2IMV15 Simulation in Computer Graphics - Particle systems

Boris Rokanov (1396331), Georgi Kostov (1396773), Tar van Krieken (1244433)

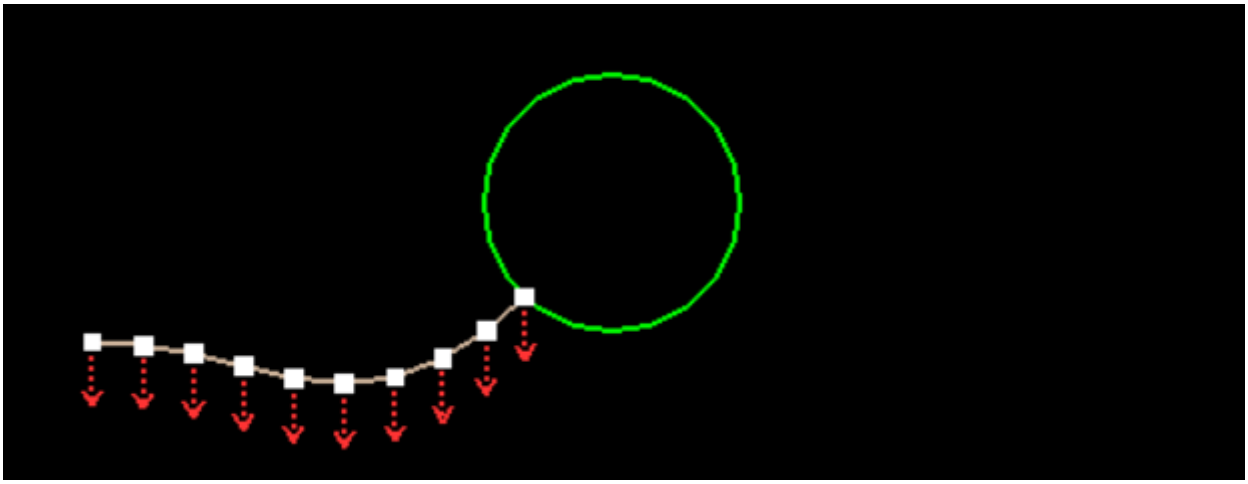

Fig. 1. Our particle system

Abstract—In this project a particle system was created in C++. This system contains a generalized force system, constraints system, various integration schemes, and a couple of different demos including a simple cloth simulation.

Index Terms—Particle system, Forces, Constraints, Cloth

1 Introduction.

Video games, digital art, simulations and many other modern software make wide use of particle systems. Due to their innate nature of defining physics on "particle-level", the implementation allows for a simple code definition that produces many complex simulation environments. Therefore, many of the difficult to recreate natural occurring phenomena such as fire, clouds, flocks of animals and others become easier to visualize and analyze [1].

In its essence, a particle system represents the movement of the particles in some environment after they got different forces applied. Often, the simulation involves different constraints that restrict the movement of some particles. In these cases, the forces and constraints have to be synchronized to produce a realistic particle movement. Yet another important factor for the particles is how they react on collisions. There are many types of collisions, such as particle-with-particle and particle-with-wall.

In this project, a simple particle system was created in C++. This system contains a generalized force system, constraints system, various integration schemes, simple collision handling and a couple of different demos including a simple cloth and a hair simulation.

2 Implementation

2.1 Forces

Together with the particles, the forces are one of the most basic elements of any particle system. To handle the different types of forces, we added two properties in the Particle class - the number w, which is the weight, and the force f, which is a $Vec2f$. The two-dimensional vector f defines the direction and size of the force applied on a given particle.

- *Boris Rokanov. E-mail: b.m.rokanov@student.tue.nl.*
- *Georgi Kostov. E-mail: g.t.kostov@student.tue.nl.*
- *Tar van Krieken. E-mail: t.m.k.v.krieken@student.tue.nl.*

At the end of each frame, the simulation uses an integration scheme (see section 2.3) to calculate the new proper velocity of the particles and calculate their new positions thereafter. Note that due to the implementation of the schemes, heavier objects are less susceptible to force, thus being both harder to move and slow down.

2.1.1 Gravitational Force

The first force that we implemented in this project was the gravitational force. This force can be defined as:

$$f_{grav} = mG$$

where m is the mass of the particle and G is the gravitational force. We extended the base implementation of the Particle class by adding a property called *mass*. Furthermore, we defined the gravitational constant as $Vec2f$ type as it is added to the *Force* field, which is of the same type. We know that $G \approx 9.81 m/s^2$ on the surface of Earth. In our case, we had to decrease this value to 0.05 because the particles were approaching the ground unrealistically fast. Note that based on the way that we scale down the velocity when a force is applied to the specific particle, we can see that the mass is eventually cancelled out.

2.1.2 Spring Force

Another requirement of this project was to implement spring force between two particles p_1 and p_2. We can define the spring force of the particle p_1 as:

$$f_{p_1} = -[k_s(||l|| - r) + k_d \frac{i \cdot l}{||l||}] \frac{l}{||l||}$$

where k_s is the spring factor, k_d is the damping factor, $l = p_1 - p_2$ (the distance between the two particles), $i = v_{p_1} - v_{p_2}$ (the difference in the velocity of the two particles) and r is the rest length. The spring force of the particle p_2 is defined as:

$$f_{p_2} = -f_{p_1}$$

During our experiments, we tuned the damping and spring coefficients. Eventually, we got the most interesting and natural results by setting $k_s = 0.1$ and $k_d = 0.2$. Regarding the rest distance value between the two particles, we used $r = 0.2$.

2.1.3 Angular Spring Force

The last force that we implemented is called Angular spring force, which uses a triplet of particles p_1, p_2, and p_3 (Figure 2). The key idea of this force is to approach a rest angle between the particles and is used to create a hair-like simulation. The angular spring is applied

Fig. 2. Triplet of particles

between the three particles, whereas there were two normal spring forces added between the tuples of particles - (p_1, p_2) and (p_2, p_3) to make the simulation more realistic. The angular spring force of the particle p_1 can be defined as:

$$f_{p_1} = -[k_s(||l|| - \sqrt{((||l_1||^2 + ||l_2||^2) - 2(||l_1|| \cdot ||l_2|| \cdot cos(angle)} + k_d \frac{(l \cdot l_d)}{||l||}] \cdot \frac{l}{||l||}$$

and the force of the particle p_2 is defined as:

$$f_{p2} = -f_{p1}$$

where the variables are:

- k_s - the spring force
- k_d - the damping force
- l - the distance between the particles p_1 and p_3
- l_1 - the distance between the particles p_1 and p_2
- l_2 - the distance between the particles p_2 and p_3
- $angle$ - the angle that has to be kept between the three particles
- l_d - the difference in the velocity of p_1 and p_3

One can notice that it is the same as the normal spring force, except that it is concerned with keeping the angle in a consistent rest position. In our experiments, we used $\frac{\pi}{2}$ as a rest $angle$ value. We found that by keeping the damping and spring coefficients higher ($k_s = 1$ and $k_d = 3$) than with the spring force, we get a more realistic hair-like simulation.

2.2 Constraints

The constraint system is based on the cancellation of forces. The system first calculates all forces that apply to each given particle, and then goes through all constraints to cancel out forces that would cause invalidation of these constraints. When only canceling forces, the particle may still slowly drift away from the constraint path, and thus a small force will also be applied to course correct for such drift.

Each constraint needs to adhere to a specific interface, in order for the general constraint solver to solve the system for multiple constraints at once. This interface consists of getters for the following properties:

- C : *float*; A value representing some form of distance between the target variable and the constraint (0 if adhering to the constraint)
- $\dot{C}$: *float*; The derivative of C with respect to time
- *particles* : $Vector < Particle >$; The particles that this constraint applies to
- J : $vector < Vec2f >$; The Jacobian matrix of C, I.e. the amount that each dimension of each particle affects the value of C.
- $\dot{J}$: $vector < Vec2f >$; The Jacobian matrix of $\dot{C}$, I.e. the amount that each dimension of each particle affects the value of $\dot{C}$.

We then specify the following vectors:

- q; $2n$-long (row) state vector containing the x and y coordinates of each particle.
- Q: $2n$-long (row) force vector containing the force applied to the x and y axis of each particle.
- W; $2n \times 2n$ diagonal inverse mass matrix, containing $\frac{1}{mass}$ for the mass corresponding to the particle for each axis.
- C; The m-long (row) constraint distance vector, derived by taking the C value of each constraint
- J: The $m \times 2n$ Jacobian matrix of C and the particle axis, derived by combining the values of J derived from each constraint

Next, the system solves the following equation to calculate the factors required for each constraint to cancel out the forces

$$\lambda^T = (JWJ^T)^{-1}(-\dot{J}\dot{q}^T - JWQ^T - k_sC - k_d\dot{C})$$

Here k_s and k_d are spring and damping force factors respectively, to course correct for when drifting occurs. This is solved using the Conjugate Gradient method, solving λ^T in:

$$(JWJ^T)\lambda^T = -\dot{J}\dot{q}^T - JW\ddot{q}^T - k_sC - k_d\dot{C}$$

Finally using λ the actual compensation forces for each particle can be calculated:

$$\hat{Q} = \lambda J$$

These constraint correction forces are then added to the forces of each of the axis of each particle.

In the actual implementation q, $\dot{q}$ and Q are represented by a vector of n particle objects. Each particle object stores a $vec2f$ for its current position (q), velocity ($\dot{q}$), and force (Q), as well as its own mass. Additionally, because most of these vectors are really sparse, they aren't explicitly created. Instead, implicit representations - such as the one specified by the interface of the constraint - are used. We for instance calculate vector d:

$$d = -\dot{J}\dot{q}^T - JW\ddot{q}^T - k_sC - k_d\dot{C}$$

In accordance to the following pseudocode:

Algorithm 1: calculating vector d

Input : $constraints : Vector < Constraint >$
Output : $-\dot{J}\dot{q}^T - JW\ddot{q}^T - k_sC - k_d\dot{C}$

```
Let d = 2n zero vector
for i ← 0 to constraints do
    Let c = constraints[i]
    d[i] = d[i] − c.C ∗ k_s − c.Ċ ∗ k_d
    for j ← 0 to c.particles do
        Let p = c.particles[j]
        d[j] = d[j] − c.J̇[j] × p.velocity − 1/P.mass · (c.J[j] ×
          p.force)
    end
end
```

Calculating the product of the matrix with λ (namely $(JWJ^T)\lambda^T$) is done similarly. Here JWJ^T is simply represented as a function mapping one vector to another. Hence, no explicit matrix has to be created and multiplied by. This is much more efficient since all these matrices are very sparse and would end up wasting many instructions by multiplying by 0, and adding 0.

Next, 3 specific constraint types were added:

- A CircularWireConstraint, which forces 1 particle to be a certain distance away from a fixed point.
- A RodConstraint, which forces 2 particles to be a certain distance apart.
- A LineConstraint, which forces 1 particle to be on a certain line

2.2.1 Circular Wire Constraint

The circular wire constraint takes 1 particle p, a target position (x_c, y_c) and a distance r. Let (x, y) be the coordinates of the particle, and let $(\dot{x}, \dot{y})$ its velocity. Then C was defined as

$$C(x,y) = (x-x_c)^2 + (y-y_c)^2 - r^2$$

From this the other attributes of the constraint were derived:

- $\dot{C} = 2((x-x_c)\dot{x} + (y-y_c)\dot{y})$
- $particles = \{p\}$
- $J = \{(2(x-x_c),\ 2(y-y_c))\}$
- $\dot{J} = \{(2\dot{x},\ 2\dot{y})\}$

2.2.2 Rod Constraint

The rod constraint takes 2 particles p_1 and p_2, and a desired distance r. Let (x_1, y_1) and (x_2, y_2) be the particles' coordinates, and let $(\dot{x}_1, \dot{y}_1)$ and $(\dot{x}_2, \dot{y}_2)$ their velocities. Then C was defined as

$$C(x_1,y_1,x_2,y_2) = (x_1-x_2)^2 + (y_1-y_2)^2 - r^2$$

From this the other attributes of the constraint were derived:

- $\dot{C} = 2((x_1-x_2)(\dot{x}_1-\dot{x}_2) + (y_1-y_2)(\dot{y}_1-\dot{y}_2))$
- $particles = \{p_1, p_2\}$
- $J = \{(2(x_1-x_2),\ 2(y_1-y_2)),\ (-2(x_1-x_2),\ -2(y_1-y_2))\}$
- $\dot{J} = \{(2(\dot{x}_1-\dot{x}_2),\ 2(\dot{y}_1-\dot{y}_2)),\ (-2(\dot{x}_1-\dot{x}_2),\ -2(\dot{y}_1-\dot{y}_2))\}$

An alternative definition was also tested:
Let $l = \sqrt{(x_1-x_2)^2 + (y_1-y_2)^2}$

- $C = \sqrt{(x_1-x_2)^2 + (y_1-y_2)^2} - r$
- $\dot{C} = ((x_1-x_2)(\dot{x}_1-\dot{x}_2) + (y_1-y_2)(\dot{y}_1-\dot{y}_2))/l$
- $particles = \{p_1, p_2\}$
- $J = \{((x_1-x_2)/l,\ (y_1-y_2)/l),\ (-(x_1-x_2)/l,\ -(y_1-y_2)/l)\}$
- $\dot{J} = \{((\dot{x}_1-\dot{x}_2)/l,\ (\dot{y}_1-\dot{y}_2)/l),\ ((\dot{x}_1-\dot{x}_2)/l,\ -(\dot{y}_1-\dot{y}_2)/l)\}$

2.2.3 Line constraint

The line wire constraint takes 1 particle p, a point on the line (x_c, y_c) and an angle in radians a. From this, 3 other constants are derived:

- $a = sin(a)$
- $b = cos(a)$
- $c = -a\dot{x}_c - b\dot{y}_c$

Let (x, y) be the coordinates of the particle, and let $(\dot{x}, \dot{y})$ its velocity. Then C was defined as

$$C(x,y) = a*x + b*y + c$$

From this the other attributes of the constraint were derived:

- $\dot{C} = a*\dot{x} + b*\dot{y}$
- $particles = \{p\}$
- $J = \{(a,b)\}$
- $\dot{J} = \{(0,0)\}$

2.3 Integration schemes

Our project successfully implements three explicit integration schemes that can be swapped at runtime. All three of them run on every frame, thus making the simulation step the same as the animation step.

2.3.1 Euler

The Euler integration scheme is the most straightforward. It initially applies all forces on all the particles, accumulating the total force applied to each particle. Next, the velocity and position of the particles get updated with the following formulas:

$$v = v + d_t \cdot \frac{f}{m}$$
$$p = p + d_t \cdot v$$

where v is the velocity, p is the position, f is the force, m is the mass of the particle and d_t is the time delta of the simulation.

2.3.2 Mid-point

The Mid-point integration scheme works in a similar manner with the difference being that it also takes into account where the particle will be and what forces will it have on the next step. This is calculated by storing the initial velocity and position of a particle, then temporarily applying half an Euler integration step. Next, another step is calculated on top of that, and finally the initial velocity and position are updated in accordance with this step.

This integration schema results in a smoother simulation compared to Euler, as the particles are less likely to overshoot their target.

2.3.3 Runge-Kutta 4

Following a similar strategy to the mid-point integration scheme, Runge-Kutta 4 does four half Euler steps before updating the velocity and position of each particle. However, looking multiple steps in the future with Euler integration would give incorrect position and velocity prediction.

Therefore, the algorithm makes use of the incremental delta of each of the Euler steps instead, updating the initial velocity and position with a third of the delta on the second and third step and a sixth of the delta on the first and forth steps.

This algorithm provides the smoothest, most real-like simulation of the three integration schemes.

2.4 Mouse Interaction

To create a better demonstration of the spring force, we implemented a mouse interaction with the particles. When the user selects a particle p, we create a temporal particle q that follows the mouse cursor (drag-like force). Between the selected and the mouse particle, we create a spring force which can easily be triggered by moving the mouse. We noticed that the temporal particle q gets the negative force of the particle p, which glitched our demonstration. Therefore, we extended our *SpringForce* class to get a boolean variable called *isMouseForce*. When this boolean is set to *true*, we do not apply force to the second particle (particle q). The particle q is being deleted on the mouse button release.

2.5 Cloth Simulation

Simple cloth simulation was achieved by placing particles in a rectangular grid-like structure and connecting them to their neighbors. Therefore, one can define a cloth by giving it width and height (i.e. number of particles), weight of each particle, rest distance between the particles and damping and spring factors. Additionally, we also experimented with two configurations of connecting neighbors - four-way and eight-way connections, as shown in figure 3.

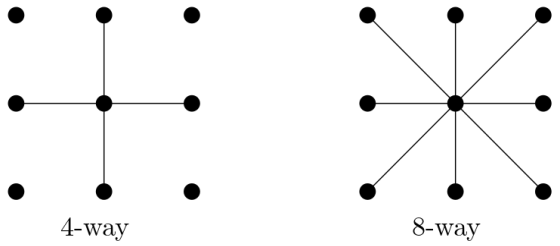

Fig. 3. 4-way cloth vs 8-way cloth

Note that in the eight way cloth, the rest distance between diagonal particles is $\sqrt{2}$ times the vertical/horizontal distance in order to preserve the rectangle-like shape of the cloth.

2.6 Collision

Collision handling helps create more realistic particle interactions, with a downside of slowing down the simulation. Nevertheless, we decided that we can safely ignore the speed reduction by implementing collision only for objects that we define as collidable (note that particles can be collidable). Therefore, we implemented a simple collision system, which in its roots is composed of two parts - collision detection and velocity change.

In order to facilitate the collision detection, we decided to limit collidable objects only to rectangular shapes, as both walls and particles are that shapes. That allowed the simulation to effortlessly check if any two-dimensional point p is currently intersecting the collidable object, by calculating the following Boolean value:

$$x_1 \leq p_x \leq x_2 \wedge y_1 \leq p_y \leq y_2$$

where x_1, x_2, y_1, y_2 are the bounding box coordinates of the collidable object and p_x, p_y are the x and y coordinates of p.

By looping through each particle just before it has moved and checking if its future position will intersect any of the collidable objects, the application can find all particles whose velocity needs to change in order to avoid intersection.

Next, the new direction and magnitude of the velocity of all those particles needs to be calculated, so that in the next frame they won't intersect with the collidable objects. This is done with the following formula:

$$p_v = (s \cdot n_x \cdot |p_{v_x}|, s \cdot n_y \cdot |p_{v_y}|)$$

where p_v is the two-dimensional velocity vector of the particle p, s is the velocity dissipating factor of the collidable object, and n is the normal vector of the collision. Note that n_x is calculated using the following formula (same formula applies for n_y, but concerning p_y, y_1 and y_2):

$$n_x = \begin{cases} -1 & \text{if } |p_x - x_1| < |p_x - x_2| \\ 1 & \text{otherwise} \end{cases}$$

3 Experiments and Results

3.1 Constraints

The constraints seem fairly stable, but in certain conditions high enough forces can still be applied to cause the system to explode. Constants $k_s = 60$ and $k_d = 10$ were arbitrarily chosen, and seemed to work well. When the chosen constants are either too great or too small, the system is more likely to explode under smaller forces. This primarily happens when a chain of rod constraints is fully extended, and does not have the freedom to contract, for instance when the endpoints have additional circular wire constraints. Both implementations of the rod constraint appear to be working similar well, with nearly identical behavior in various tests, to the degree that we couldn't tell which implementation was used. We however expect better performance from the implementation that doesn't make use of the square root, since this adds a lot of overhead to all calculations. Especially given that calculating a square root is a rather involved calculation to begin with. The number of steps done by the solver appeared to be around linear in the number of constraints are in the system when a high precision parameter was used. If the number of steps is limited below the desired number, everything still seems to operate stably, but the desired precision may not be reached. This means the system doesn't explode, but E.g. rod constraints may stretch a little.

3.2 Integration schemes

The performance and stability of each of the 3 integration schemes was tested in various versions of the scene, shown in figure 4. In each of these tests, no external forces other than gravity were applied. The scene was adapted by changing the number of particles that make up the chain: the chain length (cl). Additionally, various delta time steps (dt) were tested. Tables 3.2 to 3.2 show the results of these tests. Note that the duration of each time step was tested over 10 samples and averaged, before the system became unstable and exploded. Changing the time step size has virtually no effect on the duration of the time step, meaning that taking a slower method may be worth it if the time step can be increased to make up for it. We can see that Euler's method is very unstable, and choosing the midpoint method instead seems like a safe choice in every scenario. The Runge-Kutta 4 method seems like it may be worthwhile in some scenarios. Going from the midpoint method to Runge-Kutta 4 makes the computation about twice as slow, but we can more than double the time step size without the system exploding. Hence, while losing some precision, this may be a worthwhile move.

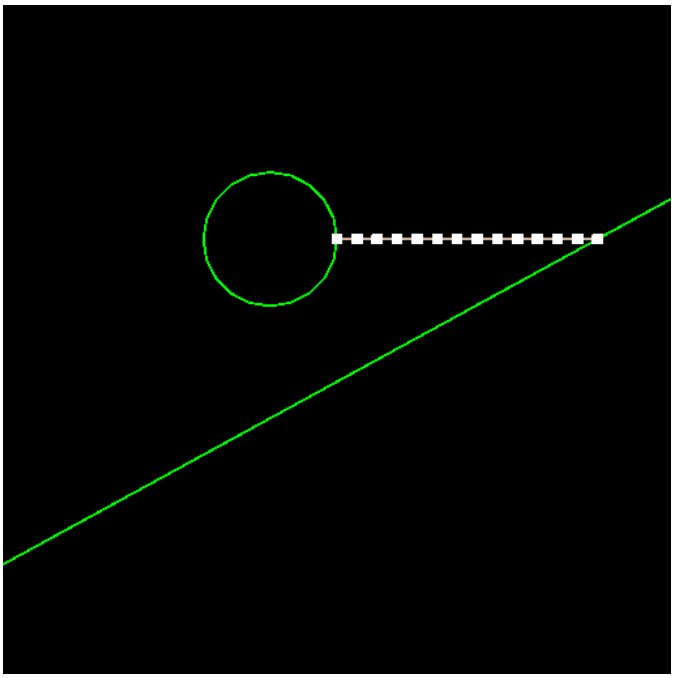

Fig. 4. Integration test scene

	dt = 0.2	dt = 0.1	dt = 0.05
Euler	0	2	2
Midpoint	1	12	52
Runge-kutta 4	15	40	72

Table 1. Maximum chain length

	cl = 6	cl = 12	cl = 24
Euler	0.01	0.01	0.008
Midpoint	0.1	0.1	0.085
Runge-kutta 4	0.25	0.22	0.115

Table 2. Maximum step size

	dt = 0.1, cl = 80	dt = 0.2, cl = 80
Euler	2	2
Midpoint	5	5
Runge-kutta 4	10	9

Table 3. Computation time per step in milliseconds

3.3 Cloth simulation

In order to visualize a realistic cloth interaction, one of the most important factors to simulate is how the cloth handles its collision. Therefore, our initial task was to design a proper testing ground. As seen in Figures 5, 6 and 7, we created two collidable walls - a big one on the left which stops almost the whole cloth, and a small one on the right that acts more like a nail. Additionally, we decided that it will be more interesting to test the cloth as a type of curtain, so we decided to add line constraint to the top row of particles.

Next, we experimented with different parameters for the cloth itself. The first interesting configuration was a 4-way 10x10 cloth with $dist = 0.05$, $k_s = 0.9$, $k_d = 0.6$ and $w = 0.2$, $dist$ is the rest distance, k_s is the spring factor, k_d is the damping factor and w is the weight of each particle in the cloth (see Figure 5).

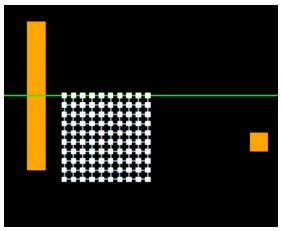

Fig. 5. 4-way cloth configuration

When there were small or no forces applied, this configuration seemed to preform well. For example, when colliding with either of the walls, it performed realistically, where the curtain cloth would bounce off the wall and close a bit. However, when adding gravity and/or strongly pulling on a certain particle of the cloth, it started to perform a little non-cloth-like, as the freedom of movement of each particle did not correctly reflect the freedom of movement of the cloth. In order to fix that, we began increasing the k_s value, as that should make the cloth stiffer, thus creating a better link of the cloth and its particles' movement. Nonetheless, that still did not manage to make the object fell cloth like, as it started becoming too stiff.

Therefore, in order to define a more realistic cloth, we also experimented with the 8-way cloth counterpart of the previous configuration - 10x10 cloth, with $dist = 0.05$, $k_s = 0.9$, $k_d = 0.6$ and $w = 0.2$ (Figure 6). Using 8-way connections instead of 4-ways helped improve the realism of the cloth, as it linked the particle and cloth movement without adding stiffness. The collision remained realistic as well. Additionally, sometimes particles would get stuck on the right wall, similarly that a cloth can get entangled when brushing against a nail (Figure 7).

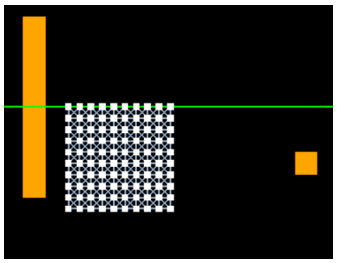

Fig. 6. 8-way cloth configuration

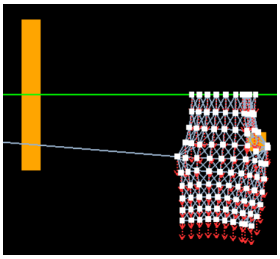

Fig. 7. 8-way entangled cloth example, pulled to the left

3.4 Hair simulation

To demonstrate our implementation of the Angular spring force, we added a hair-like simulation to our project. The hair simulation is composed of particle-triples connected with an angular spring and two normal springs. For our experiments, we added several curls, where the first particle of each curl was attached to a line constraint (Figure 8). By tuning the spring and damping coefficients, we decided to keep $k_s = 1$ and $k_d = 3$ for our demo presentation. To make the subtending angle between the three particles approach a rest angle of 90°, we set the $angle = \frac{\pi}{2}$. Additionally, to improve our demonstration, we added a particle-with-particle collision.

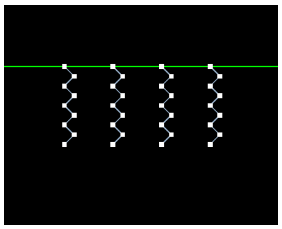

Fig. 8. Hair-like simulation

4 Optional Features

4.1 Visualization of forces

As an additional feature to our particle system, we implemented a visualization of the forces that are applied to the specific particle(s). Figure 9 shows an example of how the gravity force is displayed in the simulation.

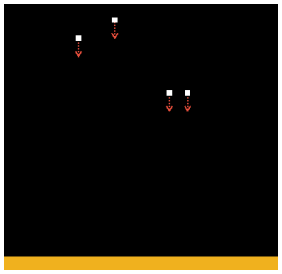

Fig. 9. Visualization of gravity

5 Conclusion and Future Work

During this project, we developed a particle system to which the user could apply various forces such as gravitational, normal spring, and angular spring forces. Additionally, the particles could be constrained with a circular wire, a line, and a rod. We created all the required integration schemes (Euler, Mid-point, Runge-Kutta 4) and experimented by tuning their parameters (dt and cl). To demonstrate the capabilities of the particle system, we implemented a cloth simulation and a hair simulation, which were using forces, constraints, and collisions.

In the current implementation of the hair simulation, the curls can get flipped due to the forces, but it would be more natural if they are always pointing to the ground. As a future work, we would like to add a small gravity force which will cause the curls to always point down.

In the future, developing implicit integration will be very beneficial to the simulation. It will likely improve the efficiency of the application by allowing for more realistic results to be calculated faster, which will in turn reduce the breaking points of the application caused by inaccuracies.

Another feature which were not able to implement due to time constraint is the energy transfer during collision. Currently in the simulation, particles bounce off the collidable object, while it stays in place. In reality, any two objects that collide will move at least a tiny amount, depending on their weight and collision impact.

6 CONTRIBUTIONS

Table 4 shows the contribution of each of the group members. Note that we were with 4 group members, but one group member quite the course before starting on the project. Note that the mentioned times exclude the time spent in the lectures, but do include general code as well as environment (E.g. virtual machine) setup and debugging.

Student	**Tasks**	Hours estimation
Boris Rokanov	Gravity force Spring force Mouse interaction Angular spring Hair simulation	38
Georgi Kostov	Cloth simulation Collision Cloth interaction	36
Tar van Krieken	Generalized constraint structure Integration schemes	40

Table 4. Personal contribution.

REFERENCES

[1] W. T. Reeves. Particle systems—a technique for modeling a class of fuzzy objects. *ACM Transactions On Graphics (TOG)*, 2(2):91–108, 1983.