Animation & CGI Motion: Krusty's Crazy Collisions

Theme 2 Milestone 1

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 add efficient and robust collision handling to the particle system you worked with in Theme 1. In this milestone you will implement basic but functional collision detection and response, which will be built upon in Milestones 2 and 3.

2 New XML Features

In addition to the xml tags from Theme 1, this milestone adds the following new features:

1. The *halfplane* node adds a half-plane to the scene:

   ```
   <halfplane px="0.0" py="0.0" nx="0.0" ny="1.0"/>
   ```

   The half-plane consists of all points $\mathbf{x}$ satisfying $[\mathbf{x} - (px, py)] \cdot (nx, ny) \leq 0$. So the example half-plane node above would consist of a floor taking up all points with $y \leq 0$. The vector (nx, ny) must have nonzero magnitude but need not be normalized; this magnitude does not affect the behavior of the half-plane. This XML tag is associated with the half-plane feature described in section 3.2.3.

2. The *halfplanecolor* node changes a half-plane's color.

   ```
   <halfplanecolor i="0" r="0.1" g="0.2" b="0.3"/>
   ```

   The integer attribute i identifies the half-plane. The scalar r, g, b attributes set the half-plane's color. The three color attributes must have values between 0.0 and 1.0.

3. The *collision* node specifies how collisions are handled during the simulation:

   ```
   <collision type="simple" COR="0.5"/>
   ```

   or

   ```
   <collision type="penalty" k="100" thickness="0.1"/>
   ```

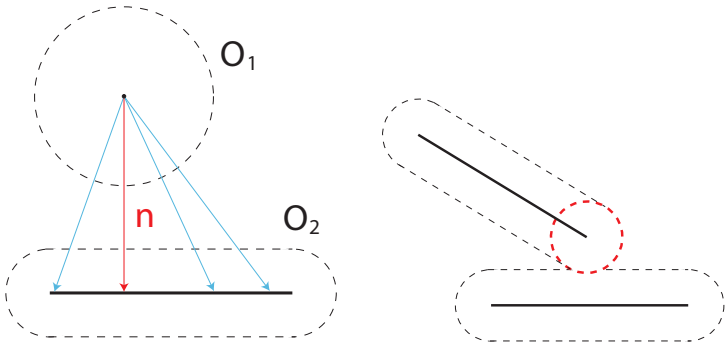

Figure 1: Left: Checking whether or not a particle and an edge are colliding. Several vectors between the particle and and the edge are drawn in blue; the shortest one **n** is drawn in red. Since **n** has length greater than the sum of the two objects' radii, the objects are not overlapping and so are not colliding. Right: Two edges cannot collide without there also being a particle-edge collision (particle drawn in red).

For this milestone, the valid values for *type* are "simple", "penalty" and "none". Later milestones will add new types of collision handling. The optional attribute *COR* sets the coefficient of restitution (described below) for contact. Its default value is 1.0. Also, *collision* has optional attributes k and *thickness* specifying the stiffness and thickness of the penalty forces used by the penalty method (described below). Attributes k and *thickness* have no effect unless *type* is "penalty".

4. To give you greater control over the rendering of your simulation, we have added an optional *viewport* node:

   ```
   <viewport cx="0" cy="0" size="5.0"/>
   ```

   The viewport specifies the center and size of the default camera viewport, in object coordinates; these settings override the default auto-centering behavior. The camera can still be moved around and resized during an OpenGL simulation using the mouse.

   The headless rendering viewport has been completely changed to coincide with what you see during OpenGL rendering, and in particular respects the *viewport* settings.

3 Required Features for Milestone I

3.1 Overview of Detection

Suppose you have two objects (each of which can be a particle, edge, or half-plane) O_1 and O_2 with thicknesses r_1 and r_2. Conceptually, the algorithm for checking whether or not O_1 and O_2 are colliding is to

1. look at all vectors between points in O_1 and O_2,
2. find the vector **n** that is the shortest,
3. and compare its length to $r_1 + r_2$. If **n** has length less than $r_1 + r_2$, the objects are overlapping.
4. Lastly, check if they are approaching or moving apart. If they are approaching, then the objects are colliding.

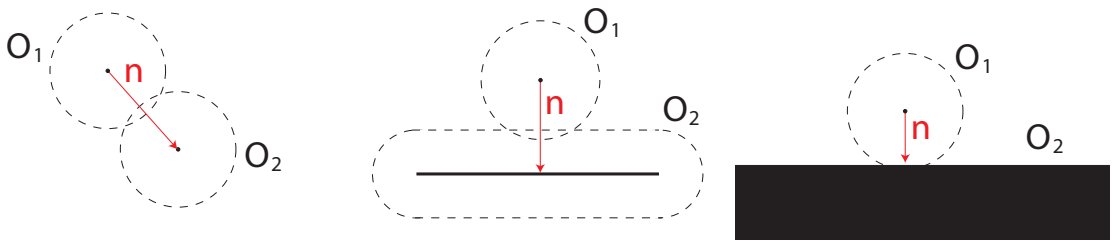

Figure 2: Left: A particle–particle collision. Middle: A particle–edge collision. Right: A particle–half-plane collision. In each case, the shortest vector between the objects, $\mathbf{n}$, has been drawn in red.

Figure 1, left, illustrates this algorithm for a particle and an edge.

Although this algorithm gives a good overview of the big picture, it cannot be practically implemented as written above: for instance, you wouldn't want to write code to actually find the set of all possible vectors between points in O_1 and O_2. Instead, by examining the geometry of the colliding objects, we will derive formulas for the shortest vector, skipping the first step entirely.

The types of collisions that can occur are particle–particle, particle–edge, and particle–half-plane. Edges can also collide with edges, and edges can collide against half-planes – but in both of these cases there is also guaranteed to be a particle–edge or particle–half-plane collision as well, so they do not need to be checked or responded to separately. (For this theme you may assume that the endpoints of an edge have radius at least as large as the edge itself.) See figure 1, right, for an illustration.

3.2 Implement Detection Routines for Primitive Pairs

Edit the provided source file *SimpleCollisionHandler.cpp* and, for each of the following pairs of primitives, implement a function that takes two such primitives and computes $\mathbf{n}$ and determines whether or not the two objects are colliding.

3.2.1 Particle–Particle

Finding the shortest vector from O_1 to O_2 is trivial when both objects are particles: there is only one vector to choose from. When checking the length of $\mathbf{n}$, don't forget that the two particles might have different radii.

To check if the particles are approaching, look at the difference in velocity along the $\mathbf{n}$ direction, $(\mathbf{v}_1 - \mathbf{v}_2) \cdot \mathbf{n}$. This scalar is proportional to the speed at which the particles are moving toward each other; if it is positive, then the particles are approaching.

Important Note: The oracle uses a *strict* inequality ($<$) when comparing $|\mathbf{n}|$ to $r_1 + r_2$. It also uses a strict inequality when checking if the relative velocity along $\mathbf{n}$ is positive. Be sure you follow this convention – otherwise you may detect collisions that the oracle does not, and fail a test.

3.2.2 Particle–Edge

If O_1 is a particle and O_2 an edge, there are many possible vectors from O_1 to O_2: although there is only one choice for the vector's tail, the tip can be any point on the edge O_2. To find the shortest vector, you must find the *closest* point on the edge to the particle.

Let $\mathbf{x}_1$ be the position of the particle, and $\mathbf{x}_2$ and $\mathbf{x}_3$ the two endpoints of the edge. We will first find the closest point to $\mathbf{x}_1$ on the infinite line passing through these two endpoints. Note that any point on the

line can be written as $\mathbf{x}(\alpha) = \mathbf{x}_2 + \alpha(\mathbf{x}_3 - \mathbf{x}_2)$ for some scalar parameter α. Our goal is to find the α that gives us a point closest to $\mathbf{x}_1$.

Minimizing the distance between $\mathbf{x}_1$ and $\mathbf{x}(\alpha)$ is the same as minimizing the *squared* distance, $\|\mathbf{x}_1 - \mathbf{x}(\alpha)\|^2$, between them. As usual when trying to find a minimum, we take the derivative with respect to α, set it equal to zero, then solve for α:

$$\begin{aligned}
\frac{d}{d\alpha}\|\mathbf{x}_1 - \mathbf{x}(\alpha)\|^2 &= 0 \\
\frac{d}{d\alpha}\left[(\mathbf{x}_1 - \mathbf{x}(\alpha)) \cdot (\mathbf{x}_1 - \mathbf{x}(\alpha))\right] &= 0 \\
-2(\mathbf{x}_1 - \mathbf{x}(\alpha)) \cdot \frac{d}{d\alpha}\mathbf{x}(\alpha) &= 0 \\
(\mathbf{x}_1 - \mathbf{x}(\alpha)) \cdot \frac{d}{d\alpha}\left[\mathbf{x}_2 + \alpha(\mathbf{x}_3 - \mathbf{x}_2)\right] &= 0 \\
(\mathbf{x}_1 - \mathbf{x}_2 - \alpha(\mathbf{x}_3 - \mathbf{x}_2)) \cdot (\mathbf{x}_3 - \mathbf{x}_2) &= 0 \\
(\mathbf{x}_1 - \mathbf{x}_2) \cdot (\mathbf{x}_3 - \mathbf{x}_2) &= \alpha\|\mathbf{x}_3 - \mathbf{x}_2\|^2 \\
\alpha &= \frac{(\mathbf{x}_1 - \mathbf{x}_2) \cdot (\mathbf{x}_3 - \mathbf{x}_2)}{\|\mathbf{x}_3 - \mathbf{x}_2\|^2}.
\end{aligned}$$

Plugging this value of α into $\mathbf{x}(\alpha)$ tells us the closest point on the *infinite line* to $\mathbf{x}_1$. What we really want to know is the closest point on the *segment* to $\mathbf{x}_1$. If $0 \leq \alpha \leq 1$, then those points are one and the same. Otherwise, we have to *clamp* α to the range $[0, 1]$.

The full algorithm for finding $\mathbf{n}$ is thus:

1. Calculate α using the formula above.
2. If $\alpha < 0$, set $\alpha = 0$. If $\alpha > 1$, set $\alpha = 1$.
3. Calculate the point $\mathbf{x}(\alpha)$.
4. The vector we need is then $\mathbf{n} = \mathbf{x}(\alpha) - \mathbf{x}_1$.
5. Lastly check if this vector $\mathbf{n}$ is less than the combined radius of your particle and edge to confirm it is overlapping.

Implement this algorithm.

This only gets us the normal vector, which tells us how far apart the particle and edge are at their closest points but not if there is a true collision. One must remember to check if this vector $\mathbf{n}$ is less than the combined radius of the particle and edge, in order to ensure they are in fact overlapping.

Lastly, to check if the particle is approaching the edge, we want to look at the velocities in the $\mathbf{n}$ direction of the particle and the closest point on the edge. This closest point is $\mathbf{x}(\alpha)$; the velocity of this point is given by taking the velocities of the endpoints and interpolating using the same α :

$$\mathbf{v}(\alpha) = \mathbf{v}_2 + \alpha(\mathbf{v}_3 - \mathbf{v}_2).$$

The difference in velocity along $\mathbf{n}$ is then proportional to $(\mathbf{v}_1 - \mathbf{v}(\alpha)) \cdot \mathbf{n}$. Once again, the objects are approaching if this scalar is positive.

3.2.3 Particle–Half-plane

Let $\mathbf{x}_h$ and $\mathbf{n}_h$ be the position and normal of the half-plane, as specified in the XML file. The shortest vector between the particle and half-plane has direction $-\mathbf{n}_h$ and magnitude equal to the distance between

the particle and boundary of the half-plane. This distance is equal to

$$\frac{(\mathbf{x}_h - \mathbf{x}) \cdot -\mathbf{n}_h}{\|\mathbf{n}_h\|},$$

so

$$\mathbf{n} = \frac{(\mathbf{x}_h - \mathbf{x}) \cdot \mathbf{n}_h}{\|\mathbf{n}_h\|^2}\mathbf{n}_h.$$

The radius of the half-plane is zero for the purposes of checking the length of $\mathbf{n}$.

The velocity of the particle in the direction $\mathbf{n}$ is proportional to $\mathbf{v} \cdot \mathbf{n}$. If this quantity is positive, the particle is approaching the half-plane.

3.2.4 Automatic Testing of Detection by the Oracle

To aid you in debugging your code, we have added functionality to the oracle to automatically compare the collisions you found against those the oracle is expecting. When running the oracle in binary input mode, during each frame it will show you:

1. Missed collisions: those that the oracle detected this frame, but weren't detected by your code. The shortest vector $\mathbf{n}$ between the two objects whose collision was missed is drawn in green to indicate the missed collision.

2. Superfluous collisions: those that your code detected, but that the oracle didn't. The vector $\mathbf{n}$ of the superfluous collision is drawn on top of the simulation in red.

3. Incorrect $\mathbf{n}$: both the oracle and your code agree that a collision occurred, but disagree on the shortest vector $\mathbf{n}$. The incorrect vector is drawn in red, and the correct one in green.

3.3 Overview of Contact Response

Once we detect a collision, we must apply *impulses* – instantaneous changes to momentum – to *respond* to the collision. Edit the provided source file *SimpleCollisionHandler.cpp* to correctly respond to each of the three types of possible collisions.

3.3.1 Particle–Particle

Denoting post-collision-response velocities with tildes, we apply an impulse to the velocities of the particles O_1 and O_2 using the equations

$$\tilde{\mathbf{v}}_1 \leftarrow \mathbf{v}_1 + \mathbf{I}_1/m_1$$
$$\tilde{\mathbf{v}}_2 \leftarrow \mathbf{v}_2 + \mathbf{I}_2/m_2$$

for some as-yet-undetermined impulse vectors $\mathbf{I}_1$ and $\mathbf{I}_2$. By choosing these impulse vectors carefully, we stop the particles from approaching any further, while obeying the laws of physics.

Denote by $\hat{\mathbf{n}}$ the unit vector in direction $\mathbf{n}$. $\hat{\mathbf{n}}$ is called the *contact normal*. Pushing the objects apart in the direction of this vector most quickly increases the distance between them, so we want the impulses $\mathbf{I}_1$ and $\mathbf{I}_2$ to lie in the same direction as $\hat{\mathbf{n}}$:

$$\mathbf{I}_1 = I_1\hat{\mathbf{n}}$$
$$\mathbf{I}_2 = I_2\hat{\mathbf{n}}$$

for scalars I_1 and I_2.

To find formulas for these scalars, we write down the laws of conservation of momentum and energy. Conservation of momentum tells us that the total momentum of the two objects before and after we apply the impulses should be equal. Therefore

$$\begin{aligned} m_1\mathbf{v}_1 + m_2\mathbf{v}_2 &= m_1\tilde{\mathbf{v}}_1 + m_2\tilde{\mathbf{v}}_2 \\ m_1(\mathbf{v}_1 - \mathbf{v}_1 - \mathbf{I}_1/m_1) &= m_2(\mathbf{v}_2 + \mathbf{I}_2/m_2 - \mathbf{v}_2) \\ -\mathbf{I}_1 &= \mathbf{I}_2 \\ -I_1\hat{\mathbf{n}} &= I_2\hat{\mathbf{n}} \\ -I_1\hat{\mathbf{n}}\cdot\hat{\mathbf{n}} &= I_2\hat{\mathbf{n}}\cdot\hat{\mathbf{n}} \\ -I_1 &= I_2. \end{aligned}$$

Similarly, energy before and after applying the impulse must be equal. In particular, since applying an impulse only modifies velocities and not positions, and so doesn't change potential energy, kinetic energy before and after the impulse must be equal:

$$\begin{aligned} \frac{1}{2}m_1\mathbf{v}_1\cdot\mathbf{v}_1 + \frac{1}{2}m_2\mathbf{v}_2\cdot\mathbf{v}_2 &= \frac{1}{2}m_1\tilde{\mathbf{v}}_1\cdot\tilde{\mathbf{v}}_1 + \frac{1}{2}m_2\tilde{\mathbf{v}}_2\cdot\tilde{\mathbf{v}}_2 \\ m_1\mathbf{v}_1\cdot\mathbf{v}_1 + m_2\mathbf{v}_2\cdot\mathbf{v}_2 &= m_1\mathbf{v}_1\cdot\mathbf{v}_1 + 2I_1\mathbf{v}_1\cdot\hat{\mathbf{n}} + I_1^2/m_1\hat{\mathbf{n}}\cdot\hat{\mathbf{n}} \\ &\quad + m_2\mathbf{v}_2\cdot\mathbf{v}_2 + 2I_2\mathbf{v}_2\cdot\hat{\mathbf{n}} + I_2^2/m_2\hat{\mathbf{n}}\cdot\hat{\mathbf{n}} \\ 0 &= 2I_1\mathbf{v}_1\cdot\hat{\mathbf{n}} + 2I_2\mathbf{v}_2\cdot\hat{\mathbf{n}} + I_1^2/m_1 + I_2^2/m_2. \end{aligned}$$

Since we know $I_2 = -I_1$,

$$\begin{aligned} 0 &= 2I_1\mathbf{v}_1\cdot\hat{\mathbf{n}} - 2I_1\mathbf{v}_2\cdot\hat{\mathbf{n}} + I_1^2/m_1 + I_1^2/m_2 \\ 0 &= 2(\mathbf{v}_1 - \mathbf{v}_2)\cdot\hat{\mathbf{n}} + I_1/m_1 + I_1/m_2 \\ I_1(m_1 + m_2) &= 2m_1m_2(\mathbf{v}_2 - \mathbf{v}_1)\cdot\hat{\mathbf{n}} \\ I_1 &= \frac{2(\mathbf{v}_2 - \mathbf{v}_1)\cdot\hat{\mathbf{n}}}{\frac{1}{m_1} + \frac{1}{m_2}}. \end{aligned}$$

Therefore the final formulas for updating velocities when a collision is detected are

$$\begin{aligned} \tilde{\mathbf{v}}_1 &\leftarrow \mathbf{v}_1 + \frac{2(\mathbf{v}_2 - \mathbf{v}_1)\cdot\hat{\mathbf{n}}}{1 + \frac{m1}{m_2}}\hat{\mathbf{n}} \\ \tilde{\mathbf{v}}_2 &\leftarrow \mathbf{v}_2 - \frac{2(\mathbf{v}_2 - \mathbf{v}_1)\cdot\hat{\mathbf{n}}}{\frac{m_2}{m_1} + 1}\hat{\mathbf{n}}. \end{aligned} \tag{1}$$

3.3.2 Particle–Edge

When a particle hits an edge, we again want to apply an impulse to the particles associated to that particle and edge. Unlike the particle-particle case, there are three particles involved in a particle-edge collision: the colliding particle $\mathbf{x}_1$, and the two endpoints of the colliding edge $\mathbf{x}_2$ and $\mathbf{x}_3$. As before, we want to apply an impulse in the direction $\hat{\mathbf{n}}$ to each of these particles:

$$\begin{aligned} \tilde{\mathbf{v}}_1 &\leftarrow \mathbf{v}_1 + I_1/m_1\hat{\mathbf{n}} \\ \tilde{\mathbf{v}}_2 &\leftarrow \mathbf{v}_2 + I_2/m_2\hat{\mathbf{n}} \\ \tilde{\mathbf{v}}_3 &\leftarrow \mathbf{v}_3 + I_3/m_3\hat{\mathbf{n}}. \end{aligned}$$

We need to solve for the values of the unknown scalars I_1, I_2, and I_3 by applying conservation laws.

Let $\mathbf{x}_e$ be the closest point on the edge to the particle. We know we can write this point as $\mathbf{x}_2 + \alpha(\mathbf{x}_3 - \mathbf{x}_2)$ for a scalar α (see the section on particle-edge detection for the formula for α.) Rewriting this equation yields

$$\mathbf{x}_e = (1 - \alpha)\mathbf{x}_2 + \alpha\mathbf{x}_3.$$

We can now look at the angular momentum of the particle and edge about the point $\mathbf{x}_e$; we want it to be the same before and after the impulse is applied:

$$m_1\mathbf{v}_1\times(\mathbf{x}_1-\mathbf{x}_e)+m_2\mathbf{v}_2\times(\mathbf{x}_2-\mathbf{x}_e)+m_3\mathbf{v}_3\times(\mathbf{x}_3-\mathbf{x}_e) = m_1\tilde{\mathbf{v}}_1\times(\mathbf{x}_1-\mathbf{x}_e)+m_2\tilde{\mathbf{v}}_2\times(\mathbf{x}_2-\mathbf{x}_e)+m_3\tilde{\mathbf{v}}_3\times(\mathbf{x}_3-\mathbf{x}_e).$$

Substituting for the updated velocities and canceling terms from both sides, we get

$$0 = I_1\hat{\mathbf{n}}\times(\mathbf{x}_1-\mathbf{x}_e)+I_2\hat{\mathbf{n}}\times(\mathbf{x}_2-\mathbf{x}_e)+I_3\hat{\mathbf{n}}\times(\mathbf{x}_3-\mathbf{x}_e).$$

We know that $\mathbf{x}_1-\mathbf{x}_e$ lies in the same direction as $\hat{\mathbf{n}}$: in fact, the latter is the normalized version of the former, by definition. Therefore $\hat{\mathbf{n}}\times(\mathbf{x}_1-\mathbf{x}_e)=0$ and

$$\begin{aligned}
0 &= I_2\hat{\mathbf{n}}\times(\mathbf{x}_2-\mathbf{x}_e)+I_3\hat{\mathbf{n}}\times(\mathbf{x}_3-\mathbf{x}_e)\\
0 &= I_2\hat{\mathbf{n}}\times(\mathbf{x}_2-(1-\alpha)\mathbf{x}_2-\alpha\mathbf{x}_3)+I_3\hat{\mathbf{n}}\times(\mathbf{x}_3-(1-\alpha)\mathbf{x}_2-\alpha\mathbf{x}_3)\\
0 &= I_2\hat{\mathbf{n}}\times\alpha(\mathbf{x}_2-\mathbf{x}_3)-I_3\hat{\mathbf{n}}\times(1-\alpha)(\mathbf{x}_2-\mathbf{x}_3\\
0 &= (\alpha I_2-(1-\alpha)I_3)\hat{\mathbf{n}}\times(\mathbf{x}_2-\mathbf{x}_3).
\end{aligned}$$

If we assume that $\hat{\mathbf{n}}\times(\mathbf{x}_2-\mathbf{x}_3)$ is not the zero vector (this assumption is valid except in the corner case where the particle is travelling along the infinite line coincident to the edge), we must have that the coefficient is 0:

$$\begin{aligned}
\alpha I_2-(1-\alpha)I_3 &= 0\\
I_2 &= \frac{1-\alpha}{\alpha}I_3.
\end{aligned}$$

If we define the new quantity $I_e=\frac{I_3}{\alpha}$, we thus have

$$\begin{aligned}
\tilde{\mathbf{v}}_1 &\leftarrow \mathbf{v}_1+(I_1/m_1)\,\hat{\mathbf{n}}\\
\tilde{\mathbf{v}}_2 &\leftarrow \mathbf{v}_2+((1-\alpha)I_e/m_2)\,\hat{\mathbf{n}}\\
\tilde{\mathbf{v}}_3 &\leftarrow \mathbf{v}_3+(\alpha I_e/m_3)\,\hat{\mathbf{n}},
\end{aligned}$$

and only two unknowns remain.

The impulse must also conserve linear momentum. Therefore

$$m_1\mathbf{v}_1+m_2\mathbf{v}_2+m_3\mathbf{v}_3 = m_1\tilde{\mathbf{v}}_1+m_2\tilde{\mathbf{v}}_2+m_3\tilde{\mathbf{v}}_3.$$

Substituting and simplifying yields

$$\begin{aligned}
0 &= (I_1+(1-\alpha)I_e+\alpha I_e)\hat{\mathbf{n}}\\
0 &= I_1+(1-\alpha)I_e+\alpha I_e\\
0 &= I_1+I_e\\
I_e &= -I_1.
\end{aligned}$$

Lastly, we have conservation of energy:

$$\begin{aligned}
\frac{1}{2}m_1\mathbf{v}_1\cdot\mathbf{v}_1+\frac{1}{2}m_2\mathbf{v}_2\cdot\mathbf{v}_2+\frac{1}{2}m_3\mathbf{v}_3\cdot\mathbf{v}_3 &= \frac{1}{2}m_1\tilde{\mathbf{v}}_1\cdot\tilde{\mathbf{v}}_1+\frac{1}{2}m_2\tilde{\mathbf{v}}_2\cdot\tilde{\mathbf{v}}_2+\frac{1}{2}m_3\tilde{\mathbf{v}}_3\cdot\tilde{\mathbf{v}}_3\\
-2I_1\mathbf{v}_1\cdot\hat{\mathbf{n}}+2(1-\alpha)I_1\mathbf{v}_2\cdot\hat{\mathbf{n}}+2\alpha I_1\mathbf{v}_3\cdot\hat{\mathbf{n}} &= (I_1^2/m_1+(1-\alpha)^2I_1^2/m_2+\alpha^2I_1^2/m_3)\hat{\mathbf{n}}\cdot\hat{\mathbf{n}}\\
-2\mathbf{v}_1\cdot\hat{\mathbf{n}}+2\mathbf{v}_e\cdot\hat{\mathbf{n}} &= I_1(1/m_1+(1-\alpha)^2/m_2+\alpha^2/m_3)\\
\frac{2(\mathbf{v}_e-\mathbf{v}_1)\cdot\hat{\mathbf{n}}}{\frac{1}{m_1}+\frac{(1-\alpha)^2}{m_2}+\frac{\alpha^2}{m_3}} &= I_1,
\end{aligned}$$

where

$$\mathbf{v}_e = (1-\alpha)\mathbf{v}_2+\alpha\mathbf{v}_3.$$

The final formulas for the changes in velocity are thus

$$\begin{aligned}
\tilde{\mathbf{v}}_1 &\leftarrow \mathbf{v}_1 + \frac{2(\mathbf{v}_e - \mathbf{v}_1)\cdot\hat{\mathbf{n}}}{1 + \frac{(1-\alpha)^2 m_1}{m_2} + \frac{\alpha^2 m_1}{m_3}}\hat{\mathbf{n}} \\
\tilde{\mathbf{v}}_2 &\leftarrow \mathbf{v}_2 - \frac{2(1-\alpha)(\mathbf{v}_e - \mathbf{v}_1)\cdot\hat{\mathbf{n}}}{\frac{m_2}{m_1} + (1-\alpha)^2 + \frac{\alpha^2 m_2}{m_3}}\hat{\mathbf{n}} \\
\tilde{\mathbf{v}}_3 &\leftarrow \mathbf{v}_3 - \frac{2\alpha(\mathbf{v}_e - \mathbf{v}_1)\cdot\hat{\mathbf{n}}}{\frac{m_3}{m_1} + \frac{(1-\alpha)^2 m_3}{m_2} + \alpha^2}\hat{\mathbf{n}},
\end{aligned} \tag{2}$$

Before implementing these formulas, it may be helpful for you to think about the case when $\mathbf{x}_e$ is one of the edge's two endpoints, and verify that the formulas above reduce to the particle–particle case.

3.3.3 Particle–Half-Plane

Since the half-plane is fixed in place, we only have a particle to which we must apply an impulse:

$$\tilde{\mathbf{v}} \leftarrow \mathbf{v} + I/m\hat{\mathbf{n}}.$$

Conservation of energy tells us the right value of I:

$$\begin{aligned}
\frac{1}{2}m\tilde{\mathbf{v}}\cdot\tilde{\mathbf{v}} &= \frac{1}{2}m\mathbf{v}\cdot\mathbf{v} \\
2I/m\,\mathbf{v}\cdot\hat{\mathbf{n}} + I^2/m^2\,\hat{\mathbf{n}}\cdot\hat{\mathbf{n}} &= 0 \\
I/m^2 &= -2/m\,\mathbf{v}\cdot\hat{\mathbf{n}} \\
I &= -2m\,\mathbf{v}\cdot\hat{\mathbf{n}}.
\end{aligned}$$

The final formula for the change in the velocity of the particle is thus

$$\tilde{\mathbf{v}} \leftarrow \mathbf{v} - 2(\mathbf{v}\cdot\hat{\mathbf{n}})\hat{\mathbf{n}}.$$

3.4 Fixed Particles

Fixed particles and edges require special consideration. On the one hand, we don't want to apply any impulses to them that will change their velocity, since they are supposed to be fixed in place. On the other, we can't ignore them completely during collision detection and response, since we do want non-fixed particles and edges to interact with them.

Since changes in velocity are equal to impulse divided by mass, setting the mass of fixed particles to infinity is one way of ensuring that they are "immovable objects" during collisions. Implement this handling of fixed objects by doing the following:

- Instead of applying an impulse to a fixed particle, do nothing instead. (This step is necessary to avoid $\frac{\infty}{\infty}$ in certain degenerate cases.)
- Any time you would use the mass of a fixed particle while applying an impulse to a non-fixed particle, use infinity for that mass instead.

Coding tip: In C++, calling `std::numeric_limits<double>::infinity()` generates infinity.
Important note: Ensure that you have implemented the formulas for applying impulses (equations 1 and 2) exactly as they are written in this PDF. They are specifically formulated to work correctly when plugging in infinite masses. It is possible to rewrite these formulas in ways that are mathematically equivalent for finite masses, but give $\frac{\infty}{\infty}$ when fixed particles are involved.

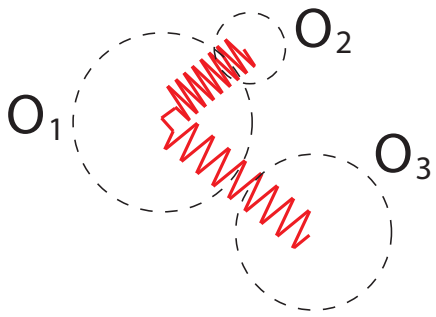

Figure 3: A conceptual illustration of the penalty method. A spring-like penalty force repels O_1 and O_2, since they are interpenetrating, and O_1 and O_3, since they are sufficiently close to each other (less than distance T apart, where T is the penalty method thickness). No force is exerted between O_2 and O_3 because they are far apart.

3.5 Coefficient of Restitution

When you drop a ball on the ground, conservation of energy suggests that the ball must bounce back up to exactly its original height. Of course, balls in the real world do not do this: bowling balls barely bounce up at all, and even the bounciest rubber ball only returns to a fraction of its original height.

Many complicated factors contribute to this dissipation of energy: plastic deformation, internal friction during elastic deformation, and production of sound waves are just a few examples. Simulating all of these small-scale phenomena is not feasible, so instead many simulations approximate the net coarse-scale behavior of these effects by a *coefficient of restitution* (COR). There are multiple ways of defining the COR, and we adopt the *Newtonian* COR as discussed in class (note: this is not the COR which is the ratio of the height of bounce to the dropping height).

Although the derivation is beyond the scope of this course, it turns out that *Newtonian* COR can be simulated by multiplying all impulses applied during contact response by the following scalar:

$$\frac{1+\text{COR}}{2}.$$

Notice that when COR is 1.0, the impulses don't change; this is as expected, since COR=1.0 corresponds to perfect conservation of energy.

Implement COR.

3.6 Penalty Method

For the second part of Milestone 1, you will switch gears and explore a completely different approach to handling collision: the penalty method. The main idea behind the penalty method is simple: for every pair of objects in the simulation, we add a new force – a *penalty force* – that does nothing if the objects are far apart. If the objects are close or colliding, on the other hand, the force acts like a spring, pushing them apart. The closer the two objects get, the stronger the force becomes, and the harder the simulation tries to push them apart. Figure 3 illustrates the penalty method conceptually.

For two objects O_1 and O_2 of radius r_1 and r_2, their penalty potential is given by the formula

$$V = \begin{cases} 0, & \|\mathbf{n}\| > r_1 + r_2 + T \\ \frac{1}{2}k(\|\mathbf{n}\| - r_1 - r_2 - T)^2, & \|\mathbf{n}\| \leq r_1 + r_2 + T, \end{cases} \tag{3}$$

where k is a global penalty force stiffness and T a global penalty force thickness (set in the scene XML file), and $\mathbf{n}$ is the usual shortest vector between O_1 and O_2. Notice that this potential bears a lot of similarity to that for a spring, given in Theme 1 Milestone 2, with two notable differences: firstly, the potential is constant (0) whenever $\|\mathbf{n}\| > r_1 + r_2 + T$, which holds whenever the objects are further than distance T apart. Since force is the gradient of potential, a constant potential when the objects aren't penetrating translates into zero force when the objects are far apart. Secondly, the length of the spring has been replaced with the length of the shortest vector between the objects, and the rest length has been replaced with the sum of the objects' radii plus T.

The penalty method has several pros and cons when compared to detecting and applying impulses as you've been doing in the first half of this milestone. As you will see, it is quite easy to implement: it's just another force you add to the simulation, without needing to do any separate collision detection or modification of the main simulation loop. On the other hand, the penalty force requires that you choose a value for the spring stiffness k: set it too high and the simulation becomes unstable, unless you decrease the time step and thereby slow down the simulation. Set it too low and the penalty force is very weak, resulting in a lot of "give" when objects collide, and in the worst case allowing objects to tunnel completely through each other. You also have to choose a value for T. Setting T to zero means that no force is applied on the objects until they are already interpenetrating, so they will visibly sink into each other during a collision. Setting T large prevents such artifacts since the springs will have time to repel the objects before they interpenetrate, but at the cost of unnatural-looking action at a distance: objects appear to slow down even though there is still a large gap separating them.

As usual, the penalty force is derived from the potential by taking the gradient with respect to position:

$$F = -(\nabla V)^T = \begin{cases} 0, & \|\mathbf{n}\| > r_1 + r_2 + T \\ -k(\|\mathbf{n}\| - r_1 - r_2 - T)(\nabla \mathbf{n})^T \hat{\mathbf{n}}, & \|\mathbf{n}\| \leq r_1 + r_2 + T, \end{cases}$$

where the precise formula for the rectangular matrix $\nabla \mathbf{n}$ depends on the two objects involved (particle–particle, particle–edge, or particle–half-plane; formulas for each case are derived below). Modify *Penalty-Force.cpp* and implement the gradient of the penalty potential $(\nabla V)^T$ in each of these three cases. You do *not* need to implement the corresponding Hessians – all test scenes and grading scenes will only use the penalty method with explicit Euler or forward-backward Euler.

3.6.1 Particle–Particle

For a pair of particles, $\mathbf{n} = \mathbf{x}_2 - \mathbf{x}_1$, where $\mathbf{x}_1$ and $\mathbf{x}_2$ are the positions of the first and second particle, respectively, so

$$\nabla \mathbf{n} = \begin{pmatrix} -\mathbf{I} & \mathbf{I} \end{pmatrix}.$$

Important Note: This matrix is expressed in local indices, with the left block corresponding to the degrees of freedom for the first particle, and the right block corresponding to those for the second particle. See Theme 1 Milestone 3 for a discussion on local versus global indices.

3.6.2 Particle–Edge

We have that $\mathbf{n} = \mathbf{x}_2 + \alpha(\mathbf{x}_3 - \mathbf{x}_2) - \mathbf{x}_1$, where $\mathbf{x}_1$, $\mathbf{x}_2$ and $\mathbf{x}_3$ are the positions of the particle and the edge's two endpoints. The gradient is a bit tricky to derive, since α depends also on the positions, and since we must take into account that α is clamped to $[0, 1]$. It turns out that the formula (in local coordinates) is

$$\nabla \mathbf{n} = \begin{pmatrix} -\mathbf{I} & (1-\alpha)\mathbf{I} & \alpha \mathbf{I} \end{pmatrix}.$$

3.6.3 Particle–Half-plane

From the first half of this milestone we have that

$$\mathbf{n} = \frac{(\mathbf{x}_h - \mathbf{x}) \cdot \mathbf{n}_h}{\|\mathbf{n}_h\|^2} \mathbf{n}_h,$$

so

$$\nabla \mathbf{n} = -\frac{\mathbf{n}_h \mathbf{n}_h^T}{\|\mathbf{n}_h\|^2}.$$

4 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.

4.1 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.