Final Project: Neural Hair Simulation

Animation is one of the main task in computer graphics, aside from rendering and modelling. Animating every minute detail to be physically plausible or accurate would be impossible without simulation. Simulation aims to recreate real world phenomena at reasonable speeds and, in the case of artistic animation, with physically believable motion. One such phenomenon is the behavior of hair on top of a head/scalp in motion. This can often be very computationally demanding, and therefore quite slow depending on the complexity of the model. In the production of computer animated films, animators often work with course representations of the final motion for simulated parts because they can't be fully simulated and rendered in real time. This often results in a disconnect between the artist's intentions and the end result. Animators may need to wait for a rough render of a scene before they're able to get a good look at how the simulation turned out. This slows down production and can make the workflow of an artist quite awkward. My final project hopes to investigate both how hair can be simulated. And to try and see if the simulation can be learned by a neural network to hopefully speed things up.

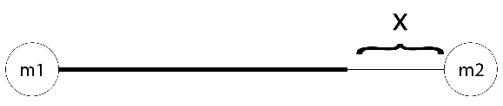
Initial Plans and Interest

The main reason I chose this area to pursue is my interest in animation and film. In computer graphics a lot of the problems surrounding modelling, rendering and animation by now have pretty great solutions. With enough time you're able to get an image or sequence that is near indistinguishable from reality. The main issues in the animation industry, apart from fine-tuning and enhancing some of those solutions, is increasing performance and sometimes finding alternative methods that work faster than older algorithms. Today, for example, Pixar films typically take about a couple of days to render just a single frame at full resolution. Animators and Artists can't possibly wait for full-renders while working on a project, so heavily-simplified versions of scenes are used in-software that can be rendered in real time. For simulations, this often means leaving out the simulation entirely, or only including a very course simulation for the artist to work with. To actually see the final motion of hair, artist will usually have to run a separate render, to see the fine detail.

My initial plan was to take the pipeline for hair simulation in film and try and cut down the time needed at a particular step. Inspired by the work of Zahng et al. (2021)[1] "Dynamic Neural Garments", which tackled a similar problem for cloth simulation, I planned to use a neural network to generate the finished render from course simulation data. I then changed this to just focus on the simulation itself, since I was limited to implementing my own hair simulator.

Final Solution and Implementation

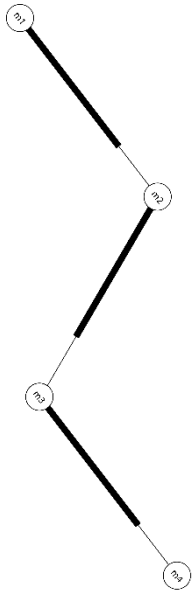
The first step was to implement a hair simulator to collect data. A simple model for a hair is a mass-spring system. At its core, we have two masses connected by an extensible spring of some fixed starting length. At any stage, the force that the spring exerts on both masses towards its center (-) is given by how far it is stretched (extension) multiplied by a spring constant k (sometimes called the *stiffness* of the spring). The diagram below illustrates the model:



$$F = -kx$$

fig1

The force described in fig1 is in the direction m1->m2 and it is felt my m2. That to say that m2 is pulled towards m1. m1 experiences an equal force in the opposite direction.



The plan to go from this model to hair, would simply be to have each hair modelled as a chain of springs and masses all obeying this law (fig2). These chains would be arranged across an invisible hemisphere representing a head. To connect the chains to this imaginary scalp, the first mass of each hair will be connected with a spring that has an initial length of zero, whose other end is connected to a point that moves in relation to the centre of the head as instead of another mass.

fig2

The simulator is written in C++. As a starting point I used the template code of an assignment (*Assignment 1* found here: https://www.cs.cmu.edu/~scoros/cs15467-s16/ [2]). This code (available in its original form in the attached zip) consisted of a GUI, and a few classes to aid the mass spring system, including a *ParticleSystem*, which contained placeholders for integration methods, OpenGL draw methods, stacked vectors for position, force and velocity and mass (i.e [x1,y1,z1,x2,y2,z2 …xn,yn,zn]), and two kind of springs (one with a fixed end and one mass and another with two masses attached). The masses were usually just referenced by their index so a few lines of code could extract the values from one of the stacked vectors for an attribute of a specific mass. There was also a system stored in *ParticleSystemLoader* for loading in vertex and spring data.

Missing from the assignment code, and therefore my task to implement, was:

- Force calculators for both *Spring* and *ZeroLengthSpring* as well as gravity on the particles
- Integration methods, which essentially updated the position of each particle (through the stacked vector *positions* for the draw method), including any code that updated the velocities, forces, or accelerations.
- Implementation of an invisible “head”: an invisible sphere around a center defined in *ParticleSystem*, which has its own velocity and radius. I had to slightly augment *ParticleSystemLoader* to read this information from the *.mss* files that I generated.
- Collisions with the head
- System to take in a batch of input conditions and run them all in sequence and a format for storing the results of the simulation (the position of each particle and the centre of the head at each frame)
- A few UI additions and hotkeys to make running the program easier
- A generator script (python) for the vertices and spring connections around the head as well as all variables/initial conditions to be read by the simulator.
- A script (python) to read the results output by the simulator into numpy data to be used to train a model.

The core of the simulator is the class *ParticleSystem* whose integrator is run once per frame. This contains several attribute vectors, the lists of springs and methods for handling collisions. etc. I used both Forward Euler (press F) integration and Symplectic Euler (default or press S) integration. (My implementation of Backwards Euler was far too slow to collect all the data in a reasonable time, but

is available in the program by pressing B and has damping sort of built in as it collapses to zero energy eventually.

Below are the integrators for both FE and SE integration:

```
// Integrate one time step with Forward Euler.
void ParticleSystem::integrate_FE(double delta) {

	// TODO (Part 1): Implement forward Euler.
	//enoch's code

	//p2 = p1+ vdt
	positions += velocities * delta;
	moveHead(delta);
	dVector a = computeAccelerations(positions, velocities, masses);
	//v=u+at
	velocities += a * delta;
	manageCollisions();

}

// Integrate one time step with Symplectic Euler.
void ParticleSystem::integrate_SE(double delta) {
//@enoch
	dVector force = computeForceVector(positions, velocities);

	// TODO (Part 2): Implement symplectic Euler.

	for (int i = 0; i < positions.size(); i++)
	{
		velocities[i] = velocities[i] + (force[i] / masses[i / 3]) * delta;
		positions[i] = positions[i] + (velocities[i] * delta);
	}
	moveHead(delta);
	manageCollisions();

}
```

FE simply takes the current velocities and uses them to update the positions to get the next positions, then updates the accelerations (by calculating the forces created by each spring) then updates the velocities ready for the next frame. SE just looks to the future time step to use that velocity as a better approximation by computing the new velocities first before updating the positions with them. *computeAccelerations* uses newtons second law F = ma. F is calculated by the function *computeForceVector*, which adds gravity to each particle as well as asking each Spring and *ZeroLengthSpring* in *ParticleSystem* to add spring forces to each particle based on hooks law demonstrated in fig1. Here is the force calculator:

```
dVector ParticleSystem::computeForceVector(const dVector& x, const dVector& v) {
	// The force vector has the same length as the position vector.for abou
	dVector forceVector = dVector::Zero(x.size());
	recordPositions();

	// TODO (Part 2): Accumulate the forces from all springs in the system.
	// Note that there are two separate lists of springs:
	// springs and zeroLengthSprings. Don't forget to add up the forces from
both.

	//add zero length springs to vector
	for (list<ZeroLengthSpring>::iterator i = zeroLengthSprings.begin(); i !=
zeroLengthSprings.end(); ++i)
```

```
	{
		i->addForcesToVector(x, forceVector);
	}

	//add normal springs to vector
	for (int i = 0; i < springs.size(); i++)
	{
		springs[i].addForcesToVector(x, forceVector);
	}



	if (enableGravity) {
		// TODO (Part 1): Implement gravity.
		for (int i = 1; i < x.size(); i += 3) {
			forceVector[i] += G;
		}

	}



	// NOTE: Velocity currently unused because we aren't modeling drag forces.
	return forceVector;
}
```

The methods addForcesToVector takes the positions vector *x* and lets each spring read it to figure out how far apart its endpoints are to calculate the spring force it should add to each one. At this step I also made sure to record the current positions of everything (*center* and *particles*) to be written to a results file later.

Collisions are handled by the method manageCollisions() called at the end of each integration step. Since the only thing to collide with is the invisible head, modelled as a sphere about a given center, this is a fairly straightforward process. For each particle, I perform a test to determine whether the particle is inside of the radius (x-center < radius), if so then I compute the normal vector to the sphere at the point on the surface nearest to the particle. I then just reposition the particle at that point on the surface and calculate its new speed by reflecting its current speed in the normal vector and having the sphere "absorb" some of its energy so that the reflected speed is less than the incoming speed to reduce bounciness. In fig 3, the the offending particle is in pink and its updated position and velocity are in green. Below is the code for this implementation.

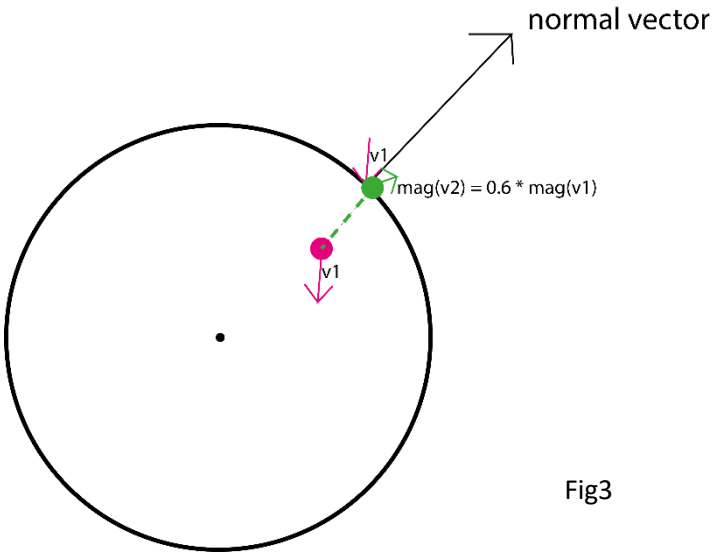


Fig3

```
//get hair to collide with the head-sphere
void ParticleSystem::manageCollisions() {
	if (enableCollision) {
		//go through particles
	int n = numParticles;
	auto c = getCenter();
	for (int i = 0; i < n; i++) {
```

```
            //get x and check to see whether or not it's inside the sphere
            V3D x = V3D(positions[3 * i], positions[3 * i + 1], positions[3 * i
+ 2]) - c;
            double dist = x.norm();
            if (dist < radius) {
                  //particle is in the sphere, so im gonna push it back to the
surface
                  x.normalize();
                  x *= radius; //basically have it be at the same anggle, kinda
just pushed backwards if loooking at the center
                  V3D new_pos = c + x;
                  setPosition(i, P3D(new_pos));

                  //now to figure out its new velocity. first, i need the
current velocity
                  V3D v = getVelocity(i);
                  V3D v_2 = reflectVector(v, x.normalized());
                  //the same momentum atm, but i think it will be better w some
damping, how about v = 0.6 original v
                  v_2 *= 0.6;
                  setVelocity(i, v_2);
            }
      }
      }


}
```

Note. Collisions are turned on in the program by pressing C, and gravity can be turned on with G. although there are buttons for each on the left hand side of the GUI.

There is a bit more to the simulator and accompanying UI such as the attachment of some springs to the head and the updating of the *center* variable which is handled by the *moveHead()* method. My code is pretty well commented so further details can be found either on the GitHub link, or the attached files.

Now that the simulator works, a method to actually generate these particles' and springs' initial positions would need to be written. I used python for this. The file generate_hairs.py (in the /python stuff folder), which is to be manipulated manually from inside an IDE, is responsible for coming up with a distribution of points around the invisible sphere about a given centre based on a radius and the density of hairs.

Here is the code for generating vertices:

```
def newHair(root,direction, step, length):
    points = []

    norm = np.linalg.norm(direction)
    if norm:
        d = direction / norm
    else:
        d = np.array([1,0,0])
    for i in range(length):
        points.append(root + i* step * d)
    return(points)
```

```
def generateRoots(r = 1, n = 6, mode = 0):
    """
    generates a pattern that should lie on the surphace of a sphere centered
at 0 with radius r.
    n determines the density.
    """
    roots = []
    if(mode == 0):
        for i in range(1, n):
            radius = i * r/ (n)
            theta = 0
            while theta <= 2*math.pi:
                roots.append(np.array([radius* math.sin(theta), math.sqrt(r **
2 - radius ** 2), radius * math.cos(theta)]))
                theta += 2 * math.pi / n
    elif(mode == 1):
        step = 4/n
        a = 1/n
        theta = 0
        rad = a * theta ** 0.7
        while rad < 0.95*r:
            x = rad * math.cos(theta)
            z = rad * math.sin(theta)
            y = math.sqrt(r ** 2 - rad ** 2)
            rt = np.array([x, y, z])
            if(math.isnan(x)):
                print(f"x: {x}, y: {y}, z: {z}\nr:{rad}, theta: {theta}")
            roots.append(rt)
            print(rt)
            theta += step
            rad = a * theta ** 0.7

    return roots

def generateHair(radius, density, spread = "radial", mode = 0):
    roots = generateRoots(radius, density, mode)
    n = len(roots)
    print
    hair = []
    for base in roots:
        if spread == "radial":
            direction = base
        elif spread == "horizontal":
            direction = base * np.array([1,0,1])
        elif spread == "up":
            direction = np.array([0,1,0])
        hair.append(newHair(base, direction, .025, 8))
    return hair
```

I essentially projected a spiral pattern down onto the top hemisphere of the circle for the roots. Then depending on whether “radial”, “horizontal” or “up” is selected, the new method adds 8 points in space to connect to the each root that either go out from the centre, out horizontally or straight up. Here is an example of the distribution function visualised:

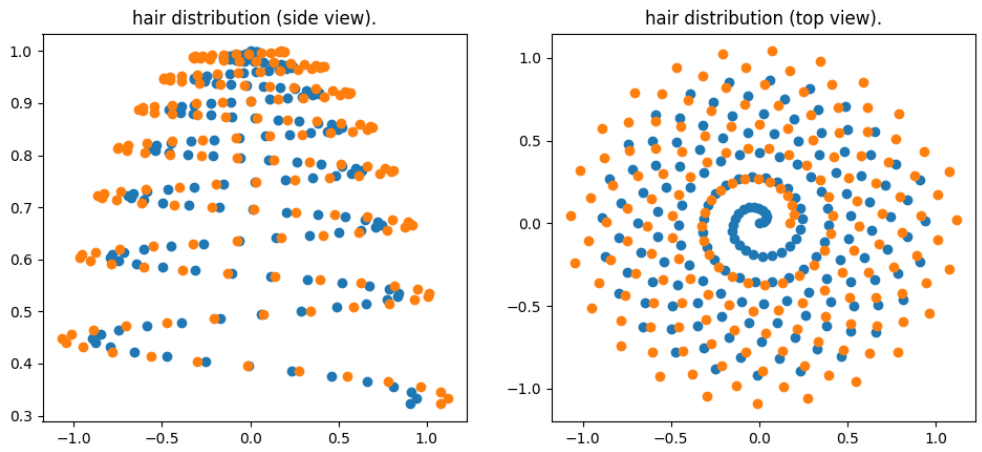


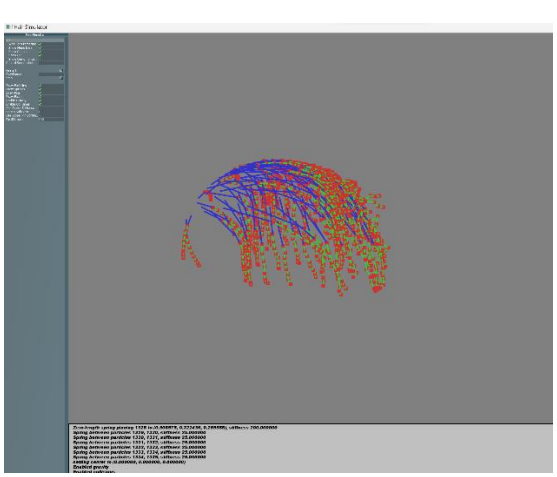
fig4

The orange points are the ends of the hair and the blue points are the roots. The left image isn’t to scale. The issue with this method is that the hair isn’t evenly distributed, there is more at the top and this is exaggerated due to the projection of the roots onto the surface (just taking the y value of the sphere at that point). Using my default parameters will five you a head of 167 hairs with 1336 total vertices. Increasing the density and the length of hair, as well as reducing the length of each chain link will increase that vertex count.

I then take this distribution and translate the points so that they are around a specified centre. each vertex is then added to an input file housed in the meshes folder under /batch. In this file we have the list of vertices with each line “v Px Py Pz Vx Vy Vz m” representing a vertex at point P with velocity V and mass m. I used 0 for the starting velocies of all hair particles. Then I have a list of springs where “s v1 v2 k” represents a *Spring* between the vertex on lines v1 and v2 with spring constant k. “z v Px Py Pz k” specifies a ZeroLengthSpring attached to vertex v and anchord to point P on the head. I then added “c Px Py Pz” and “h Vx Vy Vz” to represent the *center* and *head_speed* for the system. Running the generation script creats about 330 experiment cases, which have the center at various random positions and moving in several different directions at different speeds.

The program was altered start looping through these experiments for 200 frames each once you hit play (Spacebar), though be sure to turn on gravity (G) and collisions (K [c was taken for captions]) before starting.

The results can be found in the attached videos, but here are some sceenshots and images for a rough idea:



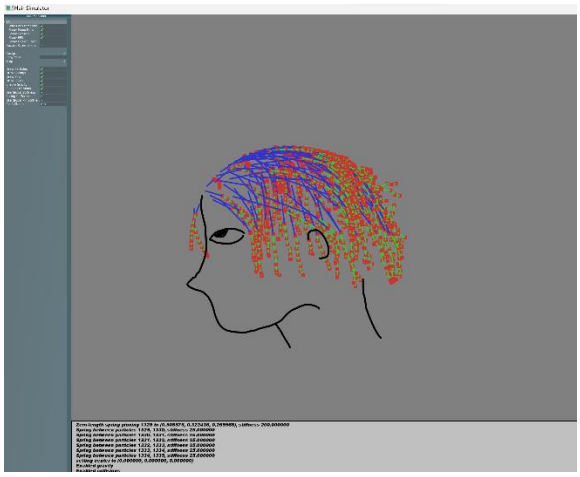


Fig5 and fig6 (I drew over the simulation to illustrate the direction of travel).

fig7 hair distribution in the hair simulator.

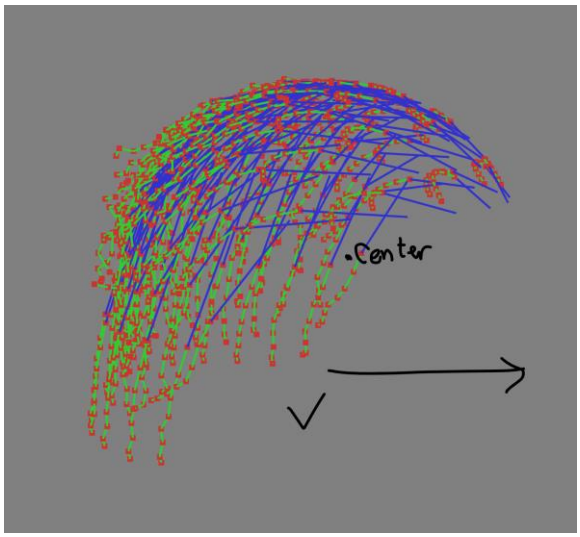


fig8 motion once play is hit.

Please see attached video(s) and code for more examples and details.

I then used the data gathered in my recordPositions() method (you can view the exact format by looking in the results folder of the project, though it is just rows of stacked vectors for each time step. The script read_data.py parses these lines and puts each experiment into a data frame containing x (a matrix containing the positions of the head's centre for each frame) and y a matrix which holds the stacked position vector at each frame. X has dimensions 3xT where T is the number

of frames (in my case 200 by default) and Y has dimensions 3N x T where N is the number of particles. To reduce this size I sampled only a portions of the frames (roughly 1/3) and a portion of each vertex. The hope being that a model trained on this data would be able to generate a point cloud Y for a given X which will be basically the same shape even with a portion of its vertices evenly removed. Here is an image example of the data visualised:

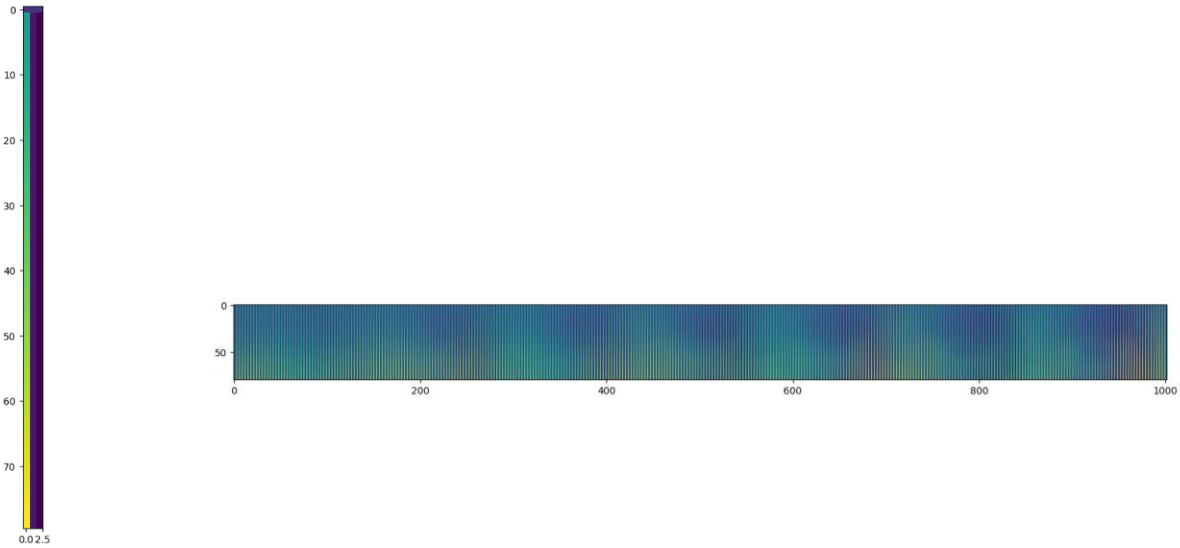


Fig9 x (left) and y(right)

Looking at X, you can see that in this example the head is moving in just the x direction (the second and third columns are constant). Looking at y is more difficult to interpret, but you can see that the motion is more chaotic towards the right hand where you have the vertices furthest from the centre and lowest down on the head. Time goes down in both images.

Unfortunately I was unable to begin training, but the aim was to use a modified image to image translation (such as Pix2Pix, Isola et al. 2016[3]) between the motion of the head and the output particle positions (the two “images” x and y. This is quite a different method to the technique used in Dynamic Neural garments in which states at times t, t-1 t-2 etc. were passed into the network to return t+1. This is because that method wouldn’t significantly speed up simulation as you would still have to pass in a matrix of positions for each new frame calculated. For my model it would be similar to just simulating it. The idea was to have this model pass in the motion of the head and in one go compute the output matrix within a reasonable buffer period. That way once an animation was played, instead of simulating each step or getting each step from a neural network, the program in use would just need to look up the values in Y. The good thing about his is that if a translation of x can be directly applied to y by just adding the shift to it.

Another extension I wanted to look into was making my hair model more complex. More sophisticated models use many more spring types to make the hair hold its shape better (mine was rather jellyfish-like), such as loose springs between alternating vertices or springs to hold a chain in a coiled position. Each addition would change the behaviour of the hair and lead to more complicated (and therefore slower) calculations per frame. It would have also been nice to use such extensions to simulate hair of many different types, representative of many ethnic differences in hair behaviour. There are also clumping techniques, grooming techniques, hair-hair friction implementations and interpolation methods (for simulating more particles than you could easily compute explicitly) that could have been explored.

Most of the interesting advancements that I've been following are from Pixar's studio and research group. For decades they've added to their techniques for modelling simulating and rendering hair using methods to style and guide the hair for simulators to add additional motion. Figure 10 shows their most publication (Ogunseitan, 2022[4]). And figure 11 shows an early example of simulations for different hair types (Hayley Iben et al., 2013[5]).

Figure 3: Process of generating guides that form a sphere of curls. ©Pixar.

Fig10

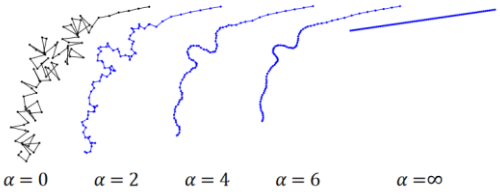


Figure 4: *Example of a stylized curly hair (far left) and the smoothed curves (blue) computed with α at 2, 4, 6, and ∞.*

fig 11 (©Disney/Pixar)

Example of stylized curly hair. © Disney/Pixar.

fig 12

References

1. * Meng Zhang, Duygu Ceylan, Tuanfeng Wang, Niloy J. Mitra: "Dynamic Neural Garments", 2021; [http://arxiv.org/abs/2102.11811 arXiv:2102.11811].
2. Stelian Coros , Carnegie Mellon University CS 15-467: Simulation Methods for Animation and Digital Fabrication [https://www.cs.cmu.edu/~scoros/cs15467-s16/].
3. * Phillip Isola, Jun-Yan Zhu, Tinghui Zhou, Alexei A. Efros: "Image-to-Image Translation with Conditional Adversarial Networks", 2016; [http://arxiv.org/abs/1611.07004 arXiv:1611.0700].
4. Sofya Ogunseitan: "Space Rangers with Cornrows - Methods for Modeling Braids and Curls in Pixar's Groom Pipeline", 2022; [https://graphics.pixar.com/library/Cornrows/paper.pdf]

5. Hayley Iben, Mark Meyer, Lena Petrovic, Olivier Soares, John Anderson, Andrew Witkin: “Artistic Simulation of Curly Hair”, 2016; [https://graphics.pixar.com/library/CurlyHairB/paper.pdf]