08214 Lab 4_2: Physically Based Simulation

In this lab we are going to build on the animation and collision detection implemented in the last lab and add some physics, some integration methods and momentum for sphere-sphere collisions.

Before we begin, update your repository to pull any changes that might have been made to the labs from SVN.

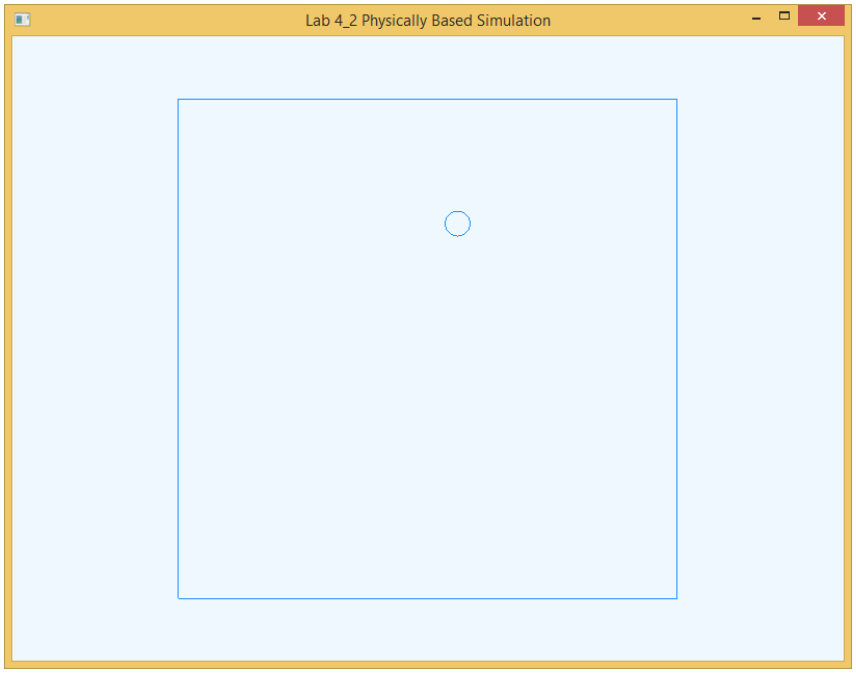
If you have any problems (or spot a mistake) one way you can get help is on the forum thread for lab 4.2 here at

http://forums.net.dcs.hull.ac.uk/t/1959.aspx

In the Program.cs file change the m_CurrentLab value to Lab.L4_2

```
private static Lab m_CurrentLab = Lab.L4_2;
```

This will run the correct window for this lab. You should see something like this:



Check out the code to ensure that you understand what is going on. Code to update the position of the circle is already written. In the OnLoad method add code that changes the circle's velocity from zero to (2,0,0).

```
mCircleVelocity = new Vector3(2f, 0, 0);
```

The circle should move off to the right. Commit your code to SVN with the message:

L42T1 Set my circle moving off to the right

Next use what you learnt in the last lab to add collisions between the circle and the box. We won't be resizing the square at all, so you can implement a pretty naïve collision detection algorithm.

Once you have your circle bouncing around the square commit your code to SVN.

L42T2 Added circle square collisions

The next step towards a more physically based simulation is to add some gravity. Gravity is a force between two masses that is dependent on both those masses, and the distance between them. However, on the earth because everything is approximately the same distance from the centre of mass of the earth, and because everything has a very small mass in comparison to the earth we can treat the acceleration due to gravity as a constant (9.81 ms^{-2} down).

After you calculate the new position, calculate the new velocity, including acceleration due to gravity.

```
mCircleVelocity = mCircleVelocity + accelerationDueToGravity * timestep;
```

You will also have to declare a vector for accelerationDueToGravity somewhere. For efficiency purposes it might be best to declare it somewhere at the class level.

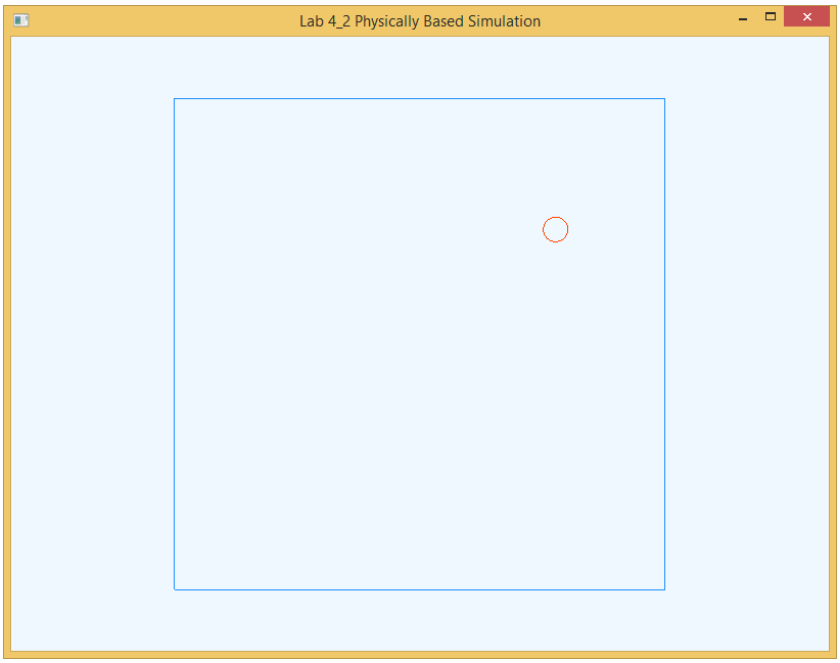
```
Vector3 accelerationDueToGravity = new Vector3(0, -9.81f, 0);
```

Run your code, and you should see the circle bouncing around under the effect of gravity.

L42T3 Added gravity using Euler Integration

But what does Euler integration mean? Well, in this case we use integration because we have to work using discrete timesteps. In this case we used Euler (pronounced "Oiler") integeration. Even though the velocity of the sphere is constantly changing throughout the timestep when we use Euler integration we assume that the timestep is so small that the change in velocity during the timestep is constant. This adds an error that has a tendency to increase the amount of energy in the system.

To constrast, lets add a second circle. Give it exactly the same starting parameters as the first circle but colour it a different colour. Again use Euler integration to simulated the second circle's path. The two circles should remain synchronized with one another. When you have done that commit your code to SVN.



In the screenshot above there is a blue circle perfectly aligned behind the red circle (trust me ;))

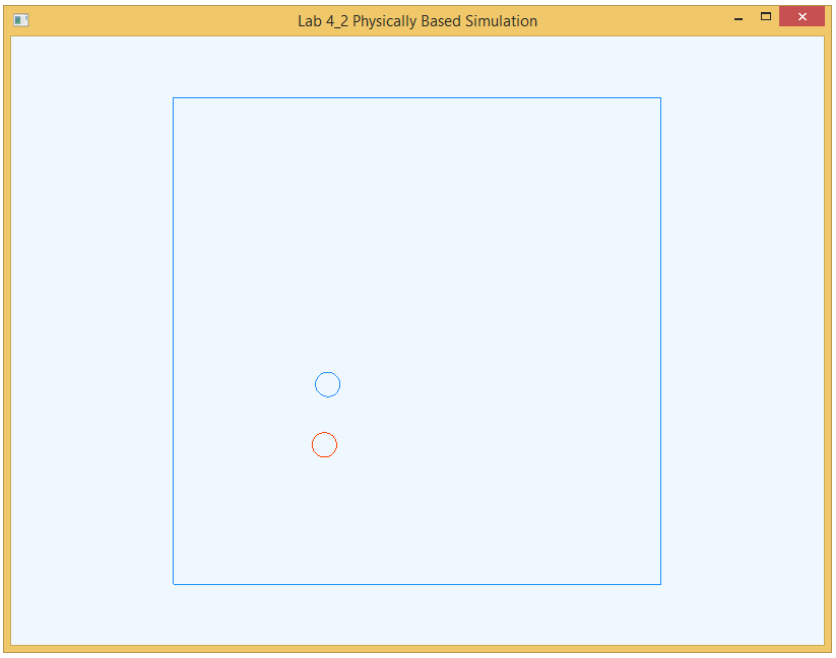
L42T4 Have two circles perfectly synchronized

Now, change the way the second circle is integrated. For the second circle only, instead of using the velocity at the beginning of the timestep, use the velocity at the end. Basically this means that we calculated the new velocity **before** we calculate the new position.

Your code might look something like this:

```
oldPosition = mCirclePosition2;
mCircleVelocity2 = mCircleVelocity2 + accelerationDueToGravity * timestep;
mCirclePosition2 = mCirclePosition2 + mCircleVelocity2 * timestep;
```

This method is called Symplectic Euler (or Semi-Implicit Euler). Run your code and see if there is any difference between the two circles.



At times the difference can be pretty significant. Commit your code to SVN.

L42T5 Integrating one circle using Euler integration and one using Symplectic Euler

There are other methods of integration available. For constant acceleration Verlet integration can offer a perfect solution, but only if combined to stable timesteps and perfect collisions. Arguably one of the best integration methods is fourth order Runge Kutta (RK4), but that takes up significantly more processing time than other methods.

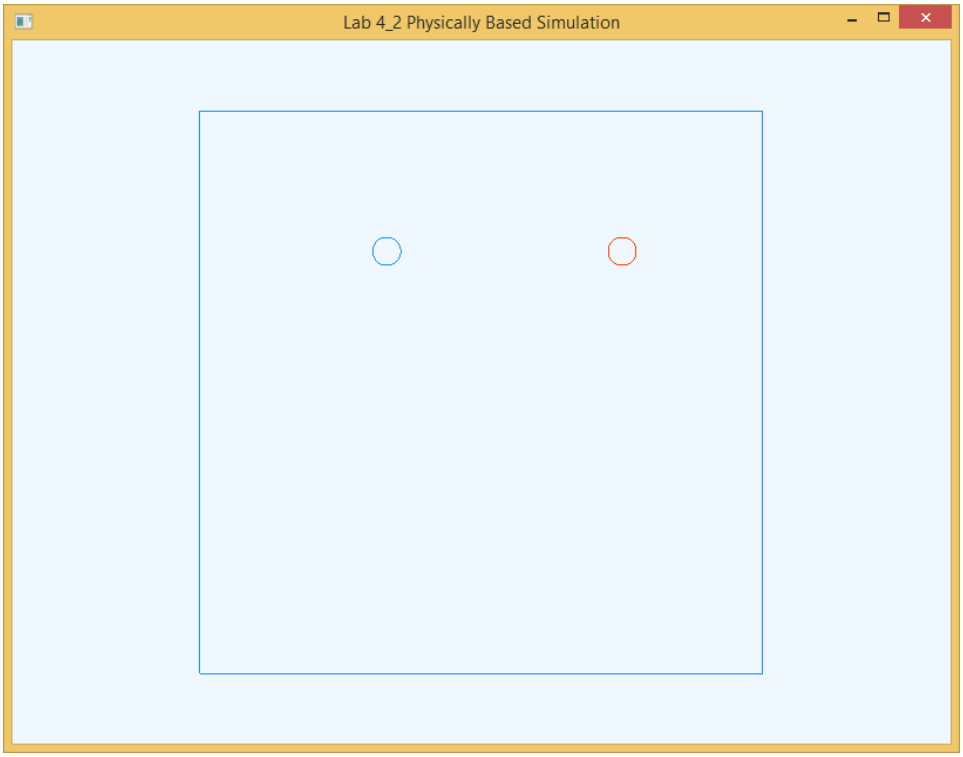
The next step in this lab is to add collision response for two moving circles. To keep things simple pick an integration method and stick to it for both circles. When you've done that commit your code to SVN.

L42T7 Two circles moving under gravity, using the <insert integration method> method of integration

Next, set the acceleration due to gravity to zero. Then and change the starting location and velocity of the circles.

```
mCirclePosition = new Vector3(-2, 2, 0);
mCircleVelocity = new Vector3(2, 0, 0);
mCirclePosition2 = new Vector3(2, 2, 0);
mCircleVelocity2 = new Vector3(0, 0, 0);
```

You should end up with something like the screen shot below. The blue circle is moving towards the red circle and passes straight through it.



L42T8 One circle moving without gravity. Need to add circle to circle collision detection

Use what you learnt from the last lab to add circle to circle collision detection. For now, fix the velocities to zero when the circles collide. Make sure your solution works from a number of angles.

L42T9 One circle bouncing off of another

After your testing set your circles back to these positions:

```
mCirclePosition = new Vector3(-2, 2, 0);
mCircleVelocity = new Vector3(2, 0, 0);
mCirclePosition2 = new Vector3(2, 2, 0);
mCircleVelocity2 = new Vector3(0, 0, 0);
```

This time we're going to transfer some of the velocity from one circle to another. As an initial hack, when a collision happens let's just swap the velocity from one circle to another. Your collision response code might look something like this.

```
Vector3 temp = mCircleVelocity;
mCircleVelocity = mCircleVelocity2;
mCircleVelocity2 = temp;
```

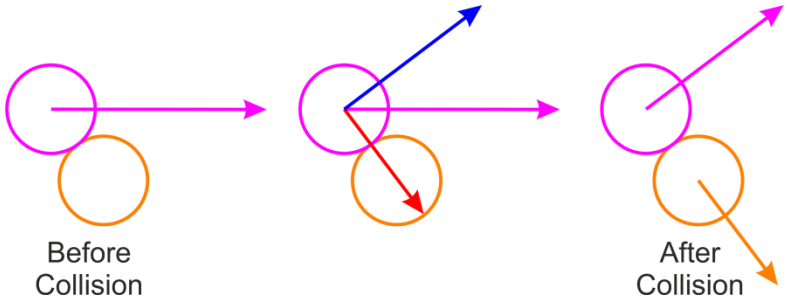
You should end up with a Newton's Cradle type effect. When the balls hit each other all the velocity from one ball is passed to the next. Once that works commit your code to SVN.

L42T10 Made a 2 circle Newton's Cradle

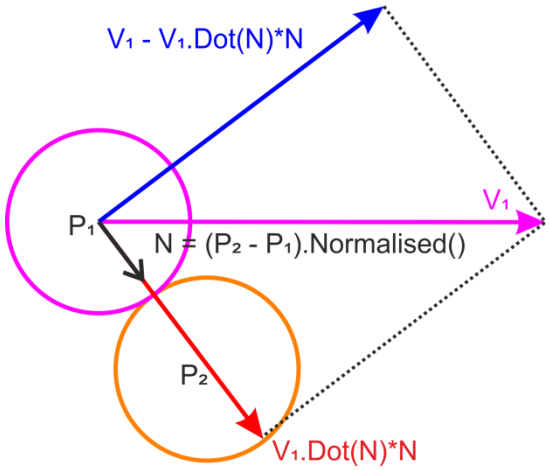
Now move one of the circles a little so that they collide at a glancing angle.

```
mCirclePosition = new Vector3(-2, 2.3f, 0);
mCircleVelocity = new Vector3(2, 0, 0);
mCirclePosition2 = new Vector3(2, 2, 0);
mCircleVelocity2 = new Vector3(0, 0, 0);
```

The collisions should still work, but the response is wrong. It worked before when the velocity was in line with the collision, but not now we're colliding at a glancing angle. We only want to transfer the component of the velocity that parallel to the direction of the collision. We want to retain the part of the velocity that is perpendicular to the direction of the collision.



To do this, we can use a dot product and a little bit of trigonometry. First we calculate the direction of the collision. Then we calculate the length of the component of the original velocity in the direction of the collision. Then we multiply the normalized collision direction by the dot product of the normalized collision vector and the initial velocity. That gives us the component in the direction of collision. To calculate the component that is perpendicular to the direction of the collision simply subtract the component of velocity that is perpendicular to the collision from the original velocity.



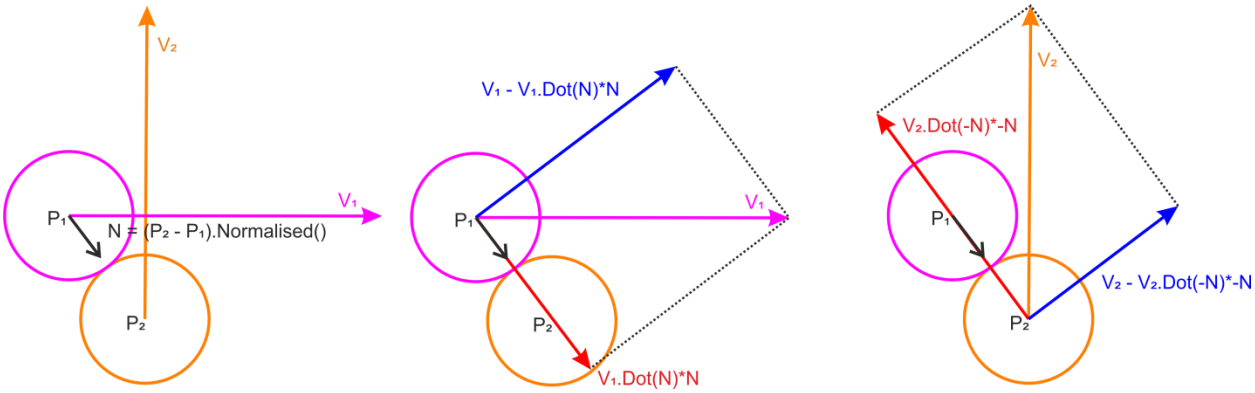
Hopefully the diagram above, along with the description should help you understand the calculation we're performing. Replace your collision response code with the appropriate calculation, and then commit your code to SVN with the following message.

L42T11 Calculated correct collision response for a collision with a glancing angle

So far, so good, but what if both circles are moving? Set up your two circles like this:

```
mCirclePosition = new Vector3(-2, 2, 0);
mCircleVelocity = new Vector3(2, 0, 0);
mCirclePosition2 = new Vector3(2, -2, 0);
mCircleVelocity2 = new Vector3(0, 2, 0);
```

Now the collision looks odd again. That's because we're only considering the velocity from one of the circles.



The diagram above should help you take into account the velocity for both spheres. At this stage we're just swapping the parallel components of each circles velocity (the red arrows), and we're retaining the perpendicular components (the blue arrows). Think carefully about the order that you move the spheres and process the collisions! Once you've made that work commit your code to SVN.

L42T12 Calculated correct collision response for two moving spheres, colliding at any angle

The next step is to change the radius of one of the circles. Let's double the radius of the seconds circle and see how our code holds up! With any luck you shouldn't need to do too much to get the collisions working correctly. Once again though, the response could be better. Commit your code to SVN.

L42T13 Colliding with circles of different radii

Now one circle is bigger than the other we need to think about the mass of the two circles when they collide. So far our collision model assumes that the circles have the same mass. If they are made of the same material then the bigger the circle is the bigger its mass is. In order to calculate the mass we need a density. For example, create a member variable mSteelDensity and set it to 7.8 (1000 kg/m^3).

If we assume each circle is actually a sphere then its volume is $\frac{4}{3}\pi r^3$ so the mass of that sphere is the density multiplied by the volume. The mass of each circle (or sphere) affects how much of the parallel component of each circle is retained, and how much is given up to the other circle. For the first circle

$$v_1 = \left(\frac{m_1 - m_2}{m_1 + m_2}\right)u_1 + \left(\frac{2m_2}{m_1 + m_2}\right)u_2$$

And similarly for the second circle

$$v_2 = \left(\frac{m_2 - m_1}{m_2 + m_1}\right)u_2 + \left(\frac{2m_1}{m_2 + m_1}\right)u_1$$

Note that if (as we have assumed so far) the two masses are equal these equations reduce down to a simple swapping of velocities. Note also that these equations only apply in the direction of the collision. Any velocity perpendicular to the collision is still retained by the original circle.

Modify your code to take the masses of the circles into account. To test your code you should check the total momentum before and after collisions. It should be the same (within a small margin of error). Momentum of a single circle is calculated by multiplying its mass by its velocity. Calculate the total momentum by adding the individual circle's momentum together. Momentum should be conserved despite the collisions.

Another way to test your code is to use some known values.

```
mCircleRadius = 0.2f;
mCircleRadius2 = 0.252f;
mCirclePosition = new Vector3(-2.5f, 2, 0);
mCircleVelocity = new Vector3(1, 0, 0);
mCirclePosition2 = new Vector3(0, 2, 0);
mCircleVelocity2 = new Vector3(-1, 0, 0);
```

A sphere of radius 0.252 should have approximately double the volume of a sphere that has a radius of 0.2. If these two collide the larger sphere should travel backwards with a velocity of 1/3 of a unit, whilst the lighter sphere should bounce off with a velocity of 5/3 units. Once you're satisfied that your code is correct commit your code to SVN.

L42T14 Correct collision responses including mass calculations

Another way to for the mass of a sphere to change is to change the density of the sphere.

```
mDensity = 1f;
mDensity2 = 2f;
mCircleRadius = 0.2f;
mCircleRadius2 = 0.2f;
mCirclePosition = new Vector3(-2.5f, 2, 0);
mCircleVelocity = new Vector3(1, 0, 0);
mCirclePosition2 = new Vector3(0, 2, 0);
mCircleVelocity2 = new Vector3(-1, 0, 0);
```

Check that if you double the density and keep the volume the same you get similar results, then check in.

L42T15 Checked the doubling the density of sphere with the same volume gives the same result as doubling the volume with the same density

So far we've only considered perfectly elastic collisions. This means that all energy is conserved. In reality with each collision some energy would be converted from kinetic (movement) energy to other types of energy, like sound, or heat. We can model this using a *Coefficient of Restitution 'e'* which is a value between 0 and 1 that is calculated using the ratios of the relative velocities of the colliding bodies before and after the collision. A coefficient of restitution of 1 will result in a perfectly elastic collision where all energy involved in the collision is retained. A coefficient of 0 will result in all energy in the collision being lost. The coefficient of restitution is not a property of a particular object, but of a particular pair of objects. When a coefficient of restitution is used to refer to a single object is it assumed that the second object is perfectly rigid and elastic. Note that the coefficient of restitution is only applied to the relative velocities. Taking the momentum equations from earlier

$$v_1 = \left(\frac{m_1 - m_2}{m_1 + m_2}\right)u_1 + \left(\frac{2m_2}{m_1 + m_2}\right)u_2$$

$$v_1 = \frac{m_1u_1 - m_2u_1 + 2m_2u_2}{m_1 + m_2}$$

$$v_1 = \frac{m_1u_1 + m_2u_2 + m_2(u_2 - u_1)}{m_1 + m_2}$$

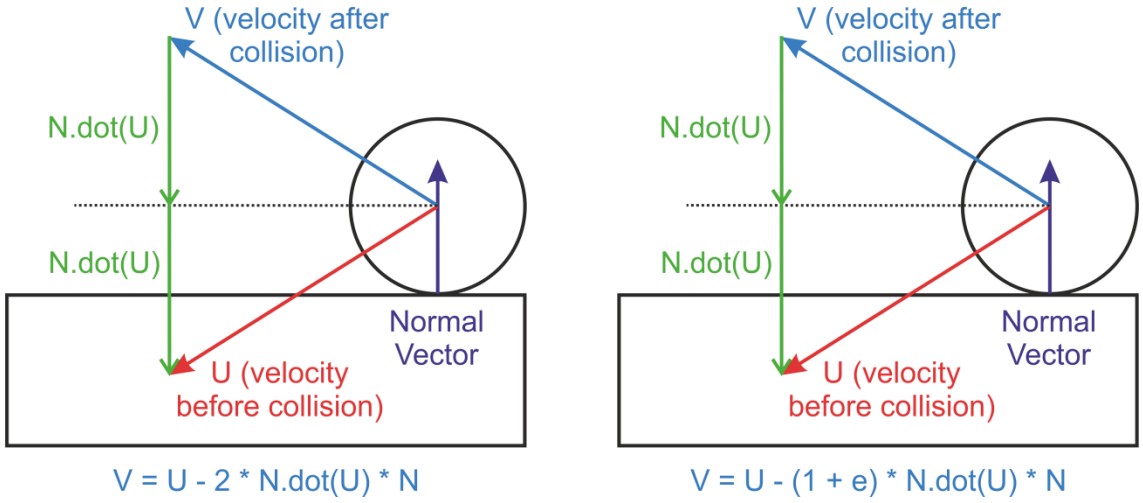
The coefficient of restitution *e* should be applied to the final term of the numerator.

$$v_1 = \frac{m_1u_1 + m_2u_2 + em_2(u_2 - u_1)}{m_1 + m_2}$$

Set a value for the coefficient of restitution for your collisions, and multiply the resultant velocities in the collision direction by the coefficient of restitution. What happens if the co-efficient is 1? What happens if the coefficient is 0? What happens if the coefficient is 0 and there is a large difference between the masses of the moving objects?

L42T16 Added the coefficient of restitution

The final step in this lab is to add gravity back in to the lab. This should just be a case of changing the gravity value we included at the beginning of the lab back to a realistic value. You should also apply the coefficient of restitution to collisions with the walls of the square. As the square doesn't move we have to remove it from the equation or risk getting odd results. The diagram below shows a comparison of the perfectly elastic collisions we've implemented previously with inelastic collisions. This should help you apply the coefficient of restitution when one object is perfectly rigid and elastic.



L42T17 Added gravity, and the coefficient of restitution for perfect collisions

Congratulations, you've made it to the end of the lab. In this lab we've focused on physical simulation. We've identified the need for numerical integration to add acceleration, and we've considered conservation of momentum in our collision response. Finally we've added elasticity to our simulation.