CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

Fast, Interactive Soft-Body Animation

A Real-Time Physics Engine using Position-Based Dynamics

Bachelor's thesis in Applied Information Technology

Patrick Andersson
Pontus Eriksson
Adam Ingmansson
Love Westlund Gotby

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg

Bachelor's thesis 2017:05

Fast, Interactive Soft-Body Animation

A Real-Time Physics Engine using Position-Based Dynamics

Patrick Andersson
Pontus Eriksson
Adam Ingmansson
Love Westlund Gotby

UNIVERSITY OF
GOTHENBURG

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg

Fast, Interactive Soft-Body Animation
A Real-Time Physics Engine using Position-Based Dynamics
Patrick Andersson
Pontus Eriksson
Adam Ingmansson
Love Westlund Gotby

© Patrick Andersson, Pontus Eriksson, Adam Ingmansson, and Love Westlund Gotby, 2017.

Supervisor: Marco Fratarcangeli, Department of Computer Science and Engineering
Examiner: Daniel Sjölie, Department of Interaction Design and Technologies

Bachelor's Thesis 2017:05
Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

The Authors grants to Chalmers University of Technology and University of Gothenburg the non-exclusive right to publish the Work electronically and in a non-commercial purpose make it accessible on the Internet. The Authors warrants that they are the authors to the Work, and warrants that the Work does not contain text, pictures or other material that violates copyright law. The Authors shall, when transferring the rights of the Work to a third party (for example a publisher or a company), acknowledge the third party about this agreement. If the Authors has signed a copyright agreement with a third party regarding the Work, the Authors warrants hereby that they have obtained any necessary permission from this third party to let Chalmers University of Technology and University of Gothenburg store the Work electronically and make it accessible on the Internet.

Cover: A piece of swaying cloth consisting of 100×100 particles and 39402 constraints, running at 19 fps on an Intel i5-3570K at 3.8 GHz.

Typeset in LaTeX

Abstract

Physics simulations for realistic computer graphics is a subject that has been around for a long time. This paper covers the implementation of a framework called *Position-Based Dynamics* with the goal to simulate real-time soft-body dynamics in an interactive setting. The implemented simulator can simulate a number of different models, including cloth, in a number of different scenes. The paper also compares two techniques for solving linear systems, *Jacobi* and *Gauss-Seidel*, as well as looks at the advantages and problems with using parallelization with several CPU cores.

For a CPU implementation it is found that the octree performs better than the uniform grid. A parallel Gauss-Seidel which ignores order of computations is evaluated to show that the incurred overhead error is outweighed by the time gained from parallelization, at least for a low amount of parallel threads.

Keywords: Particle, Simulation, Physics, Position-Based Dynamics, Soft-body dynamics

Sammandrag

Fysiksimulationer för realistisk datorgrafik är ett ämne som funnits länge. Denna rapport går igenom implementationen av ett ramverk kallat *Position-Based Dynamics*, med målet att i realtid simulera fysiken för mjuka kroppar i en interaktiv miljö. Den implementerade simulatorn kan simulera en mängd olika modeller, inklusive tyg, i ett antal olika scener. Rapporten jämför också två olika tekniker för att lösa linjära ekvationssystem, *Jacobi* och *Gauss-Seidel*, samt diskuterar fördelarna och problemen med att använda parallellisering på flera CPU-kärnor.
För en CPU-implentation så finnes det att den så kallade octree metoden presterar bättre än den så kallade uniform grid metoden. En parallell implementering av Gauss-Seidel som inte tar hänsyn till ordningen i vilken beräkningar utförs, undersöks och visar att det pådragna extra felet som uppstår vägs upp av den kortare tid med vilken beräkningarna utförs, åtminstonde för ett lågt antal parallella trådar.

Acknowledgements

We want to thank our supervisor Marco Fratarcangeli for his support and suggestions, as well as answering questions during the project. Without you this project would not have come as far.
We also thank Jennifer Linder, Hampus Rönström, Johan Carlshede, Adam Kjellgren, Erik Karlkvist, and Olle Trens for their valuable feedback.
We thank Ludwig Friborg for letting us use his AMD Ryzen CPU.
We also want to thank our respective moms for their deep love and understanding when raising us. Without you we would not be where we are today.

The Bunny model is used courtesy of the Stanford Computer Graphics Laboratory.

Contents

List of Figures **viii**

1 Introduction **1**
1.1 Background . . . 1
1.2 Purpose . . . 3
1.3 Task . . . 3
1.4 Scope . . . 4
1.5 Structure of the report . . . 5

2 Method **6**
2.1 Planning phase . . . 6
2.2 Development phase . . . 6
2.2.1 Version control system . . . 6
2.2.2 Scrum . . . 7
2.2.3 Work flow . . . 7
2.2.4 Milestones . . . 8
2.3 Testing and results phase . . . 9

3 Theory and Technical Background **10**
3.1 Position-Based Dynamics . . . 10
3.1.1 Unified Particle Physics framework . . . 10
3.1.2 Constraints . . . 11
3.2 Solving the nonlinear system . . . 11
3.2.1 Derivation of particle update . . . 11
3.2.2 Jacobi . . . 12
3.2.3 Gauss-Seidel . . . 12
3.2.4 Successive over-relaxation . . . 13
3.2.5 Number of iterations . . . 13
3.2.6 Other methods . . . 13
3.3 Numerical integration . . . 14
3.3.1 Explicit and symplectic Euler integration . . . 14
3.3.2 Verlet integration . . . 14
3.4 Extended Position-Based Dynamics . . . 15
3.5 Languages, libraries and tools . . . 15
3.5.1 C++ . . . 16

3.5.2 Open Graphics Library (OpenGL) and GL Shading Language (GLSL) . . . 16
3.5.3 CMake . . . 16
3.5.4 Intel Threading Building Blocks . . . 16
3.5.5 GLM (OpenGL Mathematics) . . . 17
3.5.6 SDFGen . . . 17
3.5.7 Blender . . . 17
3.5.8 Glad . . . 17
3.5.9 GLFW . . . 17
3.5.10 ImGui (dear imgui) . . . 18
3.5.11 Dirent . . . 18

4 Implementation 19

4.1 Visually plausible versus physically accurate . . . 19
4.2 Application pipeline . . . 19
4.3 Particles and their representation . . . 20
4.4 Rendering . . . 21
4.4.1 Particles . . . 21
4.4.2 Mesh rendering . . . 22
4.5 Models . . . 23
4.6 Scenes . . . 24
4.7 Constraints . . . 24
4.7.1 Distance constraint . . . 24
4.7.2 Fixed point constraint . . . 25
4.7.3 Environment collision . . . 25
4.7.4 Particle collision . . . 25
4.7.5 Particle to particle friction . . . 25
4.7.6 Breakable constraints . . . 25
4.8 The solver . . . 26
4.8.1 Pre-stabilization pass . . . 27
4.9 Collision detection . . . 27
4.9.1 Triangle to particle collision . . . 28
4.9.2 Particle to particle collision . . . 29
4.9.3 Collision response . . . 29
4.10 Optimizing collision detection . . . 30
4.10.1 Uniform grid . . . 30
4.10.2 Octree . . . 31
4.10.3 k-d tree . . . 31
4.11 Parallelization . . . 31
4.11.1 Constraint solver . . . 32
4.11.2 Uniform grid . . . 32
4.12 Input handling and user interface . . . 33

5 Results 34

5.1 Collision handling . . . 34
5.1.1 Sequential and parallel grid . . . 35
5.2 Convergence speed . . . 35

5.3 XPBD . . . 36
5.4 Core workload balance . . . 36

6 Discussion **39**
6.1 Data structure for collision detection . . . 39
6.2 Parallelization . . . 39
6.2.1 Grid . . . 39
6.2.2 Residual . . . 40
6.3 Gauss-Seidel versus Jacobi . . . 40
6.4 Extended Position-Based Dynamics . . . 40
6.5 Project post-mortem . . . 41
6.5.1 What went right . . . 41
6.5.2 What went wrong . . . 41
6.5.3 What could have been done differently . . . 42
6.6 Future work . . . 42
6.6.1 GPU implementation . . . 42
6.6.2 Plastic deformation . . . 42
6.6.3 Optimization . . . 43
6.7 Implications of Position-Based Dynamics for society . . . 43

7 Conclusion **45**

Bibliography **46**

List of Figures

1.1 Demonstration of physics in the Source engine. Images ©Valve Corporation. 1
1.2 Physically accurate simulations are becoming more and more sophisticated, as can be seen both in the game and move industries. 2
1.3 A model of Saturn, including rings. 4

4.1 An overview of the steps the application goes through each frame. . . 20
4.2 A mesh rendered on top of a voxelized particle representation and two different particle representations. 23
4.3 Models from left to right: cone, cube with hole, cylinder, sphere. . . . 23
4.4 Scenes from left to right: axe, ledge, spike, stairs. 24
4.5 A torus being split in two. When the separation of two particles in a distance constraint reaches a given threshold, that constraint is removed from the simulation. The effect, as is seen in the image, is that of an object fragmenting. 26
4.6 Particle is penetrating the surface at the beginning of the time step. When projected to a valid position, too large of a distance has been covered causing the system to gain kinetic energy. Image by Macklin et al. [1]. 27
4.7 A visualization of the four steps of the collision detection. Step 1 checks the distance between the particle and the plane spanned by the triangle. Step 2 checks the distance between particle and every vertex. Step 3 checks if projected particle position onto plane is inside triangle. Step 4 checks the shortest distance between a line segment (edge) of the triangle and the particle position. 28
4.8 The performance graph used to gain qualitative assesments of how much time is spent in different parts of the program. 33

5.1 Time per frame for the different collision detection methods *brute-force*, *octree*, *grid*, and *parallel grid*. Results were measured using 8000 particles. 34
5.2 Time per frame using the uniform grid for collision detection. The sequential and parallel implementation is compared with the number of particles ranging from 500 to 20 000. Note how the difference increases with the number of particles when going from sequential to parallel. 35

5.3 Comparison of the total constraint residual using Gauss-Seidel and averaged Jacobi. A cloth model with 50×50 particles and 9702 constraints was stretched to 16 times its original size and then released to let the distance constraints have it pull itself together. Notice that Gauss-Seidel has converged after about 1000 iterations while averaged Jacobi has not converged. 36
5.4 Comparison of the total constraint residual using parallelized Gauss-Seidel, running on one core/one thread, two cores/two threads, four cores/four threads, and four cores/eight threads. A cloth model with 100×100 particles and 39 402 constraints was stretched to 16 times its original size and then released to let the distance constraints have it pull itself together. Notice that a more parallelized solver converges faster in time, but that the sequential implementation has a better convergence rate per iteration. 37
5.5 Comparison of normal PBD and XPBD, where PBD is top row and XPBD bottom row. Model used is a 64×64 cloth with a time step of 0.16 ms. The different images correspond to an iteration count of 20, 40, 80, and 160 iterations, from left to right. The α value used for XPBD is 10^{-6} and the stiffness value used for regular PBD is 0.9. Notice how PBD gets more stiff for higher iteration counts, and how that effect is reduced with XPBD. 37
5.6 An AMD Ryzen 5 1600 Hexacore processor under heavy load from our application, showing the effectiveness of Intel TBB's load balancer. *Note: Text under 'CPU' translates to "Load in % over 60 seconds"* . 38

6.1 Simple model for how to handle both plastic deformation and material capable of breaking with distance constraints. If the particle distance is within the plastic deformation region, l_0, the rest length value of the distance constraint is updated. 43

1

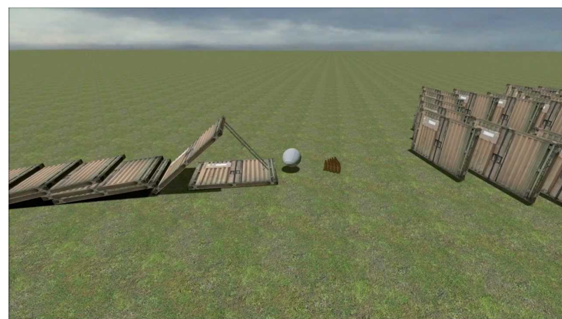
Introduction

1.1 Background

Ever since computer generated graphics were first shown on computer screens, there has been a striving for a more realistic look. At the same time, because of the nature of interactive applications, speed is of utmost importance. Up until a couple of years ago, real-time applications like games seldom used realistic physics models, for example soft-body dynamics, because of the immense computational power required. Instead, developers tried to estimate physics with rough approximations: focus was on speed and 'visual plausibility' instead of physically correct simulations.

One of the first successful game engines that used accurate physics as part of the game mechanics was the *Source* engine [2] developed by Valve Corporation. This engine can be used for physics-based games, where objects can be picked up and thrown, giving them momentum which can be transferred to other objects upon collision, as seen in Figure 1.1 where a 'Gravity gun' is used to pick up an oil drum in the game *Half-Life 2*, as well as a domino set created in *Garry's Mod* [3].

(a) Gravity gun in Half Life 2, lets the wielder pick up objects.



(b) Player-created domino set in Garry's Mod.

Figure 1.1: Demonstration of physics in the Source engine.
Images ©Valve Corporation.

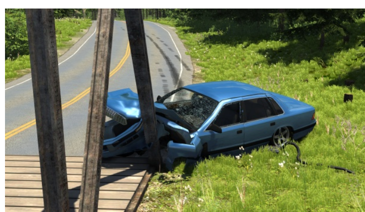
This basic type of physics simulation where objects with momentum can affect other objects, causing them to fly away or be knocked over, opens up for some interesting possibilities in regards to game mechanics. Even so, when objects start moving and interacting with each other, it is important that the computations are fast and the movements displayed are physically plausible. It is also important for the simulations to be stable: one box resting on another should keep still, not appear to be compensating for the upper box trying to 'fall through' the bottom one. Rigid body simulations are the easiest and fastest to calculate and have for this reason been the most common simulations used for a long time.

As computer engineering is still evolving, it is now becoming a more realistic idea to implement simplified real-time versions of already existing pre-rendered (frames rendered without time constraint, hereafter referred to as 'offline rendering') simulations. Two examples are Eulerian physics, which models the world as a grid with internal forces, and Lagrangian particle-based mechanics, which models objects as a collection of particles. This opens up for a world of possibilities, including (but not limited to) deformable objects, physically more accurate collisions, and accurate simulations of cloth and hair. One example of this is the vehicle simulator *BeamNG* [4] which uses soft-body dynamics in real-time to simulate damage done to cars (Figure 1.2a).

One aspect of making graphics look real are materials that look and behave physically correct. It is generally easier to simulate physically correct materials in a specialized (focusing on one type of material) and/or offline manner. Today, offline rendering can produce a very realistic look and because of this, it is often used in the film industry. One example of this is the snow in the movie *Frozen*, which was entirely simulated [5] (Figure 1.2b).

On the other hand, in situations where you do not have several minutes or hours to render each frame, for instance in a video game, offline techniques will not suffice. In these instances one usually only have $1/60^{\text{th}}$ of a second, $16.\overline{6}$ ms, to calculate the simulation and render the scene.

Even though computers grow faster and faster, the increase in computational power



(a) BeamNG features soft-body mechanics for deformable cars.

(b) The snow in Frozen was simulated with a novel snow simulation method. Images ©Disney.

Figure 1.2: Physically accurate simulations are becoming more and more sophisticated, as can be seen both in the game and move industries.

has started to even out and focus have shifted to increasing the number of cores. This makes proper usage of data structures and parallelism an even more important task than before, which naturally fits the nature of particle-based physics. This type of physics is very flexible in the way that the precision of the simulation can be scaled by varying the number of particles used. A simulation with a low-performance computer of the future can, in comparison to a low-performance computer of today, be used with a higher resolution of the models (i.e. smaller and more particles), for a more physically accurate simulation.

To have an interactive physics engine, it is essential to simulate the physics in real-time. These simulations are useful for a range of applications, e.g. video games and pedagogic tools. One of the current methods of achieving real-time physical simulations is presented in the paper *Unified Particle Physics for Real-Time Applications* [1] which is based on the method of *Position-Based Dynamics* [6]. The ideas and methods presented in these two papers is the foundation for the project presented in this thesis.

1.2 Purpose

The purpose of this report is not only to explain the methods and implementations required for an interactive particle-based physics engine based on the method of *Position-Based Dynamics*, but also to compare the different approaches and algorithms needed for such an implementation.

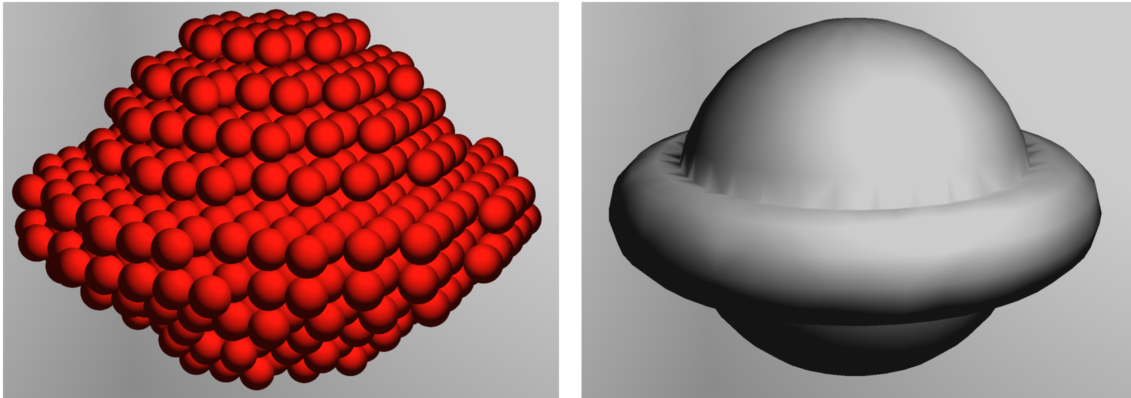
1.3 Task

The task assigned is to create a cross-platform, fast and interactive physics engine from scratch, able to simulate physics with the position-based dynamics approach and render the result in real-time. Because of the computationally heavy nature of such an application, focus is on accuracy, robustness, performance, and controllability.

The project is composed of several modules:

- discretization (voxelization) of the objects in grids of particles, in C++;
- numerical integration of the particle dynamics, in C++;
- visualization of and interaction with the simulated objects written in OpenGL and C++.

The engine is written in C++ and makes use of optimized algorithms and parallel computing on a multicore CPU in order to achieve real-time performance (10-60



(a) Voxelized particle representation of Saturn.
(b) Rendering of Saturn's surface mesh.

Figure 1.3: A model of Saturn, including rings.

FPS).

Additional features

These improvements beyond the initial plan were implemented:

- breakable constraints
- multiple collision detection methods
- XPBD
- a performance graph
- different scenes
- generated models for cloth and box
- parallelization of construction of data structure
- parallelization of the constraint solver

1.4 Scope

The scope of the project is to implement voxelization of arbitrary triangle meshes coupled with an iterative linear solver for solving a constrained particle system. The implementation makes use of an algorithm capable of finding the nearest neighboring particles for collision detection and a shader for displaying the particles. Additional

implementations are particle and constraint parameters, such as to be able to simulate different types of objects.

The main core of features that are to be present in the completed prototype is soft bodies and collisions, with which the user can interact. The simulation is expected to handle up to 100 000 particles and have support for rendering a mesh model associated with a particle representation.

The project also include dynamic tuning of the program in real-time, letting the user explore different settings for both the objects and their constraints.

1.5 Structure of the report

Chapter 2 will go into detail on how the project was organized and how the task itself was divided into different phases with different milestones. Chapter 3 will cover the technical background and the theory behind the techniques used in the project, as well as the language, library, and tools that are used for the project. This will give the reader an insight into how the theory relates to the implementation. Chapter 4 is a detailed rendition of how the theory was implemented and what algorithms were used to combat the different problems presented by the task. In chapter 5 the reader is presented with information on how the final version of the application performs as well as a comparison between different techniques and algorithms. In chapter 6 the authors give their view on how the project turned out, some learned lessons, and a reflection on the results gained. The last chapter will provide a brief summary of the report.

2

Method

In an initial planning phase, the project was divided into three major phases: planning, development, and testing and results. The development phase was furthermore divided into several stages: a pre-prototype, a working sequential prototype and a parallel and optimized prototype. This chapter is aiming to explain the goals of these different phases.

2.1 Planning phase

The first part of the project focused on planning how it was to be executed, writing a planning report, and putting many hours into reading previous publications on the subject. The project was reasonably well defined from the start, so research was begun immediately. A time plan was set up with several milestones for the project, which allowed the group focus on the most important part at every stage of the development.

2.2 Development phase

During this phase the group used an iterative approach to building the application from scratch, adding more and more functionality according to the weekly meetings. Initially, focus was on implementing as much as seemed reasonable, while later focus moved to refactoring for a better software architecture and optimizing the code.

2.2.1 Version control system

Any modern software development project benefits from using a version control system. For any project that is not a solo project, the benefits are so large that it can be considered mandatory. This project uses a version control system called *Git* [7]. *Git* was used in conjunction with *GitHub* [8] as a central repository. The project

adopted a git branching model called ”Git flow” [9]. The purpose of this model is to facilitate the collaboration by having a centralized repository, while still allowing localized collaboration between peers. Three main branch types have been used in this project. The first one is the *master* branch and is considered to always be in a release-ready state. This has seen very little use as the project is mainly concerned with the development of a prototype. The second branch is *develop* and contains a buildable version of various completed features. The last branch type is *feature* and it is a branch where a single feature is currently being developed.

2.2.2 Scrum

As this was primarily a software development project, the group utilized an agile working method based on *Scrum* [10]. Work was delegated in a weekly meeting with the intent on efficiently utilizing each group members knowledge and skills. For each task, a time estimation was done to facilitate the distribution of tasks among the members of the group. The same meeting also served as to reflect and discuss on the work done during the previous week. No Scrum master was assigned as the group only had four members and did not feel this was a necessity.

During the meeting, the past week was reviewed with discussions about what had been done, what had not been done, why, and what was prioritized to be done the coming week. During the weeks the group kept communication via a service called *Slack* [11], a chat tool designed for project groups with the possibility to have different channels for different topics. This made it possible to communicate and discuss separate problems at the same time, without mixing the discussions with each other.

Combining weekly meetings with efficient online communication made it possible to both identify problems and potential time sinks quickly, while also keeping all group members up-to-date on all parts of the project.

2.2.3 Work flow

Upcoming tasks, bugs, and other problems that were delegated during the weekly meeting were, when needed, summarized and put into an *Issue* on *GitHub*. This let the group have a list of things that were being worked on at the moment and at the same time serving as the base for a branch name in *Git*. During the week, work was done mostly individually except for when discussions about problems and solutions were needed. When a feature is finished a pull request is created on *GitHub*, which was reviewed at the next weekly meeting and merged if deemed completed.

2.2.4 Milestones

The development was split into three different prototypes to allow having a working product early on, which was improved upon incrementally. This seemed like the obvious approach since the group's supervisor advised to have a fully working sequential program before moving on to a parallel, and optimized, implementation.

Pre-prototype

The first milestone was to have a foundation for the simulations working, with no advanced algorithms implemented but rather a 'square one' that was to serve as a base for further development and discussion. This version features

- a simple forward Euler simulation with gravity;
- no interaction between particles or between particles and the scene;
- particles rendered as simple points without any kind of shading;
- simple wire-frame box signifying the scene;
- movement controls to move the camera and look around

Working sequential prototype

The next goal was to have a fully working prototype with all major parts of the simulation implemented. This was to avoid accumulating bugs and/or other problems, and not having to consider further improvements and optimizations faulty when in fact the program was faulty from the beginning. By knowing that the code works properly before focusing on performance it is easier to rule out potential causes of unwanted behavior.

Parallel and optimized prototype

For the final development phase of the software, focus was put into increasing the performance as much as possible. This was done by parallelizing and optimizing the computations wherever possible. Note that at this stage the product is not supposed to be extended anymore, but rather improved without adding new features. Two major areas were optimized, where the first was by introducing better data structures for collision detection and the second by parallelization of collision detection, constraint solving, and construction of the data structures.

2.3 Testing and results phase

The final phase of the project included taking several measurements. These were measurements for:

- a comparison between two linear iterative solver algorithms for *Position-Based Dynamics*;
- a comparison between sequential and parallel implementations;
  - Vary the number of cores, single up to quad core (4 CPUs)
  - See core utilization
  - See the effect of parallelization at different particle counts
- a comparison between the collision detection algorithms of two different data structures.

All measurements regarding a single comparison were taken on the same computer without restart to ensure that all measurements were taken in the same environment.

3

Theory and Technical Background

This chapter aims at providing the reader with an overview of previous work and research in related areas. Here the libraries, technical structures, and techniques used in this project will be briefly explained while the implementations of them and more in-depth details will be covered in chapter 4, Implementation.

3.1 Position-Based Dynamics

In 2007 Müller et al. proposed a framework for real-time physics using a point-based approach which quickly became popular for its unconditional stability and high performance [6]. Today, the Position-Based Dynamics (PBD) method is used in many applications and games, for example *The Witcher 3* [12], *CryEngine 3* [13], *Unreal Engine* [14] as well as Nvidia's *PhysX* engine [15].

3.1.1 Unified Particle Physics framework

A major selling point of PBD is that it can be used in creating a framework for simulating multiple types of physical objects such as deformable solids, rigid bodies, gases, liquids, and cloth, all interacting with each other as demonstrated by Macklin et al. [1]. Note that even rigid bodies can be modeled with particles, which effectively reduces the number of different types of collision handling needed.

Furthermore, the paper by Macklin et al. proposes a parallel solver, in comparison to the original sequential solver in *Position-Based Dynamics* [6]. This plus some other minor improvements such as better friction and a pre-stabilization collision loop makes it an even better solution for particle-based physics.

3.1.2 Constraints

A constraint is a mathematical function from the set of particle positions to the real numbers. As described in *Position-Based Dynamics* [6]:

> The constraint j [..] consists of
>
> - a cardinality n_j,
> - a function $C_j : \mathbb{R}^{3n_j} \to \mathbb{R}$,
> - a set of indices $\{i_1,...i_{n_j}\}, i_k \in [1,...N]$,
> - a stiffness parameter $k_j \in [0...1]$ and
> - a type of either *equality* or *inequality*.
>
> Constraint j with type *equality* is satisfied if $C_j(x_{i_1},...,x_{i_{n_j}}) = 0$. If its type is *inequality* then it is satisfied if $C_j(x_{i_1},...,x_{i_{n_j}}) \geq 0$.

Using this structure a wide range of constraints can be created, two examples being distance constraints and bending constraints. With these properties, it is possible to model different kinds of materials and objects.

3.2 Solving the nonlinear system

The system produced by the PBD method is a large, sparse, nonlinear equation system. To solve this system in real-time, two different methods are used. First, the system is linearized around the current positions of the particles. Secondly, an iterative linear solver is used to solve the linearized system. In short, these two steps correspond to the Newton–Raphson method [6]. The solution will contain errors because the system being solved is linearized. To get a better result, the system is once again linearized and solved, but this time with the new positions as calculated from the first iteration.

3.2.1 Derivation of particle update

A quick derivation of how the particle delta is calculated is given here to give the reader better insight. For a complete version see [1, 6, 16]. The initial, nonlinear, system can be represented as

$$C(\mathbf{p}) = 0 \tag{3.1}$$

where $\mathbf{p}$ is the vertical concatenation of all particle positions and $C(\mathbf{p}) : \mathbb{R}^n \to \mathbb{R}^M$ is the vertical concatenation of all constraint functions. Here, n is the number of

particles and M the number of constraints. This system is linearized around the current particle positions. Solving for the position delta yields

$$C(\mathbf{p} + \Delta\mathbf{p}) \approx C(\mathbf{p}) + \nabla C(\mathbf{p})\Delta\mathbf{p} = 0 \tag{3.2}$$

$$\Delta\mathbf{p} = -\frac{C(\mathbf{p})}{\nabla C(\mathbf{p})} \tag{3.3}$$

which is equivalent to the Newton-Raphson method. The position delta is then restricted to lie along the gradient of the constraint function and weighted according to the inverse of the mass matrix $\mathbf{M}$ which has the particle masses on its diagonal:

$$\Delta\mathbf{p} = \mathbf{M}^{-1}\nabla C(\mathbf{p})\boldsymbol{\lambda}. \tag{3.4}$$

Putting all pieces together, solving for $\boldsymbol{\lambda}$ by way of Equation (3.3) and (3.4), and then replacing $\boldsymbol{\lambda}$ in Equation (3.4) altogether gets the final expression for the particle delta

$$\Delta\mathbf{p} = -\frac{\mathbf{M}^{-1}C(\mathbf{p})\nabla C(\mathbf{p})^T}{\nabla C(\mathbf{p})\mathbf{M}^{-1}\nabla C(\mathbf{p})^T}. \tag{3.5}$$

Something to take into consideration is the fact that some of the constraint are not equalities but rather inequalities. To accommodate this fact, should a given constraint evaluate greater than zero, its position update is zero. This linear system is going to be solved one constraint at a time and to this effect a stationary iterative method is used.

3.2.2 Jacobi

The first and most simple stationary iterative method to consider is the Jacobi method [17, 18], but this is generally not used due to there being more powerful methods such as Gauss-Seidel [19]. Applied to Equation (3.5), the Jacobi method amounts to calculating the first position delta, then the second and so on. When all deltas have been found, the particle positions are updated. For *Position-Based Dynamics* it is also prone to being rank-deficient (which can be alleviated by under-relaxation, for example by the current number of bound constraints [1] which is referred to as *averaged Jacobi*), but it does have one advantage; parallelizing this method is trivial, as each new update is independent of all the other updates.

3.2.3 Gauss-Seidel

The Gauss-Seidel method [17, 18] is a natural extension to the Jacobi method. By realizing that the previously computed position deltas can be applied to the particles positions directly, a faster convergence rate can be achieved. Effects can propagate further in the system for each iteration. The drawback of this is that the updates are now dependant of the order in which they are calculated. It is

however still possible to leverage parallelism out of a system with this method. By synchronization mechanisms or, for example, by coloring it to find particles which are not directly connected [19], one can proceed in a manner which does not give unpredictable results.

3.2.4 Successive over-relaxation

Successive over-relaxation [17, 18], or simply SOR, is usually considered a stationary iterative method of its own. For the purpose of this project, using SOR means that each computed position delta is multiplied by a scalar value commonly denoted ω. Notable is the fact that when the scalar value is one, this method reduces to Gauss-Seidel. Most of the time a value greater than one is used in order to reach faster convergence by moving particles further along each calculated change in position.

3.2.5 Number of iterations

For each frame, the solving step is run a multiple number of times to reach a better result. For deciding how many iterations to run there are two different criteria to choose from: either there is no time for more steps in order to fit a given time frame, or the total residual has reached a given level which is considered acceptable.

3.2.6 Other methods

While outside the scope of the project, it is worth mentioning some alternatives to the stationary iterative methods. One type of method is Krylov subspace methods, which have a large body of work for linear sparse systems. Different well known methods include preconditioned conjugate gradient [20], generalized minimum residual [21], and biconjugate gradient stabilized [17, 18]. Müller successfully incorporates the multigrid method into the existing PBD framework with good results [22]. Wang uses the Chebyshev semi-iterative method and report good results when combined with Jacobi [23], even though it is worth noting that Chebyshev's method is, depending on the time frame, not always fast enough and should therefore mostly be considered when computational time is not an issue [24].

3.3 Numerical integration

There are several methods to chose from to do the numerical integration. A few of these are the explicit and symplectic Euler, and Verlet integration, which are described below.

3.3.1 Explicit and symplectic Euler integration

Explicit Euler is a basic numerical integration method. When used in *Position-Based Dynamics*, the positions and velocities of each particle are updated in every step as

$$\mathbf{v}(t+\Delta t) = \mathbf{v}(t) + \frac{\mathbf{f}(\mathbf{x}(t),\, \mathbf{v}(t))}{m}\Delta t \tag{3.6}$$

$$\mathbf{x}(t+\Delta t) = \mathbf{x}(t) + \mathbf{v}(t)\Delta t, \tag{3.7}$$

where $\mathbf{v}$ is the velocity and $\mathbf{x}$ is the position of a particle, $\mathbf{f}$ is the external forces acting on the particle at the time t, m is the particles mass, and Δt is the iteration time step.

The symplectic Euler was the most popular method for *Position-Based Dynamics* in 2015 [25]. This method conserves energy better than the explicit one and therefore gives a better result [26]. Going from explicit to symplectic means that the velocity from the next step is used when calculating the position of the next step, instead of the velocity from the current step, such that (3.7) becomes

$$\mathbf{x}(t+\Delta t) = \mathbf{x}(t) + \mathbf{v}(t+\Delta t)\Delta t. \tag{3.8}$$

Explicit and symplectic Euler are both first order accurate integration schemes, meaning that if the time step is reduced by half, the error is reduced by half as well.

3.3.2 Verlet integration

Verlet integration is another popular numerical integrator. The idea is to use the position of the previous time step, i.e. to use where the particle was the step before the current step to better predict where it will be in the next. This is done using the Taylor expansion of $\mathbf{x}(t)$ in the $+\Delta t$ and $-\Delta t$ directions to the third order as

$$\mathbf{x}(t+\Delta t) = \mathbf{x}(t) + \dot{\mathbf{x}}(t)\Delta t + \frac{\ddot{\mathbf{x}}(t)\Delta t^2}{2} + \frac{\dddot{\mathbf{x}}(t)\Delta t^3}{6} + \mathcal{O}\left(\Delta t^4\right) \tag{3.9}$$

$$\mathbf{x}(t-\Delta t) = \mathbf{x}(t) - \dot{\mathbf{x}}(t)\Delta t + \frac{\ddot{\mathbf{x}}(t)\Delta t^2}{2} - \frac{\dddot{\mathbf{x}}(t)\Delta t^3}{6} + \mathcal{O}\left(\Delta t^4\right). \tag{3.10}$$

Adding (3.9) to (3.10) and solving for $\mathbf{x}(t + \Delta t)$ yields

$$\mathbf{x}(t + \Delta t) = 2\mathbf{x}(t) - \mathbf{x}(t - \Delta t) + \frac{\ddot{\mathbf{x}}(t)\Delta t^2}{2} + \mathcal{O}\left(\Delta t^4\right) \tag{3.11}$$

$$= \mathbf{x}(t) + \underbrace{\mathbf{x}(t) - \mathbf{x}(t - \Delta t)}_{\text{cf. velocity}} + \frac{\mathbf{f}(t)\Delta t^2}{m} + \mathcal{O}\left(\Delta t^4\right). \tag{3.12}$$

This means that Verlet integration is a fourth order accurate integration scheme, i.e. if the time step is reduced to half, the error is reduced to one sixteenth.

3.4 Extended Position-Based Dynamics

Position-based dynamics exhibit different behavior depending on the value of the time step and the number of iterations used in the solver for each frame. The most noticeable feature is that the stiffness of objects increases with decreasing time step or increasing number of solver iterations. To combat this Macklin et al. [27] present an extended version of Position-Based Dynamics (called XPBD) in which both time step and iteration count are taken into consideration when solving constraints.

The updated algorithm works by storing an extra variable for each particle, a constraint multiplier, which is updated each solver iteration. This multiplier serves to introduce both time step and current iteration number into the position update. The new position update is calculated as

$$\Delta\boldsymbol{\lambda} = -\frac{\mathbf{C}(\mathbf{x}) + \tilde{\boldsymbol{\alpha}}\boldsymbol{\lambda}}{\nabla\mathbf{C}(\mathbf{x})\mathbf{M}^{-1}\nabla\mathbf{C}(\mathbf{x})^T + \tilde{\boldsymbol{\alpha}}} \tag{3.13}$$

$$\Delta\mathbf{x} = \mathbf{M}^{-1}\nabla\mathbf{C}(\mathbf{x})^T\Delta\boldsymbol{\lambda} \tag{3.14}$$

where $\tilde{\boldsymbol{\alpha}} = \frac{\boldsymbol{\alpha}}{(\Delta t)^2}$ and $\boldsymbol{\alpha}$ are block diagonal compliance matrices corresponding to inverse stiffness. The $\boldsymbol{\lambda}$ that shows up in the calculation of $\Delta\boldsymbol{\lambda}$ is the total sum of all $\Delta\boldsymbol{\lambda}$ up until this point in the current frame. $\boldsymbol{\lambda}$ is initialized as zero at the beginning of each frame.

3.5 Languages, libraries and tools

Most modern complex application uses external tools and libraries and this application is no exception. The application is written in C++, where the graphics implementation is done using OpenGL. A handful of libraries are used, most notably Intel's Threading Building Blocks [28] for easy parallelization. This section aims at briefly explaining and motivating these choices.

3.5.1 C++

C++[29] is well known for its efficiency and performance, since this is a language where code can be written at a relatively high abstraction level but still with a very low (or often even *without* any) performance overhead [30]. For a problem where computation time is a very important factor, this is an extremely coveted property.

3.5.2 Open Graphics Library (OpenGL) and GL Shading Language (GLSL)

OpenGL is an application programming interface (API) for rendering 3D and 2D vector graphics [31], usually with the help of a GPU. It is a cross-language and cross-platform interface and, with different group members using different platforms, is the only viable option. OpenGL uses a shader language that is called GLSL, which has a syntax similar to C.

3.5.3 CMake

When building a cross-platform application it is best not to make any assumptions about what compiler, platform, or integrated developer environment the user has. Therefore, an abstraction of the software build process is needed. For this project, CMake [32] is used. This allows generation of native makefiles and workspaces for most compiler environments, such as Unix makefiles and Visual Studio projects. CMake uses compiler independent configuration files which contain information such as dependency paths, dependency configuration, and linking information.

3.5.4 Intel Threading Building Blocks

Particle-based physics is an exceptional problem for parallelizing computations since a lot of computations need to be done for every particle, every frame. Intel Threading Building Blocks [28] (hereafter referred to as TBB) is a library developed by Intel for easy implementations of parallelized computations. TBB suits this project well since particle-based physics is an exceptional problem for parallelization, TBB works well in conjunction with object oriented implementations and user-defined types and has native support for nested parallelism [33], which suits the nature of this project. Furthermore, according to the comparison in ”Benchmarking Usability and Performance of Multicore Languages” [34] from S. Nanz et al., TBB had the quickest coding time while still being one of the two fastest libraries. The motivation for using TBB over other similar libraries is that it is easy to implement, robust, and has a balanced scheduler which handles thread spawning, scheduling, and determining how much work a given thread should do, automatically.

3.5.5 GLM (OpenGL Mathematics)

GLM [35] is a mathematics library designed for computer graphics and based on the GLSL specification, which makes it both suitable and easy to use for anyone who already knows GLSL. It includes many vector-based operations, such as addition, subtraction, dot product, cross product and many more.

3.5.6 SDFGen

SDFGen [36] is a tool to generate a signed distance field from a triangle mesh. The program generates a file with the signed distance value sampled over a three-dimensional grid whose size is the minimum bounding box for the mesh. These values are negative when a point is inside of the mesh. Thus a particle is added for every point that has a negative or small enough value. The file does not contain any data about the material, such as constraints, so these are added separately by the application.

3.5.7 Blender

Blender [37] is an open source graphics and 3D modeling software used for creating anything from images to video games. The main use of Blender in this project was to create models for use inside the application. These models, or triangle meshes, are mostly simple 3D geometrical objects and by using SDFGen particle representations were created from these models. Additionally, the actual scenes where the simulation is taking place are constructed inside Blender and then exported as .obj files, which were then loaded into the program.

3.5.8 Glad

Glad [38] is a loader for OpenGL. It exposes OpenGL functionality to the developer and abstracts away the hardware. There are a couple of different loaders available such as GLEW, GL3W, glLoadGen, glSDK, and glbinding [39], but as long as a cross-platform loader is chosen, the rest of the application will not be affected. These loaders are very similar and glad was chosen as the group had prior experience with it.

3.5.9 GLFW

For window creation and handling the GLFW [40] library was chosen. Considered alternatives were glut [41] and sdl2 [42] which are all cross-platform libraries.

3.5.10 ImGui (dear imgui)

ImGui [43] is a minimalistic graphics library for C++, focusing on speed and performance over having many features. It fits well into the project as it focuses not on creating end-user GUIs, but rather debug and visualization for the developers.

3.5.11 Dirent

To deliver cross-platform interaction with folders and files, the program makes use of the *Dirent* [44] interface when building the application for Windows.

4

Implementation

This chapter aims at describing the actual execution of the project by first discussing the application architecture on a high abstraction level, as well as giving a brief overview of how particles and objects are modeled, before moving on to the more technical details including implementations of algorithms and structure of the code.

4.1 Visually plausible versus physically accurate

Due to the nature of computer graphics, the result is in most cases not required to be an exact replication of reality. In fact, to maximize performance and be able to run the application in real-time, some approximations or conscious ”mistakes” in the calculations are acceptable if this saves a considerable amount of computational power. For example, when doing particle to particle collision detection, there is no guarantee that no collisions will be missed. In this case, it is considered acceptable since this would take too long.

4.2 Application pipeline

On a very high abstraction level, the simulation consists of two boxes. The first handles physics by calculating the collisions and constraints. This output data is fed into the box for the graphics, which renders the scene to the screen and also takes input from the user, which it feeds back into the physics box.

The application consists mainly of a display loop, which runs as long as the window is open, where every iteration calls a display function. Input events are handled every iteration before the display call, along with the general GUI set up. The display function handles both physics simulation and the rendering. Every display call is one frame and consists of a number of steps, which can also be seen in Figure 4.1:

1. Render scene
2. Physics step:
    (a) Remove constraints
    (b) Detect collisions (Parallel step)
        i. Build structure
        ii. Find collisions
    (c) Add collision constraints
    (d) Resolve collisions
    (e) Resolve constraints (Parallel step)
3. Render Particles/Model
4. Render UI

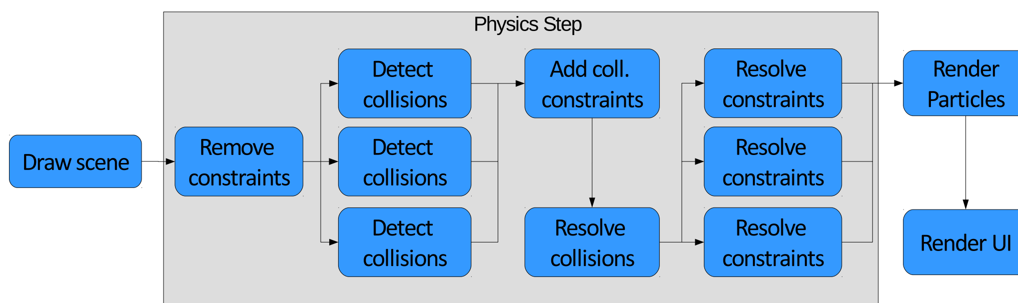
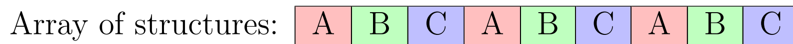


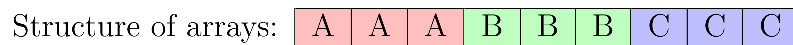
Figure 4.1: An overview of the steps the application goes through each frame.

4.3 Particles and their representation

Representing geometry using particles is an adaptable approach since all kinds of geometry can be modeled and, depending on the material, the particles can be assigned different properties and couplings. An additional benefit of this is that collision computations become more generalized as there is only one primitive object, spheres. A downside, on the other hand, is that some geometry, e.g. planes, require a lot of particles in order to be modeled at a given resolution. Therefore, deformable, interactive objects are modeled with particles. For static objects such as the scene, triangle meshes are used. Each particle is represented by some attributes, such as position. Because the application handles a very large number of particles, the implementation of how the particles are stored becomes very important.

The architecture of a program can vary in many ways, one of them being how to handle arrays of objects, such as particles. One way is to have an array of structures, which in this case would mean that each particle structure element would hold information about that particle e.g. position, velocity *et cetera*, and have one single array for all particles. The other way is to have a structure of arrays, which means that for each attribute of the structure there is an array. The difference in how the attributes are stored in memory is visualized below where A, B, and C are some variable properties:

Array of structures: 

Structure of arrays: 

The way in which particles are stored in this project is in the form of a structure of arrays. The reason for this is that most of the time only one attribute will be accessed at a time, giving a SoA a higher cache hit rate [45].

```
struct ParticleData
{
    std::vector< glm::vec3 >  position;
    std::vector< glm::vec3 >  predictedPosition;
    std::vector< glm::vec3 >  velocity;
    std::vector< int >   phase;
    std::vector< float > invmass;
    std::vector< float > radius;
}
```

The displayed model also include particle radius and the phase parameter to allow for selective collision detection.

4.4 Rendering

The rendering is done using two main shader programs. The first is used to display objects as particles connected to each other. The second shader is used to render the surfaces of the scene in which the simulation takes place, and also to render the objects in the simulation with surfaces instead of just as particles.

4.4.1 Particles

For the particle representation view, the particles are rendered as 'GL_POINT' with some simple, yet effectful, uses of the fragment shader to give them a 3D-look.

```
// Get fragments 2D-coordinate on point in [-1, 1]
vec2 pointCoord = vec2(2 * (gl_PointCoord.x - 0.5), 2*(-gl_PointCoord.y + 0.5));
// Get distance from middle of point to fragment
float length = length(pointCoord);
// Keep the particle round
if (length > 1.0) discard;
// Get normal of fragment on the imagined sphere
vec3 normal = normalize(vec3(pointCoord, cos(length * M_PI / 2)));
// Get fragment position in viewspace
vec3 fragPos = centerPos + particleSizeOut * vec3(normal.x,-normal.y,-normal.z);
// Calculate direct diffuse lighting
float diffuse = max(0.0, dot(normalize(viewSpaceLightPos - fragPos), normal));

outColor = vec4((0.2 + diffuse) * color, 1.0);
```

After this, a diffuse lighting is calculated, which does not affect performance but significantly improves the look of the particles.

4.4.2 Mesh rendering

When dealing with computer graphics it is usually not an acceptable alternative to render the particles as is. Instead, it is desirable to see the objects with their surfaces as described by the triangle meshes. To do this there were two alternatives: surface scattering [46] or mapping the positions of the object's triangle mesh to the particle representation using barycentric coordinates [47]. Surface scattering is a technique where discs are rendered on the positions of the surface particles. If the discs are large enough, they will cover the whole surface. One downside of this is that the quality of the surface depends on the number of particles used, making this a bad choice for low-performance computers.

This application maps the surface vertices to the particle representation. For each vertex in the object's triangle mesh the three closest particles are found. When this is done the barycentric coordinates for the vertex in relation to the particles are computed. For all vertices, the indices for the three closest particles as well as the barycentric coordinates are saved. When rendering the scene the procedure is a lot faster: for finding a vertex position simply take the positions of the three closest particles and apply the barycentric coordinates. Using this approach the surface is always the same as the original model, no matter how many particles are used for the physics.

This approach is very simple to implement and is also very fast. The computationally heavy part is when loading a new object, where for v vertices and p particles the complexity is $\mathcal{O}(v \cdot p)$, but since this is not something done during run-time it is not considered a problem.

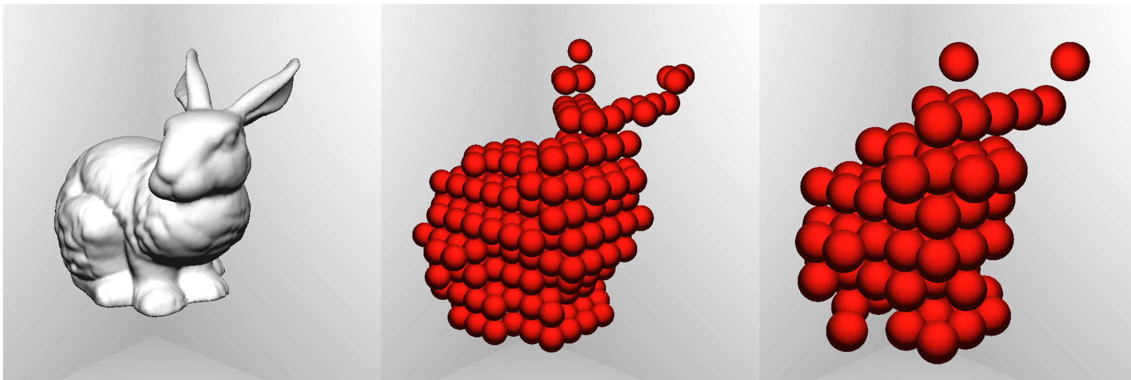


Figure 4.2: A mesh rendered on top of a voxelized particle representation and two different particle representations.

4.5 Models

3D models are stored in .obj files, containing the positions of all vertices and how they make up the surface mesh. To get a particle representation as well, a signed distance field is generated using *SDFGen* [36]. When loading objects in the application, both the .sdf and .obj files are used (see 4.4.2). Setting up the particle model is a simple matter of traversing a cube and sampling the .sdf file to find out if the current position is inside the model or not. The two special objects, cloth and box, are generated according to user inputs and thus have no corresponding .obj file. Constraints for the box and arbitrary meshes is generated by the distance between particles, resulting in constraints to all particles closer than $\sqrt{d_x^2 + d_y^2 + d_z^2}$, where d_i is the distance to the closest particle in dimension i. Similarly, for cloth, the constraints are added to neighboring particles but the diagonal constraints have a stiffness of 1% of the vertical/horizontal constraints.

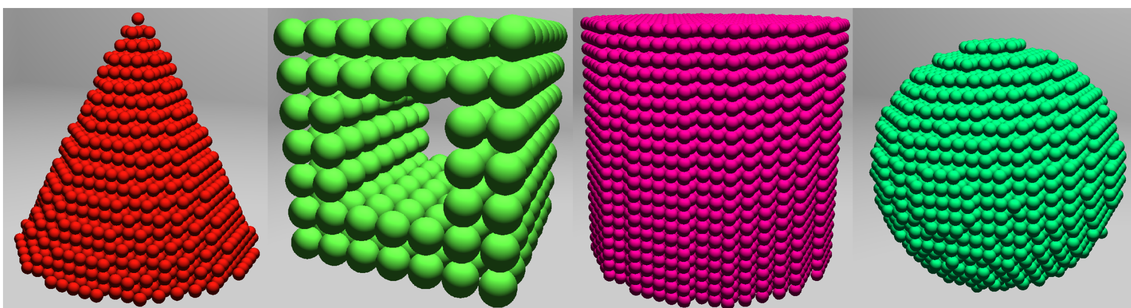


Figure 4.3: Models from left to right: cone, cube with hole, cylinder, sphere.

4.6 Scenes

Scenes are stored in .obj files, just like the models. When loading a scene, the .obj file for the desired scene, containing all vertices, the normal vector to each face, and indices to which vertices and which normal vector makes up a face, is parsed into separate vector representations. These are then sorted into a proper order and buffered onto the GPU, which draws the faces as triangles from the supplied vertices and normal vectors.

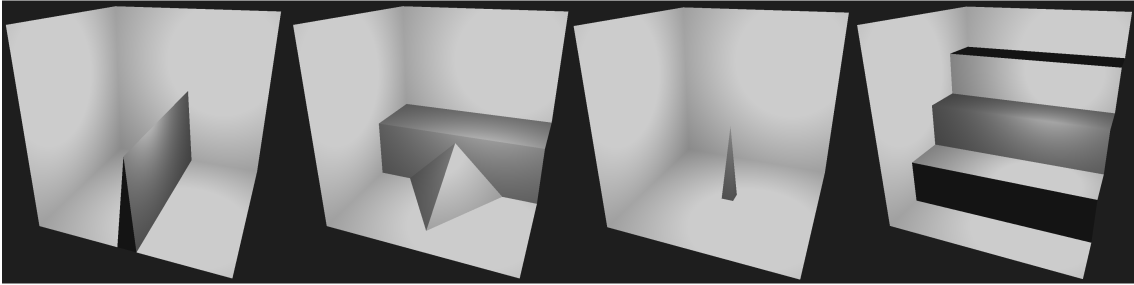


Figure 4.4: Scenes from left to right: axe, ledge, spike, stairs.

4.7 Constraints

Here the different kinds of constraints are described in both words and mathematical equations. For simplifying the equations, $\mathrm{C}(\mathbf{p}_a, \mathbf{p}_b) = \mathrm{C}(\mathbf{p}_{a,b})$ and $\mathbf{p}_a - \mathbf{p}_b = \mathbf{p}_{a-b}$.

4.7.1 Distance constraint

A distance constraint, perhaps as first mentioned by Jakobsen [48], constrain two particles to be a fixed distance apart. For an inequality constraint, the distance constraint can be interpreted as two particles should be at least a certain distance away from each other.

$$\mathrm{C}(\mathbf{p}_{1,2}) = +\|\mathbf{p}_{1-2}\| - l_0 \tag{4.1}$$

$$\nabla_{\mathbf{p}_1} \mathrm{C}(\mathbf{p}_{1,2}) = +\frac{\mathbf{p}_{1-2}}{\|\mathbf{p}_{1-2}\|} \tag{4.2}$$

$$\nabla_{\mathbf{p}_2} \mathrm{C}(\mathbf{p}_{1,2}) = -\frac{\mathbf{p}_{1-2}}{\|\mathbf{p}_{1-2}\|}. \tag{4.3}$$

By noting that the gradient of the constraint is the vector which points from one particle to the other, one can realize that $\Delta\mathbf{p}$ as computed by the solver, will always move the particles along this line.

4.7.2 Fixed point constraint

The fixed point constraint is very similar to the distance constraint, with the difference that there is only one particle that is constrained to be at a given fixed position $\mathbf{p}_f$, i.e. the rest length $l_0 = 0$.

$$\mathrm{C}(\mathbf{p}) = \|\mathbf{p} - \mathbf{p}_f\| - l_0 \tag{4.4}$$

$$\nabla_{\mathbf{p}} \mathrm{C}(\mathbf{p}) = \frac{\mathbf{p} - \mathbf{p}_f}{\|\mathbf{p} - \mathbf{p}_f\|}. \tag{4.5}$$

4.7.3 Environment collision

For collision between particles and regular static triangle meshes, a constraint is added that limits the particles to be at least a certain distance d_0 away, along with the triangle normal, from the surface.

$$\mathrm{C}(\mathbf{p}) = \mathbf{n}^T \mathbf{p} - d_0. \tag{4.6}$$

4.7.4 Particle collision

Particle collisions can be modeled by adding a constraint such that the distance between the colliding particles needs to at least equal their combined radius, r_1, r_2. Furthermore, it is possible to reuse the previously mentioned distance constraint with $d_0 = (r_1 + r_2)$ given that the constraint is only active during that frame.

$$C(\mathbf{p}_{1,2}) = \|\mathbf{p}_{2-1}\| - (r_1 + r_2). \tag{4.7}$$

4.7.5 Particle to particle friction

To include static friction in the model, friction is calculated during the constraint solve directly on the positions instead of just damping the velocities afterward. After a collision has been handled and the particles are no longer penetrating, a frictional position delta is calculated from the movement of the particles tangential to the collision normal.

4.7.6 Breakable constraints

When an object is under substantial deformation, it would be expected to see the object break or tear into pieces. In order to model this, an extra parameter is saved for each constraint which defines that constraints breaking threshold. At the

beginning of each frame, a check is run to see whether the evaluation of a constraint is greater than its threshold. Should this be the case, that constraint is removed from the list of constraints. In order to speed up this check, the constraints should be saved in a linked list. The reason behind this is that deletion of an element in the list becomes $\mathcal{O}(1)$ as opposed of the $\mathcal{O}(n)$ of a regular array. Should the array approach be used, and an object put under deformation such that all constraints break, this functionality will turn into a temporary bottleneck until all constraints have been removed. The way that this is done is as described by Macklin et al. [1].

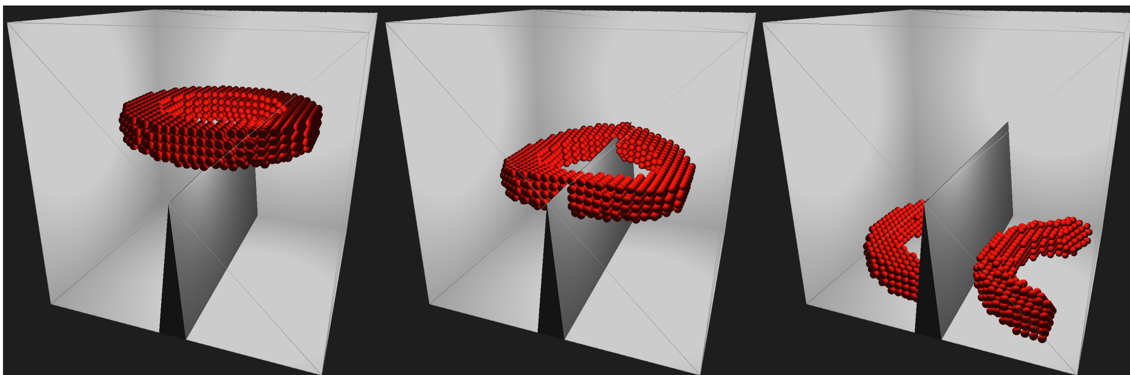


Figure 4.5: A torus being split in two. When the separation of two particles in a distance constraint reaches a given threshold, that constraint is removed from the simulation. The effect, as is seen in the image, is that of an object fragmenting.

4.8 The solver

The chosen stationary iterative method is both SOR and averaged Jacobi, in order to evaluate them against each other. Position deltas are computed by iterating over the constraints in the scene. As an example, the position deltas for two particles connected by a distance constraint is

$$\begin{aligned}\Delta\mathbf{p}_i &= -\frac{w_i(\|\mathbf{p}_i - \mathbf{p}_j\| - l_0)}{w_i + w_j}\frac{\mathbf{p}_i - \mathbf{p}_j}{\|\mathbf{p}_i - \mathbf{p}_j\|} \\ \Delta\mathbf{p}_j &= +\frac{w_j(\|\mathbf{p}_i - \mathbf{p}_j\| - l_0)}{w_i + w_j}\frac{\mathbf{p}_i - \mathbf{p}_j}{\|\mathbf{p}_i - \mathbf{p}_j\|}.\end{aligned} \tag{4.8}$$

When several solver iterations are computed for a single frame, simply multiplying the position updates with the constraint stiffness k would result in nonlinear behavior. In order to achieve a more predictable effect the position updates are instead multiplied by $1-(1-k)^{\frac{1}{i}}$ where i is the current iteration number (starting at one) as proposed by Müller et al. [6]. On top of this, the position deltas are also scaled by ω, the SOR parameter. The number of iterations that the solver is run for is decided from the GUI inside the program. XPBD is also implemented in order to evaluate the difference between the methods.

4.8.1 Pre-stabilization pass

It is not uncommon to have unsolved collisions at the beginning of a time step, one possible reason being that convergence was not reached in time during the last time step. When this happens, the solver might cause the objects to gain velocity in an unnatural way when solving interpenetrations (see Figure 4.6). To counter this, a pre-stabilization pass, as proposed in [1], is implemented. This optional pass solves interpenetrations before the main solver loop, updating both the predicted next position as well as the starting position for the time step without affecting velocity. This efficiently stabilizes initial conditions, which in turn enables the use of a lower solver iteration count since unsolved constraints no longer cause particles to gain unwanted energy. This sums up to a faster implementation overall without negatively affecting visual plausibility.

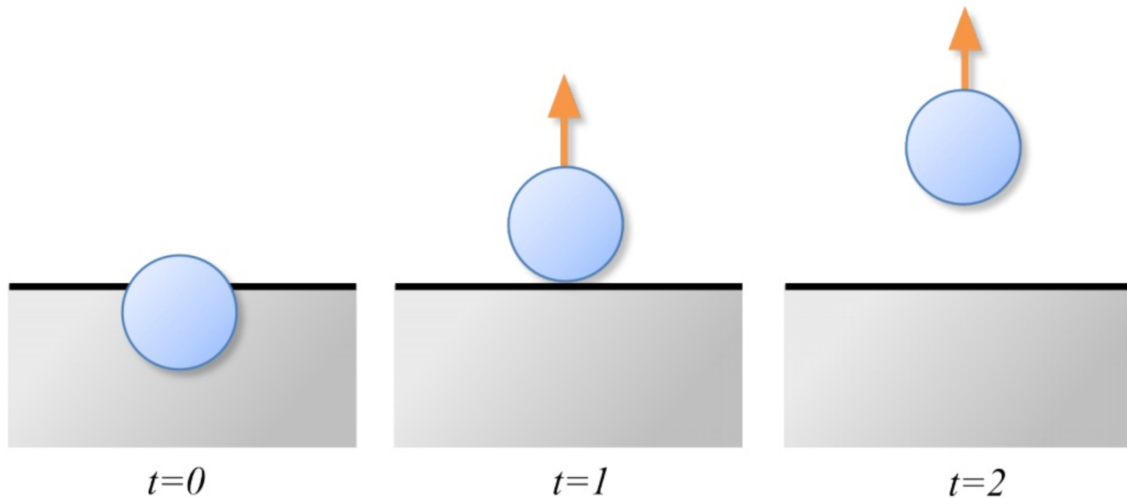


Figure 4.6: Particle is penetrating the surface at the beginning of the time step. When projected to a valid position, too large of a distance has been covered causing the system to gain kinetic energy. Image by Macklin et al. [1].

4.9 Collision detection

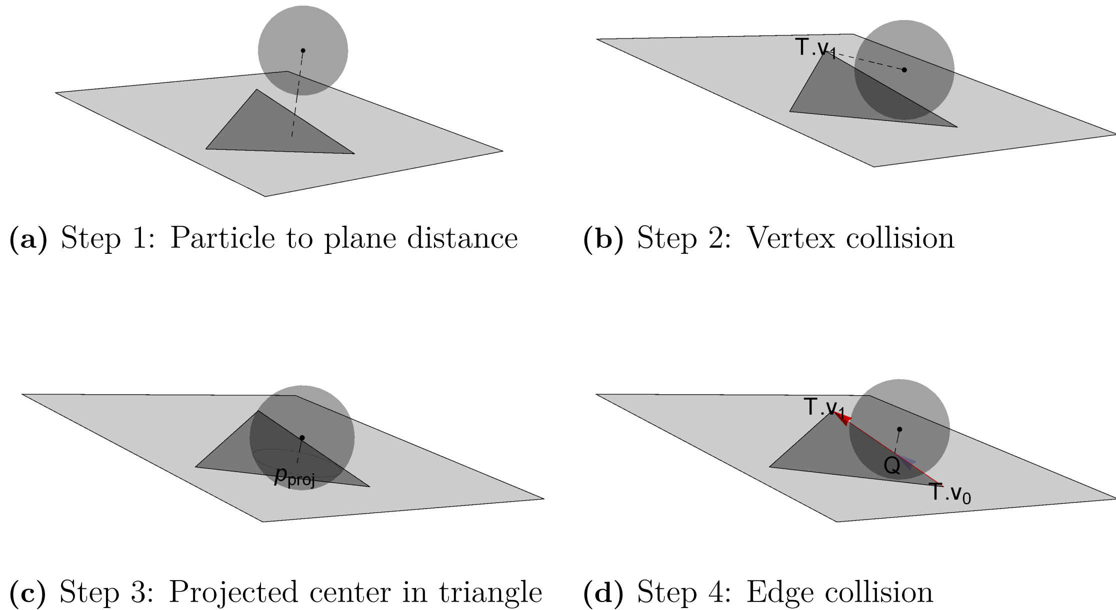
Collision detection is a very important step of the solver. Not only because a simulation without collisions is very unrealistic, but also because it is very time-consuming and optimization is paramount. There are two types of collisions handled by our application.

4.9.1 Triangle to particle collision

The implementation of collisions between a triangle T, and a particle p, which are represented as a sphere, is discrete. It works by going through four type of checks: plane, vertex, inside, edge. The first check is done by checking the distance between the plane spanned by the triangle and the particle (Figure 4.7a). Thus we get that the distance d can be calculated with the equation

$$d = (\mathbf{p} - \mathbf{T}_{v0}) \cdot \mathbf{T}_n \tag{4.9}$$

where $\mathbf{T}_{v0}$ is the triangle T's first vertex and $\mathbf{T}_n$ is the normal of the triangle. This makes sure that there is no collision if $d >$ radius.



(a) Step 1: Particle to plane distance **(b)** Step 2: Vertex collision

(c) Step 3: Projected center in triangle **(d)** Step 4: Edge collision

Figure 4.7: A visualization of the four steps of the collision detection. Step 1 checks the distance between the particle and the plane spanned by the triangle. Step 2 checks the distance between particle and every vertex. Step 3 checks if projected particle position onto plane is inside triangle. Step 4 checks the shortest distance between a line segment (edge) of the triangle and the particle position.

The second check is whether any of the three vertices of the triangle is inside the sphere. This is done by a simple check of the distance between the particle position $\mathbf{p}_p$ and the vertex position $\mathbf{T}.\mathbf{v}_i, i \in \{0,1,2\}$ (Figure 4.7b).

$$radius < (\mathbf{p} - \mathbf{T}.\mathbf{v}_i) \cdot (\mathbf{p} - \mathbf{T}.\mathbf{v}_i) \implies \text{Collision.} \tag{4.10}$$

The purpose of the third check is to check whether the projected position of the particle on the triangle plane $\mathbf{p}_{proj} = \mathbf{p} - d * \mathbf{T}.\mathbf{n}$ is inside the triangle (Figure 4.7c).

This is done by first calculating the Barycentric coordinates of $\mathbf{p}_{proj}$. For this the following calculation is used:

$$\mathbf{u} = \mathbf{T}.\mathbf{v}_1 - \mathbf{T}.\mathbf{v}_0 \tag{4.11}$$

$$\mathbf{v} = \mathbf{T}.\mathbf{v}_2 - \mathbf{T}.\mathbf{v}_0 \tag{4.12}$$

$$d_u = \frac{(\mathbf{v}\cdot\mathbf{v})(\mathbf{u}\cdot\mathbf{p}_{proj}) - (\mathbf{u}\cdot\mathbf{v})(\mathbf{v}\cdot\mathbf{p}_{proj})}{(\mathbf{v}\cdot\mathbf{v})(\mathbf{u}\cdot\mathbf{u}) - (\mathbf{u}\cdot\mathbf{v})^2} \tag{4.13}$$

$$d_v = \frac{(\mathbf{u}\cdot\mathbf{u})(\mathbf{v}\cdot\mathbf{p}_{proj}) - (\mathbf{u}\cdot\mathbf{v})(\mathbf{u}\cdot\mathbf{p}_{proj})}{(\mathbf{v}\cdot\mathbf{v})(\mathbf{u}\cdot\mathbf{u}) - (\mathbf{u}\cdot.\mathbf{v})^2}. \tag{4.14}$$

d_v and d_u can then be used to check if $\mathbf{p}_{proj}$ is inside the triangle:

$$d_u, d_v \geq 0 \wedge d_u + d_v \leq 1 \implies \text{Collision}. \tag{4.15}$$

By this stage it is known that the sphere is less than the radius away from the plane, it does not intersect any of the vertices, and its center projected on the plane is not inside the triangle. The last check is done to determine if there is any intersection between any of the edges of the triangle and the sphere. This is done by calculating the closest point $\mathbf{Q}$ on the line defined by two points (vertices) $\mathbf{T.v}_i$ and $\mathbf{T.v}_j$ (Figure 4.7d).
Let $\mathbf{e} = \mathbf{T.v}_j - \mathbf{T.v}_i$, then

$$t = \frac{\mathbf{e}\cdot\mathbf{p} - \mathbf{e}\cdot\mathbf{T.v}_i}{\mathbf{e}\cdot\mathbf{e}} \tag{4.16}$$

$$\mathbf{Q} = \mathbf{T.v}_i + t\mathbf{e}. \tag{4.17}$$

After this it is certain that there is an collision *if and only if* $|\mathbf{p} - \mathbf{Q}| < radius$ and $0 < t < 1$.

4.9.2 Particle to particle collision

Collision detection between particles is done with a very simple check of the distance between the particles. Thus there is a collision between particle i and j if $r_i + r_j < \sqrt{\|\mathbf{p}_i - \mathbf{p}_j\|}$ where r_i is the radius of the i:th particle.

4.9.3 Collision response

Whenever a collision is detected, a constraint is added. The constraint added depends on collision type where particle to particle collision is handled by a particle collision constraint and a triangle/particle collision is handled by an environmental collision, both of which have been detailed earlier.

4.10 Optimizing collision detection

Collision detection is one of the more taxing computational steps that are done during the simulation. Even worse is the naïve approach to check each of the $\frac{n(n-1)}{2}$ pairs of particles, also called the brute force approach.

To ease this, a proper data structure is a necessary alternative. The main contenders for this was a uniform grid, an octree, and a k-d tree, each of which has its set of advantages and disadvantages.

4.10.1 Uniform grid

For particle to particle collisions a parallel uniform grid is implemented [49]. Dividing the world space into equal-sized cells results in a uniform grid, where each cell will have a vector of particles. The main advantage of this structure is the ease of implementation, as it only needs a three-dimensional array or a one-dimensional array to hold each cell. The grid cells sizes and positions are defined by a lower corner c, a side length l and the desired number of cells per side n. The first thing to implement is the grid itself. For this, a single one-dimensional vector was used, because contiguous memory provides more coherent memory access. Next, all the particles are iterated over in order to assign them to one of the cells in the cell array. Insertions of particles into the grid is a $\mathcal{O}(1)$ operation per particle. The cell index of where to place a particle is easily calculated from the position of the particles as

$$(i_x,i_y,i_z) = \left(\left\lfloor \frac{n(\mathbf{p}_x - \mathbf{c}_x)}{l} \right\rfloor, \left\lfloor \frac{n(\mathbf{p}_y - \mathbf{c}_y)}{l} \right\rfloor, \left\lfloor \frac{n(\mathbf{p}_z - \mathbf{c}_y)}{l} \right\rfloor \right) \tag{4.18}$$

where i_x, i_y, i_z are indices for an equivalent three-dimensional grid, $\mathbf{p}$ is the position of a particle, and $\mathbf{c}$ is the position of the grid corner. These three-dimensional indices can then be converted into the one-dimensional case as

$$i = i_x + ni_y + n^2 i_z. \tag{4.19}$$

The particle with index i is then added to the cell array. Next up is to detect collisions. With the grid, this is done by iterating over the cells and for each cell, look for collisions with other particles both within the cell itself and in the 26 nearest neighboring cells. Think of this as a cube with three cells per side, and the middle cell is the one that is iterated over. When doing this, to avoid indexing out of bounds when calling the neighboring cells, the grid is padded with an extra cell on each side which is placed outside of the world space so that no particles will be assigned to those cells.

The main disadvantage of a uniform grid is the cell size, which needs to be larger than the biggest particle radius. A larger cell size leads to fewer cells and will negatively impact performance, as more collision checks have to be performed. With a smaller

cell size fewer particles will be placed in each cell, and therefore fewer collision checks have to be done for each cell. The trade-off is that a higher resolution grid requires more memory. As the goal is to localize the area of which to search for collisions, and memory for desktop computers is rarely limited, this trade-off is in most occasions worthwhile.

4.10.2 Octree

An octree is another type of space dividing data structure. In contrast to the uniform grid, the octree recursively subdivides space into eight equally sized, cubic, subvolumes. The condition of when to stop recursion either depends on the number of particles inside a cube or the volume of the cube. There are a couple of different ways to implement an octree. Instead of mentioning all the different ways, the chosen implementation is mentioned briefly, while the interested reader is referred to *Real-time rendering* [50].

The octree is built top-down in a normal recursive manner each frame. The tree is completely constructed from scratch each time. Nodes are stored scattered throughout memory as per the discretion of the operating system, no effort is put into maintaining a cohesive memory structure. If a particle should overlap the volumes of two nodes, that particle is considered as being a part of both nodes. Lastly, in order to detect collisions, brute force collision detection is run amongst all particles in each leaf node.

4.10.3 k-d tree

The k-d tree is, like the octree, a space dividing data structure, but unlike the octree space, it is not divided uniformly. The k-d tree is built by first sorting all particles in the x dimension, picking the median particle. Using this particle as a node in the tree, the rest of the particles are split into two parts, one containing all particles with x-coordinates lower than the nodes particles and the other with particles with a higher x-coordinate. This process is then recursively repeated, until the number of particles has reached some lower bound, with the twist that the sorting dimension is alternating $x \rightarrow y \rightarrow z \rightarrow x$ [..] for each level.

4.11 Parallelization

Physics simulation by using particles are, as mentioned in 3.5, a problem that lends itself well to parallelization. In this project, focus has been mainly on the construction of the data structure for collision detection, the collision detection itself and the solving of the constraints. By using TBB the parallelization of for loops becomes

trivial. TBB automatically chooses the number of threads that are created and then distributes the iterations of the for loops among these threads.

4.11.1 Constraint solver

For the constraint solving there are a number of ways to parallelize the algorithm. The ”Gauss-Jacobi” method [1] and graph coloring approach [24] are two ways to proceed. However, for both of these methods, it is assumed that the GPU is utilized for parallelization. For the CPU there are fewer parallel cores to work with. Under the assumption that parallel writes into RAM do not result in undefined behavior, but rather the memory will read either of the writes, it is possible to motivate a different kind of approach. If the order is completely disregarded the upfront cost of graph coloring can be cut, or as in the case of the Jacobi approach, do more updates which are based on new particle positions. The method implemented therefore proceeds as in the sequential version except that everything is now parallel. No synchronization is used for updating position values. This might result in lost updates, but the solution at large will still move towards a better solution.

4.11.2 Uniform grid

For the uniform grid, there are two parts that are parallelized: the insertion of particles into the grid and the collision detection. For the construction of the grid, the only part that needs precaution is when trying to insert two particles into the same cell at the same time. This is solved by making use of the concurrent vector class from the TBB library. It is capable of growing concurrently, and since it is not important in which order the particles are inserted, it fits the needs perfectly.

For the collision detection, a similar problem appears when adding items to an array. Constraints are to be added to the scene, however unlike building the grid, the concurrent vector can not be used in the same way as constraint parameters are stored in SoA style (see section 4.3). To solve this the whole addition of a constraint has been made into an atomic action, where only one thread can access that piece of code at a time.

4.12 Input handling and user interface

Input handling in this application is handled in two different ways. Input for the user interface (UI) is purely handled by dear ImGui [43]. There is also input to the rest of the application, so that movement and interactivity can be implemented. Any input to ImGui is not passed along, as to prevent movement when using the UI. When input is not handled by the UI, it is handled by an observer-like pattern. This pattern is comprised of functions and a list of observer objects. When some input event is triggered, such as mouse clicks, a specific function registered with GLFW to handle the event, calls the corresponding function in all subscribed observers.

The construction of the UI is delegated to each section of the program, e.g. the model UI is constructed by the section that handles models. The purpose of the UI is to provide the user with configuration options regarding the scene, model and general parameters of the simulation. There is also a performance window, whose purpose is to visualize a qualitative estimation of how much time each separate step (see section 4.2) of building the frame takes. The window consists mainly of an area chart, where each of the measurements is colored separately and stacked in order of the first measurement at the bottom, as shown in Figure 4.8. The timings are taken by using a platform specific function, *QueryPerformanceCounter* for Windows [51] and *clock_gettime* using *CLOCK_THREAD_CPUTIME_ID* for Unix [52].

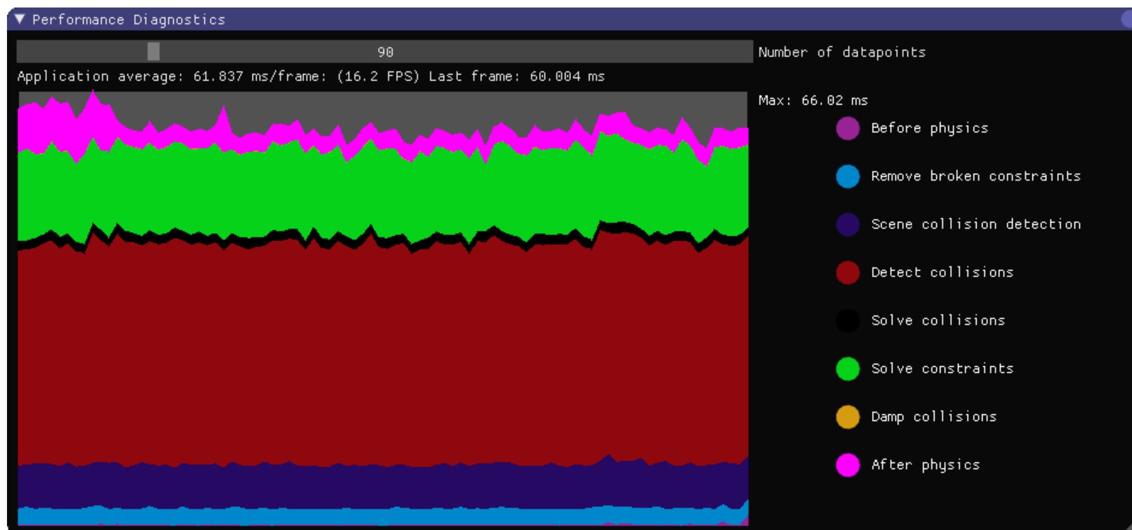


Figure 4.8: The performance graph used to gain qualitative assesments of how much time is spent in different parts of the program.

5

Results

This chapter will present a general view on how the application turned out, as well as some specific tests that were done in order to more precisely measure the capacity of the application and the techniques implemented. The application was tested on a number of different computers, but the main trials as described in this chapter were run on an Intel i7-4710HQ CPU clocked at 3.5 GHz on eight threads, unless stated otherwise.

5.1 Collision handling

As mentioned in 4.10, a proper data structure is imperative to the performance. In our application, we compared the slow brute force method, with using a uniform grid, and also with the use of an octree. This comparison, which was done on an Intel Xeon E3-1230 V2 clocked at 3.3 GHz, can be seen in 5.1.

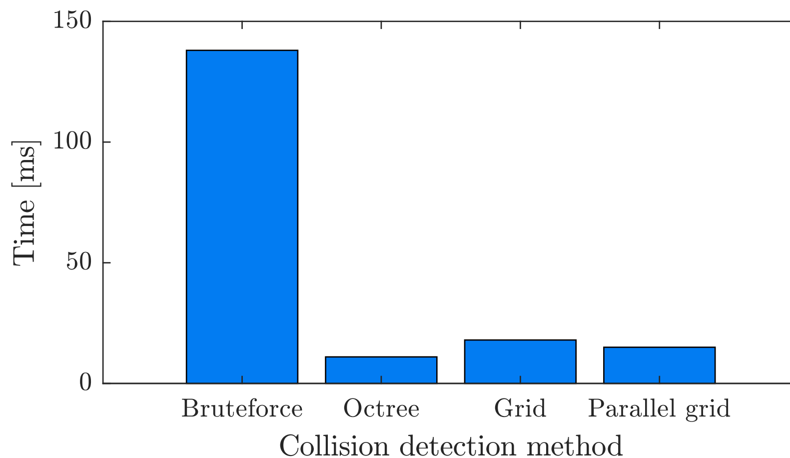


Figure 5.1: Time per frame for the different collision detection methods *bruteforce*, *octree*, *grid*, and *parallel grid*. Results were measured using 8000 particles.

5.1.1 Sequential and parallel grid

The grid is a data structure that lends itself well to parallelization. To see how effective the parallelization is, the averaged frame time over 462 data points was compared, which was also done on an Intel Xeon E3-1230 V2 clocked at 3.3 GHz, when using the sequential versus the parallel implementation. Figure 5.2 shows the comparison with different particle counts ranging from 500 up to 20 000

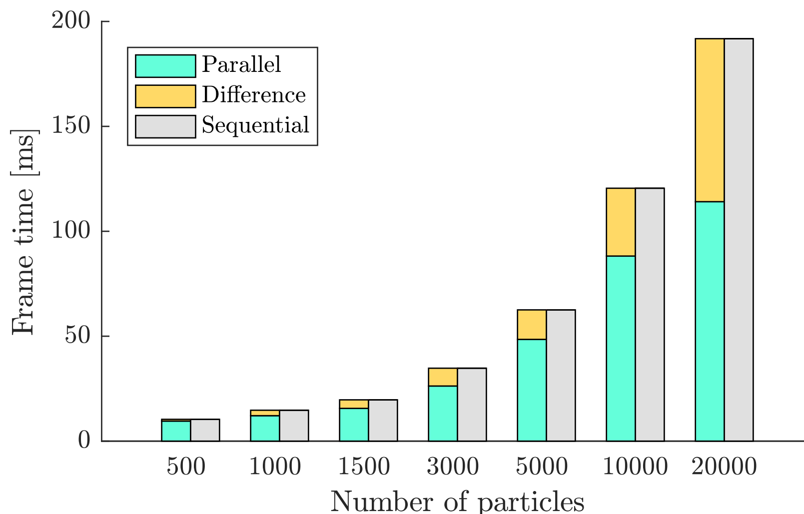
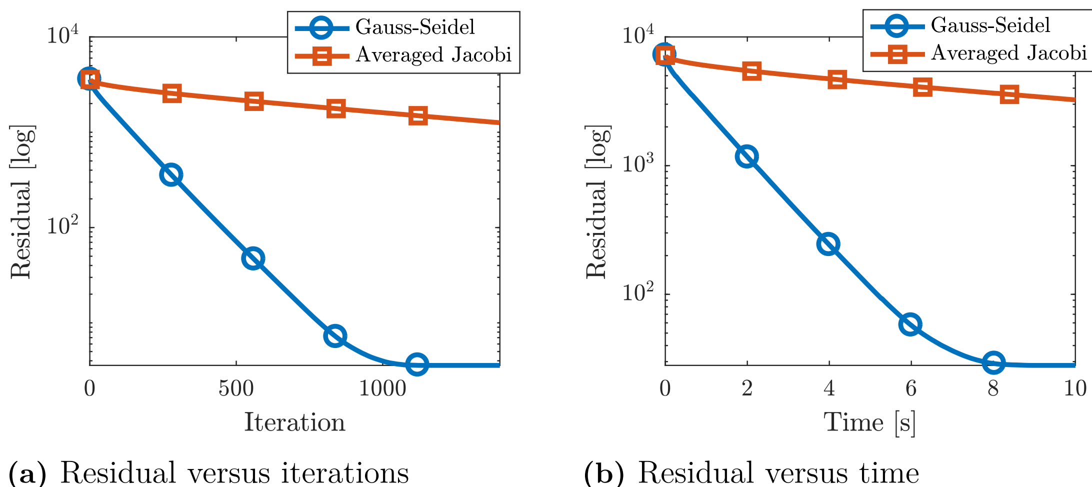


Figure 5.2: Time per frame using the uniform grid for collision detection. The sequential and parallel implementation is compared with the number of particles ranging from 500 to 20 000. Note how the difference increases with the number of particles when going from sequential to parallel.

5.2 Convergence speed

The main focus of comparing two main different techniques for solving linear systems, Gauss-Seidel (SOR with $\omega = 1$) and Jacobi, was to examine and compare the convergence rates. The convergence rates were tested by stretching a cloth model, which consisted of 50 by 50 particles and 9702 constraints, to 16 times its original size. The model was then released for it to converge back into its original form by solving the system over a large number of iterations. The result, as seen in Figure 5.3, was that Gauss-Seidel converged much faster than Jacobi over both the course of 1000 iterations and in actual computational time.

Convergence speed was further improved using a parallel solver. When doing this, the residual was seen to decrease much faster in time, but slightly slower regarding iteration count, see Figure 5.4.



(a) Residual versus iterations **(b)** Residual versus time

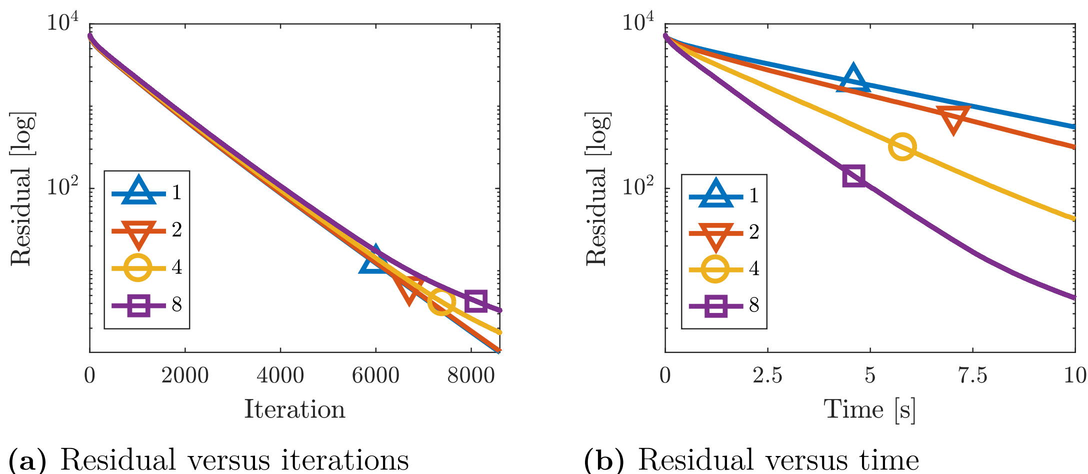
Figure 5.3: Comparison of the total constraint residual using Gauss-Seidel and averaged Jacobi. A cloth model with 50 × 50 particles and 9702 constraints was stretched to 16 times its original size and then released to let the distance constraints have it pull itself together. Notice that Gauss-Seidel has converged after about 1000 iterations while averaged Jacobi has not converged.

5.3 XPBD

To gain a qualitative comparison of the benefits of XPBD when included into the physics engine, a test scene with a piece of cloth consisting of a 64 by 64 particles was set up. A time step of 0.16 ms was used. Figure 5.5 shows regular PBD on the top row with a stiffness value of 0.9. XPBD is on the bottom row with $\alpha = 10^{-6}$. The columns are 20, 40, 80, and 160 iterations respectively. For XPBD, it can be seen that the distance constraints that are fixed to a given point, which is holding the cloth up, seem to be less stretched in general. The stiffness increase with iteration count in PBD is also reduced with XPBD. The actual incurred overhead of the computations done in XPBD is, as reported in [27], barely noticeable.

5.4 Core workload balance

In order to gain a better understanding of how well the computation work is split onto available cores, core load was measured for a set of different computers. The result was very clear, all of the cores showed high utilization under TBB. Figure 5.6 shows an example, an AMD Ryzen 5 1600 under load from the application, where the load balancer's efficiency can be seen as all 12 cores are under very similar load.



(a) Residual versus iterations

(b) Residual versus time

Figure 5.4: Comparison of the total constraint residual using parallelized Gauss-Seidel, running on one core/one thread, two cores/two threads, four cores/four threads, and four cores/eight threads. A cloth model with 100×100 particles and 39 402 constraints was stretched to 16 times its original size and then released to let the distance constraints have it pull itself together. Notice that a more parallelized solver converges faster in time, but that the sequential implementation has a better convergence rate per iteration.

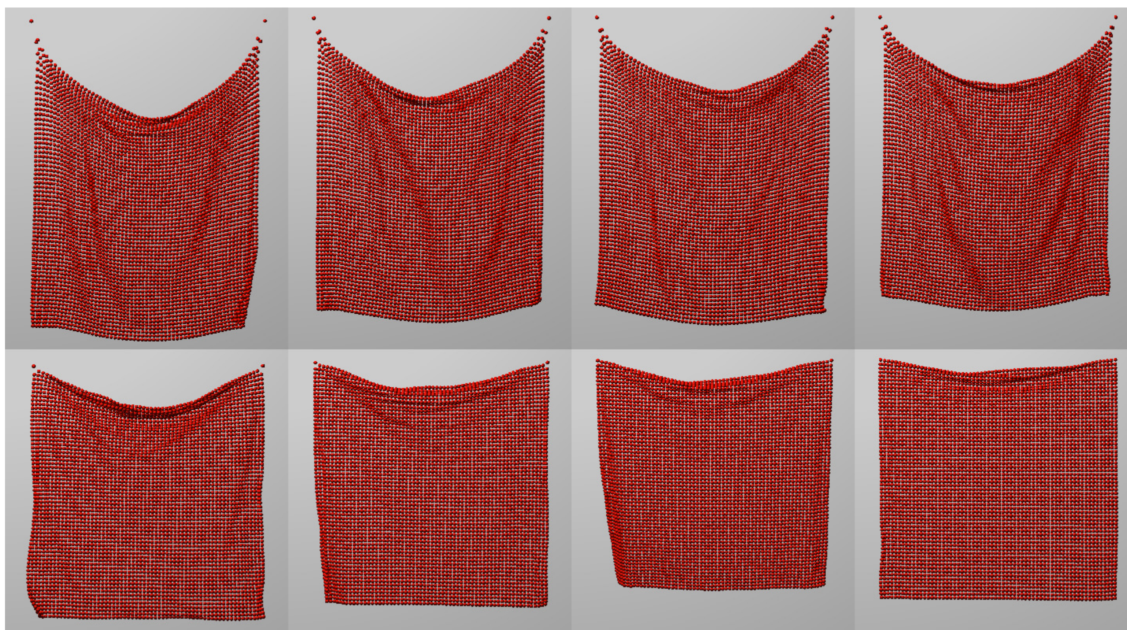


Figure 5.5: Comparison of normal PBD and XPBD, where PBD is top row and XPBD bottom row. Model used is a 64×64 cloth with a time step of 0.16 ms. The different images correspond to an iteration count of 20, 40, 80, and 160 iterations, from left to right. The α value used for XPBD is 10^{-6} and the stiffness value used for regular PBD is 0.9. Notice how PBD gets more stiff for higher iteration counts, and how that effect is reduced with XPBD.

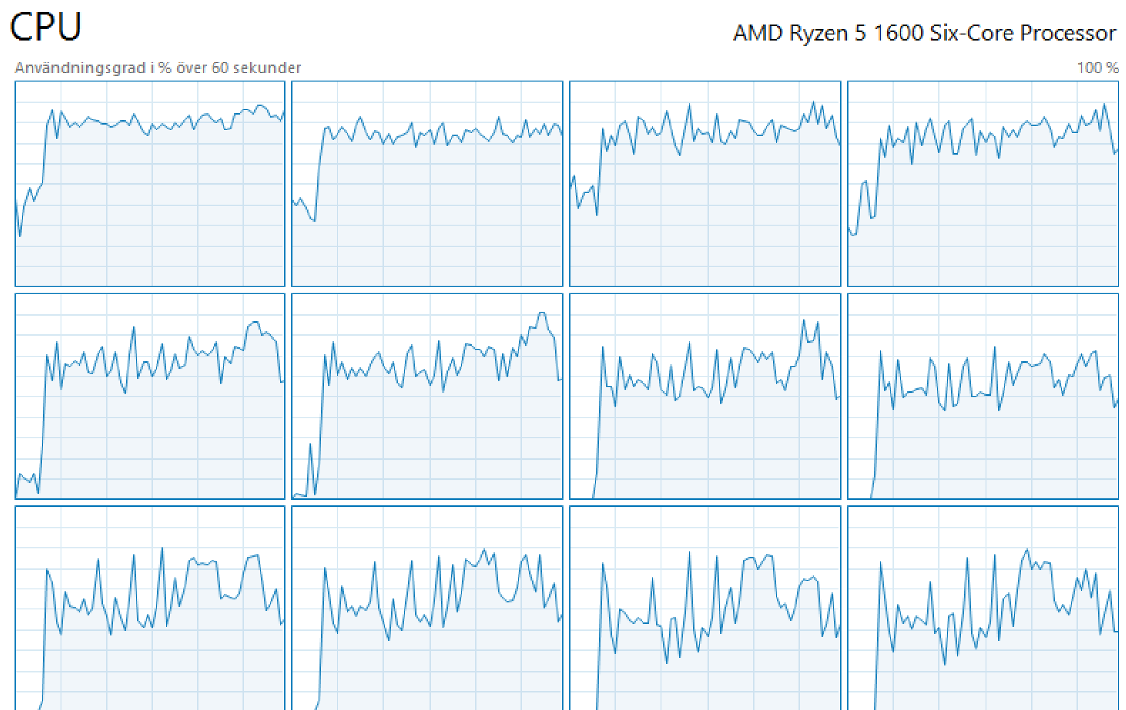


Figure 5.6: An AMD Ryzen 5 1600 Hexacore processor under heavy load from our application, showing the effectiveness of Intel TBB's load balancer.
Note: Text under 'CPU' translates to "Load in % over 60 seconds"

6
Discussion

In this chapter, we discuss the results and how they compare to our initial thoughts, as well as how the project turned out. We also discuss the problems and difficulties met and what can be done differently in a future project similar to this. Finally, we look into what can be improved on the project in the future should more time be available.

Coding a particle-based physics engine from scratch turned out to be a feasible task, while at the same time taking longer time than expected if all desirable constraint were to be implemented.

6.1 Data structure for collision detection

As shown in 5.1, brute force collision detection is unreasonably slow. Other than that, the implementation of grid and octree produced very similar results. Furthermore, it is no surprise that a parallel implementation is faster than a sequential.

6.2 Parallelization

The parallelization was facilitated greatly as a result of using TBB. Original expectations were for it to be more complex implementation wise, but as thread spawning, scheduling, and determining how much work a given thread should do is all done automatically, this turned out to be false. TBB was also very effective in balancing the load over all available cores, which is shown in Figure 5.6.

6.2.1 Grid

While the parallelization of the grid was faster (see Figure 5.2), its performance was underwhelming with regards to our expectations. The possible sources for this are the fact that it is not implemented for the GPU, there is an alternative method

for building the grid that could have better performance, or there could be flaws in the implementation. It is still very clear that the benefit of using a parallel implementation only increases as the number of particles grows.

6.2.2 Residual

As can be seen in the logarithmic scale of Figure 5.4b, parallelization of the Gauss-Seidel solver made the convergence rate for a low residual significantly faster. This proves how important parallelization is for these types of simulations. This being said, it is worth noting in Figure 5.4a, that the residual per iteration is reversed. This is because our parallel Gauss-Seidel does not update one particle at a time, preventing changes from propagating through the objects. Because of this, *per iteration* there will be lower residual with fewer threads while *over time* parallelization is better because of the overall speedup.

6.3 Gauss-Seidel versus Jacobi

The convergence rate of Gauss-Seidel was, as expected, faster than that of Jacobi (see section 3.2). What was not expected was the magnitude of the difference between the two, as can be seen in Figure 5.3. The difference means that much fewer iterations are needed for the simulation to look physically correct, which in turn allows for a higher frame rate. The reason for this large disparity lies in the fact that Jacobi is under relaxed in order to achieve stability.

6.4 Extended Position-Based Dynamics

Comparing PBD and XPBD is not necessarily straightforward as it can be hard to match the stiffness parameters of the two methods in order to conduct a fair test. Overall XPBD does a give more iteration and time step decoupled behavior as compared to PBD. Since XPBD barely incurs any additional computation cost, its effect is decidedly positive. From our experiences, XPBD seems to be the clear victor in most scenarios.

6.5 Project post-mortem

Given the initial scope that was set up, all features have been implemented. On top of this, a significant number of additional features has been implemented, as listed in section 1.3. The attitude and atmosphere have been positive, pleasant and helpful. The group has worked together without any internal issues, and each member has made essential contributions to the development process and the surrounding work. Every member has had a positive attitude and has been helpful towards others.

6.5.1 What went right

Many student, and even professional, projects are plagued by an increasingly disorganized version control system. Branches connected to finished and unfinished tasks alike are left in branches that are cluttering the system.

For this project, the workflow that was decided upon has been followed strictly. Proper routine where tasks have been properly finished, reviewed, merged and the corresponding branch deleted have led to a version control system usage that has benefited the work without being confusing for the user. Another thing that often gets increasingly disorganized and complex as the project moves along is the code base itself. It naturally grows over time and as each feature is added it inevitably becomes more complex. This project is no exception, but the code base is in a much better state than what could be expected of a project of this size, according to our experiences. Some refactoring has been done during the project, but mostly as a part of some implementation and not as a task in itself.

The project structure of having a weekly meeting on Thursday and a supervisory meeting on Friday worked well. Thursday was a good point, schedule-wise, in which to meet up, discuss progress and tasks, review code, merge finished tasks and prepare for the supervisory meeting the next day. The supervisory meeting gave a good chance to have a dialog with our supervisor and ask any question that came up during the week that we did not find the answer to ourselves.

6.5.2 What went wrong

Even though we have arrived in a satisfactory state, the project has been lacking real tangible milestones. Four milestones were set up during the planning phase, but these were not honored during the course of the project. The first milestone was reached within just two weeks and the second and third were slightly overlooked and skipped as the fourth was partially begun too early.

6.5.3 What could have been done differently

Due to the inexperience of the members about the subject and the lack of a clear leading party, many tasks were prioritized wrongly. If we would do this project again, due to the experience we now have we would have prioritized many tasks differently. Thus, some things, such as a cloth model and planning the tests, could have been done much earlier while other things could have waited until later, such as mesh rendering.

6.6 Future work

Because of the limited time frame, some interesting parts like liquids, smoke, and additional constraints, were not included in the scope. Thus, we would like to implement more constraints, so the simulation is extended to handle many more material types, such as gases and liquids. Some example of constraints that can be implemented are tetrahedral volume constraints, pressure constraints and constraints for fluids, which can, for example, be done as Macklin and Müller suggest [53].

6.6.1 GPU implementation

To further utilize the inherent parallelism in particle-based physics, moving the parallel part of the algorithms to the GPU would be a good idea. However, as it stands, the current parallel constraint solve algorithm would not work well with increased parallelism. The reason being as the number of parallel computations done, the higher amount of overwritten values. Instead one could look towards the graph coloring method, for example as shown by Fratarcangeli et al. [24].

6.6.2 Plastic deformation

One possible further implementation is that of plastic deformation. In the current application, constraints will break upon large deformation, but it is also possible to consider an intermediate state in which constraints are updated in such a way as to make the current deformed state the rest state. As a simple example, depicted in Figure 6.1, consider the distance constraint. While the current implementation of breakable constraints evaluates the whole constraint to decide if it should be removed, if plastic deformation is to be modeled it might be better to use the particle distance directly. The reason for this being that a plastic deformation might update the rest length and thus make the object infinitely extensible. Now, if two particles that make up a distance constraint is inside the region of plastic deformation, l_0 of the distance constraint is updated with a new value influenced by

their current separation distance. If the distance is large enough to be outside the plastic deformation region, it is simply removed as before.

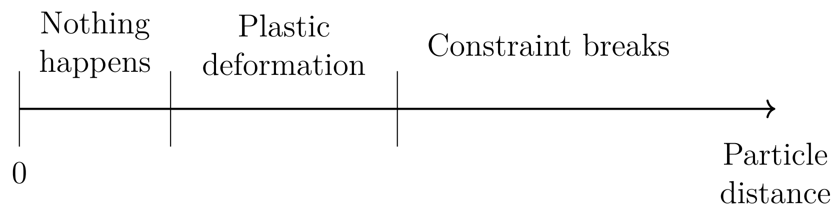


Figure 6.1: Simple model for how to handle both plastic deformation and material capable of breaking with distance constraints. If the particle distance is within the plastic deformation region, l_0, the rest length value of the distance constraint is updated.

6.6.3 Optimization

There are also further optimizations that can be done, such as manually adding SIMD (single instruction, multiple data) to speed up computations and better memory management to utilize caching better. Additionally, collision detection has many areas of improvement. There are more data structures that can be experimented with, the collision algorithms themselves can be looked upon and *Sweep and prune* [54] can be implemented.

6.7 Implications of Position-Based Dynamics for society

This kind of physics can see many uses in society, ranging from recreational areas to educational tools. One obvious application is games, where PBD can be used both for increased realism, but also for interesting game mechanics where whole environments can, or maybe even have to, be destroyed or deformed. This might lead to the game economy seeing an upswing.

Another application area can be in educational simulations, where better physics makes for a better learning experience. Both driving simulators and medical simulators can make use of this.

Finally, these techniques can see use in the movie industry, where computer generated scenes usually are rendered in a preview mode, which does not look like the final image, but can be rendered fast, to show the director. Better real-time physics could be used to make this preview look more similar to the final result, giving the director a better preview.

Performance wise, there are other implementations of this kind that will outperform it due to having a higher budget, more experienced creators, and a longer time frame.

It is difficult to make any clear statements about these things as they might be several steps removed from the immediate use of the idea.

7

Conclusion

Real-time physics is a computationally heavy task, requiring clever solutions and approximations to achieve the desired performance. Position-based dynamics provide a very stable foundation for a project like this and can be further improved with techniques such as XPBD.

Due to the large number of particles, brute force with its computational complexity of $\mathcal{O}\left(n^2\right)$ is not an option. A different approach such as a grid or octree is essential. In a comparison between an octree implementation, a sequential, and a parallel uniform grid, we found that the octree exhibits the best results for the given tests on the CPU.

Parallelization is a must for this amount of computations, but as always it is important to avoid race conditions and other potential problems due to multithreading.

After a comparison between the Gauss-Seidel and averaged Jacobi method in convergence per time and iteration, we found that Gauss-Seidel is the better option for sequential use.

We have implemented a SOR solver which ignores order when solving equations. It is shown that the loss of convergence in a given iteration is outweighed by the lower time each iteration requires, given the assumption that a low number of parallel threads are used.

There is still room for improvements and with more time many areas could be improved with more complex but faster algorithms. These areas include, but are not limited to, continuous collision detection, better construction of spatial dividing data structures moving all parallel algorithms to the GPU.

Bibliography

[1] M. Macklin, M. Müller, N. Chentanez, and T.-Y. Kim, "Unified particle physics for real-time applications," *ACM Transactions on Graphics*, vol. 33, no. 4, Article 153, July 2014, DOI = 10.1145/2601097.2601152.

[2] Valve developer Community. [Online]. Available: https://developer.valvesoftware.com/ (Visited 2017-05-30).

[3] Garry's Mod: A sandbox game for the PC, Mac and Linux. [Online]. Available: http://www.garrysmod.com/ (Visited 2017-05-30).

[4] BeamNG. [Online]. Available: http://www.beamng.com (Visited 2017-05-30).

[5] A. Stomakhin, C. Schroeder, L. Chai, J. Teran, and A. Selle, "A material point method for snow simulation," *ACM Trans. Graph.*, vol. 32, no. 4, pp. 102:1–102:10, Jul. 2013.

[6] M. Müller, B. Heidelberger, M. Hennix, and J. Ratcliff, "Position based dynamics," *Journal of Visual Communication and Image Representation*, vol. 18, no. 2, pp. 109 – 118, 2007.

[7] Git –everything-is-local. [Online]. Available: https://git-scm.com/ (Visited 2017-05-30).

[8] GitHub – IMP-Engine/Engine. [Online]. Available: https://github.com/IMP-Engine/Engine (Visited 2017-05-30).

[9] A successful Git branching model. [Online]. Available: http://nvie.com/posts/a-successful-git-branching-model/ (Visited 2017-05-09).

[10] The Scrum Guide. [Online]. Available: www.scrumalliance.org/why-scrum/scrum-guide (Visited 2017-05-30).

[11] Slack - Where work happens. [Online]. Available: https://slack.com/ (Visited 2017-05-30).

[12] The Witcher 3. [Online]. Available: https://thewitcher.com/en/witcher3 (Visited 2017-05-30).

[13] CryEngine 3. [Online]. Available: www.cryengine.com (Visited 2017-05-30).

[14] Unreal Engine. [Online]. Available: https://www.unrealengine.com/what-is-unreal-engine-4 (Visited 2017-05-30).

[15] Nvidia's PhysX. [Online]. Available: developer.nvidia.com/physx (Visited 2017-05-30).

[16] J. Bender, M. Müller, M. A. Otaduy, M. Teschner, and M. Macklin, "A survey on position-based simulation methods in computer graphics," in *Computer Graphics Forum*, vol. 33, no. 6. John Wiley & Sons Ltd., 2014, pp. 228–251.

[17] J. W. Demmel, *Applied numerical linear algebra.* SIAM, 1997.

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

[19] M. Fratarcangeli and F. Pellacini, "Scalable partitioning for parallel position based dynamics," in *Computer Graphics Forum*, vol. 34, no. 2. Wiley Online Library, 2015, pp. 405–413.

[20] D. Weber, J. Bender, M. Schnoes, A. Stork, and D. Fellner, "Efficient gpu data structures and methods to solve sparse linear systems in dynamics applications," in *Computer Graphics Forum*, vol. 32, no. 1. Wiley Online Library, 2013, pp. 16–26.

[21] J. M. Bahi, R. Couturier, and L. Z. Khodja, "Parallel gmres implementation for solving sparse linear systems on gpu clusters," in *Proceedings of the 19th High Performance Computing Symposia.* Society for Computer Simulation International, 2011, pp. 12–19.

[22] M. Müller, "Hierarchical Position Based Dynamics," in *Workshop in Virtual Reality Interactions and Physical Simulation "VRIPHYS" (2008)*, F. Faure and M. Teschner, Eds. The Eurographics Association, 2008.

[23] H. Wang, "A chebyshev semi-iterative approach for accelerating projective and position-based dynamics," *ACM Transactions on Graphics (TOG)*, vol. 34, no. 6, p. 246, 2015.

[24] M. Fratarcangeli, V. Tibaldo, and F. Pellacini, "Vivace: a practical gauss-seidel method for stable soft body dynamics," *ACM Transactions on Graphics (TOG)*, vol. 35, no. 6, p. 214, 2016.

[25] J. Bender, M. Müller, and M. Macklin, "Position-based simulation methods in computer graphics," EUROGRAPHICS 2015.

[26] Numerically integrating equations of motion. [Online]. Available:

http://pages.physics.cornell.edu/~myers/teaching/ComputationalMethods/ComputerExercises/Pendulum/NumericalODE.pdf (Visited 2017-05-02).

[27] M. Macklin, M. Müller, and N. Chentanez, "Xpbd: Position-based simulation of compliant constrained dynamics," in *Proceedings of the 9th International Conference on Motion in Games*, ser. MIG '16. New York, NY, USA: ACM, 2016, pp. 49–54.

[28] Threading Building Blocks (Intel TBB). [Online]. Available: https://www.threadingbuildingblocks.org/ (Visited 2017-05-30).

[29] C++. [Online]. Available: www.cplusplus.com (Visited 2017-05-30).

[30] R. Hundt, "Loop recognition in c++/java/go/scala," in *Proceedings of Scala Days 2011*, 2011. [Online]. Available: https://days2011.scala-lang.org/sites/days2011/files/ws3-1-Hundt.pdf

[31] OpenGL. [Online]. Available: www.opengl.org (Visited 2017-05-30).

[32] CMake. [Online]. Available: https://cmake.org/ (Visited 2017-05-30).

[33] M. V. (2011) Intel® Threading Building Blocks, OpenMP, or native threads? [Online]. Available: https://software.intel.com/en-us/intel-threading-building-blocks-openmp-or-native-threads (Visited 2017-03-03).

[34] S. Nanz, S. West, K. S. Da Silveira, and B. Meyer, "Benchmarking usability and performance of multicore languages," in *Empirical Software Engineering and Measurement, 2013 ACM/IEEE International Symposium on.* IEEE, 2013, pp. 183–192.

[35] OpenGL Mathematics (GLM). [Online]. Available: glm.g-truc.net/0.9.8/index.html (Visited 2017-05-30).

[36] SDFGen. [Online]. Available: github.com/christopherbatty/SDFGen (Visited 2017-05-30).

[37] Blender. [Online]. Available: www.blender.org (Visited 2017-05-30).

[38] glad. [Online]. Available: github.com/Dav1dde/glad (Visited 2017-05-30).

[39] OpenGL Loading Library. [Online]. Available: https://www.khronos.org/opengl/wiki/OpenGL_Loading_Library (Visited 2017-05-30).

[40] GLFW. [Online]. Available: www.glfw.org (Visited 2017-05-30).

[41] GLUT. [Online]. Available: www.opengl.org/resources/libraries/glut (Visited 2017-05-30).

[42] Simple DirectMedia Layer. [Online]. Available: http://www.libsdl.org/ (Visited 2017-05-30).

[43] Immediate Mode Graphical User interface (imgui AKA ImGui. [Online]. Available: github.com/ocornut/imgui (Visited 2017-05-30).

[44] Dirent - C/C++ library for retrieving information on files and directories . [Online]. Available: github.com/tronkko/dirent (Visited 2017-05-30).

[45] Introduction to Data-Oriented Design. [Online]. Available: http://www.dice.se/wp-content/uploads/2014/12/Introduction_to_Data-Oriented_Design.pdf Slides from a talk from DICE, motivating a Data-Oriented Design.

[46] Kobbelt and Botsch, "Point-Based Rendering," *Computers & Graphics*, 2004. [Online]. Available: http://www.cs.rug.nl/~roe/courses/acg/rendering (Visited 2017-03-03).

[47] E. W. Weisstein. [Online]. Available: http://mathworld.wolfram.com/BarycentricCoordinates.html (Visited 2017-05-30).

[48] T. Jakobsen, "Advanced character physics," in *Game Developers Conference*, vol. 3. Citeseer, 2001.

[49] S. Green, "Cuda particles," *NVIDIA whitepaper*, vol. 2, no. 3.2, p. 1, 2008.

[50] T. Akenine-Möller, E. Haines, and N. Hoffman, *Real-Time Rendering 3rd Edition*. Natick, MA, USA: A. K. Peters, Ltd., 2008.

[51] Windows Dev Center: Acquiring high-resolution time stamps. [Online]. Available: https://msdn.microsoft.com/en-us/library/windows/desktop/dn553408(v=vs.85).aspx (Visited 2017-05-30).

[52] clock_gettime(3) - Linux man page. [Online]. Available: https://linux.die.net/man/3/clock_gettime (Visited 2017-05-30).

[53] M. Macklin and M. Müller, "Position based fluids," *ACM Trans. Graph.*, vol. 32, no. 4, pp. 104:1–104:12, Jul. 2013. [Online]. Available: http://doi.acm.org/10.1145/2461912.2461984

[54] D. Baraff, "Dynamic simulation of non-penetrating rigid bodies," Ph.D. dissertation, Computer Science Department, Cornell University, 1992, pp. 52–56.