Dissertation Type: research

DEPARTMENT OF COMPUTER SCIENCE

Real-Time Hair Rendering and Simulation

Matthew Woodhouse

A dissertation submitted to the University of Bristol in accordance with the requirements of the degree of Master of Engineering in the Faculty of Engineering.

7th May 2017

Declaration

This dissertation is submitted to the University of Bristol in accordance with the requirements of the degree of MEng in the Faculty of Engineering. It has not been submitted for any other degree or diploma of any examining body. Except where specifically acknowledged, it is all the work of the Author.

Matthew Woodhouse, 7th May 2017

Contents

List of Figures

Executive Summary

Abstract

This paper explores the possibility of the rendering and simulation of hair in real-time applications. It is shown that by simulating the dynamics of a subset of hairs, and interpolating those remaining, individual strand simulation can be achieved. Hair to hair interactions are be successfully approximated by using volumetric methods modelled with three-dimensional textures. Rendering can be implemented through the use of look-up tables and simplified scattering models dependent on the available hardware. The application of these ideas is shown to achieve performance orders of magnitude beyond current industry-standard implementations.

Main Contributions

- A hair simulation model which handles the motion of individual strands and interactions between hairs in a realistic manner.
- An approximated shading model for rendering hairs with intuitive controls for artists.
- A novel method for performing the self-shadowing of a hair volume without the need for large amounts of additional computations.
- A naive implementation of the simulation and rendering models, illustrating the most straightforward approach to implementation.
- An optimised implementation showcasing the potential of the proposed models in real-time applications.

Supporting Technologies

- OpenGL 4.3 (core profile) is used as the API for GPU compute and rendering (`www.opengl.org`, visited 5/05/2017).
- GLFW 3 is used for OpenGL context and window creation (`www.glfw.org`, visited 5/05/2017).
- GLEW is used to handle OpenGL extensions (`glew.sourceforge.net`, visited 5/05/2017).
- C++14 (`en.cppreference.com/`, visited 5/05/2017) and GLSL (`www.khronos.org/opengl/wiki/OpenGL_Shading_Language`, visited 5/05/2017) are used for the implementations.
- Blender 3D is used for the creation of scalp meshes from which hair strands are grown (`www.blender.org/`, visited 5/05/2017).

Chapter 1

Contextual Background and Motivation

We live in an era of incredible computer graphics and near photo-realistic virtual worlds. Extraordinary material and lighting quality is achieved in real-time through techniques such as physically based rendering [32] and even fully dynamic global illumination [20]. Despite advances in so many other areas, one aspect of real-time rendering that has not progressed to the same degree is that of rendering and simulating perceptively realistic hair. That is not to say that there have been no developments, but rather that in most cases these newer techniques have not been adopted and used in applications.

An example of this can be seen in the games *Metro: 2033 Redux* and *Metro: Last Light Redux* [12]. Developer 4A Games did an excellent job of telling a compelling and exciting story which was greatly enhanced by the outstanding graphical fidelity provided by the 4A Engine. The only area which breaks the immersion is the hair visible on many of the human characters. It simply does not look real, and due to how impressive the rest of the game looks, the contrast in quality is quite distracting. There is also a noticeable lack of characters with longer hair. This is in stark contrast to *Tomb Raider* [8], which made realistic hair simulation a major feature of their game. Unfortunately, the simulation was so resource intensive that many users opted to turn it off. Despite being different problems, both are firmly centred aroud the representation of hair. So why is this such a challenge to implement?

The most obvious answer to this question is that there are simply many more important aspects of a rendered scene which fame time could be spent on. The current techniques available for the simulation and rendering of hair are too expensive to perform within the time available for a single frame in an interactive application. Other components of the scene provide a greater visual impact and as such hair is less of a priority.

As such, the aim of this report is to provide a method which allows for the real-time rendering and simulation of hair in applications without blowing the frame budget. This imposes certain requirements, which are to be discussed below.

The first point is, as would be expected, that the simulation must be fast. In most applications, physics simulation is performed on the CPU. As such, prior to rendering the updated position information for all physics objects must be sent to the graphics card. With updates occurring every frame, it's easy to see how this would quickly become a bottleneck. In order to avoid this data transfer overhead, we will instead perform all simulation in shaders on the GPU. This means the position information is already stored in GPU memory, and hence there is no longer a data transfer overhead. This therefore becomes a requirement of the project - all simulation must take place on the GPU to avoid unnecessary overhead.

Working on a GPU introduces its own problems, the largest of which is the high level of parallelism. It may seem strange to state this as a disadvantage, and it is of course beneficial to correctly written programs. The difficulty is writing software that is efficient for the GPU to execute. GPUs execute instructions in "lockstep" - that is to say, the many cores all execute the same instruction at the same time [29]. While this may not seem problematic at first, the real issue arises when branching is performed. How can all cores perform the same instruction if only a portion follow the same code path? In many cases, the result is that all cores will perform the same instructions, but for those which did not take the executing branch the result will be discarded. Alternative methods have been used in modern GPU architectures, providing better results with respect to dynamic branching [29]. In addition to this, reading from memory can also be a huge overhead in GPU applications depending on the memory layout. As such, it is essential that the method proposed is efficient and well-written for GPU systems.

The final, and rather obvious, requirement is that the rendered simulation looks believable. When working in computer graphics, it is commonly required to make approximations, and as such it is unlikely that the model will be a fully accurate depiction of hair motion. However, as long as the approximation is sufficiently close to realism, the overall impression the hair will make is far beyond that currently used in real-time applications.

As such, the following list provides a requirements specification to which the implementation should conform.

- **GPU Based**
  All computation will take place on the GPU to remove the overhead of data transfer with every update and draw. This will be achieved by storing all simulation state in GPU memory, with simulation calculations being performed in compute shaders.

- **Real-Time Performance**
  The final simulation must run in real-time, even on lower-end hardware. For the purposes of this document, real-time is defined as spending less than 30ms per frame for the combined cost of the simulation and rendering of the hair volume.

- **Realistic Visuals**
  Once rendered, a frame depicting the hair volume must be of a realistic appearance. While this is a subjective requirement, an image which is clearly representative of hair is considered to be acceptable.

- **Realistic Motion**
  The simulated hair dynamics must provide a close approximation of the real motion of a hair volume. Results should be of comparable quality to the currently available industry-standard implementations.

Chapter 2

Technical Background

Hair is a defining characteristic in the appearance of many creatures in the natural world, and as such many virtual characters are depicted as being at least partially hairy. As a result of this, a large amount of research has been put into how hair should be simulated and rendered for the best visual quality and most convincing results. Potentially the largest area concerned with realistic virtual hair is the motion picture industry, where rendered characters need to look natural when composited into a scene with other, real characters. With that being said, a fair amount of research has also been put into how to model the hair efficiently, such that it can be used in interactive applications.

It's important at this point to define the difference between *interactive* and *real-time*. For an application to be considered interactive, it must be able to provide some level of interactivity, but there is not a strict requirement on the latency in the system. An example of an interactive system would be 3D modelling software - the user is making changes which appear as they are made, but the response time between making a change and it appearing is not of prime concern (within reason). In a real-time application, however, keeping the latency low is essential to the user experience. The classic example of a real-time application in computer graphics is a video game - if the delay between a user performing an action and it taking place is too high, the game becomes unplayable [2].

The remainder of this chapter is split into two main sections - rendering and simulation. Much of the research presented is concerned with one or the other, with a few considering both, making this a logical division to impose. Considering rendering and simulation as separate problems is in fact beneficial to the result implementation, leading to two individual systems which could be interchanged with others should the situation arise.

2.1 Simulation

Simulation refers to the movement and dynamic deformation of a character's hair and is an essential aspect of rendering a believable scene. Without it, hair will remain fixed in the same position, causing character motion to look unnatural and lacking in momentum.

The simulation of hair can be further divided into two additional sections - the dynamics of an individual strand of hair, and the dynamics of the hair volume as a whole. While it is possible to simulate individual hairs using more general purpose numerical integration techniques such as Euler or Runge Kutta integration [11], a more specialised approach would be more likely to provide better results. For the hair volume, brute force is not an option due to the sheer computational complexity involved. As such, the remainder of this section is intended to give the reader an understanding of techniques that are used for the simulation of hair, with a particular focus on those which are built upon in this report.

2.1.1 Strand Representations

Before simulation can begin, it is necessary to determine how the hair strands will be represented. The choice of strand representation directly influences the choice of simulation method, as they are often tailored towards a certain implementation. We examine a subset of methods from the comprehensive list provided in [41].

Mass-Spring

In a mass-spring system, strands of hair are formed of a set of connected particles. The connections between particles are modelled as stiff springs, with bending angle constraints achieved through the use of angular springs located at each joint. An example of a mass-spring system can be seen in figure 2.1.

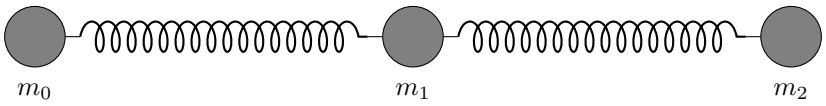


Figure 2.1: A simple mass-spring system with three particles.

Particles in a mass-spring system have three degrees of freedom composed of a single translation and two rotations. This is a popular model for hair and cloth modelling due to its simplicity, allowing for simple and efficient implementations with little overhead. However, the model does not account for the strand's resistance to stretching without the use of strong forces in the spring, which can in turn lead to an unstable simulation [41]. A detailed description of this method with respect to hair can be found in [35].

Strips

Neighbouring hair strands can be grouped together into flat patches referred to as strips. Grouping together multiple strands into a single cluster is beneficial to performance as fewer vertices will need to be simulated. As multiple strands are being modelled by a single primitive, a custom dynamics model is required. An example model is that proposed in [23], which is based on the projective equations covered in section 2.1.1.

In [23], the strips are modelled as NURBS [1] surfaces, with their control points being manipulated during simulation. Springs are also used to prevent different strips from intersecting, and also to prevent excessive stretching or compression. This method is good for performance and simplicity, but limiting in the types of hair styles which can be reproduced and lack of freedom of motion.

One-Dimensional Projective Equations

In this model, proposed in [1], hair strands are represented as a sequence of inextensible sticks, parametrised in polar coordinates as shown in figure 2.2. The stick shown in figure 2.2 would be defined as the direction $\mathbf{s}_i = \mathbf{p}_i - \mathbf{p}_{i-1}$, allowing for the parametrisation by angles θ and ϕ. During simulation, external forces are projected onto the planes defined by both angles. The fundamental principles of dynamics are then applied to both θ and ϕ leading to two differential equations which must be solved at each time step [1].

While this method may be attractive due to its simplicity, problems arise as torsional hair stiffness cannot be accounted for and as such full three-dimensional motion is not simulated [41].

Rigid Multi-Body Serial Chain

In a rigid multi-body serial chain, hair strands are modelled as shown in figure 2.3. The motion is computed using forward dynamics, described from a robotics point of view in [10]. Reduced spatial coordinates are used to prevent anything other than bending or twisting caused by the stretching of the degrees of freedom [41]. Forces are applied to each link, accounting for bending and torsional rigidity. By using the Articulated-Body Method [10] for forward dynamics, simulation can be performed with a linear time complexity.

2.1.2 Verlet Integration

Verlet integration is a numerical integration method for solving the equations of motion proposed by Loup Verlet in [40]. The method works by storing the position at the current time and that of the previous

[1] Non-Uniform Rational B-Spline

time step, which can then be used to generate the position at the end of the next time step. As an equation,

$$\mathbf{p}_i(t + \Delta t) = \mathbf{p}_i(t) + k(\mathbf{p}_i(t) - \mathbf{p}_i(t - \Delta t)) + \Delta t^2 \mathbf{f}_i(t)$$

where $\mathbf{p}_i(t)$ is the position of the particle $\mathbf{p}_i$ at time t, Δt is the size of the time step, $\mathbf{f}_i(t)$ is the force acting on the particle $\mathbf{p}_i$ at time t, and k is a damping coefficient. Constraints between particles can also be solved with only the positions of the particles required.

$$\begin{aligned}
\mathbf{dv} &= \|\mathbf{p}_i(t) - \mathbf{p}_j(t)\| \\
d &= \frac{r_{ij} - \|\mathbf{dv}\|}{\|\mathbf{dv}\|} \\
\mathbf{p}_i(t + \Delta t) &= \mathbf{p}_i(t) + \frac{d}{m_i}\mathbf{dv} \\
\mathbf{p}_j(t + \Delta t) &= \mathbf{p}_j(t) - \frac{d}{m_j}\mathbf{dv}
\end{aligned}$$

In addition to those previously defined, r_{ij} is the resting length of the constraint between particles $\mathbf{p}_i$ and $\mathbf{p}_j$, and m_i is the mass of particle $\mathbf{p}_i$. The intuition here is that each particle is being moved in opposite directions to enforce the constraint such that the distance a particle moves is proportional to its mass.

While Verlet integration is not designed with hair in mind, with the update and constrain methods outlined above it is clearly a suitable method for simulation with the mass-spring strand model. It is often used for particle systems in video games due to its simplicity [30].

2.1.3 Position Based Dynamics

Position based dynamics is another simulation method that works with positions instead of forces, and was first introduced in [27]. A rough outline of the algorithm is shown below.

$$\begin{aligned}
\mathbf{x}_i &= \mathbf{p}_i(t) + \Delta t \mathbf{v}_i(t) + \Delta t^2 \mathbf{f}_i(t) \\
\mathbf{x}_i &= \text{SolveConstraints}(\mathbf{x}_i) \\
\mathbf{v}_i(t + \Delta t) &= \frac{\mathbf{x}_i - \mathbf{p}_i(t)}{\Delta t} \\
\mathbf{p}_i(t + \Delta t) &= \mathbf{x}_i
\end{aligned}$$

In the above example, $\mathbf{p}_i(t)$ is the position of particle $\mathbf{p}_i$ at time t, $\mathbf{v}_i(t)$ is that particle's velocity at t, Δt is the size of the time step, $\mathbf{f}_i(t)$ is the force acting on the particle, and $\mathbf{x}_i$ is a temporary variable.

Solving a constraint is a non-linear operation, and is generally achieved by repeatedly iterating through all constraints and projecting the particles involved into valid locations, but only with consideration of the current constraint [27]. In the aforementioned paper, constraint projection is illustrated through solving a simple distance constraint between two particles, as seen in figure 2.4, although other constraint types such as bending resistance are also covered in detail.

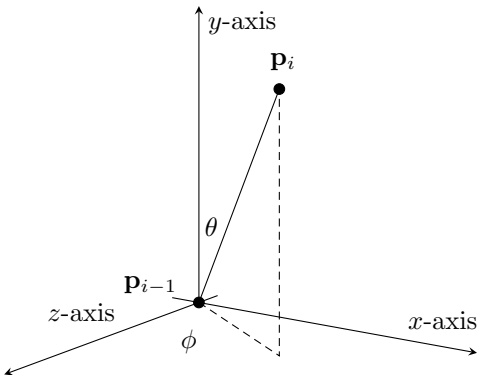


Figure 2.2: Example of a single segment in a hair strand using one-dimensional projective equations. The stick $\mathbf{s}_i$ is drawn between nodes $\mathbf{p}_i$ and $\mathbf{p}_{i-1}$, and is parametrised by zenith angle θ and azimuth angle ϕ.

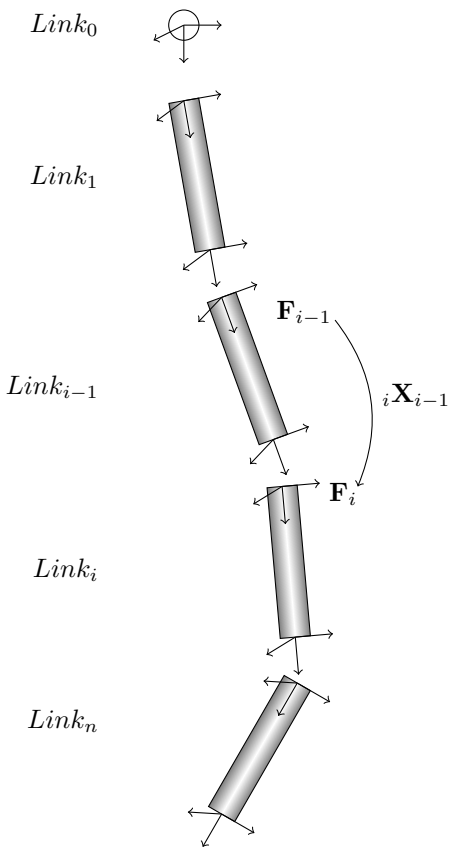


Figure 2.3: An example of a rigid multi-body serial chain representation of a hair strand.

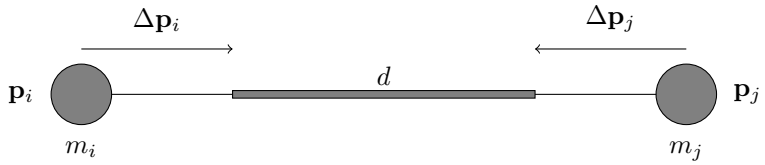


Figure 2.4: An example distance constraint between two particles $\mathbf{p}_i$ and $\mathbf{p}_j$. The constraint would be defined as $C(\mathbf{p}_i, \mathbf{p}_j) = \|\mathbf{p}_i - \mathbf{p}_j\| - d$, leading to corrections $\Delta\mathbf{p}_i$ and $\Delta\mathbf{p}_j$.

The correction vectors for solving the constraint presented in figure 2.4 are given by equation (2.1). It's noticeable how similar the method for solving constraints is to that used in Verlet integration [40], although there are subtle differences. The most important point is that again the particles are being weighted relative to their respective masses.

$$\begin{aligned} w_i &= \frac{1}{m_i} \\ w_j &= \frac{1}{m_j} \\ \mathbf{dv} &= (\|\mathbf{p}_i - \mathbf{p}_j\| - d)\frac{\mathbf{p}_i - \mathbf{p}_j}{\|\mathbf{p}_i - \mathbf{p}_j\|} \\ \Delta\mathbf{p}_i &= -\frac{w_i}{w_i + w_j}\mathbf{dv} \\ \Delta\mathbf{p}_j &= +\frac{w_j}{w_i + w_j}\mathbf{dv} \end{aligned} \tag{2.1}$$

Despite its seeming similarity, position based dynamics does have multiple advantages over other

simulation methods such as Verlet [27]. Results tend to be more stable than other methods, and generic constraints can be easily added and solved in a position based system. In addition, position based dynamics can also be used to simulate soft-body physics, by treating each vertex of a mesh as a particle [27]. This makes it a powerful tool for modelling realistic collisions involving deformable objects such as cloth.

Position based dynamics would work best with the mass-spring hair system, as it is designed to work with sets of constrained particles. It is also similar to Verlet integration in that the algorithm can easily be split into separate parallel threads for improved performance.

2.1.4 NVIDIA

The NVIDIA Corporation has put a great deal of work into the real-time rendering of hair, with the majority of their contributions related to simulation.

Nalu Demo

The first major progression was the release of the *Nalu* demo [5], the development process of which is discussed in [28].

Nalu uses a simplified mass-spring strand model, in which the only constraint between the particles is that of distance. A scalp model is generated, from which hair strands are procedurally grown along the normals as seen in figure 2.5. In the mass-spring model used, each strand is composed of seven particles which are simulated using Verlet integration with several iterations of constraint solving [28]. It is also noted in the article that particles are not evenly spaced along the strand, but instead decrease in density the further from the root. This allows for longer hairs without requiring a linear increase in the number of particles.

In addition to the constraints between particles in the same strand of hair, collisions with the character are also accounted for. This is achieved efficiently by using a simplified collision geometry composed of multiple spheres [28]. For better collision detection, the collisions between particles and the character were modelled with increasingly large spheres instead of points. This was required as some of the hair particles at the ends of strands were reaching beyond the collision mesh [28].

Once simulation is completed, the hair is tessellated to increase the number of vertices along a strand, leading to a smoother visual once rendered. In *Nalu*, the number of particles was increased from seven to thirty-six [28]. To ensure the resulting strand was smooth, the hair was tessellated as a Bézier curve [33], using the particles of the control hair as anchors and calculating the tangents at runtime.

To allow the simulation to run at real-time speeds, the number of hair strands being simulated is relatively sparse and leads to an underwhelming result. To accommodate this, interpolation was used between control hairs to give a fuller and much more impressive head of hair. This interpolation was performed using barycentric coordinates across the scalp mesh's triangles [28].

With all simulation complete, the hair was rendered using the Marschner model and opacity shadow maps, covered in section 2.2.2 and section 2.2.5 respectively. The final result can be seen in figure 2.6.

The demo does not perform any modelling of the hair to hair interactions that occur between strands.

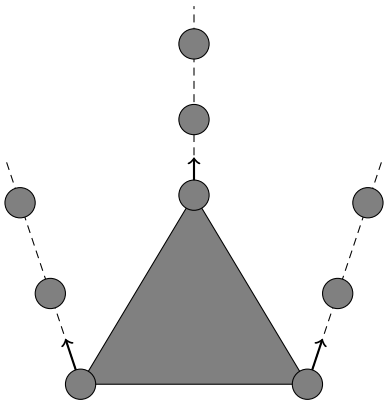


Figure 2.5: Illustration of how hair strands grow along vertex normals in the *Nalu* demo.

Figure 2.6: The final rendered hair as seen in the *Nalu* demo. Image from [5].

Cloth Simulation on the GPU

In [44], a method is proposed to perform cloth simulation on the GPU entirely through the use of shaders. While not directly related to hair, it is still of relevance due to all simulation being performed on the GPU, removing the overhead of needing to send updated vertex data to the graphics card for each frame.

The simulated cloths are modelled as particles connected by springs, with each spring modelled as a distance constraint. The particles are updated using Verlet integration, with the constraints being solved through relaxation, described previously in section 2.1.2, which is repeated for multiple iterations. Collisions of cloth with another object is enforced by moving particles that enter other geometry back to the geometry surface.

The positions of the particles are stored in floating-point textures [44], allowing points to be updated through a series of draw calls. Two textures are required for the position update, one for the state before the update, and one after [44]. To invoke the simulation, a single quad is drawn covering the whole render target, which invokes the pixel/fragment shader for every element stored in the texture. For more information about using rendering pipelines such as OpenGL for general purpose computation on the GPU (GPGPU), see [14].

For handling forces, Verlet integration requires three textures, one each for the current, old, and new states [44]. The distance constraints are solved by performing odd and evenly indexed constraints in different draw calls. This allows for maximum parallelism on the GPU to be utilised, and still leads to a relatively stable state after iterating [44].

On the GeForce 6800 Ultra, a performance of 400 frames per second was achieved, truly illustraing how effective the use of GPUs are for performing simulation.

Fast Simulation of Inextensible Hair and Fur

Up to this point the simulations that were being performed modelled only a subset of the hairs, and then used other methods such as interpolation to generate the rest. However, in [26], a method is proposed that allowed for the simulation of 47k hairs at interactive rates.

Once again, hairs are modelled as a chain of particles connected by inextensible links. The particles are updated using position based dynamics [27]. Instead of using the standard multiple iteration approach to solving the distance constraints, in [26] a modified version of the *Follow the Leader* algorithm [4] is used, which the authors name *Dynamic FTL*.

In the classic, or *Static*, FTL algorithm [4], distance constraints are solved by iterating along the chain of particles from the root. At each step, the current particle is moved such that it is in the closest position to where it was prior to solving the constraint, while still enforcing the distance constraint. This

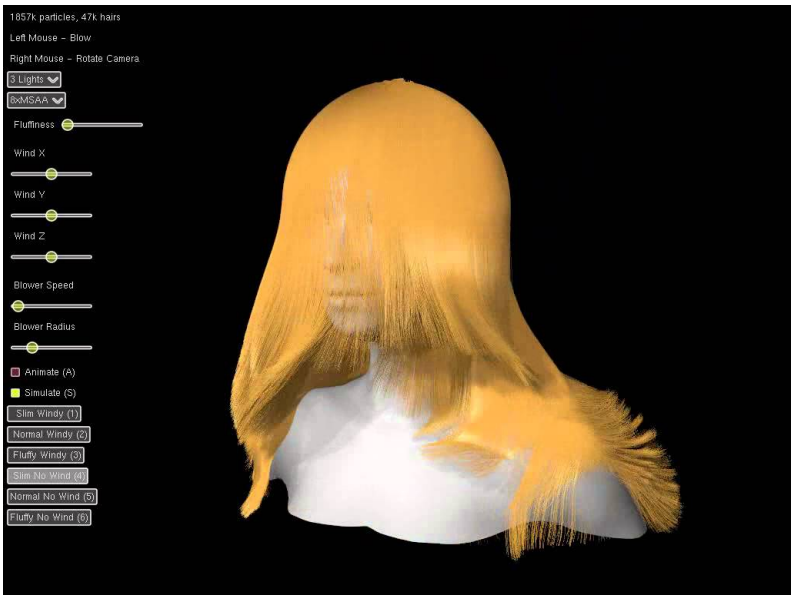


Figure 2.7: Final result of rendered hair using the Follow the Leader algorithm for the solving of constraints. Of particular note is the number of hairs and how smooth they appear even without interpolation due to the high number of particles and strands being simulated. Image from the results video referenced in [26].

is illustrated below.

$$\begin{aligned} \mathbf{dv} &= \mathbf{p}_i(t) - \mathbf{p}_{i-1}(t + \Delta t) \\ \mathbf{p}_i(t + \Delta t) &= \mathbf{p}_{i-1}(t + \Delta t) + l\frac{\mathbf{dv}}{\|\mathbf{dv}\|} \\ \mathbf{d}_i &= \mathbf{p}_i(t + \Delta t) - \mathbf{p}_i(t) \end{aligned}$$

In this example, the particle chain is formed of $\mathbf{p}_0$, $\mathbf{p}_1$, ..., $\mathbf{p}_{n-1}$ from root to tip, and l is the length of the distance constraint between particles. The vector $\mathbf{d}_i$ is said to be the *correction vector* for $\mathbf{p}_i$. With static FTL, there is an uneven mass distribution [26], leading to unnatural projections. In the original paper [4], the knots being modelled remain mostly stationary except when manipulated by the user, but this is not the case with strands of hair.

To account for this uneven mass distribution, the authors of [26] propose the following solution: given that the correction vectors $\mathbf{d}_i$ have already been computed for the constraints between the particles of a strand, the velocity can be corrected such that the new velocity update step of the position based dynamics algorithm becomes

$$\mathbf{v}_i(t + \Delta t) = \frac{\mathbf{p}_i(t + \Delta t) - \mathbf{p}_i(t)}{\Delta t} + s_{damping}\frac{-\mathbf{d}_{i+1}}{\Delta t}$$

where $s_{damping} \in [0, 1]$ is a numerical damping factor. This addition hides the uneven mass distribution [26], producing much improved motion.

The reason behind using FTL as the projection step is that it only requires a single iteration to solve the constraint, allowing for a greater overall performance. In addition, the implementation also accounts for hair to hair interactions using the method originally proposed in [31], discussed in section 2.1.6.

The final simulation produced in [26] was able to achieve interactive rates when simulating up to 47k hairs, or 1.9m, particles on a NVIDIA GeForce GTX 480 using CUDA. An image of this in action can be seen in figure 2.7.

HairWorks

The previous research performed at NVIDIA seems to have led to the creation of HairWorks, a complete product used for the simulation and rendering of hair. A selection of key features include full control over hair shape, style, and colour; self-shadowing and shadow casting; continuous level of detail, and real-time

Figure 2.8: Hair rendered using NVIDIA HairWorks. Image sourced from the gallery provided by [6].

editing [6]. An example of HairWorks in action is the game *The Witcher 3: Wild Hunt* [34]. A demo can also be seen in figure 2.8. Source code is available to NVIDIA Developer account holders online at `www.github.com/NVIDIAGameWorks/HairWorks`.

One major criticism of the simulation is that it is extremely resource intensive, resulting in many users disabling the option entirely. It doesn't matter how impressive a feature is if a large portion of users end up having to turn it off to be able to play the game.

2.1.5 AMD TressFX

TressFX [18] is to AMD what HairWorks is to NVIDIA, providing developers with a set of features such as continuous level of detail, self-shadowing, robust scalability across a range of GPUs, and correct physical modelling. An image of the method in action can be seen in figure 2.9.

The presentation [3] provides a breakdown of how TressFX works under the hood. Firstly, rendering is considered. Two hair lighting models are used, Kajiya-Kay [19] and approximated Marschner [25], discussed in detail in section 2.2.1 and section 2.2.2 respectively. Hair strands are anti-aliased manually through the computation of pixel coverage on their edges. Self-shadowing is achieved through a simplified version of Deep Shadow Maps [24], as explained in section 2.2.4. To give the hair volume a more realistic appearance, transparency is also used through Order Independent Transparency achieved by using Per-Pixel Linked Lists [3]. As of TressFX 2.0, lighting and shadowing is delayed until the later passes to improve performance.

Simulation is performed on the GPU through DirectCompute [7] in a series of steps. Once forces have been applied and the particles integrated, a global shape constraint is used to direct particles towards their goal positions to maintain the desired shape of the hair, at the cost of losing some simulation detail [3]. Local shape constraints aim to achieve a similar effect with respect to local frames. Constraints between

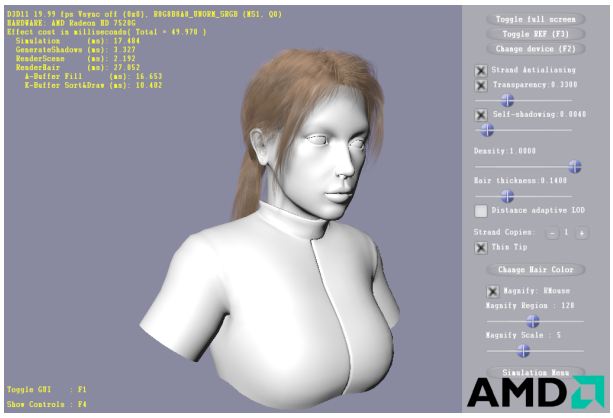


Figure 2.9: Hair rendered using the TressFX demo. Image taken by the author directly while running the demo.

particles are solved with position based dynamics.

As with HairWorks, TressFX also has something of a reputation for consuming more resources that it's worth. With that being said, it has been used in games such as *Tomb Raider* [8] and *Deus Ex: Mankind Divided* [38].

2.1.6 Pixar

Due to Pixar's extensive history of making animation films, it's unsurprising that they have put a great deal of resources into researching the best techniques for hair rendering. Both the research papers discussed in this section consider both rendering and simulation.

Figure 2.10: Hair rendered using the volumetric method proposed in [31]. Image from [31].

Volumetric Methods for Simulation and Rendering of Hair

The first paper to consider is [31], which presents a volumetric approach to rendering hair and simulating the complex interactions between strands. The initial simulation is performed through the use of a standard keyhair model, interpolating between them to generate a fuller head of hair as was previously discussed in [28]. However, this does not account for any hair to hair interactions, leading to potentially unrealistic motion and other visual artifacts such as self-intersection of hair geometry.

It is of course not feasible to calculate the interactions between thousands of hairs in a reasonable time frame, and as such the values are approximated through the use of a three-dimensional voxel grid [31]. The first step is to find the *density* at each voxel, which is the sum of the contributions of hair particles in the current voxel and its neighbours, as shown in equation (2.2), where $\mathbf{p}_i(t)_x$ is the x component of the particle $\mathbf{p}_i$'s position at time t. It's important to emphasise that only particles that are in the current voxel or its neighbours are iterated over by i.

$$D_{xyz} = \sum_i (1 - |\mathbf{p}_i(t)_x - x|)(1 - |\mathbf{p}_i(t)_y - y|)(1 - |\mathbf{p}_i(t)_z - z|) \tag{2.2}$$

$$\mathbf{V}_{xyz} = \frac{\sum_i (1 - |\mathbf{p}_i(t)_x - x|)(1 - |\mathbf{p}_i(t)_y - y|)(1 - |\mathbf{p}_i(t)_z - z|)\mathbf{v}_i(t)}{D_{xyz}} \tag{2.3}$$

$$\mathbf{v}_i(t) = (1 - s_{friction})\mathbf{v}_i(t) + s_{friction}\mathbf{V}_{xyz} \tag{2.4}$$

Once the density has been computed, the average velocity at each voxel is calculated using equation (2.3). These averages are further smoothed using a filter kernel designed to conserve the total energy. This velocity is then used to adjust the velocity of a particle by finding which voxel it lies in and calculating a weighted average between the voxel's and the particle's current velocity [26], shown in equation (2.4). While not an entirely accurate model, the motion of hair when using this approximation is greatly enhanced and more believable. Hair directing can also be achieved by using the gradient between the current voxel grid and that of a target. This is useful as it allows for artists to maintain some level of control over the simulation which is preferred over a purely procedural approach. A full explanation can be found in [31].

The density volume can also be used to enhance the shading of the hair. The authors of [31] noted that using Kajiya-Kay illumination, the approximated normal often leads to incorrect shading due to the hairs being modelled as infinitely thin cylinders. To overcome this issue, an isosurface of the density is generated using a signed distance field [31]. The normals are then obtained for the lighting calculations by finding the gradient of the density grid. [26] also use a normalised density gradient to account for hair to hair repulsion between strands, allowing for curly hair to be approximated with a higher repulsion during simulation and twisted geometry at the rendering stage. The final rendered hair can be seen in figure 2.10.

Figure 2.11: Resulting curly hair using the method proposed by Pixar, here shown in the film *Brave*. Image from [17].

Artistic Simulation of Curly Hair

The main focus of [17] is to find how to accurately simulate curly hair, motivated by the Pixar film *Brave*, the results of which can be seen in figure 2.11. The hair model used for a single strand is a combination of a mass-spring system and one of infinitesimally thin elastic rods. The motivations behind this choice of model were to support artistic controls over hair stretch, hairs that maintain the curl's shape while allowing for flexing, and curls that do not lose their shape during acceleration [17].

In addition to the properties of a classical mass-spring system, two additional springs were added to control the bend along the curl and longitudinal stretch of curls [17]. It's worth noting that the focus of this paper was not to create physically accurate hair, but rather a plausible model that allows for greater artistic control over the end result.

In addition to individual strand dynamics, hair to hair interactions were also modelled. Unlike the method proposed in [31], hair to hair contacts are calculated on a point-to-point basis, as opposed to approximating them using a voxel grid [17]. However, to prevent an overwhelming amount of computations being required, two types of pruning are used to reduce the load and make the problem more easily parallelisable. Particles of hair strands are each encapsulated by a sphere, which acts as a bounding volume through which collision points are evaluated. As many of these will overlap, those which are beyond a certain threshold are pruned away to reduce the load [17]. In addition, not all hairs interact with each other as dictated by a predefined graph. By defining a maximum number of hair contacts required for an interaction to be considered valid, additional edges of the graph can be pruned away without significant loss in visual quality [17].

The entire simulation process is designed to scale well on a cluster or group of processors. As the simulation is so finely-grained, computation times are well beyond the acceptable range for real-time applications.

2.2 Rendering

Once the hairs have been fully simulated, the next task is to render them such that they are visually appealing. The rendering aspect of hair simulation can be broadly considered as the combination of two shading models - global, and local. A global shading model is concerned with the volume as a whole, including effects between strands such as self-shadowing, while a local model looks only at a single strand.

2.2.1 Kajiya-Kay Shading

One of the earliest methods of shading which is still used today is that proposed in [19]. While the main focus of the paper was on the rendering of fur through the use of three-dimensional textures, the contribution still used to this day is how the diffuse and specular components of the local shading model are calculated. A comparison between this model and that described in section 2.2.2 [25] can be seen in figure 2.13.

Kajiya and Kay observed that if the tangent of a hair is aimed directly at a light source, the hair appears to be dark. As such, it was concluded that the diffuse lighting component is proportional to the sine between the tangent vector t and light vector l [19], as shown in equation (2.5). K_d is a constant representing absoption.

$$\begin{aligned}\Psi_{diffuse} &= K_d \frac{1-(t\cdot l)^2}{\sqrt{1-(t\cdot l)^2}} \\ &= K_d \sin(t,l)\end{aligned} \tag{2.5}$$

For specular highlights, it was observed that light is reflected by the hair at a mirror angle along the tangent, and as the hairs are modelled as cylinders, reflected light should always be independent of the azimuthal component of the eye vector, e [19]. The highlight's intensity can be calculated using the additional propertiy p, which represents the Phong exponent dictating the sharpness of the highlight, and K_s, the specular reflection coefficient.

$$\Psi_{specular} = K_s(t\cdot lt\cdot e + \sin(t,l)\sin(t,e))^p$$

2.2.2 Marschner Scattering

While the method proposed in [19] was a step in the right direction, there was only one specular highlight, whereas in real hair there are two that are clearly visible [25]. This was observed by Marschner et al. in [25], where measurements were performed on hair strands of different types to gain a better understanding and hopefully a better local shading model.

In this model, a ray of light that makes contact with a strand of hair is divided into three components, R, TT, and TRT. The R component is the main specular reflection (light reflected from the surface of the strand), as was modelled in [19]. TT represents the light that refracts through the hair and passes out the other side, and TRT represents light that was initially refracted, but then reflected off the inside surface of the hair strand and back out again. This is illustrated in figure 2.12.

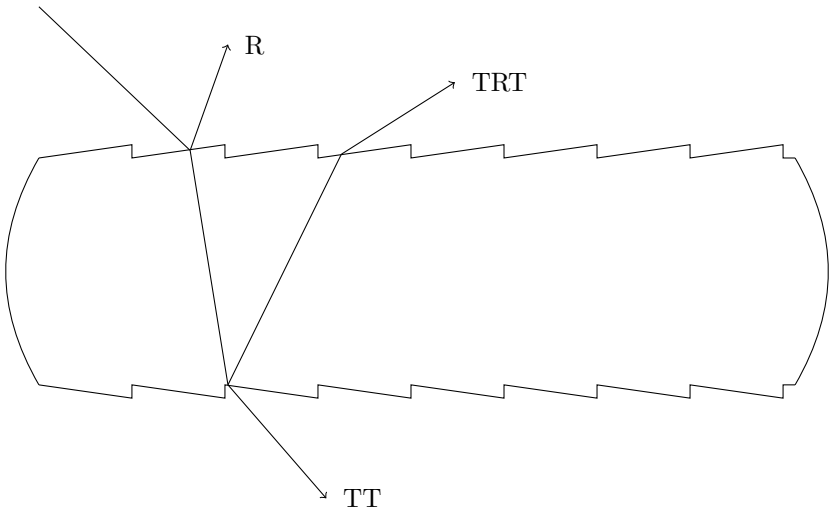


Figure 2.12: A ray of light passing through a strand of hair, illustrating the three reflection paths.

Through performing a series of measurements of scattering in the incidence plane for strands of hair in a variety of different types and colours, the authors were able to create a shading model supporting the three visible highlights previously listed. Although we do not go into detail on the measuring process here as it is beyond the scope of this report, the full description and results can be found in [25].

In the proposed model, the following angles are defined. ϕ_i and ϕ_r represent the azimuths around the hair, while θ_i and θ_r are the inclinations relative to the normal plane of the strand. Additional derived angles are also defined, with $\theta_d = \frac{1}{2}(\theta_r - \theta_i)$, $\phi = \phi_r - \phi_i$, $\theta_h = \frac{1}{2}(\theta_i + \theta_r)$, and $\phi_h = \frac{1}{2}(\phi_i + \phi_r)$. In addition, the vectors ω_i and ω_r represent the direction of illumination and direction of light scattering respectively. Both directions are defined from the point of view of the strand. As such, the bidirectional

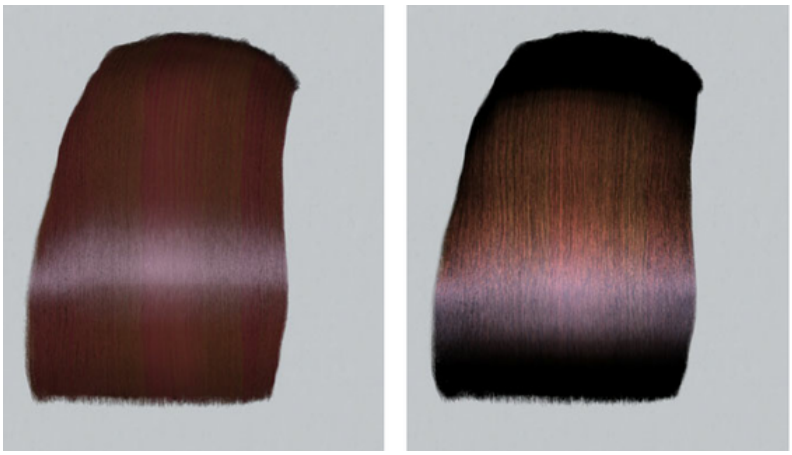


Figure 2.13: A comparison between the shading models proposed in [19] and [25], with Kajiya-Kay on the left and Marschner on the right. Image from [25].

scattering function for a strand of hair can be defined as

$$\begin{aligned} S(\phi_i, \theta_i; \phi_r, \theta_r) = \sec^2 \theta_d (&M_R(\theta_h) N_R(\eta'(\eta, \theta_d); \phi) + \\ &M_{TT}(\theta_h) N_{TT}(\eta'(\eta, \theta_d); \phi) + \\ &M_{TRT}(\theta_h) N_{TRT}(\eta'(\eta, \theta_d); \phi)) \end{aligned} \tag{2.6}$$

where η is the index of refraction, and M_p and N_p are the separate longitudinal and azimuthal scattering terms for $p \in \{R, TT, TRT\}$ respectively. A full derivation of this scattering function can be found in [25].

In order to obtain a more practical shading model for use in rendering, approximations are used.The following variables are defined, with typical values provided in [25].

Parameter	Purpose
Fiber Properties	
η	Index of Refraction
σ_a	Absorption Coefficient (RGB)
Surface Properties	
α_R	R-Lobe Longitudinal Shift
α_{TT}	TT-Lobe Longitudinal Shift
α_{TRT}	TRT-Lobe Longitudinal Shift
β_R	R-Lobe Longitudinal Width
β_{TT}	TT-Lobe Longitudinal Width
β_{TRT}	TRT-Lobe Longitudinal Width

The longitudinal scattering function M is approximated by using a unit-integral, zero-mean lobe function $g(\beta, x)$ [25] where β is the width. In the implementation described by Marschner et al. a normalised Gaussian function with standard deviation β was used. As such, M can be defined for each term as follows.

$$\begin{aligned} M_R(\theta_h) &= g(\beta_R; \theta_h - \alpha_R) \\ M_{TT}(\theta_h) &= g(\beta_{TT}; \theta_h - \alpha_{TT}) \\ M_{TRT}(\theta_h) &= g(\beta_{TRT}; \theta_h - \alpha_{TRT}) \end{aligned}$$

The definition of N is more complicated, requiring additional functions to be defined. If we have a function $A(p, h)$ describing the attenuation factor, N can then be defined as

$$N(\phi) = \sum_p N_p(p, \phi)$$

$$N_p(p, \phi) = \sum_r A(p, h(p, r, \phi)) \left| 2 \frac{d\phi}{dh}(p, h(p, r, \phi)) \right|^{-1}$$

where summing over r corresponds to the summation of all possible roots. This leads to the final definitions of the N scattering terms,

$$N_R(\phi) = N_p(0, \phi)$$
$$N_{TT}(\phi) = N_p(1, \phi)$$
$$N_{TRT}(\phi) = N_p(2, \phi)$$

It's important to note that the definition given here of $N_{TRT}(\phi)$ can cause visual artifacts of infinite intensity that do not match the realistic properties of hair. To avoid this, a more complex definition of the term is given in [25] which replaces these caustics with a smooth lobe. The final specular component for the lighting of the hair strand can be obtained using equation (2.6). The final results of the shading model can be seen in figure 2.13.

This type of local shading model was used in the *Nalu* demo by NVIDIA [28]. In order to achieve real-time performance, the values were precomputed and stored in a lookup texture indexed by the sines and cosines of θ and ϕ [28]. As each of the M_p terms are a single value, these can be packed into a single three channel texture.N_r is a single channel, but N_{TT} and N_{TRT} are each three values. To avoid needing an additional texture, the assumption is made that $N_{TT} = N_{TRT}$, allowing N_R and N_{TT} to be packed into a single four channel texture. As such, all local shading information can be obtained through a two texture reads in the application's vertex/fragment shader.

2.2.3 Disney

While the results of [25] are visually impressive, as it is based on the physical properties of hair changing values can often lead to unexpected results. An example of this is the absorption factor, σ_a, where a value of $(0.03, 0.07, 0.15)$ is blonde but $(0.3, 0.6, 1.2)$ is brown [45]. A solution to this was proposed in [36], which provides an artist-friendly approach through the further approximation of the model proposed by Marschner et al. Hair rendered using this model can be seen in figure 2.14.

Figure 2.14: Hair rendered using the artist friendly shading implementation in [36]. Image from [36].

The process used in [36] is simple in theory but difficult to implement. First, an examination of the physically based function f_s (the function $S(\phi_i, \theta_i; \phi_r, \theta_r)$ in [25]) is performed to extract the behaviour over the domain of the physically based components defining the hair strand's properties. The function f_s is then decomposed into separate and meaningful sub-functions f_{si}, each of which performs a single task. Artist friendly controls are defined for each f_{si}, exposing properties such as colour, intensity, and shape, allowing for control over visuals in an intuitive manner. Pseudo scattering functions f'_{si} are defined to approximate the qualitative behaviour of the functions f_{si} for the artist friendly controls [36]. These pseudo functions are then recombined to give f'_s which is now an artist friendly approximation of the input function f_s.

The longitudinal scattering function M is approximated by

$$M'_p(\theta) = g'(\beta_p^2, \theta_h - \alpha_p)$$

where $p \in \{R, TT, TRT\}$, β_p^2 is the longitudinal width of term p, and α_p is the longitudinal shift. This is similar to those originally proposed in [25], but here g' is a unit-height zero mean Gaussian. By enforcing unit-height as opposed to unit-area, artists will be able to control the width of the highlight without affecting its intensity [36].

Each component of the N function for azimuthal scattering is defined separately. The primary highlight N_R can be approximated by

$$N'_R(\phi) = cos(\phi/2)$$

where the constraint $0 < \phi < \pi$ is true. For N_{TT}, the Gaussian g' can be reused as the shape is that of a single lobe. Hence,

$$N'_{TT}(\phi) = g'(\gamma_{TT}^2, \pi - \phi)$$

for azimuthal width γ_{TT}^2. The final component N_{TRT} is further decomposed into the glint and N_{TRT} minus glint, denoted N_G and N_{TRT-G} respectively. Glints are separated out in this model to allow for separate colour and intensity controls, beneficial to artists when working in certain lighting conditions [36]. This gives the following,

$$\begin{aligned} N'_{TRT-G} &= cos(\phi/2) \\ N'_G &= I_g g'(\gamma_g^2, G_{angle} - \phi) \\ N'_{TRT} &= N'_{TRT-G} + N'_G \end{aligned}$$

where γ_g^2 is the azimuthal width of a glint, I_g is the intensity of the glint, and G_{angle} is the half angle between two glints. G_{angle} is made to be different for each hair strand with a random value from 30° to 45° [36].

For each component, the approximation f'_p for $p \in \{R, TT, TRT\}$ is generated by

$$f'_p = C_p I_p M'_p(\theta) N'_p(\phi)$$

where C_p and I_p are the colour and intensity of term p. These are then recombined into the complete approximation, which is the same as in [25], and given by

$$f'_s = \sec^2 \theta \sum_p f'_p$$

In addition to the approximation of [25], [36] also considers the effect of multiple scattering. However, we do not discuss this in detail as the computation is far too expensive to be considered for use in real-time applications. A full implementation is discussed, including multiple scattering, in [36].

2.2.4 Deep Shadow Maps

Self-shadowing is an essential property of rendered hair. Without it, hair appears unrealistic and extremely bright. Up to this point, only local reflectance models have been considered, meaning the presence of one strand does not affect the lighting of another. Deep shadow maps, introduced in [24], aim to remedy this.

A deep shadow map is effectively a texture, each pixel of which stores a visibility function. The value returned by this visibility function is the fraction of the original light that reaches the pixel at the current depth [24]. As an example, consider a ray cast from a light through a volume. The visibility at the pixel would then be defined as the integral of the light transmittance from the ray origin to the pixel at the current depth. The integration is performed numerically by sampling along a ray at specific intervals.

Figure 2.15: Self-shadowing rendered using Deep Shadow Maps.. Image from [24].

In order to generate a deep shadow map, a subset of points are taken from the point of view of the light at which the visibility function is evaluated. All other pixels are evaluated by a weighted combination of the original sample points.

The original paper [24] also covers the details of the compression of deep shadow maps to improve the amout of space required. It is also worth noting that this implementation is not suitable for real-time applications due to the high number of samples required to obtain results of sufficient quality. An example of self-shadowed hair rendered using this technique can be seen in figure 2.15.

2.2.5 Opacity Shadow Maps

An attempt to improve upon [24] was proposed in [21], giving an algorithm that can perform at interactive rates. Hair rendered using this technique can be seen in figure 2.16. In opacity shadow maps, the scene is split into N layers of varying depths, which can be of any distance apart, such that the entire hair volume is contained. Layers are orientated such that they are perpendicular to the direction of the light. The scene is rendered to the alpha buffer N times, clipped at each plane's depth. As such, the alpha

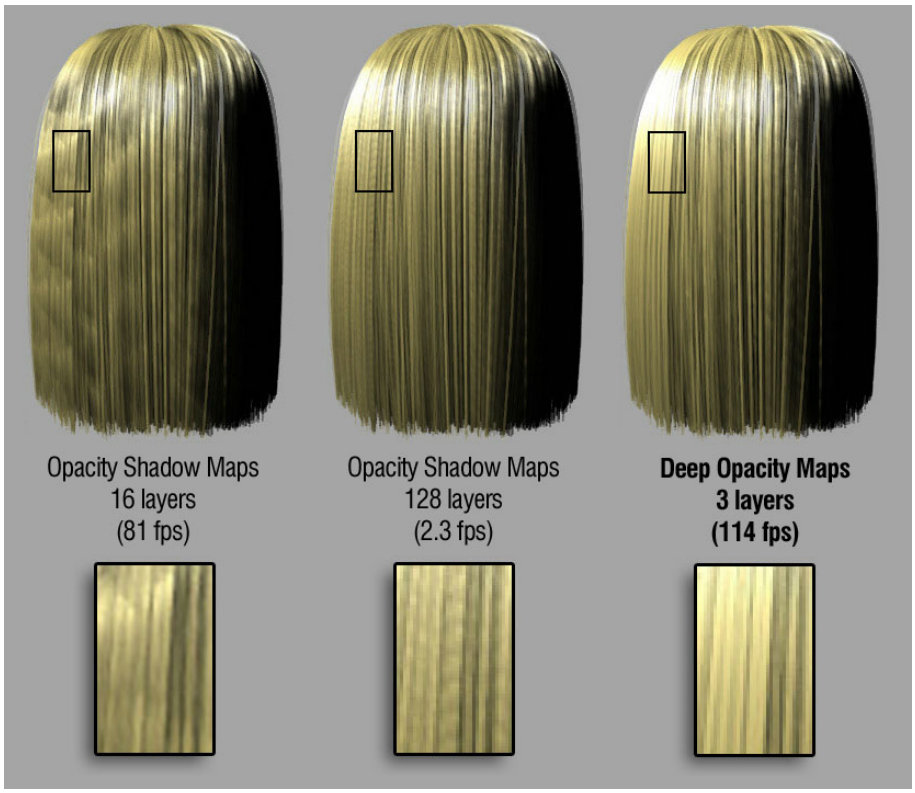


Figure 2.16: A comparison between Opacity Shadow Maps [21] and Deep Opacity Maps [43]. In addition to showing the visual differences, the figure also indicates the large difference in performance between the two implementations. Image from [43].

buffer for each layer now contains the opacity of the volume from the point of view of the light in each pixel [21]. To obtain the opacity at an arbitrary point, the application finds which two layers the point lays between, and linearly interpolates between the two. Full pseudo code can be foud in [21].

While this method provides improved performance over [24], it is lacking in that at least N draw calls are required to obtain the opacity at the different depths. The team developing the *Nalu* Demo found a solution to this problem through the use of multiple render targets (MRT) [28]. With MRT, an application can specify a number of target buffers to write to in a single draw call. In [28], the value $N = 16$ was used. At the time of *Nalu*'s creation, the hardware limit for the number to targets in MRT was 4, allowing the number of required draw calls to be reduced to 4. However, the opacity at each layer is a single value, and as such four layers of opacity data can be stored in a single four channel texture. This allowed for all shadowing information to be rendered in a single draw call [28], greatly improving performance over the implementation in [21].

2.2.6 Deep Opacity Maps

Further improvements to volume self-shadowing were made in [43]. Deep opacity maps build on opacity shadow maps in that a similar layer architecture is used. However, using as few as three layers produces results better than those obtained using 128 layers of opacity shadow maps in a fraction of the time. A comparison between this method and opacity shadow maps can be seen in figure 2.16.

In the first step of the algorithm, the depth of the scene is rendered from the point of view of the light. For each pixel, the application now has the initial depth of the volume, z_0. In addition, the depth of each layer is decided upon. For a three layer implementation, the depths d_0, d_1, and d_2 are defined, meaning the first layer is from z_0 to $z_0 + d_0$, the next from $z_0 + d_0$ to $z_0 + d_1$, and so on. The important point to note here is that as for each pixel z_0 is different, the layers will tend to mould themselves to the shape of the volume [43].

The next step renders the opacity map. The hairs strands are rendered once, and for each fragment the appropriate layer is located. The contribution of the hair is then added to the selected layer through the use of hardware additive blending. Note that opacity contributions for a layer should also be added

to all layers behind it. For an application using only three layers, all of the layers can be stored in a single texture along with the original volume depth z_0 [43]. The number of layers can be increased in a single draw call by using MRT as in [28].

A disadvantage noted in [43] is that it can be difficult to find the correct distances between layers, leading to hairs at the back of the volume not being assigned to any. It is then up to the developer to decide how to handle these hairs. A variety of solutions are discussed along with their respective merits in [43].

An advantage of the method, in addition to the much improved performance over [21], is that as the initial depth z_0 is only used for calculating the starting points of each layers, high precision is not required. It is noted in [43] that there is little to no visible difference when using and 8-bit buffer as opposed to one which is 16-bit.

2.3 Summary

As is clearly indicated by content of the previous sections, a great deal of time and effort has been put into researching optimal methods for the rendering and simulation of hair. At this point, it is evident that simulation models exist which can produce believable motion for thousands of strands of hair. The same can be said for rendering, where the approximations proposed for both the local and global shading models are able to produce images of outstanding quality and realism. The one area in which a lesser amount of research has been performed is with respect to achieving similar results in real-time applications. Although attempts have been made to achieve real-time performance, current implementations either do not provide results on par with the offline methods, or by doing so cause an unreasonable amount of latency in the application. As such, it is clear that further research into this area is required to attain the desired real-time results. In most cases, modern real-time graphics applications are based around a rendering pipeline, which is discussed in detail in the next chapter.

Chapter 3

Rendering Pipeline

In applications which deal with the more complex components of computer graphics, the use of dedicated graphics hardware is often required to provide the required level of performance. This hardware is commonly called the graphics card, or GPU, which is accessed by an application executing on the CPU through an API. There are a wide variety of APIs available for utilising graphics hardware, and when developing portable applications, OpenGL is a good choice due to its cross-platform support, and as a result is one of the most commonly used graphics APIs. However, with the recent advent of Vulkan [15], this may shift in the future. While it is assumed the reader has a basic understanding of how OpenGL functions, this chapter is included to give an explanation of the newer and more complex features that are utilised in the implementation (chapter 4). Links to the documentation on specific components are provided in footnotes where required to avoid referencing the same specification document multiple times. All referenced URLs were last visited 5/05/2017.

While the main focus of this chapter is on the rendering pipeline defined by OpenGL, it is often the case that an application uses a renderer abstraction instead of working directly with the API. This means that the underlying graphics API is hidden by the renderer, meaning the application code is no longer littered with direct API calls. This provides a number of advantages, the biggest of which is that the underlying API used by the renderer can be interchanged without the need for modifications to the application code. This is further discussed in section 4.1, in which the use of a renderer for the project's implementation is considered.

3.1 Compute Shaders

Compute shaders[1] are a new feature which have been in the OpenGL core since version 4.3, and offer a method through which general purpose computations can be performed on the graphics card. A fair question to ask would be why use compute shaders in OpenGL when a similar feature set is provided by using OpenCL, which can even interact with OpenGL using built-in functionality [46]. There are two major advantages to using compute shaders over OpenCL, the first of which is ease of use. When working with OpenCL, before any computation can take place a lot of background work is required. First the application must discover the platform, select the appropriate device, and then create a command queue to invoke kernels. With OpenGL, the background state is already handled by the driver - all the application is required to do is compile the shader and dispatch it.

Another benefit is the use of GLSL[2] to write the shaders. OpenCL uses a variation of the C language to write kernels. While this is sufficient for many purely compute applications, compute shaders in graphics applications often need to use functions dealing with textures and other rendering related features. With OpenCL this would need to be implemented by the developer, while in OpenGL the shading language provides types and functions for working with textures, which includes mipmapping, interpolation, and blending. These additional features of GLSL save a lot of development time and allow for greater efficiency at runtime.

[1] Compute Shader: `www.khronos.org/opengl/wiki/Compute_Shader`
[2] GLSL: `www.khronos.org/opengl/wiki/OpenGL_Shading_Language`

3.1.1 Terminology

A number of new features have been introduced for working with compute shaders, which are discussed here in their respective sections.

Compute Space

The number of compute shader invocations is controlled by two parameters, the global size and the local size. Both are three-dimensional integer vectors, allowing compute shaders to operate in three-dimensional space.

In OpenGL, the local size of a compute shader controls how many invocations will be executed in parallel. Invocations that are executed in parallel are said to be in the same *work group*. The local size of the shader is set by a special `layout` directive at the start of the shader source, just after the `version` directive. In this example, the local size would be $(16, 8, 1)$, meaning a total of $16 \cdot 8 \cdot 1 = 128$ shader invocations in each work group.

```
#version 430 core
layout (local_size_x = 16, local_size_y = 8, local_size_z = 1) in;
```

Global size is used to control the number of work groups to launch. As opposed to local size, global size is set from the client application and not directly by the shader itself. Following on from the example above, if the global size was $(5, 2, 10)$ the total number of shader invocations launched would be $5 \cdot 2 \cdot 10 \cdot 128 = 12800$.

Dispatch

As compute shaders are not a part of OpenGL's rendering pipeline, they must be manually invoked by the host application. When compute shader invocations are launched, this is called *dispatch*. The global size of a compute shader is decided when it is dispatched, which is performed using the `glDispatchCompute` function. If we use the same example as in the previous section, the call to launch the shader would be

```
1 glUseProgram(compute_shader);
2 glDispatchCompute(5, 2, 10);
3 glMemoryBarrier(GL_SHADER_STORAGE_BARRIER_BIT);
```

Note that prior to calling the dispatch function, the compute shader must first be set to active, as with any other shader type, through the use of `glUseProgram`. In addition, it may be neccessary to include a barrier to enforce the completion of memory transactions before any other operations take place. This is performed through `glMemoryBarrier`[3], which takes a bitfield representing the various barriers to enable.

Invocation ID

Each computer shader invocation is assigned a selection of unique identifiers, which are available as built-in inputs for access in GLSL. `gl_LocalInvocationID` is of the type `uvec3`, and contains the current invocation's ID for each dimension of the compute space inside the current work group. `gl_GlobalInvocationID` is the same, but instead identifies the invocation uniquely among all potential shader invocations.

Memory Types

After learning about the invocation ID inputs, it's reasonable to question the need for the local ID if the global ID is unique among all invocations. The reason for this is related to the different types of memory available to compute shaders. There are three different tiers - *global*, *shared*, and *private*. Consider the sample GLSL code below.

```
1 layout (std430, binding = 0) buffer ssbo_data
2 {
3     data[];
4 };
5
```

[3]Memory Barriers: `www.khronos.org/opengl/wiki/GLAPI/glMemoryBarrier#Description`

```
6  shared float workspace[gl_WorkGroupSize.x];
7
8  void main()
9  {
10     int gid = gl_GlobalInvocationID.x;
11     int lid = gl_LocalInvocationID.x;
12
13     workspace[lid] = data[gid];
14     barrier();
15
16     float tmp = workspace[lid] * workspace[lid + 1];
17
18     data[gid] = tmp;
19 }
```

The code listing above gives a simple example compute shader which contains all three of the different memory types. Lines 1-4 declare a SSBO (see section 3.1.1), which is a buffer residing in *global* device memory. Global memory is accessible by all shader invocations, and can be read and written to from the host application. Line 6 declares an array of floats of size `gl_WorkGroupSize.x` in *shared* memory. This type of memory is only accessible to invocations in the same work group. Finally, the variables `gid`, `lid`, and `tmp` are all in *private* memory, which is only accessible to a single shader invocation.

The reason for these different types of memory is the need for performance. Reading and writing to memory is slow relative to the core's operating frequency, so in an attempt to make accesses less expensive, a memory hierarchy is introduced. The slowest to access is global memory, with local memory being several orders of magnitude faster to access, and finally private as the fastest. This is of similar design to the L1, L2, and L3 caches used in processors, except in this case it is the developer's responsibility to control where each variable resides. This means the same restriction of size is also in place, with private memory being small but fast, and global being large but slow.

Shader Storage Buffer Objects

Shader Storage Buffer Objects[4], or SSBOs, are generic buffers used to store data in device memory. They are similar to Uniform Buffer Objects[5], but have a lot fewer restrictions. The most obvious is that SSBOs are not uniform, and as such can be both written to and read from. SSBOs can also be much larger. While UBOs can only be a maximum of 16KB, SSBOs can be up to 128MB. This is however not without cost, as SSBO memory accesses are going to be slower than those to a UBO.

SSBOs are created in the same way as the other device buffers through `glGenBuffers` and the `glBuffer*` functions, with `GL_SHADER_STORAGE_BUFFER` as the target. In addition, the created SSBO must be bound to a specific point using the `glBindBufferBase` function. Using the previous code snippet as an example, the SSBO would need to be bound to index 0.

Certain constraints are also imposed upon the layout of data in an SSBO to enforce specific alignments to allow for efficient memory transactions. These constraints have been greatly relaxed as of OpenGL 4.3, allowing for tighter packing of data into the buffer. The `std430`[6] layout parameter enables this property.

Atomic Operations

Atomic operations[7] are a special type of function that can be applied to SSBOs and variables declared as `shared` in a compute shader. These atomic functions enforce ordering between multiple attempts to work with the same memory location to prevent race conditions from developing. Atomic functions return the original value of the memory location prior to the current call. This makes it useful for a variety of purposes, such as allocating a unique identifier between multiple parallel shader invocations. Examples of atomic operations include `atomicAdd` and `atomicMax`.

[4]Shader Storage Buffer Object: `www.khronos.org/opengl/wiki/Shader_Storage_Buffer_Object`
[5]Uniform Buffer Object: `www.khronos.org/opengl/wiki/Uniform_Buffer_Object`
[6]Memory Layout and Constraints: `www.khronos.org/opengl/wiki/Interface_Block_(GLSL)#Memory_layout`
[7]Atomic Operations: `www.khronos.org/opengl/wiki/Shader_Storage_Buffer_Object#Atomic_operations`

3.1.2 Usage

We now consider an example of how to use a compute shader from creation to obtaining results. For the sake of simplicity, we will use the standard GPGPU example of the addition of two vectors. As a GLSL compute shader, this is written as follows.

```
#version 430 core

layout (local_size_x = 128, local_size_y = 1, local_size_z = 1) in;

layout (std430, binding = 0) ssbo_a { float A[]; };
layout (std430, binding = 1) ssbo_b { float B[]; };
layout (std430, binding = 2) ssbo_c { float C[]; };

void main()
{
    C[gl_LocalInvocationIndex] = A[gl_LocalInvocationIndex] +
        B[gl_LocalInvocationIndex];
}
```

This shader is compiled in the same way as any other shader programs, with the exception that the target is `GL_COMPUTE_SHADER` and only a single shader can be attached to a program. This is illustrated below. Note that for the sake of being concise, error checking is not present in the following listings.

```
GLuint tmp, vadd;

const GLchar* vadd_src = /* ... */;

tmp = glCreateShader(GL_COMPUTE_SHADER);
glShaderSource(tmp, 1, &vadd_src, NULL);
glCompileShader(tmp);

vadd = glCreateProgram();
glAttachShader(vadd, tmp);
glLinkProgram(vadd);

glDeleteShader(tmp);
```

With the shader ready, the next step is creating the SSBOs. The setting up of SSBO `A` is shown below.

```
GLfloat a[128];

// ...

GLuint ssbo_a, bind_point = 0;

glGenBuffers(1, &ssbo_a);
glBindBuffer(GL_SHADER_STORAGE_BUFFER, ssbo_a);
glBufferData(GL_SHADER_STORAGE_BUFFER, 128 * sizeof(GLfloat), a, GL_STATIC_DRAW);
glBindBufferBase(GL_SHADER_STORAGE_BUFFER, bind_point, ssbo_a);
glBindBuffer(GL_SHADER_STORAGE_BUFFER, 0);
```

This is also fairly self explanatory as it is similar to the creation of other buffers such as VBOs[8] and EBOs[9]. Note the call to `glBindBufferBase` sets the binding point to zero. This process would be repeated for SSBOs `B` and `C`, with `bind_point` set to `1` and `2` respectively. With this completed, it's time to dispatch the compute shader. This would be achieved by the following code.

```
glUseProgram(vadd);
glDispatchCompute(1, 1, 1);
glMemoryBarrier(GL_ALL_BARRIER_BITS);
```

[8] Vertex Buffer Object: `www.khronos.org/opengl/wiki/Vertex_Specification#Vertex_Buffer_Object`
[9] Element Buffer Object: `www.khronos.org/opengl/wiki/Vertex_Specification#Index_buffers`

Note that the global size is one in all dimensions - this is because in the shader, the local size is set to 128, which corresponds to the number of elements in the input arrays. We ensure the computation is complete with the memory barrier. Finally, the results can be read back from the SSBO `C` as follows.

```
GLfloat c[128];

glBindBuffer(GL_SHADER_STORAGE_BUFFER, ssbo_c);
glGetBufferSubData(GL_SHADER_STORAGE_BUFFER, 0, 128 * sizeof(GLfloat), c);
glBindBuffer(GL_SHADER_STORAGE_BUFFER, 0);
```

At this point, the contents of the array `c` will contain the result of adding `a` and `b`. Once they're no longer required, resources can be destroyed as normal using the `glDelete*` functions.

3.2 Tessellation Pipeline

OpenGL 4.0 introduced tessellation to the rendering pipeline. In computer graphics, tessellation is commonly used to further subdivide surfaces at runtime for added detail, or to generate surfaces from a set of given points. While OpenGL 4.0 was released in 2010, it is still relatively underused when compared to other features. As such, this section examines the new rendering pipeline from raw data to rendered fragments.

3.2.1 Vertex Shader

The first step of the pipeline is the vertex shader[10]. The vertex shader is used to perform operations on a per-vertex basis, with each shader invocation given access to a single vertex as referenced by the draw call. In order to access vertex data, the most common method is through the use of vertex attributes[11]. Consider the following snippet from a vertex shader.

```
layout (location = 0) in vec3 v_Position;
layout (location = 1) in vec3 v_Colour;
```

This declares that every input vertex will have two attributes, a position and a colour, both of which are three-dimensional single-precision floating-point vectors. The inputs `v_Position` and `v_Colour` can then be used in the shader as if they were constants. Vertex attributes are defined and enabled using the functions `glVertexAttribPointer` and `glEnableVertexAttribArray` respectively. For the example above, the attributes would be enabled as follows. This assumes the VBO has already been bound and created, and that vertex data is stored in a tightly-packed interleaved fashion.

```
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 6 * sizeof(GLfloat), (GLvoid*)0);
glEnableVertexAttribArray(0);

glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, 6 * sizeof(GLfloat),
    (GLvoid*)(3 * sizeof(GLfloat)));
glEnableVertexAttribArray(1);
```

Per-vertex outputs can also be defined, which allow the passing of generic data on to the next shader stage. There must be a one to one mapping from input vertex to output vertex - that is to say, the vertex shader cannot generate additional vertices.

An additional but useful input is `gl_VertexID`, which holds the index of the vertex in the VBO which is being processed by the current shader invocation. This is particularly useful when using indexed rendering through an EBO, as will be made clear in chapter 4.

3.2.2 Tessellation Terminology

Before the actual tessellation shader stages can be discussed, some additional terminology must first be introduced.

[10] Vertex Shader: `www.khronos.org/opengl/wiki/Vertex_Shader`
[11] Vertex Attributes: `www.khronos.org/opengl/wiki/Vertex_Shader#Inputs`

Patches

Patches are a primitive type defined in OpenGL by `GL_PATCHES` which denotes a generic type of primitive composed of a specified number of vertices, n. This means that when a draw call is issued using patches as the primitive type, every n vertices will be assigned to a different patch. Once vertices are assigned into a patch, they are tessellated into an abstract space, defined by coordinates in the range of $[0, 1]$. Note that it is up to the developer to ensure tessellated patches blend together seamlessly.

The size of the input patch, n, is set through the function `glPatchParameteri` with `GL_PATCH_VERTICES` as the target. The number of vertices in the output patch, which dictates how many tessellation control shader invocations will be active for each patch, is set in the source of the tessellation control shader.

Tessellation Levels

The degree of patch tessellation is controlled by *outer* and *inner* tessellation levels. These are defined by four- and two-dimensional floating-point vectors respectively. While there are differences depending on the primitive type being tessellated, in general the level defines how many sections an edge will be subdivided into. The outer level defines this for edges on the border of the primitive, while the inner is for those entirely inside the abstract patch.

In addition to the degree of tessellation, how the generated coordinates are spaced within the abstract patch can be controlled by the spacing directives `equal_spacing`, `fractional_even_spacing`, and `fractional_odd_spacing`. Each has their own ideal use-cases, but by far the most common is `equal_spacing`, which evenly distributes the coordinates across the patch.

Tessellation Primitives

Only three types of primitive can be tessellated in OpenGL - triangles, quads, and isolines. When working with triangles, generated points are given as barycentric coordinates. Quads are defined as a square space, and isolines are defined by a square space of horizontal lines. While triangles and quads are not discussed in detail as they are not relevant to this project, the tessellation process for isolines is fully discussed in section 3.2.4.

3.2.3 Tessellation Control Shader

The next shader stage after the vertex shader is the tessellation contol shader[12], or TCS. The TCS is responsible for setting the tessellation levels for a given patch, and is invoked as many times as is specified by the size of the output patch. This is set in the TCS as follows, where `output_size` is a positive integer which defines the size of the output patch.

```
layout (vertices = output_size) in;
```

To set the tessellation levels, the shader sets the values of `gl_TessLevelOuter` and `gl_TessLevelInner`, which are both floating-point arrays. The values set here control the degree of tessellation and consequently the number of invocations of the tessellation evaluation shader.

The TCS has two types of outputs. Per-vertex outputs are defined as an array which automatically has its length set to the size of the output patch. A TCS invocation is only permitted to write to its own position in an output array, which is defined by `gl_InvocationID` in a similar way to compute shaders. An additional "patch" output type is also available to the TCS. Patch outputs are defined by prefixing the output definition with the `patch` keyword. All invocations of the TCS operating on the same patch will have access to the same patch outputs. This is particularly useful for passing the same information to all invocations of the tessellation evaluation shader operating on the same patch.

Along with compute shaders, the TCS is the only other shader stage to support synchronisation via the `barrier` function.

The TCS is an optional shader stage. If the developer chooses to, parameters can be set using the `glPatchParameter*` functions from the host application.

3.2.4 Tessellator

The tessellator[13] is a fixed-function stage of the pipeline that the developer cannot modify directly, and is instead controlled by arguments set by the TCS. The tessellator performs the actual tessellation of

[12] Tessellation Control Shader: `www.khronos.org/opengl/wiki/Tessellation_Control_Shader`
[13] Tessellator: `www.khronos.org/opengl/wiki/Tessellation#Tessellating_primitives`

the abstract patch primitive as specified by the TCS, and forwards the calculated coordinates on to the tessellation evaulation shader. As was previously mentioned, the tessellator can operate on triangles, quads, and isolines. As isolines are a common representation of hair, we now discuss in detail the tessellation process for this type of primitive.

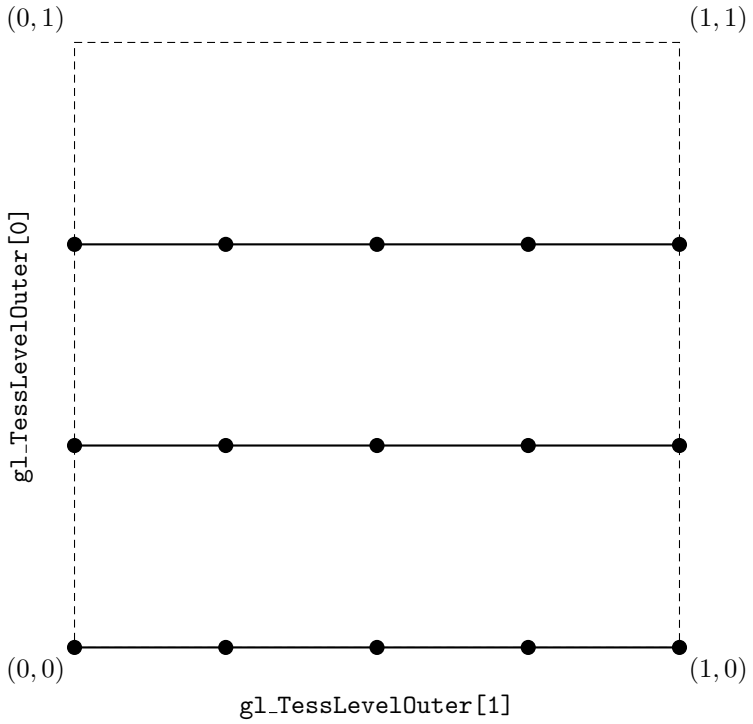


Figure 3.1: The abstract patch of isolines in the OpenGL 4.0 tessellator. In this example, the tessellation level is (4, 3) with `equal_spacing`.

The abstract patch of isolines is a series of horizontal lines, and is shown in figure 3.1. In the diagram, the dashed line represents the extent of the abstract patch's shape, whereas the solid lines are the resulting isolines produced by the tessellator. The solid circles indicate where the coordinates are placed within the patch. When working with isolines, only the outer tessellation levels have any effect. In the diagram, one can see how each level affects the result. `gl_TessLevelOuter[1]` controls how many *edges* are in a line. It's important to stress that the level sets the number of edges, *not* the number of vertices. Meanwhile, `gl_TessLevelOuter[0]` controls the number of isolines to generate. Notice how in this dimension the generated coordinates do not reach to the edge of the space, whereas for `gl_TessLevelOuter[1]` the tessellator guarantees that the first and last coordinates along an isoline are at zero and one respectively.

When working with isolines, the resulting geometry is a series of line strips. For both triangles and quads, the result will be a set of triangles.

3.2.5 Tessellation Evaluation Shader

The tessellation evaluation shader[14], or TES, receives the coordinates generated by the tessellator. In the shader, these can be accessed in the three-dimensional vector `gl_TessCoord`. Of course, when working with isolines only the first two components will have meaningful values. The number of TES invocations is not strictly defined, but the TES is guaranteed to be invoked at least once for each vertex provided by the tessellator.

The TES is responsible for setting the abstract patch type (`isolines`, `triangles`, or `quads`) and optionally the spacing values and primitive winding order. These are controlled by the `layout` directive at the start of the shader.

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

[14]Tessellation Evaluation Shader: `www.khronos.org/opengl/wiki/Tessellation_Evaluation_Shader`

The main job of the TES is interpolation - it uses the coarser data provided as inputs to generate geometry at a finer level of detail. The inputs are equivalent to the outputs defined in the TCS, or alternatively the vertex shader if a TCS was not attached to the program. Outputs can also be defined, but as with the vertex shader the TES can only produce a single vertex. The presence of the TES is required for tessellation to be enabled.

3.2.6 Geometry Shader

The geometry shader[15] is another optional vertex processing shader stage. The shader is passed a single primitive as its input, and by using it as a base is able to generate additional vertices and primitives. This is useful for expanding geometry such as lines into triangles without needing to store the additional vertices in the VBO.

Input	TES Output	Vertices
`points`	`point_mode`	1
`lines`	`isolines`	2
`triangles`	`triangles`, `quads`	3

Table 3.1: Input options to the geometry shader, how many vertices each takes, and the corresponding TES output type.

The shader can take one of primitive types listed in table 3.1 as its input. Note that this is not a comprehensive list of primitive types - a full list can be found in the OpenGL specification. As the geometry shader operates on multiple input vertices, the per-vertex outputs of the previous shader stage are aggregated into arrays of appropriate length. In addition to the input type, the geometry shader must also specify its output primitive type as one of `points`, `line_strip`, or `triangle_strip`, and the maximum number of vertices the invocation may emit. An example of defining the shader properties is given in the listing below.

```
layout (isolines) in;
layout (line_stip, max_vertices = 4) out;
```

When the geometry shader wishes to emit a vertex, the function `EmitVertex` is called. Before doing so, any per-vertex outputs must be set to their intended value, including those that remain the same for all emitted vertices. This is essential as after `EmitVertex` has been called, outputs no longer have a defined value. Once all the vertices of a primitive have been emitted, the shader calls `EndPrimitive`. Note that multiple primitives may be emitted by the same geometry shader invocation as long as the number of emitted vertices does not exceed `max_vertices`.

Another useful feature of geometry shaders is layered rendering. This allows for the rendering of a primitive to multiple images[16] without the need to bind a different render target. Note here that we use the OpenGL definition of an image, which is an array of pixels of a certain dimensionality with defined size and format. This means that a three-dimensional texture can contain multiple images as separate layers, and as such layered rendering allows for the rendering of primitives to multiple layers in the same texture. Layered rendering is achieved by setting the built-in integer output `gl_Layer` to the zero-based index of the target layer.

3.2.7 Fragment Shader

The final shader stage of the rendering pipeline is the fragment shader[17], which sets the output colour of the current fragment being processed. Fragment shaders are of particular use for performing an operation which needs to be calculated on a per-pixel basis, with the classic example being per-pixel lighting for smooth shading. The shader can also optionally reject fragments using the `discard` keyword, which prevents them from being written to the render target.

Prior to the introduction of compute shaders, fragment shaders were used to perform GPGPU operations using OpenGL. This would be achieved by storing the data inside a texture, which would be

[15] Geometry Shader: `www.khronos.org/opengl/wiki/Geometry_Shader`
[16] Textures and Images: `www.khronos.org/opengl/wiki/Texture`
[17] Fragment Shader: `www.khronos.org/opengl/wiki/Fragment_Shader`

bound as the render target. By using a VBO containing the vertices of a screen-space rectangle covering the entire framebuffer[18], the fragment shader could be invoked for each pixel and hence each data item. Input data would be accessed through the use of texture samplers, and outputs written by setting the fragment colour. A full discussion on how to use OpenGL for GPGPU before the introduction of compute shaders is discussed in [14].

[18] Framebuffer Object: `www.khronos.org/opengl/wiki/Framebuffer_Object`

Chapter 4

Project Execution

In this chapter the implementation of the real-time hair system is discussed, including the main features, subsequent enhancements and optimisations, and the motivations behind the decisions made. There are two different versions of the implementation available, `gpu` and `optimised_gpu`. The reason for this is to illustrate the importance of how the same operations are performed with respect to efficiency on the graphics card. To enable access to compute shaders, the OpenGL 4.3 core profile is used. The host application is written in C++14, with GLSL used for shaders.

It is also worth emphasising at this point the reason behind the implementation and model being so closely coupled. Due to the complexities of developing for a GPU over a CPU, a number of constraints are imposed upon the underlying models, dictating when and where they are appropriate. Of course, this also applies in the other direction, as the model itself will constrain the implementation. While in general it may be considered preferable to separate the mathematics and implementation, they are kept together in this chapter as the constraints between them are of such importance, to the point that separating them would lead to a more confusing and less concise narrative.

4.1 Renderer Abstraction

Before starting the development of any graphically intensive application, it is often a good idea to question whether the use of a *renderer* is appropriate. The purpose of the renderer is to hide the true underlying graphics API that is being used beneath one or more layers of abstraction. This is often performed by providing a set of classes which wrap and abstract the functionality into a more logical set of components, making the code easier to read, write, and maintain.

Another benefit of using a renderer is that the underlying graphics API is no longer tightly coupled and interleaved with application code. Instead, the only calls that interact with the graphics card are wrapped and hidden behind the renderer. As such, it becomes possible to provide multiple renderer implementations, each of which uses a different underlying API. For example, two renderers could be provided, with the default being implemented in DirectX, falling back to OpenGL if the first choice is not supported. Another example is software rendering, which is performed on the CPU when there is no dedicated graphics hardware available. As long as a renderer implementation is provided, the application can seamlessly operate even when there is insufficient hardware, leading to a highly portable and flexible application.

As a result of these benefits, a renderer was implemented for the project. The underlying implementation used is, as has previously been stated, the OpenGL 4.3 core profile. Adding the renderer meant that working directly with OpenGL was not required where the renderer's implementation allowed it, leading to a more logical development approach. While this did take some time to create, it allowed for much faster development once completed due to the simpler interface and automated error checking and logging capabilities. However, it is worth noting that the renderer does not wrap all OpenGL functionality, and hence some direct API calls are still required. This was to avoid spending an unreasonable amount of time on development, and as such could easily be remedied if required.

One component of this wrapper which deserves special mention is the custom GLSL shader preprocessor. As GLSL shaders must be compiled by the host application at runtime, it is invoked by an OpenGL function which unfortunately does not allow for any parameters to be passed. This meant that the host application could not change the constants in the shader prior to compilation, requiring the hardcoding of values before the application is launched. To get around this problem, a custom shader

preprocessor was written which replaces any defined constants, prefixed with an `@` symbol, with their set values. This was preferable to having to change their values directly in the source file or pass in the values as uniforms from both a development and performance perspective.

4.2 Simulation

The simulation aspect of the hair was considered first, taking into account the dynamics of both individual strands as well as interactions between strands of hair.

4.2.1 Main Loop

The entire application is powered by a single super-loop, as is often seen in games. While in most cases this would not be worth mentioning, it is necessary here to provide information on how the simulation is kept stable. In a lot of applications a variably-sized update time step is used. This is predominantly because it is straight-forward - simply calculate the time between the current update and the previous, and that's the amount of time that should be accounted for. However, this leads to problems when performing physics simulations, often leading to the simulation's state becoming unstable. The alternative is to lock the frame rate of the application to the desired update rate, but this is also undesirable as it prevents the full utilisation of more powerful hardware when it is available.

As such, an alternative method is proposed which allows for both a fixed time step and unlocked frame rate. The idea is that instead of performing an update every iteration of the main loop, the application instead keeps track of the amount of latency since the last update. When this latency is greater than or equal to the time step of the simulation, the update method is called. This is better illustrated by the listing below.

```
1  double current_time, previous_time, elapsed_time, latency = 0.0;
2
3  // ...
4
5  while (main_loop_running)
6  {
7      current_time = get_time();
8      elapsed_time = current_time - previous_time;
9      previous_time = current_time;
10     latency += elapsed_time;
11
12     process_input();
13
14     while (latency >= TIME_PER_UPDATE)
15     {
16         update();
17         latency -= TIME_PER_UPDATE;
18     }
19
20     draw(latency / TIME_PER_UPDATE);
21 }
```

Note that a `while` is used as opposed to a simple `if` statement to allow the application to catch up should it be neccessary. In addition, notice that the rendering function takes an argument, `latency / TIME_PER_UPDATE`. One problem with this system is that the draw call does not always align with updates, meaning the state remains the same for multiple draws and suddenly switches to the new state when `update` is eventually called. To avoid this, the aforementioned argument is passed to the `draw` function which represents an extrapolation factor. This allows for the application to attempt to predict how the scene will look after the next update, allowing for smoother transitions between states. While this is not exactly ideal, it is definitely preferable to having an unstable simulation or a locked frame rate.

4.2.2 Hair Model

For the modelling of each hair strand, it was decided to use a version of the mass-spring model. In this model, each hair strand is modelled as a chain of particles sequentially constrained by inextensible links. This model was selected for a variety of reasons, the first of which is simplicity. This simplicity is advantageous for two main reasons - performance and space. Simulating the forces acting on particles is a well explored problem, allowing for a wide selection of potential integration methods. As these methods often consider particles individually, this model also allows for efficient parallelisation on the GPU. The other benefit is in regard to the amount of memory required to store the hair state. As the particles are connected by inextensible links, the only information on the link that needs to be known is its length. This means that only particle information needs to be stored in addition to a single floating-point value for every constraint, saving memory on the GPU.

Of course, this simplicity can also work as a disadvantage. As was stated in section 2.1.1, the use of strong forces on the springs can lead to an unstable simulation. However, this was deemed acceptable as in a majority of cases the hairs will not be put under larger forces, and as such the situation is not worth accounting for due to the potential loss of performance in more common situations. In addition, by using a fixed time step as discussed in the previous section, the simulation should be sufficiently stable.

Modelling all the particles on the hair is of course not an option in a real-time environment, and as such only a subset of strands, henceforth referred to as keyhairs, are stored. The simulation update steps are performed on this subset of hairs, allowing for a sufficiently detailed approximation within the available time budget. When the rendering stage is reached, the keyhairs are interpolated both along and between the strands. This is explained in greater detail in section 4.3.1, but the intuition here is that interpolating between a smaller number of keyhairs will lead to a full head of hair without the need for additional simulation.

4.2.3 Strand Dynamics

In order to update the hair strands, the method decided upon was a modified version of that proposed in [26]. Position based dynamics [27] is used to perform the integration step of the physics update, and additionally the DFTL algorithm is used to solve constraints between particles. As position based dynamics is used, it is only necessary to store the particle positions and their velocities. DFTL is a good algorithm to use for constraint solving as it does not require multiple iterations to converge.

While the integration step divides easily into a parallel model, with each particle being considered separately, this is not the case for the solving of constraints as it requires sequentially traversing down each strand of hair. In the `gpu` version, the constraint solving is fairly simplistic, in that each shader invocation handles the sequential solving of a single hand. While this guarantees correct results, it does not lend itself to good performance, taking an increasingly large amount of time the more particles are added to a strand.

The constraint solving method is parallelised in `optimised_gpu`, allowing for each particle to be considered individually and hence allowing for a greater degree of parallelism. Instead of taking the constraints into account sequentially, even and odd constraints are solved alternatively. If a particle is valid to be solved, denoted by whether the shader invocation's assigned identifier is even or odd, it is projected as per the DFTL algorithm, correcting its position. The next set of particles, which are guaranteed to be in-between those that have just been solved, are in turn projected, correcting themselves but potentially invaliding those that have already taken place. As such, multiple iterations are required. As one of the key reasons for using the DFTL algorithm was the fact it only requires a single iteration, the requirement of multiple iterations could be seen as a problem. However, as few as five iterations of this parallel solving method achieved visually similar results when compared to the serial version at a much greater performance. In addition, this implementation will scale better as the number of particles in a strand are increased due to the higher degree of parallelism. The listing below outlines the solving process.

```
1 for (uint i = 0; i < ftl_iterations; ++i)
2 {
3     if ((id % 2) == 1)
4         project_position();
5
6     barrier();
7
```

```
8       if ((id % 2) == 0)
9           project_position();
10
11      barrier();
12  }
```

In addition to moving correctly, the hairs must also correctly collide with other solid geometry, most importantly that of the character. In order to achieve fast collisions, a simplified collision geometry was used to approximate the model to allow for fast collision detection. Initially, a method was proposed to enforce collision constraints and distance constraints between the particles of the keyhair in a single step. To allow for this, two equations were derived from the fact that the projected position of a particle after DFTL is on a sphere of radius equal to separation of a hair, and if it is inside the collision geometry it follows that the particle must also be projected onto the surface of the sphere. This is illustrated in figure 4.1.

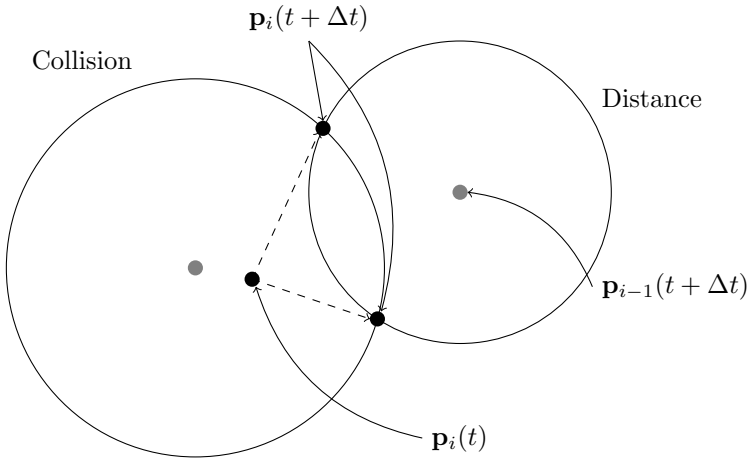


Figure 4.1: Showing how to solve both collision and distance constraints based on the spherical model used for collision and DFTL.

If the length of the distance constraint between particles in a strand is d_{strand} and the radius of a collision sphere is $d_{collision}$, we can obtain the following.

$$|\mathbf{p}_i(t + \Delta t) - \mathbf{p}_{i-1}(t + \Delta t)| = d_{strand}$$
$$|\mathbf{p}_i(t + \Delta t) - \mathbf{c}_{collision}| = d_{collision}$$

where $\mathbf{c}_{collision}$ is the centre of the collision sphere. By solving the two equations to obtain $\mathbf{p}_i(t+\Delta t)$ the resulting location of the particle can be found which conforms to both constraints. Of course, this leads to two potential solutions, but as long as the system for selecting a solution is consistent visual artifacts such as flickering between states would not occur.

The downside with this method is that the equations are deceptively simple - that is to say, they are difficult to solve in an efficient manner. As this operation would need to be performed for all particles against every collision sphere, this is obviously not a desirable trait. As such, a much simpler method is also proposed which simply iterates through all constraints, solving in a sequential manner. While this does not provide a completely accurate result, with a sufficiently small time step the resulting simulation state is stable enough to avoid visual discrepancies. Due to its performance benefit over the already discussed method while producing visually similar results, it is the preferable solution and as such used in the implementation.

4.2.4 Hair to Hair Collisions

For the solving of hair to hair collisions, the volumetric approximation method proposed in [31] seemed most appropriate as it could provide tunable performance dependent on hardware availability by changing the resolution of the voxel grid. In order to calculate the density of each voxel, the voxel itself must know which particles are contained by it and its neighbours. This is a typical nearest neighbour problem, which is non-trivial to implement on a graphics card.

The first attempt was through the use of locality-sensitive hashing, or LSH, first introduced in [13]. LSH can be used to find approximate nearest neighbours by dividing the space into a series of planes. Each particle is evaluated with respect to every plane, indicating whether it is above or below it. The result of each test is combined into a signature, which can then be used to store the particle in a bin as is performed in a hash table. This means that for each voxel, its position can be evaluated to build a signature, and then its neighbours should be located in the same bin.

In theory, this would be well suited to the problem as it is fast, with each evaluation requiring only a single dot product, and additionally being precise is not of paramount importance for the problem due to the generation of visually similar results. Unfortunately, due to the requirement of taking place entirely on the graphics card, problems arise with the implementation of the hash table. Storing objects in bins is problematic with respect to memory accesses. As for each particle there is no way to know which bin it will fall into until it has been hashed, the memory accesses will likely be scattered, causing poor performance. Additionally, the number of particles which will be hashed into each bin is unknown. The way to handle this would be to either have a maximum capacity for each bin, or set all bins to have the capacity matching the total number of particles to handle the worst case. Obviously, neither are ideal as the former leads to a potentially large number of missed neighbours, while the latter is extremely memory inefficient. On top of this, each bin would require its own atomic counter. The sheer cost of this method on a GPU means it not worth using unless no other solutions are available. Having said this, GPU implementations of hash tables are available [42], but are of considerable complexity.

Another attempt was made through the adaption of a GPU nearest neighbours algorithm, as outlined in [16]. The presentation cited details an efficient method to find the nearest neighbours of a particle within a fixed radius for applications such as fluid simulations. The space is partitioned into a set of bins, each of which is represented as a counter holding the number of particles assigned to it. The particles are then sorted by bin by performing a prefix sum and counting sort. The reason behind this is to ensure that memory transactions are ordered, and hence more efficient - the main weakness of the LSH method.

This method can be adapted to hair to hair interactions by effectively ignoring the radius constraint around the particles. This means that particles in the same bins are considered to be neighbours. The main weakness of this method was the amount of copying between buffers required. In [16], re-indexing is performed to put the particles into bin-sorted order. This cannot be performed for the hair application, as the order of the particles is important for the solving of constraints between particles in the same strand. As such, an additional buffer is required to re-index into, which leads to a certain amount of overhead due to the additional reads and writes. While this does offer an improvement over the LSH method, it is still far from ideal.

The solution that was settled upon for the `gpu` implementation was to make use of the `atomicAdd` function provided by GLSL. To calculate particle density, a compute shader is invoked for each particle, which calculates which voxel it is located inside. For each voxel in the 3x3x3 volume around this centre voxel, the densities are calculated and atomically added to the current voxel's total density. Those familiar with the GLSL `atomic*` operations would have noticed that the voxel density is a floating-point value, but the GLSL atomics only support integer operations. As such, after the density calculation is completed in floating-point, the value is converted to 16.16 fixed-point, which is stored in the buffer as a `uint`. This is a much simpler method than those previously discussed and provides real-time performance.

However, it is not without its problems. As with LSH, which particle will end up in each bin is initially unknown, and as such a bin needs to be stored for each voxel, which scales cubically in memory with the number of voxels per dimension. This isn't only inefficient with respect to consumed space - the more scattered memory transactions also lead to slower performance. The cost to clear the buffers to zero is also significant, requiring potentially multiple milliseconds to complete. In addition, the average velocity also needs to be stored, which takes up three additional positions in the buffer, and as such requires three more atomic adds. That's not to say the method is without its merits, as it theoretically scales linearly with the number of particles, not the number of voxels. Of course, due to the more scattered accesses this is not always the case in practise.

The best solution found makes use of OpenGL's built-in rendering pipeline for the bulk of the computation, as opposed to using compute shaders. Instead of storing the voxels in a buffer, they are stored as a three-dimensional texture, such that each texel represents a single voxel. By setting the texture as the `GL_RGBA32F` format, full single-precision floating-point is enabled in the hardware, with automatic clamping also being disabled by default. This texture is attached as the colourbuffer of a framebuffer object, or FBO, which is traditionally used for off-screen rendering in techniques such as shadow mapping.

To calculate the densities, the FBO is set to active and the hair volume is rendered as points. Most of the vertex processing stage of this program is straight-forward, simply applying the transformation into

voxel space. Once the geometry shader is reached, it makes use of the `gl_Layer` output to set which layer of the texture should be rendered to in the fragment shader. This layer is calculated based on the depth component from the particle once it has been transformed to voxel space. To handle adding contributions to neighbouring voxels in the other layers, the geometry shader also emits a vertex for the layer before and the layer after. To account for neighbours in the same layer, `glPointSize` is set to `3.0f`.

The fragment shader itself performs the density calculation. The particle's velocity contribution is stored in the RGB channels of the output colour, with density stored as alpha. By enabling additive blending, the contributions are summed by OpenGL automatically, resulting in each texel storing the corresponding voxel's total density and velocity. When these are used to calculate the average velocity in a compute shader, a single texture read is required using a `sampler3D`, then dividing the RGB channels by the alpha to get the average.

This method provided performance several orders of magnitude faster than its competitors. As the combination is managed by OpenGL, the performance is mostly limited by the quality of the OpenGL vendor's implementation of additive blending. In addition, accessing neighbours is efficent within the volume due to how textures are stored in OpenGL - neighbouring texels are located close together in memory to allow for maximum performance. Resetting the density and velocity to zero is also efficient in the method thanks to using an FBO. When clearing is required, the FBO is bound and then `glClear` is used to clear the colourbuffer to zero. Due to the high quality of this method, it was used in the `optimised_gpu` implementation to good effect.

4.2.5 Memory Layout

While this is not a big concern in general applications programming, memory layout is extremely important in high performance computing, especially when working on a GPU. In the `gpu` version, the particles are laid out in memory as the GLSL type `vec3`, but due to the restrictions on the `std430` buffer format, these are padded to `vec4`s to ensure aligned reads and writes. The particles are stored such that all the roots of each strand are stored together, followed by the next, and so on. This is in an attempt to allow for coalesced memory transactions. A coalesced transaction occurs when multiple GPU cores are accessing neighbouring memory locations, at which point all of the operations are combined into a single transaction, improving performance.

However, this memory layout is not ideal. Consider the solving of constraints, which requires sequentially stepping through each particle of a hair. Due to the particle positions being stored as padded three-dimensional vectors, a quarter of the bytes for each transaction are effectively wasted. This is bad for performance as it leads to both wasted memory bandwidth and a smaller amount of meaningful data being transferred in a coalesced transaction.

A better solution is to decompose the vectors such that all the x components are stored together, then the y components, and finally the z. This removes any need for additional padding and also ensures all data in a transaction is meaningful. This method is used in the `optimised_gpu` version, and provides greatly enhanced performance.

4.2.6 Compute Shaders

The `gpu` implementation uses a total of five compute shaders, listed below.

- `update_position` - Calculates the next position based on external forces ignoring all constraints, storing the result in a temporary buffer to preserve original position.
- `apply_constraints` - Solves all constraints acting on the particles. Updates temporary positions and calculates correction vectors for use in DFTL velocity correction.
- `update_velocity` - Calculates the updated velocity based on the difference between the original and new velocities and the correction vector as per DFTL. Additionally overwrites original positions with those stored in the temporary buffer.
- `calculate_densities` - Performs space partitioning and density calculation using the fixed-point 16.16 method described above.
- `correct_velocities` - Modifies velocities of particles by using the averages calculated for each voxel to simulate friction.

As the shaders were decomposed such that each only performs a single major function, it was necessary to create five SSBOs. Note that the position SSBO is manually double buffered for storing the temporary positions.

For `optimised_gpu`, the following shaders were required.

- `update` - Performs all steps of the physics update in a single compute shader. This is essentially combining the first three shaders listed in the previous section into a single compute shader for better performance.
- `voxels` - This is actually a combination of a vertex shader, geometry shader, and fragment shader which implements the density and average velocity contributions as per the three-dimensional texture method described above.
- `friction` - Compute shader that performs velocity adjustments by reading the results stored in the three-dimensional texture.

This implementation also only requires a single SSBO, in which positions and velocities are stored at different offsets. Note that it is no longer necessary to store the temporary positions and correction vectors in global memory as they do not need to be persistent between shader stages. A single three-dimensional texture is also used for the voxels. Using fewer SSBOs is beneficial as it frees the other binding points for use in other operations by the application the hair is used in. Additional optimisations were performed to the way the GLSL was written to allow for more efficient execution on the GPU.

4.3 Rendering

With simulation complete the next step is to perform rendering. This was made to be an intentionally separate component from simulation so the rendering method could be adapted for various artistic styles, although this modular design is somewhat sacrificed for efficiency in the `optimised_gpu` implementation.

4.3.1 Interpolation

There are two forms of interpolation performed in the rendering stage, the first of which is smoothing. The simulation stage uses a relatively low number of particles per strand to allow for the high simulation speeds required for real-time applications, but this can lead to blocky geometry in which it is clear where the particles are due to the sharp corners visible along the strand. In order to avoid this without adding more particles, cubic hermite interpolation is performed along the strand, increasing the number of vertices in the hair by a factor decided by the host application. Cubic hermite interpolation was chosen over other methods such as Bézier as the resulting curve is guaranteed to pass through the original control points. While this may not appear to be a major concern, without it the rendered strand could end up clipping through geometry which it should not, such as the character model. The following GLSL function shows the implementation of the cubic hermite interpolator.

```
vec3 cubicHermite(vec3 A, vec3 B, vec3 C, vec3 D, float t)
{
    vec3 a = -0.5f * A + 1.5f * B - 1.5f * C + 0.5f * D;
    vec3 b = A - 2.5f * B + 2.0f * C - 0.5f * D;
    vec3 c = -0.5f * A + C * 0.5f;
    vec3 d = B;

    return a * t * t * t + b * t * t + c * t + d;
}
```

In addition to interpolating along strands, the keyhairs are interpolated between. This leads to a greater number of rendered hairs without the additional cost of simulating them. Interpolation is performed across a triangle by using randomly generated barycentric coordinates as their root locations and interpolation factors. These random values are seeded using the keyhair indices, ensuring the interpolated root locations do not shift between iterations.

Interpolation is handled by the tessellation control shader and tessellation evaluation shader. The TCS generates and sets the parameters for interpolation to take place in the TES, including the barycentric coordinates, which are passed to the TES as a `patch` output. The TES then performs the interpolation

in both dimensions. The reason for passing this data as a patch output is to ensure that the randomly generated coordinates are the same for each TES invocation which works on the same hairs. Without this, multiple segments of the same strand would appear in different positions in space.

In order to obtain the vertices for rendering, an indexed draw call is issued for `GL_PATCHES` with an input and output patch size of three. The vertex shader simply passes these indices to the TCS, which fetches the data from the position SSBO and passes them to the TES as a `patch` output. It would seem more logical to instead store the indices of every vertex in a strand, then simply have the vertex shader fetch a single position using `gl_VertexID` to pass to the TCS with a larger input patch size. Unfortunately, this does not work in practise on a variety of hardware, where larger patch sizes which are valid in that they are below the maximum size cause driver crashes.

The implementation of interpolation remains relatively unchanged between `gpu` and `gpu_optimised`. The major exception is where the majority of calculations take place. In `gpu`, most of the expensive calculations such as lighting are performed in the TCS, the intuition being that this will prevent the re-computation of values in the TES for use in interpolation. However, it actually turned out to be better if the lighting calculations were moved to the TES instead, with the TCS only setting parameters and fetching data. The reason behind this is likely due to the number of active shader invocations. Due to the limitations discussed in the previous paragraph, there are only three TCS invocations for each patch, and as such each invocation would need to perform the lighting calculations for every particle in the strand sequentially. On the other hand, the TES has a separate invocation for each vertex and due to increased level of parallelism performs faster. It comes down to the fact that the lighting calculations performed, while not trivial, are sufficiently simple to allow for faster performance when spread over multiple invocations. It is worth noting however that for more complex operations this may not be the case.

4.3.2 Geometry Creation

Modern graphics hardware is optimised for the rendering of triangles, but the interpolated hairs are provided as line strips. As such, it is necessary to expand these lines into triangles. The triangles are orientated such that they are always facing the camera, similar to the billboarding technique often used for rendering far-away objects in video games. The geometry shader is passed two vertices of the line, allowing for the calculation of the vector between the two. By using the camera's `front` vector[1], it is possible to obtain the perpendicular vector to the two by using the cross product.

Using this perpendicular offset vector it is possible to emit four vertices as a triangle strip which represents a rectangle of equal length to the original isoline, with a width set as a uniform or constant. By allowing the changing of the width variable, it would be trivial to implement simple distance-dependent level of detail in combination with interpolation factors - the further the hair is from the camera, the thicker the strand is drawn and the fewer interpolated strands are generated.

If the isoline segments are at vastly different angles, using this method cracks become visible. The alternative would be to create seamless links between the segments by taking into account the tangent of the curve at each particle. This is however not implemented as the additional computational cost is not worth it for the visual improvement. Due to the density of the hairs, where cracks are present they are often not visible as the gaps are filled by similarly coloured hairs behind those in front. Additionally, the only cases when the simulated hair would have large cracks is when it's moving at higher speeds, and as such it will not be visible for more than a few frames anyway.

4.3.3 Local Shading Model

The local shading model is concerned with the lighting of each hair strand individually, and for the implementation a combination of Kajiya-Kay and Marschner was used. The first aspect of the lighting model to consider is diffuse, and even now the most commonly used method to compute the diffuse lighting along a strand of hair is to use the method proposed in [19]. The reason for its prevalence is simply that it is good enough and provides suitable results, and as such there is little reason to improve upon it.

That's not to say it hasn't been attempted, as evidenced by the use of the implicit surface normal in [31] for better shading along the extremes of the volume. While this would produce better results, it is not used in this implementation due to the additional computation required to compute the surface

[1] A camera's `front` vector is the direction in which the camera is facing.

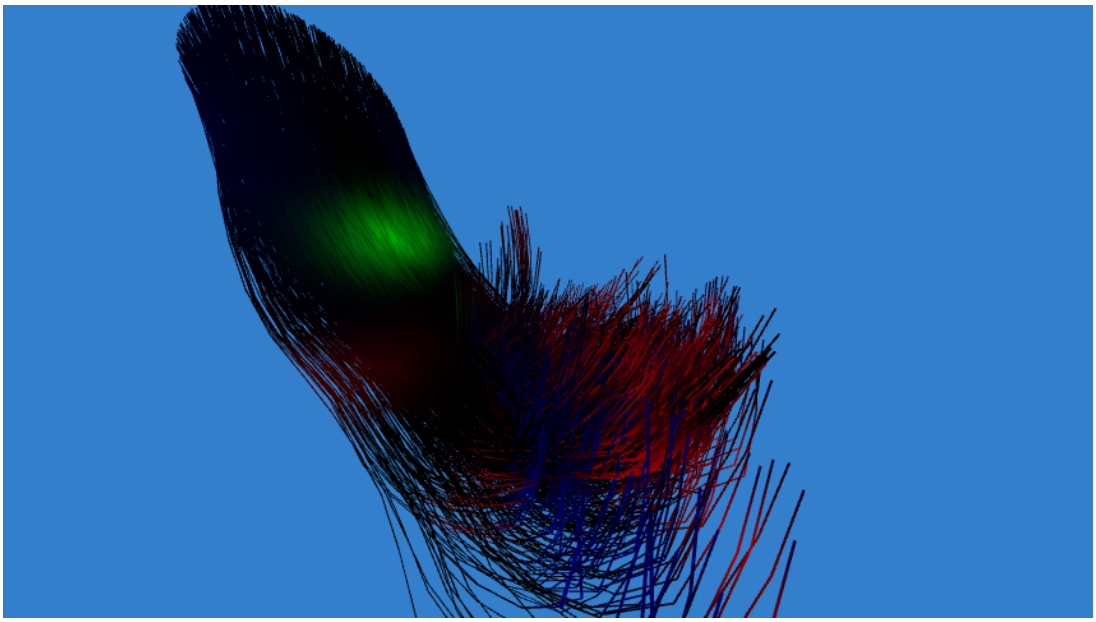


Figure 4.2: In the image above, the hairs are rendered without any diffuse lighting or shadows, and instead only use the local shading model. For the puposes of visualisation, each component of the Marschner model has been assigned a different colour, with R being red, TT being green, and TRT blue.

and then its normals for use in the lighting calculations. As such, Kajiya-Kay was used, but only for the diffuse component as the specular calculation has been greatly improved upon since [19].

For the specular component, the model proposed by Marschner et al. was used in two capacities. The advantage of this model is that it allows for the three primary scattering contributions which are not accounted for in [19]. A visualisation of the separate components of the model used in the final implementation can be seen in figure 4.2. For the first attempt, the shading model was precomputed at full detail and stored in lookup textures, similar to the method used in [28]. The advantage of this was that the lighting is completely correct (according to the proposed model) as there is no need to approximate due to the removal of the real-time constraint. In contrast to [28], three textures were used to allow for individual TT and TRT components to be stored. This means that at runtime three texture reads were required to calculate the lighting at a vertex.

However, this meant that the values would need to be recomputed for every different hair colour as a large number of properties controlling the colour are embedded into the lookup tables. To avoid this, another model was implemented where the approximations proposed in [36] were used, which allows for the setting of colours independently of other properties. While this is no longer a physically correct model, the differences in results are negligible when appropriate parameters are chosen. Additionally, the easier to use artistic controls are also advantageous, making changes to each component of the model possible without having knock-on effects elsewhere.

This approximation model was not precomputed, but instead performed at runtime. This was in fact more efficient than the texture lookup method, most likely due to the simplicity of the calculations allowing for computations to take place faster than a memory transaction. This is highly dependent on the clock speed of the GPU versus the memory latency, and as such on other hardware it may be more efficient to precompute. Precomputing the approximation model in [36] removes some of its flexibility at runtime, but as intensities and colour are not required for the lookup table this can still be altered at runtime.

The lighting calculations were performed in the tessellation stage of the rendering pipeline, as opposed to the fragment shader as is more traditional for high quality lighting. However, due to the fine width of the hair strands and frequency of the vertices, linearly interpolating the lighting values between vertices gives visually near identical results to performing the calculation in the fragment shader at a fraction of the cost.

The diffuse and specular contributions are summed and multiplied by the light's colour and intensity, with an optional ambient component should it be required. This combined value gives the total shading for a given point on an individual hair strand.

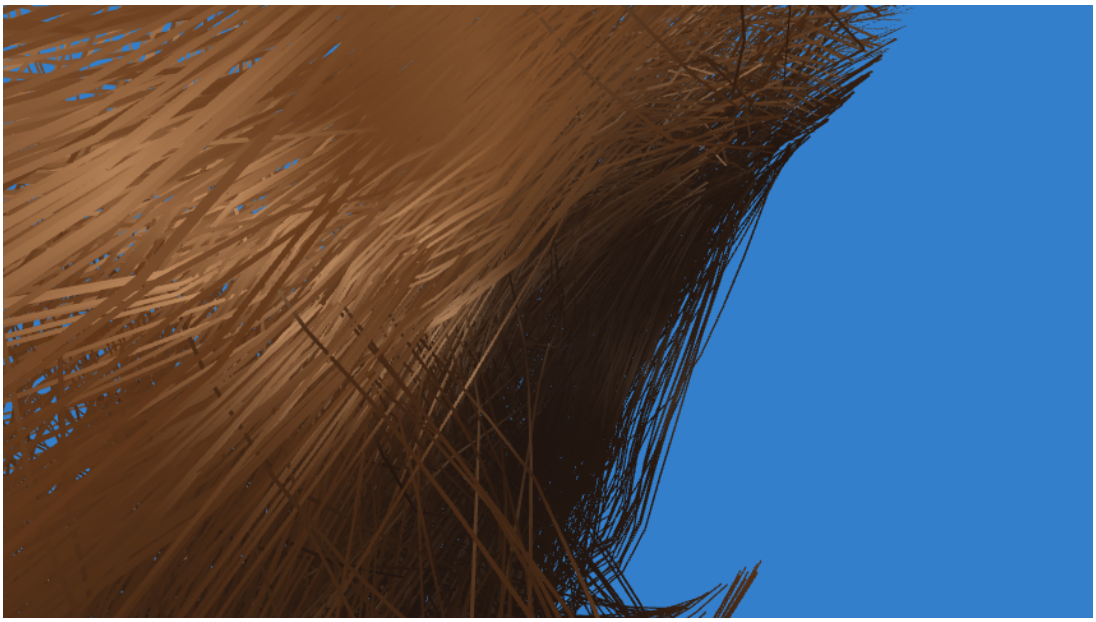


Figure 4.3: This figure provides a detailed view of the self-shadowing implementation used in `optimised_gpu`. As the light source is coming from the left, the strands to the right are being partially occluded and as such appear darker. Note that this is not a binary yes or no decision with respect to whether a fragment is in shadow, but instead its intensity is dictated by the density of the hair volume at the same location.

4.3.4 Shadows

With only the local model, hair appears to look overly bright and unnatural. This is remedied through the addition of a global shading model, which in this case is the casting of shadows between strands in the hair volume. The initial plan was to use opacity shadow maps, as proposed in [21] using the same techniques as described in [28]. However, since those papers were published the much improved deep opacity maps [43] was introduced, and as such was used in favour due to its better performance. The deep opacity maps were implemented as described in the aforementioned paper, packing all layers and initial depth z_0 into a single two-dimensional texture with four channels.

Rendering shadows using this method requires two passes. In the first path, the depth of the scene is rendered off-screen from the point of view of the light source using an FBO with `GL_DEPTH_ATTACHMENT` only. After this is complete, the scene is again rendered from the point of view of the light, but this time the fragment shader is used to calcualte the contributions to each layer. With the second pass complete, the whole opacity map is available as a two-dimensional texture which can be accessed in another shader using a `sampler2D`. Of course, one more additional pass will be required to render the scene from the camera. In this pass, points are also transformed by the view and projection matrices of the light source, allowing for the lookup of the current fragment in the opacity map. This method of shadowing was used in the `gpu` implementation.

While this did provide sufficient performance for real-time applications, by reusing the density calculated earlier during the simulation stage it is possible to achieve better performance. As was discussed in section 4.2.4, the density is rendered to a three-dimensional texture, which is still available at the time of rendering and as such can be used for calculating opacity.

The intuition in this new method is to perform ray casting from the light source to the current fragment's position in voxel space. As the density volume is accessed as a `sampler3D` through the overloaded `texture` function, it expects lookup coordinates in a normalised u, v, w format, which in this context is equivalent to the normalised voxel space. To remove the need to recalculate the value in the shader, the world position of the light is transformed and normalised by the host application into voxel space before being passed as a uniform. In the TES, the generated vertex position after interpolation is also transformed to normalised voxel space, and the difference vector between them is calculated. A ray is then cast from the light to the vertex, with samples taken at defined intervals.

The light intensity is initialised to a value of `1.0f` prior to sampling. At each sample point, the density is read from the volume texture's alpha channel. The intensity of the light is multiplicatively

updated according to the density at each point multiplied by an absorption constant and scale factor to account for the fact the density is not normalised over the whole volume. This is shown in the listing below. Note that the value is clamped between zero and one to prevent any unnatural visual artifacts occurring. Additionally, the calculated intensity is also multiplied by the artist-defined intensity of the light to allow for additional control.

```
float intensity = 1.0f;
vec3 sample_position = light_position;

// ...

for (uint i = 0; i < light_samples; ++i)
{
    float density = texture(volume, sample_position).a *
        density_scale_factor;
    intensity *= clamp(light_absorption /
        (1.0f + density), 0.0f, 1.0f);
    sample_position += sample_step;
}

intensity *= light_intensity;
```

Under normal circumstances it would be reasonable to disregard this method as slow, as it requires the calculation of the entire density volume. However, using our already proposed simulation method the density is already available as a three-dimensional texture. This removes any associated generation cost from the rendering component, which can therefore be performed in a single pass. The fact the density is stored as a texture is also beneficial at this stage, as it allows for built-in interpolation and mipmapping should it be required.

Of course, the biggest disadvantage with this method is that the quality of shadowing is directly linked to the resolution of the voxel space. However, even without a particularly high resolution reasonable results can be obtained, as can be seen in figure 4.3. It's also important to note that this resolution limitation is present with all shadowing methods, as all those previously described require the rendering of the scene off-screen to an FBO, which itself has a set resolution.

This method proved to be highly efficient, as it requires little computation in the rendering stage. The intensity calculation is performed in the TES with the local shading model. By multiplying the local value by the intensity the final shading value can be obtained, although ambient can be added if required. Due to its impressive speed and results, this method was used in the `optimised_gpu` version of the implementation.

Chapter 5

Results and Evaluation

This chapter serves to provide a detailed analysis and evaluation of the proposed models and their implementations. The project is evaluated with respect to three main points: performance, quality, and completeness. Performance refers to the execution time of the implementation, quality refers to the standard of the proposed model and how well it compares with other implementations, and completeness is concerned with how well the model proposed conforms with the specification.

5.1 Performance

The performance measure used in this case is the amount of time required for each component of the implementation to complete, the timings of which include the cost of synchronisation where appropriate. While this may increase the resulting times by a small fraction, the synchronisation is required to ensure the times recorded accurately reflect the duration of the entire computation.

In addition to the results for the two implementations of the proposed method, the TressFX demo is also provided as an industry-standard comparison. While it was originally planned to also have some NVIDIA demos, these were unfortunately unsupported by the available hardware and as such are not present. In order to remain fair, transparency was disabled for TressFX to avoid adding any additional overhead.

Tests were performed on two graphics cards, the AMD Radeon HD 6670 and the AMD Radeon HD 7520G. The HD 6670 is an entry-level dedicated graphics card for use in desktops, while the HD 7520G is an entry- to low mid-range laptop GPU. It is worth emphasising that these are both low-level graphics cards, which is the reason behind the relatively large numbers seen in the results, including the industry standard demo.

The implementations are operating with eight particles per strand, which is smoothed to sixteen during rendering. With 242 simulated strands, there is a total of 1936 particles requiring 46464 bytes of memory for the simulation state in `optimised_gpu` and 123904 bytes in `gpu`. In addition, the storage of voxels requires 250000 bytes for both implementations, leading to a total of 296464 bytes for `optimised_gpu` and 373904 bytes for `gpu`. As such, `gpu` requires approximately 1.25 times the amount of memory when compared to `optimised_gpu`.

Method	**Simulation/ms**		**Rendering/ms**		**Total/ms**	
	HD 7520G	**HD 6670**	**HD 7520G**	**HD 6670**	**HD 7520G**	**HD 6670**
TressFX 2.0	17.15	9.21	28.20	11.05	45.35	20.26
`gpu`	6.84	4.94	38.46	15.57	45.30	20.52
`optimised_gpu`	0.93	0.65	16.55	8.83	17.49	9.47

Table 5.1: A comparison between the two implementation versions against AMD TressFX 2.0. Times were measured with transparency disabled and anti-aliasing enabled.

The first measurement considered is the total time the simulation and rendering requires, presented in table 5.1. The total is divided between the cost of simulation, which in the case of the presented implementations is everything prior to interpolation, while rendering covers everything else. First we consider simulation. The most notable point here is that TressFX performs considerably worse than even

the naive implementation. It is also evident that the TressFX simulation does not perform as well on less powerful hardware, almost doubling in computation time. It is possible that the reason for this is due to the amount of memory accesses that take place. The greatest degradation to performance in GPU programming is caused by memory latency, making this a likely contender. This is further evidenced by the poorer performance on the HD 7520, which shares main memory with the CPU. As such, as more memory transactions are performed by TressFX due to a denser simulation model, the execution time is greatly amplified.

The difference between the optimised and naive versions of the implementation are also significant, with the optimised performing approximately seven times faster on both graphics cards. There are a number of likely causes for this, the first of which is the memory layout and coalesced transactions. Memory accesses are performed in smaller chunks in the naive implementation, which is fixed in the optimised version with a different layout in memory more suited to a GPU's execution model. The compute shaders dispatched in the optimised implementation are also launched with many more invocations per work group, leading to a higher number of parallel invocations and hence greater performance.

The combination of the majority of the computer shaders into a single shader once optimised is also of note. As only one shader is required, the temporary values required between stages can be stored in private memory as opposed to global. Private memory is much faster, and as such this is preferable. A single synchronisation is also required, as opposed to multiple memory barriers between shader dispatches, further improving performance. On top of this, the implementation of the optimised shader is simply superior to its predecessor, taking a greater advantage of the parallelism provided by the hardware.

When considering rendering, the first thing that one is likely to notice is how much more expensive it is than simulation, taking from four to almost twenty times longer. This is due to a variety of reasons, the first of which is that more is performed at the rendering stage. When considering that for each particle multidimensional interpolation and complex lighting and shadowing calculations are performed, it is not surprising that it takes longer. In addition, there is a considerable amount of work going on behind the scenes that is hidden by the OpenGL driver, including complex processes such as rasterisation. While on high-performance graphics hardware this is not as significant, it can take a significant toll on lower-end devices. An example of this is the low pixel rate[1] of the cards used for testing. This means that the GPU cannot work with a large amount of pixels at any given point, creating a bottleneck that is beyond the developer's control. Anti-aliasing is also well known for requiring large pixel rates for optimal performance, as working with more pixels requires higher rates to maintain performance [37].

The amount of memory reads that take place before interpolation is also considerable. For each root vertex, the entire strand of hair needs to be read from memory. While this is laid out optimally for coalesced reads during simulation, this is no longer the case when rendering as the order in which strands are accessed is no longer under strict control due to the use of indexed rendering. It is possible that by sorting the indices of the strand roots in the element buffer prior to the draw call that performance could be increased due to the higher potential for coalesced reads, but further research is required to support this hypothesis.

Another difference between `gpu` and `optimised_gpu` is where lighting calculations are performed. In the former, all calculations are performed in the tessellation control shader, with their results used for interpolation in the tessellation evaluation shader. Meanwhile, in `optimised_gpu` the lighting calculations are all performed directly in the TES. It would seem to be counter intutive that performing more calculations would lead to greater performance, but the reason behind this is grounded in the number of shader invocations. Using the proposed model, the TCS is only launched with three invocations for a single patch, with the reasons for this discussed previously in section 4.3.1. However, the TES has an invocation launched for each generated vertex of the abstract patch. This means that more operations are taking place in parallel, whereas in `gpu` the results are obtained in serial along a hair strand.

With that being said, the effectiveness of this optimisation is likely linked to the clock speed of the graphics hardware used for execution. On more modern GPUs with a faster clock speed, performing the calculations in the TCS may prove to be more efficient, as the cost of recalculating the results in parallel becomes more expensive than doing so once in serial. Another aspect effecting this trade off is the number of particles in a hair strand. The longer a hair strand, the longer the calculation would take to perform in serial and hence the more preferable performing the calculation in the TES becomes. As such, the problem becomes a balancing act, trading off between the degree of parallelism against core clock speed. Further research into this area would be required to support this, with access to a wider range of hardware for testing.

The times listed in table 5.1 can be further subdivided into the underlying components. While this

[1] Maximum number of pixels writeable to memory per second.

is possible for the compute shaders, the draw calls are still timed as a single event due to the profiling method used. To time the execution of a GPU event, the OpenGL query `GL_TIME_ELAPSED` is used. However, this considers a shader program to be a single event, and as such the draw call remains a single chunk. Despite this, an analysis of the differences in performance are provided. These more detailed results can be found in table 5.2.

Component	`gpu`/ms		`optimised_gpu`/ms	
	HD 7520G	**HD 6670**	**HD 7520G**	**HD 6670**
Strand Dynamics	1.04	0.43	0.12	0.12
Clear Voxels	0.47	0.15	0.31	0.28
Voxel Calculations	5.28	4.36	0.40	0.22
Draw	38.46 / 26.27	15.57 / 10.58	16.55 / 8.16	8.83 / 4.38
Total	45.30 / 33.07	20.52 / 15.52	17.49 / 8.98	9.47 / 5.00

Table 5.2: A breakdown of the different components of the simulation and rendering. Draw is split into two sections, the first with anti-aliasing enabled, the second without. Times are given in milliseconds.

The strand dynamics measurement is the time taken for the update of all strands of hair when considered individually. That is to say, no hair to hair interactions are considered at this stage of the simulation. When considering `gpu`, it's evident that more powerful hardware results in a better execution time. Memory accesses are again a strong contender for this disparity due to the lack of dedicated memory on the HD 7520G, as well as the other key differences between the simulation methods discussed earlier in this chapter. The solving of constraints in the naive implementation is also dependent on the operating frequency of the hardware, as due to the true data dependency between particles it is performed in a serial loop. The faster speed of the HD 6670 is therefore another reason for its improved performance.

The optimised implementation appears to be much more efficient, performing the complete computation at least three times as fast as its counterpart. The coalesced memory reads, combination of simulation stages into a single shader (and hence less memory transactions), and increased number of invocations per work group all contribute heavily to this, as has been addressed previously in the chapter. Another key factor in the increased speed is the use of shared memory. By having each shader invocation first load the particle position into a shared memory buffer accessible by all invocations, the amount of time required for a memory access is significantly reduced, the reasons for which have been previously explained in detail in section 3.1.1. This reduces the amount of time required when performing the constraint solving calculations, as the overhead for accessing neighbouring particles is reduced. Another benefit of using shared memory is that each particle is only read from global memory a single time, whereas in the `gpu` implementation the position of a particle could potentially be fetched from global memory every loop iteration when solving constraints, even if it had already been fetched for the previous iteration.

Another benefit of the optimised implementation is the removal of the serial handling of constraints through the odd-even solving method proposed in section 4.2.3. Through the introduction of this method, the solving of constraints along a hair strand is no longer dependent on the number of particles in the strand or directly tied to the operating frequency of the hardware.

Somewhat surprisingly, the `optimised_gpu` implementation runs at the same execution speed on both of the tested hardware configurations. As the same measurements were recorded on different hardware, it is probable that the total cost of the simulation is being limited by one or more factors. The most plausible bottleneck in this situation is the access to shared memory. When considering the computations that are performed during strand dynamics, they are largely simple and as such can be computed at a fraction of the speed of a memory access. As both graphics cards tested are using memory with similar frequencies and types (DDR3), this would explain the similar measurements as the memory transactions take up the majority of the recorded time, dominating the results. In order to fully support this claim, further testing would be required on GPUs with faster memory accesses to see whether a trend emerges as expected. The execution models are also similar for the two cards with respect to OpenGL support, with both having the same values for the maximum number of invocations per work group. This is also a possible reason for the measurements being so similar, as both cards only support the minimum required values by OpenGL due to them being low-range cards. This claim is supported by performing `glGet*` queries, yielding the same results for both configurations.

The next point of consideration is the clearing of voxels. These times are similar between both implementations, with the unoptimised version even performing faster on the HD 6670. In the `gpu`

implementation, the voxels are stored in an SSBO, and as such are cleared by the `glClearBufferSubData` function. The value passed to this function is replicated across all entries in the buffer, and as such passing a value of zero clears the data. In the `optimised_gpu` version, the voxels are stored in a three-dimensional texture as described in section 4.2.4. As the texture is attached to an FBO, the entire voxel space can be cleared through a `glClear` call for the colour buffer with a value of zero. The reason that this is slower on the HD 6670 is most likely due to the additional state modifications performed behind the scenes by the OpenGL driver when working with FBOs. However, the difference is small and the performance gains for the hair to hair interactions greatly outweigh this minor overhead.

The differences between the voxel calculations in the two implementations are arguably the most impressive part of the project, with `optimised_gpu` performing between ten and twenty times faster. The reason behind this is due to the use of a three-dimensional texture for storing data and making use of the built-in rendering pipeline to accelerate the computation. While the implementation is explained in detail in section 4.2.4, further explanation as to why this causes such a difference is provided. By making use of the built-in rendering pipeline, a number of advantages are made available. Firstly, by making use of `glPointSize` and `gl_Layer`, it is possible to automatically invoke the computation for the target voxel and all its neighbours without any additional computation. This is in stark contrast to the naive implementation, where it is the responsibility of the shader to find all the neighbours of the target voxel and iterate over them. This iteration is also performed in a single shader invocation, whereas in `optimised_gpu` each voxel is launched in its own invocation, further increasing performance.

It is also necessary in the density and velocity voxel calculations to perform summations of the calculated values for each voxel, allowing for an average velocity to be obtained. This was performed in `gpu` by converting values to 16.16 fixed-point and using the `atomicAdd` function to perform the summation across the parallel invocations. This process is made entirely implicit in the optimised implementation through the use of OpenGL's built-in blending capabilities. As the voxel calculations are performed in the fragment shader, by setting the appropriate blending mode all the individual values are automatically combined when written into the colour attachment of the FBO. This removes the need for converting to fixed-point, and shifts the responsibility for the summation to the OpenGL driver. As blending is an extremely common occurrence in rendering when working with transparency, this is a highly optimised operation and as such provides the excellent performance exhibited by the results.

Another benefit of storing the results in a texture is that when values are needed they can be accessed using a `sampler3D`. This means that all the benefits such as automatic interpolation and mipmapping built into texture samplers are provided by default, allowing for higher quality results even when working with low quality textures. This means the resolution of voxel grid can be decreased without a noticeable loss of visual quality, further improving voxel performance. The method OpenGL uses for storing textures in memory is also beneficial, as neighbouring texels are stored close together in memory to provide faster accesses.

Drawing efficiency is also increased in the optimised implementation, performing approximately two times as fast as the naive implementation. The interpolation and lighting aspects of this difference have already been discussed earlier in the chapter, and as such the focus here is on the different methods used for performing self-shadowing.

In `gpu`, shadowing information is calculated using deep opacity maps, described in section 2.2.6. This is one of the more advanced methods available for volume self-shadowing, providing real-time results in a variety of situations. This point is made to emphasise that deep opacity mapping is still an excellent method for self-shadowing in real-time applications. The reason that in this case the method does not perform particularly well is due to the number of render passes required. In order to obtain the opacity map used for calculating the shadows, two render passes are required, with the first generating a depth map and the second the opacity map. As such, this requires two additional framebuffer objects to which the depth and opacity maps are attached. The values stored in the maps are accessed via texture reads, meaning for a single fragment a minimum of two texture reads need to be performed. While on modern hardware this is not a major concern, this cost can be significant on lower-end hardware due to the lack of available resources. In total three passes are required, with the last pass rendering the final shadowed scene.

The method proposed in `optimised_gpu` achieves a similar effect at a fraction of the cost by using the density volume calculated during the voxel stage of simulation. Ray casting is performed through the volume from the light source to the position of the current fragment, sampling at specified intervals to calculate the final light intensity reaching the fragment. It's commonly known that ray casting is not an efficient method of rendering on modern graphics cards, as they are much more suited to the rasterisation model of rendering as used by OpenGL, and as such it may seem strange that this is the

faster method. However, the speed-up comes from the lack of any additional draw calls to generate shadowing information as the density volume has already been calculated and as such is accessible via a texture sampler during rendering. As such, the only extra work required when rendering is the texture lookups performed during ray casting to obtain the density values. By keeping the number of samples performed along a ray within a reasonable limit, the cost of the texture lookups is considerably less than that of the additional computations in `gpu` and as such the final draw time is significantly reduced. The one disadvantage of using this method is that the resolution of the shadowing information generated is limited by the resolution of the voxel grid. This is discussed in greater depth in section 5.2.

Each draw in the table has been further divided into two values, one representing the cost with anti-aliasing, and the other without. The anti-aliasing method used in the implementations described is the default `GL_MULTISAMPLE`, performing four samples per pixel. The main reason behind enabling anti-aliasing is to highlight that the proposed method does not explicitly require its own anti-aliasing method, and instead can be used with whichever method is used for the rest of the scene in the application making use of the hair while still producing natural looking results. Of course, there are arguments in which a dedicated method is preferred, as discussed in section 5.2. Additionally, it is noticeable how much the time required for rendering increases when using anti-aliasing, which is a direct result of the hardware being low-end. As such, using a dedicated method for anti-aliasing may enable lower-end machines to still have access to smoothed edges for the hair. The issue with many hair and fur anti-aliasing methods is that it causes a dithering effect, which while it does remove hard lines replaces them with a heavily fragmented edge which doesn't look good when the camera is close to the hair. As such, I feel that further research into the area would be beneficial, and may also provide a more efficient method for anti-aliasing in general.

5.2 Quality

In order to assess the quality of the implementation and hence proposed model, the best method is to compare the resulting motion and visuals to other existing implementations. For the purposes of this section, comparisons will be made between the provided implementations, AMD TressFX, and NVIDIA HairWorks. Both of these are considered to be the industry standard in real-time hair simulation for video games, a privilege currently limited to high-end hardware.

It may appear strange that the physical correctness of the model is not being evaluated, but the resulting motion and visuals being perceived as real is the only true concern. This is often the case in computer graphics, where short-cuts are taken to provide results that, while not technically correct, still look close enough to the truth for the difference to be negligible. This is the same reason behind not matching the number of particles to those in the industry implementations - if the same results can be achieved using fewer particles, the implementation has a clear advantage over other methods. As per the specification, a comparison is now made between the industry-standard implementations and the proposed method, followed by further analysis of the results. Stills of the hair while in motion can be seen in figure 5.1. In addition, the provided video `motion_normal.mp4` (also available online at `https://youtu.be/zIgm3mVr2mw`, visited 7/05/2017) shows a recording of the hair in motion.

For TressFX, we look at the implementation used in *Tomb Raider*, a video of which can be seen here: `https://youtu.be/nKYRkm9n0us` (visited 7/05/2017). The motion appears to be visually very similar to the provided implementation, with the biggest difference being the hair appears to be heavier. That is to say, when the hair is affected by a force the strands react in a more responsive manner than the method discussed in this report, giving a more realistic impression of weight. I suspect that the reason behind this is due to the velocity smoothing applied to mimic hair to hair interaction. In `optimised_gpu`, the voxel grid resolution is 25 in each dimension, and as such the voxels each take up a fair amount of volume when compared to the implementation proposed in the original paper [31]. Due to this, velocities can effectively be smoothed too much, leading to a less intense response to external forces.

There are two ways with which this can be fixed. The first and most obvious is to increase the number of voxels. This would lead to a finer grained and more detailed result, removing the potential for over-smoothing, and resulting in a more realistic motion. The more voxels per dimension, the more detailed the approximation will appear. Of course, this also leads to performance degradation as the amount of computations required increases. However, there is an approach to avoid this without the need for additional computation. By modifying the value of $s_{friction}$ used to control how much smoothing is applied, it is possible to reduce the effect of the hair to hair interactions on the hair particles. While this would help to solve the immediate problem, it still introduces issues in that this change will effect all motion, which could potentially lead to issues such as self-intersection as discussed in [31]. It may be

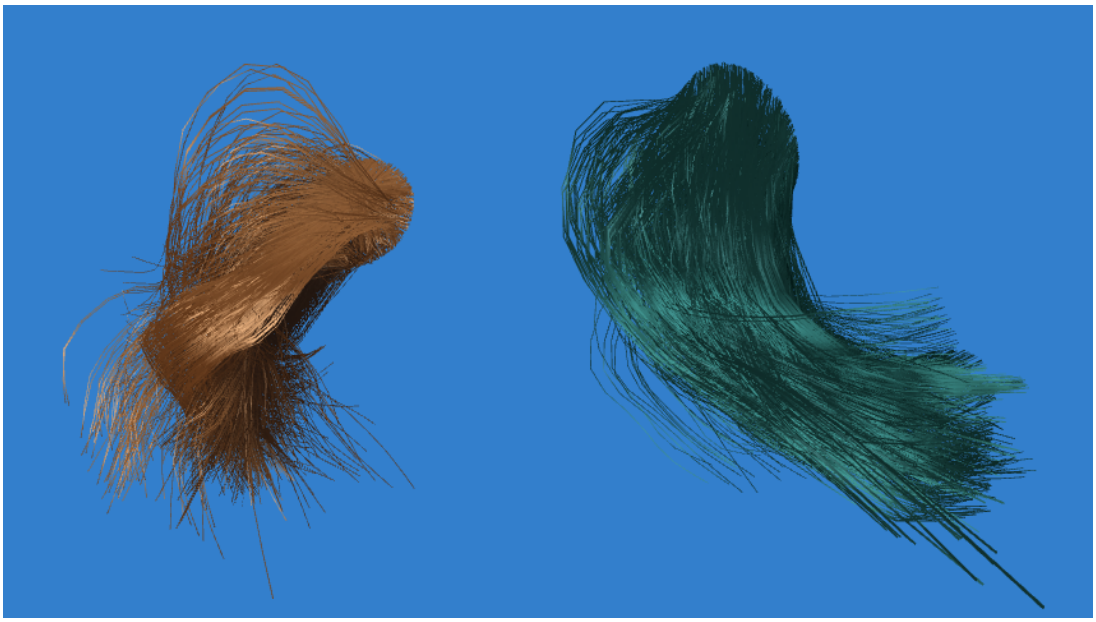


Figure 5.1: Hair volumes simulated using the final implementation while being moved at higher velocities.

possible to alter the magnitude of the friction depending on the strength of the force applied, or perhaps the impulse, but further research would need to be pursued to find an appropriate solution.

For HairWorks we consider its usage in *The Witcher 3: Wild Hunt*, which can be seen in action at `https://youtu.be/Md4Hmgtl8q0` (visited 7/05/2017). For the human characters, the hair motion achieved through the proposed model appears very similar, with no largely outstanding differences. However, its worth noting that in the example video the hair length is not as long as that used in TressFX or the implementation described in this report. As such, we also consider the *Nalu* Demo [5], as the character has long hair similar to that used in the proposed implementation. This motion is also very similar to the proposed implementation, with the exception being a much higher interpolation value is used along strands [28], leading to visually smoother motion. While this is possible for the proposed implementation to achieve, the interpolation factor was intentionally left lower to allow for it to perform well on low-end hardware. As this is a tunable value, the particle count and interpolation factor can both be increased with ease should the hardware allow it. The provided video `motion_no_gravity.mp4` (available online at `https://youtu.be/AhfnE9120nw`, visited 7/05/2017) shows the hair motion when gravity is disabled, leading to results more similar those that seen in the *Nalu* demo.

The area in which the NVIDIA implementation is superior is with respect to hair strands resisting bending, which is visible in both *Nalu* and the listed demonstration video. The points at which this is most obvious is when parts of the strands are close to the character's scalp in *Nalu*, and on the assortment of animals and monsters in *The Witcher 3*. This effect is not present in the proposed model due to a lack of a bending constraint as is addressed in section 6.1.

While in general the model provides a good simulation of motion, there are two areas which could use further improvement. The first of these is that when particles are put under very strong force which rapidly changes direction, some strands will begin to shake and jitter in an extremely unnatural and noticeable fashion. This in turn causes an increase in velocity due to the rapid movement of particles, prolonging the issue. The reason behind this is due to the Follow the Leader projection method used for solving constraints. As was explained in section 2.1.4, this method of projection works by finding the closest point on a sphere, centred at the previous particle in a strand, to the current particle's position, and by doing so enforces the length constraint. While this works well when the hair's motion is flowing and consistent, problems arise when strands are put under rapid changes of direction as the solution to the closest point shifts from one side of the sphere and back to the other again in quick succession due to small variations in position between updates. This propagates along the chain of particles, resulting in the jittering problem observed. The most obvious method to solve this problem would be to use another projection method, but the benefits of the DFTL method are significant and as such it would perhaps be better to put further research into the problem in an attempt to find a better solution.

The other situation in which the motion of the hair breaks immersion is when in some situations all the hair strands in the volume appear to move in the exact same manner, making the motion look strange

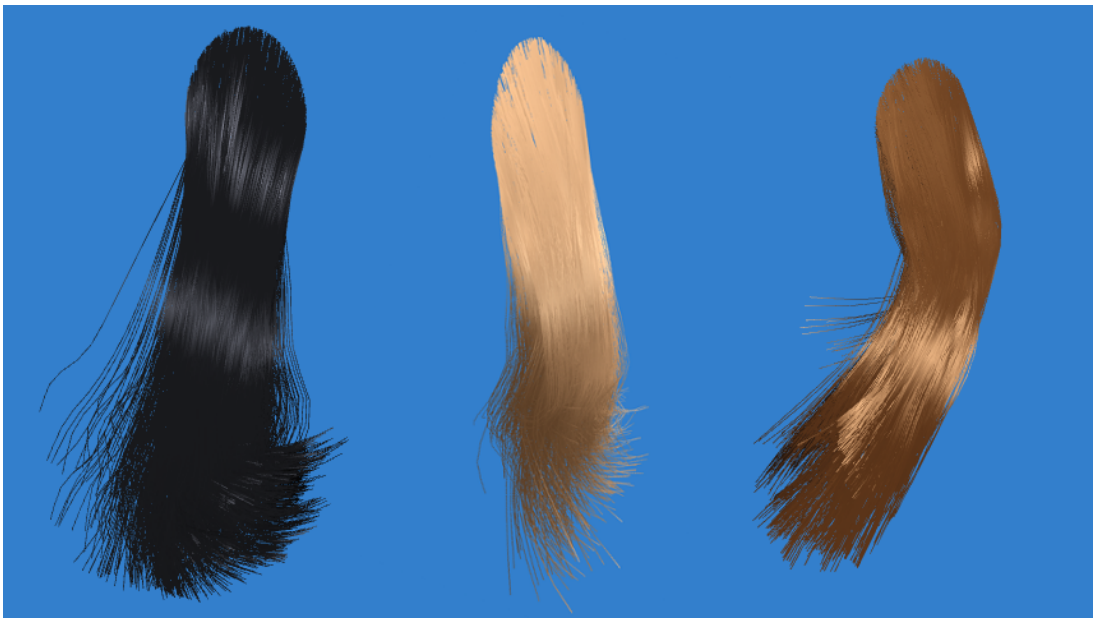


Figure 5.2: Results from the final implementation in a variety of positions and colours.

and unnatural. Although not obvious, it does make sense if considered - all particles are having the same force exerted on them during simulation, so with the exception of velocity correction from the voxels the motion would be expected to be the same. An easy method to combat this would be to provide a very slight random alteration to the velocity of each particle, resulting in motion that, while technically less correct, looks far more natural. Increasing the resolution of the voxel grid also causes a less uniform motion, as there are fewer particles assigned to each voxel. This means fewer particles are modified with the same value with respect to friction, and as such the resulting motion has a larger variance between particles.

In addition to simulation quality, the rendering quality is also of great importance as it dictates all of the visual properties of the hair volume. The proposed model's implementation can be seen in figure 5.2 and figure 5.3, with the latter providing a closer look at the rendered hair. The provided video `shading.mp4` (available online at `https://youtu.be/fG3jbfOJ9j4`, visitied 7/05/2017) shows the shading model in action. For comparison, we first consider TressFX. This was implemented using a similar lighting model to this implementation, as explained in section 2.1.5. In the demo video previously linked, the hair is often very dark due to the art style used throughout the game. As such, a better comparison is possible by considering the TressFX demo used in the previous section for performance measurements, which can be seen in figure 6.1. The appearance of the hair in the demo is very convincing, with the shading model producing realistic results.

This is also the case in the HairWorks demo, which produces highly realistic results when used in outdoor environments. One area in which the HairWorks implementation seems to look out of place is when under heavily tinted lighting, such as light from a fire or burning torch. In these situations, it seems as though the hair shading does not take the colour of the light source into account, causing the hair to often look out of place relative to the rest of the scene. In this respect, the hair in the *Nalu* demo appears to be more convincing, with the hair fitting more naturally into the environment due to it taking account of the colour of the light source. This is likely due to the fact that *Nalu* was produced as a stand-alone project, whereas HairWorks is for use in multiple situations, and as such is more finely tuned for its single use-case.

As such, all implementations use the same lighting model as a basis, allowing for a fair comparison to be made. The most obvious point when considering the implementations side-by-side is that the proposed method does in many cases appear to be much shinier - that is to say, the specular component seems much more prominent. While it would be reasonable to presume this is due to a problem with the implementation, this is in fact not the case. Instead, the issue is simply that I am not a digital artist, and as such selecting the appropriate colours is difficult for me to achieve. In fact, I would argue that the lighting system offers a greater range of colour controls, allowing for the creation of both more fictional and realistic hair volumes with ease. An example of this can be seen in figure 5.4, in which more exotic colours are used. However, as a result of the choice of colours, obtaining realistic looking results is much

Figure 5.3: A closer look at the shading model in the final implementation.

harder for someone such as myself without the appropriate training.

There are two main aspects of rendering that I would like to put more research into, the first of which is self-shadowing. Although I am very happy with the model proposed, as it performs extremely fast and produces sufficient results, the quality of the shadows produced is not always to a high standard. I hypothesise that this is due to the insufficient resolution in the voxel grid to allow for detailed shadows. Increasing the resolution of the grid would lead to more detailed shadows, but would also have the potential to introduce an array of performance issues associated with the additional computations and memory consumption. This is undesirable, and I feel this can be improved upon in other ways. One such potential solution could be to use a hybrid method of deep opacity maps and the current implementation, where the density of the voxel grid is increased in the x and y dimensions, but decreased in z. This would provide greater detail in the plane facing the light, allowing for more detailed shadowing, without the requirement of a large amount of extra memory due to the reduction in the resolution of the grid's depth. To avoid a loss of detail, the voxel calculations could be performed using the same method, although the influence on neighbouring voxels would need to be adjusted to account for the reduced number of layers. By using the multiple render target method proposed in [43] for many layers, it would potentially be possible to generate the opacity map and density volume using the same draw call. Of course, this is all hypothetical at this point, but I feel with further research into the area a breakthrough could definitely be made.

The addition of a custom anti-aliasing method could also be of use, as it would allow for greater control over the appearance of the hair. When using standard multisampling, the rendered edges are simply smoothed, but with a custom method I hypothesise greater performance could be achieved by dynamically controlling the number of samples. When considering hairs that are closer to the centre of the volume, in most cases the centre of the character's scalp, there are more strands overlapping. Because of this, the need for a large number of samples is not as important, as the hard edges are mostly obscured by the texture of the combined strands. On the other hand, strands that are on the edges of the volume are more likely to be drawn over the background scene, and as such require smoothing to avoid looking out of place. If it were possible to find a measure of how central a strand is inside the hair volume, it would consequently be possible to dynamically control the number of samples used for anti-aliasing. For example, it may be possible to use the density calculated for hair to hair interactions as an indicator of how many samples are required. This would reduce the load on the GPU and as such give improved performance. This is once again a hypothetical solution, and as such further research would be required to determine its feasibility.

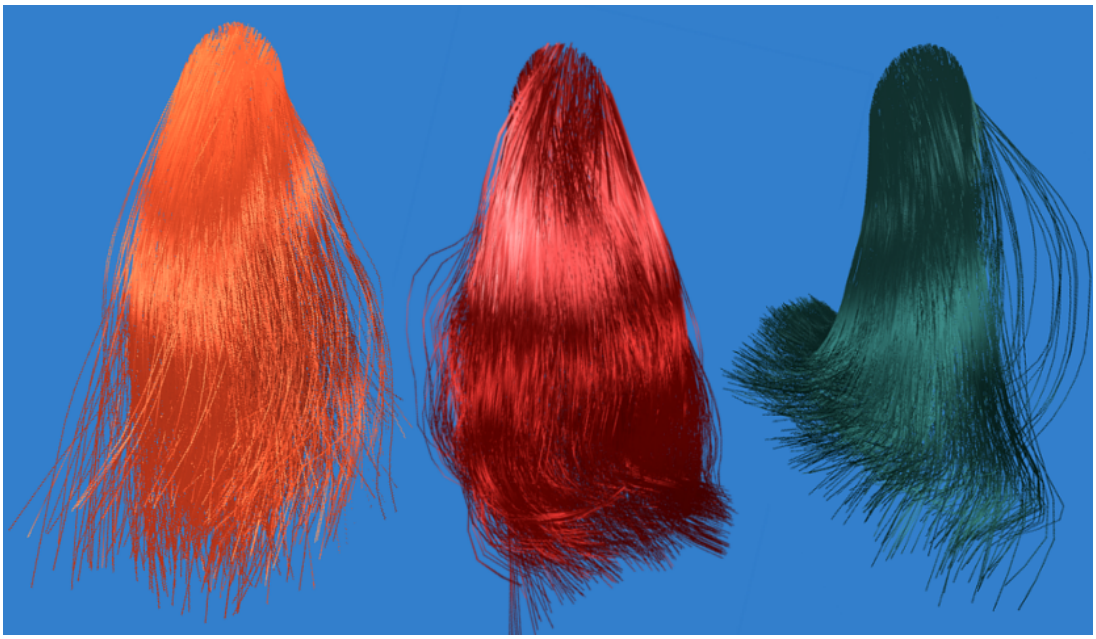


Figure 5.4: The final implementation when used with a less natural selection of colours.

5.3 Completeness

For the evaluation of completeness, how well the method and implementation conforms to the original specification is considered. The first point listed in the requirements was that the implementation is performed entirely on the GPU to avoid the overhead of needing to send the updated simulation state to device memory for every frame. As is made obvious by the previous content of the report, this requirement was fully realised, with all simulation and rendering performed directly on the GPU. The CPU generates the initial simulation state by procedurally growing hair strands along the vertex normals of a scalp mesh, but once this has been transferred to the graphics card the only communication performed is through the OpenGL functions to invoke compute and draw calls.

The next point of the specification was that real-time performance must be achieved, even on lower-end hardware. As is evidenced by the results in table 5.1 and table 5.2, both implementations operate at less than 30ms, with the single exception of `gpu` on the HD 7520G. As this version was completely unoptimised, this is not a major concern. In addition, with `optimised_gpu` the performance far exceeded real-time, with the entire simulation and rendering being performed in 5ms (200 frames per second) in the best case when operating on the HD 6670. This is a point worth emphasising, as the HD 6670 was considered to be entry-level when it was launched in 2010, and therefore achieving such performance is an impressive achievement.

The next point was realistic visuals. I believe the groundwork for this has been implemented to a high level, as the implementation provides clear controls for artists to fine tune the colours and material properties which would result in high quality lighting and convincing visuals. The model was a combination of [19] and [25], approximated using a similar method to [36] to allow for both improved performance and the introduction of intuitive controls for artists. In addition, self-shadowing was implemented using the newly proposed method of reusing the density volume, which is a strong starting point for further research, as well as providing shadows of sufficient quality for a variety of applications. I feel this point was definitely achieved, but my lack of experience as a digital artist leads to results that in some cases appear underwhelming.

Realistic motion was also achieved, producing comparable results to industry-standard applications at a fraction of the compute cost. The implementation of the interactions between strands is also impressive, with density and velocity calculations achieved through the use of a three-dimensional texture and the re-purposing of the rendering pipeline. This is a strong contribution, providing results in a fraction of the original times. The simulated motion looks convincing, although a few minor bugs occur when under rapid acceleration and deceleration.

5.4 Summary

It can as such be summarised that the proposed model and its implementations perform beyond expectations. The optimised implementation out-performs an industry-standard implementation while producing visually similar results, composed of believable motion and realsitic shading. The novel self-shadowing implementation proposed also performs well, producing shadows of sufficient quality at unparalleled speeds. Of course, that is not to say that the proposed methods are without fault, of which the potential remedies are discussed above. The next chapter goes on to cover potential extensions to the work proposed that would provide further enhancements to the end results.

Chapter 6

Future Work

While the implementation described in this report does support simulation and rendering of hair with self-shadowing, there are a variety of additional features that could be implemented.

6.1 Strand Model

The strand model used in the implementation, while sufficient for long strands of hair, does not appear as realistic for shorter strands. The main reason behind this is that there is no bending constraint between the hair segments, and because of this the implemented model is more similar to that of a rope segment.

There are a variety of different methods through which bending constraints can be introduced, with the most straight forward being the placing of constraints on the angle between two segments. By introducing a minimum and maximum angle, the hair will be prevented from flopping as it currently appears to in the implementation. The constraint could be made to be less strongly enforced, or have a larger range, the further along a strand the segment is situated to give the impression of the compounded mass. The tuning of these bending constraints could then be used by artists to enforce a certain stiffness. As position based dynamics is used, angle constraints should be easy enough to implement [27].

6.2 Styling Hair

Another point in which the implementation is lacking is the ability to style hair. While the proposed method is suitable for the simulation of long hair, it has no method through which shorter hair can be directed into a certain shape. The most obvious way to enforce this would be to record the particles' initial positions, and then throughout the simulation provide a restoring force towards them. This would give the impression of hair being loosely fixed in place depending on the strength multiplier on the restoring force. One issue with this method is that this can lead to particles being overly constrained, resulting in unnatural looking motion.

An alternative method was provided in [31] which allows for directing the hair volume through the use of the density grid. Instead of recording the original positions of the particles, the original density volume is stored. To direct the volume, a gradient is calculated between the current and original, or target, volume. For density volumes D_s and D_t representing the current and target densities respectively, the force $\mathbf{F}$ on a particle P can be found as follows.

$$
\begin{aligned}
E &= \frac{1}{2}\sum_{x,y,z}(D_s - D_t)^2 \\
\mathbf{F} &= \left(\frac{\partial E}{\partial P_x}, \frac{\partial E}{\partial P_y}, \frac{\partial E}{\partial P_z}\right) \\
&= \sum_{x,y,z}(D_s - D_t) \cdot \left(\frac{\partial D_s}{\partial P_x}, \frac{\partial D_s}{\partial P_y}, \frac{\partial D_s}{\partial P_z}\right)
\end{aligned}
$$

E represents the energy in the voxel grid. The resulting calculation for $\mathbf{F}$ can be further simplified as shown in [31]. It's worth noting that this method does not completely match the target shape, as is to be expected when working with the volume as opposed to individual strands.

The problem with this method is that the calculation would be very expensive, potentially too much so to perform in real-time. However, the most expensive part would likely be the amount of memory reads required, as the neighbours of each voxel will need to be sampled. However, by storing the volumes as three-dimensional textures it should be possible to take advantage of the hardware optimisations accessible through GLSL to achieve at the very least interactive perfromance. Additionally, it may not be necessary to perform the calculation every frame unless very strong external forces are acting on the hair particles. With that being said, to enforce a hair style the resolution of the voxel grid would need to be increased to allow for the finer details to be maintained, which may prove a problem with respect to performance.

If a styling method such as that outlined above was to be implemented, it may be possible to get by without the improved strand model described in section 6.1. The intuition behind this statement is that if the target density grid is calculated for a hair style, the particles will be directed into their desired positions regardless of the strand model, assuming constraints aren't violated. This assumes that the external forces acting on the particles are not extremely strong, which is a reasonable assumption to make in many real-time applications.

6.3 Level of Detail

Level of detail is an important aspect in real-time rendering, most notably when considering geometric complexity. There are a wide variety of level of detail algorithms for terrain such as CDLOD [39], while other geometry such as character models morph between meshes of different complexity. However, there have not been many well defined methods for level of detail in hair, most likely due to the lack of a commonly used solution to real-time hair rendering.

With the model proposed in this report, level of detail can be implemented in a variety of ways. Firstly, as discussed in section 4.3.2, reducing the number of interpolated hairs and increasing the width of those rendered is a simple but effective method. As such, the visual quality could be automatically selected based on the distance from the hair volume to the camera viewing the scene. As an example, this could be performed as a series of thresholds, where the distance being within a certain range maps to a specific number of hairs and strand width. Alternatively, this could be a smooth factor which blends between multiple values.

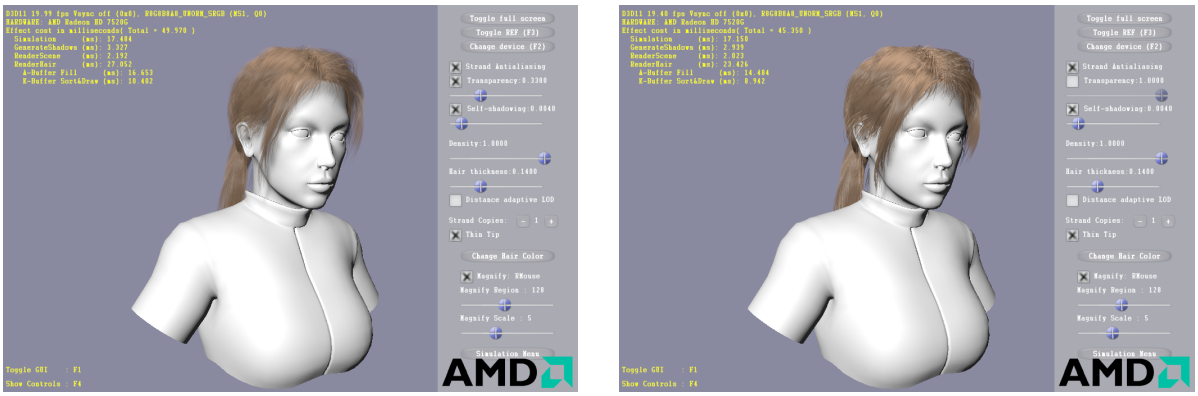
The quality of the simulation could also be modified based on the distance between the hair volume and camera. The first step in reducing simulation quality would be to increase the time step, reducing the frequency and hence number of simulation ticks that are performed. Note that it would be inadvisable to have this as a smooth value, and rather use a threshold, due to the instability that would be introduced by a variable time step [41]. When the camera is sufficiently far away, entire steps could be skipped in the simulation. The most obvious point is the hair to hair interactions, as this would be difficult to notice beyond a certain distance anyway. Note that as the density volume is used for shadowing calculations, a fall-back method would be required, or shadows could simply be ignored, when the hair volume is viewed beyond a certain range.

Additionally, the simulation could be disabled when the bounding box of the hair volume is entirely out of frame, similar to the process of frustum culling used in rendering. There's no sense in rendering hair that cannot be seen, and as such there is also no need to simulate it.

It's also worth noting that while these level of detail options were considered as interactive values changing depending on the position and orientation of the camera, they could also be set before simulation begins as options of the program. This would allow users to turn off features that their hardware is unable to efficiently support should it be required. In the worst case, when working with very old hardware, a fall-back static hair mesh could be provided.

6.4 Transparency and Blending

One of the larger setbacks with respect to the visual appearance of the rendered hair is how solid the strands appear. The way in which this would be fixed is through the introduction of some amount of alpha transparency, allowing for different strands to blend together, leading to a more realistic image. The reason adding transparency is beneficial to the appearance of the hair volume stems from the fact that the hair strands rendered after simulation are thicker than an actual strand of hair. Of course, this is intentional as it counteracts the smaller number of hairs which are being rendered when compared to those present on a real head of hair. By allowing the strands to blend together, the visual effect is that



(a) Transparency Enabled

(b) Transparency Disabled

Figure 6.1: A comparison between hair rendered with and without transparency, showing its importance to creating a more realistic image. Both images were captured in the TressFX 2.0 demo.

the distinctive lines between hairs are less visible, leading to a more textured seemingly more detailed result. On top of this, the hair volume appears to fit better into the surrounding environment as the strands on the edge will blend with the rest of the scene.

Once the alpha values have been set, blending is enabled in OpenGL as shown below.

```
1 glEnable(GL_BLEND);
2 glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
```

This enables blending between fragments, and sets the blending function such that it best matches real transparency. However, when working with hairs, this introduces problems as the strands are not depth sorted. Blending is implemented such that the blending function is applied to the fragment that is stored in the currently bound framebuffer and the new fragment located at the same offset. As such, it becomes apparent that the operation is not commutative, meaning fragments need to be sorted relative to depth before rendering is performed otherwise visual artifacts will occur. Typically this is not an issue, as the meshes can be sorted on the CPU, with draw calls then being issued in the appropriate order. This is not easy to achieve for two main reasons when using the proposed simulation method.

Firstly, the hair strands cannot be reordered in memory without causing issues. The particle positions in memory are assumed to remain constant to allow for efficient accesses and the solving of constraints during simulation. To allow for sorting, an additional buffer would be required, along with some mapping between the two to allow for re-indexing afterwards. This requires a lot of space, and more importantly memory transactions - a significant detriment to performance.

In addition, the data is all stored on the GPU. Sorting on the GPU is difficult to perform in an efficient manner, although performant solutions have been suggested [22]. However, this would also require additional buffer space and memory transactions, leading to further degraded performance.

A solution to this problem is through Order Independent Transparency, outlined in [9]. This not only solves the problem, but also provides up to ten times the performance of classical blending [3]. First, all opaque objects in the scene are rendered, providing a base framebuffer from which to work. The next render pass draws all objects with transparency, but instead of drawing to the framebuffer, the fragments are stored in the link buffer. The link buffer has an entry for each pixel of the framebuffer, where each entry contains a linked list of all the fragments which contribute to the fragment's final colour. Once this has been completed, a final render pass is invoked over all fragments using a screenspace quad as the primitive type. The elements in the linked lists are sorted in the fragment shader with respect to depth, and when combined with the background the final colour can be calculated and written to the framebuffer. Evidence of this method working in hair is its inclusion in AMD's TressFX 2.0 [3]. The importance of transparency is illustrated in figure 6.1.

6.5 Curly Hair

The hair simulated in the proposed implementation is straight by definition due to the underlying model used. This limits the types of hair style that can be implemented, and as a result curly hair cannot

currently be produced. However, there is a relatively simple method for adding support for curly hair as proposed in [26].

The intuition in the proposed method is that the strand dynamics simulation remains unchanged, and instead an alternative set of vertices are used for rendering. These vertices are generated by subdividing the strand segments into multiple vertices, and offsetting these along the particle normals. By linking the magnitude of the offset with a repeating function such as a sine wave, it is possible to generate hair curls procedurally at runtime.

However, this leads to the curls intersecting with each other, as the hair strands now appear to be too close together. As such, repulsion is introduced between strands, the magnitude and direction of which is calculated using the already computed density volume. The density grid is normalised, and its value at the particle's position, denoted by $D_{\mathbf{p}}$, is then used to update the particle velocity:

$$\mathbf{v}_i(t) = \mathbf{v}_i(t) + s_{repulsion} \frac{D_{\mathbf{p}}}{\Delta t}$$

The constant $s_{repulsion}$ can be used to control the amount of repulsion between hairs, and as such can be thought of as a measure of hair thickness for the simulation. By setting this constant to a value such that the intersections between curls are no longer visible, curly hair can be achieved with little additional code.

The biggest issue here is the normalisation of the density grid. This requires keeping track of the total density of the grid, which is not easy to obtain when working in GLSL without losing a significant amount of performance. If this were to be implemented efficiently, it would most likely be necessary to access the density grid through an alternative compute API such as OpenCL or CUDA, which allows for a wider range of approaches when dealing with this type of problem. The normalised grid would then be written back into an OpenGL texture object to be accessed via a sampler as before.

Chapter 7

Conclusion

In this dissertation I have proposed a model for the simulation and rendering of hair which is suitable for use in real-time applications, supporting realistic shading and self-shadowing. In addition, I have described a novel method through which self-shadowing can be achieved without requiring significant amounts of additional computation through the reuse of the density volume calculated for the evaluation of the interactions between hair strands. An efficient implementation of this new method has also been described, which makes use of three-dimensional textures to accelerate performance during both simulation and shadowing.

The real-time performance achieved by the project has exceeded expectations, outperforming industry-standard implementations by serveral orders of magnitude while producing visually similar results. Experimentation has also shown that the implementation performs at real-time rates even on low-end and outdated GPUs, leading to the expectation of far greater performance on high-end hardware. This opens up the potential for applications such as video games to provide a realistic approximation of hair that does not have a server detrimental impact on performance.

Bibliography

[1] Ken-ichi Anjyo, Yoshiaki Usami and Tsuneya Kurihara. *A Simple Method for Extracting the Natural Beauty of Hair*. In: *ACM SIGGRAPH Computer Graphics*. Vol. 26. 2. ACM. 1992, pp. 111–120.

[2] Grenville Armitage. *An Experimental Estimation of Latency Sensitivity in Multiplayer Quake 3*. In: *Networks, 2003. ICON2003. The 11th IEEE International Conference on*. IEEE. 2003, pp. 137–141.

[3] Bill Bilodeau and Dongsoo Han. *TressFX 2.0 and Beyond*. 2013. URL: `www.slideshare.net/DevCentralAMD/gs4147-billbilodeau` (visited on 12/04/2017).

[4] Joel Brown, Jean-Claude Latombe and Kevin Montgomery. *Real-Time Knot-Tying Simulation*. In: *The Visual Computer* 20.2 (2004), pp. 165–179.

[5] NVIDIA Corporation. *Nalu Demo*. 2004. URL: `www.nvidia.co.uk/coolstuff/demos#!/nalu` (visited on 12/04/2017).

[6] NVIDIA Corporation. *NVIDIA HairWorks*. 2014. URL: `developer.nvidia.com/hairworks` (visited on 12/04/2017).

[7] Microsoft Developer. *Direct Compute*. 2010. URL: `blogs.msdn.microsoft.com/chuckw/2010/07/14/directcompute/` (visited on 12/04/2017).

[8] Crystal Dynamics. *Tomb Raider*. Crystal Dynamics. 2013. URL: `www.crystald.com/projects/tomb-raider` (visited on 12/04/2017).

[9] Wolfgang Engel. *GPU Pro 2: Advanced Rendering Techniques*. In: A K Peters Ltd., 2011. Chap. VII GPGPU: 2. Order-Independent Transparency using Per-Pixel Linked Lists.

[10] Roy Featherstone. *Robot Dynamics Algorithms*. In: (1984).

[11] Glenn Fiedler. *Game Physics: Integratoin Basics*. 2006. URL: `www.gafferongames.com/game-physics/integration-basics/` (visited on 12/04/2017).

[12] 4A Games. *Metro Redux*. 4A Games. 2013. URL: `www.4a-games.com/metro-redux.html` (visited on 12/04/2017).

[13] Aristides Gionis, Piotr Indyk, Rajeev Motwani et al. *Similarity Search in High Dimensions via Hashing*. In: *VLDB*. Vol. 99. 6. 1999, pp. 518–529.

[14] Dominik Göddeke. *GPGPU: Basic Math Tutorial*. 2007. URL: `www.mathematik.uni-dortmund.de/~goeddeke/gpgpu/tutorial.html` (visited on 12/04/2017).

[15] Khronos Group. *Vulkan - Next Generation Graphics and Compute API*. 2016. URL: `www.khronos.org/vulkan/` (visited on 12/04/2017).

[16] RC Hoetzlein. *Fast Fixed-Radius Nearest Neighbors: Interactive Million-Particle Fluids*. In: *GPU Technology Conference*. 2014, p. 18.

[17] Hayley Iben et al. *Artistic Simulation of Curly Hair*. In: *Proceedings of the 12th ACM SIGGRAPH/Eurographics Symposium on Computer Animation*. ACM. 2013, pp. 63–71.

[18] Advanced Micro Devices Inc. *TressFX Hair, Real-Time Hair Physics System*. 2014. URL: `www.amd.com/en-us/innovations/software-technologies/technologies-gaming/tressfx` (visited on 12/04/2017).

[19] James T Kajiya and Timothy L Kay. *Rendering Fur with Three Dimensional Textures*. In: *ACM Siggraph Computer Graphics*. Vol. 23. 3. ACM. 1989, pp. 271–280.

[20] Anton Kaplanyan and Carsten Dachsbacher. *Cascaded Light Propagation Volumes for Real-Time Indirect Illumination*. In: *Proceedings of the 2010 ACM SIGGRAPH symposium on Interactive 3D Graphics and Games*. ACM. 2010, pp. 99–107.

[21] Tae-Yong Kim and Ulrich Neumann. *Opacity Shadow Maps*. In: *Rendering Techniques 2001*. Springer, 2001, pp. 177–182.

[22] Peter Kipfer and Rüdiger Westermann. *GPU Gems 2: Programming Techniques for High-performance Graphics and General-purpose Computation*. In: Addison Wesley Professional, 2005. Chap. 46. Improved GPU Sorting.

[23] Chuan Koon Koh and Zhiyong Huang. *A Simple Physics Model to Animate Human Hair Modeled in 2D Strips in Real Time*. In: *Computer Animation and Simulation 2001* (2001), pp. 127–138.

[24] Tom Lokovic and Eric Veach. *Deep Shadow Maps*. In: *Proceedings of the 27th annual conference on Computer graphics and interactive techniques*. ACM Press/Addison-Wesley Publishing Co. 2000, pp. 385–392.

[25] Stephen R Marschner et al. *Light Scattering from Human Hair Fibers*. In: *ACM Transactions on Graphics (TOG)*. Vol. 22. 3. ACM. 2003, pp. 780–791.

[26] Matthias Müller, Tae-Yong Kim and Nuttapong Chentanez. *Fast Simulation of Inextensible Hair and Fur*. In: *VRIPHYS* 12 (2012), pp. 39–44.

[27] Matthias Müller et al. *Position Based Dynamics*. In: *Journal of Visual Communication and Image Representation* 18.2 (2007), pp. 109–118.

[28] Hubert Nguyen and William Donnelly. *GPU Gems 2: Programming Techniques for High-performance Graphics and General-purpose Computation*. In: Addison-Wesley Professional, 2005. Chap. 23. Hair Animation and Rendering in the Nalu Demo.

[29] John Nickolls and William J Dally. *The GPU Computing Era*. In: *IEEE micro* 30.2 (2010).

[30] Lars Nyland, Mark Harris and Jan Prins. *GPU Gems 3: Programming Techniques for High-Performance Graphcis and General-Purpose Computation*. In: Addison-Wesley Upper Saddle River, NJ, 2007. Chap. 31. Fast N-Body Simulation with CUDA.

[31] Lena Petrovic, Mark Henne and John Anderson. *Volumetric Methods for Simulation and Rendering of Hair*. In: *Pixar Animation Studios* 2.4 (2005).

[32] Matt Pharr, Wenzel Jakob and Greg Humphreys. *Physically Based Rendering: From Theory to Implementation*. Morgan Kaufmann, 2016.

[33] Pomax. *A Primer on Bézier Curves*. 2011. URL: `pomax.github.io/bezierinfo/` (visited on 12/04/2017).

[34] CD PROJEKT RED. *The Witcher 3: Wild Hunt*. 2015. URL: `www.thewitcher.com/en/witcher3` (visited on 12/04/2017).

[35] Robert E Rosenblum, Wayne E Carlson and Edwin Tripp. *Simulating the Structure and Dynamics of Human Hair: Modelling, Rendering and Animation*. In: *Computer Animation and Virtual Worlds* 2.4 (1991), pp. 141–148.

[36] Iman Sadeghi et al. *An Artist Friendly Hair Shading System*. In: *ACM Transactions on Graphics (TOG)*. Vol. 29. 4. ACM. 2010, p. 56.

[37] Jens Schneider, Martin Kraus and Rüdiger Westermann. *GPU-Based Real-Time Discrete Euclidean Distance Transforms with Precise Error Bounds*. In: *VISAPP (1)*. 2009, pp. 435–442.

[38] Jason Stewart and Uriel Doyon. *Augmented Hair in Deus Ex Universe Projects: TressFX 3.0*. 2015. URL: `cdn.wccftech.com/wp-content/uploads/2015/03/Augmented-Hair-in-Deus-Ex-Universe-Projects-TressFX-3-0.pdf` (visited on 12/04/2017).

[39] Filip Strugar. *Continuous Distance-Dependent Level of Detail for Rendering Heightmaps*. In: *Journal of graphics, GPU, and game tools* 14.4 (2009), pp. 57–74.

[40] Loup Verlet. In: *Physical review* 159.1 (1967), p. 98.

[41] Kelly Ward et al. *A Survey on Hair Modeling: Styling, Simulation, and Rendering*. In: *IEEE Transactions on Visualization and Computer Graphics* 13.2 (2007).

[42] Patrick Wieschollek et al. *Efficient Large-Scale Approximate Nearest Neighbor Search on the GPU*. In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 2016, pp. 2027–2035.

[43] Cem Yuksel and John Keyser. *Deep Opacity Maps*. In: *Computer Graphics Forum*. Vol. 27. 2. Wiley Online Library. 2008, pp. 675–680.

[44] Cyril Zeller. *Cloth Simulation on the GPU*. NVIDIA Corporation. 2005. URL: `download.nvidia.com/developer/presentations/2005/SIGGRAPH/ClothSimulationOnTheGPU.pdf` (visited on 12/04/2017).

[45] Arno Zinke et al. *A Practical Approach for Photometric Acquisition of Hair Color*. In: *ACM Transactions on Graphics (TOG)*. Vol. 28. 5. ACM. 2009, p. 165.

[46] Intel Developer Zone. *OpenCL and OpenGL Interopability Tutorial*. 2014. URL: `software.intel.com/en-us/articles/opencl-and-opengl-interoperability-tutorial` (visited on 12/04/2017).