Simulation of Curly Hair

5th semester Project Laboratory report

Barnabás Börcsök
borcsok.barnabas@simonyi.bme.hu
Budapest University of Technology and Economics

Dr. Szécsi László
Advisor
Budapest University of Technology and Economics
Computer Graphics Group
Department of Control Engineering and Information Technology

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]).

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

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

Keywords: hair simulation, position based dynamics, OpenGL

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 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 avaiable 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 modelled 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 in 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 Simulation of a Single Hair Strand

3.1 A Particle

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

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

Constraint c with type *equality* is satisfied if $C_c(\boldsymbol{x}_{p_1}, \dots, \boldsymbol{x}_{p_{n_j}}) = 0$. If its type is *inequality*, then it is satisfied if $C_c(\boldsymbol{x}_{p_1}, \dots, \boldsymbol{x}_{p_{n_c}}) \geq 0$. 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:

TODO decide about syntax: array of particles (OOP) or array of v, x and m?

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
```

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 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 a Gauss-Seidel type fashion *TODO See section ....* Once the solver finishes with the iteartions, 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 modelled 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.

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.

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

4 Putting Hair on the Head

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. The

An example result shown in figure *TODO include figure*, and can be seen in the supplementary videos *TODO Google Drive (?) link with the videos*.

An 8-minute-video was 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 nature 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 extensive 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 Petrovic et al. [4] paper, which was also utilized in the Müller et al. [3] paper for handling hair-hair collisions.

7.2 Refinement and proper customizability of the propagation of hair strands on the head object.

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.

7.3 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.

7.4 Halton sequence for randomizing hair positions

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

7.5 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.

7.6 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 parametrization 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 Supplementary development

A.1 OBJ Reader

An OBJ reader utility was implemented as part of the project. As the OBJ file format [2] describes a wide range of properties for objects. The present OBJ reader handles only the subset of these describtion options needed for the project. Lines containing other type of object descriptions were ignored, resulting in a successful file read if possible.

- "`v x y z`" defines a vertex with position (x, y, z).[3]

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).[4]
- "`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.[5]

A.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 on. In the implementation[6], toggling the capturing is mapped to the `C` key.

For handling the export of the rendered image into files, the stb_image_write.h[7] 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-sorce ffmpeg command line utility[8]

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

[3] 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.

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

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

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

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

[8] https://ffmpeg.org/