Jello Simulation

Roy Wellington Ben Boeckel

May 12, 2010

Contents

1 Abstract **3**

2 Motivation **3**

3 Previous Work **3**

4 Algorithm **3**
- 4.1 Core mass-spring algorithm . . . 3
- 4.2 Mesh definitions . . . 4
- 4.3 Spring connections . . . 5
- 4.4 Point-in-mesh . . . 6

5 Results **11**
- 5.1 Performance . . . 11
- 5.2 Challenges . . . 11
- 5.3 Backend Library . . . 12

6 Future Work **12**
- 6.1 Performance Enhancements . . . 12
- 6.2 Skins . . . 13
- 6.3 MPI . . . 13
- 6.4 Magic constants . . . 13

7 Availability **13**

1 Abstract

Jello is an attempt at simulating realistic 3D movement similar to the properties of Jello. The project attempts to replicate the jiggle motion of Jello through a three-dimensional mass-spring model, whereby various point masses are connected by springs.

2 Motivation

Jello is one of those materials that is just interesting, and its movement fascinates the eye. Jello, and Jello-like materials, can have various applications — from modelling the snack-time favorite to making a horrific blob monster, providing a believable simulation of a gelatinous material could add realism and a sense of attention to detail. Further, accurately modelling the movement of a large or complex gelatinous object could be difficult or tedious.

3 Previous Work

Mass-spring models have been used previously in producing realistic cloth.

4 Algorithm

4.1 Core mass-spring algorithm

Our core algorithm to deal with the simulation is based on the algorithm we used in our cloth simulation homework. First, each of the points' positions are updated according to its velocity and then the velocities are updated using the points' acceleration. After this is done, the acceleration is then updated using the forces on the point. This involved doing the spring calculations using the spring formula:

$$a_i = \frac{k \sum_{n \in N_i} (\Delta d_{i,n}) - \delta v_i}{m_i} \tag{1}$$

where k is the spring constant, N_i is the set of all "neighbor" points (those points connected to this point by springs)$\Delta d_{i,n}$ is the displacement between

the current distance and original distance between points i and its nth neighbor, δ is the damping factor, v_i is the velocity of the point, m_i is the mass of the point. Points are defined as being neighbors of each other depending on the input. The various input data accepted are discussed below. Once all of the new forces are calculated, a few corrections are made. The spring calculations are parallelized using OpenMP. This doubled the amount of points we were able to simulate in real time. It also allows our simulation to scale across multiple core machines with ease.

We iterate over every point in the Jello mesh and for each of its neighbors, we do Provot correction as described in (Provot, 1995). This involves adjusting the positions of points along the spring to ensure that they are not too short nor too long. The amount that the spring is allowed to be out outside of the original spring length, τ, is a configurable fraction of the original spring length.

Unfortunately, the weight of the Jello is too much for any sane k. To avoid this, we assume that there are springs between all points when doing the Provot correction. Although incorrect, it results in a believable looking Jello simulation.

While calculating the Provot corrections, we also do hit detection with the ground (assumed to be at the $y = 0$ plane in our demo) and corrected so that the velocity in the y direction is 0 and the velocities in the x and z dimensions are multiplied by a factor, μ, to simulate a coefficient of friction. In addition, the acceleration in the y direction is slowed by a factor ν. In our simulations, we currently use a μ of 0.1 and a ν of 0.2.

$$v_i' = \mu v_{i_x}\mathbf{i} + \mu v_{i_z}\mathbf{k} \tag{2}$$

$$a_i' = a_i - \nu a_{i_x}\mathbf{j} \tag{3}$$

The Provot correction loop is run multiple times until the displacements done are minor. Our tests indicated that 5 loops was sufficient.

4.2 Mesh definitions

Jello meshes can be defined multiple ways. We support generating a random field of points inside of an area, creating a volumetric mesh from a surface mesh, and a custom file format that define points and springs for the simulation.

The random points can be generated so that they are distributed uniformly within the field or using a distribution which concentrates them along the edges of the field. By concentrating points at the edges, we allow the faces of the field to be better defined. The edge distribution we use is:

$$\frac{1 + \cos(2\pi n)}{2} \tag{4}$$

where n is a value in the range $[0, 1)$. When given a surface mesh, we generate points within the mesh (the algorithm that determines whether a random point is inside the mesh is below). These points are then passed along the spring connection algorithms. Using VTK[1], we support many file formats, including, but not limited to PLY and OBJ. Our custom file format needs no further processing before being handed off to the spring connection algorithms.

4.3 Spring connections

We offer three ways to connection points in the mesh with each other, an "implicit", a marching tetrahedron, and another which uses a Delaunay tetrahedralization algorithm provided by the VTK toolkit.

The "implicit" connection algorithm uses a kd-tree structure to store all of the points in the mesh and then defines that a point's neighbors are those within a radius r. We use a BSD licensed library[2] as our kd-tree implementation which we then wrap for ease of use. We then query the kd-tree for points that are within radius r of the vertex in question and then connect them together with springs. This results in strongly connected meshes and seems to perform the best out of all of our spring connection methods. A downside is that it creates highly-connected meshes which slows down the force calculations.

An algorithm loosely based on the marching tetrahedrons partitions the mesh's bounding box into a grid of cubes, each cube divided into several tetrahedra, as in the marching tetrahedra algorithm. If the various endpoints of the tetrahedra are found to be inside the mesh, then the edges of those tetrahedra are added as springs.

When using Delaunay tetrahedralization, we first clean up the input so that no two points are too close to each other since this would cause there to

[1] `http://vtk.org`

[2] `http://code.google.com/p/kdtree`

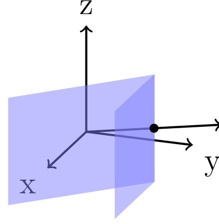

Figure 1: Collision with an edge, grazing mesh

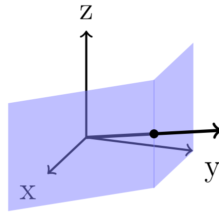

Figure 2: Collision with an edge, crossing mesh

be very narrow degenerate tetrahedrons. Another degenerate form is where the input points form a uniform grid since there are many points that lie on the circumsphere of tetrahedrons.

4.4 Point-in-mesh

In order to generate a mass-spring structure for the Jello movement algorithm, it was necessary to be able to test if a point was inside or outside a mesh. A simple algorithm for this test is to shoot a ray in any direction from the point in question, and count the number of times you intersect the mesh. As long as the mesh is well-formed (no loops or self-intersections), if the number of intersections is odd, then the point is inside the mesh.

This algorithm works well in theory, however, its implementation is difficult. Within a mesh composed of triangles (such as ours, and almost any 3D model), problems arise if the ray goes through a vertex or a triangle's edge. If this happens, the implementation must look at the surrounding edges to see if it is actually crossing the mesh. Figures 1 and 2 demonstrate this with edges, similar problems arise with vertices.

To make this work in practice, it is considerably easier to run the algorithm in a 1D perspective. This algorithm, in 2D, is counting the number

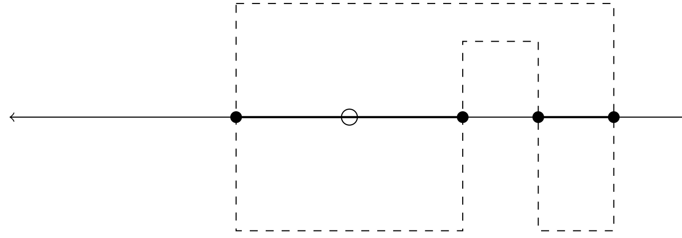

Figure 3: A 1D contour example. The original 2D contour is shown, and the point of interest is the open circle. Here, it is inside the mesh.

of times a ray crosses a 2D contour, or a polygon. In 1D, you can think of a number line, with the mesh being points along this line, and our point of interest also being on the line.

The 2D contour is a cross section of our 3D mesh, in any plane (for convenience, we choose xy), and the 1D contour is a cross section of our 2D contour. Thus, we can boil our 3D mesh down to a 2D contour, then to a 1D contour, and then run the algorithm.

The advantage of getting this 1D projection is that in 1D, the edge cases we experience in 3D are non-existent. Effectively, we handle these edge cases as we move from 3D to 1D.

Generation of a 2D contour is relatively easy, and boils to a few simple cases. For each triangle:

1. *it's complete above or below the plane of interest* — ignore it.

2. *it's intersecting the plane completely (2 vertices on 1 side, 1 on the other)* — find the segment on the triangle where it intersects. This is a line, and is part of your contour.

3. *a edge of the triangle is in the plane* — choose one side of the plane (use the same side for all the triangle). If the point is on that side of the plane, then use the edge, otherwise, ignore it.

4. *the triangle lies in the plane* — make a 2D contour from the triangle itself, and run a subtest wit this 2D contour, but only if points on the mesh's surface count. (This is a case where the point is neither inside or outside, but on the mesh.)

The same algorithm works in 2D, and will generate a 1D contour. (Segments on a number line.) Note that working with meshes formed from

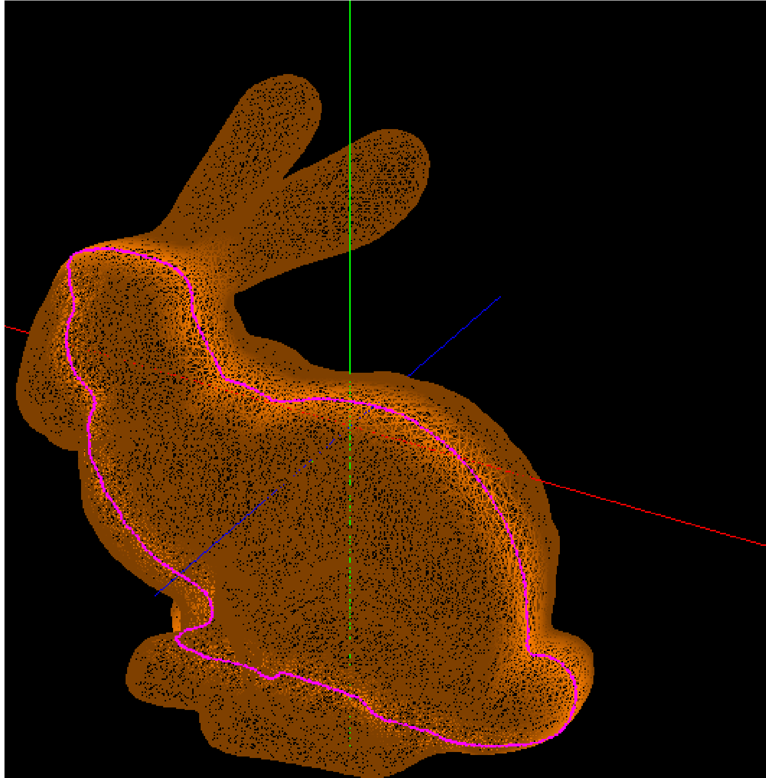

Figure 4: 2D Contour of the Stanford Bunny at $z = 0.75$

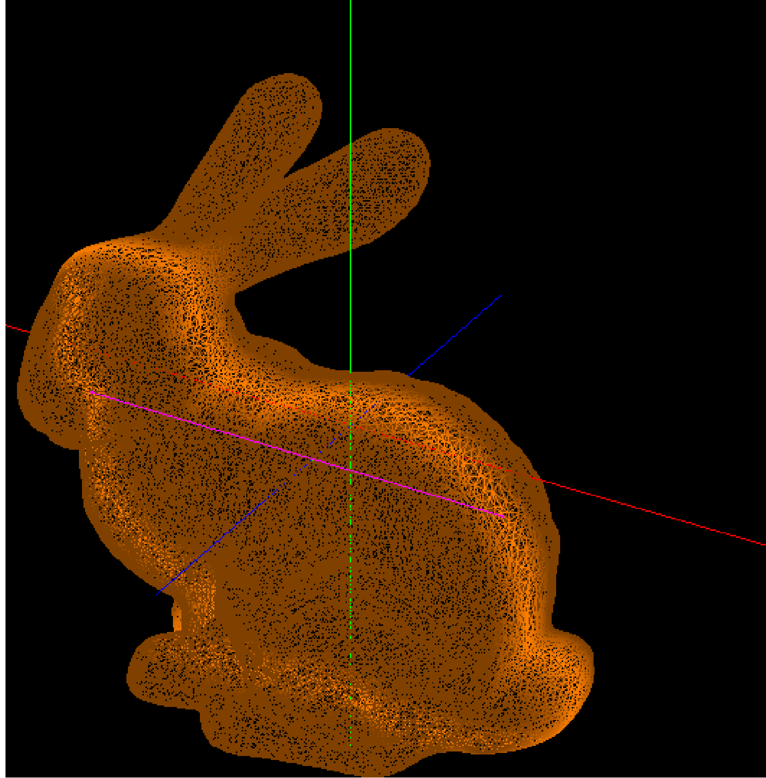

Figure 5: 1D Contour of the Stanford Bunny at $y = 0.5$

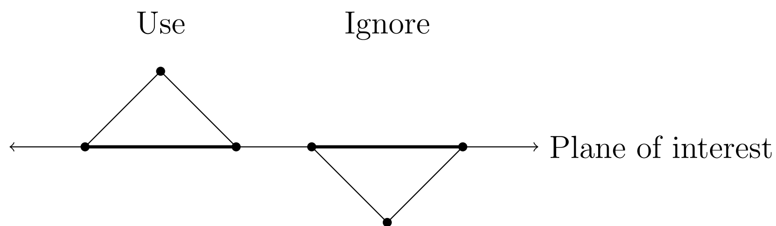

Figure 6: Algorithm case: Triangle is grazing the plane with an edge.

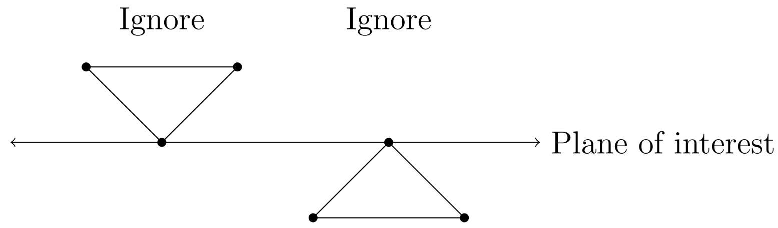

Figure 7: Algorithm case: Triangle is grazing the plane with a vertex.

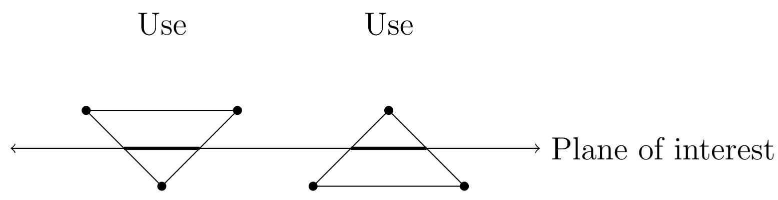

Figure 8: Algorithm case: Triangle intersects plane.

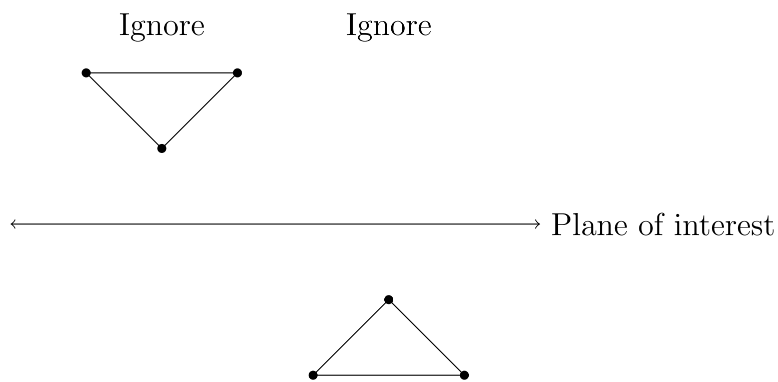

Figure 9: Algorithm case: Triangle isn't near plane.

floating-point data, or generating a 2D contour with floating point data, often results in a disconnected 2D contour, due to floating point rounding. It is often necessary to stitch the resulting 2D contour together to form a coherent 2D contour. It is worth mentioning that this algorithm will correctly handle concave meshes, and meshes with "bubbles".

5 Results

5.1 Performance

Our Jello simulation looks like real Jello, but it not very efficient. On one test machine, we achieved 18 timesteps per second with a distribution of 600 points using the edge-favoring distribution in Equation 4. Each timestep was 0.01 s, for a total of 1 s of simulation. This was using the implicit spring connections within a cube a meter to a side, a connection radius r of 0.3, a damping factor δ of $0.8\,\mathrm{kg}\cdot\mathrm{s}^{-1}$, a tolerance τ of 0.1%, one Provot correction loop, a spring stiffness k of $2\,\mathrm{N}\cdot\mathrm{m}^{-1}$, a coefficient of friction μ of 0.1, and a acceleration factor when hitting the floor ν of 0.2. The simulation ran on an AMD Opteron quad-core processor running at 2.21 GHz. The simulation is extremely CPU-bound and RAM usage was negligible at around 77 MiB.

5.2 Challenges

We also found that without a few hacks that our simulations didn't quite work. The first is that if Provot correction correction is only done between points with actual springs between them, they are not strong enough to hold the Jello up. Our spring constant and mass calculations are also not very true to the actual physical constants. They work fine when we use Provot correction between all points, but are entirely insufficient when only actual spring connections are considered. Our models tended to fall onto the floor and then spread out across it with these values.

Also, like most mass-spring models, constant values must be chosen carefully. If the values for spring constants, masses, and some physical constants like gravity are not carefully chosen, the simulation can end up melting into a pool of vertices or flinging itself into oblivion.

Our physics when the model hits the floor is one of the most hackish parts of the code. We do not have a normal force which pushes back into

Figure 10: Bunny to Infinity

the model so we cannot actually simulate a bounce off of the table with the current implementation. The friction is, however, necessary since without it, our model would slowly rotate and move across the surface of the table, not unlike a slug.

5.3 Backend Library

A fair portion of the code dealing with three dimensional manipulations, inclusion testing, etc., can be easily siphoned off of this project into a side library. The portions of the code relevant to general computer graphics have been informally named "`libacg`" (advanced computer graphics) by the developers. It includes code for managing 3D points, matrices, and even some of the basic mesh loading, point inclusion testing, and mesh algorithms could be potentially separated into this library. The code follows modern C++ development principles, and aims to be easy to use.

6 Future Work

6.1 Performance Enhancements

Spring force calculation is done from the perspective of points, meaning that for a particular spring, it is calculated twice, once from each side. These calculations should be cached, in order to only perform them once.

6.2 Skins

Some code exists within the project to allow the layering of a mesh over the Jello spring structure. This would allow a mesh to be attached to the mass-spring structure, adding visual form to the model, and allowing a user to more easily see the Jello itself. This is fairly straight forward for many forms of the Jello, such as the model based off the marching tetrahedra algorithm: the spring structure itself is derived from mesh data, so the mesh already exists.

6.3 MPI

Currently, OpenMP allows the code to run in parallel over multiple CPUs within the same machine. While this is useful, for larger simulations, more CPU power may be needed, especially since the simulation itself is CPU bound. MPI would allow the simulation to be spread among many machines, using multiple machines and multiple processors to boost speed. Note that only intra-timestep calculations can be done in parallel, since timestep $t+1$ depends on the output of timestep t — thus, two machines can't be doing timesteps t_1 and t_2 in parallel.

6.4 Magic constants

Much work needs to be done on many of the constants within the program. Many of the constants in use currently are used because they look visually appealing. Research into the actual constants of Jello has been done, and it is our hope to bring the constants in the program in line with the actual physical constants that could be measured in the real world. Currently, gravity is set to $-9.8\,\mathrm{m}\cdot\mathrm{s}^{-2}$, matching the real world constant. The spring constant and density however, do not closely match. Research on the Internet indicated that Jello had a density of $1141\,\mathrm{kg}\cdot\mathrm{m}^{-3}$ and that it had a spring constant of $606\,\mathrm{N}\cdot\mathrm{m}^{-1}$.

7 Availability

Our code is licensed under the BSD license and available via a `git clone git://cledwyn.benboeckel.net/rpi/jello.git` or via the gitweb interface at `http://cledwyn.benboeckel.net/git?p=rpi/jello.git`.

List of Figures

1 Collision with an edge, grazing mesh 6
2 Collision with an edge, crossing mesh 6
3 A 1D contour example. The original 2D contour is shown, and the point of interest is the open circle. Here, it is inside the mesh. 7
4 2D Contour of the Stanford Bunny at $z = 0.75$ 8
5 1D Contour of the Stanford Bunny at $y = 0.5$ 9
6 Algorithm case: Triangle is grazing the plane with an edge. . . 9
7 Algorithm case: Triangle is grazing the plane with a vertex. . 10
8 Algorithm case: Triangle intersects plane. 10
9 Algorithm case: Triangle isn't near plane. 10
10 Bunny to Infinity 12

References

X. Provot. Deformation constraints in a mass-spring model to describe rigid cloth behavior. In W. A. Davis and P. Prusinkiewicz, editors, *Graphics Interface '95*, pages 147–154. Canadian Human-Computer Communications Society, 1995. URL `citeseer.ist.psu.edu/provot96deformation.html`.