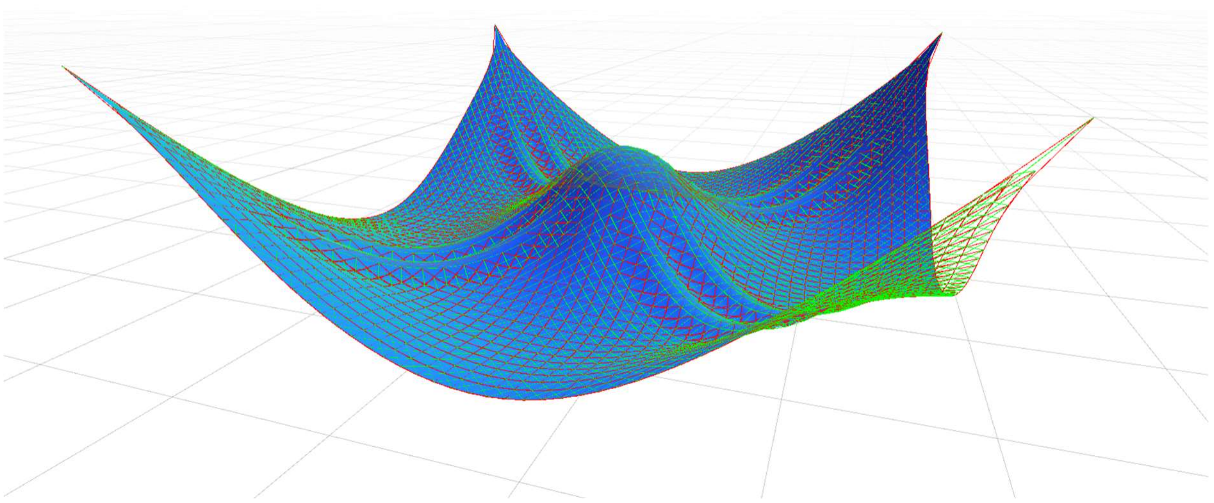
Cloth Simulation

Using Hooke's Law and Verlet Integration



C2 Computational Physics Module

William Yu

Table of Contents

Overview

Cloth modelling is the simulation of cloth computationally. It is a form of modelling soft-body physics (distinct from rigid body physics) as it deals with a deformable object in 3D space. For the realistic depiction of fabric and soft membranes, an understanding of the underlying physics is necessary. The areas of importance are as follows:

- **What are the physics in a cloth in relation to its characteristics?**
  Identifying the properties in a cloth, and using these properties to derive a physical model to demonstrate these properties (Sections 1 - 2)
- **How do we use the physics to model its movement and behaviour over time?**
  Developing a computer simulation that shows how the system will change over time by repeatedly applying the model (Sections 3 - 6)
- **How do we visualise the model?**
  Applying graphical techniques to effectively demonstrate what the system is actually doing in a visual and understandable way (Section 7 - 8)
- **How realistic is the model?**
  The goal of creating the simulation is to explore how accurately the interactions in cloth can be modelled. A good evaluation of its success is therefore how 'realistic' the simulation looks, in subjective terms, in comparison to a real-life piece of fabric.

The plan is therefore to develop a realistic looking fabric model, applying known physical concepts.

[1] Fabric Characteristics

What are the characteristics of cloth? We observe first that it is soft. In a physical context, this means that an object made of cloth is free to deform, changing its shape. In doing so, it can stretch. Fabric is elastic (though the extent of its elasticity is dependent on the type of cloth – the material, the thread density, the thread pattern etc).

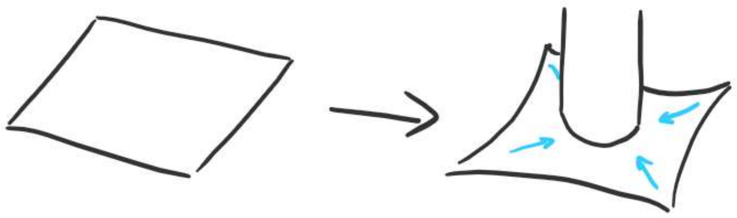


Fig 1.1: Mesh shape deforms and stretches under force

This would make it difficult to model as a single object – the shape is constantly changing. Moreover, deformations only affect nearby areas of cloth: applying a force and raising a corner of a piece of fabric only affects the shape of that corner, if no other uneven force is acting on the cloth.

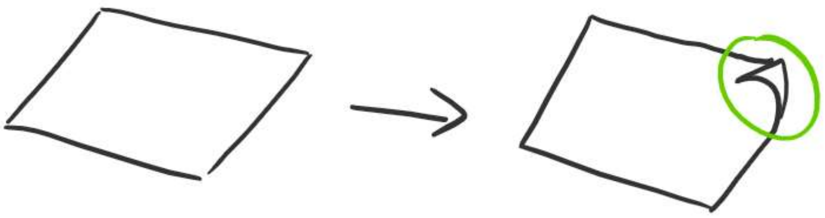


Fig 1.2: Local area of influence

[2] Physical Model

A good model for a piece of cloth is therefore an array of particles (which represent masses) in a grid mesh. Each particle is only connected to neighbouring particles, so that any influence is local (but changes can still propagate through the mesh). Computing changes on the cloth is therefore limited to iterating over each particle and applying changes (e.g. adding forces). The connectors between particles can be modelled as springs: this accounts for the deformation, by allowing particles to hinge freely within the constraints of the spring, as well as allowing for stretching. It simultaneously applies repulsion to near particles and tension to distant ones.

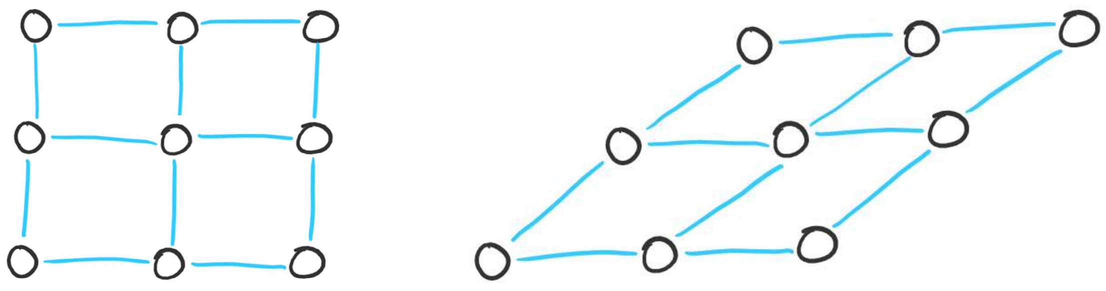


Fig 2.1: Structural springs (blue) between particles but prone to shear collapse

There are 3 types of spring in this mesh. The first of these are **structural springs**, which follow the grid lines horizontally and vertically. These ensure a distance between particles, and the overall geometry of the mesh is constant. However, as shown in the right of Fig 2.1, simply having structural springs makes the mesh prone to collapse via shear forces.

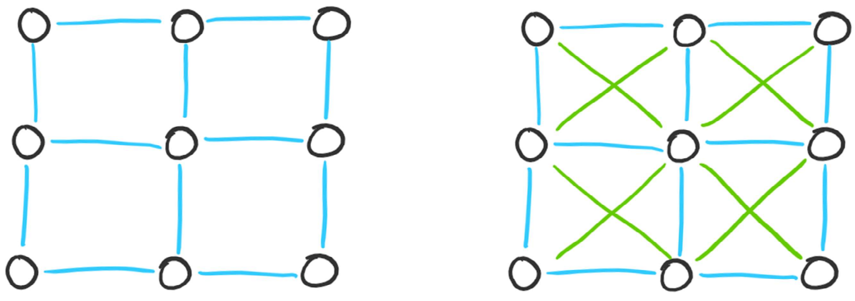


Fig 2.2: Shear springs (green) act as diagonal constraints and prevent lateral motion

To counteract this, we add **shear springs**, which connect diagonal particles.

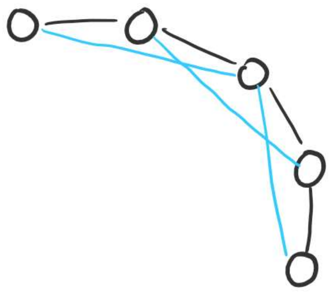


Fig 2.2: Bending springs (blue) apply tension across particles.

Finally, to prevent the mesh from folding over like an infinitely thin piece of paper, **bend springs** which span over two masses are added. These add tension across joints over the mesh.

[3] Defining Geometry

Grid definition

I initially modelled a strictly 2D mesh of the fabric in Processing, an open source visualisation program. I did this to first get the mesh mathematics correct, especially in working with arrays and linking neighbouring particles.

I shall be denoting a **structural spring** as **χ**, a **shear spring** as **σ**, and a **bend spring** as **β**.

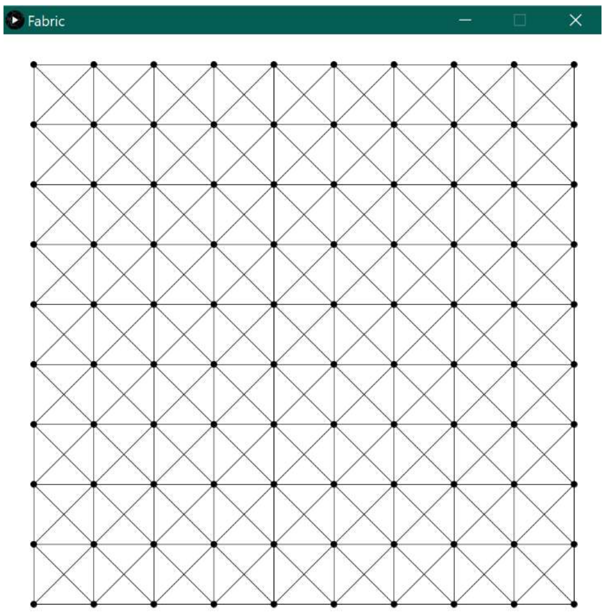


Fig 3.1: Example output from Processing software with structural and shear springs

I derived the formula for the number of **structural springs** to be as follows:

$$n_\chi = 2hw - h - w$$

The number of **shear springs**:

$$n_\sigma = 2(w-1)(h-1)$$

The number of **bend springs**:

$$n_\beta = 2(hw - h - w)$$

Where h and w are the height and width of the grid respectively, in terms of the number of particles.

The distance between particles, and therefore the length of the springs, horizontally and vertically across the lattice is defined as the **resting distance**. It is defined as a constant before runtime. This scalar is used for defining the dimensions of the grid and the length of the springs with no extension. It is immediately clear that the different springs have different resting distances. These distances in terms of r for each type of spring is as follows:

χ	r
σ	$r\sqrt{2}$
β	$2r$

As a scripting language, Processing is capable enough for this project. However, in the pursuit of improved visuals, better performance with 3D renderings, and an environment I was more familiar in, I decided to migrate the project to Unity.

Object Classes

I defined two classes for the grid: A **Particle** that is a point on the grid, and a **Spring** that connects them. The particles have masses and can have forces applied to them. The springs constrain the movement of the particles in a spring-like fashion. See the end for a complete documentation of the individual classes.

Object Arrays

I defined an array to contain the particles. Since the array is one dimensional, finding the particle at given cartesian coordinates was done with the formula

$$index = xh + y$$

where x and y are iterations through the height and width respectively, and h is the height. I further created three arrays to contain each of the three types of spring. I iterated through every x and y coordinate in the grid and created a particle at that location. I then iterated though them again, adding springs to adjacent particles (if they existed). Since particles have more than one spring attachment, I wrote a function to find the next available index in an array to correctly index the springs.

[4] Equations of Motion

Spring Forces: Hooke's Law

The spring forces throughout the mesh are calculated using Hooke's Law:

$$F_{spring} = -ke$$

where k is the spring constant and e is the extension of the spring, as calculated by the difference between the distance between such particles P_1 and P_2 and the resting distance, r:

$$e = |\overrightarrow{P_1P_2}| - r$$

In code, e is calculated via vector calculation (`ext` in the snippet below):

```
Vector3 offset = origin.pos - target.pos;
//Find distance between particles
float d = offset.magnitude;
//Find extension
float ext = d - r;
```

Hooke's Law is then applied in the direction of the spring:

```
//Hooke's Law and direction vector
Vector3 f = ext * -k * offset.normalized;
```

The resulting force is applied to both particles connected by the spring. This is repeated for every spring to give a sum of spring forces. This method can achieve good results; however, this requires fine tuning of the k constant in tandem with the damping, mass, etc. For too low levels of k, the cloth cannot support its own weight and sags unrealistically. However, if k is too large, the cloth becomes too tight, and can even gain energy as a function of time. The system undergoes positive feedback, and vertex positions tend towards infinity.

Environmental Forces: Gravity, Wind, and Air Resistance

Gravity is a simple calculation:

$$F_G = mg$$

where m is the mass of a particle, and g is the acceleration downwards due to gravity and is defined as a constant before runtime.

Wind is applied based on the position of the particle in space. I define it as a function of the mesh that takes the position of the particle, and applies a force based on that particle's position. First, I define a function for computing a wind force in the x axis based on a time offset:

$$x(\theta) = |\sin(P_x + \theta)|$$

where P_x is the particle's original x position, and θ is an offset increased every frame.

I then add a periodic sideways oscillation based on the z component:

$$z(\theta) = \frac{1}{2}\sin(\theta)$$

Where θ is the same offset as above. The value is halved to reduce the sideways oscillation proportional to the parallel wind. The function for returning the wind force is therefore

$$F_{wind}(\theta) = \langle x(\theta), 0, z(\theta)\rangle$$

This force is then scaled by a constant wind coefficient, defined before runtime, before being applied to the particle. This method varies the wind force both spatially and temporally.

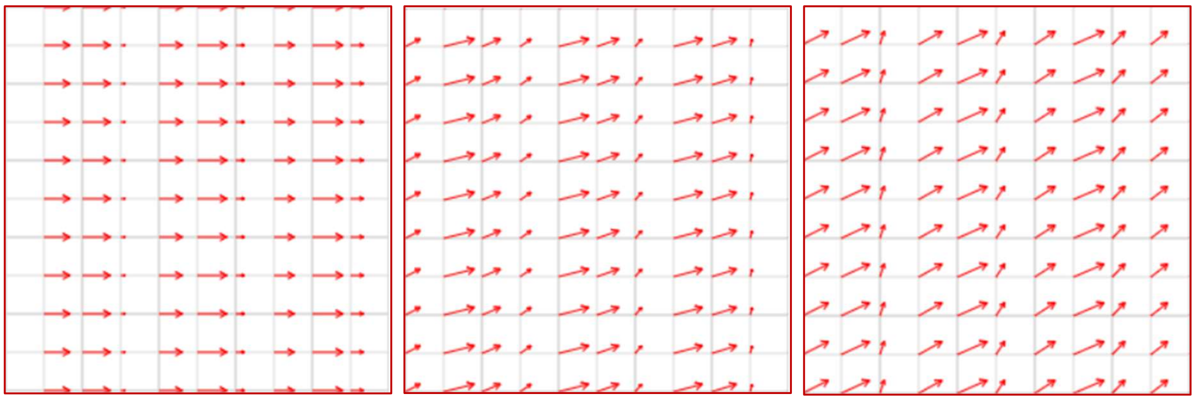


Fig 4.1: 2D vector fields showing x variation at $\boldsymbol{\theta} = \mathbf{0}$, $\boldsymbol{\theta} = 0.5$ and $\boldsymbol{\theta} = 1$

Applying Forces

Forces can be applied to the particles using a Particle method, `applyForce`. This function augments the sum of forces acting on the particle, and is repeated for any force acting on the particle. The forces can be internal to the cloth and particle (i.e. springs) or environmental (e.g. gravity, wind), such that sum of forces on a particle $\boldsymbol{P}$ are:

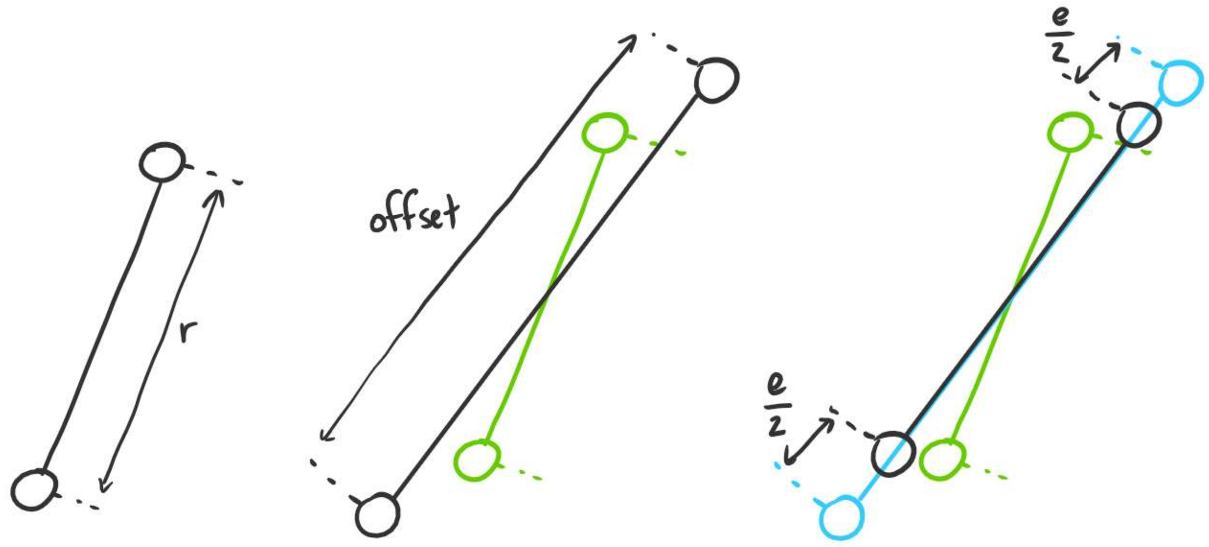
$$F_{Total} = \sum_{Springs \in \boldsymbol{P}} F + F_G + F_{wind} + \cdots$$

where $Springs \in \boldsymbol{P}$ represents all the springs attached to $\boldsymbol{P}$. This equation also gives us leeway to add further environmental forces later (for example to apply collision forces). The sum of the forces at a given time step is stored as a property of the Particle.

Aside: Constraint Calculations

While using Hooke's law is more accurate physically, using a constraint method can sometimes be better in delivering a visual representation. This involves finding the positions of the particles, and then adjusting them so that the distance between them is equal to the resting distance r. This is therefore not a physical model but can be useful for a consistent visual output when struggling to define e.g. k.

Instead of multiplying the extension by k, the percentage extension is found by dividing by the offset distance. This is halved to account for the adjustment to both particles, and then the particles are multiplied by this offset percentage to bring them back to r distance apart.



Default Length Constraints **After Updating Points** **Relaxing**

Fig 4.1: Applying constraints to keep distance constant

In code, the method for finding extension is the same as for finding it for Hooke. However, instead of applying Hooke's Law, the following is done:

```
//Find percentage extension
float perc = (ext / d);
//Divide by 2 since distance is applied to both particles
perc /= 2;

//Adjust particles
origin.pos -= offset * perc;
target.pos += offset * perc;
```

While this method does not employ actual physics to the system, it can give a more consistent visual output, as it forces an 'artificial' equilibrium. It can be more stable and less prone to velocity runaway.

However, since it modifies the position directly, it overrides collision contact forces and therefore is unable to demonstrate interactions between objects.

I originally used this method while I was struggling to get the Hooke visualisation working. I have kept it as a non-physical alternative that still produces a viable simulation.

[5] Collision

Collision is calculated against another particle. As such, it is limited to spherical objects in my demo. Inter-mesh collision is not available, due to the complexity of checking particles against each other. Collison is calculated using the vectors involved, and then applying a contact force.

We first check to see if the particle has collided with another particle, using the function **`checkCollision`**. Fig 6.1 shows an example collision between particles p and q. The distances r and R are the radii of p and q respectively. We compare the sum of the radii to the actual distance between the origins of p and q; in the case that

$$r + R \geq \left|\overrightarrow{PQ}\right|$$

or equivalently,

$$r^2 + R^2 \geq \left|\overrightarrow{PQ}\right|^2$$

a collision has occurred.

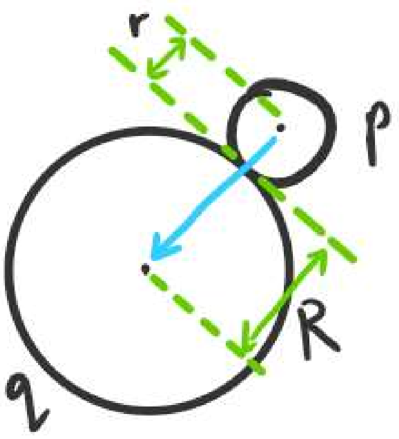


Fig 6.1: p, q, *and their radii*

The latter is used because getting the square magnitude of the offset is a cheaper operation than getting the pure magnitude, since that requires finding the square root.

Once a collision has been confirmed, the forces are calculated. In Fig 6.2, F_{Total} is the sum of forces acting on p as a result of the equations of motion. At the point of collision, an additional normal force, F_N, is exerted on p. To calculate F_N, we need to know F_C – that is, the component of F_{Total} in the central direction. To do this, we normalize $\overrightarrow{PQ}$ to find the normal direction, then use

$$\begin{aligned} comp_v u &= |u|\cos(\theta) \\ &= |u|\frac{u \cdot v}{|u||v|} \\ &= \frac{u \cdot v}{|v|} \end{aligned}$$

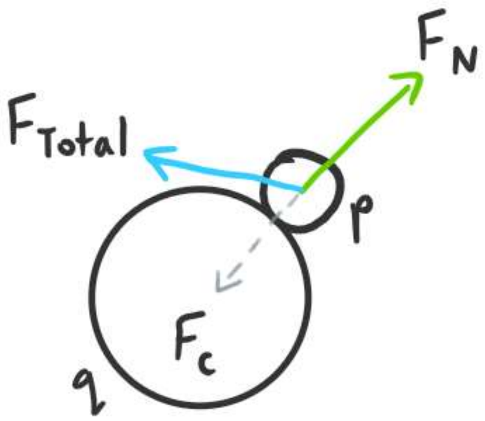


Fig 6.2: Free body diagram for p

substituting $\widehat{PQ}$ for v and F_{Total} for u. This gives us the scalar representing how much of F_{Total} is in the direction of $\widehat{PQ}$. Multiplying through by $-\widehat{PQ}$ returns the final force in the normal direction: F_N. This force is then applied to P.

The full code is therefore as follows:

```
Vector3 offset = Q.pos - pos;
if (offset.sqrMagnitude < r * r + Q.r * Q.r)
{
        Vector3 offDir = offset.normalized;
        //find component of force in collison d
        float product = Vector3.Dot(offDir, f);
        Vector3 normal = -offDir * (product / c
        applyForce(normal);
}
```

[6] Integration

Acceleration and Velocity

Acceleration is calculated simply using Newton's 2nd Law on the net force of the particle:

$$a = \frac{F}{m}$$

having calculated F using the equations of motion. This gives the instantaneous acceleration on the particle.

The average velocity of the particle since the last time step is calculated as follows:

$$v = \frac{x(t) - x(t - \Delta t)}{\Delta t}$$

where x is the position of the particle as a function of t, the time. The velocity is only calculated to apply **damping** to the particle – that is, resistance to acceleration. Damping in this simulation is defined as a deceleration at every time step, is proportional to the velocity, and inversely proportional to the mass. It is calculated with the following equation:

$$a_d = \frac{vd}{m}$$

where v is the velocity as calculated above, d is a damping constant defined before runtime, and m is the mass of the particle. This value is subtracted from a every time step before integrating for position.

Verlet Integration

This is a method of working out the position of an object as it moves. Essentially, it is a solution for the kinematic equation for the motion of any object:

$$x = x_0 + v_0 t + \frac{1}{2}at^2 + \frac{1}{6}bt^3 + \cdots$$

where x is the position, v is the velocity, a is the acceleration, b is the jerk term, and t is time. Therefore, solving for the x term in the next time step is as follows:

$$x(t + \Delta t) = x(t) + v(t)\Delta t + \frac{1}{2}a(t)\Delta t^2 + \frac{1}{6}b(t)\Delta t^3 + \mathcal{O}(\Delta t^4)$$

This equation is complex, requiring knowledge of the current x, v, a etc. Moreover, since few people calculate the jerk term, the error is typically $\mathcal{O}(\Delta t^3)$. However, suppose we want to calculate the position of the previous timestep. The equation would be as follows:

$$x(t - \Delta t) = x(t) - v(t)\Delta t + \frac{1}{2}a(t)\Delta t^2 - \frac{1}{6}b(t)\Delta t^3 + \mathcal{O}(\Delta t^4)$$

Now adding the two equations together and rearranging for $x(t + \Delta t)$ yields:

$$x(t + \Delta t) = 2x(t) - x(t - \Delta t) + a(t)\Delta t^2 + \mathcal{O}(\Delta t^4)$$

This equation derives the next position of an object purely from the current position, the previous position, and the acceleration, leaving an error of only $\mathcal{O}(\Delta t^4)$. The previous position doesn't need to be solved, since we can simply record the position each frame, and use it in the next calculation.

In code, this is implemented as:

```
// Verlet Equation
Vector3 newPos = pos * 2 - prevPos + acc * deltaTime * deltaTime;
```

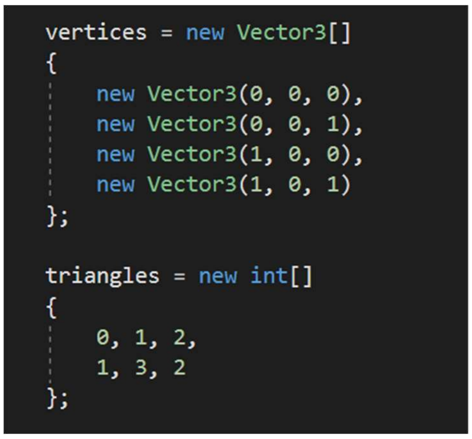
`deltaTime` is the change in time since the last calculation (i.e. the timestep). Originally, I used Unity's inbuilt `Time.deltaTime` attribute, which returns the time elapsed since the last frame. This has the benefit of matching the simulation speed to the performance, so that even at a lower framerate, the output would occur over the same timeframe and appear to run at the same speed. However, in the event of lag spikes during the simulation that would briefly raise the framerate, the simulation would briefly receive a very large value for Δt and lead the maths to explode, collapsing the simulation.

To counteract this, I changed Δt to be a constant initialised at runtime. This meant that the simulation became much more stable, and further allows for multiple integrations per frame. This could theoretically improve compute speed.

[7] Rendering

Visualising the simulation is just as important as creating the simulation itself. To do that, I defined a **mesh** in Unity. This would be used for the visual render for the grid. Unity meshes are defined by **vertices** and **triangles**, each stored in their own arrays. Vertices are defined by cartesian coordinates, while triangles are defined by the ordinal location of the points which comprise it in the array.

For example, a quadrilateral could be defined by its corners at (0, 0), (0, 1), (1, 0), (1, 1). There are two triangles in the shape, separated by a diagonal, and the points which make up the triangles correspond the coordinates at index 0, 1, 2 and 1, 2, 3 in the vertex array. Together, these arrays define the geometry of the mesh. Moreover, triangles in Unity are always drawn clockwise: this allows the renderer to identify and cull the back of the triangle, increasing performance. As such, an exemplary quadrilateral mesh is as follows:



```
vertices = new Vector3[]
{
    new Vector3(0, 0, 0),
    new Vector3(0, 0, 1),
    new Vector3(1, 0, 0),
    new Vector3(1, 0, 1)
};

triangles = new int[]
{
    0, 1, 2,
    1, 3, 2
};
```

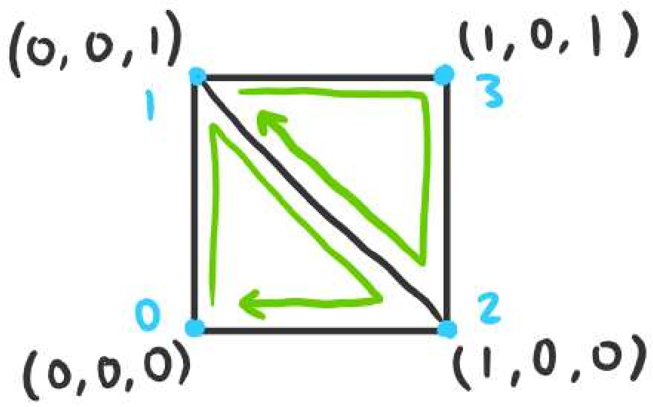


Fig 7.1: Quad defined by vertices and triangles arrays, with cartesian points (black), ordinal points (blue), and triangle cycles (green)

The vertices array is derived simply by iterating through the positions of each particle every frame.

Defining the triangles array was made easier by the fact that there are an equal number of σ springs, and therefore it is possible to use the same formula:

$$n_\sigma = 2(w-1)(h-1) = n_{triangles}$$

to create the array. I iterated through every square in the grid. Since there is one less link than particle in each row / column, going through each square is simply iterating through to $(\mathrm{w}-1)$ and $(\mathrm{h}-1)$ respectively. For each square, I then defined the two triangles which comprise it. Given the structuring of the array, adjacent points in the y axis differ by an index of 1, while adjacent points in the x direction differ by an index of h.

Every frame, the Unity mesh is updated with the new vertices array derived from the particle positions, and the mesh is re-rendered. The triangle array is unchanged for the same dimensions.

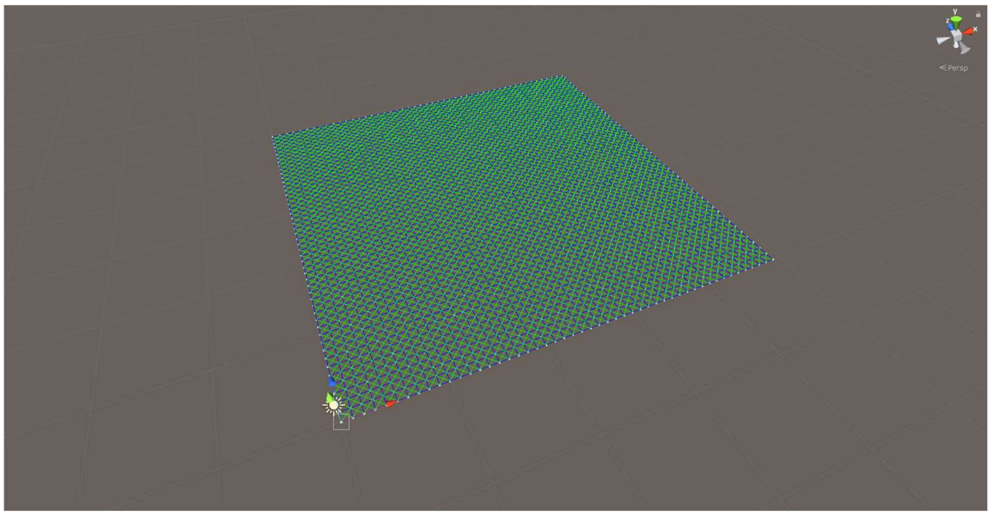


Fig. 7.2: A 50x50 grid of particles with χ (blue) and σ (green) springs; no mesh

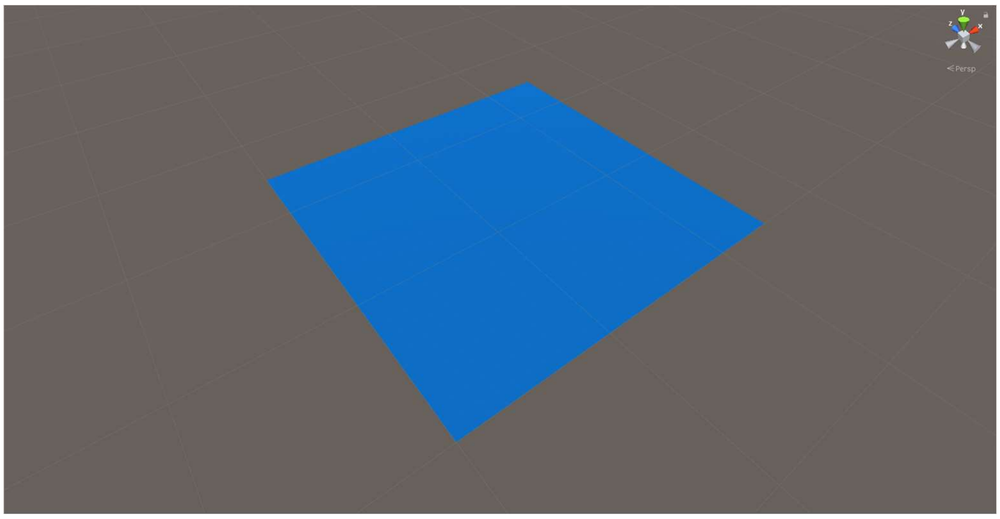


Fig. 7.3: Same grid but with a mesh renderer

Camera Controller

To ease visualising the mesh, I calculate a position for the camera looking over the mesh proportional to its size. Given a mesh with vector (x, y, z) representing its bounds, the centre of the mesh is identified with vector $(\frac{x}{2}, -\frac{y}{2}, \frac{z}{2})$ (The y component is negative as the cloth is defined flat the origin x plane, and thus when it falls the fabric gains a negative y component).

The camera is then translated backwards half the length of the mesh, translated up vertically, and rotated down to look at the centre of the mesh. Additionally, it can be made to rotate around the central y-axis of the mesh to deliver a panning shot.

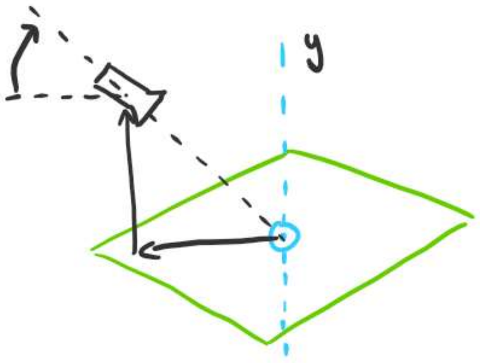


Fig. 7.4: Camera transformations (black) around mesh (green)

[8] Gallery

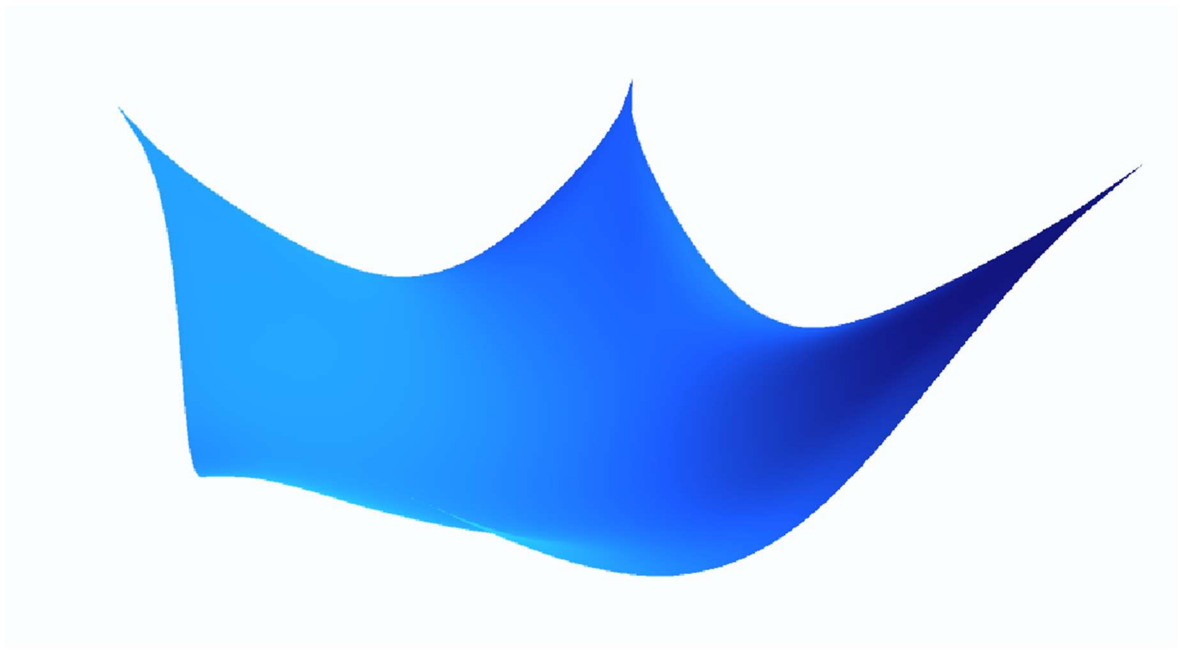


Fig 8.1: Cloth pinned at four corners

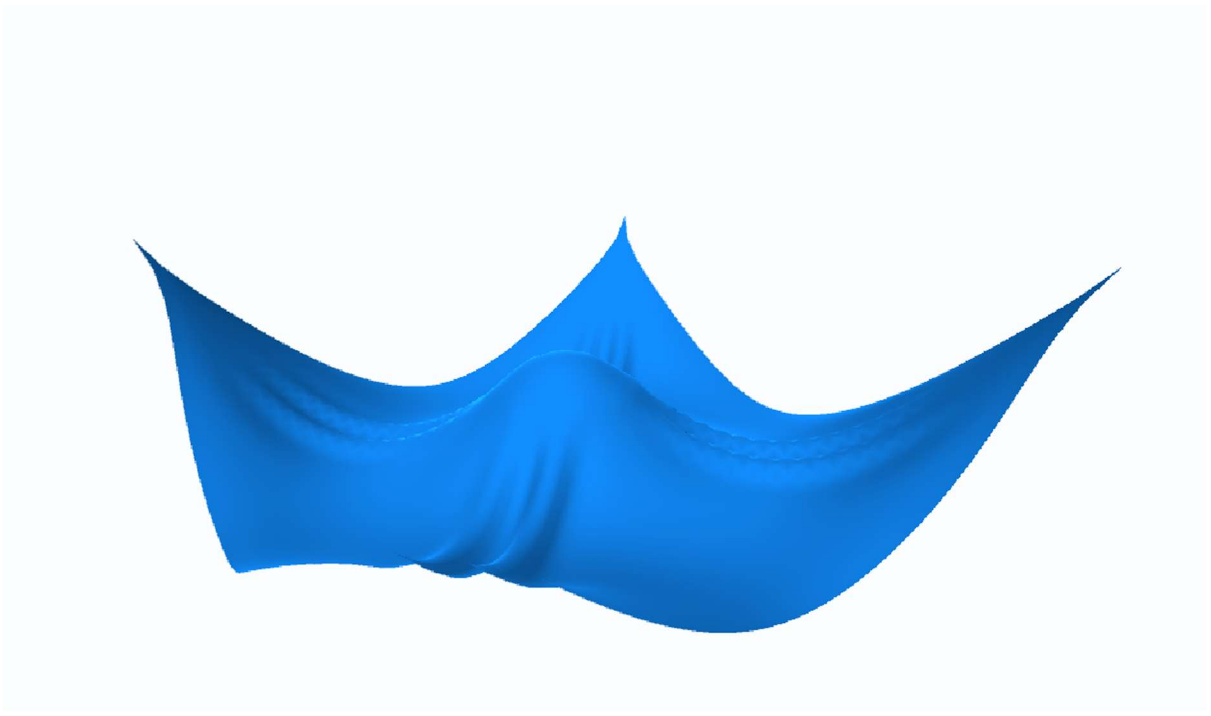


Fig 8.2: Cloth pinned at four corners draped over sphere

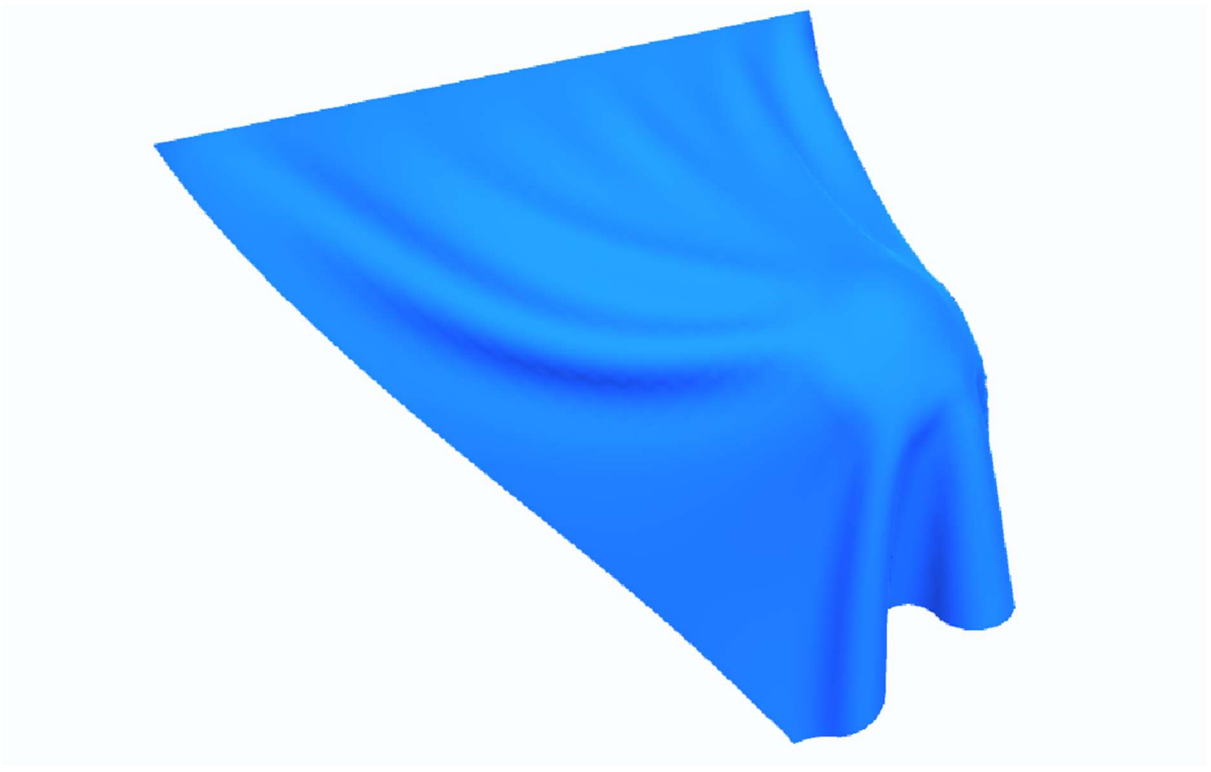


Fig 8.3: Cloth pinned along one edge draped over sphere

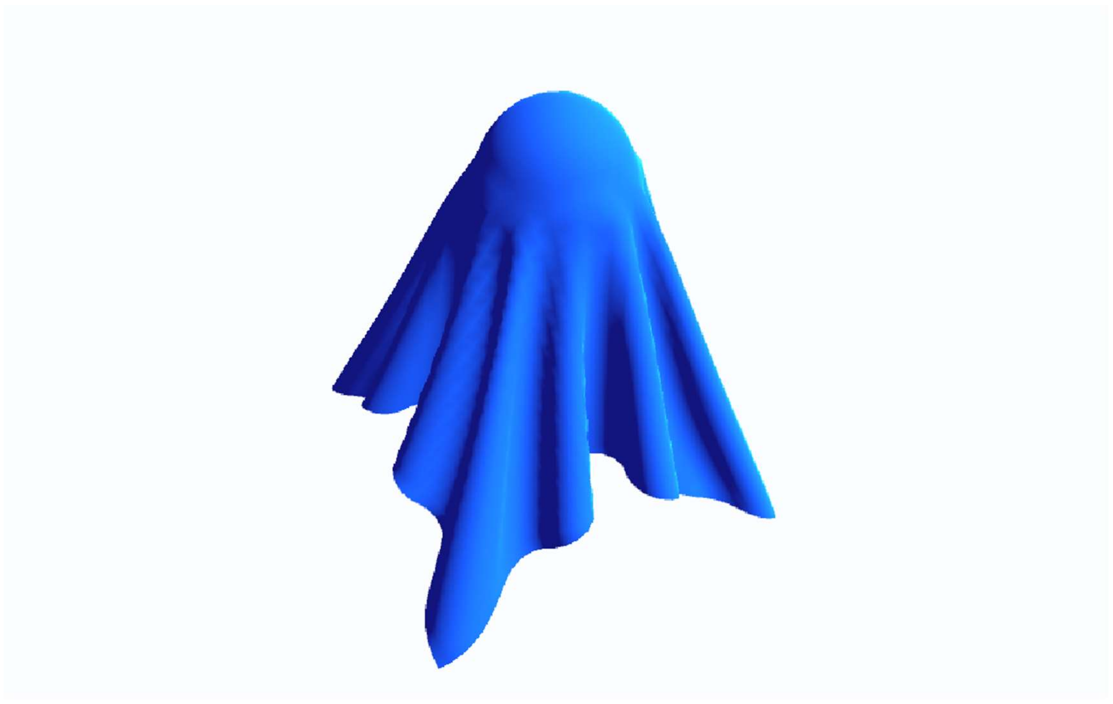


Fig 8.4: Cloth unpinned draped over sphere

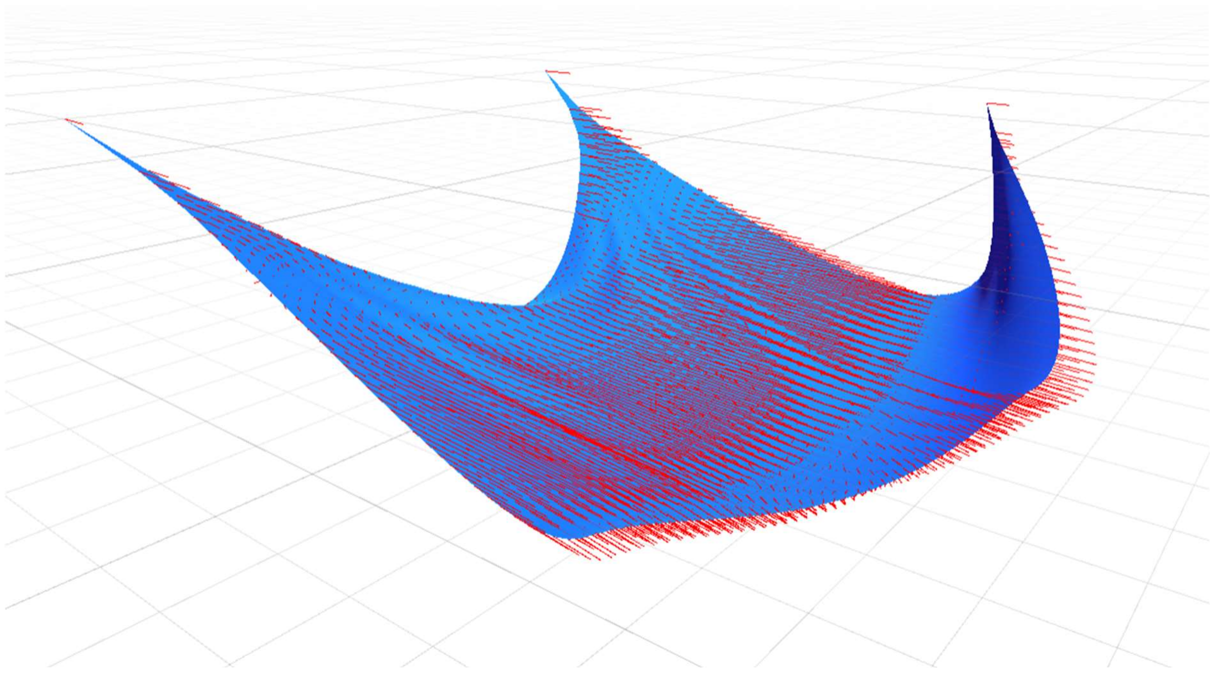


Fig 8.5: Cloth pinned at four corners with position-based wind vectors (red)

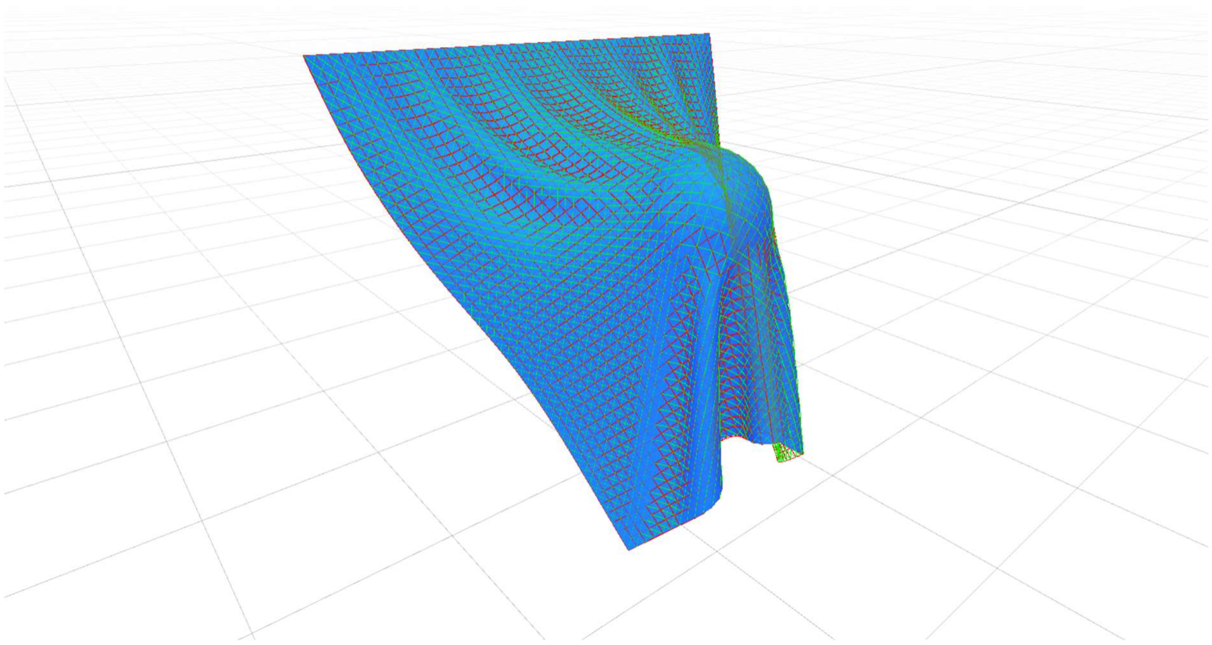


Fig 8.5: Cloth pinned along one edge draped over sphere with wireframe

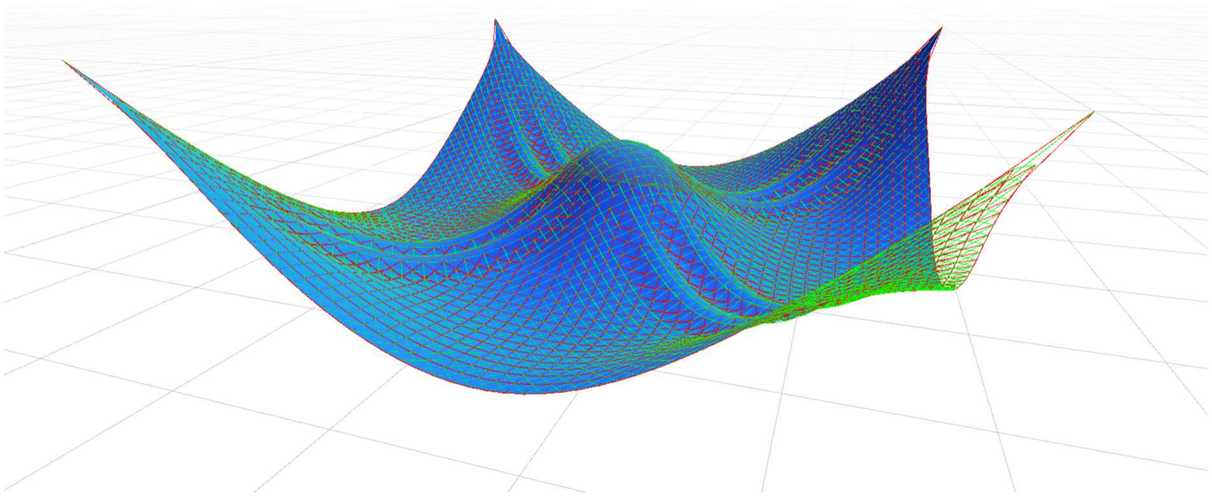


Fig 8.6: Cloth pinned at four corners draped over sphere with wireframe

[9] Class Documentation

Particle

Properties:		
Name	**Description**	**Type**
`pos`	Current position of the particle	`Vector3`
`prevPos`	Position of particle in previous time frame	`Vector3`
`m`	Mass of the particle	`float`
`g`	Gravitational constant	`float`
`d`	Damping constant	`float`
`f`	Sum of forces on particle	`Vector3`
`r`	Radius of particle (usually set to m for convenience)	`float`
`pinned`	Whether the particle is pinned in space	`bool`
`pinPos`	Position the particle is pinned	`Vector3`

Methods:			
Name	**Description**	**Arguments**	**Return**
`Particle`	Object constructor; defines internal attributes	`float M, float G, float D, float R, Vector3 position`	`void`
`applyForce`	Applies a force onto the particle by changing its acceleration according to Newton's 2nd Law	`Vector3 f`	`void`
`gravForce`	Applies force of gravity using f = ma; returns force	`none`	`Vector3`
`integratePosition`	Applies Verlet Integration to find the new position of the particle, using the change in time	`float deltaTime`	`void`
`checkCollision`	Uses vector arithmetic to work out collision with another particle	`Particle Q`	`void`
`pin`	Pins the particle to its current position	`none`	`void`
`unpin`	Unpins particle	`none`	`void`

Spring

Properties:		
Name	**Description**	**Type**
`origin`	The Particle that the base of the spring is connected to	`Particle`
`target`	The Particle at the other end of the spring	`Particle`
`k`	Stiffness constant	`float`
`r`	Radius constant	`float`

Methods:			
Name	**Description**	**Arguments**	**Return**
`Spring`	Object constructor; defines internal attributes	`float K, float R, Particle o`	`void`
`attach`	Sets the target Particle to t	`Particle t`	`void`
`applyConstraints`	Moves particles based on percentage extension to correct position	`none`	`void`
`applyHooke`	Applies Hooke's Law on connected particles and applies corresponding force	`none`	`void`
`drawSpring`	Used for visualising springs in editor	`none`	`void`

[10] Class Code

```
class Particle
{
    public Vector3 pos;
    public Vector3 prevPos;
    public float m;    //mass
    public float g;    //gravity
    public float d;    //damping
    public Vector3 f;    //forces
    public float r;    //radius;
    public bool pinned = false;
    public Vector3 pinPos;
    public Particle(float M, float G, float D, float R, Vector3 position)
    {
        pos = position;
        prevPos = position;
        m = M;
        g = G;
        d = D;
        r = R;
    }

    public Vector3 gravForce()
    {
        //F=ma in downwards direction
        return (g * m) * Vector3.down;
    }

    public void applyForce(Vector3 F)
    {
        //Add force to the sum of forces
        f += F;
    }

    public void integratePosition(float deltaTime)
    {
        //Verlet Integration
        if (!pinned)
        {
            //work out velocity for damping via change in position
            Vector3 v = (pos - prevPos) / deltaTime;
            //Newton 2
            Vector3 acc = f / m;
            //apply damping
            acc -= v * (d / m);
            //Verlet Equation
            Vector3 newPos = pos * 2 - prevPos + acc * deltaTime * deltaTime;
            //Set new prev pos
            prevPos = pos;
            //set pos
            pos = newPos;
        } else
        {
            pos = pinPos;
        }
        //reset forces
        f = Vector3.zero;

    }
    public void pin()
    {
        pinned = true;
        pinPos = pos;
    }

    public void unpin()
    {
        pinned = false;
    }

    public void checkCollision(Particle Q)
    {
        if (Q != this)
        {
            Vector3 offset = Q.pos - pos;
            if (offset.sqrMagnitude < r * r + Q.r * Q.r)
            {
                Vector3 offDir = offset.normalized;
                //find component of force in collision direction to find normal
                float product = Vector3.Dot(offDir, f);
                Vector3 normal = -offDir * (product / offDir.magnitude);
                applyForce(normal);
            }
        }
    }
}
```

```
class Spring
{
    public Particle origin;
    public Particle target;

    public float k;    //spring constant
    public float r;    //radius

    public Spring(float K, float R, Particle o)
    {
        k = K;
        r = R;
        origin = o;

    }

    public void attach(Particle t)
    {
        target = t;
    }

    public void applyHooke()
    {
        Vector3 offset = origin.pos - target.pos;
        //Find distance between particles
        float d = offset.magnitude;

        //Find extension
        float ext = d - r;
        //Hooke's Law and direction vector
        Vector3 f = ext * -k * offset.normalized;

        target.applyForce(-f); //pulling towards origin
        origin.applyForce(f); //pulling towards target
    }

    public void applyConstraints()
    {
        Vector3 offset = origin.pos - target.pos;

        float d = offset.magnitude;

        //Find extension
        float ext = d - r;
        //Find percentage extension
        float perc = (ext / d);
        //Divide by 2 since distance is applied to both particles
        perc /= 2;
        //Adjust particles
        origin.pos -= offset * perc;
        target.pos += offset * perc;
    }
}
```

[11] Diary

Week	Notes
1	Thought of project ideas. Thought that something with soft body mechanics would be fun. Looked into cloth modelling and the types of model. Researched existing models and found examples. Looked into visualisation programs for the project (e.g. Processing, Unity). Wrote up a plan for the project.
2	Created an initial grid in Processing to represent the particles. Struggled a little with grid geometry and indexing maths for arrays. Derived some formulae for the number of each type of spring in terms of the width and height. Created classes for the particles and links.
3	Implemented correct arrays for particles and springs. Used a basic constraint method to move the particles. Made some basic collisions with the floor. Moved the project to Unity. Researched render methods for meshes in Unity. Watched tutorials on vertex/triangle definitions and created a basic mesh. Redefined Particle and Spring classes in C# and defined arrays for both. Wrote functions for applying forces to particles and updating the position using the Verlet Algorithm and Unity's inbuilt Δt.
4	Corrected a mistake in my implementation of the Verlet, which doubled position vectors and led to a positive feedback loop. Added gravity via force addition. Wrote functions to define mesh vertex array from particle array and index triangles based on width and height. Struggled to implement Hooke's Law and applying the derived force. Eventually discovered that problem was due to incorrect signage and implemented the function. Spent time experimenting with the simulation to find values for e.g. k and d that would not lead to a positive feedback loop. Started working on collision.
5	Struggled to implement Hooke's Law and applying the derived force. Eventually discovered that problem was due to incorrect signage and implemented the function. Spent time experimenting with the simulation to find values for e.g. k and d that would not lead to a positive feedback loop. Started working on collision.
6	Implemented collision. Worked only with Constraint calculations but not with Hooke's Law. Rewrote the collision function; discovered that I wasn't normalizing a position vector in the equation to solve for the Normal Force. Normalizing the vector solved the problem; collisions work great with Hooke's Law. Positive feedback loops were still occurring; changed Δt to a constant which fixed the problem. Added basic UI to Unity application and linked it to parameters in the code. Added functions to pin the fabric in different ways, enable/disable collision with world object, change calculation mode etc. Wrote camera control functions to view the mesh in full.
7	Wrote up report in full. Built application.

Evaluation

Overall, the model provides an effective way of modelling the behaviour of cloth in real-time. It looks realistic, and interacts nicely with environmental forces (e.g. collision, wind). Effects on the cloth propagate across it accurately. Using the constraint method of calculating internal interaction was an effective way of making a realistic looking simulation, even if physically inaccurate. The Hooke method was physically accurate and generally produced realistic simulations.

The rendering of the cloth is effective, clearly showing how the cloth is behaving as a result of the model.

Unrealistic behaviour can occur for values of k which were less than d, or for too small values of k altogether. Moreover, because the collision system only checks for inter-object and not intra-object collision, when fully resting on a sphere the fabric can intersect itself and produce unrealistic behaviour (e.g. passing through the sphere; vertex values tending towards infinity). This is definitely an area to develop further in the future; I have started some code that uses spatial hashing to check for collisions.

References

Valid as of 23/2/2020

Example projects

https://steven.codes/blog/cloth-simulation/

https://graphics.stanford.edu/~mdfisher/cloth.html

https://journals.sagepub.com/doi/pdf/10.1177/0040517506057169

http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.127.8719&rep=rep1&type=pdf

Explanation on Verlet Integration

https://www.algorithm-archive.org/contents/verlet_integration/verlet_integration.html

Vector visualisation tool

https://academo.org/demos/vector-field-plotter/

Constraint example

http://datagenetics.com/blog/july22018/index.html

Explanation on Vector Maths

https://colalg.math.csusb.edu/~devel/IT/main/m08_vectors/src/s05_vectorprojection.html