Co-rotational Finite Element Solid Simulation with Collisions

Patrick Riordan

2015-12-11

Contents

1 Introduction

Physically Based Animation is becoming a bigger part of real time graphics as both our hardware and software get faster. Deformable Solid Simulations create more complex phenomena than Rigid Bodies and are more suited for integration with character models. In this paper I will present the Co-rotational Linear FEM algorithm applied to deformable solids covering some of the same topics as Matthias Mueller's Real Time Physics Course Notes, specifically chapters 3 and 4. This paper should not only allow the reader to implement a deformable solid simulation but also allow them to reason about these algorithms. This understanding is essential in order to apply them in practice.

2 Continuum Mechanics

In order to derive discrete equations for deformable solids we first have to get familiar with the continuous equations that govern deformable solid movement. What I present in this section is a condensed version of what appears in Mueller's course notes.

2.1 Stress and Strain

Hook's Law In the one dimensional case a deformable solid is simple. Given a deformation of length Δl there will be an internal force f. Using Hook's law this force depends the proportion of the total length l and is applied across the cross sectional area A.

$$f/A = E\Delta l/l. \tag{1}$$

This relation depends on E, Young's Modulus. This equation illustrates the relationship between a deformation and the resulting force.

$\Delta l/l$ is called the strain. Strain refers to the proportional deformation.

f/A is the stress. Stress is the internal forces resulting from deformation. Stress directly relates to force while strain only measures the deformation. Our derivation of the continuous equations will be similar to this simple formula but in 3 dimensions. Given a deformation what is the resulting force? In order to answer this question we need to first figure out how we formulate the deformation. Then we need to measure the strain resulting from that deformation. From the strain we can calculate the stress and finally from the stress we can calculate the force at any point in the solid.

2.2 Deformation Gradient

In a deformable solid a point x will be deformed to another point $p(x)$. This deformation can thought of as a vector field $u(x) = p(x) - x$. Translation for example would mean that $u(x) = k$ for every point x, where k is a constant. In 3D these quantities are all 3 component vectors.

Gradient As we can see from $u(x)$ knowing one offset doesn't help us calculate the strain. We want to know if the local point is compressed or stretched somehow. To do this we calculate a deformation gradient $\nabla u(x)$ around some point x.

$$\nabla u(x) = \begin{bmatrix} u_{1,x} & u_{1,y} & u_{1,z} \\ u_{2,x} & u_{2,y} & u_{2,z} \\ u_{3,x} & u_{3,y} & u_{3,z} \end{bmatrix}, \tag{2}$$

where $u_{1,x}$ is the derivative of the first component of $u(x)$ with respect to the first or x component of x.

$\nabla u(x)$ is the derivative of a deformed offset with respect to the original point. We can intuitively understand this as three column vectors, the first describing what direction a point p' one unit in the undeformed x direction would move if the deformation gradient was constant. The same logic applies for the 2nd and 3rd columns for y and z respectively.

2.3 Strain Measures

A strain measure is an in-between step from the deformation gradient to the stress. We want the strain to only represent deformations that would affect the stress. If the object is at rest with no external forces than the strain should be 0.

There are two strain measures we will go over: Green strain and Cauchy strain.

Green Strain

$$\varepsilon_G = \frac{1}{2}(\nabla u(x) + \nabla u(x)^{\mathrm{T}} + \nabla u(x)^{\mathrm{T}} \nabla u(x)). \tag{3}$$

Cauchy Strain

$$\varepsilon_C = \frac{1}{2}(\nabla u(x) + \nabla u(x)^{\mathrm{T}}). \tag{4}$$

Green Strain has some useful properties, the magnitude of the strain does not change under rotation or translation. The drawback to Green strain is that it's quadratic. Cauchy Strain on the other hand is linear but does not handle large deformations such as rotation properly. This linear property will prove to be useful when want to calculate velocities of a discrete domain. See Mueller's course notes for the motivation for these strain measures.

2.4 Calculating Stress

We want to generate a 3×3 stress tensor we will label as σ. Using this type of stress tensor we can calculate the force f on any cross-sectional area A through the material using this formula:

$$f/A = \sigma * (-n), \tag{5}$$

where n is the outward pointing normal of the area.

2.5 Hookean Elasticity

There's only one piece missing now if we want to get the internal forces from the deformation gradient. In the dimensional case it would simply be E, but now were are dealing with a 3×3 strain tensor instead of a scalar value.

The diagonals of the strain measure σ represent how stressed and compressed the material is while the off diagonal entires represent shear. Hookean elasticity defines a linear relationship between strain and stress. Both strain and stress are symmetric which means there are 6 degrees of freedom. We can represent a linear mapping from each component in strain to stress using a 6×6 matrix. Now we can calculate the stress from strain using Hookean elasticity as follows:

$$\begin{bmatrix} \sigma_{xx} \\ \sigma_{yy} \\ \sigma_{zz} \\ \sigma_{xy} \\ \sigma_{yz} \\ \sigma_{zx} \end{bmatrix} = \frac{E}{(1+v)(1-2v)} \begin{bmatrix} 1-v & v & v & 0 & 0 & 0 \\ v & 1-v & v & 0 & 0 & 0 \\ v & v & 1-v & 0 & 0 & 0 \\ 0 & 0 & 0 & 1-2v & 0 & 0 \\ 0 & 0 & 0 & 0 & 1-2v & 0 \\ 0 & 0 & 0 & 0 & 0 & 1-2v \end{bmatrix} \begin{bmatrix} \varepsilon_{xx} \\ \varepsilon_{yy} \\ \varepsilon_{zz} \\ \varepsilon_{xy} \\ \varepsilon_{yz} \\ \varepsilon_{zx} \end{bmatrix}. \tag{6}$$

E is Young's modulus, a scalar describing the stiffness, and v is Poisson's ratio, a scalar between 0 and $\frac{1}{2}$ ($\frac{1}{2}$ exclusive, 0 inclusive) describing how volume conserving the material is.

3 Discretization

Now we can take a deformation gradient and find the force on any given area, but how do we find the deformation gradient? We can't solve it for every point in a solid, so we have to break it up into manageable pieces. These are the basics of FEM. To numerically solve a PDE we break up the domain into finite elements and simplify the equations to be easily solvable for each finite element. For this problem there are two commonly used finite elements: a tetrahedron or a hexahedron with quadrilateral sides. We will use the tetrahedron for now because it is easier to calculate the deformation gradient for it.

3.1 Calculating Deformation Gradient

For each tetrahedron we can simplify the deformation gradient, a continuos vector field, into a linear mapping (3×3 matrix) per tetrahedron. This introduces errors but these errors should be proportional to the how small the tetrahedra are. To obtain the deformed position of point within undeformed tetrahedron with undeformed corner points x_0, x_1, x_2, and x_3, and deformed corner points p_0, p_1, p_2, and p_3 we use this formula:

$$P = [p_1, p_2, p_3][x_1, x_2, x_3]^{-1}. \tag{7}$$

This allows us to take a point x and and find its deformed position using the formula $p(x) = Px + p_0$. The deformation gradient $\nabla u = P - I$. This formula

takes every point within the tetrahedron as a linear combination of the other 4 points and then applies the deformation gradient to it. See Mueller's course notes for the full derivation.

3.2 Explicit Euler Implementation

Because we have a constant deformation gradient throughout the tetrahedron we will have a constant strain as well. We can now plug in our deformation gradient into a strain measure to get the strain. Then we can use this strain to get the stress and this stress to derive the nodal forces. Using this we simulate a deformable solid using explicit Euler and the normal equations of motion. Each frame we calculate the nodal forces i.e. the forces per corner point and use this to calculate the acceleration on that frame. If we assume that the acceleration is constant for this frame, we can calculate the new velocity by adding the acceleration multiplied by the times step. Using this velocity we can update the positions. This integration scheme will work for flexible materials, but for very stiff materials the strong nodal forces will make the simulation unstable. See Mueller's course notes for a pseudo code implementation of this.

4 Linear FEM

To ensure stability even with stiff materials we want to use implicit Euler. Here's the basic formula for implicit Euler for one particle with position p^t and velocity v^t at time t and p^{t+1} and v^{t+1} at time $t + \Delta t$. Mass is m and the force is $f(p)$.

$$mv^{t+1} = mv^t + \Delta t f(p^t + \Delta t v^{t+1}). \tag{8}$$

See the Mass Spring section of Mueller's course notes for more details about implicit Euler.

For multiple particles m turns into M, a diagonal matrix with particle masses. p and v turn into $3n$ sized vectors, where n is the number of particles. The issue with applying this formula is that we don't know v^{t+1} and we need to be able to solve for it. If $f(p)$ is complex this becomes very hard. If $f(p)$ is linearized it becomes much simpler and can be solved using linear system solvers.

4.1 Stiffness Matrix K

One can linearize $f(p)$ for any stress-strain by performing a Taylor expansion of $f(p)$, this involves finding the Jacobian of $f(p)$. Luckily there are 3 simplifications that we can use to linearize $f(p)$ without performing a Taylor expansion of the previous equations:

1. Use a linear strain measure like Cauchy Strain.
2. Use a linear stress-strain relationship model like Hookean Elasticity.

3. Calculate the face area and normal using the undeformed positions.

Now we have:

$$f(p) = K(p - x). \tag{9}$$

x is the undeformed positions of the vertices and K is a sparse $3n \times 3n$ matrix called the stiffness matrix. This matrix is symmetric and is constant throughout the simulation.

4.2 Constructing K

K describes how the positions of every pair of points affects the forces on those two points. It is can be thought of as an $n \times n$ matrix of 3×3 sub-matrices each relating two points i and j. We write this as $K_{i,j}$. I won't go over deriving the formula for these sub-matrices. Although the concept is the same as what we've been doing so far: Deformation gradient $\rightarrow$ Cauchy Strain Measure $\rightarrow$ Hookean Elasticity $\rightarrow$ Calculate nodal forces using face area and normal of undeformed positions. There are two simplifications we made that makes this invalid for large deformations such as rotation, specifically using Cauchy Strain and the calculating with undeformed areas + normals. Here's the formula taken from Mueller's course notes for calculating $K_{i,j}$ for nodes i and j:
First we create four vectors: y_0, y_1, y_2, and y_4 for both i and j.

$$\begin{bmatrix} y_1^{\mathrm{T}} \\ y_2^{\mathrm{T}} \\ y_3^{\mathrm{T}} \end{bmatrix} = X^{-1} = \left[x_1 - x_0, x_2 - x_0, x_3 - x_0\right]^{-1}. \tag{10}$$

$$y_0 = -(y_1 + y_2 + y_3). \tag{11}$$

Now we assemble $K_{i,j}$.

$$K_{i,j} = \begin{bmatrix} y_{i,x} & 0 & 0 \\ 0 & y_{i,y} & 0 \\ 0 & 0 & y_{i,z} \end{bmatrix} \begin{bmatrix} a & b & b \\ b & a & b \\ b & b & a \end{bmatrix} \begin{bmatrix} y_{j,x} & 0 & 0 \\ 0 & y_{j,y} & 0 \\ 0 & 0 & y_{j,z} \end{bmatrix} \tag{12}$$

$$+ \begin{bmatrix} y_{i,y} & 0 & y_{i,z} \\ y_{i,z} & y_{i,z} & 0 \\ 0 & y_{i,y} & y_{i,x} \end{bmatrix} \begin{bmatrix} c & 0 & 0 \\ 0 & c & 0 \\ 0 & 0 & c \end{bmatrix} \begin{bmatrix} y_{j,y} & y_{j,x} & 0 \\ 0 & y_{j,z} & y_{i,y} \\ y_{y,z} & 0 & y_{j,x} \end{bmatrix}. \tag{13}$$

$$a = VE\frac{1 - v}{(1 + v)(1 - 2v)}. \tag{14}$$

$$b = VE\frac{v}{(1 + v)(1 - 2v)}. \tag{15}$$

$$c = VE\frac{1 - 2v}{(1 + v)(1 - 2v)}. \tag{16}$$

$$V = det(X). \tag{17}$$

E is Young's Modulus (how stiff the material is) and v is Poisson's ratio (how volume conserving the material is). Now to construct the full stiffness matrix K we start out with a zeroed out $3n \times 3n$ matrix where n is the number of corner nodes. We go through every tetrahedron and calculate $K_{i,j}$ for every pair of the 4 nodes (including $K_{i,i}$) and add it to K beginning at row $3i$ and column $3j$.

4.3 Implicit Integration using K

Once we have K we update our implicit integration formula. This time not for one particle but the entire system. M is a diagonal $3n \times 3n$ mass matrix. v, p, and x are $3n$ vectors, and K is the stiffness matrix.

$$Mv^{t+1} = Mv^t + \Delta t(K(p + \Delta t v^{t+1} - x)). \tag{18}$$

Now with some algebra we can solve for v^{t+1}.

$$Mv^{t+1} = Mv^t + \Delta t K(p - x) + \Delta t^2 K v^{t+1}. \tag{19}$$

$$Mv^{t+1} - \Delta t^2 K v^{t+1} = Mv^t + \Delta t K(p - x). \tag{20}$$

$$(M - \Delta t^2 K)v^{t+1} = Mv^t + \Delta t K(p - x). \tag{21}$$

This is now in the form $Ax = b$ and we can solve it with a sparse symmetric matrix solver such as the Conjugate Gradient method. Once we have v^{t+1} we use the new velocities to calculate the new positions: $p^{t+1} = p + \Delta t v^{t+1}$.

5 Co-rotational FEM

After implementing Linear FEM with the previous equations you'll notice that the simulation works well until the model undergoes any rotation. This happens because Cauchy Strain is not rotationally invariant and using undeformed normals will give incorrect force when rotated. In order to fix this we separate out the rotational deformation to force the strain to be rotationally invariant. This is called *warped stiffness*.

5.1 Warped stiffness

The basics of warped stiffness involve calculating the rotational part R of the deformation gradient before we calculate the forces and then rotating the forces back into world space. The formula for the nodal forces from one tetrahedron would be

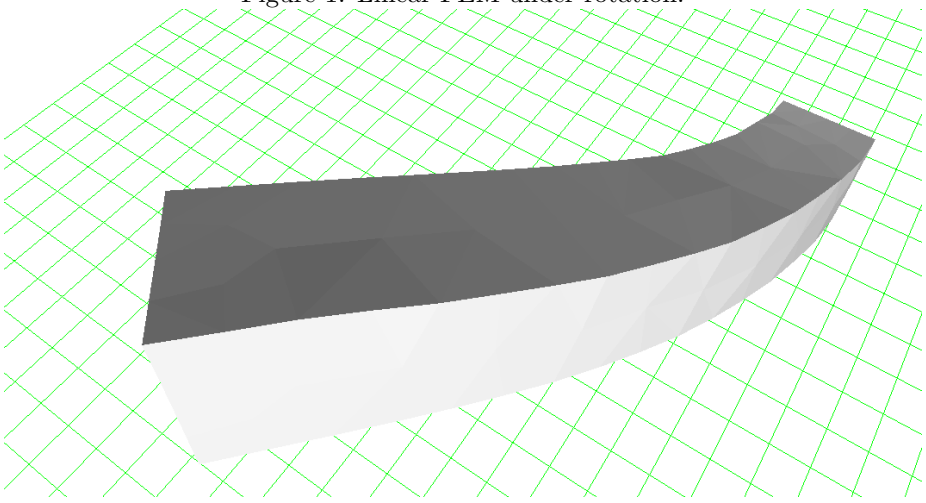
$$f(p) = R(K_e(R^{-1}p - x)) \tag{22}$$

or simplified

$$.f(p) = RK_e(R^{-1}p - x) \tag{23}$$

K_e is a 12×12 stiffness matrix relating the 4 points of the tetrahedron, composed the same way as K. p and x are the 4 deformed and undeformed points respectively and R is the rotation of tetrahedron.

Figure 1: Linear FEM under rotation.



5.2 Updated Integration Formula

Using our new force equation Eq.23 and we will rewrite Eq.21 develop a new implicit integration formula that is rotationally invariant. We start again with our implicit integration formula. We will first derive it with a single tetrahedron.

$$Mv^{t+1} = Mv^t + \Delta t R K_e (R^{-1}(p + \Delta t v^{t+1}) - x). \tag{24}$$

$$Mv^{t+1} = Mv^t + \Delta t (R K_e R^{-1} p + \Delta t R K_e R^{-1} v^{t+1} - R K_e x). \tag{25}$$

$$Mv^{t+1} - \Delta t^2 R K_e R^{-1} v^{t+1} = Mv^t + \Delta t (R K_e R^{-1} p - R K_e x). \tag{26}$$

$$(M - \Delta t^2 R K_E R^{-1}) v^{t+1} = Mv^t + \Delta t (R K_e (R^{-1} p - x)). \tag{27}$$

This is the simplest form of the equation, but we have to create less matrices in our final implementation if we define $K'_e = R K_e R^{-1}$ and use

$$(M - \Delta t^2 K'_e) v^{t+1} = Mv^t + \Delta t (K'_e p - R K_e x). \tag{28}$$

If we want to add in external forces f_{ext} the equation becomes

$$(M - \Delta t^2 K'_e) v^{t+1} = Mv^t + \Delta t (K'_e p - R K_e x + f_{ext}). \tag{29}$$

To find the rotation matrix we can use the Gram-Schmidt method or Polar decomposition. See Mueller's course notes for an efficient Gram-Schmidt like method.

Note R^{-1} can be easily calculated as R^{T} because it is a rotation matrix.

5.3 Element Assembly

We have derived equations that work with one tetrahedron but not with the entire system. Because this is a linear system this is an easy switch to make. The new stiffness matrix K', the equivalent of K'_e, now also includes the rotations of the tetrahedra and is no longer constant (although $K_{i,j}$ still is). To create this we use a new sub-matrix $K'_{i,j}$:

$$K'_{i,j} = RK_{i,j}R^{-1}. \tag{30}$$

We construct K' the same as way we construct K except with $K'_{i,j}$ instead of $K_{i,j}$.

6 Fixed Points

In order to implement points that connect with the rest of the mesh but do not move and are not affected by forces we create a separate list of nodes called fixed points. These nodes do not follow equations of motion so they do not need to be represented with a position in the stiffness matrix. These fixed points can be handled within the implicit integration schemes as follows.

6.1 Fixed Points in Linear FEM

In Linear FEM we handle fixed points by simply omitting them. They are used to create the y vectors but do not otherwise contribute to the stiffness matrix. This can be easily derived if we consider what must happen when we have masses that approach infinity and have a starting velocity of zero. Let's consider fixed point i's relationship with a non-fixed point j that share a tetrahedron. Here is Eq.21 shown again for reference.

$$(M - \Delta t^2 K)v^{t+1} = Mv^t + \Delta tK(p - x). \tag{31}$$

In Eq.21 there are two places where we must consider how i interacts. First when calculating its new velocity and second when calculating j's velocity.

6.1.1 Calculating a Fixed Point's new velocity

It is impossible to use our system of equations to *calculate* the velocity of a fixed point because fixed points violate force equals mass multiplied by acceleration. But we can model them as points with mass approaching infinity and reason how the interact with our integration scheme.

The diagonal entires in the mass matrix M approach infinity. On the right hand side the value Mv^t at index i is indeterminate because the starting velocity is zero. Arguably this evaluates to zero because the starting velocity must be at absolute zero, while the mass is a bit more flexible. There are still forces applied to i so $\Delta tK(p - x)$ at index i is non-zero but finite. On the right hand side we have M (approaching infinity) at index i minus some non-zero stiffness

matrix. Any solution that has non-zero velocities for i would have to equate the lefthand side, infinity multiplied by a finite number, and the righthand side, another finite number. Although this isn't formal logic, it gives credence to our handling of fixed point within the implicit integration equation.

6.1.2 Fixed Point's effect on j

On the right side of the equation, i would never contribute to the $k(p-x)$ term of any point because $p = x$ for all fixed points.

Let's take a look at the left side. Because i and j are both part of the same tetrahedron there is some non-zero value within the stiffness matrix at row j and column i. The new velocity of i must be zero and so this non-zero value never contributes to j's new velocity and also doesn't need to be considered.

Therefore removing a fixed point from the stiffness matrix K and the vectors v, p, and v is valid because these added values would never contribute to any velocities.

6.1.3 How does a Fixed Point contribute?

However we still care about the effects of fixed points on non-fixed points. If we don't use their positions within our integration scheme how does a fixed point interact with the simulation? The answer can be understood intuitively when we consider moving a point away from a fixed point it is connected to. When a point j gets further from its starting point, $(p-x)$ increases at index j. The diagonals of the stiffness matrix are non-zero which means that $K(p-x)$ at index j will become large and change the j's velocity unless there is some term in $K(p-x)$ that cancels it out. When a tetrahedron is translated, the canceling force comes from the other 3 points. If they are also far from their starting points in the same direction their $(p-x)$ terms will cancel out these large terns. If this were not the case our nodal forces would not be invariant to translation.

When a fixed point is a corner node of a tetrahedron the absence of a canceling value let's j's $(p-x)$ term affect the velocity of j, bringing it towards its starting point. Therefore not including the fixed points within the calculation is reasonable and produces correct results.

6.2 Fixed Points in Co-rotational FEM

If we exclude fixed points from the implicit integration equation within FEM we get instability for all nodes connected to a fixed point. This is because even though $p = x$, they do not cancel out within the implicit integraiton equation. Points at their original position contribute values to the right hand side. Here is our co-rotational implicit integration formula, Eq.28, after Element Assembly.

$$(M - \Delta t^2 K')v^{t+1} = Mv^t + \Delta t(K'p - RKx). \tag{32}$$

The same reasoning for calculating the new velocity of a fixed point can be reused but we can no longer discount i's contribution to the right hand side for

non-fixed points. Since we still want to leave fixed points out of the sparse matrix we need to derive their contribution and manually add it back in. For every point j that has non-zero entries in K at column i we use the term $(K'p - RKx)$ to find the fixed point i's individual contribution:

$$K'_{j,i}p_i - RK_{j,i}x_i. \tag{33}$$

Since i is a fixed point, $p_i = x_i$ and we have

$$(K'_{j,i} - RK_{j,i})x_i. \tag{34}$$

This contribution from fixed points creates a new term in the full equation we will call f_{fixed}. f_{fixed} is made by adding for every point j in a tetrahedron that connects a fixed point i $(K'_{j,i} - RK_{j,i})x_i$ into a $3n$ vector at index $3j$. The full equation with external forces and fixed points for Co-rotational FEM is:

$$(M - \Delta t^2 K')v^{t+1} = Mv^t + \Delta t(K'p - RKx + f_{fixed} + f_{ext}). \tag{35}$$

Note that even though we referred to a fixed point as i, the fixed points do not have an index into the stiffness matrix K or the vectors p and x. The positions of fixed points are stored separately.

7 Mesh generation using TetGen

For my implementation I used TetGen to create meshes. TetGen accepts a water-tight model with no intersecting faces and gives a tetrahedralized mesh back. To generate a tetrahedralized mesh from an existing surface mesh I first used the program MeshLab to simplify the mesh and to try and remove any intersecting faces and duplicated vertices. Although TetGen does have some mesh simplification options, MeshLab has immediate visual feedback and is easier to specify. This simplification is needed if the model is complex and the simulation is to be run at interactive frame rates. After simplifying the mesh I exported it to a text PLY file and used "-pnnqa.5" as the command line options to TetGen.

- -p Use this option to take in a piece-wise linear mesh surface (i.e. a triangulated mesh surface).
- -nn Give the tetrahedrons that border on each face. Use this to help with visualization.
- -q Enables adding new points to improve mesh quality. -q by itself uses default options specifying how to do this.
- -a.5 Use a max tetrahedron volume of .5 units cubed. With a coarse surface mesh this option determines how tessellated the mesh is. Adjust this to help get interactive frame rates.

There are three values I use for simulation that are output from TetGen: *pointlist*, *tetrahedronlist*, and *trifacelist*. pointlist gives the nodes/particles within the simulation. tetrahedronlist gives which nodes are assembled into which tetrahedra. trifacelist gives the external faces for collision detection. Be careful to note TetGen uses 1 based indexing.

8 Collision

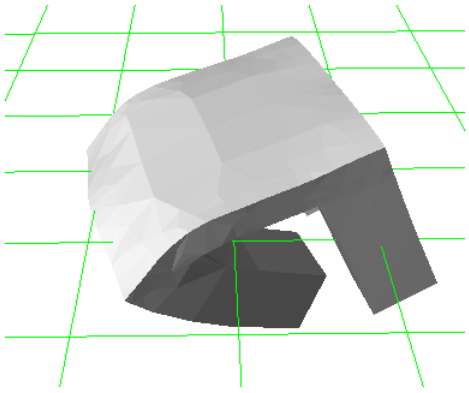
Without collision detection and response deformable solid simulations are much less visually interesting. In this section I will go over methods to implement real time collision detection and response. All the methods I present here handle each vertex collision separately. Although this introduces errors, it is more efficient and easier to implement than a global solver. First I will go over handling collisions with a plane or floor-like surface and then I will discuss collision with arbitrary meshes.

8.1 Collision with the Floor

Let's define a point i and a height h that where we want the ground be. Any point less than h in the y component is within the ground. Collision detection is easy in this case, we can simply loop over all outer points of the mesh and check if they are below h. Collision response is a bit more complex.

8.1.1 Penalty Method

Figure 2: A table colliding with the ground using the penalty method with a low k_g.



The simplest type of collision response is an explicit spring penalty force proportional to the depth of the penetration. We apply the following force to a penetrated node. Let p be the closest point to the ground's surface and k_g be a

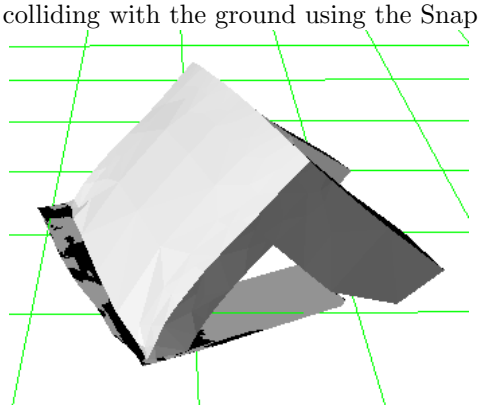
constant representing the stiffness of the penalty force.

$$f_{ground} = k_g * (p - i) * \Delta t. \tag{36}$$

$(p-i)$ is in the direction of the ground's normal. The issue with this approach is that the penetrated nodes undergo no friction and they are artificially bouncy. The visual effect can seem more like the model is floating on a geyser rather than colliding with the ground. Although making the penalty force implicit might help, it will still exert force on the nodes colliding with the ground. This creates the artificial bounce.

8.1.2 Snap to Floor

Figure 3: A table colliding with the ground using the Snap to Floor method.



Because we can easily calculate p one method is to simply displace the node i so that it no longer penetrates the ground.

$$i_{new} = p. \tag{37}$$

Although the collision no longer seems bouncy the opposite effect has taken place. It seems as though the model melts onto the ground. Once a node is forced into the ground our implicit integration scheme doesn't push other nodes away because i still has a large velocity into the ground.

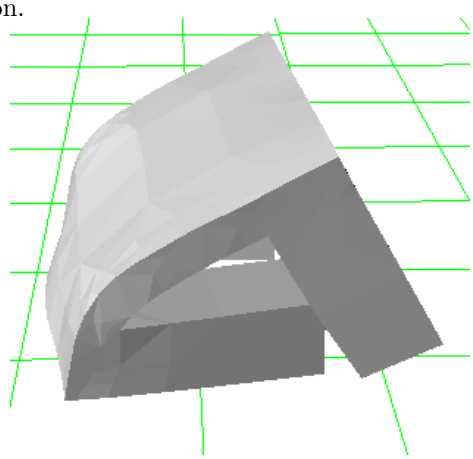
8.1.3 Snap to Floor and Infinite Friction

If we also set the velocity of a node to zero once it has penetrated the ground as well as displacing it we get a much more pleasant effect.

$$i_{new} = p, \tag{38}$$

$$v_i = 0. \tag{39}$$

Figure 4: A table colliding with the ground using the Snap to Floor method with Infinite friction.



This makes the ground seem hard and no longer let's vertices sink into it. One consequence of zeroing out every component of the velocity when a node penetrates the ground is that this emulates a large friction coefficient. This is mitigated by the fact that if a node penetrates the ground at an angle it keeps the displacement perpendicular to the ground normal, even though this motion happened after the particle intersected the ground.

I have tried snapping to the last intersection point of the ground but this causes nodes to attach themselves to the ground and refuse to move unless there is upward force. This might be considered true infinite friction because the particles are never allowed to slide on the ground. Yet this type of infinite friction doesn't look like a rough ground and instead feels as though the vertices' positions are stuck to the floor.

8.1.4 Snap to Floor and Partial Friction

If we again use the snap method but only eliminate the velocity perpendicular to the surface, we are left with a hard surface with no friction. The visual effect is close to ice where an object will slide along the ground.

Ideally to handle the interaction with the ground one could use a combination of Infinite and Partial friction and handle the motion of particle that has happened after intersection with the same formulas. For our purposes infinite friction without handling this motion is enough to model a rough ground.

8.2 Collision with a Mesh

A collision mesh behaves the same way as the ground, it doesn't move and resists penetration from the object mesh.

8.2.1 Collision Detection

Collision detection with an arbitrary triangular mesh is more complex and less efficient than an analytic model such as the floor. For this part of my implementation I tried two different collision detection libraries: SelfCCD and PQP.

SelfCCD SelfCCD is continuous collision detection library that supports detecting an object colliding with itself. Continuous collision detection is a method where we only check is if the motion to the next frame creates new intersections.

I had various issues when working with this. It turned out to be relatively slow and whenever a collision slipped through my collision response code there was no way to correct it.

PQP PQP is a non-continuous collision detection library that only detects objects colliding with other objects. Because of this, if we wanted self collisions we would have to break up models into different objects that can collide with each other. Another issue with PQP is it only gives you a pair of triangles it determines have collided and not any information such as the intersection line. It might be possible to refactor PQP to do this, but my solution was to run another collision detection algorithm between the two triangles afterwards which gave me the intersection line. Other than these issues PQP was faster and more stable for my collision response code than SelfCCD.

8.2.2 Object Vertex To Face Collisions

Once we determine that a object vertex has collided with the face of the collision mesh we can use a similar method of collision response as with the floor. We have a static surface with a normal and can apply the same technique. To do this we have to calculate the normal and project the point onto the plane.

When using SelfCCD we have to be careful about projecting the point onto the surface of the triangle. If we put the point exactly on the surface, it could be considered below the surface by SelfCCD. When this happens the next frame's collision detection will not catch this penetration because it was already beneath the surface the previous frame. To fix this we have to add a small value to the new position in the direction of the normal. PQP doesn't have this problem because there is no issue if the node is considered to be below or above after our collision response code.

8.2.3 Other types of Collisions

There are 2 other types of collisions we need to handle to fully take into account mesh to mesh collision. These are edge to edge collisions and object face to collision mesh vertex collisions. When both meshes have smooth surfaces we can ignore these collisions without visual artifacts. When our mesh has strong edges, for example a cube, These artifacts can easily be seen as parts of the meshes clipping.

I didn't cover these types of collisions within my implementation. For face to mesh vertex collisions the idea would be to find the barycentric coordinates of the collision point and move the object nodes proportional to how close they are to the collision point. In this case infinite friction does not work because the whole triangle will be frozen of just a single point. We want to change the velocities proportional to the barycentric coordinates as well.

For edge to edge collisions, it becomes bit harder. We no longer have surface normals. Instead we have two pairs of vertices whose edges have collided. One way of trying solve this is setting up least squares to move the edges to touch while trying keep as the much of the position and velocities of the vertices the same.

8.3 SelfCCD and Rollback

SelfCCD returns the time within the frame that the collision occurred. We can use this to implement a rollback scheme where we rollback to the first collision instead of trying fix every collision at the end of the frame. Once we rollback to the exact time of a collision, we only change the velocity of that vertex and run implicit integration to the end of the frame. We continue doing this until we find no collisions within the frame. This slows down the simulation immensely when we have multiple collisions per frame because we have to rerun implicit integration each time. Another issue is that resting contact causes the simulation to freeze because the vertices collide even with corrected velocities. In order to use this properly we would have to develop a resting contact model that avoids this situation.

9 Visualization

There are several visualization techniques that are useful to see the intermediate steps of the simulation. Implementing these and interactivity within the simulation can also help debug implementation errors.

9.1 Mesh Visualization

When calibrating TetGen it's helpful to see what the mesh that it generates. It's also useful to see how the individual tetrahedra perform within the simulation. A simple solution to this is to display tetrahedron edges. For coarse tessellation this works properly, but when are too many tetrahedra overlapping we cannot tell which edges belong to what tetrahedron. To help mitigate this we can color the lines based on their distances from the camera.

This still does not completely solve the problem. No natural way to display all layers of an object. Other possible solutions are to use transparency or allow slicing of the mesh to see the tetrahedra within the object.

Figure 5: A wireframe rendering of a mesh output from TetGen in the simulation

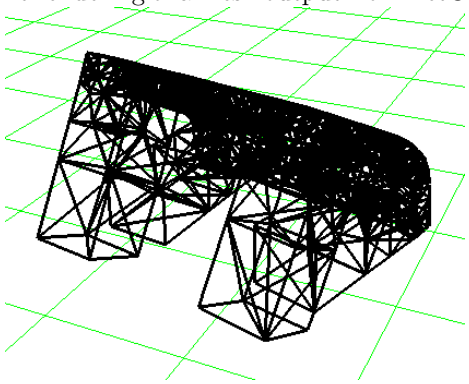
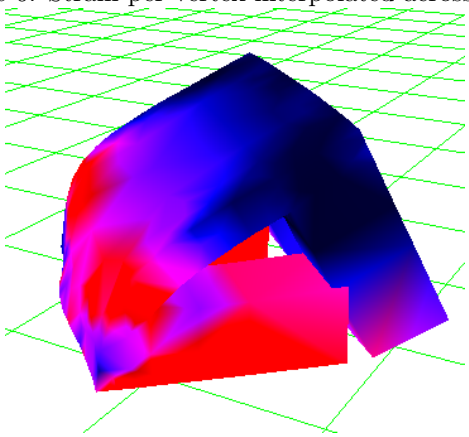


Figure 6: Strain per vertex interpolated across faces



9.2 Strain Visualization

We can visualize the strain per tetrahedron by coloring each tetrahedron face according to its strain. This creates hard edges and reveals the tessellation of the mesh. Instead we instead accumulate strain on each vertex and calculate the average strain of tetrahedra connected to the each vertex. Then we can display this strain by interpolating the color across each outer face. This looks much better and gives the illusion of a continuous material.

To properly display strain we also have to develop a scale relating colors to norm of the strain. This scale depends on what deformations we want distinguish within the mesh which usually depends on the stiffness. In my implementation I added a slider to manually adjust this value.

10 Conclusion

I'd like to thank Matthias Mueller for compiling such wonderful tutorials and my senior project advisor Shinjiro Sueda for guiding my implementation. You can view the source code for my project by going to https://github.com/patrickriordan/mass-spring.