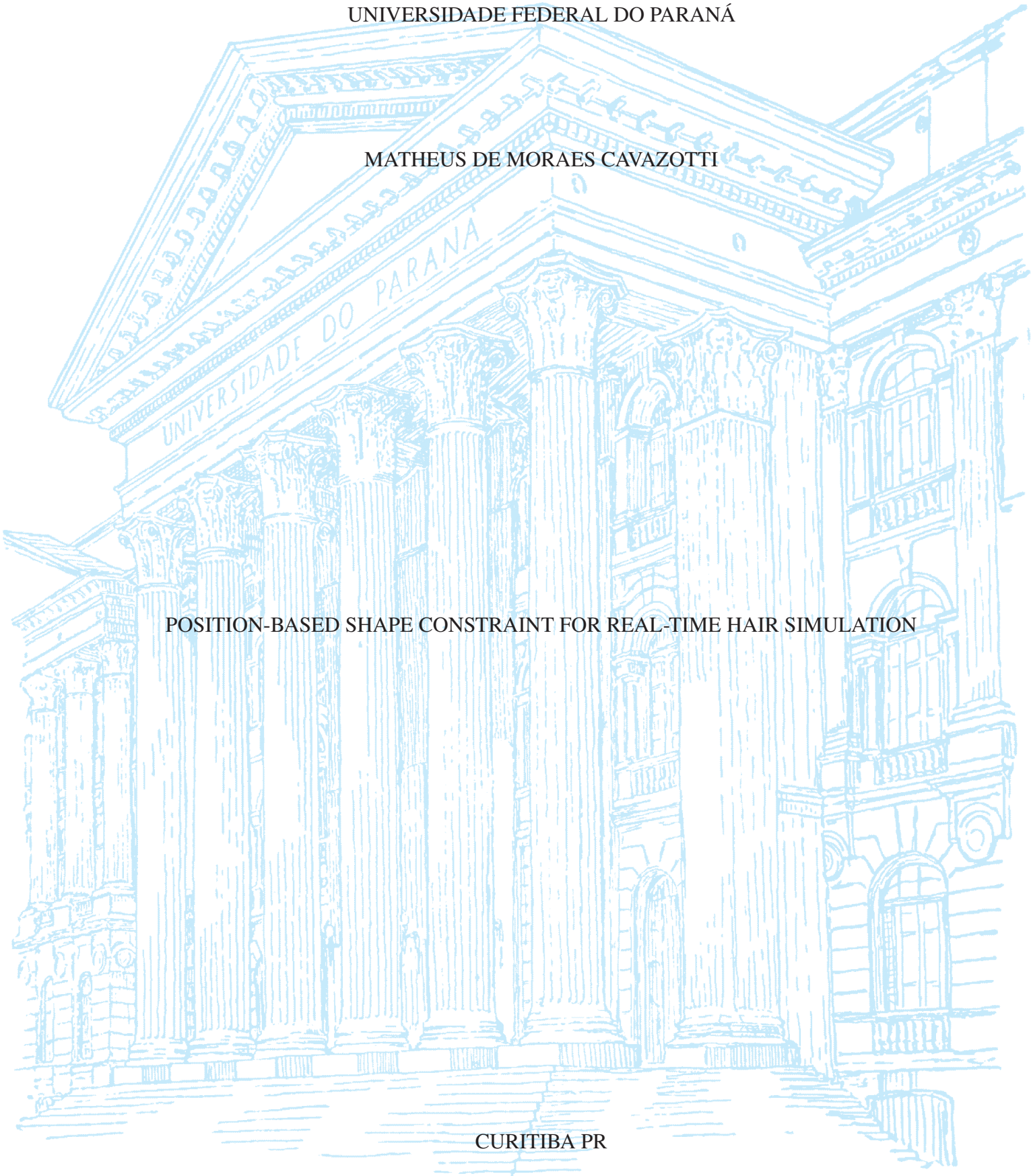
UNIVERSIDADE FEDERAL DO PARANÁ

MATHEUS DE MORAES CAVAZOTTI



POSITION-BASED SHAPE CONSTRAINT FOR REAL-TIME HAIR SIMULATION

CURITIBA PR

2025

MATHEUS DE MORAES CAVAZOTTI

POSITION-BASED SHAPE CONSTRAINT FOR REAL-TIME HAIR SIMULATION

Dissertação apresentada como requisito parcial à obtenção do grau de Mestre em Informática no Programa de Pós-Graduação em Informática, Setor de Ciências Exatas, da Universidade Federal do Paraná.

Área de concentração: *Computação*.

Orientador: André Luiz Pires Guedes.

CURITIBA PR

2025

DADOS INTERNACIONAIS DE CATALOGAÇÃO NA PUBLICAÇÃO (CIP)
UNIVERSIDADE FEDERAL DO PARANÁ
SISTEMA DE BIBLIOTECAS – BIBLIOTECA CIÊNCIA E TECNOLOGIA

Cavazotti, Matheus de Moraes
Position-based shape constraint for real-time hair simulation. /Matheus de Moraes Cavazotti. – Curitiba, 2025
1 recurso on-line : PDF.

Dissertação (Mestrado) - Universidade Federal do Paraná, Setor de Ciências Exatas, Programa de Pós-graduação em Informática.

Orientador: André Luiz Pires Guedes.

1. Cabelo. 2. Computação gráfica. 3. Física. I. Universidade Federal do Paraná. II. Programa de Pós-Graduação em Informática. III. Guedes, André Luiz Pires. IV. Título.

Bibliotecária: Roseny Rivelini Morciani CRB-9/1585

TERMO DE APROVAÇÃO

Os membros da Banca Examinadora designada pelo Colegiado do Programa de Pós-Graduação INFORMÁTICA da Universidade Federal do Paraná foram convocados para realizar a arguição da Dissertação de Mestrado de **MATHEUS DE MORAES CAVAZOTTI**, intitulada: **POSITION-BASED SHAPE CONSTRAINT FOR REAL-TIME HAIR SIMULATION**, sob orientação do Prof. Dr. ANDRÉ LUIZ PIRES GUEDES, que após terem inquirido o aluno e realizada a avaliação do trabalho, são de parecer pela sua APROVAÇÃO no rito de defesa.

A outorga do título de mestre está sujeita à homologação pelo colegiado, ao atendimento de todas as indicações e correções solicitadas pela banca e ao pleno atendimento das demandas regimentais do Programa de Pós-Graduação.

CURITIBA, 13 de Junho de 2025.

Assinatura Eletrônica
14/06/2025 22:29:51.0
ANDRÉ LUIZ PIRES GUEDES
Presidente da Banca Examinadora

Assinatura Eletrônica
23/06/2025 14:00:14.0
FABIANO SILVA
Avaliador Interno (UNIVERSIDADE FEDERAL DO PARANÁ)

Assinatura Eletrônica
15/06/2025 13:42:26.0
CLÁUDIO ESPERANÇA
Avaliador Externo (UNIVERSIDADE FEDERAL DO RIO DE JANEIRO - UFRJ)

Às minhas meninas, Bruna e Thalia

ACKNOWLEDGEMENTS

I would like to acknowledge and thank the support I've received from my wife Bruna. I also want to acknowledge my parents for all they've taught me throughout the years and Professor Guedes for his guidance and support.

RESUMO

Neste trabalho apresentamos publicações relevantes no campo de simulação de cabelos, com foco em simulações em tempo real, no contexto de Computação Gráfica. Também explicamos o *framework* de simulação *PBD* e mostramos como foi usado para simular cabelos. O ponto central de nosso trabalho é a formulação de uma *constraint* que permite manter a forma dos fios de cabelo durante a simulação. Apresentamos, também, os resultados quantitativos e qualitativos de nossa contribuição.

Palavras-chave: Física em tempo real, Simulação de cabelos, Position-based dynamics

ABSTRACT

In this work we present relevant publications in the field of hair simulation, with focus on real-time simulations, in the context of Computer Graphics. We also explain the simulation framework PBD and show how it was used to simulate hair. The central point of our work is the formulation of a constraint that allows the hair strands to keep their shape during the simulation. We also present the quantitative and qualitative results of our contribution.

Keywords: Real-time physics, Hair simulation, Position-based dynamics

LIST OF FIGURES

LIST OF ACRONYMS

API	Application Programming Interface
CPU	Central Processing Unit
DFTL	Dynamic Follow-The-Leader
DINF	Departamento de Informática
FPS	Frames per Second
GLSL	OpenGL Shading Language
GPU	Graphics Processing Unit
HLSL	High Level Shading Language
JIT	Just In Time
NURBS	Non-Uniform Rational B-Splines
ODE	Ordinary Differential Equation
PBD	Position-Based Dynamics
PPGINF	Programa de Pós-Graduação em Informática
SDF	Signed Distance Field
SIMD	Single Instruction, Multiple Data
SPH	Smoothed Particle Hydrodynamics
UFPR	Universidade Federal do Paraná
VR	Virtual Reality
XPBD	Extended Position-Based Dynamics

CONTENTS

1 INTRODUCTION

This work is fruit of the curiosity to understand more deeply how real-time hair simulation works. What began as a personal project eventually evolved into this Masters dissertation, and as we studied the topic, the relevance of our research became even more apparent.

Whenever there is a human character or some other fur-covered creature portrayed in a computer generated image, there is the need to represent visually what hair looks like. That's not a trivial task, since there are between 100.000 and 120.000 strands of hair in the human head, with each hair strand measuring between $40\mu m$ and $120\mu m$ in diameter (Rosenblum et al., 1991). The challenge is even bigger if we need not just a single image, but a sequence of images in form of an animation, because now we need to emulate the motion of hair, which can be influenced by many things, such as hair length, water and humidity, wind, and collision among the hair strands and other objects. Add to that the time budget available to real-time and interactive applications, which is at most $33ms$ per frame (Akenine-Möller et al., 2018), and we've got in our hands a pretty interesting challenge.

How realistic should be the motion of the hair, or how much artistic freedom should one have when styling a character's hair, depends much on the application for which it's being used. For instance, in animation feature movies such as Tangled (Walt Disney Animation Studios, 2010) and Brave (Walt Disney Animation Studios, 2012), hairstyle and movement are important aspects of storytelling (see Figure 1.1). On the other hand, there are several applications of Computer Graphics outside the entertainment industry that would benefit from enhanced realism that comes from natural motion of hair.

The use of virtual reality is ever more common in education and professional training (Renganayagalu et al., 2021), as well as in therapy and other health-related activities (for example, see Hoffman et al. (2020), Smith et al. (2020) and Emmelkamp and Meyerbröker (2021)). Many of the works cited in this paragraph mentions that one of the shortcomings of VR in therapy and education is the poor feeling of social presence and place illusion caused by technical limitations. Zibrek and McDonnell (2019) showed evidence that photorealistic characters can improve such feelings. Therefore, techniques that enable simulation of natural hair motion in VR applications can improve the efficacy of education and health tools that use that medium.

Many works have been published in the last three decades about techniques to simulate hair movement and the many ways it can interact with the environment. Those techniques vary in levels of realism, artistic freedom and compute efficiency. Our work will focus on approaches

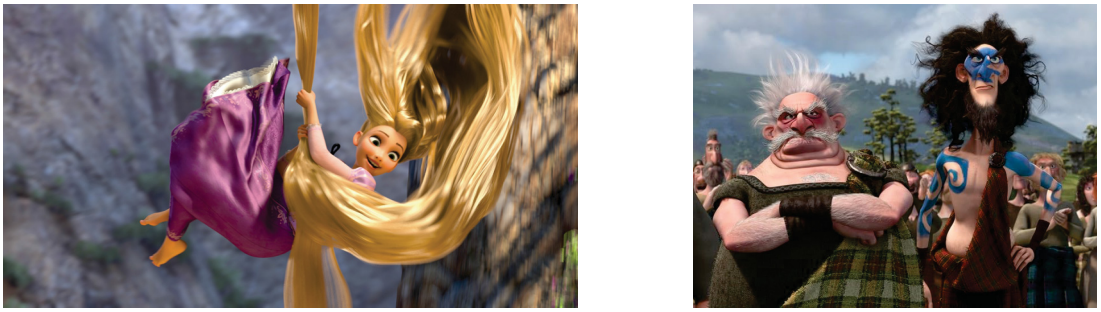


Figure 1.1: Examples of hair as part of storytelling. *On the left*: Rapunzel from the movie Tangled. *On the right*: Lord Dingwall and Lord Macintosh from the movie Brave

based on Position-Based Dynamics (PBD) (Müller et al., 2007) and will present an improvement to an existing approach, so that hair can keep its curls and hairstyle.

This work has the following structure: in Chapter 2 we will present some notation conventions and fundamental concepts of Computer Graphics and real-time simulations. In Chapter 3 we'll discuss some of the most relevant works related to hair simulation, with focus on approaches that are suited for real-time and interactive applications. Chapter 4 will expose in detail what is Position-Based Dynamics and how it has been used to simulate hair. In Chapter 5 we will present our contribution: a new way to maintain hair shape using PBD. Implementation details will be discussed in Chapter 6. In Chapter 7 we will outline and evaluate our results. Finally, in Chapter 8 we will present the conclusion of our work, reviewing our contribution, its results and opportunities for future studies.

2 NOTATIONS AND CONCEPTS

Before we can start discussing hair simulation and the many techniques developed to solve the challenges related to it, we will present some notation conventions that we'll use throughout this text, as well as some core concepts in Computer Graphics and real-time simulation that are necessary for the reader to follow this dissertation.

2.1 NOTATION CONVENTIONS

A lowercase roman letter or Greek letter in italics (for example: a,x, λ, etc.) represents a **scalar value**, with exception of Greek letters that have special meaning in Mathematics, such as Δ, which denotes variation or change, ∇, which represents the gradient of a function, and ∇^2, commonly used to represent the Lagrangian multiplier. A lowercase roman letter or Greek letter in italics with a hat represents an **angle** (*e.g.* $\hat{a}$, $\hat{\beta}$).

A lowercase roman letter in boldface (for example: $\mathbf{v}$) represents a **vector**. In this text, a vector is usually a 3D vector (in other words, it is defined in $\mathbb{R}^3$), unless stated the contrary. Vectors can be interpreted either as directions or positions in 3D space. $\mathbf{v}.x$, $\mathbf{v}.y$ and $\mathbf{v}.z$ represents the first, second and third components of such vector. An uppercase roman letter in boldface (*e.g.* $\mathbf{M}$) represents a **matrix**. Any other type of diacritics besides the one defined above is used only as means of distinction (*e.g.* $\tilde{\mathbf{x}}$).

A subscript in the right-hand side (m_i) is used for indexing, whereas a right-hand side superscript ($\mathbf{v}^{i+1}$) denotes different time-steps or iterations in the simulation. A superscript prime or star on the right ($\mathbf{x}'$, $\mathbf{x}^*$) or a left-hand side superscript might be used *ad hoc*.

2.2 DEFINITIONS AND CONCEPTS

Now we will present definitions and assumptions for some terms, which might be different from other fields. We'll also explain some fundamental concepts in Computer Graphics.

2.2.1 Real-time

In the book "Real-Time Rendering, fourth edition", Akenine-Möller et al. (2018) wrote:

> Real-time rendering is concerned with rapidly making images on the computer. It is the most highly interactive area of computer graphics. An image appears on the screen, the viewer acts or reacts, and this feedback affects what is generated next. This cycle of reaction and rendering happens at a rapid enough rate that the viewer does not see individual images, but rather becomes immersed in a dynamic process.

While this quote specifically addresses rendering, simulations of any kind are part of the feedback loop described above. The authors continue by stating that at 6 FPS there starts to be a sense of interactivity. 24 FPS is the rate at which projectors show frames. Video games aim for at least 30 FPS, and VR applications need at least 90 FPS to not cause discomfort to users.

If we consider that lowest FPS rate acceptable for a game or other interactive application is 30 FPS, it gives us at most $30ms$ to process input, run simulations for next frame, apply updates in the scene and render the result.

2.2.2 Numerical integration of differential equations

Complex physical phenomena can be described by differential equations. It's not rare for such equations to not have an exact analytical solution, so the alternative we have is to solve them using a **numerical integration method**, which is a way to approximate the result by iteratively converging towards the exact solution.

How fast we can reach a result within an acceptable error margin or how resilient to approximation errors our result can be depends on the method used. For instance, let's consider that $\mathbf{Y}(t)$ describes the state of a physical system at moment t and that we want to compute the state of the system in a later moment $\mathbf{Y}(t + \Delta t)$. An **explicit** integration method would use the previous state to compute the next:

$$\mathbf{f}(\mathbf{Y}(t)) = \mathbf{Y}(t + \Delta t).$$

On the other hand, an **implicit** method would solve an equation using both the previous and the next state:

$$\mathbf{g}(\mathbf{Y}(t),\ \mathbf{Y}(t + \Delta t)) = 0.$$

Implicit methods usually require more computation to solve a time step, however, they don't suffer as much from numerical stiffness as explicit methods, which requires impractically small time steps Δt to maintain error bounded. There are other types of integration methods and more nuances to each one of those types, but a complete analysis of them goes beyond the scope of this work.

2.2.3 Simulation

Also known as *physics-based animation*. One usually resorts physics simulation in Computer Graphics when the desired motion or deformation is too complex to be done manually, such as the motion of a fluid, the shattering of an object or the deformation that happens with cloth.

While in Engineering and Physics precision is paramount, in Computer Graphics some inaccuracy in the solution is tolerable as long as it *looks* correct, it is efficient to compute and it is stable. Marschner and Shirley (2018) explain what a stable solution is in the book "Fundamentals of Computer Graphics, fourth edition":

> *Stability* of a difference scheme [also known as "integration method"] is related to how fast numerical errors, which are always present in practice, can grow with time. For stable schemes this growth is bounded, while for unstable ones it is exponential and can quickly overwhelm the solution one seeks [...]. It is important to realize that while some inaccuracy in the solution is tolerable [...], an unstable result is completely meaningless, and one should avoid using unstable schemes.

2.2.4 Polygonal Mesh

There are many ways to represent an object in three dimensional space. One could use *signed distance fields* (SDFs), curves, polygonal meshes, among others (Akenine-Möller et al., 2018). Polygonal meshes are one of the most common methods used in games and movies due to its versatility.

In this representation, the surface of the object is discretized by **vertices**, which are points in 3D space. Those points are connected by **edges**, forming polygons called **faces**. Faces

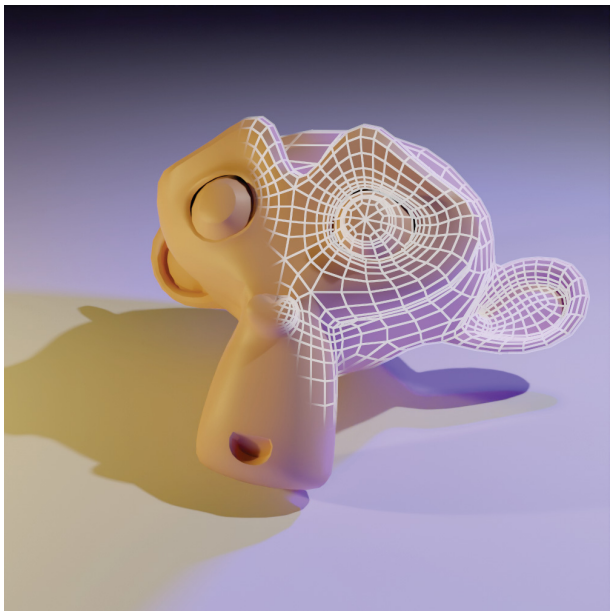


Figure 2.1: A 3D monkey head represented by a polygonal mesh

are usually triangles, but they can have any arbitrary shape. One vertex can be part of many faces, resulting in a **mesh**, hence the name (see Figure 2.1).

While we're in topic of surface representation, it might be useful to expand our mention on curves. Instead of using vertices and edges to form faces and create a mesh that discretizes a surface, we can have a sequence of control points that produces a curve. By expanding this idea to a mesh of control points, one can interpolate a surface that is continuous. How the control points will be used to calculate the curve and the surface depends on the type of schema used. For instance, the most common types of curves in Computer Graphics are: **Bézier Curves**, **Quadratic** and **Cubic B-Splines** and **Non-Uniform Rational B-Splines (NURBS)**. The reader may consult Salomon (2007) for more details about each type of curve.

2.2.5 Affine Transformations and Homogeneous Coordinates

Linear algebra gives us the tooling to place an object in a scene. By multiplying each vertex of the object's mesh by a 3×3 transformation matrix, we can scale and rotate the object. However, as noted by Marschner and Shirley (2018), we cannot translate an object using only a 3×3 matrix.

The standard way of implementing affine transformations (rotation, scaling, translation, shearing, etc.) is by using a 4×4 transformation matrix and **homogeneous coordinates**. Suppose we have a 3×3 transformation matrix $\mathbf{M}$ that represents some rotation and scaling we want to perform in a vertex $\mathbf{v}$:

$$\mathbf{M}\mathbf{v} = \mathbf{v}' \tag{2.1}$$

Now, if we also want to translate the same vertex by

$$\mathbf{t} = \begin{bmatrix} \Delta x \\ \Delta y \\ \Delta z \end{bmatrix}$$

we would need to sum both vectors:

$$\mathbf{v} + \mathbf{t} = \mathbf{v}'' \tag{2.2}$$

The way of doing both transformations in one single operation is to create a new transformation matrix $\mathbf{M}' \in \mathbb{R}^{4\times4}$ with the elements of $\mathbf{M}$ plus the elements of $\mathbf{t}$ (Equation 2.3) and convert $\mathbf{v}$ to its homogeneous form $\mathbf{v}^* \in \mathbb{R}^4$ (Equation 2.4). To convert a vector in $\mathbb{R}^3$ to its homogeneous form, we simply set the last element of the homogeneous vector to 0 if the vector represents a direction (not affected by translation), or to 1 if the vector represents a position (affected by translation).

$$\mathbf{M}' = \begin{bmatrix} m_{11} & m_{12} & m_{13} & \Delta x \\ m_{21} & m_{22} & m_{23} & \Delta y \\ m_{31} & m_{32} & m_{33} & \Delta z \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{2.3}$$

$$\mathbf{v}^* = \begin{bmatrix} \mathbf{v}.x \\ \mathbf{v}.y \\ \mathbf{v}.z \\ 1 \end{bmatrix} \tag{2.4}$$

We can understand better why it works by expanding and manipulating the equation $\mathbf{M}'\mathbf{v}^* = \mathbf{v}'''$:

$$\mathbf{M}'\mathbf{v}^* = \mathbf{v}''' \tag{2.5}$$

$$\begin{bmatrix} m_{11} & m_{12} & m_{13} & \Delta x \\ m_{21} & m_{22} & m_{23} & \Delta y \\ m_{31} & m_{32} & m_{33} & \Delta z \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} \mathbf{v}.x \\ \mathbf{v}.y \\ \mathbf{v}.z \\ 1 \end{bmatrix} = \begin{bmatrix} m_{11}\mathbf{v}.x + m_{12}\mathbf{v}.y + m_{13}\mathbf{v}.z + \Delta x \\ m_{21}\mathbf{v}.x + m_{22}\mathbf{v}.y + m_{23}\mathbf{v}.z + \Delta y \\ m_{31}\mathbf{v}.x + m_{32}\mathbf{v}.y + m_{33}\mathbf{v}.z + \Delta z \\ 1 \end{bmatrix} \tag{2.6}$$

$$= \begin{bmatrix} \mathbf{v}'.x + \Delta x \\ \mathbf{v}'.y + \Delta y \\ \mathbf{v}'.z + \Delta z \\ 1 \end{bmatrix} \qquad \text{(from Eq. 2.1)} \tag{2.7}$$

2.2.6 Parenting 3D Objects

It's extremely common in video games and animations to have the transformation of one object being dependent of the transformation of another. For example, if there is a character holding a sword, the sword's transformation depends, among other things, on the character's transformation. In this example, it's said that the sword is a **child** of the character and that character is the sword's **parent**.

Now armed with transformation matrices and homogeneous coordinates, we can express this dependency relationship by having the children's transformation matrices being relative to their parents. To place an object in a scene, we must transverse this parent-child relation graph, also called **scene graph** (Marschner and Shirley, 2018), multiplying the transformation matrices along the way.

To make it clearer, suppose we have a scene with a camera, a character with a hat and a tree. Figure 2.2 illustrates the scene graph for such scene. We can know where to place the hat's vertices in the scene with this expression $\mathbf{M}_{character}\mathbf{M}_{hat}\mathbf{v}$.

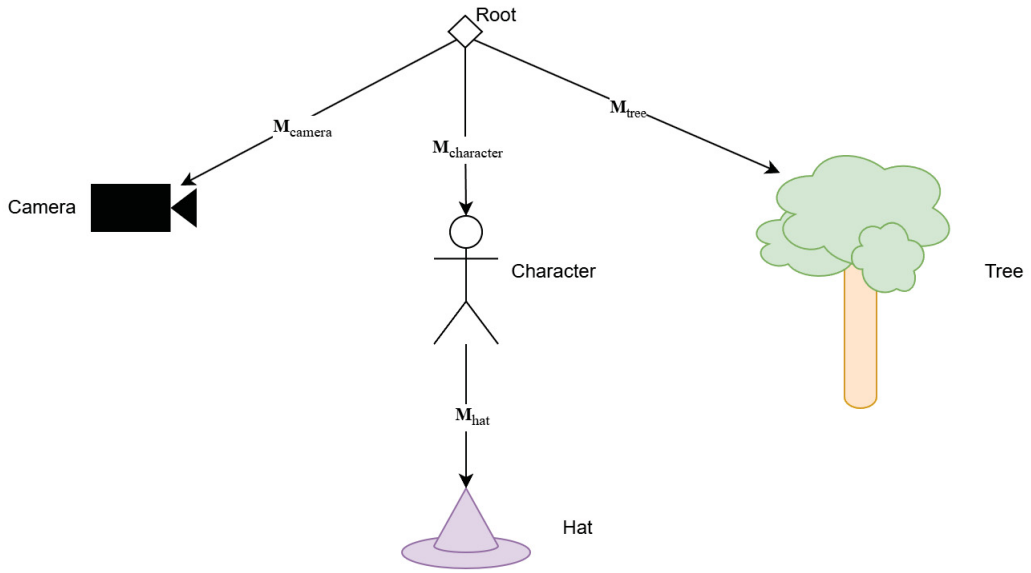


Figure 2.2: Scene graph example

2.2.7 Particles

In Computer Graphics, a particle is a point in 3D space with one or more attributes, such as mass, velocity, age, etc. Many of the simulation techniques that we'll discuss in this work use some type of particles. It's also interesting to notice that the vertices of a mesh can be treated as particles during a simulation.

2.2.8 Graphics Pipeline and Shaders

There are many graphics APIs that make the work of rendering objects a little easier. The most popular of them are OpenGL and its successor Vulkan (both cross-platform), DirectX (Windows specific) and Metal (Apple specific). All these APIs define graphics pipelines, which can be defined as "special software/hardware subsystem that efficiently draws 3D primitives in perspective" (Marschner and Shirley, 2018). The graphics pipeline perform a series of operations, starting with some data (such as vertices positions) and ending by updating the pixels in an image, usually displayed in a screen. These APIs make graphics programming a somewhat easier by abstracting away some hardware-specific aspects and leveraging GPU features such as triangle intersection and triangle rasterization under the hood.

While the main structure of the pipeline is fixed, there are some parts that can be customized with **shaders**, which are simply programs written in a programming language (such as GLSL or HLSL) that will be compiled for the GPU. There are also shaders that can be executed outside a graphics pipeline and don't have a predefined role, in other words, they don't need to fit in a pipeline with predefined inputs and outputs. That type of shader is called **compute shader**.

2.3 CONCLUSION

With our notation conventions defined and the fundamental concepts outlined, we are ready to discuss previous works related to real-time hair simulation (Chaps. 3-4) and to explain our contribution to this field (Chaps. 5-7).

3 RELATED WORKS

One of the first published works about hair simulation was written more than thirty years ago (Rosenblum et al., 1991). It described not only how to simulate hair but also how to model and render it (those two topics evolved now to become research branches of their own). Although the simulation procedures described there are simple and, to a certain degree, inefficient, they laid the foundation to almost all the works that came later, either by people looking for ways to improve it or by people that were looking for alternative approaches.

In this chapter we will discuss the aforementioned work, as well as other works that might offer a panorama of the field of hair simulation, with an emphasis, as expected, in real-time techniques. However it will not be a comprehensive survey on the field because that is not the goal of this dissertation.

Unfortunately there seems to be a lack of systematic reviews about hair simulation: besides the brief review of the state of the art present in research papers, the only in-depth review of the topic that we know of was published almost twenty years ago (Ward et al., 2007). A more recent survey focused on hair modeling (Yang, 2024) cites briefly some works about hair simulation.

We can separate the works on hair simulation in two main groups: the ones that present techniques to simulate individual hair strands and the ones that treat hair collectively. There are also some works that fall in between these two groups. We could say that all of them could be placed on a scale where, on one end, we have the finest detail at a high computational cost and, on the other, the coarsest detail with the highest efficiency. This idea is illustrated in Figure 3.1.

From our studies, we identified that every work followed one of two trends: either advance what "the finest" and most physical-accurate details possible are; or optimize performance, realism and artistic freedom.

As examples of works that follow the first trend, we mention Bertails et al. (2006), which developed one of the first approaches to realistically simulate curls and strand torsion, and Fei et al. (2017), which presented a novel technique to simulate the interactions of hair and liquids (see Figure 3.2).

On the other hand, Yuksel et al. (2009) created a technique that allows artists to create arbitrarily complex hairstyles and perform basic real-time simulations with it. Later, Wu and Yuksel (2016) improved the simulation capabilities of such technique. Another example is the

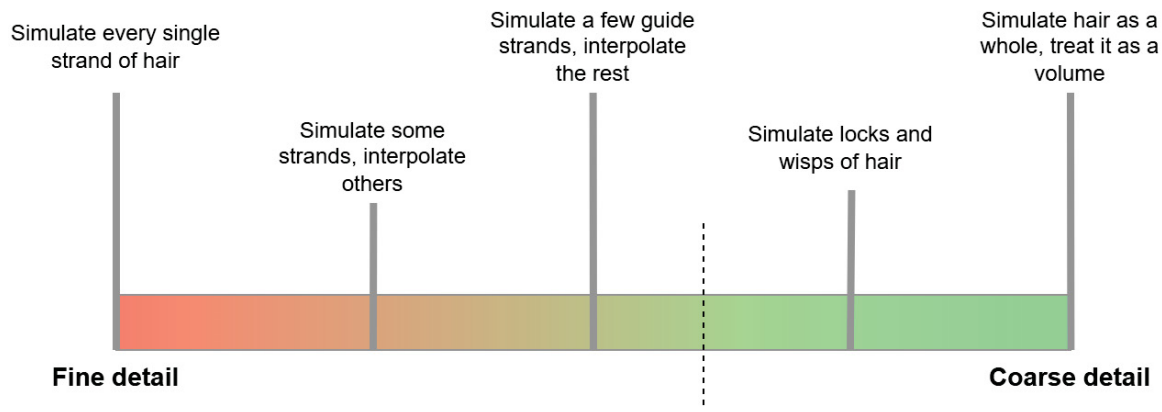


Figure 3.1: Hair simulation techniques in a scale of detail and computational cost

Figure 3.2: The clumping effect of wet hair (Fei et al., 2017)

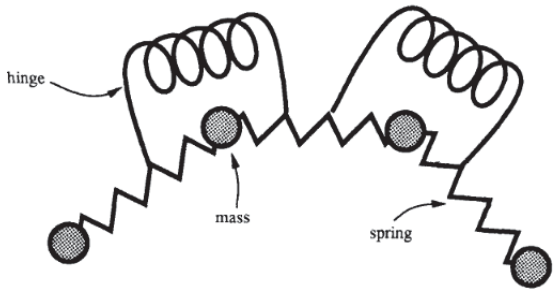


Figure 3.3: Mass-spring scheme by Rosenblum et al. (1991)

work by Jiang et al. (2020), which introduces a new hybrid integration method that achieves realistic simulations in real-time.

Now we'll dive deeper in each one of the two groups mentioned at the beginning of this chapter, describing in more details the works presented. We will also try to place them in the scale represented in Figure 3.1, as well as identify which trend each work seems to follow.

3.1 STRAND-BASED SIMULATION

In this section, we'll present works that describe techniques and frameworks that simulate individual hair strands. Some of those works simulate every single hair strand, while others simulate a small number of them, which are usually called **guide strands**, and interpolate the rest from the guide strands.

Rosenblum et al. (1991) presented a technique that sought to simulate every strand of hair, placing it in the left-most end of the scale in Figure 3.1. They represented the hair strands as chains of particles interconnected by springs, which is an instance of a mass-spring system (Fig. 3.3). Their simulation method is simple to implement and was a pioneer in the field, however, it suffers from numerical instability if the time step is not small enough. That happens because the authors used forward Euler, an explicit numerical method, to integrate particles' positions. This limitation is intensified by the fact that the springs connecting the particles must be very stiff to prevent shrinking and stretching of the hair strands, therefore requiring even smaller time steps.

One technique that pushed the left end of the scale further and that also uses the mass-spring was published by Selle et al. (2008). In that work, the authors managed to simulate thousands of hair strands using a new spring layout (Fig. 3.4) and a new semi-implicit integration method (in opposition to forward Euler's explicit method). Their technique is able to simulate

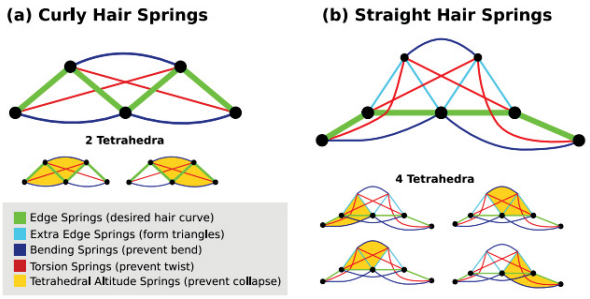


Figure 3.4: Mass-spring scheme by Selle et al. (2008)

Figure 3.5: Hair simulation by Jiang et al. (2020)

realistically torsion, bending and inextensibility characteristic of hair strands, however it took the authors several minutes to simulate a single frame. It is interesting to notice that their simulations ran on a multi-core CPU, therefore, porting their implementation to GPU could allow this technique to run in real-time.

Naturally, the next step for mass-spring hair simulations would be to simulate a full head of hair in real-time. Jiang et al. (2020) achieved this feat by innovating in the integration method. They proposed a framework that produces realistic hair dynamics with self-collisions and other volumetric phenomena (see Fig. 3.5). The authors linearized the implicit Euler method to integrate position of the particles and used a Eulerian/Lagrangian hybrid to handle self-interactions. All of that was done in a way that it is efficient to be computed with the parallel architecture of GPUs.

Still in the topic of mass-spring simulations, but now going towards the middle of the scale in Figure 3.1, we have a work (Chai et al., 2014) that presents a data-driven technique to achieve detailed hair simulation in real-time. In contrast with the works previously described, this one does not attempt to simulate every single strand of hair in real-time. It rather simulates only a few guide strands and then interpolates the rest using a a machine learning model. The guide strands are selected by an algorithm that finds which strands would be the best guides, and the model is trained on simulation data generated beforehand. This framework was later improved by Chai et al. (2016) to handle interpolation of hair strands in scenarios with complex collisions and interactions (see Figure 3.6).

Mass-spring systems are probably the most used representation for hair strands in hair simulations. However, there are other approaches that are worth mentioning. One of such is the work by Bertails et al. (2006). It represents hair strands as elastic rods, named "Super-Helices" by the authors, and draws from Cosserat and Kirchhoff theory of rods instead of Hooke's law of elasticity. This technique is able to simulate many types of hair (straight, curly, clumpy, etc.), but is not well suited for real-time simulation. At its time of publication, the way it handled bending, torsion and bending was a novelty, however we can't place this work in the extreme left end of

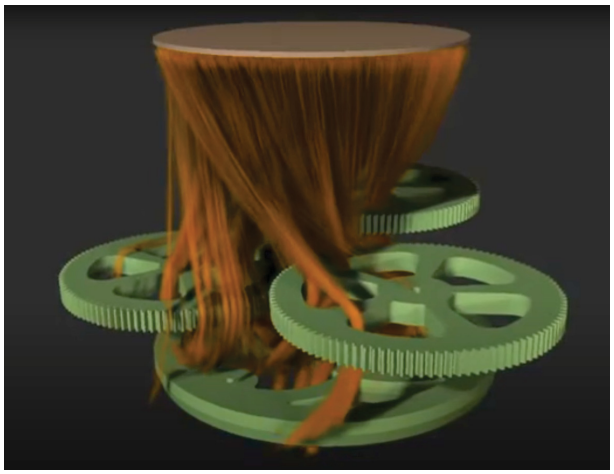


Figure 3.6: Complex collisions with hair (Chai et al., 2016)

the scale (Fig. 3.1) due to the fact that the authors interpolated some of the hair strands after simulating the guide strands.

There are also some works that are based on Position-Based Dynamics (PBD) framework. Since the subject of research of this dissertation is also based on PBD, we'll only mention briefly those works for now, because they will be explained in depth in the next chapter.

The first works about PBD hair simulation were published in the same year by Han and Harada (2012) and Müller et al. (2012). Even if both works are based on the same framework, they employ different strategies to solve the same challenges related to hair simulation. For example, while Han and Harada (2012) use many iterations per frame to preserve inextensibility, Müller et al. (2012) propose a modification in the PBD algorithm. Some time later, Sánchez-Banderas et al. (2015) combine the best of each technique in order to improve PBD hair simulation. We must remark that, although all those works produce realistic results in real-time, they are not the most accurate when it comes to physics fidelity. The works by Han and Harada (2012) and Müller et al. (2012) don't interpolate any hair strand, while the technique by Sánchez-Banderas et al. (2015) interpolates some of them. Going back to our scale in Figure 3.1, the first two works could be placed in the left side of the scale, but not in the extreme left due to the limited physical rigor. The last work would also be in the left side, but placed near the middle.

3.2 GROUP-BASED SIMULATION

Now we are going to present some works that don't simulate individual hair strands, instead, they try to capture the global hair dynamics, or in some cases, the local dynamics of groups of hair. The techniques in this section would be placed after the dashed line in the scale (Fig. 3.1).

One way to achieve simple hair dynamics is to simulate hair strips (Guang and Huang, 2002), which are strips of polygonal mesh or surfaces generated by curves (Liang and Huang, 2003) textured as hair wisps (Fig. 3.7). This approach can be used in real-time applications with little computational cost, since the number of objects simulated is extremely small. However it is limited to straight hair and simple hair motion, and the illusion of a head full of hair falls apart if viewed too close.

Around the same time, another approach was developed. It simulates hair dynamics by simulating wisps in a layered fashion (Plante et al., 2001): in the first layer, the skeleton curve (similar to a guide strand) moved according to a spring-mass system. This determines the global movement of the hair wisp. Then, in the second layer, a polygonal mesh around the skeleton curve

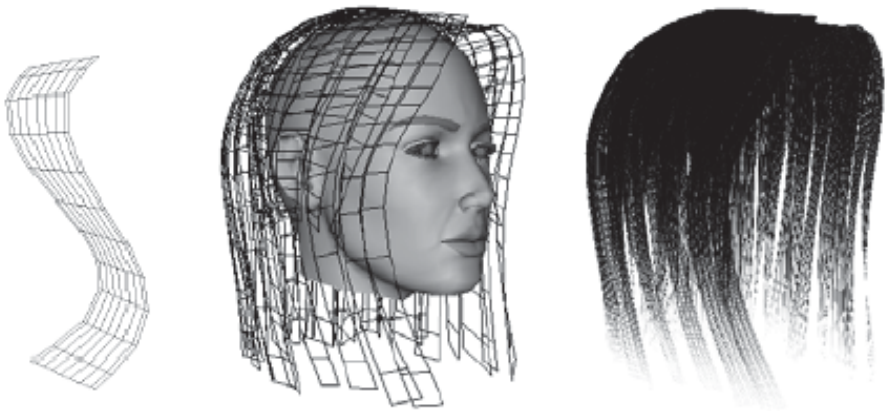


Figure 3.7: Representation of hair by Liang and Huang (2003)

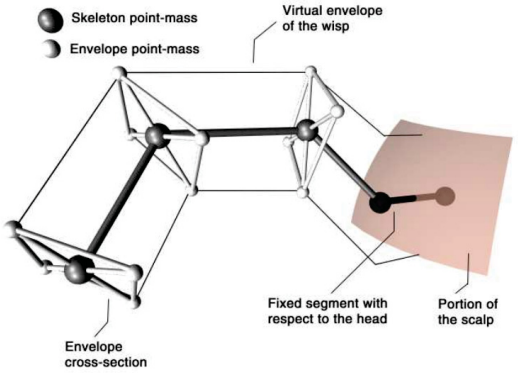


Figure 3.8: Scheme of polygonal mesh around a skeleton curve (Plante et al., 2002)

(called wisp envelope by the authors) is animated using another spring-mass system, capturing the wisp deformations. Finally, in the third layer, individual hair strands are generated inside the envelope. This approach was further improved by Plante et al. (2002) (see Fig. 3.8).

Still on the theme of generating hair strand from the inside of a polygonal mesh, Yuksel et al. (2009) presented a framework called "Hair Mesh", which gives the artist using it great control and the ability to create very complex hairstyles. This framework allows the artist to model the hairstyle by extruding faces of a polyhedron following some specific constraints. After the modeling phase, an algorithm fills the extruded mesh with hair strands, according to parameters specified by the artist. Different parameters might give different results from the same hair mesh, as seen in Figure 3.9. Initially, the authors managed to simulate hair dynamics by treating the vertices of the hair mesh as chains of rigid bodies. Hair mesh simulation was later improved by Wu and Yuksel (2016). In the new version, the authors used sheet-based cloth simulation models and introduced a new volumetric force model to handle collisions better.

There is also an approach that could be considered a hybrid between strand-based and group-based simulation: simulate hair strands individually but at the same time, treat the hair as a continuum fluid. Hadap and Magnenat-Thalmann (2001) were one of the first to take that approach. They modeled hair-hair, hair-body and hair-air interactions using smoothed particle hydrodynamics (SPH) and the hair geometry in a strand-by-strand fashion using elastic fiber dynamics.

Figure 3.9: Hair meshes (Yuksel et al., 2009)

3.3 CONCLUSION

Now that wee have a panorama of the field of hair simulation, we'll dive deeper in PBD hair simulation in the next chapter by explaining our motivation to study that simulation framework, how it works and how it was used to simulate hair dynamics. After that, in Chapter 5, we will discuss our contribution to that family of hair simulation.

4 POSITION-BASED DYNAMICS

As mentioned in the Introduction of this dissertation, this work is simply a fruit of curiosity, and so was the choice to explore Position-Based Dynamics. It is conceptually simple and easy to understand and to modify and expand. In fact, that was the main motivation of choosing to work with PBD. In the beginning of this Master's degree, we considered using the simulation scheme by Selle et al. (2008), however, when we discovered the work by Müller et al. (2012), we decided to make it our main research subject, as it is more beginner-friendly. Only after many months "playing around" with PBD is that we came to know the works of Han and Harada (2012) and Sánchez-Banderas et al. (2015).

In this chapter we will explain what PBD is and will discuss the details of each of the works mentioned above.

4.1 INTRODUCTION TO POSITION-BASED DYNAMICS

PBD was first introduced by Müller et al. (2007) as a simulation framework that is stable, robust and fast to compute, which are characteristics that many times are preferred over accuracy in real-time applications. While many techniques for simulating dynamic objects work with forces or impulses, this approach works with the position of the dynamic bodies (hence the name).

According to the authors:

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

In this framework, the main components that drive the types of dynamics being simulated are the **constraints**, which are, in short, functions that represent numerically how off the state of a dynamic body is from what it should be. For example, if we want to simulate particles colliding with a plane surface, we can devise a constraint (function) that returns how far the particle has penetrated into the plane. From the value returned by the constraint, we can apply corrections to the particle's position accordingly. If we correct so that the constraint is fully satisfied, we call it a **hard constraint**, otherwise it is a **soft constraint**.

In PBD, the way to implement a soft constraint (which is commonly used in soft-body and elastic simulations) is to have a *stiffness parameter* that behaves like the factor of a linear interpolation between fully enforcing the constraint and ignoring it. A limitation of this approach is that the stiffness of a constraint is dependent on the size of the time step, leading to a inconsistent behavior if it varies throughout the simulation.

To remedy that, Macklin et al. (2016) proposed an improvement to PBD and called it Extended Position-Based Dynamics (XPBD). This new version of PBD updated the general formula of constraints and introduced the concept of a total Lagrange multiplier so that soft

constraints could be solved independent of time step. XPBD, therefore, has correspondence to well-defined elastic and dissipation energy potentials.

Since the derivation of XPBD is considerably different from PBD, the simulation solver would also end up being very different and it wouldn't benefit from the simplicity and performance of PBD. To address that, some approximations were made. Although XPBD is not an exact solver for dynamic objects, the authors demonstrated that the error can be negligible for most applications and that, given enough iterations, it converges closely to the exact result. Some time later, Macklin et al. (2019) made the discovery that performing sub-steps results in better convergence than iterating over the constraints multiple times in one single time step.

4.1.1 The XPBD Algorithm

Algorithm 1 XPBD algorithm

1: // Simulation loop
2: **loop**
3: $\Delta t_s \leftarrow \frac{\Delta t}{n_{subs}}$
4: **for all** elements i **do**
5: $n \leftarrow 0$
6:
7: **while** $n < n_{subs}$ **do**
8: predict inertial state $\tilde{x}_i$ // see Eq. 4.1
9:
10: **for all** constraints j **do**
11: compute $\Delta\lambda_{ij}$
12: compute $\Delta\mathbf{x}_i$
13:
14: update $\tilde{x}_i \leftarrow \tilde{x}_i + \Delta\mathbf{x}_i$
15: **end for**
16: update state $\mathbf{x}_i^{n+1} \leftarrow \tilde{x}_i$
17: update velocity $\mathbf{v}_i^{n+1} \leftarrow \frac{\mathbf{x}_i^{n+1} - \mathbf{x}_i^n}{\Delta t_s}$
18: $n \leftarrow n + 1$
19: **end while**
20: **end for**
21: **end loop**

In Algorithm 1 we outline the main steps of XPBD after the contributions from Macklin et al. (2019). In the algorithm an element can be a vertex, a particle or even something else. $\mathbf{x}_i^n$ usually denotes the position of the i-th element in the n-th sub-step, but it can also represent any generalized coordinate model, such as rigid-body transforms. $\mathbf{v}_i^n$ is the velocity of the i-th element in the n-th sub-step, and m_i is its mass. If needed, we can also include the tensor of inertia, angular velocity, etc. In a real application, at the beginning of the main loop (line 2) we would process user's inputs and before the end of the loop (line 21) we would render the results for that simulation step. The number of sub-steps executed at each frame is given by the constant n_{subs}.

If we are simulating particles, for example, we would predict the inertial position [1] of the i-th particle for the next sub-step (line 8) using the following expression from classical Newtonian physics, where Δt_s is the time sub-step and $\mathbf{f_{ext}}$ are the external forces acting over that particle:

$$\tilde{\mathbf{x}}_i = \mathbf{x}_i^n + \Delta t \mathbf{v}_i^n + \Delta t^2 \frac{1}{m_i} \mathbf{f}_{ext} \tag{4.1}$$

To solve a constraint C_j, we need to compute the Lagrange multiplier for that constraint in respect to the i-th particle ($\Delta\lambda_{ij}$) and the position correction for the same particle ($\Delta\mathbf{x}_{ij}$). The Lagrange multiplier can be found with Equation 4.2. In that equation, $\tilde{\alpha}_j = \alpha_j / \Delta t_s^2$, where α_j is the compliance of said constraint, or the inverse of stiffness.

$$\Delta\lambda_{ij} = \frac{-C_j(\tilde{\mathbf{x}}_i)}{\frac{1}{m_i}|\nabla C_j(\tilde{\mathbf{x}}_i)|^2 + \tilde{\alpha}_j} \tag{4.2}$$

Having $\Delta\lambda_{ij}$, we can find $\Delta\mathbf{x}_{ij}$ with the following equation, which will be used to correct the initial guess ($\tilde{\mathbf{x}}_i$):

$$\Delta\mathbf{x}_{ij} = \frac{\Delta\lambda_{ij}}{m_i} \nabla C_j(\tilde{\mathbf{x}}_i) \tag{4.3}$$

In the works related to XPBD, there is a term called Total Lagrange Multiplier that is stored across sub-steps (or iterations). It was omitted in Algorithm 1 because is optional for sub-steps implementation (Macklin et al., 2019), although it can provide useful information about the total constraint force, which can be used for force dependent effects (like breakable joints) and haptic feedback (Macklin et al., 2016).

4.2 POSITION-BASED HAIR SIMULATION

We can use PBD (or XPBD) to simulate hair dynamics by representing hair strands as chains of particles with distance constraints connecting each pair of consecutive particles. Those constraints will force the particles to stay at a certain distance from each other. To simulate the hair, we can first move the first particle of the strand (we'll call it **root particle**) to follow the motion of the head or any other surface it is attached to, then we process the next particle in the chain, predicting its inertial state and then solving its constraints. We repeat this process for all subsequent particles and for all hair strands.

One problem of the algorithm we just described is that, if we move the i-th particle towards or away from the $i+1$-th particle to comply with the distance constraint, we'll be violating the constraint between the particles $i-1$ and i.

Han and Harada (2012) solved this issue by applying the constraints multiple times for each frame so that the particles' positions converge to a solution, however hair inextensibility is not strictly guaranteed. They've also introduced a **global shape constraint** and a **local shape constraint**. Both constraints are soft constraints, as formulated by the original PBD. In their algorithm, they specify the initial particles' positions as target positions for the global shape and the local shape constraints. For the first constraint, the target positions are represented in the global coordinate system, as for the second constraint, the positions are represented in local coordinate systems relative to the predecessors of each particle. This scheme of constraints allow the representation of curly hair and different hair styles.

[1] **Inertial position** or **inertial state** refers to the state of an object in which the resulting force acting over it is constant

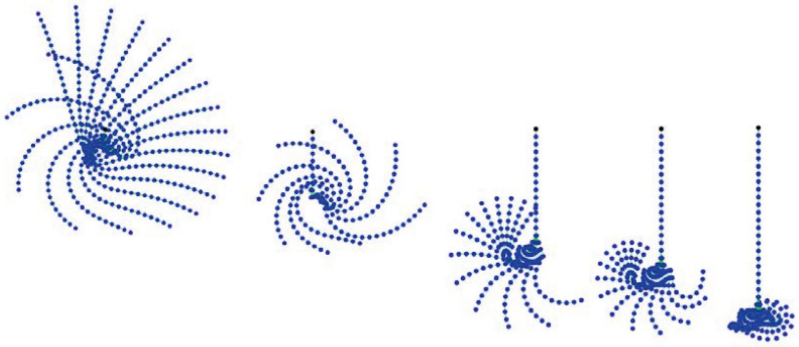


Figure 4.1: Unrealistic motion noted by Müller et al. (2012)

Figure 4.2: Hair simulations by Sánchez-Banderas et al. (2015)

During simulation, this algorithm has to compute a transformation matrix for each particle that is dependent on the transformation matrices of the particles that came before in the chain. Because of that dependency, this approach has limited space for parallelization. The authors proposed variations of their main idea that are more suited for GPUs, although being less precise.

In the same year, Müller et al. (2012) published a work that presented another approach for the problem we stated in the beginning of this section. Instead of iterating many times over the constraints to converge to a better solution with less elongation or shrinking of hair strands, the authors proposed a modification to PBD that guarantees inextensibility in only one iteration per time step. They called it **Dynamic Follow-The-Leader** (DFTL). The rationale for that idea probably came from the question "what if, instead of moving both particles to comply with the distance constraint, we moved only the child particle?" (*child particle* referring to the particle more distant from the root).

If we move only the child particle, the authors noted, it would be the equivalent of the parent particle (the one that is closer to the root and comes before in the chain) having infinite more mass than its child, leading to unrealistic motion (see Figure 4.1). To circumvent that problem, Müller et al. (2012) proposed a correction term that should be applied to the parent's velocity, to be applied on the next step. This term compensate the displacement of the child particle, while introducing damping. The authors also managed to produce curly hair, but only as a post-processing effect for rendering, with minimal influence on the way the hair moves.

Some time later, the local shape constraint (Han and Harada, 2012) and DFTL (Müller et al., 2012) were combined into one simulation framework by Sánchez-Banderas et al. (2015) (Fig. 4.2). They've made two more contributions to PBD-based hair simulation: a method to simulate hair-hair interactions and a data layout to leverage the GPU's memory architecture.

The authors addressed hair-hair interactions by considering the hair volume as a fluid continuum, like Hadap and Magnenat-Thalmann (2001). The way Sánchez-Banderas et al. (2015) modeled fluid behavior using SPH allowed them to use the same particle-based representation in the fluid simulation and in PBD's distance and shape constraints. The goal of such approach is to create a volumetric hairstyle and give the impression that the hair strands have a finite thickness.

One of the main problems when trying to parallelize PBD hair simulation is the data dependency that exists when solving constraints. Processing in parallel constraints that have no particle in common (which is related to graph-coloring problem) is a way to address this issue and it is, in fact, the method used by Han and Harada (2012). However, it breaks DFTL's motivation of solving all constraints in a single pass. The solution proposed by Sánchez-Banderas et al. (2015) provides good GPU occupancy and memory coalescence.

Instead of having all the particles' data from a single strand in a contiguous block of memory, they suggest to have all the root particles contiguous to each other, then all the next particles, and so on. In other words, the memory is organized in batches in respect with the local particle index(*e.g.* root particles, then particles with index 1, then 2, etc.). This way the particles are processed in batches and the serial dependency between particles is maintained.

Unfortunately, that data layout is not efficient for the fluid simulation. To remedy that, the authors employ a spatial subdivision strategy and store the particles' data in temporary buffers to enhance memory coalescence and thread coherence.

4.3 CONCLUSION

Now that we have explained in details what is Position-Based Dynamics and how it was used to simulate hair, we will now introduce in the next chapter our contribution: a new local shape constraint.

5 NEW SHAPE CONSTRAINT

Our motivation for the new constraint was to expand the work by Müller et al. (2012) and add to it the capability of simulating curls and keeping the hairstyle. As we mentioned previously, there is no recent comprehensive survey on the topic of real-time hair simulation, therefore, at the beginning of our research, we still had no knowledge of the works by Han and Harada (2012) and Sánchez-Banderas et al. (2015). Only when we were near the end of our formulation that we became aware of those works, fortunately, although similar, our technique has enough unique features to be considered a new work of its own. We took inspiration from the data layout from Sánchez-Banderas et al. (2015) to parallelize our algorithm.

5.1 MOTIVATION AND OVERVIEW

Our first attempts to simulate curly hair consisted of having soft distance constraints connecting pairs of particles one particle apart (*e.g.* particles of index 1 and 3, 2 and 4, and so on), however we were unable to get satisfying results. Then, we decided to change our approach and work on a constraint that eventually became the one we are presenting in this dissertation.

At a high level, before starting the simulation, our algorithm stores the target position of each particle after the root in the local coordinate system of their respective parents. During the simulation, our shape constraint tries to minimize the angle between each particle and its target position in respect to the coordinate system of its predecessor. Conceptually, our approach is very similar to that of Han and Harada (2012), however there are some key differences. For instance, while Han and Harada's local shape constraint minimizes the Euclidean distance between particle and its target position, ours minimizes the angle between the particle and the target, passing by the particle's parent.

Another distinction is the method used to compute the local coordinates for each particle. Han and Harada take a more traditional approach by first defining the root's coordinate system and then rotating and translating it for all the next particles. On the other hand, we explored an unorthodox approach that involves projecting an auxiliary point for each particle in the strand. Although our approach is a simplification of the algebraic principles applied by Han and Harada, it brought satisfactory results. Yet another distinction from Han and Harada (2012) and Sánchez-Banderas et al. (2015) is the fact that we used XPBD's formulation for soft constraints.

5.2 DETAILED EXPLANATION

We can separate our method in two phases: the preparation and the simulation. The first must be executed only once, while the last will happen every frame.

5.2.1 Preparation

Before starting the simulation, we have to compute and store some data that our algorithm will use during the simulation. Since our shape constraint uses the target position in local coordinates, we need a way to compute such coordinate systems. We will now describe the process to do it and all the other procedures involving our method focusing only on a single strand, therefore we'll index particles in respect with their position in the chain.

We get the local coordinates centered at the root particle in a similar way to Han and Harada (2012): let $\mathbf{p}_0$ be the position of said particle, then we choose an arbitrary point $\mathbf{Aux}_0$ (called auxiliary point) that satisfies the following properties:

1. $|\mathbf{Aux}_0 - \mathbf{p}_0| = 1$
2. $\mathbf{Aux}_0$ is on the plane perpendicular to the unit vector $\mathbf{n}_0$, which is the normal vector of the surface from which the strand is protruding at $\mathbf{p}_0$

With $\mathbf{Aux}_0$ and $\mathbf{n}_0$, we can compute the basis vectors $\mathbf{i}_0$, $\mathbf{j}_0$ and $\mathbf{k}_0$ for the local coordinates relative to $\mathbf{p}_0$:

$$\mathbf{i}_0 = \mathbf{Aux}_0 - \mathbf{p}_0 \tag{5.1}$$
$$\mathbf{j}_0 = \mathbf{n}_0 \tag{5.2}$$
$$\mathbf{k}_0 = \mathbf{i}_0 \times \mathbf{j}_0 \tag{5.3}$$

Let $\mathbf{T}_0$ be the basis matrix from those basis vectors. To simplify our explanation, let's consider that the particles' target positions are their initial positions. Therefore, the target position for the next particle in the chain can be computed by $\mathbf{p}_1^* = \mathbf{T}_0(\mathbf{p}_1 - \mathbf{p}_0)$ (where $\mathbf{p}_1$ is the position of the particle right after the root). For the simulation phase, we'll need to store the auxiliary point $\mathbf{Aux}_0$, the normal vector $\mathbf{n}_0$ and the target position $\mathbf{p}_1^*$.

To compute the next target positions $\mathbf{p}_i^* = \mathbf{T}_{i-1}(\mathbf{p}_i - \mathbf{p}_{i-1})$, we will need to find the local basis matrix $\mathbf{T}_{i-1}$, relative to $\mathbf{p}_{i-1}$. To do so, we need the basis vectors $\mathbf{i}_{i-1}$, $\mathbf{j}_{i-1}$ and $\mathbf{k}_{i-1}$, which are given by:

$$\mathbf{i}_{i-1} = \mathbf{Aux}_{i-1} - \mathbf{p}_{i-1} \tag{5.4}$$
$$\mathbf{j}_{i-1} = \frac{\mathbf{p}_{i-1} - \mathbf{p}_{i-2}}{|\mathbf{p}_{i-1} - \mathbf{p}_{i-2}|} \tag{5.5}$$
$$\mathbf{k}_{i-1} = \mathbf{i}_{i-1} \times \mathbf{j}_{i-1} \tag{5.6}$$

We compute $\mathbf{Aux}_{i-1}$ by projecting $\mathbf{Aux}_{i-2}$ onto the plane defined by the point $\mathbf{p}_{i-1}$ and the normal vector $\mathbf{p}_{i-1} - \mathbf{p}_{i-2}$, and adjusting it so that $|\mathbf{Aux}_{i-1} - \mathbf{p}_{i-1}| = 1$. This time, we need to store only $\mathbf{p}_i^*$ for the simulation phase.

5.2.2 Simulation

In the simulation phase, our algorithm will do something similar to what was done in the preparation phase, except that instead of storing the target position, it will compare the particles' current positions with their respective target position using our formulation for the shape constraint. Algorithm 2 describes what we've just written:

The function $computeBasisMatrix$ (lines 7 and 24) computes the basis matrix by finding the basis vectors $\mathbf{i}_i$, $\mathbf{j}_i$ and $\mathbf{k}_i$ like in Equations 5.4, 5.5 and 5.6 respectively. The function $projectAuxPoint$ (line 23) computes the next auxiliary point by projecting the previous one onto the plane defined by the point $\mathbf{p}_i$ and the normal vector $\mathbf{n}$ and then moving it so that $|\mathbf{Aux} - \mathbf{p}_i| = 1$.

One might ask why do we do this projection step instead of computing a rotation matrix to find $\mathbf{T}$, like Han and Harada (2012) did. Our rationale for that is the fact that projecting the auxiliary point takes less trigonometric operations than computing a rotation matrix.

Algorithm 2 Shape Constraint algorithm for a single strand

```
// Inside the simulation loop
// Some elements from XPBD were omitted for brevity

update n_0 and Aux_0 according to base surface transformations
n ← n_0
Aux ← Aux_0
T ← computeBasisMatrix(p_0, Aux, n)

i ← 1
while i < number of particles in strand do
    predict inertial state p̃_i

    Δp_i ← shapeConstraint(T(p̃_i − p_{i−1}), p*_i)
    correct p̃_i using Δp_i

      // Other constraints can be placed here

    p_i ← p̃_i
    update v_i
    correct v_{i−1} using Δp_i

    n = (p_i − p_{i−1}) / |p_i − p_{i−1}|
    Aux ← projectAuxPoint(p_i, n, Aux)
    T ← computeBasisMatrix(p_i, Aux, n)

    i ← i + 1
end while
```

The function $shapeConstraint$ computes $\Delta\lambda$ (Eq. 4.2) and $\Delta\mathbf{x}$ (Eq. 4.3), returning the later. The formulas for C and ∇C in our constraint are given by Equations 5.7 and 5.8, where $\mathbf{v}_1$ and $\mathbf{v}_2$ are the arguments of the function $shapeConstraint$.

$$C = acos\left(\frac{\mathbf{v}_1 \cdot \mathbf{v}_2}{|\mathbf{v}_1| \cdot |\mathbf{v}_2|}\right) \tag{5.7}$$

$$\nabla C = \begin{bmatrix} \dfrac{\mathbf{v}_1.x \cdot a - \mathbf{v}_2.x \cdot b}{b\sqrt{bc - a^2}} \\ \dfrac{\mathbf{v}_1.y \cdot a - \mathbf{v}_2.y \cdot b}{b\sqrt{bc - a^2}} \\ \dfrac{\mathbf{v}_1.z \cdot a - \mathbf{v}_2.z \cdot b}{b\sqrt{bc - a^2}} \end{bmatrix}, \text{ where } a = \mathbf{v}_1 \cdot \mathbf{v}_2,\ b = |\mathbf{v}_1|^2, \text{ and } c = |\mathbf{v}_2|^2 \tag{5.8}$$

5.3 CONCLUSION

In this chapter we have explained our contribution, which consists in a new shape constraint and the algorithm to be used with it. We have also highlighted the main distinctions from our approach to other similar works. In the next chapters we will describe our implementation and present the results from our experiments.

6 IMPLEMENTATION

We've wrote two implementation for our hair simulation. The first one served as a proof of concept for our constraint formulation, while the later was used to evaluate performance and stress test it. In this chapter we will present both.

6.1 IMPLEMENTATION 1: TYPESCRIPT AND WEBGL

We chose to use web-based technologies for three main reasons: first, modern web frameworks have the feature "hot reload", in which the code running in the browser is updated as soon as the source code is modified. This feature allows fast iterations during development. The second reason is level of abstraction: even when it comes to graphics programming, there are many frameworks and libraries that abstract away much of the boilerplate code and low-level memory management. Third, having a simulation that runs in the web browser is very useful for live demonstrations. We've also chose to use Typescript instead of pure JavaScript because static typing helps to write less error-prone code.

We've used the library *Three.JS* (Cabello, 2025) to handle rendering and user input. This is a graphics library for the web built on top of WebGL which takes care of low-level buffer management, as well as scene graph management and input processing, allowing users to interact with the virtual 3D space.

As we've mentioned before, our goal for the web implementation was to use it as proof of concept and as a tool to refine our constraint formulation and our algorithm, therefore, performance was not our main concern. A screenshot of our application can be found in Figure 6.1. As we can see, it features a parameter menu in the right-hand side of the screen. With that menu, we can show or hide the target points and auxiliary point of each particle, change simulation parameters, such as compliance and damping, and configure hair strand characteristics, like number of segments per hair strand or whether the hair is curly or not.

Besides the menu, our application featured interactive camera and real-time interactions with the object in the scene. In other words, users can pan, orbit, zoom in and out the camera and can drag the object around the scene, causing the hair to react to the movement in real-time.

We've developed a second version of this application, this time with predefined parameters and simplified user interface optimized for mobile devices (Figure 6.2). It was used as a live demo for a presentation done at *2024 ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games* (I3D) (Cavazotti and Guedes, 2024).

6.2 IMPLEMENTATION 2: PYTHON AND CUDA

The goal of our second implementation was to run the simulation on a GPU. To do that, we needed a way to write our simulation algorithm for GPU and then render the results in real-time. We've considered many options to accomplish our goal: we could use WebGPU, which is a new web API and specification for browsers that allows platform-agnostic GPU programming, however, since it's still something new in the space of web programming, there's not much support from existing frameworks and libraries. That being said, we would need to program the rendering pipeline ourselves if we wanted to use WebGPU's compute shaders to run our simulation, and that would be too much work for the scope of our project.

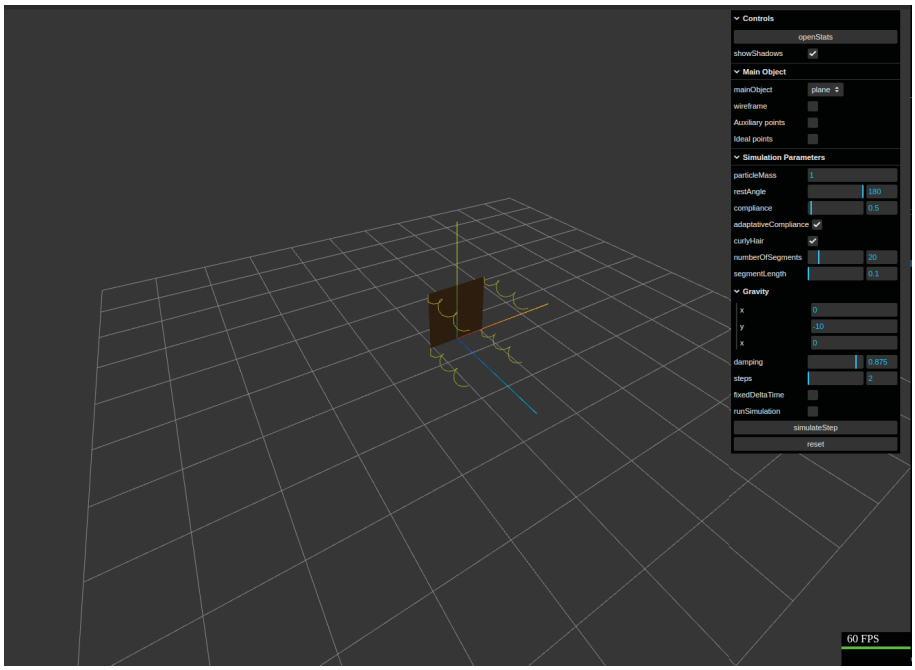


Figure 6.1: Web-based virtual simulation environment

We have also thought about using OpenGL or Vulkan, but we would run into the same issues as WebGL. Another option was to use a game engine such as Unity or Godot, but it felt that this choice would bloat our project and would require learning how to use those tools. In short, we couldn't find a solution that would allow us to easily implement our algorithm and have the performance we needed.

Finally, we discovered *Taichi*, "an open-source library [...] which alleviates this practical issue by providing an accessible, portable, extensible, and high-performance infrastructure that is reusable and tailored for computer graphic" (Hu, 2018). This library is a "domain-specific language embedded in Python" (Hu, 2025) that JIT compiles Python-like code into machine code for the CPU or GPU, depending on the client's hardware. In our case, our simulation was compiled into CUDA because we have NVIDIA GPUs.

In our second implementation, we focused on performance and tooling rather than interactivity, therefore, users can't orbit the camera nor move the object around. On the other hand, we've implemented mechanisms to run automated tests, gather data and generate reports.

In this implementation, as well as the one written in Typescript, we generate the hair strands procedurally from the vertices of a base mesh. We can define how many particles the strands will have and the distance between each particle. This implementation also features the ability to place spherical colliders and have the hair react with it.

Our simulation kernel closely follows the structure outlined in Müller et al. (2012): first, we move the strands' roots to match the base mesh transform, then, for all the next particles in the strand, we predict the inertial position, then we solve the constraints by correcting the predicted position. Next, we update the particle position, its velocity, and apply the DFTL correction on the previous particle's velocity.

We've also added some additional steps necessary to our shape constraint. For instance, when updating the root particle, we retrieve the normal vector as well as the auxiliary point stored in the preparation phase, since these information will be used in the next particle's shape constraint. Similarly, after updating the position, but before processing the next particle, we compute the next normal vector and auxiliary point.

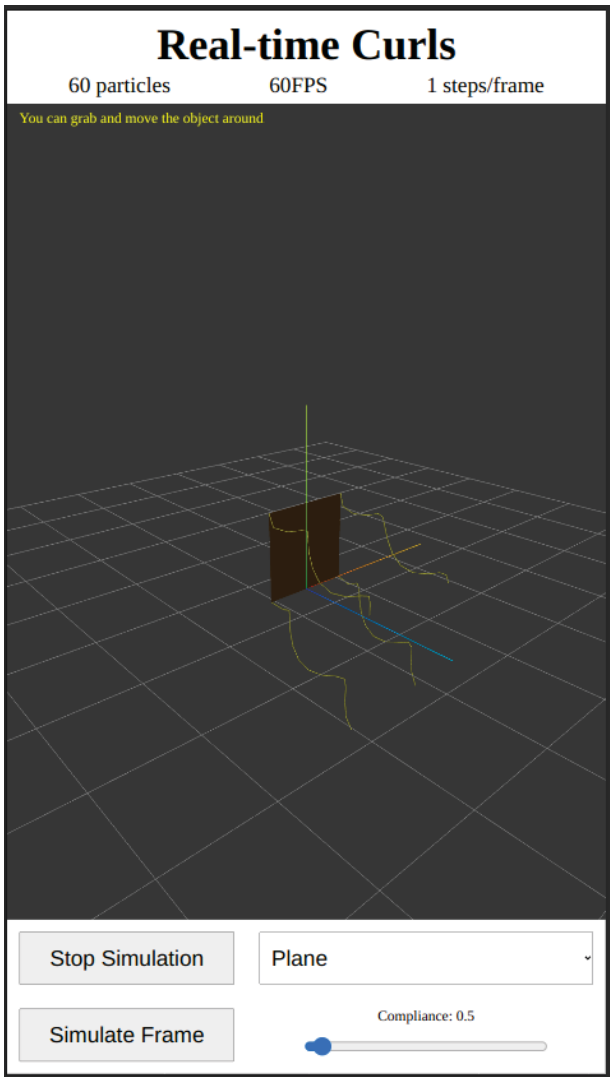


Figure 6.2: Mobile version of web-based simulation

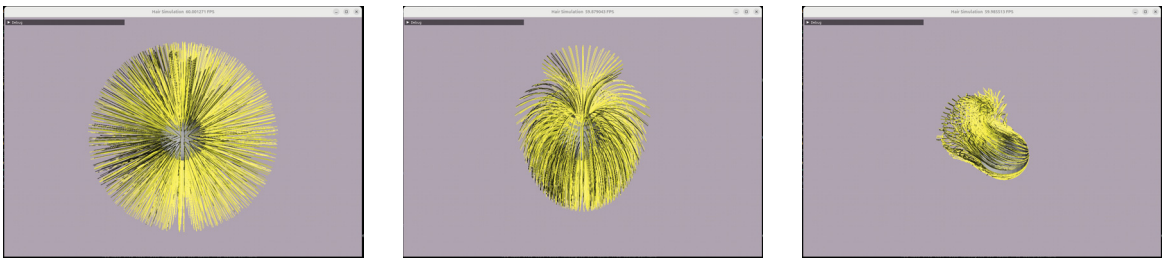


Figure 6.3: Screen shots of simulation written with Taichi

Regarding parallelization, we assigned one hair strand for each thread, which iterates over all the particles that constitutes the strand, performing operations as we've just described. We could have implemented a more sophisticate parallelization approach to reduce serial computation, like the one used by Han and Harada (2012), however that would require more sub-steps and defeats the purpose of DFTL, which is to simulate hair dynamics with only one sub-step per frame. That being said, our parallelization strategy was similar to the one employed Sánchez-Banderas et al. (2015).

We've implemented three constraints in our application: the new shape constraint, a distance constraint like the one used in Müller et al. (2012) and a simple penetration constraint that moves the particles out of the spherical colliders.

6.3 CONCLUSION

Now that we have presented how we've implemented the constraint from the last chapter, we will discuss in the next chapter the tests we've performed and the results we've observed. After that, in Chapter 8, we will present the conclusion of our work.

7 TESTS AND RESULTS

After arriving at the formulation and implementations described above, we put our ideas to test. We've performed most of the qualitative tests using the web implementation. Quantitative tests regarding performance and error margin were done in the GPU implementation. Before presenting the data we've collected, we will describe the hardware used for the tests.

While the web version was tested on many devices, the GPU version was tested on a *Acer Predator Helios 300* notebook with 16 *GB* of RAM, a 9th generation *Intel Core i7* processor and a *NVIDIA GeForce RTX 2060* GPU. See Table 7.1 for more details.

NVIDIA GeForce RTX 2060	
Clock Frequency	1.37-1.68 GHz
CUDA Cores	1920
VRAM	6GB GDDR6
Memory Bus Width	192 bits
Intel Core i7-9750H	
Clock Frequency	2.60-4.50 GHz
Cache	12 MB Intel Smart Cache
CPU Cores	6 (12 logic cores)
Memory	
Size	1x 16GB DDR4
Clock Frequency	2666 MHz

Table 7.1: Hardware details

7.1 QUANTITATIVE TESTS

Our goal was to simulate as many hair strands as possible. In our web implementation, we managed to simulate around 11 thousand particles at frame rates between 37 and 58, depending on the machine that was running it. However, we're more interested in the performance of the GPU implementation, since that implementation was the final goal of our project.

7.1.1 Data Layout and Performance

We compared two levels of optimization in data layout with a naive implementation. In the naive version, we used a "struct-of-arrays" approach, in other words, we had an array for particle positions, another for particle velocities, and another for target positions, and so on, as shown in Figure 7.1.

If we analyze Algorithm 2 and consider that each thread has to iterate through many particles, it's clear that the naive data layout is not a good approach. To be clearer, in the algorithm

Figure 7.1: Naive data layout

we can see that we need $\mathbf{p}_i$, $\mathbf{v}_i$ and $\mathbf{p}_i^*$ at each iteration. If those values are distant from each other, the GPU will have to load different blocks of data, also known as pages of memory, increasing memory bandwidth consumption, blocking threads and decreasing throughput, leading to worse performance. It's worth remembering that modern GPU architecture is based on the concept of *single instruction, multiple data* (SIMD), so every thread in a block of threads is executing the same single instruction at any given time. If there is divergence in execution flow or if the data needed by a thread is unavailable, the other threads are blocked until the data is available or the execution flow can be reconciled[1].

This leads us to the first level of optimization: "array-of-structures". In this layout, we group together all the data related to each particle, as represented in Figure 7.2. This way, the GPU doesn't need to load different blocks of data to simulate one particle. However, both the naive layout and this one have a problem in common: two neighboring threads are not processing data that live in neighboring memory addresses. That also causes increased memory bandwidth usage and lower throughput, since threads from the same group might be working with data from different pages.

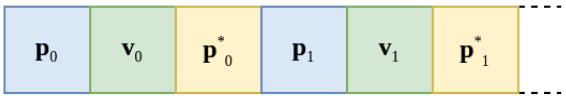


Figure 7.2: Array-of-structs

This issue was addressed by Sánchez-Banderas et al. (2015) in their "batch" data layout. Instead of having the data of all the particles of a strand in contiguous spaces of memory (see Figure 7.3), we can allocate the first particles of every strand together, then the second particles, and so on, as illustrated in Figure 7.4. In this layout, it's less likely that threads will be blocked because of memory access, and although the stride done on memory addresses in the kernel's main loop is bigger and may cause new pages of memory to be loaded, it will not block any threads.

We've tested our implementation with each one of the three data layouts, and for each version, we've experimented with varying number of hair strands and particles per strand.

7.1.2 Tests description

We chose an icosahedron as our base mesh from which the hair strands would be generated. This geometric primitive wasn't an indexed mesh, so each face had its own vertices, even if they coincided with the vertices from neighboring faces. Therefore, the mesh had 60 vertices instead of 12, resulting in 60 hair strands. Then, we applied the Catmull-Clark subdivision algorithm (Catmull and Clark, 1998) and obtained a mesh with 80 faces and 240 vertices. We repeated this procedure three more times, obtaining meshes with the following number of faces and vertices:

Since we can only do strand-level parallelization, we wanted to test how the serial part of the algorithm (the number of particles in each strand) impacted performance, so we tested with strands 10, 25, 50, 100 and 200 (only for subdivision levels 1,2 and 3) particles long.

Our tests consisted of running 500 frames and measuring the kernel execution time using *Taichi*'s profiling tools, then repeat the test using NVIDIA NSight Compute to collect other GPU metrics.

[1]This explanation is a simplification of how GPUs actually work. In reality, there are many optimizations that can lessen the impact of these issues.

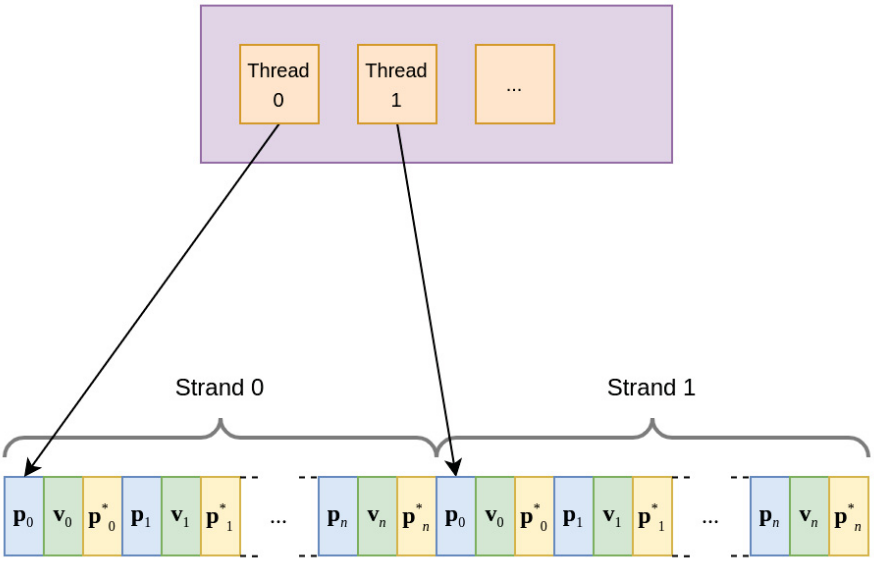


Figure 7.3: Schema of memory access in “array-of-structs” layout

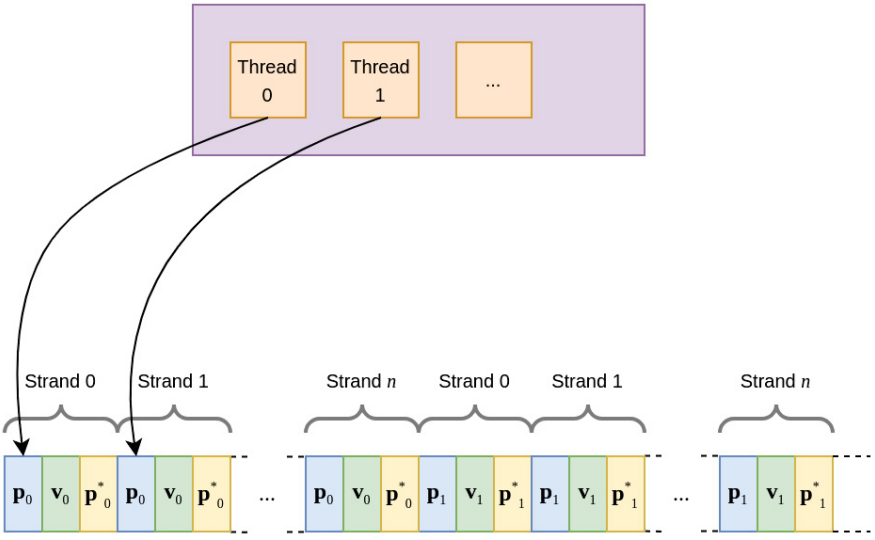


Figure 7.4: Schema of memory access in “batch” layout

Subdivision Level	Faces	Vertices
0	20	60
1	80	240
2	320	960
3	1280	3840
4	5120	15360

Table 7.2: Number of faces and vertices in each subdivision level

7.1.3 Results

We will start the presentation of our results with Figure 7.5, which depicts kernel execution times with the heaviest workloads we tested:

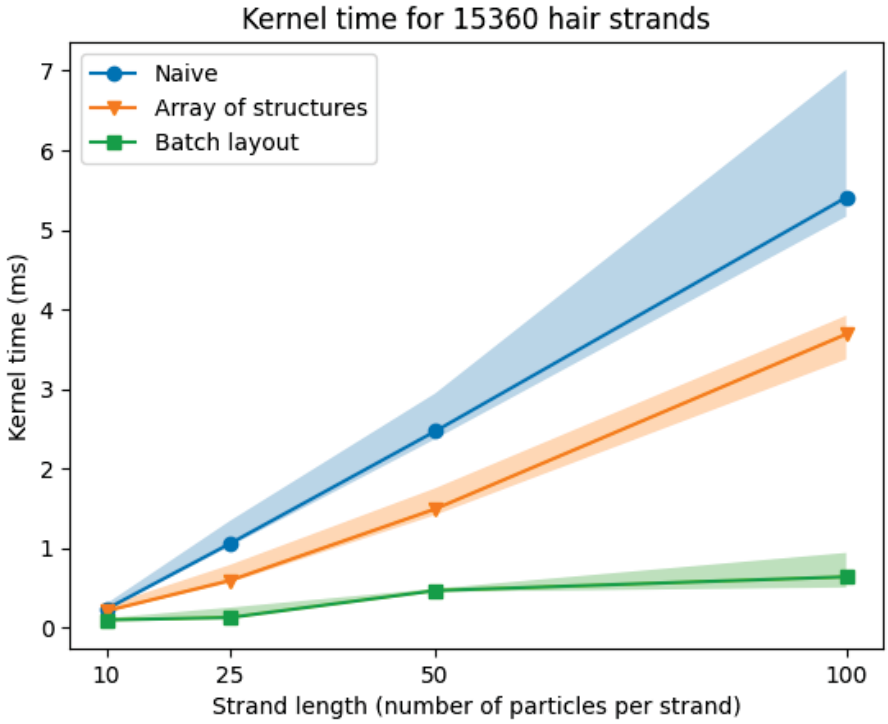


Figure 7.5: Kernel time for 15360 hair strands

In this figure, the colored areas represent the maximum and the minimum time recorded throughout the 500 frames simulated, and the lines represent the average. The number of particles simulated ranged from around 153 thousand to more than 1.5 million. In the chart we can see how much the "batch" data layout improved performance, with average kernel execution time for 100 particles per strand dropping from 5.4 *ms* in the naive layout to only 0.64 *ms* in the optimized version. NVIDIA NSight Compute's *SM Throughput* raised from 3.11% to 23.73%. The performance gain from the "array of structures" optimization is relatively small due to the issue we've described earlier: threads from the same group are working with data far apart from each other.

If we compared the kernel execution times with hair strands with the same number of particles but with different number of strands, in theory we should expect the times would be the same for all executions, since in all of them, the kernel would process the same number of particles. However that is not what we observed:

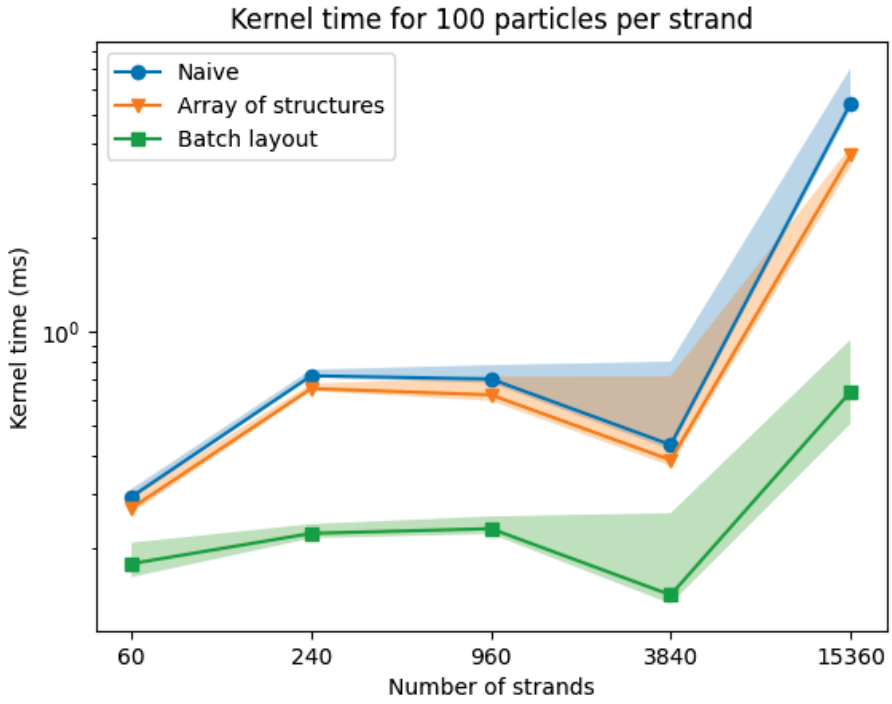


Figure 7.6: Kernel time for 100 particles per strand

The increase of execution time observed when comparing 60 and 240 strands is due to memory access pattern: the less optimized versions had to load more pages of memory, and that caused the significant increase in execution time. Then, the time is reduced at 960 and 3840, probably indicating that the data was more aligned with memory page size. The spike in execution time at 15360 is probably also caused by inefficient memory access patterns, but we couldn't pinpoint the specific cause, and our knowledge only goes so far to make assumptions.

The careful reader might notice that, when solving constraints, each constraint "undoes" what the previous one did. This effect can be attenuated with multiple passes of constraint solving: although not perfect, the particles position will converge so that the error is minimized. In our case that doesn't happen, since we made a strong point of only performing a single pass of constraint solving for each particle.

In our implementations, the order in which we solved the constraints are: shape constraint first, then distance constraint, then penetration constraint. Therefore, from the observation that we've just made, there's no real guarantee that the hair strands are inextensible. We claim that, for the purposes of our algorithm, the resulting error is negligible. To back that claim, we've calculated the error in each frame by summing the length of all hair strands after the simulation and, then, dividing it by the expected total length. In all our tests, the average error was never greater than 0.2%.

All the data we've collected in our tests can be found in Appendix A.

7.2 QUALITATIVE TEST

Now, regarding the visual quality of our algorithm, we can see, as an example, in Figure 7.7 the rendered result of a simulation with more than 17 thousand hair strands and more than 1.3 million particles. We exported the hair strands from our application to the 3D modeling software Blender (Blender Online Community, 2025), where we rendered the images. Some inaccuracies

seen in the second image where caused by the way we approximated the geometry of the head using only spheres.

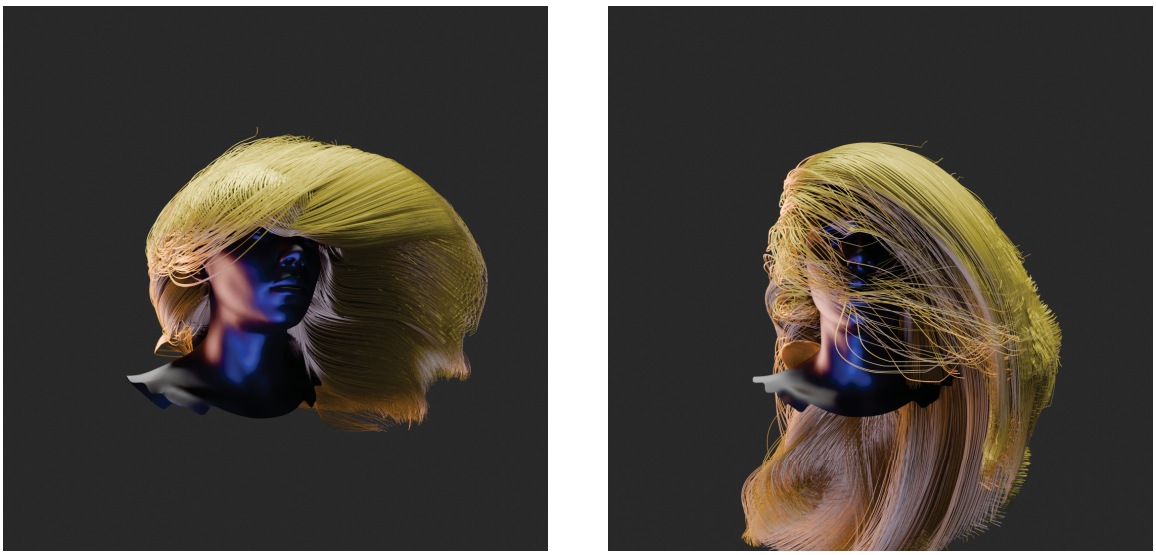


Figure 7.7: Rendered hair simulations

We've observed that our constraint can emulate appearance of volumetric hair as seen in Figure 7.8 and that it can recover its shape even after violent motions, such as the one shown in Figure 7.9.

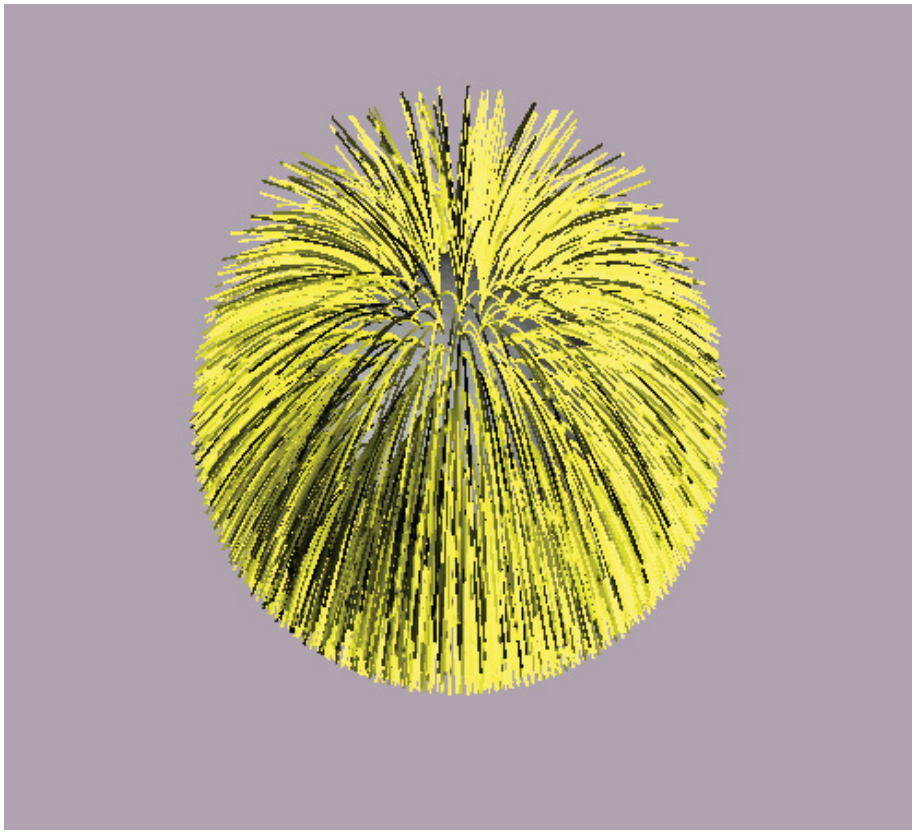


Figure 7.8: Appearance of volumetric hair

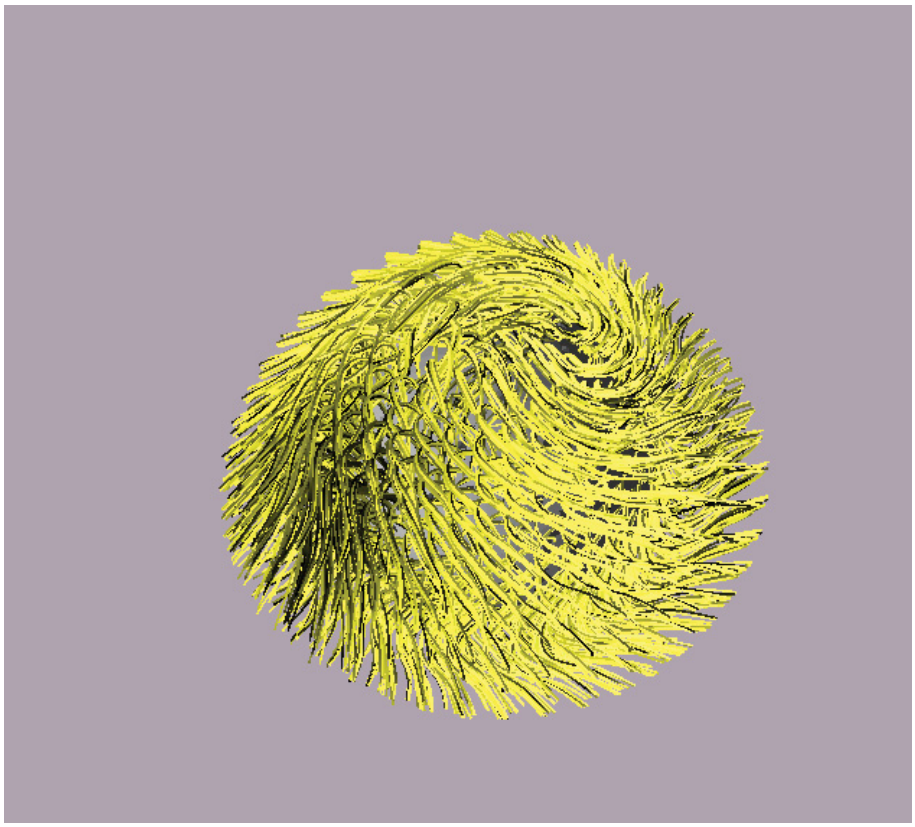


Figure 7.9: Hair undergoing violent motion

Our constraint can also simulate curly hair, as seen in the minimal demo shown in Figure 7.10. Each image is a few frames apart from each other. As we can see, the curls can recover their shape even after undergone severe deformation, even if the strands suffer from sagging. This phenomenon can be corrected if we apply the technique described by Hsu et al. (2023).

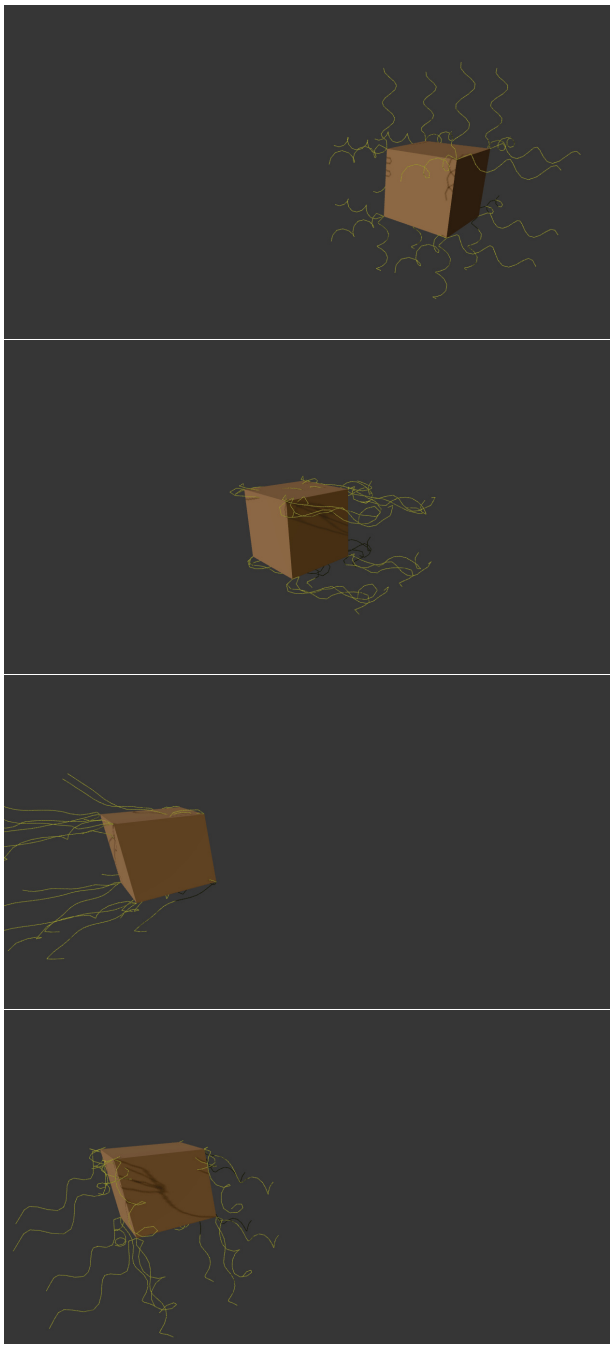


Figure 7.10: Recovery of curls after deformation

We would like to compare our results both in terms of performance and visual quality with the works that closely related to ours (Müller et al. (2012), Han and Harada (2012) and Sánchez-Banderas et al. (2015)), but we couldn't find their implementations, making it impossible to do any type of comparison.

7.3 PUBLIC RECEPTION

In May 2024, we presented our work at *2024 ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*. Even if it was not finished at the time, our work was received with interest by the research community, in so much so that it was awarded the prize for "Best Poster" by community vote (Cavazotti and Guedes, 2024).

7.4 CONCLUSION

We are satisfied with the results we've obtained, both in terms of performance and visual quality. Although the motion produced by our algorithm is not as precise as those produced some other works presented in Chapter 3, it's good enough for real-time applications. The time it takes for our CUDA kernel to simulate more 1.5 million particles makes our algorithm suitable for VR applications, which are known for having strict frame rate requirements.

8 CONCLUSION

In this Masters dissertation we've explained some fundamental concepts of Computer Graphics, then we've presented many works from the same field as our research, with greater focus on techniques based on PBD, which we've also explained in depth. We have identified the need for a comprehensive systematic survey about hair simulation, since the latest work of this kind that we have knowledge of was published almost two decades ago.

We have reached our goal to expand the technique developed by Müller et al. (2012), enabling it to simulate curly hair and other hair styles. We also managed to develop an algorithm that is simple to implement and has good performance, making it suitable real-time applications.

Although the simulation quality of our algorithm is not on par with some other techniques presented in Chapter 3, it received interest from the computer graphics community, showing its potential. In terms of performance, we are very pleased. We've managed to simulate more than 1.5 million particles in such a small amount of time that it is possible to integrate our technique into VR applications. The code we've written and the data we've produced are available at: `https://gitlab.c3sl.ufpr.br/teoria/hair-simulation`.

We've identified some areas of improvement, such as reducing the effects of sagging in order to maintain the original shape of the hair unaffected by its own weight. We can also implement some mechanisms to simulate hair-hair interaction, making our simulations even more realistic. Finally, we could try to implement other PBD-based hair simulation techniques and compare them to ours.

REFERENCES

Akenine-Möller, T., Haines, E., and Hoffman, N. (2018). *Real-time rendering*. CRC Press.

Bertails, F., Audoly, B., Cani, M.-P., Querleux, B., Leroy, F., and Lévêque, J.-L. (2006). Super-helices for predicting the dynamics of natural hair. *ACM Transactions on Graphics (TOG)*, 25(3):1180–1187.

Blender Online Community (2025). *Blender - a 3D modelling and rendering package*. Blender Foundation, Blender Institute, Amsterdam.

Cabello, R. (2010–2025). Three.js. `https://threejs.org/`. Accessed: 2025-04-21.

Catmull, E. and Clark, J. (1998). Recursively generated b-spline surfaces on arbitrary topological meshes. In *Seminal graphics: pioneering efforts that shaped the field*, pages 183–188.

Cavazotti, M. d. M. and Guedes, A. L. P. (2024). Real-time curls. `https://i3dsymposium.org/2024/posters.html`. SIGGRAPH - I3D - Awarded Best poster - audience choice - `https://i3dsymposium.org/2024/awards.html#best-poster---audience-choice`.

Chai, M., Zheng, C., and Zhou, K. (2014). A reduced model for interactive hairs. *ACM Transactions on Graphics (TOG)*, 33(4):1–11.

Chai, M., Zheng, C., and Zhou, K. (2016). Adaptive skinning for interactive hair-solid simulation. *IEEE transactions on visualization and computer graphics*, 23(7):1725–1738.

Emmelkamp, P. M. and Meyerbröker, K. (2021). Virtual reality therapy in mental health. *Annual review of clinical psychology*, 17(1):495–519.

Fei, Y., Maia, H. T., Batty, C., Zheng, C., and Grinspun, E. (2017). A multi-scale model for simulating liquid-hair interactions. *ACM Transactions on Graphics (TOG)*, 36(4):1–17.

Guang, Y. and Huang, Z. (2002). A method of human short hair modeling and real time animation. *10th Pacific Conference on Computer Graphics and Applications, 2002. Proceedings.*

Hadap, S. and Magnenat-Thalmann, N. (2001). Modeling dynamic hair as a continuum. In *Computer Graphics Forum*, volume 20, pages 329–338. Wiley Online Library.

Han, D. and Harada, T. (2012). Real-time hair simulation with efficient hair style preservation. In Bender, J., Kuijper, A., Fellner, D. W., and Guerin, E., editors, *Workshop on Virtual Reality Interaction and Physical Simulation*. The Eurographics Association.

Hoffman, H. G., Boe, D. A., Rombokas, E., Khadra, C., LeMay, S., Meyer, W. J., Patterson, S., Ballesteros, A., and Pitt, S. W. (2020). Virtual reality hand therapy: A new tool for nonopioid analgesia for acute procedural pain, hand rehabilitation, and vr embodiment therapy for phantom limb pain. *Journal of hand therapy*, 33(2):254–262.

Hsu, J., Wang, T., Pan, Z., Gao, X., Yuksel, C., and Wu, K. (2023). Sag-free initialization for strand-based hybrid hair simulation. *ACM Transactions on Graphics (TOG)*, 42(4):1–14.

Hu, Y. (2018). Taichi: An open-source computer graphics library. *arXiv preprint arXiv:1804.09293*.

Hu, Y. (2018–2025). Taichi. `https://taichi.graphics/`. Accessed: 2025-04-22.

Jiang, J., Sheng, B., Li, P., Ma, L., Tong, X., and Wu, E. (2020). Real-time hair simulation with heptadiagonal decomposition on mass spring system. *Graphical Models*, 111:101077.

Liang, W. and Huang, Z. (2003). An enhanced framework for real-time hair animation. *11th Pacific Conference on Computer Graphics and Applications, 2003. Proceedings.*

Macklin, M., Müller, M., and Chentanez, N. (2016). Xpbd: position-based simulation of compliant constrained dynamics. In *Proceedings of the 9th International Conference on Motion in Games*, pages 49–54.

Macklin, M., Storey, K., Lu, M., Terdiman, P., Chentanez, N., Jeschke, S., and Müller, M. (2019). Small steps in physics simulation. In *Proceedings of the 18th annual ACM siggraph/eurographics symposium on computer animation*, pages 1–7.

Marschner, S. and Shirley, P. (2018). *Fundamentals of Computer Graphics, 4th Edition*. A K Peters/CRC Press.

Müller, M., Heidelberger, B., Hennix, M., and Ratcliff, J. (2007). Position based dynamics. *Journal of Visual Communication and Image Representation*, 18(2):109–118.

Müller, M., Kim, T.-Y., and Chentanez, N. (2012). Fast simulation of inextensible hair and fur. *VRIPHYS*, 12:39–44.

Plante, E., Cani, M.-P., and Poulin, P. (2001). A layered wisp model for simulating interactions inside long hair. *Eurographics Workshop on Computer Animation and Simulation (EGCAS)*.

Plante, E., Cani, M.-P., and Poulin, P. (2002). Capturing the complexity of hair motion. *Graphical Models*, 64:40–58.

Renganayagalu, S. K., Mallam, S. C., and Nazir, S. (2021). Effectiveness of vr head mounted displays in professional training: A systematic review. *Technology, Knowledge and Learning*, pages 1–43.

Rosenblum, R. E., Carlson, W. E., and Tripp, E. (1991). Simulating the structure and dynamics of human hair: Modelling, rendering and animation. *The Journal of Visualization and Computer Animation*, 2(4):141–148.

Salomon, D. (2007). *Curves and surfaces for computer graphics*. Springer.

Sánchez-Banderas, R. M., Barreiro, H., García-Fernández, I., and Pérez, M. (2015). Real-time inextensible hair with volume and shape. In *CEIG*.

Selle, A., Lentine, M., and Fedkiw, R. (2008). A mass spring model for hair simulation. In *ACM SIGGRAPH 2008 papers*, pages 1–11.

Smith, V., Warty, R. R., Sursas, J. A., Payne, O., Nair, A., Krishnan, S., da Silva Costa, F., Wallace, E. M., Vollenhoven, B., et al. (2020). The effectiveness of virtual reality in managing acute pain and anxiety for medical inpatients: systematic review. *Journal of medical Internet research*, 22(11):e17980.

Walt Disney Animation Studios (2010). Tangled. Directed by Nathan Greno and Byron Howard.

Walt Disney Animation Studios (2012). Brave. Directed by Mark Andrews and Brenda Chapman.

Ward, K., Bertails, F., Kim, T.-y., Marschner, S. R., Cani, M.-p., and Lin, M. C. (2007). A Survey on Hair Modeling: Styling, Simulation, and Rendering. *IEEE Transactions on Visualization and Computer Graphics*, 13(2):213–234.

Wu, K. and Yuksel, C. (2016). Real-time hair mesh simulation. In *Proceedings of the 20th ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, pages 59–64.

Yang, J. (2024). A survey on hair modeling. *Highlights in Science, Engineering and Technology*, 115:512–526.

Yuksel, C., Schaefer, S., and Keyser, J. (2009). Hair meshes. *ACM Transactions on Graphics (Proceedings of SIGGRAPH Asia 2009)*, 28(5):166:1–166:7.

Zibrek, K. and McDonnell, R. (2019). Social presence and place illusion are affected by photorealism in embodied VR. In *Proceedings of the 12th ACM SIGGRAPH Conference onf Motion, Interaction and Games*, pages 1–7.

APPENDIX A – TEST DATA

In the following pages are all the data we've collected in our tests.

A.1 NAIVE IMPLEMENTATION

Kernel execution time (ms)		Number of hair strands														
		60			240			960			3840			15360		
		Min	Avg	Max	Min	Avg	Max	Min	Avg	Max	Min	Avg	Max	Min	Avg	Max
Number of segments per strand	10	0.029	0.032	0.053	0.064	0.067	0.086	0.070	0.073	0.090	0.074	0.076	0.094	0.218	0.235	0.301
	25	0.072	0.075	0.096	0.174	0.178	0.191	0.188	0.192	0.220	0.196	0.199	0.227	1.022	1.056	1.340
	50	0.143	0.151	0.178	0.350	0.357	0.369	0.377	0.383	0.392	0.207	0.221	0.399	2.370	2.469	2.943
	100	0.288	0.294	0.311	0.710	0.720	0.753	0.682	0.703	0.779	0.416	0.432	0.799	5.165	5.407	7.010
	200				1.485	1.498	1.541	0.809	0.848	1.617	1.406	1.501	1.653			

		Elapsed Cycles					SM Active Cycles					Compute (SM) Throughput (%)					Occupancy (%)				
		60	240	960	3840	15360	60	240	960	3840	15360	60	240	960	3840	15360	60	240	960	3840	15360
Number of segments per strand	10	30339.48	63418.01	69476.58	70578.30	235897.66	3821.82	4883.20	10876.56	35150.12	223554.64	2.78	1.61	2.54	6.55	7.56	44.66	43.29	32.85	27.50	52.08
	25	75267.96	172464.79	180427.33	191547.84	1161280.40	5291.02	8425.36	25030.80	93341.55	1076281.16	1.26	0.84	1.78	6.28	4.04	34.92	35.57	26.52	25.70	49.14
	50	145277.26	340540.61	364950.26	375032.37	2757393.03	7599.93	14120.54	48181.77	185965.12	2567678.75	0.76	0.61	1.65	6.33	3.46	25.30	30.32	24.39	24.40	47.79
	100	283733.10	687173.87	747659.72	759962.72	6083920.85	12142.14	25668.84	95544.50	377392.77	5759462.24	0.51	0.49	1.62	6.24	3.11	18.27	27.22	23.42	23.93	46.69
	200		1450629.03	1540927.83	1573539.16			50943.84	195578.73	782816.46			0.45	1.55	6.01			25.34	22.46	23.49	

A.2 ARRAY OF STRUCTS

Kernel execution time (ms)		Number of hair strands														
		60			240			960			3840			15360		
		Min	Avg	Max	Min	Avg	Max	Min	Avg	Max	Min	Avg	Max	Min	Avg	Max
Number of segments per strand	10	0.029	0.031	0.066	0.066	0.069	0.084	0.071	0.074	0.092	0.074	0.077	0.092	0.193	0.211	0.278
	25	0.064	0.066	0.084	0.156	0.161	0.197	0.169	0.173	0.191	0.174	0.177	0.190	0.564	0.592	0.787
	50	0.122	0.129	0.149	0.315	0.321	0.347	0.339	0.345	0.373	0.183	0.197	0.373	1.415	1.492	1.751
	100	0.260	0.270	0.294	0.642	0.655	0.678	0.598	0.625	0.714	0.369	0.386	0.717	3.371	3.688	3.920
	200				1.341	1.354	1.371	0.735	0.769	1.475	1.311	1.408	1.503			

		Elapsed Cycles					SM Active Cycles					Compute (SM) Throughput (%)					Occupancy (%)				
		60	240	960	3840	15360	60	240	960	3840	15360	60	240	960	3840	15360	60	240	960	3840	15360
Number of segments per strand	10	27738.93	64247.27	68953.16	70974.95	227048.30	3645.51	4844.81	10890.20	35059.24	214531.06	2.94	1.55	2.35	6.21	7.73	50.64	47.63	34.55	29.02	62.93
	25	65353.13	152336.31	163567.11	167878.86	672070.28	4888.89	7782.47	22779.55	82614.17	632337.03	1.40	0.89	1.80	6.87	6.85	39.36	38.30	27.74	25.52	50.01
	50	128281.26	304422.38	328575.22	336110.36	1412837.56	6998.82	12849.01	43468.45	165761.24	1332067.54	0.82	0.64	1.79	6.96	6.61	29.35	32.20	24.77	24.26	46.39
	100	256874.89	619333.94	665922.84	679160.93	2978659.35	11271.94	23349.74	85932.37	336304.27	2807007.08	0.53	0.52	1.77	6.93	6.32	20.62	27.95	23.10	23.56	44.77
	200		1296194.23	1392051.01	1413690.12			45911.17	177360.50	702254.18			0.49	1.70	6.67			25.27	22.05	22.91	

A.3 BATCH

Kernel execution time (ms)		Number of hair strands														
		60			240			960			3840			15360		
		Min	Avg	Max	Min	Avg	Max	Min	Avg	Max	Min	Avg	Max	Min	Avg	Max
Number of segments per strand	10	0.020	0.026	0.050	0.029	0.034	0.064	0.031	0.038	0.058	0.031	0.035	0.057	0.090	0.097	0.120
	25	0.041	0.048	0.082	0.059	0.065	0.079	0.061	0.067	0.080	0.066	0.070	0.085	0.117	0.127	0.252
	50	0.082	0.089	0.100	0.113	0.119	0.135	0.115	0.123	0.135	0.070	0.080	0.137	0.448	0.466	0.490
	100	0.162	0.179	0.209	0.216	0.224	0.240	0.223	0.232	0.254	0.133	0.142	0.260	0.504	0.639	0.940
	200				0.422	0.439	0.456	0.239	0.261	0.459	0.262	0.273	0.477			

		Elapsed Cycles					SM Active Cycles					Compute (SM) Throughput (%)					Occupancy (%)				
		60	240	960	3840	15360	60	240	960	3840	15360	60	240	960	3840	15360	60	240	960	3840	15360
Number of segments per strand	10	19943.71	27249.44	28352.74	31212.18	84426.88	3418.75	3664.59	5851.91	14758.71	73878.10	4.12	3.64	5.67	14.44	20.98	54.27	55.27	44.05	34.43	49.37
	25	44157.90	57747.78	60413.48	64840.68	201641.65	4225.58	4685.99	9941.03	31079.65	180138.91	2.06	2.30	4.89	17.98	23.02	45.11	48.44	35.34	29.09	48.06
	50	84157.66	108916.50	114159.13	121931.89	399071.96	5556.01	6383.98	16787.14	58906.40	361843.26	1.25	1.75	5.21	19.31	23.49	35.81	42.06	30.29	26.82	47.37
	100	163119.06	211220.90	220554.27	234984.91	796037.92	8186.30	9769.35	30560.45	114629.80	722769.60	0.82	1.52	5.39	20.10	23.73	26.33	36.00	26.93	25.62	47.33
	200		419122.43	434450.92	463485.91			16664.03	58251.71	227064.65			1.52	5.47	20.42			31.24	24.98	25.01	

Proportional length error (total actual length / expected total length)		Number of hair strands														
		60			240			960			3840			15360		
		Min	Avg	Max	Min	Avg	Max	Min	Avg	Max	Min	Avg	Max	Min	Avg	Max
Number of segments per strand	10	0.000%	0.150%	6.796%	0.000%	0.118%	5.842%	0.000%	0.110%	5.696%	0.000%	0.107%	5.694%	0.000%	0.108%	5.689%
	25	0.000%	0.188%	3.480%	0.000%	0.179%	2.943%	0.005%	0.177%	2.952%	0.006%	0.177%	3.005%	0.006%	0.176%	3.055%
	50	0.000%	0.162%	1.704%	0.000%	0.166%	1.442%	0.001%	0.164%	1.446%	0.002%	0.164%	1.472%	0.002%	0.164%	1.496%
	100	0.000%	0.113%	0.844%	0.000%	0.122%	0.714%	0.001%	0.121%	0.716%	0.000%	0.121%	0.728%	0.001%	0.121%	0.741%
	200				0.000%	0.078%	0.408%	0.000%	0.081%	0.373%	0.000%	0.081%	0.372%			