**Real-Time Balloon Simulation** 🎈

Annie Pa Michael Kelly

**Abstract**

We developed a real-time balloon simulation application using a mass-spring simulation model and explicit Euler integrator. Our application reads in a user-specified quad mesh .obj file and generates a balloon with structural, shear, and flexion springs between the vertices of the balloon. Our application also allows the user to inflate and deflate the balloon in real time. Optionally, the user can apply a vertex position correction algorithm to the vertices to constrain the balloon’s shape. Additionally, we implemented sphere-balloon collision detection so that the user

can throw spheres at the balloon.

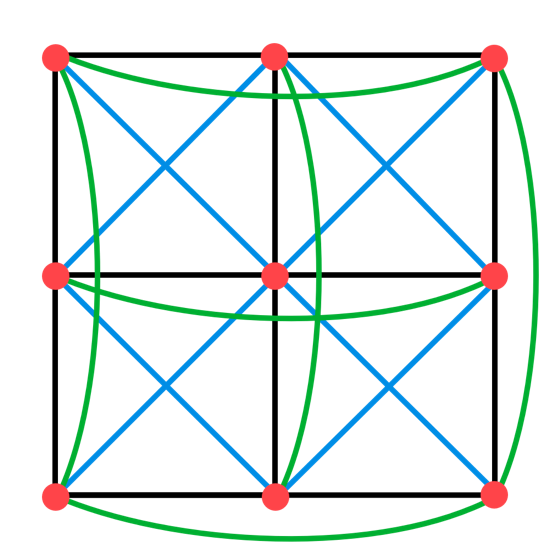


Figure 1: A standard cloth model. The red dots indicate cloth particles, the black lines indicate structural springs, the blue lines indicate shear springs, and the green lines indicate flexion springs.

**Introduction**

Mass-spring simulation is one of the simplest cloth simulation models. The simulation model works by creating particles with mass, generating springs to connect the particles, and then simulating the spring forces, as well as any gravity and damping forces on the particles. This simulation method leads to decently realistic results and runs in real time. We chose to use this model in our balloon simulation application because it is fast and easy to implement.

Similar to cloth, we use three types of springs in our balloon model: structural springs, shear springs, and flexion springs [Figure 1]. Our springs are linear Hookean springs. We also experimented with torsion springs, which are similar to flexion springs, but enforce an angle between two particles instead of a distance. However, we found that these springs had negligible impact on the balloon’s structure.

**Previous Work**

Some bullshit annie please do I don’t really want to but I will if you want.

**Technical Challenges**

Since we wanted our application to generate balloons out of arbitrary quad meshes, we could not make any assumptions about the topology of our balloons, specifically the valence of each vertex. As a result, we needed to write a .obj parser that would generate springs between an arbitrary number of faces.

Additionally, we had starter code that simulated cloth with Provot correction. However, this code assumed that cloth particles were laid out in a 2D grid and that the number of springs attached to each particle was the same for each particle. This meant we needed to do a bit of code refactoring to get the starter code to work.

Next, we had to determine how to simulate balloon inflation forces and what (if any) particle position correction should be applied to our balloon particles at each timestep.

Finally, we wanted the ability to throw spheres at our balloons, so we needed to implement collision detection and resolution.

**The Data Structures**

Balloon

Our Balloon data structure contains a vector of Faces, a vector of BalloonParticles, and a vector of Spheres. It also contains various constants used in the simulation code. Additionally, each Balloon can be attached to a string, so we include the position of the bottom of the spring and the ID of the BalloonParticle attached to the string.

Face

Each Face struct contains the IDs of the BalloonParticles in the face. The IDs of BalloonParticles are calculated based on the order they appear in the .obj file. Faces also contain a normal and area.

BalloonParticle

The BalloonParticle struct contains the original position, position, velocity, acceleration, and mass of the particle. Each particle also stores the IDs of the Faces in the balloon that the particle is a part of and the IDs of the particles that are in the Faces that the particle is a part of. This allows us to access the local neighborhood of faces and particles around a given particle. This data is used to generate the springs. Speaking of springs, BalloonParticles store vectors of structural, shear, and flexion springs attached to the particle.

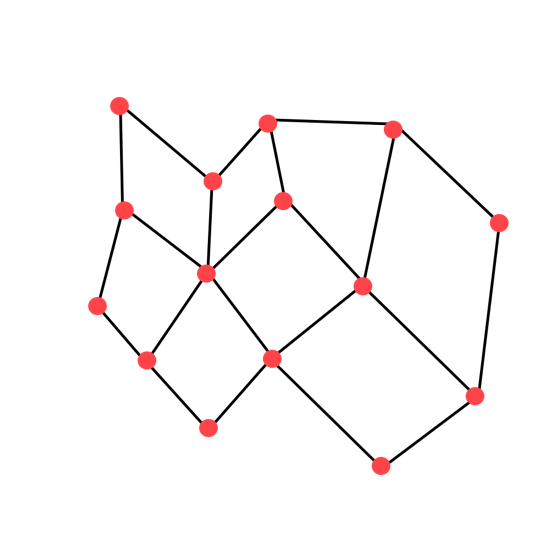


Figure 2: An example quad mesh.

Since we need the particle’s normal for shading and for the inflation simulation, we cache this normal in the particle and use it in the rendering code and simulation code. Specifically, the normal used in the simulation is a cached normal from the previous render frame. Since we complete multiple simulation timesteps before rendering a new frame, it is highly likely that the truly correct particle normal diverges from the cached normal as the simulation progresses. However, we assume that the timesteps are small enough that the cached normal and correct normal are close enough that the difference is negligible.

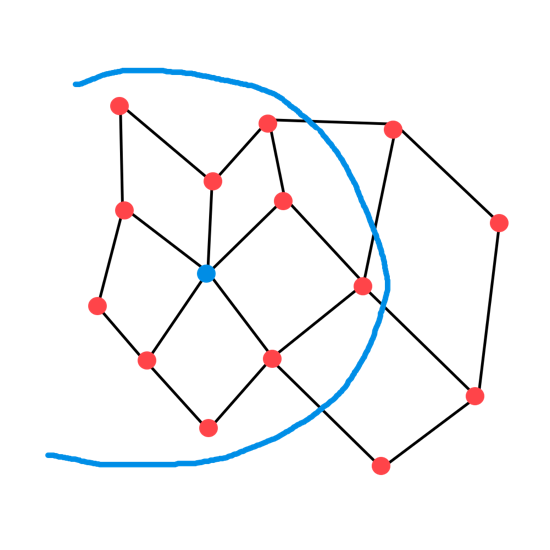


Figure 3: The blue vertex is the selected vertex and the vertices in the blue circle are the nearest vertices.

Spring

Springs store pointers to two BalloonParticles, as well as a spring constant.

Sphere

Spheres contain a mass, radius, position, velocity, and acceleration.

**.obj Loader and Spring Generation**

The first half of the .obj loader reads in the vertices and faces and turns the vertices into BalloonParticles and faces into Face structs. It also finds the nearest particles and nearest faces for each BalloonParticle.

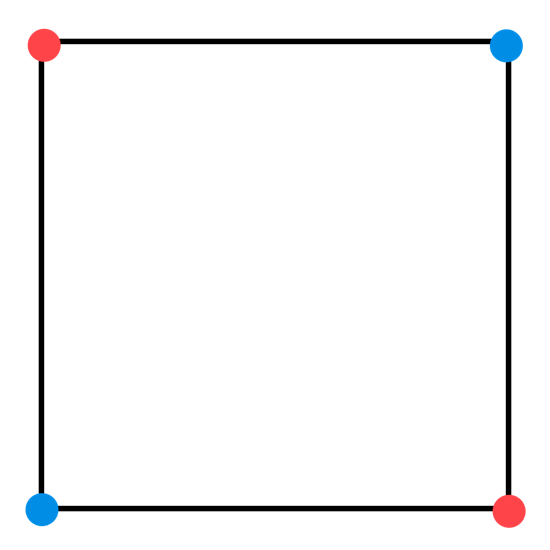


Figure 4: Red particles are across from each other in the face, and red particles are next to blue particles in the face.

The spring generation code is slightly more complex. In [Figure 2], we see an example quad mesh. In [Figure 3], we see a selected vertex, highlighted in blue, and the vertices in the local neighborhood, contained in a blue circle. We can create structural springs between vertices in the local neighborhood that are connected to the blue particle by an edge/black line and create shear springs between vertices that are not connected to the blue particle by an edge/black line. Some pseudocode is below.

Structural and Shear Spring creation

for each particle p:

for each p.nearest\_particles p2:

for each p.nearest\_faces f:

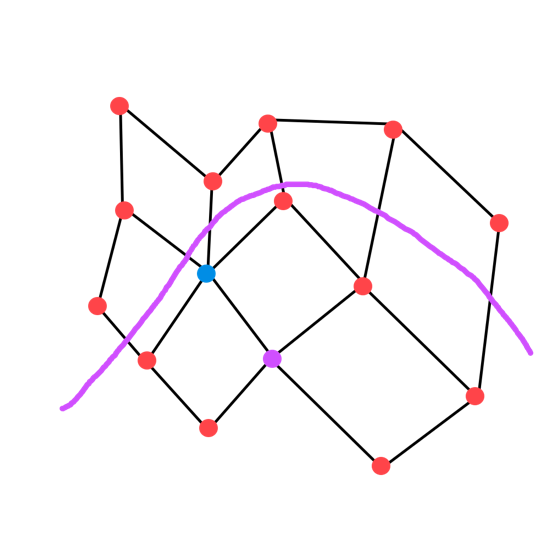


Figure 5: The purple vertex is the blue vertex’s neighbor. Vertices under the purple line are in the purple vertex’s nearest vertices.

if f.shouldCreateStructualSpring(p, p2):

Create structural spring between p and p2

Add spring to p

if f.shouldCreateShearSpring(p, p2):

Create shear spring between p and p2

Add spring to p

shouldCreateStructuralSpring(p, p2)

if this.containsParticles(p, p2):

if p and p2 are adjacent to each other in the face: [Figure 4]

return true

return false

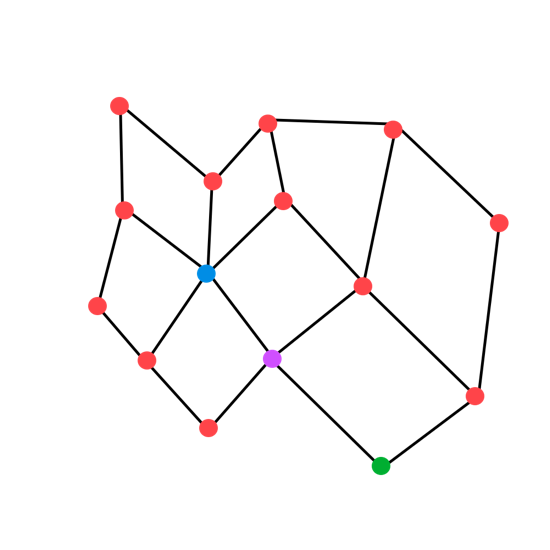


Figure 6: A flexion spring would be generated between the blue and green vertices.

shouldCreateShearSpring(p, p2)

if this.containsParticles(p, p2):

if p and p2 are across from each other in the face: [Figure 4]

return true

return false

Flexion spring creation is more complicated. Flexion springs are created between particles have one particle in between them that they are both connected to [Figures 1 and 6]. In order to generate flexion springs, we take a test particle, find that particle’s neighbors, then for each neighbor particle that generates a structural spring with the test particle, we search the neighbor particle’s neighborhood for a third particle that creates a structural spring between the neighbor particle and the third particle (the neighbor’s neighbor of our test particle). The neighbor’s neighbor particle and our test particle have a flexion spring between them. However, there are a few edge cases we need to account for. In [Figure 7], we see a potential particle combination from our algorithm. The blue and green particles do have one particle between them, but they are a part of the same face, and the blue and green particles should create a shear spring, not a flexion spring. So, before creating a flexion spring between two particles, we must check and ensure they do not create a different type of spring.

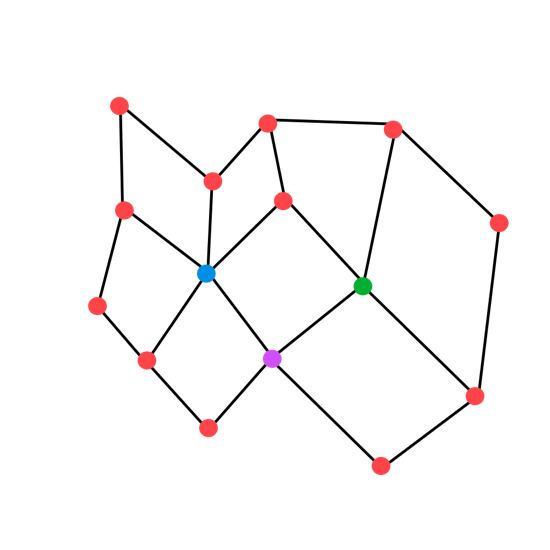


Figure 7: A flexion spring would not be created between the green and blue vertices, since they should create a shear spring.

Flexion Spring creation

for each particle p:

for each p.nearest\_particles p2:

for each p.nearest\_faces f:

if f.shouldCreateStructualSpring(p, p2):

for each p2.nearest\_particles p3:

for each p3.nearest\_faces f2:

if f2.shouldCreateStructuralSpring(p2, p3):

if (p and p3 don’t create a structural or shear  
 spring and p != p3):

Create flexion spring between p and p3

Add spring to p

**Particle Simulation**

Annie

**Particle Position Correction**

Annie

**Collision Detection and Resolution**

After new particle positions, velocities, and accelerations are calculated, and particle correction is optionally applied, we detect and resolve collisions. Our application currently only handles balloon-sphere collisions, since this collision type is relatively inexpensive. Thus, our collision detection algorithm is straightforward.

for each sphere s:

for each balloon particle p:

if distance(p.position, s.center) < s.radius:

Move p outside the sphere

Apply a penalty force to s

Calculate new acceleration, velocity, and position for s

When moving balloon particles outside of a sphere, we move it along the direction from the sphere’s center to the particle’s position. The penalty force is applied in the opposite direction of the particle’s movement and is proportional to how inflated the balloon is. While this penalty force calculation is not physically accurate, it does give convincing results, especially for a real-time application. By varying the mass of a collision sphere and the inflation of a balloon, we can achieve different collision scenarios.

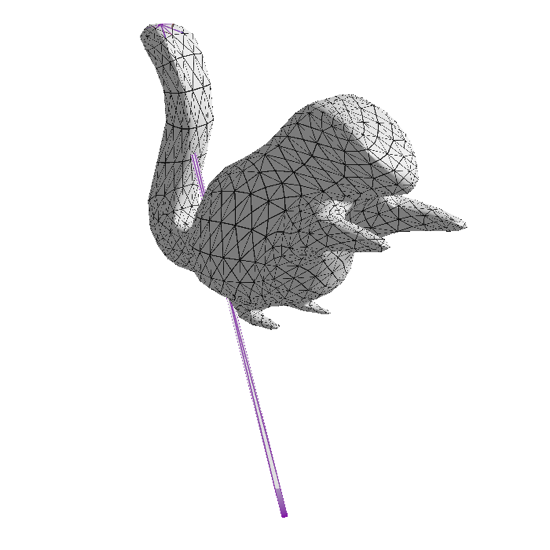


Figure 6: A string is attached to the ground and the very end of the squirrel’s tail.

**INSERT BOWL BALLOON FIGURES MICHAEL**

**Additional Examples**

In computer graphics, researchers often use the same models for demonstrations. Some popular models are the Stanford bunny and the Stanford armadillo. However, for this project, we propose adding a new animal friend to the collection of example models: the RPI squirrel. We chose to use this model because it was readily available as a quad mesh, which is necessary for our application. Additionally, we enjoyed seeing the squirrel get turned into a balloon.

In first demo [Figures 6, 7, and 8], we have the squirrel has its tail attached to a string that is attached to the ground. As the simulation progresses, the squirrel balloon successfully turns upside down, as would be expected in real life.

In the second demo [Figures 9 and 10], we drop four spheres of equal mass on to a pillow-shaped balloon. The balloons bounce off the top of the balloon.

In the third demo [Figures 11 and 12], we see how our balloon simulation handles concave surfaces. The balloon is initially shaped like a bowl, but as it is inflated, it takes on the shape of a sphere.

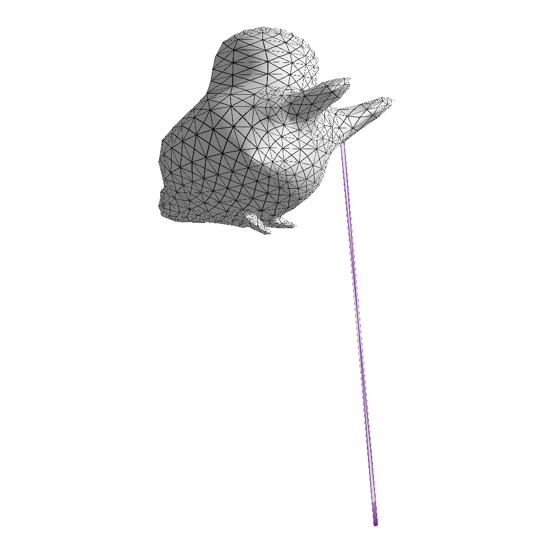


Figure 7: Squirrel is turning upside down.

**Discussion**

Talk about experimentation, played with real provot correction, also messed with torsional springs but they didn’t really do anything, don’t really need correction on lower poly objects. Might not even want to use provot correction since balloons are stretchy.

**Conclusion**

a

**Special Thanks**

We would like to thank Professor Barb Cutler for her mentorship and for providing starter code for our application.

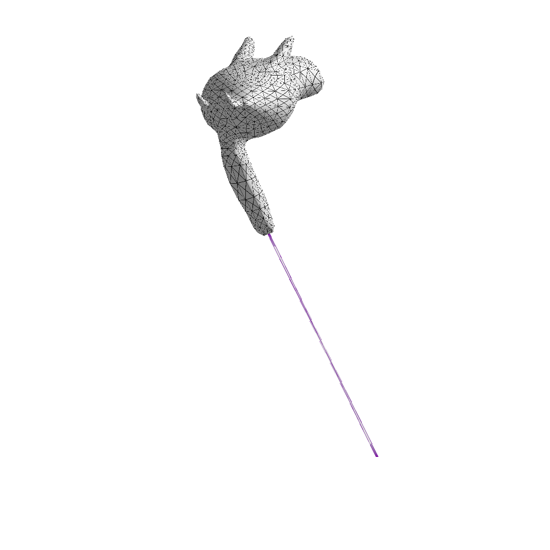


Figure 8: Our squirrel has successfully turned upside down. Hooray.

**References**

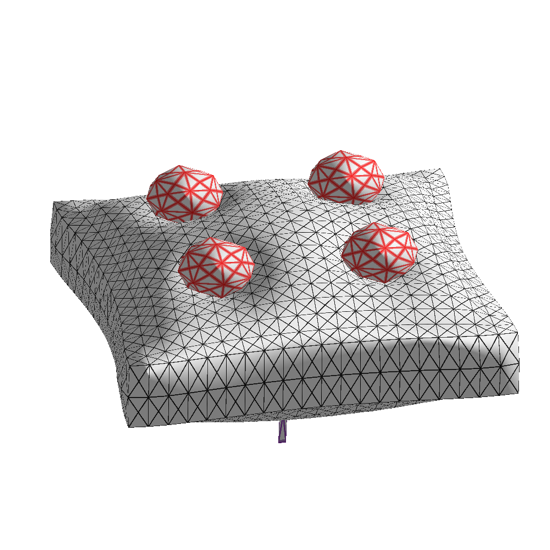


Figure 9: Four spheres colliding with a pillow-shaped balloon.