Report project 1

Tom Sandmann
1208785

Abdullah Rasool
1194155

I. Introduction

For this project we implemented a constrained particle system. In this paper we discuss the details behind this particle system, which include some implementation details and decisions we made.

This paper has the following sections:

- In section 2 we discuss the generalized force structure and how we extended it to create several forces;
- Section 3 covers the constraint dynamics, how they are computed and an example of how we can combine several constraints.
- Section 4 deals with how interaction with the user is addressed.
- In section 5 we look at the three numerical integration schemes which have been implemented.
- We conclude in section 6 with our implementation of a cloth using a spring-mass model and some experiments with different spring constants.

II. Generalized Force Structure

In this section we described how the generalized force structure was implemented. We constructed a class that represents a very general force. All forces extend this general force class and implement the computeForce method. This method is responsible to compute and apply the forces given a set of particles. In addition, there is also a function that draws the direction and the magnitude. This is for example shown as the blue line in figure 1.

A. Gravity

The most basic force is gravity which simply takes a list of particles and computes the gravitational force by multiplying the mass of each force with the gravitational constant. Instead of directly adding this force to each particle, this force is scaled down to fit the window of the application. If this was omitted the particle would disappear in a few milliseconds time, due to the big gravitational constant and the size of the window.

B. Horizontal force

Another similar constraint is the horizontal force, this is applied to slide a particle along a horizontal line. This force accepts a particle and a magnitude and applies the force in the x direction of the particle. After the evaluation of the integration scheme (see later) the particle moves in the direction specified by the force.

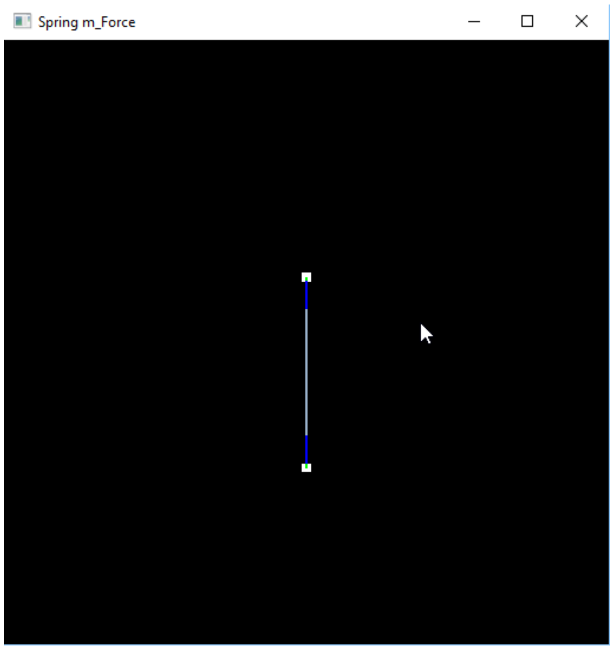

Fig. 1. Particles attached with spring.

C. Spring force

This force simulates a spring between two particles, given a certain rest length and two constants for the springs. The computeForce method implements the formula for spring force which is given in equation 1.

$$
\begin{aligned}
f_{p_1} &= (k_s(||l|| - r) + k_d \frac{\dot{l} \cdot l}{||l||}) \cdot \frac{l}{||l||} \\
f_{p_2} &= -f_{p_1}
\end{aligned} \tag{1}
$$

Where:

- l is the distance between p_1 and p_2
- $\dot{l}$ is the difference between velocities of the two particles
- r is the rest length
- k_s is the spring constant
- k_d is the damping constant

This force has also implemented the draw function, executing this force on two particles results in a scenario which is shown in figure 1.

D. Wall force

Another force which was created for the sliding cloth scenario is the wall force. If the wall force detects that particles have gone through the wall or are at exactly the same position then it applies a small bouncing force on the affected particles. For a first attempt, the force only checked if particles were exactly at the position of the wall. However, due to the time step that solution did not yield desired results. Below we show a small code snippet of how this force is computed.

```
void WallForce::computeForce() {
if (p->m_Position[0] >= x){
p->m_Velocity -= Vec2f(p->m_Velocity[0] +
     0.1f, 0.0);
p->force -= Vec2f(p->force[0] + 0.1f,
    0.0);
}
}
```

In order to achieve the bouncing effect when a particle hits a wall the current force is subtracted and a small bouncing force and velocity is added.

E. Angular spring force

An angular spring force involves three particles. A specific angle is specified. This angle indicates the angle the triplet should be in rest. This angle is also called the rest angle. Our first idea of an angular spring force was that such a spring force consisted of a spring between the two outer particles. In addition, we thought that we had to calculate the length between these two edges during every evaluation and apply the right forces to the particles. However, some literature research showed us that this was not really the case. Eventually, we found a couple of equations that we were able to use in our system. The equations on which we based are angular spring force are described in [1]. The equations given for the angular spring describe the spring forces (f_1, f_2) and (f_3, f_4) between two pairs of intersecting points $(1, 2)$ and $(3, 4)$:

$$f_1 = -k_s \left[\frac{I_{21} \cdot I_{43}}{||I_{21}|| \quad ||I_{43}||} \right] \frac{I_{43}}{||I_{43}||} \qquad f_2 = -f_1$$

$$f_3 = -k_s \left[\frac{I_{21} \cdot I_{43}}{||I_{21}|| \quad ||I_{43}||} \right] \frac{I_{21}}{||I_{21}||} \qquad f_4 = -f_3$$

where

- $I_{21} = x_1 - x_2$ and $I_{43} = x_3 - x_4$
- k_s is the stiffness constant
- c is the cosine of the rest angle between I_{21} and $I43$

In the beginning, we did not really understand how to apply these forces to our particles. However, we managed to get some sort of working example in the end, although we were not completely sure whether this is how the angular spring force would like.

III. Constraints

In this section we describe the development of the constraints. Again, a general constraint force object was created that requires some methods to be implemented. These are used when computing the constraint force. The constraint force is computed after all the forces have been applied to their respective particles. In order to compute the constraint force a matrix is build, this matrix consists of $\#C$ columns and $2n$ rows for a two-dimensional particle. $\#C$ is the number of constraints and n is the number of particles. In this matrix the value of cell (i, j) is the partial derivative of C_i with respect to particle j's state vector.

Each constraint holds a vector with this partial derivative $\frac{\partial C}{\partial q}$ which looks like $\{\frac{\partial C}{\partial x_1}, \frac{\partial C}{\partial y_1}, \frac{\partial C}{\partial x_2}, \frac{\partial C}{\partial y_2}\}$ if $q = (x_1, y_1, x_2, y_2)$. These values should then be placed in the appropriate row belonging to the particle and to the column which belongs to constraint C. In order to achieve this we first assign all particles a unique row. Next, we loop through all the constraints and get all the particles that are associated to that constraint. We get the row of these particles and using the partial derivative vector we place the values in the correct rows. This is shown in the code snippet below:

```
    //Build the J matrix
for ( int column = 0 ; column <
    constraints.size() ; column++ ) {
ConstraintForce* constraint = constraints
    [column];
for ( int particlePos = 0; particlePos <
    constraint->particles.size();
    particlePos++ ) {
//Set x value of particle in J
Particle* particle = constraint->
    particles[particlePos];
J(column, particle->row) = constraint->
    getJ()(0, particlePos*2);
Jdot(column, particle->row) = constraint
    ->getJDot()(0, particlePos*2);

//Set y value of particle in J
J(column, particle->row+1) = constraint->
    getJ()(0, particlePos*2+1);
Jdot(column, particle->row+1) =
    constraint->getJDot()(0, particlePos
    *2+1);
}

C(column) = constraint->getC();
Cdot(column) = constraint->getCdot();
}
```

Now that we have created the matrixes $J, \dot{J}, C$ and $\dot{C}$ we can compute the constraint force $\hat{Q}$. For this we use the conjugate gradient algorithm [2] to solve the following equation:

$$JWJ^T\lambda = -\dot{J}\dot{q} - JWQ - k_sC - k_d\dot{C} \qquad (2)$$

Using the conjugate gradient algorithm we can solve λ. In order to obtain the constraint force we need to multiply it with J^T. This results in a vector which is of size $2n$, this way the constraint forces for each particle is decomposed into the x

and y direction. Using a similar indexing method as was used to build J we can extract the constraint forces and add these to the forces of each particle. All of the matrix and vector arithmetic is provided by a third-party library called Eigen [1].

A. Sliding wire constraint

One custom constraint which was added is to make sure that a particle stays on a horizontal wire. This is used for the sliding cloth scenario. In that scenario one such constraint is created for all top particles in the cloth square, see figure 2.

In order to satisfy this constraint we use the following function:

$$C(y_p, y_w) = y_p - y_h \tag{3}$$

Where y_p is the y coordinate of the particle and y_w is the height of the wire. With this function we want to make sure that the particle p is at the same height as the wire, which simulates a sliding particle on a wire.

B. Mixture of constraints

One scenario which is present in our implementation is a mixture of constraints:

- Circular wire constraint: a particle has to stay on a circle of a certain radius;
- Sliding wire constraint: a particle can only move along a horizontal line;
- Rod constraint: two particles should always stay at a specified distance of one another.

Lets take a look at the constraint formulas used for the circular wire and rod constraint. In order to keep a particle on a circle, which is equivalent to having a certain distance to a specified point (center of the circle). We can describe this using the following formula:

$$C(x, y) = (x - x_c)^2 + (y - y_c)^2 - r^2 \tag{4}$$

Where x, y is the coordinate of the particle, x_c, y_c is the coordinate of the center and r is the radius of the circle. From this we can compute the derivative of C with respect to time, $\dot{C}$.

Next, lets look at the constraint formula for the rod constraint. If we look at the descriptions of both constraints we can observe that they are equivalent. Again, we have two coordinates whose distance should exceed a certain constraint length r:

$$C(x_1, y_1, x_2, y_2) = (x_1 - x_2)^2 + (y_1 - y_2)^2 - r^2 \tag{5}$$

Again we can compute $\dot{C}$ from this constraint function. We can use these functions to build the matrix J as was shown before and using the conjugate gradient algorithm to compute the constraint forces $\hat{Q}$ and apply these forces to all particles.

[1]More information, see: http://eigen.tuxfamily.org

IV. User interaction

Users are able to draw additional springforces by clicking, dragging and releasing the mouse. This will create two particles: one at the position at which the user started to drag his mouse and one where the user released his mouse. For clarity, we decided to make an additional check to see whether the user is already clicking (or releasing) the mouse on another particle or whether a particle is within a specific threshold of the mouse. If this is the case, we do not draw both particles. When the user clicks and releases the mouse immediately (i.e. not dragging), then a single particle will be placed at the given position (again under the proximity check that we mentioned before).

V. Numerical integration scheme

Numerical solvers are being used to approximate the solution of the ordinary differential equation (ODE) that are used in the particle simulator. In our application, we make use of a couple of different ODE solvers. In our description of the methods, we assume that the initial value problem is as follows:

$$\begin{aligned} y(t_0) &= y_0 \\ \dot{y} &= f(t, y) \end{aligned}$$

- **Euler**. The Euler method (or also called forward Euler) is one of most simple methods for numerically solving ODEs. It is a first order method, and is often use as a building block for more complex methods. Although it can be described in a very intuitive way, we only list the formula that described the method:

  $$y_{n+1} = y_n + hf(t_n, y_n)$$

  In the formula above (and for the resulting formulas), h is the step size.
- **Midpoint** Although the Euler method is very easy, it comes at the price of less accuracy. Therefore, Euler's method is not used in practice. One can do better by using a centered estimate instead of the one-sided estimate used by Euler. As mentioned before, this gives us some additional accuracy. The explicit midpoint method, which is the one we implemented, is defined as follows:

  $$y_{n+1} = y_n + hf(t_n + h/2, 1/2(y_n + y_{n+1})$$
- **Fourth Order Runge-Kutta (RK4)**. RK4 is one of the more famous methods of the Runge-Kutta family. In fact, the midpoint method can also be called the 2nd order Runge-Kutta. RK4 is given by the following formula (with $n > 0$):

  $$y_{n+1} = y_n + h/6(k_1 + 2k_2 + 2k_3 + k_4)$$

  where

  $$\begin{aligned} k_1 &= hf(t_n, y_n) \\ k_2 &= hf(t_n + h/2, y_n + h/2k_1) \\ k_3 &= f(t_n + h/2, y_n + h/2k_2) \\ k_4 &= f(t_n + h, y_n + h/2k_2) \end{aligned}$$

We also did some tests with the different solvers under different step sizes. In one of our configurations, we have the circle and a line, on which two particles reside. if we started the simulation in using Euler, we could see that the particle on the circle did not need a lot of time before its position was on the circle. However, when we were using the midpoint and RK4 methods, we would see that there was a little hop in the particle's position before its position stabilized on the circle. When we decreased the step size to 0.001, we saw this 'converging effect' in even more detail for both RK4 and midpoint.

In our cloth simulation (and again using a step size of 0.001), we saw that in the RK4 case, the cloth only lifted over the line by a couple of centimeters. This was different in the case when the Euler and Midpoint methods were being used. In these cases, the cloth lifted much more over the line compared to the other methods. In addition, the simulation seemed to run faster in the RK4 case.

While testing the gravity force, we had to increase the step size a little bit, otherwise it would take forever to see the particles move under the application of this force. 0.1 turned out to be a value in which the simulation would look nice. Again, there was little to no difference between the solvers being used.

In the end, we did not see a lot of difference between all the solvers. Although we would expect to see some accuracy differences when the time step would be set to a very small value, we were not able to reach those cases. We did however find it weird that simulations using RK4 seemed to run faster compared to the simulations that were performed using the other solvers. In the end we had no time to dive further into this phenomenon, but will definitely take a closer look at it once the deadline for the code submission is expired. We were glad to see that we managed to get all of the solvers working. From all of the solvers, We had the most trouble with understanding RK4. However, in the end everything fell in place.

VI. Cloth

For the cloth model we have used the spring-mass model. In this model a cloth is created with a set of particles which are interconnected using springs. In figure 2 there is a graphical representation of a cloth.

When we apply a horizontal force on this cloth we get a cloth similar as in figure 3.

As was mentioned earlier we implemented a wall force against which the cloth can collide and will bounce of.

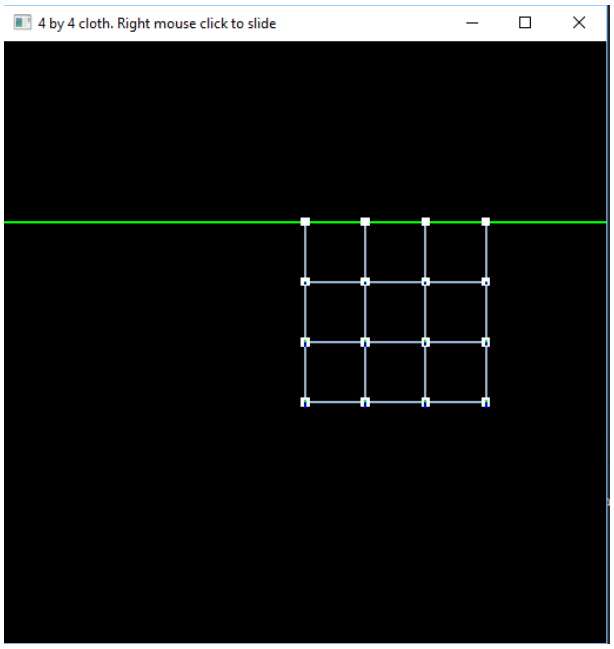

Fig. 2. Cloth model without forces

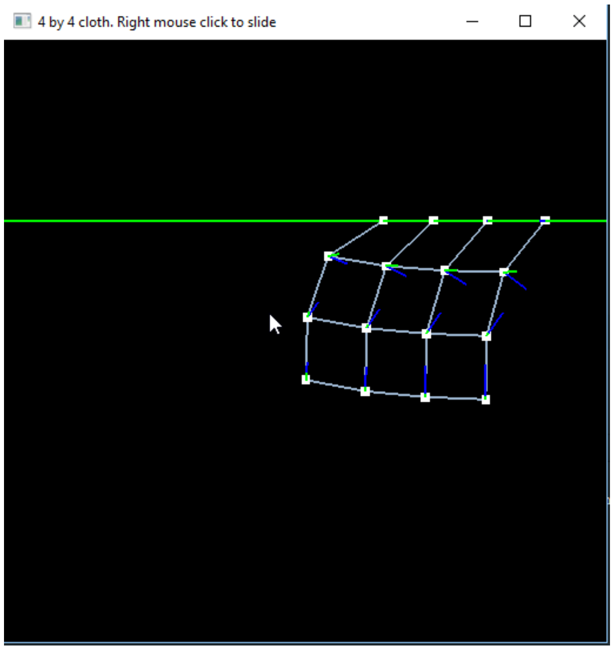

Fig. 3. Cloth model with horizontal forces applied

A. Spring constants

We experimented with the values of the spring constants, these results are presented here.

- Low spring constant (eg. 1), low damping constant (eg. 1): When starting the cloth springs get some force applied on them. The particles move very closely to each other but do not intersect each other.
- High spring constant (eg. 10), low damping constant (eg. 1): The spring bounces a lot, even going above the horizontal wire and intersecting the cloth. This is because the spring constant is way higher than the damping constant. So the particles are not damped, therefore can go above this horizontal line.
- Low spring constant (eg. 1), high damping constant (eg. 10): The particles in the spring bounce a lot less, they come to rest a lot quicker. This is because of the big damping force.

References

[1] David Bourguignon and Marie-Paule Cani. Controlling anisotropy in mass-spring systems. In *Computer Animation and Simulation 2000*, pages 113–123. Springer, 2000.

[2] Jorge Nocedal and Stephen J Wright. Conjugate gradient methods. *Numerical optimization*, pages 101–134, 2006.