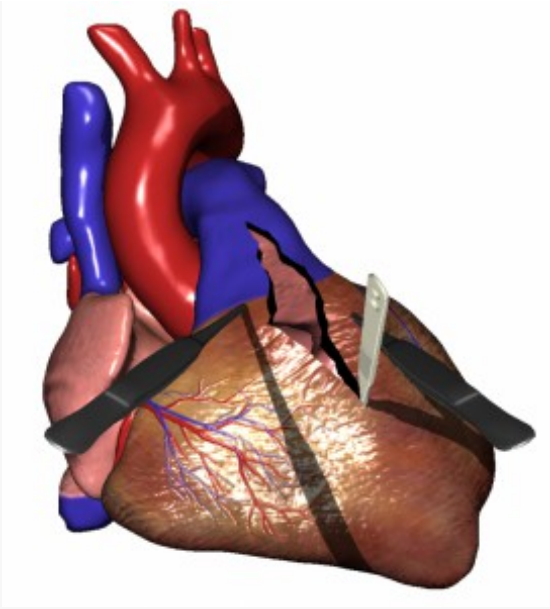
Motivation

I wrote my first cloth simulation in 2001 - and it was one of the most gratifying coding experiences I have had. The math involved can be easily understood intuitively, it does not take a long time to code, and the dynamic animation is (to me at least) an impressive emergent property of the very simple rules of particles of mass and interconnecting constraints. Furthermore I have found there to be many interesting applications of related techniques. One such example is the cardiac surgery simulator:



Cardiac Surgery Simulation - calculating tissue deformation

A cloth consists of particles and constraints

The simulation of cloth, is really about the simulation of particles with mass and interconnections, called constraints or springs, between these particles - think of them as fibers in the cloth. Particles move around in space due to forces that affect them - e.g. wind, gravity, or springs between particles.

Particles: from forces to positions

As a first step we would like to be able to affect particles with forces and calculate how much that shifts their position - within some small amount of time. To do that we need to use two pieces of knowledge.

1.Force can be "translated" into acceleration through Newtons second law: acceleration = force/mass
2.Acceleration can be "translated" into position by numerical integration - since position twice differentiated is equal to acceleration.

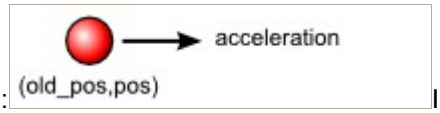
Item 1 is easy, and simply translates into the following method (on Particle class):

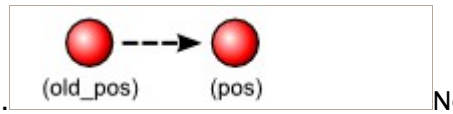
```
void addForce(Vec3 f)
{
        acceleration += f/mass;
}
```

If item number 2 does not make intuitively sense to you, then think about a car moving along a road. If you plot a graph of the distance traveled over time, the gradient at a given time is the velocity of

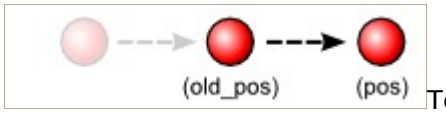
the car. If you plot the velocity on a graph and take the curvature at a given time you get the acceleration of your car (the change in velocity/speed). To go from acceleration to position we need to do the "inverse" of differentiation, namely integration. The integration routine we will use is called verlet integration. Pure verlet integation looks like this (but notice that we will modify it below to include damping in the source code)

```
Vec3 temp = pos;
pos = pos + (pos-old_pos) + acceleration*TIME_STEPSIZE2
old_pos = temp;
```

To understand this intuitively I like to think about the problem geometrically. Lets start with a particle that is not moving (pos and old_pos are the same). Lets say that we would like to push the particle by applying some acceleration vector: 

Intuitively, the particle should move in the direction of the acceleration vector. With that scenario in mind, the above verlet integration formula simply says to offset the current position by the acceleration vector. (The offset is scaled with the time step size to account for the fact that a particle should move a less amount when the piece of time is small, and vice-versa). In our current scenario the consequense is that the current position changes, and the old position stays the same. 

Now lets assume that we do not affect the particle with acceleration for the next time step. Intuitively, the particle should keep moving in the direction that we pushed it. The is realised through the term (old_pos-pos), a vector from old_pos to pos, in the verlet integration formula - implicitly representing a velocity vector. The next time step would consequently look like this:



To get to the final code used in the project we will include some damping (to heuristically account for gradual loss of velocity due to e.g. air resistance). An easy way to do is to simply multiply the velocity vector (old_pos-pos) with an amount between 0 and 1. Closer to zero makes the particle mover slower and slower each time step. An amount closer to one allows the particle to move unrestricted. Since we would like to specify damping, the amount that we will multiply by is (1.0-DAMPING). This writes out like so:

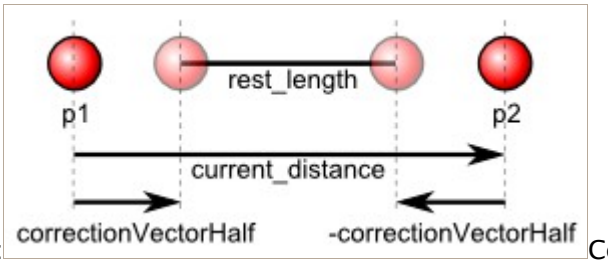
```
pos = pos + (pos-old_pos)*(1.0-DAMPING) + acceleration*TIME_STEPSIZE2
```

The final source code for our verlet integration method in the Particle Class is then:

```
void timeStep()
{
	if(movable)
	{
		Vec3 temp = pos;
		pos = pos + (pos-old_pos)*(1.0-DAMPING) + acceleration*TIME_STEPSIZE2;
		old_pos = temp;
		acceleration = Vec3(0,0,0); // acceleration is reset since it HAS been translated into a change in position (and implicitely into velocity)
	}
}
```

Constraints: move particles back into place !

Moving particles around is fine if you just want a simple particle system with no connections between them - but we want more! We would like to constrain the movement of particles, so that they try to stay in a grid. The more "effective" we are at returning all particles to the same relative position, the more rigid the cloth will behave. Notice that it is not an option to simply "reset" all particles to the original positon, since that contradicts our wish to affect the cloth with other forces such as wind and gravity. We will base our connections on constraints of distances between pairs of particles. Hence, each constraint between two particles has a distance that it would like to return to for the cloth to be at rest - we call that distance rest_distance. During the verlet integration, as explained above, particles move around, resulting in particles that are too far away from each other or too near each other. Therefor we introduce constraint satisfaction to modify the position of the two particles so that their distance is once again rest_distance. The following illustrations show two particles too far away from each other:

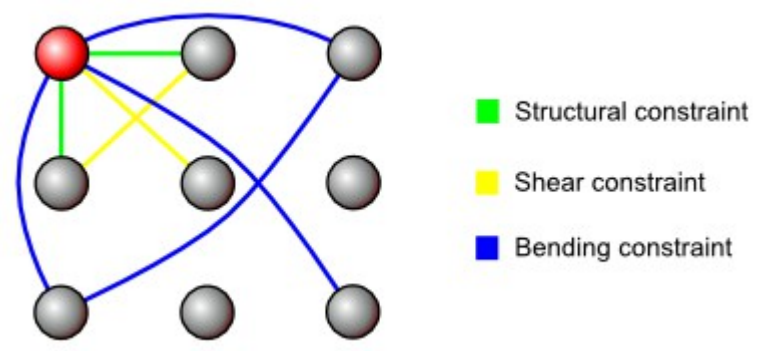


Constraint satisfaction works by moving p1 and p2 along the line connecting them, so that they once again have a distance equal to rest_length. To maintain symmetry we will move p1 and p2 by the same amount (either towards each other, or away from each other to satisfy the constraint). As it turns out, if we find the vector (we call it corrrectionVectorHalf) to move p1 by, we can use that same vector in opposite direction to move p2. Consider the vector from p1 to p2. By itself, it is too long an offset vector to be used on p1 (it would bring it all the way to p2). We will find a factor to multiply this vector with, to allow p1 to satisfy (half) of the constraint. That factor is exactly (1 - rest_distance/current_distance)*0.5. The 0.5 comes from the fact that we move p1 only half the way to satisfy the constraint (p2 moves the other half). The rest of the factor can be interpreted as finding the percentage of the whole vector p1 to p2 that is beyond rest_length - that is, we take 100%, and subtract rest_distance as a percentage of the whole distance. All in all the satisfyConstraint method on the Constraint class looks like this:

```
void satisfyConstraint()
{
        Vec3 p1_to_p2 = p2-&gt;getPos()-p1-&gt;getPos(); // vector from p1 to p2
        float current_distance = p1_to_p2.length();
        Vec3 correctionVector = p1_to_p2*(1 - rest_distance/current_distance);
        Vec3 correctionVectorHalf = correctionVector*0.5;
        p1-&gt;offsetPos(correctionVectorHalf);
        p2-&gt;offsetPos(-correctionVectorHalf);
}
```

The Cloth Class: Building the cloth

We will connect particles in a certain pattern to make the cloth behave realistically. The cloth should both stay in a grid in the plane of the cloth (structural constraints), resist shearing in the plane of the cloth (shear constraint), and resist bending (bending constraint). Below is illustrated how constraints are constructed for each particle (in red). Notice that the current (red) particle will end up having more constrains than shown since this pattern of constraints is repeated for each particle.



In the accompanying source code the build-up of the cloth connections is realized in the Cloth constructor using Particles and Constraints.

The Cloth Class: blowing in the wind, gravity and collision with a ball

To make the cloth have an interesting dynamic behavior we will add wind forces and gravity. We will also handle collision with a moving ball. Adding gravity is very easy, we simply add an acceleration vector pointing down to all particles (through the addForce method on the Cloth object). Collision detection and handling is also relatively easy, and I will not go into much detail. The ball is defined by a center and a radius. Detecting collision is done by asking whether the particle is within the radius of the ball, if yes the collision is resolved by moving the particle out of the ball along the vector from center to particle so that the distance is equal to the radius of the ball.

Adding wind is a bit more complicated, but results in a very nice effect. We will start by looking at the cloth as a collection of triangles of three particle each, and solve the problem of adding wind one triangle at a time - adding forces to the three particles the triangle consists of. The wind is coming in a certain direction, but forces should only be added in the direction of the normal of the triangle (or negative normal), and the amount of force affecting the triangle should be proportional to the angle (dot product) between the triangle and the wind. This all boils down to a simple formula for a single triangle:

```
void addWindForcesForTriangle(Particle *p1,Particle *p2,Particle *p3, const Vec3
direction)
{
        Vec3 normal = calcTriangleNormal(p1,p2,p3);
        Vec3 d = normal.normalized();
        Vec3 force = normal*(d.dot(direction));
        p1-&gt;addForce(force);
        p2-&gt;addForce(force);
        p3-&gt;addForce(force);
}
```

Conclusion

I hope that you have enjoyed this tutorial and the source code, and that it helped you in some way. Please leave a comment below or email me (clothTutorial {at} jespermosegaard(.)dk) - that would mean a lot to me !

Further Reading

I have knowingly skipped a lot of theory, given intuitive explanations instead of complete mathematical proofs, and focused exclusively on a single set of choices for the representation of a cloth simulation. There is a lot more to explore on the subject of physics based animation, and below are a few pointers.

My PhD on simulating heart surgery with GPU acceleration spring mass systems
Advanced Character Physics by Thomas Jakobsen
Cloth Simulation using Mass and Spring System