Real-Time Soft Body Simulation Using an Approximation of the Pressure Force

Patrick Martin

Table of Contents

1 Abstract

In this paper I wish to present a method for simulating soft body objects in real time using a combination of springs and internal pressure as well as integrating with rigid body systems already prevalent in video games and related simulations. I will also present a method for performing efficient collision detection between deforming soft-body objects as well as the related impulse based response. Additionally I will propose a basic collision detection algorithm between soft and rigid objects as well as realistic and believable impulse based collision response in the soft/rigid interface. Using these impulses, I will provide a method for simulating static friction in the soft body simulation.

2 Introduction/Related Work

My goal for this simulation was to realistically model soft-body objects in real-time interactive simulations such as video games. On top of providing a model for soft-body objects, I needed to get the simulation to run as quickly as possible on modern hardware, to provide a real-time solution for collision detection, and to develop a method for interacting with rigid-body objects already prevalent throughout related simulations.

The idea of using a pressure model for soft body objects was proposed in *Pressure Model of Soft Body Simulation* (1). This served as an excellent starting point, but the published paper did not detail the areas of volume calculation and collision detection. In both of these areas, they elected to restrict themselves to bounding volumes for which the volume was already known.

To more accurately calculate the volume while maintaining efficiency, I referred to the paper *Efficient Feature Extraction For 2D/3D Objects in Mesh Representation* (2) in which Zhang and Chen proposed a fast and iterative method for computing the volume. Additionally I used the technique of *Particle-based Collision Detection* (3) which provided a basic collision detection algorithm similar to V-clip (4) that worked well with concave objects. I then expanded this algorithm to take into account features other than vertices and use geometry intersection algorithms to get more accurate collision detection and responses.

3 Soft Body Simulation

3.1 Soft Model

Soft body objects are represented as a point-mass membrane completely enclosing a finitely sized volume similar to (1). The behavior of the object is primarily dependent on the spring force and the pressure force.

3.1.1 Surface Spring Network

The surface membrane consists of various triangles with a point mass at each vertex. The springs are all governed by Hooke's Law, which states that $\overrightarrow{F_x} = k(x - x_0)\hat{n}$ where k is the spring constant, x is the current length, x_0 is the resting length, and $\hat{n}$ is the direction of the spring.

3.1.2 Pressure Simulation

Pressure is simulated using the law of ideal gases which states that $PV = nRT$ where p is the pressure, V is the volume, n is the number of moles, R is the universal gas constant, and T is the absolute temperature. The pressure force is expressed as $\overrightarrow{F_p} = P\hat{n}A$, with $P = V^{-1}nRT$, $\hat{n}$ is the surface normal, and A is the area the pressure is being applied to.

For this simulation, $\hat{n}$ is the normal at the surface. In this case I use the normal of each triangle on the surface. Given a triangle defined by the counter clockwise vertices a, b, and c, I can define two edges $e_0 = b - a$ and $e_1 = c - a$. The normal is then $\hat{n} = e_0 \times e_1$ and the area is $A = \frac{1}{2}\|e_0 \times e_1\|$.

Finally, the volume is calculated iteratively for each mesh (2). To do this, I consider the tetrahedron formed between each triangle on the mesh and a random point in space, which I choose as the origin for simplicity. The volume of this tetrahedron is known as $V = \frac{1}{6}|-c_x b_y a_z + b_x c_y a_z + c_x a_y b_z - a_x c_y b_z - b_x a_y c_z + a_x b_y c_z|$. If the chosen point lies outside of the mesh (as is likely), this volume is added to the running total if the triangle's normal faces the point, or subtracted from the total if the normal faces away. Using the origin as the fourth point in the tetrahedron, this can be done by taking the dot product of the surface normal and the position vector of any vertex on the triangle. In practice, rather than taking the absolute value of the volume and changing its sign based on orientation it is possible to get identical results by simply negating the volume in each step. For example: $V_{total} = V_{total} - \frac{1}{6}(-c_x b_y a_z + b_x c_y a_z + c_x a_y b_z - a_x c_y b_z - b_x a_y c_z + a_x b_y c_z)$.

As a final point of interest, if the vertices of the mesh are not evenly distributed some edges may appear stiffer than others or the model may appear lopsided. Changing the spring force through the model will result in an unstable simulation. This is known as the stiff problem resulting from the springs failing to oscillate in frequency with one another. Instead, scaling the pressure force by $\frac{number\ \ of\ springs}{average\ \ number\ \ of\ springs}$ before applying the pressure to each vertex improves the visual issues this may cause (5).

3.1.3 Damping

I found that having many stiff springs could quickly become unstable even running RK4 at 60Hz. I therefore have two damping variables to correct this behavior expressed as $\rho = \rho_g + \rho_k$. ρ_g is the global damping applied to all objects in the simulation, and ρ_k is scaled based on the strength of the spring by $\rho_k = (1 - \rho_g)\frac{k}{k_{max}}$ where k is the spring constant. The $(1 - \rho_g)$ term ensures that ρ never exceeds 1. Using this damping, the final velocity of a point under the effects of a spring is $\vec{v}' = \vec{v} - \rho\vec{v}$. In my simulation I found that a suitable ρ_g is 0.01 and a suitable k_{max} is 400 through trial and error, although it should be noted that my springs rarely exceed $k = 300$.

3.1.4 Friction

To calculate friction, I first wait for a vertex of the soft object to drop below the ground plane. The impulse, j, is computed from the collision and move the vertex back to the surface. The object's velocity is then projected to the ground plane as $\overrightarrow{v_\parallel} = \vec{v} - \hat{n} \cdot \vec{v} \cdot \hat{n}$. Friction is therefore calculated as $\overrightarrow{v_f} = -\mu \widehat{v_\parallel} \max(j, 0)$ where μ is the friction coefficient, and is added to the final velocity.

3.2 Runge-Kutta 4

I use a Runge-Kutta 4 integrator to compute the velocity in each frame, followed by an implicit Euler integrator for position. I first get an estimation of acceleration by incrementing the velocity by the acceleration during the time step and use that to update the position so that I could calculate the pressure and spring forces. This acceleration is then stored and I repeated the process. If $F(x)$ is the total force at position x, m is mass, and x_0 and v_0 are the starting position and velocity respectively, then the whole process proceeds as:

$$a_0 = \frac{F(x_0)}{m}, v_1 = v_0 + \frac{a_0 t}{2}, x_1 = x_0 + \frac{v_1 t}{2}$$

$$a_1 = \frac{F(x_1)}{m}, v_2 = v_0 + \frac{a_1 t}{2}, x_2 = x_0 + \frac{v_2 t}{2}$$

$$a_2 = \frac{F(x_2)}{m}, v_3 = v_0 + a_2 t, x_3 = x_0 + v_3 t$$

$$a_3 = \frac{F(x_3)}{m}$$

The final acceleration is then $a = \frac{a_0 + a_3 + 2(q_1 + a_2)}{6}$. Velocity follows as $v = v_0 + at$, and position as $x = x_0 + vt$. This integrator must be applied to each vertex in the soft body object independently and every vertex in any single object must be acted on at the same time. For example, if one vertex is at its x_0 position and another at its x_3 position, then the spring and pressure forces will be computed incorrectly between these vertices resulting in visibly incorrect behavior.

4 Rigid Body Simulation

4.1 Rigid Model

Every rigid object in my simulation stores a mass m, inertia $\overleftrightarrow{I}$ and inverse inertia $\overleftrightarrow{I^{-1}}$ tensors, a quaternion orientation q, velocity $\vec{v}$, angular velocity pseudo-vector $\vec{\omega}$, and position $\vec{p}$. Since I know all my rigid objects are cubes, $\overleftrightarrow{I}$ is precomputed as $\overleftrightarrow{I} = \begin{bmatrix} \frac{m}{12}(w^2 + h^2) & 0 & 0 \\ 0 & \frac{m}{12}(h^2 + d^2) & 0 \\ 0 & 0 & \frac{m}{12}(w^2 + h^2) \end{bmatrix}$ where w, h, and d are the width, height, and depth respectively. The inverse then can be easily calculated by taking the reciprocal of each diagonal element.

The rigid body objects update is implemented using an Euler-Cromer integrator for both orientation and position. The update loop for position with an acceleration of $\vec{a}$ then becomes $\vec{v} {+}{=} \vec{a}t, \vec{p} {+}{=} \vec{v}t$. For quaternions I compute an axis of rotation as $\hat{\omega}$ and a rotation amount as $\theta = \|\vec{\omega}\| t$, and update q as $q' = \left[\cos\left(\frac{\theta}{2}\right), \sin\left(\frac{\theta}{2}\right)\hat{\omega}\right] \cdot q$.

5 Collision Detection

5.1 Broad Phase

Broad phase collision detection is performed using bounding spheres. Although not ideal, a reasonably fitted sphere can be computed by finding the axis aligned bounding box of a mesh and constructing a sphere about the center of the box that encloses all corners.

5.2 Narrow Phase

5.2.1 Soft Vs. Soft

Soft objects present an interesting challenge with regard to collision detection. I chose to implement a closest features algorithm that attempts to find the extrema on objects using a particle system (3) that loosely imitates the electromagnetic force.

These particles are initially randomly distributed across the surface of each soft body and are limited to resting in the middle of a surface feature. In the case of this simulation, a particle may only rest on a vertex, the center of an edge, or the center of a surface triangle. The physical position of each particle can then be calculated by averaging the positions of the vertices belonging to the feature the particle is resting on.

To update these particles and eventually find the closest features, I calculate the cost of a particle to stay where it is as well as the cost to move to each neighboring feature. The cost is calculated by first incrementing the cost by $\frac{1}{distanc\ e^2}$ for each particle belonging to the same object as the particle I am preparing to move, and then decremented by $\frac{1}{distanc\ e^2}$ for each particle belonging to the object being collided with. The particle then migrates to the cheapest neighboring feature, or stays where it is if the current location is currently the cheapest.

Once the particles find stable positions, I iterate through each pair and first determine if they are close enough to be considered colliding before performing a feature intersection test.

In my system, each object is allocated a pre-set number of particles. I chose to generate 1/8th as many particles as an object had vertices, although ideally there should be enough particles to place one on each convex feature of an object. I store a separate list of particles for each pair of soft body objects to minimize the number of iterations when updating the particles since the closest features of any two objects rarely change between frames.

Additionally, I found that the force attracting the particles should be twice as strong as the force repelling them. This seems counter-intuitive at first as it would seem that it is desirable to force the particles to spread out to prepare for future collisions outside the current region of interest. In practice, having the particles coalesce causes more collisions to be detected earlier minimizing the amount of visible interpenetration. Additionally, I only artificially separate a collision between a vertex and a triangle, so having the particles congregate increases the chance of this particular interaction to manifest itself and leads to a believable visual response.

Unfortunately the particle update cycle is a slow process, so it would therefore be ideal to minimize the number of times it executes. In practice, a particle usually does not move to a new feature between frames so terminating the update loop if no particles move leads to a significant speed improvement. Unfortunately, it is common for a few particles to oscillate between features and is therefore necessary to terminate if fewer than four particles are moving (this number was determined purely through experimentation). Since this simulation is intended to be used in a real-time simulation, it is reasonable to restrict the number of particle refinement loops even farther since there will be a very small amount of time between frames. I find that a maximum of 10 updates per frame yields accurate closest feature results before the objects are actually interpenetrating.

5.2.1.1 Feature Tests

Once two features are paired, the type of feature the particle is resting on is determined and used for collision tests. Each feature has a skin thickness defining how far away they can be from each other and still be considered in collision. In my simulation I found a thickness of 0.5 units worked best (my basic objects had a starting radius of 3 units before inflating). For all of my tests I always choose the normal of the highest order feature for collision resolution. If both features are identical (i.e. two triangles), I use the normal of the second one purely for simplicity reasons.

5.2.1.1.1 Vertex-Vertex Feature Test

Two vertices are considered in collision if they are closer than $2 \cdot Skin\,Thickness$ from each other. The normal of collision is calculated by averaging the normal of all the neighboring triangles.

5.2.1.1.2 Vertex-Edge Feature Test

Vertex-edge collision is treated as sphere vs. capsule. Given the two endpoints of the edge $\vec{e_0}$ and $\vec{e_1}$ as well as the vertex $\vec{v}$, I can define the edge's axis as $\vec{e} = \vec{e_1} - \vec{e_0}$ and the projection of the vertex to this point as $v_p = (\vec{v} - \vec{e_0}) \cdot \hat{e}$ with the corresponding position of this projection defined as $\vec{v_p} = \vec{e_0} + v_p \cdot \hat{e}$. Next, I compute the distance of the vertex from this projected point as $d = \left\|\vec{v} - \vec{v_p}\right\|$, and can break out if this distance is greater than the sum of the skin thicknesses. If the vertex is close enough, I first check to see if $0 \leq v_p \leq \|\vec{e}\|$ which would indicate that the projection is placed within the finite bounds of the cylinder. If this fails I fall back to a vertex-vertex test for each edge endpoint to take into account the capsule's caps.

The normal is calculated at the edge by linearly interpolating between the normals at each end point by v_p. It would be ideal to simply average the normals of the two triangles connected at this edge.

5.2.1.1.3 Vertex-Triangle Feature Test

For vertex-triangle collision tests I check the distance of the vertex from the triangle and if it is close enough I check to see if the vertex projected to the triangle lies within the triangle. If not, I fall back to three edge-edge tests then three vertex-vertex tests. The normal of the collision can trivially be taken as the triangle's normal.

The algorithm is given a triangle defined by the three counter-clockwise vertices $\vec{t_0}$, $\vec{t_1}$, and $\vec{t_2}$, the triangle's normal which can easily be calculated as $\vec{n} = \left(\vec{t_2} - \vec{t_0}\right) \times \left(\vec{t_1} - \vec{t_0}\right)$, and the colliding vertex $\vec{v}$. A projected vertex can trivially be found as $\overrightarrow{v_p} = \vec{v} - (\vec{v} - \overrightarrow{t_0}) \cdot \hat{n}$ with the distance from the triangle easily calculated as $d = |(\vec{v} - \overrightarrow{t_0}) \cdot \hat{n}|$.

If the calculated d is close enough for collision, I then perform a point in triangle test with $\overrightarrow{v_p}$. For this, I compute six more vectors:

$$\vec{a} = \overrightarrow{t_0} - \overrightarrow{v_p}$$

$$\vec{b} = \overrightarrow{t_1} - \overrightarrow{v_p}$$

$$\vec{c} = \overrightarrow{t_2} - \overrightarrow{v_p}$$

$$\vec{p} = \vec{b} \times \vec{c}$$

$$\vec{q} = \vec{c} \times \vec{a}$$

$$\vec{r} = \vec{a} \times \vec{b}$$

A point is then considered to be inside a triangle if and only if $\vec{p} \cdot \vec{q} < 0$ and $\vec{p} \cdot \vec{r} < 0$.

5.2.1.1.4 Edge-Edge Feature Test

For edge-edge tests I actually check for a line segment colliding with a capsule of thickness $2 \cdot Skin\ Thickness$, and fall back to vertex vertex collision if this fails. Therefore, I consider the line segment defined by vertices $\vec{A}$ and $\vec{B}$ colliding with the capsule defined by vertices $\vec{P}$ and $\vec{Q}$ and capsule radius $r = 2 \cdot Skin\ Thickness$. I further compute $\vec{n} = \vec{B} - \vec{A}$, $\vec{m} = \vec{A} - \vec{P}$, and $\vec{d} = \vec{Q} - \vec{P}$. I then need to solve the quadratic equation $t = \frac{-b \pm \sqrt{b^2 - ac}}{a}$ (note, the term b was preceded with a 2 which factored out the 4 in 4ac and 2 in the denominator) with $a = (\vec{d} \times \vec{n})^2$, $b = (\vec{d} \times \vec{m}) \cdot (\vec{d} \times \vec{n})$, and $c = (\vec{d} \times \vec{m})^2 - r^2 \vec{d}^2$. If there are no real roots then the edges are not in contact, if there is one real root then that is considered the closest point to the capsule, otherwise I consider the closest point to be in the middle of the two real solutions for t. After finding this point, I can plug it into the normal vertex-edge test.

5.2.1.1.5 Edge-Triangle Feature Test

In edge-triangle collision detection, I first try to intersect a line segment representing the edge with the plane of the triangle and plug the result into my vertex-triangle test. If the closest point is outside the bounds of the edge, I take the closest endpoint and plug that into the vertex-triangle test. I ignore the case of an edge being perfectly parallel to the triangle.

I consider the triangle defined by the points $\overrightarrow{t_0}$, $\overrightarrow{t_1}$, and $\overrightarrow{t_2}$ with normal $\vec{n} = (\overrightarrow{t_2} - \overrightarrow{t_0}) \times (\overrightarrow{t_1} - \overrightarrow{t_0})$, and the edge defined by endpoints $\overrightarrow{e_0}$ and $\overrightarrow{e_1}$. Then the time of intersection is $t = \frac{(\overrightarrow{t_0} - \overrightarrow{e_0}) \cdot \hat{n}}{(\overrightarrow{e_1} - \overrightarrow{e_0}) \cdot \hat{n}}$, gives the position as $\vec{v} = \overrightarrow{e_0} + t \cdot (\overrightarrow{e_1} - \overrightarrow{e_0})$. If $t < 0$ then I use $\overrightarrow{e_0}$ for collision and similarly if $t > 1$ I use $\overrightarrow{e_1}$.

5.2.1.1.6 Triangle-Triangle Feature test

For triangle-triangle tests, I simply run every edge and every vertex of one triangle against the other and vice versa until either I've exhausted every feature or I've found a collision.

5.2.2 Soft Vs. Rigid

For soft rigid collision resolution I only consider vertex intersections. I perform the collision detection in two stages, first checking to see if any vertex in the soft body object lies within the rigid then again checking if a vertex of the rigid body object lies within the soft object. In both cases, I can find a minimum penetration and the corresponding triangle, and use that triangle's normal for the collision normal.

Checking for a soft vertex in a rigid object is trivial since the rigid object is guaranteed to be convex in my simulation. Since every triangle stores an outward-facing normal $\hat{n}_i$, a point on the triangle t_i(this can be any one of the vertices), and given a colliding vertex v, a vertex can be said to be inside an object if for every triangle in the rigid object $\left(\vec{p}-\vec{t_i}\right)\cdot\hat{n}_i < 0$. The minimum collision depth is just the largest scalar value from $\left(\vec{p}-\vec{t_i}\right)\cdot\hat{n}_i$ if the intersecting test passes for all triangles.

The rigid object in soft body test is slightly more complicated. I take advantage of a feature of the Jordan Curve Theorem (6), mainly that if I cast a ray from a point in any direction and it intersects an even number of polygons then that point lies within the object being tested, otherwise it lies outside.

I therefore loop through each vertex of the rigid body object $\vec{v_i}$ and test against each triangle defined by vertices $\overrightarrow{t_{j,0}}$, $\overrightarrow{t_{j,0}}$, and $\overrightarrow{t_{j,0}}$ as well as triangle normal $\hat{n}_j$. To reduce the computational complexity, I choose a ray direction of $\vec{d}=\begin{bmatrix}0\\0\\1\end{bmatrix}$. The point at which this ray intersects the triangle's plane is $\vec{p}=\vec{v_i}+t\cdot\vec{d}$ where $t=\frac{\left(\overrightarrow{t_{j,0}}-\vec{v_i}\right)\cdot\hat{n}}{\vec{d}\cdot\hat{n}}$. This point can then be plugged into the point in triangle calculation with $\vec{a}=\vec{t_0}-\vec{p}$, $\vec{b}=\ \vec{t_1}-\vec{p}$, $\vec{c}=\vec{t_2}-\vec{p}$, $\vec{p}=\vec{b}\times\vec{c}$, $\vec{q}=\vec{c}\times\vec{a}$, and $\vec{r}=\vec{a}\times\vec{b}$ and a point considered in the triangle if $\vec{p}\cdot\vec{q}<0$ and $\vec{p}\cdot\vec{r}<0$.

5.2.3 Rigid Vs. Rigid

Rigid vs. rigid collision uses the separating axis test to determine collisions. The separating axis test simply loops through a preset list of axes and doesn't return a collision if there is any axis on which the two objects do not overlap. For axes, I use each triangle normal of each object, then the cross product between every edge on one object against every edge on the second. The separating axis test could be further improved by ignoring redundant and co-planar faces.

5.3 Resolution

Collision resolution in every case is implemented using impulses. The general case impulse is calculated as $j=\frac{-(\varepsilon+1)\cdot\overrightarrow{v_{rel}}\cdot\hat{n}}{\frac{1}{m_0}+\frac{1}{m_1}+\hat{n}\cdot\left[\left(I_0^{-1}\cdot(\vec{r_0}\times\hat{n})\right)\times\vec{r_0}+\left(I_1^{-1}\cdot(\vec{r_1}\times\hat{n})\right)\times\vec{r_1}\right]}$ where ε is the elasticity of the collision, $\overrightarrow{v_{rel}}=\vec{v_0}-\vec{v_1}$, $\hat{n}$ is the collision normal from the narrow phase, $\overrightarrow{v_{0,1}}$ each object's velocity, $I_{0,1}^{-1}$ each object's inverse inertia, and $\overrightarrow{r_{0,1}}$ each object's vector from the center of mass to the point of collision.

This impulse is then used to calculate each object's linear and angular velocity change as:

$$\overrightarrow{\Delta v_0} = \frac{j \cdot \hat{n}}{m_0}$$

$$\overrightarrow{\Delta v_1} = \frac{-j \cdot \hat{n}}{m_1}$$

$$\overrightarrow{\Delta \omega_0} = [\overrightarrow{r_0} \times (j \cdot \hat{n})] \cdot I_0^{-1}$$

$$\overrightarrow{\Delta \omega_1} = [\overrightarrow{r_1} \times (j \cdot \hat{n})] \cdot I_1^{-1}$$

For all rigid objects, I store the impulse in the object's local coordinate system. In order to ensure correct behavior of rigid objects, the inertia must be translated into world coordinates. I simply use the model-world rotation matrix R and set $I^{-1'} = RI^{-1}R^T$ (I^{-1} because impulse uses the inverse inertia).

5.3.1 Soft-Soft

Collisions between soft objects present a particularly interesting issue. Objects interpenetrating but moving away from each other can create an impulse actually pulling them towards each other. Under normal circumstances, this can trivially be avoided by ignoring impulses that are directed away from the object's center compared to the collision normal. This will not work in cases such as an object colliding with the center of the torus in my demo, since all valid impulses should be pointing towards the center along the collision vector.

To resolve this, I first impose a restriction to the collision normals specifying that each normal must point away from a higher order face (triangle being the highest) and towards a lower-order face (vertex being the lowest). If both features are of the same order then I specify that the normal must point away from the second object passed into my resolution step. This should come by default from the narrow phase collision detection algorithms I mentioned above. I can then specify that j must always be positive. Since j is multiplied with $\hat{n}$, the impulse applied to the first object should always be in the positive $\hat{n}$ direction and therefore automatically exclude any impulses applied in the wrong direction. A nice feature, though, of pure soft collisions is that there is no rotation component to the impulse so the impulse equation can be simplified to $j = \frac{-(\varepsilon+1)\overrightarrow{v_{rel}} \cdot \hat{n}}{\frac{1}{m_0}+\frac{1}{m_1}}$.

Similarly, objects may already be intersecting when a collision is detected. It would then be imperative to move intersecting features so that the objects are no longer intersecting by the end of the collision resolution step. I found that the easiest solution is just to move the vertex away from the triangle by the specified skin thickness. Thanks to the spring force, I find that this results in visibly correct behavior of the simulation when a large number of potential intersection is detected, which is one of the reasons why I weight the attraction force of particles stronger than the repelling force. I chose to resolve a collision with a triangle since there is a physical plane and trivial axis (the normal) to separate along. Further, I chose to use the vertex instance since in a vertex triangle intersection the vertex is guaranteed to lie within the Veroni region of the triangle. An edge or triangle may have endpoints far outside the range of another triangle which results in visibly incorrect behavior if they are simply moved along the triangle normal.

For impulse based collision resolution it is still necessary to provide a mass to compute an impulse. I use sum of the vertex masses for each object involved. Therefore, a triangle's mass is the sum of all three of its vertices, an edge's the sum of each endpoint's mass, and a single vertex only provides its own mass. It may then take several frames to fully reverse the velocity of a soft body, but this should be expected.

5.3.2 Soft-Rigid

Similar to soft-soft intersections, I only consider the vertices in collision on the soft body object when computing the impulse, but I do use the mass of the entire rigid object. This means that if a soft and rigid body have the same mass, then the rigid object will bounce less and the vertices under collision on the soft object bounce more, but the pressure force counteracts this behavior leading to a realistic interaction. Since all collisions are essentially vertex triangle in nature, I correct interpenetration the same way I would in triangle vertex collision in soft-soft interactions.

5.3.3 Rigid-Rigid

I use the axis of minimal penetration as my collision axis. I correct for interpenetration by simply moving the objects away from each other by the collision depth. I do not restrict the impulse direction when resolving rigid-rigid collisions.

6 Results

6.1 System Specifications

- Amd X2 4600+
- 2GB DDR2 RAM
- nVidia GeForce 8600 GTS

6.2 Geometry Tests

- Sphere: 8 stacks x 8 slices
- Torus: 16 stacks x 16 slices
- Cube: 6 faces, 2 triangles each

6.3 Benchmarks

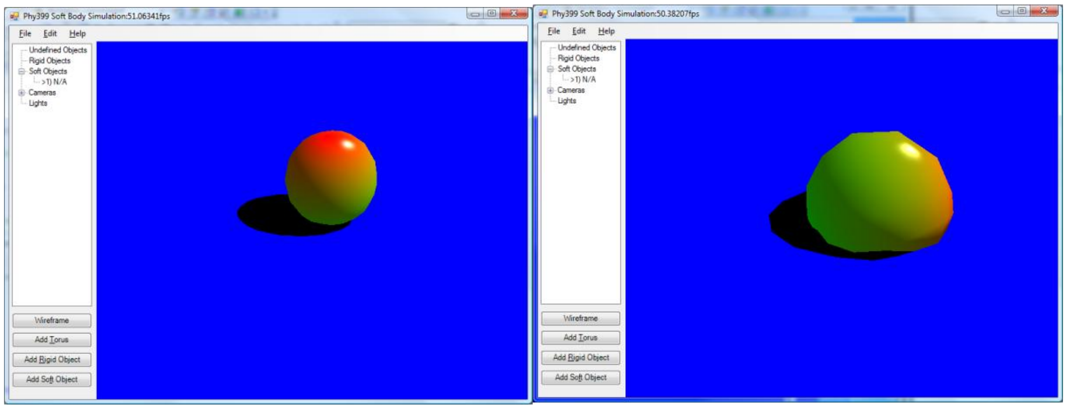
All benchmarks attained by visually monitoring a framerate counter updated every frame.

- Single soft body sphere: ~60fps
- Single soft body torus: ~45fps
- Two soft body spheres: ~50fps
  - In contact: ~30fps
- Three soft spheres: ~30fps
  - in contact: ~20fps

- Soft sphere and torus in constant contact (sphere in middle): ~20fps
- Soft torus with four rigid cubes in middle: ~18fps

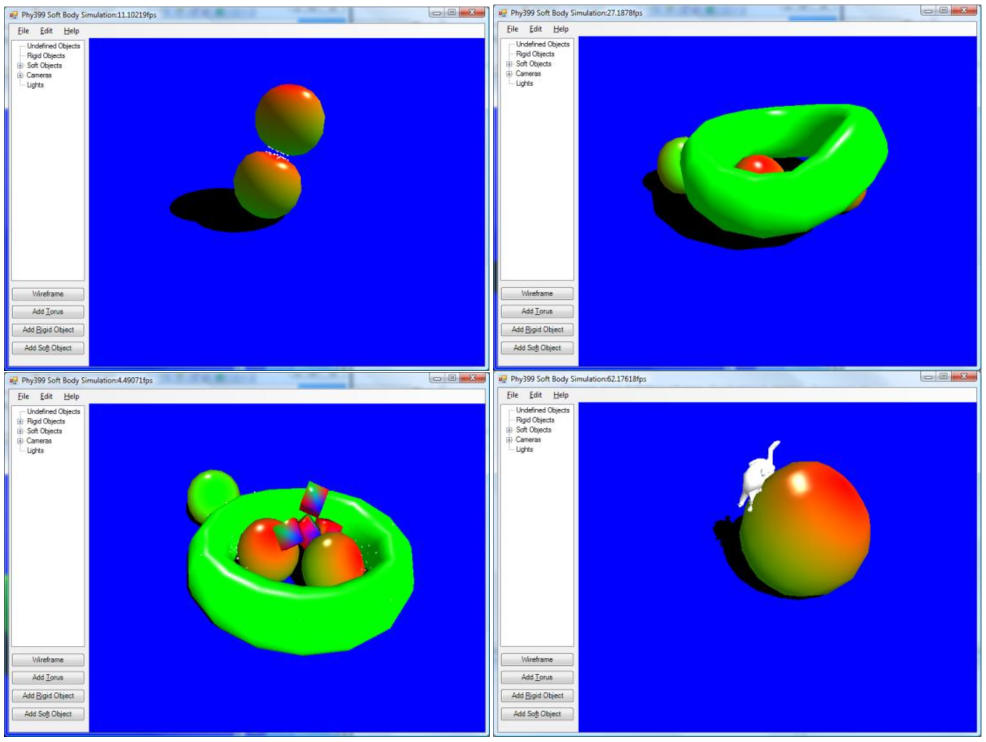
6.4 Pressure Model

Sphere at n=100 k=150 (left) and n=1 k=10 (right).



6.5 Collision Detection

Two soft spheres in collision (top left) displaying collision particles, three soft spheres and a soft torus in contact (top right), three spheres soft spheres, a soft torus, and four rigid cubes (bottom left) displaying collision particles, and a soft cat model colliding with a soft sphere (bottom right).


Phy399 Soft Body Simulation:11.10219fps
Phy399 Soft Body Simulation:27.1878fps
Phy399 Soft Body Simulation:4.49071fps
Phy399 Soft Body Simulation:62.17618fps
File Edit Help
Undefined Objects
Rigid Objects
Soft Objects
Cameras
Lights
Wireframe
Add Torus
Add Rigid Object
Add Soft Object

7 Future Research

Some immediate possible improvements include better procedural generation of collision points, accurately computing the number of particles required to cover the entire convex surface of an object (i.e. center of the torus) or scaling the number of particles based on various metrics such as the number of particles that actually are generating intersections. Additional areas of interest would be creating a spatial subdivision algorithm that could be implemented in real-time to augment or replace the particle based system and alternative particle distribution functions to ensure collisions are not missed when particles coalesce.

Other physically interesting areas to explore would be simulating viscosity, allowing for physical splitting and joining of soft bodies, and the simulation of various surface properties such as friction between soft bodies.

8 Works Cited

1. **Matyka, Maciej and Ollila, Mark.** Pressure Model of Soft Body Simulation. [Online] November 20, 2003. [Cited: April 13, 2009.] http://www.ep.liu.se/ecp/010/007/ecp01007.pdf.

2. **Zhang, Cha and Chen, Tsuhan.** Efficient Feature Extraction For 2D/3D Objects in Mesh Representation. [Online] May 18, 2001. [Cited: April 13, 2009.] http://amp.ece.cmu.edu/Publication/Cha/icip01_Cha.pdf.

3. *Particle-based Collision Detection.* **Savchenko, Vladimir, et al.** Spain, Granada : s.n., 2003. Short papers proceedings of Eurographics EG2003.

4. **Mirtich, Brian.** V-Clip: Fast and Robust Polyhedral. *Mitsubishi Electric Research Laboratories.* [Online] December 1997. [Cited: April 25, 2009.] http://www.merl.com/papers/docs/TR97-05.pdf.

5. *A Fluid Based Soft-Object Model.* **Nixon, Daniel and Lobb, Richard.** July/August 2002, IEEE Computer Graphics and Applications, vol. 22, no.4, pp. 68-75.

6. **Ericson, Christer.** Real Time Collision Detection. *Real Time Collision Detection.* San Francisco : Morgan Kaufmann Publishers, 2005, 5.4, p. 203.