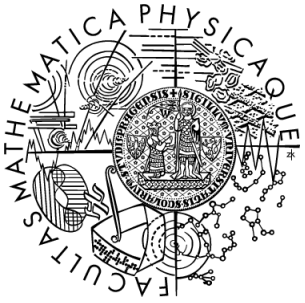
Charles University in Prague
Faculty of Mathematics and Physics

BACHELOR THESIS



Šimon Soták

Simulating Hair with the Loosely-Connected Particles Method

Department of Software and Computer Science Education

Thesis supervisor: Mgr. Petr Kmoch
Study program: Computer Science, Programming

2010

I would like to thank my supervisor Mgr. Petr Kmoch for his advice on the topic. My gratitude also goes to Martin Ščavnický, mainly for his programming tips.

I declare that I have written my bachelor thesis independently and solely by using the cited sources. I agree with lending and publishing of the thesis.

Prague, 3 August 2010 Šimon Soták

Contents

Název práce: Simulace vlasů pomocí metody Loosely-Connected Particles
Autor: Šimon Soták
Katedra (ústav): Kabinet software a výuky informatiky
Vedoucí bakalářské práce: Mgr. Petr Kmoch
e-mail vedoucího: petr.kmoch@mff.cuni.cz

Abstrakt: Tahle práce prezentuje implementaci metódy pro animaci vlasů nazvanou Loosely Connected Particles (LCP). Metodu byla aktualizována několika moderními postupy. Prvně byly implementovány dvě varianty paralelního zpracování simulace. Ukázali jsme, že práce je rozdělena rovnoměrne a teda dynamická distribuce není potřebná. Následně jsme aplikovali metódu Deep Opacity Maps pro stínování vlasů na metodu LCP a zavedli jsme alfa-ořezávání pro eliminaci artefaktů. Poskytujeme přehledné uživatelské rozhraní pro ladění parametrů simulace, a pro stříhaní vlasů pomocí myši. Pro implementaci jsme zvolili jazyk C# a platformu .NET 3.5.

Klíčová slova: simulace a zobrazování vlasů, částicový systém, paralelizace

Title: Simulating Hair with the Loosely-Connected Particles Method
Author: Šimon Soták
Department: Department of Software and Computer Science Education
Supervisor: Mgr. Petr Kmoch
Supervisor's e-mail address: petr.kmoch@mff.cuni.cz

Abstract: This thesis presents an implementation of the Loosely Connected Particles (LCP) method of hair animation proposed by Bando et al. We updated the method with several modern approaches. Firstly, we implemented two variations of parallel processing for the simulation which differ in work distribution among threads. The results indicate that work is distributed evenly, and thus dynamic distribution is not needed. Secondly, we applied the Deep Opacity Maps method of hair shadowing on the LCP and introduced alpha thresholding to eliminate artifacts. We supply a clean user interface for parameter tweaking with instant response and an easy-to-use interface for cutting hair using mouse controls. We discuss the suitability of using C# with .NET for developing a performance-heavy application.

Keywords: hair, simulation, rendering, particle system, parallelization

Chapter 1

Introduction to Hair Animation

1.1 Motivation

Creating complex and believable virtual characters has always been one of the ultimate goals of computer animation. Perhaps one of the greatest challenges in this area is producing realistic hair. This is simply due to hair's complex nature. It usually consists of hundreds of thousands of individual strands which interact with one another. However, this large number is not the only obstacle for realistic hair simulation. A hair strand itself is unlike other well-studied physical bodies such as rigid bodies or fluids. A strand is stiff but bendable, and strands tend to clump, but may separate at any point.

A natural consequence of these characteristics of hair is that various simulation and rendering approaches emerged in the past decades. Some of them aim for the highest level of realism and simulate each strand individually, in order to model individual hairs' friction, static attraction, bending, torsion or curliness. Naturally, the consequence is usually a high performance cost.

Other methods, on the other hand, strive for interactive or even real-time frame rates, and must therefore make compromises. These approaches are based on the observation that hair behaves collectively, suggesting that we do not necessarily need to simulate every single strand to produce believable results. One of these emergent aggregate phenomena hair produces is clumping. The approaches based on this assumption introduce the notion of 'guide strands'. The other, non-simulated strands are generated by

interpolating between the guide strands.

1.2 Conventional approaches

Selle et al. proposed a mass-spring model for simulating full hair geometry [SLF08]. This means that hair-hair interactions are actually modeled at the scale of single strands with parameters such as friction or static attraction. This approach is therefore very accurate and produces stunning visual results, but at very high simulation times (i.e. 10 to 30 minutes per frame).

One of the approaches that rely on aggregate hair behavior and use a notion of guide strands is the work of Bertails et al. [BAC+06]. They introduced a model where each guide strand is represented by a Super-Helix, a piecewise helical rod. These elastic rods are built upon the Cosserat and Kirchhoff theories, which account for behavior of hair strands such as bending and twisting. Notably, they demonstrated simulation of curly hair, which is a tricky phenomenon in general. However, Bertails reported that for real-time frame rates, only up to 10 strands can be simulated.

Today's state-of-the-art in real-time simulation of hair using guide strands is the work of Tariq and Bavoil from NVIDIA [TB08]. They combined a relatively simple simulation model with a number of cutting-edge renderings techniques to produce beautifully-looking human hair. The hair is simulated as a particle constraint system. To model the inelasticity of hair, Tariq imposes distance constraints between hair vertices (particles). 2D angular forces are applied at each vertex in order to maintain hair shape. Hair-hair collisions are handled in a grid-based framework. The hair particles (vertices) are pushed out of voxels with high density, thus preserving volume of hair. Rendering is done by tessellation of a strand's vertices into a finely curved hair. Strands generated at render time are produced by combining inter-strand interpolation with clump-based interpolation. All work is done on the GPU, from simulation, through tessellation and interpolation of the strands to shadowing and shading. Tariq and Bavoil make use of the newest advances in graphics hardware, by means of DirectX 11 and GPGPU. They achieved a frame rate of 41 for 166 simulated guide strands.

1.3 Fluid approaches

A completely different kind of approach is based on a paradigm shift proposed by Hadap and Magnenat-Thalmann [HMT01]. They treat hair volume as a continuum, and model hair interactions as fluid dynamics. The justification for this assumption is that hair mass produces emergent phenomena such as pressure and density in each point of the volume, like fluids. Hadap used smoothed particle hydrodynamics (SPH) as a numerical model. Hair-hair collisions are approximated by pressure term from the continuum mechanics whereas friction forces are estimated by viscosity. However, individual strands are still simulated as a rigid multibody serial chain with a model for inertial and stiffness dynamics.

As a result, utilizing fluid dynamics captures complex interactions of strands with certain limitations. For instance, the SPH does not produce hair behavior such as clumping. As far as individual strands are concerned, using multibody chains results in individual hairs maintain their stiffness, but this model is limited to straight hair. This approach still require a large number of strands to be simulated, and therefore the computation times remain high (2 minutes per frame).

1.4 Loosely-Connected Particles

This thesis is primarily concerned with the Loosely-Connected Particles (LCP) method [BCN03]. It was proposed by Bando et al., who brought Hadap's proposition to treat hair as a continuum even further. Hair volume is sampled by particles while the strands are not modeled at all. Hair dynamics is simulated by particle interactions which include not only hair-hair interactions, but also spring forces maintaining the overall stiffness of hair. By making these connections loose, they allow for dynamic splitting and merging of hair.

Bando et al. first initialize the particles by distributing them over a polygonal scalp. For this, they use the cantilever beam simulation method [AUK92] and distribute the particles evenly in the hair volume based on the Poisson disk sampling. Based on these initial positions and directions of the particles, the spring connections for lengthwise coherence are established. The consequence of this is that the initial distribution of particles and their directions influences the behavior of hair in the simulation.

For the simulation, Bando et al. borrow the idea of SPH to use a smooth-

ing kernel for defining various properties of the hair volume, such as density. They choose a kernel in which only particles a certain distance away can contribute to the properties, thus making neighbor-search efficient (by means of a voxel grid). In each step, they activate spring forces arising from the lengthwise connections. Additionally, they apply particle interaction forces, including attraction/repulsion, inelastic collisions and friction. Other forces, such as gravity, air friction and head collisions are also modeled. They represent wind as a set of SPH particles, interacting with the hair particles by drag forces.

For rendering of the hair, volume rendering techniques are used. Since hair is represented by the distribution of particles, Bando et al. use place textured billboards at the positions of the particles. These textures contain opacity information, so alpha blending is used, requiring sorting the particles in every pass. For shading, the Kajiya-Kay model [KK89] with an added directionality factor from the Goldman model [Gol97] is used. They make several more adjustments to make up for the relatively small number of particles.

Bando et al. reported interactive frame rates when simulating 2000 particles, showing collective hair behavior such as dynamic splitting and merging, maintaining volume, or hair-torso interactions. However, the fact that hair is represented by particles is clearly visible in the videos they presented, where an occasional billboard floats, separated from the remaining volume.

1.5 Deep Opacity Maps

A lot of work has been done in matters concerning only specific parts of the hair animation process. An important aspect in the appearance of hair is self-shadowing. Classic approaches such as opacity shadow maps [KN01] provide flawed visuals when using few layers while being costly with a large number of layers. Yuksel and Keyser proposed a method specifically suited for hair rendering called the Deep Opacity Maps (DOM) [YK08]. His approach is, in principle, very similar to opacity shadow maps. That is, to divide the hair volume into layers and accumulate in each layer the opacities of the hair belonging to that layer (or to previous layers). However, in DOM, the layers mimic the shape of the hair, while opacity shadow maps use regularly spaced, planar layers.

Yuksel proposes an efficient way to create the layer texture, using modern hardware. In the first pass, a depth map of hair is rendered from the light's

point of view. This defines the 'shape' of the hair, thus serving as the separator between the layers. In the second pass, hair is again rendered as seen from the light. However, depending on the distance of the hair from the light, it is assigned to a layer, based on the separator from the first pass. All work is done in the fragment shader. The layers of the opacity map are represented by color channels of a texture, reserving the alpha channel for depth (separator). Additive blending is then used to accumulate the contributions of rendered hair into the layers in the texture.

In the final pass, from the camera's POV, the hair position is transformed to light coordinates, just like in a typical shadow mapping algorithm. The amount of transmitted light on this hair is produced by interpolating between the two layers corresponding to the depth of this fragment in light coordinates.

By using only one texture, we obtain 3 layers (R, G and B channels), which would be desperately little for opacity shadow maps. However, Yuksel demonstrated that 3 layers is sufficient in most cases for DOM, producing little to no layering artifacts, as opposed to opacity shadow maps.

1.6 Goals

The primary objective of this work is to create a hair simulation program using the LCP method. The main purpose of this implementation is to update the method with modern technologies and see how well it performs on today's hardware.

The program shall supply a user interface for modifying parameters to allow for studying the effects of different parameter values. Where possible, changing parameter values should be allowed without having to restart the simulation.

There are three specific matters we plan to address, concerning updating the LCP method with advanced features. They are:

1. Implement some form of parallel processing for the simulation.
2. Propose a suitable application of Deep Opacity Maps for the LCP method of hair rendering.
3. Include a usable interface for cutting hair in the application.

There is another, albeit minor, objective for this work. The application will be written in C#, utilizing the Microsoft .NET Framework. This is quite atypical for a graphics application which requires high performance. Our aim is, therefore, to assess the extent to which performance, effectiveness and other aspects of development will be affected by this language and framework.

Chapter 2

Implementation

2.1 Environment and Technologies

The application is written in C# 3.0, using .NET Framework 3.5 in Microsoft Visual Studio 2008. Although C# may seem like an unusual choice for developing performance-heavy applications, it was chosen partly because we would like to test its 'unsuitability' and point out the main reasons for this claim, if any. After all, C# is a robust, modern, object-oriented language with strong support and frequent updates.

We utilize the GPU for rendering, by means of the OpenGL standard. We need, however, a wrapper for the OpenGL function calls. The Open Toolkit library [1] provides OpenGL bindings for C# as well as several simple math tools for tasks associated with computer graphics. These include vector and matrix structures, and their typical methods.

No additional tools or libraries are used.

2.2 Design Outline

There are several logically distinct sides to our application, namely: Entities, Initialization, Simulation, Rendering and User Interface. Implementation of each of these aspects, as well as their relation to each other will be discussed in the following sections.

[1] http://www.opentk.com

2.3 Entities

In the application, entities represent the actual data that is initialized, simulated, rendered and interacted with. The primary entities we need to model are hair, air and a body (a head with a neck and shoulders; we will refer to this simply as 'head'). The secondary (auxiliary) entities are mainly data structures which increase efficiency or aid the simulation in some other ways. These include, for example, particle pairs and a voxel grid. The entities are structured hierarchically through composition, in a logical manner (see Figure 2.1). There is also a super-entity Scene, which serves as the root of this hierarchy.

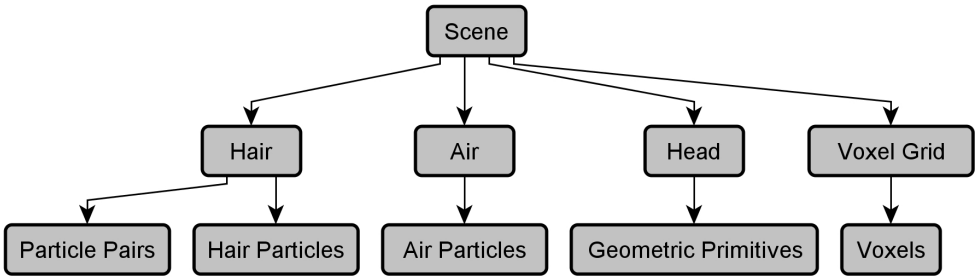


Figure 2.1: Entity Hierarchy

Inheritance is used to define entities that are actually simulated (i.e. they have positions, velocities, and forces acting on them). The inheritance graph is shown in Figure 2.2. As shown, the class `PointMass` defines the basic linear properties of a physical body, such as mass, position or velocity and already at this level has a method `IntegrateForce`. This method applies the Runge-Kutta 4 integration method to advance the body one simulation step forward in time.

The class `RigidBody` adds angular properties such as angle, angular velocity or angular acceleration to `PointMass` and overrides the virtual method `IntegrateForce` to incorporate angular integration.

Similarly, a general `SPHParticle` adds density and current neighbor particles while a `HairParticle` adds direction and sustained neighbors for lengthwise coherence. Make a strong note that these are two logically distinct kinds of 'neighbor' particles. For more detail, see [BCN03].

We also define the `Hair` and `Air` fluids through inheritance. The abstract class `Fluid<SPHParticle>` provides common methods such as density calculation.

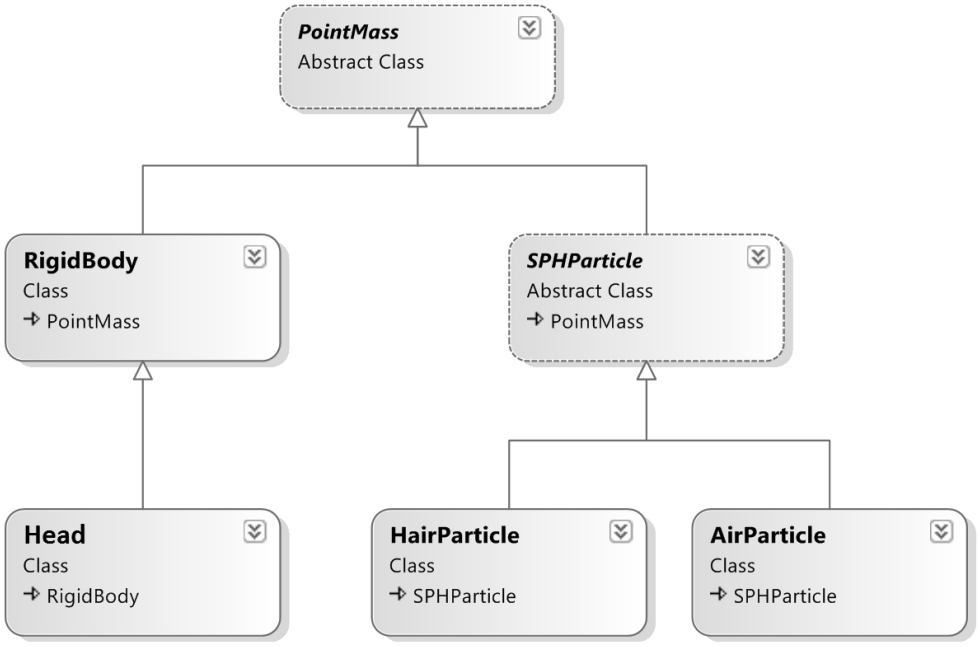


Figure 2.2: Inheritance Hierarchy (not all members are shown)

These inheritance hierarchies provide a useful way of treating entities on different levels of abstraction. For example, the voxel grid only works with the interface of a `SPHParticle`, no need to specify whether it's hair or air. This allows for lower dependency.

The head-neck-shoulders is hardcoded to be 1 sphere for the head and two cylinders for neck and shoulders (plus two additional spheres for the two ends of the shoulders, so that together the shoulders are represented by a capsule).

2.4 Initialization

We incorporate initializer classes for the entities scene, hair, air and head. Their responsibility is to create and initialize these objects. Once again, their structure is hierarchical and its functioning is demonstrated in Figure 2.3.

The figure seems a bit overwhelming, but it only aims to show that just as the entities are structured in a hierarchy, so are their corresponding initializers. The call to create and initialize the whole scene triggers calls to

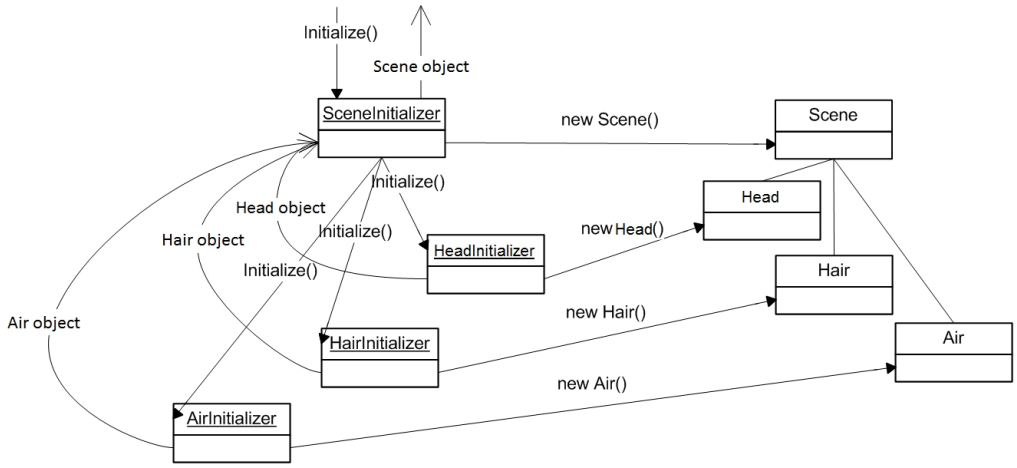


Figure 2.3: Initialization

create and initialize the components, which are then grouped together into the final `Scene` object and returned.

The initialization process is very straightforward for most of the entities. For the head, the geometric primitives are created and positioned. The air is distributed into a cylinder in front of the head. But the hair initialization is trickier.

In the hair initialization process, the most important part is particle distribution. There are two classes that are used for hair particle distribution over the head. One of them is `CantileverBeamDistributor` and is programmed according to the Cantilever beam simulation method proposed by [AUK92].

We currently use the `SemiCantileverBeamDistributor` class for hair distribution. It was created as a result of a programming error. However, by luck, it proved to have satisfactory visual results, and is therefore favored over the 'true' cantilever method. Describing how this erroneous method works is not easy, obviously, since it was not even intended to work. Nevertheless, a short description follows.

Before the true cantilever beam simulation was implemented, there was a heuristic distributor, just for simulation testing purposes. It distributed the hair particles by simulating particle motion. The position of a particle was determined by 'tossing' the particle from the root position on the scalp in the direction of the particle's root direction. The particle's motion was

'simulated' until it traveled a distance of l, where l is a random value from the interval (0, *hair length*). It was not intended to be visually convincing.

The faulty class that is used currently emerged as a hybrid of this heuristic distribution and of the actual cantilever distributor. It combines both methods. For a particle with the length parameter l, it uses the heuristic distributor for the first half of the length, and the cantilever distributor for the second half of l. Surprisingly, it provides reasonable results, even better than those produced by the genuine cantilever beam simulator. In fact, the results we obtained from using the `CantileverBeamDistributor` were quite unconvincing, in our opinion. However, there were a lot of input parameters to the genuine method, which we, perhaps, failed to tweak enough to achieve a good result.

From the initial distribution of hair particles, the smoothing length h_1 and h_2 are computed. After this, the particle connections are established according to the LCP method and the data structures are prepared.

2.5 Simulation

The simulation functionality is realized by methods of the entities hair, head and air. Calling the method `Step()` of the class `Scene` triggers a top-down propagation of method calls in the entity structure.

In each step, the scene tells hair and air to apply forces on selves (pressure forces, etc.), then calls the head to apply collision forces on the fluids, then itself applies hair-air interaction forces, and finally, it applies movement forces triggered by the user from the GUI. After all this, the forces are integrated into velocities and velocities into positions either by Euler integration, or using the well-known Runge-Kutta 4 method.

In between these steps, an auxiliary data structure for efficient space partitioning — a voxel grid — is updated and inquired. We will talk about the voxel grid in Section 2.7.

There are several steps `Hair` takes when applying forces on the hair particles. They will be described in order of calling. For how and why the force calculation works, refer to [BCN03] and the source code.

1. `prepareParticles()`

   Sets the forces of particles to zero and calculates values necessary for the following computations as specified by the method.

2. `prepareParticlePairs()`

   The same, but for lengthwise particle pairs.

3. `applySpringForces()`

   Computes and adds the spring forces which account for the lengthwise coherence of the hair fluid. This is where the particle pairs are utilized.

4. `applyNeighborForces()`

   Applies SPH fluid forces (maintaining density), collision and friction forces. This is where the voxel grid's neighbor search is utilized.

5. resetRootParticleForces();

   Since the root particles need to remain in the head, they do not move, and thus we set the forces acting on them to zero.

The air particles also exert forces on one another, but only one kind – the SPH pressure forces maintaining average density. This is completely analogous to the hair's attraction-repulsion force activation.

Note that most of the particle interactions are symmetrical. The calculations can be made much more effective taking this into account.

For the head-fluid physical interaction, a simplified model of head, neck and shoulders is used. The head is represented by a sphere for the head, a cylinder for the neck and a capsule (cylinder with two spheres at its ends) for the shoulders.

We check if any of the particles are too close to one of the geometric shapes. If any particle gets a certain distance away from the surface of the shape, its velocity component that is perpendicular to the shape's surface is damped. If the particle still manages to penetrate the surface, a repulsive force pushes it out of the shape (see Figure 2.4). The static class `Geometry` is used for point-shape collision checking. Most of these methods are either straightforward or refer to an outside source in the code.

The purpose of air's presence in the simulation is to interact with hair and thus further demonstrate hair dynamics. According to [HMT01], for each hair particle, we need to calculate the 'velocity estimate of air' at the particle's position and vice versa for the air. However, it is not specified how. Therefore, an intuitive estimation was used. We obtain the estimate by integrating the velocities of the air particles around the hair particle (the

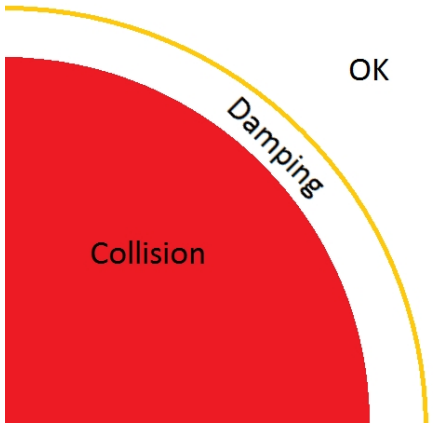


Figure 2.4: Shape interaction stages

individual velocities are smoothed by the kernel with respect to the distance of the two particles) and use the equation from the method.

There are two options programmed for how to integrate the forces and velocities of the physical entities: Euler integration and RK4 integration. The former is very simple and straightforward, but also numerically unstable and imprecise. That is why Runge-Kutta 4 method will be discussed.

The RK4 is a well-known method for approximating ordinary differential equations and we won't go into detail about how it works. What we need to know is that it has several stages. First, we need to initialize some values. Then, we apply the forces on all physical entities and make a small Euler step. RK4 stores these integrated values. We do this 4 times for different time steps.

After that, in the RK4 finalization step, the stored values are used in such a manner that we obtain the final velocities and positions of the entities at the end of the whole step. This one big RK4 step is actually more precise than 4 smaller Euler steps together (otherwise the process would be pointless).

The way all of this can be achieved for every single particle is by calling a function in the `Scene` class, which tells the head, hair and air to do a RK4 step. These, in turn, call the respective method for every single particle, telling it to do an integration step. Head is not a particle, so it does the step directly.

2.6 Rendering

We only need to render the hair and the head, since air is transparent. For rendering, two additional entities are required: the camera and the light source. These entities only appear in the rendering classes and are not simulated.

With an approximate method such as LCP, hair rendering needs to make up for the simplified physical model. Since hair strands are not modeled at all, Bando et al. proposed to use a splatting method with billboards, which fits the particle representation of hair well. A billboard is a camera-oriented quad on which we map a hair texture. We utilize the vertex shader for orienting the billboards at the camera. For the hair texture, the one presented by Bando et al. is used (Figure 2.5). It compensates for the small number of particles used to represent hair (compared to the actual number of strands) by imitating the appearance of several strands.

Figure 2.5: The hair splat texture used.

However, the billboards are not opaque – the texture has an alpha channel. For correct transparency, we render the hair using alpha blending. In order to do this, the hair particles need to be sorted according to their distance from the camera each time they are to be rendered. We first render all the fully opaque objects in the scene and then disable depth-buffer writing. Afterwards, we render the hair quads from the furthest to the nearest and only then enable depth-buffer writing again.

Hair shading is implemented in the fragment shader utilizing the Kajiya-Kay model [KK89] and incorporating a directionality factor from the Goldman model [Gol97]. Hair shadows are rendered using a different method than the one proposed by Bando et al. and will be discussed in the next chapter.

The head, for the purposes of rendering, is represented as a triangle mesh[2]. We store the head in two .obj[3] files – one for the head and one for the shoulders. This way, the head can be rotated while the shoulders remain

[2]http://www.turbosquid.com/3d-models/3d-woman-head/354953

[3]A simple geometry file format developed by Wavefront

still. The head is sent through the fragment shader too, in order to apply shadows to it (caused by the hair and the head itself).

We use the standard shadow mapping technique [SD02] for the head to cast shadows on itself as well as on the hair.

We also include several 'debug' renderers, which are not used in the production of the final image, but are very useful for debugging or for tweaking parameters of the simulation. For example, it is possible to display the air particles so that we can see how they interact with the hair particles.

2.7 Miscellaneous

The voxel grid is a simple structure. Its purpose is to make neighbor-searching efficient. Since this process is done in each simulation step, its speed has significant impact on the overall swiftness of the simulation.

It partitions space into a 3D grid of cube-shaped cells called voxels (Figure 2.6). Each voxel stores references to the particles that are contained in it. For efficiency, each particle also stores a reference to the voxel it is currently in.

In the SPH method, the particles that are more than a given distance away from each other do not affect one another. This value is called the smoothing length. Therefore, we only need to search for particle pairs that are this distant. A voxel grid is ideal for this purpose, since the voxel size is fixed to the smoothing length, and it is thus sufficient to search only the surrounding 27 voxels. This makes the asymptotic complexity of neighbor-search $O(n)$ (assuming even distribution of particles in space).

The main data stored in the voxel grid class is the 3-dimensional array of voxels. Each voxel contains a list of all particles it contains, as well as its logical coordinates (0-indexed) in the grid.

Ideally, we would partition the whole space in the grid, but this is, of course, not possible. We therefore need to have some boundaries for the grid. These are represented by a position of the grid's corner and its size.

Still, some particles might venture outside of the voxel grid's partitioned space. The solution to this problem is to store an 'extra' `Voxel` object in the `VoxelGrid` class to keep track of these outside particles. This object is not a genuine voxel, since it has no size or position, but in effect, it represents the complementary space to what the grid partitions. Thus, when a neighbor search is performed for a particle on the edge of the grid, we also need to look inside this 'outside' voxel, for it might contain its neighbors.

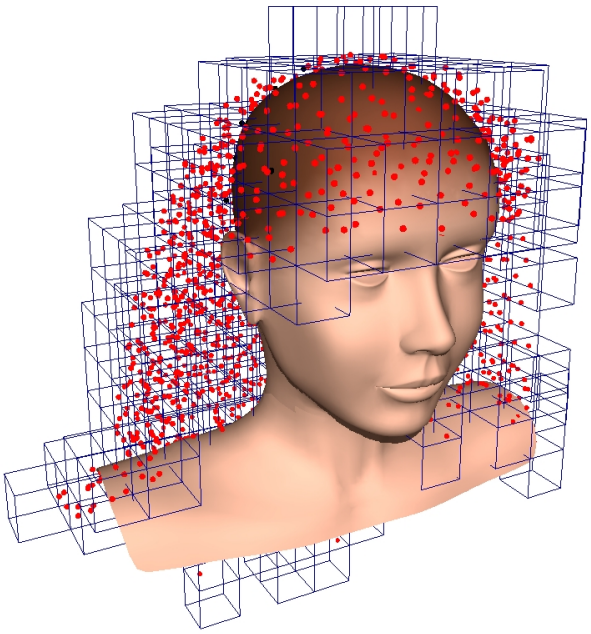


Figure 2.6: A voxel grid. Showing only voxels occupied by at least one hair particle.

In the simulation, after the particle motions are integrated, we update the voxel grid. This means that for each particle, we calculate the logical coordinates of the voxel it belongs to (based on the particle's position and voxel grid boundaries). Then we unassign the particle from the voxel it was in during the previous step and assign it to the new voxel (if required).

The `ParticlePair` is a very useful class, because the connections that are formed in the beginning for lengthwise coherence of hair persist throughout the simulation. But we need to be able to iterate through all of the pairs and also access any pair by the particles' IDs. Therefore, we keep two structures.

The first structure is a simple 1-dimensional array of all the pairs established in the beginning. Its purpose is fast iteration over the particle pairs. For every pair (i, j), there is one corresponding element in this array. This is because iteration over this array is used only when applying the spring forces which are symmetrical.

The other structure is a 2-dimensional static array of size $N \times N$, where N is the total number of hair particles. This allows for fast random-access to any pair. For one particle pair (i, j), there is one element stored at the

index $[i, j]$ and one at the index $[j, i]$ of this 2D array. It is because this array is used for calculating new directions, which, unlike calculating spring forces, are not symmetrical. This array might seem an unnecessary waste of memory, since it is always very sparse. But even with this structure, and with $N = 2000$ particles, the whole application consumes about 140MB of RAM, which is acceptable by today's standards.

Particle pairs are also indirectly stored in every `HairParticle` object through collections `NeighborsRoot` and `NeighborsTip`. This is because in certain situations, we need to iterate through a given particle's lengthwise neighbors.

2.8 User Interface

For designing a user interface, we used the Windows Forms API (WinForms). The interface is very simple – most of the application window is occupied by a control provided by OpenTK, which serves as the OpenGL viewport. This way, we can utilize common WinForms controls (such as buttons or track bars) for user interaction. We avoid the unnecessary work of making a custom HUD with controls inside the OpenGL viewport, while still allowing for an easy-to-use interface.

The way we connect the UI with the simulation parameters is through data binding. All WinForms controls have a way of binding themselves to data. We designed a very handy custom control for variable manipulation called `VisualTrackBar`. It is a composite control made of combining a `TrackBar` and a `TextBox` along with a `Label` (see Figure 2.7).

Figure 2.7: A group-box containing two visual track-bars.

To use a visual track-bar, we need to call either the `BindFloatData` method or `BindIntData` method, depending on what type of variable we wish to bind it to. This needs to be done once, in the UI initialization. For example, the call

`visualTrackBar.BindFloatData(Const.Instance, "Gravity", -2, 2);` binds the track-bar with a variable of the `Const.Instance` object named `Gravity`. Note that WinForms data-binding requires this variable to be a property. It also sets -2 as the minimum value, and 2 as the maximum value the track-bar goes to. However, for convenience, the user is allowed to type in a custom value into the text-box, even if it is outside the defined range. Additionally, the text-box is foolproof.

The way we utilize the visual track-bars in our program is through singleton classes. The variables often represent data which need to be shared across the application classes, thus resembling global variables. This was initially done through static classes with static data. However, data-binding in WinForms is not available for static variables, so the singleton pattern was introduced. There are two singleton classes used for this purpose, namely `Const` and `RenderingOptions`. The former contains simulation parameters such as gravity or hair elasticity while the latter stores information about current rendering preferences, such as whether to display hair in debug mode or not.

The user can also save and load variable configurations to and from a text file. This is useful if one wishes to compare different sets of simulation variable values or if the user wants to save an interesting configuration for later.

Chapter 3

Improvements

In this chapter, we will discuss the updates that were made to the LCP method.

3.1 Parallelization

Initially, the application was designed and developed without much consideration for parallelism. Over time, it grew bigger and more complex. The design was, however, deficient in the parallel aspect. It became clear that in order for a thorough parallelization, a proper design taking into account thread management and distribution would have been required. Therefore, the application would require a major overhaul.

Before completely redesigning the simulation, however, we decided to try and implement parallel calculations in another manner. It was done in a way which was much less invasive and required little design change. If this approach should not provide satisfactory performance improvements, only then would we continue in the radical redesign. For this purpose, we decided to utilize the managed thread pool in .NET Framework. This way we also conformed to one of our aims – using modern technologies and evaluating their suitability for this kind of application.

The approach was to only parallelize for-loops. There were several reasons for this decision. Firstly, according to a performance report provided by the VS2008 profiler, about 85% of the simulation work was spent in only several for-loops. Secondly, all of the calculations done in these loops were independent and distributing them among separate threads would not give rise to any race conditions. Additionally, using a managed thread pool could

provide a transparent way of multi-threading. Hence, the code impact would be minimal.

This idea was primarily inspired by the new Task Parallel Library included in .NET Framework 4.0. It was introduced by Microsoft with the intention of making parallelism easier to implement for developers. It exposes, among others, a parallel for-loop construct. However, migrating to a higher version of the .NET Framework (we are using .NET 3.5) was not required, since it is possible to implement such construct on our own.

On his blog, Martin Koníček contemplates how Microsoft might have implemented their parallel for-loop and provides a custom implementation [Kon08]. In this solution, the number of threads used is equal to the number of CPU's. These threads process the for-loop by chunks, and use a synchronization mechanism when 'grabbing' another chunk of tasks. In this context, a task denotes an iteration of the loop. As noted by Martin Koníček, "bigger chunk size should reduce lock waiting time and thus increase parallelism". If the chunks are too large, however, efficiency starts to drop. The reason for this drop is that by processing a big chunk, a thread can leave another thread without anything to do. If the chunk was smaller, each of the threads would get a portion of it to process.

We can see that this implementation requires locking. One could therefore argue against its efficiency. As a counter-approach, we propose a similar method, but without the need for thread synchronization. The difference lies in the distribution of tasks. While the former approach distributes work dynamically (which gives rise to the need of synchronization), the work can also be distributed statically. By static, we mean that all work is distributed in the beginning of the parallel-for routine. However, this approach requires using a larger number of threads, since the actual work load might be unevenly divided.

To sum up, the dynamic approach uses a constant number of threads (dependent on the CPU used) and we specify the chunk size used for dynamic distribution of work. For the static approach, on the other hand, we specify the number of threads to use, and the chunk size is directly computed from this number.

These simplified approaches to parallelization of the computations provided satisfactory efficiency, so the redesign was not needed. How well these two approaches performed will be discussed in the next chapter.

3.2 Deep Opacity Maps

Hair shadows are not mere eye candy. Without shadowing, hair is visually very unconvincing, since it appears without depth and volume. We have decided to implement a novel approach of rendering hair self-shadows proposed by [YK08]. The Deep Opacity Maps (DOM) are an alternative to the classic Opacity Shadow Maps [KN01]. Yuksel claims his method is faster, produces few or no artifacts and requires fewer rendering passes and memory.

We encountered several obstacles when applying DOM to the LCP method. The first problem stems from rendering the hair with a splatting method. This does not suit the depth pass of DOM very well. Z-buffer decisions are true/false, but the splat texture is semi-transparent and represents depth. Without any changes to the algorithm, the 'shape' of the hair that is rendered to the depth buffer ends up very blocky and is not a good representation of shape at all (Figure 3.1a).

(a) (b)

Figure 3.1: Depth maps (a) without alpha thresholding; (b) with alpha thresholding 0.08

To ameliorate this effect of splatting, we introduce alpha thresholding

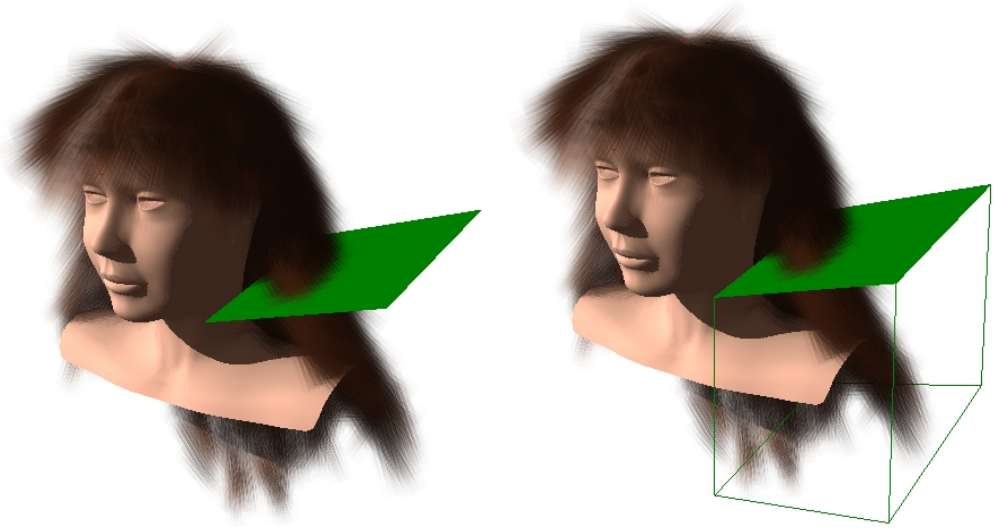
(Figure 3.1b). If a fragment has an alpha value lower than the threshold, it is discarded altogether in the DOM rendering passes. The alpha value that is compared with the threshold includes the opacity scale from the LCP method. If we then need to read the depth and opacity values for a fragment which is to be rendered in the final pass but the corresponding values have been discarded, we treat the fragment as unshadowed. Note that it is important to set texture filtering to NEAREST for the depth map.

As recommended by Yuksel, we used $d, 2d, 3d$ as the distance between the first, second and third pair of layers, respectively. The value of d can be specified through the UI.

See Figure 5.2 for a comparison of hair shadowed without and with alpha thresholding.

3.3 Cutting

In the LCP method, hair strands are not modeled explicitly. Instead, spring forces are used to maintain lengthwise coherence. To cut hair in this model would mean to remove these spring connections.



(a) (b)

Figure 3.2: Cutting hair (a) the user specified rectangle; (b) what is actually cut

We implemented cutting in a user-specified rectangle (Figure 3.2a). After the cutting mode has been switched on, a green rectangle appears and the user can now use the mouse to change the position, size, and orientation of the rectangle. After the cut has been executed, we need to check all particle pairs and see which ones are intersected by the rectangle.

This alone, however, does not produce the desired effect. The intuitive expectation is that after the hair has been cut by a horizontal rectangle, the hair under this rectangle would fall off, which is not the case. This is because the lengthwise connections are not linear, meaning that a hair particle can be connected to the root particles by many paths (Figure 3.3b). Because of this, even if the specified connections are cut, the hair underneath still holds onto the root particles indirectly, through springs connected in sequence up to the root particles.

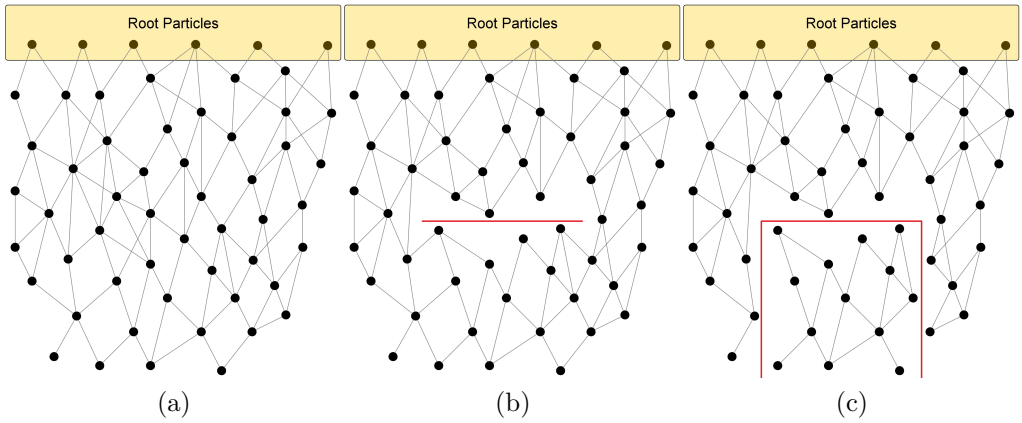


Figure 3.3: The problem with cutting lengthwise spring connections.

To make the hair under the rectangle fall off, we need to remove all the connections that lead outside from the volume we want to cut off (Figure 3.3c). The way we tackled this problem is that in addition to cutting the connections by the specified rectangle, we make a few more cuts. These additional cuts are specified by 4 parallelograms, which 'hang' from the edges of the user-specified rectangle (Figure 3.2b). The height h of the column formed by the rectangle and the parallelograms needs to be large enough, so as to cut all the required connections, while being small enough not to accidentally cut other connections. Alas, the fluid representation of hair has its limitations, making hair cutting a little like cutting water.

Chapter 4

Testing and results

The application was tested on an Intel Core 2 Duo 3.0GHz (two CPU cores) with ATI Radeon HD4870 GPU. Most of the time, we use 2000 hair particles so that we can compare our results with the original work of Bando et al.

4.1 Simulation Performance

All tests in this section were performed on 2000 hair particles and 0 air particles. Therefore, most of the usages of the parallel-for structuress get 2000 iterations to process. The only exception is iteration over the particle pairs.

For reference, we first tested the application with serial processing, using basic loops. This configuration ran at 5.9 FPS.

To be able to properly assess the effectiveness of our parallel implementation, we ran tests with disabled rendering for this section. Let us denote the first parallel-for approach 'dynamic parallel-for' and the second one 'static parallel-for'.

The dynamic parallel-for distributes work among threads by chunks of constant size. This approach was tested against various values of chunk size, see Figure 4.1.

As expected, we can see that if the chunk size values are too low, FPS is below its peak, because the overhead produced by thread synchronization. However, we did not expect the algorithm to maintain its peak efficiency all the way up to when chunk size is equal to half of the actual data size (chunk size = 1000 when there are 2000 actual particles (iterations) to process). This, in fact, means that both threads immediately grab one thousand iter-

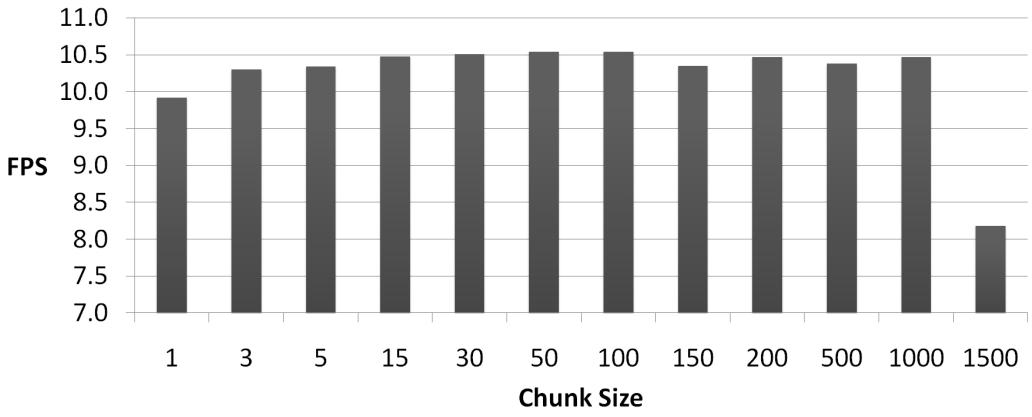


Figure 4.1: Average simulation FPS versus chunk size.

ations to process and no other distribution occurs. Why this did not slow down the algorithm is probably because the work is distributed quite evenly after all.

Obviously, the efficiency had to suffer a drop at chunk size equal to 1500, since one thread would get three times more iterations to process than the other. In other cases, the impact of the chunk size on FPS seems negligible.

The static parallel-for distributes iterations evenly among a constant number of threads. We therefore tested it against various numbers of threads used, see Figure 4.2.

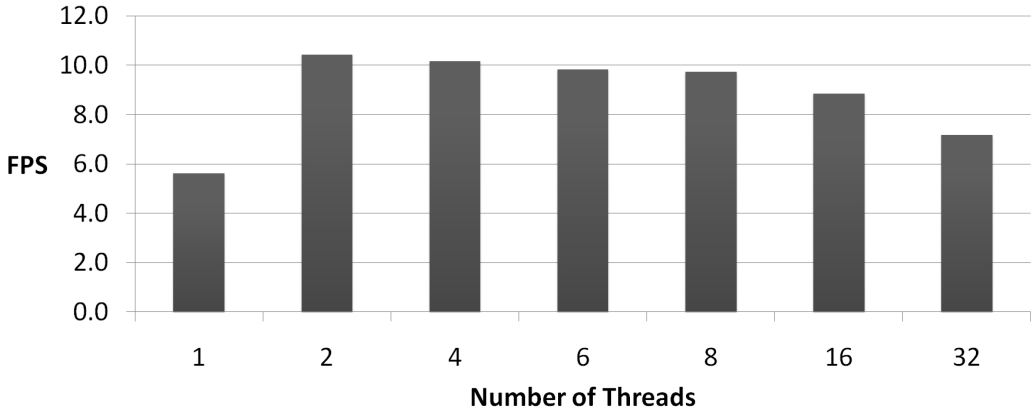


Figure 4.2: Average simulation FPS versus thread count.

Clearly, increasing the number of threads results in lower efficiency. This

is likely caused by the increased thread-context-switching overhead. The results from the previous paragraphs indicated that the work is distributed evenly among the iterations. We can therefore assume that increasing the number of threads has no positive impact on performance at all.

And finally, if we compare the actual numbers of the two approaches, their peak efficiency is almost the same. The former's max FPS is 10.5 while that of the latter's is 10.4. Note, however, that at chunk size equal to 1000 for the dynamic approach and thread count equal to 2 for the static, these algorithms are the same.

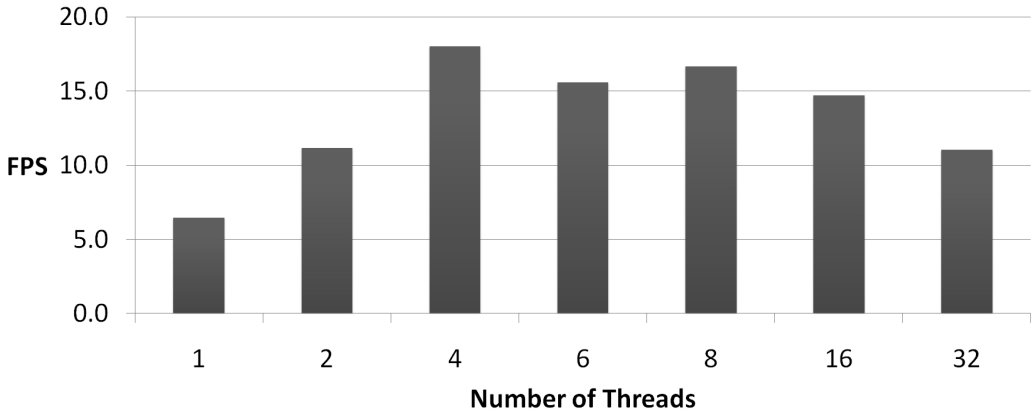


Figure 4.3: Average simulation FPS versus thread count (quad-core CPU).

So, with 5.9 FPS with serial processing, and 10.5 FPS with multi-threaded simulation, we achieved a parallelism factor of 1.78 for 2x the number of CPUs. We also performed a brief series of tests on a quad-core CPU (Intel Core i5-750 2.66Ghz with ATI Radeon 5750), and this resulted in a 2.77 higher efficiency for 4x the number of CPUs (see Figure 4.3 for the static-for performance).

4.2 Overall Performance and Appearance

When running with normal settings (rendering enabled, 2000 hair particles, 0 air particles), we achieve a frame rate of 9.3.

Figure 4.4 shows a screenshot of the scene viewport along with the interface and controls. We see several visual track-bars in use, all of which immediately affect the behavior or looks of the simulation.

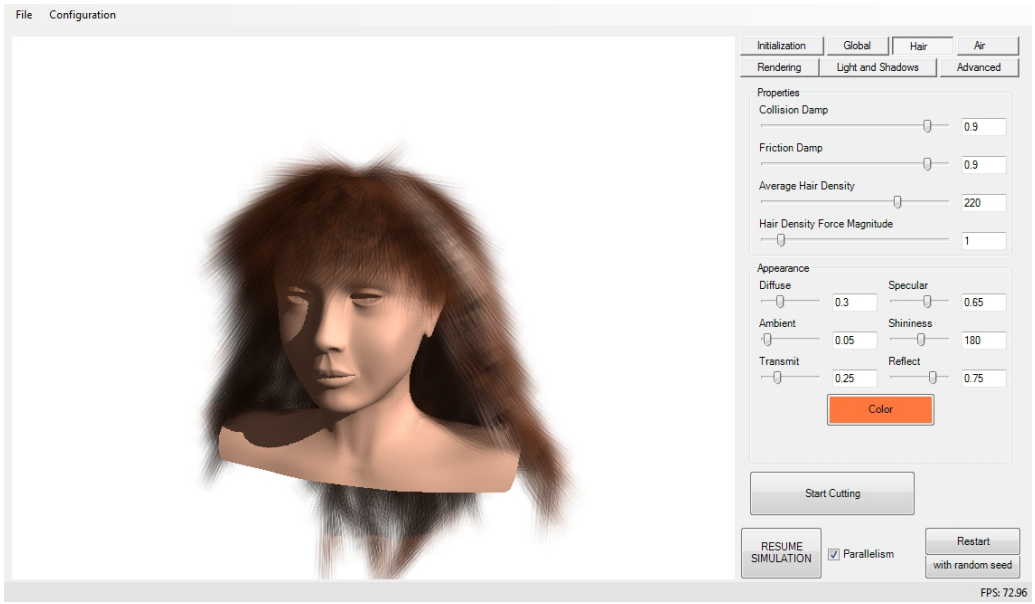


Figure 4.4: The actual application window.

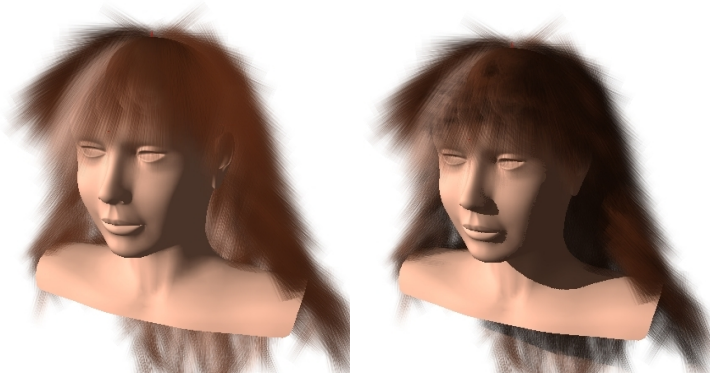
After tweaking the appearance of the hair in the GUI, we can achieve different types of hair. Figure 4.5 demonstrates different hair color with a wind effect. Figure 5.2 shows blonde hair.

The implementation of Deep Opacity Maps with added alpha thresholding works well, and adds a significant amount of depth to the hair volume (Figure 4.6).

However, we encountered several kinds of visual deficiencies, which we did not manage to eliminate. Most of them are only clearly visible in a video or in the application itself. First such flaw is caused by billboarding – that the rendered hair is not independent of camera position. When the camera moves, the billboards rotate and thus the shadows cast upon them may change their shapes or orientation, thus losing consistency. The same goes for an analogous phenomenon. When the camera is still, and the light source moves, billboards in the DOM rendering pass orient themselves toward the light. Again, shadows change their shape, in a way that is not realistic for a scene with a moving light. However, we believe these issues emerge naturally from merging billboarding with DOM and are a principal part of this fusion.

We tried several workarounds for this, but with little or no success. For instance, we tried keeping the billboards oriented toward the camera (instead

Figure 4.5: A redhead type of hair with wind blowing from the front.

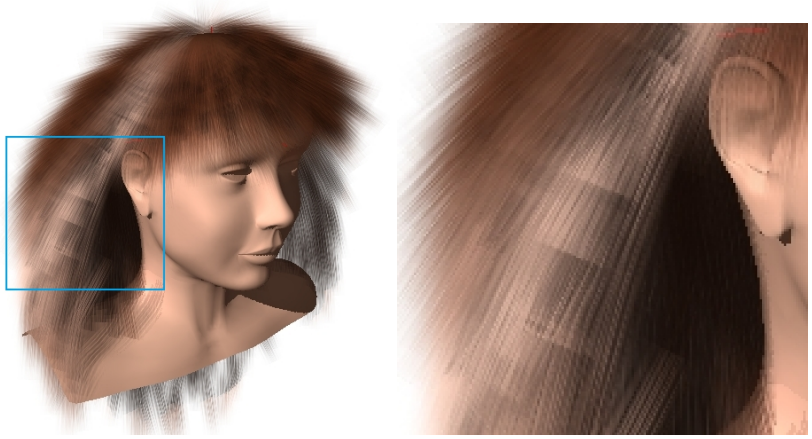


(a) (b)

Figure 4.6: Comparison of hair (a) without shadows; (b) with shadows

of the light) in the shadow passes. However, it is evident that when the particle-camera and particle-light vectors are close to perpendicular, the 2D billboard disappears completely in the shadow pass.

The second type of visual artifact appears when the number of hair particles is high enough (at least 500 particles), and it appears in places where the billboards overlap, or are close to each other (Figure 4.7). It is best visible against specular highlight. We were unfortunately unable to trace the cause of this effect and thus did not succeed at its removal.



(a) (b)

Figure 4.7: A layering artifact of Deep Opacity Maps.

4.3 Using C#

Exploring the suitability of C# (as opposed to C++) for performance-heavy applications such as ours was one of the aims of this thesis. In course of the development, we stumbled upon several noteworthy matters.

The .NET Framework is closed-source. Many of our optimizations had to be reverted because the performance dropped, while we had no way of knowing it would happen, because we are using .NET classes. For instance, a linked list should, in theory, perform better than a dynamic array when it is only used for adding and enumeration. However, that was not the case. That is not to say that the linked list is inefficient, we just don't really know what the program does at all times.

Another example is the speed of `for` versus `foreach` loops. For iteration over a `List`, these two delivered noticeably different performance, the basic for loop being faster.

The last considerable disadvantage of using C# was our inability to use a program for debugging OpenGL shaders, glslDevil – or at least we did not manage to get it working.

On the other hand, the environment provided a great deal of convenience. As we saw in Section 3.1, using the managed thread pool was both simple and efficient. In languages such as C++, the parallelization process would have been far more complicated. Also, designing the GUI and binding it to our data was very easy, thanks to a possibility to design a custom component.

Chapter 5

Conclusion

5.1 Summary and Accomplishments

We have successfully created an application for simulation and rendering of human hair. We used the Loosely-Connected Particles method and updated it with modern technologies. These include parallel computations, Deep Opacity Mapping and hair cutting.

For an implementation of parallelism, we used simple parallel-for constructs, which turned out to be suitable for this kind of particle system simulation. These parallel loops utilize the managed .NET 3.5 thread pool. In .NET 4.0, Microsoft provides an explicit `Parallel.For` implementation, see Figure 5.1 for a comparison of performance.

We have also tested the performance of the static and dynamic approaches to handling the issue of work distribution in parallel processing. The results of these tests show thread synchronization to outperform thread context switching.

Overall, we achieved a 178% simulation speed increase for a dual-core CPU and a 271% increase for a quad-core CPU.

The Deep Opacity Mapping algorithm required some adjustments to fit the billboarding in LCP. The alpha thresholding method we proposed makes the DOM suitable for a hair rendering method such as billboarding with transparency. See Figure 5.2 for comparison of shadows with and without alpha thresholding.

Even though the implementation of hair cutting is heuristic and a not very robust (using the additional cutting parallelograms), it produces nice and believable results (Figure 5.3).

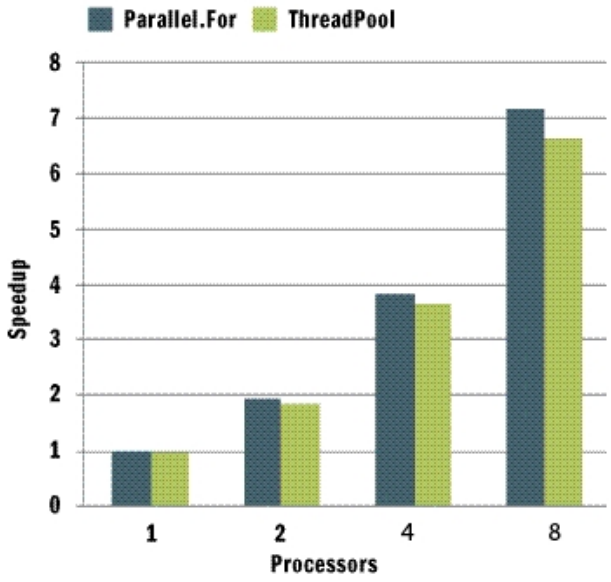


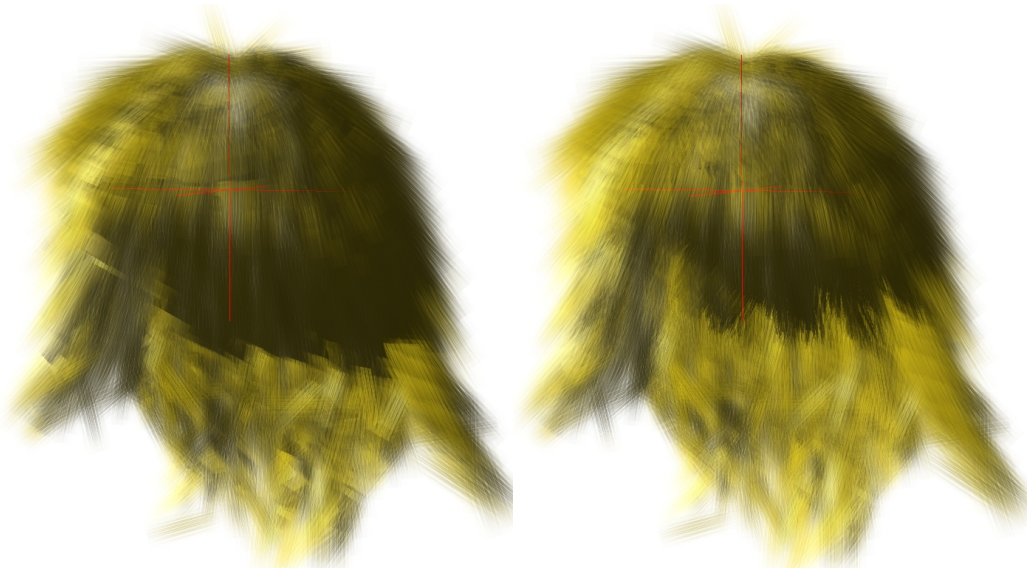
Figure 5.1: Comparison of speedups for different parallel-for realizations. Image courtesy of [LH07].

The author of this text himself gained a lot of knowledge and skills in the area of real-time computer graphics through the course of this work. Knowing nothing about 3D rendering, having little practice with parallel algorithms and no knowledge at all about hair animation in the beginning, there was a lot learned.

5.2 Discussion and Future Work

To make an honest comparison of our 2010 implementation with Bando's 2003 implementation, we need to look at the frame rates. When simulating 2000 particles, Bando reported an FPS of 6.7 while we achieved 9.3 on a dual-core CPU and 14.4 on a quad-core CPU. However, we used mult–threading to do the simulation. If we were to only use one CPU core (5.9 FPS), Bando's implementation running on 2003 hardware would outperform ours. This is presumably due to little low-level optimization and no usage of third-party numeric libraries.

Also, when investigating the performance issues, we can only trace them so far, since .NET is not open-source. The main issue with relying solely on



(a) (b)

Figure 5.2: Shadowed hair with alpha thresholding (a) disabled; (b) enabled.

.NET classes is one never knows how they are implemented. Knowing the asymptotic complexity of a method is far from enough when every frame per second gained is crucial. We do not, however, draw the conclusion that C#, or similar languages, should be disregarded for development of graphic applications. Strong typing, easy debugging, performance profiling – these are all great advantages of this environment that make the life of a programmer much more pleasant. In the end, the choice of language is about compromise between application efficiency and development efficiency.

It is up to the reader to judge whether the images we have produced are convincing and believable. But it is evident that the splatting method produces certain 'fuzziness' in the appearance of the hair, which is undesirable, but hard to eliminate. This may be a proposition for future research – to introduce a completely different rendering method for a particle system representing hair, perhaps one that is not based on billboarding.

Future investigation might also include using the actual .NET 4.0 parallel for construct, as provided by Microsoft. Or the parallelism could be taken to the next level by utilizing the GPU for simulation, not only for rendering, by means of GPGPU.

Figure 5.3: The hair, only moments after it has been cut by the green rectangle.

Bibliography

[AUK92] K. Anjyo, Y. Usami, and T. Kurihara. A simple method for extracting the natural beauty of hair. In *SIGGRAPH '92: Proceedings of the 19th annual conference on Computer graphics and interactive techniques*, pages 111–120, New York, NY, USA, 1992. ACM.

[BAC+06] Florence Bertails, Basile Audoly, Marie-Paule Cani, Bernard Querleux, Frédéric Leroy, and Jean-Luc Lévêque. Super-helices for predicting the dynamics of natural hair. In *SIGGRAPH '06: ACM SIGGRAPH 2006 Papers*, pages 1180–1187, New York, NY, USA, 2006. ACM.

[BCN03] Y. Bando, B. Chen, and T. Nishita. Animating hair with loosely connected particles. *EUROGRAPHICS*, 22(3), 2003.

[Gol97] Dan B. Goldman. Fake fur rendering. In *SIGGRAPH '97: Proceedings of the 24th annual conference on Computer graphics and interactive techniques*, pages 127–134, New York, NY, USA, 1997. ACM Press/Addison-Wesley Publishing Co.

[HMT01] S. Hadap and N. Magnetat-Thalmann. Modeling dynamic hair as a continuum. *EUROGRAPHICS*, 20(3), 2001.

[KK89] J. T. Kajiya and T. L. Kay. Rendering fur with three dimensional textures. *SIGGRAPH Comput. Graph.*, 23(3):271–280, 1989.

[KN01] Tae-Yong Kim and Ulrich Neumann. Opacity shadow maps. In *In Proceedings of the 12th Eurographics Workshop on Rendering Techniques*, pages 177–182. Springer-Verlag, 2001.

[Kon08] Martin Koníček. Coding time - Martin Konicek: Implement your own Parallel.For in C#. `http://coding-time.blogspot.com/`

2008/03/implement-your-own-parallelfor-in-c.html/, 2008. [Online; accessed 19-July-2010].

[LH07] Daan Leijen and Judd Hall. Parallel Performance: Optimize Managed Code For Multi-Core Machines. http://msdn.microsoft.com/en-us/magazine/cc163340.aspx, 2007. [Online; accessed 29-July-2010].

[SD02] Marc Stamminger and George Drettakis. Perspective shadow maps. In *SIGGRAPH '02: Proceedings of the 29th annual conference on Computer graphics and interactive techniques*, pages 557–562, New York, NY, USA, 2002. ACM.

[SLF08] Andrew Selle, Michael Lentine, and Ronald Fedkiw. A mass spring model for hair simulation. *ACM Trans. Graph.*, 27(3):1–11, 2008.

[TB08] Sarah Tariq and Louis Bavoil. Real time hair simulation and rendering on the gpu. In *SIGGRAPH '08: ACM SIGGRAPH 2008 talks*, pages 1–1, New York, NY, USA, 2008. ACM.

[YK08] Cem Yuksel and John Keyser. Deep opacity maps. *Computer Graphics Forum (Proceedings of EUROGRAPHICS 2008)*, 27(2), 2008.

Appendix A

DVD Contents

The attached DVD contains the following folders:

- Documentation – The Doxygen html documentation of the implementation.
- Install – Contains setup.exe, the installation file.
- Media – Includes screenshots and videos from the application.
- Source – The application source code.
- Thesis – Contains the PDF version of this thesis, along with originals of the figures (pictures) used in it.

Appendix B

AnimatingHair User Reference

B.1 Introduction

Animating Hair is a software project concerned with implementation and showcasing a method of *hair physics simulation and rendering* proposed by Bando et al. in 2003.

B.2 Requirements

The following requirements need to be met in order to run the program:

- Microsoft Windows operating system
- .NET Framework 3.5
- OpenGL 2.0 compatible graphics card
- 256MB memory

B.3 Installation

To install the program, run setup.exe from the Install directory on the DVD. The installation wizard will guide you through the installation process.

B.4 Usage

B.4.1 Overview

The application uses a typical Windows interface. The major part of the window is occupied by the rendering frame, in which the simulated world can be observed. On the right-hand side of the interface, there is a tabbed group of controls for modifying the simulation and rendering variables.

B.4.2 Rendering frame

In the rendering frame, the user can see the simulation entities and their interactions. To rotate the view, left-click on the frame and drag the mouse. To zoom in and out, either right-click on the frame and drag the mouse up and down, or use the mouse wheel.

Moving and rotating the head to see the hair reaction is possible. Use the numpad 8, 5, 4, 6 keys to move the head and the 7 and 9 keys to rotate it. (*Note: the air particles' positions stay the same relative to the head.*)

B.4.3 Variables panel

The variables panel is divided into tabs, separating simulation variables into groups. The programs responds to variable changes immediately, wherever possible (meaning that shifting the gravity track bar, for instance, has an immediate effect on the simulation, without any need for confirmation).

Appendix C

Performance Measurements

In this appendix, we present the tables of measurements made regarding the multi-threaded simulation.

Chunk Size	FPS
1	9.920
3	10.302
5	10.344
15	10.478
30	10.511
50	10.539
100	10.541
150	10.346
200	10.467
500	10.384
1000	10.465
1500	8.178
3000	6.087

Table C.1: Dynamic-for implementation performance on the dual-core configuration.

Without any parallelism, using normal for-loops, we measured 5.923 FPS on the dual-core configuration and 6.498 FPS on the quad-core configuration.

No. of Threads	FPS
1	5.629
2	10.440
4	10.160
6	9.825
8	9.730
16	8.850
32	7.183

No. of Threads	FPS
1	6.437
2	11.175
4	18.026
6	15.577
8	16.649
16	14.723
32	11.030

Table C.2: Static-for implementation performance on the dual-core configuration (left) and on the quad-core configuration (right).