Position-based Complementary Dynamics: Unified Pipeline for Physically Plausible Character Animations

by

Andrew Mahisa Halim

Student Number: 1242547

Supervisor: A/Prof. Jorge Goncalves & Dr Hailong Guo

A thesis submitted in partial fulfilment for the
degree of Master of Computer Science (MC-CS)

in the
Department of Computing and Information Systems
Melbourne School of Engineering and IT
THE UNIVERSITY OF MELBOURNE

March 2023

THE UNIVERSITY OF MELBOURNE

Abstract

Department of Computing and Information Systems
Melbourne School of Engineering and IT

Master of Computer Science

by Andrew Mahisa Halim
Student Number: 1242547
Supervisor: A/Prof. Jorge Goncalves & Dr Hailong Guo

To accommodate the ever-increasing demand for realism, contemporary character animations are not only required to portray the character's main course of action, but also the subtle complementary movements that accompany it. While the character's main pose tends to be quite simple to animate, complementary movements such as cloth swaying and body jiggles are painstakingly tedious to animate by hand. For this reason, numerous works have tried to build a pipeline that integrates traditional animation with physics-based animation, allowing these complementary effects to be captured via simulation. However, because physics-based animation deals with physical quantities (e.g. forces, torques) that require translational mechanisms to work together with handcrafted animation, these pipelines sacrifice artistic control and performance in the process.

In this thesis, we propose a novel animation pipeline that is "position-only". By only working with positions, the most intuitive quantity, our pipeline produces physically plausible character motions without foregoing intuitiveness and artistic control. To this end, we develop a physics-based animation algorithm capable of producing the aforementioned complementary effects without requiring forces or other physical quantities as input. In addition, the performance overhead stemming from having to transform our input into physical quantities are circumvented completely. To put our claims to the test, we run several experiments and a user study utilising characters of varying complexity. We demonstrate that, apart from a few visual artefacts, our method produces comparable visuals to other pipelines all while being much faster and more intuitive to control. To conclude our work, we also provide avenues for future research based on insights gathered from our experiments.

Keywords: character animation, physics simulation, animation control

Declaration of Authorship

I, Andrew Mahisa Halim, declare that this thesis titled, "Position-based Complementary Dynamics: Unified Pipeline for Physically Plausible Character Animations" and the work presented in it are my own. I confirm that:

- This thesis does not incorporate without acknowledgement any material previously submitted for a degree or diploma in any university; and that to the best of my knowledge and belief it does not contain any material previously published or written by another person where due reference is not made in the text.
- The thesis is between 25.000-30.000 words in length (excluding text in figures, tables, code listings, bibliographies, and appendices).

Signed: 

Date: March 2023

Acknowledgements

Words cannot express my gratitude to both my supervisors, A/Prof. Jorge Goncalves and Dr. Hailong Guo for their endless patience and feedback throughout my thesis journey. The outcome of this thesis owes much to their compassionate support and continuous guidance. Furthermore, I am especially grateful for my course coordinators, A/Prof. Nic Geard and Dr. Jey Han Lau, for their invaluable advice throughout the entirety of my degree. I would also love to thank the Faculty Members and University of Melbourne Special Consideration Team that has given me the necessary extension to complete my thesis despite my medical condition.

This past year has been filled with mental struggles, and without the help of my Clinical Pyschologist Ms. Hayley Zarb, i would not have made it this far; I am highly indebted to her for her limitless compassion and mental support. The same goes to my three closest friends: Albert Darmawan, James Barnes and Wildan Anugrah who continously checked to make sure i was sane and well.

Lastly, I would be remiss in not mentioning my family, especially my parents and my sister. Their belief in me has been my biggest motivation and their financial support allows me to keep going when part-time work is not possible. I am extremely fortunate to have them as family.

Contents

List of Figures

List of Tables

CHAPTER 1

Introduction

CREATING believable and compelling movement of characters is one of the main challenges in computer animation (McLaughlin et al., 2011). One critical aspect of creating such movement is the presence of both *primary* and *secondary* motion (Thomas et al., 1995). Whereas primary motion delivers the main action of a character within a scene, secondary motion complements it with a subtle movement that accentuates the primary motion. Take a look at the character animation shown in Figure 1.1 for example. The impact of the stomping motion (primary) is made more prominent by the presence of jiggles in the character's belly and cheeks (secondary). These additional effects add nuance and authenticity to an otherwise dull and lifeless character (Whitaker and Halas, 2013).

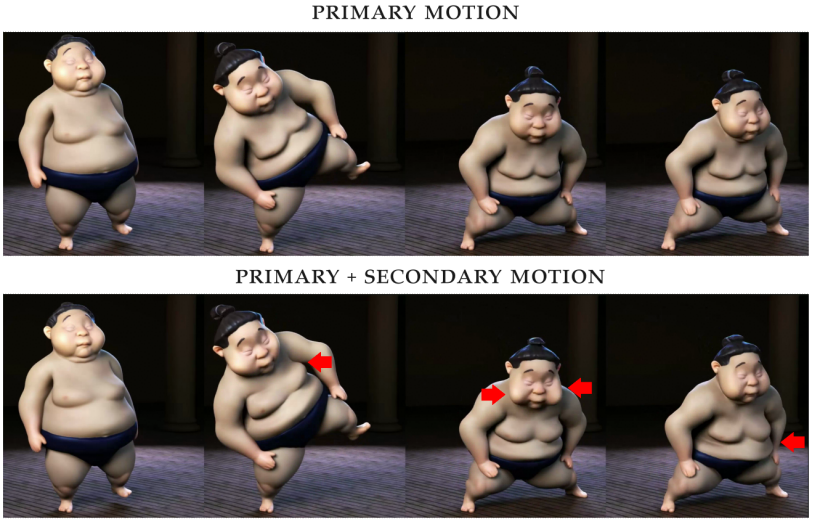


FIGURE 1.1: A sequence of animated characters without (top) and with (bottom) secondary motion. Notice the additional movements in the character's belly and cheeks pointed by the red arrow. Adapted from Hahn et al. (2012).

Despite both being vital to the visual quality of an animation, the means of generating primary and secondary motion are significantly different. Generating primary motion such as running, walking and punching are often seen as an artistic endeavour, supported by various artistic tools such as skeletal deformation (Kalra et al., 1998, Lewis et al., 2000, Jacobson et al., 2014) and motion capture (Jauregui, 2011). In the contrary, secondary motions such as hair swaying, body jiggling, and facial deformation come as a consequence of physical laws in the real world, making them arduous to capture through manual artistic work (Bargteil and Shinar, 2018). Instead, animations that involve such phenomena are more appropriately generated through *physics-based animation* (Figure 1.2). Physics-based animation encodes various laws of physics inside a computer simulation, making it better at capturing the behaviour of physical phenomena compared to its artisanal counterparts. In a vacuum, both artistic and physics-based animation handle their speciality task extremely well, however, combining them to be in a single pipeline has proved to be a massive undertaking.

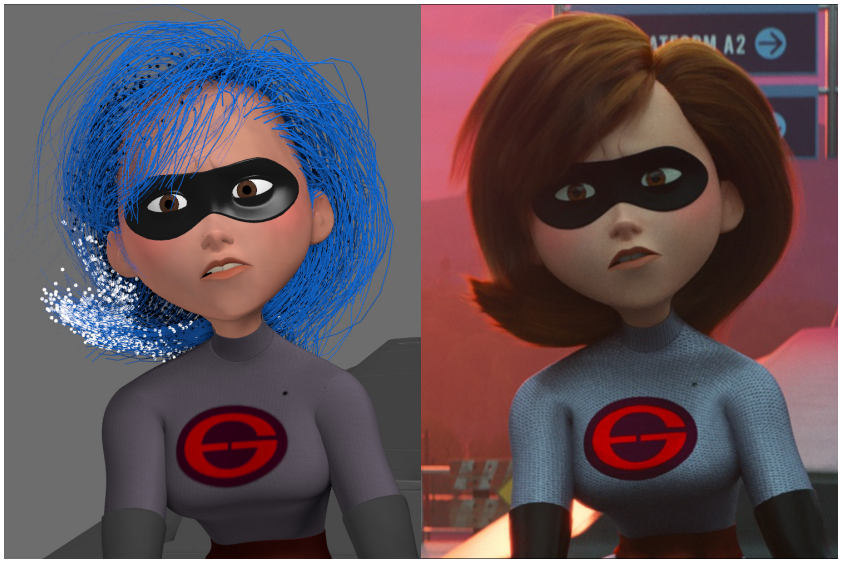


FIGURE 1.2: Physics-based hair simulation (left) and the generated image (right) in the movie *The Incredibles*. Taken from Iben et al. (2013) without permission.

In the early days of character animation, artistic and physics-based animation are joined simply by running them one after another in a layered structure, without any form of information exchanged between them (Daldegan et al., 1993, Capell et al., 2002). It is trivial to see why this will not work; in the real world, secondary motion is the underlying material's natural reaction to primary motion (Willett et al., 2017). Ignoring this cause–effect relationship would lead to crude and *incoherent* secondary motion — as if this complementary motion does not originate from the primary motion.

Fast-forward to more than 20 years later, numerous works have attempted to remedy this situation by developing some form of translational mechanisms to allow the two layers to "talk" to each other (Kim and Pollard, 2011, Li et al., 2019). By propagating some form of input

to the physics layer, the appropriate secondary motion can be generated, achieving physical plausibility. While this brought tremendous improvement to the quality of the animation, it also proved to be a major step back in terms of performance. Moreover, these translational mechanisms rely on complex mathematical operations, which strip away the intuitiveness of the system (Hahn et al., 2012). Given that performance and artistic control are an essential trait for widespread adoption in the industry (McLaughlin et al., 2011), there is a clear need for a method that can do the job without sacrificing performance and controllability.

The long-standing history of this problem immediately raises a question: what makes information exchange between these two animation paradigms costly and difficult? Perhaps unsurprisingly, the answer is difference in *representation* (House and Keyser, 2016). Whereas artist-oriented tools define animation by manipulating the *position* of a vertex inside a 3D object, physics-based animation works by enumerating and applying *forces*. Seeing numerous attempts failing to truly establish proper communication between the two begs another question: could it be that instead of trying to bridge a communication between the two paradigms, we should be *unifying* their representation instead?

A new perspective in physics simulation

Physics-based animation stems from the field of scientific computing and engineering, where physical accuracy is of paramount importance (Erleben et al., 2005). Unfortunately, this origin is apparent in the way most works insist on keeping the force-based representation to stay true to theoretical physics, taking precedence over satisfying the practical requirements of directability and speed (De Goes, 2017). That being said, some works did come as an answer to the distinct requirements of the animation industry, with the most notable being *position-based dynamics* (PBD).

Position-based dynamics (PBD) (Müller et al., 2008, 2020, Macklin et al., 2016) offers a new perspective in physics simulation, deviating from the norm of conventional force-based simulation. PBD works by strictly manipulating positions. Physical laws that are conventionally encoded as forces are being encoded as geometrical *constraints*, which are used to drive vertex to the physically correct position. With the introduction of position-based dynamics, the notion of a unified representation between artistic and physics-based animation is no longer a mere concept. Through this thesis, we present the concrete realisation of this idea — a pipeline that combines artistic animation with position-based dynamics.

As a first step of making this happen, a change needs to be introduced to the physics simulation algorithm itself, position-based dynamics. Secondary motion is composed of predominantly *elastic* motion (e.g. jiggling and swaying), but position-based dynamics cannot handle this type

of motion naturally. As one can see in Figure 1.3, enforcing constraints on positions leads to a stiff (rigid) motion. Before integrating it with artistic animation, we propose an extension to position-based dynamics by introducing *soft* constraints — a new construction that works by guiding the position of a vertex following a physically correct trajectory, instead of immediately moving it to the optimal position.

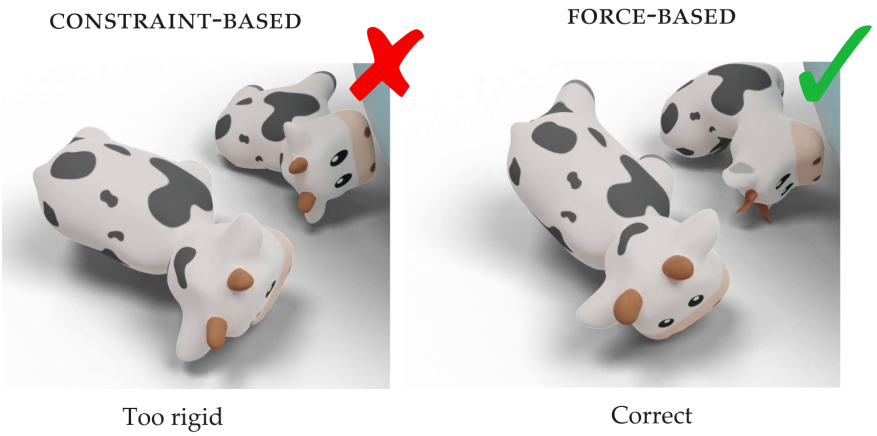


FIGURE 1.3: Simulating rubber-like material with constraint-based (left) and force-based simulation (right). We can see that using constraint to simulate elastic materials tends to result in the resulting motion being too stiff. Adapted from Zhang et al. (2020).

Finally, we plug in the extended position-based dynamics to an animation pipeline alongside artistic animation in a simple layered architecture. Because both sides now work with positions, an exchange of information (and therefore coherent secondary motion) can be achieved without computationally expensive augmentation or communication strategies. Additionally, we expect that the unified representation should also make the fine-tuning process trivial.

1.1 Research Objectives

In this thesis, we present an animation framework capable of producing primary motion along with coherent secondary motion in real-time, by leveraging the power of position-based dynamics. Within this setting, our research objectives are as follows.

Extending Position-based Dynamics

Research Question 1 (RQ1): How can we extend position-based dynamics to be able to accurately simulate elastic materials commonly occurring in secondary motion?

The current iteration of position-based dynamics handles elastic motion in a physically incorrect way, making it unfeasible for elastic-dominant use cases such as secondary motion. In this work,

we proposed position-based dynamics with soft constraints. Soft constraints work by "guiding" the motion instead of enforcing it. With this, elastic and soft materials can be represented naturally without the need of a physically incorrect workaround.

Combining Artistic Animation with Position-based Dynamics

Research Question 2 (RQ2): How can we combine our newly extended position-based dynamics with traditional artistic animation to generate compelling character animation in real-time?

To generate compelling character animation, both primary and secondary motion must be present. However, the limitations of the current work forced us to choose between an animation with incoherent secondary motion or one that is extremely costly to compute. We aim to change this situation by combining our newly extended position-based dynamics with an artistic animation technique called linear blend skinning. We hypothesise that the position-only nature of PBD allows it to work coherently with linear blend skinning without additional performance overhead, filling the needed gap in the literature.

Human-based Evaluation of Character Animations

Research Question 3 (RQ3): What are the benefits and drawbacks of our proposed animation framework compared to previous approaches?

Currently, the majority of research in character animation does not include any form of visual evaluation. The evaluation section is often limited to runtime performance metrics and an animation showcase. We deemed this inadequate, especially when the improvements are not abundantly obvious visually. The lack of visual evaluation in this context is caused by the absence of a metric that can quantify the quality of animations that involve artistic work. To solve this issue, we conduct a user study to perform comparative evaluation between our method and previous approaches.

1.2 Thesis Organisation

The remainder of this thesis is organised as follows:

Chapter 2, Literature Review and Theoretical Background. This chapter provides comprehensive background review into the computational framework of artistic and physics-based

animation. In addition, we discuss various attempts at combining the two paradigms to produce lifelike character animations. We also highlight strengths and weaknesses of each method and point out the research gap left by the current state-of-the-art.

Chapter 3, Extending Position-based Dynamics. This chapter presents our extension to position-based dynamics. We show the full derivation of our algorithm from the well-known Newton's second law. Furthermore, we demonstrate how the original PBD can be thought of as a special case of our extended PBD. Finally, we carry out several evaluations to verify our claims both from the correctness and performance standpoint.

Chapter 4, Position-based Complementary Dynamics. In Chapter 4, we introduce Position-based Complementary Dynamics (PBCD), a sequential animation pipeline that utilises our extended PBD alongside artistic primary motion. We then present performance and visual quality evaluation through a user study to determine how it compares to various state-of-the-art animation pipelines. To complete our research, we dive deeper into the result of our visual evaluation and investigate the theoretical reasoning behind our framework's strengths and pitfalls.

Chapter 5, Conclusions and Future Directions. In this chapter, we summarise the main contribution of this thesis and suggest possible directions for future work.

CHAPTER 2

Literature Review and Theoretical Background

CREATING a believable and compelling character animations is a multidisciplinary problem that can largely be divided into three different tasks. In this chapter, we review the wealth of literature that surround each task, as well as establishing fundamental knowledge necessary to recognise the limitations that exist in the current landscape.

1. In Section 2.1, we tackle the task of **generating high-quality primary motion via artistic animation**. We review concepts, tools and previous works that allow artists to define such motions in a digital medium.

2. In Section 2.2, we look into the task of **producing realistic secondary effects through physics-based animation**. We formally introduce the mathematical frameworks of physics-based simulation alongside numerous works that brought it to the field of computer animation.

3. In Section 2.3, we outline the latest attempts of **building an animation pipeline (*deformers*) containing both the aforementioned motions**. We summarise the merits and weaknesses of each pipeline and set up the scene for our method.

There has been a considerable amount of work on each of these subjects; therefore, it is beyond the scope of this thesis to exhaustively survey the vast literature. This chapter will largely discuss the literature that is most relevant to our work. For a more thorough treatment, we refer the reader to comprehensive surveys by Abu Rumman and Fratarcangeli (2016), Nealen et al. (2006), Huang et al. (2019).

2.1 Artistic Animation

The first and most important ingredient of a visually pleasing character animation is a set of motions that conveys the main message of the scene. Before diving deeper into the means of creating such an animation, let us define what it means to animate a character. To create the illusion of motion in a computer, multiple images are rendered and played in quick succession (Marschner and Shirley, 2018). Bringing this concept to characters, animating a character means generating multiple time-dependent state / variation of our character model. In the context of artistic animation, we will use the notation $\boldsymbol{\nu}^t = \left[\boldsymbol{\nu}_1^t, \cdots, \boldsymbol{\nu}_N^t\right]^T$ to represent the state (*pose*) of an object mesh at time t, where $\boldsymbol{\nu}_i^t$ represents the position of an individual vertex i for that time [1]. An animation $\mathbf{A}$ with k frames is then defined as a sequence of states $\mathbf{A} = \{\boldsymbol{\nu}^0, \boldsymbol{\nu}^1, \cdots, \boldsymbol{\nu}^k\}$ that is played sequentially with constant time in between. Another important terminology is the *rest-pose*, which refers to the initial state of the character mesh $\boldsymbol{\nu}^0$.

2.1.1 Per-vertex Animation

The simplest form of animation can be achieved by requiring artists to define the position of every vertex at a specific time t (Marschner and Shirley, 2018). This technique has a multitude of names, sometimes called *per-vertex* animation, *shape interpolation*, *blendshapes*, *shape keys* or *morph-targets* animation. In per-vertex animation, the position of each vertex is defined independently of each other; meaning that the movement of a single vertex does not influence other vertices, and there are no correlations between each position in time. This animation paradigm offers absolute control over all aspects of the animation, as it is possible to define the individual position of each vertex without any restriction (Liu, 2009). Evidently, specifying every single vertex for each frame of an animation is no easy task. As such, the main goal of various works in per-vertex animation is to find a way to perform this task more efficiently.

In hand-drawn animation, the key (senior) artists often only draw a few important poses (*key* frames), the poses in between are filled by their assistant (Williams, 2012). Digital animation borrows the same idea to help reduce the workload of the animator. But instead of an actual assistant, we use a technique known as *tweening* or *interpolation*. The key idea is to limit the number of poses ($\boldsymbol{\nu}$) we have to specify by hand. By automatically filling the gap between key frames with predicted/transitional frames, we can take a significant chunk of work off the animator's hands.

This concept is best understood with an example. Suppose we have the position of 3 key poses: $\boldsymbol{\nu}^5, \boldsymbol{\nu}^{10}$ and $\boldsymbol{\nu}^{15}$ (shown as teal dots in Figure 2.1). To find the position in between, we can

[1] In physics-based animation, the term "particle positions" or just "positions" is more commonly used. It also uses a different symbol, $\mathbf{x}$. However, they ultimately meant the same thing.

use a piecewise linear function (straight lines) to connect each dot, and pick points in this line to create "intermediary" poses (pink dots in Figure 2.1). In practice, using a piecewise linear function to interpolate between poses results in "rough" and "jumpy" motion, observable by the presence of sharp edges/turns between key poses. For this reason, interpolation that produces smoother trajectory, such as Spiro (Levien, 2009) and Catmull-Clark (Halstead et al., 1993), are preferred over linear interpolation despite their more complex nature. The green dots in Figure 2.1 shows the difference in trajectory when smooth interpolation is used.

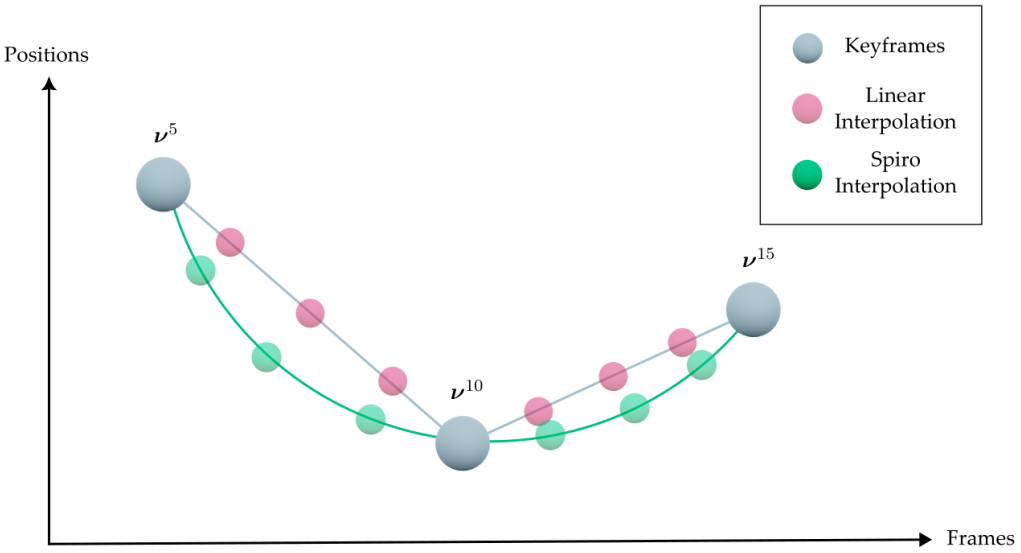


FIGURE 2.1: Trajectory created from interpolation between 3 key poses. Notice the sharp turns on every key poses with linear interpolation, in contrast to the gradual turns on poses with Spiro interpolation.

Another attempt at reducing the required workload of per-vertex animation comes in the form of motion capture (Kitagawa and Windsor, 2020). Unlike interpolation, motion capture does not help to reduce the number of poses we need to specify, but makes it easier to define a single pose. Traditionally, poses are created by manually moving vertices around inside a 3D software (Williams, 2012). Motion capture instead uses specialised hardware to capture the vertex position from a corresponding object in the real-world (Figure 2.2). Not only does this help with creating a more accurate depiction of a real-life motion, it also completely frees the artist from the laborious task of specifying vertex positions manually.

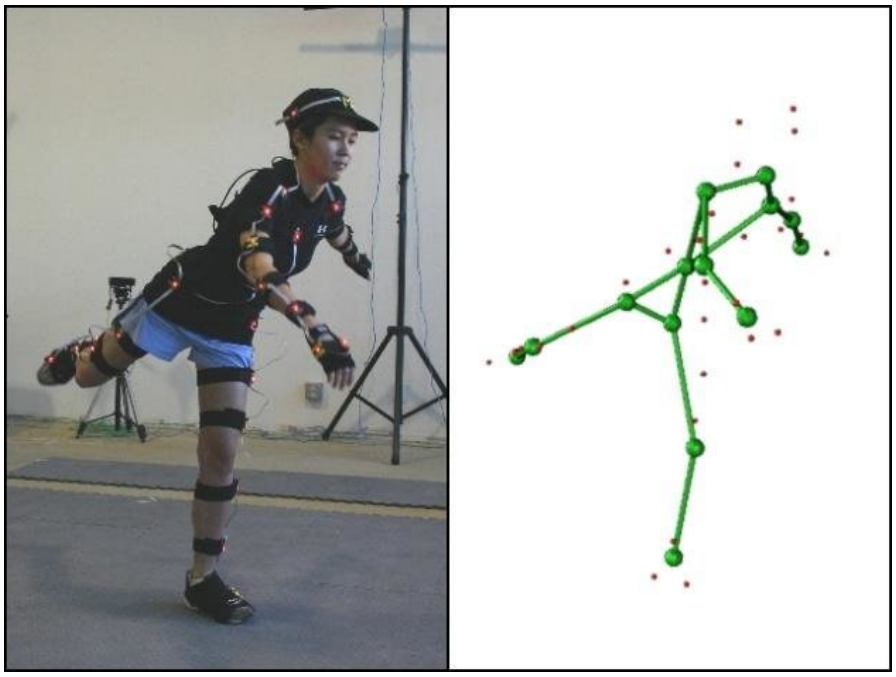


FIGURE 2.2: Figure of a motion capture device alongside a visualisation of its captured data. Notice how vertices (points in the right image) are defined by the position of the markers (red dots in the left image). Taken from Jauregui (2011) without permission.

2.1.2 Deformation

Advances in per-vertex animation do ease off the painstaking process of creating an animation, but we cannot deny that creating an animation by manually specifying vertex position one-by-one still requires a significant amount of labour (McLaughlin et al., 2011). This might be acceptable for small, simpler 2D animations that were the standard decades ago, but as we move towards animating complex 3D geometry, this approach gets considerably less feasible. To illustrate the complexity of per-vertex animation, a modern, high quality 3D object has at least 50,000 vertices. This implies that an artist has to specify the positions of 50,000 points for a single frame of animation, which also entails high requirement of artistic capability, as the probability of moving 50,000 vertices while making sure the resulting shape looks "right" is astonishingly small.

Nowadays, most artistic animations are created through a strategy called deformation (Marschner and Shirley, 2018). The realisation that vertices should be moved together as a group — as opposed to individually, gave birth to this strategy. This is achieved by using intermediary *handles*[2] that only allow indirect shape changes. In this method, the artist can only manipulate the state of the intermediary handles, but cannot modify the position of vertices directly. This

[2]In this context, handles are *any* modifiable parameters that affect the shape of the object. While it is highly synonymised with skeletons, other simpler, adjustable forms of parameters like simple scalars and bounding boxes are also a valid handle.

review will focus on work that utilise *skeleton*, the most common form of handle used in character animation. We refer the readers to the comprehensive survey by Jacobson et al. (2014) for other forms of handles used in deformation.

Skeleton-based deformation is inspired by the internal bone-joint structure of the human body, where the rotation of bones and joints define the visible shape of the visual exterior (Figure 2.3). The purpose of having an internal structure like skeleton is twofold: firstly, since the number of skeletons are significantly less than the number of vertices, it still satisfies the reduced-workload goal of deformation; secondly, this internal structure works as a constraint to limit the changes we made to the one that preserve the object's shape (Abu Rumman and Fratarcangeli, 2016).

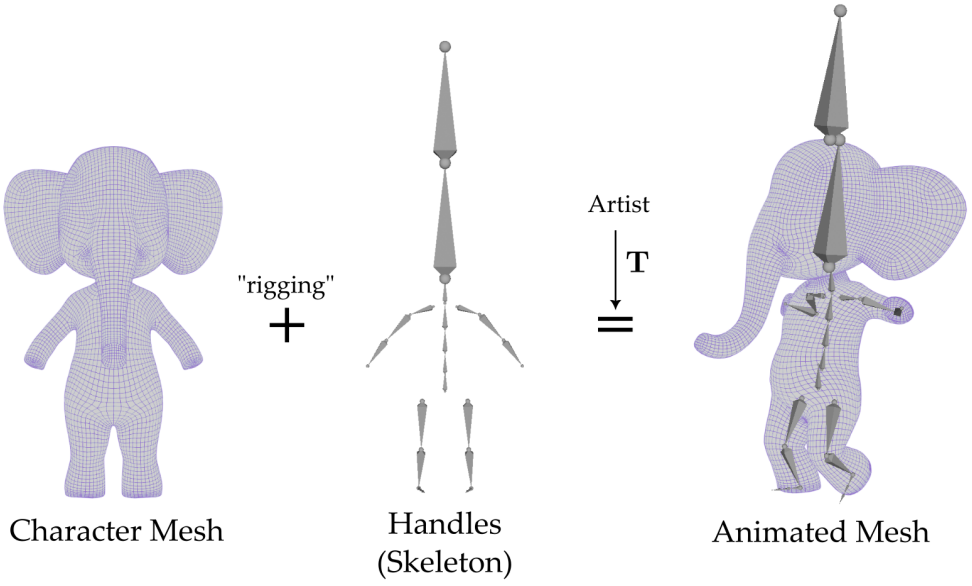


FIGURE 2.3: The standard workflow of skeletal deformation. A mesh (left) is combined with a skeleton handle (middle) whom transformations are defined by an artist. The result (right) is a deformed mesh that represents the next frame of the animation.

All skeleton-based deformation algorithms require 2 key inputs (Lewis et al., 2000). Given a character mesh with N vertices and m bones, these inputs include:

- **Rest pose**, $\boldsymbol{\nu}^0 = \left[\boldsymbol{\nu}_1^0, ..., \boldsymbol{\nu}_N^0\right]^T$, is the state of each vertex in the object we are animating before any animation is applied.
- **Skeleton Transforms**, denoted by the transformation matrices $\mathbf{T}_1, \mathbf{T}_2, \cdots, \mathbf{T}_m \in \mathbb{R}^{3\times3}$, represent the state (position, and rotation) of all bones for the current animation frame. This input is manually defined by the artist with the help of a 3D animation software. It is the only quantity that changes overtime in skeletal deformation.

The works in skeletal-based deformation differ in how they map these two inputs into a change in vertex position, while satisfying the conflicting requirements of believability and real-time

performance (Abu Rumman and Fratarcangeli, 2016). On one hand, running in real-time is an absolute requirement since skeletal deformation cannot be precomputed and has to be computed in runtime. On the other hand, many simple deformation functions will produce unsightly artefacts or unrealistic bulging in the character skin. To better understand the development of deformation functions, we will start with the simplest possible mapping: *rigid* skinning (Marschner and Shirley, 2018).

In rigid skinning, each vertex is attached to a single skeleton, and the deformation function is the identity function. In other words, any positional changes applied to the skeleton are mapped one-to-one to changes in vertex positions, as exemplified in Figure 2.4.

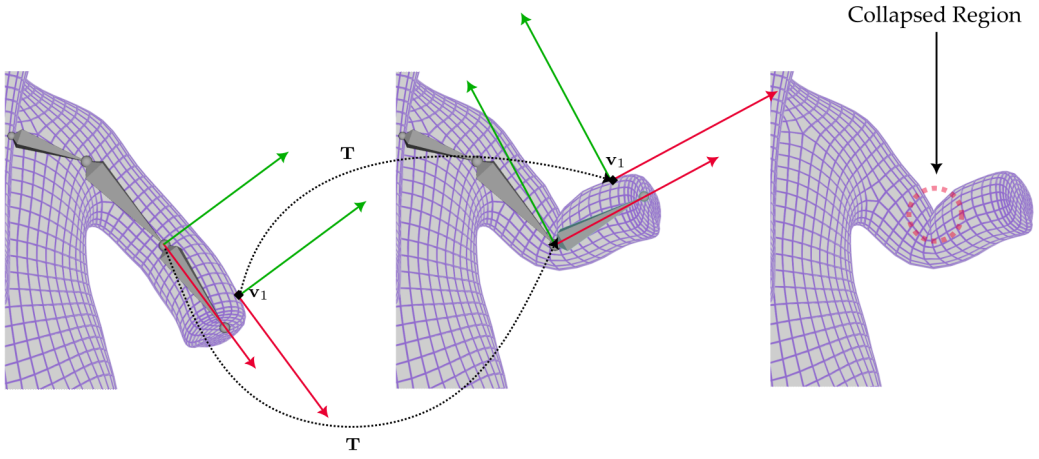


FIGURE 2.4: An example to illustrate the idea of rigid skinning. A transformation $\mathbf{T}$ is applied to the bone whom the vertex $\boldsymbol{\nu}_1$ is attached to. The result (middle) shows the vertex $\boldsymbol{\nu}_1$ receiving the same transformation. Rigid skinning often causes collapses near the joint (connection point of two bones), as shown in the rightmost figure.

Formally, for the i-th vertex that is attached to the j-th skeleton, the position of that vertex in the next frame $\boldsymbol{\nu}'_i$ is given by:

$$\boldsymbol{\nu}'_i = \mathbf{T}_j \boldsymbol{\nu}^0_i. \tag{2.1}$$

It goes without saying that rigid skinning is both fast and easy to prepare, but produces a conspicuous artefact around the joints (connection points between skeletons). The vertices around this area will penetrate each other, creating discontinuities in the character skin (rightmost illustration in Figure 2.4).

This artefact around joints makes rigid skinning impractical for any character that has multiple skeleton connections[3] (Vasilakis and Fudos, 2009). In an attempt to remedy this discontinuity around joints, Kalra et al. (1998) introduced the first version of smooth skinning, *linear blend skinning* (LBS). LBS can be seen as a generalisation of rigid skinning, where a vertex can attach

[3]This, unfortunately, includes all forms of humanoid character.

to multiple skeletons and their influence *blended* using a linear function. To achieve this, LBS introduced a third input, *weights*. For every vertex i and bone j, we have a scalar $w_{i,j}$ that represents the *influence* of bone j to the position of vertex i. A common requirement is that $w_{i,j} \geq 0$ and $w_{i,1} + w_{i,2} + ... + w_{i,m} = 1$ (partition of unity). Now, the next state of the i-th vertex can be computed with the formula:

$$\boldsymbol{\nu}'_i = \left(\sum_{j=1}^{m} w_{i,j} \mathbf{T}_j \right) \boldsymbol{\nu}^0_i. \tag{2.2}$$

The impact of using a linear combination instead of one-to-one mapping to the smoothness of the resulting deformation is quite significant. Linear blend skinning offers controllable, smooth deformation while keeping the computational cost at a minimum (Jacobson et al., 2014). For this reason, despite being introduced more than 3 decades ago, linear blend skinning is the industry standard skinning algorithm and is used inside the 3D animation software Blender (Foundation, 2018).

Being the industry standard does not mean LBS is without its own sets of problems. LBS introduced an additional authoring cost: determining weights. Selecting good weights is critical, as poorly selected weights hampers the visual quality worse than having no blending at all (Jacka et al., 2007). While automated weight-computing algorithms exist (Abu Rumman and Fratarcangeli, 2015, Jacobson et al., 2012, Dionne and de Lasa, 2013), production teams usually still leave this in the hand of another artist, often involving an endless cycle of trial-and-error. The unfortunate thing is, even with the most perfectly crafted weights, some artefacts are still bound to happen because of the linear nature of LBS (Jacobson et al., 2014), with the most infamous one being the "candy-wrapper" effect shown in Figure 2.5.

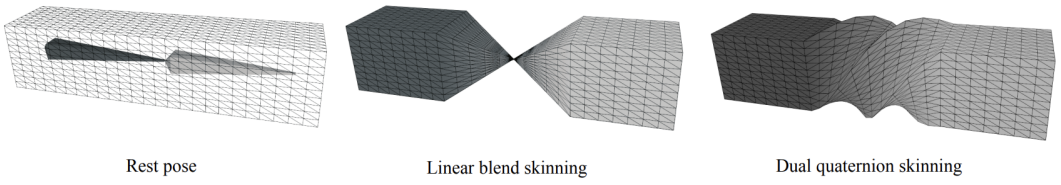


FIGURE 2.5: The infamous *candy-wrapper* effect on linear blend skinning. This happens when two rotations of opposing direction are linearly blended, causing surrounding areas to collapse to a single point. Notice that the same problem does not happen when nonlinear blending (right) is used. Taken from (Jacobson et al., 2014) without permission.

This is a strong motivator to then replace linear blending with nonlinear blending (Alexa, 2002), giving birth to work such as *spherical* skinning (Kavan and Žára, 2005) and *dual-quaternion* skinning (DQS) (Kavan et al., 2007). Both methods completely avoid the candy-wrapper effect, but with an expensive cost of simplicity and performance. For now, these additional costs are

not justified when nearly the same effect can be achieved with cheaper methods (Lee et al., 2013). It should be pointed out, however, that as the computational budget for interactive applications increase, we should see increased adoption of non-linear skinning algorithms, and eventually replacing LBS completely. The introduction of non-linear blending algorithms marks the end of the "blending era" in skeletal animation.

The next generation of skeletal deformation algorithm involves what is known as *implicit* skinning. Implicit skinning no longer focuses on finding a direct mapping between skeleton transformation and vertex positions, rather emphasising on fixing the imperfections of no-weights blending (rigid skinning) using sophisticated geometry processing algorithms (Jacobson et al., 2014). These more advanced algorithms are largely beyond the scope of this thesis, but we will quickly mention a few standouts to give the reader an up-to-date idea of the current landscape of skeletal deformation algorithms. *Delta Mush* (Mancewicz et al., 2014, Le and Lewis, 2019) starts with rigid skinning, then smooths out the surface of the character using Laplacian smoothing, hence the name "mush". This is a costly iterative method that is primarily targeted towards non-real time use cases like Weta's Digital Tissue System[4] and Maya Muscle[5]. A substantial amount of work (Xu et al., 2020, Ouyang and Feng, 2020, Chen et al., 2021) also utilises machine learning style function approximators (neural networks) in place of the traditional blending algorithms. To help the readers differentiate all the methods mentioned here, we summarised all skeletal deformation methods in Table 2.1.

Table 2.1: Summary of various skeletal deformation strategies.

Quality ↑

Source	Year	Operation
Xu et al. (2020), Ouyang and Feng (2020)	2020	Neural network
Kavan and Žára (2005)	2005	Singular-value decomposition (SVD)
Mancewicz et al. (2014), Le and Lewis (2019)	2014;2019	Laplacian smoothing
Kavan and Žára (2005)	2007	Dual-quaternion blending
Kalra et al. (1998)	1988	Linear blending
Rigid skinning	unknown	One-to-one map

↓ Performance

[4] urlwww.wetafx.co.nz
[5] https://www.autodesk.com/products/maya

2.2 Physics-based Animation

In spite of the various improvement in the field of artistic animation allowing artists to produce high quality animation at a rapid pace, artistic animation is not a one-size-fits-all solution. Certain animation tasks, such as animating fluid, smokes, and the swaying of hair strands are physical in nature. Trying to tackle these tasks artistically will result in unrealistic motions that are jarring to the common observer. To achieve the goal of realism, researchers have used simulation techniques from physics to model the behaviour and movement of these entities. Today, these efforts have culminated in what is usually referred to as *physics-based* animation (Erleben et al., 2005).

In contrast to artistic animation where a human animator provides the state of a character one way or another, physics-based animation takes a set of physical quantities (positions, velocities, and forces) from the previous state and uses a mathematical model to determine the next state. Other than some small subset of modifiable parameters, the animator has no control on how the resulting motion will look like — the main reason for its lack of use in primary motion (Marschner and Shirley, 2018). In this section, we will dive deeper into the inner workings of physics-based animation and the various works that surround it.

2.2.1 Governing Equations

The world around us is governed by physical laws, many of which can be formalised as sets of mathematical equations. In classical mechanics, the equations that govern the motion of a physical object are known as Newton's differential equations of motion (Taylor, 2005):

$$\mathbf{F}_{external} + \mathbf{F}_{internal} = \mathbf{M}\frac{\partial^2 \mathbf{x}(t)}{\partial t^2}. \tag{2.3}$$

In this equation, t represents time, $\mathbf{M}$ represents the mass matrix, $\mathbf{F}_{external}$ represent the forces that are applied to a point (or object), and $\mathbf{F}_{internal}$ represents the response forces that are exerted by the object/material. These forces push and pull an object around in space, which causes acceleration ($\frac{\partial^2 \mathbf{x}(t)}{\partial t^2}$) and, in turn, changes in position.

For computational convenience, Newton's differential equation is often rewritten into a system of first order differential equations by introducing the *velocity* $\mathbf{v}$ (not to be confused with vertex position $\boldsymbol{\nu}$):

$$\frac{\partial}{\partial t}\begin{pmatrix}\mathbf{x}(t)\\ \mathbf{M}\mathbf{v}(t)\end{pmatrix} = \begin{pmatrix}\mathbf{v}(t)\\ \mathbf{F}_{external} + \mathbf{F}_{internal}\end{pmatrix}. \tag{2.4}$$

Much of physics-based animation revolves around solving this equation inside a computer, which allows one to compute the future state of a character's mesh in a physically realistic manner. Much to our dismay, computer animations are discrete collections of states — not immediately computable from a continuous equation like Equation 2.4. In order to get the state of an object at a specific time t, we need to *discretise* the equation with a numerical integration scheme (LeVeque, 2007).

Numerical integration is done by replacing (continuous) derivative operator (∂) with its finite difference counterparts (Δ). In computer graphics, the simplest form of numerical integration is *explicit* Euler (Nealen et al., 2006), where $\frac{\partial \mathbf{v}(t)}{\partial t}$ is replaced by $\frac{\mathbf{v}(t+\Delta t)-\mathbf{v}(t)}{\Delta t}$ (and similarly for $\frac{\partial \mathbf{x}(t)}{\partial t}$). Δt is a value denoting small changes in time, often called timestep. To help differentiate the governing equation and its discretised solution, it is a common convention to use the subscript notation f^{t+i} to denote $f(t + i\Delta t)$. Adopting this notation and performing the aforementioned substitution into Equation 2.4 gave us:

$$\begin{pmatrix} \mathbf{x}^{t+1} \\ \mathbf{v}^{t+1} \end{pmatrix} = \begin{pmatrix} \mathbf{x}^t + \Delta t \mathbf{v}^t \\ \mathbf{v}^t + \Delta t \mathbf{M}^{-1}(\mathbf{F}_{external} + \mathbf{F}_{internal}) \end{pmatrix}. \tag{2.5}$$

This discrete scheme in theory gave us all the required information to generate our animation, however, it is rarely used in computer graphics because of its instability and inaccuracy. If the timestep Δt being used is too large, the object simulated with this scheme often overshoots its intended trajectory and eventually explode (gains velocity indefinitely) (Müller et al., 2005). The solution is to reduce the timestep size, but that comes at the cost of performance. The smaller steps we take, the more times we need to evaluate Equation 2.5.

The alternative scheme, which is unconditionally stable even with larger timestep, is first brought to the field of physics-based animation by Baraff and Witkin (1998). This alternative scheme, often called *implicit* Euler, is remarkably similar to explicit Euler, with the main difference being the use of the quantities at time $t + \Delta t$ on both sides of the equation:

$$\begin{pmatrix} \mathbf{x}^{t+1} \\ \mathbf{v}^{t+1} \end{pmatrix} = \begin{pmatrix} \mathbf{x}^t + \Delta t \mathbf{v}^{t+1} \\ \mathbf{v}^t + \Delta t \mathbf{M}^{-1}(\mathbf{F}_{external} + \mathbf{F}_{internal}) \end{pmatrix}. \tag{2.6}$$

The scheme is called implicit since according to Equation 2.6, the formula to compute unknowns $\mathbf{x}^{t+1}$ and $\mathbf{v}^{t+1}$ are not immediately obvious, and instead is given implicitly as the solution to the system of equations above. As we will see later in this section, depending on the choice of $\mathbf{F}$, Equation 2.6 can be *non-linear*. For this reason, the textbook approach to compute a single step of implicit Euler requires Newton-Raphson method (Bargteil and Shinar, 2018) — an iterative algorithm that solves a system of nonlinear equations by making an educated initial guess and improving it over time. This makes the cost per timestep of implicit Euler much more

expensive than explicit Euler. However, the fact that a way larger timestep size can be used usually makes up for it. Additionally, implicit Euler exhibits phenomena known as *numerical damping*, where a small amount of energy will dissipate on each timestep (LeVeque, 2007). Motions in the real world also exhibit similar visible energy dissipation caused by natural forces like wind, gravity, and friction (Cutnell and Johnson, 2009), making this scheme desirable for simulation of physical phenomena in computer graphics.

2.2.2 Material Model

One important quantity that we have not discussed from the Newton's differential equation is the force ($\mathbf{F}$). As shown in Equation 2.4, there are 2 forms of forces, internal and external. External forces are the "input" of the simulation and are used to allow interaction between physics-based animation and other forms of animation (such as primary motion). Internal forces, on the other hand, are the innate reactionary forces from the object's underlying material to the "input". In layman terms, internal forces are the physical quantity that actually determines the realism of the generated motion.

The way we compute internal forces of a specific material depends on how an object is modelled physically. There are a multitude of ways to physically model an object, but this review will focus on three of the most commonly used material model in computer graphics: mass-spring systems, continuum-based model, and position-based dynamics. For less commonly used material models, refer to the comprehensive survey by Nealen et al. (2006).

2.2.2.1 Mass-spring systems

Arguably the simplest and most intuitive way to model a physical object is by considering the object's vertices as point masses connected by a network of springs (Nealen et al., 2006). The deformation (shape changes) of the object then comes as a result of the spring compression and extension. Appropriately, in this model, the internal forces are the spring forces as defined by Hooke's law (Taylor, 2005):

$$\mathbf{F}_{internal}(t) = -k\mathbf{x}(t), \tag{2.7}$$

where k is the spring constant that defines the stiffness of the object. The pioneering work that brought this formulation to the field of computer animation was conducted in 1987 by Terzopoulos et al. (1987), where they showed that the mass-spring model can simulate the behaviour of objects made from rubber and paper. Because of its simplicity and ease of implementation, mass-spring systems quickly gained traction in simulation of various objects such

as faces (Zhang et al., 2001, Terzopoulos and Waters, 1990, Lee et al., 1995), muscles (Nedel and Thalmann, 1998) and hairs (Selle et al., 2008). Shortly after, we started to see the aggregation of these works in the form of full character locomotion, which animate an entire body of characters such as worms (Miller, 1988) and fish (Tu and Terzopoulos, 1994). Depicted in Figure 2.6, is a simple example of a fish character modelled with a mass-spring system.

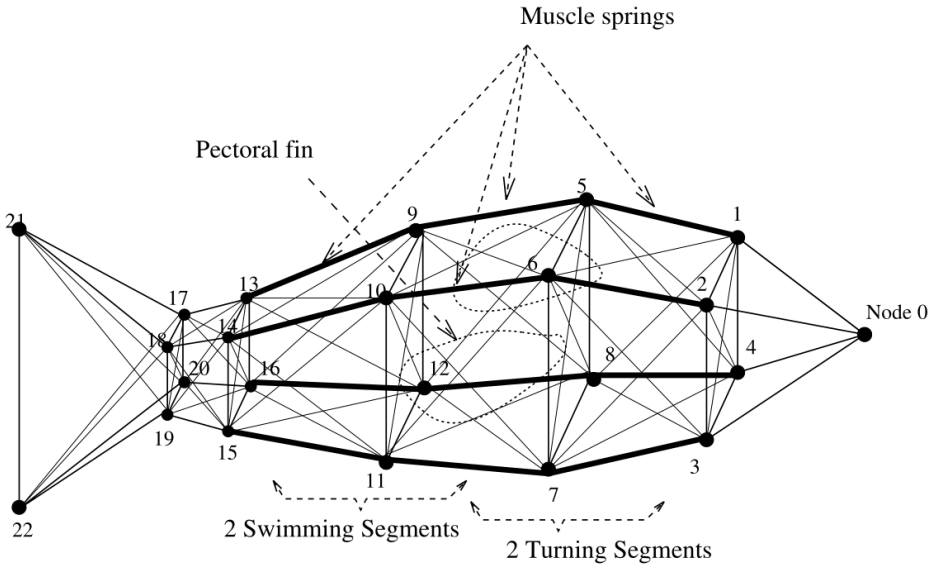


FIGURE 2.6: Internal structure of a fish character modelled with a mass-spring system. Taken from (Tu and Terzopoulos, 1994) without permission.

With almost all forms of application in place, the next challenge is how to simulate mass-spring systems more efficiently. A plethora of works (Kenwright et al., 2011, Hong et al., 2016) have tried to improve the simulation speed of mass-spring systems, but most of them are considered obsolete because of the rapid growth of computational power available to us in the last 20 years. A couple of works, however, stood out as its contribution benefited models beyond the mass-spring system. In Section 2.2.1, we mentioned that to compute physically realistic motion using an implicit scheme, we would need to solve a system of equations. Martin et al. (Martin et al., 2011) introduced the idea of *variational-implicit* Euler, which reformulate the problem into an optimisation problem. They showed that computing the next state $\bar{\mathbf{x}}$ is mathematically equivalent to:

$$\arg\min_{\bar{\mathbf{x}}} \frac{1}{2\Delta t^2}\text{tr}((\bar{\mathbf{x}}-\mathbf{y})^T\mathbf{M}(\bar{\mathbf{x}}-\mathbf{y})) + E(\bar{\mathbf{x}}), \tag{2.8}$$

where $\mathbf{y}$ is a shorthand for $2\mathbf{x}^t - \mathbf{x}^{t-1}$ and $E = \int \mathbf{F}_{internal}\, d\mathbf{x}$. Reframing it into an optimisation problem gave us access to more tools with varying characteristics. For example, Liu et al. (2013) utilised the aforementioned formulation alongside an optimisation algorithm called *Block-coordinate descent*. Compared to Newton-Rapshon, the traditional solver, block-coordinate

descent is faster but less accurate (Figure 2.7). For visual applications where speed and stability are of greater importance, this work offers visually acceptable results at a fraction of the computational cost.

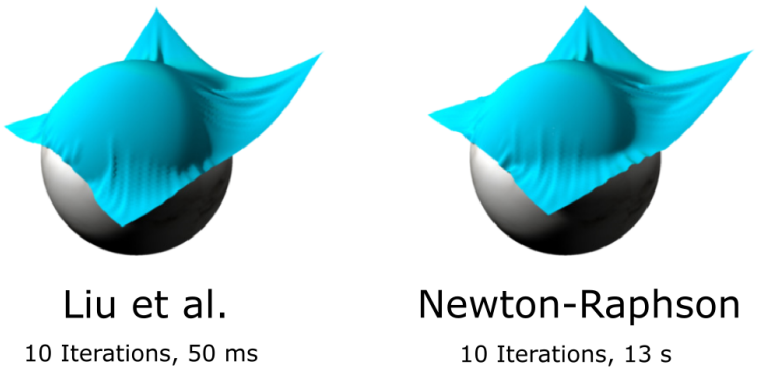


FIGURE 2.7: Visual and performance differences of two mass-spring cloths simulated with Block-coordinate descent (Liu et al., 2013) and Newton-Raphson. Block-coordinate descent offers comparable result despite being orders of magnitude faster. Adapted from Liu et al. (2013).

Another reason for the swift and wide adoption of mass-spring systems is that they innately support effects that require topological changes in the character's mesh. Effects such as fracture, tearing, and cutting that relies on runtime addition/removal of edges and vertices can be simulated with no issue (Nealen et al., 2006). Since masses are synonymous to vertices and springs are synonymous to the edges, it is theoretically possible to add a single mass or spring to the simulation for every vertex and edge we add to the character mesh (Figure 2.8). In practice, however, this trivial strategy does not always generate satisfactory results. Adding springs arbitrarily will result in the object becoming more rigid — an unexpected behaviour for an object that is being torn or cut. Approximately 15 years after the first introduction of mass-spring systems, Bridson et al. (2005) introduced *dynamic mesh reconstruction*, a technique to support cutting/tearing effects while preserving the original material behaviour. Instead of adding or removing springs with one-to-one correspondence to the addition/removal of edges, the springs are *redistributed* to the surrounding area affected by the edge addition/removal. Until today, dynamic mesh reconstruction is still the preferred way to model fracture and cutting for a deformable object, and is heavily used in use cases like virtual surgery and other forms of interactive simulations (Hammer et al., 2011).

One might notice that despite being simple and highly applicable, most of the work mentioned above is relatively old. This has to do with the main drawbacks of mass-spring systems, *visual quality* and *accuracy*. The reasoning has been hinted in the previous paragraph, where we touch

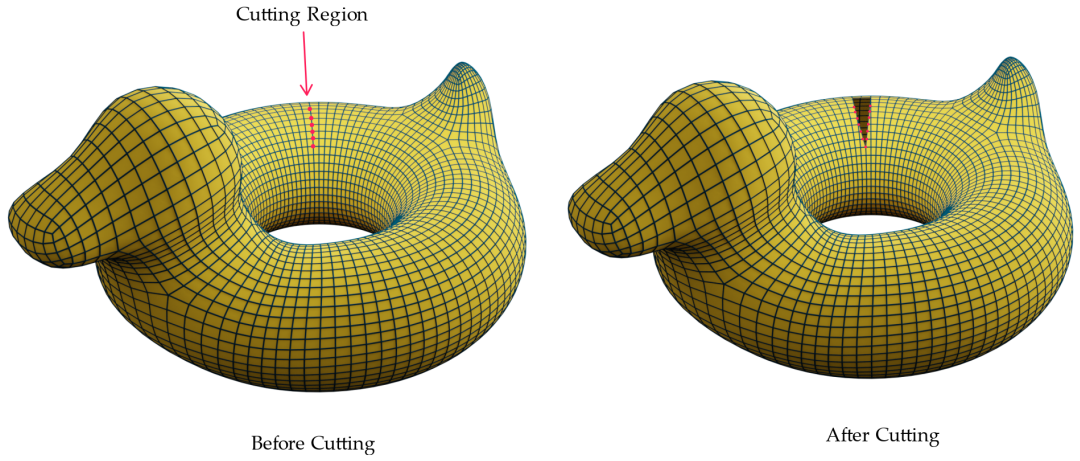


FIGURE 2.8: A rubber duck modelled with mass-spring systems before (left) and after (right) a cut. The red dots represent the removed vertices (before cutting) and the newly added vertices (after cutting).

on the fact that the behaviour of mass-spring systems is *topology-dependent*. Even with identical simulation parameters[6], the runtime behaviour of the simulation can vary depending on the shape of the character being simulated. While it seems harmless at first, the implication can be a dealbreaker. For instance, given two different characters composed of the same material, the two can behave very differently if modelled through mass-spring systems — a definite consistency nightmare (Huang et al., 2019). Additionally, mass-spring systems offer only 1 modifiable parameter, the *spring constant*. This parameter has little to no meaning in terms of the material being modelled and is often chosen arbitrarily (Nealen et al., 2006). This is fine when the goal is to simulate a generic squishy deformable object, but is insufficient when the animation calls for a specific material (e.g. rubber, fat tissue, or plastic). Lastly, since the simulation of mass-spring system happens only on each mass (vertex) and spring (edges), the empty spaces between vertices and edges would be unresponsive to any input (external forces). This can obviously be remedied by using a higher density (high resolution) mesh where the gaps between vertices and edges are smaller, but that would slow down the simulation speed. These inaccuracies might be tolerable back then, but as the visual fidelity requirement increase, most works have made the jump to a different construction known as *continuum-based* models (Huang et al., 2019).

2.2.2.2 Continuum-based Models

Consistent with its name, continuum-based models refer to a group of models which are derived from *continuum-mechanics* (Müller et al., 2007). Instead of spring forces, continuum-based

[6]Simulation parameters in this context refers to the stiffnes constant k.

models deal with elastic forces:

$$\mathbf{F}_{internal}(t) = -\nabla \cdot \mathbf{P}. \tag{2.9}$$

where ∇ is the vector differential operator and $\mathbf{P}$ is often called the *first Piola-Kirchhoff stress tensor*. Each continuum-based model has a different definition of $\mathbf{P}$. The simplest continuum-based model, the *linear elasticity* model, defines $\mathbf{P}$ as below:

$$\mathbf{P} = \mu(\mathbf{f} + \mathbf{F}^T - 2\mathbf{I}) + \lambda \text{tr}(\mathbf{f} - \mathbf{I})\mathbf{I}. \tag{2.10}$$

$\mathbf{f}$ here represents a quantity called deformation gradient (not to be confused with $\mathbf{F}$ which represents force) and $\mathbf{I}$ represents identity matrix. μ and λ, often called *Lamé* coefficients, are modifiable parameters that define the material being modelled. If we substitute $\mathbf{P}$ (Equation 2.10) back to Equation 2.9, followed by substituting $\mathbf{F}_{internal}(t)$ back to the Newton's differential equation (Equation 2.3), we would arrive at a *partial differential equation*. Partial-differential equation (in contrast to ordinary differential equation), requires additional technique called *finite element method* to solve — which are both mathematically involved and computationally expensive (Sifakis and Barbič, 2015). As a tradeoff, being based on continuum mechanics allows tight integration with material science. This allows *Lamé* parameters to be obtained experimentally, allowing any types of material to be captured with innate accuracy (Nealen et al., 2006). To showcase this innate accuracy, Figure 2.9 presents a simulation of an elephant's flesh with continuum-based model.

FIGURE 2.9: Four frames of elephant flesh animation simulated with a continuum-based model. Taken from Zhang et al. (2020) without permission.

Consequently, the majority of works surrounding continuum-based model are not targeted at the visual features of the model, but instead its performance. These performance works can be categorised into 3 major groups: *model reduction*, *computational optimisation*, and *non-model improvements*.

The first body of work, *model reduction*, relied on deriving a simplified version of our model by reducing the *degrees of freedom* (DOF) of the system (James and Pai, 2002). Essentially,

reduced-order models tried to utilise mathematical insight to reduce the amount of variables needed to represent a system. One trivial example is a fixed-length pendulum in 2D. Although we can trivially represent the state of the system with 2D Euclidean coordinates (2 variables x and y), we can better represent the state with only one variable θ, the angle of the pendulum relative to its suspension point (Amirouche, 2006). This reduces the system's degrees of freedom from two to one, effectively transforming it to a simulation of a 1D object. The weakness of this approach is quite clear. Unlike the pendulum example, not all problems have a reduced-order version without significantly compromising visual quality (Teng et al., 2015). At the same time, a successful application of this approach, such as the one seen on Figure 2.10, can lead to the biggest improvement in performance as it significantly cuts down the state space.

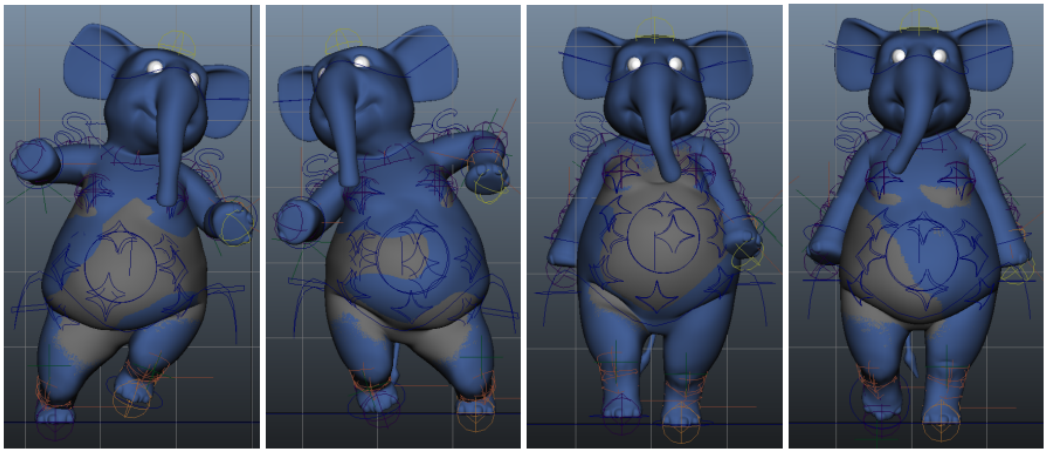


FIGURE 2.10: Application of model reduction to an elephant model by Hahn et al. (2012). The reduced-order model (grey, 6 DOFs) is laid on top of the complete model (blue, 30 DOFs). Taken from Hahn et al. (2012) without permission.

The works in the *computational optimisation* category aimed to find a way to accelerate the numerical method used to solve the governing equation. The majority of the work in this category made heavy use of variational implicit Euler formulation introduced previously (Equation 2.8). Similarly to mass-spring systems, the optimisation formulation opened doors to many more potential solvers (Huang et al., 2019). The first order necessary condition for optimality is that the gradient of the cost function equals zero (LeVeque, 2007). If we do so according to Equation 2.8, evaluation of $\nabla^2\mathbf{F}_{internal}$, the *hessian* of the internal force, is required. Unlike mass-spring systems where the hessian will be constant, the hessian in a continuum-based model is time-dependent, meaning that it will need to be recomputed every timestep.

Liu et al. (2017) noticed this computational bottleneck, and proposed the idea of using *Quasi-Newton* in place of the Newton-Raphson method. Quasi-Newton method works by approximating the hessian with $\mathbf{M}$ (the mass matrix). Without the information from hessian, the optimisation understandably requires a few extra iterations to converge. This is a justifiable side effect, however, since skipping hessian computation enables significantly more iterations

per second (Figure 2.11). In the same spirit, Wang and Yang (2016) adopted gradient-based solver as the main driver of the simulation. Although it is well known that gradient-based approach is not the best choice for physics-based animation due to its weak convergence rate (Barbič et al., 2009), this work showed that clever precomputation and step-size acceleration can compensate for the lack of theoretical performance.

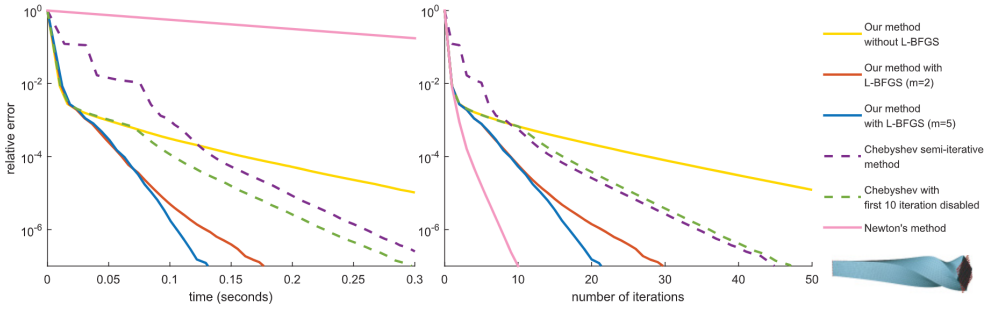


FIGURE 2.11: Convergence characteristics of conventional methods (pink, purple, and green lines) and Liu et al. (2017) quasi-newton methods (yellow, red, and blue lines). From the right graph, it is clear that quasi-newton methods require much higher iteration count to reach comparable levels of error to conventional methods. However, the cost of iterations are significantly cheaper that given any reasonable time budget (left graph), the relative error is significantly less than conventional methods. Taken from Liu et al. (2017) without permission.

Rather than changing the underlying behaviour of the physics simulation, the final group of work aims to non-intrusively improve simulation performance. *Vivace* (Fratarcangeli et al., 2016) used graph colouring techniques to delegate computational task to different threads for parallelization, which kick-started various other research that tried to delegate this workload to cloud and other powerful devices (Li et al., 2013, Danevičius et al., 2018). In addition to parallelisation, non-intrusive performance improvement can also be done geometrically, for example, through *mesh embedding* (Sederberg and Parry, 1986). Mesh embedding refers to the use of a coarse (low-resolution) version of an object during simulation coupled with a high-resolution mesh for display. This means that the final animations are rendered with high-quality 3D mesh with the computational cost of a low-quality mesh. Kim and Pollard (2011) applied this strategy to the simulation of deformable body characters as shown in Figure 2.12 — eventuating in two times speedup compared to using higher-quality mesh directly for simulation. It is important to note that even though the result is mapped to a high-quality mesh, the motion still originates from low-quality simulation, meaning that some visual details might be lost in the process.

In summary, researches have shown various ways to reduce the computational cost of continuum-based models. Model reduction offers the greatest speedup without penalty in accuracy, but is inflexible and problem-dependent. In contrast, non-model methods offer the greatest flexibility since they do not alter the behaviour of the simulation, but often negatively impact the fidelity of the resulting animation. Approaches based on computational optimisation are somewhere in between, offering incremental speedup with a minor loss in quality. Even though it might

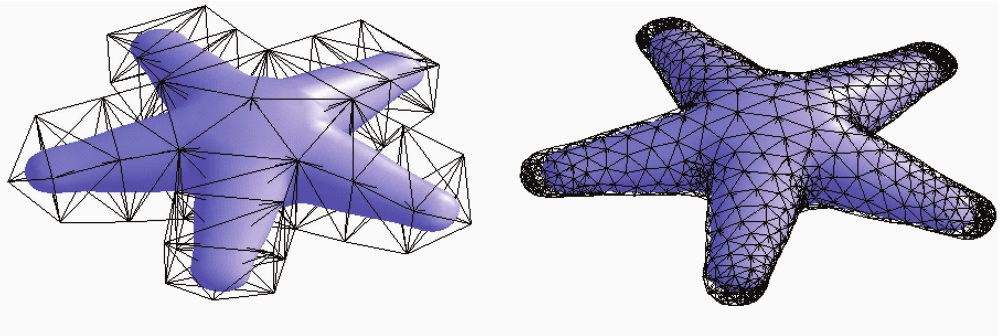


FIGURE 2.12: Visual illustration of mesh-embedding. The wireframes represent the simulation mesh, while the shaded starfish represents the mesh used for display. Traditional simulation uses the same mesh as the display mesh (right), while mesh-embedding uses a significantly lower resolution simulation mesh (left). Taken from Xu and Barbič (2016) without permission.

seem that these improvements are vastly different, these works in are, in the end, incremental improvements to the original continuum-based formulation described in Müller et al. (2007). Principally, they are complementary in nature and offer the most benefit when used together.

2.2.2.3 Position-based Dynamics

The two material models we discussed so far largely follow the *force-based* framework: they modelled the reactionary forces of a material and computed the new state of each entity according to the governing equation (Müller et al., 2008). The third material model we will discuss deviate heavily from this norm. Instead of representing the interaction between entities as forces, we use a set of geometrical constraints applied directly to the position of each entity (Müller et al., 2005). Since this model operate strictly by manipulating positions, it is often called *position-based dynamics* (PBD).

To better illustrate the idea of PBD, we will go through a simple physics-simulation scenario: *bead on a wire* (Erleben et al., 2005). In bead on a wire problem, a non-massless bead is threaded to a rough wire that follows the shape of a circle. Our task is to calculate how the bead moves along the wire. With force-based methods (top half of Figure 2.13), this task translates to enumerating all the forces affecting the beads: (1) the external force ($\mathbf{F}_{external}$), which in this case is gravity; (2) the internal force ($\mathbf{F}_{internal}$), which in this case is the reactionary force from the wire that keeps the beads in it. We then plug every force into the governing equation and solve the equation numerically.

PBD takes a completely different view of the problem (bottom half of Figure 2.13). Instead of enumerating forces, we will impose constraints. Being threaded to a wire means that the bead's distance from the circle's centre point is guaranteed to be the same, and this is the constraint the bead must satisfy. At the start of every PBD step, the bead is allowed to move freely according to gravity. Then, we "fix" the position so that it satisfies the constraint.

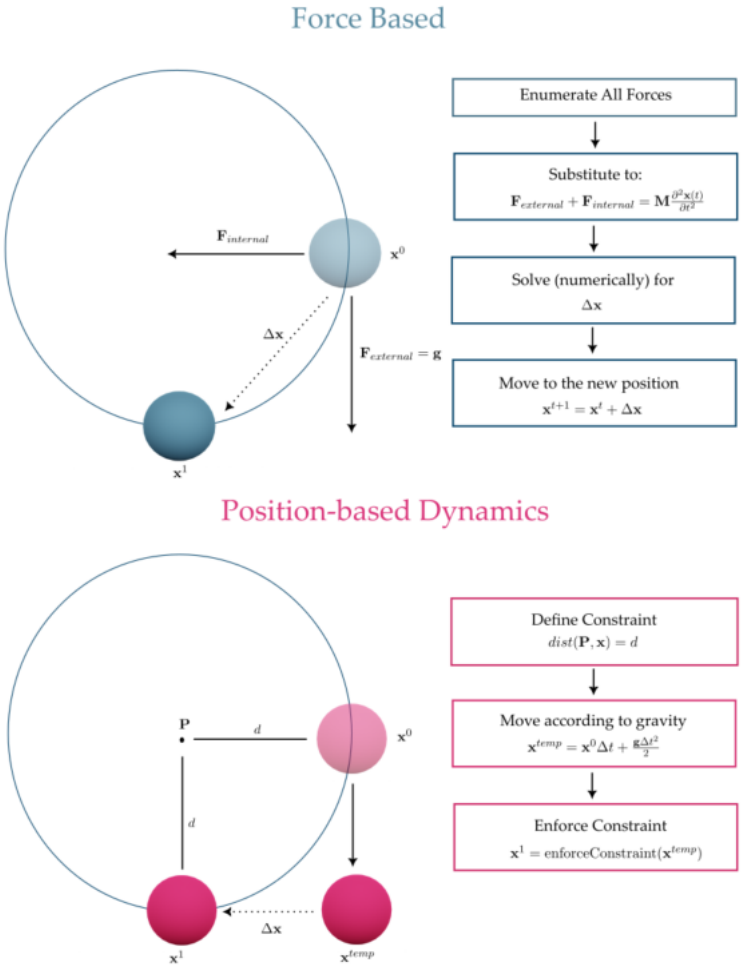


FIGURE 2.13: Steps taken in a force-based approach (top) versus a PBD simulation (bottom) for a classical problem bead on a wire.

Even though it might be clear that both versions can arrive at the same result, one might question whether there is a benefit of using this alternative construction. It turns out, PBD is the preferred simulation method for applications like video games and military simulations due to the significant advantage it has on interactive environments (Bender et al., 2015). No additional step is required to make our PBD simulation interactive, as we simply implement the user interaction normally and bring the system back to a physically realistic state by applying the constraints at the end of each frame. The same could not be said for force-based models, as we are required to find the force representation of every single user interaction in our program. Additionally, the use of constraints guarantees stability and predictability, which is desirable in less restrictive environments like surgical simulation (Pan et al., 2020, Liu et al., 2021). Last but not least, PBD simulation is the fastest of the three material models, as it does not require any of the expensive numerical methods normally used to solve the governing equation (Müller et al., 2008).

Despite being the backbone of the two widely used commercial physics engines: Bullet (Coumans and Bai, 2016–2021) and NVDIA Flex (Macklin and Müller, 2013), PBD is not free from criticism. The physically inclined reader might already question the correctness of PBD as it lacks physical intuition. In fact, the absence of connection to well-established law of physics has been PBD's main weakness since its inception. The original position-based dynamics paper (Müller et al., 2007) only claimed that PBD exhibits *physics-like* behaviour. This concern has since been addressed by Macklin et al. (2016), which showed that PBD can be seen as an approximation of force-based simulation.

Another key challenge for PBD is how we can handle soft/elastic materials (Macklin and Müller, 2013). While force-based approaches do not require any special strategies for soft materials, the same cannot be said for position-based dynamics. If we pay close attention to the premise of position-based dynamics, we would notice that it naturally creates a completely rigid object. Whereas forces operate by gradually moving the object to the correct position, constraint "snaps" the object to the correct position in an instant. Due to this, many early works in PBD introduced their own special constraint to specifically handle particular elastic material. This includes things like cloth (Bender et al., 2011), rubber (Bender et al., 2014), balloon (Bender et al., 2011) and fluids (Macklin and Müller, 2013). Figure 2.14 shows that these special constraints can definitely generate some impressive visuals. However, considering the infinitely many materials in the real world, this approach is not sustainable.

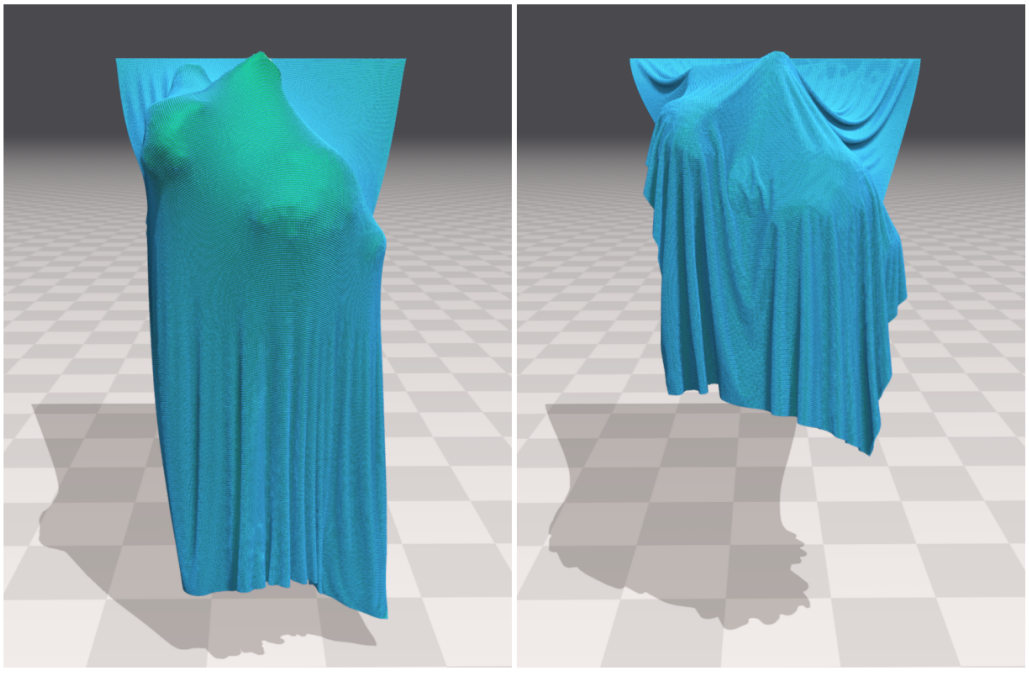


FIGURE 2.14: The cloth wrinkles simulated with a special bending constraint. Taken from Macklin et al. (2019) without permission.

A more general solution to this is actually included in an older work by Müller et al. (2007), which scales the positional change as a result of constraint enforcement by a simple scaling factor $k \in [0, 1]$. This practically delays the constraint enforcement and creates the illusion of an elastic movement. However, as pointed out by Macklin et al. (2016) sometime later, this approach is not ideal either. Elastic motion is not strictly linear (Sifakis and Barbič, 2015), therefore, scaling it linearly does not capture many behaviours properly. Additionally, this workaround instigated another issue with simulation consistency. PBD, like all physics-based animation methods, is an iterative numerical method. With the scaling mentioned above, the more iterations we do, the stiffer the material will be. In other words, our simulation becomes *iteration-dependent*. Being iteration dependent unfortunately is not a desirable trait (McLaughlin et al., 2011), as it means there is no way to tune the visual profile via the usual accuracy-performance tradeoff.

Macklin et al. (2019), the same author that pointed out the above flaws, tried to come up with a solution using the idea of *substepping*. With substepping, we limit PBD to only use a single iteration every timestep. To compensate for the loss of accuracy, we split the timestep into multiple smaller subtimesteps, and perform a single PBD iteration for each subtimestep. Another similar attempt is *higher-order* position-based dynamics (Bender et al., 2015). Higher-order position-based dynamics is a variant of PBD that is derived from BDF2 (backward-difference 2) as opposed to implicit Euler. The main difference this makes is that higher-order PBD uses information from the previous two timesteps as opposed to just the previous timestep, implying that a single iteration in higher-order PBD is much more accurate than standard PBD. The end goal that these two works tried to achieve is quite similar — minimising the number of iterations needed to reach a certain level of physical accuracy. If, for example, we only need a single iteration per timestep, then the impact of the iteration-dependent nature of PBD can be completely nullified.

While these improvements are clearly welcomed, these works are simply an attempt to minimise the visual impact caused by a flaw in PBD's formulation. The proper solution is, of course, to address the incorrectness in the formula itself. As this flaw has not been addressed until today, PBD is mostly used to simulate phenomena that involve infinitely stiff or rigid objects (Müller et al., 2020).

2.3 Animation Pipelines: Combining Physics-based Animation and Artistic Animation

In the previous two sections, we have seen the inner workings of two independent animation paradigms: artistic and physics-based animation. The creative and nonrestrictive nature of

artistic animation made it the clear-cut choice to convey the message of a scene (primary motion). In the same manner, physics-based animation suits the job of simulating the secondary motion perfectly, given their ability to model physical phenomenon. As such, to produce compelling character animation that includes both motions, artistic animation and physics-based animation need to be joined together in a single pipeline.

Unfortunately, knowing which tool is right for each job does not solve the problem in its entirety. These two paradigms differ significantly in what they want to achieve, meaning that proper care is required to ensure they can coexist in harmony. Adding to that, a life-like secondary motion should come as a reaction to primary motion, implying that a strategy to exchange information between the two needs to be established. These are the main goals that all the following works have attempted to achieve.

2.3.1 Multi-layer / Sequential Deformer

The most intuitive way to combine two vastly different approaches is to run them together, either sequentially or in parallel. This is the exact idea of a multi-layer deformer. Multi-layer deformer, also known as sequential deformer, uses a stack (or chain) of methods to additively add effects to characters. In our context, it concatenates artistic and physics-based animation so that the resulting animation exhibits both primary and secondary motion (Abu Rumman and Fratarcangeli, 2016).

The idea to use a layered model is first introduced two decades ago by Capell et al. (2002), where they introduced a pipeline composed of 2 layers: the *skin* layer and the *control* layer. Mass-spring systems introduced in Section 2.2.2.1 is in charge of the skin layer, while the control layer is controlled by a skeletal deformation (Section 2.1.2). At first glance, this strategy seems to be both performant and logical. However, the fact that these layers work as two completely separate entities pose a major problem: *coherence*. The main reason we want artistic animation and physics-based animation in the same object is to let artist defines the primary motion and allow physics-based animation to generate the appropriate *response* to it in the form of secondary motion. Being two separate entities mean that the motion generated by mass-spring systems will not come as a response of the motion from the control layer, which led to crude and unrealistic secondary effects.

Years later, Li et al. (2016) proposed a sequential deformer that brought some massive improvements to the field. Their work stacked skeletal deformation with continuum-based model. Up until this point, we were limited to full-body simulation, meaning that the entire object has to be involved in the physics simulation. Considering the fact that secondary motion generated by methods up until this point are incoherent, it can be beneficial to only limit such motion to a specific region (e.g. certain body parts of a character). This work imposed boundary conditions

to Newton's differential equation (Equation 2.3) to immobilise certain regions (*free* regions) and prevent them from receiving secondary motion (Figure 2.15). In addition, this work also introduced *blending*, where the motion from physics is blended with the motion from artists the same way in-between frames are blended from two keyframes (Section 2.1.1). Blending helps to reduce the visual impact of incoherent secondary motion, although it does not remove it completely.

FIGURE 2.15: The fixed ($\mathbf{\Omega}''$) and free ($\mathbf{\Omega} \setminus \mathbf{\Omega}''$) region of a sumo character (left) and its impact to secondary motion (right). The top row on the right shows several frames of the input motion. The bottom row on the right shows the result with secondary motion added. Notice how only the fixed region (belly, shoulder, and knees) received a secondary motion, while the rest remains static. Taken from Li et al. (2016) without permission.

The first official attempt at solving the issue of incoherent secondary motion uses a technique called *coupling* (Kim and Pollard, 2011). The idea of coupling is to run multiple physics-based simulation, with one of the simulation running in the same layer as the artistic animation. The goal is to use this simulation to compute forces generated by the artistic motion, which can then be passed as an external force to the other physics simulation in the next layer. With forces computed, the simulation in the next layer can react accordingly, achieving the goal of coherent secondary motion. In a sense, coupling serves as "translational mechanism" that converts the output of the artistic layer to the representation needed by the physics simulation. This technique is perfected in the same year with *two-way* coupling (Liu et al., 2013), allowing the physical simulation in the second layer to similarly provide forces that affect the simulation in the first layer. The main drawback of using coupling to create coherence is performance. Running multiple physics simulation is costly, especially if continuum-based models are used.

Alternatively, Li et al. (2019) used collision detection to create coherence without having to run more than one physics simulation. Every time collision between layers is detected, contact forces are generated based on impulse, and are sent as external forces to the physics simulation.

2.3.2 Kinematic-based Approaches

While it is typical to use full-blown physics-based animation to generate physically realistic secondary motion, not every application of computer animation requires physical realism. Cartoon figures are one great example. To maximise expressiveness, cartoons are usually animated with severely exaggerated motions which do not adhere to standard physical laws, hence the term "cartoon physics" (Thomas et al., 1995). Computer graphics researchers have offered a variety of approaches related to cartoon physics, and most of them fall into the category of *kinematic*-based approaches (Rohmer et al., 2021). Kinematic-based approach uses artistic animation plus kinematic information (position, velocity, acceleration), but does not plug it into Newton's equation of motion.

The groundwork for kinematic-based approaches is a solution to a problem called *shape-matching* (Müller et al., 2005). Shape matching is a geometric problem where, given two collections of vertices $\boldsymbol{\nu}^0 \in \mathbb{R}^{N\times 3}$ and $\boldsymbol{\nu}^1 \in \mathbb{R}^{N\times 3}$, we are tasked to find the rotation matrix $\mathbf{R}$ and two translation matrices $\mathbf{t}^0$ and $\mathbf{t}^1$ which minimises:

$$\sum_i \mathbf{M}_i(\mathbf{R}(\boldsymbol{\nu}_i^0 - \mathbf{t}^0) + \mathbf{t}^1 - \boldsymbol{\nu}_i^1)^2. \tag{2.11}$$

In simple terms, the problem requires us to figure out how to transform one shape to another. The detailed solution to this problem is discussed in the seminal paper by Müller et al. (2005), and is outside the scope of this review. In our context, the shape matching problem can be reinterpreted as finding transformation between two artistically defined poses. In addition to rotation and translation matrix, being able to algorithmically compute the mapping between two poses will let us compute $\mathbf{v}_i$ and $\mathbf{a}_i$, the immediate velocity and acceleration between two poses. This information serves as the base input for all kinematic-based approaches.

From the 12 principles of animation (Thomas et al., 1995), there are two main secondary motions required for cartoon physics, *squash* and *flop*. Squash is an effect where a local elongation happens in the direction of the motion (in this case, the direction of artistic motion). Flop, is an effect where a looser part of an object will move at a slower speed than the rigid parts. This trailing behind effect is sometimes also called drag and is vital to liveliness. In his work titled "Velocity Skinning", Rohmer et al. (2021) proposed a formula for squash and flop using only kinematic information:

$$\Psi_{squash}(\mathbf{v}_i) = (\mathbf{R}\mathbf{S}\mathbf{R}^T)(\mathbf{v}_i - \mathbf{c}_i), \tag{2.12}$$

where $\mathbf{c_i}$ is the centroid of the skeleton, $\mathbf{R}$ is the rotation matrix computed earlier. $\mathbf{S}$ is a simple scaling matrix that depends on the velocity $\mathbf{v}_i$:

$$\mathbf{S} = \begin{bmatrix} 1+s & 0 & 0 \\ 0 & 1/\sqrt{1+s} & 0 \\ 0 & 0 & 1/\sqrt{1+s} \end{bmatrix}.$$

For floppy secondary motion, the formula is much simpler:

$$\Psi_{flop}(\mathbf{v}_i) = -k_{flop}(\mathbf{v}_i), \tag{2.13}$$

with the floppiness constant, k_{flop} reflecting how pronounced this effect must be. These flop and squash secondary motion is then added to the vertex position after we deform each vertex position according to a primary motion, e.g. linear blend skinning to get our complete animation:

$$\boldsymbol{\nu}_i' = \underbrace{\left(\sum_{j=1}^{m} w_{i,j}\mathbf{T}_j\right)\boldsymbol{\nu}_i^0}_{\text{primary motion}} + \underbrace{\Psi_{flop}(\mathbf{v}_i) + \Psi_{squash}(\mathbf{v}_i)}_{\text{secondary motion}}. \tag{2.14}$$

The formula for squash and floppy secondary motion is then extended to also use acceleration information a year later (Kalyanasundaram et al., 2022). Figure 2.16 shows these effects in action. Needless to say, no matter how much information is used in kinematic-based approaches, it is not meant to replace secondary motion that comes from fully-fledged simulation. Instead, they are catered for a specific use of secondary motion and are able to do so in real-time.

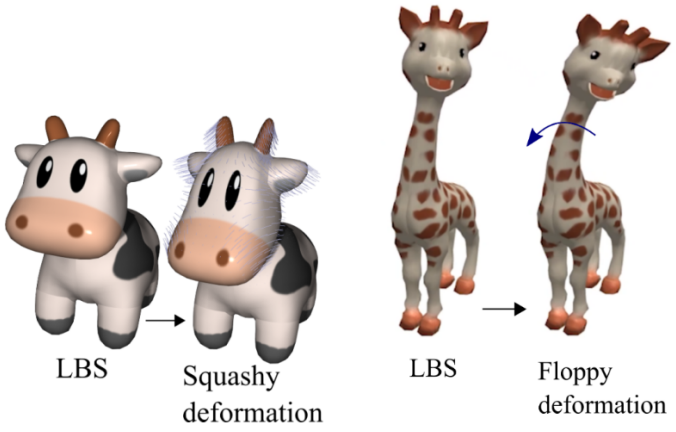


FIGURE 2.16: The squash and flop secondary motions applied to different characters. Adapted from Rohmer et al. (2021).

2.3.3 Rig-space Methods

The use of coupling and collision does provide a degree of coherence into the secondary motion, but on top of the performance problems, they are also proven to be impractical to work with (McLaughlin et al., 2011). To start, the workflow used to create artistic animation and physics-based animation are vastly different. This disconnection makes controlling and tuning animation that contains both paradigms a stressful and demanding task. Coupling adds another physics simulation to the system, which makes this situation worse. Moreover, this additional simulation is used unnaturally, striping away the physical intuitiveness that often helps to tune the system.

Research by Hahn et al. (2012) addressed these shortcomings with a method that *fuses* artistic primary motion and physics-based secondary motion. To this end, they developed a technique called rig-space methods. Rig-space methods uses skeletal-based deformation for their primary animation, and continuum-based model for their secondary animation. The key difference is that the continuum-based simulation is done in the subspace spanned by the skeleton. In simple terms, it tries to represent a physics-based motion as an *instance* of skeletal deformation, unifying their mechanics. To quickly understand how this works, we need to understand the key idea of skeletal deformation. Skeletal deformation can be thought of as a function $\mathbf{S}$, which maps a skeleton transformation $\mathbf{T} \in \mathbb{R}^{3\times3}$ to a vertex position $\boldsymbol{\nu} \in \mathbb{R}^3$; $\mathbf{S} : \mathbb{R}^{3\times3} \rightarrow \mathbb{R}^3$. In physics-based animation, the object position $\mathbf{x}$ is analogous to the vertex position $\boldsymbol{\nu}$. With this understanding, rig-space method treats $\mathbf{S}(\mathbf{T})$ as $\mathbf{x}$, and substitute it into the variational implicit Euler (Equation 2.8). Instead of minimising for $\mathbf{x}$, we are minimising for $\mathbf{T}$:

$$\arg\min_{\mathbf{T}^*} \frac{\Delta t^2}{2} \left(\frac{\mathbf{S}(\mathbf{T}^*) - \mathbf{S}(\mathbf{T}^-)}{\Delta t^2} - \frac{\mathbf{w}^-}{\Delta t} \right)^T \mathbf{M} \left(\frac{\mathbf{S}(\mathbf{T}^*) - \mathbf{S}(\mathbf{T}^-)}{\Delta t^2} - \frac{\mathbf{w}^-}{\Delta t} \right) + E(\mathbf{S}(\mathbf{T}^*)), \tag{2.15}$$

where the $-$ notation indicates the previous timestep, and $\mathbf{w}$ is a value analogous to the velocity of a skeleton transformation. The solution $\mathbf{T}^*$ is a skeleton transformation that *represents* the motion normally generated by physics-based animation. A follow-up work by the same author (Hahn et al., 2013) then approximated $\mathbf{S}(\mathbf{T})$ with its first-order Taylor series approximation, instead of evaluating them in a black-box manner. This is done because some skeletal deformation techniques are non-linear, creating a bottleneck when solving Equation 2.15.

Coherence in rig-space methods came naturally, as physics-based motion and skeletal motion are now entities with the same "type", meaning they can simply be blended together. The failure modes, however, are two sides of the same coin. Because movement from physics and artist lies on the same space, they are *adversarial* against each other, meaning that it is now

possible for our secondary motion to undo the primary motion by creating movement in the opposite direction, as shown in Figure 2.17. Additionally, secondary motion generated via rig-space method is limited to one that can be generated by the skeleton. To get the same level of expressiveness as other methods, it might be needed to augment the mesh with additional skeleton which will never be used otherwise.

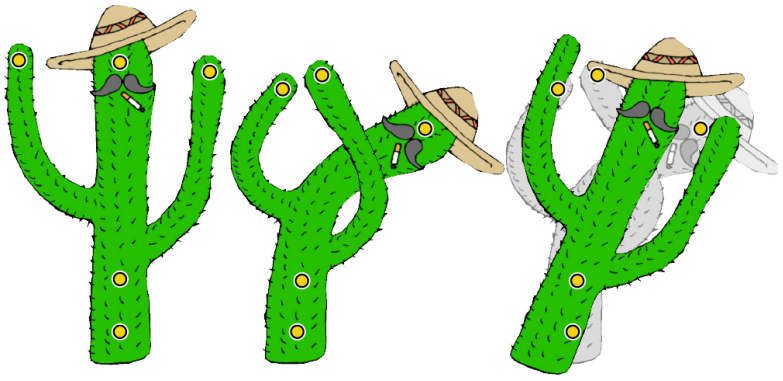


FIGURE 2.17: An instance where rig-space physics may undo artist's intention. The leftmost figure represents the condition where no motions are applied (resting pose). The middle figure shows the artist's intended pose. Finally, the rightmost figure shows what happened if physics is added using rig-space physics without any restriction. Taken from Zhang et al. (2020) without permission.

In an attempt to ameliorate this problem, *complementary dynamics* (Zhang et al., 2020) imposed orthogonality constraints on the rig-space formulation. Orthogonality constraints restrict the formula to only create secondary effects in the orthogonal direction of the primary motion. That is, the physical simulation could not add a motion that could have otherwise been added by the artist. This implements a contract with the artist, ensuring that their creative intentions are not undone by the physical dynamics, while keeping the process relatively plug-and-play. Unfortunately, complementary dynamics are not complimentary. They incur a significant computational cost due to the need to solve a large constrained optimisation problem. In fact, when simulating complex characters, the process can take up to 45 seconds per frame (Zhang et al., 2020), approximately 20 times slower than unconstrained rig-space with the same material model.

In the end, current methods in the literature seem to exhibit a clear tradeoff between performance, realism, and practicality. To give a better general overview of the field, Table 2.2 summarises all augmentation strategies mentioned above according to three aspects: material model used, architecture, and their main drawbacks.

TABLE 2.2: Summary of various animation pipelines.

Methods	Material Model	Architecture	Drawbacks
Capell et al. (2002)	Mass-spring	Sequential	Coherence
Daldegan et al. (1993)	Continuum-based	Sequential	Coherence
Li et al. (2016)	Continuum-based	Sequential with blending	Coherence
Kim and Pollard (2011)	Continuum-based	Sequential with coupling	Performance
Li et al. (2019)	Continuum-based	Sequential with collision	Tunability
Rohmer et al. (2021)	None	Velocity information	Limited applications
Kalyanasundaram et al. (2022)	None	Acceleration information	Limited applications
Hahn et al. (2012)	Continuum-based	Rig-space formula	Performance
Hahn et al. (2013)	Continuum-based	Rig-space formula	Undoing artist's motion
Zhang et al. (2020)	Continuum-based and mass-spring	Constrained Rig-space Formula	Performance

2.4 Research Gap and Problem Statement

Based on our assessment of the literature, current approaches of generating compelling character animation[7] — through the means of combining artistic and physics-based animation, leaves much to be desired. Methods based on layered deformers (Capell et al., 2002, Halstead et al., 1993) are simple and intuitive, but they lack the innate ability to create realistic and coherent secondary motion. Coupling and collision resolution techniques that supposedly bring that coherence also brought performance (Hahn et al., 2012) and tunability (Li et al., 2019) issues. Kinematic-based approaches, on the other hand, are fast and practical, however they are aimed at a specific type of secondary motion and are not universally useful. Approaches based on rig-space physics (Zhang et al., 2020, Hahn et al., 2013) boast great potential because of their

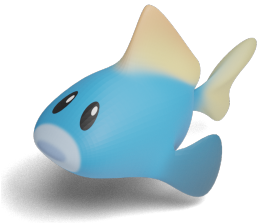
[7] In this context, a compelling character animation is defined as the one that includes *both* primary and secondary motion.

innate coherence, but they entail a significant computational burden that prohibits their use in interactive environments.

A core common problem that seems to be universal to all the methods mentioned above is that generating animation with coherent (physically plausible) secondary motion always seem to come at an expensive cost of performance and practicality — severely hampering their adoption beyond the research domain (McLaughlin et al., 2011). The lack of a method that can produce physically plausible character animation, all while remaining performant and intuitive, is a clear gap within the literature that we aim to address in this thesis.

We notice that this gap stems from the "translational mechanism" — such as coupling and rig space method, causing large performance overhead and practicality issues within the pipeline. To avoid this translational mechanism, the two components of the pipeline (physics and artistic animation) must share the same representation. As all artistic animation algorithms work with positions, finding a physics-based animation algorithm that fit the same mould is vital to solving our problem. Unfortunately, as pointed out by Macklin et al. (2016), the only simulation method that works with position, *position-based dynamics*, is not designed to handle nonrigid material — the type of material that produces secondary motion. As such, before ultimately building a character animation pipeline that is physically plausible, fast, and intuitive in Chapter 4, we first address a smaller gap: developing a physics-based animation algorithm that is position-only, but capable of simulating elastic material such as fat tissues. This will be the main topic of the next chapter.

CHAPTER 3



Extending Position-based Dynamics

The iteration count-dependent stiffness and unphysical handling of elastic motion have prevented position-based dynamics from being a serious competitor in the secondary motion simulation space (Macklin and Müller, 2013). Through this chapter, we aim to change that situation by introducing an extension to the formulation of position-based dynamics (PBD). In particular, we present our PBD extension following the below structure:

1. In Section 3.1, we **establish some fundamental technical definitions and conventions used throughout this chapter**, as well as introducing the input of our simulation.

2. In Section 3.2, we formally **introduce our extended PBD formulation**. To start, we derive the problem from Newton's equation of motion. Then, we detail the three key aspects of our algorithm: The solver (Section 3.2.1), iteration commitment (Section 3.2.2) and constraints (Section 3.2.3).

3. In Section 3.3, we **draw a connection to the original PBD from the theoretical standpoint**. We emphasise the difference in our formulation and how it translates to more physically realistic simulation.

4. In Section 3.4, we **provide an easy-to-read pseudocode of our PBD algorithm**, summarising the changes and corrections that we made to the original PBD. Readers that are primarily interested in implementing our simulation in their software are invited to advance to this section directly.

5. In Section 3.5, we **demonstrate the efficacy of our algorithm for elastic object simulations** (Research Question 1) by performing series of experiment testing the correctness and performance characteristics of our method.

3.1 Notational Conventions and The Input

The idea of position-based dynamics has been briefly touched in Section 2.2.2.3. As with all algorithms based on PBD, our simulation takes two inputs: a set of N particles, and a set of L geometrical constraints. Each particle $\mathbf{p}_i$ in PBD is synonymous to a vertex in a 3D geometry. That being said, particles carry additional information, namely:

M_i	mass
$\mathbf{x}_i$	position
$\mathbf{v}_i$	inertial velocity

The other input, geometrical constraints, are kinematic restrictions in the form of equations and inequalities. These constraints replace the role of internal forces in traditional *force-based* simulation, defining the material being modelled. Mathematically, a constraint is a function that takes information from multiple particles and returns a scalar denoting how "close" the participating particles are from satisfying the constraint. Being a *position-based* simulation, the information used by each constraint is limited to $\mathbf{x}$ (positions). Each constraint is defined according to 4 attributes, namely:

n_j	cardinality
$\mathbf{C}_j : \mathbb{R}^{3n_j} \rightarrow \mathbb{R}$	constraint function
$\{i_1, ..., i_{n_j}\}, i_k \in [1, ...N]$	participating indices
$unilateral$ or $bilateral$	type

The ultimate goal of each simulation step in PBD is to move each particle in a way that satisfy these constraints.[1] With type $bilateral$, a constraint is considered satisfied if $C(\mathbf{x}_i, ..., \mathbf{x}_{n_j}) = 0$. In contrast, a constraint with type $unilateral$ is satisfied if $C(\mathbf{x}_i, ..., \mathbf{x}_{n_j}) \geq 0$. We will see the reasoning of defining constraints this way in Section 3.2. For now, it is sufficient to think of a constraint as a geometric restriction that drives particles into a physically plausible position.

The input particle information can trivially be extracted from any 3D file format. As such, it is valid to interpret the first input as a *3D model of an object*. The second input, the constraints, are typically defined programmatically according to the material being simulated. This is historically a laborious process since the set of constraints used differ from material-to-material. Fortunately, since the introduction of generalised constraints (Macklin et al., 2016, Müller et al., 2007), the same set of constraints can be used for most materials, by evaluating the

[1] Remember, since constraints replace the role of forces in position-based simulation, satisfying a constraint also means being in the physically correct position.

characteristics of said material. We detail what constraints we include later in Section 3.2.3, as it requires full understanding of the solver to introduce.

The last aspect that we need to introduce is the concept of *global* view and *local* view. In physics, certain laws are defined globally, meaning that it affects the system as a whole. Other laws, like the Hooke's law, are defined as an interaction between individual elements (local). When deriving our method, we will often switch between laws that are applied globally and locally. To make this process clear, we will use the notation $\mathbf{x}$ and $\mathbf{C}$ to represent the concatenation of particle positions $[\mathbf{x}_1, ..., \mathbf{x}_N]^T$ and constraints $[\mathbf{C}_1, ..., \mathbf{C}_L]^T$ respectively. These concatenated values will be used when we are referring to the system as a whole (global), while the subscripted notation will be used when we are referring to a *single* quantity (local). For example, when we are defining a constraint function, we want to use the local notation $C(\mathbf{x}_i, ..., \mathbf{x}_{\mathbf{n_j}})$ to accentuate that the required information is taken from each participating particle. When those constraint functions are used in a formula somewhere, we want to use the global notation, $C(\mathbf{x})$, to emphasise that the formula is applicable regardless of the constraint function's definition.

3.2 Core Algorithm

Unlike the original position-based dynamics (Müller et al., 2007) which is derived intuitively, our method will be derived mathematically from the Newton's equation of motion. Following Macklin et al. (2016), we can represent internal forces as negative gradient of potential energy $-\nabla U^T(\mathbf{x})$:

$$\underbrace{-\nabla U^T(\mathbf{x})}_{\text{force}} = \underbrace{\mathbf{M}}_{\text{mass}} \underbrace{\frac{\partial^2 \mathbf{x}(t)}{\partial t^2}}_{\text{acceleration}} . \tag{3.1}$$

The key supporting theory of our simulation is the formulation known as *compliant-constrained dynamics* (Servin et al., 2006) — a relaxation of constrained dynamics (Witkin, 1997). Constrained dynamics work, to the extent they do, because they calculate direct displacement to restore the particle position to satisfy the constraints. However, this is not a good mechanism for elastic objects, since physically correct position cannot be achieved without stiffness.

Force-based simulation does not suffer from this problem, because their notion of "correction" happens through internal (*restoring*) forces, which affect the position of each particle in an indirect manner. Compliant-constrained dynamics are motivated by the mechanics behind restoring forces but does so in the context of satisfying constraints. The result is the formula for *constraint energy*, an energy stored within the system to bring the object back to a constraint-satisfying position. In particular, this constraint energy is defined as:

$$U(\mathbf{x}) = \frac{1}{2}\mathbf{C}(\mathbf{x})^T \boldsymbol{\alpha}^{-1}\mathbf{C}(\mathbf{x}), \tag{3.2}$$

where $\boldsymbol{\alpha}$ represents block diagonal compliance matrix. The entries in this compliance matrix are scalars that represent how "nonconforming" we are to each constraint in $\mathbf{C}$. The bigger the value, the less compliant we are to a constraint. By definition, constraint energy is a potential energy, allowing us to create a connection between constraint functions and Equation 3.1 by substituting the definition of $U(\mathbf{x})$:

$$\underbrace{-\nabla\mathbf{C}(\mathbf{x})^T \boldsymbol{\alpha}^{-1}\mathbf{C}(\mathbf{x})}_{constraint\ \text{force}} = \underbrace{\mathbf{M}}_{\text{mass}}\ \underbrace{\frac{\partial^2\mathbf{x}(t)}{\partial t^2}}_{\text{acceleration}}. \tag{3.3}$$

One excellent intuition to understand this formulation can be formed by looking at the type of quantities on the left-hand side, the "force". $\nabla\mathbf{C}(\mathbf{x})^T$ is a vector, which can be thought as the *direction* of the force. $\boldsymbol{\alpha}^{-1}\mathbf{C}(\mathbf{x})$ is a scalar, which can be thought of as the *magnitude* of the force. The optimal strategy to minimise a convex function is to step in the negative direction of its gradient (Boyd et al., 2004). Similarly, the best course of action to satisfy the given constraints is to step in the negative direction of the constraint gradient (Servin et al., 2006). In our formulation, our force (left-hand side) always points to the negative direction of the constraint gradient, which can be interpreted as a force that *takes the optimal direction to satisfy the given constraint.* The term *constraint force* will be used to refer to this quantity. In simple terms, our simulation is a dynamical system with constraint forces controlling its dynamics. With this construction, we can even see the resemblance to the traditional force-based simulation, where constraint forces are used in place of spring forces (mass-spring systems) or elastic forces (continuum-based models).

With the governing equation set up, we can start building our simulation scheme. Unconditional stability is one of the core characteristics of position-based dynamics (Müller et al., 2005), and our extension will continue this trend. To devise an unconditionally stable simulation, *implicit* Euler integrator will be used (Baraff and Witkin, 1998). Applying implicit Euler to the governing equations yields us:

$$-\nabla\mathbf{C}(\mathbf{x}^{t+1})^T \boldsymbol{\alpha}^{-1}\mathbf{C}(\mathbf{x}^{t+1}) = \mathbf{M}\left(\frac{\mathbf{x}^{t+1} - 2\mathbf{x}^t + \mathbf{x}^{t-1}}{\Delta t^2}\right). \tag{3.4}$$

We then applied the following mathematical manipulations to help simplify Equation 3.4:

1. Δt **folding**. Following Macklin et al. (2016), we define $\tilde{\boldsymbol{\alpha}} = \frac{\boldsymbol{\alpha}}{\Delta t^2}$ to incorporate Δt into $\boldsymbol{\alpha}$. This makes the notation less cluttered and does not come with any real ramifications since Δt is a simple scalar.

2. **Auxiliary variables**. Following Servin et al. (2006), we introduce an auxilliary variable $\boldsymbol{\lambda} = -\tilde{\boldsymbol{\alpha}}^{-1}\mathbf{C}(\mathbf{x})$. This allows us to split Equation 3.4 into two smaller equations, which simplify the solving process.

This yields the final systems of equations that we need to solve every animation frame:

$$\mathbf{M}\left(\mathbf{x}^{t+1} - 2\mathbf{x}^{t} + \mathbf{x}^{t-1}\right) - \nabla\mathbf{C}(\mathbf{x}^{x+1})^{T}\boldsymbol{\lambda}^{t+1} = \mathbf{0} \tag{3.5}$$

$$\mathbf{C}(\mathbf{x}^{t+1}) + \tilde{\boldsymbol{\alpha}}\boldsymbol{\lambda}^{t+1} = \mathbf{0}. \tag{3.6}$$

In short, generating the next frame of animation requires us to compute $\mathbf{x}^{t+1}$ (particle positions for the next timestep) according to Equation 3.5 and Equation 3.6. However, constraint functions included in $\mathbf{C}$ can be arbitrarily complex[2]. As such, there will not be a simple closed-form formula to compute $\mathbf{x}^{t+1}$. Instead, we are required to employ another numerical method to approximate $\mathbf{x}^{t+1}$. This sets up the scene for the first key component of our simulation: the *solver*.

3.2.1 The Solver

Solver is a part of the simulation that computes the next state according to the problem set by our numerical integrator. Because we are only looking at a single timestep, the superscript notation is no longer relevant and therefore will be dropped. It is a good idea to look at the current and the previous state as constant values, and the next state as the unknown. In our context, this implies two things: (1) the current and the previous state will be merged into a single constant $\tilde{\mathbf{y}} = 2\mathbf{x}^{t} - \mathbf{x}^{t-1}$ [3]; (2) our two unknowns ($\mathbf{x}^{t+1}$ and $\boldsymbol{\lambda}^{t+1}$), will simply be called $\mathbf{x}$ and $\boldsymbol{\lambda}$. This is a common convention to avoid clashing in notation, as solvers are typically iterative methods which use another superscript notation to indicate the iteration number (Macklin et al., 2016)[4].

Keeping these notational changes in mind, if we label Equation 3.5 and Equation 3.6 as $\mathbf{g}$ and $\mathbf{h}$ respectively, the goal of our solver is to find two unknowns $\mathbf{x}$ and $\boldsymbol{\lambda}$ that satisfies:

[2] In particular, it can be *non-linear*.

[3] This seemingly arbitrary definition has a vital interpretation. By substituting the definition of $\mathbf{x}^{t}$ and $\mathbf{x}^{t-1}$ to $\tilde{\mathbf{y}}$, we get $\tilde{\mathbf{y}} = 2\mathbf{x}^{t} - \mathbf{x}^{t-1} = \mathbf{x}^{t} + \mathbf{v}^{t}\Delta t$. In other words, $\tilde{\mathbf{y}}$ represents the so-called *inertial position* (Taylor, 2005) — a position obtained by letting an object move according to its current velocity, without any restriction.

[4] Iterative methods typically use $^{(i)}$ for this purpose.

$$\mathbf{g}(\mathbf{x}, \boldsymbol{\lambda}) = \mathbf{0} \tag{3.7}$$

$$\mathbf{h}(\mathbf{x}, \boldsymbol{\lambda}) = \mathbf{0}. \tag{3.8}$$

As hinted earlier, the possibly *nonlinear* nature of $\mathbf{C}$ complicates the solving process. The canonical way to solve a nonlinear system of equations is through Newton-Raphson method (Burden et al., 2015). Newton-Raphson method solves a system of equations by successive approximation of $\mathbf{x}$ and $\boldsymbol{\lambda}$ starting at the initial guess $(\mathbf{x}^{(0)}, \boldsymbol{\lambda}^{(0)})$. Assuming our initial guess is close enough to a solution, the "hard-to-solve" equations $\mathbf{g}$ and $\mathbf{h}$ can be approximated with their first-order taylor expansion around $(\mathbf{x}^{(0)}, \boldsymbol{\lambda}^{(0)})$:

$$g(\mathbf{x}^{(0)}, \boldsymbol{\lambda}^{(0)}) + (\mathbf{x} - \mathbf{x}^{(0)})\frac{\partial}{\partial \mathbf{x}} g(\mathbf{x}^{(0)}, \boldsymbol{\lambda}^{(0)}) + (\boldsymbol{\lambda} - \boldsymbol{\lambda}^{(0)})\frac{\partial}{\partial \boldsymbol{\lambda}} g(\mathbf{x}^{(0)}, \boldsymbol{\lambda}^{(0)}) = 0 \tag{3.9}$$

$$h(\mathbf{x}^{(0)}, \boldsymbol{\lambda}^{(0)}) + (\mathbf{x} - \mathbf{x}^{(0)})\frac{\partial}{\partial \mathbf{x}} h(\mathbf{x}^{(0)}, \boldsymbol{\lambda}^{(0)}) + (\boldsymbol{\lambda} - \boldsymbol{\lambda}^{(0)})\frac{\partial}{\partial \boldsymbol{\lambda}} h(\mathbf{x}^{(0)}, \boldsymbol{\lambda}^{(0)}) = 0. \tag{3.10}$$

If we let $\Delta\mathbf{x} = \mathbf{x} - \mathbf{x}^{(0)}$ and $\Delta\boldsymbol{\lambda} = \boldsymbol{\lambda} - \boldsymbol{\lambda}^{(0)}$, we can express the above equations in a nicely compressed matrix form:

$$\begin{bmatrix} \frac{\partial}{\partial \mathbf{x}} g(\mathbf{x}^{(0)}, \boldsymbol{\lambda}^{(0)}) & \frac{\partial}{\partial \boldsymbol{\lambda}} g(\mathbf{x}^{(0)}, \boldsymbol{\lambda}^{(0)}) \\ \frac{\partial}{\partial \mathbf{x}} h(\mathbf{x}^{(0)}, \boldsymbol{\lambda}^{(0)}) & \frac{\partial}{\partial \boldsymbol{\lambda}} h(\mathbf{x}^{(0)}, \boldsymbol{\lambda}^{(0)}) \end{bmatrix} \begin{bmatrix} \Delta\mathbf{x} \\ \Delta\boldsymbol{\lambda} \end{bmatrix} = - \begin{bmatrix} g(\mathbf{x}^{(0)}, \boldsymbol{\lambda}^{(0)}) \\ h(\mathbf{x}^{(0)}, \boldsymbol{\lambda}^{(0)}). \end{bmatrix} \tag{3.11}$$

The key computation in Newton-Raphson method is solving the matrix equation above for $\Delta\mathbf{x}$ and $\Delta\boldsymbol{\lambda}$. We can then find a better approximation for $\mathbf{x}$ and $\boldsymbol{\lambda}$, $(\mathbf{x}^{(1)}, \boldsymbol{\lambda}^{(1)})$, by adding $\Delta\mathbf{x}$ and $\Delta\boldsymbol{\lambda}$ to our initial guess:

$$\mathbf{x}^{(1)} = \mathbf{x}^{(0)} + \Delta\mathbf{x} \tag{3.12}$$

$$\boldsymbol{\lambda}^{(1)} = \boldsymbol{\lambda}^{(0)} + \Delta\boldsymbol{\lambda}. \tag{3.13}$$

The process of computing the two deltas and adding it to the current guess constitute a single *solver iteration*. Unsurprisingly, we need to do this step multiple times each frame to generate an animation that is accurate enough physically.

Applying the Newton-Raphson method to our problem gives us the following matrix equation:

$$\begin{bmatrix} \mathbf{M} + \boldsymbol{\lambda}^{(i)} \nabla^2 \mathbf{C}(\mathbf{x}^{(i)}) & -\nabla \mathbf{C}^T(\mathbf{x}^{(i)}) \\ \nabla \mathbf{C}(\mathbf{x}^{(i)}) & \tilde{\boldsymbol{\alpha}} \end{bmatrix} \begin{bmatrix} \Delta \mathbf{x} \\ \Delta \boldsymbol{\lambda} \end{bmatrix} = - \begin{bmatrix} \mathbf{M}\left(\mathbf{x}^{(i)} - \tilde{\mathbf{y}}\right) - \nabla \mathbf{C}^T(\mathbf{x}^{(\mathbf{i})}) \boldsymbol{\lambda}^{(i)} \\ \mathbf{C}(\mathbf{x}^{(i)}) + \tilde{\boldsymbol{\alpha}} \boldsymbol{\lambda}^{(i)}, \end{bmatrix} \tag{3.14}$$

where i denotes the iteration number. While it is technically possible to solve this matrix equation directly, several works (Macklin et al., 2016, Liu et al., 2017) have shown that this is not optimal in terms of performance. As one of our research goal (RQ1) requires our method to be as performant as PBD, we decided that a few performance-enhancing approximations are necessary.

In Equation 3.14, the term $\mathbf{M} + \boldsymbol{\lambda}^{(i)} \nabla^2 \mathbf{C}(\mathbf{x}^{(i)})$ requires computation of a *time-varying* constraint hessian, which is costly as we need to recompute this value every time step. To avoid computing the constraint hessian, we adopt the quasi-newton approximation that $\mathbf{M} \approx \mathbf{M} + \boldsymbol{\lambda}^{(i)} \nabla^2 \mathbf{C}(\mathbf{x}^{(i)})$ (Liu et al., 2017). Since the mass-matrix is constant throughout the simulation, precomputation can be performed before the simulation starts, significantly reducing the computational cost of a single iteration.

Next, following Macklin et al. (2016), we assume that $\mathbf{M}\left(\mathbf{x}^{(i)} - \tilde{\mathbf{y}}\right) - \nabla \mathbf{C}^T(\mathbf{x}^{(\mathbf{i})}) \boldsymbol{\lambda}^{(i)} = \mathbf{0}$. This assumption is justified by noting that in the first iteration, this is trivially true if we set our initial guess to $\mathbf{x}^{(0)} = \tilde{\mathbf{y}}$ and $\boldsymbol{\lambda}^{(0)} = \mathbf{0}$. As mentioned previously, $\tilde{\mathbf{y}}$ represents the inertial position of the current particle. Setting our initial guess to $\tilde{\mathbf{y}}$ means that at the start of every frame, our simulation allows the object to move freely according to its inertial velocity, before bringing it to a physically correct position through Newton-Raphson iterations.

Including these two approximations lead to a simpler Newton subproblem:

$$\begin{bmatrix} \mathbf{M} & -\nabla \mathbf{C}^T(\mathbf{x}^{(i)}) \\ \nabla \mathbf{C}(\mathbf{x}^{(i)}) & \tilde{\boldsymbol{\alpha}} \end{bmatrix} \begin{bmatrix} \Delta \mathbf{x} \\ \Delta \boldsymbol{\lambda} \end{bmatrix} = - \begin{bmatrix} 0 \\ \mathbf{C}(\mathbf{x}^{(i)}) + \tilde{\boldsymbol{\alpha}} \boldsymbol{\lambda}^{(i)} \end{bmatrix}. \tag{3.15}$$

In this simpler version, the value of $\Delta \boldsymbol{\lambda}$ can be easily computed by taking the Schur complement with respect to $\mathbf{M}$:

$$\Delta \boldsymbol{\lambda} = \frac{-\mathbf{C}(\mathbf{x}^{(i)}) - \tilde{\boldsymbol{\alpha}} \boldsymbol{\lambda}^{(i)}}{\mathbf{C}(\mathbf{x}^{(i)}) \mathbf{M}^{-1} \nabla \mathbf{C}^T(\mathbf{x}^{(i)}) + \tilde{\boldsymbol{\alpha}}}. \tag{3.16}$$

With that, the change in position (*correction vector*) can be computed through simple substitution:

$$\Delta \mathbf{x} = \mathbf{M}^{-1} \nabla \mathbf{C}^T(\mathbf{x}^{(i)}) \Delta \boldsymbol{\lambda}. \tag{3.17}$$

These formulae are then used to compute the successive approximation $(\mathbf{x}^{(i+1)}, \boldsymbol{\lambda}^{(i+1)})$ in the same manner as Equation 3.12. The intention is that by repeating this iteration over and over, we will eventually arrive at a "good enough" solution. This good enough solution is often referred to as the point of *convergence*. To determine if we have reached this point, we need to understand what it means to reach convergence. In numerical analysis, convergence is defined as a point where our solution is no longer improving (Burden et al., 2015). Bringing this definition to our context, our approximation is no longer improving if the changes from iteration-to-iteration are marginal. We decide if this is the case by checking if $\Delta \mathbf{x}$ is less than a very small value ϵ (usually 10^{-7}). Once we reach convergence, the guess at the point of convergence serves as our next state and is rendered to the screen.

3.2.2 Iteration Commitment

The formula stated in Equation 3.16 involves $\mathbf{C}(\mathbf{x}^{(i)})$. As we mentioned earlier, $\mathbf{C}$ is a concatenated vector composed of all L constraints ($[\mathbf{C}_1, ..., \mathbf{C}_L]^T$) within the system. In the case of $L > 1$ [5], Equation 3.16 actually represents a *system* of equations[6] — one for each constraint. In other words, a single solver iteration consists of L "mini" updates. This section deals with the nuances that arise because of these mini updates.

To perform the j-th mini update in the i-th iteration, in addition to computing the changes in lambda and position ($\Delta\lambda_j$ and $\Delta\mathbf{x}_j$), we need to commit that changes by adding them to the state of the particles in the current iteration ($\mathbf{x}^{(i)} + \Delta\mathbf{x}_j$). However, as pointed out by Macklin et al. (2014), the timing to when the commitment happens can lead to vastly different end results, particularly because the $j+1$-th update *potentially* use the mini-updated $\mathbf{x}^{(i)}$. In the literature, there are 2 possible ways of handling this: *Jacobi* and *Gauss-Seidel*.

The Jacobi strategy applies any update to the position only *at the end* of each iteration. This means every $\Delta\mathbf{x}$ computation uses the state at the beginning of the iteration ($\mathbf{x}^{(i)}$) as an argument. The immediately noticeable benefit of such strategy is its thread-safety, permitting direct parallelisation without lock and mutexes (Fratarcangeli et al., 2016). In addition, it is also symmetric, meaning that the order of which constraints are processed does not affect the result (Macklin et al., 2014).

The Gauss-Seidel strategy, on the other hand, applies $\Delta\mathbf{x}$ *immediately* after its computation. With this strategy, only the computation of the first $\Delta\mathbf{x}$ utilises $\mathbf{x}^{(i)}$ as its argument. Then,

[5] Which is typical and will be the case for any meaningful physical system.
[6] A system of L equations, to be exact.

the subsequent updates use the state that includes all updates leading to them: $\mathbf{x}^{(i)} + \Delta\mathbf{x}_1$, $\mathbf{x}^{(i)} + \Delta\mathbf{x}_1 + \Delta\mathbf{x}_2, \cdots$ and so on. On paper, this seems like a more logical strategy, but there is no evidence a priori that points to which strategy is "better". For this reason, we will implement our simulation using both strategies and rely on the evaluation at the end of this chapter to decide which strategy will be used in our final algorithm.

3.2.3 Constraints to be Solved

Once the formula for our solver is derived, another necessary aspect of PBD simulation is to decide what constraints will be imposed in the system. In our case, our target material will be any material capable of producing secondary motion, so our constraint will be designed around its behaviour. Material that are capable of exerting secondary motion are materials that are elastic but fundamentally solids (Metaxas, 2012). These materials are also known as *deformable solids* or *soft bodies*.

Deformable solids are characterised by their structural rigidity (Allen, 2012), roughly maintaining their shape but are able to deform moderately in response to forces. In the context of 3D geometry, maintaining shape translates to conserving two quantities: the distance between each vertex (edge length), and the volume of each composing element in the geometry. The two constraints we will be using are a direct mathematical translation of these requirements. The deformable qualities can then be achieved by making these constraints soft, such that they allow slight deviation but eventually lead to constraint satisfaction given enough time.

3.2.3.1 Edge Length/Distance Constraint

Consistent with its name, edge length constraint limits the distance between two particle positions $\mathbf{x}_1$ and $\mathbf{x}_2$ to be a specific distance d. This constraint is applied individually to each pair of vertices that is connected with an edge inside our object's mesh. As pointed by Bender et al. (2015), such constraint can be formulated as follows:

$$C(\mathbf{x}_1, \mathbf{x}_2) = \underbrace{\|\mathbf{x}_1 - \mathbf{x}_2\|}_{\text{current edge length}} - \underbrace{d}_{\text{resting distance}} . \tag{3.18}$$

In our edge length constraint, the distance d is chosen to be equal to the rest distance of the edge. This means that the object will eventually maintain the shape of its edge, thus satisfying our structural rigidity requirement.

In addition to the formula for the constraint itself, another important information that we need to provide here is how the solver can move the particle from an arbitrary state to satisfy this

constraint. In other words, we want to derive the constraint gradient with respect to each individual particle $\nabla_{\mathbf{x}_1} C(\mathbf{x}_1, \mathbf{x}_2)$ and $\nabla_{\mathbf{x}_2} C(\mathbf{x}_1, \mathbf{x}_2)$. Using the chain rule, we can compute this analytically by taking the gradient of Equation 3.18:

$$\nabla_{\mathbf{x}_1} C(\mathbf{x}_1, \mathbf{x}_2) = \frac{\mathbf{x}_1 - \mathbf{x}_2}{\|\mathbf{x}_1 - \mathbf{x}_2\|} \tag{3.19}$$

$$\nabla_{\mathbf{x}_2} C(\mathbf{x}_1, \mathbf{x}_2) = -\frac{\mathbf{x}_1 - \mathbf{x}_2}{\|\mathbf{x}_1 - \mathbf{x}_2\|}. \tag{3.20}$$

With the help of illustration in Figure 3.1, we can visually verify that taking a step in the direction pointed by $-\nabla_{\mathbf{x}_1} C$ and $-\nabla_{\mathbf{x}_2} C$ is the optimal way to satisfy the edge length constraint.

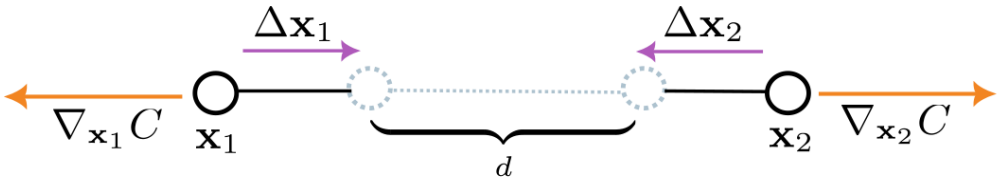


FIGURE 3.1: Visualisation of edge length constraint. Notice how the position update $\Delta\mathbf{x}$, which is in the negative direction of constraint gradient, is in the optimal direction to satisfy the constraint.

3.2.3.2 Volume Constraint

Another important characteristic of elastic objects is how they maintain their volume even after significant amount of forces are applied to them (Sifakis and Barbič, 2015). Importantly, 3D objects in computers are stored as a collection of smaller connected polygons/polyhedrons as opposed to one big geometry. As a result, it is generally not possible to fully preserve the global volume of an object. The best we can do is to preserve the volume of each polyhedron, hoping that doing so will maintain the volume of the entire object to an acceptable degree. This constraint is built under the assumption that the 3D models used in the simulations are made up of tetrahedron, the most commonly used polyhedron in computer graphics.

This constraint involves 4 particle positions $(\mathbf{x}_1, \mathbf{x}_2, \mathbf{x}_3, \mathbf{x}_4)$ at the corners of each tetrahedron. Our constraint function (Equation 3.21) is then defined similarly to the edge length constraint, as a difference between the current volume and the resting/initial volume. Note that once again, the constraint is satisfied (has a value of zero) if and only if the current volume matches the initial volume.

$$C(\mathbf{x}_1, \mathbf{x}_2, \mathbf{x}_3, \mathbf{x}_4) = \underbrace{\frac{1}{6}[(\mathbf{x}_2 - \mathbf{x}_1) \times (\mathbf{x}_3 - \mathbf{x}_1)] \cdot (\mathbf{x}_4 - \mathbf{x}_1)}_{\text{current volume of the tetrahedron}} - \underbrace{V_0}_{\text{resting volume}} . \tag{3.21}$$

The gradient of this constraint function is slightly less trivial to derive and interpret than the edge length constraint. To maximally change the volume of the tetrahedron, we want to move in the direction perpendicular to the triangle created by the other three points. This is equivalent to changing the height of the tetrahedron, exemplified in Figure 3.2.

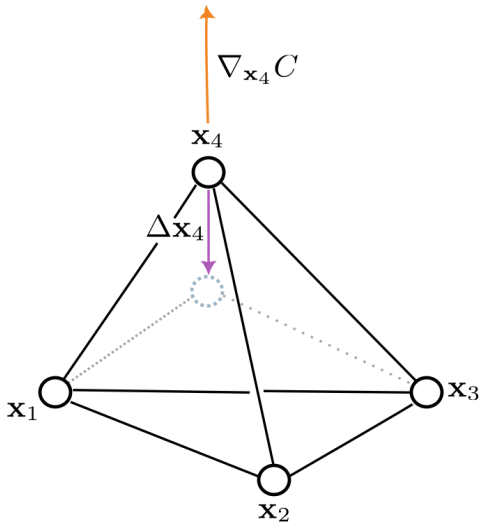


FIGURE 3.2: Visualisation of volume constraint. Notice that the correction vector $\Delta\mathbf{x}$ maximally alters the volume of the tetrahedron by changing its height.

Mathematically, this direction is represented by the cross product of the two vectors that made up the base of the tetrahedron. Applying this idea to all particles, we arrive at the following formula:

$$\nabla_{\mathbf{x}_1} C(\mathbf{x}_1, \mathbf{x}_2, \mathbf{x}_3, \mathbf{x}_4) = \frac{1}{6} \|\mathbf{x}_4 - \mathbf{x}_2\| \times \|\mathbf{x}_3 - \mathbf{x}_2\| \tag{3.22}$$

$$\nabla_{\mathbf{x}_2} C(\mathbf{x}_1, \mathbf{x}_2, \mathbf{x}_3, \mathbf{x}_4) = \frac{1}{6} \|\mathbf{x}_3 - \mathbf{x}_1\| \times \|\mathbf{x}_4 - \mathbf{x}_1\| \tag{3.23}$$

$$\nabla_{\mathbf{x}_3} C(\mathbf{x}_1, \mathbf{x}_2, \mathbf{x}_3, \mathbf{x}_4) = \frac{1}{6} \|\mathbf{x}_4 - \mathbf{x}_1\| \times \|\mathbf{x}_2 - \mathbf{x}_1\| \tag{3.24}$$

$$\nabla_{\mathbf{x}_4} C(\mathbf{x}_1, \mathbf{x}_2, \mathbf{x}_3, \mathbf{x}_4) = \frac{1}{6} \|\mathbf{x}_2 - \mathbf{x}_1\| \times \|\mathbf{x}_3 - \mathbf{x}_1\| \tag{3.25}$$

3.3 Connection to Original Position Based Dynamics

We can draw a connection back to the original PBD (Müller et al., 2007)[7] through an examination of how it is derived. In their work, their correction formula is derived by examining a *single* constraint. Given a single constraint function that takes the current state $C(\mathbf{x})$, their solver aims to find a correction vector $\Delta\mathbf{x}$ such that $C(\mathbf{x} + \Delta x) = 0$ (constraint is satisfied). To deal with the nonlinearity of C, PBD linearises each constraint individually (local). This is in lieu of Newton-Raphson method that we use, which linearises and solves the entire system (global). Nevertheless, in their work, the constraint is approximated by:

$$C(\mathbf{x} + \Delta\mathbf{x}) \approx C(\mathbf{x}) + \nabla C(\mathbf{x}) \cdot \Delta\mathbf{x} = 0. \tag{3.26}$$

We have not considered any physical laws at this point. To incorporate physical plausibility, a single scalar s – the scaling factor, has to be found such that the Δx limits itself to the direction of $\nabla C(\mathbf{x})$[8]:

$$\Delta\mathbf{x} = s\mathbf{M}^{-1}\nabla C(\mathbf{x}). \tag{3.27}$$

Here, we can see a slight resemblance to how our constraint forces work, by moving in the direction of constraint gradient. The value of s can be found by substitution to Equation 3.26, giving us:

$$s = -\frac{C(\mathbf{x})}{C(\mathbf{x})\mathbf{M}^{-1}\nabla C^T(\mathbf{x})}.$$

This step then needs to be repeated multiple times per timestep since the computed correction $\Delta\mathbf{x}$ is not exact[9]. Ignoring differences in notation, we can see how s corresponds to our $\Delta\boldsymbol{\lambda}$ with one minor difference, the absence of the $\tilde{\boldsymbol{\alpha}}$ term. The original PBD can be thought of as a *special case* of our extension, where the compliance matrix $\tilde{\boldsymbol{\alpha}} = \mathbf{0}$. Since the matrix $\tilde{\boldsymbol{\alpha}}$ indicates the inverse compliance level, a value of $\mathbf{0}$ indicates that constraints are enforced strictly. This is consistent to PBD's intended design above; by their definition (Equation 3.26), applying $\Delta\mathbf{x}$ to the state should immediately lead to the constraint being satisfied — hence the term "*hard*" constraint. This is in contrast to our $\Delta\mathbf{x}$, which is designed as a means to compute proper

[7] This work is considered as the de facto "original" PBD. The latest work on PBD like Macklin et al. (2016, 2019), Bender et al. (2017) still maintain the same formula.

[8] Doing so *conserves* the linear and angular momenta. Unfortunately, the reason is not so obvious, we refer the reader to the full text (Müller et al., 2007) to find out why.

[9] This is partly because of the approximation performed in Equation 3.26. Additionally, locally satisfying one constraint can *invalidate* the others.

constraint forces (solving Equation 3.5), leading to gradual fullfilment of the constraint — hence the term "*soft*" constraint.

From a strictly formulaic standpoint, this means that the original PBD should not be able to simulate elastic motion. To work around this limitation, PBD applies an intuitive workaround, scaling the value of $\Delta\mathbf{x}$ with a stiffness factor $k \in [0, 1]$ (Müller et al., 2007). By linearly scaling the positional change, PBD can imitate the gradual enforcement of soft constraints. Figure 3.3 shows the different visuals these three approach produce.

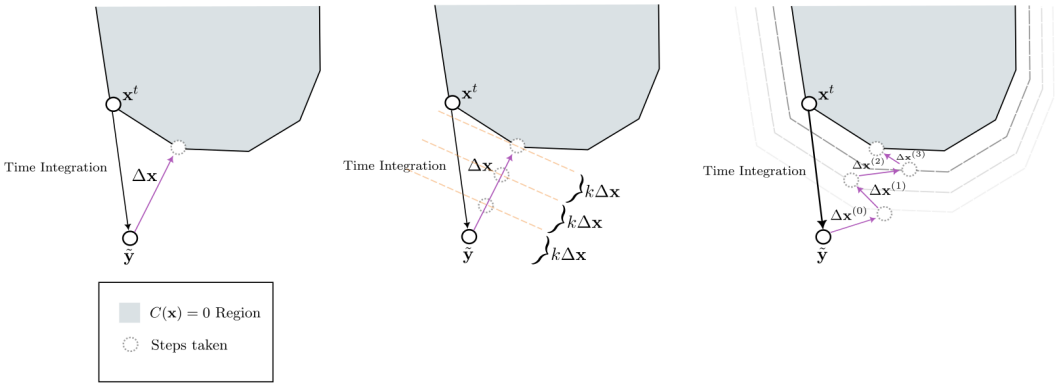


FIGURE 3.3: Visual interpretation of hard constraints (left), PBD's workaround (middle) and true soft constraints (right). $\tilde{\mathbf{y}}$ represents the initial guess of the solver, while $\mathbf{x}^t$ represents the particle position in the current frame. Whereas PBD's workaround is a hard constraint broken down into few smaller steps, soft constraints truly walk towards the constraint-satisfying region gradually.

As one might be able to deduce from the Figure 3.3, this workaround is the main culprit of PBD's infamous iteration-dependent stiffness. When a specific value of k is chosen, we are bound to perform exactly $\frac{1}{k}$ iterations to get the material we designed for. Performing more or less iterations will lead to a material that is too soft or too rigid. This is impossible to do in practice, since there is no way to know exactly how many iterations we will need to get a good enough $\Delta\mathbf{x}$ for every problem. With "true" soft constraints, the computation of $\Delta\mathbf{x}$ takes the softness of the material into account, and each "step" towards the constraint manifold is seen as a problem for the next timestep. Consequently, it does not add additional stiffness the more iteration that we do.

Another issue with this workaround, as pointed out by Macklin et al. (2016), is that it does not cover all types of material. Some materials are inherently nonlinear (Figure 3.4). Because of this, representing their softness with a linear scaling will not be accurate. Even worse, some materials are *anisotropic*, meaning that they react differently in different directions — clearly not capturable by a scalar scaling.

Our soft constraint concept is derived mathematically from compliant-constrained dynamics (Servin et al., 2006), meaning that it has clear theoretical support. In addition, to tune the

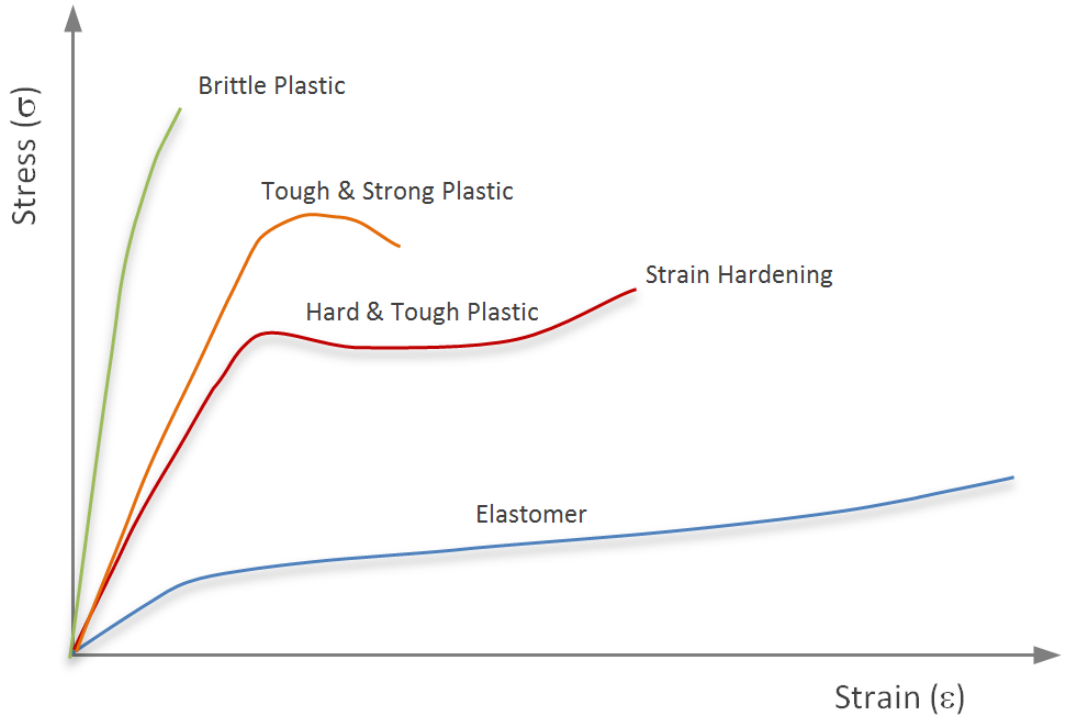


FIGURE 3.4: The stress-strain relationship of many real-life materials. We can see that the stress (force applied per area) does not always have a linear relationship with strain (change in position). The PBD workaround will only work for material that has a roughly linear relationship between them, like the brittle plastic in this graph.

softness of each material, we can adjust the compliance matrix $\tilde{\boldsymbol{\alpha}}$. With this compliance matrix, not only we can represent linear isotropic material (with a scalar-valued diagonal matrix), we can also represent non-linear and anisotropic material by putting values outside the matrix's main diagonal. To conclude, every conceptual differences between the original PBD and our extension can be found in Table 3.1.

TABLE 3.1: Summary of conceptual differences between PBD and our extension.

	PBD (Müller et al., 2007)	**Ours**
Scaling Factor	$\frac{C(\mathbf{x}^{(i)})}{C(\mathbf{x}^{(i)})\mathbf{M}^{-1}\nabla C^T(\mathbf{x}^{(i)})}$	$\frac{C(\mathbf{x}^{(i)})-\tilde{\boldsymbol{\alpha}}\boldsymbol{\lambda}^{(i)}}{C(\mathbf{x}^{(i)})\mathbf{M}^{-1}\nabla C^T(\mathbf{x}^{(i)})+\tilde{\boldsymbol{\alpha}}}$
Solver	Local	Global
Constraint	Hard / soft with workaround	Soft
Stiffness	Iteration-dependent	Iteration-independent

3.4 Putting Everything Together

Due to the lengthy nature of our method derivation, it might be challenging for the reader to see the full picture. To help with this, we dedicate this section to summarise our PBD extension as a fully functioning simulation algorithm. This section also serves as a practical reference for readers that are interested in implementing our extended PBD, but are not particularly interested in how it is derived.

All design decisions and derivations in this chapter lead to the algorithm we call secondary motion position-based dynamics (SMPBD) shown below:

Algorithm 1 Secondary Motion Position-based Dynamics

1: **Input:** $model$, $\mathbf{M}$,$\tilde{\boldsymbol{\alpha}}$ ▷ Takes 3D model of an object, its mass and its material
2: **procedure** SIMULATE($model$, $\mathbf{M}$, $\tilde{\boldsymbol{\alpha}}$)
3: $\mathbf{x^0} \leftarrow$ LOAD3DMODEL($model$) ▷ Input initialisiation (Section 3.1)
4: $t \leftarrow 0$
5: **while** $t \leq$ animationLength **do** ▷ Animation Loop (Section 3.2)
6: compute predicted position $\tilde{\mathbf{y}} \leftarrow 2\mathbf{x}^t - \mathbf{x}^{t-1}$
7: $\mathbf{x}^{(0)} \leftarrow \tilde{\mathbf{y}}$
8: $\lambda^{(0)} \leftarrow \mathbf{0}$
9: **while** $\Delta\mathbf{x} \geq \epsilon$ **do** ▷ Solver loop (Section 3.2.1)
10: **for each** $C_j \in \mathbf{C}$ **do**
11: $C_j(\mathbf{x}^{(i)}) \leftarrow$ EVALUATECONSTRAINT($type, \mathbf{x}^{(i)}$)
12: $\nabla C_j(\mathbf{x}^{(i)}) \leftarrow$ EVALUATECONSTRAINTGRADIENT($type, \mathbf{x}^{(i)}$)
13: compute $\Delta\boldsymbol{\lambda} \leftarrow \frac{-\mathbf{C}_j(\mathbf{x}^{(i)}) - \tilde{\boldsymbol{\alpha}}\boldsymbol{\lambda}^{(i)}}{\mathbf{C}_j(\mathbf{x}^{(i)})\mathbf{M}^{-1}\nabla\mathbf{C}_j^T(\mathbf{x}^{(i)}) + \tilde{\boldsymbol{\alpha}}}$
14: compute $\Delta\mathbf{x} \leftarrow \mathbf{M}^{-1}\nabla\mathbf{C}_j^T(\mathbf{x}^{(i)})\Delta\boldsymbol{\lambda}$
15: COMMITUPDATE(*strategy*)
16: **end for**
17: $i \leftarrow i + 1$
18: **end while**
19: $\mathbf{x}^{t+1} = \mathbf{x}^{(i)}$ ▷ Set the next state to the result given by the solver
20: RENDEROBJECT($\mathbf{x}^{t+1}$)
21: $t \leftarrow t + 1$
22: **end while**
23: **end procedure**

Now, lines in Algorithm 1 perform the following:

- In line 1–4, the inputs to the simulation are initialised. The 3D model of the simulated object is loaded, and the position of its vertices is used as the initial state of the system ($\mathbf{x}^0$). The mass and compliance matrix are given as a parameter according to the material being simulated.

- Line 5 denotes the start of the animation loop, which is repeated until the animation ends. Each animation loop represents a single frame (image) denoted by its time t. In each animation loop, we solve a single instance of Equation 3.5 & Equation 3.6, which requires a solver.

- Line 6–8 shows the solver initialisation. Our solver is based on Newton-Raphson method, which requires an initial guess close enough to the solution. Both of our state variables are initialized according to the values we set in Section 3.2.1.

- Line 9–16 denotes our solver loop. Here, our guess is refined until we reach a good enough solution for the current state. In line 10-11, we are required to evaluate the value of the constraint function and the constraint gradient for the given state. The formula for each constraint is described in detail in Section 3.2.3. After each refinement, the change in our guess are commited via Gauss-Seidel or Jacobi update (Section 3.2.2).

- Line 19–20 happen after our solver converged. The solution to Equation 3.5 & Equation 3.6 are given by the solver and used as the next state. This new state is rendered to the screen, and we advance to the next animation loop.

3.5 Evaluating Secondary Motion PBD

Before integrating it with artist-made animation in the next chapter, it is necessary to ensure that as a standalone physics-based animation algorithm, SMPBD does exhibit a physically correct behaviour. Thus, we run a quick experiment to evaluate the *correctness* of our method and see how it stacks up against previous approaches. To test correctness, we will measure a number of quantitative metrics and compare the measurements to other PBD-based methods. While a user study is a common strategy to judge the "goodness" of an animation (we even adopted it in Chapter 4), we avoid using it here because it has been shown to not be the best strategy when evaluating physical correctness[10].

[10]While humans are good at detecting physically *implausible* motions, they are not particularly good at differentiating which physically plausible motions are the most correct (O'Sullivan et al., 2003).

3.5.1 Setup

Throughout this evaluation, we will refer to the implementation of our method as SMPBD (secondary motion position-based dynamics). We include two variants of our method: SMPBD_{GS} and $\text{SMPBD}_{\text{Jacobi}}$ in our evaluation. SMPBD_{GS} and $\text{SMPBD}_{\text{Jacobi}}$ uses Gauss-Seidel and Jacobi update respectively (as described in Section 3.2.2). These variants are both included to help us decide which one will be included in our animation pipeline in the next chapter. For simplicity, we use the term SMPBD when referring to our method more generally.

For baselines, we choose three different methods that are all based on position-based dynamics, all of which have been reviewed in the previous chapter. The first and most obvious choice, PBD (Müller et al., 2007), is the original formulation of position-based dynamics. The second baseline we choose is $\text{PBD}_{\text{substep}}$ (Macklin et al., 2019) — formulaically equivalent to original PBD but uses substepping instead of solver iterations. For the last baseline, PBD_{BDF2}, otherwise known as *higher-order* PBD (Bender et al., 2015), is chosen. This PBD uses second-order accurate time integration called BDF2, as opposed to the more commonly used implicit Euler. Both $\text{PBD}_{\text{substep}}$ and PBD_{BDF2} are selected to represent two latest attempts of improving the physical correctness of position-based dynamics, albeit with a completely different means to ours.

We implemented all methods mentioned above[11] in C++ with the *Eigen*[12] library. To deal with importing and exporting 3D file formats, we utilised *libIGL*[13], a geometric processing library that is capable of parsing common research 3D file formats (`.obj`, `.mesh`, `.ply`) into Eigen matrices and vice versa. To generate the fully rendered figures in this chapter, we exported the simulated object into one `.obj` file per timestep/frame, and used the stop-motion plugin [14] to load the sequence of `.obj` files that we exported into the 3D tool suite Blender[15]. The Blender's default renderer is then used to render the images.

All timings and performance measurements are measured on a mass-market laptop equipped with AMD Ryzen™ 6800HS @ 4.7 GHz and 16 GB of RAM. The program is executed within Windows Subsystem for Linux 2 under Ubuntu™ 22.04.1 LTS. To ensure consistency and parity for every method, we run every computation on the CPU and allow OpenMP (and therefore Eigen) to use 4 out of 16 available CPU threads via the command `OMP_NUM_THREADS=4`. The Δt (timestep) used in all experiments is the standard 41.6ms, which equates to 24 frames per second. For all methods, the particle's mass is uniformly initialised to 1 kg, unless specified by the scenario. To get the stiffness parameter k[16], which unfortunately does not have real-world

[11] For readers that are interested to play around with SMPBD, we built an interactive web version available here: https://conrev.github.io/PBCD

[12] https://eigen.tuxfamily.org/

[13] https://libigl.github.io

[14] https://github.com/neverhood311/Stop-motion-OBJ

[15] https://blender.org

[16] Used by PBD, $\text{PBD}_{\text{subtep}}$ and PBD_{BDF2}.

physical meaning, we searched its entire range ($[0, 1]$) in a grid-search manner and reported the results with the least amount of error[17]. Another important point is that all measurements are performed at the point of convergence, indicating each method's maximum potential.

3.5.2 Testing Scenarios

Physics-based animation is, at heart, a numerical solution to Newton's differential equation. Therefore, evaluation of correctness can only be performed in instances where either the solution is analytically known, or a widely accepted approximate solution exist. These instances are called *scenarios* (Bargteil and Shinar, 2018). In layman terms, scenarios are special animation tasks where the "ground truth" animation is present. Because scenarios have a known solution, this solution can serve as a reference that represents how the simulated object should move. This allows us to use a quantitative metric to measure the correctness (accuracy) of our method.

In this thesis, we will be running our evaluation against three scenarios (Figure 3.5): *harmonic oscillator* (Müller et al., 2007), *cantilever beam* (Sifakis and Barbič, 2015), and *monster jello* (Liu et al., 2017). Since our extension mainly alters how PBD deals with elastic movements, all three selected scenarios involve an object with elastic material and are chosen based on how often they showed up in the previous work related to ours. Additionally, they represent different levels of complexity to ensure our method performs well at various scale. These scenarios are conceptually 3D objects equipped with a set of constraint specifically designed to match their intended real-life behaviour.

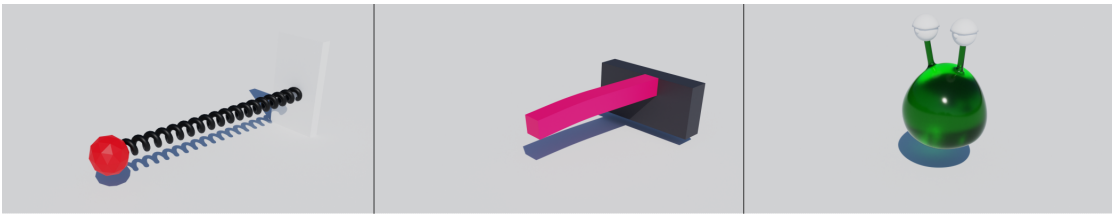


FIGURE 3.5: Visual depiction of our three testing scenarios: *harmonic oscillator* (left), *cantilever beam* (middle), and *monster jello* (right).

Harmonic Oscillator Harmonic oscillator remains the most commonly used sanity test for constraint-based simulation (Müller et al., 2005). In this scenario, a non-massless particle is attached to a wall via a frictionless spring. From the simulation perspective, this scenario comprises 2 particles (vertices) and a single distance constraint. One of the vertex is assigned infinite mass to lock it in place (simulating a wall). In the initial configuration, the spring is mildly stretched beyond its initial distance, resulting in a harmonic (back-and-forth) motion

[17] Average error at convergence, see Section 3.5.3.

with constant amplitude and frequency. The fact that the system consists of only a single mobile vertex permits us to track and compare its position throughout the simulation, which will not be possible with more complex scenarios.

Cantilever Beam Cantilever beam is a rigid structural element that is supported at one end and free at the other. In the initial configuration, an external stress is applied to the free end and then released, resulting in a deflection followed by a wiggle motion. The beam is modelled with 180 vertices (particles), 384 edges and made of 173 tetrahedra. The system comprises 557 constraints total — made of 384 distance constraints (one for each edge) and 173 volume constraints (one for each tetrahedron). This scenario tests the ability to simulate volumetric effects (e.g. volume preservation), which requires at least a few hundred vertices to be visible.

Monster Jello This last testing scenario represents a complex full-body animation. The jello scenario depicts a character made out of jelly-like material dropped into a flat plane, resulting in large deformations in the character body. The character's geometry consists of 3634 vertices, 7204 edges and 7228 tetrahedra. Similar to cantilever beam, one distance constraint is imposed per edge, and one volume constraint is imposed per tetrahedron — totalling to 14432 constraints in the system. The high number of vertices and tetrahedral elements in this character allows it to serve as the "stress" test scenario. Additionally, the material used in this scenario is anisotropic, meaning that it has different "softness" characteristics in different directions. In particular, the material is softer (*less* stiff) in the vertical direction (y axis).

3.5.3 Evaluation Metrics

Evaluating physics-based animation generally boils down to two inseparable things: performance and accuracy. Most physics-based solvers (including ours) are iterative in nature, meaning they work by repeatedly improving their solution until a certain condition is met. They are designed to eventually converge to the theoretical solution given enough iterations. The differentiating factor is often how efficient they are at reaching this point, and how close their convergent solution is to the theoretical solution. A poorly designed method can arrive at the same solution as the state-of-the-art method, but might do so in significantly more time/solver iterations. For this reason, an independent measurement of performance and accuracy is often meaningless.

With this in mind, we can measure the following *basic* metrics to evaluate our method:

- **#Iterations until convergence** (Liu et al., 2017). This metric measures the rate of which a method reach convergence. A method that reaches convergence in the least number of

iterations approach the theoretical solutions the fastest, and therefore can be considered the most performant.

- **Average time cost per iteration** (Baraff and Witkin, 1998). This metric quantifies how expensive it is to perform a single solver iteration. Combined with the number of iterations, this metric offers insight into the total runtime of a method. Perhaps unsurprisingly, a method with high convergence rate typically requires expensive computation on every iteration and vice versa. In a time-constrained applications like physics-based animation, finding a method that can strike a good balance between the two is often the main goal.

- **Average error at convergence** (Liu et al., 2017). This metric measures the accuracy of the method at its maximum potential. It is defined as the absolute difference between the position of each particle in our simulation compared to the ground truth, averaged for every particle:

$$\frac{1}{N}\sum_{i=1}^{n}|\mathbf{x}_i^{(+)} - \mathbf{x}_i^*| \tag{3.28}$$

  where $\mathbf{x}^{(+)}$ is the position at the point of convergence, and $\mathbf{x}^*$ the ground truth position. A high error simply means our simulation deviates heavily from the real-world, implying that it is not physically accurate.

Such basic metrics are often capable of showing the performance differences of our method compared to the previous work. However, there are instances where we want to know when, where and why such differences happen. To perform this more nuanced analysis, we will, in some cases, measure the following *advanced* metric:

1. **Trajectory graph** (Macklin and Müller, 2013). The trajectory graph plots the position of a specific particle for every frame of the animation. This allows us to visually compare the trajectory of each method to the theoretical solution, and decide which phenomena create and propagate error. It is important to note, however, that this is only possible when the scenarios are simple, as those are the only times when we can keep track of every movement in the system.

3.5.4 Evaluation Results

Table 3.2 reports the measurements of three basic evaluation metrics for the harmonic oscillator scenario. First thing we notice is that SMPBD_{GS} and $\text{SMPBD}_{\text{Jacobi}}$ give the exact same result. This aligns with the theoretical difference between them; on a system with a single constraint, the difference in constraint update mechanism will not influence the result. In general, perhaps surprisingly, our two SMPBD variants show lacklustre performance in this scenario, landing in the middle of the pack in both speed and accuracy fronts. In terms of average error, SMPBD ranked second, trailing behind PBD_{BDF2}. They both are not particularly fast either, placing at rock bottom in terms of total frame time (5 iterations $\times$ 0.9 ms/frame = 4.5 ms). Both $\text{PBD}_{\text{substep}}$ and PBD_{BDF2} deliver on their promise of minimising the number of iterations[18], converging after just a single iteration. SMPBD showing similar performance profile to PBD makes sense, as noted in Section 3.3, the actual calculation done every frame is nearly identical. Nevertheless, the accuracy result might seem unintuitive, especially after seeing how theoretically "more correct" our PBD extension is.

TABLE 3.2: Resulting measurements for harmonic oscillator. Columns are **n**umber of **i**terations until **c**onvergence (NI), **c**ost per **i**teration (CI), **a**verage **e**rror at convergence (AE).

	Method	**NI**	**CI** (ms)	**AE** (m)
Baselines	PBD (Müller et al., 2007)	5	0.8	2.682
	$\text{PBD}_{\text{substep}}$ (Macklin et al., 2019)	1	1.6	2.664
	PBD_{BDF2} (Bender et al., 2015)	1	2.3	0.887
Ours	SMPBD_{GS}	5	0.9	2.087
	$\text{SMPBD}_{\text{Jacobi}}$	5	0.9	2.087

With a more careful inspection, however, it is not difficult to see why this is. The setup of harmonic oscillator results in the so-called *ideal* Hookean material. Such material can be precisely modelled with spring forces we introduced in Equation 2.7. Given no external forces and a single particle, substituting the spring forces to Newton's differential equation gave us[19]:

$$kx(t) = M\frac{\partial^2 x(t)}{\partial t^2}.$$

The differential equation above does have an analytical solution, in particular:

[18] Of course, this comes at a slightly more expensive iteration cost.

[19] Note that we drop the bold (vector) notation here. Since the harmonic oscillator moves in a single direction, it is effectively a 1-D object.

$$x(t) = A\,cos(\frac{2\pi}{T}t + \varphi)$$

where A represents the amplitude of the oscillation, φ represents the phase, and T represents the period of the oscillation. The presence of this analytical solution allows us to deduce two things a priori: firstly, the scenario has a *linear* stress-strain relationship [20]; secondly, the task is relatively trivial. These two implications largely answers why our three baselines are able to produce animations on par to SMPBD. Because hard constraints cannot capture the behaviour of nonrigid objects, PBD has a workaround in play. This workaround happens to be able to handle linear material (Section 3.3). The infamous issue of iteration-dependent stiffness will not come into play either, given the miniscule (≤ 5) number of iterations required by all methods — a testament to the triviality of this scenario.

Another factor we need to highlight is why, despite our claim of the problem being trivial, most methods seem to still have reasonably high error. In Figure 3.6, we can see the that the final configuration of all methods, except for PBD_{BDF2}, differ quite a bit from the reference simulation. One feasible explanation is the presence of *numerical damping*, a dissipation of mechanical energy that happens when we use implicit Euler to integrate Newton's equation of motion (Bargteil and Shinar, 2018). However, looking at a static image will not tell the whole story. To investigate, we plot the exact position of the moving particle at every frame to see its trajectory. This gives us the tool to investigate where all the simulation methods went wrong.

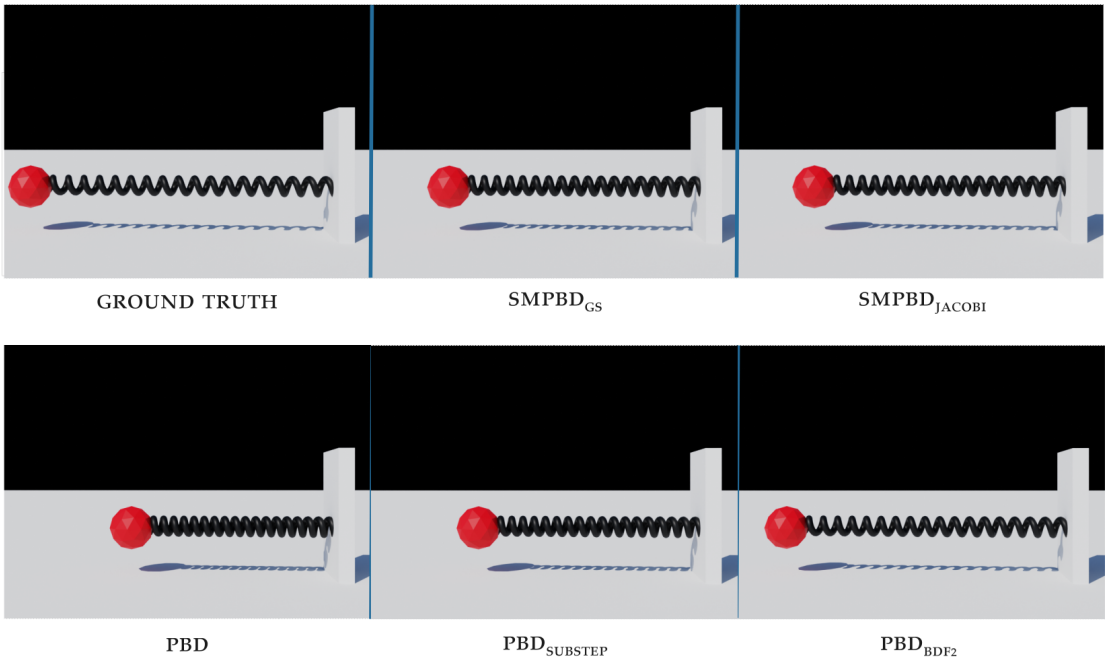


FIGURE 3.6: Final frame of harmonic oscillator for all participating methods. Notice that PBD_{BDF2} looks close to indistinguishable from the ground truth

[20] Hookean material is, by definition, linear.

Looking at Figure 3.7, we can see that initially, the trajectory of the simulated particle matches the ground truth. However, as more simulation steps are taken, the simulation amplitude and period gradually stray away from the analytical solution. In particular, the amplitude seems to gradually decrease as if frictions/dampeners are in effect — thereby confirming our suspicion. Compared to the other four methods, PBD_{BDF2} seems to lose the amplitude the slowest, explaining its similarity to the ground truth in Figure 3.6. This also makes sense mathematically. BDF2, the numerical integrator used in PBD_{BDF2}, has significantly less numerical damping than implicit Euler — the integrator used in the other four methods (Bender et al., 2015).

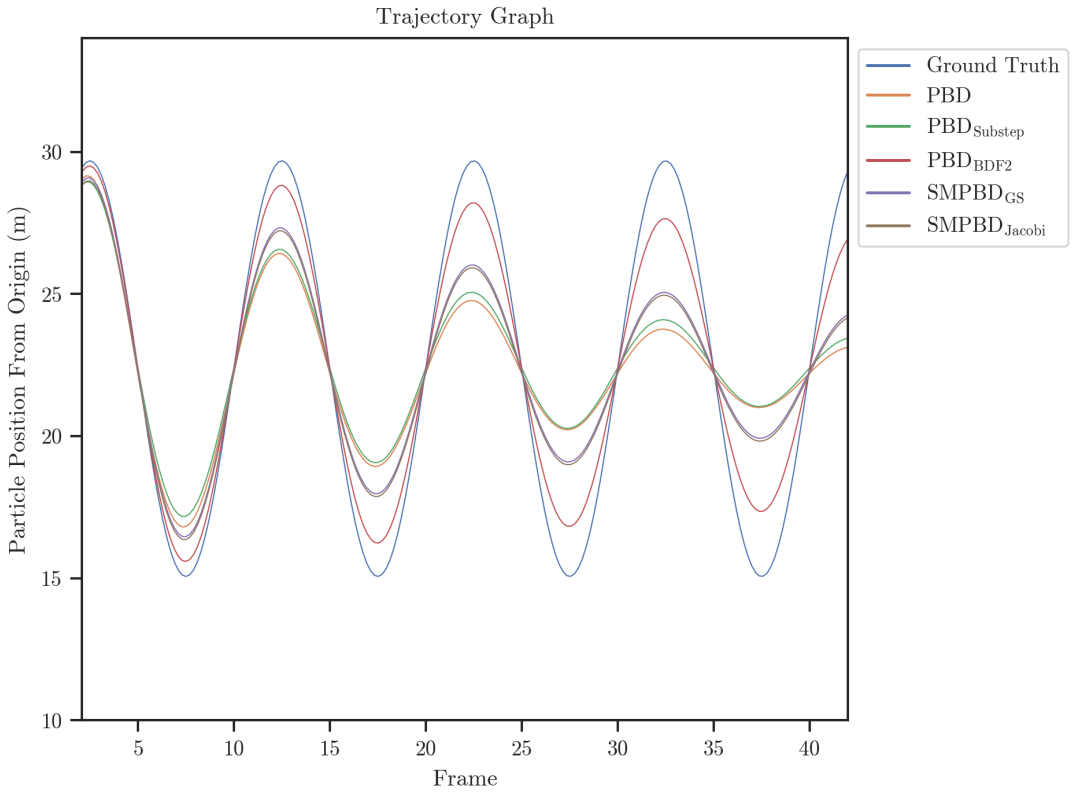


FIGURE 3.7: Trajectory graph of the sole moving particle in harmonic oscillator. Notice the reduction in the oscillation's amplitude for all methods as time passes. PBD_{BDF2}'s oscillation decay happens the slowest, followed by both of our SMPBD variants.

While having non-negligible amount of error in a trivial task does sound concerning, it happens solely because of the lack of friction in the reference simulation of this scenario — a condition which does not happen in the real-world. As such, we do not regard this as particularly concerning.

All in all, this scenario shows the intricacies of building a physics-based animation algorithm. Simply employing the most "correct" formula does not guarantee the best result. To complicate things even further, even in the case of a clear best performer (like PBD_{BDF2} in this scenario), the decision might not be that black and white. Notice that while PBD_{BDF2} converge in a single iteration, that iteration took 2.3 ms to finish. Because it only does a single iteration, if our

frame budget is tighter[21], we cannot dial-down the method by reducing its iteration count. On that case, SMPBD_{GS} or PBD might be the better choice, since they can be tuned-down via the iteration count–accuracy tradeoff.

We now turn to the next scenario, the cantilever beam. Table 3.3 shows the metrics for the cantilever beam scenario. In terms of performance, the same trend seems to apply here. PBD completes every iteration in the least amount of time, followed by our two SMPBD variants, then PBD_{BDF2}. $\text{PBD}_{\text{substep}}$, which by definition only performs a single iteration, lead the overall performance category with 1 iteration $\times$ 20.6 ms/frame = 20.6 ms of total frame time.

TABLE 3.3: Resulting measurements for cantilever beam. Columns are **n**umber of **i**terations until **c**onvergence (NI), **c**ost per **i**teration (CI), **a**verage **e**rror at convergence (AE).

	Method	**NI**	**CI** (ms)	**AE** (m)
Baselines	PBD (Müller et al., 2007)	14	1.2	2.950
	$\text{PBD}_{\text{substep}}$ (Macklin et al., 2019)	1	20.6	0.294
	PBD_{BDF2} (Bender et al., 2015)	5	5.3	2.870
Ours	SMPBD_{GS}	10	2.1	0.142
	$\text{SMPBD}_{\text{Jacobi}}$	15	1.4	0.159

Interestingly, our two SMPBD variant seems to exhibit a tradeoff between number of iterations and cost. There are few experiments from the previous work (Bender et al., 2015, 2017) that have discussed why Gauss-Seidel update strategy naturally converges faster, and it has to do with the bias made by Jacobi iteration. Because the positional correction for the j-th constraint $\Delta\mathbf{x}_j$ is not committed immediately, the positional correction for the successive constraint $\Delta\mathbf{x}_{j+1}$ will use outdated information, as if the previous constraint is never enforced. Since multiple constraints can *share* the same particle, it is very possible that when $\Delta\mathbf{x}_j$ is finally applied, the correction value of $\Delta\mathbf{x}_{j+1}$ is no longer correct. We can see this easier in Figure 3.8 where we illustrate the problem for a much simpler object.

At the same time, this bias should not cause any difference in iteration cost. We hypothesise the difference in iteration cost is caused by lock mechanism in parallelisation. In Gauss-seidel implementation, the $\Delta\mathbf{x}$ that we compute following each constraint has to immediately be added to the global state vector $\mathbf{x}^{(i)}$. Unfortunately, the same global state vector is needed to compute $\Delta\mathbf{x}$. When parallelised, each thread would need to read and write to a shared resource, which requires resource locking. This can lead to threads waiting for each other to continue, inflating the cost. Fortunately for the Jacobi strategy, their mini update is accumulated and written once at the end of each iteration, avoiding this overhead entirely.

[21] say, 2.0 ms which translates to 500 frames per second.

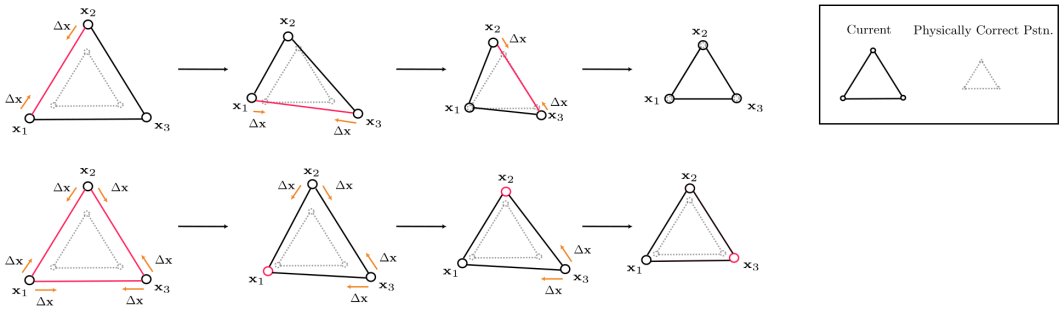


FIGURE 3.8: Visualisation of the "mini" update within an iteration done through Gauss-Seidel update (top) and Jacobi update (bottom). Since Jacobi computes correction using the state at the beginning of the iteration, it does not take the changes within the iteration into account, amounting to a slight bias from the desired result (rightmost image). To get the same result, Jacobi might require a few extra iterations.

In terms of accuracy, we finally see the overwhelming advantage in average error exhibited by our two SMPBD variants compared to the three baselines. SMPBD_{GS} and $\text{SMPBD}_{\text{Jacobi}}$ exhibit 2000% less error than their PBD counterparts. We attribute this to the iteration-dependent stiffness issue to finally come into play; especially since $\text{PBD}_{\text{substep}}$, which by definition runs only one iteration, performs really well with only 0.294 m of average error. To verify this suspicion, we rendered the resulting animation and showed their range of motion in Figure 3.9.

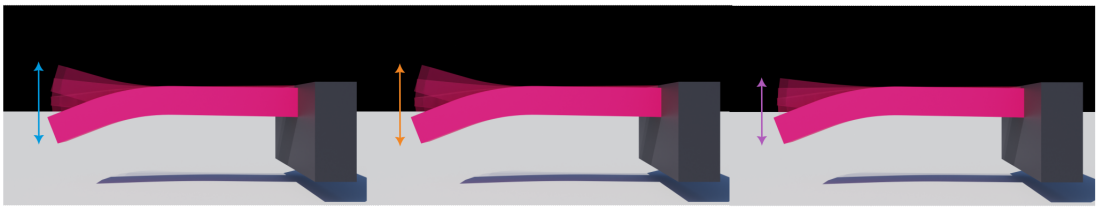


FIGURE 3.9: Range of motion of a cantilever beam simulated with the reference simulation (left), SMPBD_{GS} (middle), and PBD (right). The fully opaque beam represents the initial configuration, while the motion frames are rendered with 90% transparency. The two ends of the arrow represent their range of motion.

As suspected, PBD converges to a very stiff beam, noticeable by how narrow its range of motion is. PBD_{BDF2} — which shares the same formula as PBD, understandably shares the same issue. Its result is almost identical to PBD, which we omit from the above figure to improve readability. On the contrary, both SMPBD_{GS} and $\text{SMPBD}_{\text{Jacobi}}$ generate a result that is exceptionally similar to the ground-truth. We have seen the theoretical reasoning for this in Section 3.3, but this scenario has showcased these benefits experimentally.

To wrap up this section, we move on into the animation task that involve a fully featured character — the monster jello. Table 3.4 displays the metric for this last scenario. The first thing that stands out here is the fact that $\text{SMPBD}_{\text{Jacobi}}$ does not converge (hence the ∞ entry in the related column). This is likely also a culmination of the bias in the Jacobi update. In particular,

this bias is likely more influential to the convergence when the task is harder, which is definitely the case here. Looking strictly at the numbers, the original PBD takes the crown for the fastest overall method, totalling to 35 iterations $\times$ 11.4 ms/iteration = 399 ms to render a single frame. As we have seen numerous times already, SMPBD_{GS} converges in a similar iteration count but with a slightly more expensive iteration cost (11.4 ms/iteration vs 14.1 ms/iteration).

TABLE 3.4: Resulting measurements for monster jello. Columns are **n**umber of **i**terations until **c**onvergence (NI), **c**ost per **i**teration (CI), **a**verage **e**rror at convergence (AE). The ∞ on the NI column means the solver does not converge. If a method does not converge, the average error cannot be measured.

	Method	**NI**	**CI** (ms)	**AE** (m)
Baselines	PBD (Müller et al., 2007)	35	11.4	4.155
	$\text{PBD}_{\text{substep}}$ (Macklin et al., 2019)	1	456.6	2.121
	PBD_{BDF2} (Bender et al., 2015)	31	29.2	1.790
Ours	SMPBD_{GS}	35	14.1	0.342
	$\text{SMPBD}_{\text{Jacobi}}$	∞	11.9	∞

In this stress test scenario, we can see that the SMPBD_{GS} takes another big lead in accuracy compared to the baselines. As great as these results might be, it is understandably hard to interpret. Just how much more "correct" a method with 0.342 m average error is compared to a method with, say, 1.790 m of average error? To help with the interpretation, we provided a few animation frames for the ground truth, the best performer (SMPBD_{GS}), and the second-best perfomer (PBD_{BDF2}) in Figure 3.10.

In the third column (where differences between frames are most observable), the concaving region near the antenna does not curve as deep in PBD_{BDF2}. This indicates incorrect material and is likely a combination of two factors: (1) the inability to handle anisotropic material; (2) the object being too stiff. The jello material used in this scenario is anisotropic, being slightly softer in the vertical direction. Receiving a similar amount of force should lead to more deformations in this direction (shown by the inward curve near the eye area). However, since this is not possible with the PBD formula, it leads to the difference shown in Figure 3.10. Additionally, the problem of the material being too stiff is also observable by how the object does not bend as deep compared to the ground truth and SMPBD_{GS}.

At this point, a question might arise: If performing more iterations triggers the issue with PBD formulation, is it not better to just limit the number of iterations that we do? Unfortunately, that would not help the situation. Iteration count is designed to work as a "knob" that we can tune to adjust how accurate or fast we want our method to be. In a more typical case, performing

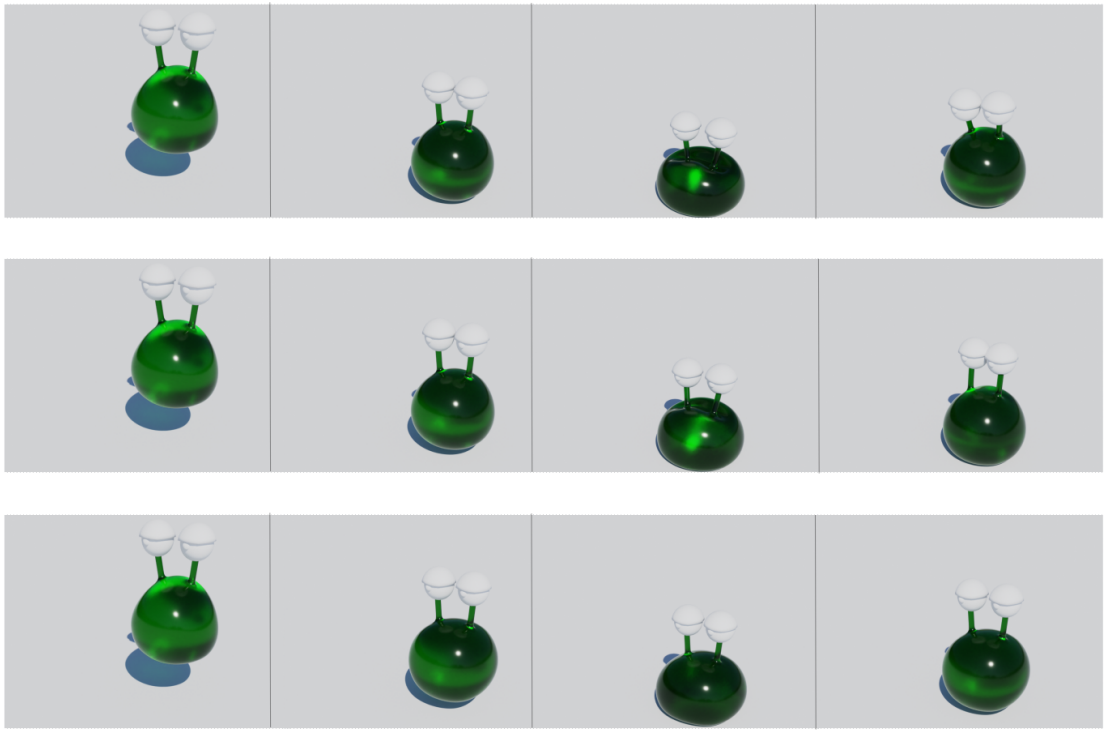


FIGURE 3.10: Four frames of monster jello animation generated by three different methods. The top sequence represents the ground truth, while the middle and bottom sequences represent SMPBD_{GS}, and PBD_{BDF2} respectively.

more iteration improve our solution to Newton's equation of motion, and lead to more accurate dynamics and material depiction. However, in the case of a method with iteration-dependent stiffness, performing more iterations improves the dynamics but might worsen the material depiction — putting us in a lose-lose situation. If we do not perform enough iteration, we are bound to get incorrect dynamics; but if we do, we will get an infinitely stiff material (Figure 3.11). Even when a good middle ground is found, it will be nowhere near as accurate as a system that can be iterated until convergence without affecting its material accuracy.

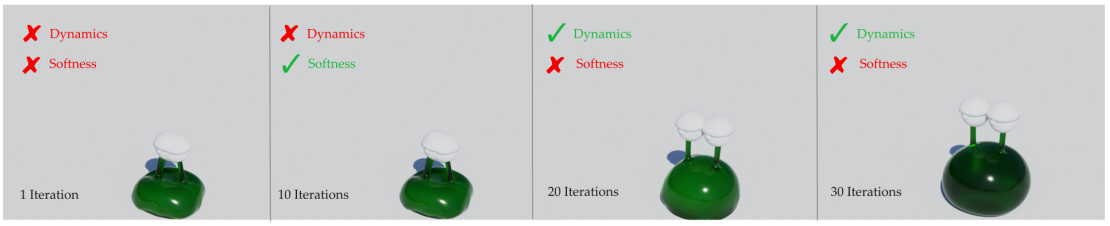
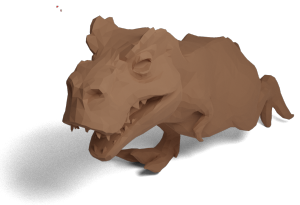


FIGURE 3.11: The visual impact of performing more iterations in an iteration-dependent method. As we can see, the more iterations we do, the more accurate dynamics we get (correct shape), but stiffer the material will be.

To summarise, the benefit of proper soft constraints in PBD shows in how much more accurate SMPBD is at handling objects with soft material. We also noticed that this benefit gets noticeably more pronounced as the complexity of the scenario increases. While doing that, we maintained the performance profile similar to the original PBD, converging at around the same iteration and

only being marginally slower at computing a single iteration. While $\text{PBD}_{\text{substep}}$ and PBD_{BDF2} do deliver on their promises by converging in the smallest amount of iterations, they do not manage to compensate for PBD's improper formulation, giving relatively high error numbers. Their unique performance profiles might, however, warrant them a consideration when the errors are a little closer. In terms of which variants should be chosen for our animation pipeline in the next chapter, SMPBD_{GS} is the obvious choice. Not only that it always converges in fewer iterations compared to its Jacobi counterparts, it also exhibits slightly better average error in all cases. Furthermore, $\text{SMPBD}_{\text{Jacobi}}$ does not converge in the monster jello scenario, which we deem unacceptable as the task will only grow in complexity in the next chapter.

Chapter 4



Position-based Complementary Dynamics

With the introduction of our SMPBD in the last chapter, we showed that elastic material can be accurately simulated through sole manipulation of positions. In this chapter, we leverage its capability to handle secondary motion as a part of a complete character animation pipeline. Within this setting, we proposed position-based complementary dynamics — a two-step sequential deformer combining artistically made primary motion (linear blend skinning) and physically simulated secondary motion (SMPBD) to generate expressive and physically plausible character animations. To put things into perspective, the first section (Section 4.1) will be dedicated to highlighting conceptual differences between the task presented in this chapter and the previous chapter, while simultaneously introducing our input. Following that, the rest of this chapter is organised as follows:

1. Section 4.2 **describes our preprocessing and mesh preparation**. We describe the duality of mesh representation in computer graphics and how to juggle between them.

2. Section 4.3 **bring forth the technical details of position-based complementary dynamics**. We start by giving an overview of the entire pipeline, followed by a detailed description of the artistic layer (Section 4.3.1) and the physics layer (Section 4.3.2).

3. Finally, in Section 4.4, we provide an answer to our third and final research question (RQ3) by **performing a thorough evaluation of position-based complementary dynamics**. This evaluation comprises two large components: performance (Section 4.4.1), and visual quality (Section 4.4.2).

4.1 Notational Conventions and Input

To ease the reader into this chapter, let us discuss what differentiates this chapter from the previous chapter, and how these differences translate to us needing an additional form of input. In the last chapter, we demonstrate how to simulate objects according to physical laws (physics-based animation). In that instance, our input generally includes initial configuration (3D model) and simulation parameters (mass matrix, material parameters). Because the resulting animation is purely a byproduct of a simulation, the only way we influence the result is by changing the initial configuration or the simulation parameters[1]. In this case, our input is a *configurative* input. Instead of merely simulating physics, the main purpose of this chapter is to generate a compelling character animation. This requires us to combine physics simulation with a form of artistic animation. Artistic animation is heavily controlled by an artist, which in addition to configurative input, also requires a *runtime* input. In contrast to configurative input which is taken before the animation starts, runtime input is taken at every animation frame, and represents artist's influence to the resulting animation[2].

With this in mind, we now introduce the configurative inputs used in this chapter. The most important configurative input is, once again, a 3D mesh of a character. Specifically, this input serves as the *initial* state (rest pose) of our character before any animations are applied to it. The rest pose $\boldsymbol{\nu}^0$ is represented by a collection of N vertex positions $\boldsymbol{\nu}^0 = \left[\boldsymbol{\nu}_1^0, \cdots, \boldsymbol{\nu}_N^0\right]^T$. For the purpose of artistic animation, the input mesh is paired with an internal structure called *bones/skeletons*. The term *rigged mesh* is often used to refer to this mesh–skeleton pairing. For most algorithms, it is not necessary to have an explicit representation of the character's skeleton. However, as we will see in Section 4.3.1 and Section 4.3.2, this information will be extensively used to set up our linear blend skinning and SMPBD. As such, we denote the m bones inside $\boldsymbol{\nu}$ by the position of their endpoints $\mathbf{B} = [\mathbf{B}_1, \mathbf{B}_2, \cdots, \mathbf{B}_{2m}]^T$, where the endpoints of the j-th bone are found in the j-th and $j+1$-th entries respectively.

In addition to the mesh itself, we also require some configurative inputs that take the form of parameters. These inputs control the behaviour of algorithms within our animation pipeline, and are primarily used to fine-tune the final animation. Because their meaning and importance will be discussed in-depth when we explain linear blend skinning (Section 4.3.1) and SMPBD (Section 4.3.2), we will not go through them here. However, we include them in the summary of configurative inputs in Table 4.1.

With the configurative inputs out of the way, we now move on to the runtime input. As touched briefly in Section 2.1.2, the most common way artists influence a character's shape is by changing

[1] This is mostly why, in character animations, no pipelines use only physics-based animation, as controlling it to get the exact character pose that we want is extremely challenging

[2] Having said that, this does not mean runtime input has to be made during runtime, it is simply *consumed* during runtime

TABLE 4.1: Summary of our configurative inputs. The last two columns represent which of the two layers (artist and physics) will utilise that input.

		Used By	
Notation	**Description**	**Artist**	**Physics**
$\boldsymbol{\nu}^0$	Set of vertices representing character's rest pose	✓	✓
$\mathbf{B}$	Set of bone endpoints within a character's mesh	✓	✓
w	Level of influence (weight) of a bone to a vertex	✓	
$\mathbf{M}$	Mass matrix		✓
$\tilde{\boldsymbol{\alpha}}$	Compliance matrix describing character's material		✓

the state of the character's skeleton. As runtime input, our pipeline requires a description of this "change". Particularly, in the form of a transformation matrix $\mathbf{T} \in \mathbb{R}^{3\times3}$. Provided that there are m bones within our geometry, we require m transformation matrices $\mathbf{T}_1, \mathbf{T}_2, \cdots, \mathbf{T}_m$ every frame, where $\mathbf{T}_i$ refers to the transformation applied to the i-th bone. Our pipeline (artistic layer, to be specific) will then translate these bone transformations into deformation in the character geometry. Through this input, artists can incorporate their intended character pose into the current frame[3].

4.2 Mesh Preparation: The Duality of Mesh Representation

An observant reader might notice that we used $\boldsymbol{\nu}$ (artist-oriented notation) instead of $\mathbf{x}$ (physics-oriented notation) to represent the input mesh in the preceding section. This is deliberate, as the input mesh commonly fed into an animation pipeline is usually prepared "the artist way". Unfortunately, such mesh differs in many aspects from the one used in physics simulation. In the following two sections, we point out these aspects and introduce the preprocessing needed to make the input mesh *simulation friendly*.

4.2.1 The First Duality: Level of Detail

The first differentiating factor between mesh used for artistic animation and physics simulation is their complexity. Seeing that artistic animation is mostly a creative endeavour, meshes designed for this purpose usually have high vertex and polygon count (*fine* mesh). They are designed to not restrict artistic expressiveness by permitting as much detail as possible. Algorithmic tools that are used to produce such animation are also highly scalable[4]. Physics-based animation algorithms are quite costly in comparison, and are largely unfeasible in very fine meshes. To accommodate both means in a single animation pipeline, the input mesh needs to be *decimated* to reduce its complexity.

[3] Hence the term artistic animation, despite the involvement of an algorithm to produce it.
[4] Partly because artistic animation already involves a lot of manual work.

In order to do that, we will perform *progressive* mesh decimation as a part of our preprocessing routine[5]. In particular, we repeatedly apply the edge collapse operation (Hoppe, 1996) — by selecting two vertices and joining them, effectively reducing the vertex count by one. The selection criteria follow a specific cost function, designed to minimise topological change caused by the operation[6]. For any two vertices $\boldsymbol{\nu}_1$ and $\boldsymbol{\nu}_2$ connected with an edge E, the topological cost to remove it is defined as:

$$\text{cost}(\boldsymbol{\nu}_1, \boldsymbol{\nu}_2) = s_1 cos^{-1}(\mathbf{m_1} \cdot \mathbf{m_2}) + s_2 |\boldsymbol{\nu}_1 - \boldsymbol{\nu}_2|. \tag{4.1}$$

where $\mathbf{n}_1$, $\mathbf{n}_2$ are the normal unit vectors of two polygons that share E as its edges, while k_1 and k_2 are user defined constants that can be utilised to tune the decimation. The removal cost is computed for every adhering vertex pairs, and the least cost pair is removed. A new vertex — $\boldsymbol{\nu}_*$, is then placed in the midpoint of the two removed vertices, as exemplified in Figure 4.1.

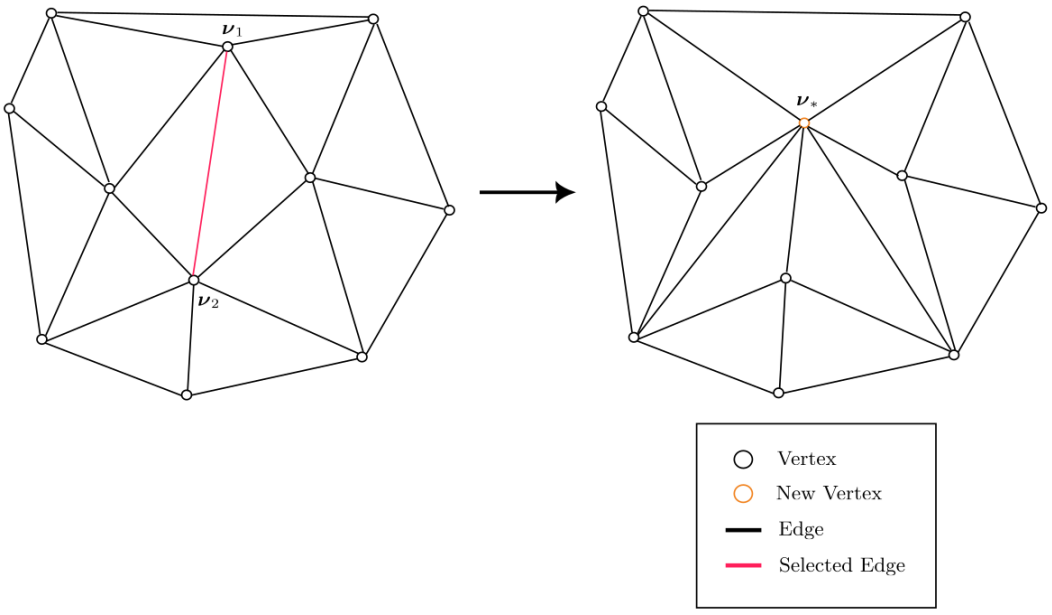


FIGURE 4.1: Visual depiction of a single edge collapse operation. Notice that a new vertex $\boldsymbol{\nu}_*$ is placed in between the removed vertices, and is connected to all neighbours of the old vertex.

By doing such an operation progressively, we can control the complexity of the decimated mesh as desired. It allows us to explore Pareto optimality, finding a good balance between artistic expressiveness and simulation performance. Additionally, a progressive approach to mesh decimation gives us practically unlimited access to mesh with varying complexity; saving us

[5]We note that mesh simplification/decimation is a well-researched problem and warrant an in depth research on its own (Algorri and Schmitt, 1996, Cignoni et al., 1998). The method described here is quite simplistic and by no means the best solution.

[6]In Layman terms, we aim to simplify the mesh while trying to preserve its general "shape".

the time of having to scour for test beds with specific complexity. When applied to a character mesh, the visual difference observable after rounds of decimation can be seen in Figure 4.2.

FIGURE 4.2: The impact of various levels of mesh decimation applied to our testing character: SPOT.

4.2.2 The Second Duality: Composing Elements

Real world objects, including character geometries, are made of infinitely many particles and atomic molecules. From the lens of material science, these particles bonded with each other, creating no physical gap between them. The concept of infinitely many "things" does not fit well within the context of computers. To represent that same object within computers, we discretise them using a finite number of smaller elements. The relevant issue here, is that there are two ways to discretise a continuous object; we can discretise the *exterior* of the object, leaving it hollow in the inside. Or, we can discretise both the *interior* and *exterior* of the object, representing its shape as well as its volume. By discretising only the exterior, we get a *surface* mesh, composed of planar elements such as triangle and quadrilateral polygon. In contrast, discretising both interior and exterior gave us *volumetric mesh*, constructed of polyhedron elements such as tetrahedrons.

Artistic animation, including tools pertaining to it, deals with surface mesh. The reason is quite pragmatic: the interior points are simply never visible, effectively wasting CPU resources. However, physics-based animation almost strictly uses volumetric mesh. Otherwise, simulation of volumetric effects such as volume preservation and jiggling will not be possible. In simple terms, physics-based animation *requires* volumetric/tetrahedral mesh, while artistic animation *prefers* surface/triangular mesh. Because the modelling and rigging process is exclusively done in artist-oriented 3D software, it uses surface mesh as the default representation. For our pipeline, we need to transform this into a volumetric mesh through a process called *tetrahedralization* (Figure 4.3).

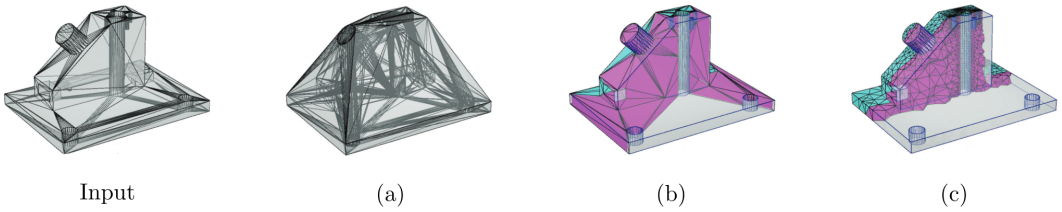


FIGURE 4.3: The input used in TetGen (leftmost image) and its three processing steps: (a) point insertion; (b) constrained tetrahedralization; (c) Delaunay refinement. Adapted from Hang (2015).

To perform tetrahedralization, we make use of the widely adopted TetGen library[7], which is included in our library of choice, libIGL[8]. Tetrahedralization in TetGen is composed of three large processing steps[9], namely:

1. **Point insertion**. In this step, the "rough" version of our tetrahedral mesh is generated. We begin by taking the four outermost points from our surface mesh and connecting them; forming a large tetrahedron that covers every point in the surface mesh. The result is (a) in Figure 4.3.

2. **Constrained tetrahedralization**. In this step, we remove every tetrahedron that is non-conforming to the original input mesh. One way to do this by removing tetrahedra whose barycenter is located outside the input mesh. The result is (b) in Figure 4.3.

3. **Delaunay refinement**. In this step, all tetrahedra that violate the *Delaunay condition* are removed. Delaunay condition states that a circumsphere of a tetrahedron should only contain its own four points. This gives us a quality tetrahedral mesh seen in Figure 4.3 (c).

In short, tetrahedralization will generate a tetrahedral (volumetric) mesh from the given triangular (surface) mesh. When applied to our character mesh, the before and after comparison can be seen in Figure 4.4.

4.3 Method Overview

Our method, position-based complementary dynamics (PBCD), generates compelling character animation by first applying artist desired poses to the character geometry, followed by adding

[7] https://wias-berlin.de/software/tetgen
[8] https://github.com/libigl/libigl/blob/main/include/igl/copyleft/tetgen/tetrahedralize.h
[9] TetGen on its own is a large piece of software composed of many geometry processing algorithms. It is beyond the scope of this thesis to cover it in detail. Instead, we refer readers particularly interested in this step to Hang (2015).

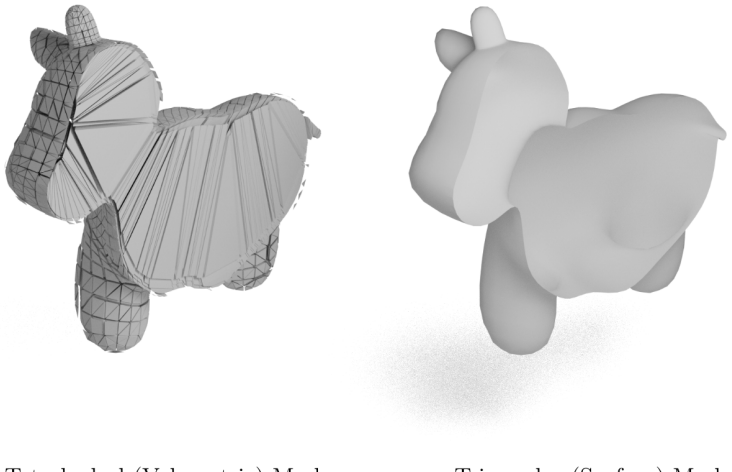


FIGURE 4.4: The tetrahedral mesh (left) generated by passing triangular mesh (right) to TetGen. To highlight the difference, we added small offsets to the tetrahedral mesh and slice both meshes in half. Notice how the triangular mesh is hollow, while the tetrahedral mesh has elements on the inside.

physically plausible secondary motion via physics simulation. Conceptually, our pipeline is remarkably simple. We run two highly robust methods: *linear blend skinning* and *secondary motion position-based dynamics* (SMPBD) sequentially one after another. The extremely simple nature of our pipeline is made possible because of our work in the previous chapter, where we proposed SMPBD as a position-only simulation method that can produce effects such as skin jiggling, drag and swaying — the three most commonly occurring secondary motions (Willett et al., 2017). Being position only means that linear blend skinning and SMPBD can provide inputs to one another without any translation mechanism in between. As a result, coherent character animation can be created without costly mechanisms (e.g. coupling (Kim and Pollard, 2011)) or complex architectures (e.g. rig-space method (Hahn et al., 2012)).

The performance benefit owing to this simple sequential architecture is accompanied by ease-of-use and intuitive fine-tuning process. Whereas tuning a translation mechanism such as coupling can be confusing (Hahn et al., 2012), tuning a simple two-layered deformer is trivial. For example, if the resulting pose does not align with what the artist had in mind, skinning weights in the artistic layer can be adjusted. If, instead, the secondary motion is not as expected, the compliance or mass matrix within the simulation layer can be changed. In PBCD, the relationship between the two layers are much more transparent. Consequently, the impact each layer has to the resulting animation is much more predictable. All things considered, PBCD serves as a character animation pipeline that better aligns with the three core requirements of computer graphics: (1) speed; (2) visual flair; (3) directability (Erleben et al., 2005). With the introduction of this pipeline, we provide an answer to our research question 2 (RQ2).

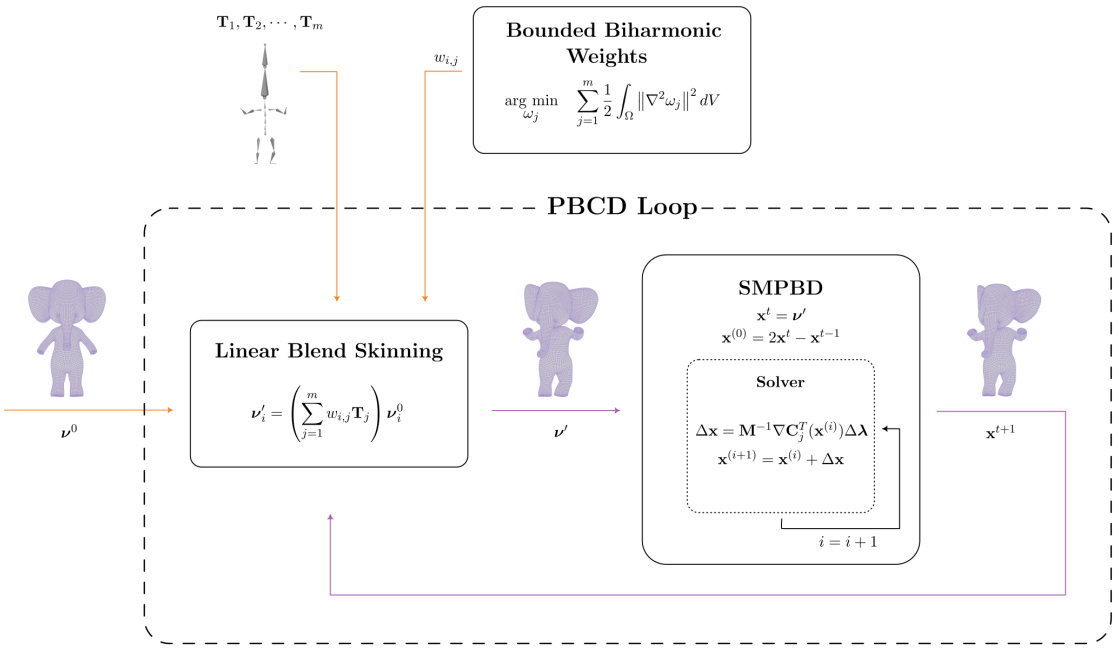


FIGURE 4.5: Full schema of our animation pipeline.

Position-based complementary dynamics follows the simplest form of sequential deformer we discussed in Section 2.3.1. In the first layer (Section 4.3.1), linear blend skinning algorithm is used to deform the character geometry according to bone transformations provided by the artist. We take the reference pose of our character $\boldsymbol{\nu}^0$, and change its shape to output $\boldsymbol{\nu}'$ — the *deformed* character geometry that exhibits the artist's desired pose. This new pose reflects *an action the character is meant to do in the current frame, also known as the primary motion.*

Our second layer (Section 4.3.2) follows this up by setting its initial particle positions to $\boldsymbol{\nu}'$, essentially interpreting $\boldsymbol{\nu}'$ as $\mathbf{x}^0$. Then, the character undergoes a full body physics simulation, generating physically believable reactionary motion according to the initial positions given. Because this process is gradual visually, it produces *a wiggle/drag effect that we often call secondary motion.* The results from this layer is afterwards rendered, producing the desired output: an animated character that shows both primary and secondary motion. Despite our simple structure, we acknowledge that visualising each process can be challenging. To assist understanding, we schematised our entire pipeline in Figure 4.5.

In the following two sections, we provide a comprehensive explanation regarding computations that happen within each layer, including additional adjustments that we made to each algorithm. To begin every section, we also provided justification to why each method is selected to handle that layer.

4.3.1 Handling Primary Motion: Linear Blend Skinning

In this first layer, our task is to translate bone/skeleton transformations into changes in vertex positions. To tackle this task, we went with a skeletal deformation algorithm called linear blend skinning (LBS) (Kalra et al., 1998). The rationale behind this decision can be summed up into two reasons: (1) LBS is lightweight, fast, and intuitive; (2) LBS primary weakness, the *candy-wrapper* effect, is naturally remedied by our physics simulation. The main expected benefit of having a unified-representation pipeline like ours is performance. Linear blend skinning fits right into this idea by being one of the fastest methods available out there (Table 2.1). Even better, the reasoning why LBS is sometimes avoided is because of candy-wrapper effect (Jacobson et al., 2014). Fortunately, the output of LBS in our pipeline is not immediately rendered but instead is fed to SMPBD for simulation. The distance and volume constraints within SMPBD will remove the candy-wrapper effect during the simulation stage — giving us the benefits of LBS without its drawbacks.

Linear blend skinning works by attaching each vertex to one or more skeletal bones; each attachment is then given a strength/influence value called weights (w). When artists specify their desired bone transformations as transformation matrices $\mathbf{T}_1, \mathbf{T}_2, \cdots, \mathbf{T}_m$, we take the weighted average of those transformations and apply it to the position of each vertex. For consistency, the transformation matrix $\mathbf{T}$ is computed with respect to the initial state $\boldsymbol{\nu}^0$. As such, we require the reference shape (rest pose) as an input to this step. Now, for the i-th vertex within a character mesh, LBS can be summarised with just one equation:

$$\boldsymbol{\nu}_i' = \left(\sum_{j=1}^{m} w_{i,j} \mathbf{T}_j \right) \boldsymbol{\nu}_i^0, \tag{4.2}$$

where $w_{i,j}$ represents the influence level of the j-th bone to the current vertex ($\boldsymbol{\nu}_i$). A good set of weights is vital to ensure that the artist's intended pose is faithfully represented without any noticeable artefacts (Jacka et al., 2007). These weights are normally "painted" by the artist during the rigging process. Unfortunately for us, our preprocessing alters the character geometry, invalidating those painted weights. The replacement weights can be computed algorithmically; for that, we first need to define what constitutes good weights. According to Fedkiw (2018), a good set of weights should satisfy the following criteria:

1. (Req1) **Locality and sparsity:** each skeleton should mainly affect vertices around its vicinity. Similarly, for any vertex in $\boldsymbol{\nu}$, the nonzero weights should be localised to its nearby bones. This ensures intuitive interaction between bone transformation and shape deformation. For example, if the skeleton around the elbow is transformed, it should not affect the shape of the legs.

2. (Req2) **Smooth:** the influence of a bone should diminish smoothly as we get further away from it. This prevents unsightly discontinuity in the character's skin.

3. (Req3) **Partition of unity:** i.e. the sum of all weights for a single vertex $\boldsymbol{\nu}_i \in \boldsymbol{\nu}$ should be one ($\sum_j w_{i,j} = 1$). This means that if all skeletons are transformed by the same transformation $\mathbf{T}$, the entire object should be transformed by $\mathbf{T}$. Additionally, it prevents undesired extrapolation and scaling errors.

One algorithm we deem to possess all the desired properties above is the *bounded biharmonic weights* (BBW) by Jacobson et al. (2011). In their approach, the j-th bone is assigned its own weight function that maps a vertex position into its corresponding weights; $\omega_j : \mathbb{R}^3 \to \mathbb{R}$. For a given vertex $\boldsymbol{\nu}_i \in \boldsymbol{\nu}$, we can compute the influence of the j-th bone to it by using this function $w_{i,j} = \omega_j(\boldsymbol{\nu}_i)$. The weight function ω_j can be computed by minimising a higher order smooth functional commonly known as Laplacian energy. Let Ω be the volumetric domain enclosed by our character's mesh, then, ω_j is defined as the solution to the following problem:

$$\underset{\omega_j,\ j=1,2,\cdots,m}{\arg\min} \quad \sum_{j=1}^{m} \frac{1}{2} \int_{\Omega} \left\| \nabla^2 \omega_j \right\|^2 dV \tag{4.3a}$$

$$\text{s.t.} \qquad \omega_j|_{H_k} = \delta_{jk} \ , \tag{4.3b}$$

$$\sum_{j=1}^{m} \omega_j(\boldsymbol{\nu}_i) = 1, \quad \boldsymbol{\nu}_i \in \boldsymbol{\nu}, \tag{4.3c}$$

$$0 \leq \omega_j(\boldsymbol{\nu}_i) \leq 1, \quad j = 1, 2, \cdots, m, \tag{4.3d}$$

where H_j denotes all the points in the line segment of the j-th bone's endpoints, and δ_{jk} is the Kronecker's delta. This is a quadratic programming (QP) problem that can be solved with any QP solver. In our case, we use the implementation built in into libIGL[10], which solves this by calling MOSEK[11]. While the proof might not be obvious, ω_j does satisfy the aforementioned requirements for good weights. The smooth property (Req2) is satisfied by the fact that the result of Equation 4.3a is a shape-aware harmonic function, which is consequently smooth (Axler et al., 2013). Additionally, the requirement for partition of unity (Req3) is clearly enforced in Equation 4.3c. The locality and sparsity requirement is the one with the weakest proof; nonetheless, the authors have proved this experimentally as a part of their work (Jacobson et al., 2011). We can see the resulting weights from this method visualised in Figure 4.6.

With all the ingredients prepared, we can proceed with the computation following Equation 4.2, and pass the output ($\boldsymbol{\nu}'$) to the next layer.

[10] https://libigl.github.io/tutorial/#bounded-biharmonic-weights
[11] https://www.mosek.com/

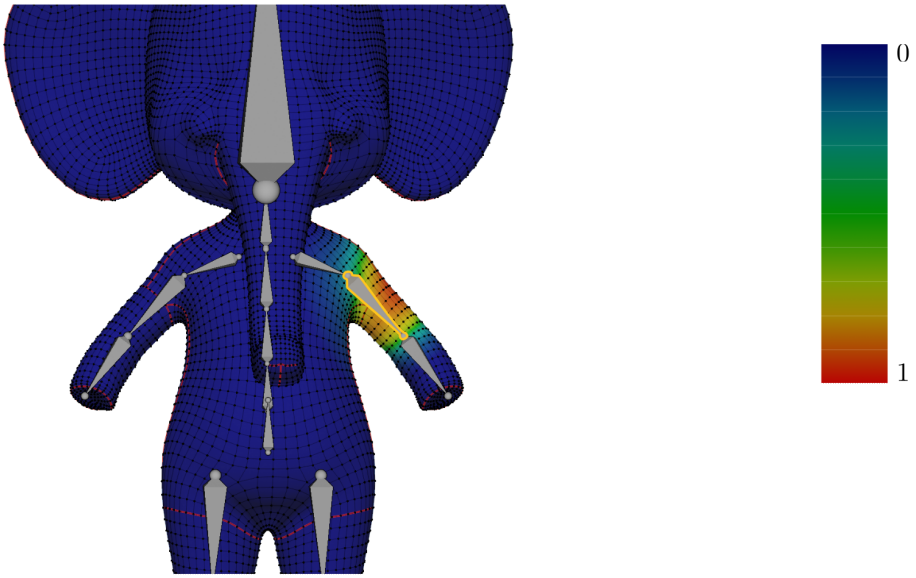


FIGURE 4.6: Colormap visualisation of the weight function of the glowing bone. As we can see, only the area around the right forearm have nonzero weights. Additionally, the weight values smoothly diminish as we get closer to other bones.

4.3.2 Handling Secondary Motion: SMPBD

To create realistic jiggles that add personality to a character, we perform a full deformable body simulation following linear blend skinning. Unsurprisingly, the method of choice in this layer will be secondary motion position-based dynamics (SMPBD) — or more specifically, the SMPBD_{GS} variant. For a method to take this spot, it has to satisfy two requirements: (1) able to simulate soft and elastic body in a physically plausible way; (2) strictly works with positions. To our knowledge, secondary motion position-based dynamics is the only simulation algorithm that satisfies these criteria.

The core idea of SMPBD has been discussed throughout Chapter 3. As one might be able to guess, SMPBD fits right into this layer with minimal adjustments[12]. For the first frame, SMPBD simulation requires the initial position for every particle ($\mathbf{x}^0$). Exploiting the position-only nature of both PBCD layers, we straightforwardly take the output of the previous layer and use it as our initial position — by setting $\mathbf{x}^0 = \boldsymbol{\nu}'$. The same assignment also applies for each of the following frame. Coupled with our two configurative input ($\mathbf{M}$ and $\tilde{\boldsymbol{\alpha}}$) which are provided to us as parameters, we have all the necessary ingredients to begin the simulation. Having said that, there are still a couple of aspects differentiating SMPBD in PBCD to the standalone SMPBD — with the most important thing being one extra constraint called the *bind* constraint.

Previously, we mentioned that SMPBD utilise two types of constraints: the edge length constraint (Section 3.2.3.1), and the volume constraint (Section 3.2.3.2). The two aforementioned

[12]Understandably so. After all, SMPBD is carefully designed to fit here.

constraints maintain the structural rigidity of our object with respect to its "resting" (reference) shape. In layman terms, they ensure that the character's shape is maintained throughout the simulation. These two constraints establish a fully working standalone soft body simulation, but missing a key piece for interoperability with artistic animation.

When our simulation is run on a rigged mesh, the shape of our object is not only defined by its resting shape but also by the current state of the skeleton. As such, in addition to being structurally rigid with respect to its resting shape, the character mesh is also required to be structurally rigid with respect to its skeleton. Following Abu Rumman and Fratarcangeli (2015), we achieve this by preserving the distance between a particle and its projection to the nearest skeleton. We refer to this third and final constraint as the *bind* constraint.

To set this up, given the position of the two ends of the j-th skeleton $\mathbf{B}_j$ and $\mathbf{B}_{j+1}$, the projection of our participating particle ($\mathbf{x}_1$) to the j-th skeleton can be computed with:

$$\text{proj}_{\mathbf{x}_1} j = \frac{(\mathbf{x}_1 - \mathbf{B}_j) \cdot (\mathbf{B}_j - \mathbf{B}_{j+1})}{\|\mathbf{B}_j - \mathbf{B}_{j+1}\|^2}(\mathbf{B}_j - \mathbf{B}_{j+1}).$$

This projection represents the distance between $\mathbf{x}_1$ and the closest point within the j-th bone line segment (Figure 4.7). Then, we want to find the index of the closest bone j^*, defined as the bone whom projection has the least distance to $\mathbf{x}_1$:

$$j^* = \arg\min_{j^*} \left\|\mathbf{x}_1 - \text{proj}_{\mathbf{x}_1} j^*\right\|.$$

Following this, the value of $\text{proj}_{\mathbf{x}_1} j^*$ when $\mathbf{x}_1 = \mathbf{x}_1^0$ (i.e. the distance before the simulation starts) is stored as d (the resting distance). Similarly to all of our previous constraints, we then define the constraint function as the difference between the current distance to the resting distance:

$$C(\mathbf{x}_1) = \underbrace{\left\|\mathbf{x}_1 - \text{proj}_{\mathbf{x}_1} j^*\right\|}_{\text{current distance to the nearest skeleton}} - \underbrace{d}_{\text{resting distance}}. \tag{4.4}$$

Conceptually, this is an edge length constraint, except the second particle ($\mathbf{x}_2$) is replaced by $\mathbf{x}_1$'s projection to the nearest bone. This implies that the gradient can be computed the same way as edge length constraint (Equation 3.19). To help the readers visualise this constraint, we provided a visual illustration of its mechanics in Figure 4.7.

On a different note, there is an additional minor adjustment we need to make to the algorithm itself. Particularly, the fact that we are no longer simulating in a loop. When performing simulation, we typically have a notion of "continuity"; that is, we start at $\mathbf{x}^0$ to produce $\mathbf{x}^1$, then

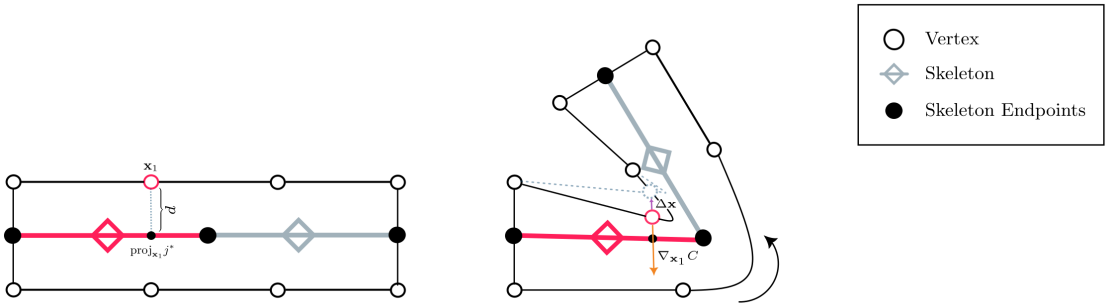


FIGURE 4.7: Inner workings of a bind constraint. The left image demonstrates its "binding" stage, where the red vertex is bound to the red skeleton. If the state of the skeleton changes (right), the bind constraint ensures that a vertex and the skeleton it binds to stays d unit distance apart.

repeat the process to generate $\mathbf{x}^2$, and keep repeating until the animation ends. In this context, however, we want to only advance "the clock" by one step — say, from $\mathbf{x}^t$ to $\mathbf{x}^{t+1}$, every time this layer is called. This keeps our simulation in sync with the artistic animation, such that the time that passes between frames are equal. To summarise, all the adjustments we describe resulted in the "single step" SMPBD algorithm included below:

Algorithm 2 "Single Step" SMPBD

Input: t,$\boldsymbol{\nu}'$, $\mathbf{M}$,$\tilde{\boldsymbol{\alpha}}$ ▷ Now takes input from the artist layer, and the current time t
procedure SIMULATE(t, $\boldsymbol{\nu}'$, $\mathbf{M}$,$\tilde{\boldsymbol{\alpha}}$)
  $\mathbf{x}^t \leftarrow \boldsymbol{\nu}'$
  compute predicted position $\tilde{\mathbf{y}} \leftarrow 2\mathbf{x}^t - \mathbf{x}^{t-1}$
  $\mathbf{x}^{(0)} \leftarrow \tilde{\mathbf{y}}$
  $\lambda^{(0)} \leftarrow \mathbf{0}$
  while $\Delta\mathbf{x} \geq \epsilon$ **do**
    for each $C_j \in \mathbf{C}$ **do** ▷ Now includes one bind constraint for every vertex
      $C_j(\mathbf{x}^{(i)}) \leftarrow$ EVALUATECONSTRAINT($type, \mathbf{x}^{(i)}$)
      $\nabla C_j(\mathbf{x}^{(i)}) \leftarrow$ EVALUATECONSTRAINTGRADIENT($type, \mathbf{x}^{(i)}$)
      compute $\Delta\boldsymbol{\lambda} \leftarrow \frac{-\mathbf{C}_j(\mathbf{x}^{(i)}) - \tilde{\boldsymbol{\alpha}}\boldsymbol{\lambda}^{(i)}}{\mathbf{C}_j(\mathbf{x}^{(i)})\mathbf{M}^{-1}\nabla\mathbf{C}_j^T(\mathbf{x}^{(i)}) + \tilde{\boldsymbol{\alpha}}}$
      compute $\Delta\mathbf{x} \leftarrow \mathbf{M}^{-1}\nabla\mathbf{C}_j^T(\mathbf{x}^{(i)})\Delta\boldsymbol{\lambda}$
      COMMITUPDATE(*strategy*)
    end for
    $i \leftarrow i + 1$
  end while
  $\mathbf{x}^{t+1} = \mathbf{x}^{(i)}$ ▷ Set the next state to the result given by the solver
  RENDEROBJECT($\mathbf{x}^{t+1}$) ▷ No longer loops and passes the control to back to artistic layer
end procedure

4.4 Evaluating Position-based Complementary Dynamics

The idea of having a pipeline with unified representation does sound enticing on paper. Yet, it boasts no practicality if its conceptual merits do not translate into practical benefits. Once again, quantifying these benefits boil down to measuring two inseparable and often conflicting factors: performance and visual quality. The rationale is that neither of these qualities is particularly useful without the other. For example, a simulation method from the field of engineering is capable of simulating the dynamics of a bridge with pinpoint accuracy but might take a week's worth of computation time. In contrast, an animation pipeline proposed decades ago will be considered lightweight by today's computers but might not adhere to the visual quality standards we have today. Neither of these methods are particularly desirable in our context because they lack a healthy balance between the two factors. Following this rationale, we include both factors in our evaluation. The performance evaluation is covered first in Section 4.4.1, followed by the visual quality evaluation afterwards (Section 4.4.2).

4.4.1 Performance Evaluation

We will begin the thorough evaluation of our method by measuring its comparative performance against other animation pipelines. It is important to note, however, that performance measurement in this section will not be done to the same level of granularity as the previous chapter, due to each method having vastly different (and not directly comparable) inner workings.

4.4.1.1 Setup

In this evaluation, we will rely on a general performance metric known as *frame time*. In the context of animation pipelines, this refers to the total compute time (end-to-end)[13] to produce a single frame of animation. To get a grasp about the overall performance of a method, this value will then be averaged over the duration of the animation.

The animation pipeline proposed in this chapter will be referred to as PBCD (position-based complementary dynamics). To represent the state-of-the-art in our line of work, we selected two methods that similarly augment artistic animation with physics-based animation[14]: (1) *articulated projective dynamics* (Li et al., 2019) — a sequential deformer combining continuum-based simulation with artistic animation through collision mechanism referred to as the "joint" step; (2) *complementary dynamics* (Zhang et al., 2020), the latest improvement to the rig-space method first coined by (Hahn et al., 2012). These methods will be referred to as APD and CD

[13] Preprocessing is excluded from this measurement, as it only needs to be run once during the entire animation duration.

[14] Both methods have been reviewed previously in Chapter 2, Section 2.3.

respectively. While we would like to include more methods as baselines, unfortunately, code replicability is a prominent issue in computer graphics (Bonneel et al., 2020). It is not a rare occasion for methods to lack important implementation details or source code to replicate their results. For completeness, we also included LBS+PBD, an almost carbon copy of PBCD, but uses Müller et al. (2007) original PBD instead of our SMPBD. This baseline represents what would happen if we try to integrate linear blend skinning and PBD together, but skipping the necessary work described in Chapter 3.

The testing instrument used to perform the measurement is exactly the same as in the previous chapter[15]. The same applies to the programming language (C++) and the visualisation tools (Blender). Whenever applicable, each particle/simulated object will have the exact mass of 1 kg. As Li et al. (2019) recommends 5 solver iterations for his APD, to ensure fairness, PBCD will also employ for 5 SMPBD iterations for its secondary motion.

4.4.1.2 Scenarios

As test beds, we adapted and designed four character animation tasks shown in Figure 4.8. In contrast to the scenarios depicted in the previous chapter, these tasks have no inherent theoretical meaning[16], and are instead designed to reflect various real world tasks possible with our animation pipeline. Despite that, they are carefully crafted to provide unique challenges and represent different tiers of character complexity. For instance, Blissful Blub and Sprightly Suzanne are designed as sanity tests that utilise simpler characters. Tender T-rex, on the other hand, represents character with high quality rigging; its high skeleton count facilitates the artist to encode finer movements in the character geometry, which makes it challenging to get right. Lastly, Scenic Spot is a stress test scenario with multiple characters within a single scene.

Each scenario will provide us with three things: (1) character geometry alongside its skeleton; (2) m skeleton transformations $(\mathbf{T}_1, \cdots, \mathbf{T}_m)$ every frame to represent the input from the artist; (3) physical measurement to describe the character's material. The mesh decimation preprocessing mentioned in Section 4.2.1 is used to tailor the provided meshes to the level of complexity shown in Table 4.2.

The character's underlying material is tuned to match animal flesh (Kalra et al., 2016), animal skin (Chen et al., 1996), muscle (Chen et al., 1996), and an inflatable (Zhang et al., 2020) respectively. To incorporate such material information into the animation, we must translate the value included above to parameters acceptable by the physics simulation of each method. APD and

[15] AMD Ryzen™ 6800HS @ 4.7 GHz,16 GB of RAM, WSL2 under Ubuntu™ 22.04.1 LTS.

[16] Scenarios in the previous chapter are special because they all represent cases where Newton's differential equation has a known or widely accepted theoretical solution.

FIGURE 4.8: Our four testing characters: BLUB (top left), SUZANNE (top right), T-REX (bottom right), SPOT (bottom left)

TABLE 4.2: Summary of geometry and material information for each of our testing scenario. The material composing each character is represented by its Young's modulus (κ) and Poisson's ratio (υ). The Young's modulus is stated in Gigapascal (GPa)

	Geometry				**Material**	
Scenario	**Vertices**	**Edges**	**Poly.**	**Skeleton**	κ (GPa)	υ
BLISSFUL BLUB	700	2092	1394	2	0.0007	0.500
SPRIGHTLY SUZANNE	1250	2100	1403	3	0.1	0.499
TENDER T-REX	3620	10590	7000	50	0.01	0.400
SCENIC SPOT	11700	23428	23426	24	4.33	0.350

CD use continuum-based simulation under the hood, which is tunable through the two *Lamé* coefficients: μ and λ. Fortunately, μ and λ can be computed from κ and υ through the formula below (Sifakis and Barbič, 2015):

$$\mu = \frac{\kappa}{2(1+\upsilon)} \tag{4.5}$$

$$\lambda = \frac{\kappa\upsilon}{(1+\upsilon)(1-2\upsilon)} \tag{4.6}$$

For SMPBD — whom material parameter is defined according to compliance matrix $\tilde{\boldsymbol{\alpha}}$, we utilise the theorem by Macklin et al. (2016) to connect it with the two *Lamé* coefficients:

$$\tilde{\boldsymbol{\alpha}} = \frac{1}{\Delta t} \begin{bmatrix} \lambda + 2\mu & \lambda & 0 \\ \lambda & \lambda + 2\mu & 0 \\ 0 & 0 & 2\mu \end{bmatrix}^{-1} \tag{4.7}$$

4.4.1.3 Results

Table 4.3 reports the performance measurements for all scenarios. At first glance, the performance of PBCD aligns with our expectations — beating both APD and CD in all but one scenarios. As also expected based on our measurements in the previous chapter, LBS+PBD outperforms PBCD slightly on all scenarios. This boils down to its physics simulation (PBD) being slightly faster than PBCD's physics simulation (SMPBD), as that is the sole distinction between our method and LBS+PBD. Nevertheless, as we will see in the evaluation of its visual quality later, this marginal performance difference is most certainly not justifiable. Throughout all scenarios, complementary dynamics (Zhang et al., 2020) performs the worst comparatively[17], even reaching slightly above 4 seconds per frame in TENDER T-REX.

TABLE 4.3: Frametime measurements for all scenarios. Columns are frametime for **BLISSFUL BLUB** (BB), **SPRIGHTLY SUZANNE** (SZ), **TENDER T-REX** (TT), **SCENIC SPOT** (SS). The ‡ is used to denote our method.

	Method	**BB** (ms)	**SZ** (ms)	**TT** (ms)	**SS** (ms)
Baselines	APD (Li et al., 2019)	17.3	46.8	659.1	1414.7
	CD (Zhang et al., 2020)	53.4	78.6	4092.1	3417.4
	LBS+PBD	20.3	22.4	99.4	159.9
‡	PBCD	22.4	27.3	101.5	180.0

Bringing the focus back to our method's comparative performance against the state-of-the-art, the speed-up factor offered by our method ranges between $1.7\times$ (vs. APD in SPRIGHTLY SUZANNE) and $40.3\times$ (vs. CD in TENDER T-REX) — a clearly remarkable improvement. Unfortunately, the overall frametime measurements provided above do not equip us with enough information to pinpoint exactly *why* such significant difference exist. Based on our intuition, the difference should come from the lack of additional overhead in our method, which is no longer necessary because both of our motions are generated the same way (by manipulating positions). Fortunately, APD, which is our closest competitor, has a similar structure to our pipeline. APD and PBCD both works by *stacking* approaches together in a sequential manner.

[17]The author (Zhang et al., 2020) did note that complementary dynamics focuses on demonstrating general applicability rather than performance optimisation, so this is expected.

Utilising this fact, we can break down the above's metric to a sum of its comprising components : (1) the cost of the artistic layer; (2) the cost of the physics layer; (3) additional operation cost (overhead) if any. We can then put our intuition to the test by comparing each cost individually, which produces the bar chart shown in Figure 4.9.

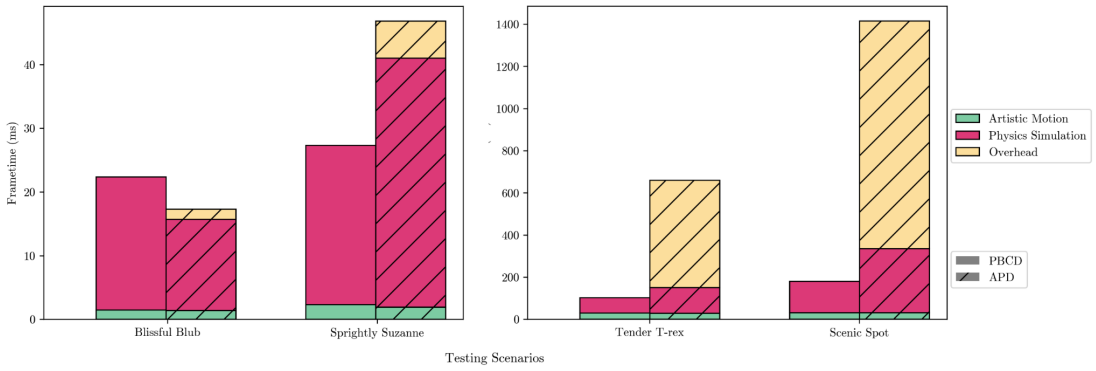


FIGURE 4.9: The frametime measurement for APD and PBCD broken down into its comprising elements. The figure is split into two subfigures to allow different y-scale without scaling any of the measurements.

Li et al. (2019) stated that their method uses *affine skinning* to handle artistic motion, which from our understanding, is yet another name for linear blend skinning. Given that we share the same approach, it makes sense that their cost to compute the artistic motion (green bar) is almost identical to ours. As also evident in Figure 4.9, the overhead cost (yellow bar) makes up most of the frame time difference between APD and our method, confirming our intuition. This overhead starts small, taking only 9% of APD's total frame time in BLISSFUL BLUB. However, it scales poorly with increased complexity — taking nearly 78% of APD's total frametime in TENDER T-REX. To reiterate Section 2.3, this overhead is necessary to provide "glue" between the two different representation APD use (position in the artist layer vs. forces in physics layer). In APD, this glue relies on detecting collision between vertices and skeletons within a character mesh. Naïve collision detection has a worst-case runtime of $O(mn)$, where n is the number of vertices and m is the number of skeletons. If the overhead includes naïve collision detection, it makes sense why it is particularly costly in TENDER T-REX, which has both: (1) significantly more bones; (2) almost three times as many vertices as SPRIGHTLY SUZANNE. By having physics simulation that also works with positions (like SMPBD), PBCD completely axe this costly glue.

That being said, this is not the sole culprit of the large performance gap between APD and PBCD. APD's physics simulation, which is based on continuum-based model, also seem to scale a lot worse compared to SMPBD. While it is theoretically true that position-based simulations generally run faster than its continuum-based counterparts (Bender et al., 2014), the difference should be the largely attributed to the solver iteration complexity. APD (and in this case, also CD) uses the standard Newton-Raphson method as their solver. The iteration cost of Newton-Raphson is $O(d^3)$ (Byrd et al., 1995), where d is the "dimension" or the solution search space.

In simulation of 3D characters, d will be the number of vertices times three. The $O(d^3)$ term comes from the cost of computing inverse hessian. As stated in Section 3.2.1, our SMPBD solver approximates the hessian with the mass matrix, reducing the cost to $O(d^2)$. The tradeoff of performing an approximation like this is that our solver will take more iterations to converge. However, since the number of iterations are fixed in our experiment, the ramifications likely come later in the form of reduced visual quality.

When all is said and done, the performance improvement offered by PBCD is evident. Even so, being an order of magnitude faster means nothing if the resulting visual is also an order of magnitude worse. In order to gauge how meaningful this improvement really is, we need to couple our performance measurements with a quantification of visual quality. To this end, we designed a user study to quantify the visual quality of our method in the next section.

4.4.2 Visual Quality Evaluation

A holistic assessment of an animation-generating pipeline would not be complete without any evaluation regarding the "goodness" of the generated animation. Unfortunately, this is also the part commonly lacking in many latest studies in character animation (Li et al., 2019, Zhang et al., 2020, Rohmer et al., 2021, Kalyanasundaram et al., 2022). The cause is likely the lack of the notion of *ground truth* in this context. When we evaluate our SMPBD in the last chapter, we are evaluating a pure physics-based animation. In that context, there is such thing as an animation that is "right" or "wrong" — in accordance with the theory of physics. In the context of full animation generation (i.e. one that involve artistic animation), there exist no general metric that can measure the visual fidelity of an animation (O'Sullivan et al., 2003). As such, to investigate the effectiveness of our approach beyond performance grounds, we conducted a user evaluation comparing the *subjective* visual quality of our method to the previous work.

4.4.2.1 Setup

Our user evaluation takes the form of an online survey developed specifically for this study [18]. We recruited 35 participants from computer science, computer animation and graphics programming backgrounds. The motivation behind this selection comes from Chamberlain et al. (2019) — which indicates that some degree of expertise increases one's sensitivity towards differences in the same realm. Since the differences between the generated animations are often not visually abundant, sensitivity towards small differences is a necessary trait. This study collects two forms of demographic data: (1) age; (2) field of expertise — both of which are self-reported. The summary of participant's demographic information for this study is shown

[18] Accessible through https://conrev.github.io/PBCD-Survey, please note that responses are **no longer recorded**.

in Figure 4.10. Unless otherwise stated, the testing scenarios and evaluated methods in this section are exactly the same as the previous section.

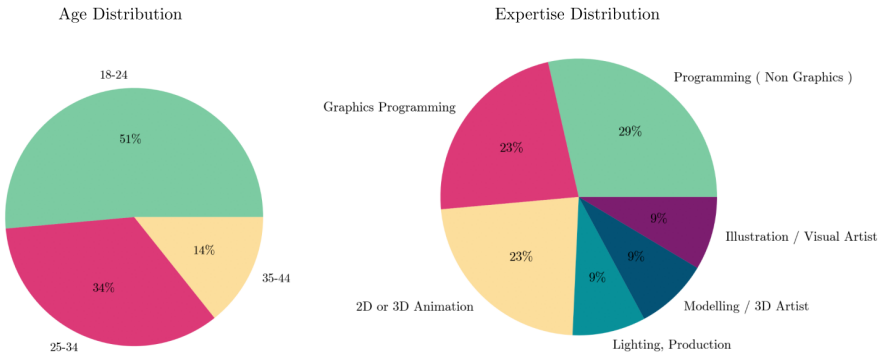


FIGURE 4.10: Pie charts of participant's age (left) and expertise (right). The majority (51.8%) of our participants were within the range of 18–24, with general programming being the most common expertise.

Tasks The survey is composed of four set of tasks. Each task corresponds to a single testing scenario in Section 4.4.1.2. The page for each task (Figure 4.11) includes two subtasks:

1. ***Motivating* subtask**. The motivating subtask is used to gather data to motivate the importance of our study. In this subtask, participants are given a side by side video of an animated character. One side has only artistic motion, while the other side has both artistic and physics-based motion (generated by our method). The question here is a simple yes–no question, whether the animation that includes both motions are more compelling[19]. In the case of a "no", participants are also given a chance to provide their reasoning via a text box.

2. ***Comparative* subtask**. This subtask investigates the effectiveness of our approach compared to previous work. In this subtask, participants are provided 4 animations — labelled from "A" to "D". Each label represents one of the methods in Section 4.4.1.1, which are randomised and hidden from the participants to avoid carryover bias (Ferris et al., 2001). The participants are then required to rate the visual quality of each animation on a scale from one (not visually compelling at all) to five (very visually compelling). They are also given the option to put their reasoning into words by filling the text box right below every rating.

[19]Needless to say, if people end up preferring animation with only artistic animation, there is no point of building a method that combines both motions like ours.

FIGURE 4.11: The page for a single set of evaluation tasks. Each page consists of a motivating block (top) and a comparative block (bottom). Each block starts with a quick instruction detailing information about the scene. This is followed by the generated animation and a set of input fields to complete the related task.

Data Analysis for the comparative subtask The main purpose of this user study is to quantify the visual fidelity of each method and to decide whether there is a method that generates visually more appealing animation than others (if so, which one). In statistical terms, our *dependent* variable is the visual quality rating of the resulting animation and the *independent* variable is the method used to generate such animation. Given that our dependent variable is on a likert scale and the difference between each point in the scale is not uniform, parametric statistical tests that rely on comparing sample means (ANOVA, t-test) are not appropriate (Boone et al., 2012). While some experts have argued that such tests are robust enough to still provide "the right answer" under multiple violations of assumptions (Sullivan and Artino Jr, 2013), we took the conservative route and utilised nonparametric tests. In particular, the *Kruskall-Wallis* test ($\alpha = 0.05$, $N = 35$, $df = 3$) is used to determine whether there is a statistically significant difference in visual fidelity — determined by their *mean ranks*, between each method. To pinpoint which method pairs exhibited the difference, the *Dunn's-Bonferonni* (adjusted $\alpha = \frac{0.05}{\text{number of pairs}} = 0.0083$) test is chosen as the post-hoc test.

4.4.2.2 Results

Motivating subtask The experimental results for the motivating subtask is shown in Figure 4.12. Generally speaking, an overwhelming majority of the participants prefers animations with both physics-based and artistic animation in it. The percentage of participants preferring it the other way around is non-existent in the first scenario, and at best, 9 percent (rounded up) in the Tender T-rex scenario. 9 percent translates to 3 participants given the 35 total participants, which are not much at all. While this by no means provides justification for our method specifically, it demonstrates the importance of our line of work — combining artistic and physics-based animation.

Nonetheless, to gauge which aspect of physics-based secondary motion is not welcomed within an animation, we manually inspected the reasoning (Table 4.4) provided by every participant that prefer strictly artistic animation. Regrettably, the comments targeted towards result in Sprightly Suzanne and Scenic Spot are vague and do not offer any further insights. However, the 3 reasoning provided in the Tender T-rex scenario do point to a single factor, and that is that the secondary motions added by our method are too jiggly, especially in the nose area.

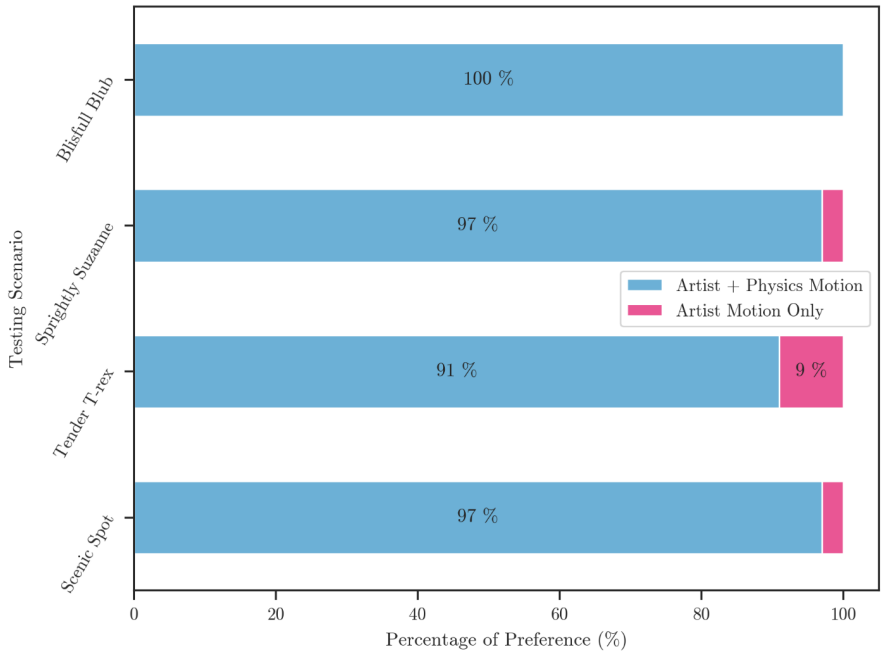


FIGURE 4.12: Bar plot representing participant's responses to the motivating task, transformed to rounded percentage. Each bar in the y-axis represents a single testing scenario.

TABLE 4.4: Reasoning provided by participants who prefer solitary artistic animation, divided for each scenario. The (P#) at the beginning of each quote is used to denote the participant number.

Testing Scenario	**Reasoning**
SPRIGHTLY SUZANNE	(P14) "*The added jiggles does not add value to the animation*"
TENDER T-REX	(P7) "*In the physics version, the nose is too bouncy while the areas that have fat e.g. thighs does not have enough*"
	(P14) "*Less secondary motion on the nose*"
	(P15) "*Could be slightly less secondary animation*"
SCENIC SPOT	(P31)"*The added physics feels unrealistic*"

We then manually investigated each frame of the aforementioned scenario. Interestingly, as seen in Figure 4.13, the nose in the physics-enhanced version does noticeably jiggle more than its sole primary motion counterparts. Whether these additional jiggles are desirable is difficult to quantify; but this does raise an important point. The SMPBD simulation used to generate these jiggles are tuned to have the material of a "muscle" (with diagonal entries of $\tilde{\boldsymbol{\alpha}}$ being $0.31 \times 10^{-1}\ m^2/Newton$, following Section 4.4.1.2). However, the main issue is that this material is applied to the *entire* body. In other words, the character exhibits uniform bounciness

throughout the entire body. This is unfortunately not a faithful representation of the real world (Holzapfel et al., 2001), as different parts of our body have slightly different material behaviour. The closest "workaround" we have in our method currently is to set particle mass to different values in each region, which might mute the jiggles in the region with a higher mass. Nonetheless, the ability to apply different material to different parts of the character is actually a solved problem (Abu Rumman and Fratarcangeli, 2015, Li et al., 2016), which might be worth revisiting for our method moving forward.

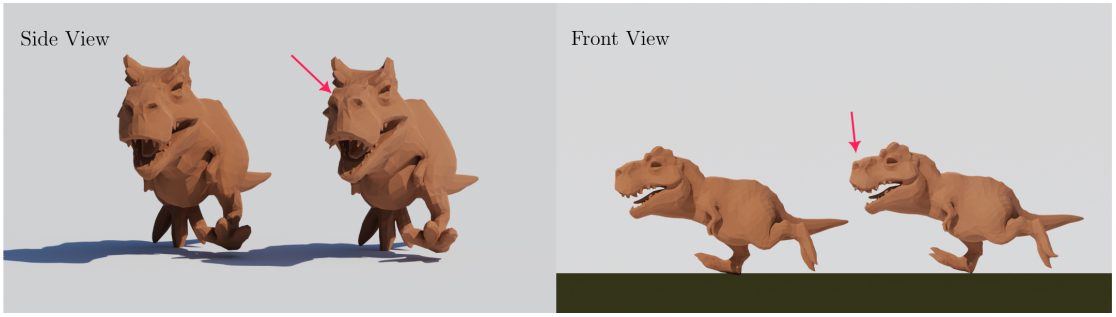


FIGURE 4.13: The bouncy nose phenomenon mentioned by Participants 7 and 14. For comparison, we also included the sole artistic animation version on the left.

Comparative Subtask Before diving into the statistical test results, we begin by providing interpretation to the key statistic used in this evaluation: the *mean rank*. When evaluating ordinal data, it is customary to use the *median* (IQR) as the main descriptive statistics (Boone et al., 2012). Throughout this section, we will still occasionally refer to the median, but the main interpretation will be decided by mean rank. The issue with median in our data is that it is often not granular enough, given that our sample rating can only take a value between 1 and 5[20]. Mean rank offers higher granularity while keeping a similarly simple interpretation: the higher the mean rank for a method is, the higher visual rating it tends to get from the participants.

Now, Table 4.5 provides the necessary evaluation statistics *per* scenario. In BLISSFUL BLUB, our test indicates that there is a statistically significant difference in visual quality of the generated animations. In particular, the Dunn's post-hoc test with Bonferroni adjustments (adjusted $\alpha = 0.0083$) revealed that those significant differences happen between APD – CD ($p = 1.9 \times 10^{-11}$), CD – LBS+PBD ($p = 0.39 \times 10^{-5}$) and CD – PBCD ($p = 5.2 \times 10^{-9}$). In short, the visuals generated by CD (Zhang et al., 2020) is significantly different from the rest. A quick glance at its statistics (mean rank = 30.0, median rating = 1) shows that these differences are for the worse.

The extreme nature of CD's rating measurements hinted that the defect in visual quality is likely very evident. This is quite peculiar considering its quite costly nature (Section 4.4.1.3) — if anything, we expected it to top the visual quality evaluation. Once we inspected the animation

[20] On this occasion, it is possible to have a statistically significant rating difference with the exact same median.

TABLE 4.5: User evaluation results for comparative subtask. The red or green mark on p-value denotes whether the null hypothesis is rejected.

	Mean Ranks				Test Statistic	
Scenario	**APD**	**CD**	**LBS+PBD**	**PBCD**	***H***	***p*-value**
BLISSFUL BLUB	93.3	30.0	73.5	85.1	53.59	1.37×10^{-11}
SPRIGHTLY SUZANNE	74.2	96.0	42.6	69.1	33.15	3×10^{-7}
TENDER T-REX	79.8	92.4	37.4	72.38	38.19	2.8×10^{-8}
SCENIC SPOT	66.1	80.6	61.4	74.2	7.37	0.6×10^{-1}

(Figure 4.14), however, two things become clear: (1) the issue in the generated animation is indeed obvious (2) it is likely not an intended behaviour. We singled out the reasoning of this artefact to the material used in this scenario. The material used in BLISSFUL BLUB ($\kappa = 0.0007$ & $\upsilon = 0.500$), if translated into *Lamé* second coefficient (parameters used by continuum-based model within CD), will give us an undefined value ($\lambda = \frac{0.0007 \times 0.5}{(1.5)(0)} = \frac{0.00035}{0}$). This will cause the Piola-Kirchhoff stress tensor ($\mathbf{P}$) to have an infinite value[21]. Having an infinitely large stress will cause a simulation to behave unpredictably, and it depends on the implementation to handle this corner case. CD's implementation does not have this case handled, thereby causing the issue.

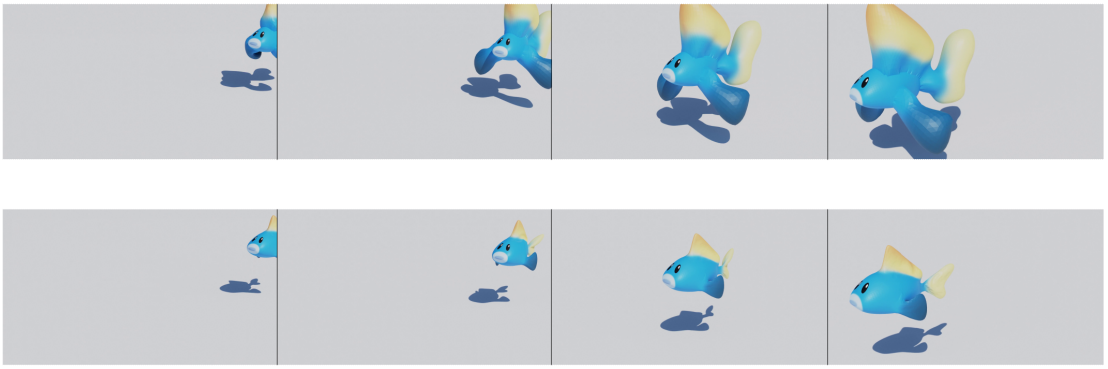


FIGURE 4.14: BLISSFUL BLUB animation generated by complementary dynamics (top). To provide context about what a "good" animation looks like for this scenario, the results of the highest mean rank method (APD) is provided at the bottom.

Focusing on the results for the rest of the method (APD, LBS+PBD and PBCD), our series of Dunn's-Bonferonni post-hoc tests (adjusted α = 0.083) indicate that there is no statistically significant rating difference between the three ($p = 0.22$, $p = 0.036$, $p = 0.38$) — implying that they produce comparable visuals. Still, the slightly lower statistic (mean rank = 85.1, median rating = 3) of our method compared to APD (mean rank = 93.3, median rating = 4) indicates that there are some parts of our animation that are less satisfactory. Comments from the participants attribute this lower score to the behaviour of the tail and fins in our animation: (P6)"*The tail*

[21] In numerical sense.

does not look right."; (P13)"*the fish tail looks like paper*"; (P28) "*The tail has a lot of artefacts*". We show the full render of the problematic tail from different angle in Figure 4.15.

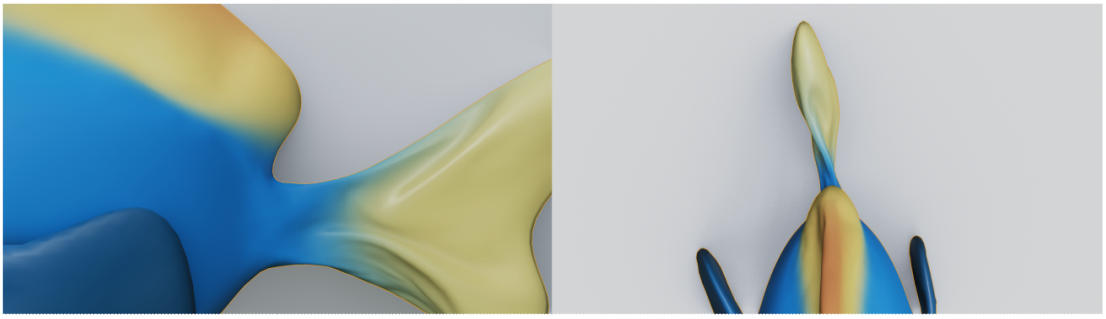


FIGURE 4.15: The problematic paper-like fins mentioned by participants 6, 13 and 28. The left and right image depict the top and side view respectively.

Clearly, the tail of the fish looks somewhat "crumpled". Upon further investigation, the cause of this is the fact that in PBCD, we have no mechanism that handles self-collision (collision between vertices/edges within the character mesh). As evident in Figure 4.16 by the darker colours around the circled region, there are a lot of edges penetrating each other, creating a *crease-like* effect on the fins. The reason why this is becoming an issue now, and not observable in the standalone evaluation of our physics simulation (SMPBD), is due to the especially thin region in BLUB's tail. Under normal circumstances, the edge length constraint in SMPBD will keep the vertices within d distance to each other, preventing self-collision. However, when the region is thin, d will be initialised to a very small value. Given that constraints in SMPBD are soft (enforced gradually), when rest pose distance d is small, there is a higher probability for an edge distance to hit zero during the gradual process of bringing the particle into a constraint satisfying position.

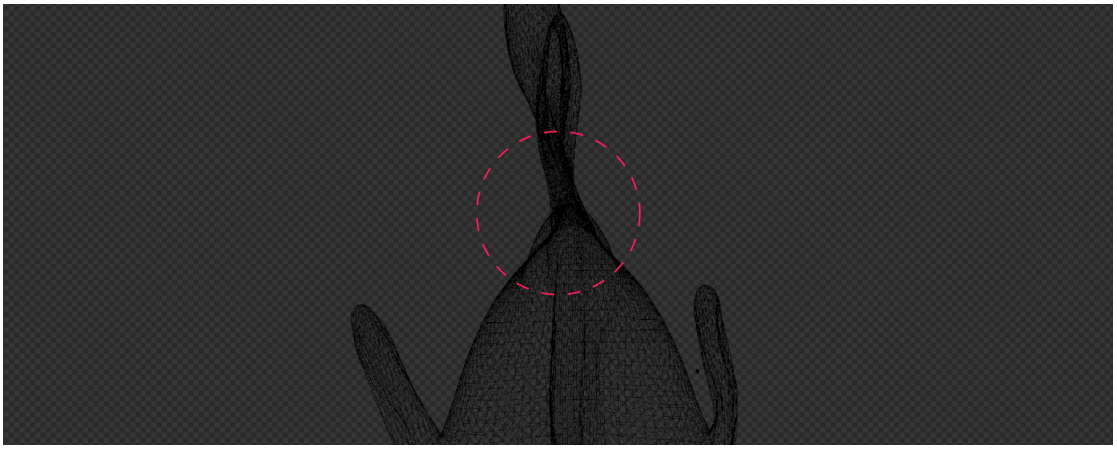


FIGURE 4.16: Wireframe render of the Fish's problematic tail.

Moving on to SPRIGHTLY SUZANNE, a Kruskall Wallis test ($\alpha = 0.05$, $N = 35$, $df = 3$) indicates that there is a statistically significant visual rating between samples ($p = 3 \times 10^{-7}$). The disparity happens between LBS+PBD and the other three methods, with a p-value of 0.00071

(vs. APD), 1.1×10^{-8} (vs. CD), and 0.0045 (vs. PBCD). Once again, there is a method that receives significantly worse ratings. Except this time, this result is expected. LBS+PBD uses original position-based dynamics to handle the secondary motion. We expected it to generate less compelling animation, since its iteration dependent stiffness will create infinitely stiff objects which have no secondary motion at all. This is exactly what happened in this instance, as observable in Figure 4.17.

FIGURE 4.17: LBS+PBD (top) producing no secondary motion, observable by its indistinguishable result from the pure artistic motion version (bottom).

In absence of visual artefacts and instability issues, CD (mean rank = 96, median rating = 4) is able to top all methods in terms of animation quality, even showing statistically significant rating difference to our method (p-value = 0.0039). A great point is made by participants 21 and 34[22], stating that PBCD changes the trajectory intended by the artist, while CD preserves it. This is theoretically correct, because of the way we fundamentally apply the secondary motion. In PBCD, the primary motion is first applied to the character mesh via linear blend skinning; then, that very mesh is fed into a physics simulation to augment it with secondary motion. However, by performing simulation on that same mesh, we are bound to alter the look and trajectory of the primary motion slightly (Figure 4.18). In contrast, CD works by finding physically correct wiggles that are in the *orthogonal* direction of the primary motion (Zhang et al., 2020), meaning it actively avoids impacting the primary motion. In this sense, complementary dynamics, is strictly *complementary*.

The rating data from TENDER T-REX tells a similar story to SPRIGHTLY SUZANNE, that is, there is a statistically significant difference between the rating received by each method (p-value = 2.8×10^{-8}). The post-hoc Dunn's test also attributed this difference to LBS+PBD (mean rank = 37.4, median rating = 2) being worse than the other three. Participant's comments (Table 4.6) suggest that even the reasoning is a carbon copy of the previous scenario. By this point, we have enough evidence to support our claim in the previous section, that the extra performance

[22]Interestingly, they both declared "Modelling/ 3D artist" as their expertise.

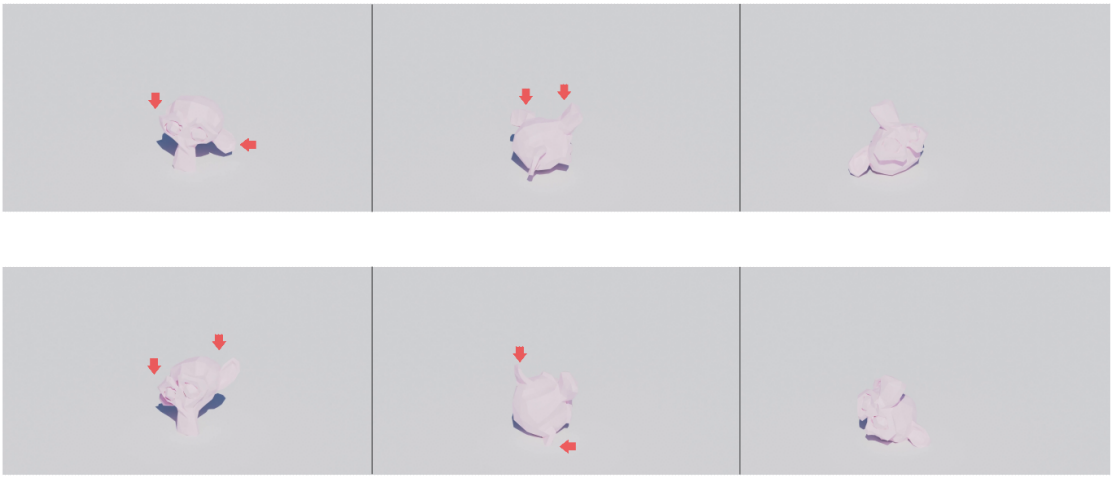


FIGURE 4.18: The impact of added jiggles in SPRIGHTLY SUZANNE. The results are from CD (top), and PBCD (bottom). Despite both method being successful at adding realistic jiggles (red arrow), The result from CD is still reminiscent of the artist's version in Figure 4.17, whereas PBCD's version looks like a completely different animation.

offered by LBS+PBD is not justified. Furthermore, the lacklustre animation quality of LBS+PBD demonstrates the importance of our work in Chapter 3. Without our modification, combining PBD with linear blend skinning offers little to no benefit at all.

TABLE 4.6: Reasoning for LBS+PBD lower rating, randomly selected from the 35 participants.

Testing Scenario	**Reasoning**
TENDER T-REX	(P1) "*Barely any wiggles at all.*"
	(P20) "*No secondary animation.*"
	(P22) "*This method has no discernible secondary motion, which I could see being realistic if the material was very stiff.*"
	(P30) "*Feels very slippery.*"
	(P34) "*No observable difference to the input.*"

Strangely enough, SCENIC SPOT — our last and most complex scenario, is the only scenario where our Kruskall-Wallis test ($\alpha = 0.05$, $N = 35$, $df = 3$) suggest no statistical difference in the rating received by each method. Needless to say, this is especially odd after knowing that LBS+PBD tends to not produce any secondary motion at all. One viable explanation is that the secondary motion added here is not in the usual form of wiggles or swaying. The material we use here (inflatable) has a Young's modulus of 4.33 (Section 4.4.1.2). Young's modulus is the essence of material stiffness, telling us how easily a material can stretch and deform. As depicted

in Figure 4.19, the value of 4.33 represents a stiffness between polymer and wood, both of which can be difficult to bend or deform[23]. Because the material is quite stiff, the "complementary" effect here is more of a ballooning (overpressure) effect, rather than the usual wiggles (Bender et al., 2013). The problematic issue of LBS+PBD creating an overly stiff depiction of a character is, of course, less jarring to the eye when the character itself is meant to be stiff.

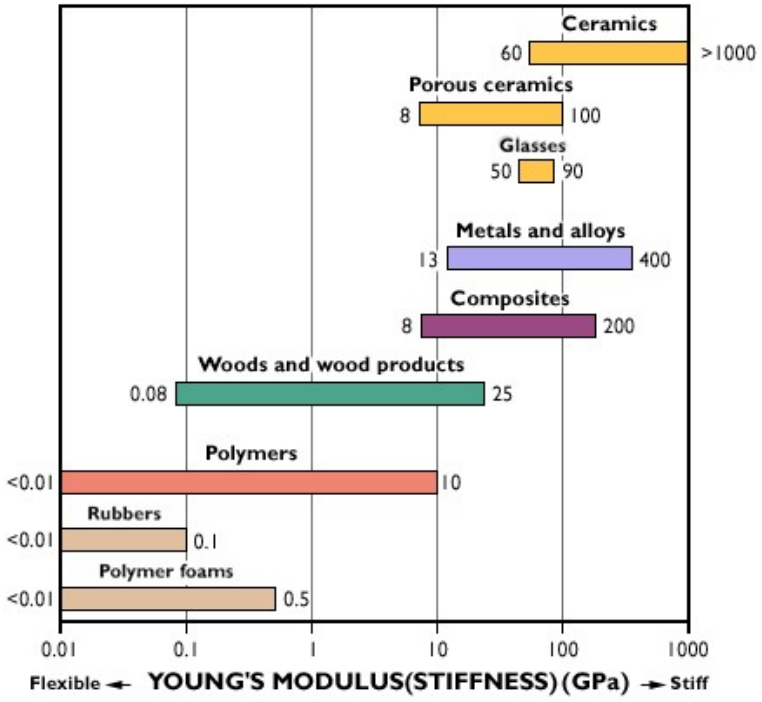


FIGURE 4.19: Young's modulus of many real life materials, measured in gigapascal (GPa). Taken from Cambridge (2023) without permission.

That being said, having a stiff material does not necessarily mean that the generated animation will be the identical. The ability to simulate an accurate depiction of a stiff material will still be variable method-to-method. In fact, as shown in Figure 4.20, the difference can be quite abundant. Interestingly, despite the results arguably being "different enough", those differences are not reflected in the given ratings (Table 4.5). Likely, this has to do with the notion of plausibility vs. accuracy (Barzel et al., 1996). Because humans have an imprecise mental model of physical dynamics, we are only capable of differentiating physically implausible motions from the plausible ones. It is rare for us to be able to pinpoint exactly which animations are the most accurate (O'Sullivan et al., 2003). As long as the resulting animation is within the realm of plausibility, it is unlikely to be perceived as significantly less compelling. In Figure 4.20, the two rendered frames are quite distinct. However, none stands out as being particularly physically "implausible" (compare this to, say, Figure 4.14). While it does not imply that our method generates "similar" or "better" animations, this finding is certainly welcomed.

[23] Reminiscent of a real-life inflatable.

it demonstrates that in this scenario, PBCD produces as *equally believable* character animations as the state-of-the-art.

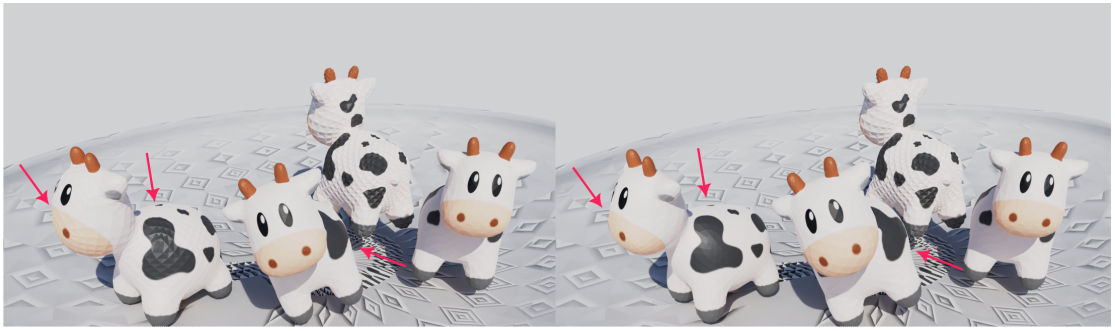


FIGURE 4.20: SCENIC SPOT frame 6 render by PBCD (left) and CD (right), notice that the distinctions between the two (red arrows) are quite clear, despite the two not having statistically significant mean rank difference.

To conclude, based on our user study, the visual quality of PBCD is within our expectations. Rating wise, our method consistently ranks in the second or the third spot behind either CD or APD. Even better, in three out of four scenarios, the difference it has to the first place is not statistically significant. Given that our method is an order of magnitude faster than its competitors, being able to produce visually comparable animations is an impressive feat. Having said that, PBCD can definitely benefit from additional work focusing on its visual quality. In particular, to remedy minor quirks and visual artefacts noted by our participants.

CHAPTER 5

Conclusion

In this thesis, we tackle the problem of producing lifelike character animations by integrating hand-crafted and physics-based animation. Particularly, we address the need for a method that can perform such a task in an intuitive and computationally efficient manner. To this end, we proposed *position-based complementary dynamics* (PBCD), a two-layer animation pipeline that works by sole manipulation of vertex positions.

In the centre of PBCD, is our novel physics simulation method: *secondary motion position-based dynamics* (SMPBD). SMPBD is built upon position-based dynamics, inheriting its core constraint mechanisms and position-only nature. In contrast to its predecessor, however, SMPBD is capable of accurately simulating elastic motion and jiggle effects. This is made possible by formulating our problem into compliant constraint dynamics, where constraints "steer" the behaviour of the system but need not be satisfied immediately (RQ1). This system is then discretised and solved with Newton-Raphson method to culminate a realistic simulation of elastic objects.

Harnessing SMPBD's ability to simulate elastic behaviour, it is used within PBCD alongside *linear blend skinning* (LBS). To achieve the ultimate goal of believable character deformations, PBCD employs a simple sequential architecture that begins by applying the artist's desired primary motion using linear blend skinning. Given the matching representation between LBS and SMPBD, the output produced by LBS can be immediately used as an initial condition for SMPBD simulation, yielding the desired secondary effects all without the need of complex architecture or any bridging mechanism (RQ2). In the absence of any translational mechanism and complex architecture, the resulting method will be efficient and easy to tune, filling the required gap within the literature.

Additionally, we have performed a comprehensive evaluation of our newfound pipeline, utilising a combination of quantitative evaluation and user study. Performance-wise, we found out that our proposed method runs at a fraction of the cost of the previous methods (RQ3). Further, we have shown that this improvement can be attributed to the simple architecture and the matching representation. That being said, we only manage to hit real-time frame rate — our initial performance goal, on simpler characters. Therefore, some additional performance work might be necessary for adoption in real-time use cases such as video games. On a different note, our user evaluation indicates that the visual quality of PBCD, while adequate, is mediocre at best. Despite rarely showing significantly worse ratings than other methods, it often receives negative attention because of the uncanny artefacts it tends to produce (RQ3). All thing considered, while PBCD does not, by any means, render all other pipelines irrelevant, it definitely raises the bar for the performance requirement of character animation pipelines moving forward.

5.1 Limitations and Future Work

Energy Preservation in Secondary Motion PBD

In Section 3.5.4, we ran into a problem where our simulation is incapable of mimicking a frictionless harmonic oscillator to a great degree, mainly because of the innate energy dissipation in the system. We mentioned that this is a byproduct of implicit Euler time integration, which is known to reduce the system's total mechanical energy over time (Bargteil and Shinar, 2018). While this is desirable most of the time as it brings stability and resemblance to the non-frictionless real-world (LeVeque, 2007), given that animation is an artistic outlet, there will be time and place where mimicking the real world is not the desired behaviour.

We have provided hints to a potential solution throughout Chapter 3 — by replacing the time integrator. There is a wealth of work in energy-preserving numerical integrators both to solve general ordinary differential equations (Gautschi, 2011), and even specifically targeting Newton's equation of motion (Dinev et al., 2018, Cetinaslan and Chaves, 2019, Löschner et al., 2020). Adapting these time integrators should not be difficult, but might come with their own quirks — absence of unconditional stability, performance issues and difficulties in implementation just to name a few. As a first step, we recommend adapting BDF2 time integrator. While this does not conserve mechanical energy per se, it has orders of magnitude less numerical damping (Bender et al., 2015), which serves a good enough workaround until a more proper solution is devised.

Another approach that can be taken is by imposing an energy preservation constraint. In particular, Pan et al. (2018) did this by imposing the below constraint:

$$C(\mathbf{x}_1, \mathbf{x}_2, \mathbf{x}_3, \mathbf{x}_4) = \frac{1}{2} \sum_{i=1}^{4} k_i (\|\mathbf{x}_i - \mathbf{x}_0\| - d_i)^2, \tag{5.1}$$

where k_i is the spring constant for the "virtual" spring between $\mathbf{x}_i$ and $\mathbf{x}_0$. Similarly, d_i is the rest length of the aforementioned spring. Like our volume constraint, this constraint involves a single tetrahedron with its barycenter located at $\mathbf{x}_0$. Introducing this to our soft constraint PBD might be challenging, however, as constraints are not enforced strictly in our method — preventing the energy to be fully conserved.

Self-Collision

Position-based complementary dynamics that we propose is able to bring the characters to life through expressive-but-realistic character animation. Nevertheless, one deficiency that is pointed out quite often during our user evaluation is the presence of "crumpled" body parts in the animated character. This stems from the lack of self-collision handling in our method. Self-collision is quite an intricate problem in computer graphics due to it consisting of two performance-intensive operations: collision detection and response.

For detection, we recommend exploiting the special characteristics of skeleton-driven characters, which is that their collision happens mostly in a localised region near the connection between two bones (i.e. joints). Knowing this, one can build a *spatial hashing* strategy and hash the bounding box of the tetrahedron near the joints. An $O(1)$ algorithm can then be devised to quickly figure out if a particle intersects any bounding boxes, resulting in a collision. The closest work that might serve as a reference is Abu Rumman and Fratarcangeli (2016).

For response, we recommend starting with a straightforward method such as the *vertex projection* (Vaillant et al., 2013). The idea of vertex projection is to first build the so-called *projection plane*. Given the contact point $\mathbf{c}$, the projection plane is a plane perpendicular to the line segments of the bone closest to $\mathbf{c}$. With the projection plane computed, responding to the collision is simply a matter of pushing the current particle $\mathbf{x}$ out to the surface of the projection plane along its normal:

$$\Delta\mathbf{x} = -\mathbf{n} \|\mathbf{x} - \mathbf{c}\|, \tag{5.2}$$

where $\mathbf{n}$ denotes the normal vector of the projection plane.

Parallelization

Another viable research direction is to optimise the computationally efficient nature of position-based complementary dynamics even further. While the results in Section 4.4.1 shows overwhelming performance improvement compared to other animation pipelines, production-level implementation of these methods often apply various optimisations that might alter the situation slightly. Of these commonly implemented optimisations, one that is problematic in the context of our method is parallelisation.

As concluded in Section 3.5, our SMPBD_{GS} shows better performance compared to its Jacobi counterparts and therefore is used in position-based complementary dynamics. However, when describing the nature of the two update (Section 3.2.2), we also mentioned that our parallelised Gauss-Seidel update might not be optimal due to threads having to wait until all constraint that uses the same particle to finish. This might not pose an issue in the context of a smaller scale research, but potentially problematic in highly parallel production environments. For future work, we recommend techniques such as graph colouring (Fratarcangeli et al., 2016) to better distribute the updates such that the waiting time is minimised. For example, by colouring constraints that share a particle with a different colour, then dispatching a single thread for each colour.

Supporting Other Sources of Primary Motion

In this thesis, we mainly deal with artistic animation that is created through skeletal deformation. As we know, this is not the only means of producing artistic animation; per-vertex animation and motion capture are two other ways artists can create primary motion. To reach a wider userbase, in the future, PBCD should also support these other means of creating primary motion. Considering that these two approaches also adhere to the position-only nature of PBCD, supporting them should be as simple as replacing the linear blend skinning layer within PBCD.

Bibliography

Tim McLaughlin, Larry Cutler, and David Coleman. Character rigging, deformations, and simulations in film and game production. In *ACM SIGGRAPH 2011 Papers*, 2011.

Frank Thomas, Ollie Johnston, and Frank Thomas. *The illusion of life: Disney animation*. Hyperion New York, 1995.

Harold Whitaker and John Halas. *Timing for animation*. Routledge, 2013.

Fabian Hahn, Sebastian Martin, Bernhard Thomaszewski, Robert Sumner, Stelian Coros, and Markus Gross. Rig-space physics. *ACM Transactions on Graphics (TOG)*, 31(4):1–8, 2012.

Prem Kalra, Nadia Magnenat-Thalmann, Laurent Moccozet, Gael Sannier, Amaury Aubel, and Daniel Thalmann. Real-time animation of realistic virtual humans. *IEEE Computer Graphics and Applications*, 18(5):42–56, 1998.

John P Lewis, Matt Cordner, and Nickson Fong. Pose space deformation: a unified approach to shape interpolation and skeleton-driven deformation. In *Proceedings of the 27th Annual Conference on Computer Graphics and Interactive Techniques*, pages 165–172, 2000.

Alec Jacobson, Zhigang Deng, Ladislav Kavan, and JP Lewis. Skinning: Real-time shape deformation. In *ACM SIGGRAPH 2014 Courses*, 2014.

David Antonio Gomez Jauregui. *3D motion capture by computer vision and virtual rendering*. PhD thesis, Institut National des Télécommunications, 2011.

Adam W. Bargteil and Tamar Shinar. An introduction to physics-based animation. In *ACM SIGGRAPH 2018 Courses*, 2018.

Hayley Iben, Mark Meyer, Lena Petrovic, Olivier Soares, John Anderson, and Andrew Witkin. Artistic simulation of curly hair. In *Proceedings of the 12th ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, 2013.

Agnes Daldegan, Nadia Magnenat Thalmann, Tsuneya Kurihara, and Daniel Thalmann. An integrated system for modeling, animating and rendering hair. In *Computer Graphics Forum*, 1993.

Steve Capell, Seth Green, Brian Curless, Tom Duchamp, and Zoran Popović. Interactive skeleton-driven dynamic deformations. *ACM Transactions on Graphics (TOG)*, 21(3):586–593, 2002.

Nora S Willett, Wilmot Li, Jovan Popovic, Floraine Berthouzoz, and Adam Finkelstein. Secondary motion for performed 2d animation. In *Proceedings of the 30th Annual ACM Symposium on User Interface Software and Technology*, 2017.

Junggon Kim and Nancy S Pollard. Fast simulation of skeleton-driven deformable body characters. *ACM Transactions on Graphics (TOG)*, 30(5):1–19, 2011.

Jing Li, Tiantian Liu, and Ladislav Kavan. Fast simulation of deformable characters with articulated skeletons in projective dynamics. In *Proceedings of the 18th Annual ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, 2019.

Donald House and John C Keyser. *Foundations of physically based modeling and animation*. AK Peters/CRC Press, 2016.

Kenny Erleben, Jon Sporring, Knud Henriksen, and Henrik Dohlmann. *Physics-based animation*. Charles River Media Hingham, 2005.

Fernando De Goes. Physics-based animation at pixar. MIT Distinguished Seminar Series in Computational Science and Engineering, 2017. URL https://cse.mit.edu/events/physics-based-animation-at-pixar/.

Matthias Müller, Jos Stam, Doug James, and Nils Thürey. Real time physics: class notes. In *ACM SIGGRAPH 2008 Classes*, 2008.

Matthias Müller, Miles Macklin, Nuttapong Chentanez, Stefan Jeschke, and Tae-Yong Kim. Detailed rigid body simulation with extended position based dynamics. In *Computer Graphics Forum*, 2020.

Miles Macklin, Matthias Müller, and Nuttapong Chentanez. Xpbd: position-based simulation of compliant constrained dynamics. In *Proceedings of the 9th International Conference on Motion in Games*, 2016.

Jiayi Eris Zhang, Seungbae Bang, David I.W. Levin, and Alec Jacobson. Complementary dynamics. *ACM Transactions on Graphics (TOG)*, 39(6):179–189, 2020.

N Abu Rumman and Marco Fratarcangeli. State of the art in skinning techniques for articulated deformable characters. In *Proceedings of the 11th Joint Conference on Computer Vision, Imaging and Computer Graphics Theory and Applications*, 2016.

Andrew Nealen, Matthias Müller, Richard Keiser, Eddy Boxerman, and Mark Carlson. Physically based deformable models in computer graphics. *Computer Graphics Forum*, 25(4):809–836, 2006.

Jin Huang, Jiong Chen, Weiwei Xu, and Hujun Bao. A survey on fast simulation of elastic objects. *Frontiers of Computer Science*, 13(3):443–459, 2019.

Steve Marschner and Peter Shirley. *Fundamentals of computer graphics*. CRC Press, 2018.

Chen Liu. *An analysis of the current and future state of 3D facial animation techniques and systems*. PhD thesis, School of Interactive Arts & Technology-Simon Fraser University, 2009.

Richard Williams. *The animator's survival kit: a manual of methods, principles and formulas for classical, computer, games, stop motion and internet animators*. Macmillan, 2012.

Raphael Linus Levien. *From spiral to spline: Optimal techniques in interactive curve design*. University of California, Berkeley, 2009.

Mark Halstead, Michael Kass, and Tony DeRose. Efficient, fair interpolation using catmull-clark surfaces. In *Proceedings of the 20th Annual Conference on Computer Graphics and Interactive Techniques*, 1993.

Midori Kitagawa and Brian Windsor. *MoCap for artists: workflow and techniques for motion capture*. Routledge, 2020.

Andreas Vasilakis and Ioannis Fudos. Skeleton-based rigid skinning for character animation. In *Proceedings of the 4th Joint Conference on Computer Vision, Imaging and Computer Graphics Theory and Applications*, 2009.

Blender Foundation. *Blender - a 3D modelling and rendering package*. Stichting Blender Foundation, Amsterdam, 2018. URL http://www.blender.org.

David Jacka, Ashley Reid, Bruce Merry, and James Gain. A comparison of linear skinning techniques for character animation. In *Proceedings of the 5th International Conference on Computer graphics, Virtual Reality, Visualisation and Interaction in Africa*, 2007.

Nadine Abu Rumman and Marco Fratarcangeli. Position-based skinning for soft articulated characters. In *Computer Graphics Forum*, 2015.

Alec Jacobson, Ilya Baran, Ladislav Kavan, Jovan Popović, and Olga Sorkine. Fast automatic skinning transformations. *ACM Transactions on Graphics (TOG)*, 31(4):1–10, 2012.

Olivier Dionne and Martin de Lasa. Geodesic voxel binding for production character meshes. In *Proceedings of the 12th ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, 2013.

Marc Alexa. Linear combination of transformations. *ACM Transactions on Graphics (TOG)*, 21 (3):380–387, 2002.

Ladislav Kavan and Jiří Žára. Spherical blend skinning: a real-time deformation of articulated models. In *Proceedings of the 2005 Symposium on Interactive 3D Graphics and Games*, 2005.

Ladislav Kavan, Steven Collins, Jiří Žára, and Carol O'Sullivan. Skinning with dual quaternions. In *Proceedings of the 2007 Symposium on Interactive 3D Graphics and Games*, 2007.

Gene S Lee, Andy Lin, Matt Schiller, Scott Peters, Mark McLaughlin, Frank Hanner, and Walt Disney Animation Studios. Enhanced dual quaternion skinning for production use. In *SIGGRAPH 2013 Talks*, 2013.

Joe Mancewicz, Matt L Derksen, Hans Rijpkema, and Cyrus A Wilson. Delta mush: smoothing deformations while preserving detail. In *Proceedings of the Fourth Symposium on Digital Production*, 2014.

Binh Huy Le and JP Lewis. Direct delta mush skinning and variants. *ACM Transactions on Graphics (TOG)*, 38(4):113–1, 2019.

Zhan Xu, Yang Zhou, Evangelos Kalogerakis, Chris Landreth, and Karan Singh. Rignet: Neural rigging for articulated characters. *ACM Transactions on Graphics (TOG)*, 39(4):1, 2020.

Xuming Ouyang and Cunguang Feng. Autoskin: Skeleton-based human skinning with deep neural networks. In *Journal of Physics: Conference Series*, 2020.

Xu Chen, Yufeng Zheng, Michael J Black, Otmar Hilliges, and Andreas Geiger. Snarf: Differentiable forward skinning for animating non-rigid neural implicit shapes. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, 2021.

John R. Taylor. *Classical Mechanics*. University Science Books, US, 2005.

Randall J LeVeque. *Finite difference methods for ordinary and partial differential equations: steady-state and time-dependent problems*. SIAM, 2007.

Matthias Müller, Bruno Heidelberger, Matthias Teschner, and Markus Gross. Meshless deformations based on shape matching. *ACM transactions On Graphics (TOG)*, 24(3):471–478, 2005.

David Baraff and Andrew Witkin. Large steps in cloth simulation. In *Proceedings of the 25th Annual Conference on Computer graphics and Interactive Techniques*, 1998.

John D Cutnell and Kenneth W Johnson. *Physics*. John Wiley & Sons, 2009.

Demetri Terzopoulos, John Platt, Alan Barr, and Kurt Fleischer. Elastically deformable models. *ACM SIGGRAPH Computer Graphics*, 21(4):205–214, 1987.

Y. Zhang, E. C. Prakash, and E. Sung. Real-time physically-based facial expression animation using mass-spring system. In *Proceedings of Computer Graphics International Conference*, 2001.

Demetri Terzopoulos and Keith Waters. Physically-based facial modeling, analysis, and animation. *Journal of Visualization and Computer Animation*, 1(2):73–80, 1990.

Yuencheng Lee, Demetri Terzopoulos, and Keith Waters. Realistic modeling for facial animation. In *Proceedings of the 22nd Annual Conference on Computer Graphics and Interactive Techniques*, 1995.

Luciana Porcher Nedel and Daniel Thalmann. Real time muscle deformations using mass-spring systems. In *Proceedings of the 25th Annual Conference on Computer Graphics and Interactive Techniques*, 1998.

Andrew Selle, Michael Lentine, and Ronald Fedkiw. A mass spring model for hair simulation. In *ACM SIGGRAPH 2008 Papers*, 2008.

Gavin SP Miller. The motion dynamics of snakes and worms. In *Proceedings of the 15th annual conference on Computer graphics and interactive techniques*, 1988.

Xiaoyuan Tu and Demetri Terzopoulos. Artificial fishes: Physics, locomotion, perception, behavior. In *Proceedings of the 21st Annual Conference on Computer Graphics and Interactive Techniques*, 1994.

Ben Kenwright, Rich Davison, and Graham Morgan. Real-time deformable soft-body simulation using distributed mass-spring approximations. In *The Third International Conference on Creative Content Technologies*, 2011.

Min Hong, Jae-Hong Jeon, Hyo-Sub Yum, and Seung-Hyun Lee. Plausible mass-spring system using parallel computing on mobile devices. *Human-centric Computing and Information Sciences*, 6(1):1–11, 2016.

Sebastian Martin, Bernhard Thomaszewski, Eitan Grinspun, and Markus Gross. Example-based elastic materials. In *ACM SIGGRAPH 2011 Papers*, 2011.

Tiantian Liu, Adam W Bargteil, James F O'Brien, and Ladislav Kavan. Fast simulation of mass-spring systems. *ACM Transactions on Graphics (TOG)*, 32(6):1–7, 2013.

Robert Bridson, Sebastian Marino, and Ronald Fedkiw. Simulation of clothing with folds and wrinkles. In *ACM SIGGRAPH 2005 Courses*, pages 3–es. Association for Computing Machinery (ACM), 2005.

Peter E Hammer, Michael S Sacks, Pedro J Del Nido, and Robert D Howe. Mass-spring model for simulation of heart valve tissue mechanical behavior. *Annals of biomedical engineering*, 39(6):1668–1679, 2011.

Matthias Müller, Bruno Heidelberger, Marcus Hennix, and John Ratcliff. Position based dynamics. *Journal of Visual Communication and Image Representation*, 18(2):109–118, 2007.

Eftychios Sifakis and Jernej Barbič. Finite element method simulation of 3d deformable solids. *Synthesis Lectures on Visual Computing: Computer Graphics, Animation, Computational Photography, and Imaging*, 1(1):1–69, 2015.

Doug L James and Dinesh K Pai. Dyrt: Dynamic response textures for real time deformation simulation with graphics hardware. In *Proceedings of the 29th Annual Conference on Computer Graphics and Interactive Techniques*, 2002.

Farid ML Amirouche. *Fundamentals of multibody dynamics: theory and applications*. Springer, 2006.

Yun Teng, Mark Meyer, Tony DeRose, and Theodore Kim. Subspace condensation: Full space adaptivity for subspace deformations. *ACM Transactions on Graphics (TOG)*, 34(4):1–9, 2015.

Tiantian Liu, Sofien Bouaziz, and Ladislav Kavan. Quasi-newton methods for real-time simulation of hyperelastic materials. *ACM Transactions on Graphics (TOG)*, 36(3):5, 2017.

Huamin Wang and Yin Yang. Descent methods for elastic body simulation on the gpu. *ACM Transactions on Graphics (TOG)*, 35(6):1–10, 2016.

Jernej Barbič, Marco da Silva, and Jovan Popović. Deformable object animation using reduced optimal control. In *ACM SIGGRAPH 2009 Papers*, 2009.

Marco Fratarcangeli, Valentina Tibaldo, and Fabio Pellacini. Vivace: A practical gauss-seidel method for stable soft body dynamics. *ACM Transactions on Graphics (TOG)*, 35(6):1–9, 2016.

Jianying Li, Yu Guo, Ping Liu, Qiong Wang, and Jing Qin. A gpu-accelerated finite element solver for simulation of soft-body deformation. In *IEEE International Conference on Information and Automation (ICIA)*, 2013.

Edvinas Danevičius, Rytis Maskeliūnas, Robertas Damaševičius, Dawid Połap, and Marcin Woźniak. A soft body physics simulator with computational offloading to the cloud. *Information*, 9(12):318, 2018.

Thomas W Sederberg and Scott R Parry. Free-form deformation of solid geometric models. In *Proceedings of the 13th Annual Conference on Computer Graphics and Interactive Techniques*, 1986.

Hongyi Xu and Jernej Barbič. Pose-space subspace dynamics. In *ACM Transactions on Graphics (TOG)*, 2016.

Jan Bender, Matthias Müller, and Miles Macklin. Position-based simulation methods in computer graphics. In *Eurographics Tutorials)*, 2015.

Junjun Pan, Leiyu Zhang, Peng Yu, Yang Shen, Haipeng Wang, Haimin Hao, and Hong Qin. Real-time vr simulation of laparoscopic cholecystectomy based on parallel position-based dynamics in gpu. In *2020 IEEE Conference on Virtual Reality and 3D User Interfaces (VR)*, 2020.

Fei Liu, Zihan Li, Yunhai Han, Jingpei Lu, Florian Richter, and Michael C Yip. Real-to-sim registration of deformable soft tissue with position-based dynamics for surgical robot autonomy. In *2021 IEEE International Conference on Robotics and Automation (ICRA)*, 2021.

Erwin Coumans and Yunfei Bai. Pybullet, a python module for physics simulation for games, robotics and machine learning. http://pybullet.org, 2016–2021.

Miles Macklin and Matthias Müller. Position based fluids. *ACM Transactions on Graphics (TOG)*, 32(4):1–12, 2013.

Jan Bender, Raphael Diziol, and Daniel Bayer. Simulating inextensible cloth using locking-free triangle meshes. In *VRIPHYS: Virtual Reality Interactions and Physical Simulations*, 2011.

Jan Bender, Matthias Müller, Miguel A Otaduy, Matthias Teschner, and Miles Macklin. A survey on position-based simulation methods in computer graphics. In *Computer Graphics Forum*, 2014.

Miles Macklin, Kier Storey, Michelle Lu, Pierre Terdiman, Nuttapong Chentanez, Stefan Jeschke, and Matthias Müller. Small steps in physics simulation. In *Proceedings of the 18th Annual ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, 2019.

Yijing Li, Hongyi Xu, and Jernej Barbič. Enriching triangle mesh animations with physically based simulation. *IEEE transactions on Visualization and Computer graphics*, 23(10):2301–2313, 2016.

Damien Rohmer, Marco Tarini, Niranjan Kalyanasundaram, Faezeh Moshfeghifar, Marie-Paule Cani, and Victor Zordan. Velocity skinning for real-time stylized skeletal animation. In *Computer Graphics Forum*, 2021.

Niranjan Kalyanasundaram, Damien Rohmer, and Victor Zordan. Acceleration skinning: Kinematics-driven cartoon effects for articulated characters. In *Graphics Interface*, 2022.

Fabian Hahn, Bernhard Thomaszewski, Stelian Coros, Robert W Sumner, and Markus Gross. Efficient simulation of secondary motion in rig-space. In *Proceedings of the 12th ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, 2013.

Martin Servin, Claude Lacoursiere, and Niklas Melin. Interactive simulation of elastic deformable materials. In *SIGRAD 2006 Conference Proceedings*, 2006.

Andrew Witkin. Physically based modeling: principles and practice constrained dynamics. *Computer Graphics Forum*, 9(1):27, 1997.

Stephen Boyd, Stephen P Boyd, and Lieven Vandenberghe. *Convex optimization*. Cambridge university press, 2004.

Richard L Burden, J Douglas Faires, and Annette M Burden. *Numerical analysis*. Cengage learning, 2015.

Miles Macklin, Matthias Müller, Nuttapong Chentanez, and Tae-Yong Kim. Unified particle physics for real-time applications. *ACM Transactions on Graphics (TOG)*, 33(4):1–12, 2014.

Dimitris N Metaxas. *Physics-based deformable models: applications to computer vision, graphics and medical imaging*. Springer Science & Business Media, 2012.

David H Allen. *Introduction to the mechanics of deformable solids: bars and beams*. Springer Science & Business Media, 2012.

Jan Bender, Matthias Müller, and Miles Macklin. A survey on position based dynamics, 2017. In *Proceedings of the European Association for Computer Graphics: Tutorials*, 2017.

Carol O'Sullivan, John Dingliana, Thanh Giang, and Mary K Kaiser. Evaluating the visual fidelity of physically based animations. *ACM Transactions on Graphics (TOG)*, 22(3):527–536, 2003.

Maria-Elena Algorri and Francis Schmitt. Mesh simplification. In *Computer Graphics Forum*, 1996.

Paolo Cignoni, Claudio Montani, and Roberto Scopigno. A comparison of mesh simplification algorithms. *Computers & Graphics*, 22(1):37–54, 1998.

Hugues Hoppe. Progressive meshes. In *Proceedings of the 23rd Annual Conference on Computer Graphics and Interactive Techniques*, pages 99–108, 1996.

Si Hang. Tetgen, a delaunay-based quality tetrahedral mesh generator. *ACM Transactions on Mathematical Softwares*, 41(2):11, 2015.

Ron Fedkiw. Lecture notes 13 in interactive computer graphics: Skinning, 2018. https://web.stanford.edu/class/cs248/pdf/class_13_skinning.pdf, Last accessed on 2023-02-28.

Alec Jacobson, Ilya Baran, Jovan Popovic, and Olga Sorkine. Bounded biharmonic weights for real-time deformation. *ACM Transaction on Graphics (TOG)*, 30(4):78, 2011.

Sheldon Axler, Paul Bourdon, and Ramey Wade. *Harmonic function theory*, volume 137. Springer Science & Business Media, 2013.

Nicolas Bonneel, David Coeurjolly, Julie Digne, and Nicolas Mellado. Code replicability in computer graphics. *ACM Transactions on Graphics (TOG)*, 39(4):93–1, 2020.

Anubha Kalra, Andrew Lowe, and AM Al-Jumaily. Mechanical behaviour of skin: a review. *Journal of Material Sciences and Engineering*, 5(4):1000254, 2016.

Eric J Chen, Jan Novakofski, W Kenneth Jenkins, and William D O'Brien. Young's modulus measurements of soft tissues with application to elasticity imaging. *IEEE Transactions on Ultrasonics, Ferroelectrics, and Frequency control*, 43(1):191–194, 1996.

Richard H Byrd, Peihuang Lu, Jorge Nocedal, and Ciyou Zhu. A limited memory algorithm for bound constrained optimization. *SIAM Journal on Scientific Computing*, 16(5):1190–1208, 1995.

Rebecca Chamberlain, Jennifer E Drake, Aaron Kozbelt, Rachel Hickman, Joseph Siev, and Johan Wagemans. Artists as experts in visual cognition: An update. *Psychology of Aesthetics, Creativity, and the Arts*, 13(1):58, 2019.

Steven J Ferris, Rob A Kempton, Ian J Deary, Elizabeth J Austin, and Margaret V Shorter. Carryover bias in visual assessment. *Perception*, 30(11):1363–1373, 2001.

Harry N Boone, Deborah A Boone, et al. Analyzing likert data. *Journal of Extension*, 50(2):1–5, 2012.

Gail M Sullivan and Anthony R Artino Jr. Analyzing and interpreting data from likert-type scales. *Journal of Graduate Medical Education*, 5(4):541–542, 2013.

Gerhard A Holzapfel et al. Biomechanics of soft tissue. *Computational Biomechanics*, 3(1): 1049–1063, 2001.

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

The University of Cambridge. Property information: Young's modulus and specific stiffness, 2023. http://www-materials.eng.cam.ac.uk/mpsite/properties/non-IE/stiffness.html, Last accessed on 2023-02-24.

Ronen Barzel, John R Hughes, and Daniel N Wood. Plausible motion simulation for computer graphics animation. In *Proceedings of the 1996 Eurographics Workshop*, 1996.

Walter Gautschi. *Numerical analysis*. Springer Science & Business Media, 2011.

Dimitar Dinev, Tiantian Liu, Jing Li, Bernhard Thomaszewski, and Ladislav Kavan. Fepr: Fast energy projection for real-time simulation of deformable objects. *ACM Transactions on Graphics (TOG)*, 37(4):1–12, 2018.

Ozan Cetinaslan and Rafael Oliveira Chaves. Energy embedded gauss-seidel iteration for soft body simulations. In *2019 32nd SIBGRAPI Conference on Graphics, Patterns and Images (SIBGRAPI)*, 2019.

Fabian Löschner, Andreas Longva, Stefan Jeske, Tassilo Kugelstadt, and Jan Bender. Higher-order time integration for deformable solids. In *Computer Graphics Forum*, 2020.

Junjun Pan, Lijuan Chen, Yuhan Yang, and Hong Qin. Automatic skinning and weight retargeting of articulated characters using extended position-based dynamics. *The Visual Computer*, 34 (10):1285–1297, 2018.

Rodolphe Vaillant, Loïc Barthe, Gaël Guennebaud, Marie-Paule Cani, Damien Rohmer, Brian Wyvill, Olivier Gourmel, and Mathias Paulin. Implicit skinning: Real-time skin deformation with contact modeling. *ACM Transactions on Graphics (TOG)*, 32(4):1–12, 2013.