Simulation of Curly Hair

5th Semester Project Laboratory Report

Barnabás Börcsök
borcsok.barnabas@simonyi.bme.hu

Dr. László Szécsi
Advisor

Figure 1. The achieved visual look

Abstract

A self-assessment of the 5th semester Project Laboratory Project is presented in this article. My goal was to achieve hair simulation of acceptable quality both in terms of look and performance. Multiple approaches were considered before arriving at a Position Based Dynamics based solution. The simulation was implemented in C++ with the Open Graphics Library (OpenGL [1]).

CCS Concepts: • **Computing methodologies** → *Physical simulation.*

Keywords: hair simulation, position based dynamics, OpenGL

[1] http://www.opengl.org

1 Introduction

In the 5th semester of their undergrad studies, BSc students from Budapest University of Tehchnology and Economics embark on their first journey of scientific research. I was always interested in and fascinated by computer graphics, it came naturally to choose a subject in this area. As I had little hands-on experience in this field, a long time had to be dedicated to research and trying out different simulation methods.

The first part of this paper reflects this, giving an overview of considered methods, and other possible routes that could have been taken to implement hair simulation.

The implementation and all of the code mentioned is available at http://git.sch.bme.hu/bobarna/brave-2.

2 Overview of considered methods

There were mainly three methods considered, two of them being substantially different:

2.1 Mass Spring System

The whole idea of doing hair simulation as my project laboratory came after reading Iben et al. [1]. They outline a method for the artistic simulation of curly hair for use in their film production. The method was featured in the 2012 movie Brave.

This approach models the hair as a chain of particles with given mass, each connected via springs. This results in a mass-spring system, which is then modeled by considering well-known physics formula's such as Newton's second law of motion $F = m * a$, and Hooke's Law, which states that the force F needed to extend or compress a spring by some distance x scales linearly with respect to that distance. The paper by Iben et al. [1] created an elaborate system of springs, with additional core springs making the simulation stable and resulting in the desired look and feel.

The main characteristic of this approach is that a Mass Spring System deals with forces, and tries to stay true to the laws of physics. It accounts for internal and external forces from which accelerations are computed based on Newton's second law of motion. A time integration method is then used to update the velocities and finally the positions of the particles. This will not be the case with Position Based Dynamics.

2.2 Position Based Dynamics (PBD)

The paper by Müller et al. [2] presents an approach that omits the velocity as well as the force layer of the then-present popular approaches for simulation methods of dynamic systems in computer graphics.

Position Based Dynamics (PBD) – as its name suggests – works with the position of particles. A big advantage of such a system lies in its controllability, and the easy reducability of overshooting problems present in force based systems. Another favorable aspect is that PBD methods are generally easier with the maths and somewhat easier to implement in terms of the needed mathematical and physical background needed to grasp its inner workings. In addition, – as it works directly with the position of particles – collision constraints can be handled easily and penetrations can be resolved completely by projecting points onto the penetrated surface. Although such measures can only be applied on the expanse of physical accuracy, position based dynamics being only a "good enough" approximation of how objects behave in real life.

Chapter 2 of the paper by Umenhoffer et al. [6] builds on the Müller et al. [2] paper and gives a concise overview of the position based dynamics method, and goes on to show the type of constraints that can be applied to control to behaviour of such a system – which is present in both papers.

2.3 Follow-the-Leader (FTL)

The Dynamic Follow-the-Leader (FTL) method outlined by Müller et al. [3] focuses on the fast simulation of hair and fur on animated characters. The sheer number of computation needed for simulating thousands of hair strands, each consisting of numerous particles presents a big challenge. Also, as each strand is inextensible, it is not trivial to come up with an algorithm that does its job in feasible time.

As it will be covered later in this article, the already introduced PBD method needs multiple iterations per frame in order to keep the system from stretching and becoming unstable. The FTL method by Müller et al. [3] presents a method that takes only a single iteration through the particles of each hair strand per frame to achieve the desired results.

As this sounds fascinating, and cuts the computation time needed substantially, in the early stages of my project, I implemented the FTL method alongside PBD. It *works*, and is fast, although due to the iteration count being one, when greater forces are being applied to the system, a substantial amount of stretching was introduced to the system. As it seemed in comparison that the "basic" PBD method yielded the same – or even more accurate – results, and it was feasible to allow myself to use multiple iterations per frame, I did not further investigate if there could have been improvements made to my implementation to eliminate the stretching in the presence of greater forces.

3 Position Based Simulation

3.1 Algorithm Overview

We model an S hair strand as a chain of N particles and a set of M constraints. Each particle $p \in [1, \ldots, N]$ has three attributes: mass (m_p), position ($\boldsymbol{x_p}$) and velocity ($\boldsymbol{v_p}$).

A constraint $c \in [1, \ldots, M]$ with cardinality n_c is a function $C_c : \mathbb{R}^{3n_c} \mapsto \mathbb{R}$. It operates on a set of indices $\{i_1, \ldots i_{n_c}\}$, $i_k \in [1, ..., N]$. The constraint function also has a stiffness parameter $k_c \in [0...1]$ and a type of either *equality* or *inequality*.

Constraint c with type *equality* is satisfied if $C_c(\boldsymbol{x}_{p_1}, \ldots, \boldsymbol{x}_{p_{n_j}}) = 0$. If its type is *inequality*, then it is satisfied if $C_c(\boldsymbol{x}_{p_1}, \ldots, \boldsymbol{x}_{p_{n_c}}) \geq 0$. The stiffness parameter k_c defines the strength of the constraint in a range from zero to one.

Given these notations, the algorithm works in the following way:

Lines `(1)-(3)` initialize the particles. We store the inverse masses w in order to be able to store infinitely heavy particles that are stationary during the simulation. ($\frac{1}{\infty} \approx 0$)

The core idea of position based dynamics is shown in lines `(9)-(11)`, `(15)-(17)` and `(18)-(21)`. In line `(10)`, estimated

Algorithm 1: pseudo code for the PBD simulation

```
(1)  foreach p in particles do
(2)  |  initialize p_x = x_p^0, p_v = v_p^0, p_w = 1/m_p
(3)  end
(4)  loop
(5)  |  foreach p in particles do
(6)  |  |  p_v ← p_v + Δt · p_w · f_external
(7)  |  end
(8)  |  dampVelocities(p_1, ..., p_N)
(9)  |  foreach p in particles do
(10) |  |  p_tmp ← p_x + Δt · p_v
(11) |  end
(12) |  foreach p in particles do
(13) |  |  generateCollisionConstraints(p_x →
     |  |    p_tmp)
(14) |  end
(15) |  loop numberOfIterations times
(16) |  |  projectConstraints(C_1, ..., C_{M+M_coll}, particles)
     |  |      /* only the temporary positions of particles are needed */
(17) |  end
(18) |  foreach p in particles do
(19) |  |  p_v ← (p_tmp − p_x)/Δt
(20) |  |  p_x ← p_tmp
(21) |  end
(22) |  velocityUpdate(particles)
     |      /* velocities of colliding particles are modified according to friction
     |  and restitution coefficients. */
(23) end
```

positions for each particle $\boldsymbol{p}$ are calculated using and explicit forward Euler integration step. In line (16) the iterative solver manipulates these temporary position estimates such that they satisfy the constraints. It does this by repeatedly projecting each constraint in an iterative manner to similar to that of a Gauss-Seidel [2] solver *(see section 3.2)*. Once the solver finishes with the iterations, in lines (19)-(20) each particle is moved to the calculated (once temporary) positions, and the velocity of each particle is updated accordingly.

Velocities are manipulated in lines (6), (8) and (19).

Line (6) allows to account for external forces in the simulation if some of the forces cannot be converted to positional constraints. *(For example the pulling force between subsequent particles are modeled as distance constraints instead of external forces, and so are the forces generated by the particles colliding with other objects.)* We use it to add gravity, and to generate wind effects for demonstration purposes. If only the gravitational force is present, then the line becomes $\boldsymbol{p_v} \leftarrow \boldsymbol{p_v} + \Delta t \cdot \boldsymbol{g}$, where $\boldsymbol{g}$ is the gravitational acceleration.

[2] https://en.wikipedia.org/wiki/Gauss%E2%80%93Seidel_method

In line (8) the velocities can be damped, if necessary. *Section 3.5* of the Müller et al. [2] paper gives some more sophisticated methods for damping, although the paper also points out that any form of damping can be used. In our case, simply applying a k_s damping coefficient with value between 98% to 99.9% seemed to be a good enough solution.

The initial constraints $C_1, \ldots, C_M$ are fixed throughout the simulation. In addition to these initial constraints, line (13) generates additional M_{coll} collision constraints. These change from time step to time step. The projection step in line (16) considers both the fixed and the collision constraints. Our implementation does not yet utilize this opportunity to dynamically generate collision constraints. This is one of the possible future works outlined in the subsection Hair-hair Collisions.

3.2 The Solver

The input to the solver are $M + M_{coll}$ constraints and the temporary ($\boldsymbol{p_{tmp}}$) positions of the particles. The aim of the solver is to modify the estimates such that they satisfy all the constraints. This yields a system of equations. For solving these for particle positions we use a sequence of Gauss-Seidel-type iterations. The original Gauss-Seidel algorithm can only handle linear systems, although we have a non-linear system of equations of hand. Even a simple distance constraint $C(\boldsymbol{p_1}, \boldsymbol{p_2}) = |\boldsymbol{p_1} - \boldsymbol{p_2}| - d$ yields a non-linear equation. In addition, the constraints of type *inequality* yield inequalities.

The part we borrow from the Gauss-Seidel method is the idea of solving each constraint independently one after the other. This makes up the essence of the method: we repeatedly iterate through all the constraints and project the particles to valid locations with respect to the given constraint alone. Modifications to point locations immediately get visible to the process. This speeds up convergence significantly, because pressure waves can propagate throughout the chain of particles in a single solver step, and effect which is dependent on the order in which constraints are solved.

In our hair simulation case, we solve the distance constraints by putting the hair particles at the desired distance from each other starting at the scalp, and moving outwards from there.

For projecting the constraints, we utilize the method proposed by Müller et al. [2]. The details of computing the desired delta positions are explained in the Appendix A Constraint Projection. An example derivation is given in Section 3.3 Distance Constraint.

There are several ways of incorporating the stiffness parameter. The simplest – and the way we chose to utilize – is simply multiplying the $\Delta \boldsymbol{p}$ correction by the $k \in [0 \ldots 1]$ stiffness parameter. For multiple iteration loops of the solver,

the effect of k becomes non-linear. The remaining error after n_s solver iterations is $\Delta \boldsymbol{p}(1-k)^{n_s}$. Müller et al. [2] propose to multiply the corrections not by k directly, but by $k' = 1-(1-k)^{1/n_s}$. With this transformation, the error becomes $\Delta \boldsymbol{p}(1-k')^{n_s} = \Delta \boldsymbol{p}(1-k)$, and thus linearly dependent on k and independent of n_s. The resulting stiffness is still dependent on the time step of the simulation. Real time environments typically use fixed time steps in which case this dependency is not problematic.

3.3 Distance Constraint

One of the most basic constraint is the Distance Constraint, forcing two given particles d distance apart. It is an *equality* type constraint, meaning that it is satisfied if $C_{distance} = 0$.

$$C_{distance}(\boldsymbol{p_1}, \boldsymbol{p_2}) = |\boldsymbol{p_1} - \boldsymbol{p_2}| - d \tag{1}$$

We are looking for $\Delta \boldsymbol{p_1}$ and $\Delta \boldsymbol{p_2}$ corrections such that

$$C_{distance}(\boldsymbol{p_1} + \Delta \boldsymbol{p_1}, \boldsymbol{p_2} + \Delta \boldsymbol{p_2}) = 0.$$

In finding the desired delta values to satisfy the constraint, we follow the method proposed by Müller et al. [2] *(see Appendix Constraint Projection)*.

The derivatives with respect to the positions are

$$\nabla_{p_1} C_{distance}(\boldsymbol{p_1}, \boldsymbol{p_2}) = \boldsymbol{n}$$

$$\nabla_{p_2} C_{distance}(\boldsymbol{p_1}, \boldsymbol{p_2}) = -\boldsymbol{n}$$

$$\boldsymbol{n} = \frac{\boldsymbol{p_1} - \boldsymbol{p_2}}{|\boldsymbol{p_1} - \boldsymbol{p_2}|}.$$

This makes the scaling factor

$$s = \frac{C(\boldsymbol{p_1}, \boldsymbol{p_2})}{w_1 |\nabla_{p_1} C(\boldsymbol{p_1}, \boldsymbol{p_2})|^2 + w_2 |\nabla_{p_2} C(\boldsymbol{p_1}, \boldsymbol{p_2})|^2}$$

$$= \frac{|\boldsymbol{p_1} - \boldsymbol{p_2}| - d}{w_1 + w_2}.$$

The final corrections are:

$$\Delta \boldsymbol{p_1} = -\frac{w_1}{w_1 + w_2} (|\boldsymbol{p_1} - \boldsymbol{p_2}| - d) \frac{\boldsymbol{p_1} - \boldsymbol{p_2}}{|\boldsymbol{p_1} - \boldsymbol{p_2}|}$$

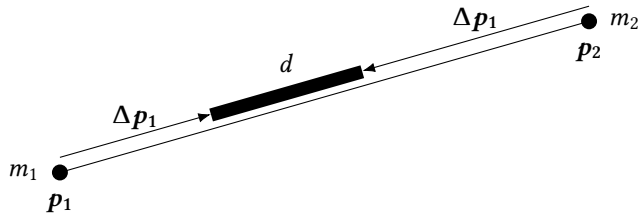
$$\Delta \boldsymbol{p_2} = +\frac{w_2}{w_1 + w_2} (|\boldsymbol{p_1} - \boldsymbol{p_2}| - d) \frac{\boldsymbol{p_1} - \boldsymbol{p_2}}{|\boldsymbol{p_1} - \boldsymbol{p_2}|}$$

Since the type of the Distance Constraint is *equality*, the projection is straight forward: we move both by $\boldsymbol{p_1}$ and $\boldsymbol{p_2}$ by $\Delta \boldsymbol{p_1}$ and $\Delta \boldsymbol{p_2}$ respectively.

3.4 Bending Constraint

Umenhoffer et al. [6] use the PBD technique to simulate cloth materials. They utilize the bending constraint to define the bending resistance of the material, giving it an extra stiffness.

Figure 2. Projection of the distance constraint $C(\boldsymbol{p_1}, \boldsymbol{p_2}) = |\boldsymbol{p_1} - \boldsymbol{p_2}| - d$. The corrections $\Delta \boldsymbol{p_i}$ are weighted according to the inverse masses $w_i = 1/m_i$.



Their goal with the Bending Constraint is to keep an initial angle between adjacent triangles formed by particles. They offer the $C(p_1, p_2, p_3, p_4) = arccos(n_1 \cdot n_2) - \phi_0$ constraint, where n_1 and n_2 are the normal vectors of the triangles, and $p_2 - p_1$ forms the common side of the two triangles. The ϕ_0 angle is measured between the normals of the two triangles.

At first, we thought about implementing something similar for our project, but then simplified the "Bending Constraint" for our use-case to essentially distance constraints $C_{distance}(p_{i-1}, p_{i+i})\ for\ i \in [1 \dots n_{particles} - 1]$ of the same hair strand with the d distance paramter of the being lower than the d parameter of other $C_distance$ constraints keeping the particles a given l_{seg} distance apart from each other, and assigning a lower $k_{bending}$ stiffness parameter.

This idea could be further developed when considering other (p_j, p_k) pairs and different distances between them. Figure 3 illustrates some example variety in the behaviour of chain of particles under different bending constraints.

3.5 Collision Constraint

One huge advantage of the position based approach is how simply collision handling can be realized. In each simulation step M_{coll} collision constraints are generated in the `(12)-(14)` lines of Algorithm 1. For a moving particle $\boldsymbol{p}$ and points $\boldsymbol{q_1}, \boldsymbol{q_2}, \boldsymbol{q_3}$ forming a triangle with normal $\boldsymbol{n}$, the collision constraint is

$$C_{collision}(\boldsymbol{p}, \boldsymbol{q_1}, \boldsymbol{q_2}, \boldsymbol{q_3}) = (\boldsymbol{p} - \boldsymbol{q_1}) \cdot \boldsymbol{n} \geq 0$$

, where we check whether $\boldsymbol{p}$ lies on the desired side of the given surface. We require that the magnitude of the vector from an arbitrary point on the surface to $\boldsymbol{p}$ projected onto the normal vector be greater than 0 by utilizing the dot product operation.

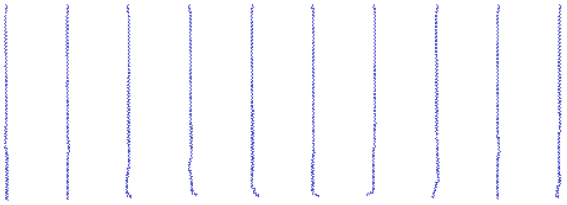
The formula for solving the triangle collision constraint is:

$$\Delta \boldsymbol{p_{tmp}} = -((\boldsymbol{p_{tmp}} - \boldsymbol{q_1}) \cdot \boldsymbol{n}) \cdot \boldsymbol{n}$$

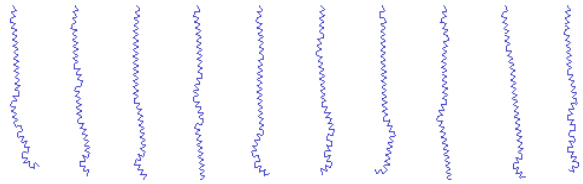
This formula moves the particle along the triangle normal exactly to the triangle's surface.

Figure 3. Different distance and particle pairing definitions yield significantly different results. Particle masses were slightly randomized for a variance between strands. In each below simulation, nr_{seg} particles of length l_{seg} were generated. Distance constraints acting as bending constraints had a stiffness parameter k. Distance constraints responsible for keeping the particles l_{seg} distance apart always had a $k_{pos} = 1$ stiffness parameter.

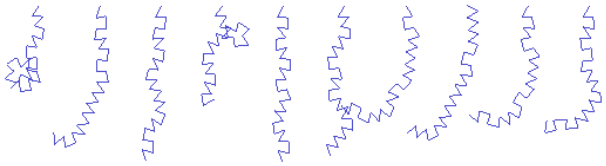
(a) $nr_{seg} = 120$, $l_{seg} = 0.005$, $k = 0.6$



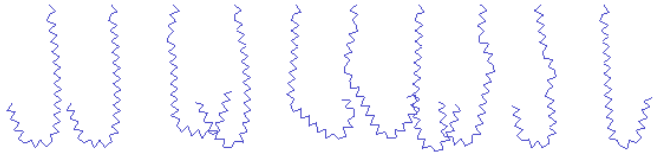
(b) $nr_{seg} = 60$, $l_{seg} = 0.01$, $k = 0.6$



(c) $nr_{seg} = 30$, $l_{seg} = 0.02$, $k = 0.6$



(d) A distance constraint between the first and last particles were given. $nr_{seg} = 40$, $l_{seg} = 0.015$, $k = 0.8$



3.6 Position Constraint

Maybe the most straight-forward constraint is the Position Constraint, given as:

$$C_{collision}(\boldsymbol{p}) = |\boldsymbol{p} - \boldsymbol{p}_{goal}| = 0$$
$$\Delta\boldsymbol{p} = \boldsymbol{p}_{goal} - \boldsymbol{p}$$

Moving the $\boldsymbol{p}$ particle to the $\boldsymbol{p}_{goal}$ position.

We utilize the Position Constraint for keeping the first particle in each strand on the surface of the head.

4 Implementation

After introducing the general position based dynamics simulation method, we dive into the specifics we settled on for our hair simulation.

The code is available at http://git.sch.bme.hu/bobarna/brave-2.

4.1 Data Structures

The properties of particles introduced in chapter 3.1 Algorithm Overview got an additional color property. This decision was made so hair with a gradient color could be simulated.

```
class Particle {
    vec3 pos;
    vec3 tmp;
    float w;
    vec3 v;
    vec3 color;
}
```

The strands are stored as a vector of vector of Particles. It is practical to keep related data packed right after each other in the memory for improving the performance of operations and function calls.

4.2 Constraints

A `Constraint` class with an abstract `solve()` method was implemented, storing each derived constraint class *(e.g. PositionConstraint, BendingConstraint)* in a heterogeneous collection and calling each overriden `solve()` method when projecting the constraints in the iteration loop (line `(16)` of Algorithm 1).

This implementation proved to be not fast enough, lagging even with low particle counts. We could not discover the cause of this increased computational time, and decided to settle on another solution. Instead of the above mentioned, we store only the parameters that constraints operate on, and the simulation calls the relevant solve functions with these stored parameters.

4.3 Putting the Hair in the Hair Simulation

The *PBDSimulation* class handles the simulation and rendering of the hair. It takes a *HeadObject* and propagates hairstrands starting from its surface. Each strands first particle is fixed in place with a *PositionConstraint*, not letting it move away from the scalp.

The geometry of the *HeadObject* is read from an `OBJ` file. (*See the Appendix B.1 OBJ Reader.*)

The hair strands are propagated throughout the surface of the *HeadObject* by proposing random $(u, v) \in [0 \dots 1] \times [0 \dots 1]$ coordinates and settling on them if they satisfy requirements for the position of the hair – like the corresponding face in world coordinates has an upward pointing normal vector. This propagation of strands on the surface of the head could be improved, as pointed out in sections More and better customization and Sequence for randomizing hair positions.

5 Results

The implementation is available at git.sch.bme.hu/bobarna/brave-2. The program is able to handle up to around 500 individual hair strands, each consisting of around 30 particles in real time. With the implemented OBJ reader, it is possible to achieve diverse visual results in no time.

An example result shown in Figure 1, and can be seen in the supplementary videos available here.[3] An 8-minute video was also made to supplement this writing. It is accessible at *TODO link to 8-minute video*

6 Conclusion

After investigating and trying out multiple methods to simulate hair, we arrived at the Position Based Dynamics (PBD) method. An overview of the PBD method, and an introduction to the constraint types used in our implementation was given. An implementation was made in C++ and OpenGL, which easily simulates the hair strands in real time, although the limit of the real-time method can easily be surpassed when adding around a thousand strands each with more than 30 particles.

7 Future Work

As this was part only of a semester-long Project Laboratory course, a great deal of time was dedicated in the early stages of the project to investigate and explore the subject area. This left only a sub-optimal time for implementing all desired aspects of the simulation.

7.1 Hair-hair Collisions

As it is computationally expensive to simulate each particle colliding with all of the other particles, a particle density field could be used. One such method of grouping hair particles and their corresponding velocities into a 3D voxel grid is outlined in the paper by Petrovic et al. [4], which was also utilized in the Müller et al. [3] paper for handling hair-hair collisions. Such a method could be used to dynamically generate collision constraints in line (13) of Algorithm 1.

[3] https://drive.google.com/drive/folders/1n79xSokrBe2lKQixX_DacF7wByXkbpZP?usp=sharing

7.2 Bending Constraint Improvements

As already mentioned in Section 3.4 Bending Constraint. A huge improvement could be made by improving the bending constraint either by further exploring the possibilities of defining different distance constraints between pairs of particles.

An alternative route could be to explore the possibility of a different type of bending constraint, for example by keeping a given angle between chunks of subsequent particles.

7.3 More and better customization

A drawable texture map is considered for the ability to tell the simulation where to put the hair strands on the surface of the head.

The `OBJ` reader could also be further developed into reading material, texture and/or color data for the objects.

A common interface for importing and exporting different hairstyles could be developed.

7.4 Better customizability

In the current implementation there is no way to properly customize the look and feel of the hair without accessing the source code. A possible improvement could be to read material and texture data for the hair from the OBJ file. Another huge improvement could be to set the constraints in real time, giving the user much better control of the hair. Other properties, such as length of the hair segments and color of the hair could be set as well.

7.5 Sequence for randomizing hair positions

In the current implementation, a standard C++ random number generator is utilized. Utilizing a low-discrepancy number sequence such as the Halton sequence for randomizing hair strand positions would be an improvement.

7.6 Rendering

This semester left no time to try out different methods for rendering and visualizing the hair. The current implementation connects the individual particles with GL_LINES, resulting in a sub-optimal visual experience. Offline (e.g. ray-tracing) methods could utilize the current simulation method. The paper by Rapp [5] discusses the real-time aspect of hair rendering in great detail, supplemented with a broad discussion on different parts of the (OpenGL) rendering pipeline that could be utilized.

7.7 Moving to the GPU

The current implementation runs solely on the CPU. A big leap forward would be the utilization of the GPU. Compute shaders for computations are considered. The paper

by Umenhoffer et al. [6] discusses a parallelization of the different constraint solves, although as they work on the same dataset, some computation problems are present during the parellelization of the problem.

Tesselation and geometry shaders could be utilized for the rendering and display of the particles, resulting in a far better image with the same simulation method and properties.

Acknowledgments

To Dr. László Szécsi, associate professor at the Computer Graphics Group (Department of Control Engineering and Information Technology), whom I had the chance to consult with during the semester.

References

[1] Hayley Iben, Mark Meyer, Lena Petrovic, Olivier Soares, John Anderson, and Andrew Witkin. 2013. Artistic Simulation of Curly Hair. In *Proceedings of the 12th ACM SIGGRAPH/Eurographics Symposium on Computer Animation* (Anaheim, California) *(SCA '13)*. Association for Computing Machinery, New York, NY, USA, 63–71. https://doi.org/10.1145/2485895.2485913 http://graphics.pixar.com/library/CurlyHairB/paper.pdf.

[2] Matthias Müller, Bruno Heidelberger, Marcus Hennix, and John Ratcliff. 2007. Position Based Dynamics. *J. Vis. Comun. Image Represent.* 18, 2 (April 2007), 109–118. https://doi.org/10.1016/j.jvcir.2007.01.005

[3] Matthias Müller, Tae Kim, and Nuttapong Chentanez. 2012. Fast Simulation of Inextensible Hair and Fur. *VRIPHYS 2012 - 9th Workshop on Virtual Reality Interactions and Physical Simulations.* https://doi.org/10.2312/PE/vriphys/vriphys12/039-044

[4] L. Petrovic, M. Henne, and John Anderson Pixar. 2006. Volumetric Methods for Simulation and Rendering of Hair. https://graphics.pixar.com/library/Hair/paper.pdf

[5] M. Rapp. 2014. Real-Time Hair Rendering. http://markusrapp.de/wordpress/wp-content/uploads/hair/MarkusRapp-MasterThesis-RealTimeHairRendering.pdf

[6] Tamás Umenhoffer, Artúr M. Marschal, and Péter Suti. 2016. Simulation methods for elastic and fluid materials. http://cg.iit.bme.hu/~umitomi/publications/GRAFGEO2016_PhysicsSimulators.pdf.

A Constraint Projection

asd

B Supplementary development

B.1 OBJ Reader

An OBJ reader utility was implemented as part of the project. As the OBJ file format [4] describes a wide range of properties for objects, our OBJ reader handles only the subset of these description options – those that are needed for the project. Unsupported line types were simply ignored, still resulting in a successful file read if possible.

- "`v x y z`" defines a vertex with position (x, y, z).[5]

Table 1. Supported OBJ Data Types

Type of data	Format
Comment	`# comment`
Geometric vertex data	`v x y z`
Texture coordinates	`vt u v`
Vertex normals	`vn i j k`
Triangular faces	`f v1/vt1/vn1 v2/vt2/vn2 ... v3/vt3/vn3`

- "`vt u v`" defines a 2D texture coordinate with position (u, v).[6]
- "`vn i j k`" specifies a normal vector with components i, j and k.
- "`f v1/vt1/vn1 v2/vt2/vn2 v3/vt3/vn3`" Specifies a face with the given indices. For the first vertex, the *v1*st previously defined vertex position is used, the *vt1*st texture coordinate and *vn1*st normal vector is used.[7]

B.2 Recording the simulation on-the-fly

A recording utility was made to work in tandem with the application. In essence, the rendered images are written into `bmp` or `png` files if the capturing is enabled. In the implementation[8], toggling the capturing is mapped to the `C` key.

For handling the export of the rendered image into files, the stb_image_write.h[9] single-file public domain library was used.

The program outputs the image files in the `renders` folder at 24 FPS, with the naming convention renderXXXX[.bmp/.png], where XXXX is the number of the frame with left-padded zeroes.

After the image files are written, the user can start the `make_video.sh` script to assemble the `output.mp4` and delete all the rendered frames. This script uses the free and open-source ffmpeg command line utility[10]

[4] http://paulbourke.net/dataformats/obj/

[5] The obj file format allows for a w coordinate (also called the weight) for describing rational curves and surfaces. The default value of *w* is 1.0. My implementation handles only the assignment of *x*, *y* and *z* values.

[6] The obj file format allows for 3D texture coordinates as well, but this implementation handles only 2D texture coordinates.

[7] The obj file format allows for face definitions far beyond only triangles, although this implementation handles only face definitions of the above format.

[8] https://git.sch.bme.hu/bobarna/brave-2

[9] https://github.com/nothings/stb/blob/master/stb_image_write.h

[10] https://ffmpeg.org/