Animation & CGI Motion: Krusty's Crazy Collisions

Theme 2 Milestone 2

Academic Honesty Policy

You are permitted and encouraged to discuss your work with other students. You may work out equations in writing on paper or a whiteboard. You are encouraged to use the discussion boards to converse with other students, the TA, and the instructor.

HOWEVER, you may NOT share source code or hardcopies of source code. Refrain from activities or the sharing of materials that could cause your source code to APPEAR TO BE similar to another student's enrolled in this or previous years. We will be monitoring source code for individuality. Source code should be yours and yours only. Do not cheat.

1 Introduction

In Theme 2, you will first learn about a collision detection method to prevent tunneling, and the learn about a new way of resolving collisions geometrically by treating the colliding particles as if they were a rigid body. The penalty method, geometric method, impulse method, and continuous-time collision detection will then be combined into one robust, hybrid solution to collision detection and response. Test scenes for this milestone are located in a new directory, *assets/t2m2*.

2 New XML Features

This milestone adds the following new feature:

1. The *collision* node's functionality has been expanded:

   ```
   <collision type="hybrid" maxiters="10" k="100" thickness="0.1"/>
   ```

   The valid values for *type* are "none", "simple", "continuous-time", "penalty", and "hybrid"; you will be implementing the continuous-time detection and the hybrid method in this milestone. In addition to the attributes described in Milestone 1, *collision* has the new optional attribute *maxiters* specifying the maximum number of times to try applying impulses before switching to a geometric failsafe, as explained in section 3.7 below. This attribute has no effect unless *type* is "hybrid".

3 Required Features for Milestone 2

3.1 Dealing With Tunneling

In Milestone 1 we implemented very simple collision detection: at the end of every simulation step, we calculated the distances between objects to check if they were overlapping; if so, we checked their relative velocity in the normal direction to see if they were approaching. If two objects were both overlapping and approaching, a collision was said to have occured.

Although this algorithm will do in a pinch when objects move slowly and/or the simulation time step is small, it is not very robust. In particular, it is possible for objects to move far enough during a single time step to tunnel completely through a second object; this collision cannot be detected without taking into account the motion of the particle throughout the entire time step, and not just its position at the end of the time step. Figure 1 illustrates this tunneling.

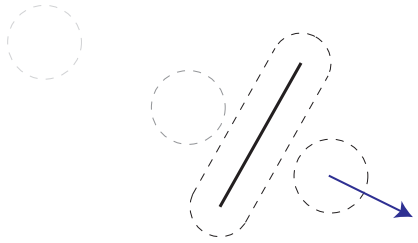

Figure 1: Simply checking for collisions at the end of the time step misses some collisions, such as this situation where a particle with high velocity *tunnels* through an edge.

Suppose we have the positions of objects at the start of a time step ($\mathbf{q}^s$) and the end of the time step ($\mathbf{q}^e$). We assume that the particles and edges moved in straight lines between the two time steps; in other words, that the positions between the two time steps are given by the continuous function $\mathbf{q}(t) = \mathbf{q}^s + t\Delta\mathbf{q}$, where $\Delta\mathbf{q} = \mathbf{q}^e - \mathbf{q}^s$. The parameter t can be thought of as the time since the start of the time step, where for convenience time has been renormalized so that the time step has duration 1.

To prevent tunneling, we must ask ourselves, **"is there any time t in $[0, 1]$ when the two objects are (a) overlapping, and (b) approaching?"** Answering this question is called performing *continuous-time collision detection*, since we are looking at all values of t instead of just $t = 1$.

For each pair of objects O_1, O_2 (with radii r_1 and r_2) we derived in Milestone 1 formulas for the shortest vector $\mathbf{n}$ between them. These formulas depended only on the end-of-time-step positions $\mathbf{x}^e$ of the objects; we can thus write down the shortest vector $\mathbf{n}(t)$ as a function of time by replacing $\mathbf{x}^e$ with $\mathbf{x}(t) = \mathbf{x}^s + t\Delta\mathbf{x}$ in these formulas. Question (a) then boils to, "is there some time t in $[0, 1]$ when $|\mathbf{n}(t)| < r_1 + r_2$?" Similarly, question (b) can be rephrased as, "is there some time t in $[0, 1]$ when the relative velocity $\mathbf{r}(\Delta\mathbf{x})$ of the two objects along $\mathbf{n}(t)$ is positive?"

Performing continuous-time collision detection is thus equivalent to checking whether the *simultaneous system of inequalities*

$$\begin{aligned} \sqrt{\mathbf{n}(t) \cdot \mathbf{n}(t)} &< r_1 + r_2 \\ \mathbf{r}(\Delta\mathbf{x}) \cdot \mathbf{n}(t) &> 0 \end{aligned}$$

has a solution between $t = 0$ and $t = 1$. Solving such systems of inequalities is in general very hard. One thing we can do to make our lives a little bit easier is to square both sides of the first inequality, getting rid of the square root; this squaring is permitted since both sides of the inequality must be positive.

$$\begin{aligned} \mathbf{n}(t) \cdot \mathbf{n}(t) &< (r_1 + r_2)^2 \\ \Delta\mathbf{x} \cdot \mathbf{n}(t) &> 0. \end{aligned}$$

Later in these milestone instructions we will see that for particle–particle, particle–edge, and particle–halfplane collisions, these equations can be written as *polynomial* inequalities, which are much easier to solve than general non-linear inequalities.

3.2 Systems of Polynomial Inequalities

Suppose we have some polynomials $f(t), g(t)$ and we want to find a simultaneous solution to

$$\begin{aligned} f(t) &\geq 0 \\ g(t) &\geq 0. \end{aligned}$$

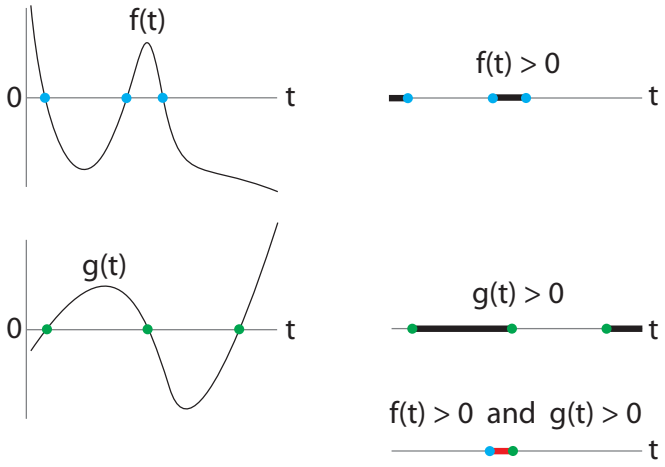

Figure 2: Solving the system of inequalities $f(t) > 0, g(t) > 0$ for two example polynomials. Left: Plots of the two polynomias. Right: The intervals of t on which f and g are positive, and the intersection (bottom, in red) of the two sets of intervals. Any point inside a red interval is a solution to the system of inequalities.

Here's the algorithm. We take one of our polynomials, f, and find all of its roots. We know that between consecutive roots, f must have the same sign (positive or negative), since f's sign can only change at a root. By evaluating $f(t)$ at any value of t between the two roots we can determine this sign.

We can thus find the *intervals* of t on which $f(t)$ is positive. For example, the polynomial $x^2 - 1$ has two intervals on which it is positive: $(-\infty, -1)$, and $(1, \infty)$. $-x^2 + 1$ has one: $(-1, 1)$. Higher-degree polynomials might have many such intervals.

We can then do the same thing for $g(t)$. If any of f's intervals overlap one of g's intervals, then any value of t inside the *intersection* of the two intervals is a solution to the system of inequalities. Figure 2 illustrates this algorithm for two example polynomials.

As a final note, although this algorithm has been described for two polynomials, it is easy to extend to the problem of solving for when an arbitrary number of polynomials $f(t), g(t), h(t), \ldots$ are simultaneously positive.

This algorithm, while conceptually simple, can be tricky to implement correctly, so in *Continuous-TimeCollisionHandler.cpp* we have provided you with the method *findFirstIntersectionTime* that takes in an arbitrary number of polynomials are returns the first time $t > 0$ when all polynomials become simultaneously positive.

Important Note: Carefully read the comments in the provided code, and ensure you are e.g. passing in polynomial coefficients in the right order.

3.3 Incorporating Continous-Time Collision Detection with Collision Response

Using the above polynomial inequality algorithm, we can determine whether or not two objects collide during a time step. But what do we do with this information? In Milestone 1, we applied impulses at the end of the time step, but there's no point detecting that two objects are going to tunnel through each other if we can't do anything about it until after the tunnelling has occured!

What we must do instead is partition time-stepping into two parts. First, we step the simulation forward using whatever integrator has been specified for the simulation, completely ignoring collisions. This gives us *predicted* positions and velocities $\mathbf{q}^e$ and $\dot{\mathbf{q}}^e$. We then perform continuous-time collision detection and response, modifying $\mathbf{q}^e$ and $\dot{\mathbf{q}}^e$ to prevent all collision during the time step. Here is the algorithm in detail:

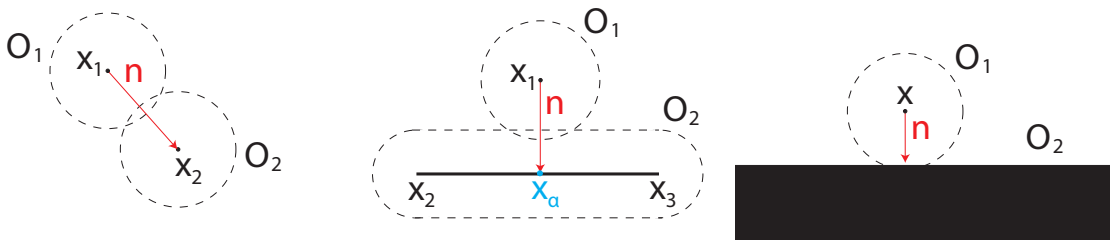

Figure 3: Left: A particle–particle collision. Middle: A particle–edge collision. Right: A particle–half-plane collision. In each case, the shortest vector between the objects, $\mathbf{n}$, has been drawn in red.

1. Step the old positions $\mathbf{q}^s$ and velocities $\dot{\mathbf{q}}^s$ forward using the numerical integrator to get predicted new positions $\mathbf{q}^e$ and velocities $\dot{\mathbf{q}}^e$.
2. Assume the particles moved with constant velocity from $\mathbf{q}^s$ to $\mathbf{q}^e$, and perform continuous-time collision detection with $\mathbf{q}(t) = \mathbf{q}^s + t\Delta\mathbf{q}$, where $\Delta\mathbf{q} = \mathbf{q}^e - \mathbf{q}^s$.
3. If any collisions were detected, apply impulses to the predicted velocities $\dot{\mathbf{q}}^e$ to get modified velocities $\dot{\mathbf{q}}^m = \dot{\mathbf{q}}^e + \mathbf{M}^{-1}\mathbf{I}$. The vector $\mathbf{I}$ is the impulse derived in Milestone 1 needed to resolve the collision.
4. Also modify positions to get new collision-free positions: $\mathbf{q}^m = \mathbf{q}^e + h\mathbf{M}^{-1}\mathbf{I}$.

Note that this algorithm assumes that applying impulses does not cause any new collisions. We will examine this assumption and further improve collision response with the hyrbrid method later this milestone.

Your first task in this milestone is to modify *ContinuousTimeCollisionHandler.cpp* and implement step 2 of this algorithm. For each of particle–particle, particle–edge, and particle–half-plane, implement continuous-time collision detection: given old and new positions, and assuming objects move in a straight line from their old to new positions, determine whether or not the objects are overlapping and approaching at some point during the motion. If so, compute the value of t at which the collision occurs, and the shortest vector between the objects at that value of t.

3.3.1 Particle–Particle

For two particles with centers $\mathbf{x}_1(t)$ and $\mathbf{x}_2(t)$, the vector $\mathbf{n}(t)$ is simply $\mathbf{x}_2(t) - \mathbf{x}_1(t)$. The first polynomial we need is therefore

$$\begin{aligned}
\mathbf{n}\cdot\mathbf{n} &< (r_1+r_2)^2 \\
(\mathbf{x}_2^s + t\Delta\mathbf{x}_2 - \mathbf{x}_1^s - t\Delta\mathbf{x}_1)\cdot(\mathbf{x}_2^s + t\Delta\mathbf{x}_2 - \mathbf{x}_1^s - t\Delta\mathbf{x}_1) &< (r_1+r_2)^2 \\
[(\mathbf{x}_2^s - \mathbf{x}_1^s) + t(\Delta\mathbf{x}_2 - \Delta\mathbf{x}_1)]\cdot[(\mathbf{x}_2^s - \mathbf{x}_1^s) + t(\Delta\mathbf{x}_2 - \Delta\mathbf{x}_1)] &< (r_1+r_2)^2 \\
(\mathbf{x}_2^s - \mathbf{x}_1^s)\cdot(\mathbf{x}_2^s - \mathbf{x}_1^s) + 2t(\mathbf{x}_2^s - \mathbf{x}_1^s)\cdot(\Delta\mathbf{x}_2 - \Delta\mathbf{x}_1) + t^2(\Delta\mathbf{x}_2 - \Delta\mathbf{x}_1)\cdot(\Delta\mathbf{x}_2 - \Delta\mathbf{x}_1) &< (r_1+r_2)^2,
\end{aligned}$$

a quadratic in t. We need to rewrite this in the form $f(t) > 0$ in order to plug it into the polynmoial inequality solver:

$$-(\Delta\mathbf{x}_2 - \Delta\mathbf{x}_1)\cdot(\Delta\mathbf{x}_2 - \Delta\mathbf{x}_1)t^2 - 2(\mathbf{x}_2^s - \mathbf{x}_1^s)\cdot(\Delta\mathbf{x}_2 - \Delta\mathbf{x}_1)t + (r_1+r_2)^2 - (\mathbf{x}_2^s - \mathbf{x}_1^s)\cdot(\mathbf{x}_2^s - \mathbf{x}_1^s) > 0.$$

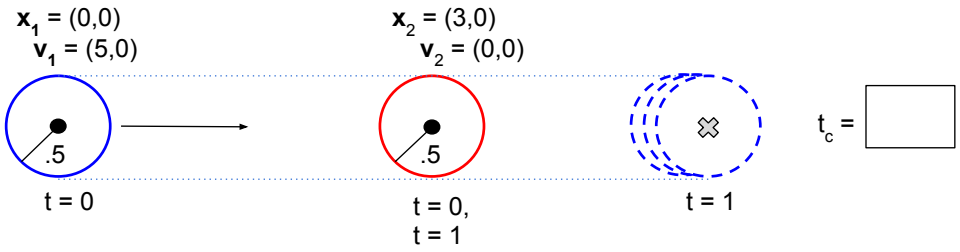

Figure 4: As a usfeul debug tool whenever deriving or confirming polynomials, try to set up example scenes where the answer is known and verifying your results.

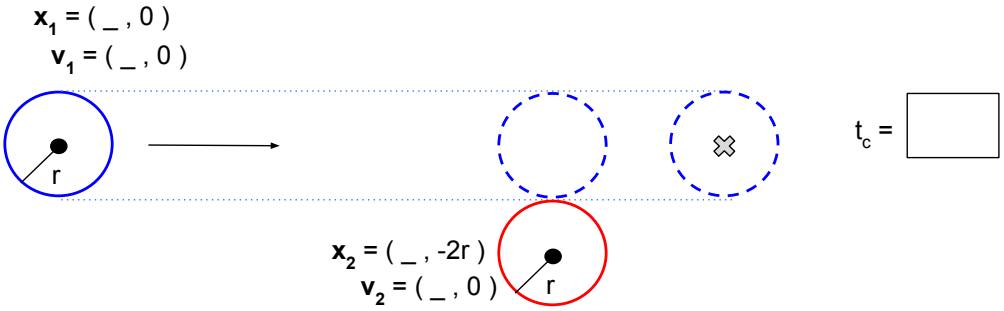

Figure 5: Is it important to also consider tricky scenarios. This example is left partially blank and may be filled in by students in order to debug.

From Milestone 1 we know that the relative velocity $\mathbf{r}(\Delta\mathbf{x})$ is $\Delta\mathbf{x}_1 - \Delta\mathbf{x}_2$, so the second inequality is just

$$\begin{aligned}(\Delta\mathbf{x}_1 - \Delta\mathbf{x}_2) \cdot \mathbf{n}(t) &> 0\\ (\Delta\mathbf{x}_1 - \Delta\mathbf{x}_2) \cdot (\mathbf{x}_2^s + t\Delta\mathbf{x}_2 - \mathbf{x}_1^s - t\Delta\mathbf{x}_1) &> 0\\ (\Delta\mathbf{x}_1 - \Delta\mathbf{x}_2) \cdot [(\mathbf{x}_2^s - \mathbf{x}_1^s) + t(\Delta\mathbf{x}_2 - \Delta\mathbf{x}_1)] &> 0\\ (\Delta\mathbf{x}_1 - \Delta\mathbf{x}_2) \cdot (\Delta\mathbf{x}_2 - \Delta\mathbf{x}_1)t + (\Delta\mathbf{x}_1 - \Delta\mathbf{x}_2) \cdot (\mathbf{x}_2^s - \mathbf{x}_1^s) &> 0.\end{aligned}$$

If both polynomial inequalities are satisfied for some t with $0 \leq t \leq 1$, then a collision has occured this time step.

3.3.2 Particle–Edge

Recall that in Milestone 1, we computed the shortest vector $\mathbf{n}$ between a particle at $\mathbf{x}_1$ with radius r_v and an edge with endpoints at $\mathbf{x}_2$ and $\mathbf{x}_3$ and radius r_e by first extending the edge to an infinite line, and finding the closest point $\mathbf{x}_\alpha = \mathbf{x}_2 + \alpha(\mathbf{x}_3 - \mathbf{x}_2)$ on that line to $\mathbf{x}_1$ by calculating α. By clamping α to the range $[0, 1]$, we found the closest point on the segment to $\mathbf{x}_1$, from which it followed that $\mathbf{n} = \mathbf{x}_\alpha - \mathbf{x}_1$.

Because of this clamping, it is not possible to translate the Milestone 1 formula for $\mathbf{n}$ into a simple function $\mathbf{n}(t)$. Instead we will have to be a little clever: instead of trying to write down a single polynomial inequality encoding that the particle overlaps with the edge, we will instead specify three simultaneous inequalities: (a) the particle overlaps with the infinite line; (b) $\alpha(t) > 0$, and (c) $\alpha(t) < 1$. (For this milestone we won't worry about collisions with the edge endcaps, when $\alpha = 0$ or $\alpha = 1$. Assuming the edge endpoints have radii at least as large as the edge radius, such collisions will be caught by particle–particle collision detection.)

To formulate polynomial (a), we take the formulas from Milestone 1 and simply omit the clamping of α:

$$\begin{aligned}\mathbf{n}(t) \cdot \mathbf{n}(t) &< (r_v + r_e)^2 \\ (r_v + r_e)^2 - [\mathbf{x}_\alpha(t) - \mathbf{x}_1(t)] \cdot [\mathbf{x}_\alpha(t) - \mathbf{x}_1(t)] &> 0 \\ (r_v + r_e)^2 - [\mathbf{x}_2(t) + \alpha(t)[\mathbf{x}_3(t) - \mathbf{x}_2(t)] - \mathbf{x}_1(t)] \cdot [\mathbf{x}_2(t) + \alpha(t)[\mathbf{x}_3(t) - \mathbf{x}_2(t)] - \mathbf{x}_1(t)] &> 0,\end{aligned}$$

where

$$\alpha(t) = \frac{[\mathbf{x}_1(t) - \mathbf{x}_2(t)] \cdot [\mathbf{x}_3(t) - \mathbf{x}_2(t)]}{\|\mathbf{x}_3(t) - \mathbf{x}_2(t)\|^2}.$$

Unfortunately the left-hand side of this inequality is not a polynomial, since $\alpha(t)$ is a rational function. But we can multiply both sides of the inequality by the (positive) denominator $\|\mathbf{x}_3(t) - \mathbf{x}_2(t)\|^2$, yielding

$$\|\mathbf{x}_3(t) - \mathbf{x}_2(t)\|^2 \left[(r_v + r_e)^2 - \|\mathbf{x}_2(t) + \alpha(t)[\mathbf{x}_3(t) - \mathbf{x}_2(t)] - \mathbf{x}_1(t)\|^2\right] > 0,$$

which after simplification *does* turn out to be a quartic polynomial in t. Simplifying inequality (a) and deriving the formula for the coefficients of the quartic polynomial is left to you as part of this assignment. You do *not* need to turn in written work; whether or not your expressions are correct will be obvious from the behavior of your code. To help with debugging, here is the correct polynomial for two example sets of old positions and change in positions:

1.

$$\begin{aligned}\mathbf{x}_1^s = (3,1) \qquad \Delta\mathbf{x}_1 = (4,1) \\ \mathbf{x}_2^s = (5,9) \qquad \Delta\mathbf{x}_2 = (2,6) \\ \mathbf{x}_3^s = (5,3) \qquad \Delta\mathbf{x}_3 = (5,8) \\ r_v = 1 \qquad r_e = 1\end{aligned}$$

$$-361t^4 - 304t^3 - 468t^2 - 288t > 0,$$

2.

$$\begin{aligned}\mathbf{x}_1^s = (1,4) \qquad \Delta\mathbf{x}_1 = (1,4) \\ \mathbf{x}_2^s = (2,1) \qquad \Delta\mathbf{x}_2 = (3,5) \\ \mathbf{x}_3^s = (6,2) \qquad \Delta\mathbf{x}_3 = (3,7) \\ r_v = 1 \qquad r_e = 2\end{aligned}$$

$$-16t^4 - 68t^2 + 36t - 16 > 0.$$

Polynomial (b) is quadratic in t:

$$\begin{aligned}\alpha(t) &> 0 \\ \frac{[\mathbf{x}_1(t) - \mathbf{x}_2(t)] \cdot [\mathbf{x}_3(t) - \mathbf{x}_2(t)]}{\|\mathbf{x}_3(t) - \mathbf{x}_2(t)\|^2} &> 0 \\ [\mathbf{x}_1^s + t\Delta\mathbf{x}_1 - \mathbf{x}_2^s - t\Delta\mathbf{x}_2] \cdot [\mathbf{x}_3^s + t\Delta\mathbf{x}_3 - \mathbf{x}_2^s - t\Delta\mathbf{x}_2] &> 0 \\ [(\mathbf{x}_1^s - \mathbf{x}_2^s) + t(\Delta\mathbf{x}_1 - \Delta\mathbf{x}_2)] \cdot [(\mathbf{x}_3^s - \mathbf{x}_2^s) + t(\Delta\mathbf{x}_3 - \Delta\mathbf{x}_2)] &> 0 \\ (\Delta\mathbf{x}_1 - \Delta\mathbf{x}_2) \cdot (\Delta\mathbf{x}_3 - \Delta\mathbf{x}_2)t^2 + [(\mathbf{x}_1^s - \mathbf{x}_2^s) \cdot (\Delta\mathbf{x}_3 - \Delta\mathbf{x}_2) + (\Delta\mathbf{x}_1 - \Delta\mathbf{x}_2) \cdot (\mathbf{x}_3^s - \mathbf{x}_2^s)]t & \\ +(\mathbf{x}_1^s - \mathbf{x}_2^s) \cdot (\mathbf{x}_3^s - \mathbf{x}_2^s) &> 0.\end{aligned}$$

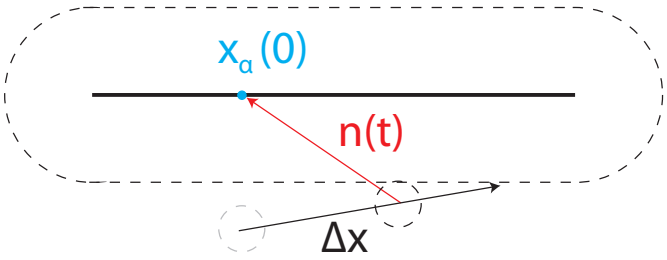

Figure 6: A fast-moving particle collides with an edge at a glancing angle. If we assume α is constant, the dot product between $\mathbf{n}(t)$ (red arrow) and Δx (black arrow) is negative at the time of collision, and we (incorrectly) conclude that the particle and edge are separating at that time.

Polynomial (c) is similar, and simplifies to

$$(\Delta\mathbf{x}_1 - \Delta\mathbf{x}_3) \cdot (\Delta\mathbf{x}_2 - \Delta\mathbf{x}_3)t^2 + [(\mathbf{x}_1^s - \mathbf{x}_3^s) \cdot (\Delta\mathbf{x}_2 - \Delta\mathbf{x}_3) + (\Delta\mathbf{x}_1 - \Delta\mathbf{x}_3) \cdot (\mathbf{x}_2^s - \mathbf{x}_3^s)]t + (\mathbf{x}_1^s - \mathbf{x}_3^s) \cdot (\mathbf{x}_2^s - \mathbf{x}_3^s) > 0.$$

The last polynomial we need is one encoding that the edge and particle are approaching. Per Milestone 1, the full formula for this polynomial would be

$$(\Delta\mathbf{x}_1 - \Delta\mathbf{x}_2 - \alpha(t)[\Delta\mathbf{x}_3 - \Delta\mathbf{x}_2]) \cdot \mathbf{n}(t) > 0,$$

which after simplification and clearing out denominators becomes a quintic polynomial. As a possible simplification, one can assume that $\alpha(t)$ is constant during a time step, but that is not valid in all cases. For example if a particle is moving quickly and collides with an edge at a glancing angle, using this approximation can cause collisions to be missed. Figure 6 illustrates one of such situations for a moving particle and a fixed edge. Because we are implementing a robust simultaneous collision handling framework (discussed further in later part of this milestone), adopting this simplification will make all our efforts futile. Therefore we need the full quintic polynomial inequality. The code for this quintic polynomial is provided to you for free. You only need to implement the other three polynomials.

If all four polynomial inequalities are simultaneously satisfied for some t between 0 and 1, then the particle and edge collide.

3.3.3 Particle–Half-plane

Since particle–half-plane collisions involve only two degrees of freedom (the position of the particle), its formulas are the simplest to derive. Using the particle–particle derivation above and the formulas from Milestone 1 as your guide, formulate the two polynomials you need for continuous-time collision detection on your own. Your polynomial encoding that the particle overlaps the half-plane should be quadratic. The one encoding that they are approaching should be linear.

3.4 Handling Multiple Simultaneous Collisions

So far in this theme we have been assuming that doing one pass of pairwise collision detection, followed by collision response in the form of applying impulses to each pair of colliding objects, is enough to produce a collision-free simulation. This assumption neglects to take into account the possibility of three of more objects colliding *simultaneously* during a single time step. Figure 7 shows one example where this assumption breaks down: only one collision is detected at first, between particles 1 and 2, but resolving this collision

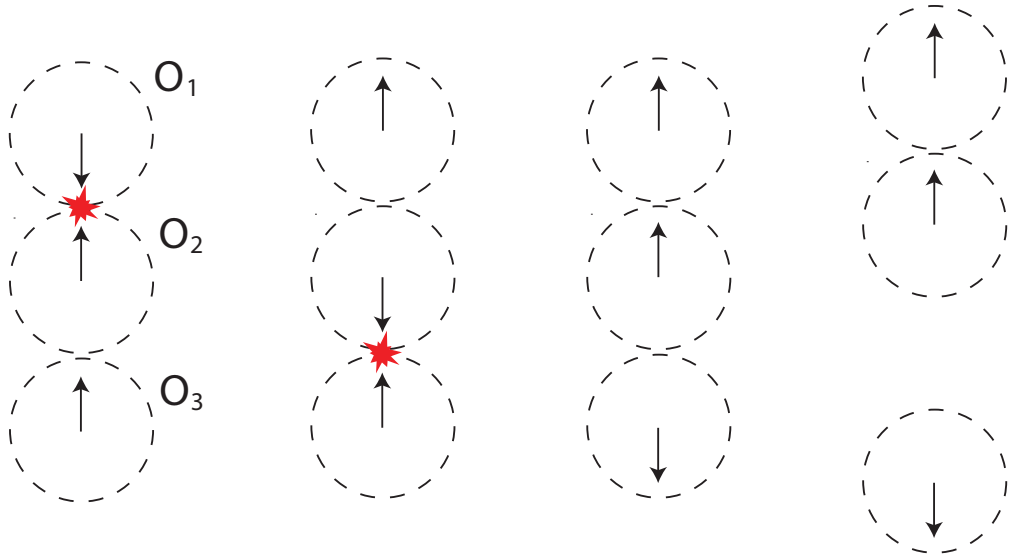

Figure 7: An example showing that a single round of collision detection and response can be insufficient. Initially, particles 1 and 2 are detected to be colliding (left); applying impulses resolves this collision, but introduces a new collision between particles 2 and 3 (middle left). A second iteration of collision detection and response (middle right) is necessary to produce a collision-free simulation at the end of the time step (right).

introduces a new one, between particles 2 and 3. Unless we do a second round of collision detection and response, we will have failed to stop this second collision, and the time step will end with the simulation in an interpenetrated state.

One way to handle the problem of simultaneous collisions is to wrap a loop around detection and response, and keep modifying positions and velocities until there are no collisions detected at the end of the time step. Here is the algorithm, a modification of that given in the previous section:

1. Step the positions $\mathbf{q}^s$ and velocities $\dot{\mathbf{q}}^s$ forward using the numerical integrator to get initial predicted end-of-time-step positions $\mathbf{q}_0^e$ and velocities $\dot{\mathbf{q}}_0^e$.

2. Assume the particles move with constant velocity from $\mathbf{q}^s$ to $\mathbf{q}_0^e$, and perform continuous-time collision detection with $\mathbf{q}(t) = \mathbf{q}^s + t\Delta\mathbf{q}_0$, where $\Delta\mathbf{q}_0 = \mathbf{q}_0^e - \mathbf{q}^s$.

3. For $i = 0, 1, \ldots$ until no collisions are detected:

4. (a) Apply impulses to the predicted velocities $\dot{\mathbf{q}}_i^e$ to get new predicted velocities $\dot{\mathbf{q}}_{i+1}^e = \dot{\mathbf{q}}_i^e + \mathbf{M}^{-1}\mathbf{I}$. The vector $\mathbf{I}$ is the sum of the impulses derived in Milestone 1 needed to resolve the detected collisions.

   (b) Also modify positions to get new predicted positions: $\mathbf{q}_{i+1}^e = \mathbf{q}_i^e + h\mathbf{M}^{-1}\mathbf{I}$.

   (c) Assume the particles move with constant velocity from $\mathbf{q}^s$ to $\mathbf{q}_{i+1}^e$, and perform continuous-time collision detection with $\mathbf{q}(t) = \mathbf{q}^s + t\Delta\mathbf{q}_{i+1}$, where $\Delta\mathbf{q}_{i+1} = \mathbf{q}_{i+1}^e - \mathbf{q}^s$.

If the above loop terminates, then it is guaranteed by construction to result in end-of-time-step position that are collision-free. Unfortunately, it can be shown that for some values of COR, there exist simulations for which infinitely many iterations are necessary to resolve all collisions: every impulse that stops one pair of objects from colliding causes a different pair of objects to collide. Moreover, you might need many, many iterations to fully resolve all collisions in a simulation with a large clump of objects. Even if the loop

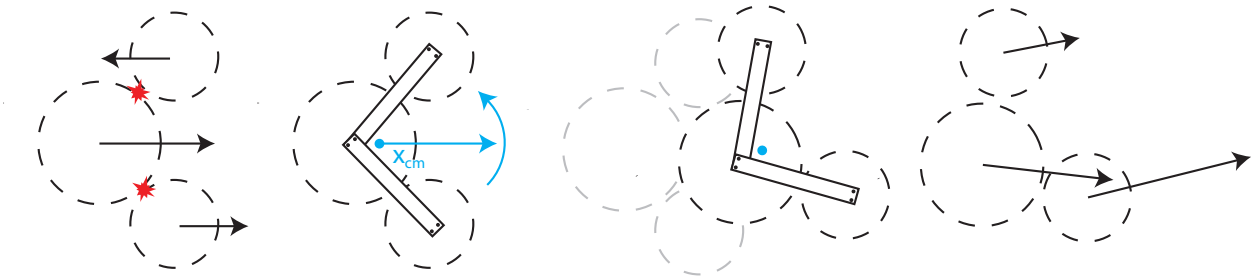

Figure 8: An illustration of the geometric collision response algorithm. Given a set of colliding vertices (left), we convert them into a rigid body (middle left), step the rigid body forward to the end of the time step (middle right), then convert the rigid body back into individual particles (right). The resulting position are guaranteed to be collision-free.

terminates, it might do so after running for a prohibitively long time. These potential pitfalls motivate us to look for an alternative approach that is guaranteed to handle multiple simultaneous collisions quickly.

3.5 Geometric Collision Response

Suppose we have some number of particles and edges that we know are about to collide with each other. There's a very simple thing we can do that is guaranteed to stop all collisions: we can simply freeze all the particles and edges in place. This solution is obviously extremely simplistic; it doesn't conserve momentum, for instance.

A more sophisticated approach is to treat the set of particles and edges as a *rigid body*: suppose every particle and edge endpoint is connected to every other one by a rigid metal beam. Since distances between particles and edges cannot change, the objects cannot collide. What we need to do, then, is calculate how the rigid body would move after the beams are attached, as a function of how the individual objects moved before attachment. For the purposes of this method, we will assume that the mass of an edge is lumped at its endpoints (so that the span between the endpoints is massless.) This assumption allows us to ignore the edge itself in what follows, and work only with particles and edge endpoints (which are also particles.)

Suppose we have a collection of particles with start-of-time-step positions $\mathbf{x}_i^s$ and (colliding) end-of-time-step positions $\mathbf{x}_i^e$. We also suppose that the particles move from their old to new positions with constant velocity $\Delta\mathbf{x}_i = \mathbf{x}_i^e - \mathbf{x}_i^s$. We want to do the following:

1. Treat the particles as if they were part of a rigid body; that is, treat them as if we connected them with rigid beams at the start of the time step.
2. Step the rigid body forward in time to the end of the time step.
3. Set each particle's modified end-of-time-step position $\mathbf{x}_i^m$ to the position dictated by the motion of the rigid body.
4. Also set the particle's modified end-of-time-step velocity to $(\mathbf{x}_i^m - \mathbf{x}_i^s)/h$, where h is the length of the time step.

Figure 8 shows this algorithm step by step.

If the masses of the particles are m_i, their *center of mass* is given by

$$\mathbf{x}_{cm}^s = \frac{\sum_i m_i \mathbf{x}_i^s}{\sum_i m_i}.$$

When all masses are equal, the center of mass is just the average of the particle positions. If masses are unequal, the center of mass is a weighted average, with the more massive particles having greater weight.

A rigid body's configuration is completely determined by only three degrees of freedom: the position of the center of mass of the body, and the orientation (rotation) of the body about the center of mass. Similarly, the motion of the rigid body is determined by the velocity $\Delta\mathbf{x}_{cm}$ and angular velocity (speed of rotation) ω_{cm} of the center of mass. At the start of the time step, we want to convert the individual particles into a single rigid body; we do so by calculating $\Delta\mathbf{x}_{cm}$ and ω_{cm}.

To find $\Delta\mathbf{x}_{cm}$, we invoke conservation of momentum. The momentum of the rigid body has to be equal to the momentum of the individual particles:

$$\left(\sum_i m_i\right)\Delta\mathbf{x}_{cm} = \sum_i m_i\Delta\mathbf{x}_i$$
$$\Delta\mathbf{x}_{cm} = \frac{\sum_i m_i\Delta\mathbf{x}_i}{\sum_i m_i}.$$

Similarly, ω_{cm} is derived by looking at conservation of angular momentum. The total angular momentum (about the center of mass) of the individual particles is

$$L = \sum_i m_i(\mathbf{x}_i^s - \mathbf{x}_{cm}) \times (\Delta\mathbf{x}_i - \Delta\mathbf{x}_{cm}),$$

where $(a, b) \times (c, d)$ is the scalar two-dimensional cross-product $ad - bc$. We want the angular momentum of the rigid body to have the same value. Angular momentum is related to angular velocity by the formula

$$L = I\omega_{cm},$$

where I is the moment of inertia $I = \sum_i m_i\|\mathbf{x}_i^s - \mathbf{x}_{cm}\|^2$ of the rigid body. Therefore

$$\omega_{cm} = L/I.$$

Now we know how to step the rigid body forward in time: the center of mass translates by $\Delta\mathbf{x}_{cm}$, and the body rotates about the center of mass by ω_{cm}. To convert the rigid body's new position into new positions for the individual particles, we use the formula

$$\begin{aligned}\mathbf{x}_i^e &= \mathbf{x}_{cm} + \Delta\mathbf{x}_{cm} + R_{\omega_{cm}}(\mathbf{x}_i^s - \mathbf{x}_{cm}) \\ &= \mathbf{x}_{cm} + \Delta\mathbf{x}_{cm} + \cos(\omega_{cm})(\mathbf{x}_i^s - \mathbf{x}_{cm}) + \sin(\omega_{cm})(\mathbf{x}_i^s - \mathbf{x}_{cm})^\perp,\end{aligned}$$

where $R_{\omega_{cm}}$ is rotation about the origin by ω_{cm}, and $(a, b)^\perp = (-b, a)$.

3.5.1 Handling Fixed Objects

There is another wrinkle that must be addressed: if the set of colliding objects includes a fixed object (such as a fixed particle, an edge containing an endpoint that's fixed, or any half-plane), we cannot move the set of objects as a rigid body since the fixed object's position is not allowed to change. For this milestone, when applying geometric collision response to a set of objects that includes a fixed object, instead of the above set $\mathbf{x}_i^e = \mathbf{x}_i^s$ and $\mathbf{v}_i^e = 0$. There are more clever things you could do (for instance, if the only fixed object is a particle, you might set that particle's position to be the center of mass, and allow the other particles to rotate about that position) but they are beyond the scope of this milestone.

3.6 Impact Zones

In the previous section we saw how to take a set of colliding objects and compute end-of-time-step positions for them that are guaranteed to be collision-free, by treating them all as part of one rigid body. What we still need is the big picture: given some start-of-time-step positions $\mathbf{q}^s$ and end-of-time-step positions $\mathbf{q}^e$, with some of the objects colliding as they move between these two positions, to which subset of the objects do

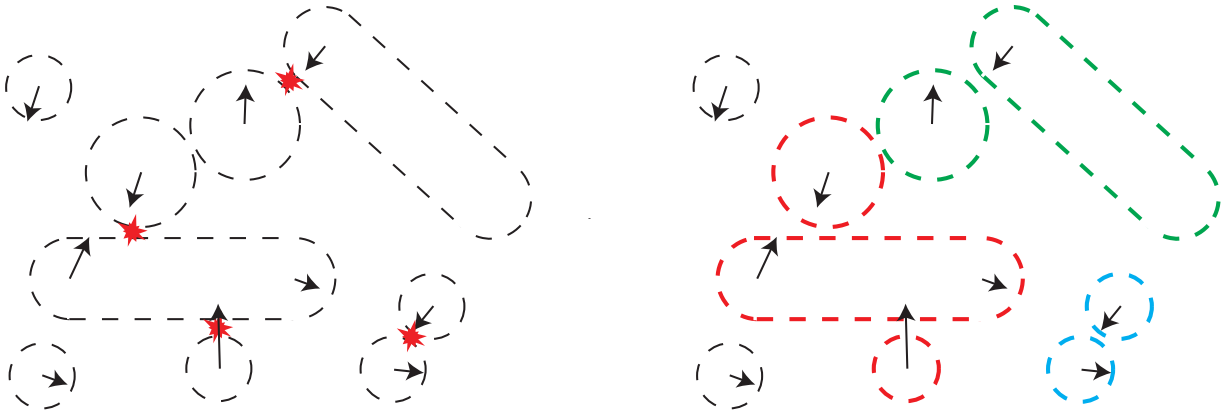

Figure 9: A number of collisions are detected during a time step (left). Based on these detected collisions, the particles in the simulation involved in collisions are separated into impact zones (right). Each impact zone is a different color.

we apply the geometric response discussed in section 3.5? Moving *all* particles and edges in the simulation as a rigid body, even those not at all involved in a collision, is clearly a poor solution. Treating all objects involved in a collision as part of the same rigid body is also suboptimal; if there are two clumps of collisions in two separate areas of the simulation, we don't want to couple them unnecessarily by gluing the two clumps together into one rigid body.

Instead, we will split up all particles involved in a collision into one or more *impact zones*. An impact zone is a set of particles, as well as a boolean flag indicating whether or not a half-plane is colliding with one of the particles in the impact zone.[1] Conceptually, each impact zone is one clump of interconnected collisions, with no particle shared by more than one impact zone. Each impact zone will be treated as a separate rigid body during geometric collision response; figure 9 illustrates the impact zones for an example set of detected collisions.

Here is the concrete algorithm for how to turn a set of detected collisions into disjoint impact zones:

1. First, create an impact zone for each detected collisions.
   (a) For each collision detected between particles i and j, create an impact zone containing the set of particles $\{i, j\}$. This impact zone does not involve a half-plane.
   (b) For each collision detected between particle i and edge whose endpoints are particles j and k, create an impact zone containing the set of particles $\{i, j, k\}$. This impact zone does not involve a half-plane.
   (c) For each collision detected between a particle i and a half-plane, create an impact zone containing only the lone vertex $\{i\}$. This impact zone does involve a half-plane.
2. Two impact zones are not allowed to share the same particle. If any two impact zones Z_1 and Z_2 share a particle, they must be *merged*: replace the two zones with a single new zone containing the union of Z_1 and Z_2's particles. The new zone involves a half-plane if either of the old zones did.
3. Repeat the previous step until all zones are disjoint.

Once we have a set of disjoint impact zones, we can adjust the end-of-time-step positions of the vertices in each zone using geometric collision response. It is important to stress that this response is applied to each zone separately: each zone acts as a different rigid body.

[1] We need to keep track of which impact zones involve a half-plane since we must treat such zones specially. See section 3.5.1.

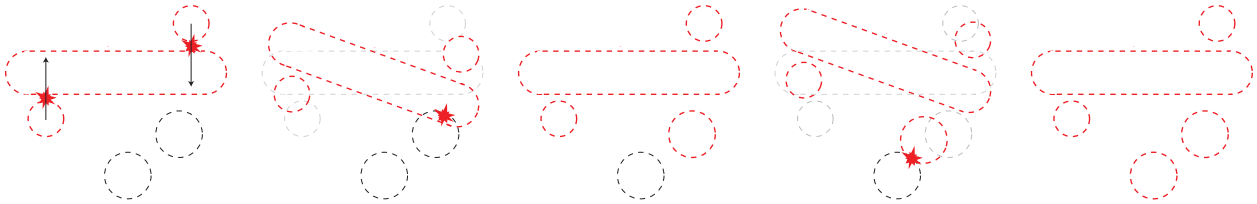

Figure 10: Iterative geometric collision handling. Some collisions are detected, and impact zones constructed (left). Stepping the simulation forward causes new collisions (middle left), so the impact zones grow (middle). This process repeats a second time (middle right and right).

Just as when resolving collisions using impulses, there is a possibility that one iteration of response could cause new collisions: a zone Z_1 might, while moving as a rigid body, collide into an object not in Z_1. One possibility discussed in class is to go back to the approach in 3.4 and apply impulses between the rigid body and the new object. How to apply impulses to rigid bodies will be covered in the next part of this milestone; for now, if such a collision is detected, we will *grow* Z_1 by adding the new object to it, and recomputing the end-of-time-step positions for the particles in the new Z_1. As impact zones grow in this way, it is also possible for them to merge. Here is the algorithm in more detail; it assumes we have start-of-time-step positions and velocities $\mathbf{q}^s, \dot{\mathbf{q}}^s$ and predicted end-of-time-step positions and velocities $\mathbf{q}^e, \dot{\mathbf{q}}^e$, and are trying to find collision-free, modified end-of-time-step positions and velocities $\mathbf{q}^m, \dot{\mathbf{q}}^m$.

1. Perform continuous-time collision detection using positions $\mathbf{q}^s$ and $\mathbf{q}^e$.
2. Initialize $\mathbf{q}^m = \mathbf{q}^e$ and $\dot{\mathbf{q}}^m = \dot{\mathbf{q}}^e$.
3. Construct a list of disjoint impact zones $\mathbf{Z}$ from the detected collisions.
4. For each impact zone in $\mathbf{Z}$, apply geometric collision response, using positions $\mathbf{q}^s$ and $\mathbf{q}^m$, and modifying $\mathbf{q}^m$ and $\dot{\mathbf{q}}^m$ for the vertices in those zones.
5. Perform continuous-time collision detection using positions $\mathbf{q}^s$ and $\mathbf{q}^m$.
6. Construct a new list of impact zones $\mathbf{Z}'$ consisting of all impact zones in $\mathbf{Z}$, plus one zone for each detected collision.
7. Merge the zones in $\mathbf{Z}'$ to get disjoint impact zones.
8. If $\mathbf{Z}$ and $\mathbf{Z}'$ are equal, the algorithm is done: $\mathbf{q}^m$ and $\dot{\mathbf{q}}^m$ are the new, collision-free end-of-time-step positions. $\mathbf{Z}$ and $\mathbf{Z}'$ are equal if they contain exactly the same impact zones; impact zones are the same if they contain the same particles and they both involve, or both don't involve, a half-plane. If $\mathbf{Z} \neq \mathbf{Z}'$, go to step 9.
9. Set $\mathbf{Z} = \mathbf{Z}'$ and go to step 4.

Figure 10 illustrates the algorithm. Unlike the interative impulse algorithm described in section 3.4, this algorithm is guaranteed to terminate after finitely many iterations: during each iteration, either an impact zone grows, or an impact zone that didn't involve a half-plane now involves a half-plane. This process must eventually stop (in the worst case, with every single particle contained in one impact zone that involves a half-plane).

3.7 Putting It All Together: Hybrid Collision Handling

Although geometric collision response is guaranteed to produce collision-free end-of-time-step positions, and conserves momentum and angular momentum, it is important to realize that it is not physically correct: for instance, a particle that hits a fixed edge at an angle should either reflect off of the edge (for COR = 1.0), slide along the edge (for COR = 0.0), or reflect at some angle between these two extremes. In every case, the particle's motion in the direction *tangential* to the edge is unrestricted. When using geometric response, on the other hand, the particle will come to a dead stop. In general, handling collisions using geometric response alone results in simulations that look "chunky" and unnatural.

However, geometric response can be a potent tool when combined with iterated impulses. Instead of iterating applying impulses until no collisions are detected – which could take a very long time – we cap the loop at a certain fixed *maximum number of iterations* n. After n iterations, if there are still unresolved collisions, we switch to geometric impulses as a failsafe. Here is the algorithm:

1. Perform collision handling using iterative impulses, as described in 3.4. Stop after n iterations, or if there are no new detected collisions. The outputs from this process are new predicted end-of-time-step positions and velocities $\mathbf{q}_n^e$ and $\dot{\mathbf{q}}_n^e$. This part you need to implement in *HybridCollisionHandler::applyIterativeImpulses*.

2. If there were still unresolved collisions, perform geometric collision handling, using these new predicted quantities. The outputs from this step are guaranteed collision-free positions and velocities $\mathbf{q}^m$ and $\dot{\mathbf{q}}^m$. This part you need to implement in *HybridCollisionHandler::applyGeometricCollisionHandling*.

If objects in a simulation become clumped in very close proximity, many simultaneous collisions occur and it becomes likely that the failsafe (step 2) is necessary to resolve these collisions. To try to create some breathing room between nearby objects and prevent this situation from occurring too often, we will add a final ingredient to our hybrid method: a weak penalty force that repels objects that are near-touching. You do not need to implement this penalty force: the starter code automatically adds it to simulations with hybrid collision detection.

Edit the *Student Code* section of *HybridCollisionHandler.cpp* and implement steps 1 and 2. To free you from low-level chores, some utility functions for manipulating impact zones are provided in the *Impact Zone Utilities* section of *HybridCollisionHandler.cpp*. You can of course implement your own utility functions (for example, functions for merging impact zones, for growing impact zones given a set of newly detected collisions, etc) to help yourself organize the code. How you implement them is completely up to you.

3.7.1 Further Reading

The hybrid method presented above is adapted from algorithms described in the following papers for handling collisions between cloth and objects in 3D simulations. Over the course of this theme, you have learned many of the main ideas that make up these algorithms, which are state-of-the-art and used by major movie studios.

- Robust Treatment of Collisions, Contact and Friction for Cloth Animation. R. Bridson, R. Fedkiw and J. Anderson, ACM Transactions on Graphics, vol 21, no 3, Proc. ACM SIGGRAPH 2002, pp. 594–603.

- Robust Treatment of Simultaneous Collisions. D. Harmon, E. Vouga, R. Tamstorf and E. Grinspun, ACM Transactions on Graphics, vol 27, no 3, Proc. ACM SIGGRAPH 2008, pp. 1–4.

4 Requirement for the Milestone

4.1 Continuous Time Collision Handler

In *ContinousTimeCollisionHandler.cpp* you need to edit the three detection methods, namely *detectParticleParticle*, *detectParticleEdge*, and *detectParticleHalfplane*. Due to numerical errors intrinsic in handling

high order polynomials, the oracle grades your submission slightly differently for this milestone. Instead of checking your simulation results, it checks whether your polynomial coefficients are correct (up to a scaling factor, because uniform scaling does not change the roots of a polynomial). For this reason, the backbone of these detection methods are already provided, and what you need to do is to fill in the coefficients. DO NOT CHANGE THE ORDER OF THE POLYNOMIALS, or you will fail the oracle. After calling the polynomial root finder, you will need to compute the contact vector **n** and decide whether to report a collision or not.

4.2 Hybrid Collision Handler

In *HybridCollisionHandler.cpp* you need to edit methods *applyIterativeImpulses*, *applyGeometricCollisionHandling* and *performFailsafe* to implement the hybrid collision handling as described in the previous section. *applyIterativeImpulses* should detect collisions with continuous time detector, attempt to resolve them with impulses, detect collisions again, and attempt to resolve them again, ... until no more collisions are detected, or impulses have been applied a number of times (n_maxiters). *applyGeometricCollisionHandling* and *performFailsafe* are really one task (the second part of the hybrid collision handler); they are only divided so that each one has a relatively simpler workflow. *applyGeometricCollisionHandling* iteratively executes continuous time collision detection, gathers the collisions into a collection of impact zones, and calls *performFailsafe* on them which rigidly moves each impact zone, guaranteeing a collision free end of step configuration. The workflow of each method is laid out in the comments in detail. You need to complete the implementation.

Note for GeometricTests: maximum iteration number is set to 0 in the GeometricTests scenes (because they are intended to test the geometric response only). When there is no collisions, the *applyIterativeImpulse* code can either do a collision detection and report "no collision found" so that the geometric response routine is never called, or it can directly report "iterative impulses failed" and let the geometric response routine figure out that the impact zones are empty by performing their own collision detection. These two implementations are equivalent, but since the oracle is grading your code using polynomials instead of simulation results, the two implementations differ because they perform different numbers of collision detection. Please follow the second convention, i.e. when maxiters is 0, do not perform any collision detection in applyIterativeImpulses().

5 Creative Scene

As part of your final submission for this milestone, please include a scene of your design that best shows off your program. Your scene will be judged by a secret of committee of top scientists using the highly refined criteria of:

1. How well the scene shows off this milestone's "magic ingredients" (a la Iron Chef).
2. Aesthetic considerations. The more beautiful, the better.
3. Originality.

Top examples will be posted to the discussion board. Please remember to generate this scene before submitting, as the Codio box will become read-only after you submit.

5.1 Making Movies

Please remember to generate this scene before submitting, as the Codio box will become read-only after you submit. The FOSSSim starter code comes with a PNG outputting utility that can help you create movies from your simulation. To enable it, create a folder named "pngs" in the directory that you are calling your executable from (i.e. your current directory) and run your code with the

```
-g 1
```

flag enabled. This will prompt your program to automatically save a PNG file for each frame of the simulation. This wont work if the "pngs" folder does not exist so you need to create it first before you run the program.

After the simulation finishes, you'll find all the frames in the pngs folder. Then you can make videos out of them with command-line tools such as mencoder and ffmpeg. Both mencoder and ffmpeg are easy-to-use tools available on the Codio boxes. You can execute this command:

```
mencoder mf://pngs/*.png -mf fps=24 -ovc lavc -lavcopts vcodec=msmpeg4v2 -oac copy -o output.avi
```

followed by:

```
ffmpeg -i output.avi -vcodec libx264 -crf 25 output.mp4
```

in order to create an mp4 video you may upload for the Creative portion of the assignment.

Note, ffmpeg can be used to create a video from the png images in one step, although sometimes this will give an error depending on the count and dimensions of input files. Nevertheless, the following command can take pngs and convert them to an mp4 movie in one:

```
ffmpeg -r 24 -f image2 -i ./pngs/frame%05d.png -vcodec libx264 -crf 25 -pix_fmt yuv420p test.mp4
```

Lastly, you may preview your movie by running the following command and looking at the virtual desktop:

```
mplayer <movie_name>
```

Please submit the mp4 file to the Peer Review Assignment portion of this week. An explanation for arguments to both mencoder and ffmpeg can be found online.

FAQ

Q: What does it mean to fail an assertion `m_x.all() == m_x.all()`?
A: The only cause for any variable to fail a comparison with itself (unless == is overloaded) in C++ is a 'nan'. nan != nan, and therefore if you have nans in your vector those types of assertions will fail.

Q: I need help simplifying the polynomial, what are my options?
A: Other than going about it by hand, MatLab and Mathematica can be of use.

A technique in Mathematica that has proved helpful: I have found that simplifying expressions by hand can lead to simpler expressions. But it can also lead to bugs. Therefore, I typically simplify by hand, but then I use Mathematica to check my simplification. In particular, if f(x) is the original expression, and g(x) is my attempt at simplification, then in Mathematica, I run FullSimplify[f(x) - g(x)] and if Mathematica gives me zero, then I know I did it right. If Mathematica doesn't give me zero, then maybe it got stuck simplifying, so it doesn't mean I necessarily got it wrong. In that case, I take the result from Mathematica (which should be zero), and I plug in different values for the coefficients and for x. If I consistently get a numerical zero, then I believe I've done the simplification correctly.

Q: How do fixed objects effect geometric collision response?
A: If the impact zone contains any fixed object, then the rule for fixed objects applies to all in the impact zone.

Tips

When doing the derivation of the quartic, using temporary variables to replace recurring terms can make the simplification and expansion a lot easier and more compact.

The function .normalize() returns a unit vector and .norm() returns a scalar magnitude of a vector.

Many people have had issues in the past and resolved them by going back over the prompt more carefully. Please read and get started on it early, so you may ask clarification questions and save yourself time.

The scalar cross product is $(x_1, y_1) \times (x_2, y_2) = x_1 y_2 - y_1 x_2$.

Random behavior is often introduced by garbage values.

Even if your residual is zero, you will fail the oracle if your coefficients for your polynomial are incorrect. It may also be an indication that you are detecting the wrong number of collisions. The oracle changes from milestone to milestone in the criteria it checks, and so a residual of zero does not necessarily mean a correct simulation.

Change the `fps=24` parameter in the mencoder call to get faster creative scene videos.