In-Class Assignment 8 – *Rolling Wheel*

In this assignment, you will write a quick program that moves an object left and right across the screen using uniform motion. It will reflect off of the edges of the screen, and animation will be used to give it the illusion of rolling motion.

*Physics Goals*: Uniform motion, reflections, relative position and trigonometric functions.

*Programming Goals*:

*Instructions:*

1. *Create your wheel.* Your wheel will be the first object you move around the screen with physics.
   1. *Create the initialization method for the* Wheel *class.* You’ll need to take each of the arguments in the \_\_init\_\_() method and make class properties for them. The four arguments are:
      1. pos – a two element list with the x and y position of the wheel on the field *in meters*. You will only change the x-component in this application, but the y-component is necessary for drawing the object to the screen. It should become two object properties.
      2. vel – a single number for the x-component of the initial velocity of the wheel. This number should be in meters per second.
      3. size – a single number representing the radius of the wheel in meters.
      4. col – a gameclass color for the color of the wheel when you draw it.
   2. *Initialize your wheel.* Create the wheel in the main() function. Its radius should be a quarter to a half of a meter, and its position should a small integer times that in each component to keep it on the screen. The velocity should be zero at this point.
   3. *Include the wheel in the MVC calls.* Add the wheel in the MVC call from the main() function. Put them in the correct positions. You can figure out where that is by looking at the functions.
   4. Create the draw() method for the Wheel class. Here you have to do two things:
      1. Find the correct position and radius of the wheel *in pixels* for a given scaling factor sent to the draw() method from the view() function.
      2. Draw the circle using the PyGame draw.circle() method.
   5. Call the draw() method of the object from the view() function. You should now see a dot in the upper lefthand corner of your screen.
2. *Create a field object.* The Field class of the gameclass package represents your physical game area. Unlike the background, its dimensions are the physical units of the game area, and the field is passed to the engine() function to interact with the objects (whereas the background is sent to the draw function). This will allow you to scale your drawing.
   1. Create a field object. The field only needs the dimensions of the field as a data when initially called – this is a tuple of two elements. For this program, make the width (x-component) of the field 10 meters, and scale the y-component accordingly using the screen\_size variable.
   2. In the view() function, read the scale from the background object using the get\_scale() method, and use it in the call to the draw() method of the wheel.
   3. Add the wheel to the engine() and view() calls. It should be the third entry in each call.
   4. Call the imprint() method of the field object with the draw\_floor() function and the background object as arguments. This will scale you’re your wheel so that it is no longer a small dot when you run the program.
   5. Call the imprint() method in the event loop right after the background is resized. This will rescale your drawing.
3. *Make the wheel move.* Uniform motion is the basis of all motion, with complicated motion is determined by its deviation from uniform motion. It’s now time to make your wheel move across the screen.
   1. In the update() method of the Wheel class, modify the wheel’s position property by adding in the wheel’s velocity times the amount of time that has passed since the last update. *Remember the interval coming in is in milliseconds and you want your program to use seconds.*
   2. Call the update() method from the engine() function. If you run your program, it should not do anything yet, but it shouldn’t throw any errors.
   3. Change the initial velocity of the wheel to be 1 m/s. Your wheel should move now.
4. *Make the wheel reflect.* Simple reflections at the side of the screen or at a pre-defined surface position are an essential feature of many games. Here, you will create a set of reflections for the wheel so that it will continuously move
   1. Determine if the wheel should reflect off of the wall. Fill in the reflects\_at() call of the wheel class. Is should return False unless the edge of the screen lays between the leftmost and rightmost points of the ball.
   2. In main(), replace the None in the engine() call to the field object so that you can use it to make determinations about whether the object has hit an edge or not.
   3. In the engine() function, find the allowed positions of the wheel using the field object’s get\_size() method (and a little bit of logic). Call the reflect\_at() method with this. (If you put a print statement in your reflects\_at() method, you can check if it’s working before you work on the reflect() function, next).
   4. Fill in the Wheel class’ get\_velocity() and set\_velocity() methods.
   5. Reflect the wheel off of the edge. You’ll need to change the wheel’s position and velocity as discussed in your reading. You’ll need to use the get\_velocity() and set\_velocity() methods you just wrote plus some logic to change the velocity. You can use either the get\_position() and set\_position() methods (which you’ll need to write) or use the wheel’s update() method to change the position.

In-Class Assignment 9 – *Colliding Wheels*

Reflections against stationary objects are a special case of a collision. Specifically, the case of the wheel hitting the wall is a collision where the wall is assumed to have infinite mass. A more general collision is one where two moving objects bounce off of each other. In this exercise, you will animate two wheels with different masses and allow them to collide occasionally.

*Physics Goals*: Collision detection, collision resolution

*Programming Goals*: Functions, Methods, Objects, Reusing Code

*Instructions:*

1. *Reuse code from the rolling wheel*. When you’ve already built something in a previous program, you can cannibalize the code for the next program. In this case, you’re going to program two wheels that roll around on the screen, so you’ll want to
   1. Copy your Wheel class from the previous class and use it in this one.
   2. Do the same with the reflect() function.
   3. Do the same in the engine() function
   4. Do the same in the view() function.
   5. Do the same in the draw\_background() function.

Your code should now run, but the screen should be blank.

1. *Create two different wheels that move on the screen*. Every collision requires two objects. You’ll need two objects that start on opposite sides of the screen and move towards each other.
   1. Initialize the first wheel in the initialization section of the main() function as you did last time.
   2. Send both wheels to the view() function by placing them in the list wheels at the end of the initialization section of the main() function.

The wheel should now appear on the screen, but it not update.

* 1. Initialize the second wheel in the same place as the first one. It must have a different name. Give it a different initial position and different initial velocity.
  2. Add the second wheel to the wheels list.
  3. Send the wheels list to the engine() function. Then, index the list to make two individual objects in the engine(), and update each individually.

Your wheels should now roll around on the screen, but pass through each other.

1. *Modify the* Wheel *class for collisions*. When a heavy thing hits a stationary light thing, the heavy thing keeps moving in the same direction. When a light thing hits a stationary heavy thing, it bounces off of it.
   1. Add a mass property to the class, calling it self.mass. Modify the \_\_init\_\_() method to ask for the mass when the object is created.
   2. Create a method get\_mass() in the Wheel class. It will return the mass property.

1. *Create the collision code*. This includes both collision detection and collision resolution. Collision detection determines if there has been a collision, and should be part of the Wheel class, and collision resolution determines what to do with the two objects that have collided.
   1. Create a collides\_with() method for the Wheel class. It should take another wheel as an argument and it should return True whenever the distance between the centers of the two wheels are close enough that they will hit and returns False otherwise.
   2. In the engine() function, check if the two wheels hit by calling the collides\_with() method of one with the other as then argument. This should be done after the wheels’ positions have updated.

Place this in a print statement for now, and you will be able to tell if it works by monitoring the output.

* 1. Create a collide() function that takes two objects that have hit each other and changes their velocities in accordance with the rules of collision.
  2. In the engine() function, remove the print statement, and replace it with an if-statement that sends the objects to the collide() function when the wheels hit each other.

Collisions

where *v* represents a velocity before the collision, *v’* a velocity after the collision, and *m* a mass.

In-Class Assignment 10 – *1D Bowling*

This worksheet will introduce you to the concept of forces. The force applied to the bowling ball in this application will be friction, which slows objects down.

*Physics Goals*: Forces, Impulses, and Friction

*Programming Goals*: Inheritance, Exceptions

1. Add friction to the bowling ball’s motion. The bowling ball will need new properties, one for the friction coefficient (called self.dynamic in the template) and one for the impulse added over the interval (called self.impulse). The friction should an argument to the initialization method so that the programmer can choose to have different balls with different friction. Because of inheritance, this property will automatically be added to the Pin class.
   1. Modify the velocity by the impulse. Just as the position is updated every cycle, now you have to update the position every cycle.
   2. Clear the impulse after every update. You’ve used the impulse to modify the velocity, and you don’t want to use it in the next cycle, so clear out the old impulse from the object’s memory.
   3. Create a method to change the impulse due to the friction. You will need to find the direction and magnitude of the impulse from friction based upon the time interval sent to it, the gravitational acceleration sent to it, and the internal properties of the ball.
   4. Add friction to the ball. The larger the number, the more effect. Keep it below 1.

Now, the ball should stop rolling …eventually.

1. Create a Pin class using inheritance. A class Ball has already been provided. This class basically does everything that the Wheel class did in the previous worksheets.
   1. To create a class using another class, define the name of the class as a parameter.

class Pin(Ball):

this will give any object make from the Pin class all of the same properties and methods that the Ball class has.

* 1. Then, create a new \_\_init\_\_() method under it. This method is already there, you just have to initialize the properties with the correct values. If you have trouble figuring things out, you can look to the \_\_init\_\_() method from the Ball class for inspiration.
  2. Consider each method in turn, starting with the \_\_init\_\_() method. Is there anything that has to be different about this method because you now have a rectangular object instead of a circular one? Does the method need to be modified to be used in the collide and reflect functions? If so, change it. Things to think about:
     1. The size of the rectangle has two components rather than one.
     2. The width is the full width and not the diameter.
     3. The x and y positions of the Rectangle should be its center (because of the physics) remember this when calling the draw\_pin() function in the draw() method..
  3. In the main() function, create a stationary pin and add it to the pins list.

When you run the program, the bowling ball should hit the pin, and it should bounce off the edge of the screen.

1. Add more pins. Add two more pins to the pins list. They should bounce around nicely.

In-Class Assignment 11 – *Plank Walking*

This worksheet will introduce you to working with reciprocal forces, action and reaction. A man will walk on a freely translating plank

*Physics Goals*: Newton’s Third Law and contact forces.

*Programming Goals*: MVC architecture

1. Make the avatar walk the plank. The main action of this activity has the avatar walking on top of the platform.
   1. Add in user control of the avatar. In the control() function, add an if-statement so that when there is a KEYDOWN event, the avatar exerts a total impulse of 300 kg·m/s. Use the add\_impulse() method of the avatar to do this.
   2. Make the avatar go. Add an update() call for the avatar below the if-else statement in the engine().
   3. Add friction. Now make the avatar stop by adding friction as he walks along the platform if he is actually on top of the platform. Replace the assignment of the friction variable with an add\_friction() call to the avatar.
2. Add in Newton’s Third Law. As the avatar pushes forward on the plank, the plank moves backward on its rollers. This will move the plank backwards and the man moves forward. This is a general property of forces: forces have to be applied on something by something else, and anything that can feel a force is a proportionate generator of that force.
   1. Send the frictional impulse back to the engine from the avatar. Send the total amount of impulse applied in the add\_friction() method of the Box class back to the function call. Save this value as a local variable in the engine() function.
   2. Send the amount of frictional impulse to plank. The plank should gain as much impulse in the opposite direction to the impulse the man imposes on himself while walking, all through the frictional force. Use the add\_impulse() method of the plank to do this.
   3. Make the platform move. Add an update() call for the plank above the if-else statement in the engine().

The plank should now move under the man when the man moves forward, but they are not completely connected. If you change the velocity of the plank, it will slide under the man.

1. Account for relative motion. As the avatar walks on the plank, he pushes it backwards, but his foot remains on the same place. Here, you need to modify the Box class so that the avatar’s coordinates are relative to the plank’s position – when the plank moves backwards, then so does the guy.
   1. Send the distance travelled back to the engine. The Box class’ update() function should send the horizontal distance travelled by the object back to function call for use in the engine() function. Save that distance to a local variable in the engine().
   2. Make the avatar move backwards. The relative position of the stick figure should move backwards by the distance that the plank moved. You should model this by sending the distance to the avatar using its move\_base() method. This way, if the plank has an initial speed, the man will move on the plank.

If you change the plank’s initial velocity (to (-1.5,0.0), say) then you can see this.

In-Class Assignment 12 – *1D Rocket Lander*

Gravity is a ubiquitous force in physics problems. Except in distant space or for elementary particles that move very fast, gravity produces effects in almost every mechanical situation, and so we

*Physics Goals*: Thrust, momentum, gravity.

*Programming Goals*: Conditional operation, keyboard control.

1. Activate gravity. Gravity impacts objects no matter what their position is.
   1. Create a get\_mass() method for the Rocket class. It needs to return the entire mass of the rocket, including both its fuel and its superstructure. Call this function in the engine() and save the result to a local variable.
   2. Calculate the gravitational force (F = mg) in the engine().
   3. Add it to the rocket’s impulse.
2. Create the ship’s thrusters. We like to control game objects through physics, not directly updating its position but imposing forces on it.
   1. Create methods for the rocket to activate and deactivate the thrusters, activate\_thrust() and deactivate\_thrust(). activate\_thrust() should take a force in Newtons to apply to the ship as long as the ship is active and put it into the self.force property of the rocket. It should also set the self.active property of the rocket to True. deactivate\_thrust() should set the force to zero and the active flag to False.
   2. Keep the thruster on only when the spacebar is depressed. Fill out the control() function so that on a KEYDOWN event, check to see if the event.key is K\_SPACE. If it has been pressed, activate the thrusters with 140 kN. On a KEYUP, deactivate the thrusters.

1. Keep track of fuel. In the rocket’s
   1. Burn fuel during use. In the rocket’s update() method, reduce the amount of fuel by 1 kg/ms of activation when self.in\_flight and self.active are both True.
   2. Stop burning fuel when it’s used up. When self.fuel is zero, there is no fuel to burn and so the force should drop to zero and the self.active flag should revert to False. The fuel should never be negative. Make this check in the update() function only when the ship is active and in flight.
   3. Don’t let the user burn any more fuel. Change activate\_thrust() so that it does turn the thrust on when there is no fuel.