



# Cloth Simulation

Rachel LaViola  
Corbin White  
Matthew Dias  
Aaron White

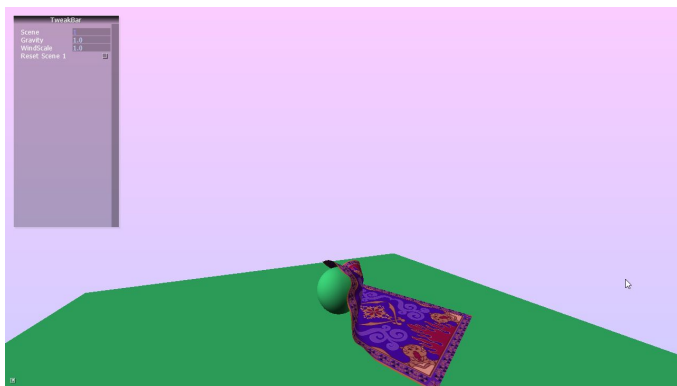
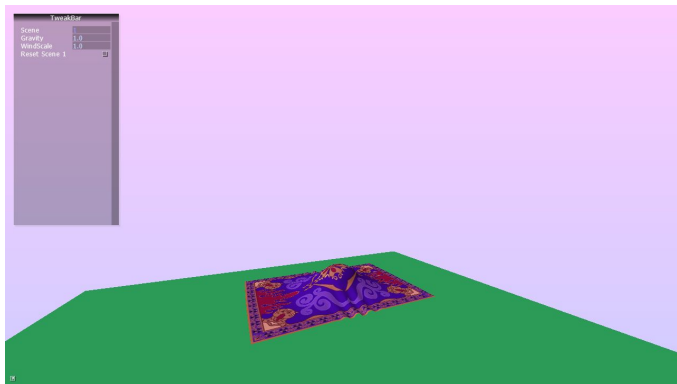


# Objective

The overarching goal of our project was to simulate the realistic motion of cloth using a spring-mass particle mesh. The particles interact with environmental forces as well as the forces that they exert upon on each other.

In pursuing this goal, we accomplished several interim goals, mainly:

- i. Familiarizing ourselves with the basic fundamentals of OpenGL programming.
- ii. Applying mathematical concepts relevant to cloth simulation, which include mass-spring models, kinematic equations, Hooke's law, and the Verlet algorithm.
- iii. Including a user-interactive element to our program; for instance, allowing the user to manipulate environmental properties.



Where We Are

<https://github.com/matthewdias/ClothSim>



*Demo*



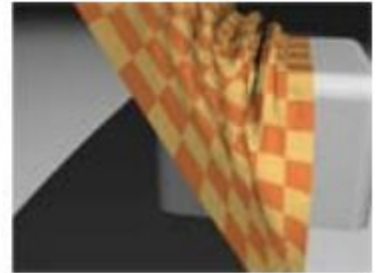
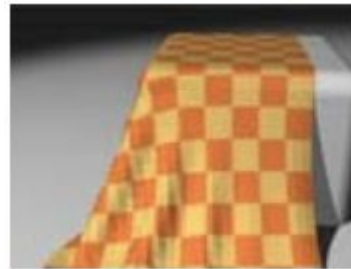
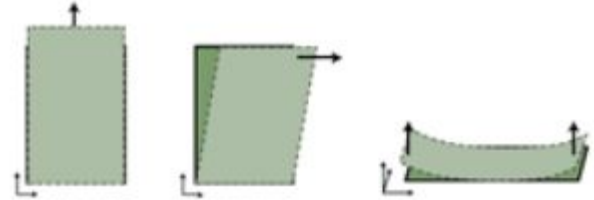
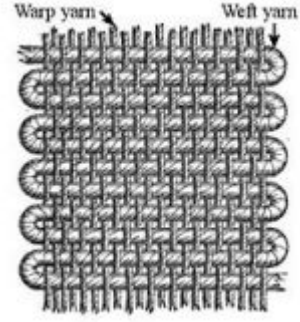
# Cloth

What is cloth?

- Flexible material consisting of a network of natural or artificial fibres.

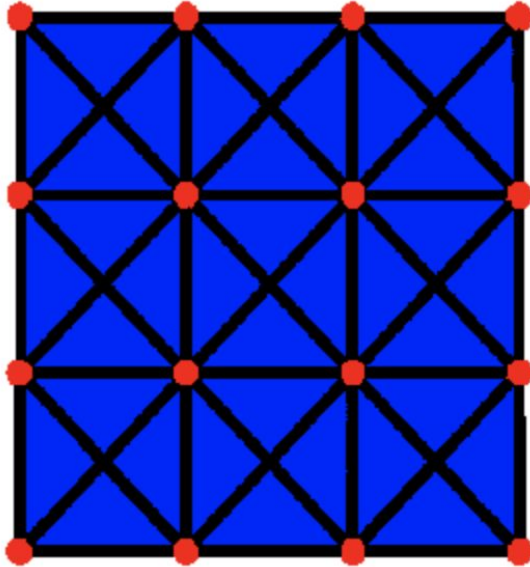
Properties:

- Stretch/Compression
- Shear (displacement along diagonal directions)
- Bend (curvature of cloth surface)
- Draping
- Wrinkling/Buckling



# Particle Mesh

We chose to create a lattice of springs in a grid structure to model deformable dynamic properties.



**Red dots** = Particles that store all of their own local information (mass, velocity, position, acceleration, etc.).

**Black lines** = Massless, spring connectors that serve as structural constraints.

# Particle Mesh

- **Good** for representing linear strain.
- **Bad** for capturing 2D/3D elastic behavior such as bending and shearing.

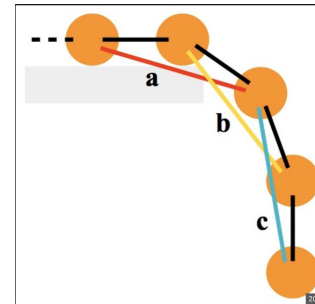
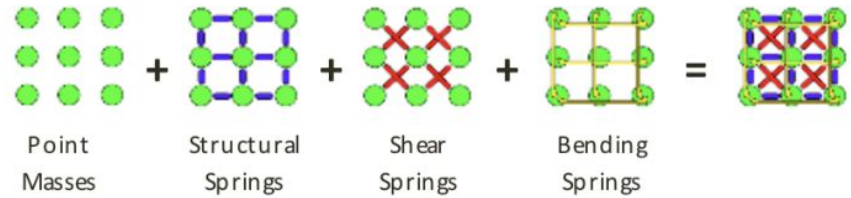
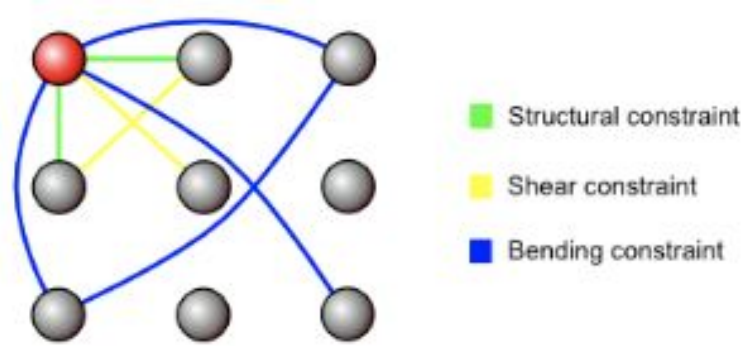
**Need another way to account for this!**

# Particle Mesh

**Structural Constraints:** Hold particles in a grid in the plane of the cloth.

**Shear Constraints:** Resist shearing in the plane of the cloth. Helps particles move in parallel of each other.

**Bending Constraints:** Resist bending of the material in an unrealistic ways.



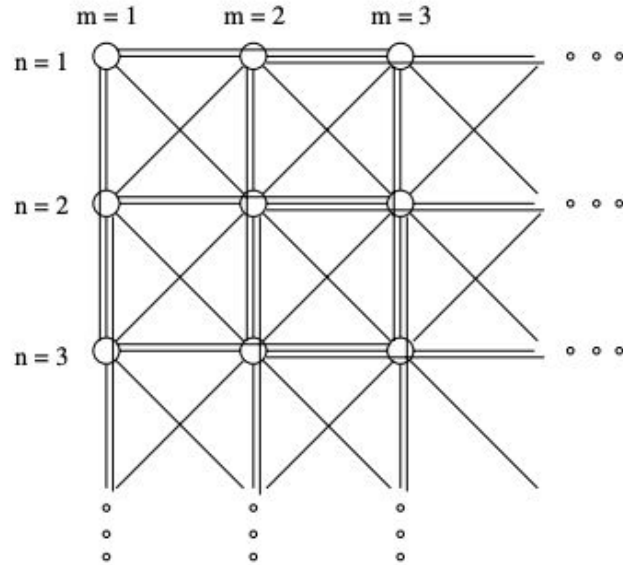


# Particle Mesh

**Structural Springs:** between  $[i, j]$  and  $[i+1, j]$  and between  $[i, j]$  and  $[i, j+1]$ .

**Shear Springs:** between  $[i, j]$  and  $[i+1, j+1]$  and between  $[i, j+1]$  and  $[i, j]$ .

**Bend Springs:** between  $[i, j]$  and  $[i+2, j]$  and between  $[i, j]$  and  $[i, j+2]$ .



mass : ○

spring : —

# Accumulation of Forces

How do we actually simulate the cloth movement?

For movement we need to sum up all forces that are acting upon the cloth.

$$F_{net} = Mg + F_{wind} - F_{Hookes}$$

# Accumulation of Forces

Gravity:

Our mass and gravity values are arbitrary values because of unit sizes in OpenGL

Mass = 1

Gravity = 0.075 Units/s<sup>2</sup>

# Accumulation of Forces

Wind:

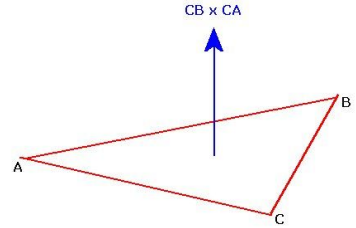
Wind is calculated per triangle rather than per particle.

We give a wind value, our flag example is using  $(2, 0, 0.1)$

We need to find the component of this wind that is in the same direction as our triangle's normal.

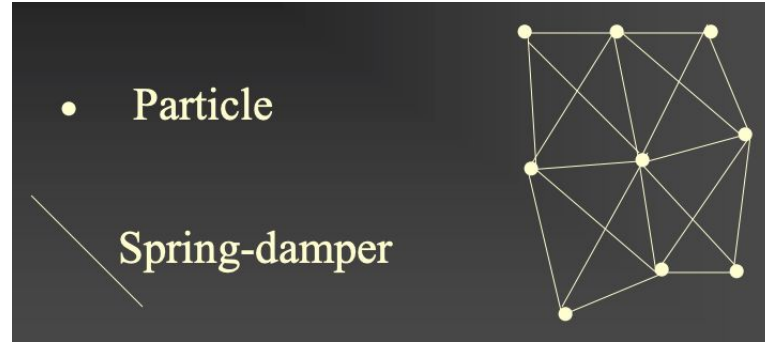
Dot product of our wind vector and our calculated normal for the triangle to produce a scalar value which represents the force on the triangle in that direction.

We can turn this back to a 3D vector by multiplying our scalar by the triangle's normal vector.



# Hooke's Law

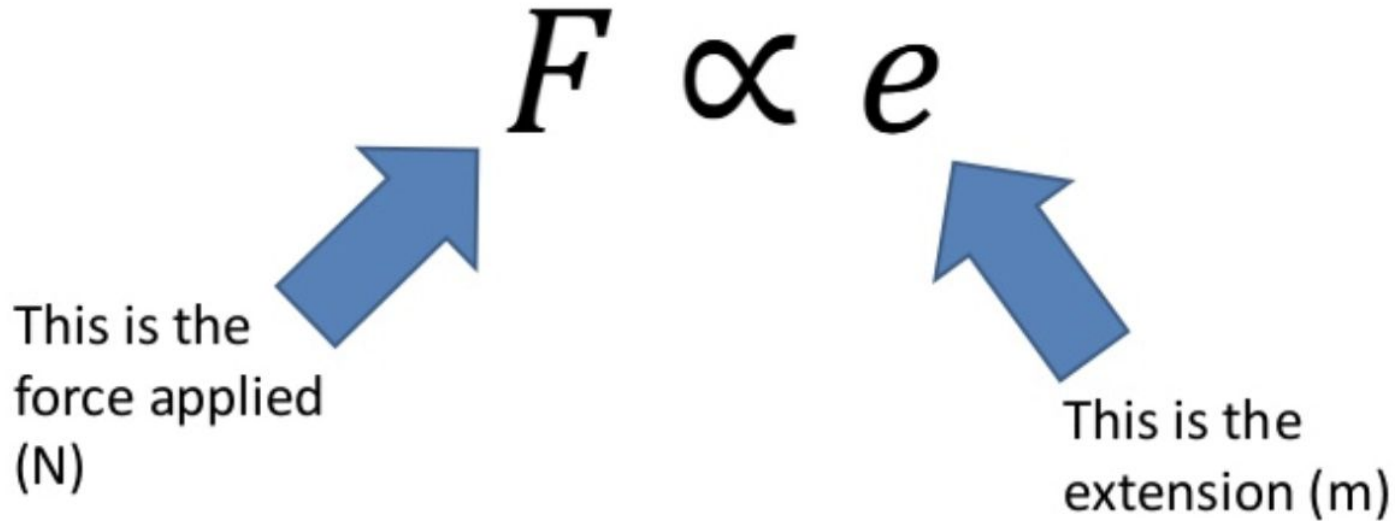
We treated the cloth as a system of particles interconnected by spring-dampers.



Each spring-damper connects two particles and generates a force upon them that is based on their positions and velocities.

# Hooke's Law

This force is calculated based off the concept of Hooke's law:



The diagram shows the equation  $F \propto e$  in a large, bold, black serif font. A blue arrow points from the text 'This is the force applied (N)' to the variable  $F$ . Another blue arrow points from the text 'This is the extension (m)' to the variable  $e$ .

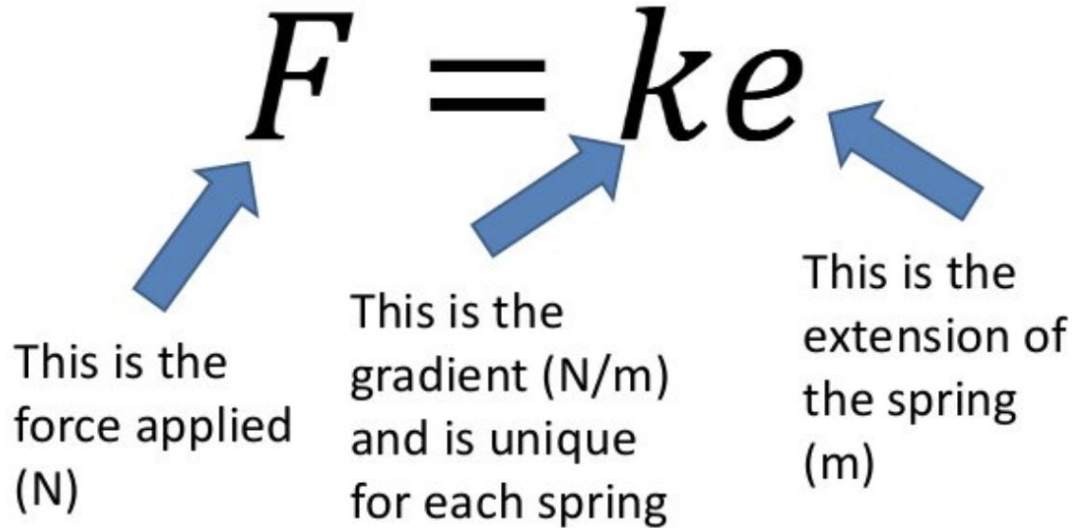
$$F \propto e$$

This is the force applied (N)

This is the extension (m)

# Hooke's Law

This force is calculated based off the concept of Hooke's law:



$F = ke$

This is the force applied (N)

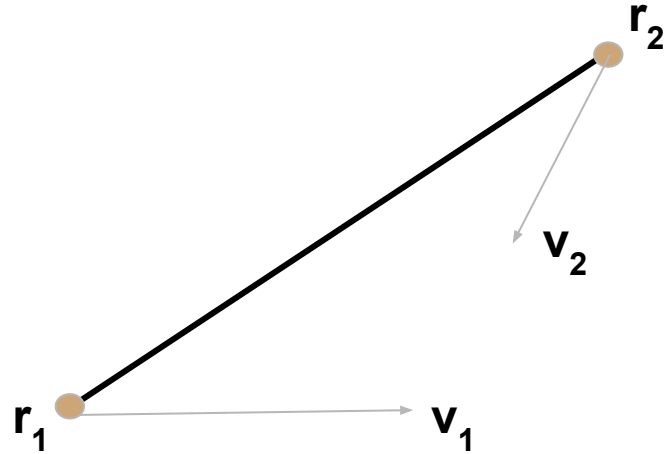
This is the gradient (N/m) and is unique for each spring

This is the extension of the spring (m)

# Hooke's Law

The basic spring-damper system connects two particles, each with their own velocities, and has three constants defining its behavior:

- Rest length:  $l_0$
- Spring constant:  $k_s$
- Damping factor:  $k_d$





# Hooke's Law

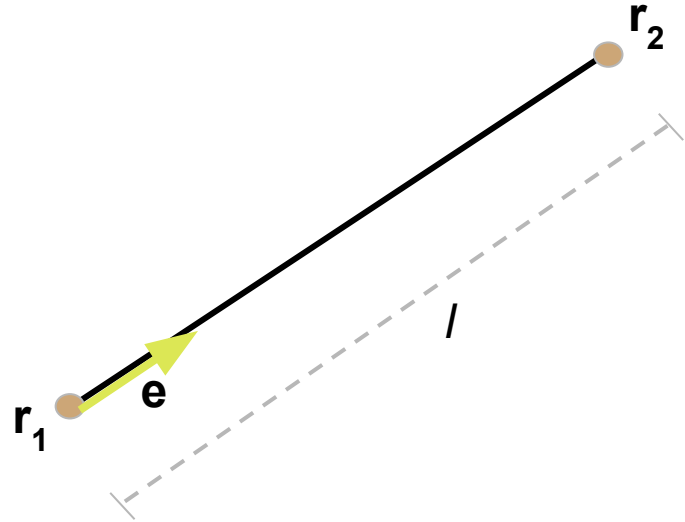
To compute the forces in 3D, we had to:

1. Turn 3D distances & velocities into 1D
2. Compute the spring force in 1D
3. Then turn 1D force back into 3D force

# Hooke's Law

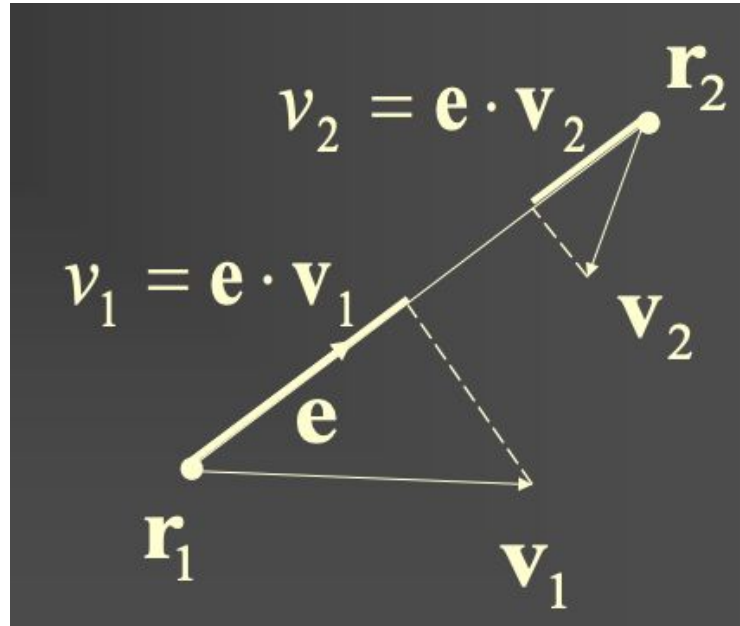
Start by computing the unit vector  $\mathbf{e}$  pointing from  $\mathbf{r}_1$  to  $\mathbf{r}_2$  and the distance  $l$  between the two points.

$$\begin{aligned}\mathbf{e}^* &= \mathbf{r}_2 - \mathbf{r}_1 \\ l &= |\mathbf{e}^*| \\ \mathbf{e} &= \frac{\mathbf{e}^*}{l}\end{aligned}$$



# Hooke's Law

Next, find the 1D velocities of each particle.



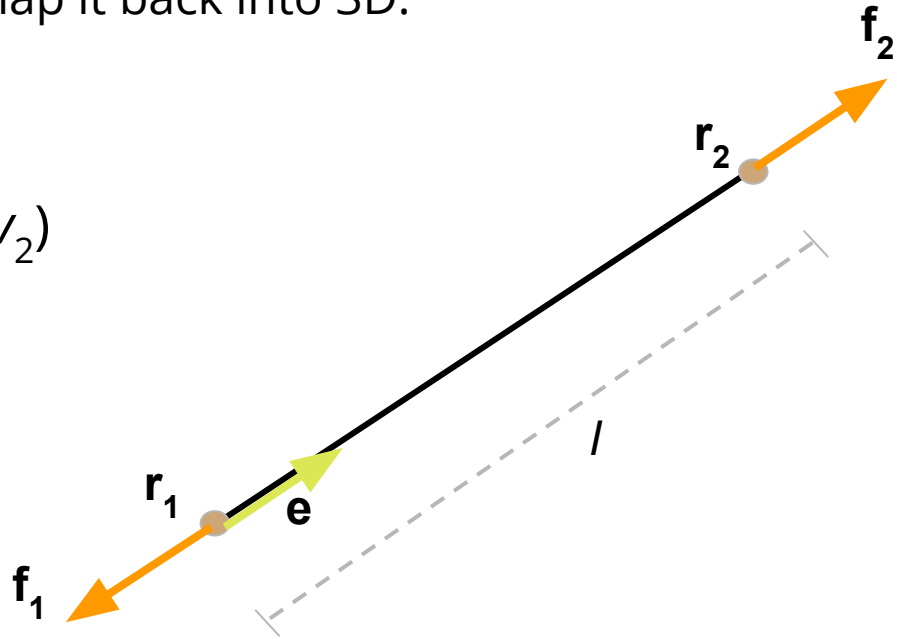
# Hooke's Law

Now find the 1D spring force and map it back into 3D.

$$f_{sd} = -k_s(l_0 - l) - k_d(v_1 - v_2)$$

$$\mathbf{f}_1 = f_{sd} \mathbf{e}$$

$$\mathbf{f}_2 = -\mathbf{f}_1$$



# Hooke's Law

Our implementation:

```
void Constraint::satisfyConstraint() {
    vmath::vec3 differenceVec = p2->getPos() - p1->getPos();
    vmath::vec3 normalDifferenceVec = vmath::normalize(differenceVec);
    float currDistance = (float)vmath::length(differenceVec);
    vmath::vec3 velocity1 = p1->pos - p1->prevPos;
    vmath::vec3 velocity2 = p2->pos - p2->prevPos;
    float v1 = vmath::dot(normalDifferenceVec, velocity1);
    float v2 = vmath::dot(normalDifferenceVec, velocity2);

    // Ks is the stiffness of the spring, when k gets bigger, the spring wants keep it's length
    float springForce = -0.1f * (restDistance - currDistance);
    float springDamp = -0.5f * (v1 - v2);
    float totalSpringForce = springForce + springDamp;
    vmath::vec3 f1 = totalSpringForce * normalDifferenceVec;

    p1->addForce(-f1);
    p2->addForce(f1);
}
```

# Acceleration

$$a = \frac{F_{net}}{M}$$

```
void Particle::addForce(vmath::vec3 givenForce) {  
    a += givenForce / mass;  
}
```

# Integration

We need a method to find how to change position after forces have been applied.

**Integrator** - formula for determining the next position for a point based on the current and/or past state.

**Explicit** - Only considers current state

- Easy to implement

**Implicit** - Uses current state and derivative at next state after a timestep

- Hard to implement

# Integration

## Euler's Method (Explicit)

- Determine next velocity using current velocity and timestep
- Determine next position using current position and computed velocity
- Explosive and inaccurate

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

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





# Integration

## Verlet Algorithm (Explicit)

- Don't need to track velocity
- Compute from previous and current position
- Much more accurate than Euler's
- Can easily be dampened to prevent explosion
- Used to calculate orbit of Halley's Comet in 1909

$$\mathbf{x}_{t+\Delta t} = 2 \mathbf{x}_t - \mathbf{x}_{t-\Delta t} + \left( \frac{d\mathbf{v}}{dt} \right)_t (\Delta t)^2$$



# Integration

## Implementation (Verlet)

```
pos = pos + (pos - prevPos) * (1.0 - dampening) + a *(t * t);
```

- `pos`: current position of particle
- `prevPos`: position of particle at previous timestep
- `dampening`: environmental constant to offset explosion
- `a`: acceleration vector incremented by accumulated forces
- `t`: timestep

# Constraint Satisfaction

Once you have moved the particles, there may be some violations that must be fixed.

The first being **Constraint Satisfaction**:

- The need to constrain the movement of particles so that they stay in a grid formation.

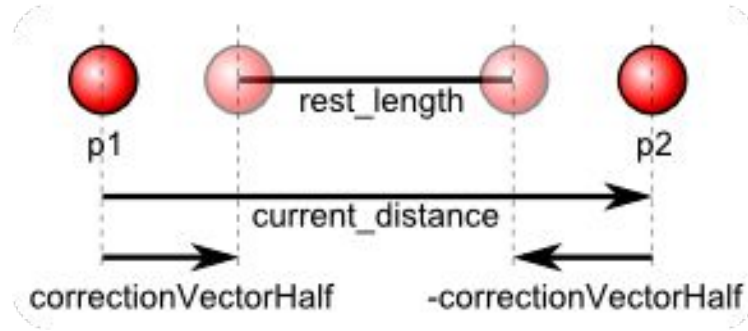
# Constraint Satisfaction

Each constraint between two particles has a distance that it would like to return to for the cloth to be at rest. We called that distance *restDistance*.

During verlet integration, our particles move around, resulting in some that are too far away from each other or too near to each other.

To remedy this, we must modify the positions of every two offending pairs of particles so that their distances are once again at rest-distance before we render the image.

# Constraint Satisfaction



Works by moving  $p1$  and  $p2$  along the line connecting them, so that they once again have a distance equal to *restDistance*.

To maintain symmetry, we move  $p1$  and  $p2$  by the same amount (either towards each other or away from each other to satisfy the constraint).

# Constraint Satisfaction

Our implementation:

```
std::vector<Constraint>::iterator c;  
for(int i=0; i<constraintIterations; i++)  
{  
    for(c=constraints.begin(); c != constraints.end(); c++)  
    {  
        vmath::vec3 differenceVec = (*c).p2->getPos() - (*c).p1->getPos();  
        float currDistance = (float)vmath::length(differenceVec);  
        vmath::vec3 correctionVec = differenceVec * (1 - (*c).restDistance/currDistance);  
        vmath::vec3 correctionVecHalf = correctionVec * 0.5;  
        (*c).p1->offsetPos(correctionVecHalf);  
        (*c).p2->offsetPos(-correctionVecHalf);  
    }  
}
```

# Collision Correction

When the cloth particles collide with a solid object, they must have their positions corrected.

There are two types of collision that we tested in our demo:

- Ball Collision
- Plane Collision

# Collision Correction

In ballCollision, we measure the distance between each particle and the center of the circle. We use this distance to calculate whether the particle is within the radius of the ball or not. If it is, we offset the particle's position.

```
245 void Cloth::ballCollision(const vmath::vec3 center, const float radius) {
246     std::vector<Particle>::iterator pIterator;
247     for (pIterator = particles.begin(); pIterator != particles.end(); pIterator++) {
248         vmath::vec3 distVector = (*pIterator).getPos() - center;
249         float distance = (float)vmath::length(distVector);
250
251         if (distance < radius) {
252             float offset = radius - distance;
253             vmath::vec3 n = vmath::normalize(distVector);
254             (*pIterator).offsetPos(n * offset);
255         }
256     }
257 }
```



# Collision Correction

In planeCollision, we take the y value of the plane and check each particle's y value to see if they are lower than the plane. If so, offset the particle in the y direction.

```
259 void Cloth::planeCollision(const float yValue) {
260     std::vector<Particle>::iterator pIterator;
261     for (pIterator = particles.begin(); pIterator != particles.end(); pIterator++) {
262         float dist = (*pIterator).getPos()[1] - yValue;
263
264         if ((*pIterator).getPos()[1] < yValue) {
265             float offset = -dist + 0.01;
266             vmath::vec3 n = vmath::normalize(vmath::vec3(0, 1, 0));
267             (*pIterator).offsetPos(n * offset);
268         }
269     }
270 }
```

# Problems We Encountered

- Cloth used to intersect the object.
  - Solved the problem by tuning of our gravity and spring force variables.
- Cloth occasionally intersects self.
  - Due to lack of bounding spheres surrounding individual cloth particles.
- Grappling with OpenGL textures.
  - Successfully loaded image into memory, adjusted for correct dimensions.

# References

- Dingliana, John. (2016). "14: Mass-Spring System & Cloth." College of Computer Science and Statistics at Trinity College Dublin, <https://www.scss.tcd.ie/~manzkem/CS7057/cs7057-1516-14-MassSpringSystems-mm.pdf>.
- Fisher, Matthew. "Matt's Webcorner." Matt's Webcorner - Cloth, [graphics.stanford.edu/~mdfisher/cloth.html](https://graphics.stanford.edu/~mdfisher/cloth.html).
- Gonzalez, Giancarlo and Wang, Muxuan. (n.d.). "Cloth Simulation with Discrete Mass-Spring and Particle System." Creative Coding at UC Santa Cruz, <https://creativecoding.soe.ucsc.edu/courses/cs488/finalprojects/cloth/cloth.pdf>.
- Mosegaard, Jesper. "Mosegaards Cloth Simulation Coding Tutorial." Visual Computing Lab, [viscomp.alexandra.dk/?p=147](https://viscomp.alexandra.dk/?p=147).
- Press, W. H.; Teukolsky, S. A.; Vetterling, W. T.; Flannery, B. P. (2007). "Section 17.4. Second-Order Conservative Equations". Numerical Recipes: The Art of Scientific Computing (3rd ed.). New York: Cambridge University Press. ISBN 978-0-521-88068-8.
- Provot, Xavier. (n.d.). "Deformation Constraints in a Mass-Spring Model to Describe Rigid Cloth Behavior." Institut National de Recherche en Informatique et Automatique, <http://graphics.stanford.edu/courses/cs468-02-winter/Papers/Rigidcloth.pdf>.
- Rotenberg, Steve. (2016). "Cloth Simulation." Computer Science and Engineering at UC San Diego, [https://cseweb.ucsd.edu/classes/sp16/cse169-a/slides/CSE169\\_11.pdf](https://cseweb.ucsd.edu/classes/sp16/cse169-a/slides/CSE169_11.pdf).
- Tutor Pace. (2015). "Hooke's Law – An Overview." <https://freeonlinetutoring.edublogs.org/2015/11/10/hookes-law-an-overview>.