Simulation of hair and fur

vorgelegt von

Andreas Henne

Ausarbeitung zum Masterprojekt
Albert-Ludwigs-Universität Freiburg im Breisgau

August 2013

Contents

1 Introduction

When creating realistic virtual humans or animals, one of the biggest challanges is often the simulation and rendering of the hair or fur. In the movie industry, sophisticated methods that simulate individual strands of hair have been used for some time already. Popular recent examples include Merida's long curly hair in Pixar's *Brave* (2012) and the fur of the tiger in *Life of Pi* (2012). It is also becoming more important in computer games. The PC Version of *Tomb Raider* (2013) includes an optional mode in which Lara Croft's hair is simulated using thousands of strands of hairs. With faster video game consoles available in late 2013, it is not unlikely that realistic hair simulation and rendering will be more common in the video games industry in the future.

Realistic depiction of hair involves several challenges. Firstly, the strands of hair must be placed on the human head or the skin of the animal. Secondly, the strands of hair must be simulated. The hair should react to external forces such as wind or gravity and it should not pass through rigid objects (for example the shoulder of a human). The simulation should ensure that the lengths of the strands of hair remain constant, because hair is perceived as inextensible. Also, the interactions between individual strands of hair must be considered to achieve realistic results. Thirdly, the simulated strands of hair must be rendered which requires an illumination model for hair. The survey by [WBK+07] provides an extensive overview over these problems and possible approaches.

The goal of this project is to be able to place strands of hair on arbitrary meshes and to simulate and render them. Thereby, the focus lies on simulation. The main source for this project is the paper by [MKC12], which is about hair simulation at interactive framerates. The implementation developed as part of this project is used to render a furry bunny and long hair.

2 Mesh Sampling

Placing the strands of hair on the skin is not a trivial task. The sampling should look natural and a bit chaotic, but the strands of hair should also be distributed evenly over the surface. While it may be possible to implement a specific solution for human heads, fur may be placed on creatures with very different shapes. The sampling algorithm should be capable of sampling arbitrary triangle meshes.

2.1 Poisson sampling

A desirable sampling is a maximal Poisson disk point set. Given a minimum distance d, two points in a Poisson disk set have a distance of at least d. A Poisson disk set is maximal if no point can be added to the set without being too close to one of the existing points in the set. Figure 2.1 shows the Armadillo mesh from the Stanford 3D scanning repository sampled with 300K particles using random sampling and Poisson disk set sampling. The Poisson disk set covers the surface better with the same number of samples.

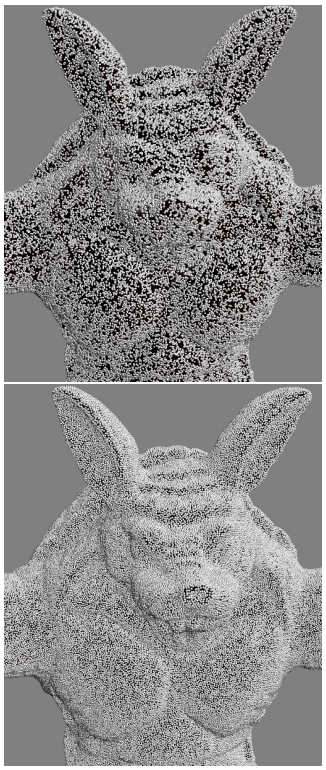


Figure 2.1: Random (top) vs. Poisson sampling (bottom) with 300K samples

For this project I implemented an algorithm proposed by [CJW+09] that allows to generate maximal Poisson disk point sets with a minimum distance d on 3D surfaces. Their technique works for triangle meshes, but also for implicit and parametric surfaces. They also propose an approach to approximate the geodesic distance metric, where the distance between two points is not the Euclidean distance but the length of the shortest path on the surface. For hair and fur that is sampled very densely the Euclidean distance is usually sufficient. Also, I only consider sampling of triangle meshes since it is the most common representation of surfaces for natural objects like animals and human heads.

The approach by [CJW+09] is based on the idea of dart throwing. A random dart is thrown on the surface to generate a sample on the surface. The sample is rejected if it is too close to one of the existing samples, otherwise

it is accepted. This is repeated until the Poisson set is maximal. This naive approach is obviously not very efficient, since most darts will be rejected. Besides, it is not clear how do determine that the Poisson disk set is maximal. Optimized dart throwing addresses these problems by excluding surface fragments that are already completely covered by samples. The algorithm terminates when the complete surface is covered by samples.

2.1.1 Throwing a dart on a triangle mesh

Throwing a dart on a triangle mesh is implemented in two steps. In the first step a triangle is chosen, in the second step a dart is thrown on this triangle. All points on the surface should be hit by the dart with the same probability, so the probability that a specific triangle is chosen should be proportional to its area. The area a of a triangle with the vertices $\mathbf{p_1}$, $\mathbf{p_2}$ and $\mathbf{p_3}$ can be computed with the following formula:

$$a = \frac{1}{2} \cdot ||(\mathbf{p_2} - \mathbf{p_1}) \times (\mathbf{p_3} - \mathbf{p_1})||$$

Figure 2.2: Armadillo sampled with 25K points

[CJW+09] propose logarithmic binning to quickly choose a triangle with a probability proportional to its area. Each triangle is inserted into one of 64 bins according to its area, so that each bin contains triangles from a minimum area B_{min} to a maximum area $B_{max} = 2B_{min}$. Each triangle is placed into the bin with index $\lfloor \log_2(A_{max}/A_t) \rfloor$, where A_t is the area of the triangle and A_{max} the maximum area of all triangles. This only works under the assumption that the ratio of the largest triangle to the smallest triangle is smaller than 2^{64} ([CJW+09]). In practice, all meshes should fulfill this requirement.
Instead of directly choosing a triangle, a bin is chosen first. The probability to choose a bin must be proportional to the average area of the triangles inside it. A bin is chosen by generating a random number between 0 and 1 and performing a linear search. Since the number of bins is constant, this operation has a constant time complexity. After a bin has been chosen, rejection sampling is used to chose a triangle inside the bin. A random triangle with area A_t inside the bin is selected and accepted with probability $A_t/(2B_{min})$. This is repeated until a triangle is accepted. The probability that a triangle is accepted is at least 50 percent, because $A_t \geq B_{min}$. This means that the rejection sampling runs in constant time on average. Since choosing a bin runs in constant time, this means that a triangle is chosen in constant time on average.

After choosing a triangle, a sample point candidate on the triangle is generated. For this purpose barycentric coordinates are used. A barycentric coordinate (u, v) for a triangle given by three points p_1, p_2, p_3 describes the following global position:

$$\mathbf{x} = \mathbf{p_1} + u(\mathbf{p_2} - \mathbf{p_1}) + v(\mathbf{p_3} - \mathbf{p_1}) = \mathbf{p_1}(1 - u - v) + u \cdot \mathbf{p_2} + v \cdot \mathbf{p_3}$$

$\mathbf{x}$ is inside the triangle if and only if $0 \leq u \leq 1$ and $0 \leq v \leq 1$ and $u + v \leq 1$. Otherwise it is on the triangle plane, but outside the triangle.
A point on the triangle can be generated by generating a barycentric coordinate (u, v) that

fulfills this condition. Generating the random barycentric coordinate is not as easy as it seems. Any point on the triangle should be chosen with the same probability. It is not enough to just generate two numbers between 0 and 1, because the resulting point could lie outside the triangle. In my implementation, I use the following equations to generate random barycentric coordinates on the triangle, given two random numbers a and b between 0 and 1 that are uniformly distributed:

$$u = 1 - \sqrt{a}$$

$$v = a\sqrt{b}$$

The global candidate position is computed from the generated barycentric coordinate. This candidate is accepted and added to the poisson disk set if no existing sample is too close to it. To accelerate this query, a spatial data structure is required that stores the generated samples. In my implementation I use (just like [CJW+09]) a grid based on spatial hashing to search neighbor samples of sample candidates. Spatial hashing is explained in more detail in section 4.

2.1.2 Optimized dart throwing on triangle meshes

The idea behind optimized dart throwing is to exclude surface fragments that are completely covered by already generated samples. When dealing with triangle meshes, this can be achieved by excluding triangles.
Checking if a set of samples exists that covers a triangle would be computationally very expensive. [CJW+09] only exclude triangles that are completely covered by one sample. A triangle is covered by a sample if all its vertices are closer to the sample than the minimal distance. The spatial hash grid is used to find out near samples that cover one of the triangle's vertices. If one of the samples also covers the other two triangle vertices, it covers the triangle completely. This test is performed after a dart was thrown on a triangle, no matter whether the dart was accepted or not. If the triangle is covered completely by one sample, it is excluded. Otherwise, it is subdivided into 4 subtriangles and the subtriangles are added to the appropriate bins. Without the subdivision, large triangles would never be covered by one sample. A triangle is split into 4 sub triangles by adding vertices at the edge midpoints as shown in figure 2.3. The algorithm terminates when all triangles are excluded.

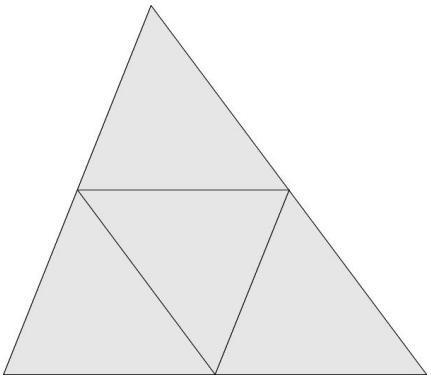


Figure 2.3: Triangle subdivision

2.1.3 Floating point issues

When adding a triangle to a bin or removing one from it, the sum of triangle areas in the bin must be updated. An intuitive and efficient solution is to subtract the triangle area when removing a triangle and adding the area when a triangle is added to the bin. However this may lead to problems due to floating point precision. When hundreds of thousands of triangles are added and removed from bins due to to triangle subdivision, floating point errors can sum up and become a serious problem. I often experienced bins with a negative sum

of areas that were obviously never chosen because their probability was technically below 0. Vice versa empty bins often had a sum of areas above 0. To ensure termination I added two checks. Whenever a triangle is added to the bin and the new sum of areas is below the triangle area, the sum of areas of the bin is set to the triangle area. Whenever a triangle is removed from a bin and the bin is empty, the sum of areas in the bin is set to 0.
Another floating point related issue is that too small triangle triangles can lead to termination problems. [CJW+09] suggest to just discard triangles with an area below a very small threshold.

2.1.4 Generating barycentric coordinates instead of positions

The algorithm by [CJW+09] generates a set of sample positions. However for hair and fur that is attached to animated objects it is more convenient to have for each sample a triangle index and the barycentric coordinates on it. The algorithm already generates barycentric coordinates to sample triangles, but these coordinates are mostly generated on sub triangles created by the algorithm that do not exist in the original triangle mesh. The first step to solve this problem is to store for each sub triangle the index of the original triangle it is part of. Now it is possible to output the index of the corresponding triangle for each sample. One possibility to find out the barycentric coordinate on the original triangle is to compute the global sample position and solve a system of linear equations. A more efficient solution is to directly convert barycentric coordinates on a sub triangle to barycentric coordinates on the original triangle. This is possible because the edges of all subtriangles are parallel to the edges of the original triangle. In my implementation I use an offset $\mathbf{o}$ (which is a two dimensional vector) that stores where the subtriangle starts and a scale factor s that encodes the size of the subtriangle. s can be positive or negative, which is used to invert the coordinates for the middle subtriangle. The following C++/pseudo code shows how a triangle is split into 4 sub triangles and how barycentric coordinates on the sub triangle are transformed to coordinates on the original triangle:

```
1  struct SubTriangle
2  {
3      int origTri;
4      float s;
5      vec2 o;
6      SubTriangle(int origTri, float s, vec2 o)
7          : origTri(origTri), s(s), o(o) {}
8
9      vec2 mapToOriginalCoords(const vec2& coords)
10     {
11         return s * coords + o;
12     }
13
14     vector<SubTriangle> splitIntoSubTriangles()
15     {
16         vector<SubTriangle> out;
17         float subS = s * 0.5f;
18         out.push_back(SubTriangle(origTri, subS, o));
19         out.push_back(SubTriangle(origTri, subS,
20                 vec2f(o.x + s * 0.5f, o.y)));
21         out.push_back(SubTriangle(origTri, subS,
22                 vec2f(o.x, o.y + s * 0.5f)));
```

```
23            out.push_back(SubTriangle(origTri, -subS,
24                    vec2f(o.x + s * 0.5f, o.y + s * 0.5f)));
25            return out;
26        }
27    };
```

Instead of representing sub triangles with 3 global positions, a sub triangle only stores the id of the original triangle (or a pointer to it) and information how to map barycentric coordinates on the sub triangle to barycentric coordinates on the original triangle. This also saves some memory which is convenient when many sub triangles are generated. In a pre-processing step, a SubTriangle object is created for each triangle of the mesh with $s = 1$ and $o = (0, 0)$ so that the algorithm does not need to differentiate between sub triangles and triangles.

2.1.5 Performance evaluation

I tested my implementation with the Happy Buddha and the Armadillo mesh from the Stanford 3D scanning repository. The Buddha has slightly more than 1M triangles, the Armadillo has roughly 350K. The test was performed on a system with an Intel i5 3450 @ 3.1 Ghz CPU and 8 GB RAM.

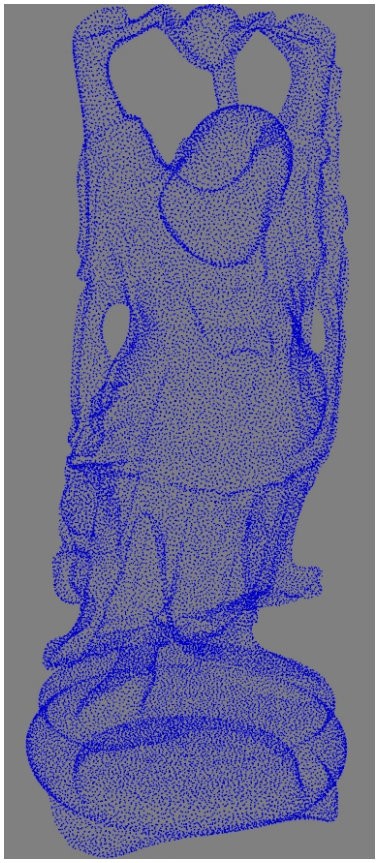


Figure 2.4: Buddha sampled with 40K points

Buddha:

Minimum distance	Samples	Time
0.01	9564	2.1s
0.005	39440	2.9s
0.003	109870	5.2s
0.001	992627	49.4s

Armadillo (which is scaled differently than the Buddha, thus the different minimum distances):

Minimum distance	Samples	Time
0.05	26286	1.1s
0.04	41174	1.3s
0.03	73298	2.2s
0.02	165335	5.0s
0.015	294017	9.4s
0.01	662063	25.2s

According to [CJW+09] the time complexity of the algorithm is $O(N + M)$, where N is the number of triangles and M the number of generated samples. In case of the Buddha the generation of samples outweighs the processing of triangles when more than a few ten thousand samples are generated. After that the algorithm scales linearly with the number of generated samples. The timings are sufficient for the purpose of this project, since hair and fur is only placed once on meshes.

2.1.6 Limitations

Hair or fur usually does not cover the mesh completely. In my implementation I added the possibility to only sample a subarea of the mesh by passing a list of triangle indices that shall be sampled to the algorithm. However for different hair styles it is often required to generate more samples in certain areas and less in others. A maximal poisson disk set of points with one minimum distance is obviously not the right choice for this kind of sampling. [CJW+09] propose an extension of their algorithm where each dart can have an individual radius. A dart is rejected if the sphere with its position and radius overlaps with one the other existing dart spheres. This could be used to sample the mesh with varying sample densities. The radii could be stored per vertex or in a texture. A more simple, but less efficient method to achieve a similar effect would be to generate a maximal poisson disk set and reject samples afterwards with varying probabilities.
A related issue is the user interface. For this project I hardcoded a specific hairstyle by defining which triangles of a sphere are sampled and which not. It would be better if the user could select the triangles in the interactive 3D editor or even paint densities on them.

3

Simulation of a single strand of hair

Simulating hair or fur requires simulating the strands of hair. The first problem that must be solved is the simulation of a single strand of hair.

3.1 Simulating a chain of particles

A strand of hair is modeled as a chain of particles with equal distances between two neighbor particles in the chain as shown in Figure 3.1. Each particle stores a mass, a position and a velocity. In each time step, external forces $\mathbf{F}$ such as gravity are accumulated for each particle and the particle's acceleration $\mathbf{a}$ is computed using Newton's second law of motion.

$$\mathbf{a} = \frac{\mathbf{F}}{m}$$

Given the acceleration and the timestep t, the particle's velocity $\mathbf{v}$ and position $\mathbf{p}$ are updated. In my implementation I use the explicit Euler-Cromer integration scheme:

$$\mathbf{v} \leftarrow \mathbf{v} + \mathbf{a} \cdot t$$

$$\mathbf{p} \leftarrow \mathbf{p} + \mathbf{v} \cdot t$$

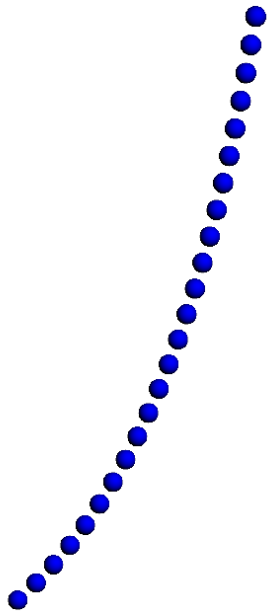


Figure 3.1: A chain of 25 particles

Obviously, two neighbor particles in the chain must be connected somehow, otherwise the particles in the chain could fly apart as a reaction to external forces. Strands of hair are perceived as inextensible, so the length of the chain should be constant or almost constant during the simulation. This can be achieved by demanding that the distance between two neighbor particles should remain constant during the simulation. Therefore geometric constraints are introduced between neighbor particles that restrict the simulation accordingly. Another constraint is required that attaches the first particle of the chain to a point on the skin.
Formally, a constraint that restricts particles with positions $\mathbf{x_1}, \mathbf{x_2}, ..., \mathbf{x_n}$ is encoded with a function $C(\mathbf{x_1}, \mathbf{x_2}, ..., \mathbf{x_n})$ that is 0 if and only if the constraint is satisfied. The following

constraint function represents the distance constraint between neighbor particles with position x_1 and x_2, where d is the distance. It is 0 if and only if the distance between $\mathbf{x_1}$ and $\mathbf{x_2}$ is d:

$$C(\mathbf{x_1}, \mathbf{x_2}) = |\mathbf{x_1} - \mathbf{x_2}| - d$$

The attachment constraint that attaches a particle to a point p can be written similar to the distance constraint, only that the second position is not an input parameter of the constraint function and d is zero:

$$C(\mathbf{x_1}) = |\mathbf{x_1} - \mathbf{p}|$$

Except for the last particle in the chain, all particles are restricted with two possibly conflicting constraints. In the case of a chain that hangs down, fulfilling one constraint usually requires to move the particle upwards, while the other constraint can be fulfilled by moving the particle downwards. The simulation must ensure that all constraints are satisfied at all times. The particles should however still obey Newton's Laws. One general approach to achieve this is to correct errors when they occur, either by adding penalty forces, modifying velocities or modifying positions.

3.2 Position based dynamics

Position based dynamics is a technique that allows to solve geometric constraints by directly manipulating positions. The velocity of a particle is computed afterwards from the difference of the old and new position just like in the Verlet integration scheme. The following C++ code gives an overview over a simulation step with timestep t using position based dynamics. The code is also close to the code I use in my implementation:

```
1  void simulationStep(particles, t)
2  {
3      for (Particle p : particles)
4      {
5          p.xOld = p.x;
6          p.F = accumulateForces(p);
7          p.a = p.F / p.m;
8      }
9      for (Particle p : particles)
10     {
11         p.v = p.v + t * p.a;
12         p.x = p.x + t * p.v;
13     }
14
15     solveConstraints(particles);
16
17     for (Particle p : particles)
18         p.v = (p.x - p.xOld) / t;
19 }
```

First, the old positions of particles are memorized and the acceleration is computed from accumulated forces. After this, the velocities and positions of particles are integrated using the Euler-Cromer integration scheme. In line 15, the particle positions are manipulated so that they satisfy all constraints. Afterwards, the particle velocities are computed according to the memorized old position and the new position.

Obviously, the tricky part is to implement the method solveConstraints. Since a particle may be restricted by multiple constraints and constraints may affect multiple particles, it is necessary to solve for all constraints at the same time. Another problem is that many constraints such as the distance constraint are non-linear, so the particle positions can not be computed by solving a system of linear equations in the general case. [MHHR06] propose an iterative algorithm to solve for the particle positions. In each iteration, the constraints are solved individually one after another. For each constraint the respective particle positions are manipulated directly, so the order in which the constraints are solved matters. They derive a general method to compute the position correction vector $\Delta\mathbf{x_i}$ for a particle with position $\mathbf{x_i}$ that is constrained by a constraint $C(\mathbf{x_1}, ..., \mathbf{x_n})$:

$$\Delta\mathbf{x_i} = -\frac{w_i}{\sum_j w_j |\frac{\partial C(\mathbf{x_1},...,\mathbf{x_n})}{\partial \mathbf{x_j}}|^2} C(\mathbf{x_1}, ..., \mathbf{x_n}) \frac{\partial C(\mathbf{x_1}, ..., \mathbf{x_n})}{\partial \mathbf{x_i}}$$

The gradient $\frac{\partial C(\mathbf{x_1},...,\mathbf{x_n})}{\partial \mathbf{x_i}}$ points into the direction of maximal change. The gradient is multiplied with $C(\mathbf{x_1}, ..., \mathbf{x_n})$ and weighted according to the mass of the particle in relation to the masses of the other particles that are affected by the constraint. w_i is the inverse mass of the i-th particle. Formulating the constraint projection with inverse masses allows to consider particles with infinite mass where $w_i = 0$. In this case the particle with infinite mass is not moved at all. For example, this is useful for the hair particle that is attached to the skin.
One advantage of position based dynamics is that attachment constraints can be satisfied by simply setting the positions of particles. This can also be derived with the formal method described above. The attachment constraint for a particle with position $\mathbf{x_1}$ attached to a position $\mathbf{p}$ is $C(\mathbf{x_1}) = |\mathbf{x_1} - \mathbf{p}|$. Since $\frac{\partial(|\mathbf{x_1}-\mathbf{p}|)}{\partial \mathbf{x_1}} = \frac{\mathbf{x_1}-\mathbf{p}}{|\mathbf{x_1}-\mathbf{p}|}$ and $|\frac{\mathbf{x_1}-\mathbf{p}}{|\mathbf{x_1}-\mathbf{p}|}| = 1$, the correction vector is:

$$\Delta\mathbf{x_1} = -\frac{w_1}{w_1 |\frac{\partial(|\mathbf{x_1}-\mathbf{p}|)}{\partial \mathbf{x_1}}|^2} (|\mathbf{x_1} - \mathbf{p}|) \frac{\partial(|\mathbf{x_1} - \mathbf{p}|)}{\partial \mathbf{x_1}} = -(\mathbf{x_1} - \mathbf{p}) = \mathbf{p} - \mathbf{x_1}$$

Applying the method yields the expected result: The constraint can be solved by simply setting $\mathbf{x_1}$ to position $\mathbf{p}$.

For the distance constraint $C(\mathbf{x_1}, \mathbf{x_2}) = |\mathbf{x_1} - \mathbf{x_2}| - d$ with $\frac{\partial(|\mathbf{x_1}-\mathbf{x_2}|-d)}{\partial \mathbf{x_1}} = \frac{\mathbf{x_1}-\mathbf{x_2}}{|\mathbf{x_1}-\mathbf{x_2}|}$ and $\frac{\partial(|\mathbf{x_1}-\mathbf{x_2}|-d)}{\partial \mathbf{x_2}} = \frac{\mathbf{x_2}-\mathbf{x_1}}{|\mathbf{x_1}-\mathbf{x_2}|}$ and $|\frac{\mathbf{x_1}-\mathbf{x_2}}{|\mathbf{x_1}-\mathbf{x_2}|}| = |\frac{\mathbf{x_2}-\mathbf{x_1}}{|\mathbf{x_1}-\mathbf{x_2}|}| = 1$ the correction vector $\mathbf{x_1}$ is:

$$\begin{aligned}\Delta\mathbf{x_1} &= -\frac{w_1}{w_1 |\frac{\partial(|\mathbf{x_1}-\mathbf{x_2}|-d)}{\partial \mathbf{x_1}}|^2 + w_2 |\frac{\partial(|\mathbf{x_1}-\mathbf{x_2}|-d)}{\partial \mathbf{x_2}}|^2} (|\mathbf{x_1} - \mathbf{x_2}| - d) \frac{\partial(|\mathbf{x_1} - \mathbf{x_2}| - d)}{\partial \mathbf{x_1}} \\ &= -\frac{w_1}{w_1 + w_2} (|\mathbf{x_1} - \mathbf{x_2}| - d) \frac{\mathbf{x_1} - \mathbf{x_2}}{|\mathbf{x_1} - \mathbf{x_2}|} \qquad \text{([MHHR06])}\end{aligned}$$

The correction vector for the second vector is derived analogous:

$$\Delta\mathbf{x_2} = -\frac{w_2}{w_1 + w_2} (|\mathbf{x_1} - \mathbf{x_2}| - d) \frac{\mathbf{x_2} - \mathbf{x_1}}{|\mathbf{x_1} - \mathbf{x_2}|}$$

Intuitively, the two particles are simply moved towards each other. If one particle is heavier than the other, it is moved less than the other one. If both poth particles have the same mass, the term $\frac{w_2}{w_1+w_2}$ is always $\frac{1}{2}$.

Now we have all the tools to implement the method solveConstraints for the particle chain. In each iteration all neighbor particles are moved towards each other. The particles are

always processed in the same order, from the attached particle towards the last particle in the chain. The particle positions are always corrected directly, so the computations of correction vectors take into account previous corrections. An infinite mass is assigned to the attached particle, so it is not necessary to process the attachment constraint inside the solver loop because the particle will never be moved by a distance constraint.

[MHHR06] also incorporate a stiffness constant k that ranges from 0 to 1 into their algorithm. They multiply the correction vectors inside the solver loop with $k' = 1 - (1 - k)^{1/n_s}$, where n_s is the number of the iteration. This ensures that k has a linear effect. For the simulation of inextensible strands of hair this is not relevant, because a stiffness of 1 is required. However in tests I noticed that the convergence of the solver can be improved by using a k that is higher than 1. The k is multiplied directly with the correction vectors, so the effect is non-linear. In the each iteration, particles may be moved closer to or too far from each other than they should be. This is alway corrected in the following iteration, so the error becomes smaller with each iteration. The intuition behind this is that information is propagated faster across the particle chain. In section 3.4.1 the convergence improvement is evaluated in detail.

3.3 Dynamic follow-the-leader

Simulating a hair strand with the method above requires multiple iterations per timestep and always allows a certain amount of stretching. Dynamic follow the leader is a technique proposed by [MKC12] that guarantees zero stretch with one single iteration, while introducing some damping. The algorithm is based on position based dynamics, only the method solveConstraints is implemented differently. The first particle in the chain is called the leader particle. In the case of a hair strand this is the particle that is attached to the skin. The other particles are processed in the order from the second particle to the last. Each particle is moved towards its predecessor so that the corresponding distance constraint is satisfied. After one iteration, all distance constraints are satisfied. This is a special case of the method described in the section above. Each particle has infinitely less mass than its predecessor in the particle chain. Each distance constraint is satisfied by only moving the successor, the predecessor is not moved at all. Obviously, such an object is physically impossible. Without further changes to the PBD velocity update it also does not show the desired behavior. Forces added to a particle in the hair strand, for example due to a collision response or wind, have no effect on predecessor particles which looks unrealistic. The second problem is that an undamped simulation with this approach shows strange and unwanted behavior. [MKC12] documented this behavior with images that show a series of simulation frames. Figure 3.2 shows one of these imagaes from their paper. The rope should just fall down, instead it spins around. [MKC12] propose a special damping that addresses both problems. They derive a velocity correction that hides the uneven mass distribution and eliminates the strange behavior.

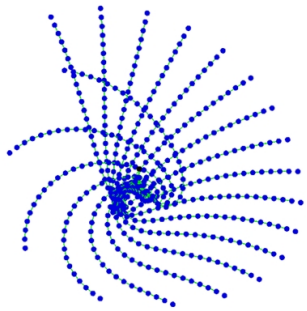


Figure 3.2: FTL without damping [MKC12]

$$v_i \leftarrow v_i + s_{damping} \frac{-d_{i+1}}{\Delta t}$$

v_i and x_i are the velocity and position of the particle i. d_{i+1} is the follow-the-leader correction vector for the successor particle. $s_{damping}$ is a damping coefficient between 0 and 1. The velocity correction is applied right after the velocity is computed using the difference of new and old position as described in section 3.2 (in the original paper, velocity update and correction are combined into one equation). For the last particle, d_{i+1} is obviously not defined, so no correction step is performed for this particle.

For a particle i, the correction vector $-d_{i+1}$ points into the direction of the particle $i+1$. The velocity correction of particle i emulates that the particle was moved towards particle $i+1$ inside *solveConstraints*, although it was not. This propagates information from successor particles to predecessor particles, so that applying forces to particles affects predecessor particles, resulting in a natural bending of the hair strand. With $s_{damping} = 1$, the velocity correction emulates that the particle i was moved towards particle $i+1$ the same amount particle $i+1$ was moved towards particle i. This completely hides the uneven mass distribution, however it also introduces a large amount of damping. Choosing $s_{damping}$ is a tradeoff between hiding the uneven mass distribution and adding damping to the simulation. In tests, $s_{damping} = 0.9$ yielded good results. $s_{damping}$ should be at least 0.8 to avoid the strange behavior shown in Figure 3.2. The strange behavior can also be eliminated by performing the velocity correction with $-d_i$ instead of $-d_{i+1}$, which equals simple point damping. This however does not hide the uneven mass distribution, so successor particles will not affect predecessor particles.

The simplest way to implement dynamic follow-the-leader is to memorize the correction vectors for all particle inside *solveConstraints* and use them later to perform the velocity correction. It is also possible to perform the velocity correction without memorizing all correction vectors. This can be done by updating and correcting the velocity of a particle i immediately after the FTL projection is performed for it.

3.4 Evaluation

3.4.1 PBD convergence

[MKC12] writes that roughly 25 PBD iterations are necessary to achieve an inextensible look in typical situations. In this section I will examine the required number ob PBD iterations a little closer and investigate how convergence can be improved by using a higher stiffness constant. Therefore I measure the required number of iterations to achieve an inextensible look. In each update step the solver performs iterations until the length of the rope is less than the original rope length multiplied with 1.05 and longer than the original rope length multiplied with 0.95, allowing a stretch of 5 percent. All tests where performed without any damping and a timestep of 0.01.

Rope falling down with 30 particles, maximal 5 percent stretch:

Particle distance	Stiffness k	Iterations avg	Iterations max
0.04	1.0	5.3	13
0.04	1.65	4.0	4
0.02	1.0	8.6	20
0.02	1.65	4.0	6

Rope with 30 particles moving with 2 units per second, maximal 5 percent stretch:

Particle distance	Stiffness k	Iterations avg	Iterations max
0.04	1.0	17.5	60
0.04	1.6	4.4	10
0.02	1.0	22.7	93
0.02	1.6	4.7	9

Rope with 30 particles moving with 4 units per seconds, maximal 5 percent stretch:

Particle distance	Stiffness k	Iterations avg	Iterations max
0.04	1.0	51.0	118
0.04	1.65	10.3	22
0.02	1.0	74.1	134
0.02	1.65	14.4	30

The tables show that the number of required iterations depends on the velocity of the rope's attached particle. It also depends on the distance between particles. With the same number of particles, more iterations are required when the distance between particles is smaller. It is not possible to give one number of iterations that achieves an inextensible look in all situations, however 25 iterations seems reasonable for typical scenarios. Especially for interactive applications, the maximal number of required iterations is crucial, because an interactive application is usually supposed to keep at least a certain framerate. In this case, one would probably use a maximum number of iterations and accept more stretching in extreme cases.

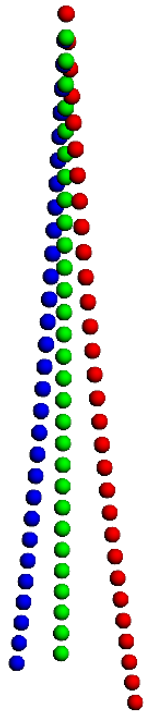


Figure 3.3: Blue: 4 PBD iterations with stiffness 1.6, Red: 4 PBD iterations with stiffness 1.0, Green: FTL

Using a stiffness multiplier of 1.65 is a huge improvement in all scenarios. However it requires a certain amount of iterations, otherwise overshooting problems become visible. For this reason a minimum iteration count of 4 was used for all tests. Also, the stiffness constant must not be chosen too high. Values higher than 1.7 almost always lead to problems. The behavior of the rope is almost identical for both stiffness constants but different iteration counts. Figure 3.3 demonstrates the improvement with a fixed iteration count. It shows a rope simulated with 4 PBD iterations and a stiffness constant of 1, a rope simulated with 4 PBD iterations and a stiffness constant of 1.6 and a rope simulated with dynamic follow-the-leader. All ropes have 30 particles and should have the same length. The rope simulated with dynamic follow-the-leader is inextensible, while the two ropes with PBD show some stretching. The blue rope simulated with a stiffness constant of 1.65 is stretched less than the red rope with the same number of iterations.

Another very simple method to speed up convergence is to assign different masses to the

particles. The second particle gets less mass then the first, the third less then the second and so on. This changes the behavior of the rope, but the result still looks decent. Introducing an uneven mass distribution is similar to what FTL does, however no damping is required. I tested the rope with 30 particles moving with $4m/s$, where the i-th particle in the rope has mass $1/i$ (the attached first particle has mass infinity like before).

Particle distance	Stiffness k	Iterations avg	Iterations max
0.04	1.0	26.6	65
0.04	1.65	5.5	11
0.02	1.0	38.6	78
0.02	1.65	7.0	14

When using this mass distribution only half the iterations are required compared to the even mass distribution. Using this uneven mass distribution and a stiffness constant of 1.65 results in a speedup of factor 10 for this scenario.

If a rope has the required length, it does not mean that all distance constraints are satisfied because the particles may be uneven distributed. For the rope with $k = 1.65$, particles tend to be closer to each other at the end of the rope (this is hardly visible however). To ensure that both algorithms compute the same result, I performed iterations until every distance between particles is correct with a tolerance of 10 percent. The scenario is again the rope with 30 particles (equal masses), moving with $4m/s$:

Particle distance	Stiffness k	Iterations avg	Iterations max
0.04	1.0	42.0	95
0.04	1.65	22.9	76
0.02	1.0	62.5	113
0.02	1.65	33.1	56

As expected, the advantage for the rope simulated with a higher stiffness constant becomes smaller. Especially the maximum number of iterations increases in some situations. Still, it is significantly faster than the version with $k = 1$.

Altogether, PBD convergence depends on many parameters. Not investigated here was the timestep and the number of particles. In any case, choosing a stiffness multiplier higher than 1 is a very simple and effective method to speed up convergence in the special case of rope simulation. Combined with an uneven mass distribution an inextensible look can be achieved with only a few iterations for typical scenarios.

3.4.2 Dynamic follow-the-leader

Dynamic follow-the-leader is especially attractive for interactive applications, because it guarantees zero stretch with one single iteration in every situation. Especially for long hair that is moved with a high velocity this is a huge advantage compared to other approaches such as iterative PBD. The performance is very good and the algorithm is very simple to implement. It introduces a large amount of damping, however that does not look very unrealistic in the case of hair strands. Also, the damping does not affect external forces such as gravity so it is in some way better than point damping. In fact it is difficult to reproduce a damping similar to the FTL damping when using the iterative PBD solver. Another issue is that with damping coefficients less than 1 the rope moves differently due to the uneven mass distribution. However this still looks believable and and may be even

a wanted behavior. All in all, the biggest advantage of the iterative solver compared to dynamic follow-the-leader is flexibility. Other constraints can be incorporated easily and any kind of damping may be used. If the special damping and uneven mass distribution are acceptable and the time budget is small, then FTL is preferable.

Dynamic follow the leader guarantees zero stretch, but even with the damping it is not unconditionally stable. In my tests I sometimes noticed strange behavior as shown in Figure 3.4, especially with high values for $s_{damping}$. This is especially a problem for low timesteps and small distances between particles. In practice it is not difficult to find a set of parameters that achieves the desired result, but it must be considered when implementing dynamic FTL.

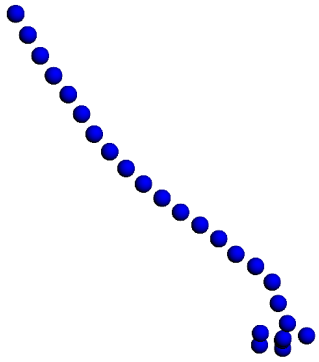


Figure 3.4: FTL is not unconditionally stable

4

Hair-hair interactions

To achieve realistic results it is necessary to take interactions between strands of hair into account. Figure 4.1 shows what can happen when hair is simulated without any interactions. The strands of hair pass through each other without resistance and often move indepentantly from each other, which looks unrealistic in many situations. Unfortunately, the interactions between strands of hair are very complex and no standard physically-based model exists yet ([WBK+07]).

4.1 Related work

Typically most strands of hair are permanantly in contact with other strands. Therefore a group of approaches model hair-hair interactions under the assumption that hair is a continuous medium. Many of these approaches are based on fluid dynamics. In contrast to hair dynamics, fluid dynamics are well researched and understood.

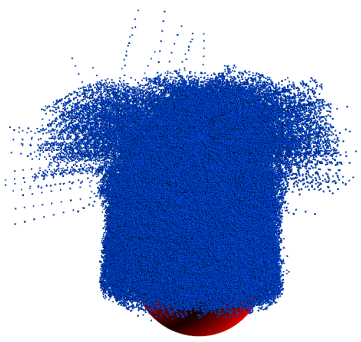


Figure 4.1: Hair falling down without hair-hair interactions

[HMT01] uses Smoothed Particle Hydrodynamics (SPH) that allows to simulate fluids represented with particles. If the strands of hair are already modeled with particles, this has the advantage that the same hair representation is used for hair-hair interactions and single strand of hair simulation. However if the distance between particles in a strand of hair is too high it may be necessary to insert additional fluid particles between them.
For each hair respectively fluid particle a density is computed based on particles in a neighborhood radius called the smoothing length. For instance, a particle with many close neighbors will get a high density. Given the difference between this density and the rest density, a pressure is computed using the state equation. A pressure force is applied which effectively moves particles away from each other that are too close. Also, artificial viscosity forces are added that approximate friction between hair particles. It is also possible to handle collision with rigids with this approach by sampling the rigids with boundary particles that exert a force on close fluid particles. It is usually not required that the hair also influences the rigids, so two-way coupling is not required.

4.2 Voxel grid based interactions

Computing the hair-hair interactions on the particle level can be very time consuming. The bottleneck is the neighborhood search which depending on the smoothing length usually takes much more time than any other part of the simulation. To accelerate the computation of hair-hair interactions, [PHA05] propose to construct a voxel grid in each simulation step that represents the hair volume and compute the interactions using this grid. The hair volume is divided into cells of equal size. Each cell is a volumetric sample called voxel that stores a density and a velocity. For this project I implemented the approach proposed by [MKC12] that is based on the work of [PHA05]. Given the grid that stores velocities and densities, two forces are computed for each particle. Hair friction approximates friction between colliding strands of hair. Hair repulsion approximates the collision response.

Figure 4.2: Hair falling down with velocity smoothing

The grid is constructed in every simulation step by inserting the particles into an empty grid where all densities and velocities are zero. A particle is inserted into the grid by distributing its velocity between the eight neighbor cells according to the trilinear interpolation weights of the particle's position. Also, the density with value 1 is distributed between the neighbor cells. After all particles are inserted, the velocities are normalized by divding them by the density stored in their cells. Each voxel now stores a weighted average of particle velocities. Now the grid can be used to simulate hair-hair interactions.

Hair friction can be approximated by replacing the velocity $\mathbf{v}$ of a particle with a (possibly weighted) average velocity of neighbor particles. A lookup of a velocity in the grid at the particle's position using trilinear interpolation yields such a weighted average velocity $\mathbf{v}_{grid}$ of neighbor particles. Instead of a particle radius, the cell size determines which neighbor particles contribute to the weighted average. Larger cell sizes mean that the velocities of more neighbor particles are integrated into the weighted average. Additionally, the parameter $s_{friction}$ is introduced to control the amount of friction.

$$\mathbf{v} \leftarrow (1 - s_{friction})\mathbf{v} + s_{friction}\mathbf{v}_{grid}$$

To simulate hair repulsion, the normalized density gradient $\mathbf{g}$ at the particle's position is computed. The density gradient points into the direction of maximum change. The particle's velocity is changed so that the particle will move into this direction, steering it away from other particles. The parameter $s_{repulsion}$ is a multiplier that controls the repulsion force strength.

$$\mathbf{v} \leftarrow \mathbf{v} + \frac{s_{repulsion}\mathbf{g}}{\Delta t}$$

[MKC12] does not explain why the density gradient is normalized. A possible reason could be that the amount of hair repulsion is simpler to control when the gradient is normalized. I also tested using the non-normalized density gradient. When using the non-normalized density gradient, it is more convenient to treat it as a force and not as a position offset, otherwise very low values for $s_{repulsion}$ are required. The velocity update changes to:

$$\mathbf{v} \leftarrow \mathbf{v} + s_{repulsion}\mathbf{g} \cdot \Delta t$$

The gradient is computed using the finite difference method. Let $d(x, y, z)$ be the function that maps from positions to densities and r the hair radius.

$$\mathbf{g} = \begin{pmatrix} \partial d/\partial x \\ \partial d/\partial y \\ \partial d/\partial z \end{pmatrix} \approx \begin{pmatrix} (d(x-r,y,z) - d(x+r,y,z))/2r \\ (d(x,y-r,z) - d(x,y+r,z))/2r \\ (d(x,y,z-r) - d(x,y,z+r))/2r \end{pmatrix}$$

$d(x, y, z)$ is implemented as a trilinear interpolation of the density values stored in the 8 neighbor cells of the position x, y, z. Hence, computing the gradient requires 6 trilinear interpolations, making it the performance critical operation in the voxel grid based approach.

4.2.1 Trilinear interpolation

To implement hair repulsion and friction as described above, two basic operations are required: The trilinear insert takes a position and a value and distributes it between the neighbor cells at that position according to trilinear interpolation weights. The second basic operation is a lookup at a position, performing a trilinear interpolation to retrieve the value at that position. Given a grid data structure that provides the method *lookup*(x, y, z) retrieving the voxel at integer coordinate x, y, z, the trilinear interpolation can be implemented with the following C++ code:

```
1  float trilinearInterpolation(const vec3& position)
2  {
3      vec3 scaledPosition = position * m_InverseCellSize;
4      int cellX = floor(scaledPosition.x());
5      int cellY = floor(scaledPosition.y());
6      int cellZ = floor(scaledPosition.z());
7      float x = scaledPosition.x() - cellX;
8      float y = scaledPosition.y() - cellY;
9      float z = scaledPosition.z() - cellZ;
10     return m_Grid.lookup(cellX, cellY, cellZ) * (1-x) * (1-y) * (1-z)
11         + m_Grid.lookup(cellX, cellY, cellZ+1) * (1-x) * (1-y) * z
12         + m_Grid.lookup(cellX, cellY+1, cellZ) * (1-x) * y * (1-z)
13         + m_Grid.lookup(cellX, cellY+1, cellZ+1) * (1-x) * y * z
14         + m_Grid.lookup(cellX+1, cellY, cellZ) * x * (1-y) * (1-z)
15         + m_Grid.lookup(cellX+1, cellY, cellZ+1) * x * (1-y) * z
16         + m_Grid.lookup(cellX+1, cellY+1, cellZ) * x * y * (1-z)
17         + m_Grid.lookup(cellX+1, cellY+1, cellZ+1) * x * y * z;
18 }
```

In line 3, the position is divided by the cell size of the voxel grid by multiplying with the inverse cell size. This means that the following trilinear interpolation code can assume a cell size of 1. Now the integer cell coordinates of the neighbor voxel cell left, below and in front of *position* can be computed by simply rounding using *floor*. If it is ensured that the x, y, and z component of *position* are always positive the integer cell coordinates can also be obtained with a cast to *int*, which is much more efficient.
The trilinear insert implementation looks very similar. Line 2 - 9 remain the same. Line 10-17 change to:

```
1  float trilinearInsert(const vec3& position, float value)
2  {
3      // ...
```

```
4       m_Grid.lookup(cellX, cellY, cellZ) += value * (1-x) * (1-y) * (1-z)
5       m_Grid.lookup(cellX, cellY, cellZ+1) += value * (1-x) * (1-y) * z
6       m_Grid.lookup(cellX, cellY+1, cellZ) += value * (1-x) * y * (1-z)
7       m_Grid.lookup(cellX, cellY+1, cellZ+1) += value * (1-x) * y * z
8       m_Grid.lookup(cellX+1, cellY, cellZ) += value * x * (1-y) * (1-z)
9       m_Grid.lookup(cellX+1, cellY, cellZ+1) += value * x * (1-y) * z
10      m_Grid.lookup(cellX+1, cellY+1, cellZ) += value * x * y * (1-z)
11      m_Grid.lookup(cellX+1, cellY+1, cellZ+1) += value * x * y * z;
12  }
```

In both examples, the grid only stores floating point values and thus the trilinear interpolation function retrieves a *float* and the insert takes a *float* as parameter. However the basic principle is the same no matter whether the grid stores floating point numbers, 3D vectors or both. In my implementation both operations and the underlying grid data structure are implemented as templates that work with any data type for which the multiplication operator and the +=-operator are defined. In C++ it is possible to overload operators, which means that structs or classes can be used that store a collection of properties.

4.2.2 Grid data structures

To implement the trilinear interpolation and insert the grid data structure only needs to provide random access to a cell at integer coordinates x, y, z.

One possibility is to implement the grid is spatial hashing. A one dimensional arrays of fixed size stores buckets of voxels. A hash function maps integer coordinates x, y, z to a single integer value in the range from 0 to the size of the array. To lookup a voxel at x, y, z, the integer coordinate is converted to an array index using the hash function. Since multiple coordinates may be mapped to the same index, the corresponding bucket may contain multiple voxels. A linear search is required to find out whether the voxel with coordinates x, y, z is stored in the bucket. If the voxel was found it is returned, otherwise the default value for empty cells is returned. This does only work for the lookup that is used for the trilinear interpolation. For the trilinear insert a new voxel with the default value is inserted into the bucket and a reference to it is returned. In my implementation the grid provides a method *lookup* that is used by the trilinear interpolation and a method *lookupOrCreate* that returns a reference and is used by the trilinear insert.
The hash function is crucial for the performance of the spatial grid. It should minimize conflicts, so that few buckets contain more than one voxel. For spatial hashing, the following hash function is usually used:

$$h(x, y, z) = ((xp_1) \text{ xor } (yp_2) \text{ xor } (zp_3)) \mod n$$

n is the size of the array and p_1, p_2 and p_3 are large primes. In my implementation the primes $p_1 = 73856093$, $p_2 = 19349663$ and $p_3 = 83492791$ are used.

Compared to a simple 3D array, spatial hashing has several big advantages. Firstly, it does not store empty cells and is therefore very memory efficient. Secondly, it is not necessary to know the boundary of the hair volume in advance. Voxels at any coordinates can be stored inside the grid. On the other hand, the lookup and insert becomes computationally more expensive, because the hash function must be evaluated and conflicts may occur. Although it is never the case in praxis, the theoretical worst case runtime complexity for a lookup is linear, because all inserted voxels may be stored in the same bucket.

Since the voxel grid method is intended for interactive framerates where even milliseconds are critical I wanted to investigate how much performance is sacrificed compared to a three-dimensional array. When allocating the 3D array the AABB of all hair particles must be considered so that all particles fit into the array. Since the hair is attached to a mesh the size of this AABB is limited. If the maximum hair length is known, it is possible to allocate the 3D array once and avoid reallocations during runtime. This is the case when using dynamic follow-the-leader, because it guarantees zero hair stretch. When using iterative PBD an adaptive number of iterations must be used to guarantee that the hair stretch is below a certain percentage. In my computation I compute the maximum AABB once when the hair is created, given the sample positions and the maximum hair length. The minimum position of this AABB is memorized and the three-dimensional array is allocated. Whenever a trilinear insert or lookup is performed the memorized minimum AABB position and the current position of the sample mesh are added to the position first.

Another issue with the 3D array concerns the normalization of the velocities. Dividing the velocity by the density for every cell may be inefficient, because typically there are more empty cells than non-empty cells. Depending on the cell size there may be more cells than particles. When using a very large hash table to minimize conflicts this is also an issue for the spatial hash grid. This is why I skip the normalization of the cells and divide the interpolated velocity by the interpolated density for each particle when computing friction. Probably a better solution would be to keep track of used cells using an additional data structure.

4.2.3 Performance evaluation

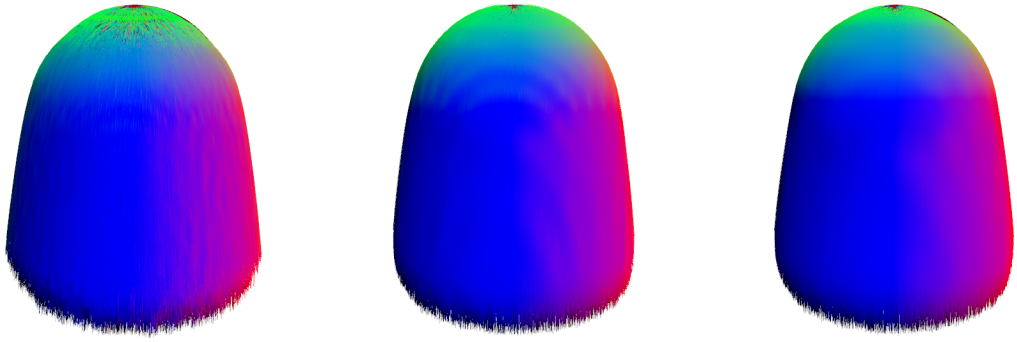
The table shows the time one simulation frame takes on average (the average was not exactly computed though) while the hair is moved around. This does not only include hair-hair interactions but also Euler-Cromer integration, FTL projection and velocity update. The cell size is set to 0.02, the distance between particles is 0.02 and the timestep is 0.01. Two versions of the 3D array were benchmarked. The safe 3D array performs a check before every lookup whether the given coordinate x, y, z is inside the array. The other 3D array implementation performs no checks and will crash if a cell is accessed that does not exist.

Particles	Grid	Frame time
240372	Hash grid	120ms
240372	3D array safe	70ms
240372	3D array	45ms
240372	No interactions	16ms
425558	Hash grid	210ms
425558	3D array safe	120ms
425558	3D array	75ms
425558	No interactions	27ms

The timings show a significant difference for the different grid implementations. The timings are more than doubled for the hashgrid compared to the 3D array. The average bucket length (not counting empty buckets) in the hashgrid is 1.05, so only a few buckets contain more than one voxel. The main reason for the performance difference seems to be that for each lookup the hash function is evaluated and the given position must be compared to the position of the voxel stored in the bucket. The timings for the safe 3D array implementation with overflow checks support this theory. The safe 3D array is significantly slower than the

3D array implementation without checks, however still faster than the hash grid because no hash function has to be evaluated. When interactive framerates are the goal, these performance differences matter and a 3D array should be used if possible.
The timings also show that hair-hair interactions are the performance critical task in the simulation step when using dynamic follow-the-leader. Computing the hair-hair interactions requires more than 70 percent of the simulation frame time even for the fastest grid implementation.

4.2.4 Grid cell size



(a) Cell size 0.01 (b) Cell size 0.02 (c) Cell size 0.03

Figure 4.3: Cell size comparison

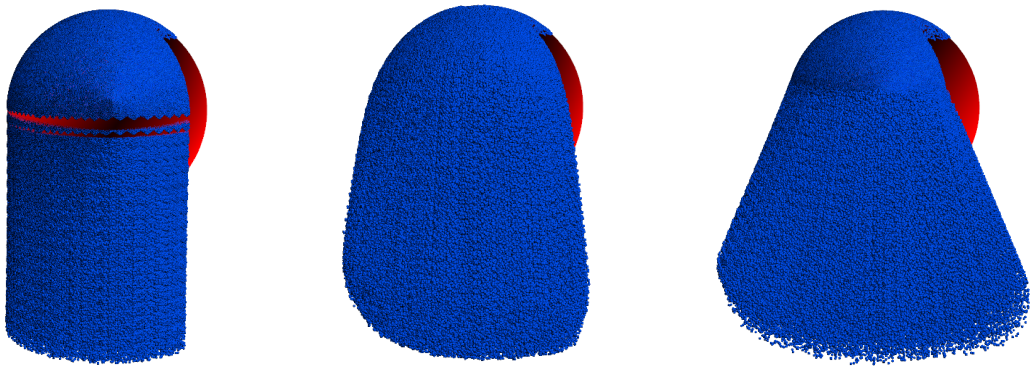
The grid cell size is crucial for the behavior of both hair repulsion and friction. Figure 4.3 visualizes the density gradients for hair that roughly consists of 450K particles. The particle distance in a strand of hair is 0.02 . The distance between two hair particles attached to the skin is at least 0.0015.
Coarser grids produce smoother gradients, however the gradients are less accurate. Figure 3.3 shows that finer grids are not always superior compared to coarser grids. With the cell sizes 0.01 and 0.02 patterns due to the trilinear interpolation become visible. For the velocity smoothing the cell size is even more important because it influences the amount of smoothing. It is also possible to use separate grids for densities and velocities with different cell sizes.
Altogether, it is difficult to state a good cell size for all situations, because it depends on many factors.

4.2.5 Collision with rigids

For this project I only considered collision detection against the rigid the hair is attached to, which is sufficient for most cases. For their demo featuring long hair, [MKC12] perform collision detection for each particle against ellipsoids that represent the head and shoulders. This requires to setup the ellipsoids and synchronize them with the rigid. For this project I tried out two other approaches based on voxel grids.

With hair repulsion

[MKC12] writes that for fur it is not required to handle collision with the character because hair repulsion moves particles away from the skin. I tried to exploit this effect further by filling the border of the rigid volume with particles of high density. Therefore the rigid is sampled with the Poisson mesh sampling algorithm discussed in the last section. The samples are generated only once and stored as barycentric coordinates. Just like the hair samples, the global sample positions for the samples are computed every frame from the barycentric coordinates. For each sample, a high density is inserted into the density grid used for hair repulsion at a position below the sample. This position is computed using the surface normal at that position. It is important to insert a velocity at that position too, because the density grid is not only used for repulsion but also to normalize the velocities. Therefore the velocity of the rigid at that position multiplied with the added density is inserted. Figure 4.4 shows an example where this approach produces good results. The advantage of this method is that rigid collision is handled with little additional cost if hair repulsion is used anyway. However it does not work with no or insufficient hair repulsion. Also, for long hair that is moved around quickly hair may enter the rigid.



(a) No repulsion (b) Non-normalized gradient (c) Normalized gradient

Figure 4.4: Repulsion comparison

The figure shows that in this case it is really beneficial to not normalize the gradient. If the gradient is normalized the repulsion force is not strong enough to even out gravity and the hairs at the top fall inside the sphere rigid. If the gradient is not normalized the hairs do not fall inside the rigid without any additional collision handling.

With a separate grid

It is also possible to use a separate grid for the collision detection and response. Each voxel inside the rigid stores a vector that points towards the closest point on the surface. Voxels outside the rigid store the null vector. Hence, a lookup in the grid yields a collision response vector. For each particle a collision response vector is computed by interpolating the grid at the particle position. The main challenge is to generate the grid. In my implementation I only implemented a very slow brute force method that iterates over all voxels and all triangles to find the closest surface point for each voxel. Even for low poly meshes this takes a lot of time and is only feasible if the grid is only generated once. The advantage of this method is that it it is independant from the amount of used hair repulsion. Since the rigid

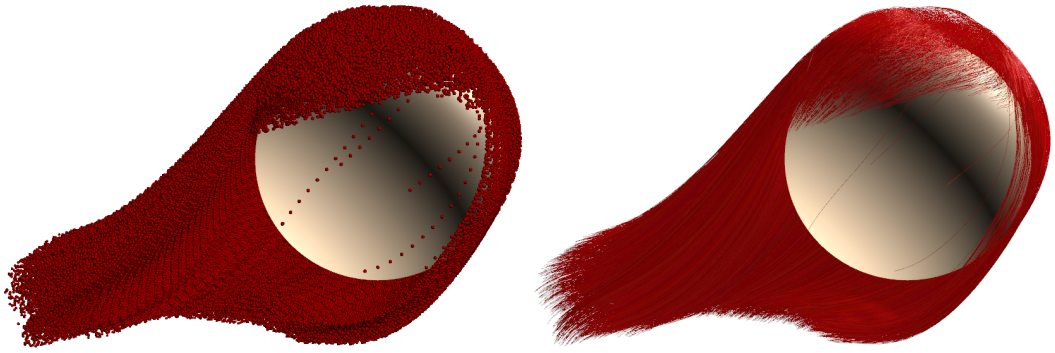
body volume is completely filled with response vectors and not only the border, this also works for high velocities when particles penetrate deeply into the volume. I had not the time to really investigate this approach further. Besides, for fur it is often possible to only use hair repulsion and for hair it is probably more efficient to check for collision against primitive objects like [MKC12] did.

5 Rendering

For this project the hair was rendered with OpenGL and some details in this section are OpenGL specific. The general approaches should also translate to DirectX and even ray tracing systems. The focus of this project was on simulation and not and rendering.

5.1 Hair geometry

The simplest method to render hair is to use a line strip for each strand of hair. Each particle in the strand of hair is a vertex of the line strip. The line width can be controlled with *glLineWidth(x)*. Especially when using a line width below 1, anti-aliasing should be enabled using *glEnable(GL_LINE_SMOOTH)*.
Another possibility is to render each strand of hair as a camera oriented triangle strip. This has the advantage that a texture can be used for the hair strand instead of only a color. For this projects I used line strips, which are sufficient for many purposes. Figure 5.1 shows how the simulated particles are rendered using OpenGL line strips.



(a) Particles (b) Line strips

Figure 5.1: From particle chains to hair

When vertices are only created at particle positions the result will only look good if the distance between particles is very small. Otherwise, edges will be visible where the hair bends. This problem can be solved by adding vertices at interpolated positions, so that the rendered hair has more detail than the simulated hair. [ND05] uses Bezier curves to

tesselate strands of hair. [Tar08] uses B-Splines that require to solve a system of linear equations for each strand of hair. Both methods make it possible to sample strands of hair with an arbitrary number of vertices, independant of the number of particles. This allows to use much fewer particles while achieving a smooth look, which es especially important when interactive framerates are the goal. In OpenGL or DirectX, hardware tesselation can be used to generate the vertices on the GPU. For this project I did not implement hair tesselation and instead use a rather small distance between simulated particles.

Tesselation allows to use fewer particles for each strand of hair. A lot of approaches do not only interpolate additional vertices inside a strands of hair but entire hairs. This approach is known as the key hair method, where additional strands of hair are interpolated from the simulated key hairs. Usually only a few hundred key hairs are simulated and all other hairs are interpolated. The key hair method is also widely used when hair is not simulated but animated by hand. This is especially important in the movie industry, where artists want to have maximum control over the hair for dramatic effect. Animating each strand of hair would be impossible, instead only the key hairs are animated and the other strands of hair are interpolated. Since the focus of this project is on the simulation, the key hair method was not implemented.

5.2 Lighting

A first simple step towards a realistic look is to render the hair strands with different colors. Small random color variations add some structure even without any lighting, as shown in Figure 5.2.

Figure 5.2: Hair strands with small color variations

[KK89] developed a phenomenological model for hair lighting. Published in 1989, their approach is still popular because it is very simple to implement and can achieve good results. Similar to the Phong lighting system, it computes a diffuse and specular component for each hair fragment. Instead of a surface normal, it requires the tangent of the strand of hair at each fragment. This means that for each vertex a tangent must be computed, so that the tangents for the fragments can be interpolated by OpenGL just like normal vectors. Let $\mathbf{x_i}$ be the position of the particle i. I use the central difference method to compute the tangent $\mathbf{t_i}$ for particle i:

$$\mathbf{t_i} = \frac{\mathbf{x_{i+1}} - \mathbf{x_{i-1}}}{||\mathbf{x_{i+1}} - \mathbf{x_{i-1}}||}$$

For the first and last particle the tangent is computed using the forward respective backward difference method. In any case, the tangent points downwards the strand of hair. The tangent is passed to the vertex shader as a texture coordinate using *glMultiTexCoord3f(GL_TEXTURE0, t.x, t.y, t.z)*.

Given the tangent $\mathbf{t}$ and the light direction $\mathbf{l}$ pointing from the fragment to the light, the diffuse component is:

$$Diffuse = \sin(\mathbf{t}, \mathbf{l}) = \sqrt{1 - dot(\mathbf{t}, \mathbf{l})^2}$$

Computing the specular component also requires the camera direction $\mathbf{e}$ pointing from the fragment to the camera:

$$Specular = [dot(\mathbf{t}, \mathbf{l}) \cdot dot(\mathbf{t}, \mathbf{e}) + \sin(\mathbf{t}, \mathbf{l}) \cdot \sin(\mathbf{t}, \mathbf{e})]^p$$
$$= [dot(\mathbf{t}, \mathbf{l}) \cdot dot(\mathbf{t}, \mathbf{e}) + \sqrt{1 - dot(\mathbf{t}, \mathbf{l})^2} \cdot \sqrt{1 - dot(\mathbf{t}, \mathbf{e})^2}]^p$$

The exponent p controls the shininess of the surface very similar to the specular exponent in the Phong illumination model.

A general problem with the specular component is that strong highlights often look too uniform, lacking structure. This problem is especially visible when the hair is not moving. To overcome this problem, [Tar08] proposed to add random offsets $\mathbf{r}$ to tangent vectors. This simple trick improves the visual quality of the specular lighting considerably in many situations, as demonstrated in Figure 5.3. It is important that the same offset is added to each particle in all frames, otherwise the result will flicker.

$$\mathbf{t_i} = \frac{\mathbf{x_{i+1}} - \mathbf{x_{i-1}} + \mathbf{r}}{||\mathbf{x_{i+1}} - \mathbf{x_{i-1}} + \mathbf{r}||}$$

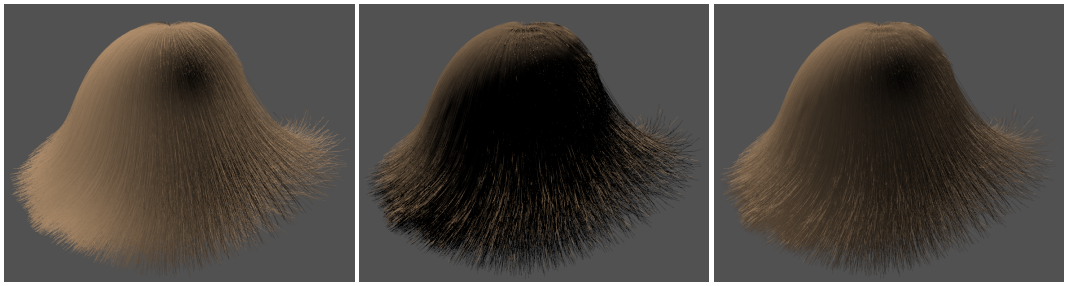
Figure 5.3: Adding a random offset to hair tangents (bottom)

The model by [KK89] produces reasonable lighting with little effort. However it is not energy conserving and does not look like real hair in many situations. Especially the diffuse term is problematic, because it usually results in very bright hair with some dark spots, which often looks unrealistic. An example is the first image in figure 5.4. [MJC+03] developed a more accurate model based on measurements that represents the state of the art in hair lighting. [ND05] demonstrated that the Marschner illumination model is also suitable for interactive applications. It it however much more difficult to setup and implement than the method by [KK89], because it requires to precompute lookup textures. For this project I just used the model by [KK89] and added a constant ambient term to the diffuse term to achieve a smoother look.

Another thing that I tried was to add some diffuse lighting based on the Phong illumination model. This requires a surface normal which is not available when rendering hair as line strips. Even if the hair is rendered using camera oriented triangle strips the normal will be inappropriate because it just points toward the camera. The correct surface normal must depend on the neighbor hairs. Conveniently, such a vector is already computed for the hair-hair interactions. The density gradient points into the direction of maximum change is computed for all particles. For particles at the hair surface, this should be the surface normal. The normalized density gradient is passed to the vertex shader as normal. In the fragment shader the interpolated normal is then used for Phong-style diffuse lighting. The new diffuse term is a combination of Phong diffuse and diffuse according to [KK89].

$$Diffuse = 0.6 \cdot dot(\mathbf{l}, \mathbf{n}) + 0.4 \cdot \sqrt{1 - dot(\mathbf{t}, \mathbf{l})^2}$$

This improves the visual quality of the lighting especially when shadows are not considered. Hair that faces away from the light source is now darker which is an improvement especially for bright hair. The biggest problem with this approach is that the density gradients computed with a voxel grid often show strange patterns as discussed in section 3. The patterns become visible in the diffuse lighting, which looks very unrealistic. The approach works best if high hair-hair repulsion forces are used and the particles are distributed evenly in the grid.
For fur it is also an option to compute the particle normal from the underlying surface. Figure 5.4 shows a comparison between diffuse lighting based Kajiya and Kay's model and Phong illumination. Both can be combined to achieve a good result. Altogether, using the density gradient as surface normal could be interesting when interactive framerates are the goal. With proper self shadowing in combination with the much more accurate illumination model by [MJC+03], the benefits of additional Phong diffuse lighting probably become less significant.



(a) Kajiya and Kay diffuse (b) Phong diffuse using density gradient (c) Combined diffuse

Figure 5.4: Diffuse comparison

[PHA05] also combines Phong diffuse lighting with Kajiya and Kay diffuse lighting, however they compute the surface normal differently to achieve a smooth look in all situations. They compute a hair isosurface from the density field (probably using the marching cubes algorithm). Subsequently, they compute a signed distance field by constructing a volumetric grid that stores signed distances from the surface. To compute the normal of a hair vertex they compute the gradient of the distance field at the particle's position. Constructing the isosurface and the distance field is time consuming. This is why the procedure is only executed once for the hair rest pose. In each frame, the generated normals are transformed according to the hair orientation.

6

Results and evaluation

6.1 Bunny with fur

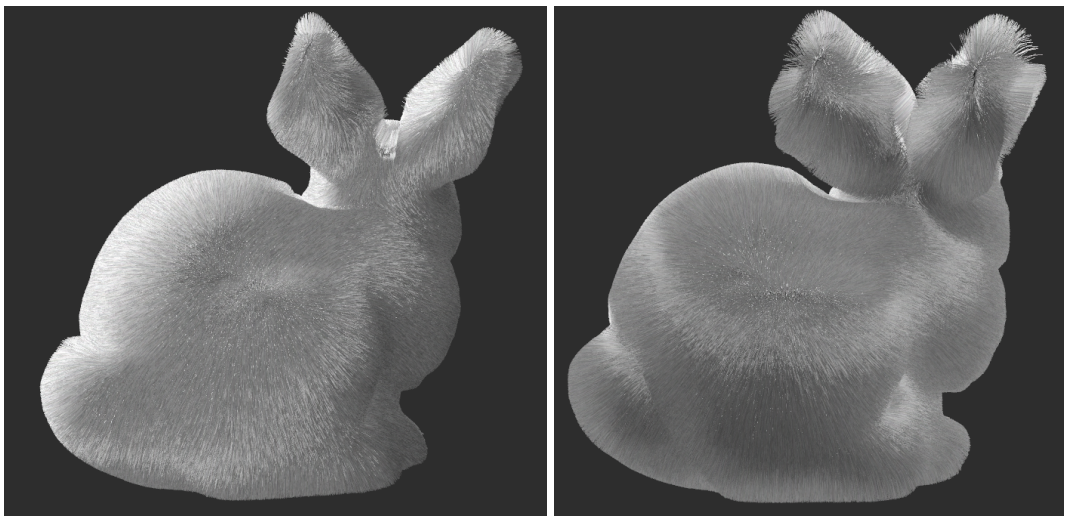


Figure 6.1: Furry Stanford Bunny

To test fur simulation and rendering, I used the famous bunny available in the Stanford 3D scanning repository with roughly 70000 triangles. The bunny is sampled with approximately 250000 strands of hair and each hair consists of 5 particles, so the total number of particles is around 1.25 million. The minimum distance between sampled particles is 0.0008.
Iterative PBD is used to simulate the individual strands of hair without any damping. With only 5 particles per hair the difference between FTL and iterative PBD should be not that huge performance wise because few PBD iterations are required to achieve an inextensible look. The number of iterations is adaptive so that the stretch is not more than 20 percent. For the density and velocity grid, a cell size of 0.03 is used. A small amount of hair friction is used ($s_{friction} = 0.05$), while $s_{repulsion}$ is set to 0.2. Collision with the bunny rigid is handled with hair repulsion. A limitation is that this does not work properly for fast velocities. When the bunny moves too fast disturbing artifacts become visible. As a simple fix, collision detection is performed for each particle against the plane given by the attached particle's position and surface normal. This is not perfect but it reduces the artifacts considerably.
The timestep is 0.01. With an Intel i5 quadcore processor with 3.1 Ghz and 8 GB Ram

it takes around 290 ms to simulate the strands of hair, including everything except for rendering. All computations except for the construction of the grid are performed with multi-threading using OpenMP. The majority of this time is spent on computing the hair-hair interactions. With roughly 50 ms it also takes a considerable amount of time to solve the distance constraints with iterative PBD.

Concerning rendering, the fur color is a very light gray with some subtle brightness variations. The method by Kajiya and Kay is used for specular lighting. Small random offsets in the range from -0.002 to 0.002 are added to the hair tangents which really improves specular lighting. For the diffuse component a mix of Kajiya and Kay's diffuse term and the Phong diffuse term is used. The normalized density gradient is used as surface normal. For the furry Stanford Bunny this works very well and improves the overall look. Also, some ambient lighting is added (factor 0.3).

Figure 6.1 shows the furry bunny and demonstrates different fur styles. In the second image the distance between hair particles is higher than in the first. Also, the amount of brightness variation is reduced resulting in a smoother look. It should also be mentioned that the light source and camera direction is not exactly the same in both images.

6.2 Long hair

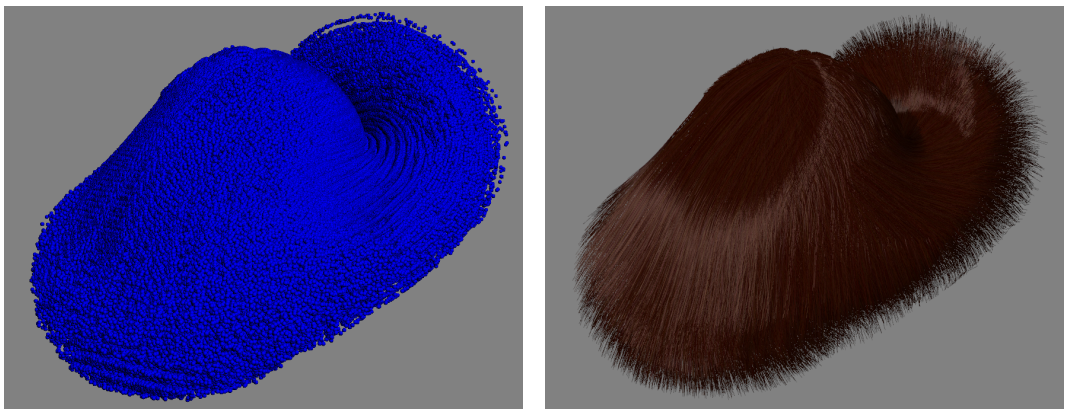


Figure 6.2: Moving Hair with 2.1 million particles

According to the german Wikipedia, a human has around 100000 strands of hair on the head. It differs for different hair colors, for red hair it is 90000 and for thin blond hair it can even be up to 150000. This does not seem very much compared to the 250000 strands of hair that were placed on the bunny, but much more particles per strand of hair are required to simulate long hair. A simple sphere is used for the head, which makes collision detection very easy. Roughly 90000 strands of hair are placed on the sphere. The haircut is hardcoded for this specific example. Only the top of the sphere is sampled and the lengths of the hairs are variable. The longest hairs consist of 35 particles. In total, approximately 2 million hair particles are used.
The strands of hair are simulated with dynamic follow-the-leader with a damping of 0.9. For the density and velocity grid a cell size of 0.0175 is used. The simulation is performed on the same system that was used for the fur simulation. It takes roughly 400-500 ms to simulate the strands of hair, not including rendering. The dynamic follow-the-leader computations only take 30-40 ms. [MKC12] simulates a similar scenario with 1.9 million particles at

8 frames per second, including rendering and collision detection against 8 ellipsoids that represent head and shoulders. However they run the simulation on a GTX 480 GPU.

Concerning rendering, it is sometimes problematic to use the density gradient as surface normal. It looks great most of the time, but sometimes patterns due to the underlying grid become visible that were discussed briefly in section 4.2.4. This is especially a problem for light hair colors such as blond. Without the Phong diffuse term however, blond hair looks too bright overall and even more unrealistic. For long blond hair, proper self-shadowing is required which I did not implement for this project.

6.3 Summary and conclusion

To place the strands of hair on the skin, I implemented an algorithm that generates Poisson disk point sets directly on triangle meshes.
The distance constraints for the individual strands of hair are solved with position based dynamics. Dynamic follow-the-leader is a viable option for hair rendering and is probably the best choice if interactive framerates are the goal. Iterative PBD is more flexible in terms of damping and provides a well defined mechanism to add other constraints by processing them inside the solver loop. Usually a few solver iterations are enough to achieve an inextensible look in most situations, but it can not compete against dynamic FTL that always only needs one iteration. Hair-hair interactions are simulated by constructing a voxel grid at each frame that stores densities and velocities. This grid is used to simulate hair friction and repulsion, which can be adjusted in order to get different hairstyles.
The strands of hair are rendered as line strips. The illumination model by Kajiya and Kay is used for lighting, which achieves good results in many situations despite its simplicity. If a surface normal is available the lighting can be improved by adding some diffuse lighting based on the Phong illumination system and reducing the diffuse Kajiya and Kay diffuse light contribution. This surface normal can be computed from the density grid. Using the density gradient as surface normal often leads to disturbing artifacts, however with careful calibrated parameters (most notably cell size) it may be an option for some applications.

Hair simulation is a huge topic in computer graphics and only a fraction of possible approaches could be discussed as part of this project. The implementation can simulate and render both fur and hair. It is however limited to smooth hair and can not handle curly hair for instance. Another limitation concerns interactions with rigids. If the interaction is handled with hair repulsion the hair may enter the rigid at high velocities. Checking only against primitive objects such as spheres and ellipsoids may be not accurate enough for some applications and it requires to setup the primitive geometry for each rigid by hand. An elegant but probably slower solution would be to compute hair-hair interactions with SPH and also handle rigid collisions with it using boundary particles.
Concerning rendering, the biggest limitation of my implementation is the absence of shadows and self-shadowing. Also, the illumination model by [MJC+03] should be implemented which is more realistic than the model by Kajiya and Kay.

Bibliography

[CJW+09] D. Cline, S. Jeschke, K. White, A. Razdan, and P. Wonka. Dart throwing on surfaces. In *Eurographics Symposium on Rendering*, 2009.

[HMT01] S. Hadap and N. Magnenat-Thalmann. Modeling dynamic hair as a continuum. In *Eurographics*, 2001.

[KK89] J. T. Kajiya and T. L. Kay. Rendering fur with three dimensional textures. In *SIGGRAPH*, 1989.

[MHHR06] M. Mueller, B. Heidelberger, M. Hennix, and J. Ratcliff. Position based dynamics. In *Workshop on Virtual Reality Interactions and Physical Simulation*, 2006.

[MJC+03] S. R. Marschner, H. W. Jensen, M. Cammarano, S. Worley, and P. Hanrahan. Light scattering from human hair fibers. In *SIGGRAPH*, 2003.

[MKC12] M. Mueller, T. Y. Kim, and N. Chentanez. Fast simulation of inextensible hair and fur. In *Workshop on Virtual Reality Interactions and Physical Simulation*, 2012.

[ND05] H. Nguyen and W. Donelly. *GPU Gems 2*, chapter 23. Nvidia Corporation, 2005. http://http.developer.nvidia.com/GPUGems2/gpugems2_chapter23.html.

[PHA05] L. Petrovic, M. Henne, and J. Anderson. Volumetric methods for simulation and rendering of hair. Technical report, Pixar, 2005.

[Tar08] S. Tariq. Real-time hair rendering on the gpu. In *SIGGRAPH*, 2008.

[WBK+07] K. Ward, F. Bertails, T.-Y. Kim, S. R. Marschner, Marie-Paule, C. Ming, and C. Lin. A survey on hair modeling: Styling, simulation, and rendering. Technical report, 2007.