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.

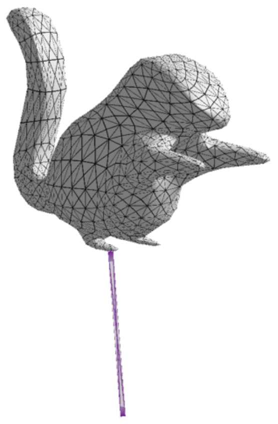
Figure 1: An initially deflated squirrel balloon.

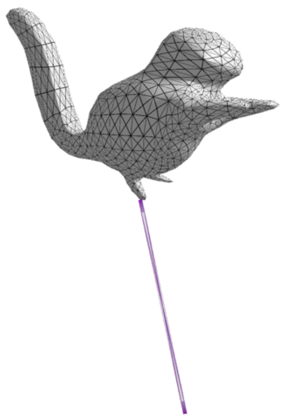
Figure 2: An inflated squirrel balloon. Compared to Figure 1, the squirrel's tail, head, and feet have changed shape.

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 spring forces, as well as any additional forces, like gravity, on the particles. This simulation method leads to realistic results and runs in real time. We chose to use this model in our balloon simulation because it is fast and easy to implement.

Similar to cloth, we use three types of linear Hookean springs in our balloon model: structural springs, shear springs, and flexion springs [Figure 3].

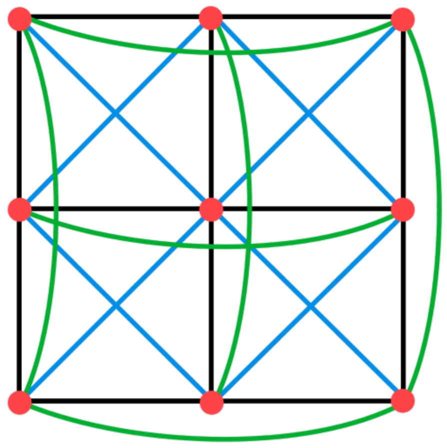
Figure 3: A standard cloth model. The red dots indicate cloth particles, the black lines indicate structural springs, the blue lines indicate shear springs, and the green lines indicate flexion springs.

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 (Provot, 1995), 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.

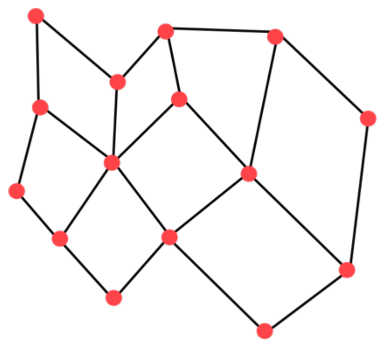
Figure 4: An example quad mesh.

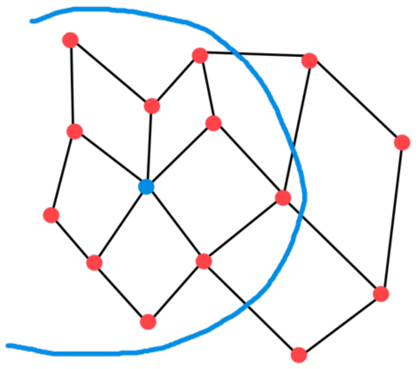
Figure 5: The blue vertex is the selected vertex and the vertices in the blue circle are the nearest vertices.

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.

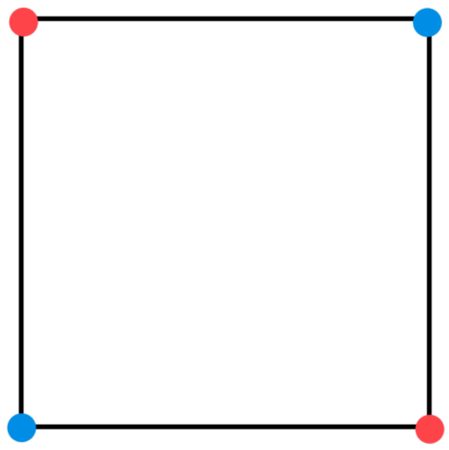
Figure 6: Red particles are across from each other in the face, and red particles are next to blue particles in the face.

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 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

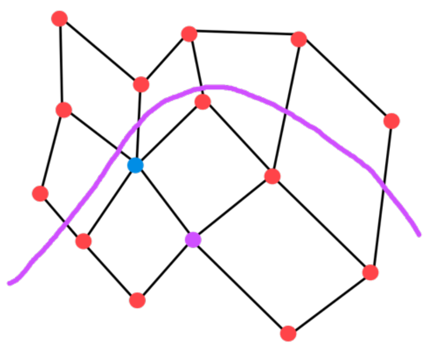

Figure 7: The purple vertex is the blue vertex's neighbor. Vertices under the purple line are in the purple vertex's nearest vertices.

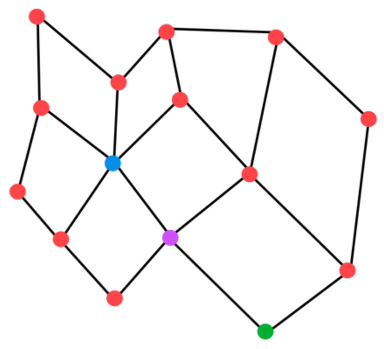

Figure 8: A flexion spring would be generated between the blue and green vertices.

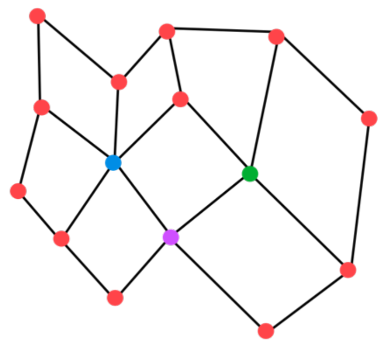

Figure 9: A flexion spring would not be created between the blue and green vertices, since they should create a shear spring.

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.

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

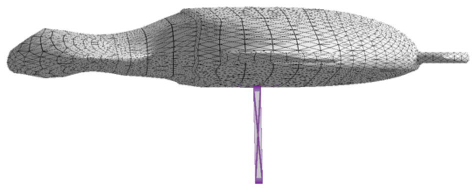
Figure 10: A deflated turtle balloon.

.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 4], we see an example quad mesh. In [Figure 5], 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.

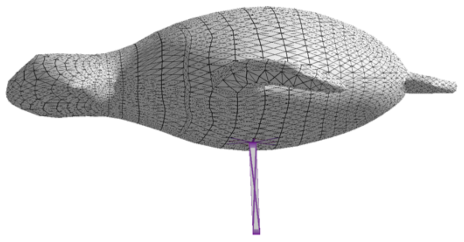
Figure 11: Inflated balloon without particle position correction.

```
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
```

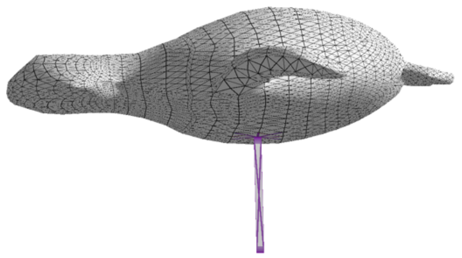
Figure 12: Inflated balloon with particle position correction. Correction is most apparent on the turtle's neck and belly.

```
Face::shouldCreateStructuralSpring(p, p2)
   if this.containsParticles(p, p2):
      if p and p2 are adjacent to each other in the face: [Figure 6]
         return true
   return false

Face::shouldCreateShearSpring(p, p2)
   if this.containsParticles(p, p2):
      if p and p2 are across from each other in the face: [Figure 6]
         return true
   return false
```

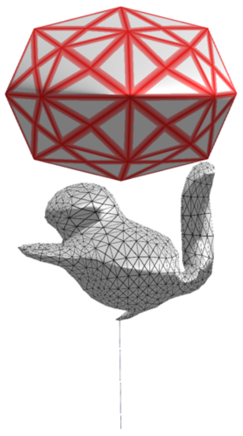
Figure 13: Squirrel and sphere before collision.

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 9], 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. We must account for this in the algorithm.

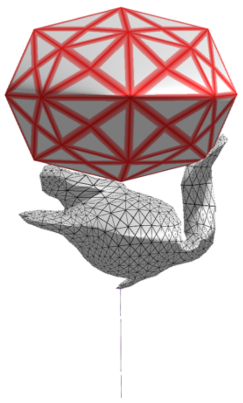
Figure 14: Squirrel getting squished by a sphere.

```
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
```

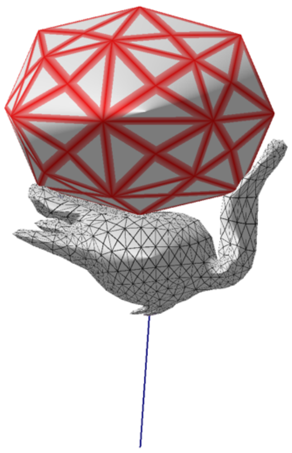
Figure 15: Squirrel getting very squished by a sphere.

Particle Simulation

We used a traditional mass-spring system with our simulation for the balloon. Traverse through each particle, add up the collective spring forces when applicable, and use explicit Euler functions to move the particles in the right direction. With the right spring constraints, the material "cloth" of the balloon simulates elastic rubber. To simulate inflation, we project a force that goes in the direction of the particles Gouraud normals, and that force is adjusted by a "k_value", which is calculated as such:

k_val = 100 * k_normal * (closest_face_to_particle / total_surface_area)

This ensures a proper adjustment of forces according to the surface area of the balloon, which is realistic to how latex is shaped for balloons. This is the pseudo code for our force calculations below:

```
for each particle p:
    springforce = 0
    springforce += shear spring forces
    springforce += structural spring forces
    springforce += flexion spring forces
    gravity = gravity_force + helium * p.mass
    damp = damping_force * p.velocity
    total = gravity - damp
    k_val = 100 * k_normal *
        (closest_face_to_particle / total_surface_area)
    total += k_val * p.normal
    p.acceleration = total/p.mass
    p.velocity = p.velocity + timestep*p.acceleration
    p.pos = p.pos + timestep*p.velocity
```

Figure 16: The sphere was on top of the balloon while it was deflated. As the balloon inflated, the sphere was launched upwards. The sphere has one unit of mass.

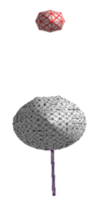
Figure 17: A sphere with 100 units of mass is launched like in Figure 16. It does not fly as far upwards because it is heavier.

Particle Position Correction

Position correction was taken from a couple of sources. We first approached this with traditional Provot correction. The results were undesirable. Using normal Provot made the balloon look like a larger version of the base model with no rounding or exaggeration of features commonly seen in weirdly shaped balloons. We had to loosen the constraints but still have some correction, so alongside raising the constraints of the springs themselves, we also adopted Tarantino's version of correction. His approach was this: if the spring is at its max, do not contribute spring forces to the particle anymore. We still correct them position-wise using Provot's methods, but the forces being taken away lead to smoother simulation and more rounded edges because particles do not move as suddenly. The effect of our correction algorithm can be seen in [Figures 10, 11, and 12].

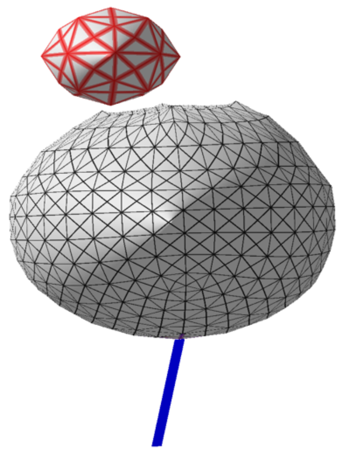
Figure 18: A sphere with 1000 units of mass is launched like in Figure 16. It barely launches off the balloon since it is so heavy.

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
```

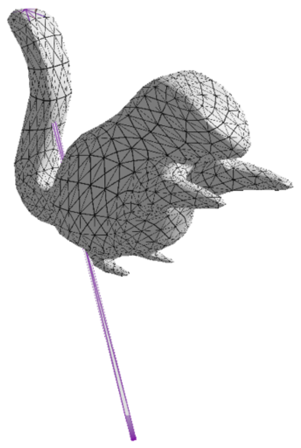
Figure 19: A string is attached to the ground and the very end of the squirrel's tail.

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 [Figures 13, 14, and 15]. By varying the mass of a collision sphere and the inflation of a balloon, we can achieve different collision scenarios [Figures 16, 17, and 18].

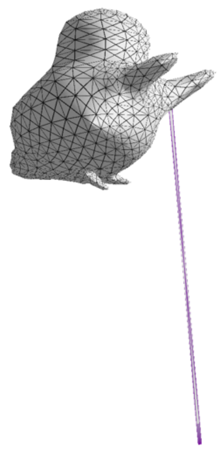
Figure 20: Squirrel is turning upside down.

Discussion

After implementing a particle correction algorithm, we noticed that the impact of the algorithm is very subtle [Figures 11 and 12]. In fact, we found that on low-poly balloon models, like our squirrel, using springs with large spring constants constrained the balloon's shape quite similarly to our correction algorithm. Additionally, our algorithm greatly constrains the balloon when the inflation force of the balloon is large. Correction algorithms like Provot's algorithm were designed to correct springs that are overstretched, since real cloth cannot stretch as much as simulated cloth without correction. However, real balloons do stretch quite a lot, so overstretched springs are not necessarily a bad thing in our simulation. Because of this, we do not use any correction algorithm in our default simulation.

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.

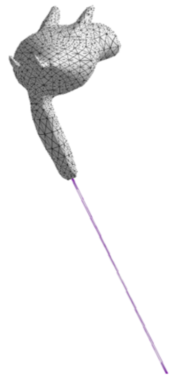
Figure 21: An upside-down squirrel balloon.

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 become a balloon.

In the first example [Figures 19, 20, and 21], 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.

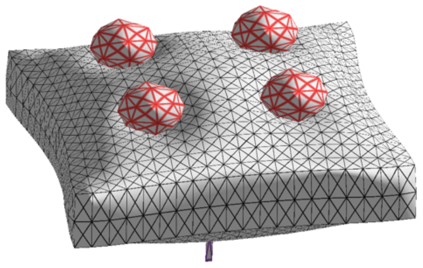
Figure 22: Four spheres colliding with a pillow-shaped balloon.

In the second example [Figures 22 and 23], 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 example [Figures 24 and 25], 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.

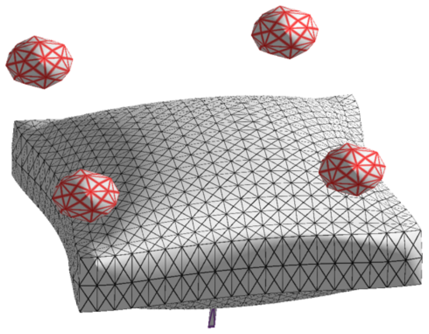
Figure 23: Four spheres bouncing off a balloon after colliding.

Conclusion

Our application simulates balloons realistically in real time. We adapt a mass-spring cloth simulation model to simulate our balloons. We apply a helium force to our balloons and apply a inflation force along surface normals. Finally, we investigated the use of a particle position correction to constrain the balloon's shape, and we found that algorithms similar to Provot correction and Tarantino's correction algorithm sometimes constrain the balloon's shape unrealistically, especially at greater inflation levels.

Acknowledgements
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.

4. Provot, Xavier. "Deformation Constraints in a Mass-Spring Model to Describe Rigid Cloth Behavior." http://www-rocq.inria.fr/syntim/research/provot/.

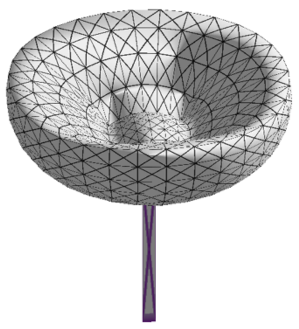
Figure 24: A deflated balloon

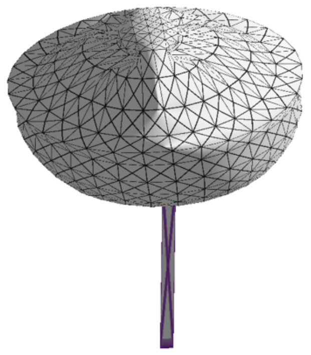
Figure 25: The balloon inflates from a bowl shape to a spherical shape.

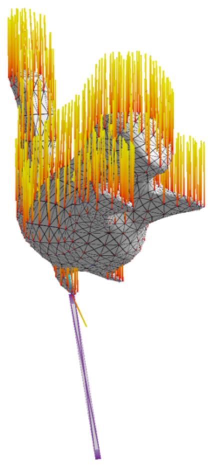
Figure 26: A visualization of the net forces on balloon particles. 🔥🐿️

🎈