Cloth Simulation

P. Schönberger & R. Naraghi-Taghi-Off

RheinMain University of Applied Sciences, Wiesbaden, Germany

Figure 1: *Simulated flag using our cloth implementation*

Abstract
The simulation of cloth is needed in various industries. Firstly, there is the game development industry, which adds atmosphere and life to the game by displaying clothes when a wind blows or an object gets caught in it. Often, companies even profit by offering exclusive, purchasable clothing for the player's character. However, the main focus of games is not the clothes, but the player experience. One industry where the accurate representation is essential is the 3D movie industry. Here, a realistic representation is desired in order to enhance the movie experience. In the context of this elaboration it is explained in more detail how cloth can be simulated and which components are needed to implement it. Based on this, a prototype was developed, which realizes a composition of the components and allows interactions by dragging the cloth and applying wind.

1. Introduction

Nowadays, the simulation of real objects and physical laws is becoming more and more authentic. These include gravity, which is omnipresent and affects all objects, but also the collision of objects so that they do not intersect. The simulation of cloth involves a complex combination of physical laws. Each area of a piece of cloth reacts differently after physical impact, because the composition consists of many points - so-called vertices - and connections that hold the vertices together and can transmit effects of forces. A common connection could be, for example, the simulation of springs, which represent a certain elasticity, but more importantly can transmit forces from vertex to vertex. By determining the position and velocity of the vertices, among other things, a collision can be determined. This paper first deals with related works to give a small overview for similar projects. Then, in the third section, the theory behind the simulation of fabric is explained. The fourth section highlights the implementation of the prototype, which has been implemented in the context of this paper.

2. How Cloth Simulation Works

This section deals with the necessary theory about how cloth simulation works and how it could be implemented. First, the equations of motions are described. Then integrators are considered to integrate them. Finally, collision is discussed.

2.1. Equations of Motion

In order for a cloth to have a texture, equations are needed to describe and regulate its motion. There are two common equations of motion, which are explained in more detail in this chapter.

Mass-Spring Model

A classic model is the mass-spring model. It is usually easy to implement. Several mass points M are placed at regular intervals and connected by means of a spring like in 2. A network is created which causes dependencies. The movement of a single point affects the springs, which transmit the force to other springs in the surrounding area via the vertices. Each vertex has its own local information, such as the current velocity, position, or the applied force.

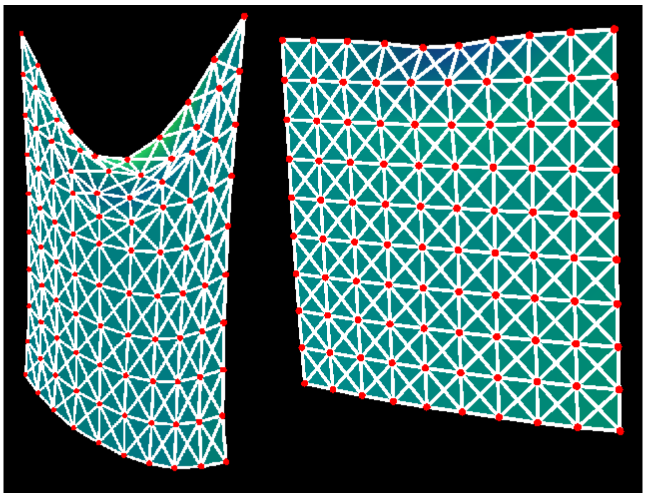

Figure 2: *Red nodes: vertices; White lines: springs; Diagonal springs: necessary to resist collapse of the face; green area: mesh faces*

The equation of motion looks as follows:

$$F_{net}(v) = Mg + F_{wind} + F_{airresistance} - \sum_{springs \in v} k(x_{current} - x_{rest}) = Ma$$

Here **M** is the mass of a vertex. To calculate its mass more accurately, it is helpful to find the area of each triangle and assign $\frac{1}{3}$ of it to the mass of each adjacent vertices. This makes the mass of the cloth the mass of the total area of all triangles times the mass density of the cloth. The gravity vector is **g** with a constant value of $(0, -9.8, 0)$, which corresponds to the average gravity of the Earth and is calculated in m/s^2. $X_{current}$ is the current length and X_{rest} the original state of the springs, where F_{wind} is a varying environmental influence by other particles. The constant **a** helps to make adjustments to the nature of the cloth to stabilize its behavior. The stiffness of the cloth is influenced by **k**. If k is too high, the cloth will hardly collapse. If k is too high, it will collapse more easily.

Elasticity Model

The Elasticity Model is a more precise model, which is based on the fact that energy is integrated over the surface of the cloth. Here, triangles and edges are preserved in size and are affected solely by energy (compression and expansion) [BW98]. In addition, edges resist bending, which means that bending the adjacent surfaces requires energy from the initial bending of that edge. Apart from this, triangles also resist deformation, as this also requires energy.
The formula requires a large vector **S** that represents every important variable in the system, which include the position and also the velocity of all vertices. Here **E(S)** is the energy as a function of the current state of the system, which allows the equation of motion for a vertex at position **(x, y, z)** to be calculated as follows:

$$F_{net} = \{\frac{\partial E(S)}{\partial x}, \frac{\partial E(S)}{\partial y}, \frac{\partial E(S)}{\partial z}\}$$

$$\frac{\partial E(x)}{\partial x} = \frac{E(x+\Delta x) - E(x)}{\Delta x}$$

2.2. Integrators

To determine the position and velocity of a vertex in the next time step, a method for integrating the equation of motion is necessary. A distinction is made between **explicit** and **implicit**.

Explicit

Explicit models are often easy to implement, but have the problem that the values of variables reach explosive proportions. The size of the time steps does not matter here. To deal with this problem, enough damping should be applied so that the energy decreases naturally. Another possible solution would be to use an implicit integrator, which is discussed in section 2.2. The term *explicit* means that the state at time **t+1** is evaluated by considering the state at time **t** [Fis14].

- **Euler's method** is a simple method for integration. The equations for determining the velocity and position look like this:

$$v_{t+\Delta t} = v_t + \Delta t(\frac{dv}{dt})_t$$

$$x_{t+\Delta t} = x_t + \Delta t v_t$$

  Here, these can be easily derived like this:

$$v_t = \frac{dx}{dt} = \frac{\Delta x}{\Delta t}$$

$$x = \Delta t v_t$$

$$x_{t+\Delta t} - x_t = \Delta t v_t$$

  This also applies to the derivation of the velocity.

- **Verlet Algorithm** is a model that does not need information about velocity. To determine this, it looks at the current position and the previous time step:

$$x_{t+\Delta t} = 2x_t - x_{t-\Delta t} + (\frac{dv}{dt})_t(\Delta t)^2$$

  By not requiring the speed, this one is easier to implement.

Implicit

Implicit integrators use the state variable of the current time step and the derivative of the next time step to obtain the state variable of the next time step.

- **Euler's method** is in the implicit version as follows:

$$v_{t+\Delta t} = v_t + \Delta t(\frac{dv}{dt})_{t+\Delta t}$$

$$x_{t+\Delta t} = x_t + \Delta t v_{t+\Delta t}$$

  In this case only $v_{t+\Delta t}$ has to be calculated, from which finally $x_{t+\Delta t}$ is derived. To find the new velocity, we need to calculate the term $(\frac{dv}{dt})_t$. To do this, we consider a large state vector **S** with all velocities and positions in the system, as a $6n \times 1$ matrix (n = number of all vertices). By linearizing the equation of motion, $\frac{dv}{dt}$ at time **t** can be represented as follows:

$$(\frac{dv}{dt})_t = QS_t$$

  Here, **Q** represents a large $3n \times 6n$ matrix, which represents the linear relationship between the change in velocity and the state of the system [Fis14]. This is finally used to calculate the new velocity.

- **Symplectic Euler's Method** is a semi-implicit method, which is a compromise between the implicit and explicit model. Its equation is as follows:

$$v_{t+\Delta t} = v_t + \Delta t(\frac{dv}{dt})_t$$

$$x_{t+\Delta t} = x_t + \Delta t v_{t+\Delta t}$$

  It is called semi-implicit because the velocity is explicit and the new position is implicit.

2.3. Collision

To make the behavior of a cloth in interaction with another solid object realistic, dealing with collision is essential. This allows the cloth to recognize the immediate overlap with another object. It should be noted that an intersection with itself is also possible. Therefore, in the following sections, the collision of the cloth with other solid objects as well as with itself will be considered and explained in more detail.

Cloth-Object

In this type of collision it is assumed that solid objects have their fixed existence. This means that the position and also motion at any time **t** is known.
The physically correct model would be as follows: First, at the time of the start **t** and the position of the cloth **x(t)**, the position **x(t + dt)** is predicted. For each face of the mesh, the initial and final positions are considered. If this falls between **t** and **t + dt**, then it means that an interaction with another object has occurred. In this case, the exact time when the surfaces first came into contact is calculated. Then we check if the face was in contact with another surface before and if it is not anymore in the meantime, so that the surfaces are separated from each other again.Following this, the time of the earliest of such a collision is determined, the simulation is reset to that time, and the face that was hit with the surface is marked as the contact. As soon as a surface has been marked, the static and kinetic friction force is calculated by the normal vectors of the contact surfaces. At this point, the simulation continues and everything repeats.
Because of this very difficult and costly methodology to implement, Matthew Fisher [Fis14] recommends a somewhat simpler approach:
Through his algorithm, a collision is viewed primarily through the position of a vertex in the next time step. This determines whether the vertex would be inside another object. The processing of this information can be either to move the vertex back to the position it came from or to repel the object.

Cloth-Cloth

To make cloth look more realistic, its collision with itself is an important factor. To implement such a collision, there is a simple method:

1. Calculation of the new position
2. Determination of an intersection (Fig. 3: middle)
3. Return to the previous time step (Fig. 3: right side)
4. Manipulate the magnitude of the velocity so that a collision no longer occurs (Fig. 3: right side)

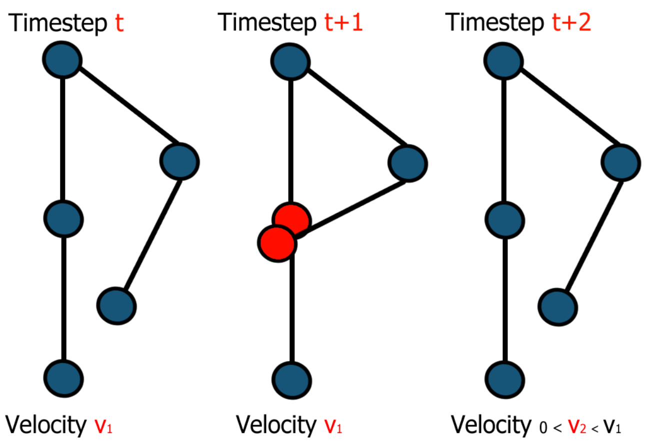

Figure 3: *At the beginning there is no collision. In the next time step a collision occurs. After the collision its position is reset and the velocity changed*

The problem is that this algorithm takes a lot of time to calculate each collision and slows it down. However, this one is easier to implement and provides safe collision avoidance [Fis14].

3. Implementation

We implemented a cloth simulation in JavaScript with the Three.js framework. The simulation runs in real time in a web browser and consists of a constraint-based representation of a rectangular piece of cloth with Verlet integration. It can interact with basic forces like gravity and wind and can be dragged by the user. It also features cloth-cloth collision, preventing it from clipping through itself and causing wrinkling, which makes it look more life-like.

3.1. Tools

The project is implemented in JavaScript using the Three.js framework for rendering the cloth in 3D. These tools were chosen because browsers are widely available, enabling easy sharing of the project, and because they make it easy to get the simulated cloth on screen without too much work.

Three.js

Three.js is a JavaScript framework which simplifies 3D programming with WebGL by providing abstraction like meshes, cameras, lights and scenes. It also includes some basic maths utilities like vectors and matrices. It alleviates some of the work involved in displaying 3D graphics in JavaScript, while still allowing for a deep level of customization. This allowed us to shift our focus from rendering to the actual simulation.

3.2. Approach

We decided to use a basic mass-spring model (see 2.1) because it is intuitive to understand and simple to implement. The springs act as a set of constraints that are to be satisfied. The acceleration calculated for each mass is integrated using the Verlet method (see 2.2), because it is more stable than similar explicit methods and less complex than implicit methods.

A simulation step consists of three actions:

1. Iterate over all masses, applying external forces and integrating them
2. Iterate over all springs, moving their masses relative to the difference between resting and current length
3. Check for cloth-cloth intersections and resolve them

The cloth is represented internally as a matrix of masses and a set of springs connecting them. Each mass has a position in 3D space and a weight. The springs consist of two confining masses and their resting length.

For cloth-cloth interaction, the cloth is interpreted as a matrix of spheres. If two non-adjacent spheres overlap, the associated masses are moved apart from each other so they no longer touch, stopping the cloth from self-intersecting.

For rendering, the masses making up the cloth are interpreted as vertices and a simple mesh is created using indexed face sets.

3.3. Components

The simulation roughly consists of three components, the mass-spring model, Verlet integration and collision detection. These are all applied for each simulation step in that order.

Mass-Spring Model

The mass-spring model defines the underlying structure of the simulated cloth. It consists of a set of masses that are arranged in a grid and springs connecting them horizontally and vertically (see Figure 4). By varying the amount of masses and springs, as well as the elasticity of the springs, different consistencies can be achieved. It is represented by the class `Cloth`, which has the following methods:

`constructor`
Creates the internal representation for the cloth and sets up the given amount of masses and springs in a rectangular shape to fit the supplied size.

`generateGeometry`
Creates a Three.js `BufferGeometry` with two triangles for each quad making up the cloth.

`updateGeometry`
Updates the `BufferGeometry` vertex positions and sets a flag that tells Three.js to send the updated vertex data to the GPU.

`simulate`
Performs a single simulation step.

`intersect`
Resolves any self-intersections.

`blow`
Adds a wind force coming from the position of the camera toward the cursor.

`drag`
Drags the supplied mass toward the cursor.

When constructing a `Cloth` instance, two arrays are generated, `masses` and `springs`, containing instances of the `Mass` and `Spring` class respectively. A `Mass` can be constructed with a position and weight and has two methods, one for adding acceleration and one for integrating it's motion. `Spring`s are constructed with two `Mass`es and a resting distance and have only a single method `satisfy`, which approaches the `Spring`s resting length by moving the confining masses towards or away from each other.

Applying the mass-spring model means satisfying the constraints represented by the springs. Every springs wants to retain it's resting length. If it is stretched, it contracts, subsequently stretching neighbouring springs. By applying these constraints in short time steps, the springs behave like an elastic material.

Verlet

Integrating the motion of a mass using the Verlet method is rather simple. Each simulation step, acceleration from the different forces

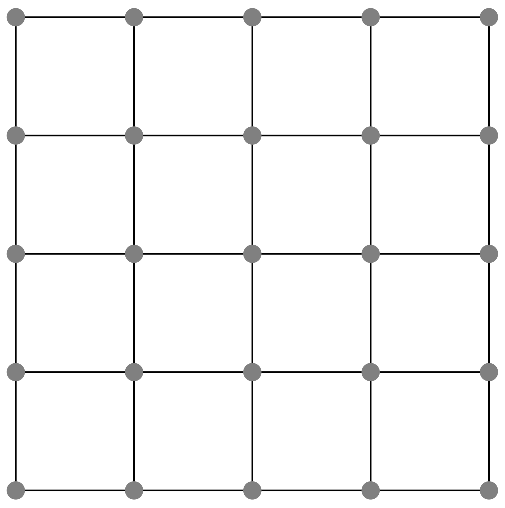
Figure 4: *Visualization of a mass-spring model with 5 × 5 masses.*

acting on a mass is added up and in the end it is integrated to calculate the position for the next simulation step. The formula used for integration can be seen in equation 1. x_t is the position at time t, $(\frac{dv}{dt})_t$ is the accumulated acceleration and Δt is the time between two simulation steps. The method has been slightly altered to apply a *Drag* constant (set to 0.97) to the velocity before adding it to the current position to improve numeric stability.

$$x_{t+\Delta t} = (x_t - x_{t-\Delta t}) \times Drag + x_t + (\frac{dv}{dt})_t \times \Delta t^2 \qquad (1)$$

The position is then updated and the acceleration reset to zero for every mass.

Collision

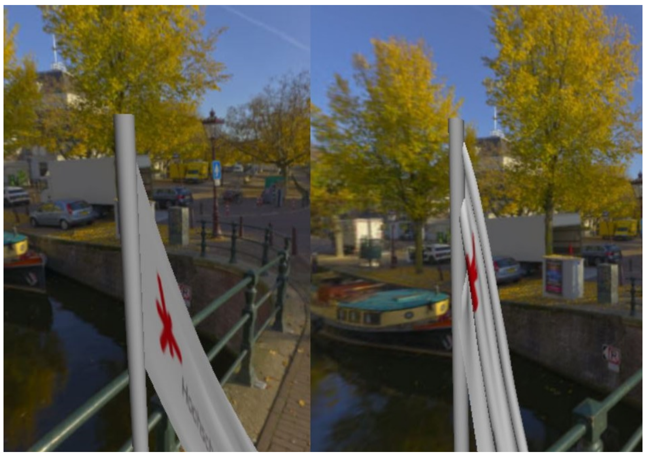

Figure 5: *Cloth-cloth collision disabled on the left and enabled on the right*

Unless the cloth being simulated is the only thing in a scene, it has to interact with its environment. For this, collision is crucial. But collision is not only needed for interaction with surrounding objects, the way the cloth acts on itself also has to be simulated for realistic results, most notably so that creases form instead of the cloth clipping through itself (see Figure 5).

We focused on cloth-cloth collision, meaning the aforementioned interaction of the cloth with itself. For this, every point making up the cloth is checked against every other one, except for its direct neighbours.

If the distance between them is smaller than a set minimum value (the distance between two masses when the cloth is in a resting state), they are moved away from each other to be exactly that distance apart.

Cloth Interaction

In the demo, the user can interact with the cloth in two ways. First, the cloth can be dragged with the mouse, and second, wind can be blown at the cloth (see Figure 6).

Both of these forces are applied in the same fundamental way, by adding to the acceleration of every point that is being affected.

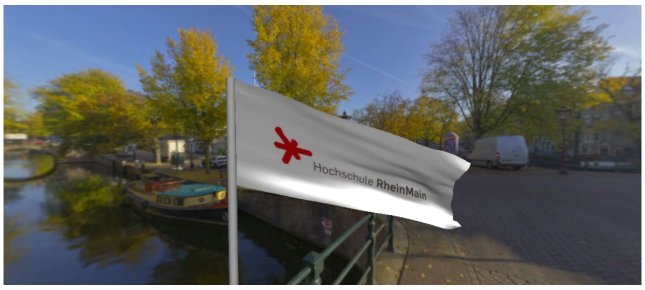

Figure 6: *Blowing wind at the cloth*

For dragging, a ray is cast from the camera through the cursor and if it intersects any of the mesh's faces, that face is "grabbed." A force is then added to the masses making up the face in the direction the user is moving the cursor until the face is "released." Wind is applied in the same way, except that the force points in the same direction as the camera to emulate a wind force coming from the user's point of view.

Rendering

For rendering, the internal representation is converted to a Three.js `BufferGeometry`. The `BufferGeometry` is a close mapping to the way geometry is actually stored and passed to the GPU by WebGL. Instead of providing variables for vertex, normal and color information, the user can set `BufferAttributes`. These `BufferAttributes` contain instances of special typed array classes to handle the data more efficiently. Since the cloth mesh could change every frame, the vertex data has to be updated and sent to the GPU every frame as well. The `BufferGeometry` makes this very easy, it allows tagging individual `BufferAttributes` as changed, and will then update them internally. When doing so, one needs to also update the mesh's bounding volume, because features like frustum culling depend on it.

A texture is added to the cloth by setting up UV values for the vertices, which is very simple since the cloth is a rectangle and the vertices are initially set up in regular intervals (see Figure 4). The texture itself can be applied using a Three.js `Material`.

4. Conclusion & Future Works

We managed to simulate a piece of cloth in real time in the browser with a relatively small amount of very concise code. The framework is modular and could easily be extended to more features. This was in part achieved by re-writing the entire project after gaining a better understanding of the underlying concepts. Cloth-cloth collision also works in real time and provides realistic creasing while preventing self-intersection.

The obvious next step would be to add cloth-object interaction, the complexity of which largely depends on the type of object, more precisely how easy it is to determine whether a given point is contained within the volume of the object.

Other improvements would be simulation of cloth thickness, improved numeric stability, for example by utilising an implicit integration method instead of an explicit one, or tearing.

References

[BW98] BARAFF D., WITKIN A.: Large steps in cloth simulation. In *Proceedings of the 25th Annual Conference on Computer Graphics and Interactive Techniques* (New York, NY, USA, 1998), SIGGRAPH '98, Association for Computing Machinery, p. 43–54. URL: `https://doi.org/10.1145/280814.280821`, `doi:10.1145/280814.280821`.

[Fis14] FISHER M.: Cloth, 2014. URL: `https://graphics.stanford.edu/~mdfisher/cloth.html`.