

# Modeling Motion Week 9 Computer Lab

## Billiards

In this lab we will use object-oriented programming and physics to model collisions between spherical objects in motion such as billiard balls. We will use the concepts of impulse and conservation of momentum to estimate the new velocities of a pair of objects just after they collide.

As always, an incremental approach usually works best. Start with a small working program and add one capability at a time until the program meets its requirements. In this case we would like our final program to be able to make any number of instances of a class called `Ball`, give them initial positions, velocities, and masses, and have them bounce properly off of each other when they collide.

As always, it's not necessary to turn in individual responses to each of the steps below, just the final version or as much as you are able to complete.

- 1) First create a `Ball` class and make two instances of it on screen. Use a VPython sphere object to make each ball's graphical representation. Create an instance of `sphere` in the `__init__(self)` method of your `Ball` class and assign it to a data attribute called `graphic`. You can store the definition of your new class in its own module, then create a program which imports your module and creates two instances of `Ball` in different positions.
- 2) Now add a `step()` method to your class which takes one argument, `dt`, (in addition to the implicit "self") and estimates the position of the ball after `dt` seconds. This implies each ball will need position and velocity attributes which can be updated each time step using Euler's method. These attributes should be VPython `vector` objects. Giving the `Ball` class its own position attribute rather than using the `sphere`'s position will make it possible to update the position more than once without drawing a new frame of animation each timestep. In your test program give your balls some initial velocity and put in a loop which calls each of their `step()` methods repeatedly for some time period so they move at constant speed. Don't forget the graphical representation of them will not change unless you also update `self.graphic.pos`.

Try to set the balls up so they crash into one another in preparation for testing the collision handling code we will add next.

- 3) Now add a method `handleCollision(self, other)` to `Ball` which takes another `Ball` object as an argument. This method will update the velocities of both `self` and `other` if a collision has occurred. The first step is simply to get it to recognize if a collision has occurred but not worry about computing the proper post-collision velocities.

In our simplified model we will assume a collision has occurred when the distance between the balls' centers is less than the sum of their radii. This implies we need to know what the radius of each ball is. Add a radius attribute and be sure to update `self.graphic` with the new radius. Making it a parameter of `__init__()` with a reasonable default value is a good idea.

To test this collision detection code, make it print out something like "Collision Detected" and either set the velocities of the two balls to zero or negate them to cause them to reverse direction. Then go to the test program and add a call to `handleCollision` after updating the positions of the balls (by calling `step(dt)`). The two Balls should now stop as soon as they are slightly overlapping and the print statement should continuously print because now the balls will be in contact forever since their velocities are set to zero.

- 4) Once collisions are being detected, add the necessary code to compute and set the proper new velocities of the two balls after a collision. To do this, you will need to compute the impulse force resulting from the collision which will depend on the masses of the two balls, their velocities just before the collision and a factor Newton called the "coefficient of restitution". This implies you will need to add a `mass` attribute to the `Ball` class.

The coefficient of restitution represents the fact in the real world collisions are not 100% elastic and therefore some energy is lost in collisions. This means the relative velocity of the two balls in the

direction of the collision normal after the collision is a little less than their relative velocity just before the collision. The number you need to multiply by to get the final relative velocity from the initial is the coefficient of restitution.

Using the law of conservation of momentum, the definition of impulse and the definition of the coefficient of restitution you should be able to set up the three equations needed to solve for the three unknown quantities (final velocities of two balls relative to the collision normal and impulse). This should be done using velocities in the direction of the collision normal which is a normal vector (length 1) along the line of action of the collision. For the case of two spheres of uniform mass distribution this is the line connecting the centers of the spheres.

The goal is to get an expression for the magnitude of the impulse in terms of the relative velocities of the balls in the direction of the collision normal, their masses and the coefficient of restitution. Once you know the magnitude of the impulse (a scalar) you can compute the impulse vector by multiplying its magnitude by the collision normal vector.

This impulse vector represents the change in momentum of one ball due to the collision and its negative represents the change in momentum of the other ball since they move apart after the collision. To get the final velocities, you need to divide the impulse by the mass of the ball its acting on, which gives the change in velocity, and add this to the ball's current velocity.

Once you manage this feat, you should have balls which collide properly in three dimensions given their masses and a coefficient of restitution. If the restitution is 1.0, the impulse expression reduces to the expression for perfectly elastic collisions and with a value of 0.0 it reduces to perfectly inelastic collision. Billiard balls are fairly elastic and would have a high value near 1.0.

## Challenge problems

- 5) Add some rolling friction to the balls. This is easy to add to the `step()` method by computing acceleration based on a force proportional to the velocity but in the opposite direction.
- 6) update your program to work for any number of balls
- 7) make a billiards table (no pockets) and add code to make the balls collide properly with the rails around the edge. The code below sets up a “break” scenario with a table on the x-y plane. It assumes your `Ball` class has a parameter `p` for setting the initial position of the ball, `v` for initial velocity, and `color` for the obvious.

```
balls=[] #this list will hold all Ball objects

# Rack 'em - make an inverted triangle of balls with random colors
for y in range(5): #rows of balls
    for x in range(y): #balls in each row
        color=(random(),random(),random())
        px = 2*(x-(y-1)/2.0)
        py = 25+1.7*y
        balls.append( Ball(p=vector(px,py,0), color=color) )

#add a cueball to the end of the list with slightly random position
balls.append(Ball(p=vector(uniform(-1,1),0,0),v=vector(0,226,0),color=(1,1,.7)))

#adjust viewing angle
scene.center=(0,25,0)
scene.forward=(0,1,-1)
scene.autoscale=0

#Make a green felt table
table = box(pos=(0,0,-1.1),size=(50,100,.2),color=(.1,.7,0))
```