**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 have implemented simple sphere-balloon collision detection so that the user can throw spheres at the balloon.

**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.

Talk about structural shear and flexion springs and include a picture in the margin. Maybe Michael will make a pretty picture.

**Previous Work**

In “Semi-Realistic Balloon Simulation”, Tarantino attempts to simulate balloons with a mass-spring system. Tarantino does not use various spring types (structural, shear, flexion, etc), but does vary the spring constants. However, Tarantino does model a viscosity force. The balloon model does not include any Provot correction, but does include a correction where if a spring is overextended, the simulation stops applying new forces to the spring (Tarantino, 1996). We investigate using Tarantino’s correction method with interesting results.

We draw inspiration for our mass-spring system with “Fast Simulation of Mass-Spring Systems”. In this paper, they approach how to simulate cloth and mass spring systems with a simpler model and less calculations to make the system faster. The biggest difference with the algorithm is how spring forces are calculated, instead of Provot correction and collecting sums of forces, it’s just done with an optimized reduction of Hooke’s law, which reduces the amount of calculations done per particle greatly (Liu et al., 2013). However, they mention that their implementation does not really take into account all 3 kinds of springs in traditional cloth simulation, which results in a less-faithful simulation of cloth. We made the decision to keep the 3 springs and cached the springs for each mass to speed up our calculations.

With the advantage of simpler spring calculations, we can fit in buoyancy calculations. The approach that Jinwook Kim and his colleagues proposed in their paper. Their algorithm is actually a bit more complicated than we need because our first goal is to have a traditionally shaped balloon to work first, but their approach is quite clever. It takes advantage of the rendered geometry and uses a “slice” of it to and approximates how mass is distributed within it to calculate how it should bounce in water (Kim et al., 2006). We ultimately did not approach this but it did lend insight on adding an even force in one direction with respect to amount of surface area to have better simulated buoyancy.

**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.

Since we need the particle’s normal for shading and for the inflation simulation, we cache this normal in the particle and use it A close up of a logo

Description automatically generatedin 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.

Spring

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

Figure 1: An example quad mesh.

Sphere

A close up of a red light

Description automatically generated**.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.

The spring generation code is slightly more complex. In Figure 1, we see an example quad mesh. In Figure 2, 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.

Figure 2: The blue vertex is the selected vertex and the vertices in the blue circle are the nearest vertices.

Structural and Shear Spring creation

for each particle p:

for each p.nearest\_particles p2:

for each p.nearest\_faces f:

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:

**A close up of a light

Description automatically generated** return true

return false

shouldCreateShearSpring(p, p2)

if this.containsParticles(p, p2):

if p and p2 are across from each other in the face:

return true

return false

Figure 3: The purple vertex is the blue vertex’s neighbor. Vertices under the purple line are in the purple vertex’s nearest vertices.

Flexion Spring creation

for each particle p:

for each p.nearest\_particles p2:

for each p.nearest\_faces f:

if f.shouldCreateStructualSpring(p, p2):

A picture containing indoor, person, sitting

Description automatically generatedfor 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**

Figure 4: A flexion spring would be generated between the blue and green vertices.

**Particle Position Correction**

Annie

A close up of a red light

Description automatically generated

**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:

Figure 5: A flexion spring would not be created between the green and blue vertices, since they should create a shear spring.

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.

**INSERT BOWL BALLOON FIGURES MICHAEL**

**Examples**

Squirrel turning upside down, bowl balloon inflating,

**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**

annie

**Special Thanks**

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

**References**

1. Tarantino, Paul. “Semi-Realistic Balloon Simulation.” alumni.soe.ucsc.edu/~pault/262paper/262paper.pdf.

2. Liu, Tiantian, et al. “Fast Simulation of Mass-Spring Systems.” ACM Transactions on Graphics, vol. 32, no. 6, 2013, pp. 1–7., doi:10.1145/2508363.2508406.

3. Kim, Jinwook, et al. “Fast GPU Computation of the Mass Properties of a General Shape and Its Application to Buoyancy Simulation.” The Visual Computer, vol. 22, no. 9-11, 2006, pp. 856–864., doi:10.1007/s00371-006-0071-x.