Real-Time Wet Hair Simulation using Afro-Textured Hair

Iris Onwa

A Dissertation

Presented to the University of Dublin, Trinity College

in partial fulfilment of the requirements for the degree of

Master of Computer Science

Supervisor: Michael Manzke

April 2025

Real-Time Wet Hair Simulation using Afro-Textured Hair

Iris Onwa, Master of Computer Science

University of Dublin, Trinity College, 2025

Supervisor: Michael Manzke

The upkeep of afro-textured hair has had a profound impact on African and coily-haired people, requiring constant care and preening to ensure it remains healthy. This comes through the use of various oils, conditioners, and shampoos. Afro-textured hair is also much drier and porous than its counterparts, which leads to interesting interactions with these fluids.

However, simulations of these interactions are sparse. Simulating afro-textured hair accurately has an impact on performance, as it tends to be far denser than straighter hair, leading to constant collision handling. Therefore, any simulation involving both hair and fluids must make sacrifices in accuracy for performance. These sacrifices are great when implemented in real-time applications, such as video games.

We present an application that showcases the interactions of afro-textured hair and fluids in real-time while minimising a loss in accuracy. We will use the Position-Based Dynamics (PBD) framework, which trades some accuracy for performance, for both the hair and fluid simulations. Cosserat theory will be used to implement angular constraints to ensure hair strands remain coily. To boost the application's performance and obtain more accuracy, we will implement both simulations on the GPU, allowing for immense parallelisation. Our application can be run on low-end consumer devices such as laptops, proving its low performance impact. Future work lies in improving the accuracy of the simulation while balancing performance.

Acknowledgments

I would like to thank my supervisor, Michael Manzke, for his support while working on this project. I would also like to thank my friends and family for supporting me throughout my years of college. I never would have been able to complete this dissertation without them.

Iris Onwa

University of Dublin, Trinity College
April 2025

Contents

List of Tables

List of Figures

List of Algorithms

Chapter 1

Introduction

In this dissertation we will introduce a method of simulating wet, afro-textured hair in real-time, providing a method that can be applied to a wide variety of solvers and software.

1.1 Motivation

Hair simulations have been a long sought-after feature in many modern games and applications. In particular, real-time simulations are considered a staple of the field, with heavy optimisation needed to run a simulation effectively. However, very little work has been done on simulating different types of hair. Many simulations feature predominately straight or lightly curly hair, but simulations of more tightly coiled hair remain sparse. (Darke et al., 2024)

One significant cause of this is angle constraints. Straight hair strands are trivial to implement in most physics solvers as they can be described as a simple series of springs separated by masses. However, introducing angle constraints to slender rods such as Cosserat rods has been implemented in many physics solvers in modern game engines.

The second issue is performance. Performance issues plague many attempts due to the density of afro-textured hair. Since it coils into itself instead of falling "out" of the head, most of the hair is constantly colliding. Simulating these collisions accurately every frame would drastically reduce the possibility of running the simulation in real-time.

The third issue is the dynamics of the hair. Afro-textured hair is often much lighter than its counterparts, leading it to remain mostly stationary when a person is moving. Due to this, there is very little reason to model the dynamics of each hair strand, since most will not change from their initial configuration if left alone, except under gravity.

These issues have lead to researchers neglecting the study of methods that combine hair simulations with other materials, such as fluids. Afro-textured hair is a very unique

type of hair regarding interactions with fluids. It is denser and more porous on average than straighter hair. Due to its low weight, wet hair will clump outwards, providing a spiky appearance. The natural dark colour of most afro-textured hair strands also means that its physical properties are the only source of telling whether hair strands are wet, instead of how lighter hair has colour. In addition, there are numerous styling products such as shampoos and conditioners that modify the texture and shape of hair. Simulations of this type exist for straight and curly hair (Ward et al., 2007b), but not for coily hair.

1.2 Research Question

The goal of this dissertation is to explore the feasibility of simulating wet, natural, afro-textured hair in real-time. By "natural", we refer to an afro hair style without braiding or locing. [1] We seek to determine the different ways this simulation can be implemented, the performance of such a method, and how our dissertation can aid future research in the field of real-time simulations.

1.3 Terminology

Human hair varies greatly from person to person. Determining the exact dynamics of any one person's hair would require knowing about their precise physical situation, such as their diet, exercising habits, and genetics. While some of these could be applicable to a video game, such as a game that simulates the human condition, for real-time contexts, only genetics are concerned. In this regard, when ignoring density and porosity, the curl of the hair strand is its most important feature. These features are predominant in afro-textured hair and are not features we wish to ignore in our implementation, but for the sake of explanation, we will them when talking about the types of hair strands.

By far, the most popular categorisation of hair types is Andre Walker's hair typing system. Zitz (2022) The system has received changes over time and can be considered controversial for focusing purely on the curliness of hair, but as mentioned, this is the most important feature for game designers.

The system can be split up into four main categories each with three subcategories, as shown in Figure 1.1. The first types, 1A, 1B, and 1C, all describe predominantly straight hair, with very little difference in curl between them. The largest difference is the thickness of each subcategory. The next set, 2A-2C, all describe "wavy" hair, with 2C having the most curl.

[1] The formation of dreadlocks, colloquially referred to as "locs".

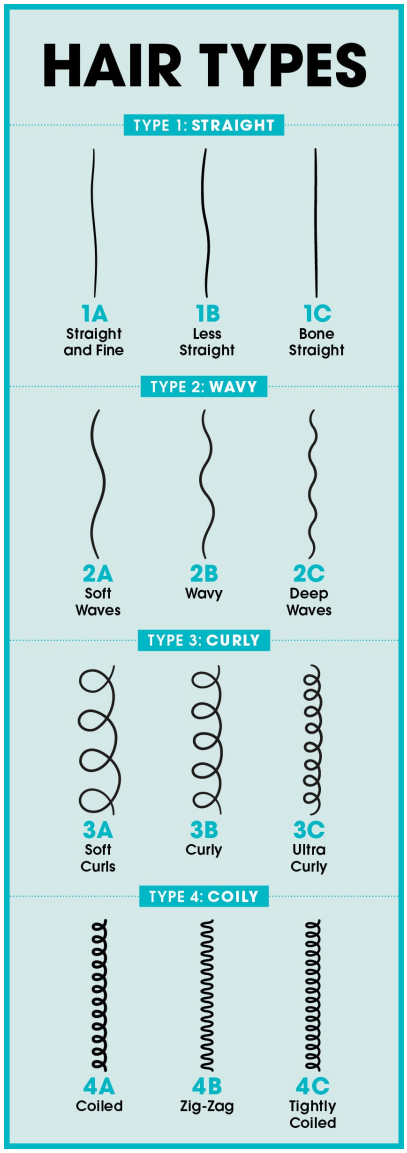


Figure 1.1: A chart of hair types. Image taken from Prevention Zitz (2022)

From here, hair strands begin to curl enough to be described as helices. 3A-3C describe hair curly enough to defy gravity; hair strands begin to grow "out" more than "down". Finally, 4A-4C defines even curlier hair, which can be described as "coily". 4C hair is considered the curliest type of hair. For the purposes of this dissertation, we will describe 3A-4C hair as "afro-textured", as these hair types share the property of having defined curls and are much lighter than 1A-2C hairs.

It is important to note that the term "afro-textured" is being used to describe hair rather than "African", as hair types of African people are not necessarily similar. An African woman may naturally have 4C, 3B, or even 1C hair, however rare that may be. Cloete et al. (2019) Therefore, we use "afro-textured" to specify the curly to coily hair that is the focus of this dissertation.

1.4 Structure

This document will be structured as follows. First, in Chapter 2, we will explore the state of the art in real-time hair and fluid simulations and how they are coupled. Chapter 3 will go more in depth on the theory behind these simulations and the terminology behind them. Chapter 4 will explain the high-level overview of our implementation and why it was chosen. Chapter 5 will explain our implementation in more detail, providing diagrams and code snippets to show how our work was done. Chapter 6 will answer our research question and evaluate how well our solution holds up and what, if anything, went wrong. Finally, Chapter 7 will explore what can be improved.

Chapter 2

The State of the Art

In this section, we will discuss the state-of-the-art of real-time hair and fluid simulations. Both types of simulations have a deep history and have resulted in numerous methods in performing simulations. Each of these techniques have their own uses, from a simple simulation of points on a string to simulating the movement of stars.

The average human head has roughly 100,000 hairs on their scalp. Milo et al. (2009) Rendering and simulating each of these is a monumental task, even for modern devices. Most research in this field has been on straighter hair, which bears the simplest representation of a strand of hair, treating it as a series of point masses connected by strings. Comparatively little research has been made on curlier hair, especially in real-time contexts.

Fluid simulations have a rich history, with different types of implementations depending on the use case. Popular methods range from treating cells on a grid as discrete packets of fluid, to considering them as smaller particles defined in free space, as well as combinations of the two. This section will briefly explore this history in chronological order and discuss their use cases.

2.1 Hair Simulations

The simplest method of discretising a hair strand is to break it up into point connected by lines. Each point, or vertex, determines the mass of the strand at that point. We refer to the line segments between vertices as "rods". Human hair is not inelastic, but even a little elasticity is noticeable by viewers, so the stretching of rods is typically prohibited. Angle constrains can also be introduced to allow for a natural curly rest shape. A hair simulation, then, dictates how these point masses move and how the rods between them constrain their lengths and angles. Several methods were detailed in Ward et al. (2007a),

which we will outline below. Despite the age of survey, these methods provide insight into how hair simulations have been traditionally performed and form a basis for how we will discuss the methods used in this paper. (Modern methods involving deep learning and reconstruction from images are outside the scope of this paper.) We will discuss methods to solve these constraints below.

2.1.1 Volumetric Methods

Before delving into strand-based methods, one popular method of simulating hairs that is important to note is volumetric simulations. Specifically, simulations that treat the entire hair as a mesh, either as a single object or as a group of objects, usually cylinders or tetrahedra. These methods are popular as simulating individual strands can be very time-consuming for real-time, especially for physically-accurate simulations. By simulating hair in bulk, the computation time and complexity can be reduced drastically, at the cost of a stiff-looking appearance.

For example, Scheuermann (2004) used polygonal patches to create surface meshes and applied textures onto the surface to represent hair. Yuksel et al. (2009) patented Hair Meshes, which allowed for more freedom in modelling hair by allowing users to create dynamic meshes that represented the volume of hair. In 2024, they added real-time capability to the program by offloading the calculations to the GPU and introducing level-of-detail (LOD) systems. (Bhokare et al., 2024)

Afro-textured hair is extremely coily and often coils into itself, creating a cluster of hair that would be simple to render using volumetric methods. Patrick et al. (2004) used volumetric methods to generate natural and braided afro-textured hair. They also provided an explicit method for generating straightened hair. [1] Coilier hair, such as 4C hair, could also cheaply be simulated using a slightly elastic soft-body system.

2.1.2 Mass Spring System

An early method for developing interactive hair simulations was created by Rosenblum et al. (1991). Rosenblum's method discretised hair strands by representing points as masses and rods as stiff linear springs. To constrain rod angles, they used another angular spring between rods. While this system works for straight hair, it fails to capture bends and twists correctly, with added angular springs increasing the complexity of the system. The stiffness of the system can also cause instabilities that require a small time-step to fix,

[1] Afro-textured hair can be straightened using a technique called "relaxing", where the coiliness of the hair is removed using heat or creams. This process is difficult to reverse and usually requires cutting the relaxed hair off so new growth returns the coils.

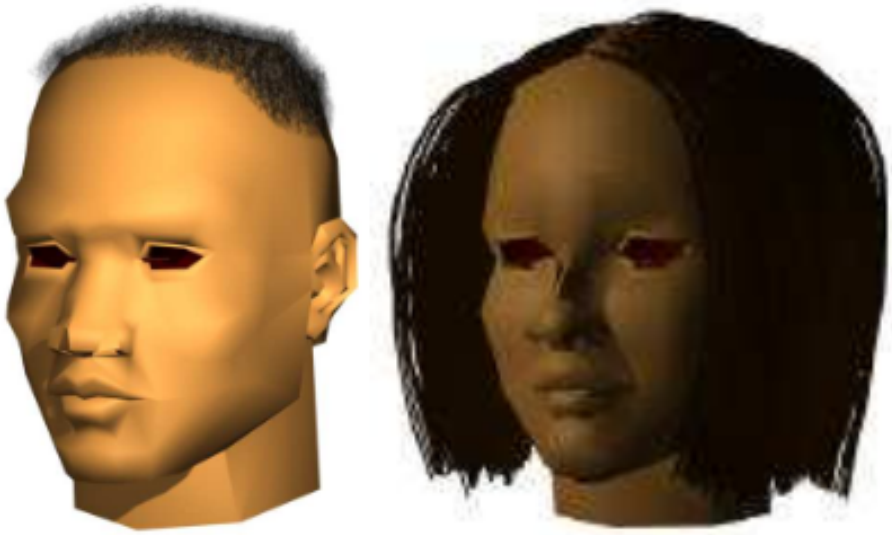


Figure 2.1: Rendered African hair. Image taken from (Patrick et al., 2004)

slowing down the simulation. Daldegan et al. (1993) added basic collisions by simulating rods as simple cylinders. Selle et al. (2008) used tetrahedra to model hair vertices by placing altitude springs on each edge of tetrahedron and solved a problem present in some volumetric simulations where the volume of the tetrahedra could collapse to zero or negative volume, breaking the simulation. This method allows for bending and twisting of hairs, as well as the ability to simulate moderately curly hair.

Despite Selle's work, the helices we seek to create are not possible with mass-spring systems. While curly hair can certainly be approximated ((Müller et al., 2012), (Li et al., 2016)), they can still only create curls on the order of 2A-2C. These curls are also only rendered using additional, non-functional vertices. The constraints required to create genuine coils have yet to be found with this method.

2.1.3 One-Dimensional Projective Equations

Projective dynamics were proposed by Anjyo et al. (1992) as a simplified simulation of a cantilever beam. [2] Each rod was initialised as a beam starting from the previous rod, with the root starting from the head. External forces were applied to its azimuth and zenith, which had to be solved using differential equations at each time-step. While this enabled inextensibility and the ability to recover bending angles, torsion was not possible.

[2] A horizontal structural beam with one fixed end.

2.1.4 Kinematic Chains

To allow torsion, forward kinematics have been proposed as an alternative to 1D projective equations. Each hair strand can be considered as a kinematic chain, similar to an arm or leg, where each vertex is considered as a rotational joint. (Chang et al., 2002) This method, well-studied in the field of robotics, enables bending and twisting while constraining stretching, but does not allow for suitable curl.

2.1.5 Cosserat Theory

Cosserat rods can be discretised implicitly or explicitly. Below, we will provide a brief explanation of Cosserat theory and discuss different methods of discretisation and how they fall under each category. We will go more in depth on Cosserat theory in Section 3.6.

Implicit discretisation

Cosserat rods were first introduced to the graphics community by Pai (2002), who used them to simulate sutures and threads in the medical field. The Cosserat theory of rods is based on Kirchhoff theory. We can describe a Kirchhoff rod as a slender, inextensible rod that passes through a centreline with a material cross-section orthogonal to the centreline at any point. Each cross-section, or frame, contains a Darboux vector that dictates how the rod bends at that frame. Cosserat rods add stretching and shearing to Kirchhoff rods.

Bertails et al. (2006) introduced "super-helices" in order to address issues with mass-spring systems, 1D projective dynamics, and kinematic chains; namely that they do not allow for suitable curl. They use Kirchhoff theory for simulating hair by treating each strand as a Kirchhoff rod. This method allows for the simulation of rigid coils in real-time, albeit slowly. They later improved on this through the use of super-helices that can run with a linear time-complexity. (Bertails, 2009)

These methods are referred to as "implicit" because they use curves that only represent the centreline. The actual centreline needs to be recovered at the end of the time-step by integrating the rod's frames from end to end. An alternative method is explicit discretisation, shown below, which uses the actual discretised frames of the rod and can provide a more dynamic simulation that makes determining collisions simpler.

Explicit discretisation

Spillmann and Teschner (2007) builds on super-helices with Cosserat Rod Elements (CoRdE) by providing a more robust self-collision system and accompanying each frame

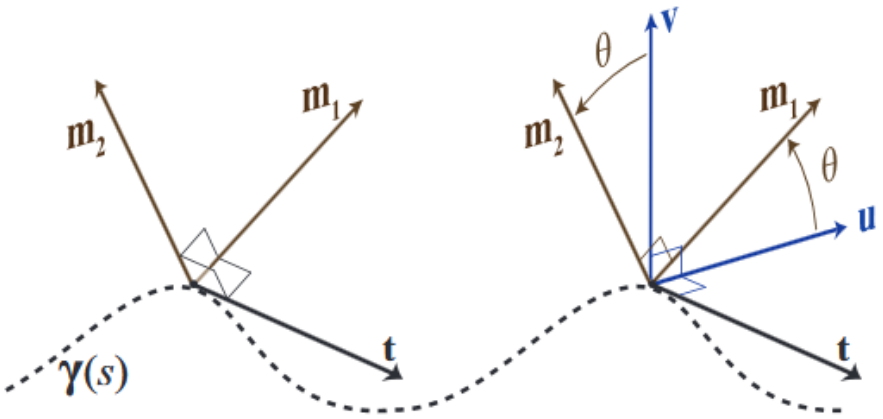


Figure 2.2: A representation of a Kirchhoff rod. $\gamma(s)$ represents the centreline, and the material frame is represented as $\{t(s), m_1(s), m_2(s)\}$. The Bishop frame is given as $t(s), u(s), v(s)$; θ is the removed twist from the frame. Image taken from (Bergou et al., 2008)

with a quaternion to represent an orientation. This method also replaces Kirchhoff rods with Cosserat rods, as Kirchoff rods suffer from stiff equations.

A very popular explicit discretisation of Cosserat theory comes from Discrete Elastic Rods (DER). (Bergou et al., 2008) They return to Kirchhoff rods and explicitly model the centreline, again using quaternions to aid in edge orientation. They define the Bishop frame as the material frame without twist and use it to find the curvature bi-normal along the centreline. Then, by taking the Darboux vector of the Bishop frame, they can use parallel transport to move a vector, such as a twisting force, along the rod.

Umetani et al. (2015) used their position-based elastic rod model. They used Position-Based Dynamics (PBD) to discretise a Cosserat rod into a set of particles connected by constraints. By defining the stretching, shearing, bending, and twisting forces as PBD constraints, they could calculate a correction displacement to add to each particle. The particle velocities are then updated using Verlet integration. Since PBD represents all bodies with particles, they represented frames with ghost particles orthonormal to the rods between particles. This way, they fit Cosserat theory into PBD. This method provides a fast and stable implementation of Cosserat rods that can be used in real-time at the cost of accuracy.

Kugelstadt and Schömer (2016) provided a similar method with their position- and orientation-based Cosserat rod model (POCR). Instead of trying to fit Cosserat theory into PBD, they worked to modify the PBD algorithm to define material frames directly. Instead of attaching a quaternion to each frame, the quaternions are the frames. Constraints are now handled between particles and quaternions to ensure unit length. This

reduces the number of particles required for the simulation, which can improve performance even further.

Deul et al. (2018) used Extended Position-Based Dynamics (XPBD) to provide a direct solver for the stretching and bending constraints. They combined the constraints into one central model and modified the XPBD model to use Karush–Kuhn–Tucker systems and Newton iterations instead of the traditional XPBD non-linear Gauss-Seidel iterations. (See Section 3.2 for an explanation on Gauss-Seidel solvers.)

Recently, Shi et al. (2023) released a method for generating and simulating tightly coiled hair. Their method, while not suitable for real-time applications, provides an excellent appearance for tightly-coiled hair. They provide a geometric method for improving the appearance and realism of their generated hair strands in Wu et al. (2024).

Overall, Cosserat theory is an ideal method of modelling physical rods as they allow for stretching constraints and bending and twisting constraints, all of which are necessary for modelling helices such as coily hair. The combination of Cosserat theory and PBD allows for stable, efficient, and visually plausible simulations without sacrificing too much accuracy. Furthermore, methods involving PBD can easily be applied to other solvers when following Macklin et al. (2014), which shows how to combine PBD fluids and rigid bodies. This makes determining the interaction between PBD particles simple to implement.

2.2 Fluid Simulations

In nature, fluid dynamics are governed by sets of physical rules. The Navier-Stokes equations provide accurate and physical rules for these dynamics, which can then be used for solving many real world problems, such as modelling air flow around an object or liquids through a pipe. However, solving these equations often requires an analytical solution which, while accurate, will run far too slowly for our purposes. Therefore, many mathematicians and engineers developed methods of deriving numerical solutions to the equations.

The majority of fluid simulations are either Eulerian, Lagrangian, or a hybrid of both. Gross and Pfister (2007) describe Eulerian simulations as simulations that discretise the *space* the fluid flows through. For simplicity, this is almost always a grid. Fluids can then be described in terms of each grid cell. This is similar to a person or camera from a fixed position watching fluid flow through a system. Conversely, Lagrangian simulations discretise the fluid itself, typically generalising it as a series of discrete parcels or "particles". These are also known as "meshfree" methods, as they do not require particles to be connected. Hybrid methods use both methods: individual fluid particles move according to their own velocity and that of the underlying grid.

2.2.1 Computational Fluid Dynamics

Computational Fluid Dynamics (CFD) is one of the oldest computational methods for simulating fluids. It uses numerical analysis to accurately simulate situations involving fluid flow. CFD methods are based on the Navier-Stokes equations, which define laws for the motion of viscous, incompressible fluids. During the mid-20th century, Francis Harlow and other researchers at the Los Alamos National Lab developed several methods for two-dimensional fluid flow. Their research led to the development of two important algorithms: Particle-in-Cell (PIC) (Harlow et al., 1955) and Marker-and-Cell (MAC) (Harlow and Welch, 1965).

Particle-in-Cell

PIC is a hybrid Eulerian-Lagrangian method. Particles add their velocity to the grid, which balances itself and makes itself incompressible, before the average velocity of each grid cell is added back to each particle it contains. The grid used is a MAC grid. A *staggered* MAC grid contains information about the fluid, such as the density and pressure, as the centre of each grid cell. Velocities are kept at the faces of each cell, which makes it easier to determine the incoming and outgoing fluid flow at each grid cell. This makes enforcing incompressibility much simpler. An alternative is a *co-located* grid where velocities are stored as the centre. This makes the code easier to write but harder to derive for more complicated boundaries. (Ferziger et al., 2020)

Fluid-Implicit Particle

PIC's faults lay in the damping caused by the repeated velocity transfers between the grid and the particles. Brackbill et al. (1988) created the Fluid-Implicit Particle (FLIP) method to acknowledge this. FLIP makes a copy of the grid before adding particle velocities and makes the grid incompressible as normal. However, it then uses the difference in velocities of the old and new grid to apply velocities back to the particles. This method solves the damping issue but amplifies a "ringing instability" that was present in PIC where data sampled at higher frequencies are indistinguishable from those sampled at lower frequencies, causing them to be missed and large density fluctuations to occur.

One option of fixing this is to use a blend of the PIC and FLIP velocities when simulating. Zhu and Bridson (2005) use this method to simulate sand like a fluid. This hybrid PIC/FLIP method has gained widespread use in software such as Unreal Engine's Niagara Fluids simulator. (Epic Games, 2025a)

However, PIC/FLIP still contains the issues of both PIC and FLIP individually; their combination only mitigates them. Modern advancements in Eulerian-Lagrangian hybrid

simulations improve the performance, stability, and appearance of the fluid. For example, the Affine Particle-In-Cell (APIC) method uses a "locally affine" velocity field to preserve individual particle velocities. (Jiang et al., 2015) The Polynomial Particle-In-Cell method (PolyPIC) generalises APIC to provide a general affine field to for any property. (Markidis et al., 2018)

2.2.2 Material Point Method

The Material Point Method (MPM) arose out of a desire to fix the ringing instability issues present in FLIP. It allows for the simple coupling of multi-phase materials, combining solids and fluids by default. It works by assigning each particle or "material point" a certain property, such as volume, stress, or temperature. Particles are stored on a grid made up of fixed nodes that transfer their information to adjacent nodes. The algorithm is very similar to other Eulerian-Lagrangian methods: particles apply their velocities to the grid, which calculates pressure gradients and updates itself using Newton's Second Law of Motion, before transferring data back to the particles.

The Moving Least Squares Material Point method (MLS-MPM) follows as a generalisation of APIC that uses an alternative strategy to make calculations easier to implement. (Hu et al., 2018) It also avoids the need to compute pressure gradients, making it even faster than regular MPM. (nialltl, 2019)

2.2.3 Smoothed Particle Hydrodynamics

In 1977, Lucy and Monaghan *et. al.* separately discovered the field of Smoothed Particle Hydrodynamics (SPH) (Lucy, 1977; Gingold and Monaghan, 1977). SPH is a Lagrangian method originally used for astrophysics simulations. It works by approximating a continuous function, usually in the form of a partial differential equation (PDE), using a smoothing kernel with a given smoothing radius. Elements outside this radius are entirely unaffected by the sampled particle, but elements within the radius are affected differently depending on their distance from the central particle.

Throughout the late 20th century, Monaghan and others made incremental progress in developing an incompressible SPH method. In 1992, Monaghan released another paper refining SPH and allowing it to form free surface flows. Monaghan et al. (1994) It also came with better defined smoothing kernels, using a cubic spline kernel instead of a Gaussian kernel. However, although this version of SPH was better suited for simulating a fluid, it couldn't do so without visibly obvious problems. Monaghan's free surface has density and pressure issues, leading to fluctuations near the fluid boundaries.

At the turn of the century, more focus was put on fixing these issues while retaining an incompressible free surface. Müller et al. (2003) produced a variant of SPH with different smoothing kernels that could run at interactive rates. This paper paved the way for the use of SPH methods in real-time games and simulations. Müller continued his work on interactive SPH, introducing coupling with deformable solids (Müller et al., 2004) and different fluid densities (Müller et al., 2005), as well as non-rigid fluids such as elastic and melting solids (Müller et al., 2004). Keiser et al. (2005) followed up on this by providing a unified Lagrangian framework for phase-transitions between solids and fluids, as well as providing surface generation to the fluid. Clavet et al. (2005) used additional "near density" and "near pressure" terms to prevent particle clustering between particles with higher local densities.

Becker and Teschner (2007) produced a weakly-compressible method (WCSPH) by forgoing the usual goal and allowing for small density fluctuations. It uses the Tait equation, which defines a relationship between density and pressure, and a high speed of sound, which is used to determine the speed of propagation of information throughout the fluid, to obtain low density fluctuations. However, it also needs to solve a pressure Poisson equation (PPE), which often requires smaller time-steps to run accurately. This could make it too slow for real-time applications.

Solenthaler and Pajarola (2009) instead use a predictive-corrective incompressible SPH method (PCISPH) to propagate a predicted density throughout the fluid for each particle that needs to update its pressure. It uses an iterative solver to solve the pressure equations until they are below some error. Not needing to solve PPEs improves the performance by an order of magnitude over WCSPH. However, the convergence of the algorithm still had room to improve.

Macklin and Müller (2013) use the PBD framework to create Position-Based Fluids (PBF). PBF forgoes calculating pressure entirely as the PBD framework operates on particle positions directly. The density of each particle is solved with a constraint that, when solved, yields a vector displacement to each particle. The particle's velocities are then updated using Verlet integration. This method improved on PCISPH by avoiding the need to calculate forces, at the cost of accuracy.

Ihmsen et al. (2014) use an implicit incompressible SPH method (IISPH). It returns the PPEs but discretises it, along with the continuity equation, which enforces incompressibility. The PPE allows for a more uniform pressure field, which allows the solver to converge far faster than PCISPH.

Bender and Koschier (2015) proposed Divergence-Free SPH (DFSPH) which sought to create a divergence-free velocity field. Their constant density solver stabilises any numerical errors that might arise from the incompressibility solver. Solvers determine a

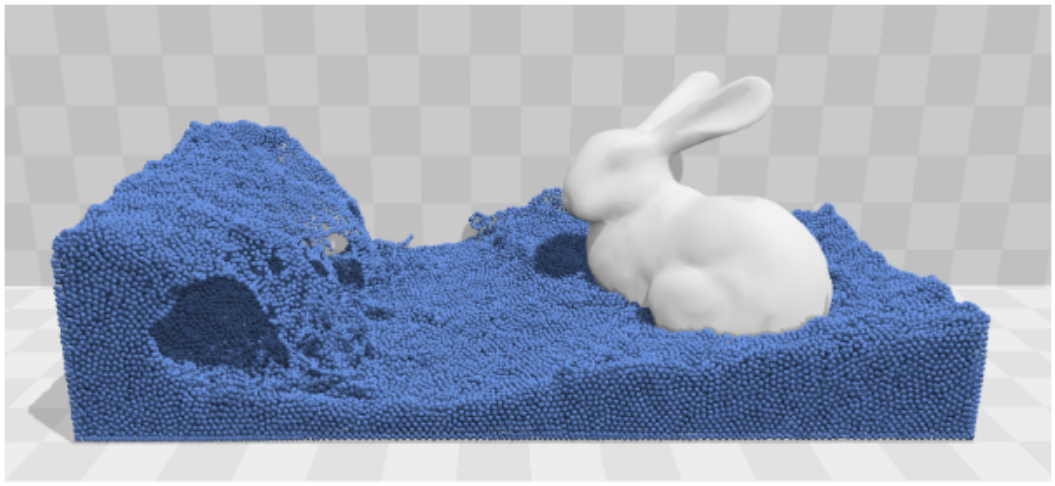


Figure 2.3: Position-Based Fluids with collision detection. Image taken from (Macklin and Müller, 2013)

local stiffness coefficient to calculate pressure for each neighbourhood of particles, then compute each of these neighbourhoods in parallel. By lowering the density error, less iterations are needed, which improves performance over PCISPH, IISPH, and PBF.

Li and Li (2023) recently published a method that removes iterations by determining a variable smoothing length (VSLSPH) for each particle. This improves the performance and accuracy over previous methods, but incurs a large computational cost when determining nearest neighbours.

SPH has had a great deal of research put into it, which is still ongoing. Its methods are simple to understand and can be applied to a wide range of situations, including simulating simple hairs.

2.3 Hair-Fluid Coupling

Hair simulations are the most visually interesting when viewed while being affected by external forces, such as wind or gravity. The interaction of hair and fluids has received some traction in recent years, but most studies instead focus on the general interactions of porous materials, such as sponge and cloth. Though we are focusing on hair for this paper, we will briefly outline these prior methods below, as they lay the foundation behind the ideas we will explore later.

2.3.1 General Permeable Media

Zhu et al. (1999) provided a pore-scale fluid simulation at the microscopic level. They used Darcy's law for modelling the flow throughout the porous material and chose SPH to

model the fluid. To deal with the boundaries in the porous media, they placed additional SPH particles that would be included in density calculations, and that would contain their own densities and velocities. Morris et al. (1999) provided a similar method that could be executed in parallel. These methods only simulated porous flow within the media itself, and did not account for macroscopic flow on the surface. The high accuracy required for these simulations also made them undesirable for real-time applications.

Lenaerts et al. (2008) provide a simulation on the macroscopic level. They build on Zhu and Morris' work by simulating porous float within the object and on the surface, providing a two-way rigid-fluid coupling between the fluid and the material. To do this, they discretise the material and add *porous particles* which contain their own porosity and permeability. These quantities could be used to calculate the pore velocity at a porous particle, which allowed for the modelling of emission to push fluid particles out of the porous media.

2.3.2 Porous Hair

Rungjiratananon et al. (2012) used these methods to focus purely on porous hair, again using SPH. They modelled the fluid flow using a Cartesian grid to determine the porous regions of wet hair, similar to an Eulerian fluid simulation. Fluid flowing onto the hair sticks to it with adhesion forces and gradually builds. Water is diffused to nearby grid segments or "voxels" over time and the mass of water increases as more flows onto it. Eventually, when the mass of the accumulated hairs grows to a certain value, it drops out of the voxel. This method used a particular type of hair strand representation and isn't easily generalised to other models. It also doesn't model forces for the hair strands, instead defining purely porous flow.

Lin (2014) generalised their method by directly defining and manipulating forces between the hair and fluids. They sample porous particles along the rods in hair strands and ignore emission, since hair strands have no real "inside" or "outside" when simulated on a macroscopic scale. They used Akinci et al. (2012) to add solid-fluid coupling and Akinci et al. (2013) to add self-adhesion between porous particles. Their method also specifies a self-adhesion force between porous particles to enable hairs to clump together. This force would then be transferred back to the hair vertices near each porous particle. This clumping force is also not applied when the hair is underwater, which is determined by a porous-fluid density attenuation factor. This method is much easier to apply to other hair models such as Discrete Elastic Rods as it only requires the overall shape of each hair strand.

Fei et al. (2017) provided a multi-scale model for simulating wet hair by simulating both the micro- and macroscopic levels of fluid-hair interactions. They used the APIC fluid simulation method instead of SPH, although they mention that their method should be generalisable to any fluid simulation. They also specify more physically-accurate phenomena such as capillary bridges that form between wet hairs, instead of clumping forces. While their methods are more accurate, they lose the real-time aspect, reporting time-step sizes of up to 57 seconds as opposed to the much smaller time-steps available in previous methods.

For realistic porous flow to be simulated, individual flow within hair strands must be considered. The real-time methods mentioned above provide methods of doing this, but either have tight memory requirements or only apply to particular types of hair. Care must be taken to provide a real-time simulation that can be generalised to any device. Afro-textured hair becomes clumpy when wet, but the amount it droops down by decreases with how coily the hair is.

2.4 Hair Rendering

Afro-textured hair has many bends and turns, so the sleek shine present in straighter hair isn't as visible. While specular highlights are more prevalent in wet hair, as 1A-2C hair becomes much straighter when wet, while 3A-4C hair has a very diffuse surface, albeit with some added shininess. Despite this, we will need to make considerations for how wet hair is rendered. In this section, we will briefly mention some common methods for rendering hair.

2.4.1 Dry Hair

One of the first comprehensive models for rendering hair came from Kajiya and Kay (1989). In their paper, they introduced texels, which were volumetric representations of geometry that could be used to determine how to scatter light. They demonstrated this by using texels to render fur and considered hairs as "infinitely thin cylindrical surface[s]," similar to modern methods. Their rendering algorithm was contrasted against Phong shading. Instead of using the normal vector of the surface, they used the tangent vector along the hairs. Their reasoning was that since a single texel would be used to represent a single hair strand, the function for determining reflectance, or bi-directional reflectance function (BRDF) would be the same everywhere, and thus have the same normals. Marschner et al. (2003) improved Kajiya and Kay's rendering model by predicting scattered light on rough cylinders, where Kajiya and Kay assumed smooth cylinders.

Figure 2.4: A sequence of images of hair with gradually increasing wetness. Image taken from (Gupta and Magnenat-Thalmann, 2007)

2.4.2 Wet Hair

Bruderlin (1999) first created a method for generating wet animal fur. Fur can clump with nearby "clump hairs" by some rate and by some amount. Ward et al. (2004) approximated the Phong illumination model by decreasing the colour of hair by a wetness parameter. Since wet hair also forms around a hair strand, the shininess of it increases, so the specular exponent increases with it. Gupta and Magnenat-Thalmann (2007) follows suit with similar changes when darkening hair and increasing specularity, although they use Marschner's model instead of Phong shading.

2.5 Summary

In this section, we have discussed various methods for simulating fluids and hair, as well as methods on rendering hair. This has not been an exhaustive catalogue, but rather a constrained list of closely-related methods to the core purpose of this dissertation. Each method in each section has their purposes, strengths and weaknesses, but certain methods fit our use-case more than others.

In terms of hair simulations, we would prefer a simulation that is fast and stable, rather than highly accurate. We also want to be able to model tight, stable, coils with the ability to bend and twist. This leads us to the methods involving Cosserat rods implemented using Position-Based Dynamics, the combination of which provides the best real-time performance, flexibility, and interactivity. Of these, Position and Orientation-Based Cosserat Rods (Kugelstadt and Schömer, 2016) is the simplest to implement.

For fluid simulations, we would desire similar properties. As mentioned, methods involving PBD can be coupled easily as outlined in Macklin et al. (2014). Position-Based Fluids (Macklin and Müller, 2013) becomes a clear pick, as it uses SPH, which is a fast and stable simulation method that doesn't need to be constrained to a grid and can thus

be used in a variety of situations.

Finally, for hair-fluid coupling, our choice of PBD and SPH leads us to Lin (2014), whose unified methods allow for the simple coupling of hairs and fluids.

Chapter 3

Theory

This chapter will delve further into the theory behind the technical aspects of this thesis. This paper will feature complicated topics that may be difficult to understand, even for more tenured computer scientists, so this chapter will break down these concepts to make them easier to understand. This will also make Chapter 5 more straightforward to read, as we will not need to stop to explain how new concepts work while implementing them.

Section 3.1 will explore the OpenGL rendering pipeline and how we can use the GPU to perform parallel computations. Section 3.2 will details how physics solvers work, how they can be discretised, and how they can be optimised to run in real time. This will aid in understanding Section 3.3, which explores the field of Position-Based Dynamics and how to model and solve PBD constraints. Section 3.4 will touch on the specifics behind Smoothed Particle Hydrodynamics. Quaternions will be touched on in 3.5 and hair simulations will be explained in more detail in Section 3.6. Finally, in Section 3.7, we will look into how to use spatial hashing apply a fixed-radius nearest neighbour search to reduce the the runtime complexity of the fluid and hair simulations.

We will denote matrices using the uppercase bold typeface (e.g., $\mathbf{A}$), vectors using the lowercase bold typeface (e.g., $\mathbf{x}$) and scalars and quaternions using lowercase italics (e.g., ω).

3.1 OpenGL

In order to render custom liquids and hair strands, we will need to create a custom program to allow for close debugging. Although superior and pre-made software for rendering already exists, including game engines such as Unity or Unreal Engine, these packages include several features that would abstract away part of the simulation process. For example, Unreal Engine's Groom package includes a hair simulator (Epic Games,

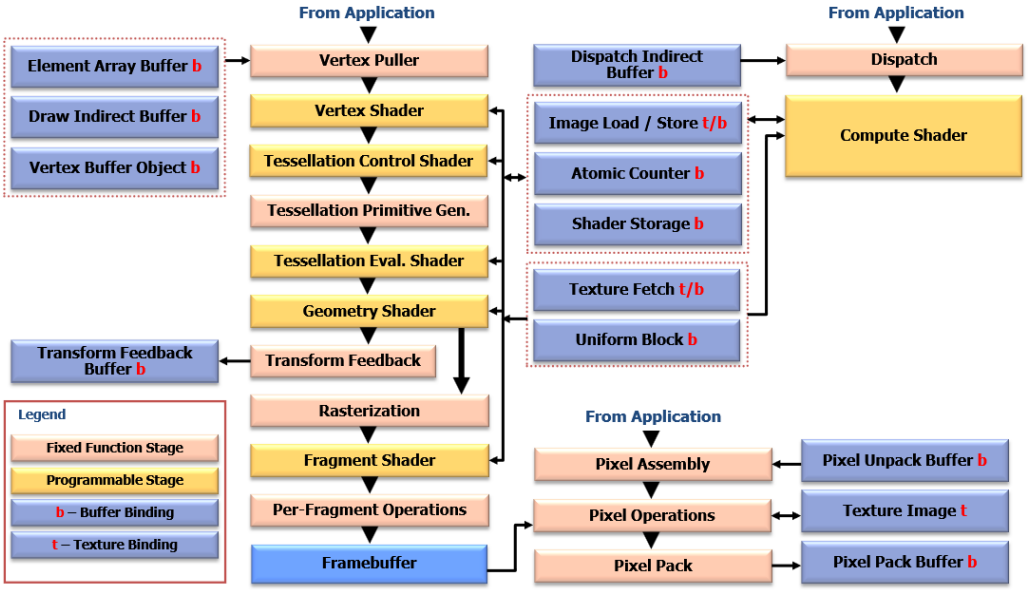


Figure 3.1: The OpenGL 4.6 rendering pipeline (Segal and Akeley, 2022)

2025b), the use of which would defeat the purpose of this thesis. OpenGL is an older but still widely accepted platform, and the concepts introduced in this section will be applicable to any platform that supports rendering and general-purpose GPU (GPGPU) programming.

3.1.1 The OpenGL Framework

OpenGL is a multi-platform graphics language used for creating graphical applications. It is available on Windows and Linux, and is available on MacOS devices up to version OpenGL 4.1. OpenGL provides several functions that allow for the rendering of various "primitives", such as points, lines, or triangles.

Points or vertices are positions in 3D space where, if following the right-hand rule, the negative z-axis is the "forward" direction. Vertices are passed through the rendering pipeline where are projected onto the screen as coloured pixels or "fragments". A shader is a program that the GPU runs; the program is usually either operating on a vertex or a fragment.

Broadly, the rendering pipeline first takes vertices from the CPU and projects them into screen-space using a vertex shader. These positions go through the rasterisation step to output them as pixels on the screen. Finally, the fragment shader assigns colours to these pixels via lights, shading, or some other technique. When rendering triangles, it may also be useful to provide indices for each point on the triangle to tell the rasteriser

which points can be reused for which triangles. Depending on the platform, different steps can be added to the rendering pipeline; the vertex, rasterisation, and fragment processing steps are permanent and cannot be removed.

3.1.2 Framebuffers

As we can see in Figure 3.1, the pipeline for modern OpenGL has several additions to give more functionality to developers. One such addition is the framebuffer (FBO), which takes place after the fragment shader and allows for the output that would ordinarily be rendered straight to the screen to be stored in a 2D texture. Textures may be passed into fragment shaders and sampled, which allows for post-processing effects. For example, a cube rendered to the screen can be stored in a framebuffer's texture. The fragment shader can then take that texture and apply a blur to only the cube, leaving all other geometry unaffected.

The screen is considered the default framebuffer. To use a framebuffer, it must be bound before any rendering happens to capture all OpenGL calls. Unbinding a framebuffer will re-bind the default framebuffer. Framebuffers may also be "offline", where they are drawn to and immediately used in another shader without being drawn to the screen first.

Framebuffers can contain renderbuffers, which contain information about the scene such as depth or the results of stencil test. Renderbuffers cannot be written to textures and sampled like framebuffers, so they are only used when the depth or stencil is needed for rendering, but not for sampling. If one wanted to use the depth or stencil for rendering, they would need to create additional colour attachments.

Fragment shaders normally only output the colour of the pixel at some location. However, they have the ability to output any kind of data by using *binding points* to output them to specific locations. By default, only the first binding point is enabled, but others can be enabled using `glNamedFramebufferDrawBuffers` and a list of the desired draw buffers. This is often used to output data like depth maps to extra textures. When a framebuffer with additional draw buffers is bound, any fragment shader that outputs to that binding point will be sent to the texture at that binding point, giving framebuffers the ability to store multiple types of data in a single draw call.

3.1.3 Tessellation Shaders

Another rendering stage, this time after the vertex shader, is the tessellation shader. (OpenGL Wiki, 2020) Tessellation shaders (TS) generate additional vertices to the program depending on the vertex shader output. They are distinct to the Geometry Shader

(GS) stage in that TS are made for *subdivision*. Given a triangle, a quad, or an isoline[1], the TS can choose how to subdivide it, how many copies to make, and where each subdivision goes.

CPU

Firstly, on the CPU, the Tessellation stage can be enabled by providing a Tessellation Control Shader (TCS) and a Tessellation Evaluation Shader (TES). The TCS is optional; as long as the TES is compiled along with the vertex shader and fragment shader, there will be no issues. [2] Instead of the primitives we are used to, we must specify a "patch". Patches are treated as groups of unspecified vertices. We must also provide the size of each patch. To draw patches, we can configure OpenGL like so:

Listing 3.1: Tessellation Shader CPU input example

```
glPatchParameteri(GL_PATCH_VERTICES, 4);
glDrawArrays(GL_PATCHES, vertexCount);
```

This will use 4 vertices per patch *input*.

Tessellation Control Shader

The TCS is the first interface in the TS pipeline. It takes place immediately after the vertex shader stage and determines how much tessellation is performed for each patch. The TCS is run for each patch and controls how much each patch is subdivided. Any outputs from the vertex shader that previously went to the fragment shader now go through here as an array to be accessed per patch. It contains a layout qualifier that specifies how many vertices to *output* and provides per-patch outputs. The `gl_InvocationID` variable holds the index of the vertex within the current patch.

The TCS controls how much to tessellate each patch primitive by using two arrays: `gl_TessLevelOuter` of length 4 and
`gl_TessLevelInner` of length 2. The definitions of these arrays depends on the type of patch primitive being used. If it is a quad, then `gl_TessLevelOuter` refers to the subdivision level of each of the outer sides and `gl_TessLevelInner` refers to the subdivision of the inner square. Triangles only use the first three outer tessellation values and the the first inner tessellation value. Isolines use only the first two outer tessellation values and no inner values.

[1] A quad is simply a quadrilateral. An isoline is a line. These terms are how they are referred to in GLSL.

[2] The TCS will use its default settings in this case. The fragment shader is also optional, though including a TES without it would be pointless.

Tessellation Primitive Generator

The Tessellation Primitive Generator (TPG) is a non-programmable fixed function stage that tessellates the vertices according to the TCS. It strives to ensure continuity between patches to prevent breaks in geometry. They work with "abstract patches", meaning that it does not actually look at the patches returned from the TCS, only how to tessellate them according to the rules specified.

Tessellation Evaluation Shader

As the final tessellation stage, the TES is responsible for determining the final positions, colours, and other values of the vertices to be given to the fragment shader, similar to the vertex shader. This stage specifies what the patch primitive will be considered as, the spacing between tessellated vertices, and the winding order in which to generate them. For example, the code in Listing 3.2 shows the values the TES takes in. `isolines` means the patch was tessellated as one or more isolines. `equal_spacing` means the tessellation used equal spacing between generated vertices. `cw` means the vertices were generated using a clockwise winding order. The winding order can be used to specify the order in which vertices are processed, which, for example, is used for face culling with `GL_CULL_FACE`. Specifying the order happens earlier than usual when using Tessellation Shaders. (OpenGL Wiki, 2022)

Listing 3.2: Tessellation Evaluation Shader input example

```
layout (isolines, equal_spacing, cw) in;
```

The TES can also interpolate values across its inputs. Values passed from the TCS are once again specified per-patch, and the tessellation coordinate is stored in the `gl_TessCoord` variable. A simple method to interpolate between values along an isoline is given in Listing 3.3:

Listing 3.3: Interpolating normals in the TES

```
layout (isolines, equal_spacing, cw) in;
in vec3 inNormal[]; // per-patch list of length 2
out vec3 outNormal; // output for *this* vertex only
uniform mat4 projView; // apply uniforms here instead
void main() {
        float u = gl_TessCoord.x;
        outNormal = mix(inNormal[0], inNormal[1], u);
        vec4 p = calculatePosition();
        gl_Position = projView * p;
```

}

When it comes to rendering hair strands, applying the appropriate curl can be done easily with Tessellation Shaders. Using a spline, such as a Catmull-Rom spline or B-Spline means that you only need to input the vertices that make up the curl, rather than a detailed model. This allows for hair strands to be user-specified, allowing for a more dynamic and personalised application.

3.1.4 Compute Shaders

While most stages of the rendering pipeline are directly tied to creating, storing, and rendering geometry, there exists a process that falls outside that avenue. Vertex shaders explicitly take in vertices and output their positions, and fragment shaders explicitly take in positions and output colours, or other values as specified by the currently-bound framebuffer. However, compute shaders do not take in or output any particular data. This means that they can be used for arbitrary computation.

General-purpose GPU (GPGPU) programming is the field of using GPUs for arbitrary computation. Compute shaders fall under this category. With the ability to manipulate vectors and matrices, compute shaders become a powerful tool for mathematical simulations in fields like graphics, physics, and mathematics. GPUs are usually only meant for simple calculations, such as basic multiplication, and have much lower clock speeds — "thinking rates". Despite this, due to the thousands of cores a GPU may have, they can perform these tasks in parallel, achieving far more work than a CPU can if utilised correctly.

An alternative is CPU multiprocessing using libraries such as OpenMP (The OpenMP ARB, 2007); however, for the purposes of this dissertation, the overhead cost of moving data to the GPU for rendering and back to the CPU for calculations is expensive. A better alternative in this case would be tools such as OpenCL (Khronos Group, 2009) or CUDA (NVIDIA, 2007), both of which provide additional tools for GPGPU.

Work Groups and Local Groups

Compute shaders operate within the abstract concept of work groups and local groups, or subgroups. A work group defines the 3D set of subgroups we dispatch in the shader. For example, if our work group is of size $(16, 16, 4)$, our shader would dispatch its local subgroup 16 times on the "x-axis" and "y-axis" and 4 times on the "z-axis", or $16 \times 16 \times 4 = 1024$ times. This could be used for image processing or matrix manipulation.

Work groups are dispatched on the CPU as shown in Listing 3.4. Since the GPU invocations may not finish at the same time, we must wait for them using `glMemoryBarrier`.

Groups don't run in any specified order, so the work group $(6, 15, 2)$ could run before $(2, 1, 3)$.

Listing 3.4: Dispatching a compute shader in OpenGL

```
int dispatchSize = ceil(particles.size() / 16) + 1;
glDispatchCompute(dispatchSize, 1, 1);
glMemoryBarrier(GL_SHADER_STORAGE_BARRIER_BIT);
```

Subgroups are specified in the shader itself. They dictate how many times the shader runs. Each invocation can be accessed with built-in variables. We can also find the index of a particular invocation across the entire dispatch with `gl_GlobalInvocationID`.

The total number of shader invocations across a dispatch can be found as a product of the work group size and the subgroup or local size. For example, consider a set of 1,024 particles and a compute shader with a local size of $(16, 1, 1)$. Here, we only want to run a 1D compute shader. To find the optimal dispatch size, we can divide our particle count by the local size on each axis and round up. This gives us $(1023/16) + 1 = (64, 1, 1)$ dispatches. We add 1 for the case that the number of particles we have is indivisible by our local size. If we overshoot, we can ignore invocations whose global ID is greater than the size of the list.

Shader Storage Buffer Objects

As mentioned, compute shaders don't have inputs or outputs. To access data from compute shaders, we can use methods such as Image Load Store or Shader Storage Buffer Objects (SSBO). SSBOs are easier to use and more general, so we will explain them briefly.

An SSBO is simply a buffer of data. The data within the buffer must be of one type, such as an integer or C-style struct, and can hold as least 128 megabytes of data. Compute shaders, like any shader, must have a bound vertex array object (VAO) to run without errors. [3] Once created, SSBOs can be bound as any type of data, but they must be bound before the dispatch or drawing call is issued. They are bound to a binding point that must be specified in the shader. Within the shader, which is not necessarily a compute shader, the data can be accessed written to or read from freely. An example of how to create and use an SSBO is shown in Listing 3.5.

Listing 3.5: Creating an SSBO in OpenGL 4.6 using C++

```
unsigned int VAO, SSBO;
```

[3] This is platform dependent. It is common to create empty "dummy VAOs" to load and dispatch compute shaders under.

```
// create VAO and SSBO
glCreateVertexArrays(1, &VAO);
glCreateBuffers(1, &SSBO);

// store Particle struct in SSBO
glNamedBufferStorage(SSBO,
        sizeof(Particle) * particles.size(),
        particles.data(),
        GL_MAP_READ_BIT
);

// bind VAO
glBindVertexArray(VAO);
// bind SSBO to binding point 0
glBindBufferBase(GL_SHADER_STORAGE_BUFFER, 0, SSBO);

glUseProgram(myComputeShaderID);
int dispatchSize = ceil(particles.size() / 32) + 1;
glDispatchCompute(dispatchSize, 1, 1);
glMemoryBarrier(GL_SHADER_STORAGE_BARRIER_BIT);

// unbind when finished
glBindVertexArray(0);
```

An example of accessing this SSBO in a shader is shown in Listing 3.6.

Listing 3.6: Accessing an SSBO in a compute shader

```
#version 460 core

layout (local_size_x = 16) in;

struct Particle {
        vec4 position;
        vec4 velocity;
};

layout(std430, binding=0) buffer Particles {
```

```
    Particle particles[];
};

layout(location = 0) uniform int speed;
layout(location = 1) uniform float deltaTime;

void main() {
        uint gid = gl_GlobalInvocationID.x;
        if (gid >= particles.length()) return;
        int idx = int(gid);

        particles[gid].velocity += speed * deltaTime;
        particles[gid].position += velocity * deltaTime;
}
```

3.2 Physics Solvers

While the exact mathematics involved behind physics solvers is outside the scope of this paper, we still need to explain what exactly these solvers are doing in order to write code for them. Position-Based Dynamics is based on this, so we will very briefly explain what it means to solve these mathematical systems.

We should note that the title of this section is technically misleading. While the concepts explained in this section will be used to solve physics equations, they are in fact *mathematical* solvers used to solve systems of linear and non-linear equations. However they will still lay the foundation behind how physics equations and constraints are solved.

3.2.1 Linear vs. Non-linear

A linear system of equations is any system of linear equations that share the same variables. By linear, we mean that the equations themselves are linear. For example, the following equations are linear:

$$3x + 4y = 10$$
$$-2(x + 16) = 3y$$

We can also define this by saying that the change input is *proportional* to the change in output.

A non-linear system is therefore any system of non-linear equations that share variables. This would include equations such as:

$$x^2 + 4x + 4 = 2y$$
$$y = ln(x)$$

Solving systems of equations like these analytically can be done easily by hand, but computers solving them numerically must approximate the correct result. This is done through *iterative methods* that repeatedly approximate the solution by making an educated guess and updating the current solution to that guess. The *time-step* Δt is the time between simulation frames. This value is unrelated to the computation itself; smaller time-step sizes generally mean a more accurate and/or stable simulation, but the machine will compute the same steps at the same speed. Only the simulation will run slower. This is repeated until the system *converges*, or comes close enough to with an acceptable error of a desired answer. Three examples of this are the Newton-Ralphson method for non-linear equations and the Gauss-Seidel and Jacobi methods for linear equations. We will discuss linear methods below.

3.2.2 Jacobi Solver

The Jacobi iteration algorithm is a method for solving a system of linear equation. Typically, this method is used for solving a diagonally-dominant system, where the diagonal is larger than the sum of all other values on that row. For example,

$$\begin{bmatrix} 1 & 0.2 & 0.02 \\ 0.1 & 1 & 0.3 \\ 0.01 & 0.01 & 1 \end{bmatrix}$$

is a diagonally-dominant system of equations. The above matrix, which must be square, contains the coefficients of a traditional system of linear equations. The matrix can be solved by decomposing it and solving it iteratively. The mathematics behind this is outside the scope of this paper, but the algorithm remains similar for our algorithms. To solve a system of linear equations in a Jacobi fashion, we can do the following as shown in Algorithm 1:

where t is the current simulation frame and $f(x_i^t, x_j^t)$ is some function that operates on each pair of values. This method does not modify the value x_i during the iterations, instead compiling them separately and applying the changes after. This method tends to converge slowly, as the entire solver loop needs to occur for every value before any get

Algorithm 1 Iterative Jacobi solver

$t \leftarrow 0$
for i: $1 \rightarrow n$ **do**
  $\Delta x_i = 0$
  for j: $1 \rightarrow n$ **do**
    if $i \neq j$ **then**
      $\Delta x_i \leftarrow \Delta x_i + f(x_i^t, x_j^t)$
    end if
  end for
end for
for i: $1 \rightarrow n$ **do**
  $x_i^{t+1} \leftarrow x_i^t + \Delta x_i$
end for
$t \leftarrow t + 1$

updated.

3.2.3 Gauss-Seidel Solver

The Gauss-Seidel method is an alternative. Instead of compiling changes separately, they are applied immediately. An example is shown in Algorithm 2:

Algorithm 2 Iterative Gauss-Seidel solver

$t \leftarrow 0$
for i: $1 \rightarrow n$ **do**
  $\Delta x_i = 0$
  for j: $1 \rightarrow n$ **do**
    if $i \neq j$ **then**
      $\Delta x_i \leftarrow x_i^t + f(x_i^t, x_j^t)$
    end if
  end for
  $x_i^t \leftarrow \Delta x_t$
end for
for i: $1 \rightarrow n$ **do**
  $x_i^{t+1} \leftarrow x_t^t$
end for
$t \leftarrow t + 1$

The Gauss-Seidel updates each value immediately, which increases the rate of convergence. However, this system is inherently serial, in that solving one value during the inner loop depends on the previous value. This makes Gauss-Seidel approaches unusable for parallel computation, which may operate on values in any order. Jacobi iterations avoid this issue by not updating values while solving, but this is not an acceptable trade-off for

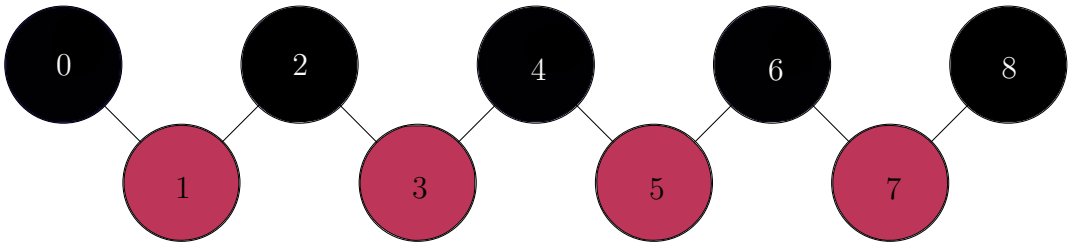


Figure 3.2: Red-black ordering. All pairs of even (black) particles will be solved before odd (red) ones in the order ($[0, 1], [2, 3], [4, 5], [6, 7], [1, 2], [3, 4], [5, 6], [7, 8]$)

our purposes, especially in real-time. We must look to methods for allowing Gauss-Seidel iterations to be performed in parallel.

3.2.4 Optimisation

While Gauss-Seidel solvers are typically faster than Jacobi solvers, they may not be fast enough for our purposes. For hair simulations, there are special considerations that must be taken, particularly if they are solved in parallel.

Red-Black Ordering

A hair strand is discretised as a series of vertices connected by rods. Each of these vertices undergoes certain forces to constrain its position and orientation, retaining its curl. Solving forces on a hair strand means solving each vertex in order. As mentioned, if we were to use the Gauss-Seidel method, each vertex would be updated immediately and used in the next rod. This method works fine for a single thread, such as on the CPU.

However, on the GPU, threads do not have a guaranteed order. Some hair vertices may be updated before prior ones, creating instabilities in the simulation that would cause it to fail. To get around this, we can use red-black ordering. (Bindel, 2010) Simply put, we can solve all vertices with an even index first, then all odd vertices. Since each vertex depends on previous one and moves the next one, this ensures that vertices will not be updated with incorrect values. This method is a specialisation of graph colouring, limited to constraints between two nodes.

Successive Over-Relaxation

Successive over-relaxation (SOR) is a simple method for quickly speeding up the convergence of a Gauss-Seidel solver. By choosing some non-zero value ω, we can linearly interpolate between the current solution and previous solutions. (Black and Moore, 2025)

This would take the form of

$$\boldsymbol{x}_i^t = \frac{\omega}{n_i}\boldsymbol{x}_i^t + (1 - \frac{\omega}{n_i})\boldsymbol{x}_i^{t-1} \tag{3.1}$$

for a position $\boldsymbol{x_i}$. n_i denotes the number of constraints the particle is involved in.

This simple addition increases the convergence of the simulation, lowering the number of iterations required and thereby boosting performance.

3.3 Position-Based Dynamics

PBD has been referenced several times throughout this paper, but is yet to be explained. PBD is a position-based physics solver. (Müller et al., 2007) Traditionally, force-based solvers have been more popular as they can create more accurate behaviour; however, for real-time applications, plausibly correct behaviour suffices. PBD is also usually more stable than force-based solvers as it operates directly on the positions of particles.

Following the convention of Macklin et al. (2014), we can define a vertex or particle i as a grouping of a position $\boldsymbol{x}_i$, a velocity $\boldsymbol{v}_i$, an inverse mass w_i, and a type t_i. The type determines the constraint and forces the particle will be subject to. PBD defines a set of *constraints* of the form $C_j : \mathbb{R}^{3n_j} \to \mathbb{R}$, where n_j represents the cardinality of constraint j. Each constraint operates on a particular type of particle and can be an *equality* or *inequality* constraint. To determine whether the constraint is solved, it returns an error value. An equality constraint is solved if $C_j(\boldsymbol{x}_i, \ldots, \boldsymbol{x}_n) = 0$ for the n points it is applied to. An inequality constraint is solved if $C_j(\boldsymbol{x}_i, \ldots, \boldsymbol{x}_n) \geq 0$. Each constraint is also associated with a stiffness parameter $k_j \in [0, 1]$ which defines how strongly the constraint is applied.

The *gradient* of a constraint $\nabla_p C(\boldsymbol{p}_1, \ldots, \boldsymbol{p}_n)$ dictates the direction in which the value of the constraint function is increasing the fastest. This value is used to determine the direction to move the particle. The correction displacement $\Delta\boldsymbol{p}_i$ is chosen to move $\boldsymbol{p}_i$ along this gradient. In this situation, we want to solve an equality constraint such that

$$C(\boldsymbol{p} + \Delta\boldsymbol{p}) = 0 \tag{3.2}$$

which we can approximate with

$$C(\boldsymbol{p} + \Delta\boldsymbol{p}) \approx C(\boldsymbol{p}) + \nabla_p C(\boldsymbol{p}) \cdot \Delta\boldsymbol{p} = 0 \tag{3.3}$$

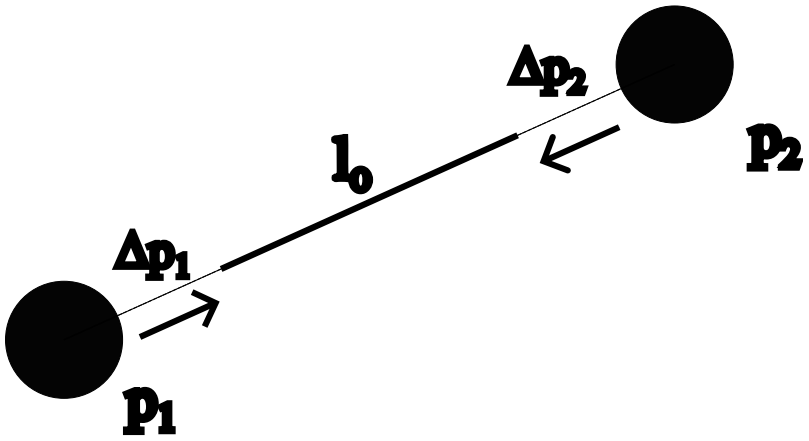


Figure 3.3: An example of a PBD distance distance constraint. p_1 and p_2 will be offset by Δp_1 and Δp_2 respectively to return the distance between them to the rest length l_0.

This eventually leads us to the full definition for the correction displacement:

$$\Delta \boldsymbol{p}_i = -s w_i \nabla_{p_i} C(\boldsymbol{p}_1, \ldots, \boldsymbol{p}_n) \tag{3.4}$$

where w is the inverse mass of particle i and s is a scaling term of the form:

$$s = \frac{C(\boldsymbol{p}_1, \ldots, \boldsymbol{p}_n)}{\sum_j w_j \|\nabla_{p_j} C(\boldsymbol{p}_1, \ldots, \boldsymbol{p}_n)\|^2} \tag{3.5}$$

As an example, we can consider a distance constraint. Let l_0 represent the target (rest, default) length between two particles $\boldsymbol{p_i}$ and $\boldsymbol{p_j}$, as shown in Figure 3.3. Solving the constraint requires finding a solution for $C(\boldsymbol{p}_i, \boldsymbol{p}_j) = 0$. We can define the distance constraint from the perspective of i to be

$$C_i(\boldsymbol{p}_i, \boldsymbol{p}_j) = \|\boldsymbol{p}_i - \boldsymbol{p}_j\| - d \tag{3.6}$$

where d is the current distance between the particles. The gradient of this constraint increases the farther the particles become from each other. Thus, we can define the gradients from the perspective of both particles as

$$\nabla_{p_i} C(\boldsymbol{p}_i, \boldsymbol{p}_j) = \boldsymbol{n} \tag{3.7}$$

$$\nabla_{p_j} C(\boldsymbol{p}_i, \boldsymbol{p}_j) = -\boldsymbol{n} \tag{3.8}$$

where $\boldsymbol{n}$ is the normalised distance. Thus, we can write out the displacements like so:

$$\Delta p_i = -\frac{k w_i(\|(\boldsymbol{p}_i - \boldsymbol{p}_j)\| - d)}{w_i + w_j} \frac{\boldsymbol{p}_i - \boldsymbol{p}_j}{\|\boldsymbol{p}_i - \boldsymbol{p}_j\|} \tag{3.9}$$

$$\Delta p_j = +\frac{k w_j(\|(\boldsymbol{p}_i - \boldsymbol{p}_j)\| - d)}{w_i + w_j} \frac{\boldsymbol{p}_i - \boldsymbol{p}_j}{\|\boldsymbol{p}_i - \boldsymbol{p}_j\|} \tag{3.10}$$

Creating constraints can be generalised to higher dimensions, such constraining the volume of a tetrahedron for use in soft-body physics.

In the original paper, the constraints alone were applied and solved with Projected Gauss-Seidel iterations (PGS), where positions were projected along the constraint gradient in each iteration loop. The original overall simulation loop would then consist of the following steps:

1. Apply external forces such as gravity to the velocities and dampen them
2. Predict positions using updated velocities
3. Generate collision constraints
4. Solve all constraints a number of times and apply them to predicted positions
5. Update velocities using Verlet integration
6. Update positions to the predicted positions

However, Macklin et al. (2019) discovered the revelation that simply stepping through the entire simulation loop repeatedly provided much faster convergence. Using *substepping*, and by dividing the time-step by the number of substeps, a more stable version of the simulation can emerge. The new general simulation loop for PBD is as follows:

Algorithm 3 A substepped PBD simulation loop

$\Delta t_s \leftarrow \dfrac{\Delta t}{n_{substeps}}$
while simulation is active **do**
  apply external forces $\boldsymbol{v_i} \leftarrow \boldsymbol{f}^{ext}$
  predict positions $\boldsymbol{p}_i \leftarrow \boldsymbol{x}_i + \boldsymbol{v}_i \Delta t_s$
  handle collisions $\boldsymbol{x}_i \rightarrow \boldsymbol{p}_i$
  solve constraints $C(\boldsymbol{p}_i, \ldots, \boldsymbol{p}_n) \Rightarrow \boldsymbol{p}_i \leftarrow \boldsymbol{p}_i + \Delta \boldsymbol{p}_i$
  $\boldsymbol{v}_i \leftarrow \dfrac{\boldsymbol{p}_i - \boldsymbol{x}_i}{\Delta t_s}$
  $\boldsymbol{x}_i \leftarrow \boldsymbol{p}_i$
end while

The expression for handling collision is chosen as if a particle's predicted position results in it becoming out of bounds, it can be set to its previous position.

3.4 Smoothed Particle Hydrodynamics

SPH discretises a continuous media into a set of discrete particles through the use of a *kernel* or *smoothing* function. (Koschier et al., 2022) We want it so that the neighbours of a particle receive forces proportional to their distance. This leads to the use of a kernel that smoothens the effects of the function depending on the distance of the neighbours. This smoothing kernel, denoted as $W(\boldsymbol{r}, h)$, determines the output for a distance $\boldsymbol{r}$ between two particles and a *smoothing radius* h that informs the function how close a particle needs to be to have an effect. The chosen kernel must be symmetrical: any force applied to its neighbours could plausibly be applied back to it from its neighbours. In other words, $W(\boldsymbol{r}, h) = W(-\boldsymbol{r}, h)$. Some other conditions, such as the kernel returning a positive value, are also desired.

For some quantity $A(\boldsymbol{x})$ for some particle at position $\boldsymbol{x}$, SPH will return an approximation of that quantity according to the smoothing kernel. A continuous approximation is given by

$$A(\boldsymbol{x}) \approx (A * W)(\boldsymbol{x}) = \int A(\boldsymbol{x}')W(\boldsymbol{x} - \boldsymbol{x}', h)dv' \tag{3.11}$$

where dv' is the volume integration corresponding to the nearby sampled position $\boldsymbol{x}'$. We can rewrite this as a discrete sum, leading us to the primary formation:

$$\sum_{j \in \mathcal{N}} A_j \frac{m_j}{\rho_j} W(|\boldsymbol{x}_i - \boldsymbol{x}_j|, h) \tag{3.12}$$

where ρ_j is the density of particle j, m_j is its mass, and A_j is the quantity we want to measure. For example, to estimate the density surrounding the particle at position $\boldsymbol{x}_i$, we can use the following approximation:

$$\rho_i = \sum_j m_j W_{ij} \tag{3.13}$$

where W_{ij} is the shorthand for the smoothing kernel; this remains the same value as in Equation 3.12.

We may also want to calculate the gradient of a quantity. To do this, we use a similar formula:

$$\nabla A_i \approx \sum_j \frac{m_j}{\rho_j}(A_j - A_i)\nabla W_{ij} \tag{3.14}$$

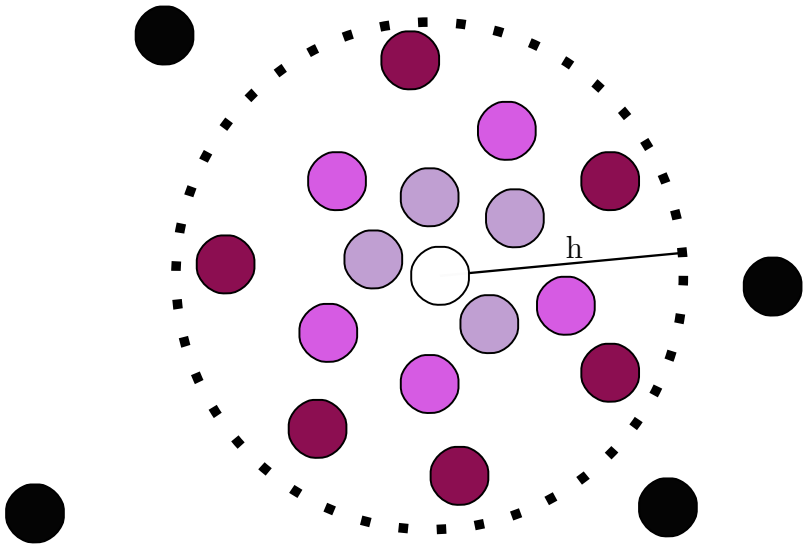


Figure 3.4: The smoothing radius of a particle. The main particle is highlighted in coloured in white; the further away from the centre of the radius a particle becomes, the less of an effect is exerted on it, which is shown here by brightness. Black particles outside the radius are entirely unaffected.

Our choice of smoothing kernel depends on the situation. A simple kernel to use is the Gaussian distribution; however, Koschier et al. (2022) reject this as the Gaussian function does not have a compact support domain. That is, the function does not return 0 for values outside the smoothing radius. For our purposes, we will use a select list of kernels to satisfy different conditions. This list is given below. We will denote the italic math symbol $r = \|\boldsymbol{r}\|$ as the length of the difference $\boldsymbol{r}$.

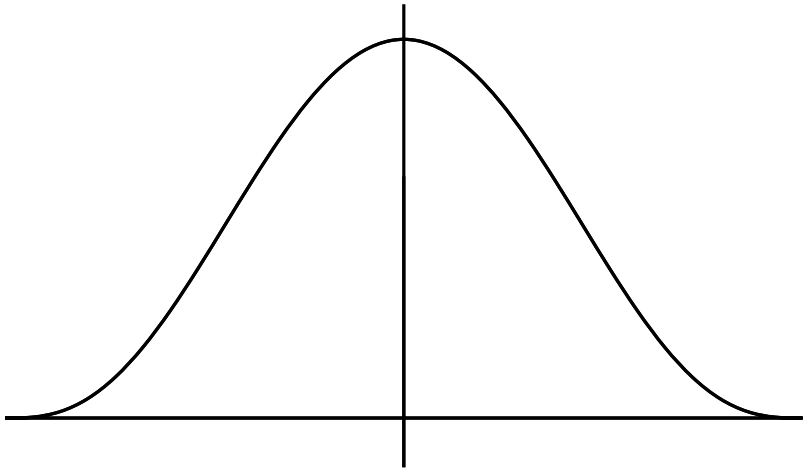


Figure 3.5: The poly6 kernel distribution.

First, the poly6 kernel and its gradient (Tychonievich, 2024):

$$W_{\text{poly6}}(\boldsymbol{r}, h) = \frac{315}{64\pi h^9} \begin{cases} (h^2 - r^2)^3 & 0 \leq r \leq h \\ 0 & \text{otherwise} \end{cases} \tag{3.15}$$

$$\nabla W_{\text{poly6}}(\boldsymbol{r}, h) = \frac{315}{64\pi h^9} \begin{cases} -6(h^2 - r^2)^2 \boldsymbol{r} & 0 \leq r \leq h \\ 0 & \text{otherwise} \end{cases} \tag{3.16}$$

This kernel is flat near $r = 0$ as noted by Müller et al. (2003), so while it may be used for density estimation, we want a more defined gradient that pushes particles out of the centre. This leads us to the spiky kernel:

$$W_{\text{spiky}}(\boldsymbol{r}, h) = \frac{15}{\pi h^6} \begin{cases} (h - r)^3 & 0 \leq r \leq h \\ 0 & \text{otherwise} \end{cases} \tag{3.17}$$

$$\nabla W_{\text{spiky}}(\boldsymbol{r}, h) = \frac{-45}{\pi h^6} \begin{cases} (h - r)^2 |\boldsymbol{r}| & 0 \leq r \leq h \\ 0 & \text{otherwise} \end{cases} \tag{3.18}$$

We also want to model fluid viscosity, so we include the viscosity kernel given by Müller et al. (2003):

$$W_{\text{visc}}(\boldsymbol{r}, h) = \frac{15}{2\pi h^3} \begin{cases} \frac{-r^3}{2h^3} + \frac{-r^2}{2h^2} + \frac{h}{2r} - 1 & 0 \leq r \leq h \\ 0 & \text{otherwise} \end{cases} \tag{3.19}$$

3.5 Quaternions

A quaternion is a number with one real part and three imaginary parts of the form:

$$q = a + b\boldsymbol{i} + c\boldsymbol{j} + d\boldsymbol{k} \tag{3.20}$$

Quaternions represent 3D rotations in space. To rotate a vector $\boldsymbol{v}$ by a quaternion q, we can multiply it like so:

$$\boldsymbol{v}' = q\boldsymbol{v}\bar{q} \tag{3.21}$$

where $\bar{q}$ is the conjugate of the quaternion:

$$\bar{q} = a - b\boldsymbol{i} - c\boldsymbol{j} - d\boldsymbol{k} \tag{3.22}$$

The imaginary parts of q form the direction of rotation, and the real part the angle.

Given a vector $\boldsymbol{v}$ and an angle θ in radians about that vector, we can construct the quaternion:

$$q = \cos\frac{\theta}{2} + \boldsymbol{v}\sin\frac{\theta}{2} \tag{3.23}$$

Multiplying two quaternions by each other returns a quaternion representing the rotation of the second, *then* the rotation of the first. They may also be normalised by dividing by their norm:

$$\|q\| = \frac{q}{\sqrt{(a^2 + b^2 + c^2 + d^2)}} \tag{3.24}$$

3.6 Cosserat Theory

As mentioned in Section 2.1.5, Cosserat rods are slender rods that pass through a centre line $\boldsymbol{r}(s)$ and allow for the modelling of stretching, shearing, bending, and twisting. s refers to where along the rod we are looking. This section will briefly touch on some of the terminology used to make their usage clearer.

A material, local reference $\boldsymbol{Q}(s) = \{\boldsymbol{d}_1, \boldsymbol{d}_2, \boldsymbol{d}_3\}$ frame is present at throughout the rod itself. (Gazzola et al., 2018) These frames are defined relative to the global laboratory frame, which is given by the basis vectors $\{\boldsymbol{e}_1, \boldsymbol{e}_2, \boldsymbol{e}_3\}$. A vector $\boldsymbol{x}$ can be described in terms of both frames as

$$\bar{\boldsymbol{x}} = x_i\boldsymbol{e}_1 + x_2\boldsymbol{e}_2 + x_3\boldsymbol{e}_3 \tag{3.25}$$

$$\boldsymbol{x} = x_i\boldsymbol{d}_1 + x_2\boldsymbol{d}_2 + x_3\boldsymbol{d}_3 \tag{3.26}$$

The values $\{\boldsymbol{d}_1, \boldsymbol{d}_2, \boldsymbol{d}_3\}$ are called *directors*. $\boldsymbol{d}_3$ normally points along the tangent of the rod $\partial_s\boldsymbol{r}(s)$, which is orthonormal to the cross-section at any given frame. We can use $\boldsymbol{Q}$ to define a rotation matrix, allowing us to freely convert between these two frames. We may also use quaternions to rotate basis vectors to directors with $\boldsymbol{d}_k = q\boldsymbol{e}_k\bar{q}$.

If $\boldsymbol{d}_3$ does not point in the tangent direction, the rod must be undergoing shearing. If the length of the tangent is not unit length, the rod must be stretched. We can define a stretch and shear strain vector (Kugelstadt and Schömer, 2016) to measure the deformation:

$$\boldsymbol{\Gamma}(s) = \partial_s\boldsymbol{r}(s) - \boldsymbol{d}_3(s) \tag{3.27}$$

To express this vector in the material frame, we can rotate it with the same quaternion along the tangent:

$$\bar{\boldsymbol{\Gamma}} = \boldsymbol{R}(q)^T\partial_s\boldsymbol{r}(s) - \boldsymbol{d}_3(s) \tag{3.28}$$

where $\boldsymbol{R}(q)^T$ is the transposed Rodrigues (Kugelstadt and Schömer, 2016) matrix formed

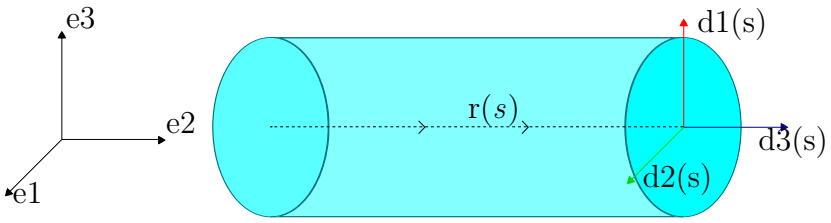


Figure 3.6: An representation of a Cosserat rod.

from the quaternion.

To model bending and twisting deformations, we can make use of Darboux vectors. Named after Gaston Darboux, a French mathematician, Darboux vectors describe how the material frame changes as it moves along the rod. (Umetani et al., 2015) It can therefore be used to determine the rotation of the rod at any frame. It is given as

$$\boldsymbol{\Omega} = \frac{1}{2}\sum_{k=1}^{3} \boldsymbol{d}_k \times \boldsymbol{d}'_k \tag{3.29}$$

where $\boldsymbol{d}'_{\boldsymbol{k}}$ represents the derivative of the director with respect to s. This representation is similar to angular velocity and can be expressed in terms of quaternion rotation:

$$\boldsymbol{\Omega} = 2\bar{q}q' \tag{3.30}$$

where $\bar{q}$ represents quaternion conjugation. Using this, we can check for bending and twisting deformations by determining how much the Darboux vector differs from its rest configuration. By measuring these *rest Darboux vectors* as Ω^0, we can define a measure for bending and twisting deformation:

$$\Delta\Omega = \Omega - \Omega^0 = 2(\bar{q}'q - \bar{q}^0 q'^0) \tag{3.31}$$

3.7 Grid Partitioning

PBD models all objects as series and groups of particles that emit forces against one another, and SPH applies its smoothing kernel on all nearby particles. Naively searching through every particle to find the closest ones would have a $O(N^2)$ time complexity and would be unsuitable for real-time applications. Instead, we can break up the simulation space into a discrete, regular grid. Each particle would be able to define its place in the grid and only need to check the surrounding grid cells in a *fixed radius* to find its nearest neighbours: the surrounding 9 cells in 2D and the surrounding 27 cells in 3D.

This method is practically necessary for Lagrangian fluid simulations such as SPH, which needs a fixed, well-defined grid cell size that works in tandem with the smoothing radius. The time complexity now only depends on the maximum amount of objects that could fit into the search radius. Eulerian-Lagrangian methods already come with this grid, so they enjoy faster searches by default.

However, one big issue with storing the grid is memory requirements. Since hybrid methods need to store information at each grid cell, they also must store the cells themselves. Lagrangian methods have no such need for this, and so can use smarter methods for saving memory. This section will detail how these methods work and how they can be used in parallel physics solvers.

3.7.1 Spatial Hashing

Spatial hashing is one such technique. By eschewing a predefined region of space, we can pick a cell size and round space towards it. A particle can simply divide its position by the cell size to find the cell it is contained in. Then, by accessing each surrounding region of space, it can locate nearby particles and decide what to do next.

This technique has two problems. Firstly, hashing is not an infallible system. Two or more regions can hash to the same region, meaning that the currently searching particle must consider particles entirely too far away to reach. Adding unnecessary forces can be avoided by ensuring that the searched particles are within the search radius first and using squared length to determine their distance to avoid expensive squaring.

The second problem is actually determining these nearby particles. We have already decided not to store information about particles at each grid cell, so any information stored will need to be large enough to hold our particles, but small enough such that we are not spending too much memory.

3.7.2 Counting Sort

A popular method for achieving this is counting sort. (Hoetzlein, 2014) Counting sort is an integer sorting algorithm that can be used to sort values into bins, such as our grid cells. Values will be sorted in an array in a stable fashion, so the indices of particles will not change while being sorted.

We can adapt it to our needs by first defining a ***startIndices*** table at least as large as our particle list, plus one slot as a guard. We also define a second ***cellEntries*** list to store particle indices. Following this, the algorithm is as follows:

1. Each particle position is discretised to a grid cell, which is then hashed and wrapped

around the length of the ***startIndices*** table using the modulo function. The index in the table is then incremented for that particular grid cell.

2. The prefix sum or scan is computed for ***startIndices***. This may be either an inclusive scan, where the first value in the array is added to the second, or an exclusive scan, where the first value in the array is set to 0.

3. Each particle position is hashed again and the value at the ***startIndices*** index is decremented. The value in the table is used to index the ***cellEntries*** array and place that particle's index in the array.

The result of this is that particles in the same cell now have their indices ordered within the ***cellEntries*** array. To access them, we can access the grid cell using ***startIndices*** and using the index there as the start of the list of nearby particles. That index plus one will store the the end of the list.

This method is very simple to implement and use on the CPU, as prefix sum is an inherently serial algorithm. However, for parallel applications, prefix sums need to be implemented in a very difficult and non-obvious manner, such as a Blelloch scan. NVIDIA provides an implementation of this (Harris et al., 2007), but as we are designing an application that can be used on a wide range of platforms, not just those with NVIDIA drivers. Therefore, for simplicity, we look to another method.

3.7.3 Atomic Bump Allocation

Atomic bump allocation (ABA) is a sparsely referenced alternative for sorting grids. (Levien, 2020) (Lewin and Barré-Brisebois, 2024) The main downside of ABA over counting sort is that while counting sort is a stable sorting algorithm, ABA is not. This means that even with the utmost care taken to ensure a deterministic GPU ordering simulation, ABA will render it nondeterministic. However, for our purposes, the nondeterminism of this method is not a deal-breaker.

ABA is overall a very similar method to counting sort. It uses atomic instructions and shared memory to organise particle indices within a list. We define a *bucket* as a tuple containing the a grid cell's start index, the number of particles in the cell, and the next available index in the bucket. The algorithm is as follows. Note that we use the terms "grid cell", "bin", and "bucket" interchangeably throughout the rest of this document.

1. Similar to counting sort, we define a ***startIndices*** table to hold our hashed grid cells. On this pass, every particle simply adds 1 to its bucket's particle count.

2. During the allocation step, we use a shared integer ***particleCountIndex*** and add to it. Atomic instructions modify some buffer in a user-defined way and then *return the previous value at that index in the buffer*. By adding the current bucket count to the shared integer, we obtain the previous size of the total number of integers, allowing us to obtain a valid location for the start of this bucket and set up the start location of the next bucket.

3. Finally, in the insertion step, we atomically add 1 to the next available bucket slot for each particle in the bucket. We again use a ***cellEntries*** list and store the bucket's start index plus the offset from the atomic addition to obtain the particle's correct global location. Figuring out the location of each particle in practice is much easier, as the bucket already holds the information about the beginning and start of the range of particles in the bucket.

Of these two methods, ABA is far easier to implement. The difference in determinism is not damaging enough to the simulation to warrant using counting sort, and, as ascertained, any such implementation would be difficult, especially in GLSL.

3.8 Summary

This section provided an overview of the topics that will be discussed in Chapters 4 and 5. Additional detail can be found in the bibliography. We also provided some more insight as to why we will make certain decisions later in this dissertation.

Chapter 4

Design

In this chapter, we will discuss the strategy we will take in order to create our wet hair simulation. We will explain our methodology, why we have taken it, and how we will implement it.

4.1 Problem Formulation

As mentioned in Chapter 1, we seek to ascertain the efficacy of creating a real-time wet hair simulation using afro-textured hair. This type of hair cannot be simulated accurately due to the large number of collisions required to model, so we instead look to discrete methods to define our hair strands by modelling them as a discrete list of points. To model wet hair, we also seek a discrete fluid simulation method. Hair simulations are typically either constrained to a single location for a demonstration or single application, or are applied in dynamic environments such as video games. We seek a model that can be used in an open environment, so we require a Lagrangian fluid simulation. As Lagrangian simulations use free particles, our hairs should be simulated with a similar method for easier coupling. To retain coil with this technique, we can use a discretised Cosserat rod model for our hair constraints.

4.1.1 Identified Challenges

There exists certain gaps in knowledge left by the lack of research in the field. More sophisticated solutions may be possible when this topic is explored further.

1. The challenges of hair-fluid coupling lie mostly in the aesthetics and the performance. Again, the density of afro-textured hair means that most hair strands are always in proximity of one another. Hair strands clump together when wet, so for

a single wet strands, all strands in its vicinity will clump with it. However, as these other strands are also within a dense field of hair, they will attempt to clump other their other neighbours. This results in an oscillating behaviour where hair strands constantly attempt to clump with all other hair strands they find. This also means that for each hair particle, their field of nearest neighbours is always dense, increasing computation costs.

2. Another challenge is the hair and fluid simulations themselves. One simulation may run given a certain time-step size and substep count, but a different simulation may require longer substeps or a smaller time-step. Combining simulations like these requires careful consideration, as increasing the time-step for one may cause it to fail and "explode", whereas decreasing it may cause the other to run too slowly.

3. Another, more technical problem is fluid mass diffusion. Lin (2014) uses this to slowly remove fluid mass from hair strands and deleting particles that shrink to some mass limit. However, since we are performing our simulation on the GPU using compute shaders and SSBOs to store particle data, we cannot easily delete particles. Attempting to remove all references to a particular particle would require writing all that information to the CPU, casting it to a common list data structure, removing all references, and then putting all data back on the GPU. This would incur a massive overhead cost, particularly if we have a lot of hair or fluid.

4.2 Overview of the Design

Our approach will use Position and Orientation-Based Cosserat Rods (POCR) to model the hair strands using Cosserat rods and PBD. For our fluid simulation, we will use Position-Based Fluids, as it is also a PBD method. Finally, for our hair-fluid coupling, we will use the clumping and adhesion forces as specified by Lin. In this section, we will denote hair particles with h_i, fluid particles with f_i, porous particles with p_i.

4.2.1 Hair Model

Since PBD works on particles, we can generate hair strands by simply generating a series of points on a helix. The radius and length of the helix can be changed before the simulation, which can be used to generate different types of hair. Each particle will contain a position, velocity, inverse mass, and type based on the model given by Macklin et al. (2014). They will be added to a global particle list. We will also generate a series of

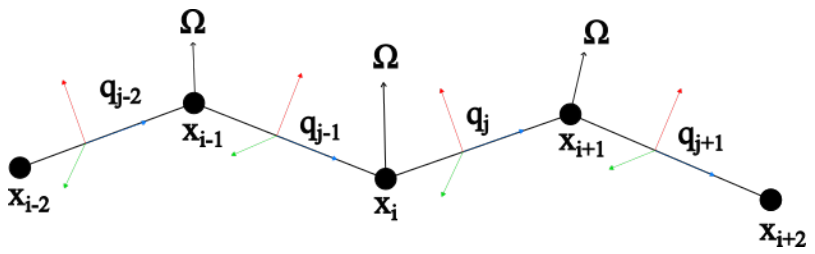


Figure 4.1: An representation of a discretised Cosserat rod. $\boldsymbol{q}_j$ represents the quaternion beginning from particle i.

quaternions representing the rods between our hair particles. Each rod will generate the rest Darboux vector between itself and the next rod for the bend and twist constraint.

These particles and rods will make up one hair strand object. Strands will be stored as a tuple containing information about the number of strands and rods, where in the particle and rod buffers they begin, the root position of the strand, and the rest length between particles for use in the stretch and shear constraint.

In addition to hair particles, we will store porous particles to be sampled between hair particles as suggested by Lin. These porous particles will be created the same way as hair particles, but will carry additional data about the hair vertices they sit between, the volume of porous particles around them, and the local fluid density.

Due to the density of the hair, we do not compute any hair-hair collisions. We model the head as a simple sphere and perform collisions with it by projecting any particles within its radius outside it.

Stretch and Shear Constraint

The stretch and shear constraint is an extension of the basic distance constraint given in Equation 3.6. Hair particles cannot just stay a certain distance from each other; they need to orient themselves in the same way as their initial configuration. We can discretise Equation 3.27 to form the stretch and shear constraint (Kugelstadt and Schömer, 2016):

$$C_{ss}(\boldsymbol{p}_1, \boldsymbol{p}_2, q) = \frac{1}{l_0}(\boldsymbol{p}_2 - \boldsymbol{p}_1) - \boldsymbol{R}(q)\boldsymbol{e}_3 = 0 \tag{4.1}$$

where l_0 is the rest length between the particles. As shown in Figure 3.6, $\boldsymbol{e}_3$ represents the global up direction, so $\boldsymbol{R}(q)\boldsymbol{e}_3$ is equivalent to the director $\boldsymbol{d}_3$. We use this constraint

to calculate the final displacement corrections:

$$\Delta p_1 = +\frac{w_1 l_0}{w_1 + w_2 + 4w_q l_0}\left(\frac{1}{l_0}(\boldsymbol{p}_2 - \boldsymbol{p}_1) - \boldsymbol{d}_3\right) \tag{4.2}$$

$$\Delta p_2 = -\frac{w_2 l_0}{w_1 + w_2 + 4w_q l_0}\left(\frac{1}{l_0}(\boldsymbol{p}_2 - \boldsymbol{p}_1) - \boldsymbol{d}_3\right) \tag{4.3}$$

$$\Delta q = +\frac{w_q l_0^2}{w_1 + w_2 + 4w_q l_0}\left(\frac{1}{l_0}(\boldsymbol{p}_2 - \boldsymbol{p}_1) - \boldsymbol{d}_3\right) q\bar{e}_3 \tag{4.4}$$

where $\bar{e}_3$ is a quaternion with a real part of 0.

Bend and Twist Constraint

Kugelstadt and Schömer define rods as quaternions between ordered pairs of particles. Like particles, each rod q_j has its own velocity and (inverse) mass. The velocity of a rod $\boldsymbol{\omega}_j$ can be determined through the formula for angular velocity:

$$\boldsymbol{\omega}_j = \boldsymbol{\omega}_j + \Delta t \boldsymbol{I}^{-1}(\boldsymbol{\tau} - \boldsymbol{\omega}_j \times (\boldsymbol{I}\boldsymbol{\omega}_j)) \tag{4.5}$$

where $\boldsymbol{I}$ is the inertia matrix for a thin rod and τ is torque. We can use this to predict the next orientation of the rods:

$$u_j = q_j + \frac{q_j \boldsymbol{\omega}_j \Delta t}{2} \tag{4.6}$$

Like the stretching constraint, we can use Equation 3.31 to form the bend and twist constraint for two rods q and u:

$$C_{bt}(q, u) = \mathbf{Im}(\bar{q}u - \bar{q}^0 u^0) = \boldsymbol{\Omega} - s\boldsymbol{\Omega}_0 = 0 \tag{4.7}$$

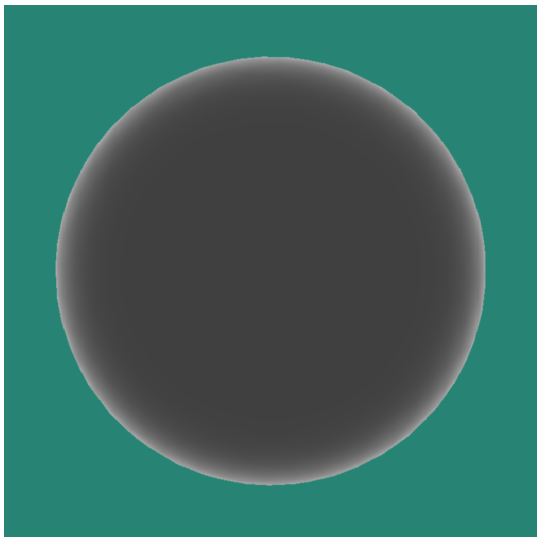
where $\mathbf{Im}(q)$ is the imaginary component of the quaternion q and s is the function

$$s = \begin{cases} +1 & \text{if } \|\boldsymbol{\Omega} - \boldsymbol{\Omega}^0\|^2 < \|\boldsymbol{\Omega} + \boldsymbol{\Omega}^0\|^2 \\ -1 & \text{otherwise} \end{cases} \tag{4.8}$$

Our bending and twisting constraints are:

$$\Delta q = +\frac{w_q}{w_q + w_u} u(\boldsymbol{\Omega} - s\boldsymbol{\Omega}_0) \tag{4.9}$$

$$\Delta u = -\frac{w_q}{w_q + w_u} q(\boldsymbol{\Omega} - s\boldsymbol{\Omega}_0) \tag{4.10}$$



(a) An example of the Fresnel effect on a diffuse ball.

(b) An example of rim lighting on an afro. Photo of Abbey Lincoln, taken by Jack de Nijs (1996). (Wikimedia Commons, 2025)

Figure 4.2: Fresnel lighting versus rim lighting. Rim lighting an afro can be approximated using the Fresnel effect.

4.2.2 Hair Rendering

We will use splines to render our hair strands. The point of generating particles in a helix was to use an interpolating spline that passes through each particle. For our use case, we have chosen Catmull-Rom splines as they provide C^1 continuity [1] and are simple to implement.

As mentioned, afro-textured hair contains many bends that all reflect light in many different directions. If considered not as individual hairs but a large, rough mass with its volume concentrated towards the centre, the intense scattering of light makes it suitable for shading techniques like subsurface scattering. However, this technique or its approximations may be too slow to compute, so we instead use a different approximation.

We use the Cook-Torrance BRDF for lighting our hair strands. This model, similar to Marschner's shading model, bounces light of off *microfacets* in diffuse objects. This makes it a physically-based rendering (PBR) method, meaning we can define the roughness of our hair to get a more diffuse surface. The edges of the hair are also a good use case for the Fresnel effect, an example of which is in Figure 4.2. The rim lighting will make the hair's edges more obvious.

Hair gets darker when wet, as seen in Figure 2.4. Afro-textured hair is typically very dark, so the darkening effects associated with wet hair will not be as visible. Regardless,

[1] C^1 continuity means that a spline has matching tangents at the end of interpolated line segments.

since afro-textured hair can come in any colour or be dyed, we will use the density of the fluid surrounding porous particles to attenuate the albedo colour and roughness value of the hair strands.

4.2.3 Fluid Model

Fluid particles will be created in the same manner as hair particles but will store additional auxiliary information required for computing the density constraint. We will follow the methodology of Macklin and Müller by using the poly6 kernel for density calculations, the viscosity kernel for viscosity calculations, and the spiky kernel for constraint gradients.

PBF uses a density constraint to enforce incompressibility. Macklin et al. (2014) modify this constraint to ensure that densities only move particles apart from each other by clamping them to be non-positive. We can define this constraint as

$$C_i(\boldsymbol{p}_1, \ldots, \boldsymbol{p}_n) = \frac{\rho_i}{\rho_0} - 1 \leq 0 \tag{4.11}$$

where ρ_0 is the target density. The gradient of the constraint for particle k is therefore

$$\nabla_{p_k} C_i = \frac{1}{\rho_0} \begin{cases} \sum_j \nabla_{p_k} W(\boldsymbol{p}_i - \boldsymbol{p}_j, h) & \text{if } k = i \\ -\nabla_{p_k} W(\boldsymbol{p}_i - \boldsymbol{p}_j, h) & \text{if } k = j \end{cases} \tag{4.12}$$

where the first case calculates the density gradient around the particle itself, and the second case represents the force accumulated from all neighbouring particles. Recall from Section 3.4 that smoothing kernels must be symmetrical.

We modify Equation 3.5 to relax the force to avoid a singularity, now writing it as

$$s_i = \frac{-C(\boldsymbol{p}_1, \ldots, \boldsymbol{p}_n)}{\sum_j w_j \|\nabla_{p_j} C(\boldsymbol{p}_1, \ldots, \boldsymbol{p}_n)\|^2 + \varepsilon} \tag{4.13}$$

where ε is a small user-specified relaxation parameter. The final correction displacement is updated to

$$\Delta \boldsymbol{p}_i = \frac{1}{\rho_0} \sum_j (s_i + s_j) \nabla_{p_i} W_{ij} \tag{4.14}$$

using the neighbour's density constraint values.

We also model viscosity and surface tension forces, which are needed for more viscous fluids such as creams. We use the XSPH viscosity term as used in Monaghan (1992):

$$\boldsymbol{f}_{i \leftarrow j}^{\text{viscosity}} = \kappa_{\text{visc}} \sum_j \frac{m_j}{\rho_j} (v_j - v_i) W_{\text{visc}}(\boldsymbol{p}_i - \boldsymbol{p}_j, h) \tag{4.15}$$

where κ_{visc} is the viscosity coefficient and v_i is the velocity of particle i. The kernel W_{visc} is provided in Equation 3.19.

For surface tension, we follow the implementation of Akinci et al. (2013) to define a cohesion force to pull particles together and a curvature force to group then in a spherical shape:

$$\boldsymbol{f}_{i\leftarrow j}^{\text{curvature}} = -\kappa_{\text{curve}} m_i(\boldsymbol{n}_i - \boldsymbol{n}_j) \tag{4.16}$$

$$\boldsymbol{f}_{i\leftarrow j}^{\text{cohesion}} = -\kappa_{\text{cohesion}} m_i m_j W_C(\boldsymbol{p}_i - \boldsymbol{p}_j, h)\frac{\boldsymbol{p}_i - \boldsymbol{p}_j}{\|\boldsymbol{p}_i - \boldsymbol{p}_j\|} \tag{4.17}$$

The kernel W_C is given as

$$W_C(\boldsymbol{r}, h) = \frac{32}{\pi h^9}\begin{cases}(h-r)^3 r^3 & 2r > h \wedge r \leq h \\ 2(h-r)^3 r^3 - \frac{h^6}{64} & r > 0 \wedge 2r \leq h \\ 0 & \text{otherwise}\end{cases} \tag{4.18}$$

For curvature, we want to reduce the surface area of the conjoined body of fluid. Akinci *et. al.* handle this by defining the curvature normal $\boldsymbol{n}_i$ to be

$$\boldsymbol{n}_i = \sum_j \frac{m_j}{\rho_j}\nabla_{\text{spiky}} W_{ij} \tag{4.19}$$

4.2.4 Fluid Rendering

Fluid rendering is not a significant portion of this paper, as we focus more on the physical interactions of hair and fluids. However, providing a clear definition of exactly which algorithms were followed would be useful for future readers, so we detail the method here.

For fluid rendering, we will use screen-space rendering by Green (2010), which generates an excellent fluid surface without needing to generate meshes like with the Marching Cubes algorithm. We will first render fluid particles to the screen as cropped point sprites. We can use this output to calculate a depth and thickness map which we will then blur. Green uses a bilateral filter in his paper to only blur surfaces within a similar depth of each other; we replace their bilateral filter with a narrow-range filter (Truong and Yuksel, 2018) that provides a more accurate depth-aware blur. Using the blurred depth map, we can generate a normal map, which we can then use for diffuse lighting. The thickness map will be used to provide the appearance of volume to the fluid by rendering it as mostly transparent and using Beer's Law to model volumetric absorption. We also use reflection and refraction to provide more accurate views of the environment of the simulation.

4.2.5 Coupling

We will use a simplified model of Lin's paper, focusing purely on adhesion and clumping effects. For transferring forces from porous particles to fluid particles, Lin uses the same adhesion forces as Akinci *et. al.*, adding a force of the type

$$\boldsymbol{f}^{\text{adhesion}}_{f_i \leftarrow p_k} = -\kappa_{\text{adh}} m_f \Psi_{p_k} \rho_0 W^A_{ik} \frac{\boldsymbol{p}_i - \boldsymbol{p}_k}{\|\boldsymbol{p}_i - \boldsymbol{p}_k\|} \tag{4.20}$$

where κ_{adh} is the adhesion coefficient, $\boldsymbol{p}_i$ is the position of the fluid particle, $\boldsymbol{p}_k$ is the position of the porous particle, Ψ_{p_k} is the representative volume of porous particle, ρ_0 is the fluid rest density, and W^A_{ik} is a kernel function specified in Akinci et al. (2013):

$$W_{\text{adhesion}}(\boldsymbol{r}) = \frac{0.007}{h^{3.25}} \begin{cases} \sqrt[4]{-\frac{4r^2}{h} + 6r - 2h} & 2r > h \wedge r \leq h \\ 0 & \text{otherwise} \end{cases} \tag{4.21}$$

The representative volume is given by

$$\Psi_{p_i} = \frac{1}{\sum_k W_{ik}} \tag{4.22}$$

For the clumping coefficient, Lin uses an equation similar to the fluid-pore adhesion force in Equation 4.20:

$$\boldsymbol{f}^{\text{clumping}}_{p_i \leftarrow p_j} = -\kappa_{\text{clump}} \bar{S} U^{\text{atten}} \Psi_{p_i} \Psi_{p_j} W^{Cl}_{ij} \frac{\boldsymbol{p}_i - \boldsymbol{p}_j}{\|\boldsymbol{p}_i - \boldsymbol{p}_j\|} \tag{4.23}$$

where κ_{clump} is the clumping force, $\bar{S}$ is the average saturation of the porous particles, and U^{atten} is an attenuation factor for submerged strands. Lin specifies W^{Cl}_{ij} as the kernel

$$W_{\text{clumping}}(\boldsymbol{r}, h) = \frac{32}{h^9} \begin{cases} (h-q)^3 q^3 & r > R \wedge r \leq h \\ 0 & \text{otherwise} \end{cases}$$

However, we found that this kernel made submerged strands too restless. We instead reuse the surface tension kernel from Equation 4.18. The attenuation factor is calculated via

$$U^{\text{atten}} = (1 - \mathbf{min}(\frac{\rho_u}{\rho_0}, 1)) \tag{4.24}$$

where ρ_u is the density of fluid around the porous particle. The saturation S_i of a porous particle is found with

$$S_i = \frac{m_i}{\rho_0 \phi \Psi_i} \tag{4.25}$$

where ϕ is the porosity of the hair strands, which defines how porous hair strands are. The higher the porosity, the lower the effective adhesion.

As mentioned, we are focused primarily on the adhesion and clumping effects of wet hair. However, fluids naturally fall out of porous materials over time as the density increases. After a time, when the density mass of fluid around a hair strand builds up to a certain point, its collective mass will cause it to fall out of the hair strand as a water droplet. Lin includes this in his paper by using Darcy's Law and the diffuse volume of a porous particle to determine how much to modify the mass of a fluid particle by. When the mass of the particle shrinks below 10% of its original volume, it is deleted.

Since we do not want to delete particles, we use a different process to imitate dripping. Instead of changing fluid mass, we increase the force of gravity fluids are affected by through the use of a global fluid mass diffusion factor γ. Each fluid particle now holds an additional diffusion factor δ_i that slowly increases the longer a fluid particle remains near a porous particle:

$$\delta_i = \begin{cases} \delta_i + \gamma\rho_i & \text{if } \exists\ \boldsymbol{p}_k \text{ s.t. } \|\boldsymbol{p}_{f_i} - \boldsymbol{p}_{p_k}\| < h \\ 1 & \text{otherwise} \end{cases} \tag{4.26}$$

We now model the gravitational force of a fluid particle with

$$f_{f_i}^{\text{gravity}} = \delta_i g \tag{4.27}$$

where g is the acceleration due to gravity. This method skips the additional iteration required calculate the pore velocity term used in the original mass diffusion calculation, saving on computational costs.

4.2.6 Algorithm

The overall algorithm for simulating the hair-fluid coupling is in Algorithm 4. Although it is not mentioned in the algorithm itself, hair rods are predicted and updated at the same time as hair particles.

Algorithm 4 Our hair-fluid coupling algorithm

$\Delta t_s \leftarrow \dfrac{\Delta t}{n_{substeps}}$
initialise neighbour grid
while simulation is active **do**
  for n_{substeps} substeps **do**
    apply external forces $\boldsymbol{v_i} \leftarrow \boldsymbol{f}^{gravity}\Delta t_s$
    calculate representative volumes $\boldsymbol{\Psi_{p_i}}$
    compute densities $\boldsymbol{\rho_{f_i}}$
    compute XSPH viscosity $\boldsymbol{v}_{f_i} \leftarrow \boldsymbol{f}^{\text{viscosity}}$
    compute fluid auxiliary quantities [2]
    predict hair/fluid positions $\boldsymbol{p}_{hf_i} \leftarrow \boldsymbol{x}_{hf_i} + \boldsymbol{v}_{hf_i}\Delta t_s$
    predict porous positions $\boldsymbol{p}_{h_i} \rightarrow \boldsymbol{p}_{p_k}$
    handle collisions $\boldsymbol{x}_i \rightarrow \boldsymbol{p}_i$
    solve constraints $C(\boldsymbol{p}_i, \ldots, \boldsymbol{p}_n) \Rightarrow \boldsymbol{p}_i \leftarrow \boldsymbol{p}_i + \Delta\boldsymbol{p}_i$
    compute clumping forces $\Delta\boldsymbol{p}_{p_i} \leftarrow \boldsymbol{f}^{\text{clumping}}$
    update hair/fluid velocities $\boldsymbol{v}_{hf_i} \leftarrow (\boldsymbol{p}_{hf_i} - \boldsymbol{x}_{hf_i})/\Delta t_s$
    update hair/fluid positions $\boldsymbol{x}_{hf_i} \leftarrow \boldsymbol{p}_{hf_i}$
    update porous positions $\boldsymbol{x}_{h_i} \rightarrow \boldsymbol{p}_{p_k}$
  end for
end while

4.2.7 Pipeline

The simulation will also take the rendering steps into account. Particle data will be kept in buffers on the GPU and sent to vertex and fragment shaders for rendering. We provide an overview of the entire simulation pipeline in Figure 3.1.

4.3 Summary

Our overall pipeline will serve at the vehicle for our simulation. Data will be kept in buffers in SSBOs to keep them on the GPU, which saves performance. This also allows them to be used in the rendering step as we can simply rebind the SSBOs whenever we need them.

[2] Curvature normals and PBF scaling factors

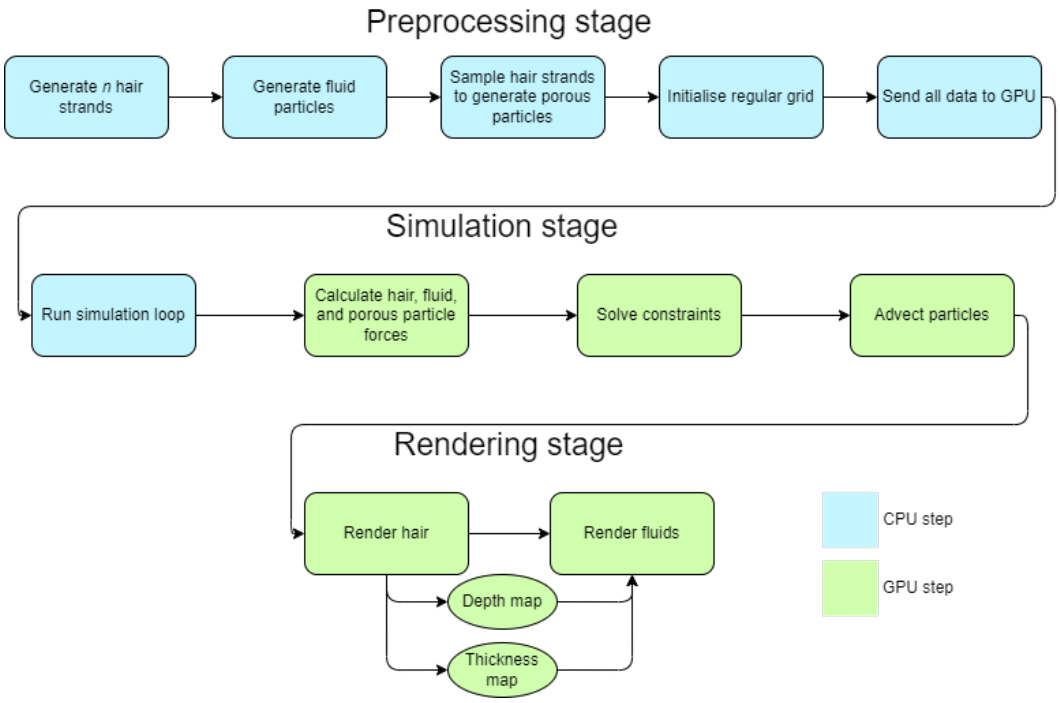


Figure 4.3: Our simulation and rendering pipeline.

Chapter 5

Implementation

Our implementation was created in OpenGL using C++ using the GLM maths library. The simulation was ran on the GPU using compute shaders. Hair strands were tessellated using tessellation shaders and our fluid was rendered using a series of framebuffers. We will detail our implementation of Chapter 4 using these tools below.

5.1 Preprocessing Stage

During this stage, we generate and process every particle that will be added to our simulation.

5.1.1 The `Simulation` class and the `Particle` struct

First, let us make clear how a particle is defined. We declare a particle to be a C-style struct

```
struct Particle {
    vec4 x;  // position
    vec4 v;  // velocity
    float w; // inverse mass
    int t;   // type
    float d; // type-specific quantity
    int s;   // type-specific quantity
}
```

Listing 5.1: Our `Particle` struct

The position, velocity, inverse mass, and type were all specified previously. The first type-specific quantity `d` is the sum of the porous particle fluid density around a hair vertex

for hair particles and the diffusion factor from Equation 4.26 for fluid particles. This value is unused for porous particles. The second quantity `s` contains the index of the hair strand a hair particle is in and is unused for other particles.

The type-specific quantities are placed here for two reasons.

1. Firstly, arrays of structs imported with the `std430` layout qualifier will be tightly packed; that is, the stride between array elements will be 0. The data within structs will also be aligned to the largest alignment in the struct, which is 16 bytes or the size of a `vec4` in our case. `vec3`s are 12 bytes long, so padding must be added to satisfy this rule. (Segal and Akeley, 2022) This would require adding 4 bytes of padding between each vector we have. Instead, we can simply import data as a `vec4`. We could alternatively use an array of 3 floats, but this would make code difficult to read and tiresome to write.

2. Secondly, there is a limit to the number of SSBOs that can be bound at one time. In our implementation, we want both good performance and readability of code, so data is split into multiple buffers where available. The reasoning behind this is that due to the rule above, adding all data to a single struct risks it becoming unaligned with a `vec4`, which would require additional padding, wasting memory. But if our data is already a single `vec4` or `float`, it will be tightly packed. This method strikes a balance between performance and readability.

The entire simulation runs as part of the `Simulation` class, which handles which code runs and when. It is broken down into a `Hair` class for hair particles, a `Fluid` class for fluid particles, and a `SpatialGrid` class for the uniform grid used for neighbour searching. Each class handles its respective settings and setup.

5.1.2 Hair Strand Generation

Hair particles were generated on the CPU. The `Hair` class takes in a list of tuples determining the properties of each hair strand, which then generates its particles and rods. We provide our hair generation method in Appendix A. Afterwards, we generate rods as quaternions pointing in the direction between each adjacent pair of vertices. To keep rotations consistent, each quaternion is rotated with the one before it, with the first quaternion rotating from the global up direction. We define our inertia matrix using the moment of inertia for a rod rotating about its centre as $\frac{1}{12}m_i l_0^2$ and applying this value down the diagonal of the 3×3 matrix. We define a `Rod` struct in Listing 5.2

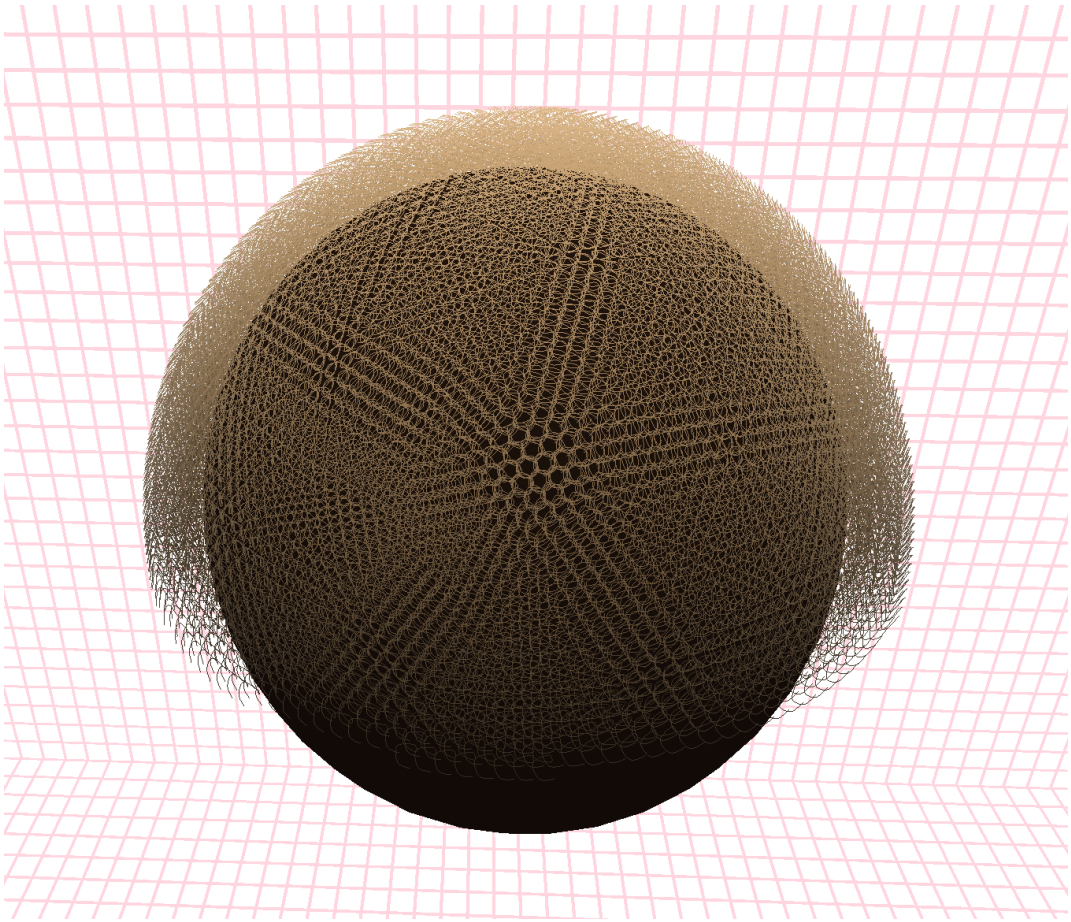


Figure 5.1: Our hair strands generated on a sphere.

```
struct Rod {
    vec4 q;          // quaternion
    vec4 v;          // angular velocity
    float w;         // inverse mass
    int s;           // rod strand index
    int pd1, pd2;    // padding
}
```

Listing 5.2: Our `Rod` struct

Again, while it is possible to reduce padding by embedding values inside others, this would make the code much harder to read and write.

As mentioned, the last type-specific quantity `s` contains the index directing each hair particle to the strand it belongs to. We can use the index of the strand to figure out the index of the quaternion whose start index is this particle. For a particle i, its quaternion is located at `i - particles[i].s`. Similarly, for a quaternion j, its Darboux vector is

located at `j - rods[j].s`. This is shown in Figure 5.2.

5.1.3 Fluid Particle Generation

Fluid particles are generated the same way as hair particles. Because they have no set location, we can arrange them in a grid to give them a nicer initial appearance:

```
void createFluidParticles(int n, PD pd, vec3 offset) {
    int cubeRoot = ceil(pow(n, 1.f / 3.f) - 1e-5);
    float spacing = smoothingRadius * .75f;
    for (int i = 0; i < n; ++i) {
        vec3 p = Util::to3D(i, vec3(cubeRoot));
        vec3 gridPos = (p - vec3(cubeRoot / 2.f + .5f)) *
            spacing;
        gridPos += centre + offset;
        Particle pf = Particle(gridPos, vec3(0), 1, FLUID);
        particles.push_back(pf);
        ps.push_back(vec4(gridPos, 0));
    }
}
```

Listing 5.3: Generating `n` fluid particles in a grid.

We also created framebuffer objects for screen-space fluid rendering. We created a `Framebuffer` class to handle the different framebuffers we need. A framebuffer can rendered to a full-screen texture for use in post-processing effects. To do this, we need to define a full-screen quad and render the framebuffer's textures onto it. We create a framebuffer like so:

```
void createFBO() {
    glCreateFramebuffers(1, &FBO);
    glCreateVertexArrays(1, &quadVAO);
    glBindVertexArray(quadVAO);

    glCreateBuffers(1, &quadVBO);
    glNamedBufferStorage(quadVBO,
        sizeof(vertexData[0]) * vertexData.size(),
        &vertexData[0], GL_MAP_READ_BIT);
    glVertexArrayVertexBuffer(quadVAO, 0, quadVBO, 0,
        sizeof(QVertex));
```

```
    glEnableVertexArrayAttrib(quadVAO, 0);
    glVertexArrayAttribFormat(quadVAO, 0, 2, GL_FLOAT,
        GL_FALSE, offsetof(QVertex, pos));
    glVertexArrayAttribBinding(quadVAO, 0, 0);

    glEnableVertexArrayAttrib(quadVAO, 1);
    glVertexArrayAttribFormat(quadVAO, 1, 2, GL_FLOAT,
        GL_FALSE, offsetof(QVertex, uv));
    glVertexArrayAttribBinding(quadVAO, 1, 0);

    glBindVertexArray(0);
}
```

Listing 5.4: Creating a framebuffer object. `QVertex` is a struct containing a 2D position and texture coordinate.

We can also add textures to the framebuffer. After creating the depth, thickness, blur, and normal map framebuffers, we can add their buffer objects to an environment framebuffer.

5.1.4 Porous Particle Generation

Porous particles are sampled between hair particles. We provided a method of determining where to place these particles; in practice, we found using any more than 1 porous particle for each adjacent pair too strong of an impact on performance. `PoreData` is a struct containing the indices of the hair particles it lies between and how much it influences either one.

```
void samplePorousParticles() {
    for (int s = 0; s < numStrands; ++s) {
        int sV = hairStrands[s].startVertexIdx;
        int sVLim = hairStrands[s].startVertexIdx +
            hairStrands[s].nVertices - 1;
        for (int i = sV; i < sVLim; ++i) {
            vec3 a = vec3(particles[i].x);
            vec3 b = vec3(particles[i + 1].x);
            vec3 d = b - a;
            for (int j = 0; j < poreSamples; ++j) {
```

```
                float strength = (j + 1) / (samples + 1.f);
                vec3 smpl = a + d * strength;
                Particle p = Particle(
                    smpl, vec3(0), 1.f, PORE);
                particles.push_back(p);
                ps.push_back(vec4(smpl, 0));
                PoreData pData = PoreData(
                    i, 1 - strength,
                    i + 1, strength);
                poreData.push_back(pData);
            }
        }
    }
}
```

Listing 5.5: Sampling porous particles. `samples` is the number of porous particles to generate between hair particles. `ps` is the predicted positions buffer.

5.1.5 Regular Grid Creation

As the regular grid is recreated every loop, we only have to specify the size of the `startIndices` buffer and the size of each grid cell. The `cellEntries` buffer will be the same size as the number of particles. We pick the table size to be 3 times the number of our particles. Our grid cells are 1 unit in size, though they can be changed during the simulation if so desired.

5.1.6 Sending data to the GPU

All of this information needs to be stored in buffers. Since all particles are of the same type, we store them all in the same buffer. The `glNamedBufferStorage` function takes in a buffer object and stores our data, after which we can do whatever we want with it. In practice, we use it in SSBOs. The data stored using this function is *immutable*; this is the primary reason we cannot delete fluid particles. We create a buffer the same way as in Listing 3.5.

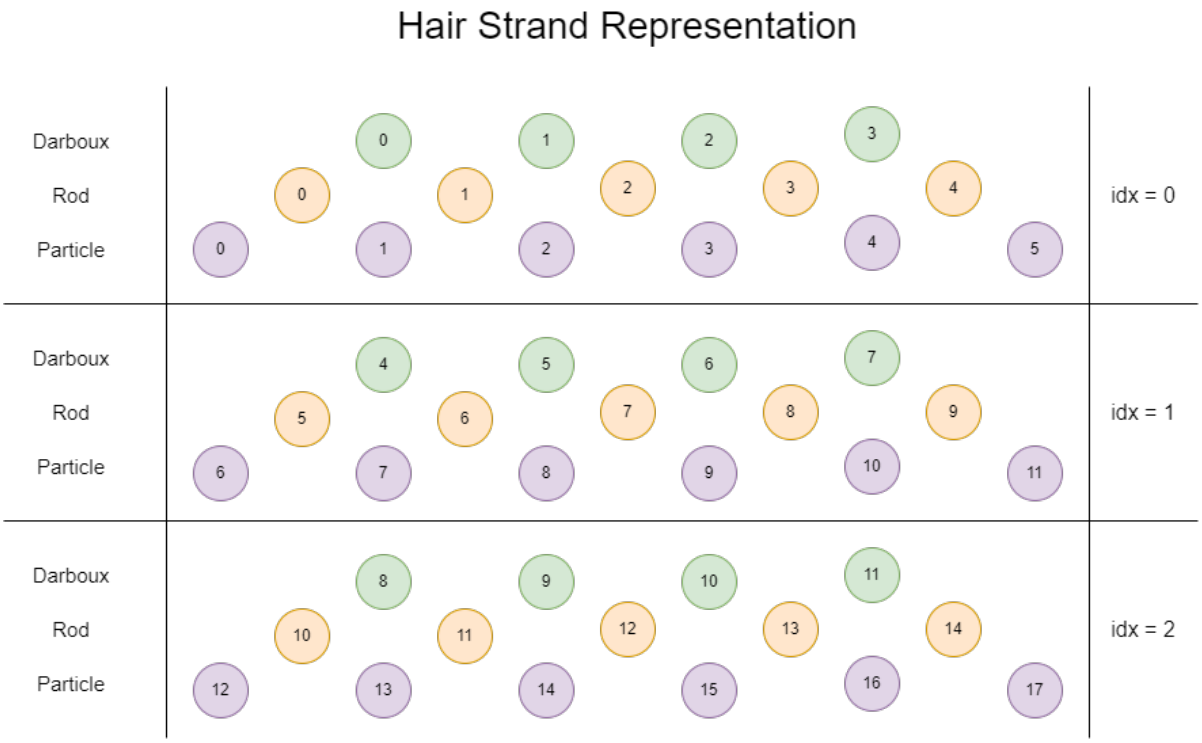


Figure 5.2: Particles can find the index of their rods using by subtracting the index of their strand from their own index. For example, the hair particle at hair index 8 (purple) has a rod at rod index 7 (yellow) that uses it as a starting index. $j = i - \mathbf{strandIdx}_x(i) \Rightarrow 7 = 8 - 1$

5.2 Simulation Stage

The simulation loop begins on the CPU as we need to tell the GPU we want to dispatch shaders. Most of the work happens on the GPU in this stage. As we have already explained the steps we will take in modelling our hair-fluid forces, we will provide more of an overview on how the code we wrote works.

5.2.1 Running Simulation Loop

At the beginning of the simulation stage, just before calculating the hair-fluid coupling, we dispatch the GPU to create our neighbour grid using atomic bump allocation. We start by resetting the grid and returning the values in all buckets to 0. Then we perform our counting, allocation, and insertion steps. The compute shaders for this step can be found in Appendix B.

After updating the grid, we bind all our buffers using `glBindBufferBase` at the binding index we want. Next, we bind our uniforms. We created a `Shader` class that handles the creation and binding of shader programs, and the passing of uniform data through them. Uniform data is information that can be passed to a shader from the CPU. The data in our uniforms consists of the time-step size scaled by the substep count, the number of each type of particle, the bounds of the playable area, and every physics constant we defined in Chapter 4.

The simulation is split up into the different sections shown in Algorithm 4. Our entire simulation is stored in a single compute shader for ease of reference, so we simply iterate over the number of simulation stages and perform a certain calculation of computation based on that stage. Each step is dispatched with the particles involved in that step. For example, computing densities requires both the fluid and porous particles, as both use their local fluid density in later equations. We then dispatch the compute shader like so:

```
switch (stage) {
    case ...:
        // ...
        break;
    case COMPUTE_DENSITIES:
        glDispatchCompute(ceil(
            ((fluidParticleCount + porousParticleCount)) /
            DISPATCH_SIZE) + 1, 1, 1);
        glMemoryBarrier(GL_SHADER_STORAGE_BARRIER_BIT);
        break;
```

```
    case ...:
        // ...
        break;
}
```

Listing 5.6: Dispatching the density computation

As mentioned, one of our main issues is with constraints that can be satisfied with different conditions. Indeed, this is the case here; the fluid simulation performs reasonably well with just one substep, but the hair simulation requires at least 10 substeps to hold up. To rectify this conflict, we make a compromise: the simulation is ran at 5 *substeps* and the hair constraints are ran with 5 *iterations*. What this means is that the fluid density constraint is ran 5 times per simulation iteration, and the hair constraints are run 25 times. Iterations are not as "valuable" as substeps are, but we found this to make a comfortable median that still prevents the fluid simulation from failing. It does require significant damping, however, which only serves to make the simulation even more unphysical. However, we feel as though this is an appropriate compromise.

Hair constraints depend on prior and later particles, so we need to use red-black ordering to avoid updating constraints in the wrong order. We do this through the use of another uniform. In a single iteration, we dispatch the hair constraint with half the hair particle count and specify that we want the *even* indices to be processed first. Then we do the same thing again with the *odd* indices.

```
switch (stage) {
    case ...:
        // ...
        break;
        case STRETCH_SHEAR_CONSTRAINT:
        case BEND_TWIST_CONSTRAINT:
            for (int iter = 0; iter < simIterations; ++iter) {
                shader->setInt("rbgs", 0); // even
                glDispatchCompute(ceil(
                    (hairParticleCount / 2) /
                    DISPATCH_SIZE) + 1, 1, 1);
                glMemoryBarrier(GL_SHADER_STORAGE_BARRIER_BIT);

                shader->setInt("rbgs", 1); // odd
                glDispatchCompute(ceil(
                    (hairParticleCount / 2) /
```

```
                    DISPATCH_SIZE) + 1, 1, 1);
                glMemoryBarrier(GL_SHADER_STORAGE_BARRIER_BIT);
            }
            break;
    case ...:
        // ...
        break;
}
```

Listing 5.7: Dispatching the hair constraints using red-black ordering

Each dispatch incurs an overhead cost, so red-black Gauss-Seidel ordering can actually be detrimental to performance if one is not careful. We do not perform RGBS on the fluid constraint since the ordering of fluid particles does not strictly matter. This does lead to more non-determinism, however.

5.2.2 GPU Solver

At this stage of our simulation pipeline, we have finally moved onto the GPU. All of the following steps will performed in parallel.

Forces

First, we apply external forces. In our case, this is just gravity. We did not implement other forces such as wind or modify torque as that would be detrimental to appraising the overall appearance of our model. We also scale the effect of gravity slightly on hair strands depending on their wetness.

```
// apply external forces (gravity)
// `i_hf` represents a hair or fluid particle
void applyExternalForces(int i_hf) {
    int i_h = gToH(i_hf); // convert global index to hair index
    int i_f = gToF(i_hf); // convert global index to fluid index

    // hair gravity
    if (particles[i_hf].t == HAIR) {
        if (i_h == getRootVertex(i_h)) return;
        particles[i_hf].v += vec4(fv_gravity *
            max(1, particles[i_hf].d / 20) * dt, 0);
```

```
        // hair torque
        if (i_h == getTailVertex(i_h)) return;
        int j_h = toJ(i_h);
        rods[j_h].v += vec4(
            inverse(inertia)*(fv_torque - cross(rods[j_h].v.xyz,
            inertia * rods[j_h].v.xyz)) * dt, 0);
    } else if (particles[i_hf].t == FLUID) {
        // fluid gravity
        particles[i_hf].v += vec4(
            fv_gravity * particles[i_hf].d * dt, 0);
    }
}
```

Listing 5.8: Calculating external (gravity) forces on hair and fluid particles.

Density is calculated like so:

```
// compute the porous hair or fluid density.
// `i_fp` represents fluid or porous particles
void computeDensity(int i_fp) {
    int i_g = fToG(i_fp); // convert to global particle index
    float density = 0;
    ivec3 cell = posToCell(ps[i_g]);
    int minX = max(cell.x - 1, 0);
    int maxX = max(1, cell.x + 1);
    int minY = max(cell.y - 1, 0);
    int maxY = max(1, cell.y + 1);
    int minZ = max(cell.z - 1, 0);
    int maxZ = max(1, cell.z + 1);
    for (int x = minX; x <= maxX; ++x) {
        for (int y = minY; y <= maxY; ++y) {
            for (int z = minZ; z <= maxZ; ++z) {
                uint key = flatten(ivec3(x, y, z));
                int start = startIndices[key].startIndex;
                int end = startIndices[key].startIndex +
                    startIndices[key].particlesInBucket;
                for (int s = start; s < end; ++s) {
                    int j_g = cellEntries[s];
                    if (particles[j_g].t != FLUID) continue;
                    density += (1.f / particles[j_g].w) *
                        poly6Kernel(vec3(ps[i_g] - ps[j_g]),
                            smoothingRadius);
                }
            }
        }
    }
    if (particles[i_g].t == FLUID) {
        fluidDensities[i_fp] = density;
    } else if (particles[i_g].t == PORE) {
        poreData[i_fp - fluidParticleCount].density = density;
    }
}
```

Listing 5.9: Calculating the fluid density for fluid and porous particles.

At this stage, we begin to use our grid for finding nearby neighbours. This method is identical across all uses, so we will omit it from future references.

After density, viscosity, and fluid auxiliaries, we get to the prediction step. Our hair is modelled around a sphere representing a head. The roots of hair strands influence the movement of their immediate neighbour, but do not move according to the constraints. Thus, to update hair strands, we performed the following:

```
// given hair indx i_h:
// hair wetness, for rendering
particles[i_g].d = max(0,
    particles[i_g].d - fluidMassDiffusionFactor/5);

if (i_h == getRootVertex(i_h)) {
    ps[i_g] = headTrans * hairStrands[getStrandV(i_h)].root;
} else {
    ps[i_g] = particles[i_g].x + particles[i_g].v * dt;
}
```

Listing 5.10: Calculating the predicted positions of hair particles.

where `hairTrans` is the transform of the head object. Fluid particles are predicted as normal. Porous particles do not move under any forces, but rather influence the hair particles they lie between. Although we only use one sample, we can distribute the *weight* of porous particles to either hair particle depending on how close they are.

```
// reset porous particle positions
// `i_p` represents porous particles
void updatePorousPositions(int i_p) {
    int i_g = pToG(i_p);
    ps[i_g] = ps[poreData[i_p].startIndex] +
      (ps[poreData[i_p].endIndex]-ps[poreData[i_p].startIndex])*
      poreData[i_p].endStrength;
}
```

Listing 5.11: Calculating the predicted positions of hair particles.

Constraints

As mentioned, we solve hair constraints in two passes using RBGS ordering. Since the constraints only depend on one other value, we can compute the correction displacements

immediately instead of going through the effort of finding all nearby neighbours to influence. However, this is required for our density constraint. In addition to the density constraint, we compute the fluid surface tension:

```
/* apply surface tension */
vec3 grad = AAI12TKernelNorm(pij, smoothingRadius);
vec3 cohesion = imass * jmass * grad * -f_cohesion;
vec3 curveDiff = vec3(curveNormals[i_f] - curveNormals[j_f]);
vec3 curvature = imass * curveDiff * -f_curvature;
float densitySum = fluidDensities[i_f] + fluidDensities[j_f];
float K = (2 * restDensity) / densitySum;
vec3 surfaceTension = K * (cohesion + curvature);
deltaP += surfaceTension;
```

Listing 5.12: Calculating the surface tension for fluid particles.

We also model the fluid-pore adhesion force

```
/* apply hair adhesion */
float imass = 1 / particles[i_f].w; // inverse of an inverse
deltaP += -f_adhesion * imass *
    (restDensity * poreData[j_p].volume) *
    adhesionKernelNorm(pij, smoothingRadius);
```

Listing 5.13: Calculating the fluid adhesion for porous particles.

Clumping

Once we've calculated our constraints, we first add the hair-fluid clumping force before updating particle velocities and positions. We scale our clumping force depending on how far out the particle is relative to the length of its, so porous particles nearer to the root will clump weakly and particles nearer to the head will clump strongly.

```
// weightI is calculated at the beginning of the function
// its density is also checked
int strandCountJ = hairStrands[getStrandV(
        poreData[j_p].startIndex)].nVertices;
int strandStartJ = hairStrands[getStrandV(
        poreData[j_p].startIndex)].startVertexIdx;
float weightJ = (poreData[j_p].startIndex -
    strandStartJ + 1.f) / float(strandCountJ);

if (poreData[j_p].density < 1e-9) continue;
float avgSat = (calculateSaturation(i_p) +
    calculateSaturation(j_p)) / 2;
float psi_i = restDensity * poreData[i_p].volume * weightI;
float psi_j = restDensity * poreData[j_p].volume * weightJ;
float U = 1 - min(1, poreData[j_p].density * restDensityInv);
if (sqLen(dff) == 0) continue;
vec3 nforce = avgSat * U * psi_i * psi_j *
    adhesionKernelNorm(dff, smoothingRadius);
totalClumpingForce += nforce;
```

Listing 5.14: Calculating the clumping force for fluid and porous particles.

5.3 Rendering Stage

In this section, we will explain how we used tessellation shaders to create hair strands and how we used framebuffers to perform screen-space fluid rendering.

5.3.1 Tessellation

To render our hair strands, we first need to specify which points are used for which line segments. Our strands will be a series of connected splines. Catmull-Rom splines require four points to generate a spline: two outer control points, and the inner start and end points. However, the layout of our particles in memory assumes that we will have only the inner points tightly packed. We would need to duplicate vertices to tell the compiler which ones go in which order, but since our vertices are part of our physics calculations,

this will not suffice. We can instead make use of index buffers to detail which particles are meant to be used and when.

Loading Index Buffers

First, when creating our SSBOs, we can also create a VAO for our hair and an EBO, or element buffer object. We generated vertex indices during our hair particle generation, which is shown in Appendix A. We can store this data with

```
glCreateVertexArrays(1, &VAO);
glCreateBuffers(1, &EBO);
glNamedBufferStorage(EBO, sizeof(indices[0]) * indices.size(),
&indices[0], GL_MAP_READ_BIT);
glVertexArrayElementBuffer(VAO, EBO);
```

Listing 5.15: Setting up our particle indices for hair tessellation.

Our hair strands will each have their own particle count and root positions. We cannot use instancing to duplicate geometry since our particles are used in physics calculations, but we also cannot send our hair particle data separately. Instead we can use *indirect rendering*. This feature was added in OpenGL 4.0 and allows us to specify a command buffer that holds the information about our draw commands. Then, whenever we want to draw our geometry, we need only bind the buffer object.

To use indirect rendering with indexing, we will call `glMultiDrawElementsIndirect`. [1] We create a draw command struct as according to the OpenGL specifications (Segal and Akeley, 2022):

```
struct IndirectElementDrawCommand {
    unsigned int indexCount;
    unsigned int instanceCount;
    unsigned int baseIndex;
    unsigned int baseVertex;
    unsigned int baseInstance;
};
```

Listing 5.16: The `glMultiDrawElementsIndirect` command buffer.

[1] Strictly speaking, `MultiDraw*` commands are meant for instancing *different types* of geometry with differing vertex counts. Although we provide an interface for creating hair strands of any type of geometry, we only test the same strands. We leave the instance count as 1 in our implementation, but note that this should be considered wasteful.

and create a struct for each hair strand. We store our commands in another SSBO and bind it before drawing our geometry:

```
glBindBuffer(GL_DRAW_INDIRECT_BUFFER, commandBuffer);
glPatchParameteri(GL_PATCH_VERTICES, 4);
glMultiDrawElementsIndirect(
    GL_PATCHES, GL_UNSIGNED_INT, 0, numStrands, 0);
```

Listing 5.17: Using `glMultiDrawElementsIndirect` to draw our hair strands.

Tessellating Hairs

We start in the vertex shader. When we render our hairs, we want to sample the wetness at each hair particle, so we need to pass the particle index through the tessellation shader stages. We also want the weight of each particle (see Section 5.2.2) to be interpolated along the entire strand, and eye space position of the particles, and the position itself. We pass all of this through to the TCS, where we start actually generating our spline. We pass in the particles SSBO and index the current particle using the built-in command `gl_VertexID`. Our implementation of Catmull-Rom splines using tessellation shaders is taken from Enoch Tsang's implementation. (Tsang, 2017). We detail her method below.

We first assign the two outer levels of `gl_TessLevelOuter`, as we are using isolines.

```
gl_TessLevelOuter[0] = float(1);
gl_TessLevelOuter[1] = float(8);
```

Listing 5.18: Selecting our level of tessellation.

We use per-patch output variables to output the control points for each 4-vertex patch and send out the vertex positions themselves. We also pass our input from the vertex shader directly to the TES.

```
if (gl_InvocationID == 0) {
    p_1 = gl_in[0].gl_Position; // start control
    p_2 = gl_in[3].gl_Position; // end control
    gl_out[gl_InvocationID].gl_Position = gl_in[1].gl_Position;
} else if (gl_InvocationID == 1) {
    gl_out[gl_InvocationID].gl_Position = gl_in[2].gl_Position;
}
```

Listing 5.19: Selecting spline controls and line points.

In the TES, we perform the calculation to generate our spline and interpolate over the data we received from the vertex shader. Tsang uses the tessellated vertex from the TPG to create a matrix and then solves it to create the output location of our vertex.

```
vec4 p0 = gl_in[0].gl_Position;
vec4 p1 = gl_in[1].gl_Position;
float b0 = (0.f) - (1.f * u) + (2.f * u*u) - (1.f * u*u*u);
float b1 = (2.f) + (0.f * u) - (5.f * u*u) + (3.f * u*u*u);
float b2 = (0.f) + (1.f * u) + (4.f * u*u) - (3.f * u*u*u);
float b3 = (0.f) + (0.f * u) - (1.f * u*u) + (1.f * u*u*u);
vec4 p = (b0*p_1 + b1*p0 + b2*p1 + b3*p_2);
```

Listing 5.20: Calculating the new position of the tessellated vertex from the Catmull-Rom spline.

Rendering

Like we mentioned before, we used the Cook-Torrance BRDF to light our strands. We first use the eye space position we computed in the vertex shader to output the depth of each tessellated vertex. This will be used later when rendering our fluid.

```
vec4 pixelPos = vec4(es.eyeSpacePos, 1);
vec4 clipSpacePos = proj * pixelPos;
float ndc = clipSpacePos.z / clipSpacePos.w;
gl_FragDepth = ndc *.5 + .5;
float outDepth = -pixelPos.z;
DepthColour = vec4(vec3(outDepth/150), 1);
```

Listing 5.21: Outputting the depth value.

Next, we use the wetness of the original hair particle, which was interpolated in the TES, to attenuate the albedo colour of the hair strand. The wetter a strand is, the darker it will become, so we multiply the colour by `1 - wetness`. We use used the GGX BRDF (Walter et al., 2007). Our implementation of this is available in Appendix C. The Fresnel lighting is attenuated by the particle weight to emphasize the appearance of a rim light. It's also damped by the distance of the hair to the camera, so that it is less apparent when closer.

5.3.2 Framebuffers

The final step in our simulation pipeline is rendering the fluid. To perform screen-space fluid rendering, we need a depth map, thickness map, narrow-range blur filter, normal map, and environment map. Before the cubemap, head, and hair strands are rendered, we bind the environment FBO, which has a two draw buffers associated with it. After everything has been drawn, we unbind it and render the fluid.

Firstly, we render point sprites out and calculate the depth the same way we calculate any other value. To calculate the thickness, we disable the depth test and enable blending, then render each point again. This allows us to use the layered opacity as a thickness map. The thickness is rendered into a framebuffer a fraction of the screen size since blending is "fill-rate intensive" (Green, 2010).

Once again, we wish to render a set of particles that is already present in another buffer. Fluid particles are single objects that don't need to be considered as part of a group, so we can take a different approach than we did for the hair strands. We use `glDrawArraysIndirect` to render our fluid particles. Since every particle will always be one particle, we have no use for the differing geometries that `MultiDraw*` commands grant us. The draw command is also different:

```
struct IndirectArrayDrawCommand {
    unsigned int vertexCount;
    unsigned int instanceCount;
    unsigned int baseVertex;
    unsigned int baseInstance;
};
```

Listing 5.22: The `glDrawArraysIndirect` command buffer.

Calling it is simple:

```
glBindBuffer(GL_DRAW_INDIRECT_BUFFER, commandBuffer);
glDrawArraysIndirect(GL_POINTS, 0);
```

Listing 5.23: Drawing fluid particles.

We use Nghia Truong's implementation of their narrow-range filter pass and normal pass. The final composition pass is based on theirs and combines all former textured together to render the final output.

5.4 Summary

This chapter explained our implementation and the steps we took to realise our idea. We generated our geometry and set up buffers to be used on the GPU. We initialised our nearest neighbour grid, dispatched our data to a compute shader, and ran all computations in parallel. We provided code snippets of the significant portions of our implementation to give a better understand of how our code works.

Chapter 6

Evaluation

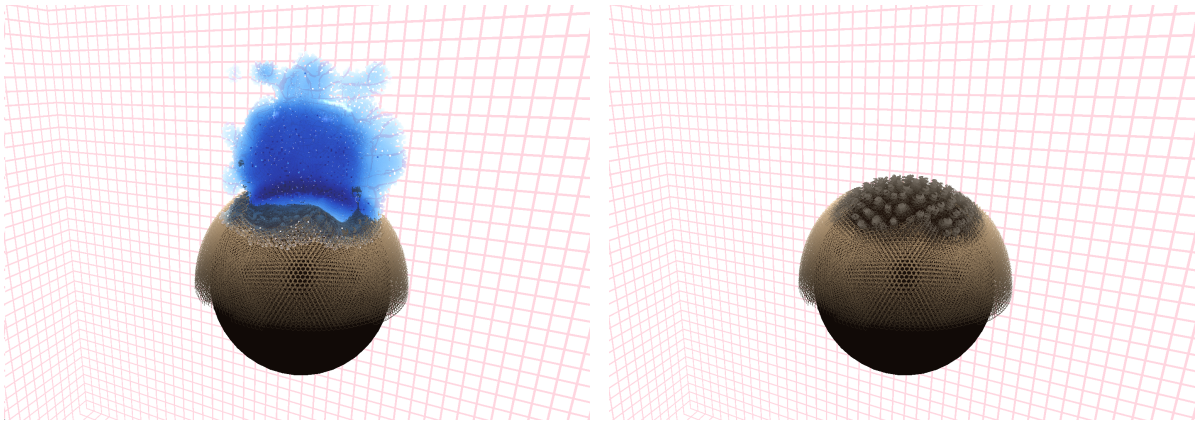
This chapter will evaluate the results of our program and answer the research question we posed in Chapter 4. We will test our program in different states with different configurations and find how usable it is for real-time contexts, as well as exposing any potential flaws in our design.

6.1 Criteria

Figuring out what constitutes "real-time" depends largely on the person. With the advent of more powerful GPUs and lower-latency screens, the threshold for what is considered playable continues to increase. Low-end devices such as mobile phones or laptops with integrated GPUs rarely enjoy the experience of an application running at 90 frames per second, because their devices are locked to 60.

We look to other modern visual media to find an answer. Cinemas and animations typically run at 24 FPS. (DeGuzman, 2023) While these standards are based on older requirements, few complain about their use in the modern day. Modern video games aim for at least 60 FPS, but for the wider field of interactive applications, this value can be lowered. Interactivity therefore is more dependent on the view, rather than the actions, so times lower than 24 FPS can be considered acceptable. For these reasons, we define "real-time" to be a minimum of 20 FPS and "interactive" to be a minimum of 10 FPS.

We also lack run-time comparisons. The simulation by Rungjiratananon et al. (2012) models only clumping forces and does not consider more complex forces. Lin (2014) model, whose ours is based on, requires time-steps on the order of $\frac{1}{500}$ to $\frac{1}{2000}$. Fei et al. (2017) reports frames taking up to 57 seconds. There are no real-time wet-hair simulations similar enough to our method to properly compare to, so we provide our research as a foundation for other researchers.



(a) Fluid visible (b) Fluid removed

Figure 6.1: Water interacting with 5,287 hair strands. Note how the hair's colour darkens and it clumps only with wet hairs.

6.2 Experiments

We tested our implementation with varying hair strand counts, hair particle counts, and solver sub-steps. All tests were run on an Intel Core i7-1260P processor with an Intel Iris Xe integrated GPU. The time-step was $\frac{1}{30}$. We used 5 substeps and 5 hair constraint iterations.

6.2.1 Physical Experiments

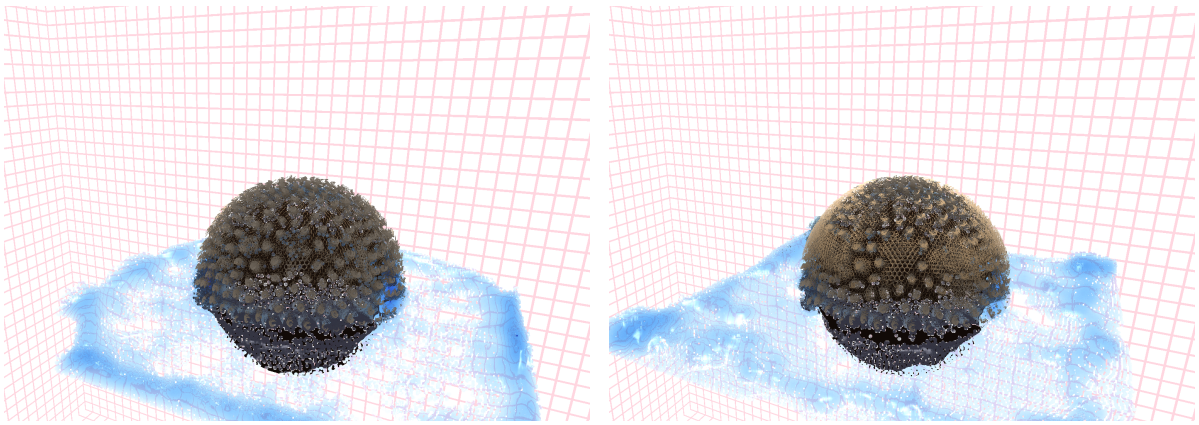
Firstly, we will test the performance. In our experiment, we dropped a cube of 20,000 fluid particles onto the head and measured the average FPS when it was fully submerged. We disabled fluid rendering during testing.

Strand Count	Particles per Strand	Average FPS
1,365	10	20
1,365	15	12
1,365	30	5
5,287	10	5
5,287	15	2

Table 6.1: The average FPS when testing different hair densities and particles.

Substeps	Average FPS
3	22
10	9
15	5

Table 6.2: The average FPS when testing different substep sizes. We used 1,365 hair strands @ 15 particles each.



(a) Hairs still wet (b) Hairs drying out

Figure 6.2: Fluid diffusing from the 5,287 strands over time. The colour returns and the hairs are no longer clumped.

6.3 Summary

We tested our implementation with different hair strand and particle counts. Using smaller numbers of hair strands and particle sizes allows us to achieve "real-time" and "interactive" rates with our simulation. Using PBD keeps the simulation stable even at time-step size of $\frac{1}{30}$, which previous works struggled to achieve.

Chapter 7

Conclusions & Future Work

From our findings, we conclude that a real-time wet hair simulation using afro-textured hair is viable given ample processing power. The benchmark machine does not have the necessary processing speed to simulate more than a few thousand strands. We have severely undershot the actual count of human hairs by a factor of over 30×; however, we are not aiming for a perfect simulation, so this loss is acceptable.

7.1 Future Work

The fluid simulation affected the substep size test. As mentioned, it runs fine at lower substep sizes, but higher ones make it far slower. Using a faster method for the fluid simulation would help in increasing the simulation speed.

The density of the hair was another factor. When calculating the clumping force, the number of surrounding porous particles is always high, so a lot of wasted checks were constantly being performed. One possible solution is restricting the porous particles that can be clumped with by their weight, since particles closer to the root will are mostly pulled along by hairs that are further up.

It would be interesting to see different types of hair styles. We only tested one length and colour, but our simulation is designed in such a way that any type of hair is possible, including straighter hair. While an afro is a good benchmark, tying hair together could be another good way to test our model.

Using tessellation shaders had good and bad outcomes. We were able to create distinct hair strands using only splines, but tessellation shaders only subdivide, not create. We cannot make the hair strands thicker and easier to see. The geometry shader has functionality for generating new vertices, but is generally considered to be much slower than tessellation shaders. We also could work on implementing a level-of-detail (LOD)

system that varies the tessellation amount depending on the distance to the camera.

Something that could help with increasing the stiffness of the hair without requiring additional substeps is Extended Position-Based Dynamics (XPBD).(Macklin et al., 2016) This method improves the original PBD method by modelling itself on *compliance*, which is the opposite of stiffness. Deul et al. (2018) uses XPBD with Cosserat rods in their direct solver. Combining their method with Lin's hair-fluid coupling remains to be seen, but could be another useful step in furthering this field of research.

Bibliography

Akinci, N., Akinci, G., and Teschner, M. (2013). Versatile surface tension and adhesion for sph fluids. *ACM Trans. Graph.*, 32(6).

Akinci, N., Ihmsen, M., Akinci, G., Solenthaler, B., and Teschner, M. (2012). Versatile rigid-fluid coupling for incompressible sph. *ACM Trans. Graph.*, 31(4).

Anjyo, K.-i., Usami, Y., and Kurihara, T. (1992). A simple method for extracting the natural beauty of hair. *SIGGRAPH Comput. Graph.*, 26(2):111–120.

Becker, M. and Teschner, M. (2007). Weakly compressible sph for free surface flows. In *Proceedings of the 2007 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, SCA '07, page 209–217, Goslar, DEU. Eurographics Association.

Bender, J. and Koschier, D. (2015). Divergence-free smoothed particle hydrodynamics. In *Proceedings of the 14th ACM SIGGRAPH / Eurographics Symposium on Computer Animation*, SCA '15, page 147–155, New York, NY, USA. Association for Computing Machinery.

Bergou, M., Wardetzky, M., Robinson, S., Audoly, B., and Grinspun, E. (2008). Discrete elastic rods. In *ACM SIGGRAPH 2008 Papers*, SIGGRAPH '08, New York, NY, USA. Association for Computing Machinery.

Bertails, F. (2009). Linear time super-helices. *Computer Graphics Forum*, 28(2):417–426.

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 Trans. Graph.*, 25(3):1180–1187.

Bhokare, G., Montalvo, E., Diaz, E., and Yuksel, C. (2024). Real-time hair rendering with hair meshes. In *SIGGRAPH 2024 Conference Papers*, SIGGRAPH '24, pages 61:1–61:10, New York, NY, USA. ACM Press.

Bindel, D. (2010). Lecture 14: Iterative methods and sparse linear algebra. PowerPoint slides. [Online; accessed 15-April-2025].

Black, N. and Moore, S. (2025). Successive overrelaxation method. From MathWorld–A Wolfram Web Resource, created by Eric W. Weisstein [Online; accessed 15-April-2025].

Brackbill, J., Kothe, D., and Ruppel, H. (1988). Flip: A low-dissipation, particle-in-cell method for fluid flow. *Computer Physics Communications*, 48(1):25–38.

Bruderlin, A. (1999). A Method to Generate Wet and Broken-up Animal Fur . In *Computer Graphics and Applications, Pacific Conference on*, page 242, Los Alamitos, CA, USA. IEEE Computer Society.

Chang, J. T., Jin, J., and Yu, Y. (2002). A practical model for hair mutual interactions. In *Proceedings of the 2002 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, SCA '02, page 73–80, New York, NY, USA. Association for Computing Machinery.

Clavet, S., Beaudoin, P., and Poulin, P. (2005). Particle-based viscoelastic fluid simulation. In *Proceedings of the 2005 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, SCA '05, page 219–228, New York, NY, USA. Association for Computing Machinery.

Cloete, E., Khumalo, N. P., and Ngoepe, M. N. (2019). The what, why and how of curly hair: a review. *Proc. Math. Phys. Eng. Sci.*, 475(2231):20190516.

Daldegan, A., Thalmann, N. M., Kurihara, T., and Thalmann, D. (1993). An integrated system for modeling, animating and rendering hair. *Computer Graphics Forum*, 12(3):211–221.

Darke, A. M., Olander, I., and Kim, T. (2024). More than killmonger locs: A style guide for black hair (in computer graphics). In *ACM SIGGRAPH 2024 Courses*, SIGGRAPH Courses '24, New York, NY, USA. Association for Computing Machinery.

DeGuzman, K. (2023). Why are movies 24 fps — the cinematic frame rate explained.

Deul, C., Kugelstadt, T., Weiler, M., and Bender, J. (2018). Direct position-based solver for stiff rods. *Computer Graphics Forum*, 37(6):313–324.

Epic Games (2025a). Fluid simulation overview. [Online; accessed 04-April-2025].

Epic Games (2025b). Hair physics overview. [Online; accessed 05-April-2025].

Fei, Y. R., Maia, H. T., Batty, C., Zheng, C., and Grinspun, E. (2017). A multi-scale model for simulating liquid-hair interactions. *ACM Trans. Graph.*, 36(4).

Ferziger, J. H., Perić, M., and Street, R. L. (2020). *Computational Methods for Fluid Dynamics, Fourth Edition*. Springer Cham.

Gazzola, M., Dudte, L., McCormick, A., and Mahadevan, L. (2018). Forward and inverse problems in the mechanics of soft filaments. *Royal Society open science*, 5(6):171628.

Gingold, R. A. and Monaghan, J. J. (1977). Smoothed particle hydrodynamics: theory and application to non-spherical stars. *Monthly Notices of the Royal Astronomical Society*, 181(3):375–389.

Green, S. (2010). Screen space fluid rendering for games.

Gross, M. and Pfister, H., editors (2007). *Point-Based Graphics*. Morgan Kaufmann.

Gupta, R. and Magnenat-Thalmann, N. (2007). Interactive rendering of optical effects in wet hair. In *Proceedings of the 2007 ACM Symposium on Virtual Reality Software and Technology*, VRST '07, page 133–140, New York, NY, USA. Association for Computing Machinery.

Harlow, F. H., Richtmyer, R. D., and Evans, M. (1955). *A machine calculation method for hydrodynamic problems*. Los Alamos Scientific Laboratory of the University of California.

Harlow, F. H. and Welch, J. E. (1965). Numerical calculation of time dependent viscous incompressible flow of fluid with free surface. *The Physics of Fluids*, 8(12):2182–2189.

Harris, M., Sengupta, S., and Owens, J. D. (2007). Chapter 39. parallel prefix sum (scan) with cuda. [Online; accessed 16-April-2025].

Hoetzlein, R. (2014). Fluids v.3: Interactive million particle fluids. [Online; accessed 16-April-2025].

Hu, Y., Fang, Y., Ge, Z., Qu, Z., Zhu, Y., Pradhana, A., and Jiang, C. (2018). A moving least squares material point method with displacement discontinuity and two-way rigid body coupling. *ACM Transactions on Graphics*, 37(4):150.

Ihmsen, M., Cornelis, J., Solenthaler, B., Horvath, C., and Teschner, M. (2014). Implicit incompressible sph. *IEEE Transactions on Visualization and Computer Graphics*, 20(3):426–435.

Jiang, C., Schroeder, C., Selle, A., Teran, J., and Stomakhin, A. (2015). The affine particle-in-cell method. *ACM Trans. Graph.*, 34(4).

Kajiya, J. T. and Kay, T. L. (1989). Rendering fur with three dimensional textures. In *Proceedings of the 16th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '89, page 271–280, New York, NY, USA. Association for Computing Machinery.

Keiser, R., Adams, B., Gasser, D., Bazzi, P., Dutre, P., and Gross, M. (2005). A unified lagrangian approach to solid-fluid animation. In *Proceedings Eurographics/IEEE VGTC Symposium Point-Based Graphics, 2005.*, pages 125–148.

Khronos Group (2009). Opencl overview - the khronos group inc. [Online; accessed 04-April-2025].

Koschier, D., Bender, J., Solenthaler, B., and Teschner, M. (2022). A survey on sph methods in computer graphics. *Computer Graphics Forum*, 41(2):737–760.

Kugelstadt, T. and Schömer, E. (2016). Position and Orientation Based Cosserat Rods . In Kavan, L. and Wojtan, C., editors, *Eurographics/ ACM SIGGRAPH Symposium on Computer Animation*. The Eurographics Association.

Lenaerts, T., Adams, B., and Dutré, P. (2008). Porous flow in particle-based fluid simulations. *ACM Trans. Graph.*, 27(3):1–8.

Levien, R. (2020). Prefix sum on vulkan. [Online; accessed 16-April-2025].

Lewin, C. and Barré-Brisebois, C. (2024). Pb-mpm. [Computer software; accessed 16-April-2025].

Li, L., Li, R., and Yu, J. (2016). A mass spring based 3d virtual hair dynamic system for straight and curly hair. In *2016 35th Chinese Control Conference (CCC)*, pages 6982–6987.

Li, M. and Li, H. (2023). An efficient non-iterative smoothed particle hydrodynamics fluid simulation method with variable smoothing length. *Visual Computing for Industry, Biomedicine, and Art*.

Lin, W.-C. (2014). Coupling Hair with Smoothed Particle Hydrodynamics Fluids. In Bender, J., Duriez, C., Jaillet, F., and Zachmann, G., editors, *Workshop on Virtual Reality Interaction and Physical Simulation*. The Eurographics Association.

Lucy, L. B. (1977). A numerical approach to the testing of the fission hypothesis. *Astronomical Journal*, 82:1013–1024.

Macklin, M. and Müller, M. (2013). Position based fluids. *ACM Trans. Graph.*, 32(4).

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*, MIG '16, page 49–54, New York, NY, USA. Association for Computing Machinery.

Macklin, M., Müller, M., Chentanez, N., and Kim, T.-Y. (2014). Unified particle physics for real-time applications. *ACM Trans. Graph.*, 33(4).

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*, SCA '19, New York, NY, USA. Association for Computing Machinery.

Markidis, S., Olshevsky, V., Sishtla, C. P., Chien, S. W. D., Laure, E., and Lapenta, G. (2018). Polypic: The polymorphic-particle-in-cell method for fluid-kinetic coupling. *Frontiers in Physics*, Volume 6 - 2018.

Marschner, S. R., Jensen, H. W., Cammarano, M., Worley, S., and Hanrahan, P. (2003). Light scattering from human hair fibers. *ACM Trans. Graph.*, 22(3):780–791.

Milo, R., Jorgensen, P., Moran, U., Weber, G., and Springer, M. (2009). Bionumbersâ€"the database of key numbers in molecular and cell biology. *Nucleic Acids Research*, 38(suppl_1):D750–D753.

Monaghan, J., Thompson, M., and Hourigan, K. (1994). Simulation of free surface flows with sph. In *Advances in Computational Methods in Fluid Dynamics*, volume 196, pages 375–380, United States of America. American Society of Mechanical Engineers (ASME). Proceedings of the 1994 ASME Fluids Engineering Division Summer Meeting. Part 9 (of 18) ; Conference date: 19-06-1994 Through 23-06-1994.

Monaghan, J. J. (1992). Smoothed particle hydrodynamics. *Annual Review of Astronomy and Astrophysics*, 30(Volume 30, 1992):543–574.

Morris, J., Zhu, Y., and Fox, P. (1999). Parallel simulations of pore-scale flow through porous media. *Computers and Geotechnics*, 25(4):227–246.

Müller, M., Charypar, D., and Gross, M. (2003). Particle-based fluid simulation for interactive applications. In *Proceedings of the 2003 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, SCA '03, page 154–159, Goslar, DEU. Eurographics Association.

Müller, M., Keiser, R., Nealen, A., Pauly, M., Gross, M., and Alexa, M. (2004). Point based animation of elastic, plastic and melting objects. In *Proceedings of the 2004 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, SCA '04, page 141–151, Goslar, DEU. Eurographics Association.

Müller, M., Solenthaler, B., Keiser, R., and Gross, M. (2005). Particle-based fluid-fluid interaction. In *Proceedings of the 2005 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, SCA '05, page 237–244, New York, NY, USA. Association for Computing Machinery.

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 . In Bender, J., Kuijper, A., Fellner, D. W., and Guerin, E., editors, *Workshop on Virtual Reality Interaction and Physical Simulation*. The Eurographics Association.

Müller, M., Schirm, S., Teschner, M., Heidelberger, B., and Gross, M. (2004). Interaction of fluids with deformable solids. *Computer Animation and Virtual Worlds*, 15(3-4):159–171.

nialltl (2019). notes and examples for the material point method. [Online; accessed 15-April-2025].

NVIDIA (2007). Cuda toolkit - free tools and training — nvidia developer. [Online; accessed 15-April-2025].

OpenGL Wiki (2020). Tessellation — opengl wiki,. [Online; accessed 15-April-2025].

OpenGL Wiki (2022). Primitive assembly — opengl wiki,. [Online; accessed 15-April-2025].

Pai, D. K. (2002). Strands: Interactive simulation of thin solids using cosserat models. *Computer Graphics Forum*, 21(3):347–352.

Patrick, D., Bangay, S., and Lobb, A. (2004). Modelling and rendering techniques for african hairstyles. In *Proceedings of the 3rd International Conference on Computer*

Graphics, Virtual Reality, Visualisation and Interaction in Africa, AFRIGRAPH '04, page 115–124, New York, NY, USA. Association for Computing Machinery.

Rosenblum, R. E., Carlson, W. E., and Tripp, E. R. (1991). Simulating the structure and dynamics of human hair: Modelling, rendering and animation. *Comput. Animat. Virtual Worlds*, 2:141–148.

Rungjiratananon, W., Kanamori, Y., and Nishita, T. (2012). Wetting effects in hair simulation. *Computer Graphics Forum*, 31(7):1993–2002.

Scheuermann, T. (2004). Practical real-time hair rendering and shading. In *ACM SIGGRAPH 2004 Sketches*, SIGGRAPH '04, page 147, New York, NY, USA. Association for Computing Machinery.

Segal, M. and Akeley, K. (2022). The opengl® graphics system: A specification. [Online; accessed 15-April-2025].

Selle, A., Lentine, M., and Fedkiw, R. (2008). A mass spring model for hair simulation. *ACM Trans. Graph.*, 27(3):1–11.

Shi, A., Wu, H., Parr, J., Darke, A. M., and Kim, T. (2023). Lifted curls: A model for tightly coiled hair simulation. *Proc. ACM Comput. Graph. Interact. Tech.*, 6(3).

Solenthaler, B. and Pajarola, R. (2009). Predictive-corrective incompressible sph. In *ACM SIGGRAPH 2009 Papers*, SIGGRAPH '09, New York, NY, USA. Association for Computing Machinery.

Spillmann, J. and Teschner, M. (2007). Corde: Cosserat rod elements for the dynamic simulation of one-dimensional elastic objects. In *Proceedings of the 2007 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, SCA '07, page 63–72, Goslar, DEU. Eurographics Association.

The OpenMP ARB (2007). About us - openmp. [Online; accessed 04-April-2025].

Truong, N. and Yuksel, C. (2018). A narrow-range filter for screen-space fluid rendering. *Proc. ACM Comput. Graph. Interact. Tech.*, 1(1).

Tsang, E. (2017). Tessellation shaders and isolines.

Tychonievich, L. (2024). Cs 418 – smoothed particle hydrodynamics. [Computer software; accessed 16-April-2025].

Umetani, N., Schmidt, R., and Stam, J. (2015). Position-based elastic rods. In *Proceedings of the ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, SCA '14, page 21–30, Goslar, DEU. Eurographics Association.

Walter, B., Marschner, S. R., Li, H., and Torrance, K. E. (2007). Microfacet models for refraction through rough surfaces. In *Proceedings of the 18th Eurographics Conference on Rendering Techniques*, EGSR'07, page 195–206, Goslar, DEU. Eurographics Association.

Ward, K., Bertails, F., Kim, T.-y., Marschner, S. R., Cani, M.-p., and Lin, M. C. (2007a). A survey on hair modeling: Styling, simulation, and rendering. *IEEE Transactions on Visualization and Computer Graphics*, 13(2):213–234.

Ward, K., Galoppo, N., and Lin, M. (2004). Modeling hair influenced by water and styling products. *Proc. of Computer Animation and Social Agents*.

Ward, K., Galoppo, N., and Lin, M. (2007b). Interactive virtual hair salon. *Presence: Teleoperators and Virtual Environments*, 16(3):237–251.

Wikimedia Commons (2025). File:abbey lincoln in 1966.jpg — wikimedia commons, the free media repository. [Online; accessed 17-April-2025].

Wu, H., Shi, A., Darke, A., and Kim, T. (2024). Curly-cue: Geometric methods for highly coiled hair. In *SIGGRAPH Asia 2024 Conference Papers*, SA '24, New York, NY, USA. Association for Computing Machinery.

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.

Zhu, Y. and Bridson, R. (2005). Animating sand as a fluid. *ACM Trans. Graph.*, 24(3):965–972.

Zhu, Y., Fox, P. J., and Morris, J. P. (1999). A pore-scale numerical model for flow through porous media. *International Journal for Numerical and Analytical Methods in Geomechanics*, 23(9):881–904.

Zitz, S. (2022). Hair type 101: Everything you need to know about straight, wavy, curly, and coily hair. [Online; accessed 02-April-2025].

Appendix

A: Hair Strand Generation

```
// Generate hair vertices for strand strandIdx
// This strand points in direction dir
void initialiseVertices(int strandIdx, vec3 dir) {
    int vS = hairStrands[strandIdx].startVertexIdx;
    int nV = hairStrands[strandIdx].nVertices;
    for (int i = 0; i < nV; ++i) {
        /* Generate hair strand vertex */
        float ang = (nCurls * 2 * PI) / nV * i;
        vec3 p = vec3(rad * cos(ang), strandLength -
            (strandLength / (float)nV) * i, rad * sin(ang));

        /* Rotate vertex to point in away from the head */
        vec3 from = Util::UP;
        vec3 to = -dir; // coils grow downwards
        vec3 axs = cross(from, to);
        // Handle similar to and from directions
        float cosTheta = dot(from, to);
        if (cosTheta < -1 + 1e-3 || length2(axs) < 1e-9) {
            axs = cross(vec3(1.0f, 0.0f, 0.0f), from);
            if (length2(axs) < 1e-9)
                axs = cross(vec3(0.0f, 0.0f, 1.0f), from);
        }
        axs = normalize(axs);
        // rotate `from` about `axs` towards `to`
        float rotAngle = acos(cosTheta);
        quat q = angleAxis(rotAngle, axs);
```

```
            p = rotate(q, p);

            /* Add vertex/particle to this hair strand */
            Particle part = Particle(p, vec3(0),
                i == 0 ? 0 : 1, HAIR);
            particles.push_back(part);
            ps.push_back(vec4(p, 0));

            /* Add spline indices */
            if (i <= 3) {
                indices.push_back(i);
            } else {
                indices.push_back(i - 3);
                indices.push_back(i - 2);
                indices.push_back(i - 1);
                indices.push_back(i);
            }
        }

        hairStrands[strandIdx].l0 =
            distance(particles[vS].x, particles[vS + 1].x);

        // offset strand to head position
        vec4 offs = hairStrands[strandIdx].root -
            particles[vS].x + headTrans[3];
        for (int i = vS; i < vS + nV; ++i) {
            particles[i].x += offs;
            ps[i] += offs;
        }
    }
```

Listing 1: Generating a hair coil. Each particle is generated in a coil pointing upwards which is then rotated to face a given normal direction. Particle indices are added to an index buffer for tessellation.

B: Atomic Bump Allocation

```
/* Reset the values of the particle tables. */
#version 460 core
#define LOCAL_SIZE 8
layout (local_size_x = LOCAL_SIZE) in;

struct BucketData {
    int startIndex;
    int particlesInBucket;
    int nextParticleSlot;
    int pd;
};

layout(std430, binding=1) buffer GridStartIndices {
    BucketData startIndices[];
};
layout(std430, binding=2) buffer GridCellEntries {
    int cellEntries[];
};
layout(std430, binding=3) buffer TotalBucketParticleCount {
    int particleCountIndex[];
};
ivec3 posToCell(vec4 v);
uint flatten(ivec3 cell);

void main() {
    uint gid = gl_GlobalInvocationID.x;
    if (gid >= startIndices.length()) return;
    atomicMin(startIndices[gid].startIndex, 0);
    atomicMin(startIndices[gid].particlesInBucket, 0);
    atomicMin(startIndices[gid].nextParticleSlot, 0);
    if (gid >= cellEntries.length()) return;
    atomicMin(cellEntries[gid], 0);
    atomicMin(particleCountIndex[0], 0);
}
```

Listing 2: Resetting the buffers used in the grid.

```
/* Add each particle to its bin. */
#version 460 core
#define LOCAL_SIZE 8
layout (local_size_x = LOCAL_SIZE) in;

struct BucketData {
    int startIndex;
    int particlesInBucket;
    int nextParticleSlot;
    int pd;
};

layout(std430, binding=0) buffer PredictedPositions {
    vec4 ps[];
};
layout(std430, binding=1) buffer GridStartIndices {
    BucketData startIndices[];
};
layout(std430, binding=2) buffer GridCellEntries {
    int cellEntries[];
};
ivec3 posToCell(vec4 v);
uint flatten(ivec3 cell);

void main() {
    uint gid = gl_GlobalInvocationID.x;
    if (gid >= ps.length()) return;

    uint key = flatten(posToCell(ps[gid]));
    atomicAdd(startIndices[key].particlesInBucket, 1);
}
```

Listing 3: The counting step. Each particle adds 1 to its bucket count.

```
/* Allocate space for this bucket. */
#version 460 core
#define LOCAL_SIZE 8
layout (local_size_x = LOCAL_SIZE) in;

struct BucketData {
    int startIndex;
    int particlesInBucket;
    int nextParticleSlot;
    int pd;
};
layout(std430, binding=0) buffer PredictedPositions {
    vec4 ps[];
};
layout(std430, binding=1) buffer GridStartIndices {
    BucketData startIndices[];
};
layout(std430, binding=2) buffer GridCellEntries {
    int cellEntries[];
};
layout(std430, binding=3) buffer TotalBucketParticleCount {
    int particleCountIndex[];
};
ivec3 posToCell(vec4 v);
uint flatten(ivec3 cell);

void main() {
    uint gid = gl_GlobalInvocationID.x;
    if (gid >= startIndices.length()) return;
    startIndices[gid].startIndex =
        atomicAdd(particleCountIndex[0],
        startIndices[gid].particlesInBucket);
}
```

Listing 4: The allocation step. Each bucket adds its particle count to a shared integer atomically, finding its starting index within the `startIndices` table.

```
/* Insert each particle into its bin. */
#version 460 core
#define LOCAL_SIZE 8
layout (local_size_x = LOCAL_SIZE) in;

struct BucketData {
    int startIndex;
    int particlesInBucket;
    int nextParticleSlot;
    int pd;
};

layout(std430, binding=0) buffer PredictedPositions {
    vec4 ps[];
};
layout(std430, binding=1) buffer GridStartIndices {
    BucketData startIndices[];
};
layout(std430, binding=2) buffer GridCellEntries {
    int cellEntries[];
};
ivec3 posToCell(vec4 v);
uint flatten(ivec3 cell);

void main() {
    uint gid = gl_GlobalInvocationID.x;
    if (gid >= ps.length()) return;

    uint key = flatten(posToCell(ps[gid]));
    int nid = atomicAdd(startIndices[key].nextParticleSlot, 1);
    cellEntries[startIndices[key].startIndex + nid] = int(gid);
}
```

Listing 5: The insertion step. Each particle inserts itself into an available bucket slot.

```
/* Helper functions. */

#version 460 core

struct BucketData {
    int startIndex;
    int particlesInBucket;
    int nextParticleSlot;
    int pd;
};

layout(std430, binding=1) buffer GridStartIndices {
    BucketData startIndices[];
};

layout(location = 0) uniform float gridCellSize;

// Get the grid cell containing point `v`.
ivec3 posToCell(vec4 v) {
    return ivec3(
        floor(v.x / gridCellSize),
        floor(v.y / gridCellSize),
        floor(v.z / gridCellSize));
}

// Convert a 3D grid cell to a 1D index.
uint flatten(ivec3 cell) {
    uint tx = (cell.x * 78455519);
    uint ty = (cell.y * 41397959);
    uint tz = (cell.z * 27614441);
    return uint(abs(tx ^ ty ^ tz)) % (startIndices.length());
}
```

Listing 6: Helper functions for accessing the grid.

C: Hair Shader

```
#version 460 core

#define PI 3.14159265358979323846264338327950288

struct PBRMaterial {
  vec3 albedo;
  float metalness;
  float roughness;
};

struct Particle {
    vec4 x;
    vec4 v;
    float w;
    int t;
    float d;
    int s;
};
layout(std430, binding = 0) buffer Particles {
    Particle particles[];
};

layout(location = 0) out vec4 FragColour;
layout(location = 1) out vec4 DepthColour;

in ES_OUT {
    flat uint particleID;
    float vWeight;
    vec3 eyeSpacePos;
    vec3 fragPos;
} es;

float wetness = 0.1;
float wetnessSub = 0.9;
```

```
layout(location = 0) uniform mat4 proj;
layout(location = 2) uniform vec3 viewPos;
layout(location = 4) uniform vec4 colour = vec4(1);
layout(location = 5) uniform float particleRadius;
layout(location = 6) uniform float fresnelPower;
layout(location = 7) uniform vec3 headPos;
layout(location = 8) uniform PBRMaterial pbrMaterial;

vec3 CalcDirLight(vec3 lightDir, vec3 normal, vec3 viewDir);
float GGX_D(vec3 normal, vec3 halfwayDir, float roughness);
float GGX_G_Schlick(vec3 normal, vec3 viewDir, float roughness);
float GGX_G(vec3 normal, vec3 viewDir, vec3 lightDir,
        float roughness);
vec3 fresnel(float cos_theta, vec3 F0);

void main() {
    vec4 pixelPos = vec4(es.eyeSpacePos, 1);
    vec4 clipSpacePos = proj * pixelPos;
    float ndc = clipSpacePos.z / clipSpacePos.w;
    gl_FragDepth = ndc *.5 + .5;
    float outDepth = -pixelPos.z;
        DepthColour = vec4(vec3(outDepth/150), 1);

    wetness = particles[es.particleID].d;
    wetness = clamp(wetness, 0.1, 1);
    wetnessSub = 1 - wetness;

        // properties
        vec3 norm = normalize(es.fragPos - headPos);
        vec3 viewDir = normalize(viewPos - es.fragPos);

        // directional lights
        vec3 result = pbrMaterial.albedo * 0.2 +
                CalcDirLight(vec3(-1, -1, 0), norm, viewDir);

        vec3 ev = normalize(es.fragPos - viewPos);
        float ff = abs(dot(ev, norm));
```

```
        result += fresnel(ff, vec3(0.05)) *
                es.vWeight *
                (outDepth/150) *
                0.5;

        FragColour = vec4(result, 1.0);
}



// calculates the color when using a directional light.
vec3 CalcDirLight(vec3 lightDir, vec3 N, vec3 V) {
        vec3 L = normalize(-lightDir);
        vec3 H = normalize(L + V);
        vec3 radiance = pbrMaterial.albedo *
                clamp(wetnessSub, 0, 1);
        float mtl = clamp(pbrMaterial.metalness + 0, 0, 1);
        float rgh = clamp(pbrMaterial.roughness - 0, 0, 1);

        float NoL = max(dot(N, L), 0);
        vec3 F0 = vec3(0.5);
        F0 = mix(F0, pbrMaterial.albedo, mtl);
        float D = GGX_D(N, H, rgh);
        float G = GGX_G(N, V, L, rgh);
        vec3 F = fresnel(max(dot(H, V), 0), F0);

        vec3 BRDF = (D * G * F) / (4.0 * max(dot(N, V), 0.0) *
                NoL + 0.0001);
        vec3 I = normalize(es.fragPos - viewPos);

        vec3 kS = F;
        vec3 kD = (vec3(1) - kS) * 2;
    kD = (vec3(1) - kS) * (1 - pbrMaterial.metalness) * 2;
        return (kD + BRDF * vec3(0.8)) * radiance * NoL;
}

// normal distribution function (Trowbridge-Reitz GGX)
```

```
float GGX_D(vec3 normal, vec3 halfwayDir, float roughness) {
        float a = roughness * roughness;
        float a2 = a*a;
        float HoN = max(dot(normal, halfwayDir), 0);
        float partial_denom = ((HoN * HoN) * (a2 - 1) + 1);
        float denom = PI * partial_denom * partial_denom;
        float D = a2 / denom;
        return D;
}

// geometry function (Schilck-GGX)
float GGX_G_Schlick(vec3 normal,
                                        vec3 viewDir,
                                        float roughness) {
        float NoV = max(dot(normal, viewDir), 0);
        float r = roughness + 1;
        float k = (r * r) / 8;
        return NoV / ((NoV) * (1-k) + k);
}

// geometry function (Smith-GGX)
float GGX_G(vec3 normal,
                        vec3 viewDir,
                        vec3 lightDir,
                        float roughness) {
        float g1 = GGX_G_Schlick(normal, viewDir, roughness);
        float g2 = GGX_G_Schlick(normal, lightDir, roughness);
        return g1 * g2;
}

// fresnel equation (Schlick approximation)
vec3 fresnel(float cos_theta, vec3 F0) {
        return F0 + (1-F0) * pow(clamp(1 - cos_theta, 0, 1),
                fresnelPower * wetnessSub);
}
```

Listing 7: Using the Cook-Torrance GGX BRDF to shade our hair.