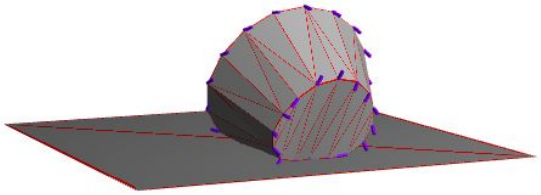
Real Time Simulation of Soft Objects Using Mass-Spring System

Haoxin Luo and Brandon Ip



Abstract

We explore a mass-spring based approach to simulating soft body deformation in real time. A mass-spring model is used to represent the 3D model, in which a set of point masses define the body's surface, and the points are connected to each other through springs. The body experiences gravitational force, spring, damping forces, and internal pressure force. These forces attempt to simulate the deformation and restoration behavior of soft bodies.

1. Introduction

Our goal for this project was to simulate plushies, which are soft objects usually modeling a character, animal, or object. We extended the mass-spring model that we had used for cloth simulation to work with 3D models. In addition to the spring forces and gravitational force, we added the internal pressure force described in the paper. Given at least one input mesh with several user defined values

including spring constant and mass, our implementation attempts to simulate and animate the behavior of soft objects. Our results are decent, though there is room for much improvement.

2. Prior Work

[1] This paper describes the mass-spring model similar to the one used in cloth simulation being used to simulate the soft bodies. Compared to other methods such as Finite Element Method or Elasticity, this is fast, uncomplicated, and easy to implement. The drawback is that it is not physically accurate. Instead, the simulation can be thought of as visually attractive. Some values like spring coefficients can be played with for more outlandish results, though this should be the case for most models. The paper describes several methods for simulation of soft bodies, but doesn't really provide any results, and it doesn't implement the methods in 3D models, which is the primary concern.

[2] This paper proposes a combined spring-mass and internal pressure system for the real time simulation of 3D soft body objects, and is the ideal goal of the project. Implicit integration is used for the animation of soft body surfaces, pressure forces for internal characteristics, and conservation of momentum for animation deformations. The object itself is modeled using a triangular mesh of connected points, where each has six neighbors. Each point experiences gravity, spring forces, damping force, and pressure. Spring force wise, the math is quite similar to the Provot implementation, but generalized for 3D meshes. The paper includes a fast and stable implicit integration improvement from Y. Kang et. al., while using Matyka et. al.’s ideal gas law based pressure implementation for internal forces.

[3] This paper describes a gas pressure based model for simulating soft body deformation. The basic idea is to imagine there is fluid inside the object providing pressure outwards, parallel to the normal of each face on the mesh. This particular paper chose gas as the internal fluid because it is faster to compute than navier stokes liquid simulation. The implementation is built on top of a working spring-mass model, specifically for cloth, to emulate soft body characteristics. The use of the Clausius

Clapeyron equation: PV = nRT simplifies calculations greatly as a particle system is not needed with the assumption that any particle to particle interaction is too small to affect the pressure and hence deformation of the object as a whole. The temperature is also assumed to be held constant, leaving pressure to be solely correlated to the volume of the object. The volume is estimated using different bounding boxes, namely, simple bounding box, bounding ball, and bounding ellipsoid.

3. System Overview

The system takes a single input file that is passed to an overarching *Mesh* class that emcomposes several *Plush* instances. Each *Plush* instance represents an independent entity in the simulation by storing its own simulation parameters, list of vertices, edges, triangles, and adjacency, and bounding box. In every step of the simulation, the sole *Mesh* instance iteratively calls each *Plush* to update its particles' attributes; e.g. acceleration,and position. The new acceleration of each particle is calculated from the summation of gravity, spring, and pressure forces. Integrated using verlet integration method, each particle's new position is set, bounding box updated, and then used for collision detection. After any collision response is handled, the *Mesh* then calls each *Plush* to push its vertex positions, and triangle indices into a common *Vertex Buffer Object* held by *Mesh* to be displayed.

4. Input Mesh Generation

We wanted our system to be able to support meshes of arbitrary shape and design, yet to use internal pressure the mesh must be closed. As discussed in section 6, internal pressure model also makes heavy use of surface area and volume of the mesh, and thus these are also primary concerns when selecting a mesh generation method. These objectives demand that the mesh be highly regular both in terms of vertex valence and triangle area. As such, it is unfeasible given the time constraints of this project to implement our own custom mesh generator. After experimenting with several free to use object modeling softwares, we settled on *Wings 3D*. This choice was made both based on ease of use and its ability to output directly in the file format we use for this project.

4.1 Input Mesh File Format

The system takes a single object file (.obj) as input. This file will contain the simulation parameters, vertex positions, polygon faces, and vertex properties for every object denoted in the file. The mesh generator we chose can output object files for every object we create, but it still needs to be further modified by hand to define several of our own simulation parameters. In addition, our system uses a different object delimiter than is standard for object files, which the modeling software follows, thus multiple objects must combined by hand into a single input file for the system. Below are some of the file format specifics that differ from the standard object file format:

- *k_structural* the spring constant for the mass springs
- *damping* the damping factor which simulates friction in the system to slow down particles
- *total_weight* the weight of the overall object which is evenly distributed to each of its vertices
- *nmol* the number of moles of gas particles inside the object. No gas particle is simulated in this system, but this is used to calculate the overall internal pressure
- *scale* the uniform scaling factor of the object. This is used to control the volume without having to recalculate vertex positions for the object.
- *translate* the 3D vector by which to translate the model. This is used to position the object in world space while still defining each object centered at the origin
- *fixed* specifies an index that refers to an already defined vertex point, and fixes its position throughout the simulation at its original location after scale and translate
- *END_OBJECT* our custom keyword that designates the end of one object

All constants are expressed in SI units unless otherwise specified. For example, distance is in meters, pressure is in Newtons per meters squared.

5. Mass Spring System

The input file is passed through the *Mesh* to consecutive *Plush* instances until the end of file is reached. Each *Plush* takes the parameters and vertexes to create a Mass Spring Model of the object. A Mass Spring Model is commonly used in soft body and cloth simulation, so we have adapted the method based off of paper [2]. To model the soft bodies, we use a triangular mesh consisting of a set of points with mass that represent the surface of the body. For now, assume that all meshes will be closed meshes. Each of the points are connected to each other through triangle edges, which will represent the springs in the model. These springs are massless and will attempt to return to their original length when stretched or compressed. The mesh is stored as half-edge structure with the addition of adjacency of vertices to speed up neighborhood detection given any vertex. To be more specific, the adjacency list is stored as a vector of sets of integer vertex identifiers , so that given each vertex we can immediately obtain all its valence vertices, and by extension, edges. This leads to a larger memory footprint, but leads to performance gain as no time is spent deleting or creating vertexes, edges, or triangles past initialization.

Once the mesh model is set up, we can simulate the behavior of the objects by iterating through their particles. In one timestep of the simulation, each of the points will experience a number of forces, which will be discussed in the following sections.

5.1 Gravitational Force

All objects on Earth experience gravitational force, and it can be computed through the equation: $F_g = mg$, where F_g is the gravitational force, m is the mass, and g is the acceleration due to Earth's gravity, which is a constant 9.81.

5.2 Spring Forces

Each of the points on the surface will experience forces from the springs connected to it. When a spring is stretched or compressed beyond its natural or original length, according to Hooke's Law, the

force exerted by the spring is linearly proportional to the difference in lengths (F = kX, where k is the spring constant and X is the difference in length). To compute this, we use the equation: $F_s = k(|v_{ij}| - l_{ij})(v_{ij})/|v_{ij}|$, where F_s is the spring force from spring ij onto point i, k is the user defined spring constant, v_{ij} is the vector from point i to point j, and l_{ij} is the natural length of the spring. The total spring force on a point i is the sum of all the forces from the springs connected by points j surrounding point i.

5.3 Damping Force

Damping force of some form is standard in simulation systems to ensure that energy in the system steadily decreases, leading to eventually convergence. In our system, the damping force is essentially air resistance as the object moves through space, and when it expands or contracts based on internal pressure and applied force from collision. Due to the verlet integration method used in our system, the damping factor is directly applied to neither acceleration nor velocity, but the position of each particle. The equation, $x(t + \Delta t) = (2 - f)x(t) - (1 - f)x(t - \Delta t) + a(t)\Delta t^2$, where f is $1/damping$ is used to update the position of every particle at each timestep [4].

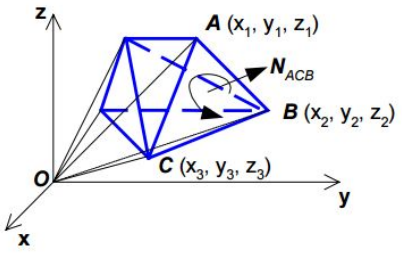
5.4 Internal Pressure Force

A closed mesh of purely mass-springs is essentially just a piece of cloth sewn into a ball. Logically, this cloth ball will still collapse into a pile or disc unless a force is present to maintain its shape. As such, [2], [3], and [5] have all proposed using internal pressure as the solution. Namely using the ideal gas law: $PV = nRT$, where P is pressure in N/m^2, n is the number of gas molecules in moles, R is the ideal gas constant, and T is the temperature. The idea is to assume there is n moles of gas inside the object, and thus the internal gas pressure would provide the force necessary to prop up the cloth ball. At the same time, gas is highly compressible and thus can convincingly simulate the deformation and rebound properties of soft objects. The key to not slowing the simulation down to a haul is to not simulate any gas particles inside the mesh, and simply assume the temperature and the particle

interactions would not affect the mesh's behavior in any other way [5]. Thus, after some manipulation of the equation, all that is needed is the surface area (*S*) and volume (*V*) of the mesh.

$$F_p = \frac{SnRT}{V}$$

The surface area of the mesh can be easily approximated by summing up the area of every triangle in the mesh. Since the mesh is assumed to be closed, this approximation is as accurate as the model itself (of the intended simulation object). The volume calculation also turned out to be trivial. [6] verifies that many features, including area and volume, of a mesh can be calculated by a simple aggregation. The idea is to pretend each face triangle forms a tetrahedron with the origin point (or any other point in fact), and calculate the volume of this shape. This volume is made to be either positive or negative based on the cross product of triangle normal and direction to the origin. The volume over each triangle-tetrahedron is then summed to form the volume of the mesh itself. Precise equation and diagram from [6]:



$$|V_{OACB}| = |\frac{1}{6}(-x_3y_2z_1 + x_2y_3z_1 + x_3y_1z_2 - x_1y_3z_2 - x_2y_1z_3 + x_1y_2z_3)|$$

Where V_{OACB} is volume of triangle *ACB* with *O*rigin, and x_1, y_1 and z_1 are the position of point *A*.

6. Collision Detection

In order to test the accuracy of our simulation, collision detection between two objects is essential. In our implementation, we have two different tests depending on what kind of objects are in the simulation: one for collision between a soft object and a floor, and one for collision between two soft objects.

6.1 Collision Between a Soft Object and a Floor

Collision between a soft object and a floor is fairly simple. Assume the floor is a mesh made up of two triangles facing upwards. Using the normal of the plane, we can test if all the points of the soft object is in that direction by using dot products. If the equation v • n, where v is a vector from any point on the floor to the point in question and n is the floor normal, returns a negative number, then the point is on the other side of the floor and collision should be handled.

6.2 Collision Between Two Soft Objects

Collision between two soft objects is much more complicated and can be very expensive if not optimized. Complex objects can consist of thousands of points, and calculating which two points are the points of collision in each timestep takes a ridiculous amount of time. For our implementation, we use some of the algorithms described in [7].

In the first phase, also known as the broad phase, we look at each object as a whole, using their bounding boxes. Using simple vector math we can find if two bounding boxes are overlapping. It is important to note that the bounding boxes are axis-aligned. Working with oriented bounding boxes requires a little bit more math, so we didn't go that route.

The second phase, also known as the narrow phase, attempts to detect collision more precisely. We base our method on the Separating Axis Theorem, which states that two convex shapes are not intersecting if and only if there exists at least one axis where the orthogonal projections of the shapes on the axis do not intersect.

Firstly, we will assume that all of our objects are convex shaped. As for the test axes, there exists an infinite amount of axes, but for this test, there are only specific ones that we need to test: the normals of all faces of each object, and the cross products of all pairs of edges, one from each object. To project the shapes of the objects onto the axis, we need to project all the points on to the axis using

dot products and find the minimum and maximum for each object. Finally, we can test these values to see if they overlap.

Unfortunately, at the moment, our implementation of this method does not seem to work properly, so we are unable to test simulations involving two soft objects.

7. Integration

The integration method is a key determiner of simulation accuracy, timestep size, and consequently simulation speed. [2], whose work we primarily based off of, proposes a "simplified implicit integration" method for calculating new acceleration values that they claim is stable and efficient enough for real time simulation. Their formula is shown below:

$$\Delta v^{t+h}{}_i = \frac{(F'_i h(|F_i{}^t| + khn_i|\Delta v^t{}_i|))}{(|F_i{}^t|\,(m_i + kh^2 n_i))}$$

Where $\Delta v^{t+h}{}_i$ is the acceleration at the next instance, $F_i{}^t$ is total force on particle *i* at time *t*, *k* is the spring constant, *h* is delta time, n_i is number of neighboring points, and m_i is the mass of particle *i*. This integration method promises a lot between stability, robustness, and efficiency; not to mention, this also looks fairly easy to implement as it does not require solving complex systems of linear equations like earlier implicit integration methods [8].

Disappointingly, none of the above properties held in our implementation of the method. Using [2]'s method caused similar instabilities and eventual explosion of the mesh as explicit euler, albeit later on in the simulation. Following this find, we implemented two other explicit integration methods and decided to stick with the one we found more stable: verlet integration. The base verlet integration method we implemented is of the form:

$$x(t + \Delta t) = 2x(t) - x(t - \Delta t) + a(t)\Delta t^2$$

Where *x(t)* is the position at time *t*, and Δt is the size of the timestep, and *a(t)* is the acceleration at time *t*. We do not claim to have found this method to be generally superior to any of the other methods we

have tried, but merely that this method seemed to work the best for the specific set of simulation parameters we have tried. Verlet integration lead to the most stable simulation at the super small timestep of 0.001 second, but showed even stronger oscillations. The mesh would repeatedly expand and contract to no end. Eventually we found that verlet integration requires a different way of calculating damping because this method does not directly consider velocity of the particles at any point. To add damping back, we modified the above equation to that described under section 5.3.

8. Conclusion

We do have a decent, working simulation of a soft object as it collides with the floor, and the result looks somewhat realistic. We were a little too optimistic about the goals we initially set out to reach. However, had we had more time, we believe we could have solved most of the persisting problems, which include robust collision detection between two soft objects, and accurate bouncing of a soft object on the floor. Currently, the objects hit the floor and compress as normal, but the objects do not seem to try to revert to its original form and bounce from the floor as we had hoped. This could be a problem with providing the correct combination of input values. We think this is a reasonable assumption as earlier we had a serious problem dealing with irregular vertices in the mesh, however, the problem fixed itself once we found better values for the input parameters.

9. Future Work

If we continue working on this project, we believe another attempt at [2]'s implicit integration method is well worth the time. The potential benefits of an accurate, robust, and efficient integration scheme would make this simulation much more realistic as a simulation of soft body objects. It will open many opportunities to explore many different types of soft body objects from balloons to pillows to animals. Another thing worth exploring is accurate collision response between soft body objects. This goal is very similar to accurately simulation cloth-cloth interactions, especially with this mass spring

model, but would prove much more interesting as different “degrees of softness” will lead to drastically different responses.

10. Division of Labor

- Approximately 65 hours spent on this project
- Mesh input generation - Haoxin Luo
- Mesh input parsing and formatting - Haoxin Luo
- Creating original code base of half-edge mesh structure and springs - Both
- Volume calculations - Haoxin Luo
- Explicit Euler, Fourth Order Runge-Kutta, and Verlet integration - Haoxin Luo
- [2]’s simplified implicit integration - Brandon Ip
- Collision Detection - Brandon Ip
- Collision Response - Brandon Ip
- Floor Mesh - Brandon Ip

11. References

[1] Ben Kenwright, Rich Davison, Graham Morgan. Real-Time Deformable Soft-Body Simulation using Distributed Mass-Spring Approximations.

[2] Jaruwan Mesit, Ratan Guha, Shafaq Chaudhry. 3D Soft Body Simulation Using Mass-spring System with Internal Pressure Force and Simplified Implicit Integration

[3] Svenska Föreningen. Pressure Model of Soft Body Simulation. The Annual SIGRAD Conference 2003

[4] Various. Maintained by wikipedia. Verlet Integration

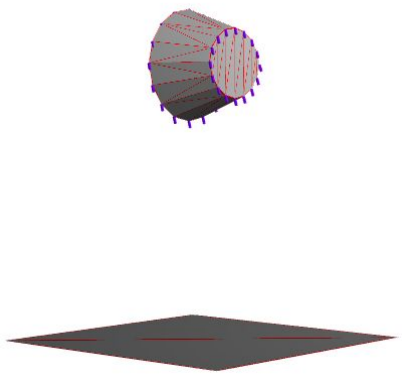
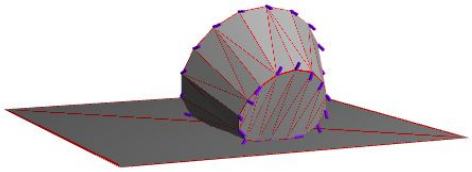
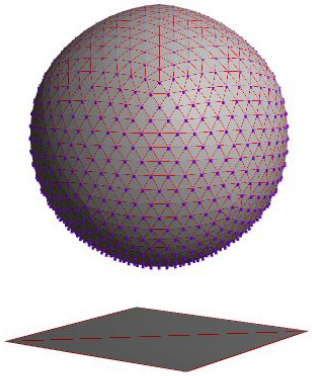
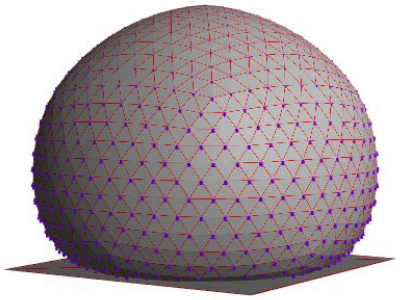
[5] Maciej Matyka, Mark Ollila. Pressure Model of Soft Body Simulation

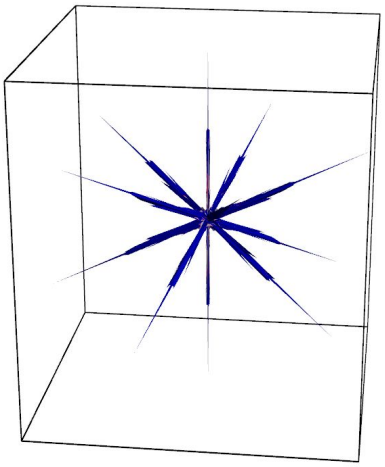
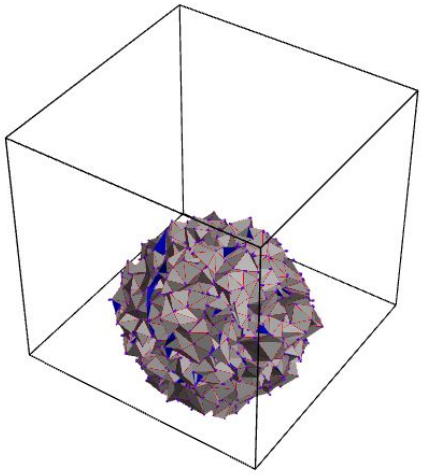
[6] Cha Zhang, Tsuhan Chen. EFFICIENT FEATURE EXTRACTION FOR 2D/3D OBJECTS IN MESH REPRESENTATION

[7] Nilson Souto. Collision Detection for Solid Objects

[8] David Baraff, Andrew Witkin. Large Steps in Cloth Simulation

12. Photos

	
In this experiment, an object shaped like pudding with 32 points is dropped to the floor.	The result of the pudding colliding with the floor
	
In this experiment, a sphere with 1002 points is dropped to the floor.	The result of the sphere colliding with the floor

	
Implicit euler (original method) with timestep=0.01, k=1, w=0.1, d=4, n=0.007	Wolfram-alpha symbol found!? Same constants but timestep=0.001
	
Simplified implicit integration method caused the mesh to collapse on itself	