Real-Time Hair Simulation and Rendering with OpenCL and OpenGL

by

Mathieu Le Muzic

Dissertation

Presented to the

University of Dublin, Trinity College

in fulfillment

of the requirements

for the Degree of

Master of Science in Computer Science

University of Dublin, Trinity College

September 2012

Declaration

I, the undersigned, declare that this work has not previously been submitted as an exercise for a degree at this, or any other University, and that unless otherwise stated, is my own work.

Mathieu Le Muzic

August 25, 2012

Permission to Lend and/or Copy

I, the undersigned, agree that Trinity College Library may lend or copy this thesis upon request.

Mathieu Le Muzic

August 25, 2012

Acknowledgments

I wish to thank Rachel McDonnell for her input as a supervisor.

I would also like to thank Fabrizio Bellicano, John Dingliana, the Feldmann's, the Honohan's and Daniel Wilson for the help and support provided.

Mathieu Le Muzic

University of Dublin, Trinity College
September 2012

Real-Time Hair Simulation and Rendering with OpenCL and OpenGL

Mathieu Le Muzic
University of Dublin, Trinity College, 2012

Supervisor: Rachel McDonnell

In computer graphics, human hair simulation represents a challenging issue, and is still an active research subject nowadays. The problem comprises two complementary dimensions: the physical simulation and the rendering. While both aspects must be treated individually for each strand, they must also be treated globally due to interactions between hair strands. Because of the complexity of the hair, a large number of strands must be taken into account in order to achieve realistic results. In such conditions, processing may be difficult, especially in real-time. This is why most of interactive implementations now rely on GPU parallel computing, for performance gains. This project presents a real-time hair simulation application, which executes in parallel on the GPU using OpenCL for the physical simulation and OpenGL for the rendering.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

The goal of this dissertation is to explore the capabilities of using parallel computing on the GPU for real-time hair simulation and rendering. Whether applied to computer animation or computer games, hair simulation has always been challenging. It is also a very complete problem because it is related to physical simulation and rendering at the same time. Hair simulation was initially an offline procedure, which is why it became popular in offline computer animation first. After years of research and improvements, such a simulation is now able to run in real-time and has even been introduced in computer games. The real-time abilities of the recent implementations mostly rely on the use of parallel computing, just as in many other computer graphics areas.

Parallel computing on multi-core CPU's is a long established field. Parallel computing on GPU's, however, is quite recent, and was made famous by NVidia's CUDA framework in 2007. Compared to a CPU the GPU possesses much more compute units (Cores) and thus, is capable of handling much more simultaneous computations. A few years later, the Khronos Group released its first implementation of OpenCL. Even though the two frameworks mostly do the same thing, the CUDA framework remains more popular. Because it was released first, it is considered as more mature than OpenCL. The OpenCL framework, however, has the great advantage of being multi-platform, which means that the same program is able to run on different kinds of hardware.

In most of the recent materials we found on hair simulation, the implementation was done with CUDA or even with the Compute Shader, an analogous technology, feature of the Direct3D API and restricted to the windows platform only. We were surprised to see that no material is available on the subject of hair simulation with OpenCL. We were also curious to determine if this kind of simulation could be implemented with such a framework. Our strategy was to implement an interactive application on a personal computer demonstrating hair simulation and rendering. The demo has been implemented with OpenCL for the simulation and with OpenGL for the rendering. One could have also relied on OpenCL for implementing more complex rendering techniques but we wished to mostly focus on the physical simulation in this project.

The following chapters of this dissertation will describe our approach from the gathering of information to the development of the software. First, we will focus on the state-of-the-art, where we will review the most relevant techniques used in hair simulation and rendering. In the next chapter, we will briefly introduce the OpenCL framework in order to get a better understanding of the implementation part. We will then describe the design of the project in the fourth chapter. Chapter five will be dedicated to the implementation of our simulation. This will be the core of this document, we will explain all the different steps of the simulation as well as our modus operandi. In the sixth chapter, we will evaluate the results of the simulation. Finally, we will conclude and discuss future directions of the work in the last chapter of this paper.

Chapter 2

State-of-the-Art

In the field of human simulation hair represents one of the most challenging problems, and is still an active research subject nowadays. A very large number of academic references are available on the subject. For the sake of clarity, we will limit this overview to the most relevant methods, mainly those pertaining to real-time-capable approaches. Previous work in real-time hair simulation has been split into two distinct parts, the physical simulation and the rendering. The simulation controls the motion of the hair, while the rendering controls its visual appearance. This subject differs from other computer graphics problems because of the difficulty of simulating a large number of geometries at interactive rates. Many techniques were implemented over the last 20 years, making such simulation possible in real-time now. In this survey we will focus on these techniques in order to understand the state-of-the-art in hair simulation. We will give special attention to the key references that are closely related to this project. We will also review the progress made in graphics hardware, which helped improving the performance of real-time hair simulation.

2.1 Physical Simulation

In this first section, we will cover the relevant research in the domain of simulation. We will study the improvements made in this domain, in chronological order. Some reviews may be more brief than to others, due to the relative relevance of the technique. Because of the complexity of the simulation, the problem is commonly treated in

two different parts, individual and global. At first, the strands were usually simulated without taking into account the inter-hair interactions, mostly for performances reasons. Then, more advanced techniques appeared, allowing to take into account these interactions. In hair simulation there is always a duality between these two aspects and a good implementation must include fast and efficient ways to process them both. First, we will review the techniques related to the individual behavior strands. The ones related to the global motion of the hair will be covered afterwards.

2.1.1 Individual Hair Strands Dynamics

In the beginning of hair simulation, the behaviour of the full hair was limited to the individual dynamics of the strands. Each strand was simulated on its own, without having any interactions with the other strands. The main reason for this was mostly because of performance issues. Indeed in the late eighties the computers used for computer graphics were far less performant than the ones used nowadays. A decade after the attempts in hair simulation began, the first simulations with inter-hair interactions appeared, but the research on individual strand dynamics was still active. Even if we manage to get a coherent motion for the full hair it is still important to get an accurate behaviour for the individual strands.

Mass-Spring-Hinge System

The first step in simulating dynamics of hair strands was made by Rosenblum et al. [1] using constrained particle systems to simulate the strands, just like cloth simulation. The strands are represented as a linear series of masses, springs and hinges. The spring forces maintain the distance between two segments while the hinges allow the realistic bending of the strands. Since human hair is not very elastic, the spring forces have to be strong to prevent excessive stretching.

The simulation supports hair-head collision detection but for the sake of simplicity the strands simulated here do not interact with each other. Only the motion of individual hair strands is described. The head is simply represented as a bounding sphere and the collision response is made by applying repulsive forces to the nodes. Note that the system previously designed does not support different kinds of hairstyles, it only

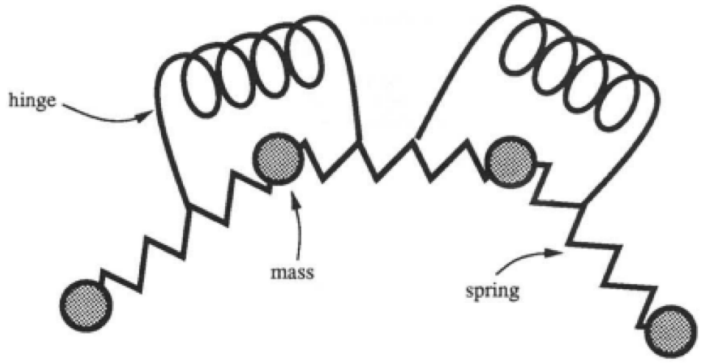


Figure 2.1: Schematic representation of the mass-spring-hinge system, Rosenblum et al. [1]

applies to straight hair. This method is fairly easy to implement; however, it cannot properly represent torsional motions of the strands. It can also cause instability because of the strong spring forces, which can lead to stiff equations. This instability, however, may be fixed by using a very small time step.

The mass-spring system was the first implementation of hair simulation and therefore has inspired a number of later projects. It has recently been reintroduced in hair simulation by Baraff and Witkin [2]. The implementation presented an integration system using implicit schemes in contrast with of one of Rosenblum et al. [1] which was using an explicit scheme. As stated above, this does not really suit interactive applications because it requires a very small time step. In contrast, implicit schemes are more difficult to implement but ensure stability in the system even when using large time steps. This is what makes the latter more preferable for interactive applications.

One-Dimensional Projective Equation

In 1992, Anjyo et al. [3] presented a new method to describe the dynamical behaviour of hair strands using differential equations. Again hair-hair interactions are not treated rigorously here, being too costly for the technologies employed at this time. Each hair strand is considered as a chain of segments and each hair segment is defined in a two-dimensional polar coordinate system (zenith and azimuth), and projected onto these two axes. The projection of the segments on each axis allows the integration

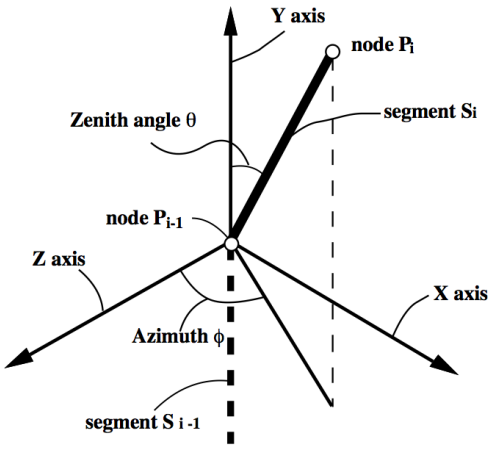


Figure 2.2: A hair strand segment in the polar coordinate system, Anjyo et al. [3]

of the forces using one-dimensional equations. On every time step of the simulation, the equations solve the new angle value for each axis after integration of the forces. The positions of the segments are then determined by these angles. The system also includes a control of the angle between two segments in order to recreate the bending properties of human hair. Once we obtain the new positions after the integration, the angle values may be readjusted based on the stiffness parameter defined for each joint.

This method has a few advantages compared to the mass-spring system. It is easy to implement, efficient, and does not cause any stretching. However, this method does not provide three-dimensional motion and thus cannot represent realistic hair strand behaviour such as torsion.

Serial Rigid Multi-Body Chain

To avoid the problems due to the one-dimensional projective equation methods, Hadap and Thalmann [4] introduced a new model for single hair strand dynamics, which was reused by Chang et al. [5] afterwards. This model uses forward kinematics and considers each strand as a serial chain of multiples rigid bodies. Unlike the previous techniques, this one does not require small time steps for the simulation and allows realistic three-dimensional motion.

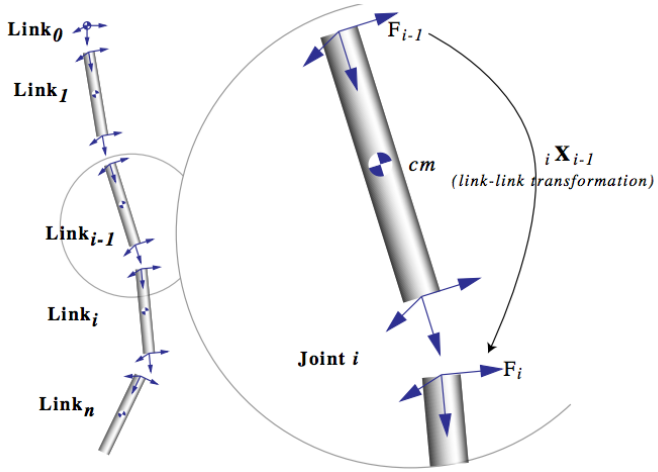


Figure 2.3: A hair strand as serial rigid multi-body chain, Hadap and Thalmann [4]

This model divides each strand into segments, and each segment is connected to the following one by a three degrees-of-freedom (DOF) spherical joint. Given a set of active forces (such as gravity, collisions, torsion and bending) the system solves the new positions by using a forward kinematics method called the Articulated-Body, well known in the domain of robotics. So far this has only been applied to straight hair.

Position-Based Dynamics

The common way to exert internal constraints in simulation is based on forces. For each time step, a set of internal forces (such as spring or hinge constraints) are accumulated in order to be applied to the elements of the simulation. These forces, according to Newton's second law of motion, are transformed into acceleration values and then integrated to determine the velocity and position values of the particles. So far, all the techniques in hair simulation were based on this method and research was focused on finding the most efficient ways to integrate these forces. This system - although easy to understand - may present some instability issues due to the force-based constraint response. Indeed, this system allows some shakiness, whereas a model that directly resolves constraints on the positions does not.

Although the position-based dynamics was introduced by Jakobsen [6] in 2001, Müller et al. [7] were the first to define a complete framework. The principle relies

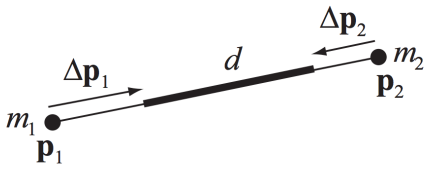


Figure 2.4: Projection of a distance constraint, Müller et al. [7]

on the Verlet integration and on the formulation of position-based constraints. In the Verlet integration the velocity is implicitly stored in the previous two frames positions. This allows the velocity to be automatically updated while manipulating the position values. The constraints are defined as functions that explicitly modify the positions in order to solve the equilibrium configuration. These position constraints replace the need of integrating internal forces, such as spring or hinge. The external forces, however, are still integrated in a normal way. In the case of hair simulation the main constraint would make sure that a given distance between two nodes is always respected. Such a constraint is called a distance constraint and is represented in Figure 2.7. There are other kinds of constraints that we may also apply to the simulation, like collision constraints or bending constraints. The use of position-based dynamics hugely simplifies the implementation of the simulation, compared to any of the previous methods. It also offers an unconditional stability to the system since it does not extrapolate blindly into the future time steps like traditionals explicit schemes would do.

In the meanwhile, some researchers have also applied the position-based dynamics principle to hair simulation like, Nguyen et al. [8] in 2005, or more recently Oshita [9] in 2007. Note that the simulation of complex hairstyles is also facilitated by the use of position-based dynamics. In 2010, Rungjiratananon et al. [10], focused their research on the simulation of curvy and curly hair based on this technique. Another advantage is that it can be easily parallelized. Tariq and Bavoil [11] demonstrated, in 2008, a hair simulation based on this method using parallel computing on the GPU with very impressive results. In a later section, we will see that the optimization of the simulation using parallel computing power is a very important aspect of the state-of-the-art in real-time hair simulation.

2.1.2 Full Hair Dynamics

The dynamics of a full hairstyle are more complex than those of a single hair, because the interactions between hair strands have to be taken into account. This part is usually the bottleneck of the simulation. These interactions, however, are essential for ensuring a realistic hair motion. It gives volume to the hair and avoids visual artifacts of over-compressed or over-stretched hair strips.

The human hair is composed of about 100 000 of hair strands, meaning that the more strands we have in the simulation the more accurate it becomes. Given the fact that each strand is usually divided into a dozen of segments, it becomes clear that it is impossible to achieve a simulation with as many strands at interactive rates. The graphics pipeline would simply be overloaded. For these reasons human hair can be represented in different ways in order to simulate a full hair and at decent frame rates. All the different representations of full hair dynamics will not be covered in this section, we will only focus on the ones we deem important for the implementation. One may for instance, represent hair as a continuous medium by using fluid dynamics to model hair interactions. One can also use large simulated hair strips to represent groups of strands. Finally, one can use a set of a few simulated guide strands which are then duplicated by interpolation. We will study these different representations in this section. The last model will turn out to be particularly interesting because it balances between the two previous ones; accordingly, it is also the most relevant one for our project.

Hair as a Continuous Medium

Hair strands that are close to each other tend to move in the same way. From this observation, one may represent the hair as an anisotropic continuous medium. Hadap and Thalmann [4] were the first to propose a model with hair-hair interactions, using fluid dynamics. The hair is considered as a set of discrete particles, using smoothed particles hydrodynamics (SPH) as numerical model. Even if this model is an approximation of hair-hair interactions, it computes much faster than handling hair-hair collisions individually.

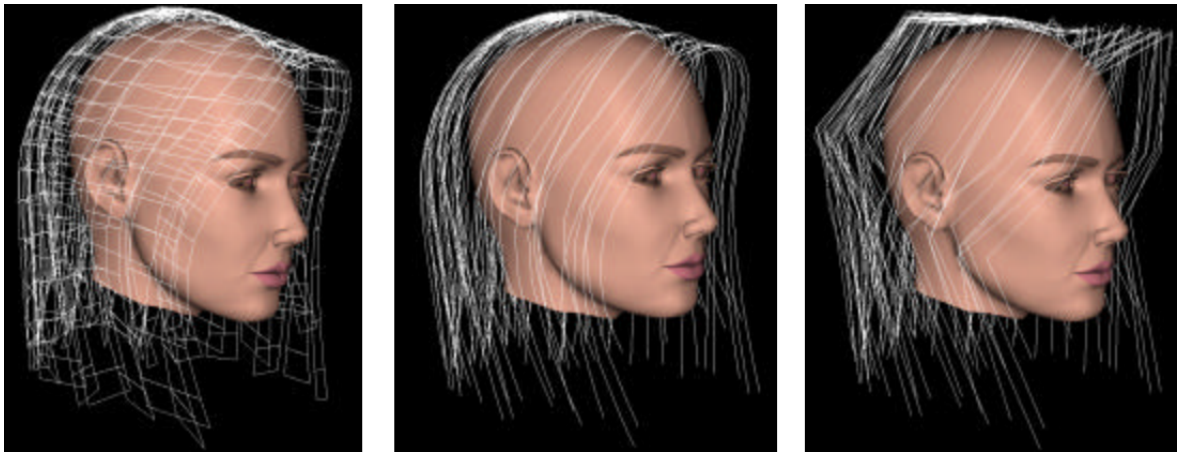


Figure 2.5: Hair representation as strips (left), smoothed lines strips (center), and raw line strips (right), Koh and Huang [12]

By treating hair as a continuum, we cannot replicate the individual behaviour of the strands. This is why the dynamics are split into two parts in this implementation. Continuum dynamics is used for the overall hair and serial rigid multi-body chains are used for the individual strands dynamics. The viscous pressure of the fluid simulation captures the complex frictional interactions between hair strands, while the serial serial rigid multi-body chains capture the individual behaviour of hair strands. By using both solid and fluid dynamics, Hadap and Thalmann [4] expressed perfectly the duality of hair simulation. This method, however, is rather slow and does not capture clustering effects observed in long human hair.

Hair Strip Representation

To overcome the difficulty of simulating a large number of geometries when dealing with full hair, Koh and Huang [12] presented a framework of human hair modeling based on large hair strips. Instead of accounting for thousands of strands, only a few hundred strips are simulated. This representation reduces hugely the complexity of the system. The strips are first represented as particles and then modeled into a curve by parametric surfaces (NURBS) using the segments joints as control points. The curve given by the control points is tessellated into a mesh (2D strip), shaded and textured using alpha blending in order to represent groups of hair strands.

The model uses the one-dimensional projection equations method, in analogy to Ajjyo et al. [3]. The simulation is only applied to the control points of the strips instead of the mesh vertices. This approximates the simulation a little but also reduces the computation cost in order to satisfy the real-time applications constraints. The use of a low number of simulated strands also makes the collision-detection between the strands feasible at interactive frame rates. This is done by introducing springs between each hair strip and its neighbours. Hair simulation became appealing to computer games with the introduction of this method. Unfortunately, this simulation is very crude and limited in terms of hairstyles and types of motion.

Guide Hair Strands Interpolation

Another way to perform hair simulation is to only simulate a small number of hair strands. The rest of the hair is generated afterwards, from the simulated strands, using interpolation techniques. Daldegan et al. [13] were the first to introduce such a method. For the sake of performance, they generated new hair strands by duplicating the guide strands and applying a small offset to them. Like in the case of hair strips, because the number of simulated strand is low, the inter-hair collision detection can be solved in a more classical way, without using complex fluids dynamics systems. In this implementation, however, there is no hair-hair interaction yet - only the hair-body interaction is handled using a discretized cylindrical presentation of the body.

Chang et al. [5] reused this technique and introduced a better hair-body collision model and mutual hair interactions between guide strands. The strand-body collision is handled by checking the hair particles penetration with the triangles of the body meshes. In order to take into account the hair mutual interactions they propose to build links between each guide segment and the closest point on the nearby strands. These connections exert spring forces on the hair segments during the simulation and are defined once, at the beginning of the simulation. One may use an octree structure to improve the performance of neighbourhood search. During simulation, these static links may break due to excessive forces. In that case the links no longer apply forces on the segments and are not rebuilt until the end of the simulation.

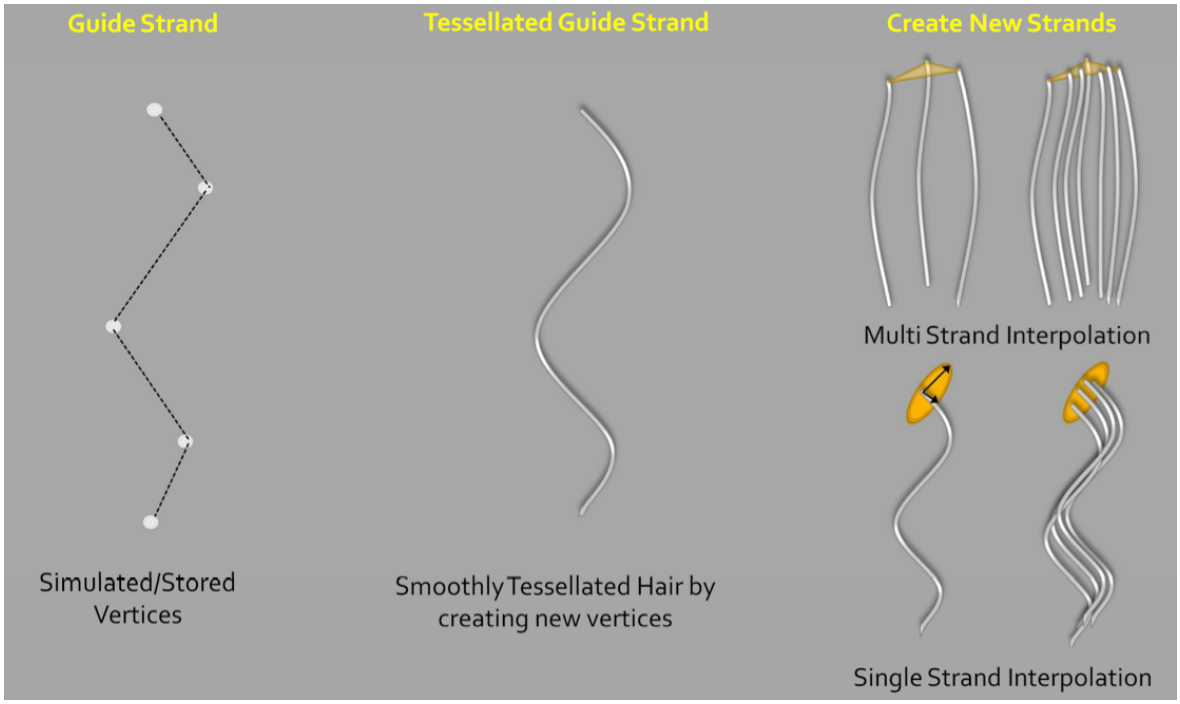


Figure 2.6: Principle of the guide strands interpolation technique, Tariq and Bavoil [11]

Collision detection is also checked between each segment of the set of guide hairs. Since this is not enough to give accurate results, additional triangle strips are built between pairs of nearby guides. These strips are only used for collision purposes, and are not rendered. In the case of collision, a strongly damped force is applied to the pair of elements to push them away from each other. Again, one may use an octree for fast collision detection.

Another interesting part of the simulation is the smoothing of the strands. The smoothing is performed by using Hermite spline interpolation between the hair segments. An average of fifty segments is sufficient for satisfactory results. The procedure used by Chang et al. [5] for generating new hair strands differs from the one used by Daldegan et al. [13], which tends to group hair strands as unnatural clusters. Here, the new strands are generated with a sophisticated interpolation using three guides as reference. Note that collision detection is not applied to the interpolated strands in this case, which can lead the strands to miss collisions with external objects.

Nguyen et al. [8] and Tariq and Bavoil [11], both from Nvidia, presented very convincing results based on this technique and at interactive frame rates. The work of these researchers mostly focuses on GPU optimizations since the company is a graphics hardware fabricant. It represents an important aspect of the most recent techniques in real-time hair simulation, as we will see in a later section. This is why the method of guide strands interpolation is preferred in this domain; it balances between realism and performance. It has the advantage of hugely simplifying the simulation stage, giving more room for optimizations, and thus more appealing results.

Tariq and Bavoil [11] proposed to use the previous two guide strands interpolation methods at the same time. For inter-hair collisions, a different approach should be taken into account. This approach is based on the work of Bertails et al. [14]. At first they generate a voxelized representation of the flul hair and then apply repulsive forces to hair vertices; from the highest to the lowest density areas. This method, although not very accurate in terms of collision detection, allows preserving a uniform hair density. Tariq and Bavoil [11] also proposed a technique to avoid collisions between interpolated strands and external objects.

2.2 Graphics Hardware and Hair Simulation

Real-time hair simulation and rendering are both very complex operations that require the maximum of computing resources. Even if some progress has been made since the first attempts, the performances of the simulation remain entirely dependent on the hardware performances. Instead of focusing on new integration methods or more efficient rendering techniques, some researchers focused on pipeline optimization. They used well-known techniques, while taking advantage of the capabilities of new generation of graphic hardware to accelerate the process.

2.2.1 GPGPU (General-Purpose Computation on Graphics Hardware)

For a long time, computer graphics simulations were executed on the CPU. Change has been introduced during the last years with the appearance of programmable graphics

pipelines. Some operations - not only related to computer graphics - can now be processed on the GPU. The later being much more efficient than the CPU for numerical calculus. The GPGPU relies on the parallelization of the operations and on the nature of the hardware specialized for numerical computation. In the beginning, GPGPU programs were computed using the rendering pipeline, since nothing on the GPU was designed for general purposes.

In 2008, Tariq and Bavoil [11] implemented very impressive real-time demos, thanks to the intensive use of graphics hardware. This implementation uses Vertex Shader (VS) programs to compute the physics simulation. The VS is normally used for vertex operations during the rendering process, but it can also be used for non-graphical operations, for performance gain. In order to simulate the motion of the guide strands, a buffer containing the nodes is passed to the VS. Then, the program updates the new positions for each vertex. Since the graphic pipeline does not allow reading and writing operation in the same buffer (breaks the parallel programing logic), the output data must be written to another buffer. An efficient way to solve this problem is to use the two buffers in ping-pong (buffer1 to buffer2, then buffer2 to buffer1, etc).

This simulation relies on the position-based dynamics, which can be easily implemented in a parallel. Some simulation steps, however, like the solving of distance constraints, work with pairs of vertices. This poses a considerable problem of parallelization since it allows the system to modify the same vertex several times and simultaneously. To avoid this issue, solving of the distance constraint must be done in a particular way and respect some rules. We will give more details about this in the implementation section.

In 2010, Yuksed and Tariq [15] brought some improvement to their previous work by introducing the use of the Compute Shader [22]. The Compute Shader is a feature of Direct3D that allows computing operations on the GPU in parallel but out of the rendering pipeline. It also provides shared memory to transfer information between threads as well as thread synchronization. In this implementation, each particle of the strands is simulated in parallel by its own thread.

The Compute Shader is similar to CUDA [23] and OpenCL [24]. It is, as previously stated, a feature of the Direct3D API and is therefore restricted to windows platforms. The CUDA framework is restricted to Nvidia hardware [16] only, whereas OpenCL is multi-platform just like OpenGL. These technologies brought the GPGPU to a whole new dimension, allowing many kinds of general-purpose computations to run faster and to be easily implemented .

Recently, a student from Stanford University, Steve Lesser [21], presented a master project on real-time hair simulation using CUDA. The challenge of this project was to understand the concepts of the simulations explained by Yuksed and Tariq [15] and to implement a similar demo by using a different technology. The idea behind our dissertation project is strongly inspired by this work. Instead of using CUDA, the challenge would be to use OpenCL to parallelize the physical simulation. Even though CUDA is considered more mature than OpenCL, the fact that OpenCL is an open standard might lead to its use as a reference in terms of GPGPU.

2.2.2 Pipeline Optimization

There are two primitive options available for the rendering of the hair strands. In both cases, the strands still need to be smoothed in order to get a curvy visual appearance. Indeed, after the simulation stage, the strands are only composed of a few segments. It is usually not efficient enough to get a smooth aspect, especially when the viewer's position is closely located to the strands. One may use the strands vertices as control points and apply a spline interpolation technique to create more vertices and segments and thus get a curvier result. This operation of spline smoothing was first introduced by Chang et al. [5] and was made "off-rendering". This means that the smoothing operation was made before the rendering and that the totality of the vertex information was sent to the renderer. This results in the risk of overloading the graphics pipeline with too much information. Nguyen et al. [8] implemented a smoothing technique using the Geometry Shader in order to improve the performances of the simulation. The Geometry Shader is a rendering step, just like the Vertex or the Fragment Shader. It allows decreasing the bandwidth of the pipeline by generating primitives dynamically. In that way can send less information to the renderer and save computing time.

The most intuitive way to represent hair strands is to render the strands as line primitives like Nguyen et al. [8]. It has the advantage of being easy to implement and speed. However, since the line width does not depend on the point of view, it needs to be updated manually according to the viewer's position. It also does not allow accurate perspective effects. For example, a part of a strand located closer to the point of view will look as wide as another part located further away from the point of view. The texturing of line is also limited to 1D textures only; therefore it is impossible to map a texture along the width of a line or to apply complex shading techniques.

The other option is to expand the line strips into camera-facing triangle strips, in analogy to Yuksed and Tariq [15]. This method solves all the issues encountered with the line representation, but is also more time consuming. Since more geometries are rendered, we might end up with a bottleneck if too many vertices are sent to the display. To avoid this issue, Yuksed and Tariq [15] propose to use a Geometry Shader to improve the performances of the graphic pipeline in analogy to the spline interpolation introduced by Nguyen et al. [8]. Accordingly, the triangle strips are generated on the fly during the rendering process and sent to the per-fragment rendering operation afterwards. Since only a small number of polygons are sent from the CPU to the GPU, this technique ensures that no bottleneck will occur at this stage. Of course, a greater amount of computation now relies on the GPU, but since the latter is more adapted for this kind of operation, it can handle a greater amount of computation.

In 2010, Yuksed and Tariq [15] also presented another improvement to the rendering pipeline compared to their previous implementation. As we know, sending information to the GPU is a very costly operation. Therefore, the less information we send, the faster the simulation will run. Sending less information to the GPU also means creating more geometries dynamically during the rendering process. In this presentation, they used a recent Direct3D feature, called the Tessellation Engine. This feature allows to dynamically generate new interpolated strands from the simulated guides and during the rendering stage. The guide interpolation step was usually done out of the rendering process. The geometries were created first on the CPU and then sent to the display for the smoothing, the tessellation, and the shading. But in this case, this step is also achieved during the rendering process. The most important advantage is that it is faster

to create data using the tessellation engine than it is to create data on the CPU and then upload it to the GPU, or even to use the Geometry Shader to create new vertices. Although the Geometry Shader would be able to perform such task, it shall only be used for small amounts of data expansion. In addition to the strand interpolation, the Tessellation Engine also performs the smoothing of the strands because it is also faster than the Geometry Shader. Once the lines are generated and smoothed, one may still use the geometry shader to expend those lines into camera-facing strips.

2.3 Hair Rendering

Realistic rendering of human hair is a well-studied issue, which present a certain amount of constraints due to the specific properties of human hair. The rendering of such a large number of strands also represents an important aspect of the challenge of hair simulation. This problem, in analogy to hair simulation is both local and global. On the local scale the properties of hair fibers define how the individual strands are illuminated. On the global scale, these properties describe how each strand cast shadows on the other fibers.

2.3.1 Individual Hair Strands Lighting

The light scattering is an important property of human hair to take into account when rendering hair strands at local scale. In reality human hair fibers are composed of various elements that absorb and also refract the incoming light. In 1989, Kajiya and Kay [16] were the first to implement a shading technique that take into account the scattering properties of human hair. This model represents the geometries in a volumetric texture and illuminates these geometries using anisotropic lighting. In analogy to Phong shading this model includes a diffuse component and a specular component. While Phong shading relies on the normal vector of the surfaces for computing the illumination, Kajiya and Kay's shading derives the diffuse and specular component from the tangent vector of the strands. Marschner et al. [17] proposed, in 2003, a more physically accurate scattering model for human hair. This model improves the previous one based on new measurements of scattering from individual hair fibers that exhibit visually significant effects not predicted by Kajiya and Kay's model.

Figure 2.7: Comparison between Kajiya's model (left), Marschner's model (middle), and real hair (right), image from Survey on Hair Modeling: Styling, Simulation, and Rendering [20]

2.3.2 Global Hair Illumination

The shadowing of hair strands becomes difficult on the global scale because hair fibers cast shadows on each other. However, they do not fully block the incoming light but rather transmit and scatter. Two main approaches are considered for this problem: by ray casting through a volumetric representation or by using shadow maps. The first one is the most naive way to proceed and is also not suitable for interactive applications because performance issues. The second one uses a texture to store a representation of the depth of the hair from the light point of view. This method, while less accurate than the ray traced ones, has a lot of potential for hair rendering in real-time. Yuksel and Keyser [18], in 2008, presented an advanced implementation of shadow maps for hair rendering, with conclusive results and at interactive frame rates. In real-time hair rendering the parallelization of the process is also an important aspect, in analogy to real-time hair physical simulation. As an optimization of the shadow map method, Bertails et al. [14] used a voxelised representation of the density of the hair in order to compute the transmittance inside the hair volume. Although very simple this method yields convincing results at interactive rate and can be easily parallelized for performance gains.

Chapter 3

The OpenCL Framework

The core of this project relies on the implementation of an efficient hair simulation using OpenCL. For the implementation, a good understanding of the framework is required. The following chapter is a short introduction to OpenCL that presents the key aspects of the framework.

As briefly explained before, OpenCL is a framework designed for general-purpose parallel programing. The framework abstracts the nuances between the different kinds of hardware and allows the same program to be executed on multiple machines. OpenCL is an intuitive tool for computation improvement. It is suitable for computer graphics but also for non-graphical computation, such as scientific programs that require high performance computing. The framework leverages the power of GPU's and multi-core CPU's and allows threads to run in parallel on the multiples cores of the devices.

3.1 OpenCL terminology

The OpenCL framework comprises a terminology that will help us to understand its operation. Among the various concepts the most important ones are the device, the host, the kernel, the memory objects and the work-items.

3.1.1 Device

The hardware on which OpenCL runs the instructions in parallel is called the "device". Each device contains several compute-units: a hardware unit capable of independent computations. The ability of a device to perform efficient computations with OpenCL depends on the number of compute units. The more compute-units we have, the more simultaneous computations we get. A CPU usually contains 2 to 8 different compute units (cores), while a modern GPU may contain tens to hundreds of compute units. This is why GPU's are more suitable for OpenCL computations, although an OpenCL program would run the same way on a CPU, only the computation time would differ.

3.1.2 Host application

The role of the host application is to call the OpenCL external functions in order to perform operations on the device. These functions may be used, for instance, to set up the OpenCL context, or to trigger the execution of the OpenCL programs on the device. The device on which the host application runs is called the host device. In most of the cases the host application runs on the CPU while the OpenCL programs run on the GPU. In some cases the host device can be the same device as the one executing the OpenCL computation, when the OpenCL device and the host device are both the CPU for example.

3.1.3 Kernels

We refer to the programs that are executed on the device as the "kernels". Their usage is somehow similar to GLSL shader programs (one function executed in parallel for each element of a dataset). The main difference is that the output data may not be related to computer graphics (other than a vertex or a fragment). The OpenCL language is a variation of the C-language just like GLSL. In order to be customized for each kind of device, the kernel programs must be compiled on the host application before being executed on the device.

Even though the same program is executed in parallel it does not mean that the same instructions are executed in parallel. For instance, when using "if" statements,

the actions that correspond to the statements will be only performed by the threads satisfying these conditions. This allows a lot of flexibility for the program, because each thread can have its own behaviour. In some cases, when the kernel program becomes more complex, special care must be taken when writing the program to avoid parallel processing artefacts. In OpenCL, these artefacts would be caused, for instance, when several threads try to access the same part of memory at the same time. In that case, we might risk erasing some important value or reading an out of date value.

3.1.4 Work-items

The purpose of OpenCL is to run kernel programs in parallel. Therefore, the same program will be executed several times simultaneously. Each instance of the program is referred to as "work-item". A work-item of a kernel program is only executed once and on one compute unit of the device. It is very important to get the meaning of the work-item because the logic of OpenCL lies in the organization of these elements. The work-items are similar to the notion of threads in parallel computing. In the other similar frameworks, such as Compute shaders or CUDA, the work-items are referred to as threads. The work-items may be synchronized or share information with other work-items. These operations, however, are only possible between the work-items of the same group. This is why OpenCL allows us to arrange the work-items within work-groups. In the following section about the OpenCL execution model, we will see how such an organization may be structured.

3.1.5 Memory objects

The memory objects are a very important aspect of the OpenCL computation, they allow the kernel programs to read from and write values to the device memory. The memory objects that need to be shared between the host and device application are declared from the host application but stored on the device memory. OpenCL provides some API functions to read from and write to the device memory objects from the host application. They may be used, for instance, when we wish to transfer the result of a calculation back to the host in order to display the results. The memory objects can either hold untyped data (value or array of values) or image data (2D or 3D images).

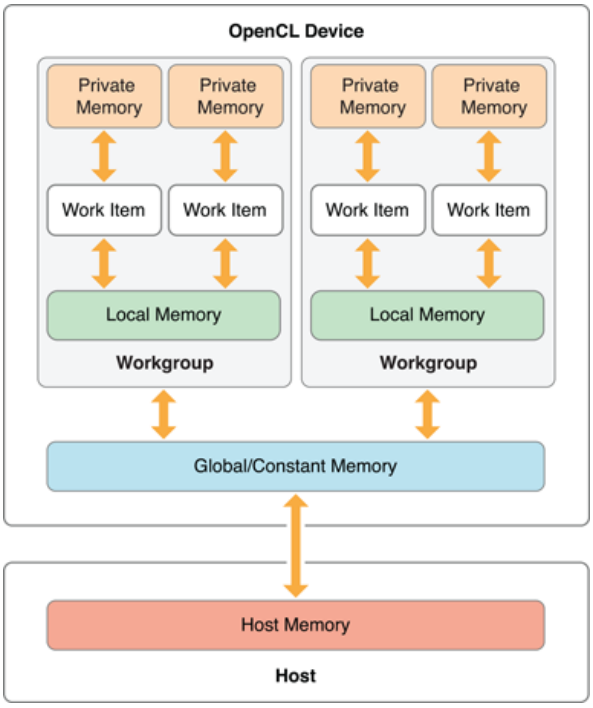


Figure 3.1: Schematic representation of the OpenCL framework

3.2 The OpenCL memory model

The memory object model describes the different domains of the memory objects. Each domain has its own properties and defines the behaviour of the memory objects. In OpenCL, a memory object can be global, local, constant or private.

Global A global memory object can be accessed (read and write) by any work-item in any work-group, just like the global memory of a C program. In comparison to the non-global memory objects, the global memory has high access latency. This is why the developer must be careful when writing a kernel, not to overload the bandwidth by accessing the global memory too frequently. The best practice would be to use cached data instead.

Constant The constant memory objects can be accessed by all the work-items, like the global memory object, but it just allows read-only access.

Local A memory object defined as local is accessible only to the work-items within the same work-group (each work-group would have its own local buffer). The access time to the local memory is much smaller than the global memory. Hence, it is wiser to use a local memory to share values between threads rather than using the global memory.

Private The memory object which is declared as private is proper to each work-item and can not be accessed by any of the other work-items. By default, if no type is specified for a memory object, this one is declared as private.

3.3 The OpenCL execution model

In this section we will briefly explain the execution model in OpenCL. The execution model defines how the work-items are organized. A coherent organization is an important task; it defines how will be executed the kernel program on the device.

The purpose of OpenCL is to execute instructions in parallel. OpenCL exploits each compute units of a device to run the kernel programs. When executing a kernel program, the host device sends as many instances of the program as needed to the device. The work-items are structured in groups, because it is usually unlikely that the device can handle the computation of all work-items at the same time. The grouping of work-items also allows to share memory and to synchronize the work-items within the same work-group. The sharing is made possible with the use of local memory objects and the synchronization with barrier and memory fence functions. Each work item is defined by a global and a local index, and the index-space can comprise in 1,2 or 3 dimensions, according to the needs of the algorithm The global ID of a work-item represents the index among all the others work-items, while the local ID refers to the index within the work-group only. This information can be accessed in a kernel program via built-in functions. By default, the organization of the work-groups is managed by OpenCL. One can also set the number and size of the work-groups manually, however, these properties must not be greater than the maximum size allowed by the device.

Chapter 4

Design

The simulation software we developed for this project is relatively simple in terms of code organization since we do not consider code reutilization or integration into a third-party application. Even though the host application has some importance, the complexity of this exercise is based on the writing of kernels and shaders programs. Yet these programs are generally short, declared in a unique file and do not allow the use of object-oriented programing. For these reasons, we will not provide any kind of diagram related to software development. Instead we provide a data flow diagram that describes the different key steps of the simulation in Figure 4.1.

The success in the implementation of such a program does not rely on a strong code design but rather on a very good understanding of the previous work and of the OpenCL framework. The state-of-the-art review had led us toward the most intuitive and efficient ways to proceed with hair simulation and rendering. However, we will not implement any of the state-of-the-art techniques in rendering. We do so because we deem that the project is complete enough just dealing with simulation. We will content ourselves with a basic lighting, without any shadows or ambient occlusion. The first part of this project is related to the physical simulation and rely on OpenCL programs while the second part is related to the rendering and was implemented with GLSL programs.

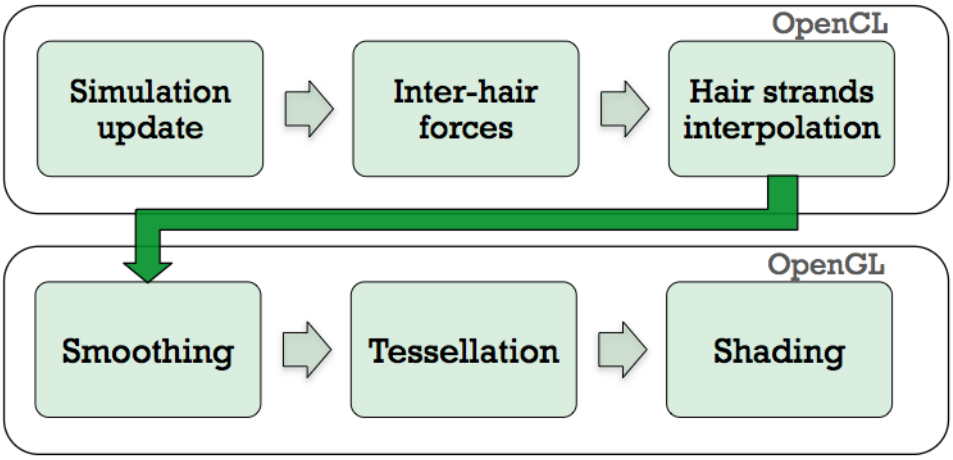


Figure 4.1: The data flow diagram of the simulation

Our project is strongly inspired by the work of Yuksed and Tariq [15]. We perform the simulation step in parallel, based on the position-based dynamics principles. Only hundreds of strands are simulated in order to lighten the calculation. Thereafter, the simulated strands are interpolated in order to give more volume to the hair. The inter-hair forces are computed in analogy to Bertails et al. [14], by using a voxelized representation of the density. The strands are then smoothed and tessellated from the Geometry Shader. Finally, we apply a shading technique similar to the one introduced by Kajiya and Kay [16].

The host application was developed with Qt, which is smart way to develop all different kinds of applications. Among its numerous advantages, the Qt SDK is multi-platform and possesses a powerful Graphic User Interface (GUI) framework that allows us to integrate a wide range of different widgets into our application. The GUI of our application is built with this framework in order to give control of the simulation to the user. We will enumerate the different parameters available to the user throughout the implementation chapter.

Chapter 5

Implementation

In this chapter we will describe the implementation of the different steps previously outlined in the design chapter. The core of this project is related to the use of OpenCL, therefore we will first focus on the implementation of the kernel programs. In other words, we will describe the different kernels programs and how to set up the OpenCL platform in order to compute such a kind of simulation. Since the physical simulation is implemented with OpenCL, we will cover this notion in the first part in this chapter. The interpolation of the guide strands is also related to OpenCL, accordingly we will cover this subject in the first section as well. In the second part of this chapter, we will focus on the visual appearance of the hair. First, we will first explain the smoothing and the tessellation steps. Finally, we will focus on the lighting technique used in this implementation.

5.1 Physical simulation with OpenCL

This section covers the physical simulation, which deals with the motion of the hair. Here, we will discuss the different techniques applied in order to get a realistic and interactive hair simulation. Most of the methods employed here have been previously described in the state-of-the-art chapter. The purpose of this section is to go further into the details of these methods. We will describe how these methods have been implemented in this project and what kinds of problem were encountered during the implementation. We have also developed some new techniques in order to facilitate the

simulation and to improve the results, which will be highlighted in this chapter. The physical simulation section is split into different subsections which follow the flow of the data during the simulation. First, we will briefly explain how the data is represented. Secondly, we will focus on the simulation and the computation of the inter-hair forces. Finally, we will approach the subject of the guide strands interpolation.

5.1.1 Hair strand representation

Before focusing on the explanations of the simulation update, let us have a brief overview of the structure and the placement of the simulated hair strands and their sub-elements.

Each guide strand is composed of particles which we call nodes. They are linked to each other and form the hair segments of the strand. The nodes represent the atomic entities of the simulation and each of them is simulated in parallel. A node is equivalent to a vertex, e.g. a position in a three dimensional space. In theory, a vertex may have several attributes in addition to the position such as the colour or the normal vector, but in the context of this exercise we only need the position information.

We must declare a structure to accommodate the nodes. For this, we simply need to declare an array of nodes. This array must be declared on the memory of the OpenCL device, which will be updated by the kernel program. By the end of the simulation update we will have acquired the new nodes positions and we will be ready for the display. The host application does not really need to access the results of the simulation in our case since the display device is also the OpenCL device (the GPU). This is why we may simply declare the array as OpenGL Vertex Buffered Objects. Indeed, OpenCL allows the manipulation of VBO allocated with OpenGL on the memory of the graphics device. This interoperation between OpenCL and OpenGL allows us to transfer data to the renderer without having to transfer any data back to the host application. Of course, this interoperation only applies if the display device and the OpenCL device share the same context, in other terms, if they share the same and unique device.

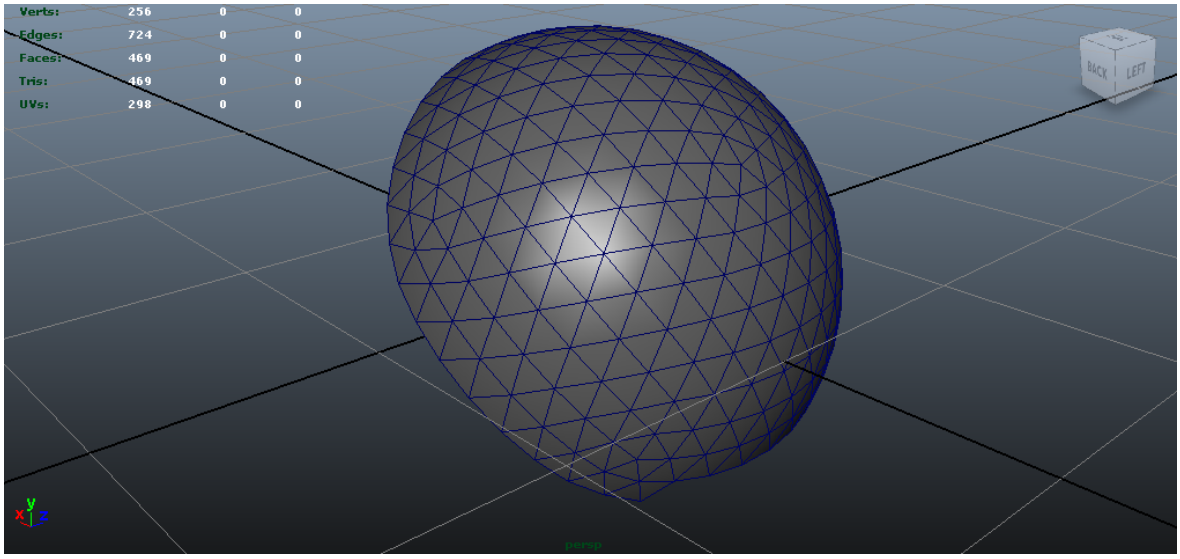


Figure 5.1: The scalp model

After the declaration of the vertex array, we must fill this array with the initial values. We write these values into the OpenCL memory objects from the host application by using the OpenCL API functions. We start the initialization by defining the position of the root nodes. The positions of these roots are determined thanks a 3D model of a scalp. Each root corresponds to one vertex of the scalp. The 3D mesh is previously modeled with a modeling software package and must respect certain rules. We will elaborate on the details of these rules in the section about guide strands interpolation. The rest of the nodes of each strand are aligned with the roots positions and the center of the scalp. This alignment respects the initial separation distance between each node.

It is important to make sure that the first segments of the strands are located below the surface of the scalp and remain static during the simulation. When a segment is declared as static, the nodes that compose the segment are not physically integrated. This is necessary for the application of angular constraints. We will explain the details in the angular constraint section below. Even though the first segments must remain static, their positions are relative to the scalp. To attach these segments to the scalp, we apply the transformation matrix of the scalp to the nodes of the segments. Thus, when translating, scaling or rotating the scalp, the roots will follow this motion. The rest of the nodes will also follow this motion, due to the distance constraints.

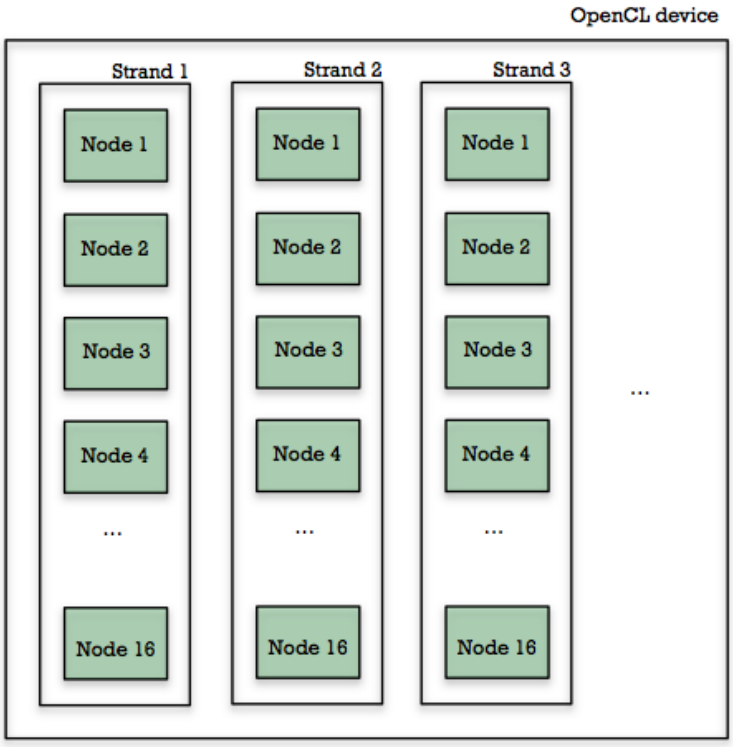


Figure 5.2: Organization of the work-items on the device

5.1.2 The simulation update

The simulation needs to be updated quite frequently in order to ensure interactivity. In this section, we will explain what exactly happens during one update step. We will study the different forces and constraints of the system and describe how this can be achieved with OpenCL.

The simulation update is the task of one dedicated kernel program. This program aims to integrate the positions of the nodes and to satisfy the different constraints as well. The kernel also allows for the construction of the hair density map. This map will be subsequently processed through another kernel during the inter-hair forces stage. The source code of the simulation kernel is available in appendix A, Listing A.1.

As explained above, the representation of the nodes is done using a Vertex Buffered Objects. The VBO comprise only one dimension. Thus, the organization of the work-items will also be done in one dimension. This facilitates the exercise because the one-dimensional organization is the easiest way to proceed. In the section about distance constraints, we will see that the nodes of the same strands need to communicate with each other. This is why they should be grouped in the same work-group, one group per simulated strand.

According to the OpenCL framework, the total number of work-items and the number of work-groups must be multiple of two and also divide together evenly, let us illustrate an example of a valid configuration: We may for instance use a set of 256 simulated strands, each of 16 nodes. In the previous case, we get a total number of nodes of 4096. Therefore, we need to declare 4096 work items (one work-item for each node), and 256 work-groups, each one containing 16 work-items. It is important to set the size and the number of work-groups manually here. Otherwise, we might end up with an invalid configuration. Before executing a kernel program on the OpenCL device, we specify the number of work-items and the size of work group with the following function:

Listing 5.1: The enqueue kernel function prototype

```
cl_int clEnqueueNDRangeKernel ( cl_command_queue command_queue,
    cl_kernel kernel,
    cl_uint work_dim,
    const size_t *global_work_offset,
    const size_t *global_work_size,
    const size_t *local_work_size,
    cl_uint num_events_in_wait_list,
    const cl_event *event_wait_list,
    cl_event *event)
```

This function triggers the execution of a kernel program of the OpenCL device. *The global_work_size* and *local_work_size* parameters correspond respectively to the total number of work-items and to the number of work-item per group.

Verlet Integration

At the beginning of the simulation we apply the internal and external forces to the nodes. This step is called the integration. It allows us to find the new position of the nodes for the current time value. As shown in chapter two, there are different ways of integrating these forces. One of these technique is particularly relevant for constraint-based systems such as hair simulation. This technique is called the Verlet Integration, made famous in interactive applications by Jakobsen [6].

Each node of the simulation is updated thanks to the Verlet Integration. The new current position is determined by the differential of the two previous positions P^{-1} and P^{-2}. With the use of the Euler integration, for instance, we would need to keep track of the velocity value for each node. This is not necessary here since the latter is implicitly stored in the previous positions. Nevertheless, we would need one extra buffer in addition to the VBO in order to store the previous positions P^{-2}. This buffer is only used for the computation and not for display purposes. For this reason, it does not need to be declared as a VBO like the P^{-1} buffer.

Before each simulation step, we have a VBO that holds the positions from the previous step P^{-1} and an additional buffer containing the previous position from the previous step P^{-2}. By the end of the simulation step we need to update the values contained in these two buffers in order to prepare the next update step. We then replace the values of the previous positions buffer with the initial positions at P^{-1} and fill the VBO with the result of the simulation update afterwards. During the integration step we also add external forces, such as gravity, wind, wind drag and inter-hair collision forces. These forces are all accumulated in the same vector and integrated according to the following formula:

$$P = P^{-1} + (P^{-1} - P^{-2}) + a\Delta t^2 \tag{5.1}$$

Where P represents the current position, P^{-1} and P^{-2}, the previous two frames' positions, and a represents the accumulation vector of the external forces of the system. A description of the different external forces of the system can be found below:

Gravity The gravity is a direction in which the nodes are steered, like in reality with the earth gravity. In our case, the direction of the gravity is perpendicular to the ground and cannot be changed by the user. The scale of the gravity, however, may be modified by the user.

Wind The wind also represents a direction vector, but unlike the gravity its direction can be fully modified by the user during the simulation. The force is made proportional to the angle between the hair orientation and the wind direction in analogy to Oshita [9]. Such a technique gives a greater influence to the wind when perpendicular to the hair segment. This also avoids the wind to stretch the hair strands when the latter blows in the same direction as the strands. Equation 5.2 gives us the wind force according to the orientation of the strands:

$$F_{wind} = \frac{|w \times v|}{|w|} w \tag{5.2}$$

w is the user-defined wind direction and v is the tangent of the node along the hair spline. Since the wind force only represents a direction, the motion due to these forces may look a bit flat, especially once all the strands are aligned with the wind direction. This is why we have added some turbulence, generated using noise functions, in order to get realistic wind motion.

Wind drag The motion of hair strands in fluids such as air is usually damped. This damping is caused by the resistance of the strands with the fluid due to their light weight. To obtain this effect in our simulation, we accumulate a wind drag force with the other external forces. The value of the wind drag force can be approximated with the following formula:

$$F_{drag} = -bV \tag{5.3}$$

V corresponds to the velocity and k to the strength of the drag. The strength value may be modified by the user during the simulation via the GUI. Since the formula requires the velocity, we must extract this value from the previous two frames' positions.

This value can be retrieved by the following formula:

$$V = \frac{P^{-1} - P^{-2}}{\Delta t} \tag{5.4}$$

Inter-hair forces The inter-hair forces are accumulated with the other external forces but are computed outside the simulation in another kernel program. We will discuss this in detail in the section dedicated to inter-hair forces.

Constraints solver

In this part we will cover the solving of the different constraints of the system. The constraints we use in this simulation are position-based. This means that the response of the constraints is directly applied to the position. The use of position-based constraints in addition to the Verlet integration represents the core of the position-based dynamics. This integration relies on the previous positions instead of on an explicit velocity value. Thus, we can maintain a consistency while simply modifying the positions.

According to that concept, we simply have to modify the positions of the nodes in order to satisfy the different constraints of the system. This step is called constraint solving. There are three constraints we have to implement to ensure a coherent hair simulation: the collision constraint, the distance constraint and the angular constraint. The last constraint of the solving step is by default the strongest constraint, because a constraint that is solved after the others is guaranteed to be satisfied by the end of the constraints solving. On the other hand, a constraint that is solved first might be corrupted by the solving of the following constraints.

When using several constraints together, one usually repeats this operation several times. These iterations ensure that all the constraints will be respected. The number of solving iterations can be modified by the user during the simulation via the GUI. The constraint solving is usually the most time-consuming stage of the position-based dynamics. This is why this value must be manipulated carefully. Indeed, we might risk seeing a significant drop of performances if this value is increased too much.

Each constraint has a coefficient that determines the strength of the constraint. These values can be modified during the simulation as well. A constraint coefficient is defined between 0 and 1. A value of 1 means that the constraint is fully satisfied and a value of 0 nullifies this constraint. These coefficients allow us to keep the control of motion of the strands. In that way we may tweak these values, following some artistic insight. For instance, we could decide to simulate a very straight hair, like a brush. We could get such results by using a strong coefficient for the angular and distance constraints. We could also simulate a softer hair, like human hair, using lower coefficient values.

When using several solving iterations, the effect of the coefficient constraint is not linear anymore. It means that we may get unexpected results for some coefficient value. According to Müller et al. [7], we must use a coefficient value defined by the following formula in order to get linear results:

$$k' = 1 - (1 - k)^{\frac{1}{n}} \tag{5.5}$$

Where n is the number of iterations and k the user-defined strength coefficient.

Collision constraints The first constraint of the solver is the collision constraint. It is also the easiest constraint to solve within the whole constraint set. In this project, the collision checking is made at its simplest. We only control collisions between the hair nodes and spherical shapes. The first object we check for is the head (which is a sphere). We can also include other spherical objects in the scene as external obstacles - to model the body, for instance. Then, for each node, we must check for potential penetrations between the nodes and these obstacles. A collision is triggered when the distance between a node and the center of a sphere is lower than the radius of the sphere. In that case we must apply a correction to the position of the node. The correction does not take into account the incoming velocity. We simply move the node back outside the sphere with a normalization operation. The normalization will move the node in the direction of its current position and the center of the sphere.

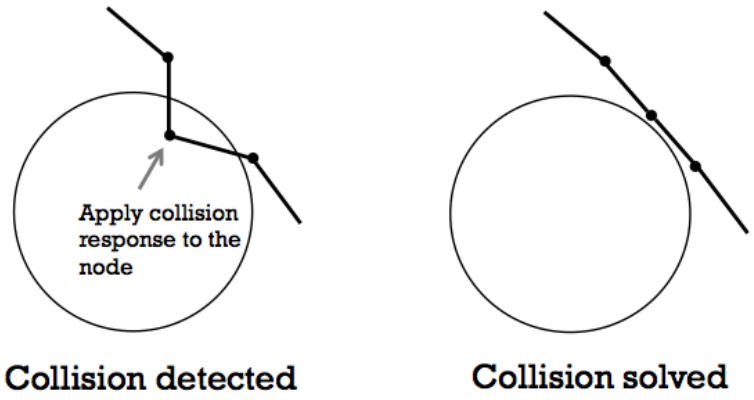


Figure 5.3: Application of a collision constraint

It is important to know that the collision is a hard constraint. A hard constraint is a constraint which needs to be fully satisfied. There is no user control for this constraint, the coefficient is fixed to 1 so the constraint is satisfied in any circumstance. It is also interesting to know why we decided to solve the collision constraint first. When we apply a correction to a node after a collision, we modify the position of the node. Afterwards, the distance constraints are applied to the subsequent nodes in order to form a coherent line strip which represents the hair strand. If we decide to apply the distance constraints before, we will not see the subsequent nodes being updated after the correction. This could lead to unpleasant visual results. This problem is usually attenuated when using several solving iterations, but it can be avoided if the collision constraint is solved first.

Distance constraint The second constraint to be applied in the solver is the distance constraint. The distance constraint is an important notion in the domain of hair simulation because it ensures the validity of the system. The principle is simple, it is to decrease or increase the distance between two nodes in order to satisfy a certain distance. Once all the distance constraints are applied to the nodes, each strand forms a chain of segments in which all the nodes respect the correct separation distance. In the case of the collision constraint, we only modify the position of one node for each constraint. Therefore, it is easy to solve these constraints in parallel. In the case of the distance constraint, however, we need to modify the position of two nodes at the

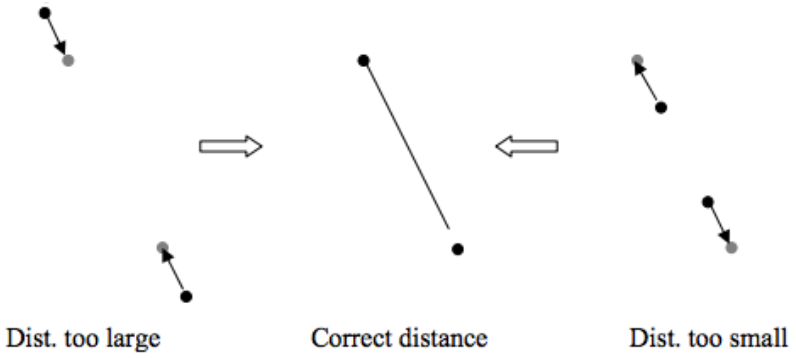


Figure 5.4: Fixing an invalid distance by moving particles.

same time. This means that two adjacent constraints cannot be updated in parallel. Otherwise we risk some incoherency in the data. Indeed, if two adjacent constraints try to access the same node in parallel, one of the two constraints would simply erase the result of the other constraint. Even if this would only occur if the threads of two adjacent constraints which are perfectly synchronized, we must consider this issue. In parallel computing, the major rule is to avoid several threads accessing the same data at the same time, even if the probability of this happening is very low.

To overcome this problem, Tariq and Bavoil [11] suggest solving the constraint in pseudo-parallel, by splitting the constraints in two different sets. Each set only contains constraints that do not share any nodes with each other. To divide the constraints in two sets we can, for instance, solve the constraints that start with nodes with local even indices first. The odd constraints will be solved afterwards. Since the splitting leaves a separation of one segment between two successive constraints, we can ensure that none of the constraints will share any nodes. Figure 5.5 schematizes the concurrency issue of the distance constraints. The full lines of the two batches represent the constraints that are included in these batches.

To decide if a work-item is solving an even or an odd constraint we simply check the parity of its local ID. Let us remember that the local ID corresponds to the local index of a node within a strand. Again, we must pay attention to synchronization issues. Indeed, as we already mentioned in the OpenCL overview, the same program does not necessarily include the same instructions. The use of conditional, for instance,

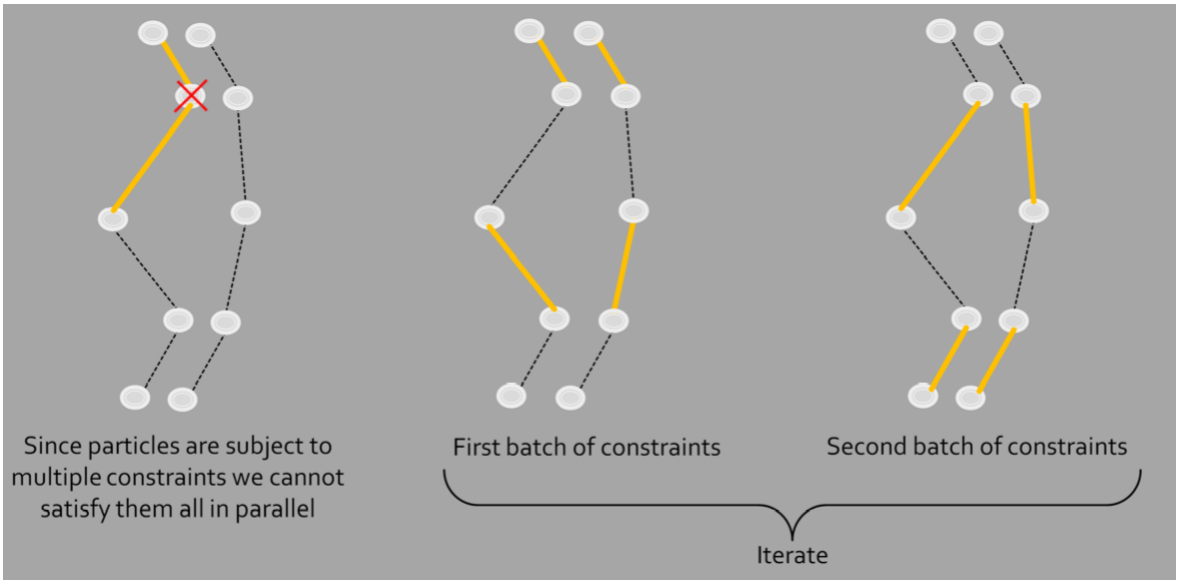


Figure 5.5: Concurrency issue of distance constraints, Tariq and Bavoil [11]

may cause a desynchronization of the threads. If the threads are not synchronized, some work-items may start to solve odd constraints before all the even constraints are solved. This is the kind of situation we need to avoid. Fortunately, OpenCL provides us with some functions to synchronize the threads within the same work-group, like the barrier() function. This function, once used, blocks the execution of the work-items of the same group until they all have reached this instruction. This function allows us to satisfy the two batches of constraints separately and to ensure the viability of the results.

Before this point, the changes made to the current position may be stored in a private variable. This allows us to spare computation time due to the fast access time of the private memory objects. The distance constraints, however, read from and write to the local memory only. This is why all the current positions must be affected in the local memory before starting to solve the distance constraints. We also need to add a barrier() function right after this affectation and just before the constraint solving, otherwise we might see some threads accessing some part of the local memory that has not been updated yet.

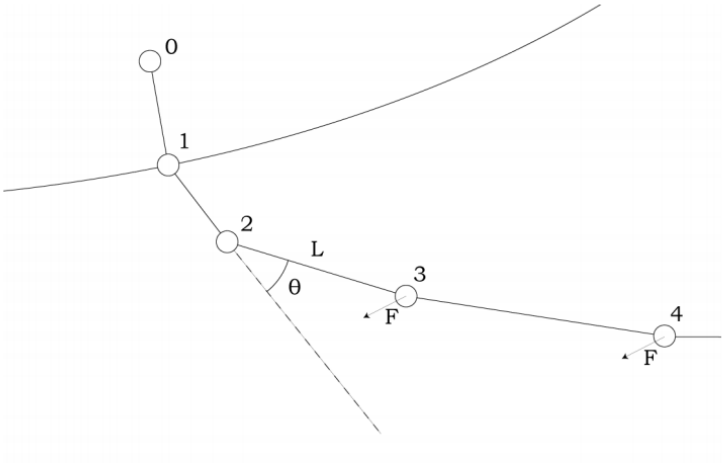


Figure 5.6: Schematic representation of the angular constraint

The user disposes of two parameters that let him modify the behaviour of the distance constraint. First, we have the stretching parameter, which is actually the strength coefficient of the constraint. This parameter has the ability to give stretchiness to the strands with a value of lower than 1. As a result, when we lower the value of this parameter, the hair strands will behave like a rubber or a loose spring. The second parameter available to the user corresponds to the spacing between the nodes. This scale value will have a direct impact on the strands size: a small spacing will shrink the strands while a greater spacing will enlarge them.

Angular constraints The last constraint of the solver is the angular constraint. The angular constraint prevents the hair from extreme bending due to strong external forces such as gravity or wind. The principle of this constraint is to push the nodes of the same strand to align with each other in order to give some rigidity to the strand. Indeed, we can notice from observations that straight human hair strands tend to erect straight from the scalp before bending under their own weight. This constraint, although not compulsory to get a believable motion, helps us to get a realistic strands behavior. It has also been implemented in most of the previous implementations presented in chapter two. This is why we included this important feature in this project.

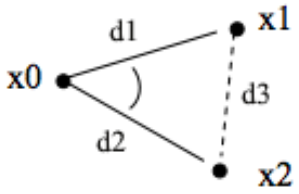


Figure 5.7: Two adjacent segments subject to an angular constraint

One may use different approaches to solve angular constraints. In previous work, Tariq and Bavoil [11] suggest to solve the constraints successively for each pair of nodes. The solving of a pair of segments is done by correcting the orientation of the second segment according to the angle between the two segments. This process must be done in serial from the scalp to the points, because each constraint needs the previous segments to be already aligned in order to remain aligned with the strand. Compared to a technique executed in parallel, the computation of this stage is slowed down. Furthermore, the transformations of each node into local space as well as the rotation operations applied to the segments would imply to use costly matrix operations. This is why we came up with an alternative method, which provides similar results with less complexity. The principle is very simple; we assume that an angular constraint can be solved with a distance constraint. This technique is effective under one conditions: the angle of the constraint must be close to 180 degrees, in other words, the two segments must be aligned. In our case, this technique may be applied because we are only dealing with straight hair. In the case where we wish to apply angular constraints with angle value different to 180 degrees we would need to implement a different technique.

This technique, illustrated in Figure 5.8, is a bit more approximate than the previous one but leads to very convincing results. The angular constraint is represented by the doted line and the distance constraints are represented by a continuous line. According to the previous principle, if we apply a distance constraint of length d3 = d1+ d2 between the nodes x1 and x2, this would results in the alignment of the two segments x0x1 and x0x2.

Another advantage is that it can be easily parallelized. Indeed, the first technique requires solving the constraint for the previous segments before solving the next pair

of segments. We use another approach here, since the pairs of segments are always composed of the same first segment. This segment is the one located below the scalp surface and declared as static. As a result, all the nodes will lay on the same line defined by the first segment and will all be aligned with each other. This first segment is the reference for all the other segments. The angle between this segment and the surface of the scalp will determine the direction of the constraint and thus the direction of the hair. In this implementation, the direction of the first segment is simply defined by the direction of the vector composed of the root and the center of the scalp. As an improvement of this software we can think of a tool that would set an angle value properly to each strand in order to model the hairstyle.

The fact that this stage does not need to be done in serial makes the last technique much more appealing than the one described before. Because of the parallel computing environment, it is always better to distribute the computation among the thread rather than relying on one thread while all the others stay idle. The only downside is that we can only constrain the strands in one direction. Thus, we cannot give any style to the hair at this stage. However, Tariq and Bavoil [11] suggest that hair styling should be done during the smoothing stage by applying a sinusoidal offset to the results of the spline interpolation.

The angular constraint comprises a set of user-defined parameters, in analogy to the previous constraints. The bending coefficient represents the strength of the constraint. This value is defined between 0 and 1 and the variation of this number between this range modifies the rigidity of the hair strands. A value of 1 makes the hair straighter while a value of 0 makes the hair become softer.

Because each constraint coexists with the other ones and with the external forces, the result of a given constraint can be attenuated sometimes, even with a strength value equal to 1. This is why we introduced the bending scale value, whose purpose is to give more strength to the constraint. This value simply enlarges the distance constraints defined within the frame of the angular constraints. These distances are by default defined according to the distances defined within the frame of the distance constraints. In the case in which the scale value is greater than one, this will result in

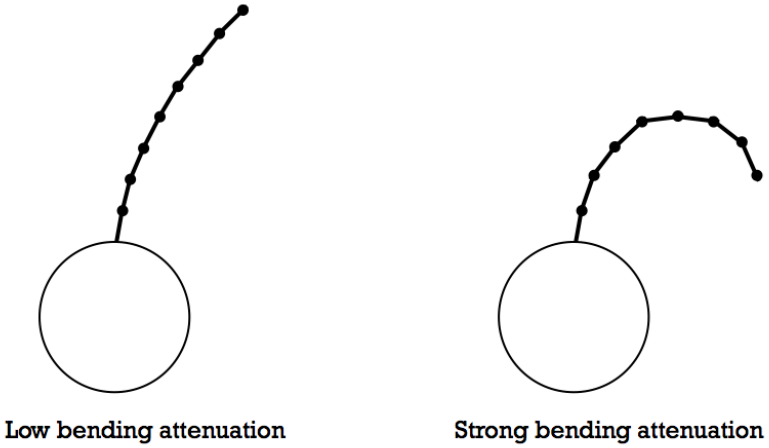


Figure 5.8: Representation of the bending attenuation parameter

a fortification of the constraint. This can be useful when trying to get very rigid hair, for instance a brush effect. The scale value also gives us a wider range of types of hair when tweaked together with other values.

Another parameter available to the user is the bending attenuation. This parameter is meant to give us more control over the style of the hair. The bending attenuation attenuates the effects of the bending progressively along the strands. Thus, the hair can come out straight from the scalp and bend harmoniously over its own weight after a few nodes. This notion came to us from the observation of real human hair. We noticed that the hair stands are subject to more bending when closer to the roots. Therefore, the ends should be less constrained than the root. This parameter allows the creation of a whole range of creative and interesting hair types. Equation 5.6 correspond to our custom bending attenuation formula:

$$k'_{bending} = \frac{k_{bending}}{i_{node}, k_{attenuation}} \tag{5.6}$$

The bending attenuation formula is applied to the bending coefficient, e.g. the strength of the constraint. The $k_{bending}$ value represents the user-defined bending parameter and $k'_{bending}$ the new bending coefficient, i_{node} represent the index of the node within the strand and $k_{attenuation}$ represents the user defined attenuation parameter.

Since the value of the denominator becomes greater as the node index increases, the result of the fraction will become smaller and the constraint weaker as we progress on the strand.

The angular constraint is the last step of the simulation stage. Once the latter is achieved we can transfer the data from the local memory into the VBO located in the global memory. We also build the density map of the hair by the end of the simulation stage. This data will be used in the kernels dedicated to the inter-hair collisions. More details about this are given in the following section.

5.1.3 The inter-hair forces

As stated above, the inter-hair forces are applied in the simulation kernel, during the verlet integration. This implies that these forces are computed somehow, either before or after the simulation step. To compute the inter-hair collision forces we use a technique introduced by Bertails et al. [14], reused afterwards by Tariq and Bavoil [11]. Once again this technique was chosen because it can run in parallel and thus, be executed at interactive frame rates. The principle is to use a volumetric representation of the hair density in order to find the areas of highest density. The hair nodes contained in these areas are then spread towards areas of lowest density. This is supposed to approximate inter-hair collisions assuming that hair-hair collisions are more likely to happen in the areas of highest density. Even though collisions are still likely to happen, this method yields very convincing results

Since this technique uses a discretized representation of the volume the system has to be bound somehow. This can represent a limit for the simulation, since we are constrained to this predefined volume. In the case of hair simulation it is not really an issue because the motion of the hair is only limited to a certain volumetric range. The hair is attached to the scalp and does not move freely. This technique is somehow close to the fluid dynamics technique introduces by Hadap and Thalmann [4]. They both use a discretized representation to solve the repartition of the particles within a confined space. The technique used in this exercise, however, is not as efficient since we just spread the particles using forces.

The technique is composed of three different steps. First, we create the voxel density map of the hair, then we smooth the values with a blur technique and finally, we determine the forces that are going to be applied to each node. We would need to implement several kernels in order to achieve these different steps. The organization of the work-items and the work-groups, however, remains the same for all the dedicated kernels of this stage.

The voxel grid

We store the hair density information in a three-dimensional structure, which represents a cubic volume surrounding the scalp. The volume is divided into smaller cubes of the same size called voxels. We use a three-dimensional array of integer values to represent the grid in this program. The integer value corresponds to the density value of the voxel. The three indices (x, y, and z) of each value correspond to the position of the voxel in the structure but also in the three-dimensional space. If we apply the offset and the scale value to the indices we can retrieve the position of a voxel in space. When filling the density map we set the density values according to the number of nodes located in each voxel.

The size of the grid must be chosen wisely. On the one hand, it must not be too small, otherwise we will have a partial representation of the density. On the other hand, it must not be too large in order to save memory. The optimum size of the grid is defined by the maximum size of a hair strand and the size of the scalp. In each dimension of the grid the size must be equal to twice the maximum size of a strand plus the radius of the scalp.

The resolution is defined by the number of voxels in each dimension of the grid. For the sake of simplicity we use the same resolution for each dimension of the grid. The accuracy of the simulation depends on the grid resolution, however, a higher resolution increases the memory as well as the computation time. This is why this value we must also be chosen carefully.

The structure must be declared on the device memory in order to be updated by the OpenCL program. OpenCL provides us a built-in structure to represent three-dimensional data. However, within OpenCL 1.0 this data type is only accessible in read-only access. Since we need to update the data every frame, we must find another way to host the density values. An easy way to represent multi-dimensional data is to flatten the array into an array of lower dimensions. In this exercise we use a single dimension array to hold the volumetric data. The array must be large enough to store all the voxels contained in the grid. This number is equal to the resolution power of the number of dimensions. In order to access the data contained in a flattened array we define an access formula. The access formula is defined below:

$$data3d(x, y, z) = data1d(x + y \times resolution + z \times resolution^2) \tag{5.7}$$

The left part of the equality represents the normal way to access three-dimensional data and the right part represents the way we access our flattened array.

Work items organization

The organization of the work-items for the kernels of the inter-hair collisions stage is different from the simulation stage kernel. In this stage, the number of work-items is not set to be equal to the number of nodes but to the number of voxels. Since we are dealing with three-dimensional data we may also use a three-dimensional work-item organization. When using a three-dimensional representation each work-item will be given three different indices, one for each dimension. Therefore, by getting the indices of the three respective dimensions (x,y,z), we can fetch the corresponding values in the three dimensional structure. These values may be used with the formula defined above 5.6 in order to access the voxel data corresponding to a work-item. We do not need to care about the organizations of the work-groups in this part since we will not make any use of the synchronization functions. Thus, we may let OpenCL decide on the organization of the work-items by setting the group size parameter to null.

Density map

The density map is used to determine the forces that will steer the nodes towards the areas of lower density. In this section, we will focus on the density map, we explain how it can be generated from the simulation kernel. But before proceeding to the filling of the density map we must ensure that all the values of the structure are cleared to zero. This step is crucial because we only perform incremental operations on data when filling the density map. The result of an incremental operation is an addition performed on the previous value. Therefore, if the values are not initialized to zero before this step, we will end up with erroneous values. In order to do so we must write a dedicated kernel for the initialization. This program is very simple, it just includes one single affectation of the density of the voxel to zero.

Listing 5.2: The clear density kernel

```
kernel void clearDensity(
    global  int* density,
    const int resolution)
{
    int x, y ,z;
    x = get_global_id(0);
    y = get_global_id(1);
    z = get_global_id(2);

    // Set the density of the current voxel to 0
    density[x + y*resolution + z*resolution*resolution] = 0;
}
```

One could think about initializing the density map in a kernel that performs another operation like the gradient kernel for instance. Indeed, at the end of the gradient kernel we do not need the density map values anymore. We could just clear these in the same kernel and thus spare us writing a dedicated kernel. This operation, however, would require synchronization on the global level. Before clearing the density value we must ensure that all the work-items have already performed their read or write operations on the density map if we do not want to corrupt the results. This kind of synchronization, unfortunately, is not possible on the global scale with OpenCL, it is only possible on the scale of a group. This is why the different operations of this stage are implemented in different kernels unlike the simulation stage.

After initializing the data, we must update the three dimensional structure with the new density values on each simulation update. The density value of a voxel is a value that counts the number of hair nodes located inside the voxel. As explained before this step is done during the simulation update. There is one work-item per node in the simulation kernel, thus for each node of the simulation, we will perform an incrementation of one to the density of the enclosing voxel. Because this operation is done individually and simultaneously for each node, we must pay attention to synchronization issues. Indeed, this allows several nodes, when located inside the same voxel, to modify the density of this same voxel at the same it. In such case, the final result of the simultaneous incrementations will be erroneous, since each thread will update the density without taking into account the results of the other threads. Let us remind that an incrementation is composed of three underlying instructions: The first one is an access instruction which copies the value from the memory into the register of the compute unit. The second one is an addition instruction performed on the register of the compute unit. The last instruction is an affectation instruction that copies the value from the register into the memory.

In order to help us understand this synchronization issue we illustrated it with an example: If two threads increment simultaneously a density value of 0, both will read this same value at the same time. As a result, the two independent additions will both be equal to 1. By the end of the incrementation, the two threads will be updating the exact same result into the memory. Therefore, the final density value, instead of being equal to the number of simultaneous incrementation will only be equal to 1.

One solution to this issue is to use an OpenCL atomic function. The atomic functions are very useful when working in a parallel environment. They allow executing a whole set of thread-safe operations. When such an operation is executed on a given part of the memory no other work-item will have access to this memory until the end of the execution of the function. Among the wide range of basic operations, the atomic functions offers us a way to perform thread-safe incrementation using the function atomic_inc(). In that way we can ensure the validity of the density map even if two or more work-items increment the same density value at the same time.

Blur

Once the density map is filled we may proceed to the blurring of this map. The blurring step allows smoothing the data and thus smoothing the forces that are going to be applied to the nodes. In that way we can get rid of some jittering due to big disparities in the density map and can get a more believable motion. There are many different ways to blur a data set, and whether the data set possesses two or three dimensions does not make a big difference, the techniques remain similar. To blur one single value of the grid we must sample the neighbour values and average somehow these values in order to determine the new blurred value. The simplest way to compute this average is to add the sampled values together and to divide this total by the number of sampled values. This kind of blurring technique is called the Box Blur. This technique is a bit trivial and does not take into account the distance between the evaluated values and the sampled values as Gaussian Blur does, for instance. In our case, all the sampled values have the same weight. This may produces less accurate blurring but it is easier to implement especially when dealing with three dimension blurring.

In any kind of blurring technique, the number of sampled values defines the strength of the blur. The more data is sampled, the blurrier results we get. It also increases the computing time because more sampled data requires more memory accesses. In this exercise, the number of sampled values for each voxel is at its simplest. We only sample three voxels in each dimension, which mean that we read the values of twenty-seven voxels for each evaluated voxel. In the case where we require a more accurate blur we may use several passes of this Box blur technique.

Density Gradient

Now that the values of the density map are smoothed we must determine the forces that will steer the nodes to the lower density areas. These forces are determined using the gradient of the density map. The gradient of a scalar field is a vector field of the same size that points in the direction of the greatest rate of increase. In our case, the scalar field is the density map. Thus, for each voxel, we will end up with a three-dimensional vector that will be applied as a force to all the nodes located inside a

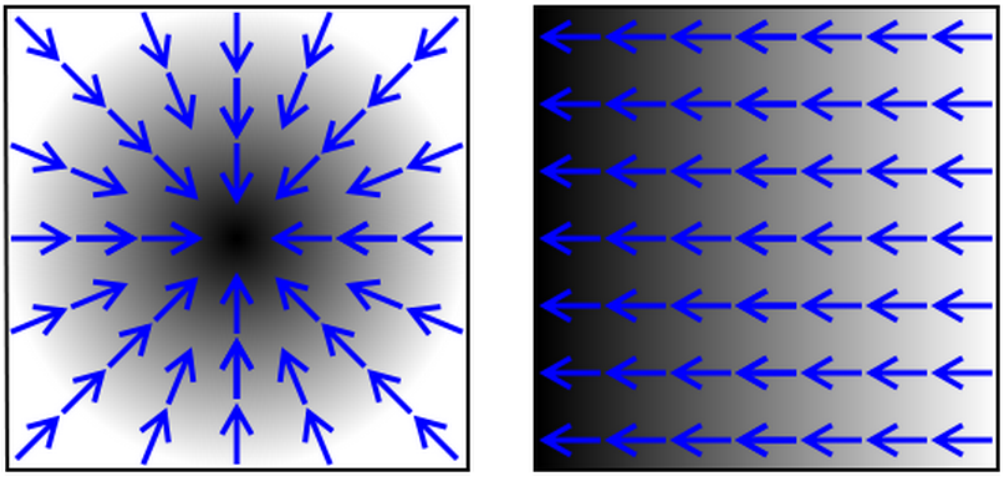


Figure 5.9: Gradients of two-dimensional scalar fields

voxel. But since the gradient is directed towards the rate of increase of the scalar field the vector forces will be directed toward the high densities areas. If we wish to move the nodes towards the lower density area we simply apply the inverse of the gradient. Figure 5.9 represents the gradient applied to a two-dimensional scalar field, the scalar field is in black and white, black representing higher values, and its corresponding gradient is represented by blue arrows.

In order to accommodate the gradient data we must declare a vector array on the device memory. Since the size of vector field is equal to the size of the scalar field, the two arrays must have the same size. The vector field is updated in a kernel program dedicated to the computing of the gradient of the density map. To determine the gradient vector of a voxel we must find the differential of this value in each dimension. The following formula defines the gradient vector.

$$\Delta = \hat{x}\frac{\delta}{\delta x} + \hat{y}\frac{\delta}{\delta y} + \hat{z}\frac{\delta}{\delta z} \tag{5.8}$$

Here Δ corresponds to the gradient vector also known as Del or Differential operator. The right part represents the respective differential value of each diminution of the field. One may find the gradient using the differential between the next and

the current value as follow. This formula, however, produce a small bias that slightly change the direction of the gradient vector. The most common technique to find the gradient vector is to use the differential between the next and the previous value and to divide this difference by two. In that way we manage to get rid of the bias and obtain the correct gradient value. Here is the pseudo code of the gradient kernel without bias:

Listing 5.3: The pseudo-code of the gradient kernel

```
gradient.x = (density[x+1][y][z] - density[x-1][y][z])/2;
gradient.y = (density[x][y+1][z] - density[x][y-1][z])/2;
gradient.z = (density[x][y][z+1] - density[x][y][z-1])/2;
```

By the end of this stage we get our force field filled with the correct values. These forces will be applied to the nodes during the next update. But before integrating these forces, we must first find the enclosing voxel of each node in order to determine the inter-hair force vector. The indices of the work-items allow us to access the gradient vector in the same way we access the density value. This vector must also be inverted before being accumulated with the other external forces of the system.

5.1.4 Guide strands interpolation

The simulation stage is an essential part of this exercise but is also very time-consuming. This is why we only simulate a few guide strands during this stage. In order to save computing time, the rest of the hair is generated out of these guide strands. Each new hair strand is generated by simple operations applied to one or several guide strands. Thus, the time needed to find the nodes' position of these strands is very low, as compared to the time needed to simulate the nodes. Although the positions of the new strands are determined from the guide strands positions, it is still likely that the new strands will collide with the scalp or other shapes. If we wish to avoid this, we would need to check for collisions during this step as well. For the sake of simplicity, this case is not implemented in this exercise: we will quite deliberately let this kind of collisions occur. There are two different ways to generate new hair strands from the guide strands. In this version, we propose a hybrid method that offers both of them. Each duplication technique, multiple or single-strand, has its own aesthetics specificities, so by combining this method with the triangular interpolation we ensure a more diverse kind of hair.

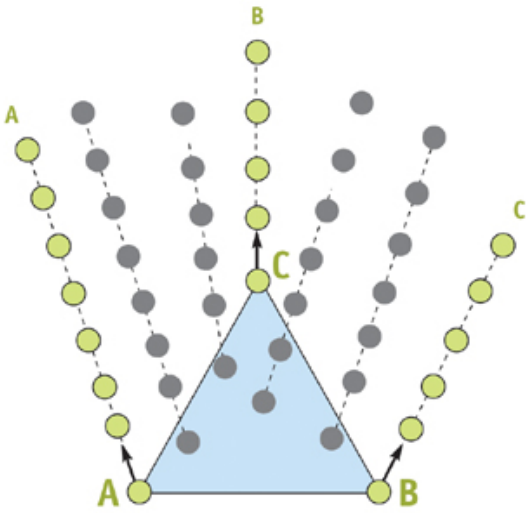


Figure 5.10: Schematic representation of the multi-strand interpolation, Nguyen et al. [8]

Multi-strand interpolation

First, we are going to focus on the method that gives better results when used alone, the multi-strand interpolation. The purpose of this technique is to generate new hair strands by interpolation of three different guide hair strands. To achieve this, we use a custom scalp model made with a modeling software package. To really facilitate this operation, the model must respect some conditions. The scalp model must be composed of triangles in order help in the research of the guide hair strands. Indeed, if the scalp only constitutes triangles, we can simply browse the triangles and generate new interpolated strands for each one of them.

This approach, similar to the one used in the Nguyen et al. [8], is more intuitive and easier than any kind of random placement of the root positions. In the first drafts of this implementation we tried to find the positions of the roots using a random uniform repartition. This approach seemed simpler than modeling a scalp because it is an easy way to get a uniform reparation of the strands over scalp. Both kinds of strands, guide and interpolated strands, were positioned using this method. In order to interpolate the generated strands, we needed to find the enclosing three guide strands for each of them and we also had to find the weight coefficients of the three guides. There are

several ways to generate uniform repartitions over simple surfaces like spheres and it is rather easy to implement. The main issue we encountered was not placing the guides or the interpolated strands, but finding the enclosing guides for each interpolated strand. Finding an enclosing triangle among a set of guide roots is not an easy task, because this set only represents points instead of triangles. Furthermore, if we wish to get accurate results we must find the smallest enclosing triangle possible. In such a case it would even be simpler to use a Delaunay triangulation algorithm in order to form regular triangles with the roots of the guide strands. For each new interpolated strand we would need to perform a hit test between the root and the triangles in order to find the enclosing three guides. Yet, this would still be much more complex than the technique we currently employ.

In this exercise, we assume that the repartition of the hair is uniform over the scalp. This is probably incorrect in reality, but easier to implement in a simulation. To make sure of this, we only use triangles of the same size for the scalp model. Indeed, if the triangles do not share the same size and if we generate the same amount of strands for every triangle, the density of the hair will not be uniform. To avoid this, we could for instance, decide to generate less hair inside the smaller triangles. This method, however, is a bit more complex than the previous one. By using triangles of the same size we can easily ensure a uniform repartition of the interpolation over the scalp. The modeling of such a shape can become quite difficult, especially without good modeling skills. To simplify this job, we define the scalp as a part of a sphere, which is one of the easiest shapes to model. However, we cannot just use any kind of sphere primitive provided by the software. The sphere must be geodesic, which means that every triangle of the sphere has the same size. Even if most of modeling software packages do not usually provide such a shape by default it is still easily achievable by using a few tricks or plugins available on the Internet. Once we get the geodesic sphere, we simply chop some vertices out of the shape until we get an aspect close to the shape of a human scalp. In this exercise we do not really care for the amount of triangles of the shape. On the other hand, we must pay attention to the number of vertices of the shape. In fact, if we use a scalp model with more vertices than guide strands, we might end up trying to interpolate some strands over a face with no guide strands defined for the vertices of the triangle.

Each new strand is thereby linked to three guide strands and also to the corresponding coefficients in order to proceed to the interpolation. These coefficients or weights are defined at random during the initialization of the simulation for each interpolated strand. Each triangle can host several interpolated strands, according to the total number of strands and the number of triangles. The values of the coefficients will determine the position of the roots within the enclosing triangle but also the positions of the rest of the nodes. We use the barycentric coordinates principle in order to determine the position of a node from the weight of the vertices. The following method allows us to generate random coefficients for each strand. Only two coefficients are necessary to determine the interpolated position. The sum of the two coefficients must be less than or equal to one in order to get a position lying inside the triangle.

This information is stored into buffers during the initialization in order to be sent to the OpenCL device for the interpolation. Next, the nodes of the generated strands are interpolated in the kernel program from the guide positions and the coefficients. We define one dedicated kernel for each kind of interpolation. We also need to declare a new OpenCL buffer to store the positions of the nodes of the interpolated strands, one for each kind of interpolation method. Just like for the guide strands, these buffers are declared as a VBO because we need to send these nodes to the graphics pipeline for display. So the total amount of hair nodes is stored in three different buffers, one for the guide strands, and two for the interpolated strands. When executing the kernel program on one node with this kind of interpolation we need to retrieve the equivalent nodes of the three guide strands in order to proceed to the interpolation. This means that when executing the kernel program for the node of local index x, the new position will be the result of the interpolation between the guide nodes of index x. Here is the kernel program of the multi-strand interpolation:

Listing 5.4: The multi-strand interpolation kernel

```
kernel void multiStrandInterpolation(
    global float4* guideNodes,
    global float4* interpolatedNodes,
    global float4*  interpolatedGuideIndices,
    global float4*  interpolatedGuideWeights)
{
    const uint global_id    = get_global_id(0);
    const uint local_id     = get_local_id(0);
    const uint group_id     = get_group_id(0);

    float4 guideIndices = interpolatedGuideIndices[group_id];
    float4 guideWeights = interpolatedGuideWeights[group_id];

    int guideIndex1 = guideIndices.x;
    int guideIndex2 = guideIndices.y;
    int guideIndex3 = guideIndices.z;

    float guideWeight1 = guideWeights.x;
    float guideWeight2 = guideWeights.y;
    float guideWeight3 = guideWeights.z;

    float4 guideNode1 = guideNodes[guideIndex1*16+local_id];
    float4 guideNode2 = guideNodes[guideIndex2*16+local_id];
    float4 guideNode3 = guideNodes[guideIndex3*16+local_id];

    float4 AB = guideNode3 - guideNode1;
    float4 AC = guideNode2 - guideNode1;

    interpolatedNodes[global_id] = guideNode1 + guideWeight1 * AB +
        guideWeight2 * AC;
}
```

We only need two coefficients in order to determine the position of a node. For a given triangle ABC, starting from the vertex A, we simply add an amount of the segments AC, and AB. The amount of segments is determined by the coefficient defined during the initialization. This is enough to give us a random position within a triangle. The repartition of these positions inside a triangle is also uniform. By the end of the multi-strand interpolation, we store the result in the dedicated VBO for the display.

Simple-strand interpolation

The second method we use for duplicating the guide strands is a bit easier and it only requires one guide strand. This technique is usually labeled as "interpolation" although it does not actually perform any kind of interpolation. The positions of the new strand are determined by simply offsetting the guide strands.

Each new duplicated hair strand is therefore linked to only one guide hair and to the offset values as well. This information is defined during the initialization process, and then stored into buffers in order to be sent to the kernel interpolation program. For each guide hair, we generate a certain amount of newy duplicated hair as well as the offset values, which are randomly chosen. The same amount of new hair is created for each guide strand. We tried two different approaches for duplicating the guide strands but we only selected one of them. In the first trial, we implemented a guide strand offset using a displacement method. We initially defined the new roots in a range around the guide roots and then applied a displacement of the difference between the two roots to the rest of the nodes. The results were convincing enough, with a low number of generated strands. This technique, however, when generating a high number of strands, tends to form clusters around the guide strands. To get better results with this technique we could, for instance, use more guide strands. However, since the simulation runs faster when using a few simulated strands a more appropriate interpolation technique should be considered.

This inspired our new technique that achieved an offset out of one single hair with better results. The purpose of this technique is to use an angular offset instead of a displacement. During the initialization process, we define random rotation values for only two axes in the scalp coordinate system. We just need two angles because we only wish to rotate the strands around the head, that is, one angle for the vertical offset and one angle for the horizontal offset. The strands are rotated according to these values afterwards, in the simple-strand interpolation kernel. By using a rotation instead of displacement we get rid of the "grouping" behaviour we previously got. This makes the interpolated strands look independent from the guide roots.

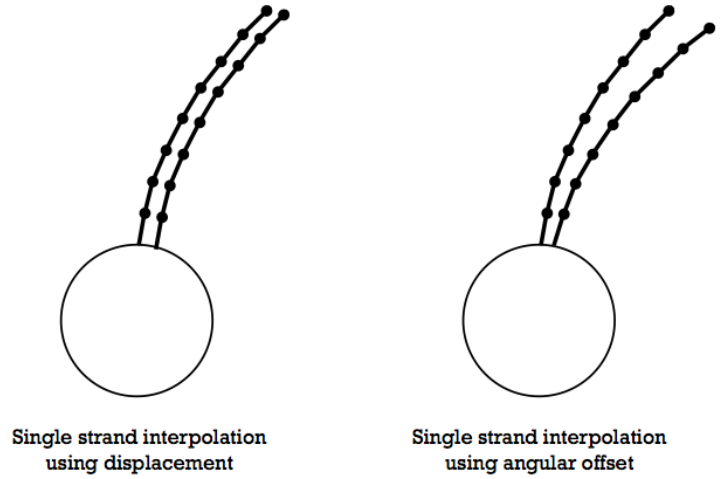


Figure 5.11: The two single-strand interpolation techniques

5.2 Rendering with OpenGL

In the second part of this chapter, we will cover the different steps related to the rendering. The rendering allows us to transform the raw data from the simulation into a realistic hair on the screen. Research in real-time hair rendering is very advanced, as compared to a few years ago. Now, the state-of-the-art techniques give very impressive results and they do so at interactive rates. The core of this project, however, is mainly related to the physical simulation. This is why, for the sake of simplicity, we will not implement any of these recent techniques. Nevertheless, we have implemented a basic rendering in order to visualize the results of the simulation. Although some important notions like the shadowing or the transparency were skipped here, the shading technique still allows us to get a believable visual representation of hair. We will go through the details of its implementation in one of the following subsection. We also cover the smoothing and the tessellation of the hair strands in the rendering section because these actions are both performed in the rendering pipeline. Since these two actions are executed before the shading in the data flow we will first focus on them.

5.2.1 Hair strands smoothing

The hair strand representation in this simulation is made of different nodes that correspond to the position of the strands at some point. The number of these nodes is deliberately low in order to accelerate the process. If we decide to simply display the strands as this, the visual aspect of the strands will suffer from this low resolution. Indeed, if we simply represent the strands by a set of lines connecting all the nodes, we will get a blocky shape made of sharp angles. The result is quite unsatisfactory and quite different from the look of real human hair. This is why we need to smooth the raw data of the simulation somehow.

Fortunately, there are some processes that allow the smoothing of such representation by using only a small amount of input data. The technique we use for this purpose is called Spline interpolation. This technique, well known in computer graphics, has the advantage of being easy to compute and stable. The principle is to determine an underlying curve behind some control points by estimating the values of the curve at any position between these points. Even if we are able to guess a curve defined by the nodes of the strands, it is still very difficult to construct a perfect curve out of it in computer graphics, in terms of performances. We only use the curve information to add more details to the strands and thus get rid of the sharp angles. Although the strands are still composed of segments, the increase of the number of nodes will give us the illusion of having a smooth and curvy visual presentation. Of course this illusion depends on the viewer's position: the closer we are to the strands, the more segments we need in order to get a smooth result.

There are many different kinds of spline interpolations and each one of them has its own properties. In the following sections we will go through some of the techniques we have implemented and explain which one is suitable for this exercise.

Cosine interpolation

The first technique we have implemented is cosine interpolation. It is also the easiest kind of interpolation after linear interpolation. To perform this interpolation between two control points, we only need the information about the two nodes and nothing

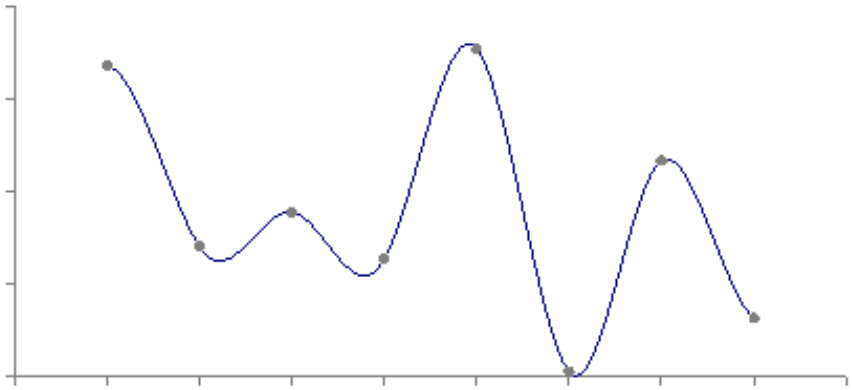


Figure 5.12: Plot of a cubic interpolation through control points

more. Later on, we will see that other kinds of interpolation may require more than two nodes for this kind of operation. Although it does not provide true continuity between the different segments, cosine interpolation still provides smooth transitions between adjacent segments.

Unfortunately, this technique tends to revert to linear interpolation when applied independently in several dimension. This of course, makes this kind of interpolation useless because we are dealing with three-dimensional interpolation in our simulation.

Cubic interpolation

Regarding the negative aspects of the cosine interpolation we had to investigate different kinds of interpolation techniques. The next one is a bit more "famous" in computer graphics and known as cubic interpolation. This is the simplest method for providing true continuity between adjacent segments. It also has the advantage of working in several dimensions. For these reasons, the cubic interpolation is quite an appealing method to solve our problem. In order to be interpolated between two endpoints, this method requires the adjacent point of each endpoint. The four control points are labelled y0, y1, y2, and y3 in the code below, which describe the cubic interpolation function.

Listing 5.5: The cubic interpolation function

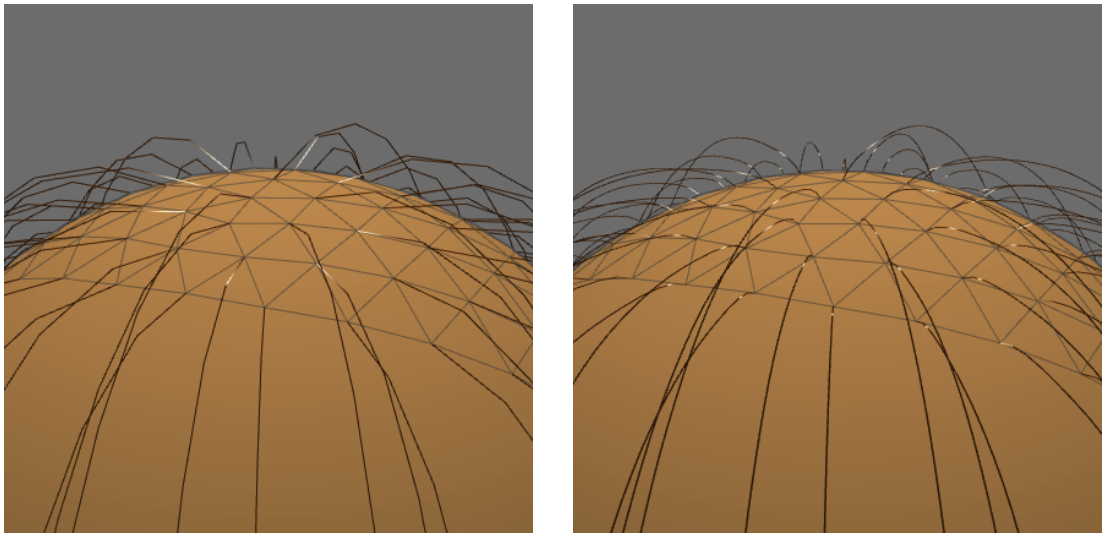
```
float CubicInterpolation(float y0,float y1,float y2,float y3,float mu)
{
    float a0,a1,a2,a3,mu2;

    mu2 = mu*mu;
    a0 = y3 - y2 - y0 + y1;
    a1 = y0 - y1 - a0;
    a2 = y2 - y0;
    a3 = y1;

    return(a0*mu*mu2+a1*mu2+a2*mu+a3);
}
```

The results of this interpolation in our case are still not satisfying because the spline is simply not smooth enough. Paul Breeuwsma [26] proposes the following coefficients for a smoother interpolated curve, which uses the slope between the previous point and the next as the derivative of the current point. In this case, the resulting polynomial is called a Catmull-Rom spline. This is the technique we use to interpolate through the strands segments in our implementation.

Listing 5.6: The Catmull-Rom coefficients

```
a0 = -0.5*y0 + 1.5*y1 - 1.5*y2 + 0.5*y3;
a1 = y0 - 2.5*y1 + 2*y2 - 0.5*y3;
a2 = -0.5*y0 + 0.5*y2;
a3 = y1;
```

In order to increase the resolution of the strands we could, for instance, perform the interpolation on the host application or via OpenCL and populate a new array of nodes with a bigger size. However, we might risk to end up with a bottleneck in our graphics pipeline. Indeed, even when using a VBO, this stage is always a bit critical. It would be wiser to send a small amount of information to the display and to generate new nodes on the fly during the rendering. The Geometry Shader, available since OpenGL 3.2, allows us to perform such an operation. In the graphic pipeline, the Geometry Shader is located right after the Vertex Shader and just before the Fragment Shader. The Geometry Shader accepts as input, the scene geometries previously transformed by the vertex shader and perform operations in order to increase their number of ver-



(a) Without smoothing (b) With smoothing

Figure 5.13: Comparison of the smoothing results

tices. The kind of input and output geometries may also be different in the Geometry Shader. In the next section, we will see that, in our case, the output of the Geometry Shader is different from the one we use as input, e.g. the line strips.

In OpenGL, there are two different primitive types available to represent lines strips, GL_LINES_STRIP and GL_LINE_STRIP_ADJACENCY. If we specify the primitive type GL_LINES_STRIP as input type for the Geometry Shader, this one will only receive two vertices for each invocation. The pair of vertices received by the shader corresponds to a segment of the overall line strip. On the other hand, if the type GL_LINE_STRIP_ADJACENCY is specified, the shader will not only receive the two vertices corresponding to a segment of the line ends but also the adjacent vertices of this segment. Since the Catmull-Rom interpolation technique requires this additional information, we must specify this type in the Geometry Shader instead of the previous one. This type must also be specified in the host application when sending the hair nodes to the display.

5.2.2 Hair strands tessellation

Directly following the smoothing operation, the segments are expanded into camera facing triangle strips. As explained in the state-of-the-art chapter, this allows more flexibility for the rendering and also provides perspective to the strands. The reason why the triangle strips must face the camera is to give the illusion of a line to the viewer, no matter what his position is. Indeed, without this trick the viewer would be able to see the 2D strips within different angles and thus notice the difference when facing the strands or when looking form side-view. Since we need to add more vertices to the strands in order to expand the strands into triangle strips we also have to perform this stage in the Geometry Shader. Therefore the primitive type GL_TRIANGLE_STRIP must be specified in the Geometry Shader in order to output triangle strips. This expansion into camera-facing triangle can easily be achieved by offsetting the nodes of the strands along the cross product of the tangent of the strand and the eye direction.

5.2.3 Hair shading

This part is the last part of our implementation and we will be covering the shading of the hair strands. Now that we have enough geometries in the scene we may proceed to the lighting of these geometries. Since the hair is represented as triangle strips we can fully enjoy the lighting techniques compared to a representation using line strips.

The first shading technique implemented for hair rendering is the Kajyia-Kay shading. This technique is still widely used, even though it can be considered as trivial compared to the one implemented by Marschner et al. [17]. The principle behind the Kajiya-Kay shader is somehow similar to Phong Shading. It is composed of three different lighting terms, ambient, diffuse and specular. The calculation of the illumination, however, is not based on the normal vector of the geometry surface, like with the Phong Shading, but on the tangent vector. This particularity allows catching the anisotropic property of human hair strands. This is why the Kajiya-Kay shading is relevant for our implementation, with the additional advantage of being relatively easy to implement. In the paragraphs below we described the different lighting terms of this technique and the source code of this shader is available in appendix A, Listing A.2.

Ambient The ambient term in analogy to Phong Shading is a constant color that describe the ambient lighting. It corresponds to the color applied to an object when this one is not directly exposed to any source light.

Diffuse Compared to Phong Shading, the diffuse term of Kajiya-Kay shader is based on the tangent instead of the normal value. The lighting is optimum when the tangent is perpendicular to the light direction and minimum when parallel to the light direction. Equation 5.9 gives us the diffuse lighting term:

$$\Psi_{diffuse} = K_d sin(t, l) \tag{5.9}$$

Where K_d is the diffuse reflection coefficient, t the tangent vector the strand and l the vector pointing to the light.

Specular Unlike the diffuse term, the Kajiya-Kay's specular term is a function of the viewer's position. It gives a shiny aspect due to direct light exposure. The lighting is optimum when the reflection of the viewer's position along the hair tangent is pointing towards the light direction. Equation 5.10 gives us the specular lighting term: vector, and p is the

$$\Psi_{specular} = K_s(t \cdot lt \cdot e + sin(t, l)sin(t, e))^p \tag{5.10}$$

Where K_s is the diffuse reflection coefficient, t the tangent vector the strand, l the vector pointing to the light, p the vector pointing to the eye, and p is the Phong exponent specifying the sharpness of the highlight.

There are many other important points to take into account during the rendering stage, such as shadowing, transparency and the ambient occlusion. But these aspects are not implemented in this project. Although this would represent an interesting challenge in terms of parallel programming and use of OpenCL, we decided to mainly focus on the simulation. Regarding the results of the simulation detailed in the next chapter it would have quite challenging to run one or several of these techniques at the same time as an efficient simulation.

Chapter 6

Evaluation

In this chapter, we will discuss the performances of our implementation, in terms of visual results but also in terms of computation speed. We estimate that the minimum limit in terms of frames per second is around 20fps. Below this rate the simulation suffers too much and looses interactivity. We must also be careful of not compromising the visual results too much for the sake of the performances. To make a good simulation there should be a balance between visual quality and performances. In the next section, we will see what represents, in our case, the right balance of these two elements.

We benchmarked this simulation on a Macbook pro 13' with a graphics chipset NVidia GeForce 320m. This may not be the most suitable configuration for such a simulation but it is sufficient enough. Although the simulation speed is a bit limited, the results are still acceptable in terms of interaction and quality. There are several parameters that can really have an impact on the performances. Some of them are unflexible, but some others may be changed in order to suit with our configuration. Here are the two alterable parameters we take into account in our benchmark: the number of interpolated strands and the number of subdivisions for each strand segments. The number of guide strands also has a great influence on the simulation performances. Since this number depends on the number of vertices of the scalp model, it is not so easy to modify this parameter freely. This is why we did not include it in our benchmark. One could say that the number of solving iteration is a important parameter as well, but compared to the other parameters the impact on the frame rate is negligible.

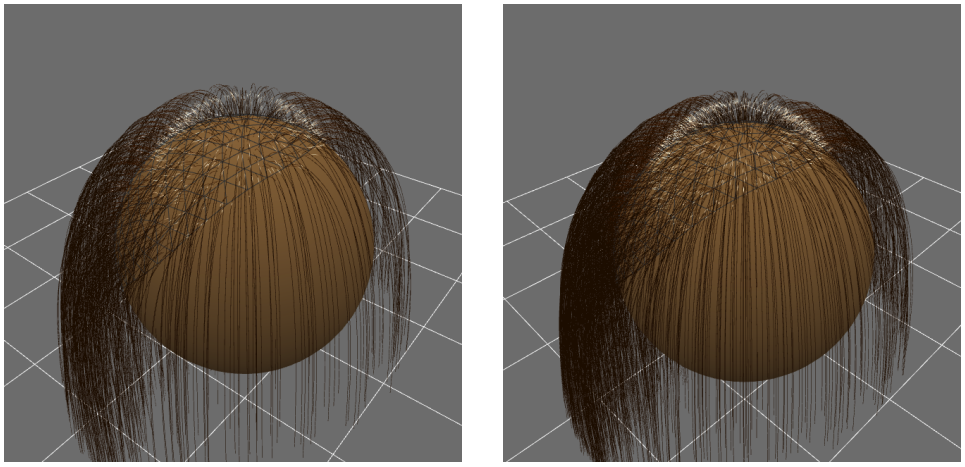
	1024	2048	4096	8192
5	26fps	24fps	23fps	19fps
10	15fps	14fps	12fps	10fps

Table 6.1: Comparison between the frame rates resultant of the different configurations

Table 6.1 shows us the average fps for the different configurations. The top row represents the number of interpolated strands and the far-left column represents the segments subdivisions. The fps may vary in function of the size of the window or the distance to the viewer. This benchmark however uses the same values all along the test. We run the simulation with 256 simulated guides and 16 nodes per guide. The full-scene anti aliasing is also enabled for this test.

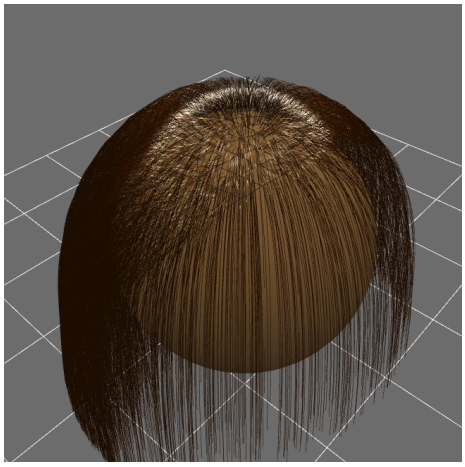
With 5 subdivisions per hair segment we get a total of 80 subsegments per strands. This is enough to give us a smooth visual appearance from a reasonable distance. If we wish to get really close to the strands we would need to increase this value. In our implementation, however, we cannot go higher than 5 without seeing a big drop of performances. This justifies why Yuksed and Tariq [15] used the Tessellation Engine for this step, as indeed, the Geometry Shader can hardly generate such a large amount of data.

There are approximatively 500 triangles in our scalp model, so with an amount of 1024 interpolated strands we get an average of two interpolated strands per triangle. This is quite few, and does not provide believable results, as with 2048. From 4096, the hair density gets a bit more interesting. It is still not as dense as a human hair would be, but the results are much more satisfying than the previous ones. One can also use a larger strip width with this configuration in order to give more the volume to the hair. When using this technique the empty spaces are filled with the larger strips and we get the illusion of a dense human hair. When using 8192 interpolated strands, the density is optimum. It is difficult to generate more strands than this, while keeping an interactive frame rate. Figure 6.1 shows us a comparison between the different densities of hair.

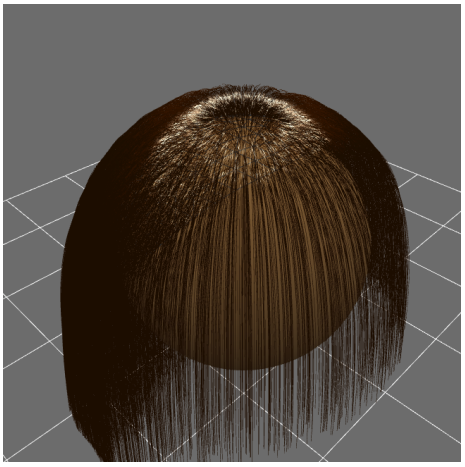


(a) 1024 interpolated strands

(b) 2048 interpolated strands



(c) 4096 interpolated strands



(d) 8192 interpolated strands

Figure 6.1: Comparison of the different hair densities

The results of this simulation can be hardly compared precisely with other techniques such as CUDA or Compute Shader, because too many parameters vary from an implementation to another. Nonetheless, the fact that the simulation is able to run at interactive frame rates proves us that OpenCL is able perform hair simulation just as well as its competitors.

Chapter 7

Conclusions & Future Work

This project presents a real-time hair simulation demo based on the GPU computing. The simulation and the rendering are respectively implemented with OpenCL and OpenGL. We used position-based dynamics for integrating the forces and solving the constraints. The inter-hair forces are computed separately by using the inverse gradient of the voxelized hair density. The smoothing, and the tessellation are done during the rendering process in the Geometry Shader. Finally, the hair is illuminated with the Kajiya-Kay shading technique.

The simulation runs efficiently at around 20 fps on a NVidia GeForce 320 with 256 simulated strands, 4096 interpolated strands and full scene anti-aliasing. Each strand is composed of 16 simulated nodes and then smoothed into 80 subsegments with spline interpolation before being expanded into camera-facing triangle strips. This configuration gives us approximately 700 000 triangles for the full hair. Although we took care of the most important aspects of real-time hair simulation, there is still some room for improvement in this implementation, in terms of level-of-details, interpolation and smoothing, collision-detection and hairstyles.

This project misses a level-of-detail (LOD) management. The LOD allows the generation of less hair when the viewer is located far away from the subject. Indeed, in such a case, the viewer does not need to get much visual details of the hair, only a few strands are enough. This feature would improve the performance, especially if the hair

is not the only subject of the scene. Since there is no other element in our scene, we did not deem it essential to implement such a technique, however, it would be a nice addition to have in a computer game for instance.

The smoothing of the hair strands only relies on the Geometry Shader. In the evaluation chapter we saw that this stage has a great impact on the speed of the simulation. To lighten the Geometry Shader of such a computation Yuksed and Tariq [15] suggested using the Tessellation Engine, which is more suitable for this task. The Tessellation Engine is the feature of the Direct3D API and therefore restricted to the windows platform only. Since we implemented our project on the mac platform, it impossible for us to use it. A similar feature is available for OpenGL and is called the Tessellation Shader. Unfortunately this feature is only available in OpenGL 4.X, while the mac platform is still limited to OpenGL 3.2. At this date, we hope that the upcoming operating systems of the mac platform will support OpenGL 4.X, in order to benefit of the advantages of the Tessellation Shader.

There is no support in our implementation for handling collisions between interpolated strands and external objects. Indeed, among the thousands of interpolated strands it is still likely to see some of them colliding with external objects. This occurs especially when using multi-strand interpolation since there is no guarantee that the results of the interpolation will not collide with any objects. The solving of such collisions is quite challenging because the strands are not simulated, which means that no constraints are actually applied to the nodes of the strand. Even if we apply a collision correction to the colliding nodes the results may still not be satisfying because the vertices below also need to be updated. Since this phenomenon only occurs when using multi-strand interpolation, one could for instance switch to single-strand interpolation when a collision is detected.

The simulation is only capable of simulating one kind of hair. During the simulation, only the motion of straight hair is taken into account. It would be quite challenging to simulate the dynamics of different types of hair because it would require using more spring constraints. However, during the smoothing of the hair, one could apply an offset to the nodes in order the get wavy results. The offset should be precom-

puted into 1D textures and applied to the perpendicular of the tangent of the strands. The offset could be manually or procedurally defined with the help of sine functions for the wave patterns. One could also add some random variations for a more realistic look.

Despite these missing elements, we managed to implemented simulation using both OpenCL and OpenGL, with conclusive results. This proves to us that OpenCL is able to run hair simulation just as well as the Compute Shader or CUDA. Accounting for the fact that we only used version 1.0 of the framework, we had all the elements and tools required for this task. Although OpenCL is still considered as less mature than its competitors, the fact that this technology is multi-platform and that the Khronos group is constantly improving the framework might restore the balance in the next few years.

Appendix A

Source Code

Listing A.1: The simulation update kernel

```
kernel void update(
    global float4* vbo,
    global float4* prev_pos,
    local  float4* shared,
    const float hairScale,
    const int numIterations,
    const float streching,
    const float bending,
    const float bendingAttenation,
    const float bendingScale,
    const float gravity,
    const float damping,
    constant  float* distances,
    constant  float* summed_distances,
    global  int* density,
    global  float4* gradient,
    const float scale,
    const int size,
    const float4 scalpPos,
    global float4* rootPositions,
    const float windX,
    const float windY,
    const float windZ,
    const int time){
```

```
// Constant values
const uint global_id    = get_global_id(0);
const uint local_id     = get_local_id(0);
const uint group_id     = get_group_id(0);
const float4 gravity    = (float4)(0.0f, g, 0.0f, 0.0f);
const float offset      = scale * -0.5f;

// Find correct stiffness
const float k1 = 1.0f - pow(1.0f-streching, 1.0f/ numIterations );
const float k2 = 1.0f - pow(1.0f-bending, 1.0f/ numIterations );

// Store values into private memory
float4 pos = vbo[global_id];
float4 p_pos = prev_pos[global_id];
float4 velocity = (pos-p_pos) / TIME_INTERVAL;
float4 wind = (float4)(windX, windY, windZ, 0);

// Find the voxel index
int3 volumeIndex = FindVolumeIndex(pos, size, scale);

// Find the flatten index
int flattenVolumeIndex =  volumeIndex.x + size * volumeIndex.y + size
    * size * volumeIndex.z;

// Fetch interHairForces from gradient map
float4 interHairForces = gradient[flattenVolumeIndex];

// If the node belongs to the first segment, do not update, apply the
    scalp transform instead
if(local_id == 0 ){
    shared[local_id] =  scalpPos + rootPositions[group_id] * (1-
        distances[0] * hairScale);
}
else if(local_id == 1 ){
    shared[local_id] =  scalpPos + rootPositions[group_id];
}
```

```
// Add verlet integration + external forces
else{
    // Determine the strength of the wind according to the tangent
    float4 v;
    if(local_id == 15) { v = pos - vbo[global_id-1]; }
    else { v = (vbo[global_id+1] - vbo[global_id-1])*0.5f ; }
    float angleFactor = length(cross(normalize(wind), normalize(v)));

    // Find turbulences
    float turbulence = turbulence3d(pos+velocity, 100000000.0f, 10.0
        f, time*2.0f, 4.0f) ;
    float turbulence1 = turbulence3d(pos+velocity,1000.0f, 10.f,
        time, 4.0f) ;
    float turbulence2 = turbulence3d(pos+velocity, 10000.0f, 10.0f,
         time*0.2f, 4.0f);
    float4 turbVec = (float4)(turbulence, turbulence1, turbulence2,
        0.0f) * 50.0f;
    wind *= turbVec * angleFactor;

    // Viscous drag
    float4 drag -= damping * velocity;

    // Accumulation of the external forces
    float4 accel = gravity + wind + drag - interHairForces;

    // Verlet integration
    shared[local_id] = (2.0f*pos - p_pos) + accel *
        TIME_INTERVAL_SQUARE ;
}

// Constraint solving
for(int i = 0; i < numIterations; i++){
    // Sphere collision constraint
    if(local_id > 1){
        shared[local_id] = SphereCollisionConstraint(shared[local_id
            ], scalpPos, 1);
    }

    barrier(CLK_LOCAL_MEM_FENCE);
```

```
        // If id is even solve distance constraints
        if(local_id \%2 == 0){
            DistanceConstraint(shared, local_id  , local_id + 1,
                hairScale*distances[local_id], k1);
        }

        barrier(CLK_LOCAL_MEM_FENCE);

        // If id is odd solve distance constraints
        if(local_id < 15 && local_id \%2 != 0){
            DistanceConstraint(shared, local_id  , local_id + 1,
                hairScale*distances[local_id], k1);
        }

        barrier(CLK_LOCAL_MEM_FENCE);

         // Bending constraints
        if(local_id > 1 && local_id < 16){
            float index = local_id -1 ;
            float d = bendingScale*hairScale*summed_distances[index];
            DistanceConstraint(shared, 0 , local_id , d, k2 * (1.0f / pow(
                index, bendingAttenation) /2.0f) );
        }
    }

    // Find the voxel index after simulation
    volumeIndex = FindVolumeIndex(shared[local_id], size, scale);

    // Find the flatten index
    flattenVolumeIndex =  volumeIndex.x + size * volumeIndex.y + size *
        size * volumeIndex.z;

    // Increase voxel density value
    atomic_inc(&density[flattenVolumeIndex]);

    // Copy value to global memory
    vbo[global_id] = shared[local_id];
    prev_pos[global_id] = pos;
}
```

Listing A.2: The hair fragment shader

```
#version 150
uniform vec3 hairColour;
uniform vec3 specColour;

out vec4 fragColour;

in vec4 tangent;
in vec4 lightDir;
in vec4 viewDir;

void main ()
{
    vec3 T = normalize(vec3(tangent));
    vec3 L = normalize(vec3(lightDir));
    vec3 V = normalize(vec3(viewDir));

    float TdotL = dot(T,L);
    float TdotV = dot(T,V);

    // The diffuse component
    float kajiyaDiff = sin( acos(TdotL) );
    if(kajiyaDiff < 0 ) { kajiyaDiff = 0; }
    kajiyaDiff = pow(kajiyaDiff, 10);

    // The specular component
    float kajiyaSpec = cos( abs( acos( TdotL ) - acos( -TdotV ) ) );
    if(kajiyaSpec < 0 ) { kajiyaSpec = 0; }
    kajiyaSpec = pow(kajiyaSpec, 100);

    // The fragment color
    fragColour = vec4(hairColour*0.6 + hairColour*0.5 *kajiyaDiff +
        specColour*kajiyaSpec, 0.725);
}
```

Bibliography

[1] R. E. Rosenblum, W. E. Carlson, and E. Tripp, "Simulating the structure and dynamics of human hair: Modelling, rendering and animation," *The Journal of Visualization and Computer Animation*, vol. 2, no. 4, pp. 141–148, 1991.

[2] D. Baraff and A. Witkin, "Large steps in cloth simulation," in *Proceedings of the 25th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '98, (New York, NY, USA), pp. 43–54, ACM, 1998.

[3] K.-i. Anjyo, Y. Usami, and T. Kurihara, "A simple method for extracting the natural beauty of hair," *SIGGRAPH Comput. Graph.*, vol. 26, pp. 111–120, July 1992.

[4] S. Hadap and M. N. Thalmann, "Modeling Dynamic Hair as a Continuum," *Computer Graphics Forum*, vol. 20, no. 3, pp. 329–338, 2001.

[5] J. T. Chang, J. Jin, and Y. Yu, "A practical model for hair mutual interactions," in *Proceedings of the 2002 ACM SIGGRAPH/Eurographics symposium on Computer animation*, SCA '02, (New York, NY, USA), pp. 73–80, ACM, 2002.

[6] T. Jakobsen, "Advanced Character Physics," in *Game Developers Converence Proceedings*, pp. 383–401, CMP Media, Inc., 2001.

[7] M. Müller, B. Heidelberger, M. Hennix, and J. Ratcliff, "Position based dynamics," *J. Vis. Comun. Image Represent.*, vol. 18, pp. 109–118, Apr. 2007.

[8] H. Nguyen and W. Donnelly, "Hair Animation and Rendering in the Nalu Demo," in *GPU Gems 2* (M. Pharr and R. Fernando, eds.), ch. 23, Addison Wesley Professional, Mar. 2005.

[9] M. Oshita, "Real-time hair simulation on gpu with a dynamic wisp model," *Comput. Animat. Virtual Worlds*, vol. 18, pp. 583–593, Sept. 2007.

[10] W. Rungjiratananon, Y. Kanamori, and T. Nishita, "Chain shape matching for simulating complex hairstyles," *Computer Graphics Forum*, vol. 29, no. 8, pp. 2438–2446, 2010.

[11] S. Tariq and L. Bavoil, "Real time hair simulation and rendering on the gpu," in *ACM SIGGRAPH 2008 talks*, SIGGRAPH '08, (New York, NY, USA), pp. 37:1–37:1, ACM, 2008.

[12] C. K. Koh and Z. Huang, "A simple physics model to animate human hair modeled in 2d strips in real time," in *Proceedings of the Eurographic workshop on Computer animation and simulation*, (New York, NY, USA), pp. 127–138, Springer-Verlag New York, Inc., 2001.

[13] A. Daldegan, N. M. Thalmann, T. Kurihara, and D. Thalmann, "An integrated system for modeling, animating and rendering hair," in *COMPUTER GRAPHICS FORUM*, pp. 211–221, 1993.

[14] F. Bertails, C. Ménier, and M.-P. Cani, "A practical self-shadowing algorithm for interactive hair animation," in *Proceedings of Graphics Interface 2005*, GI '05, (School of Computer Science, University of Waterloo, Waterloo, Ontario, Canada), pp. 71–78, Canadian Human-Computer Communications Society, 2005.

[15] C. Yuksel and S. Tariq, "Advanced techniques in real-time hair rendering and simulation," in *ACM SIGGRAPH 2010 Courses*, SIGGRAPH '10, (New York, NY, USA), pp. 1:1–1:168, ACM, 2010.

[16] J. T. Kajiya and T. L. Kay, "Rendering fur with three dimensional textures," *SIGGRAPH Comput. Graph.*, vol. 23, pp. 271–280, July 1989.

[17] S. R. Marschner, H. W. Jensen, M. Cammarano, S. Worley, and P. Hanrahan, "Light scattering from human hair fibers," *ACM Trans. Graph.*, vol. 22, pp. 780–791, July 2003.

[18] C. Yuksel and J. Keyser, "Deep opacity maps," *Computer Graphics Forum (Proceedings of EUROGRAPHICS 2008)*, vol. 27, no. 2, pp. 675–680, 2008.

[19] A. Zinke, C. Yuksel, A. Weber, and J. Keyser, "Dual scattering approximation for fast multiple scattering in hair," *ACM Trans. Graph.*, vol. 27, pp. 32:1–32:10, Aug. 2008.

[20] K. Ward, F. Bertails, T.-Y. Kim, S. R. Marschner, M.-P. Cani, and M. C. Lin, "A survey on hair modeling: Styling, simulation, and rendering," *IEEE Transactions on Visualization and Computer Graphics*, vol. 13, pp. 213–234, Mar. 2007.

[21] S. Lesser, "Fast hair simulation and rendering using cuda and opengl," 2011.

[22] "Compute shader overview." `http://msdn.microsoft.com/en-us/library/windows/desktop/ff476331(v=vs.85).aspx`. Last accessed 10/8/2012.

[23] "Cuda zone." `http://www.nvidia.com/object/cuda_home_new.html`. Last accessed 10/8/2012.

[24] "Opencl prgramming guide for mac." `https://developer.apple.com/library/mac/#documentation/Performance/Conceptual/OpenCL_MacProgGuide/Introduction/Introduction.html`. Last accessed 10/8/2012.

[25] "Interpolation methods." `http://paulbourke.net/miscellaneous/interpolation/`. Last accessed 10/8/2012.

[26] "Cubic interpolation." `http://www.paulinternet.nl/?page=bicubic`. Last accessed 10/8/2012.