \tag{2.14}$$

The terms on the left-hand side are related to the particle itself, while the terms on the right-hand side are forces acting on the mass point. The first right-hand term is a velocity-dependent damping force, g_{ij} is the force exerted on mass i by the spring between masses i and j, and f_i is the sum of other external forces, acting on mass i.

The equations of motion for the entire system are assembled from the motions of all the mass points in the lattice [28]. Concatenating the position vectors of the N individual masses into a single $3N$-dimensional position vector x, one obtains:

$$M\ddot{x} + C\dot{x} + Kx = f \tag{2.15}$$

According to [28], where M, C and K are the $3N \times 3N$ mass, damping, and stiffness matrices, respectively. Although large, these matrices are typically quite sparse. M and C are diagonal matrices and K is banded because it encodes spring forces which are functions of distances between neighboring mass points only. The vector f is a $3N$-dimensional vector representing the total external forces on the mass points.

The system is evolved forward through time by re-expressing (Equation 2.15) as a system of

first-order differential equations:

$$\dot{v} = M^{-1}(-Cv - Kx + f) \tag{2.16}$$

$$\dot{x} = v \tag{2.17}$$

Where v is the velocity vector of the system of mass points. Now, we just need to use one of the time integration methods we've already discussed in order to compute x and v as functions of time.

2.3.2 Finite Element Methods

Mass spring systems cannot capture volumetric effects. Their behaviour depends on the tesselation of the mesh [9, 41].

Finite element methods are one of the most widely used techniques in computational sciences for the simulation of solid objects [41]. The method reduces general partial differential equations to systems of algebraic equations [3]. While this provides more physically realistic results, the amount of computation required at each time step is greatly increased [28].

2.3.2.1 Continuum mechanics

According to [41], continuum mechanics provides the theoretical foundation for understanding the mechanical behavior of materials as continuous substances. It allows us to describe and analyze how materials respond to external forces and deformations by considering them as continuous media with distributed properties. The concepts of displacement and Green's strain tensor play a crucial role in continuum mechanics and form the basis for the application of finite element methods.

In the context of finite element methods, we utilize continuum mechanics to simulate and analyze the behavior of complex structures and systems. By discretizing the continuous domain into smaller elements, we can approximate the behavior of the material at each element and then assemble these local solutions to obtain a global understanding of the system [41].

According to [41], displacement, which represents the change in position of material particles within a body, is a key parameter in finite element analysis. By approximating the displacement field over each element, we can determine how the material deforms or moves under the influence of applied forces or loads. This allows us to understand the structural response and predict the displacements at different points within the material. Strain, which measures the relative change in the size or shape of an object, is another important quantity in finite element analysis. By calculating the strain distribution over the elements, we can assess the level of deformation experienced by the material. This information helps us understand the structural integrity and potential failure modes of the system under different loading conditions. Stress, closely

related to strain, characterizes the internal forces within the material. By computing the stress distribution throughout the structure, we can evaluate the material's resistance to deformation and identify regions of high-stress concentration. This enables us to analyze the structural safety and determine if the applied loads exceed the material's strength limits.

Finite element methods rely on constitutive equations that describe the relationship between stress and shear for a given material. These equations capture the material's mechanical properties and provide the necessary information to relate the applied forces to the resulting deformation. We're not going into the specific equations behind this process because it's not as covered in the context of this thesis, but for a more curious reader, [41] is a very good resource to understand the math behind the whole process.

2.3.3 Position-based Methods

According to [9], in classical dynamics simulation methods, the positions of objects are typically updated by numerically integrating their velocities, which are influenced by the applied forces. However, position-based approaches take a different approach. Instead of explicitly calculating velocities, these methods directly compute the positions of the objects based on the solution to a specific problem.

Rather than explicitly calculating forces and updating velocities, position-based methods focus on finding a set of positions that satisfy a set of desired constraints. These constraints can include maintaining certain distances between particles, preserving volume, or enforcing collision avoidance [9].

In the context of this thesis we'll only be discussing PBD (Position-based Dynamics) [40] and XPBD (Extended Position-based Dynamics) [37].

2.3.3.1 Position-based Dynamics

In PBD, the simulation starts with an initial configuration of positions for the particles or vertices of the object being simulated. These positions are then updated at each time step based on the satisfaction of various constraints.

Figure 2.3 from [40] demonstrates some deformation results using PBD

In the simulation process of Position-Based Dynamics (PBD), the positions of the next simulation iteration of the particles or vertices of the simulated object are typically initially predicted using a time integration scheme, such as the ones talked about in a previous section. This initial prediction provides a starting point for the simulation [40].

According to [40], once the initial positions are predicted, the simulation enters the constraint-solving phase. In this phase, the position-based constraints are iteratively solved to ensure that they are satisfied by the updated positions of the particles. The constraint-solving process involves

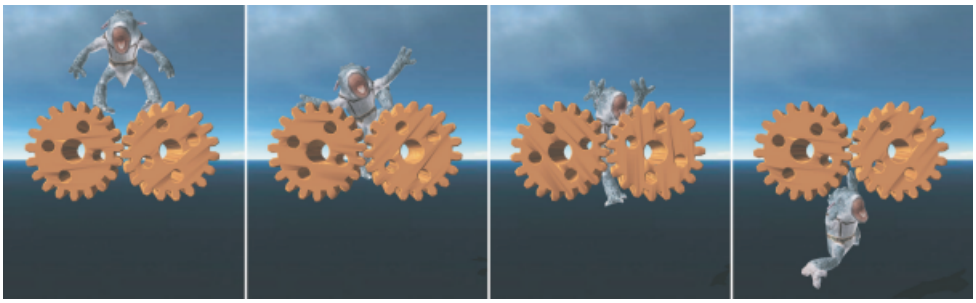


Figure 2.3: An example deformation benchmark test from [40]

iteratively adjusting the positions of the particles to enforce the constraints. The resulting system of equations in this process is non-linear. To solve such a general set of equations and inequalities, we can't apply directly a Gauss-Seidel or Jacobi solvers, since they are linear solvers. So what's normally used is a solver based on Gauss-Seidel or Jacobi which solves non-linear equations.

According to [10], this solver's objective is to determine a set of positions that fulfill all the constraints without breaching any of them. This is typically achieved through an iterative process where the positions are adjusted based on the gradients of the constraint functions. The solver applies small corrections to the positions in each iteration until the constraints are sufficiently satisfied. By iteratively solving the constraints, PBD achieves a dynamic equilibrium where the simulated object behaves realistically while respecting the specified constraints.

The main features and advantages of PBD are [40]:

- Position-based simulation gives control over implicit integration and removes the typical instability problems.
- Positions of vertices and parts of objects can directly be manipulated during the simulation.
- The formulation we propose allows the handling of general constraints in the position-based setting.
- The explicit position-based solver is easy to understand and implement

According to [11], however, one of the observed limitations in PBD is the dependence of constraint stiffness on the chosen time-step size and iteration count in the constraint solver. It has been noted that as the number of constraint iterations approaches infinity, the constraints tend to become infinitely stiff.

To address the issue of stiffness dependence on time-step size and iteration count in PBD, an extension called Extended Position-Based Dynamics (XPBD) [37] was introduced. XPBD builds upon a compliant constraint formulation [50], which associates an inverse stiffness parameter with each constraint.

The specific details about the algorithm and solvers will be touched upon in the following chapters.

2.4 Parallelization

One of the most attractive features of position-based methods is that they can be massively parallelized, so in this section, we delve into the realm of parallelization within the context of Position-Based Dynamics (PBD) and Extended Position-Based Dynamics (XPBD). We explore the potential benefits and techniques of parallel computing to accelerate PBD/XPBD simulations and enable the efficient simulation of complex physical phenomena.

According to [10], following each constant solver iteration, the positions of particles impacted by the constraints are promptly adjusted. In a parallelized implementation, multiple threads concurrently handle the constraints. However, when two constraints that affect the same particle are processed simultaneously by different threads, immediate updates to the particle's position are not permitted. This restriction arises from the potential for race conditions when multiple threads attempt to write to the same position simultaneously, which can introduce unpredictability into the process. To mitigate these challenges, a parallelized implementation of simulation must partition the constraints into distinct groups or phases. Within each phase, no constraints are permitted to involve the same particle. By enforcing this restriction, the constraints within the initial phase can be concurrently processed without conflicts. Following a global synchronization point, the subsequent phase can then be processed. This cyclic process continues until all constraints have been processed, ensuring efficient parallel execution of the simulation.

2.4.1 Graph coloring

The concept of particles and constraints in a physics-based simulation can be seen as a graph, providing an opportunity to leverage graph theory methods for optimized computations. One such method is graph coloring, where distinct colors are assigned to graph elements to ensure interconnected elements do not share the same color. By applying graph coloring to the particle system, particles can be grouped into sets that can be independently solved, enabling efficient parallel processing and optimization of the simulation [46].

One typical example of graph coloring is the greedy algorithm taken from [23]:

Algorithm 1: Greedy Graph coloring Algorithm

```
1 let v_1, v_2, ..., v_n be an ordering of V
2 for i = 1 to n do
3 |   determine forbidden colors to v_i
4 |   assign v_i the smallest permissible color
5 end
```

Figure 2.4 demonstrates a visualization for this algorithm

The main problem of this algorithm is that even though it runs on $O(n)$, it's very dependent on the ordering that it follows and as such, in general, an arbitrary ordering may perform very

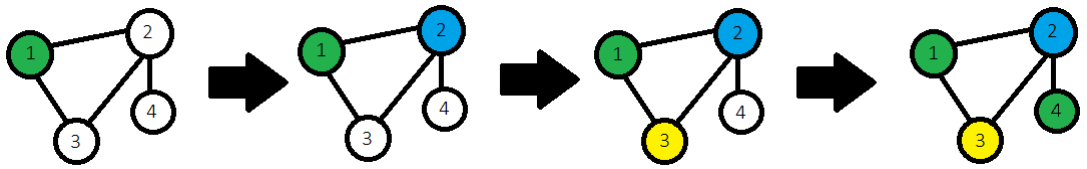


Figure 2.4: Greedy coloring algorithm visualized

poorly [10].

In the pursuit of overcoming these challenges, one prominent technique is the Vivace algorithm, introduced by [27]. Vivace employs a parallel graph coloring technique that allows for a high maximum degree of the constraint graph. It leverages randomization and heuristics to mitigate conflicts between colors, resulting in efficient parallelization. Vivace has demonstrated impressive results, enabling the handling of hundreds of thousands of constraints while maintaining interactivity and preserving visual fidelity.

Another noteworthy advancement is the work by [25], which addresses the concurrency challenges of parallel Gauss-Seidel solvers in PBD. They propose a graph coloring-based approach that partitions constraints into phases, ensuring that no constraints share a common particle within each phase. This enables conflict-free parallel processing of constraints, leading to improved efficiency in the simulation.

In later chapters, we'll also be taking a look into an algorithm created in the context of this thesis work, very inspired by DSATUR, introduced in [12].

Chapter 3

Related Work

The most widely used methods in computer graphics for physics-based visual simulation of deformable objects are force-based methods. Force-based approaches, such as the finite element method [51] and mass-spring systems [34], rely on computing the forces acting on a system and then integrating those forces to update the positions and velocities of the objects being simulated. While these methods can produce highly accurate simulations, they can be computationally expensive, especially for large or complex systems [41].

Position-Based Dynamics (PBD) [40], has found versatile applications across a wide spectrum of fields. These applications range from simulating knots [32], to animating facial expressions [24], and automating character skinning [45]. Initially devised for flexible structures like fabrics and inflatable objects, PBD's scope expanded to include various domains.

The application of PBD was not confined to soft bodies alone, with subsequent research delving into rigid bodies [19], and fluids [35]. To capture realistic animation effects, strain tensor constraints [38]. These constraints proved valuable in simulating animation effects for both cloth materials and volumetric models.

While Position-Based Dynamics (PBD) has proven to be notably efficient and capable of managing extensive deformations and intricate geometries, it is important to acknowledge that its accuracy might not universally hold up in all scenarios [11]. This realization paved the way for the introduction of Extended Position-Based Dynamics (XPBD) [37]. XPBD acts as a straightforward expansion of PBD, addressing one of its prominent limitations - the dependency of stiffness on time steps and iteration counts [37].

This method stems from a compliant constraint formulation, where each constraint is associated with an inverse stiffness, often termed compliance, denoted as α. This compliance property enables PBD to emulate diverse elastic and dissipative energy potentials. Furthermore, XPBD can offer accurate estimations of constraint forces, particularly for effects reliant on forces [11].

Constraints are an important part of any deformable object simulation. They provide

the assurance that the simulation is physically accurate and visually realistic. There are many different types of constraints that can be used in deformable object simulations, such as stretching, shear and volume conservation [9].

With the increasing computational power of modern GPUs and CPUs, PBD can be efficiently implemented with parallel computation to achieve high performance [11].

To improve performance, many approaches have been explored and implemented using parallel processing to partition the process of numerical solving. A highly parallelized version of PBD that utilized a single GPU device [20].

In [39], a hierarchical position-based approach for clothes is devised in order to accelerate the convergence of the solver. In [7], a red-black parallel Gauss-Seidel schema is used for animating inextensible clothes using a force-based system. While providing excellent performance, this method is restricted to meshes with a regular grid topology. The approach detailed in [7] involves the subdivision of the mesh into constraint strips. Notably, strips lacking shared particles are autonomous of each other, thus facilitating parallelized solutions.

The Gauss-Seidel algorithm stands as an efficient iterative technique employed for the resolution of linear equation systems, including the linearized positional constraints seen in PBD. Its convergence outpaces that of alternative solvers, such as Jacobi [23]. However, the fundamental algorithm itself is inherently sequential, with equations being sequentially resolved in an iterative manner. Following each iteration, the disparity between the current solution and the optimal one diminishes.

The utilization of pre-conditioned conjugate gradient (PCG) [56], and the application of GMRES [4], have served the purpose of leveraging the sparsity inherent in linear systems. This strategy has been instrumental in extracting parallelism and enhancing the GPU implementation of well-recognized solvers.

The unified framework presented by [36] utilizes Position-Based Dynamics (PBD) as a fundamental element for the real-time simulation of gases, liquids, deformable solids, rigid bodies, and cloth with two-way interactions. This includes the modeling of their interactions and collisions. The procedure involves the transformation of initial mesh inputs into particles, followed by the application of a parallel Jacobi solver along with an under-relaxation technique. In contrast, [23] applies a parallel Gauss-Seidel solver in a manner similar to the original PBD approach.

Iterative solvers have found significant application in contact resolution [13]. [55] introduces a parallel iterative solver tailored for rigid bodies, designed to prevent jitter artifacts even at low iteration counts. The interconnected challenge stemming from rigid body systems is approached iteratively by grouping contacts into blocks. This involves utilizing parallel Gauss-Seidel to address the contacts within each block and employing Jacobi to integrate these blocks cohesively.

If we see the concept of particles and constraints in a physics-based simulation as a graph,

we have the opportunity to leverage graph theory methods for optimized computations, such as graph coloring to group the particles into sets that can be independently solved [46].

[25] introduced an extensively parallelized rendition of Position-Based Dynamics (PBD) aimed at achieving interactive animations of deformable bodies. This implementation partitions the array of constraints into distinct independent segments using a graph-based algorithm. This enables parallel solving of each partition on a GPU, resulting in a substantial performance enhancement when compared to its sequential counterpart. A simplified form of this algorithm was also introduced by [15, 16]. Another notable innovation is the Vivace algorithm [26]. This approach leverages randomization and heuristics to alleviate conflicts between colors.

Within the context of performance enhancement, a method dedicated to maintaining internal shape consistency is presented by [21]. This algorithm is engineered to uphold the shape of a 3D deformable entity and concurrently diminish the number of internal structures upon its incorporation into PBD. Moreover, [45] expanded the utilization of PBD to encompass the simulation of soft body character models. This was achieved by implementing tetrahedral volume constraints on a volumetric mesh, followed by the reconstruction of surface attributes to facilitate rendering requirements.

Chapter 4

Method

4.1 Position-Based Dynamics

The iterative solver employed in PBD plays a vital role in achieving stability. It operates by applying corrective impulses or displacements to particles, nudging them toward their desired positions as dictated by the constraints. Through an iterative process, the solver progressively converges to a solution that satisfies the constraints, resulting in stable and controllable object motion [40].

4.1.1 Contraint Projection

Projecting a set of points according to a constraint means moving the points such that they satisfy the constraint. The Gauss-Seidel iteration process is the most significant step of this algorithm for computing the final position of each particle [16]. We let p be the concatenation $\{i_1, ...i_{n_j}\}$ of the constraint. Given p we want to find a correction Δp such that $C(p + \Delta p) = 0$. This equation can be approximated using Taylor series by:

$$C(p + \Delta p) \approx C(p) + \nabla_p C(p).\Delta p = 0 \tag{4.1}$$

Given this, we also know that in order to maintain both momenta, the direction of Δp is restricted to be along $\nabla_p C(p)$ [40]. And as such:

$$\Delta p_i = w_i \nabla_{p_i} C(p) \lambda_i \tag{4.2}$$

where w_i is the inverse mass of each particle and λ_i is the Lagrange multiplier that can be obtained by using equations (4.1)-(4.2):

$$\lambda_i = -\frac{C(p)}{\sum_i w_i |\nabla_{p_i} C(p)|^2} \tag{4.3}$$

And finally, we can get the value of Δp_i by using equations (4.2)-(4.3):

$$\Delta p_i = -\frac{w_i C(p) \nabla_{p_i} C(p)}{\sum_i w_i |\nabla_{p_i} C(p)|^2} \tag{4.4}$$

4.1.2 Constraints

In the context of this thesis, we implemented three constraints: distance, volume and shear.

4.1.2.1 Distance constraint

Let's get into the distance constraint function $C_{distance}(p_1, p_2) = |p_1 - p_2| - d$ where d is the initial distance between the two particles and with this, we want to make sure the difference between the distance of two particles is approximated to the distance to which they started.

Figure 4.1 inspired from [15] demonstrates a visualization for this constraint

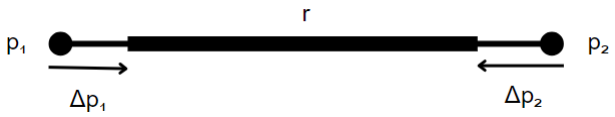


Figure 4.1: Projection of the distance constraint inspired from [15].The Δp_i are the corrections.

The derivatives with respect to the points are:

$$\nabla_{p_1} C_{distance}(p_1, p_2) = \frac{p_1 - p_2}{|p_1 - p_2|} \tag{4.5}$$

$$\nabla_{p_2} C_{distance}(p_1, p_2) = -\frac{p_1 - p_2}{|p_1 - p_2|} \tag{4.6}$$

4.1.2.2 Volume Constraint

Typically, applying the volume constraint is a common practice for tetrahedral meshes. However, in our approach, we have devised a practical solution to extend the volume constraint to triangle meshes. To achieve this, we calculate the model's center of mass during each iteration and utilize it as the apex of each triangle primitive. This straightforward technique enables us to compute the volume of the triangle mesh as if it were a tetrahedral mesh, all while reducing computational overhead [17].

According to [10], the volume constraint is mathematically expressed as $C_{volume} = \frac{1}{6}(((p_1 - p_0) \times (p_2 - p_0)) \cdot (p_3 - p_0)) - V_0$, where p_0, p_1, p_2, p_3 are the four corners of the tetrahedron, p_0

being the center of mass, and V_0 is the rest volume $\frac{1}{6}(((P_1 - P_0) \times (P_2 - P_0)) \cdot (P_3 - P_0))$, where P_0, P_1, P_2, P_3 are the four corners of the initial tetrahedron, P_0 being the initial center of mass. This formulation allows us to effectively control the volume preservation in the context of triangle meshes while maintaining computational efficiency.

Figure 4.2 inspired from [45] demonstrates a visualization for this constraint

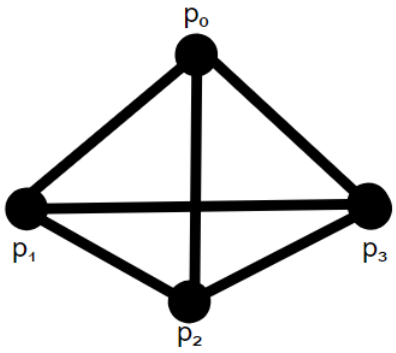


Figure 4.2: Visualization of volume constraint inspired from [45].

The corresponding gradients are obtained as:

$$\nabla_{p_1} C_{volume} = (p_2 - p_0) \times (p_3 - p_0)\frac{1}{6} \tag{4.7}$$

$$\nabla_{p_2} C_{volume} = (p_3 - p_0) \times (p_1 - p_0)\frac{1}{6} \tag{4.8}$$

$$\nabla_{p_3} C_{volume} = (p_1 - p_0) \times (p_2 - p_0)\frac{1}{6} \tag{4.9}$$

and in order to conserve linear momentum we know that [40]:

$$\nabla_{p_0} C_{volume} = -\nabla_{p_1} C_{volume} - \nabla_{p_2} C_{volume} - \nabla_{p_3} C_{volume} \tag{4.10}$$

Since p_0 is an implicit point we attribute no weight to it, and as such we apply all the position updates for the volume conservation except to p_0.

4.1.2.3 Shear Constraint

The shear constraint is a complementary component in simulations, particularly for achieving realistic cloth wrinkling effects. It plays a crucial role in maintaining the correct angles between particles, contributing to the formation of wrinkles in cloth simulations. In contrast to the bending constraint, which requires additional context, the shear constraint specifically keeps the deformation information within a 2D plane, particularly in the context of triangles [14].

According to [14], the shear constraint is mathematically expressed as $C_{shear} = cos^{-1}(p_{12} \cdot p_{13}) - r$, where p_{12} represents the unit vector $p_1\hat{p}_2 = \frac{p_1\vec{p}_2}{|p_1\vec{p}_2|}$ and p_{13} represents the unit vector $p_1\hat{p}_3 = \frac{p_1\vec{p}_3}{|p_1\vec{p}_3|}$ with $p_1, p_2, p_3 \in \mathbb{R}^3$ being the triangle vertices. While r is $cos^{-1}(P_{12} \cdot P_{13})$ where P_{12} represents the unit vector $P_1\hat{P}_2 = \frac{P_1\vec{P}_2}{|P_1\vec{P}_2|}$ and P_{13} represents the unit vector $P_1\hat{P}_3 = \frac{P_1\vec{P}_3}{|P_1\vec{P}_3|}$ with $P_1, P_2, P_3 \in \mathbb{R}^3$ being the initial points of the triangle.

Figure 4.3 inspired from [14] demonstrates a visualization for this constraint

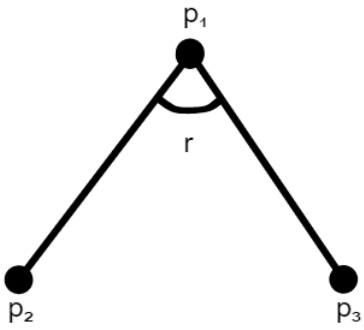


Figure 4.3: Visualization of shear constraint inspired from [14].

The corresponding gradients are obtained as [14]:

$$\nabla_{p_2} C_{shear} = -\frac{1}{\sqrt{1 - (p_{12} \cdot p_{13})^2}}(p_3 - p_1) \tag{4.11}$$

$$\nabla_{p_3} C_{shear} = -\frac{1}{\sqrt{1 - (p_{12} \cdot p_{13})^2}}(p_2 - p_1) \tag{4.12}$$

and in order to conserve linear momentum we know that [40]:

$$\nabla_{p_1} C_{shear} = -\nabla_{p_2} C_{shear} - \nabla_{p_3} C_{shear} \tag{4.13}$$

4.1.3 Damping

Incorporating a suitable damping scheme can enhance the overall quality of dynamic simulations. One significant advantage of damping is its ability to improve stability by reducing temporal oscillations in the positions of objects. This reduction in oscillations allows for the utilization of larger time steps, resulting in improved performance of the dynamic simulation [9].

However, it is important to note that damping also alters the dynamic behavior of the simulated objects. While this can have desired effects, such as minimizing oscillations in deformable solids, it can also introduce undesirable changes in linear or angular momentum throughout the object. These disturbances in momentum can impact the overall motion and

behavior of the simulated objects, which needs to be considered and carefully controlled in order to achieve the desired simulation results [11].

4.1.4 Stiffness

Although the stiffness k of the constraint hasn't been mentioned so far. There are several ways to incorporate it. The simplest variant is to multiply the corrections Δp by $k \in [0...1]$. However, for multiple iteration loops of the solver, the effect of k is non-linear [40]. And even transforming this effect into linear leaves us with a problem: the resulting material stiffness will still be dependent on the iteration count of the simulation. This introduces an extension of PBD that solves this issue and a couple of more, called Extended Position-Based Dynamics (XPBD) [37].

4.2 Extended Position-Based Dynamics

4.2.1 Why use XPBD instead of PBD?

One of the observed limitations in PBD is the dependence of constraint stiffness on the chosen time-step size and iteration count in the constraint solver. It has been noted that as the number of constraint iterations approaches infinity, the constraints tend to become infinitely stiff [11].

To address the issue of stiffness dependence on time-step size and iteration count in PBD, an extension called Extended Position-Based Dynamics (XPBD) [37] was introduced. XPBD builds upon a compliant constraint formulation based on [50], which associates an inverse stiffness parameter, known as compliance $\alpha = \frac{1}{k}$, with each constraint.

4.2.2 Main differences

The derivation of XPBD reveals an alternative interpretation of the λ values calculated for each constraint during a PBD iteration as incremental changes to a total multiplier. This modification alters (4.4) from λ_i to $\Delta\lambda_i$ in the original PBD formulation, and then applies this new compliance parameter (α) in the following manner:

$$\Delta p_i = w_i \nabla_{p_i} C(p) \Delta\lambda_i \tag{4.14}$$

$\Delta\lambda_i$ being computed as:

$$\Delta\lambda_i = -\frac{C(p) + \widetilde{\alpha}\lambda_i}{\sum_i w_i |\nabla_{p_i} C(p)|^2 + \widetilde{\alpha}} \tag{4.15}$$

where $\widetilde{\alpha} = \frac{\alpha}{\Delta t^2}$ is the time-step scaled compliance parameter, and $\lambda_{i+1} = \lambda_i + \Delta\lambda_i$ computed in

each iteration [37].

The intriguing additional terms found in the denominator of Equation 4.15 play a distinctive role in constraining the magnitude of the force exerted by a constraint. As the Lagrange multiplier λ increases, the incremental change in the constraint diminishes. This effect ensures that constraints with zero compliance ($\alpha = 0$) adopt the same formulation as regular PBD, resulting in infinitely rigid constraints (Equation 4.5) [37].

The total Lagrange multiplier λ holds remarkable significance, serving as a quantifiable measure of the collective force the constraint applies to the particles. It is worth noting that XPBD does not accelerate the convergence of PBD, it still requires the same number of iterations to attain a stiff solution [11].

In addition, the XPBD method introduces yet another enhancement. Rather than employing an external damping function relying on linear and angular velocities [40], XPBD adopts a different approach to dampening global motion. It incorporates the Rayleigh dissipation function, denoted as R, which is expressed as $R = \frac{1}{2}\dot{C}(o)^T \beta \dot{C}(p)$ [15], where $C(p)$ is the constraint function and β is the diagonal matrix of damping coefficients. This damping model stems directly from the total Lagrange multiplier, similar to the formulation presented in:

$$\Delta\lambda_i = -\frac{C(p) + \widetilde{\alpha}\lambda_i + \gamma(\nabla_{p_i} C(p) v_n)}{(1+\gamma)(\sum_i w_i |\nabla_{p_i} C(p)|^2) + \widetilde{\alpha}} \tag{4.16}$$

where $\gamma = \frac{\widetilde{\alpha}\beta}{\Delta t}$ and $v_n = (x_n - x_{n-1})$, the instant velocity without step-size.

However, a notable drawback of this approach is its reliance on the compliance parameter (α) for incorporating the Rayleigh dissipation function. Consequently, the choice of the compliance parameter value directly influences the overall motion of the simulation [15].

4.2.3 Algorithm overview

Given all this information, we can combine everything into the following algorithm [16]

Algorithm 2: XPBD Algorithm

Loop
  $x_{n+1} = x_n + (\Delta t)v_n + (\Delta t)^2 w_i f_{ext}$
  initialize total Lagrange multiplier $\lambda_0 = 0$
  while $k < iterationCount$ **do**
    for *each Constraint* **do**
      compute $\Delta\lambda$ (4.16)
      compute Δx (4.14)
      update $\lambda_{k+1} = \lambda_k + \Delta\lambda$
      update $x_{k+1} = x_k + \Delta x$
    end
    $k = k + 1$
  end
  update positions $x_{n+1} = x_k$
  update velocities $v_{n+1} = \frac{x_{n+1} - x_n}{\Delta t}$
EndLoop

4.3 Parallelizing the simulation

To optimize the solving process and achieve faster simulations, we introduced a parallel implementation of the Gauss-Seidel solver. This improvement was made possible through a carefully designed precomputing phase in the simulation. Prior to initiating the simulation, we intelligently partitioned the primitives to ensure that no two connected primitives could belong to the same group. This strategic partitioning was essential to avoid race conditions, where simultaneous threads calculating the same information from different constraints might yield different outcomes, resulting in non-deterministic behavior.

We created a graph with each constraint in the system represented as a node. Nodes were connected if the corresponding constraints shared at least one particle. In the context of this thesis the primitives that we care about the most are edges (for stretching constraint) as seen in Figure 4.4 and faces (for shear and volume constraint) as seen in Figure 4.5, these images are inspired from [25].

In these images we can see that on the left side, we have a connected mesh comprising particles, where each edge represents a stretch constraint in Figure 4.4 and each face represents a shear/volume constraint in Figure 4.5, in between adjacent particles. On the right side, we see the constraint graph, where each node corresponds to a specific constraint. Nodes are connected if the corresponding constraints share at least one common particle.

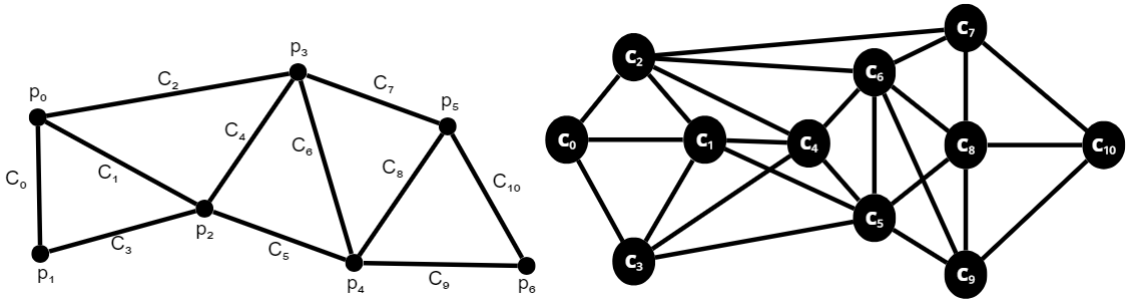


Figure 4.4: Edge constraint graph inspired from [25]

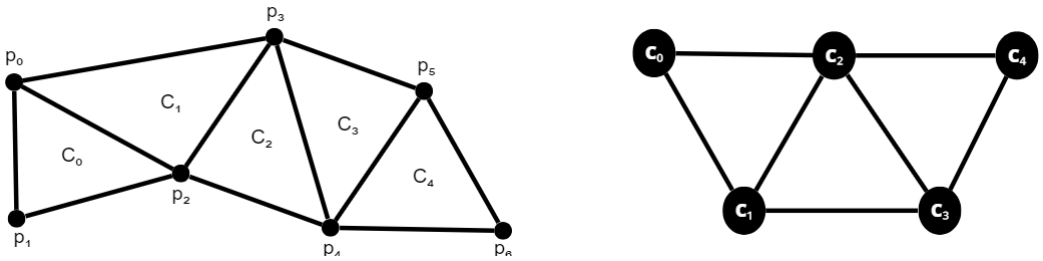


Figure 4.5: Face constraint graph inspired from [25]

After having the constraint graph, to organize the primitives into non-overlapping groups, we utilize a greedy graph coloring technique, called DSATUR (Degree of Saturation) [12], this algorithm consists of using saturation degree. The saturation degree of a vertex is defined by the number of different colored vertices to which it is adjacent [12]. To the best of our knowledge, this is the first time this algorithm has been adapted in the context of computer graphics applications.

The pseudocode goes as follows:

Algorithm 3: DSATUR pseudocode

1 Arrange the vertices by decreasing the order of degrees.
2 Color a vertex of the maximal degree with color 0 and increase the saturation degree of its neighboring vertices.
3 Choose a vertex with a maximal saturation degree. If there is equality, choose any vertex of maximal degree in the uncolored subgraph.
4 Color the chosen vertex with the least possible (lowest numbered) color and increase the saturation degree of its uncolored neighboring vertices.
5 If all the vertices are colored, stop. Otherwise, return to 3.

4.3.1 DSATUR Example Iterations

In order to better understand how this algorithm works, let's apply it to Figure 4.4, we'll be using 4 colors, 0 corresponds to red, 1 to blue, 2 to orange and 3 to yellow.

4.3.1.1 Iteration 0

In Table 4.1, we use iteration 0 to obtain all degrees and initialize the colors. A value of -1 represents an uncolored vertex, and at this stage, all vertices have a degree of saturation equal to 0 since none have been colored yet.

Table 4.1: Iteration 0 of the DSATUR Algorithm applied to Figure 4.4

Vertex	Color	Degree	Degree of Saturation
C_0	-1	3	0
C_1	-1	5	0
C_2	-1	5	0
C_3	-1	4	0
C_4	-1	5	0
C_5	-1	6	0
C_6	-1	6	0
C_7	-1	4	0
C_8	-1	5	0
C_9	-1	4	0
C_{10}	-1	3	0

4.3.1.2 Iteration 1

In Table 4.1, all vertices have a degree of saturation of 0, indicating that none of them have been colored yet. For the first iteration of the DSatur algorithm, we select the vertex with the highest degree to color. In this case, two vertices have the highest degree (6), and we choose the first one encountered, which is vertex C_5.

After coloring C_5, we increment the degree of saturation for its uncolored neighboring vertices, which are C_1, C_3, C_4, C_6, C_8, and C_9. The updated state of the vertices is shown in Table 4.2 and can be visually observed in Figure 4.6.

Table 4.2: Iteration 1 of the DSATUR Algorithm applied to Figure 4.4

Vertex	Color	Degree	Degree of Saturation
C_0	-1	3	0
C_1	-1	5	1
C_2	-1	5	0
C_3	-1	4	1
C_4	-1	5	1
C_5	0	-	-
C_6	-1	6	1
C_7	-1	4	0
C_8	-1	5	1
C_9	-1	4	1
C_{10}	-1	3	0

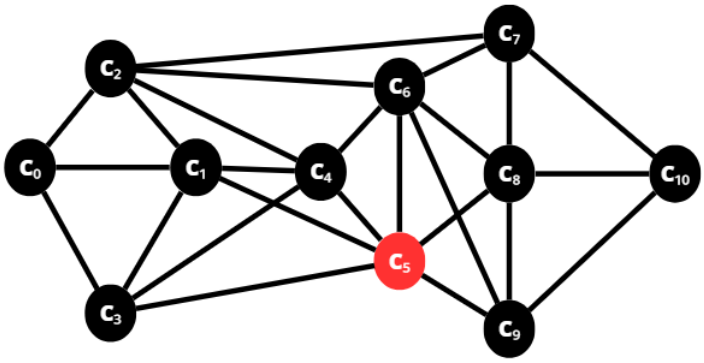


Figure 4.6: Iteration 1 of the visual representation of the DSATUR algorithm

4.3.1.3 Iteration 2

In Table 4.2, we start the main loop of the DSATUR algorithm. Among the vertices with the highest degree of saturation (1), which are $C_1, C_3, C_4, C_6, C_8, C_9$, we select the vertex with the highest degree. In this case, vertex C_6 has the highest degree (6). Since the first available color is 1 (as 0 is already taken by a connected vertex C_5), we color vertex C_6 with color 1.

After coloring C_6, we increment the degree of saturation for its uncolored neighboring vertices, which are C_2, C_4, C_7, C_8, C_9. The updated state of the vertices is shown in Table 4.3 and can be visually observed in Figure 4.7.

Table 4.3: Iteration 2 of the DSATUR Algorithm applied to Figure 4.4

Vertex	Color	Degree	Degree of Saturation
C_0	-1	3	0
C_1	-1	5	1
C_2	-1	5	1
C_3	-1	4	1
C_4	-1	5	2
C_5	0	-	-
C_6	1	-	-
C_7	-1	4	1
C_8	-1	5	2
C_9	-1	4	2
C_{10}	-1	3	0

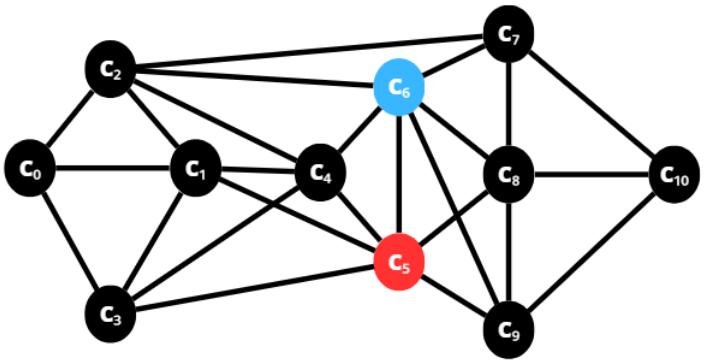


Figure 4.7: Iteration 2 of the visual representation of the DSATUR algorithm

4.3.1.4 Iteration 3

In Table 4.3, among the vertices with the highest degree of saturation (2), which are C_4, C_8, C_9, we select the first vertex with the highest degree. In this case, vertex C_4 has the highest degree (5). Since the first available color is 2 (as 0 is already taken by a connected vertex C_5, and 1 by C_6), we color vertex C_4 with color 2.

After coloring C_4, we increment the degree of saturation for its uncolored neighboring vertices, which are C_1, C_3, C_4. The updated state of the vertices is shown in Table 4.4 and can be visually observed in Figure 4.8.

Table 4.4: Iteration 3 of the DSATUR Algorithm applied to Figure 4.4

Vertex	Color	Degree	Degree of Saturation
C_0	-1	3	0
C_1	-1	5	2
C_2	-1	5	2
C_3	-1	4	2
C_4	2	-	-
C_5	0	-	-
C_6	1	-	-
C_7	-1	4	1
C_8	-1	5	2
C_9	-1	4	2
C_{10}	-1	3	0

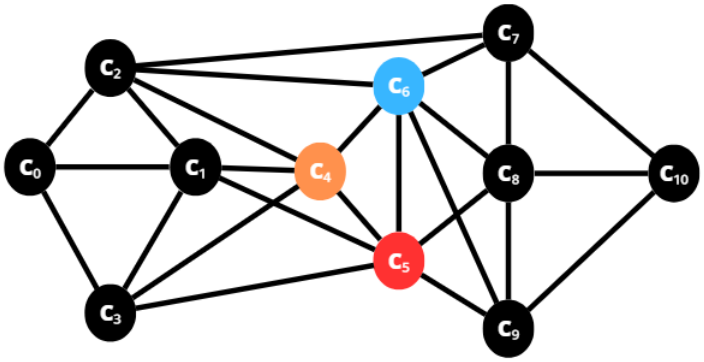


Figure 4.8: Iteration 3 of the visual representation of the DSATUR algorithm

4.3.1.5 Iteration 4

In Table 4.4, among the vertices with the highest degree of saturation (2), which are C_1, C_2, C_3, C_8, C_9, we select the first vertex with the highest degree. In this case, vertex C_1 has the highest degree (5). Since the first available color is 1 (as 0 is already taken by a connected vertex C_4), we color vertex C_1 with color 1.

After coloring C_1, we increment the degree of saturation for its uncolored neighboring vertices, which are C_0, C_2, C_3. The updated state of the vertices is shown in Table 4.5 and can be visually observed in Figure 4.9.

Table 4.5: Iteration 4 of the DSATUR Algorithm applied to Figure 4.4

Vertex	Color	Degree	Degree of Saturation
C_0	-1	3	1
C_1	1	-	-
C_2	-1	5	3
C_3	-1	4	3
C_4	2	-	-
C_5	0	-	-
C_6	1	-	-
C_7	-1	4	1
C_8	-1	5	2
C_9	-1	4	2
C_{10}	-1	3	0

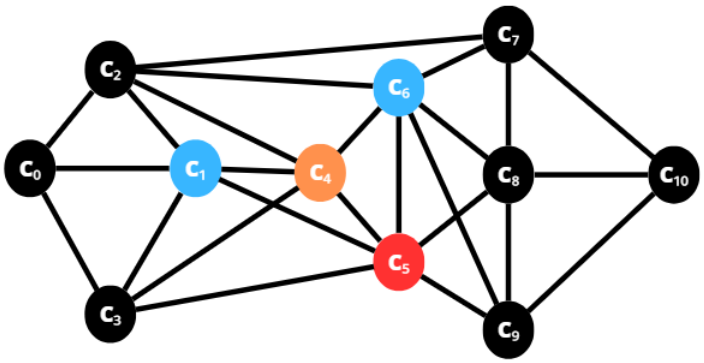


Figure 4.9: Iteration 4 of the visual representation of the DSATUR algorithm

4.3.1.6 Iteration 5

In Table 4.5, among the vertices with the highest degree of saturation (3), which are C_2, C_3, we select the vertex with the highest degree. In this case, vertex C_2 has the highest degree (5). Since the first available color is 0 (as we don't have any connected vertex with the color 0), we color vertex C_1 with color 0.

After coloring C_2, we increment the degree of saturation for its uncolored neighboring vertices, which are C_0, C_7. The updated state of the vertices is shown in Table 4.6 and can be visually observed in Figure 4.10.

Table 4.6: Iteration 5 of the DSATUR Algorithm applied to Figure 4.4

Vertex	Color	Degree	Degree of Saturation
C_0	-1	3	2
C_1	1	-	-
C_2	0	-	-
C_3	-1	4	3
C_4	2	-	-
C_5	0	-	-
C_6	1	-	-
C_7	-1	4	2
C_8	-1	5	2
C_9	-1	4	2
C_{10}	-1	3	0

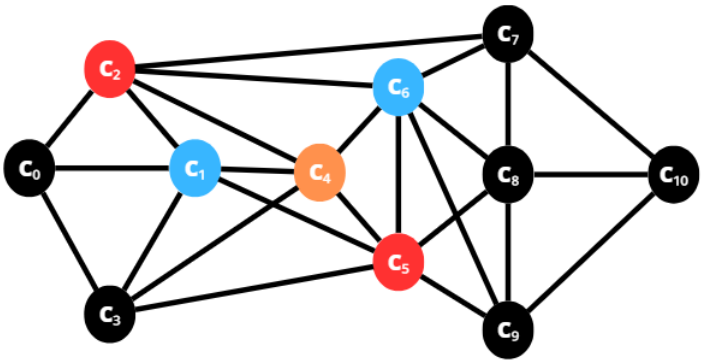


Figure 4.10: Iteration 5 of the visual representation of the DSATUR algorithm

4.3.1.7 Iteration 6

In Table 4.6, since there's only one vertex with the highest degree of saturation (3), which is C_3, we select that one. Since the first available color is 3 (as 0 is already taken by a connected vertex C_5, 1 by C_1 and 2 by C_4), we color vertex C_3 with color 3.

After coloring C_3, we increment the degree of saturation for its uncolored neighboring vertices, which is C_0. The updated state of the vertices is shown in Table 4.7 and can be visually observed in Figure 4.11.

Table 4.7: Iteration 6 of the DSATUR Algorithm applied to Figure 4.4

Vertex	Color	Degree	Degree of Saturation
C_0	-1	3	3
C_1	1	-	-
C_2	0	-	-
C_3	3	-	-
C_4	2	-	-
C_5	0	-	-
C_6	1	-	-
C_7	-1	4	2
C_8	-1	5	2
C_9	-1	4	2
C_{10}	-1	3	0

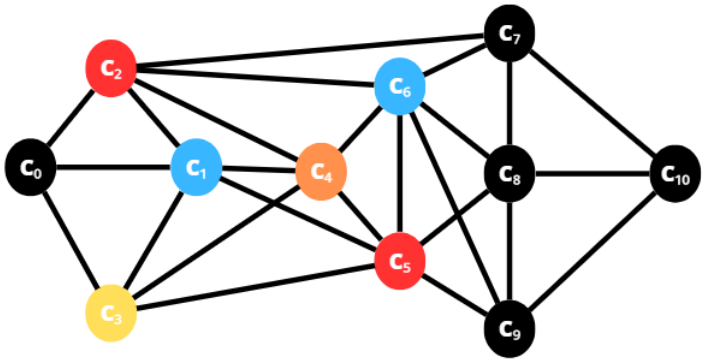


Figure 4.11: Iteration 6 of the visual representation of the DSATUR algorithm

4.3.1.8 Iteration 7

In Table 4.7, since there's only one vertex with the highest degree of saturation (3), which is C_0, we select that one. Since the first available color is 2 (as 0 is already taken by a connected vertex C_2, and 1 by C_1), we color vertex C_0 with color 2.

After coloring C_0, we don't increment any degree of saturation because this vertex has no uncolored neighboring vertices. The updated state of the vertices is shown in Table 4.8 and can be visually observed in Figure 4.12.

Table 4.8: Iteration 7 of the DSATUR Algorithm applied to Figure 4.4

Vertex	Color	Degree	Degree of Saturation
C_0	2	-	-
C_1	1	-	-
C_2	0	-	-
C_3	3	-	-
C_4	2	-	-
C_5	0	-	-
C_6	1	-	-
C_7	-1	4	2
C_8	-1	5	2
C_9	-1	4	2
C_{10}	-1	3	0

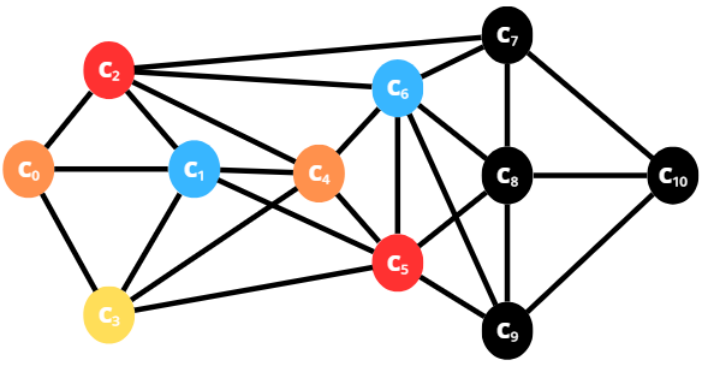


Figure 4.12: Iteration 7 of the visual representation of the DSATUR algorithm

4.3.1.9 Iteration 8

In Table 4.8, among the vertices with the highest degree of saturation (2), which are C_7, C_8, C_9, we select the vertex with the highest degree. In this case, vertex C_8 has the highest degree (5). Since the first available color is 2 (as 0 is already taken by a connected vertex C_5, and 1 by C_6), we color vertex C_8 with color 2.

After coloring C_8, we increment the degree of saturation for its uncolored neighboring vertices, which are C_7, C_9, C_{10}. The updated state of the vertices is shown in Table 4.9 and can be visually observed in Figure 4.13.

Table 4.9: Iteration 8 of the DSATUR Algorithm applied to Figure 4.4

Vertex	Color	Degree	Degree of Saturation
C_0	2	-	-
C_1	1	-	-
C_2	0	-	-
C_3	3	-	-
C_4	2	-	-
C_5	0	-	-
C_6	1	-	-
C_7	-1	4	3
C_8	2	-	-
C_9	-1	4	3
C_{10}	-1	3	1

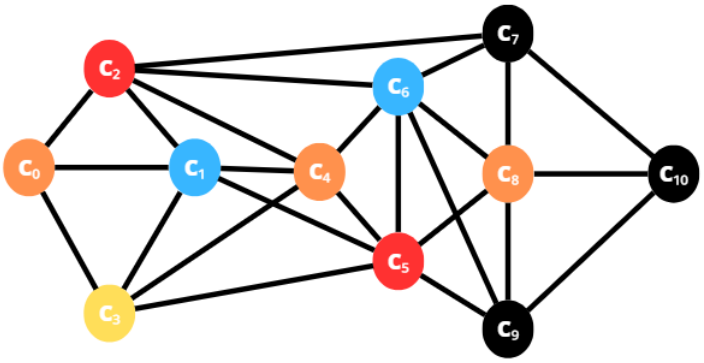


Figure 4.13: Iteration 8 of the visual representation of the DSATUR algorithm

4.3.1.10 Iteration 9

In Table 4.9, among the vertices with the highest degree of saturation (3), which are C_7, C_9 we select the first vertex with the highest degree. In this case, vertex C_7 is the first one with the highest degree (5). Since the first available color is 3 (as 0 is already taken by a connected vertex C_2, 1 by C_6, and 2 by C_8), we color vertex C_7 with color 3.

After coloring C_7, we increment the degree of saturation for its uncolored neighboring vertex, which is C_{10}. The updated state of the vertices is shown in Table 4.10 and can be visually observed in Figure 4.14.

Table 4.10: Iteration 9 of the DSATUR Algorithm applied to Figure 4.4

Vertex	Color	Degree	Degree of Saturation
C_0	2	-	-
C_1	1	-	-
C_2	0	-	-
C_3	3	-	-
C_4	2	-	-
C_5	0	-	-
C_6	1	-	-
C_7	3	-	-
C_8	2	-	-
C_9	-1	4	3
C_{10}	-1	3	2

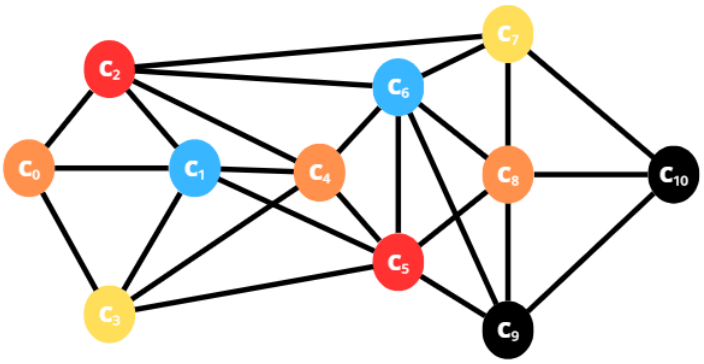


Figure 4.14: Iteration 9 of the visual representation of the DSATUR algorithm

4.3.1.11 Iteration 10

In Table 4.10, since there's only one vertex with the highest degree of saturation (3), which is C_9, we select that one. Since the first available color is 3 (as 0 is already taken by a connected vertex C_5, 1 by C_6, and 2 by C_8), we color vertex C_9 with color 3.

After coloring C_9, we increment the degree of saturation for its uncolored neighboring vertex, which is C_{10}. The updated state of the vertices is shown in Table 4.11 and can be visually observed in Figure 4.15.

Table 4.11: Iteration 10 of the DSATUR Algorithm applied to Figure 4.4

Vertex	Color	Degree	Degree of Saturation
C_0	2	-	-
C_1	1	-	-
C_2	0	-	-
C_3	3	-	-
C_4	2	-	-
C_5	0	-	-
C_6	1	-	-
C_7	3	-	-
C_8	2	-	-
C_9	3	-	-
C_{10}	-1	3	2

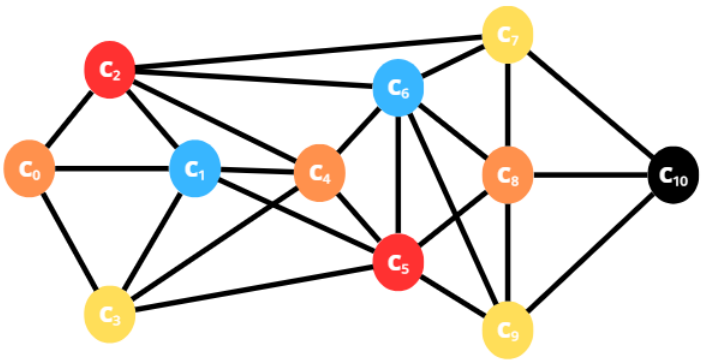


Figure 4.15: Iteration 10 of the visual representation of the DSATUR algorithm

4.3.1.12 Final Iteration

In Table 4.11, since there's only one vertex remaining which is C_{10}, we select that one. Since the first available color is 0, we color vertex C_{10} with color 0.

After coloring C_{10}, we're finished because all vertices are colored. The final state of the vertices is shown in Table 4.12 and can be visually observed in Figure 4.16.

Table 4.12: Iteration 11 of the DSATUR Algorithm applied to Figure 4.4

Vertex	Color	Degree	Degree of Saturation
C_0	2	-	-
C_1	1	-	-
C_2	0	-	-
C_3	3	-	-
C_4	2	-	-
C_5	0	-	-
C_6	1	-	-
C_7	3	-	-
C_8	2	-	-
C_9	3	-	-
C_{10}	0	-	-

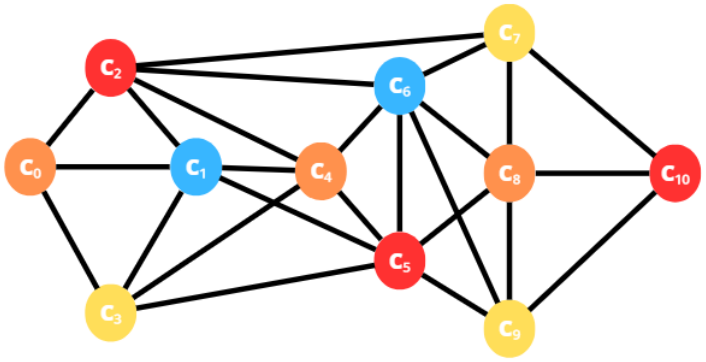


Figure 4.16: Final iteration of the visual representation of the DSATUR algorithm

Each color represents a cluster of constraints, so we have 4 clusters of constraints $\{C_2, C_5, C_{10}\}$, $\{C_1, C_6\}$, $\{C_0, C_4, C_8\}$, $\{C_3, C_7, C_9\}$ and we instantiate a thread for each of these clusters. This way, the system is solved in fewer steps than the sequential approach, leading to faster convergence.

Chapter 5

Implementation and Results

5.1 Implementation

The core of this thesis is to build a novel optimized precomputation method for the graph coloring method for a parallel XPBD algorithm implementation, serving as a plugin for Autodesk Maya using C++ and example scenes were arranged using Python scripting. This plugin operates on the deforming mesh, performing all the calculations discussed in the previous chapter, utilizing either GPU (CUDA) or CPU (standard concurrency library). In its early stages, this framework primarily involved a single-threaded CPU and multi-threaded GPU, with a specific focus on handling simulations with stretching constraints and collisions. This initial framework was employed in a previous implementation of [16] and utilized the precomputation coloring algorithm introduced there.

Subsequently, our thesis work expanded upon this foundation in several key aspects. Firstly, we introduced an alternative coloring algorithm for the precomputation phase (DSATUR). Secondly, we optimized the CPU simulation by implementing multi-threading using a concurrency system. Furthermore, we extended the GPU simulation to support a larger number of coloring groups, effectively scaling its capabilities. To broaden the scope and applicability of the XPBD algorithm, we undertook the task of implementing shear and volume constraints from scratch, enabling more comprehensive and accurate simulations. Our experimentation encompassed scenarios involving 2D and 3D volumetric triangle meshes. All the experiments and evaluations presented within this thesis were conducted on a robust hardware setup comprising an 8-core Intel i9-9900K processor with a clock speed of 3.60GHz, accompanied by 32 GB of RAM, and an NVIDIA GeForce RTX 2080S graphics card.

In this thesis, we expanded the initial framework with the main goal of speeding up the precomputation phase. And as such we conducted comparisons to measure the extent of time improvement achieved in this specific phase and to evaluate the overall performance of our simulations. This comparative analysis is a key element of our research, demonstrating the effectiveness of our contributions in optimizing the XPBD algorithm's precomputation for

practical use.

5.2 Results

We devised eight comparison test scenarios, all derived from a common scene inspired by the work presented in [16], as depicted in Figure 5.1. In each scenario, the mesh composition varies, encompassing different quantities of vertices, edges, and faces.

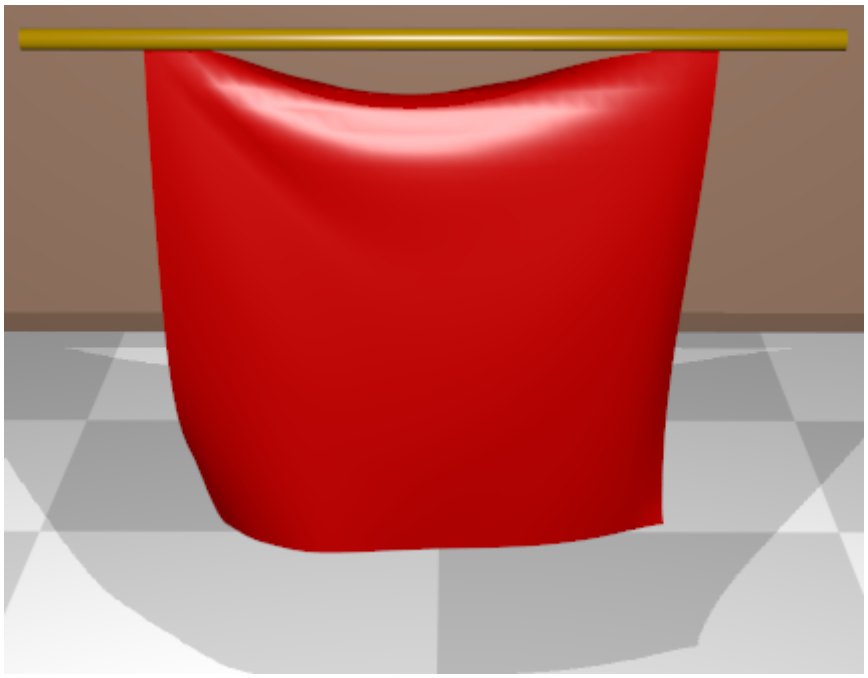


Figure 5.1: A piece of cloth is hanged under constant gravity regenerated from [16]

For each scenario, we provide a comparison of the precomputation time, which signifies the milliseconds required to generate the color groups essential for parallelization (Figure 5.2). Additionally, we present an evaluation of overall performance, measured in frames per second (FPS), for both CPU (Figure 5.3) and GPU (Figure 5.4) implementations, comparing our method to the initial framework from [16]. This can be seen in depth with the respective mesh composition for each model of the comparison in Table 5.1.

All these results were taken from an average of 10 executions of the simulation, and all were created under the same conditions, such as $\alpha = 0.00001$, iteration count $= 100$, damping coef. $= 0.3$ and step-size $= \frac{1}{24}$.

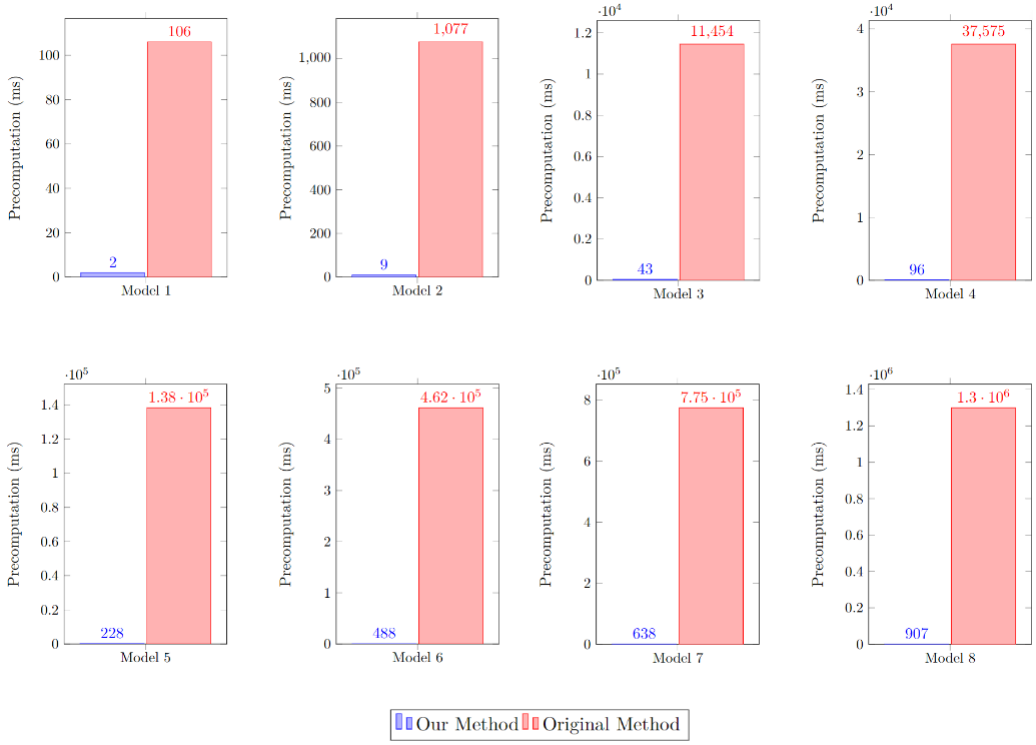


Figure 5.2: Comparison of the precomputation timings

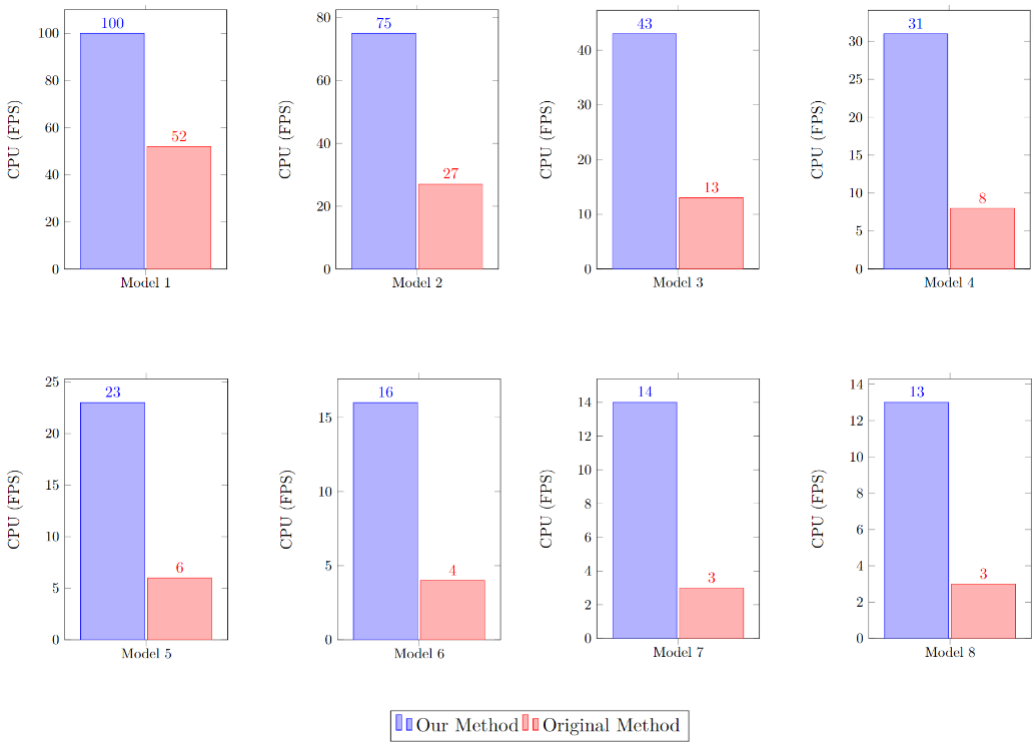


Figure 5.3: Comparison of the performance utilizing CPU implementation (Note that our method utilizes multi-threading while the original uses single-threading)

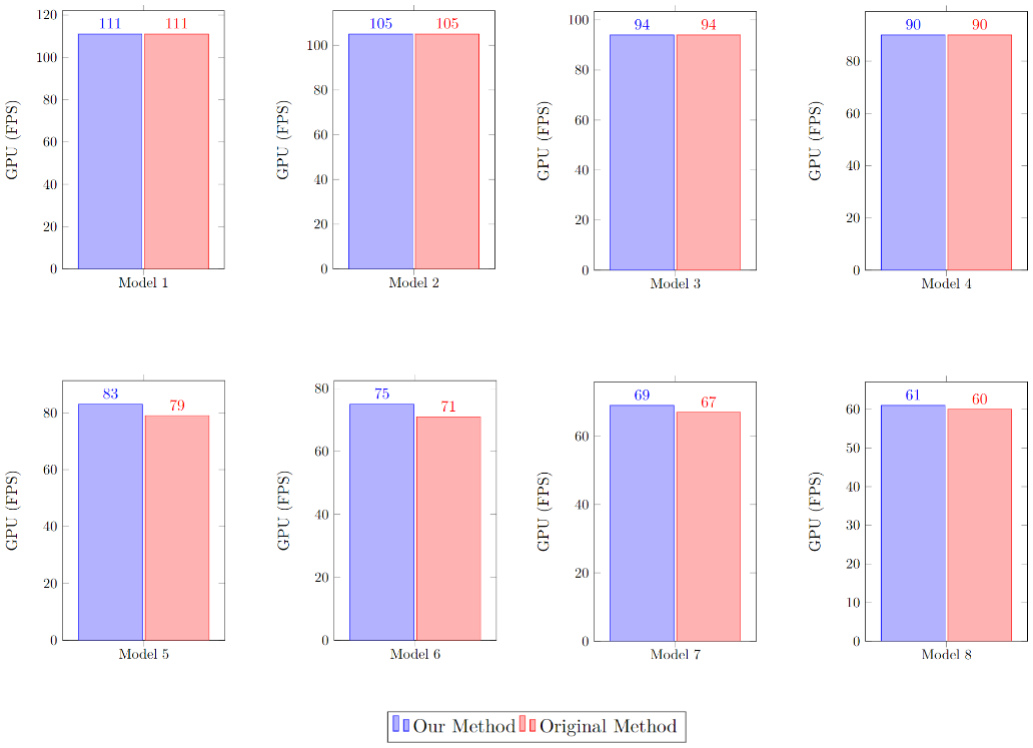


Figure 5.4: Comparison of the performance utilizing GPU implementation

Table 5.1: Comparisons

Details of the models				Our method			Original method [15]		
Model	# vertices	# edges	# faces (tri.)	Precomputation	CPU	GPU	Precomputation	CPU	GPU
Model1	441	1240	800	2 ms	100 FPS	111 FPS	106 ms	52 FPS	111 FPS
Model2	961	2760	1800	9 ms	75 FPS	105 FPS	1077 ms	27 FPS	105 FPS
Model3	2116	6165	4050	43 ms	43 FPS	94 FPS	11454 ms	13 FPS	94 FPS
Model4	3136	9185	6050	96 ms	31 FPS	90 FPS	37575 ms	8 FPS	90 FPS
Model5	4900	14421	9522	228 ms	23 FPS	83 FPS	138327 ms	6 FPS	79 FPS
Model6	7225	21336	14112	488 ms	16 FPS	75 FPS	461820 ms	4 FPS	71 FPS
Model7	8464	25025	16562	638 ms	14 FPS	69 FPS	775043 ms	3 FPS	67 FPS
Model8	10201	30200	20000	907 ms	13 FPS	61 FPS	1299625 ms	3 FPS	60 FPS

A notable reduction in precomputation time is evident in our results. Furthermore, our observations reveal that GPU performance remains relatively consistent, while CPU performance sees substantial improvement as expected since the original method used single-threading while our proposed technique uses multi-threading.

Furthermore, to provide a more detailed examination of this system's capabilities, we conducted six distinct simulations categorized into two groups. The scene design for all these simulations draws inspiration from [1].

Three of the simulations showcase the stretching and shear constraints within the context of cloth dynamics. In these scenarios, we utilized a standard plane model from Maya and adjusted the material's color texture to red. While the other three highlight the stretching and volume constraints, this time within the context of objects with volume, in our case using a dog model

sourced from [2].

We begin by highlighting the influence of external gravitational forces, maintaining the fixation of multiple vertices of the model to a pole (see Figure 5.5 and Figure 5.6).

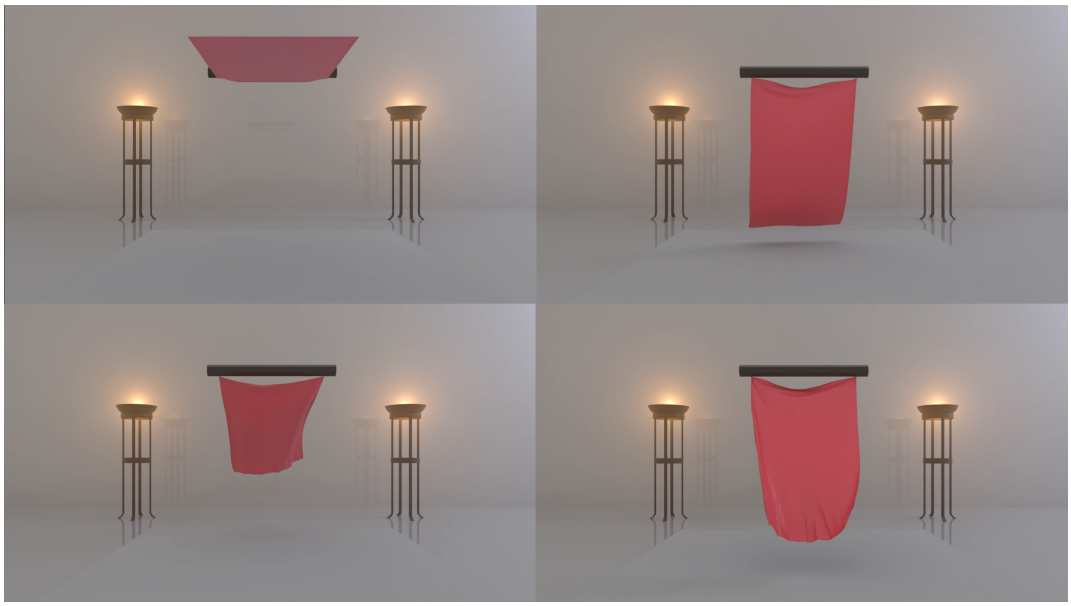


Figure 5.5: A piece of cloth is hanged under constant gravity with two vertices fixed to a pole. The conditions are α=0.000003, iteration count = 50, damping coef. = 0.12 and step-size=$\frac{1}{24}$

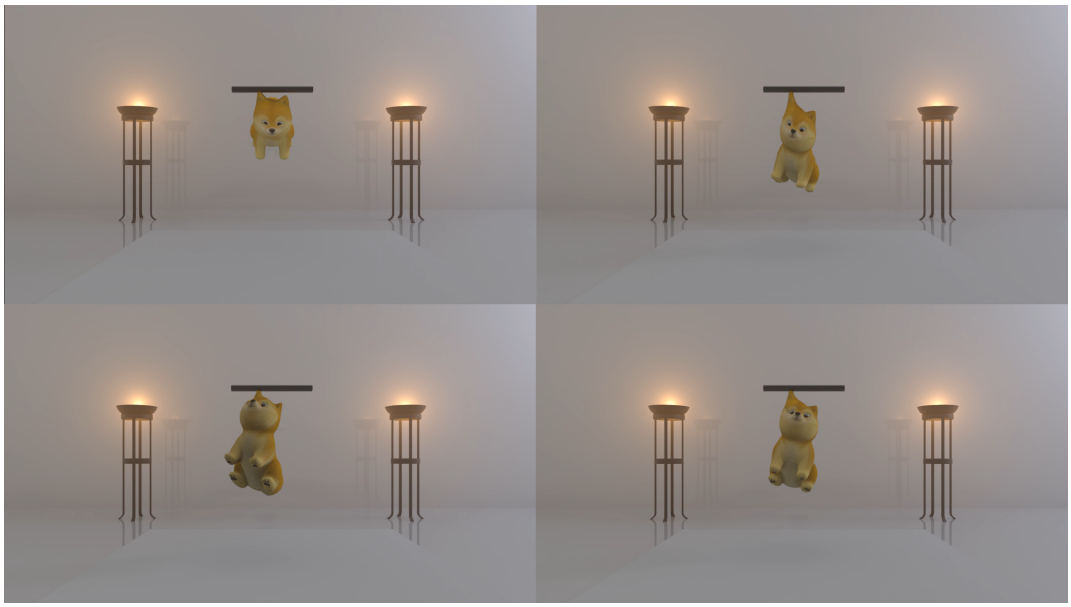


Figure 5.6: A volumetric model is hanged under constant gravity with multiple vertices fixed to a pole. The conditions are α=0.000005, iteration count = 50, damping coef. = 0.2 and step-size=$\frac{1}{24}$

In the next scenario, we replicate the same behavior as in the previous one but introduce an

animated ball to demonstrate sphere collision interactions (refer to Figure 5.7 and Figure 5.8).

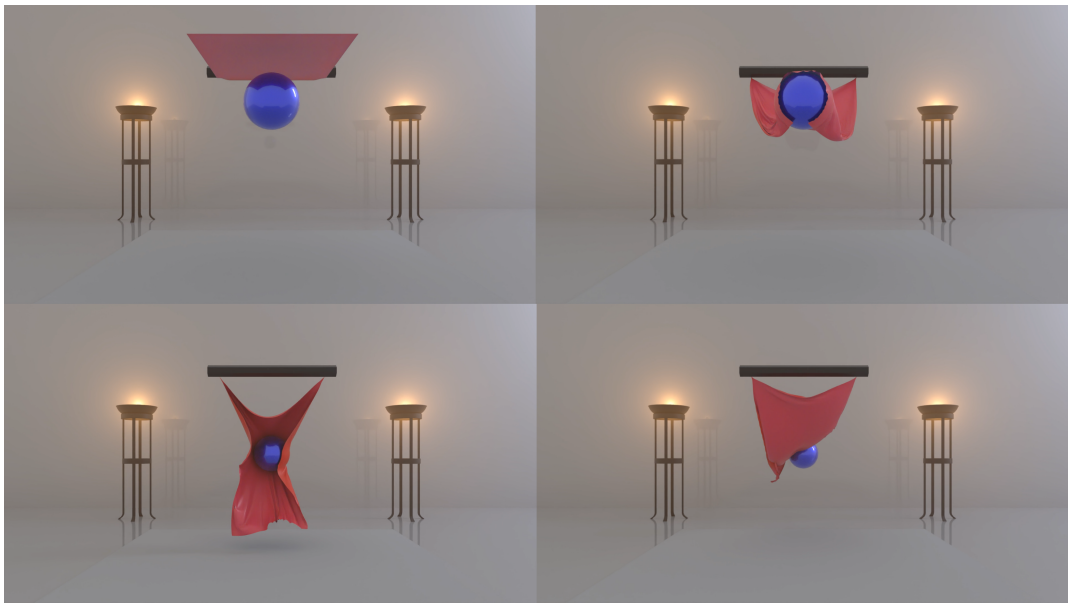


Figure 5.7: A piece of cloth is suspended under constant gravity with two vertices fixed to a pole, and an animated ball is present throughout the scene. The conditions are α=0.0000095, iteration count = 50, damping coef. = 0.35 and step-size=$\frac{1}{24}$

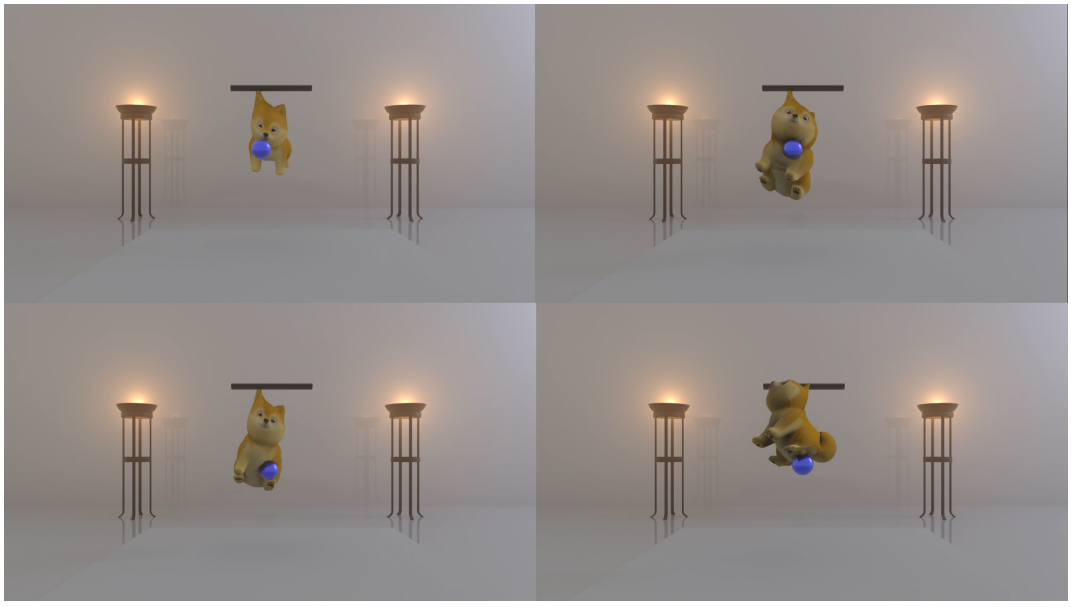


Figure 5.8: A volumetric model is suspended under constant gravity with multiple vertices fixed to a pole, and an animated ball is present throughout the scene. The conditions are α=0.000005, iteration count = 50, damping coef. = 0.2 and step-size=$\frac{1}{24}$

In the final scenario, there are no fixed vertices, enabling the model to fall freely and interact with the spheres. This setup accentuates the impact of gravitational forces and collision dynamics

(as depicted in Figure 5.9 and Figure 5.10).

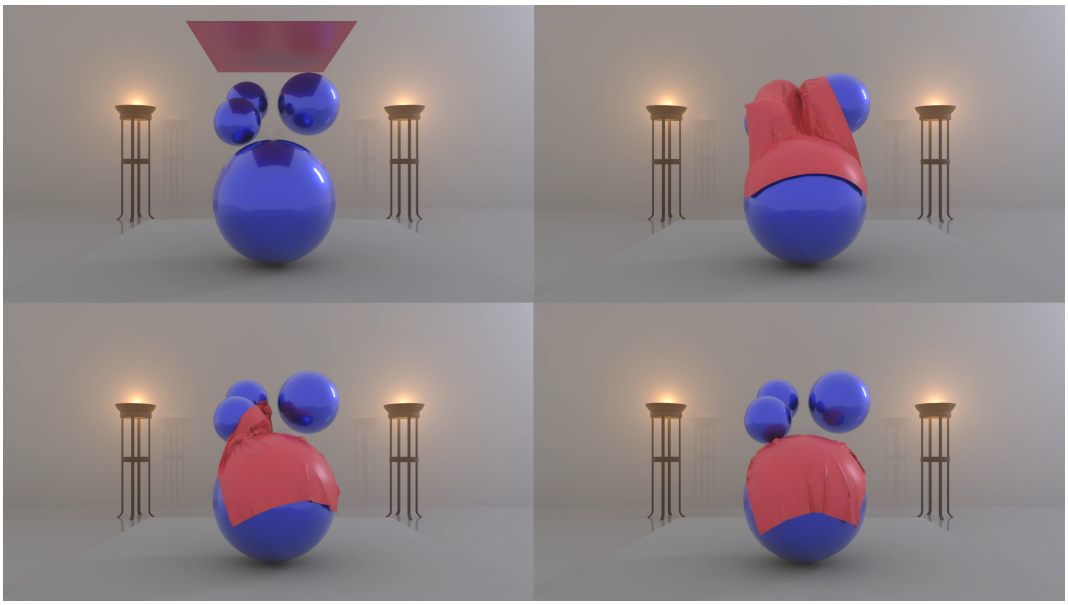


Figure 5.9: A piece of cloth is allowed to fall freely under the influence of constant gravity, engaging in interactions with multiple spheres. The conditions are α=0.000003, iteration count = 50, damping coef. = 0.2 and step-size=$\frac{1}{24}$

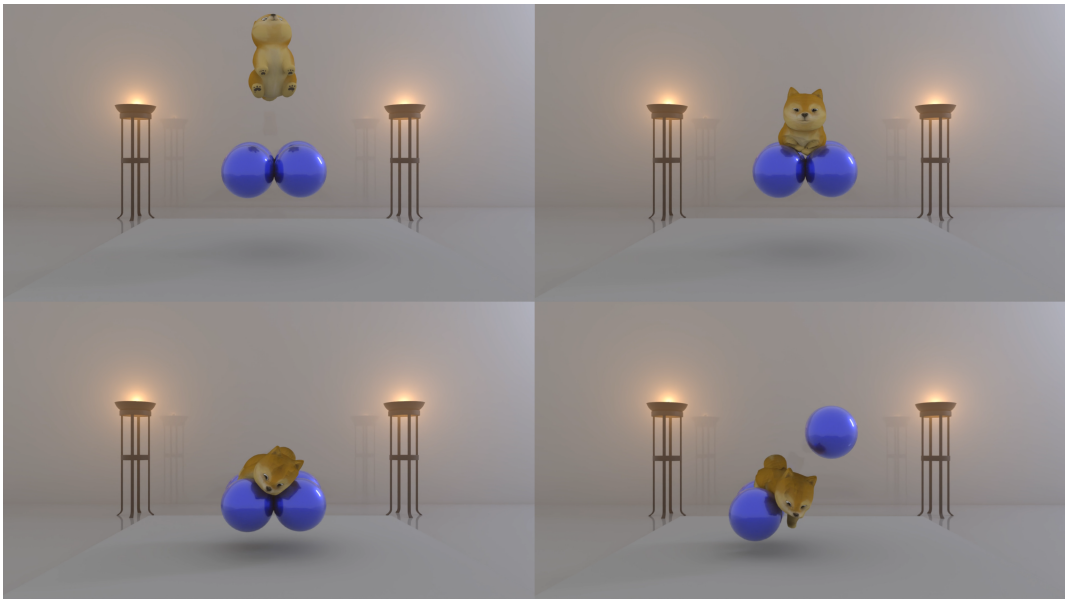


Figure 5.10: A volumetric model is allowed to fall freely under the influence of constant gravity, engaging in interactions with multiple spheres. The conditions are α=0.000006, iteration count = 50, damping coef. = 0.2 and step-size=$\frac{1}{24}$

The outcomes of these scenarios are shown in regards to the precomputation (Figure 5.11), CPU implementation performance (Figure 5.12) and GPU implementation performance (Figure 5.13). Models and performance rates are comprehensively detailed in Table 5.2. Each scenario is directly linked to its corresponding figure, offering insights into precomputation times and overall performance for both CPU and GPU.

Table 5.2: Results

Details of the models				Details of the simulation	Our method		
Figure	# vertices	# edges	# faces (tri.)	# sphere collisions	Precomputation	CPU	GPU
Figure 5.5	5041	14840	9800	0	587 ms	19 FPS	59 FPS
Figure 5.7	5041	14840	9800	1	650 ms	18 FPS	59 FPS
Figure 5.9	5041	14840	9800	5	655 ms	13 FPS	57 FPS
Figure 5.6	1856	5562	3708	0	91 ms	40 FPS	70 FPS
Figure 5.8	1856	5562	3708	1	96 ms	37 FPS	66 FPS
Figure 5.10	1856	5562	3708	5	96 ms	26 FPS	61 FPS

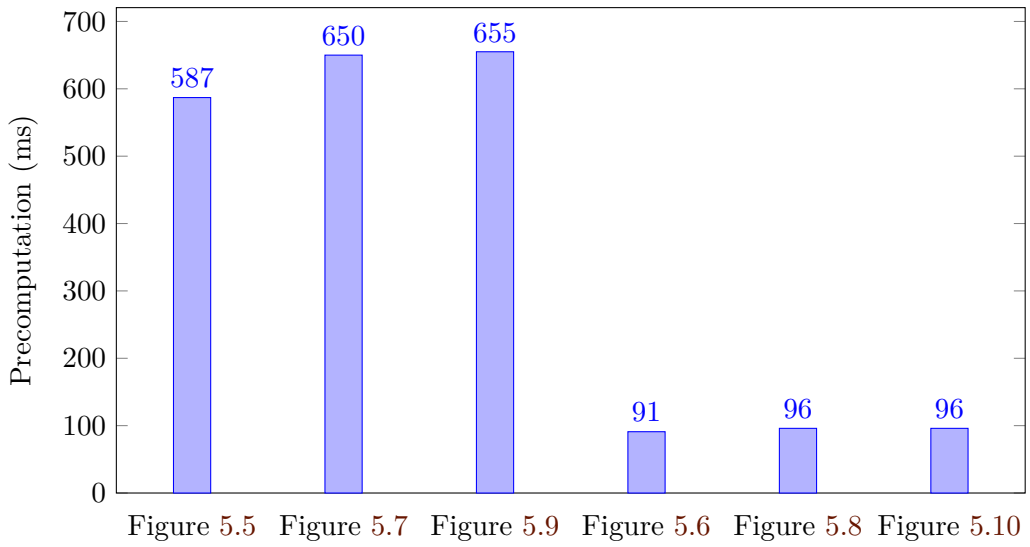


Figure 5.11: Precomputations of the simulations

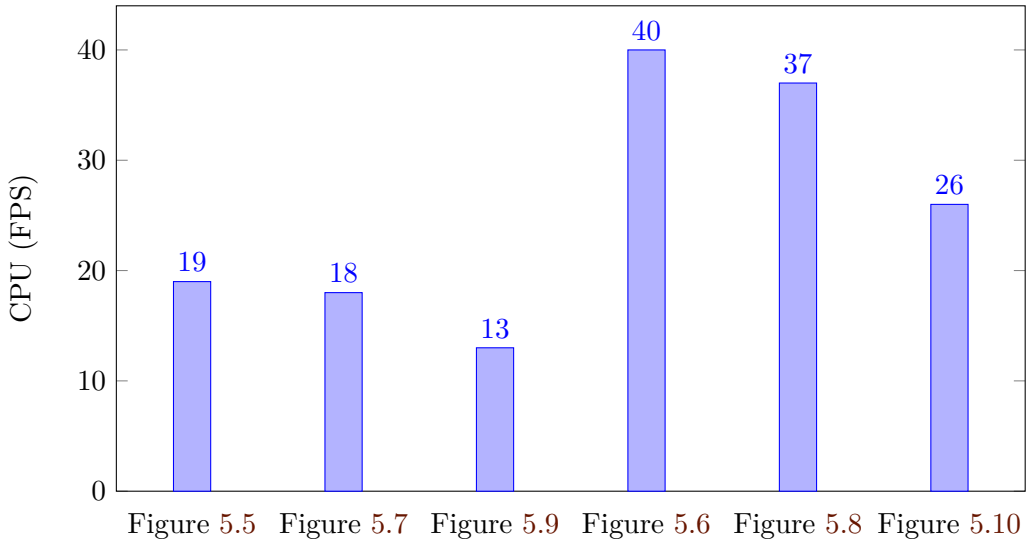


Figure 5.12: CPU implementation of the simulations

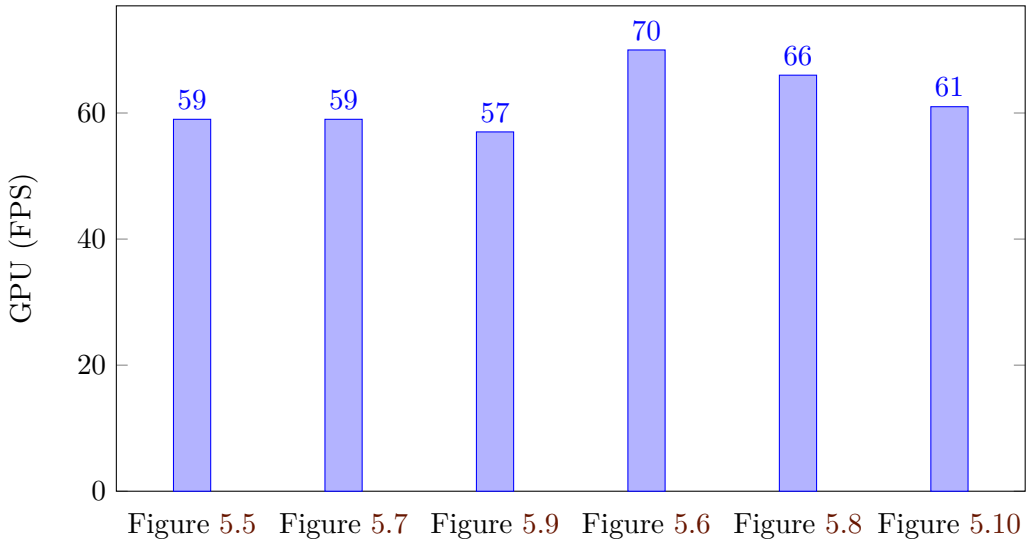


Figure 5.13: GPU implementation of the simulations

Chapter 6

Conclusion and Future Work

6.1 Conclusion

Throughout the progression of this thesis, we systematically accomplished the initial objectives we established. Our journey embarked with a comprehensive review of the foundational concepts underpinning physics-based simulations.

Subsequently, we delved deeper into Position-Based Dynamics (PBD) and, more precisely, Extended Position-Based Dynamics (XPBD), shedding light on their critical distinctions. This exploration unveiled the intricate mechanisms governing this method, encompassing the precise constraints we chose to incorporate in our research—stretching, shear, and volume constraints.

Furthermore, we conducted a thorough examination of the potential for parallelization within this system. The central focus of our thesis revolved around finding a graph-coloring algorithm that could significantly reduce precomputation times without adversely affecting simulation performance. Striking this balance was our primary goal throughout our research journey.

Addressing this challenge involved a process of experimentation, as the optimal approach wasn't initially efficient. Therefore, we attempted to enhance the existing graph-coloring algorithm by modifying data structures to improve its speed. However, it became apparent that a new coloring algorithm was necessary to achieve our performance goals. Given our requirement for strong performance and since an approximation of the optimal number of colors would suffice, we opted for a greedy algorithm. However, this approach introduced a new challenge as the standard greedy algorithm's performance heavily depends on the starting vertex.

To tackle this challenge, we sought to find an improved variant of the greedy graph-coloring algorithm, one that made more informed choices about where to begin the coloring process. Our exploration eventually led us to the DSATUR algorithm, as introduced by [12], which emerged as a crucial solution to our graph-coloring algorithm needs.

With the most significant contribution of this thesis firmly established, the next step was to

assess its effectiveness in comparison to the framework from which this entire endeavor originated, a framework developed for [16], which we can see again in Figure 6.1.

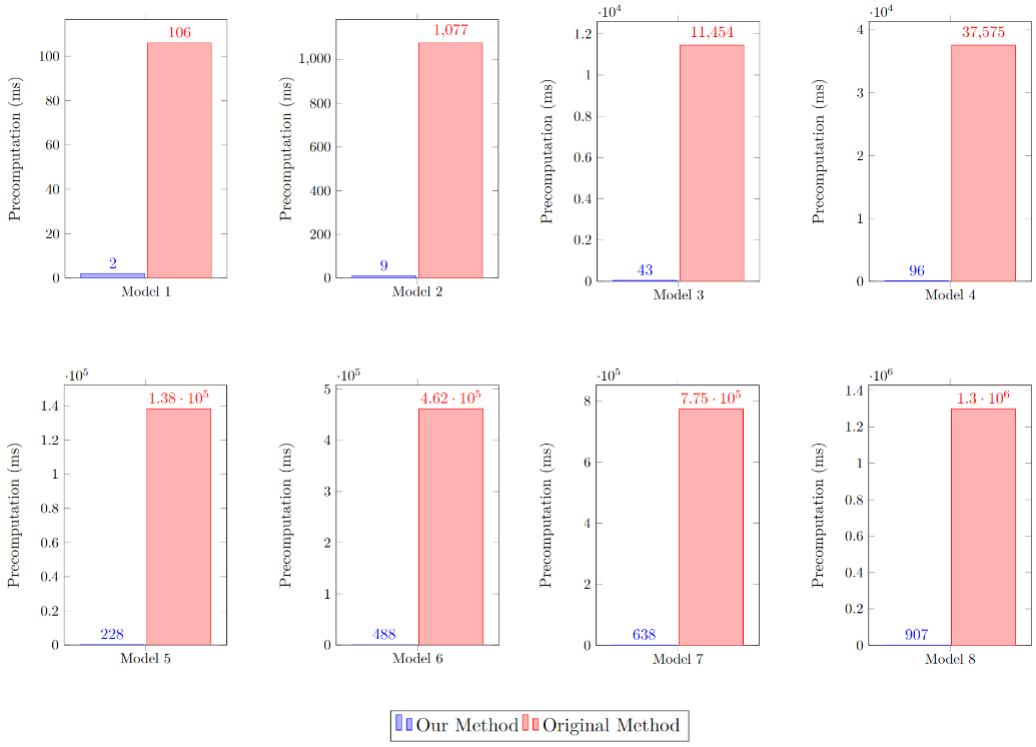


Figure 6.1: Comparison of the precomputation timings

This foundation had demonstrated impressive results, particularly for larger meshes, and further enhancements were shown in the performance through the utilization of the job system and CUDA for parallelization on both the CPU and GPU.

After our theoretical exploration and algorithmic enhancements, we conducted a series of six simulations. These simulations allowed us to demonstrate how our research findings fit into the practical framework of implemented constraints.

During this process, we engaged in a systematic approach of trial and error. We fine-tuned critical parameters such as the compliance parameter (α), damping coefficient, and iteration count. Through these simulations, we were able to validate the effectiveness of our contributions and gain practical insights into deformable object simulations.

6.2 Future Work

As we conclude this thesis, we acknowledge that the journey towards innovation in real-time physically-based simulation of deformable objects is ongoing. Building on the knowledge we've acquired during this research, we can now explore future possibilities and address some limitations:

- Optimizing Performance: A crucial avenue for further research involves optimizing the performance of our system. We can explore advanced parallelization techniques, leverage hardware-specific optimizations, and employ machine-learning approaches to predict and mitigate performance bottlenecks or to fine-tune the critical parameters of the simulation, so we don't need so much trial and error.

- Alternative Preprocessing Techniques: While our pioneering use of the DSATUR graph coloring algorithm has proven its performance, it can be sensitive to mesh complexity and hardware configurations. We can delve into research on alternative preprocessing methods, exploring a spectrum of graph-coloring algorithms and data structures.

- Exploring Additional XPBD Constraints: Deformable objects often exhibit a diverse range of behaviors, and our current framework primarily focuses on simulating cloth and volume constraints. Future research could center on enhancing our simulation capabilities by incorporating additional XPBD constraints tailored to these types of objects. This exploration may involve the integration of new constraint types, such as bending constraints and Green's strain tensor to capture material flexibility, collision constraints to simulate interactions between objects, or attachment constraints to anchor specific parts of the model to others. These extensions will further diversify our simulation framework, enabling the representation of a wider variety of deformable objects and more variety of example scenes.

Bibliography

[1] Chest scene. https://www.cgtrader.com/free-3d-models/interior/other/chest-scene. (Accessed on 07/08/2023).

[2] Dog. https://sketchfab.com/3d-models/my-doge-a2779233edfd4d3cbd27b3ce8ef336fa. (Accessed on 07/08/2023).

[3] En175: Mechanics of solids - intro to fea. https://www.brown.edu/Departments/Engineering/Courses/En1750/Notes/FEA_Intro/FEA_Intro.htm. (Accessed on 07/09/2023).

[4] Jacques M. Bahi, Raphaël Couturier, and Lilia Ziane Khodja. Parallel gmres implementation for solving sparse linear systems on gpu clusters. In *Proceedings of the 19th High Performance Computing Symposia*, HPC '11, page 12–19, San Diego, CA, USA, 2011. Society for Computer Simulation International.

[5] Alan H. Barr. Global and local deformations of solid primitives. *ACM SIGGRAPH Computer Graphics*, 18(3):21–30, jan 1984. doi:10.1145/964965.808573.

[6] Richard H. Bartels, John C. Beatty, and Brian A. Barsky. *An Introduction to Splines for Use in Computer Graphics Geometric Modeling*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1987. ISBN: 0934613273.

[7] Jan Bender and Daniel Bayer. Parallel simulation of inextensible cloth. *VRIPHYS*, 8:47–56, 2008.

[8] Jan Bender, Kenny Erleben, and Jeff Trinkle. Interactive simulation of rigid body dynamics in computer graphics. *Computer Graphics Forum*, 33(1):246–270, dec 2013. doi:10.1111/cgf.12272.

[9] Jan Bender, Matthias Müller, Miguel A Otaduy, and Matthias Teschner. Position-based methods for the simulation of solid objects in computer graphics. In *Eurographics (State of the Art Reports)*, pages 1–22, 2013.

[10] Jan Bender, Matthias Müller, and Miles Macklin. Position-based simulation methods in computer graphics, Eurographics Tutorial 2015. doi:10.2312/EGT.20151045.

[11] Jan Bender, Matthias Müller, and Miles Macklin. A survey on position-based dynamics, Eurpgraphics Tutorial 2017. doi:10.2312/EGT.20171034.

[12] Daniel Brélaz. New methods to color the vertices of a graph. *Communications of the ACM*, 22(4):251–256, apr 1979. doi:10.1145/359094.359101.

[13] Robert Bridson, Ronald Fedkiw, and John Anderson. Robust treatment of collisions, contact and friction for cloth animation. *ACM Transactions on Graphics*, 21(3):594–603, jul 2002. doi:10.1145/566654.566623.

[14] Ozan Cetinaslan. Localized constraint based deformation framework for triangle meshes. *Entertainment Computing*, 26:78–87, may 2018. doi:10.1016/j.entcom.2018.02.001.

[15] Ozan Cetinaslan. Position-based simulation of elastic models on the GPU with energy aware gauss-seidel algorithm. *Computer Graphics Forum*, 38(8):41–52, nov 2019. doi:10.1111/cgf.13759.

[16] Ozan Cetinaslan. Parallel XPBD Simulation of Modified Morse Potential - an Alternative Spring Model. In Hank Childs and Steffen Frey, editors, *Eurographics Symposium on Parallel Graphics and Visualization*. The Eurographics Association, 2019. ISBN: 978-3-03868-079-6. doi:10.2312/pgv.20191108.

[17] Ozan Cetinaslan. ESPEFs: Exponential spring potential energy functions for simulating deformable objects. In *Motion, Interaction and Games*. ACM, nov 2021. doi:10.1145/3487983.3488303.

[18] Mathieu Desbrun, Peter Schröder, and Alan Barr. Interactive animation of structured deformable objects. In *Graphics Interface*, volume 99, page 10, 1999.

[19] Crispin Deul, Patrick Charrier, and Jan Bender. Position-based rigid-body dynamics. *Computer Animation and Virtual Worlds*, 27(2):103–112, 2016.

[20] R. Diziol, J. Bender, and D. Bayer. Robust real-time deformation of incompressible surface meshes. In *Proceedings of the 2011 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*. ACM, aug 2011. doi:10.1145/2019406.2019438.

[21] Yoo-joo Choi Min Hong Do-kyeong Lee, Tae-won Kim. Volumetric object modeling using internal shape preserving constraint in unity 3d. *Intelligent Automation & Soft Computing*, 32(3):1541–1556, 2022. ISSN: 2326-005X.

[22] James D. Foley, Andries van Dam, Steven K. Feiner, and John F. Hughes. *Computer Graphics: Principles and Practice (2nd Ed.)*. Addison-Wesley Longman Publishing Co., Inc., USA, 1990. ISBN: 0201121107.

[23] M. Fratarcangeli and F. Pellacini. Scalable partitioning for parallel position based dynamics. *Computer Graphics Forum*, 34(2):405–413, may 2015. doi:10.1111/cgf.12570.

[24] Marco Fratarcangeli. Position-based facial animation synthesis. *Computer Animation and Virtual Worlds*, 23(3-4):457–466, may 2012. doi:10.1002/cav.1450.

[25] Marco Fratarcangeli and Fabio Pellacini. Towards a massively parallel solver for position based dynamics. In *Proceedings of SIGRAD 2014, Visual Computing, June 12-13, 2014, Göteborg, Sweden*.

[26] Marco Fratarcangeli, Valentina Tibaldo, and Fabio Pellacini. Vivace: a practical gauss-seidel method for stable soft body dynamics. *ACM Transactions on Graphics*, 35(6):1–9, nov 2016. doi:10.1145/2980179.2982437.

[27] Marco Fratarcangeli, Huamin Wang, and Yin Yang. Parallel iterative solvers for real-time elastic deformations. In *SIGGRAPH Asia 2018 Courses*. ACM, dec 2018. doi:10.1145/3277644.3277779.

[28] Sarah Frisken Gibson and Brian Mirtich. A survey of deformable modeling in computer graphics. Tech. Rep. TR-97-19, Mitsubishi Electric Research Lab., Cambridge, MA, 1997.

[29] Thomas Jakobsen. Advanced character physics. *In Game Developers Conference Proceedings*, 01 2001.

[30] Doug James and Dinesh Pai. Artdefo: accurate real time deformable objects. In *Proceedings of the 26th annual conference on Computer graphics and interactive techniques - SIGGRAPH '99*, volume 72, pages 65–72. ACM Press, 01 1999. doi:10.1145/311535.311542.

[31] Mark T Jones and Paul E Plassmann. Scalable iterative solution of sparse linear systems. *Parallel Computing*, 20(5):753–773, may 1994. doi:10.1016/0167-8191(94)90004-3.

[32] Blazej Kubiak, Nico Pietroni, Fabio Ganovelli, and Marco Fratarcangeli. A robust method for real-time thread simulation. In *Proceedings of the 2007 ACM symposium on Virtual reality software and technology*. ACM, nov 2007. doi:10.1145/1315184.1315198.

[33] Randall J. LeVeque. *Finite Difference Methods for Ordinary and Partial Differential Equations*. Society for Industrial and Applied Mathematics, jan 2007. doi:10.1137/1.9780898717839.

[34] Tiantian Liu, Adam W. Bargteil, James F. O'Brien, and Ladislav Kavan. Fast simulation of mass-spring systems. *ACM Transactions on Graphics*, 32(6):209:1–7, November 2013. Proceedings of ACM SIGGRAPH Asia 2013, Hong Kong.

[35] Miles Macklin and Matthias Müller. Position based fluids. *ACM Transactions on Graphics*, 32(4):1–12, jul 2013. doi:10.1145/2461912.2461984.

[36] Miles Macklin, Matthias Müller, Nuttapong Chentanez, and Tae-Yong Kim. Unified particle physics for real-time applications. *ACM Transactions on Graphics*, 33(4):1–12, jul 2014. doi:10.1145/2601097.2601152.

[37] Miles Macklin, Matthias Müller, and Nuttapong Chentanez. XPBD: Position-based simulation of compliant constrained dynamics. In *Proceedings of the 9th International Conference on Motion in Games*. ACM, oct 2016. doi:10.1145/2994258.2994272.

[38] Matthias Müller, Nuttapong Chentanez, Tae-Yong Kim, and Miles Macklin. Strain based dynamics. In *Proceedings of the ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, SCA '14, page 149–157, Goslar, DEU, 2015. Eurographics Association.

[39] Matthias Müller. Hierarchical Position Based Dynamics. In Francois Faure and Matthias Teschner, editors, *Workshop in Virtual Reality Interactions and Physical Simulation "VRIPHYS" (2008)*. The Eurographics Association, 2008. ISBN: 978-3-905673-70-8. doi:10.2312/PE/vriphys/vriphys08/001-010.

[40] Matthias Müller, Bruno Heidelberger, Marcus Hennix, and John Ratcliff. Position based dynamics. *Journal of Visual Communication and Image Representation*, 18(2):109–118, apr 2007. doi:10.1016/j.jvcir.2007.01.005.

[41] Matthias Müller, Jos Stam, Doug James, and Nils Thürey. Real time physics. In *ACM SIGGRAPH 2008 classes*. ACM, aug 2008. doi:10.1145/1401132.1401245.

[42] Andrew Nealen, Matthias Müller, Richard Keiser, Eddy Boxerman, and Mark Carlson. Physically based deformable models in computer graphics. *Computer Graphics Forum*, 25 (4):809–836, dec 2006. doi:10.1111/j.1467-8659.2006.01000.x.

[43] James F. O'Brien and Jessica K. Hodgins. Graphical modeling and animation of brittle fracture. In *Proceedings of the 26th annual conference on Computer graphics and interactive techniques - SIGGRAPH '99*. ACM Press, 1999. doi:10.1145/311535.311550.

[44] Hartmut Prautzsch, Wolfgang Boehm, and Marco Paluszny. *Bézier and B-Spline Techniques*. Springer-Verlag, 01 2002. ISBN: 978-3-642-07842-2. doi:10.1007/978-3-662-04919-8.

[45] Nadine Abu Rumman and Marco Fratarcangeli. Position-based skinning for soft articulated characters. *Computer Graphics Forum*, 34(6):240–250, mar 2015. doi:10.1111/cgf.12533.

[46] Yousef Saad. *Iterative methods for sparse linear systems*. SIAM, jan 2003. doi:10.1137/1.9780898718003.

[47] Philip J Schneider. Nurb curves: a guide for the uninitiated. In Develop, the Apple Technical Journal, Issue 25, June 1996.

[48] Thomas W. Sederberg and Scott R. Parry. Free-form deformation of solid geometric models. *ACM SIGGRAPH Computer Graphics*, 20(4):151–160, aug 1986. doi:10.1145/15886.15903.

[49] Thomas W. Sederberg, David L. Cardon, G. Thomas Finnigan, Nicholas S. North, Jianmin Zheng, and Tom Lyche. T-spline simplification and local refinement. *ACM Transactions on Graphics*, 23(3):276–283, aug 2004. doi:10.1145/1015706.1015715.

[50] Martin Servin, Claude Lacoursière, and Niklas Melin. Interactive simulation of elastic deformable materials. *Proc. SIGRAD*, 01 2006.

[51] Eftychios Sifakis and Jernej Barbic. FEM simulation of 3d deformable solids: a practitioner's guide to theory, discretization and model reduction. In *ACM SIGGRAPH 2012 Courses*. ACM, aug 2012. doi:10.1145/2343483.2343501.

[52] J. Teran, S. Blemker, V. Ng Thow Hing, and R. Fedkiw. Finite volume methods for the simulation of skeletal muscle. In *Proceedings of the 2003 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, SCA '03, page 68–74, Goslar, DEU, 2003. Eurographics Association. ISBN: 1581136595.

[53] Demetri Terzopoulos, John Platt, Alan Barr, and Kurt Fleischer. Elastically deformable models. *ACM SIGGRAPH Conference*, 21(4):205–214, aug 1987. doi:10.1145/37402.37427.

[54] Matthias Teschner, Bruno Heidelberger, Matthias Müller, and Markus Gross. A versatile and robust model for geometrically complex deformable solids. In *Proceedings Computer Graphics International, 2004.*, volume 0, pages 312–319. IEEE, 07 2004. ISBN: 0-7695-2171-1. doi:10.1109/CGI.2004.1309227.

[55] Richard Tonge, Feodor Benevolenski, and Andrey Voroshilov. Mass splitting for jitter-free parallel rigid body simulation. *ACM Transactions on Graphics*, 31(4):1–8, jul 2012. doi:10.1145/2185520.2185601.

[56] Daniel Weber, Jan Bender, Markus Schnoes, André Stork, and Dieter Fellner. Efficient gpu data structures and methods to solve sparse linear systems in dynamics applications. In *Computer graphics forum*, volume 32, pages 16–26. Wiley Online Library, 2013.

[57] Zhiliang Xu. The jacobi and gauss-seidel iterative methods the jacobi method, Lecture in University of Notre Dame (Numerical Analysis) 2012. (Accessed on 06/23/2023).