Position Based Dynamics

Pieterjan Bartels

Researching and Implementing a Particle-Based Unified Physics Library.

Bournemouth University

August, 2015

Contents

List of Figures

Abstract

In this thesis, a position based dynamics library is developed. It reviews academic research literature focused on this class of algorithms, and extracts the mathematical equations necessary to implement these techniques. It then goes on to give the details of the software implemented by the author, and its results. The resulting library is capable of simulating several types of effects: rigid and deformable bodies, cloth, granular material, hair and fluids. The project was also made thread-safe and implemented in parallel. After analysing the results of the implementation, an original improvement to the Position Based technique is proposed, and the results of an experimental implementation of this improvement are presented.

Keywords: Physical Simulation, Position Based Dynamics, Unified physics

Chapter 1

Introduction

This report describes the masters project made by the author. This section will introduce the project and its goals, and then outline the rest of the thesis.

Problem statement

One of the core jobs of the Visual effect industry is providing films with simulated versions of physical phenomena that are either too expensive or just practically impossible to actually film. Several different phenomena that are ubiquitous in human life have been extensively researched in the Computer Graphics literature in the last few decades. some examples of well-researched simulation topics are cloth, granular material, fluids, smoke, rigid bodies, ... (Parent 2012)

An obvious research goal for such simulations is accuracy, i.e. creating a simulated version of a phenomenon that is as close to the real-life thing as possible. However, the simulations for visual effects are often on a massive scale, and since creating these effects usually happens in a commercial environment, the algorithms should be as efficient as possible, so as to ensure a smooth pipeline when working towards deadlines.

It is easy to see these two goals typically form a trade-off, and different classes of simulation algorithms can be found in different places in the spectrum between efficiency and accuracy.

This thesis is concerned with a class of simulation algorithms that is located towards the efficiency end of said spectrum: Position based dynamics (PBD). These algorithms have gained a lot of attention in the past years, especially in interactive environments, because they are unconditionally stable and provide plausible, yet inaccurate, results. The stability allows large time-steps, making them efficient. Given the rise of multi-processor computers in the last decade or so, an algorithm aiming to be as efficient as possible should allow making use of these multiple processors. Many recent position based dynamics publications mention parallelism.

As mentioned, different algorithms exist for different phenomena. However, in reality these phenomena interact with one another. For example a piece of cloth or a rigid body floats on water, and creates splashes and waves in the fluid simulation of the water. Most of the research literature makes little or no mention of these interactions, but some research is specifically focused on them, be it one particular type of interaction or trying to incorporate all interactions (also called unified physics).

Due to the capabilities of the PBD technique, and its relative novelty, researching it and implementing a library based on it was deemed a suitable subject for the author's masters project. Given the mentioned considerations, the main goals of this project were the following:

- Researching how to implement and effectively implement a dynamics library based on the position based dynamics idea. These two goals, researching and implementing, are intertwined: implementing should be a means of researching and analysing the techniques described in literature.

- Implement it in a parallel manner, meaning that it makes as much use as possible of the available computing power.
- Have the library include at least a couple of different phenomena and their interactions.
- Ideally, the library should also be extensible: adding new types of phenomena should be easy, and should adhere the open/closed principle: Extending the author's library should be possible without modifying the author's software.
- While the extensibility was specifically mentioned, the end result should ideally follow all principles of good software engineering and be well-designed code.

Structure

The rest of this document will describe the authors efforts towards the given goals, and his results. The next chapter, chapter 2, will give a concise overview of related work in the research literature. This will be the base for chapter 3, which will explain the mathematics and physics of the different phenomena and how they were simulated in the project. This chapter will present pseudo-algorithms representing the inner working of the resulting library. The following chapter, chapter 4, will outline the library's code design and the work-flow used for achieving the final result. Any problems that were encountered during implementation will be shown, and any crucial decisions in the project will be highlighted. Chapter 5 then shows the results of the project. After analysing these results, this chapter shows the results of an original improvement to PBD. This improvement is discussed in section 5.4 specifically. Chapter 6 concludes.

Chapter 2

Related Work

Physical simulation has been a popular topic of research in the CGI community. Position based dynamics algorithms specifically have been growing in popularity over the last few years, and the list of publications related to this class of algorithms grows ever longer. This section aims to concisely review the work most relevant to this project. Due to the growing popularity of this class of algorithms, an exhaustive review of all previous work on PBD, let alone physical simulation, is out of scope. For an overview of the multitudes of other existing physical simulation algorithms and research, we refer to Parent (2012). For PBD specifically, Bender *et al.* (2014) and Bender *et al.* (2015) provide good and very recent overviews of the existing techniques and limitations, and have served as the main guides for this project as well, besides Macklin *et al.* (2014).

Position based dynamics was first introduced by Jakobsen (2001). They introduce a cloth system based on constraints rather than springs to keep particles at a certain distance. The constraints are iteratively applied to the system, hoping this will result in a state were all constraints are (close to) satisfied. The advantage of this technique over a mass-spring model is its stability when taking large time-steps, and the technique is specifically aimed at an interactive setting (i.e. computer games). On top of so called spring constraints, he also introduces constraints to deal with collisions: particles are kept inside a box (by keeping them within a certain range per dimension), are projected out of objects and are kept a certain distance away from each other (to avoid self-intersection).

Building on this, Müller *et al.* (2007) formalize Position Based Dynamics. They provide a mathematical framework for the constraints, including constraint weights, and a solver-algorithm that is independent of the type of constraints. This opened the door for new constraint types simulating different effects, where PBD's constraint solver could serve as a unified solver. They recognise that position based dynamics sometimes suffers from slow convergence due to slow spatial propagation of errors, and follow it up with Hierarchical Position Based Dynamics to counter this (Müller 2008). Follow-up work then introduced more constraints, and thus effects, to PBD. These new constraints usually build on already existing techniques.

Rigid bodies were introduced to PBD by Deul *et al.* (2014), building on the connectors from Witkin *et al.* (1990). However, as shown by Müller *et al.* (2005) it is also possible to do rigid body dynamics (RBD), and deformable objects, by using shape matching. Macklin *et al.* (2014) use these shape matching techniques for particle-based non-convex rigid bodies. This led to shape matching constraints for PBD. The latter approach was preferred for this project. Shape matching can also be used to simulate deformable object (Müller *et al.* 2005), providing two effects with one technique. Other solids, including cloth, have been proposed or improved separately as well. An overview is given in Bender *et al.* (2013).

Even fluids have been simulated with PBD. Macklin and Müller (2013) enforce incompressibility using a pressure constraint per fluid particle, based on the standard SPH formulas. They also provide ways to correct negative pressure at boundaries and to provide vorticity confinement (Macklin and Müller 2013).

Granular materials require friction to be simulated correctly (Jaeger *et al.* 1996). These have also been simulated with PBD. One alternative is to use the ideas of Bell *et al.* (2005), who simulate granular materials using small Rigid Bodies. The friction is then implied in the rigid body interaction. However, in PBD friction has been often modelled by damping the velocities post constraint solving (Müller *et al.* 2007). Recently the friction has been placed inside the constraint solving loop by Macklin *et al.* (2014), with a specific friction constraint.

As was mentioned, PBD was introduced for its stability. With the rise of multi-core computers, algorithms aimed at interactive applications would ideally take advantage of this increase in computing power. Position based dynamics can be implemented in a multi-threaded fashion, as was recently shown by Macklin *et al.* (2014). In order to do so efficiently, the algorithm is slightly modified from Gaussian iterations to Jacobi-style iterations, meaning that each constraint is solved independently.

On top of focusing on multi-threading, Macklin *et al.* (2014) also aim to simulate every thing using particles. Because of the generality of the PBD solver, and all effects being particle-based, their technique turns out to be an elegant unified physics algorithm, that is also easy to parallelize. For this reason, Macklin *et al.* (2014) was used as the main reference for this project. As a result, the resulting library described in the next chapters will not only be a unified, position based physics library, but will also be particle-based .

Chapter 3

Mathematical and Physical Framework

This chapter will introduce the main mathematics and algorithms that were implemented in this project. It will build on the papers mentioned in the previous chapter, picking out the parts that were implemented for the project and referencing them accordingly.

3.1 Algorithm Outline

This section outlines the general idea and maths behind position based dynamics. For this, we mostly follow the notation of Müller *et al.* (2007) and Bender *et al.* (2015).

In general terms, position based dynamics is a technique for calculating the movement of a set of vertices through 3D space (Müller *et al.* 2007). This movement can be caused by external or internal forces on these vertices. In order to know the position and velocity of these vertices, we integrate the net force on them, since force is related to acceleration, the second derivative of position, as dictated by Newton's second law:

$$a = \frac{F_{net}}{m} \tag{3.1}$$

where a is the acceleration of the vertex, F_{net} the net force on the vertex, with the net force being the sum of the internal forces and the external forces $F_{net} = F_{int} + F_{ext}$. Finally, m is the mass associated with the vertex.

The external forces are usually easy to apply, e.g. gravity, wind, drag, ... The difference between different simulation methods is how they model and calculate the internal forces. A simple example are mass-spring systems, where the internal forces on the vertices or particles are represented with linear springs. Several other models for the internal forces exist, but most of them are either slow or unstable (or both).

Constraints In position based dynamics the problems associated with internal forces are avoided by replacing them by so-called constraints that work directly on the position rather than being added to the net force. Hence, in PBD, the vertex positions are updated through simple integration of the external forces, and those positions are then directly subjected to constraints, which are functions of the positions themselves (Müller *et al.* 2007).

This project was based on the work of Macklin *et al.* (2014), which is purely with particle-based. As particles have no orientation, the notation in what follows does not mention a rotational component: for example, constraints will be functions of the particle positions only, not their orientation. For a more general notation including rotation, we refer to Bender *et al.* (2015). Hence in this thesis we represent constraints on systems with N particles as follows (notation from Müller *et al.* (2007)):

$$C(x_1, x_2, ..., x_n) = 0 \tag{3.2}$$

or

$$C(x_1, x_2, .., x_n) \geq 0. \tag{3.3}$$

All constraints are in one of these formats, called equality of inequality constraints respectively. The position of particle i is represented by x_i. For readability purposes, constraints are sometimes also represented as:

$$C(p) \succ 0 \tag{3.4}$$

where p is the vector obtained by concatenating x_i of particles 1 to N. The symbol $\succ$ is used to represent both $=$ or $\geq$. This gives a short notation for both kinds of constraint. Note that while in general constraints are functions of all particle positions, most constraint types work on a limited number of particles.

Since PBD solves the constraints individually, it basically turns the problem into many under-determined single scalar equations of the following form (Müller *et al.* 2007):

$$C(p + \Delta p) \succ 0. \tag{3.5}$$

Expressed in words this comes down to finding the correction Δp to the positions p of the system that makes the system satisfy the constraint. Since the system is under-determined, the constraints are linearized (Bender *et al.* 2015):

$$C(p + \Delta p) \approx C(p) + \nabla C(p) \cdot \Delta p \succ 0, \tag{3.6}$$

and then a solution Δp is found along the direction $\nabla C(p)$, bringing the amount of unknowns in the equation down to one Lagrange multiplier λ. The equation is hence no longer under-determined.

The Lagrange multiplier then has to be so that the correction Δp of the form (Bender *et al.* 2015)

$$\Delta p = \lambda M^{-1} \nabla C(p) \tag{3.7}$$

satisfies Equation (3.6). These simplifications lead to the correction for a particle i affected by the constraint:

$$\Delta x_i = -\lambda w_i \nabla_{x_i} C(p) \tag{3.8}$$

where $\nabla_{x_i} C(p)$ is the gradient of $C(p)$ according to a change in position x_i, w_i is the inverse mass of particle i and

$$\lambda = \frac{C(p)}{\Sigma_j |\nabla_{x_j} C(p)|^2}. \tag{3.9}$$

Where j iterates over all particles affected by the constraint. Sometimes a multiplier k is added to the right side of Equation (3.8). This acts as constraint weight, and is supposed to be set by the user in the interval $[0, 1]$.

In conclusion, to update one particle i by one constraint, one has to calculate the particle-independent λ as in Equation (3.9) (which is the same for all particles), and a particle-dependent gradient $\Delta_{x_i} C(p)$, and combine them as in Eq. (3.8).

Solver In most systems, several constraints need to be applied. In early work, all constraints were processed sequentially, updating the position of the particles after every constraint (Müller *et al.* 2007). Since a constraint might invalidate an earlier processed one, the group of constraints is iterated over, and all constraints are solved several times, every time in the same order. This way of solving the constraints is called Gauss-Seidel iterations, and can be proven to converge to a correct solution (Müller *et al.* 2007). However, a significant amount of iterations might be needed to arrive at a converged solution and in practice convergence isn't always reached.

It it important to note that the order of the constraints is important. Since the last constraint to be processed is sure to be satisfied (i.e. no other constraint can come in and invalidate it), it possible to prioritize the constraints by ordering them. When processing the more important ones last, they are more likely to be satisfied (Bender *et al.* 2015).

As is rightfully pointed out by Macklin *et al.* (2014) and Macklin and Müller (2013), this process is inherently sequential. All constraints need to be processed in the same order every iteration, and multi-threading is therefore difficult. Macklin *et al.* (2014) solve this problem by working with Jacobi iterations rather than Gauss-Seidel iterations. This means that rather than updating the position immediately after every constraint, the corrections Δp are all computed based on the same position and accumulated before actually applying them to the particle at the end of the iteration. This allows PBD to take advantage of the computational power of modern computers with multiple cores.

Macklin *et al.* (2014) note that accumulating the corrections may lead to situations that will not converge to a correct solution. For this reason, the corrections are averaged in every iteration, and the total correction Δx_i for particle i in one iteration becomes:

$$\Delta x_i = \frac{\Sigma_j \Delta x_{ij}}{n} \tag{3.10}$$

where n is the amount of constraint that affect i and Δx_{ij} is the correction of the position of i by constraint j (with j going from 1 to n). This averaging is sometimes too severe, bringing down the convergence rate. For this reason, Macklin *et al.* (2014) proposes introducing over-relaxation to Equation(3.10):

$$\Delta x_i = \frac{\omega}{n} \Sigma_j \Delta x_{ij} \tag{3.11}$$

where ω is a user-defined parameter controlling the amount of over-relaxation. Macklin *et al.* (2014) reports $1 \leq \omega \leq 2$ to work well.

An issue with Jacobi iterations is that it makes it impossible to prioritize the constraints, as their processing order is now no longer guaranteed. To allow prioritized constraints, Macklin *et al.* (2014) groups the constraints. These constraint groups are ordered according to their priority, parallelized separately and the position of the particles is updated after each group rather than after every iteration.

Algorithm Overview The above considerations result in the algorithm as in listing 1 being implemented in the project. This pseudocode algorithm strongly resembles similar listings in Müller *et al.* (2007); Macklin *et al.* (2014); Bender *et al.* (2015).

```
foreach Particle $i$ do
    $v_i \leftarrow v_i + \Delta t f_{ext}(i)$;
    $x_i^* \leftarrow x_i + \Delta t v_i$
end
foreach Particle $i$ do
    find Neighbouring Particles $N_i(x_i^*)$;
    find Environment contacts;
    generate collision constraints;
end
for $i \leftarrow 1$ to $solverIterations$ do
    foreach ConstraintGroup $G$ do
        foreach Particle $i$ do
            $\Delta x_i \leftarrow 0, n_i \leftarrow 0$;
        end
        foreach Constraint $C$ in $G$ do
            foreach Particle $i$ affected by $C$ do
                Solve Constraint for i;
                Add result to $\Delta x_i$ and increment $n_i$;
            end
        end
        foreach Particle $i$ do
            $x_i^* \leftarrow x_i^* + \omega \frac{\Delta x_i}{n_i}$;
        end
    end
end
foreach Particle $i$ do
    update velocity $v_i \leftarrow \frac{x_i^* - x_i}{\Delta t}$;
    update position $x_i \leftarrow x_i^*$ or apply particle sleeping;
end
```

Algorithm 1: Simulation loop of Position Based Dynamics (pseudo-code)

Giving an overview of the steps in Algorithm 1:

- Lines 1-4: The external forces $f_ext(i)$ on the particles are integrated with explicit euler integration, with timestep Δt. Note that particle's position is not immediately updated. The resulting position is stored in a second variable called the predicted or proposed position x_i^* (Macklin *et al.* 2014). The particle's velocity v_i is updated as well.

- Lines 5-9: The particles are checked for collisions with other particles and environment objects. Collision constraints are generated accordingly, and temporarily added to the constraint groups.

- Line 10: The outer loop of the solver, where $solverIterations$ is the amount of Jacobi iterations to perform. The ideal value depends on the simulation and should be user settable.

- Line 11: As explained earlier, the constraints are grouped and the particles are fully processed per constraint group.

- Line 12-14: The accumulators of the particles, Δx_i and n_i, are set to zero.

- Line 15-19: The constraints are solved and the results added to the accumulators. Note that this loop is presented in the constraint-centric approach. One could also write a particle-centric loop (Macklin *et al.* 2014), provided one can easily find the constraints affecting one particular particle. Note that the particle-centre approach allows rewriting the three loops in lines 12-23 into one larger loop.

- Line 21-23: The particle's proposed position is updated with the averaged accumulated correction. Note that over-relaxation is applied as well, by multiplying with ω.

- Lines 26-29: The particle's position x_i and velocity v_i are updated with the computed proposed position x_i^*.

It is important to note that all of the calculations are done on a second variable, the proposed position, before they are applied to the actual particle's position. It is hence important to keep in mind to work with the proposed position when implementing constraints.

Macklin *et al.* (2014) gives one exception to the rule of always working with the proposed position. In between the application of the external forces and the constraint solve, they place a pre-stabilization. This means that the contact constraints are solved with the positions at the start of the time-step rather than the proposed position. Any deltas computed are applied to both the actual position and the proposed position, but the velocity is not updated. Adding this to the solver loop would mean adding the pseudo-code in algorithm 2 in between line nine and ten of algorithm 1.

```
for $i \leftarrow 1$ to $solverIterations$ do
    foreach Particle $i$ do
        $\Delta x_i \leftarrow 0, n_i \leftarrow 0$;
    end
    foreach Collision Constraint $C$ do
        foreach Particle $i$ affected by $C$ do
            Solve Constraint for i with $x_i$ instead of $x_i^*$;
            Add result to $\Delta x_i$ and increment $n_i$;
        end
    end
    foreach Particle $i$ do
        $x_i \leftarrow x_i + \omega \frac{\Delta x_i}{n_i}$;
        $x_i^* \leftarrow x_i^* + \omega \frac{\Delta x_i}{n_i}$;
    end
end
```

Algorithm 2: pre-Stabilization loop as by Macklin *et al.* (2014) (pseudo-code)

This pre-stabilization is to ensure that any unsatisfied constraints in the previous time-step can not propagate into a high velocity away from the contact, as explained in Figure 3.1. This method of pre-stabilization

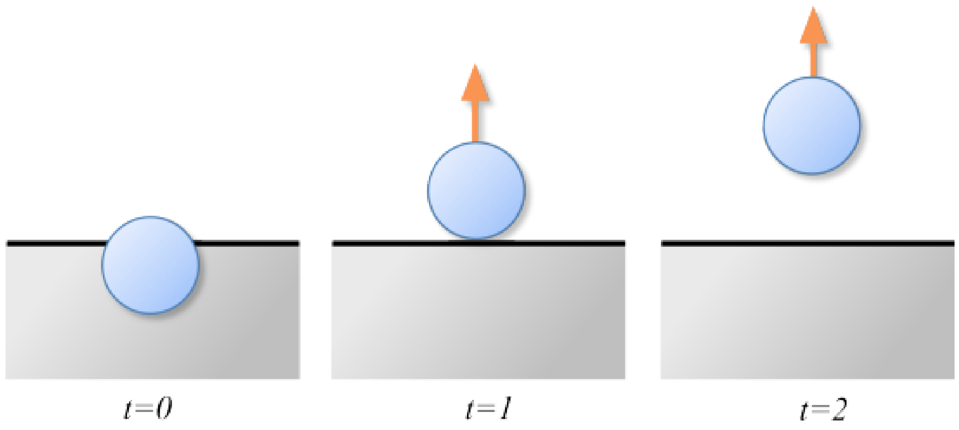


Figure 3.1: *This particle is clearly in an invalid state at timestep* $t = 0$. *Without pre-stabilization, this will be corrected in the following time-step, but the velocity will be upwards. Consequently, it seems to fly upwards for no reason at the following time-step (*$t = 2$*). Image by Macklin* et al. *(2014).*

removes a lot of the visible errors that would otherwise occur in a constraint solve with a low amount of solver iterations. This way it increases performance by allowing lower solver iteration counts. Pre-stabilization was also implemented in the project.

In the next section, the constraints are presented with x_i, but need to be computed with the proposed position x_i^*, as explained.

3.2 Constraint types

The framework explained in the previous subsection allows simulating a wide range of effects, provided we have the right constraints to simulate them. In this section, the constraints that were implemented in the project will be explained. For a full overview of existing constraints, we refer again to Bender *et al.* (2014).

Fixed Position Constraint We start out with the easiest example: a constraint that fixes a particle i at a particular position p. The easiest way to calculate the correction of the position x_i necessary is:

$$\Delta x_i = p - x_i, \tag{3.12}$$

so that $x_i + \Delta x_i = x_i + p - x_i = p$. As said earlier, in practice this will be calculated with the proposed position, but that is beside the question in this section.

However, we would like the calculations to fit Equation (3.8), which requires the calculation of λ and $\nabla C(p)$. This is again fairly easy as the direction $\nabla C(p)$ is simply the direction from x_i to p:

$$\nabla C(p) = \frac{p - x_i}{||p - x_i||} \tag{3.13}$$

and λ is the distance between them. However, λ also needs to incorporate the mass m_i of the particle, and be negated, to cancel out w_i and the -1 factor in Equation (3.8):

$$\lambda = -m_i||p - x_i||. \tag{3.14}$$

Substituting these into Eq. (3.8) gives the right answer again:

$$- - m_i||p - x_i||w_i\frac{p - x_i}{||p - x_i||} = m_i\frac{1}{m_i}(p - x_i) = p - x_i \tag{3.15}$$

This simple constraint can be of great value. By updating the goal position p every frame, one such constraint is a very simple means of animating a particle. This is particularly useful for simulating cloth and strands attached to an animated object.

Distance Constraint A step up from the fixed position constraint is the distance constraint. This constraints mimics a linear spring force in the classic mass-spring model: its goal is to keep two particles p_1 and p_2 on a particular distance d.

As pointed out by Bender *et al.* (2015), such a constraint can be formulated as follows:

$$C(x_1, x_2) = |x_1 - x_2| - d. \tag{3.16}$$

Here x_i is the position of particle p_i. Note that this constraint can be either used as an equality constraint, where the particles are supposed to be at that exact distance d, or as an inequality constraint, where the particles stay within a certain distance from each other (or a certain distance away from each other, depending on the formulation). In order to mimic the classic spring, it is usually implemented as an equality constraint, and so it is in this project.

Again, looking to evaluate Equation (3.8), we calculate the gradients with respect to the particle positions (Bender *et al.* 2015)

$$\nabla_{x_1} = \frac{x_1 - x_2}{|x_1 - x_2|} \tag{3.17}$$

and $\nabla_{x_2} = -\nabla_{x_1}$. Additionally we calculate λ as:

$$\lambda = \frac{|x_1 - x_2| - d}{w_1 + w_2}, \tag{3.18}$$

with w_i still the inverse mass of particle p_i.

It is interesting to add that, as is noted by Bender *et al.* (2015), filling these into Eq. (3.8) results in:

$$\Delta x_1 = -\frac{w_1}{w_1 + w_2}(|x_1 - x_2| - d)\frac{x_1 - x_2}{|x_1 - x_2|} \tag{3.19}$$

and

$$\Delta x_2 = \frac{w_2}{w_1 + w_2}(|x_1 - x_2| - d)\frac{x_1 - x_2}{|x_1 - x_2|}. \tag{3.20}$$

Which are exactly the corrections proposed for the spring constraint in the original PBD publication (Jakobsen 2001). It is important to note that these constraints need a large amount of solver iterations to appear stiff.

Collision Constraints This is more of a category than one particular type of constraint. First of all, most particles are not supposed to intersect with each other (although there are exceptions, e.g. a fluid). Second, most simulations happen in an environment, and the particles should respond to that environment in an appropriate manner.

Collision constraints are generated at the start of every simulation step, as can be seen in Lines 5 to 9 of the algorithm in listing 1. Before generating such constraints, the collisions need to be detected. For particle-particle interaction this is fairly easy, since the distance can simply be compared to the sum of the two radii. However naive implementations are usually too slow because of the $O(n^2)$ complexity, with n the number of particles. The fundamentals of spatial hashing techniques are not discussed as these are very common. However, such a technique was implemented for the project, as explained in the next chapter. For the collisions with the environment, the detection depends on the type of elements used for the environment. Typical examples are simple infinite planes and triangle meshes. Both were implemented for the project.

Once collision has been detected, a constraint is generated to solve the conflict. For particle-particle interaction, between particle p_1 and p_2, one can easily introduce an inequality version of the distance constraint from the previous paragraph (Bender *et al.* 2015):

$$C(x_1, x_2) = |x_1 - x_2| - (r_1 + r_2) \geq 0 \tag{3.21}$$

with x_i the position of particle p_i, and r_i the radius. In words: the distance between the two particles needs to be greater than or equal to the sum of the radii.

For the two forms of environment elements, the same collision constraint can be introduced (Bender *et al.* 2015). If the particle collides with an infinite plane or a triangle when moving from its position to its proposed position, an infinite plane constraint is generated, as this works for both cases. This constraint aims to keep the particle on the same side of an infinite plane throughout the time-step (Bender *et al.* 2015):

$$C(x) = n^T x - d_{rest} \geq 0. \tag{3.22}$$

Here x is the position of the colliding particle, n is the normal on the infinite plane and d_{rest} is the distance along that normal between the plane and the origin. Note that this distance should include the radius of the colliding particle on top of the distance in the plane's definition. Otherwise the particle's center will be kept at that distance.

Friction Constraint Most materials have some kind of friction when the interact with each other, and it should hence be included in simulation. Friction is particularly important for simulating granular materials, as their behaviour is largely due to the friction between their grains (Jaeger *et al.* 1996).

whereas some authors have included friction in PBD using a post-process (Müller *et al.* 2007), Macklin *et al.* (2014) proposed a constraint to provide friction. This places friction in the constraint loop, allowing for particle piles with high angles of repose, a typical phenomenon seen in granular materials.

Macklin's friction constraint is supposed to be processed after the collision constraints (i.e. placed in a later constraint group), and aims to take some of the relative tangential displacement away. They start by calculating said displacement between particles p_1 and p_2:

$$\Delta x_\perp = ((x_1^* - x_1) - (x_2^* - x_2)) \perp n. \tag{3.23}$$

x_i and x_i^* are the position at the start of the time-step and the current proposed position respectively. The vector n is the contact normal $\frac{x_i^*-x_j^*}{|x_i^*-x_j^*|}$, and the $\perp$ means only the part perpendicular to n is retained.

The correction for particle p_1 is then (Macklin *et al.* 2014):

$$\Delta x_1 = \frac{w_1}{w_1+w_2} \begin{cases} \Delta x_\perp & |\Delta x_\perp| < \mu_s d \\ \Delta x_\perp \cdot min(\frac{\mu_k d}{|\Delta x_\perp|}, 1) & otherwise \end{cases} \tag{3.24}$$

with μ_k, μ_k the static and kinetic friction coefficients, and d the interpenetration distance. The correction for p_2 is similar but with the weighting for w_2 and the result negated.

Rigid and Deformable Bodies Some different ways of simulating rigid bodies exist. A simple one that would easily fit the PBD framework is connecting vertices of the mesh with distance constraints. However as mentioned, these constraints need many iterations to be stiff. Stiffness if an absolute necessity for rigid bodies. Deul *et al.* (2014) introduce a position based RBD simulator but this doesn't fit in the particle-based framework we have in mind for this project.

Therefore, shape-matching constraints are used by Macklin *et al.* (2014). These were originally introduced by Müller *et al.* (2005). In this technique particles are generated evenly within the mesh to represent the shape. These particles are then simulated independently through all the other forces and constraints, with particle collision between particles of the same rigid body turned off. The shape-matching constraint then brings the particles back into the shape of the rigid body. It is clear the shape-matching constraint has a high priority and has to processed as one of the last constraints, to ensure the rigid body shape to be intact. This

Bringing the positions back into the shape involves finding the least squares mapping from the deformed state to the correct state. Macklin *et al.* (2014) use following correction for a particle i:

$$\Delta x_i = (Qr_i + c) - x_i. \tag{3.25}$$

Which means that the proposed position is interchanged with $(Qr_i + c)$, the goal position, putting the particle back in the original shape. Here c is the center of mass of the particles in the deformed state and r_i is the offset of p_i from the mass center in rest configuration. The matrix is Q is the rotational component of the polar decomposition of A, where A is computed as:

$$A = \Sigma_i^n (x_i - c) \cdot r_i^T \tag{3.26}$$

Note that $\cdot$ here represents the outer product, and this formula hence results in a 3 by 3 matrix.

In conclusion, this technique avoids having to store rotational values for the rigid bodies and (discrete) rigid body collisions are included automatically through the particle collision system in the PBD framework. It is hence a very simple way of implementing rigid body dynamics.

Due to the discrete collision detection, inter-locking may occur (Macklin *et al.* 2014). A simple way of improving this slightly is making the particles slightly overlapping, which is possible since collisions between particles belonging to one body are ignored during the collision generation. A better way of avoiding interlocking is introduced by Macklin *et al.* (2014), and comes down to storing signed distance field information on the particles. This technique was implemented in the project but not thoroughly tested. The current default of the system is to rely on the particle-particle collision detection.

As said earlier, shape matching was introduced by Müller *et al.* (2005). They intended the technique to be used not only for rigid bodies, but also for deformable ones. They provide different ways of adding deformations

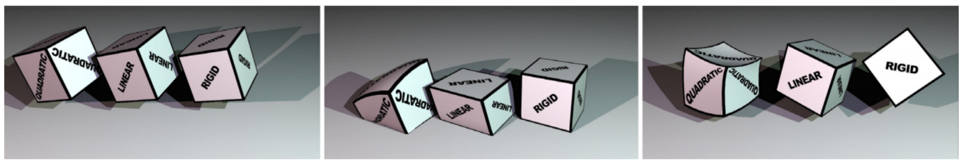


Figure 3.2: *Different kinds of deformations as introduced by Müller* et al. *(2005), compared to rigid body behaviour. The cubes are marked in accordance with their deformation behaviour (quadratic, linear or rigid). Image by Müller* et al. *(2005).*

to the above technique. The simplest one, which can only be used for small deformations, is introducing a multiplier α in Equation (3.25):

$$\Delta x_i = \alpha * ((Qr_i + c) - x_i). \tag{3.27}$$

The effect is that the particle will be placed somewhere in between its deformed and goal position. The factor α ($\in [0, 1]$) controls the amount of deformation. Note however that this very sensitive to the amount of solver iterations, as will be shown in the implementation as well.

Two type of deformations that allow bigger deviations from the base shape are presented by Müller *et al.* (2005) as well: linear and quadratic deformations. The difference between the two is illustrated in Figure 3.2. The linear one is again fairly simple, but can only represent shear and stretch. The matrix Q in Eq. (3.25) is replaced by a linear combination of Q itself and A, where A is:

$$A = (\Sigma_i^n (x_i - c) \cdot r_i^T)(\Sigma_i^n r_i \cdot r_i^T) \tag{3.28}$$

The combination of Q and A is controlled by another control parameter β, and the final matrix is $\beta A+(1-\beta)Q$. Again, β is in the interval 0 to 1.

Quadratic deformations, while visually more interesting, are mathematically a bit more involved. The goal position g_i (the position of particle i in the correct shape) is now found as follows (Müller *et al.*

2005)

$$g_i = \tilde{A}\tilde{r}_i. \tag{3.29}$$

Special attention needs to be paid to the dimensions here. g_i is a three dimensional position, and $\tilde{r}_i$ is a nine dimensional vector, based on r_i. r_i is still the offset of particle i to the center of mass in the rest position. If r_i equals $[r_{ix}, r_{iy}, r_{iz}]^T$, than $\tilde{r}_i$ is $[r_{ix}, r_{iy}, r_{iz}, r_{ix}^2, r_{iy}^2, r_{iz}^2, r_{ix}r_{iy}, r_{iy}r_{iz}, r_{iz}r_{ix}]^T$.

Consequently, $\tilde{A}$ is a three by nine matrix. As Müller *et al.* (2005) show, this matrix can be computed as:

$$\tilde{A} = (\Sigma_i m_i (x_i - c)\tilde{r}_i^T)(\Sigma_i m_i \tilde{r}_i \tilde{r}_i^T)^{-1}. \tag{3.30}$$

With all symbols as defined before. As with the linear deformation, the matrix $\tilde{A}$ is used in a linear combination $\beta\tilde{A} + (1-\beta)\tilde{Q}$. The matrix $\tilde{Q}$ is simply defined as $[QOO]$, with Q as before, and O the three by three zero matrix. Müller *et al.* (2005) note that this is a cheap version of modal analysis.

Fluid Constraint Macklin and Müller (2013) propose a constraint for simulating the incompressibility in fluids, based on SPH simulations. They introduce following equality constraint for every particle p_i in the fluid:

$$C_i(p_i, p_1, ..., p_n) = \frac{\rho_i}{\rho_0} - 1, \tag{3.31}$$

with ρ_0 the rest density in the fluid. The constraint C_i for particle p_i also influences the particles in the neighbourhood of p_i. These have been named p_1 to p_n. The density ρ_i at the p_i's position is calculated as:

$$\rho_i = \Sigma_j m_j W(p_i - p_j, h). \tag{3.32}$$

with W the SPH kernel, and h the kernel smoothing length. In (Macklin and Müller 2013), and in this project, the Spiky kernel is used for gradients and the Poly6 one for density estimation.

Macklin and Müller (2013) then go on to compute λ from the given constraint:

$$\lambda = \frac{C_i(p_i, p_1, ..., p_n)}{\Sigma_k |\nabla_{x_k} C_i|^2 + \epsilon} \tag{3.33}$$

where k ranges over all particles affected by the constraint (i and 1 to n). The term ϵ is added to counter instability problems, by relaxing the constraint. For ∇_{x_k}:

$$\nabla_{x_k} C_i = \frac{1}{\rho_0} \begin{cases} \Sigma_j \nabla W(p_i - p_j, h) & k = i \\ -\nabla W(p_i - p_j, h) & otherwise \end{cases} \tag{3.34}$$

With j ranging over the particles 1 to n in i's neighbourhood.

Macklin and Müller (2013) also include Tensile instability corrections and vorticity confinement but this was not implemented in the project.

Chapter 4

Software Design & Implementation

The previous chapter introduced the concepts and mathematics of the technique that were implemented for this project. This chapter will cover how these techniques were implemented. It will discuss the software design, explaining the major decisions that were made during the project, and the most important technical details of the project.

4.1 Class Diagrams

This section will show class diagrams of the project, which will then be discussed in the next section. The diagrams themselves are inserted into this report but will also be provided in image formats, to ensure that they are clearly readable.

The class diagrams were split up into smaller logical modules, and can be found in Figures 4.1 to 4.5. These diagrams are complemented by a glossary of the classes and their responsibilities, which can be found in Appendix A. For readability, it was preferred to keep the operation parameter names out of the class diagrams. Also worth mentioning is that some type names of class members may seem unknown. These are all type definitions to increase readability. Most common are instances where a certain classname is complemented with 'Ptr', indicating a smart pointer pointing to that class. In the same style, a class with the 'PtrVec' addition is a std::vector of such smart pointers. In any case, all type definitions are included in the aforementioned glossary.

4.2 Discussion

The core of algorithm 1 is abstracted into the class design seen in figure 4.1. In order to represent the different particles and objects consisting thereof, a Particle and ParticleObject class were designed. The different ParticleObjects are grouped in a ParticleScene. Hence, the simulation data is situated in the ParticleScene class (and its members). It is interesting to mention that a ParticleObject is more than just a collection of Particles. It also holds information on how to create new particles (if this is allowed: adding a particle to a rigid body is impossible, but adding one to a fluid simulation is common practice, hence the optional ParticleInfo member), and has some more functions ("preSim" and "afterSim"), that have to do with the extensibility of the library, and will be discussed in a later paragraph.

The SimulationController class ensures the particles in the ParticleScene are updated by the right steps, in the right order. The different steps of the algorithm are again abstracted away in their separate classes, in a style similar to a strategy pattern. This results in classes representing different parts of the algorithm: The ExternalForceSolver, Environment and ConstraintSolver classes.

The ExternalForceSolver and ConstraintSolver classes apply respectively ExternalForces and Constraints to the particles. Both of them keep track of which forces or constraints to apply, by means of a com-

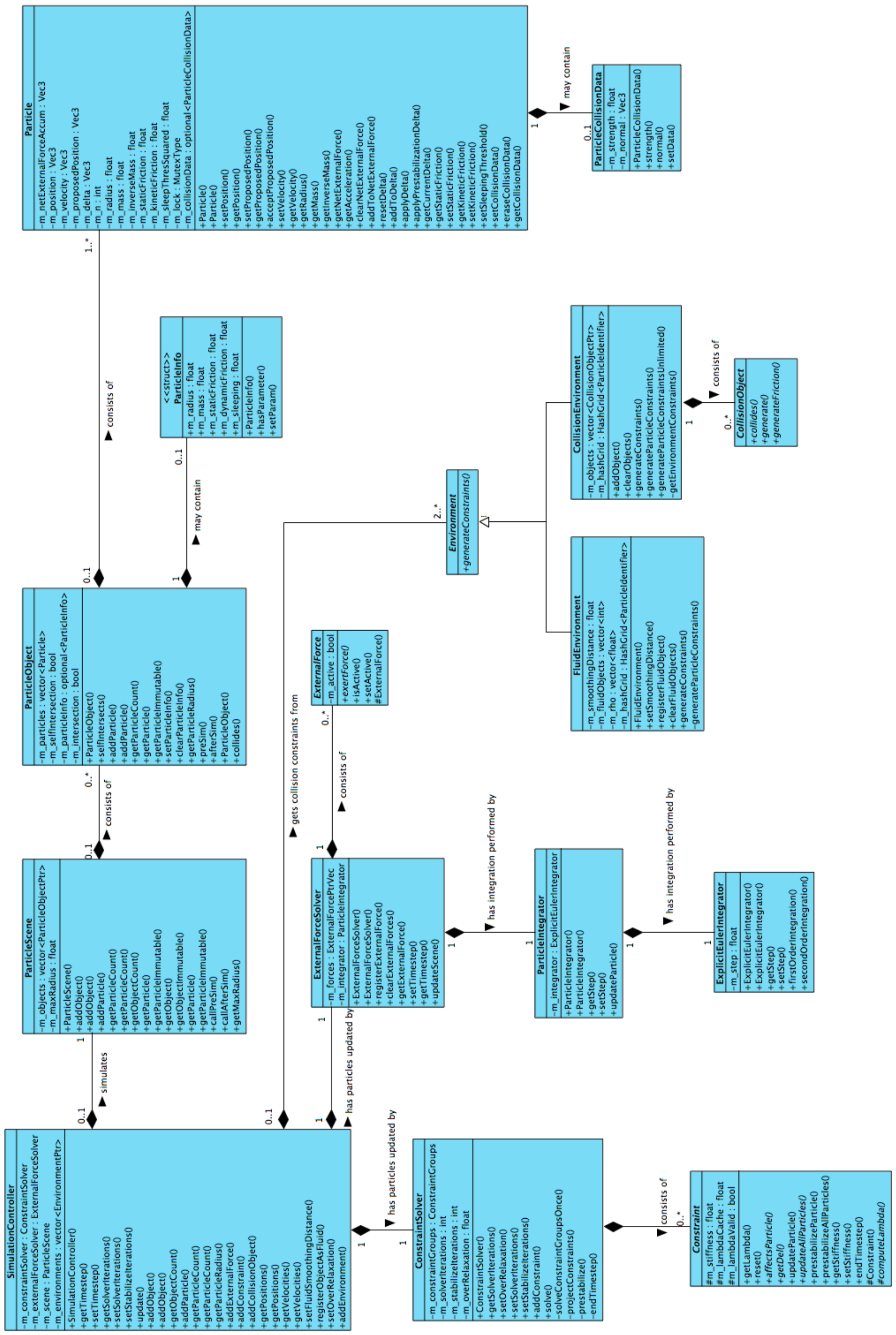


Figure 4.1: *Diagram: Core of the Position Based Dynamics Library*

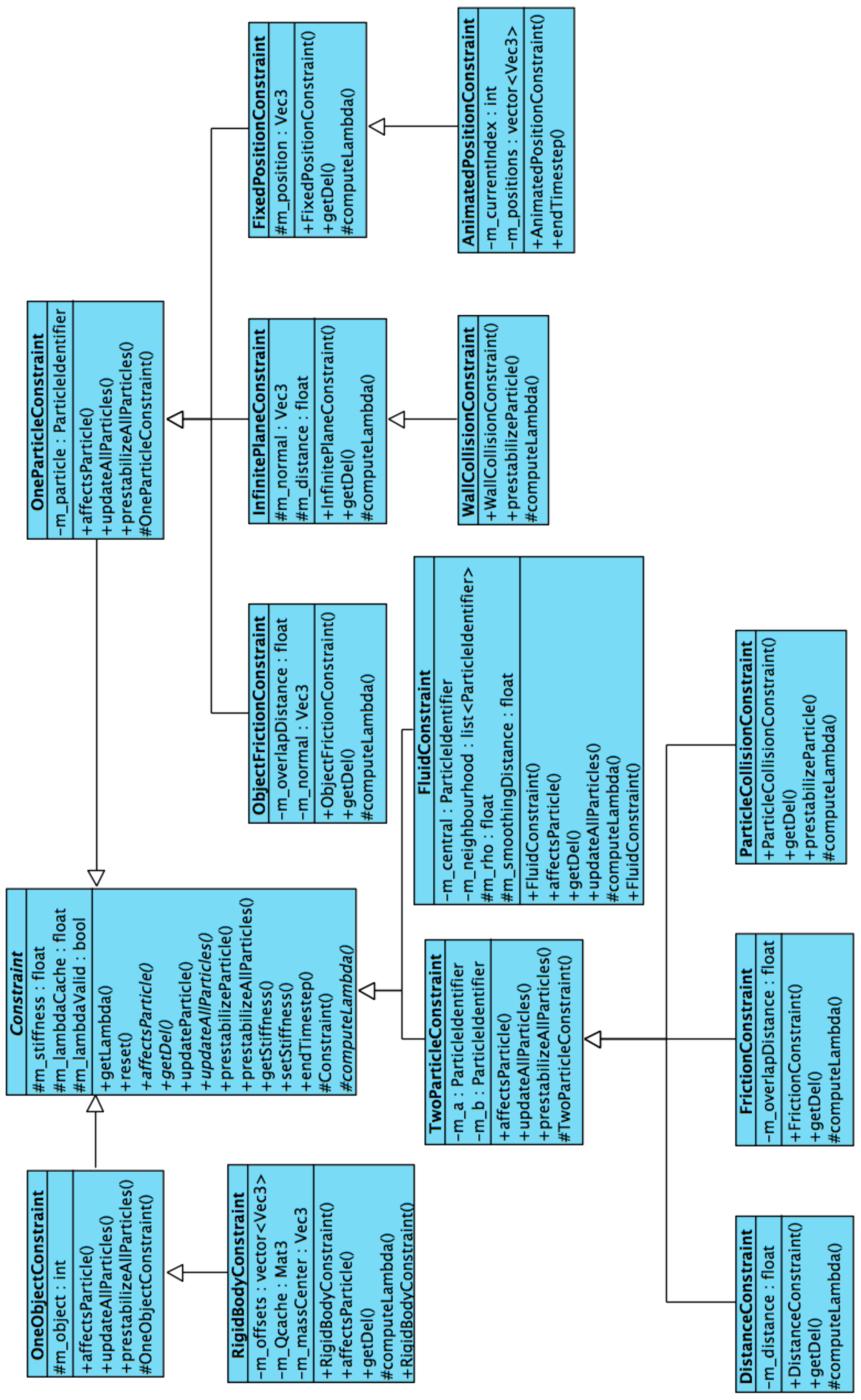


Figure 4.2: *Diagram: Inheritance diagram of all existing constraints.*

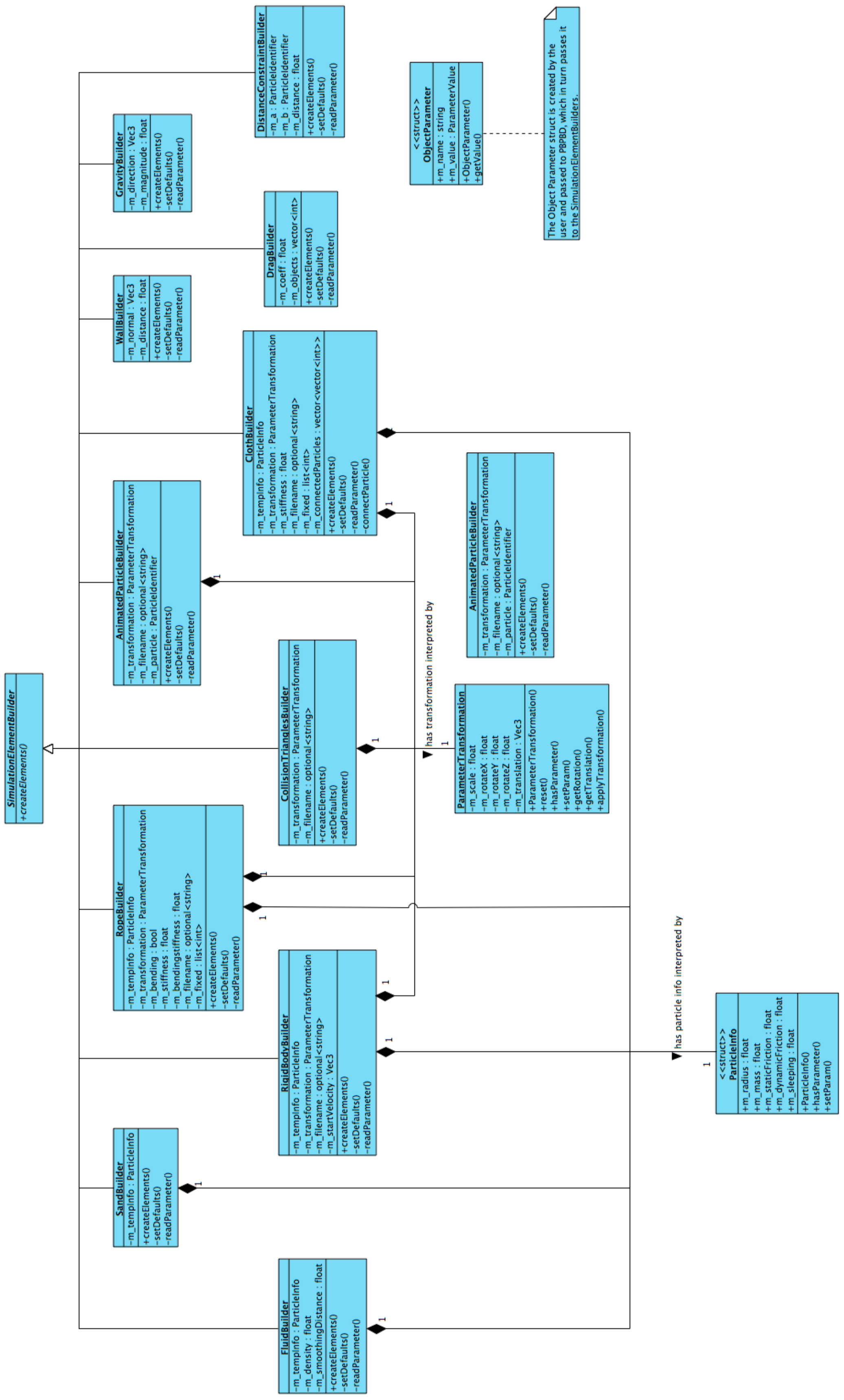


Figure 4.3: *Diagram: Inheritance diagram of all existing SimulationElementBuilders.*

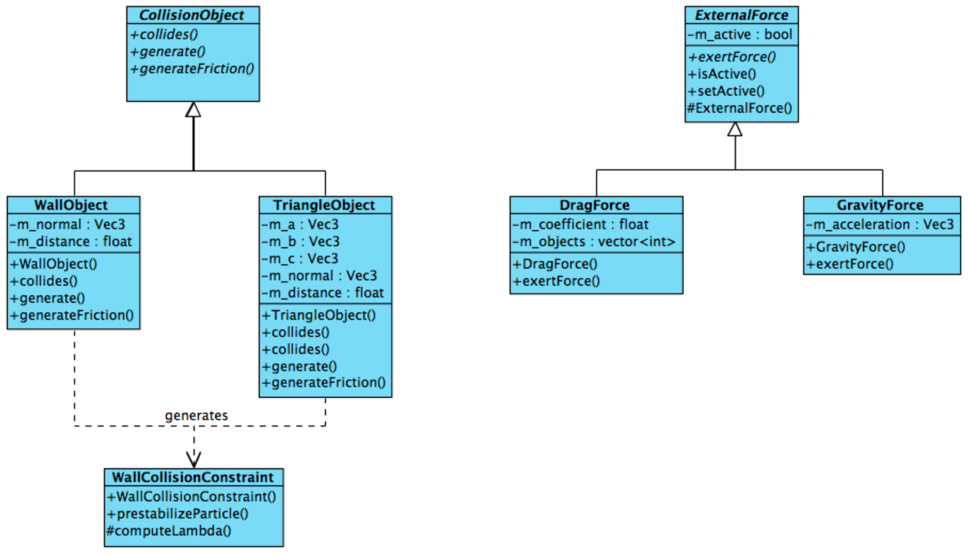
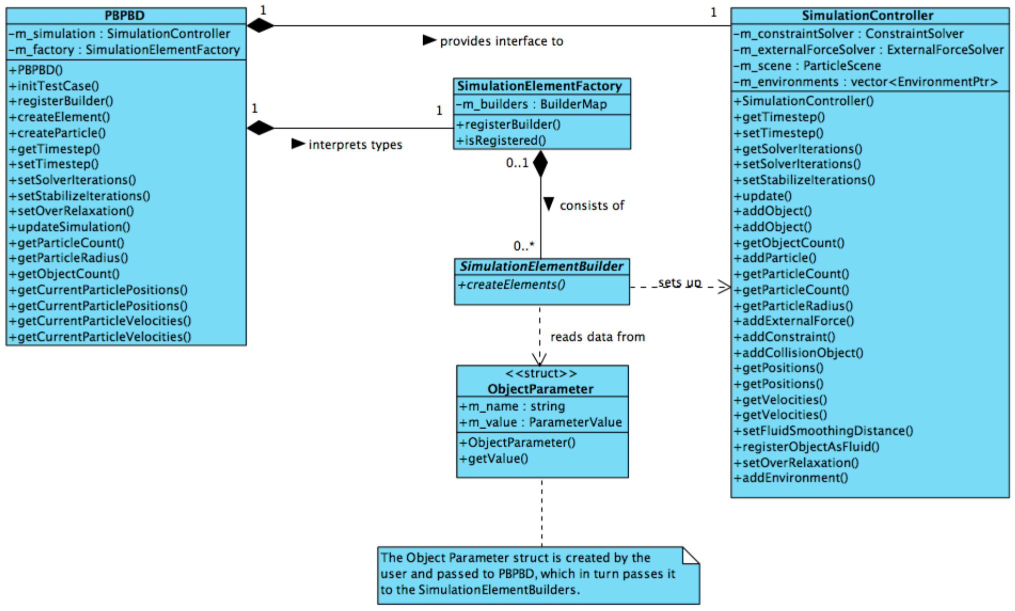


Figure 4.4: *Diagram: Inheritance diagram of ExternalForces and CollisionObjects.*



The Object Parameter struct is created by the user and passed to PBPBD, which in turn passes it to the SimulationElementBuilders.

Figure 4.5: *Diagram: Class diagram of class involved in setting up simulations.*

mand pattern: ExternalForce and Constraint are abstract classes, and any child class can be added the their respective solvers. This use of inheritance will be one of the key parts in the extensibility of the library, as explained later. The different forces and constraints that were mentioned in the previous chapter are all included in the library, and specific class diagrams can be seen in Fig. 4.4 and Fig. 4.2, respectively.

Environments are classes that calculate temporary constraints. These are constraints that are generated per time-step, solved and then forgotten. The most prominent example are collision constraints, which are computed by the CollisionEnvironment class (both collision between particles and with the physical environment). However, it might be possible in the future one wants to add different collision constraints or improve the performance, so it is possible to add new types of Environments to the SimulationController. This completes the strategy pattern.

In fact, the project contains such a different Environment: the FluidEnvironment, which calculates the neighbors of fluid particles every timestep. The CollisionEnvironment and FluidEnvironment are defaulted into the SimulationController, but can be turned off or replaced.

The CollisionEnvironment works with CollisionObjects for the physical environment. These objects know themselves when a particle collides with them, and know what kind of collision constraint to generate. Abstracting this knowledge in a separate class, combined with an abstract CollisionObject interface, allows adding more types of environment objects in the future, without having to change the CollisionEnvironment. Currently the library contains a triangle CollisionObject and an infinite plane one.

Note that the Constraints have two layers of inheritance, as Constraints have to know themselves what particles to work on. Several types of constraints work on one or two particular particles, or one particular

ParticleObject, and hence the classes OneParticleConstraint, TwoParticleConstraint and OneObjectConstraint were made.

The SimulationController is a facade pattern, providing a clear interface to controlling a particular simulation. There are functions to add forces, constraints, objects and particles, and functions to run the resulting simulations. However, one still needs to build these elements before they can be added and simulated. Ideally, one does not want to change the core of the code (Fig. 4.1) when a new type of Constraint or Force is added, and hence creation of these elements was kept away from the core.

For the creation and set-up, a software design was made that provides both an easy and uniform way of setting all kinds of simulations and easy extensibility when new types of elements are implemented. The resulting design can be found in Figure 4.5. The whole library is accessible through one simple class (again a facade pattern), called PBPBD, also the name of the library. Setting up simulations is done by telling this class what type of objects to make and to give the according parameters. These types are registered with the factory with a typename (std::string). The types of objects included in PBPBD by default, and their parameters, are specified in appendix B. The parameters are structs called ParameterObjects, which are combinations of a name and boost::Variant, and every element type accepts custom parameters of different base types. The library reads these in, and then simulates, giving back positions and velocities. This way, the library is not connected to one particular application: different applications can be written using the library. In other words, the library respects the Model-View-Controller pattern, with the PBPBD class being the controller, and the library itself the model.

As said, the system was meant to be extensible. For this reason, the SimulationElementFactory was created (see Figure 4.5). This object allows a new implemention of the class SimulationElementBuilder to register with a typename, and calls their buildElement method when their typename is called upon. This is easily recognizable as a Command Pattern and Factory Pattern. The factory is accessible through the PBPBD interface. These SimulationElementBuilders can create any kind of element and add it to the SimulationController. The parameters of the Builders are the earlier mentioned ParameterObjects, allowing them to have custom parameters. The default types in the library all correspond to one of the the default Builders (see Figure 4.3), which are all are registered with PBPBD on creation. The default Builders are specified in appendix B.

It is clear from this design that it makes the library easily extensible: a programmer wishing to extend the default library can create his or her own elements (Constraints, Objects, Environments, CollisionObjects, ExternalForces, ... almost everything) and then complement these new elements with one or more SimulationElementBuilders, possibly with custom parameters, creating the particular elements and adding them to the SimulationController.

Using this way of extending, implementing new elements does not mean having to change the core algorithm, as they can easily be added to the existing SimulationController thanks to the design. The new SimulationElementBuilders can simply be registered with PBPBD before they are used, and PBPBD will still work as normal but with added functionality. Hence the open/closed-principle is very well respected in the library design: no piece of existing code has to change in order to extend the library with most functionality imaginable. Since Position Based Dynamics relies on different types of constraints, being flexible towards new types of constraints is key for a PBD library. The author believes that his software design provides this flexibility.

Additionally, ParticleObjects have the functions "preSim" and "afterSim". In the default version these do nothing, but they are called for every ParticleObject right before the constraint solving loop and right after respectively. Adding new types of objects that have calculations that somehow not fit into a constraint (e.g. calculating normals) becomes possible this way. Again, these custom objects would then be added to the simulation with a custom builder. Similarly, the constraints have a endTimestep function that is called after every contraint solve. This is key for constraints that change throughout time, such as the Animated-PositionConstraint.

4.3 Implementation

The above software design was implemented in C++, using NGL for its 3D vector and matrix classes, and obj mesh loader. Several elements of the boost library were used as well, mainly it's smart pointers, the property tree class for xml parsing, the unordered map container as the provided environment did not include C++11, and the boost optional and variant types. The Eigen library was used for some of the matrix operations, most importantly the polar decomposition.

performance For a physical simulation library, performance is important. One does not want to wait endlessly for the results of his simulation. Especially for PBD, which was meant for interactive applications, performance is key. Because of this, a large fraction of the development time of the project was spent on improving performance. While focusing on this, an iterative workflow was used.

Using the valgrind function profiler, the code was profiled with a set of testcases. The profiler results were inspected to find the current bottleneck. After finding it, the bottleneck was improved or avoided. Once succeeding in the alleviating the bottleneck, we went back to profiling.

Giving an exhaustive list of all changes made in this iterative process would not be very worthwhile. However, one part of improving performance was quite substantial and is definitely worth explaining in more detail. Since the PBPBD library depends completely on particle-based physics, a lot of particle-particle interactions need to be computed. Detecting these interaction is usually a bottleneck in particle projects, especially since naive implementations have a squared complexity in the amount of particles, which is unacceptable.

To avoid this bad complexity, a hash-grid was implemented, somewhat separate (i.e. in a separate project and hence reusable) from the library. To avoid unnecessary delays, choosing the right containers for this hash-grid implementations was key. The hash-grid needs to be rebuilt every time-step, and the main operation is to find all particles in one cell and iterate over them.

Consequently, the hash-grid is implemented with an unordered map (from the boost library). The keys are cell ids, the values are the cell contents, which are represented with linked lists. Because of this combination, adding particles to the hash-grid has a constant complexity, as the map is unordered and there is no need for re-allocation when adding to a linked list. Look-up into a unordered map is also constant. The linked lists form no problem since all found particles need to be iterated over in the collision environment. Both operations need to happen for every particle every time-step, and hence the collision generation step has a linear complexity in the amount of particles.

Multi-threading As was explained in the introduction, one of the goals of the project was to take advantage of the multiple cores most computers have nowadays. However, before parallelizing, one has to ensure thread-safety of the operations in the code (Pacheco 2011).

As mentioned earlier, Macklin *et al.* (2014) rework PBD to Jacobi iterations to make parallelizing easier. Hence making the algorithms thread-safe was relatively easy. Adding computed corrections to the accumulated delta of a particle needs to be thread-safe, as the algorithm is implemented (and parallelized) in a constraint-centric fashion (Macklin *et al.* 2014). A simple per-particle mutex was added. To parallelize the hash-grid look-ups, the look-up functions were made thread-safe. This was done by simply using const marked functions of the boost unordered map.

After ensuring thread-safety, the code was multi-threaded using the OpenMP library. Looking at the algorithm in listing 1, the following elements were multi-threaded, as they were the most important bottlenecks. The line of the particular for loop in algorithm 1 is added between parenthesis:

- Finding the neighbors of each particle. (Line 5.)
- Pre-Stabilization. (Line 5 in Algorithm 2.)
- Solving the constraints. (Line 15.)
- Updating the proposed positions. (Line 21.)

Note that synchronization is necessary between Line 15 and 21, as all constraints need to be solved before updating the proposed positions. The result of the effect of multi-threading is discussed in the next section.

Something that came up in discussions with people in the VFX industry is the requirement for physical simulations to be robust. By this, they meant that a simulation should have the same result independent of the amount of cores it is calculated on. In theory the PBD algorithm as explained above is robust in this sense. However, since corrections (and forces) are accumulated, and the order of accumulation is non-deterministic, the results may change on a different number of cores due to a changing order of floating point additions.

While initially small, such errors can propagate and give an entirely different result. A possible solution is to keep track of all corrections in an ordered container, and then adding them together when the proposed position is updated in the same way. However, this over-head was deemed unnecessary, especially for a technique that is meant to be interactive. Hence, the results of the library in its current state might change due to floating point rounding errors when ran on a different amount of cores.

Basic Application For demonstrating the libraries capabilities, a basic application was made. This basic application reads in XML-files and adds the corresponding elements through using the PBPBD interface only (as it is intended to be used). The syntax for the XML files can be found in appendix C, and is very similar to the default element types accepted by PBPBD and its SimulationElementFactory. The basic application writes out the resulting particle information into a file per frame.

This can be read into SideFX' Houdini using a set-up as seen in Figure 4.6. Several examples of such houdini files are included in the hand-in. Since most SimulationElementBuilders can work with OBJ files, a simulation can also be set-up in Houdini (or any other 3D package) and exported into OBJs. These OBJs can then be referenced in the XML files of the basic application. Currently, these XML files are made manually, but a direction of future work is generating these straight out of the 3D package, or even better, calling PBPBD straight from a plug-in for a 3D package.

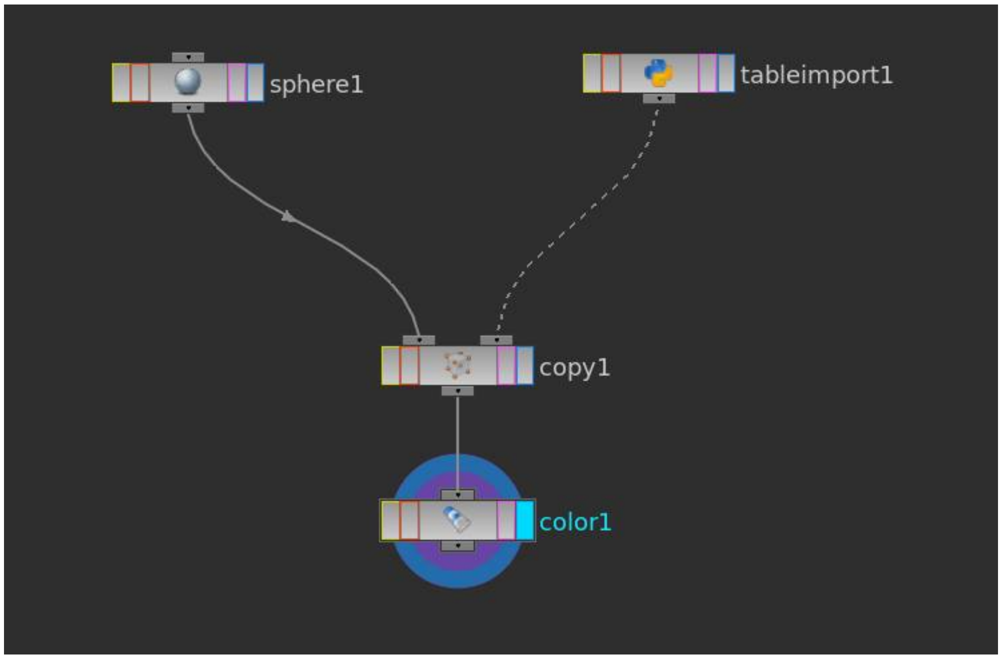


Figure 4.6: *The Houdini set-up for reading in results.*

Chapter 5

Results & Discussion

This chapter will review the final results of the project. It will discuss the functionality included in the project and shows some screenshots of how they look. However, since the project concerns physical simulation, videos are far more important than one frame. Hence, the original videos have been included in the hand-in separately, in the 'videos' folder. After reviewing the visual results in the first section, the next section will go on to discuss the performance of the system. The following section will then review the current limitations of the technique and system, and outline directions of future work. The final section describes an improvement to the current Position Based Dynamics technique, building on the discussion in the section before.

5.1 Included effects & functionality

As was mentioned in the previous chapters, constraints were implemented for the following types of effects:

- Rigid Bodies
- Deformable Bodies
- Ropes and Hair

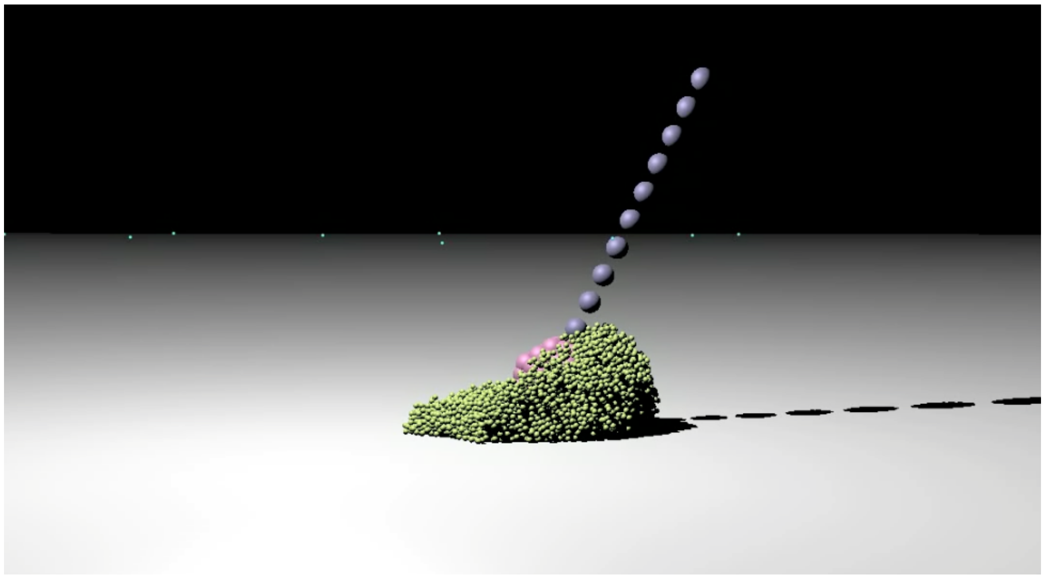


Figure 5.1: *A rigid body pulled by an animated rope, moving through a pile of sand.*

- Cloth
- Free flowing particles, such as Granular Material.
- Fluids

Most of these were extensively tested during the project. Resulting videos can be found in the Hand-in (videos folder). These have been complemented with functionality to set-up simulations with these effects working together in different ways. An exhaustive list of all functionality that is included by default in the library is given in Appendix B, and Appendix C shows how to use this functionality in the form of XML-files through the basic application. Figures 5.1 through 5.5 show some of the effects in action.

5.2 Performance

As was mentioned earlier, parts of the library were parallelized using OpenMP. This section reviews the performance gain this induced for the kind of simulations that were tested during the project.

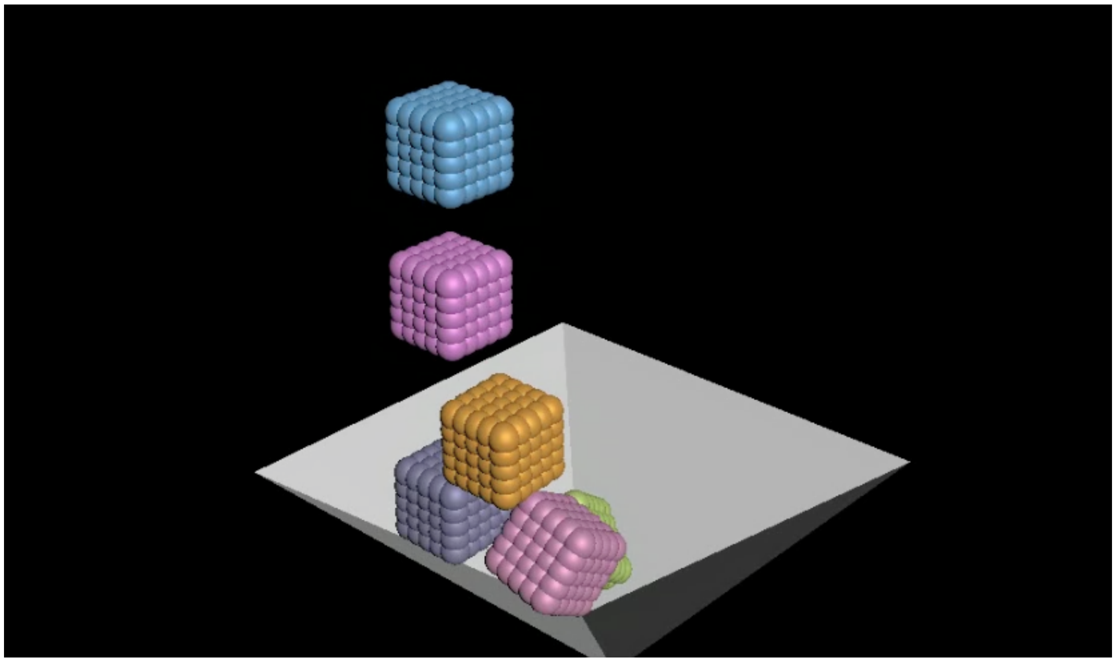


Figure 5.2: *Six rigid bodies colliding with an object consisting of triangles.*

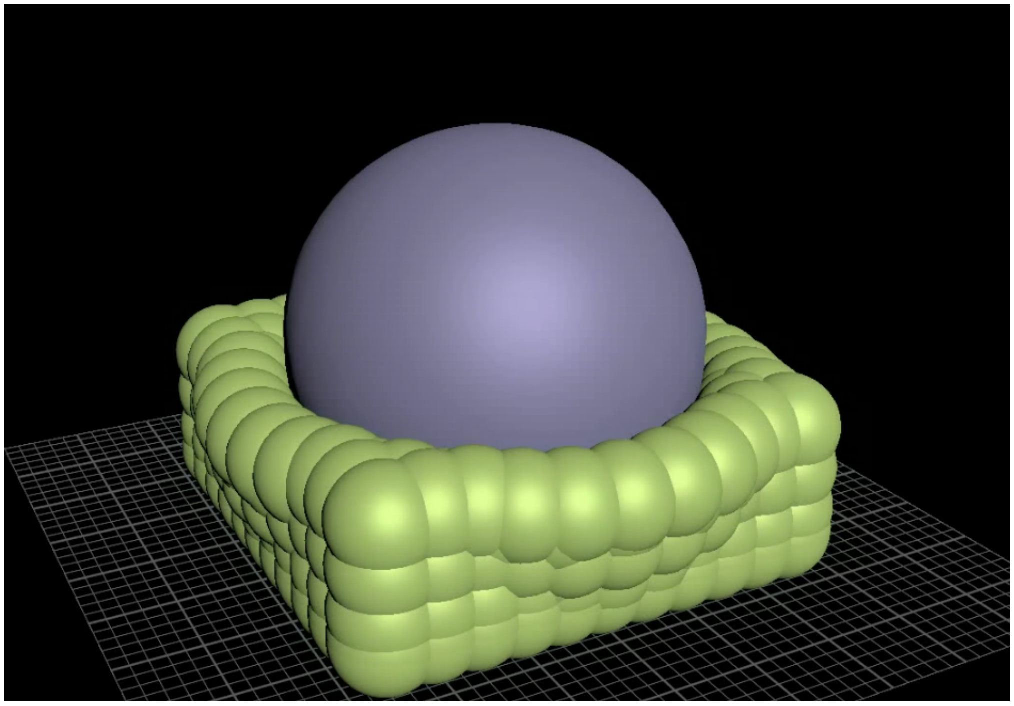


Figure 5.3: *A body deforming under the weight of another object.*

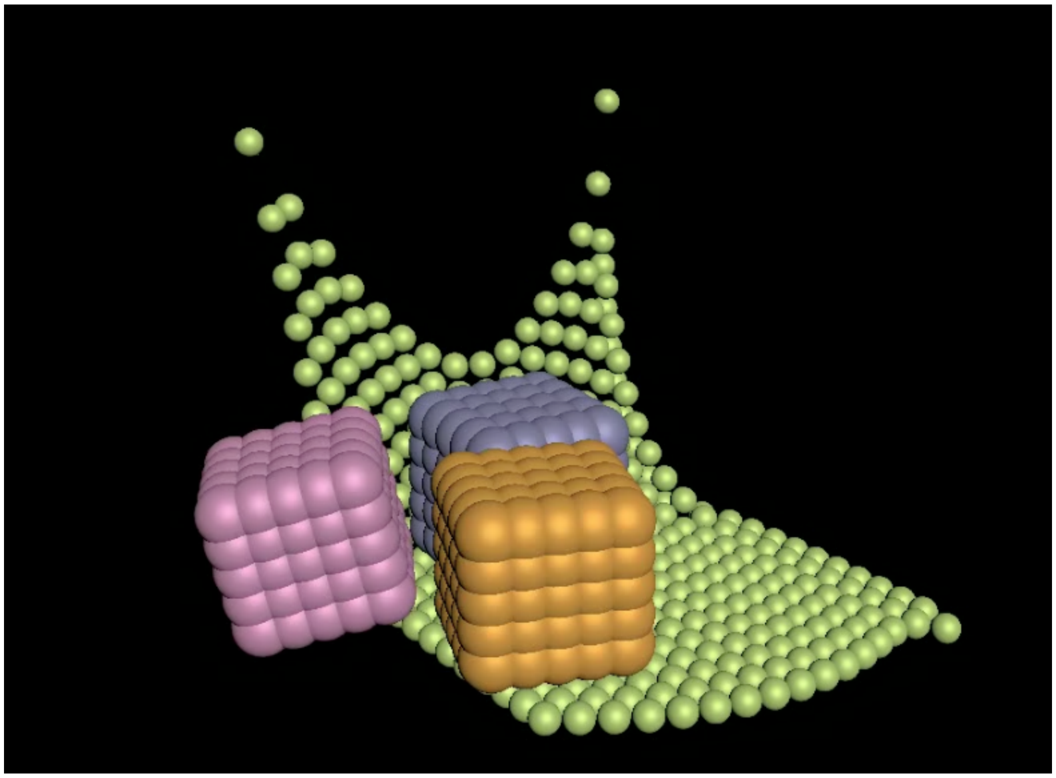


Figure 5.4: *Rigid bodies falling onto a piece of cloth.*

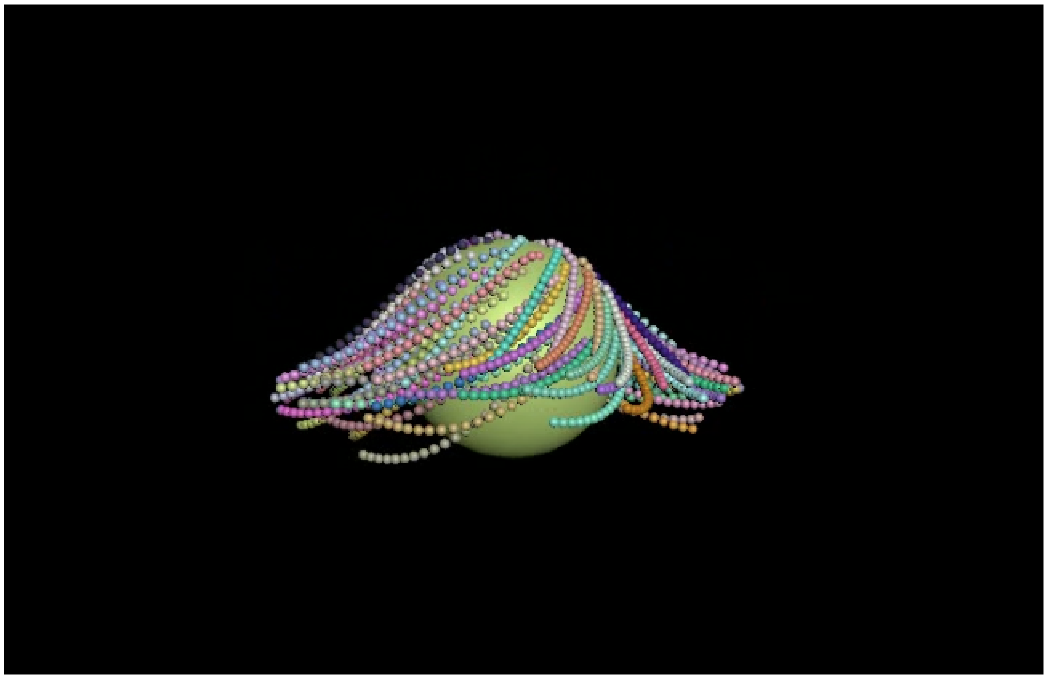


Figure 5.5: *A set of hairs connected to a head (a sphere).*

To compare the performance, three test-cases were run both single-threaded and multi-threaded. The three test-cases are:

- The hair simulation as seen in Figure 5.5. This was simulated for 251 frames, with 100 solver iterations and 25 stabilization iterations. This simulation has close to 700 particles.
- The deformation of one deformable body similar to the one in Figure 5.3, but with a less coarse particle packing for the body (about 4000 particles). This simulation was run with 70 solver iterations and 2 stabilization iterations, and was run for 201 frames.
- The sand pile simulation as in Fig. 5.1, with 70 solver iterations and 10 stabilization iterations for 301 frames. This simulation nears 10000 particles.

These test-cases were chosen because they represent different cases: the first is a simulation with many constraints that work on a small amount of particles (hair consists of a lot of distance constraints), where as the second as is a deformable that works with one body constraint that takes care of them all. The last one is a mix of the two, since there are many free-flowing particles (the sand), but also has a rigid body. The reason the choice of test-cases is inspired on the amount and type of constraints is that the simulation loop is implemented and parallelized in a constraint-centric fashion, as explained in chapter 3. Hence, we mainly want to investigate the impact of the decision to implement this constraint-centric loop over a particle-centric. An overview of the results can be seen in table 5.1.

As we can see in the table, the results show very little relation to the type of constraints. While this is a little bit surprisingly, there are some different factors explaining this. First of all, the lambda in constraints is cached. This means that for the Rigid (or deformable) Body Constraint, the heaviest part of the calculation is done only once and the per-particle calculation is fairly lightweight.

Test-Case	**Single**	**Multi**	**Improvement**
Hair	24.64	12.17	49.39%
Deformation	114.62	38.69	33.76%
SandPile	484.69	48.79	17.49 %

Table 5.1: *Performance improvement by multi-threading for the three different test-cases. The second column shows the (averaged out) wall time for the test-case when single-threaded, the third column shows the same but for the multi-threaded version. The last column shows the division of multi-threaded by single-threaded (in percentage).*

Furthermore, there is a considerable amount of collision constraints in every test-case. Even in the deformation test-case, a great deal of the deformable body particles collide with the floor or with the mass falling on top of it. Collisions also mean more work in neighbourhood searches, which are also parallelized. In the hair case, the particles are clearly less densely packed than in the other cases, giving the other two cases an edge. Hence, the collisions make the rigid body constraint less important in the general picture. This is particularly true for the SandPile case, since it has about 9000 sand-particles in a pile, very closely packed together. As a result, the amount of collision constraints is gigantic, making this a very good case for multi-threading.

We can conclude that the main factor determining the effectiveness of multi-threading is the scale of the simulation. This can be expected, since the overhead of multi-threading becomes relatively smaller as the amount of particles increases. Hence, the decision to implement a constraint-centric solver doesn't seem to have a very big effect, and the parallelization works very well as the scale of the simulation increases. It would be interesting to see if a particle-centric solver could improve performance, however this would add the overhead of keeping track of what constraints work on particular particle.

5.3 Limitations & Future Work

This section discusses the currently known problems of the library. These are divided in two categories: First the section will cover problems inherent to the position based dynamics technique that are visible in the library results. Afterwards the section will cover problems of the library itself, which will mainly be missing functionality. To complement these two categories, the section will point out with directions for future improvements to the PBD technique and project where possible.

Position Based Dynamics An issue that is often mentioned in the PBD research literature and that was found in the implementation, is the dependency of the results on the amount of solver iterations (Macklin *et al.* 2014). During the project, this dependency was found to be understated by most authors. Most of the academic literature does not fully convey the importance of the amount of solver iterations. Most parameters are meaningless since their results is so dependent on this. An excellent example are the friction constraints: the friction coefficients are usually completely overruled by the amount of solver iterations. Another very good example is the setting of α for small rigid body deformations, which will become almost meaningless with an high iteration count. The dependency on this parameter is illustrated in Figure 5.6. To illustrate this even more, the hand-in contains some videos of test-cases repeatedly run with the same settings except for an increasing number of solver iterations. These can be found in the folder 'iterationVideos'. Müller *et al.* (2007) propose using non-linear constraint weights (called constraint stiffness earlier). This reduces the effect slightly, but does not remove it.

Furthermore, the amount of solver iterations is, in all known literature, the same for every type of constraint. As a result, the amount of solver iterations depends on the weakest link in the simulation. For example: rigid bodies usually dont need many iterations due to their

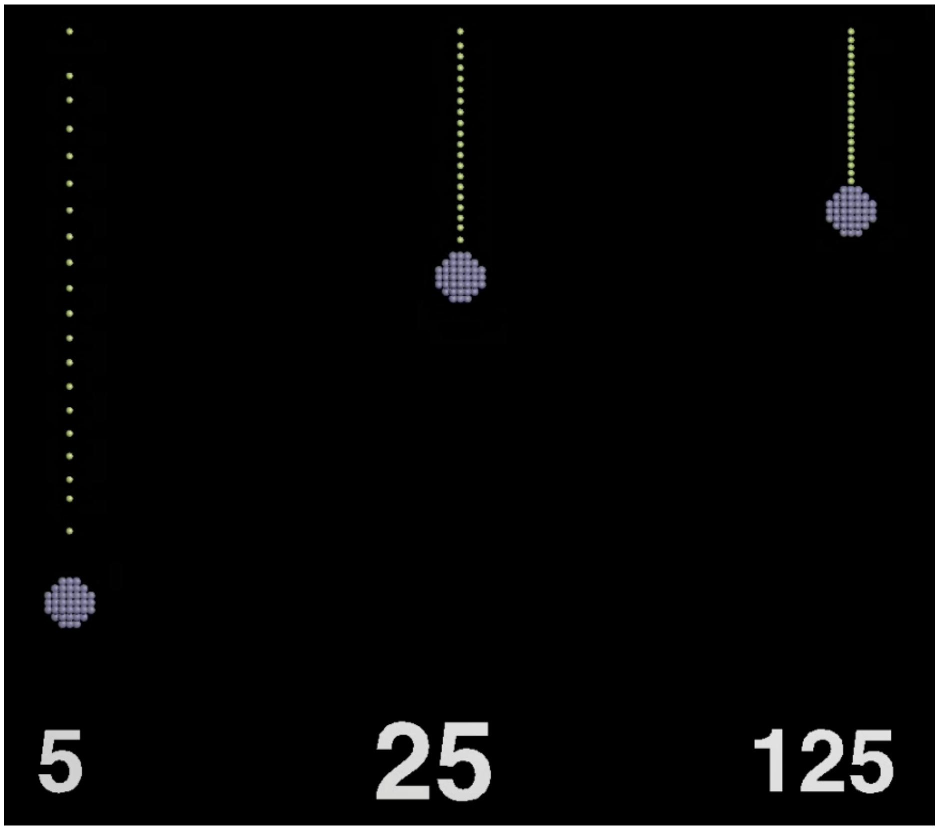


Figure 5.6: *A rigid body suspended on a piece of rope. Even with the stiffness set on the maximum, the actual stiffness largely depends on the amount of solver iterations. For each rigid body, the amount of iterations is shown below it. The rope stretches a lot with low amount of solver iterations.*

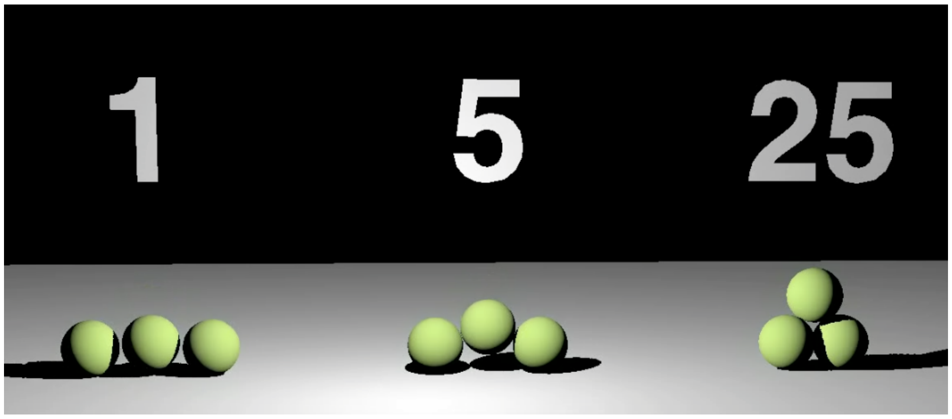


Figure 5.7: *Three free-flowing particles fall onto each other, three different times. With the same friction settings, the friction is still different in each case due to the amount of solver iterations (shown above the three cases).*

well working, high-priority shape matching constraints. Cloth and rope on the other hand need many iterations to appear remotely inextensible. Hence, a simulation containing both cloth and rigid bodies will need to have a high amount of iterations, to ensure a decent cloth simulation. These iterations spent working on the rigid bodies will be a wasted effort. Furthermore, the effect of an high iteration count might not be suitable for every effect. Consider Figure 5.7, where it is clear the amount of friction also depends on the iteration count. If one wants a medium amount of friction but inextensible cloth or rope, choosing a suitable amount of iterations will be very difficult, if not impossible.

A possible solution for this would be to have different amounts of solver iterations per type of constraint. This would obviously not work in a naive way where some constraints are just dropped after a certain amount of iterations. However, some constraints could be periodically solved. For the cloth-RBD example: after 4 iterations of solving all constraints except the shape matching (RBD) ones, we could re-include those for the next iteration. The shape matching constraint should also always be included in the final iteration. To my knowledge, this idea has not been presented in any academic literature, and it would be interesting to experiment with this idea. An initial implementation of this was

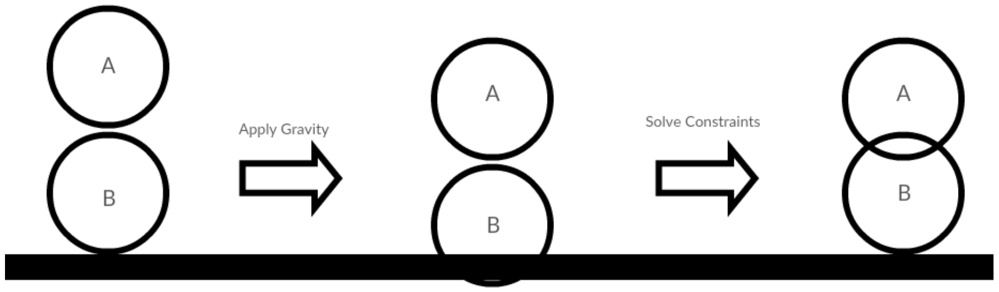


Figure 5.8: *Two particles, situated above each other but not touching, fall onto a floor. The external forces are applied, including gravity, bringing the lower particle B in contact with the floor. Hence a collision constraint is generated for B and the floor. However, A and B are not colliding in their proposed positions and no constraint is generated to resolve their eventual collision. The frame ends on in an invalid state and the collision is only resolved in the next frame. It is easy to see this becomes worse when adding another particle above A.*

made and tested in this project. This is the subject of the next chapter.

Another problem found to be inherent to PBD is a one-frame delay in collision detection due to the collision detection happening in between the external forces and the constraint solve. This is explained in Figure 5.8: two particles, one positioned above the other, falling onto the floor will overlap for at least one frame, since there will be no collision constraint between in the first frame. This is made worse when multiple particles are positioned above each other (i.e. each additional particle will take another frame to be detected), and the collision will propagate upwards. This is why this delay is called slow error propagation by Bender *et al.* (2015). This behaviour can result in stacks of supposedly rigid particles showing spring-like behaviour. Without pre-stabilization this even leads to particles being shot upwards, as was mentioned by Macklin *et al.* (2014). However, they fail to mention that the necessary amount of pre-stabilization also depends on the amount of stacked particles. Especially for piles of granular materials, a large amount of pre-stabilization iterations turned out to be necessary to avoid particles shooting out of the pile.

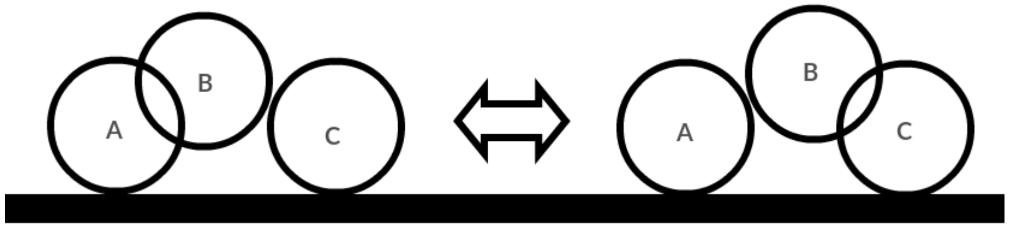


Figure 5.9: *The one-frame delay pushes particle B back and forth between A and C, especially if there is a lot of friction, since friction keeps A and C in the same place. This causes the particle to oscillate.*

In certain situations this one-frame delay causes one particle to oscillate between two positions. An example situation is explained in Figure 5.9, where a particle jumps back and forth between two positions, and where it will take a considerable amount of frames before this situation is resolved, if ever. This manifests itself as visible flickering of the particles, something that is easily observed in the result videos. A fairly simple solution to the flicker could be to add a post-stabilization step, where flickering of particles is detected and averaged out.

However, the one-frame delay is a known problem, and is mentioned by Bender *et al.* (2015). They call it 'slow propagation of local errors'. A solution is mentioned in Müller (2008), and definitely forms a possible of direction of future work for this project.

A problem inherent to the Jacobi iterations as presented by Macklin *et al.* (2014) is the constraint averaging. While necessary, this averaging can be too harsh and increase the amount of necessary solver iterations themselves. What is worse is that the amount of necessary iterations depends on the amount of constraints (in one constraint group) due to the averaging. The more constraints there are, the more every correction will be averaged out, so more iterations are necessary to reach a solution state. This is illustrated in Figure 5.10. Making things worse, consider

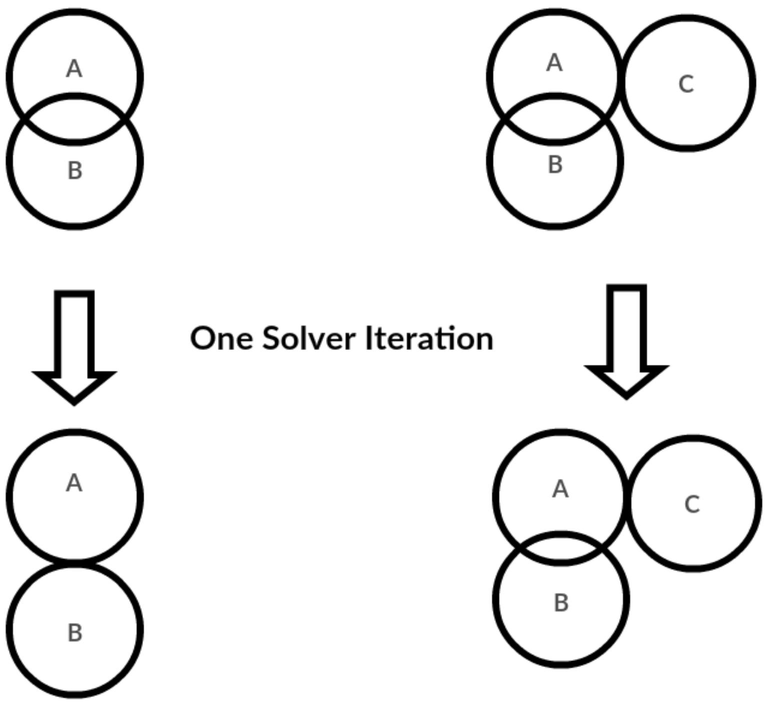


Figure 5.10: *Constraint averaging makes the amount of necessary solver iterations depend on the amount of constraints. In the situation on the left, the collision is resolved in one iteration. On the right however, the third particle has barely any influence (the particles merely touch), but because of its presence there is an extra collision constraint. Consequently, the first constraint is not resolved in one iteration since it is averaged out.*

free-flowing granular materials: the smaller they are, the more collisions will happen in one time-step (for the same magnitude of gravity), and the more iterations are necessary to resolve them all. This ties in the amount of solver iterations with the scale of the simulation which is far from ideal. This inspires one to ask the question if Jacobi iterations are truly necessary. However, as explained by Macklin *et al.* (2014), it is almost impossible to efficiently parallelize the Gauss-Seidel version of PBD.

Library The code library created for this project has been thoroughly covered in the previous chapter. The author believes the code design and implementation is good. However, there are always possible improvements for a software project of this time-frame.

A first one is the improvement of the Fluid simulation. Due to the time constraints of the project, there was not enough time to thoroughly test the fluid simulation, since fluid simulation take a fair amount of time to get completely stable. Furthermore the tensile instability and vorticity confinement from (Macklin and Müller 2013) are still missing. The same goes for the deformable rigid bodies, since not all techniques from Müller *et al.* (2005) were implemented, but just the one for small deformations based on a multiplier. Adding or improving these elements can be done without touching any of the existing code. However, due to the broad range of existing PBD techniques, this will always be the case. This is exactly the reason extensibility was valued so highly. A variety of different elements (constraints, objects, forces, ...). It would for instance be possible to add tearable cloth fairly easily.

The implemented solver was not meant to be touched but could still be tweaked. For instance, a post-stabilization step to decrease the particle flickering would be a nice addition.

Apart from that, the main improvement would be a better application. Usage of the library through the current Basic Application with XML files is quite laborious. It would be nice to build a plug-in for Houdini or Maya that calls PBPBD directly.

Off course, the performance could also still improve. As explained, the profiling happened in an iterative fashion and such iterative processes are never truly finished. It would be nice to experiment with other multi-threading libraries or techniques as well, such as Intel TBB or CUDA.

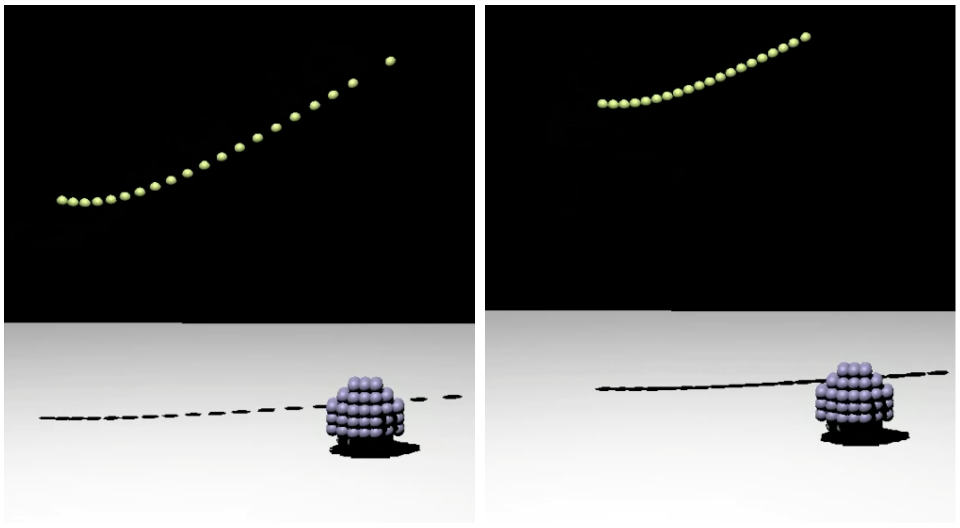


Figure 5.11: *In the figure on the left, a low amount (7) of solver iterations ensures no work is wasted on the rigid body, but the rope is very extensible. On the right we see the opposite situation, a high amount of solver iterations (70) ensures inextensible rope but wasted effort on the RBD.*

5.4 Adaptive Position Based Dynamics

As was explained in the previous section, all current PBD algorithms have a fixed amount of iterations for all constraints. However, not all constraints depend equally on the amount of iterations. In the previous section we compared cloth/rope to rigid bodies, which is a good example:

- The cloth and rope are connected with distance constraints. These are relatively cheap to evaluate but need a lot of iterations to appear stiff.
- Rigid bodies are simulated using shape matching constraints. These dont need a lot of iterations but are expensive to compute due to the matrix operations, as outlined in chapter 3.

This is illustrated in Figure 5.11.

Ideally, one wants to avoid the wasted effort of doing many iterations on the rigid bodies, since they simply dont need it. For this reason, it would be useful to have different solver iterations per constraint-type.

To the best of my knowledge, this has not been done in previous research.

The basic idea goes as follows. Assume two constraints A & B are being processed, and we want constraint A to be processed twice as much as B. We do however want that B is processed in the last iteration, otherwise there might be serious mistakes (e.g. if B is a shape matching constraint, it would be no longer Rigid Body simulation). On the other hand, we want to avoid doing A without incorporating information from B for too many iterations. Consequently, A is solved every iteration and B every other iteration.

This example does not show the full complexity of the process, since in actual simulations there will be many constraints working together, but it shows the idea. Due to the time constraints of this project, only a relatively simple experiment was conducted. The constraint-groups as introduced by Macklin *et al.* (2014) and implemented in this project were an ideal mechanism for testing this variable solver iterations idea.

The constraint solver was made adaptive by first repeating the lower priority constraints. In the current library the priorities range from 0 to 9. In one solver iterations, the constraint group are solved in the order: 0, 0, 1, 0, 1, 2, 0, 1, 2, 3, 0, Hence for for 1 solver iteration, the constraints in group 0 are solved 10 times and those in 9 only once. This gives a simple adaptive constraint solver.

To show-case the adaptive solver, we show two variants of one test-case. This test-case consists of a string of rope and a rigid body. In the first variant, the two are not connected. The regular PBD version of this case was presented in Figure 5.11. As can be seen in Figure 5.12, the adaptive solver does equally well, without wasting solver iterations on the rigid body. This is as expected when the two effects are unconnected.

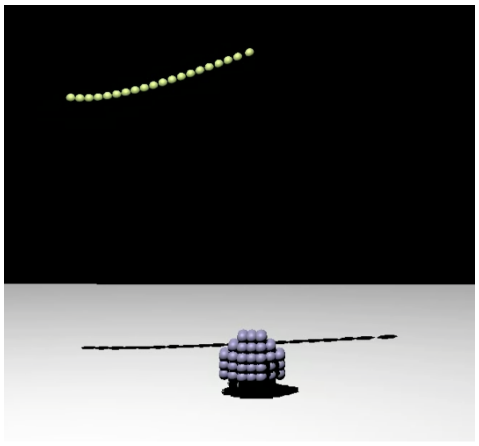


Figure 5.12: *In the adapative version, 70 iterations are used for the rope, and 7 for the RBD, resulting in the same result as in the right of Fig. 5.11, without the wasted effort on the .*

Things become more complex in the second variant. Here the rope and the rigid body are connected with a distance constraint. The stretchiness of the rope becomes worse as there is a considerable mass at the end. The result for normal, non-adaptive PBD can be seen in Figure 5.13. The adaptive solver still behaves as before but due the low iterations on the shape matching constraint, the connecting distance constraint is overstretched. However, even in this case it is still possible to reduce the efforts wasted on the rigid body, as illustrated in Figure 5.14. Hence the potential for saving on solver iterations will depend on the particular simulation, but that goes for the current amount of solver iterations already anyway.

In conclusion, this project was complemented by an experimental adaptive constraint solver, an idea that has not been proposed in current literature and has the potential to improve the performance of existing PBD algorithms. This experimental solver works as expected and shows that this idea has promise. More tests need to be performed. Ideally the amount of solver iterations would not be connected to the constraint priority in the future.

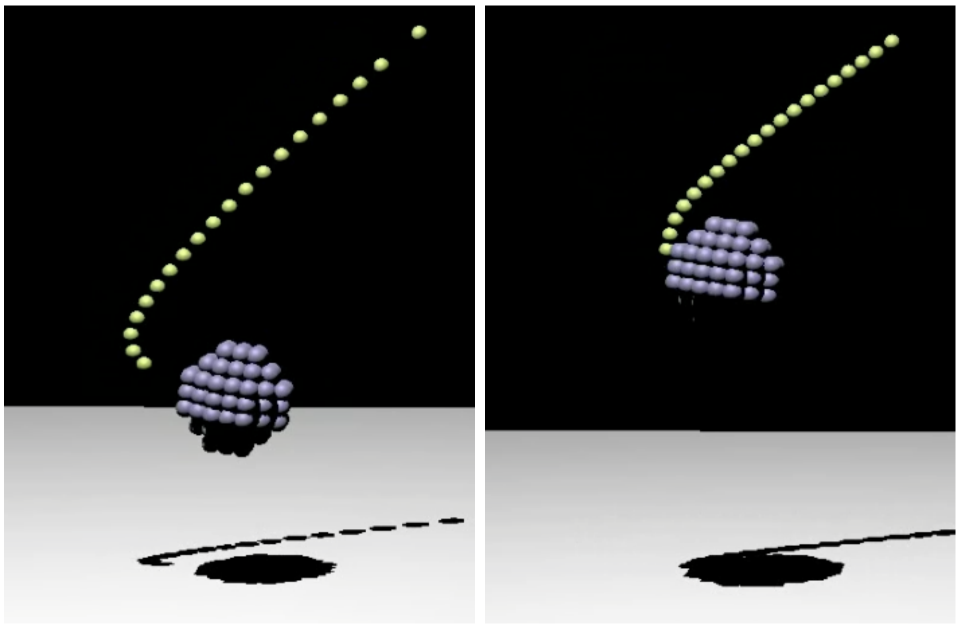


Figure 5.13: *The regular version of PBD in the connected variant with two different amounts of solver iterations, left 7 iterations, right 70.*

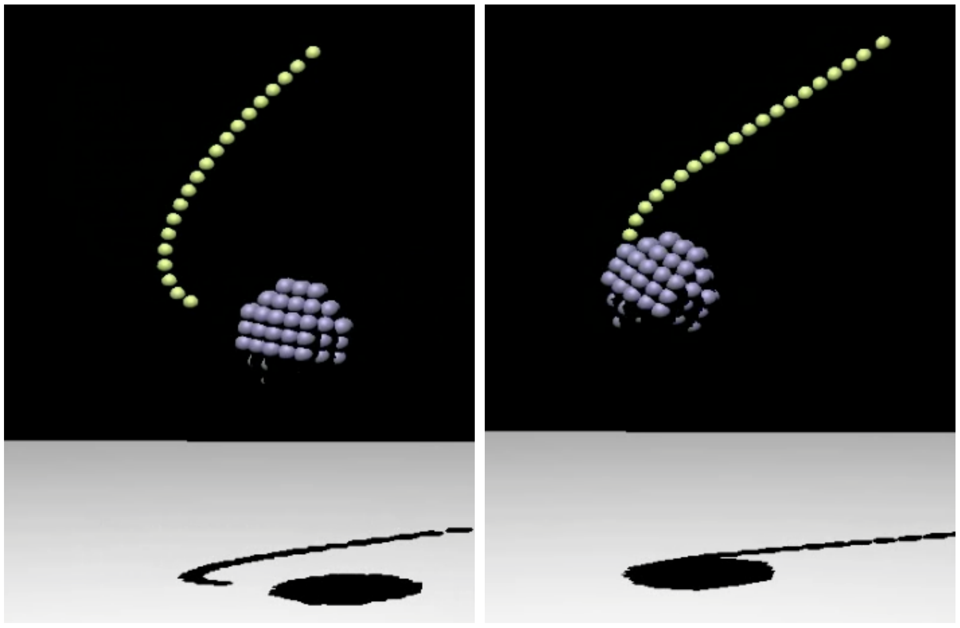


Figure 5.14: *In the figure on the left, a low amount (7) of solver iterations ensures no work is wasted on the rigid body, and the rope is inextensible. However, the distance constraint connecting gets over-stretched. On the right, we have 70 iterations for the rope and 35 for the rigid body. This gives a good result and halves the work necessary for the Rigid Body.*

The results as presented in Figures 5.11 to 5.14 can be found in the folder 'adaptiveness' in the hand-in. The videos marked with fixed are the regular, non-adaptive PBD. The ones marked with adaptive are the adaptive PBD ones.

Chapter 6

Conclusion

In this project Position Based Dynamic algorithms were researched, and a particle-based unified physics library was developed based on these techniques. Looking back at the goals as set in the introduction, this document, and the additional material in the hand-in, shows that:

- Position based dynamics was successfully researched and implemented. Implementing and analysing the results even lead to proposing an original improvement, i.e. adaptive constraint solver iterations, as explained in section 5.4.
- The resulting library was multi-threaded by using the OpenMP threading library.
- The library contains cloth, hair, granular Material,rigid and deformable bodies and fluids, although the last one needs to be improved. All of them interact with each other.
- The author believes the library to have a good design. The extensibility was singled out in the introduction, and was thoroughly discussed in section 4.

I think I can say all the goals set at the start of the project were met. It would have been nice to multi-thread it in Intel TBB as well as in OpenMP, but unfortunately there was not enough time to do this.

After implementing the techniques, they were tested to see what problems were still present in the techniques. This discussion lead to several ideas for future work. The most promising one was an adaptive constraint solver, meaning that different constraints are solved with different amount of solver iterations. This idea was actually implemented in the project, and the results look promising, since they can increase performance. This adaptive solver can be seen as an original contribution, as no existing research makes mention of such an idea, to the best of my knowledge.

Bibliography

Bell N., Yu Y. and Mucha P. J., 2005. Particle-based simulation of granular materials. In *Proceedings of the 2005 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, SCA '05, New York, NY, USA. ACM, 77–86.

Bender J., Müller M. and Macklin M., 2015. Position-based simulation methods in computer graphics. In *EUROGRAPHICS 2015 Tutorials*. Eurographics Association.

Bender J., Müller M., Otaduy M. A. and Teschner M., 2013. Position-based methods for the simulation of solid objects in computer graphics. In *EUROGRAPHICS 2013 State of the Art Reports*. Eurographics Association.

Bender J., Müller M., Otaduy M. A., Teschner M. and Macklin M., 2014. A survey on position-based simulation methods in computer graphics. *Computer Graphics Forum*, **33**(6), 228–251.

Deul C., Charrier P. and Bender J., 2014. Position-based rigid body dynamics. *Computer Animation and Virtual Worlds*.

Jaeger H. M., Nagel S. R. and Behringer R. P., Oct 1996. Granular solids, liquids, and gases. *Rev. Mod. Phys.*, **68**, 1259–1273.

Jakobsen T., 2001. Advanced Character Physics. In *Game Developers Converence Proceedings*. CMP Media, Inc., 383–401.

Macklin M. and Müller M., 2013. Position based fluids. *ACM Transactions on Graphics (TOG)*, **32**(4), 104.

Macklin M., Müller M., Chentanez N. and Kim T.-Y., 2014. Unified particle physics for real-time applications. *ACM Transactions on Graphics (TOG)*, **33**(4), 104.

Müller M., 2008. Hierarchical position based dynamics. In Faure F. and Teschner M., editors, *VRIPHYS*. Eurographics Association, 1–10.

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

Müller M., Heidelberger B., Teschner M. and Gross M., 2005. Meshless deformations based on shape matching. In *ACM SIGGRAPH 2005 Papers*, SIGGRAPH '05, New York, NY, USA. ACM, 471–478.

Pacheco P., 2011. *An Introduction to Parallel Programming.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition.

Parent R., 2012. *Computer Animation: Algorithms and Techniques.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 3 edition.

Witkin A., Gleicher M. and Welch W., February 1990. Interactive dynamics. *SIGGRAPH Comput. Graph.*, **24**(2), 11–21.

Appendix A

Glossary

This appendix contains a glossary of all classes and type definitions in the software project. The definitions are focused on the responsibility the classes have in the system.

- **AnimatedParticleBuilder** : A simulationElementBuilder that adds an AnimatedPositionConstraint to the simulation.
- **AnimatedPositionConstraint** : A constraint fixing a particular particle to a position that changes every frame, thereby animating the particle instead of simulating it. Child of FixedPositionConstraint.
- **BasicApplication** : The class containing the logic for the basic application that works with XML input and writes out the position per frame.
- **BuilderMap** : Typedef: std::map<std::string, BuilderPtr> (Used in the SimulationElementFactory, to register SimulationElementBuilders with a name)
- **BuilderPtr** : Typedef: boost::shared_ptr<SimulationElementBuilder>
- **ClothBuilder** : A SimulationElementBuilder that adds a particle object that will behave like cloth to the simulation.
- **CollisionEnvironment** : An Environment creating collision and friction constraints for rigid contacts, both inter-particle collisions

and collsions with CollisionObjects.

- **CollisionObject** : Abstract class defining objects that make up the physical environment with which the particles interact.
- **CollisionObjectPtr** : Typedef: boost::shared_ptr<CollisionObject>
- **CollisionTrianglesBuilder** : A simulationElementBuilder adding TriangleObjects to the simulation, based on a given OBJ file.
- **Constraint** : Abstract definition of a constraint.
- **ConstraintGroups** : Typedef: std::map<int, ConstraintPtrVec> (Every constraint group is a ConstraintPtrVec with the key the priority of the group)
- **ConstraintPtr** : Typedef: boost::shared_ptr<Constraint>
- **ConstraintPtrVec** : Typedef: std::vector<ConstraintPtr>
- **ConstraintSolver** : Class keeping track of the constraints in the system and solving them for the SimulationController.
- **DistanceConstraint** : A constraint keeping two particles at a specified distance.
- **DistanceConstraintBuilder** : A SimulationElementBuilder adding a DistanceConstraint between two particles to the simulation.
- **DragBuilder** : A SimulationElementBuilder adding a dragforce to the simulation.
- **DragForce** : An Externalforce representing drag (i.e. a force opposing the current velocity). Can work on all objects or one specified object.
- **Environment** : Abstract definition of an Environment that makes up temporary constraints per timestep.
- **EnvironmentPtr** : Typedef: boost::shared_ptr<Environment>
- **ExplicitEulerIntegrator** : Class containing functions to update a position and velocity given a certain net force.

- **ExternalForce** : An abstract class representing external forces such as gravity, drag, ...
- **ExternalForcePtr** : Typedef: boost::shared_ptr<ExternalForce>
- **ExternalForcePtrVec** : Typedef: std::vector<ExternalForcePtr>
- **ExternalForceSolver** : Class keeping track of the external forces in the system and solving them for the SimulationController.
- **FixedPositionConstraint** : Constraint fixing a particle at one, fixed, location. Superclass of the AnimatedPositionConstraint.
- **FluidBuilder** : A SimulationElementBuilder adding a fluid object to the simulation.
- **FluidConstraint** : A constraint meant to compute the movement due to the pressure at one particular fluid particle.
- **FluidEnvironment** : An environment creating FluidConstraints per timestep.
- **FrictionConstraint** : Constraint applying friction between two particles.
- **GravityBuilder** : A SimulationElementBuilder that adds a GravityForce to the simulation.
- **GravityForce** : An externalforce representing a force with a constant acceleration.
- **InfinitePlaneConstraint** : A constraint making particles stay on one particular side of an infinite plane.
- **MutexType** : A Mutex for ensuring that code is only executed by one thread at a time.
- **ObjectFrictionConstraint** : A constraint providing friction between the particle and a CollisionObject
- **ObjectParameter** : A struct to easily pass parameters to SimulationElementBuilders. Combination of parametername (std::string)

and a boost::Variant. The boost variant can contain an integer, float, string, boolean or a vector.

- **OneObjectConstraint** : Abstract class representing a constraint working on all particles of a particular ParticleObject.
- **OneParticleConstraint** : Abstract class representing a constraint working on one particular Particle.
- **ParameterTransformation** : A class representing a transformation built from ObjectParameters. Used in a lot of SimulationElementBuilders.
- **ParameterValue** : Typedef: boost::variant<int, float, bool, std::string, ngl::Vec3> (Used to have some flexibility in storing parameters in ObjectParameters, which are passed to SimulationElementBuilders)
- **Particle** : Class representing a Particle.
- **ParticleCollisionConstraint** : Constraint resolving a collision between two particles.
- **ParticleCollisionData** : Data representing how a collision with a particle should be resolved. In some cases it is prefferable to use another direction than the contact normal.
- **ParticleDraft** : A struct representing a particle to be added to PBPBD by the BasicApplication later.
- **ParticleIdentifier** : Typedef: std::pair<int, int> (used to represent a particle, where first is the ParticleObject ID, and second is the Particle ID.)
- **ParticleInfo** : Blueprint for a Particle. Used in ParticleObject to add Particles later on.
- **ParticleIntegrator** : Class updating Particles by applying their net force.
- **ParticleObject** : Class representing one object consisting of Particles.

- **ParticleObjectPtr** : Typedef: boost::shared_ptr<ParticleObject>
- **ParticleScene** : Class representing all ParticleObjects in the simulation.
- **PBPBD** : The main interface for the library.
- **PBPBDFileParser** : Class parsing XML files for the basic application.
- **RigidBodyBuilder** : A SimulationElementBuilder that adds a RigidBodyConstraint to the simulation.
- **RigidBodyConstraint** : A constraint keeping a ParticleObject in a pre-determined shape. The constraint might allow deformations based on its stiffness setting.
- **RopeBuilder** : A SimulationElementBuilder that adds a ParticleObject that will behave as a rope (a chain of connected particles)
- **SandBuilder** : A SimulationElementBuilder that adds a ParticleObject that will behave as sand (unconnected particles).
- **ScopedLock** : A lock in MutexType.
- **SimulationController** : The interface/facade for controlling the simulation.
- **SimulationElementBuilder** : Abstract class used for setting up different kinds of elements of a simulation.
- **SimulationElementFactory** : Factory class keeping track of the existing SimulationElementBuilders and calling them when necessary.
- **TriangleObject** : CollisionObject representing one triangle.
- **TwoParticleConstraint** : Abstract class representing a constraint working on two particular Particles.
- **WallBuilder** : A SimulationElementBuilder adding a WallObject to the simulation.

- **WallCollisionConstraint** : A constraint resolving a collision between a particle and a WallObject.
- **WallObject** : A CollisionObject representing an infinite plane.

Appendix B

Default SimulationElementBuilders

This Appendix shows the SimulationElementBuilders that are registered with PBPBD by default. This includes their type-name used to call them through PBPBD, and all of their parameters. Every parameter also has a name, and a value type. Every parameter is explained and its default is given, unless it is a mandatory parameter. A mandatory parameter is signified by **M** in the default column.

Some SimulationElementBuilders use a ParameterTransformation object to register a transformation. These objects take the following parameters and defaults:

Parameter	**Type**	**Default**
scale	float	1.0f
rotateX	float	0.0f
rotateY	float	0.0f
rotateZ	float	0.0f
translate	vector	(0, 0, 0)

Depending on the particular SimulationElementBuilder, this transformation is used in a different way. However, for all of them, the transformation has the above parameters and defaults. To keep the below tables shorter, these transformation parameters are replaced by one parameter called 'transform'. However, none of the default SimulationElement-Builders have an actual parameter called 'transform'. The presence of this parameter in their table simply means the particular builder takes all of the parameters in the above table as well. The transformation order is: uniform scaling - rotation (around X,Y and Z, in that order) - translation, as this allows scaling and rotating in place before translating.

The default SimulationElementBuilders are (**M** is mandatory, transform refers to the table above):

AnimatedParticleBuilder (type-name: 'animatedParticle')			
Parameter	**Type**	**Default**	**Remarks**
transform	T	see above	Transform the animation.
file	string	**M**	Sequence of positions (txt or OBJ).
objectID	int	0	The objectID of the animated particle
particleID	int	0	The particleID of the animated particle

ClothBuilder (type-name: 'cloth')			
Parameter	**Type**	**Default**	**Remarks**
transform	T	see above	Transform the cloth.
radius	float	1.0f	particle radius
mass	float	0.0f	particle mass
staticFriction	float	0.0f	friction coefficient
kineticFriction	float	0.0f	friction coefficient
particleSleeping	float	(0, 0, 0)	amount of particle sleeping
stiffness	float	**M**	Constraint stiffness. Between 0 and 1.
file	string	0	Mesh for cloth (OBJ)
fixed	int	none	Fix one particle (by ID). Can be specified multiple times

CollisionTrianglesBuilder (type-name: ’triangles’)			
Parameter	**Type**	**Default**	**Remarks**
transform	T	see above	Transform the mesh.
file	string	**M**	Mesh for collision object (OBJ).

DistanceConstraintBuilder (type-name: ’distanceLock’)			
Parameter	**Type**	**Default**	**Remarks**
distance	float	1	Needed distance between the particles.
objectID	int	0/0	ObjectID of locked particle (need to give 2!)
particleID	int	0/1	ParticleID of locked particle (need to give 2!)

DragBuilder (type-name: ’drag’)			
Parameter	**Type**	**Default**	**Remarks**
coefficient	float	0.1f	Drag Coefficient.
objectID	int	none	ObjectIDs of objects to apply drag to. Can be specified multiple times. If none given, applied to all.

FluidBuilder (type-name: ’fluid’)			
Parameter	**Type**	**Default**	**Remarks**
radius	float	1.0f	particle radius
mass	float	1.0f	particle mass
staticFriction	float	0.0f	friction coefficient
kineticFriction	float	0.0f	friction coefficient
particleSleeping	float	0.0f	amount of particle sleeping
density	float	0.2f	Fluid rest density
smoothing	float	4.0f	Fluid smoothing length

GravityBuilder (type-name: ’gravity’)			
Parameter	**Type**	**Default**	**Remarks**
direction	vector	(0, -1, 0)	Gravity direction
magnitude	float	10.0f	Gravity magnitude

RigidBodyBuilder (type-name: 'rigidBody')			
Parameter	**Type**	**Default**	**Remarks**
transform	T	see above	Transform the animation.
radius	float	1.0f	particle radius
mass	float	1.0f	particle mass
staticFriction	float	1.0f	friction coefficient
kineticFriction	float	1.0f	friction coefficient
particleSleeping	float	0.00001f	amount of particle sleeping
file	string	**M**	Points making up the rigid body (txt or OBJ)
velocity	vector	(0, 0, 0)	Initial rigid body velocity.
repeater	string	none	Optional list of translations. One Rigid Body per translation. (txt or OBJ)
fixed	int	none	Fix one particle (by ID). Can be specified multiple times
stiffness	float	1.0f	Controls amount of deformation allowed. Has to be between 0 and 1. 1 is fully rigid.

RopeBuilder (type-name: 'rope')			
Parameter	**Type**	**Default**	**Remarks**
transform	T	see above	Transform the animation.
radius	float	1.0f	particle radius
mass	float	1.0f	particle mass
staticFriction	float	0.0f	friction coefficient
kineticFriction	float	0.0f	friction coefficient
particleSleeping	float	0.0f	amount of particle sleeping
bending	bool	FALSE	Whether to attach bending constraints.
stiffness	float	1.0f	Constraint stiffness. Between 0 and 1.
bendingStiffness	float	0.5f	Bending Constraint Stiffness. Between 0 and 1.
file	string	**M**	List of positions (and optionally velocities) making up the rope. (txt or OBJ)
fixed	int	none	Fix one particle (by ID). Can be specified multiple times

SandBuilder (type-name: 'sand')			
Parameter	**Type**	**Default**	**Remarks**
radius	float	1.0f	particle radius
mass	float	1.0f	particle mass
staticFriction	float	0.15f	friction coefficient
kineticFriction	float	0.15f	friction coefficient
particleSleeping	float	0.0001f	amount of particle sleeping

WallBuilder (type-name: 'wall')			
Parameter	**Type**	**Default**	**Remarks**
normal	vector	(0, 1, 0)	Normal on the infinite plane.
distance	float	0.0f	Distance between origin and plane.

Appendix C

Basic Application Manual

C.1 XML skeleton

As was mentioned in the thesis, the library was complemented with a basic application to show-case its capabilities. This basic application is fairly simple: it reads in an XML file describing the simulation, and writes out one CSV file containing the simulation results per frame number. To run the application, one needs to go into the folder where it was compiled and run it with one or two parameters. The first parameter is the XML simulation description, the second, optional, parameter the folder to store the resulting files in. Hence running the basic application will simply require the following command in the terminal (provided the terminal is in the right location):
./PBPBD scenefile.xml results/
If the latter argument is not specified the application will automatically try to write to the 'results' folder.

The key here is, obviously, how to specify said XML files. This appendix will show the right XML syntax. The basic application will only work with XML files adhering to the following pattern:

```
<simulation>
    <simulationParameter>value</simulationParameter>
    ...

    <elements>
        <element>
            <type>elementType</type>
            <parameterName>value</parameterName>
            ...
        </element>
    </elements>

    <particles>
        <particle>
            <parameterName>value</parameterName>
        </particle>
        <list>
            <parameterName>value</parameterName>
        </list>
        <listSequence>
            <parameterName>value</parameterName>
        </listSequence>
    </particles>
</simulation>
```

We explain this skeleton from the first line down (Tag names are in italics):

- The whole simulation is enclosed in the *simulation* tags. These are mandatory.
- Some settings (e.g. time-step, frame count) that are global to the simulation can be set within the *simulation* tags immediately. These are the *simulationParameter* tags. What tag names to use exactly, and what values these can take, are specified in section C.2. The dots on the next line specify that multiple of these lines

can follow.

- The *elements* tag denotes the start of the elements specification. These tags are mandatory, since a simulation needs at least one element.
- The *element* tag denotes the start of the specification of one element. At least one element needs to be specified but there is no upper limit. Every element needs *type* tags, saying what kind of element it is. This type completely determines what kind of parameters can be given. The type-names that are accepted as element-Type are the type-names of the default SimulationElementBuilders from appendix B. Consequently, the parameter names and their possible values for elements can also be found in said appendix.
- The *particles* tags are not mandatory. They denote the start of the particle specification, where particles can be added to objects at particular frames. There are three different ways of doing this, as can be seen in the XML skeleton. These three different ways and their parameters are explained in section C.3.

C.2 Simulation Parameters

The global simulation parameters that are settable within the *simulation* tags directly are as follows. **M** denotes mandatory parameters:

Parameter	**Type**	**Default**	**Remarks**
iterations	int	**M**	Frame count for the simulation
solverIterations	int	5	Iterations for the constraint solver
timestep	float	0.1	The time-step between two frames
overRelaxation	float	1	The amount of over-relaxation in the system
stabilize	int	0	The amount of pre-stabilization iterations
adaptive	bool	FALSE	Whether or not to use adaptive solver iterations

C.3 Particles Parameters

There are three different ways of specifying particles within the *particles* tags. The simplest is adding one particle through the *particle* tags, with the following parameters. **M** denotes mandatory parameters:

Parameter	**Type**	**Default**	**Remarks**
time	int	**M**	The frame number the particle should be introduced
object	int	**M**	The ID of the object to add the particle to
position	vector	**M**	The start position of the particle
velocity	vector	(0, 0, 0)	The start velocity of the particle

More involved, but also more efficient, is adding a complete list of particles at once, through the *list* tags, with parameters:

Parameter	**Type**	**Default**	**Remarks**
time	int	**M**	The frame number the particle should be introduced
object	int	**M**	The ID of the object to add the particle to
file	string	**M**	List of start positions (and optionally velocities) to create particles with (txt or OBJ)

Again more efficient is adding a list of particles every frame for a certain range of frames:

Parameter	**Type**	**Default**	**Remarks**
time	int	**M**	The frame number to start adding the particles
endTime	int	**M**	The frame number to stop adding the particles
object	int	**M**	The ID of the object to add the particle to
file	string	**M**	List of start positions (and optionally velocities) to create particles with (txt or OBJ). This is a template, and exactly one '*' character is excepted. Every frame in the specified range, a file will be opened with that character replaced by the frame-number, no padding.

C.4 Example

To conclude this section, we give a simple example of an XML file. In this example a simulation is supposed to run for 50 frames with 7 solver iterations. It contains two sand particles, added at different times. More complex examples of XML-files will be included in the hand-in.

The example:

```
<simulation>
    <iterations>50</iterations>
    <solverIterations>7</solverIterations>

    <elements>
        <element>
            <type>sand</type>
            <radius>0.8</radius>
            <mass>0.05</mass>
        </element>
    </elements>

    <particles>
        <particle>
            <time>0</time>
            <object>0</object>
            <position>1 1.2 0</position>
        </particle>
        <particle>
            <time>13</time>
            <object>0</object>
            <position>5 8 3.2</position>
            <velocity>0 -1 1</velocity>
        </particle>
    </particles>
</simulation>
```