In-Class Assignment 13 – *Rolling Ball*

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. *Make the ball move.* The ball will move like the Wheel did in one dimension, but instead it will move in two dimensions..
   1. In the update() method of the Ball class, modify both of the ball’s position properties by adding in the product of the velocity components in their respective directions and the amount of time that has passed since the last update.
   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 in the x-direction and something similar in the y-direction. Your wheel should move now.
2. *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 hits\_edge\_of() call of the Wheel class. Is should return False unless the edge of the field overlaps some part of the ball, in which case it should return True. This has to work for both the top and the bottom walls.
   2. In the engine() function, call the hits\_edge\_of() method with this. (If you put a print statement in your hits\_edge\_of() method, you can check if it’s working before you work on the bounces\_off() method, next).
   3. Reflect the wheel off of the edge. When the ball bounces\_off() a Field object, you’ll need to change the wheel’s position and velocity as you did previously, but this time you’ll do it in a method using the object’s properties directly and you’ll need to do it for both the top of the field and the bottom of the field.
   4. In the engine() function, call the bounces\_off() method of the ball whenever the ball hits the edge of the field.

In-Class Assignment 14 – *Angled Wall I: Normals and Tangents*

Reflections against walls that are

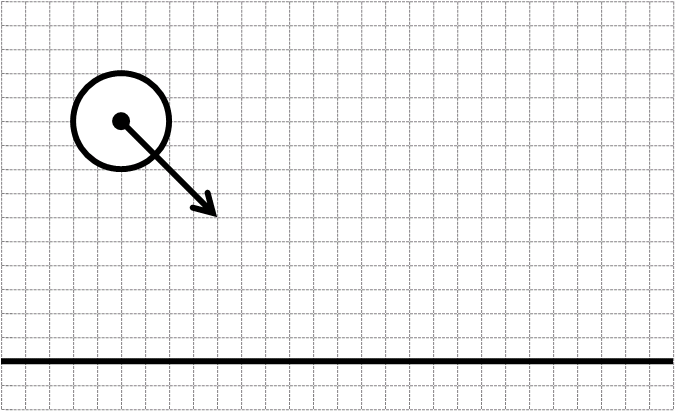
*Physics Goals*: Collision Resolution in 2D, Calculation of Normals and Tangents

*Programming Goals*:

*Instructions:*

Perform the following steps on diagram at the end of the section and symbollically:

1. Draw a line for the initial path of the ball. Extend the velocity vector to meet with the wall. We’ll call this the collision point—the point where the ball was pushed in the most in the first diagram. You will use the distance from the point of the vector to the x-intercept of the line you drew as the hypotenuse of a right triange. Label the distance from the vector’s point to the wall’s *x*-intercept ***d***.
2. Draw a right triangle. Draw the right triangle so that the diagonal is the distance that you marked off in step (1), the bottom, horizontal side ends at the line’s intersection with the wall, and the left, vertical side starts at the tip of the vector. Label the horizontal side ***x*** and the vertical side ***y***. *You will use this triangle to determine the vector components of the velocity*
3. Find the direction of the velocity. Write the unit vector for the direction of the velocity . Write it in list notation using the three values you found above. Think about it, and make sure they have the right signs, and then make sure it’s a unit vector by add the squared components to make sure that they sum to one.
4. Find the vector components of the initial velocity. Multiply the direction of the velocity by its magnitude. Write it as a list.
5. Reverse the component of velocity normal to the floor. Write the final velocity so that it has the same *x*-component as the initial velocity, but multiply the initial velocity’s *y*-component by -1.

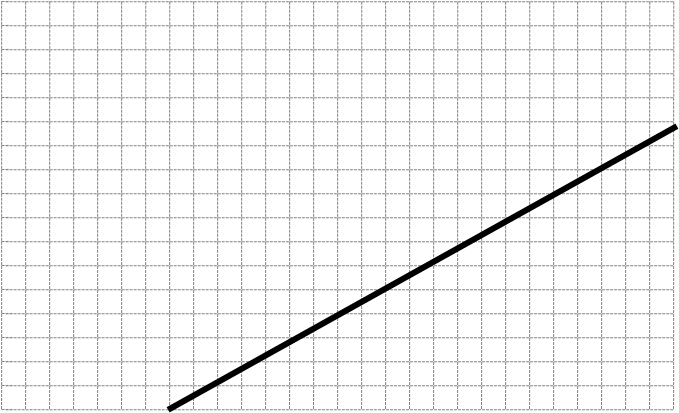


*Normal Direction of a Tilted Wall*

In the final section of this exercise, you’ll use the tangential (parallel) and normal (perpendicular) directions from the wall in the same way that you just used the *-* and - directions for the reflection off of the floor.

Perform the following steps on diagram at the end of the section and symbollically:

1. Choose a point on the wall. This will be one end point of the hypotenuse of a right triangle. Label the distance from the wall’s *x*-intercept to the point ***d***.
2. Draw a right triangle. Draw the right triangle so that the diagonal is the distance along the wall that you marked off in step (1), the bottom, horizontal side starts at the wall’s *x*-intercept, and the right side, vertical side starts at the point you chose. Label the horizontal side ***x*** and the vertical side ***y***. *You will use this triangle to determine the tangential direction in unit vector notation.*
3. Find the tangent direction. Write the unit vector for the direction tangent (parallel) to the wall in list notation. Square and add the components to make sure that they add to one.
4. Draw a perpendicular line. Draw the line perpendicular to the wall through the point that you drew on the wall.
5. Mark off a similar distance on this line. Now mark off the distance ***d*** in step (1) on the line you drew in step (4). Mark it from the point you chose on the wall in step (1), and up and to the left of it.
6. Draw a second right triangle. Now draw another right triangle with the distance from step (5) as the hypotenuse and two other sides along the *x*- and *y*-directions, respectively. *You will use this triangle to determine the normal direction in unit vector notation.*
7. Determine the lengths of its two sides. Rationally determine what the lengths and signs of the two sides of the new triangle are. Write them here.
8. Find the normal direction. Write the unit vector for the direction normal (perpendicular) to the wall in list notation . Square and add the components to make sure that they add to one.



*Use These Relationship for Simple Calculations*

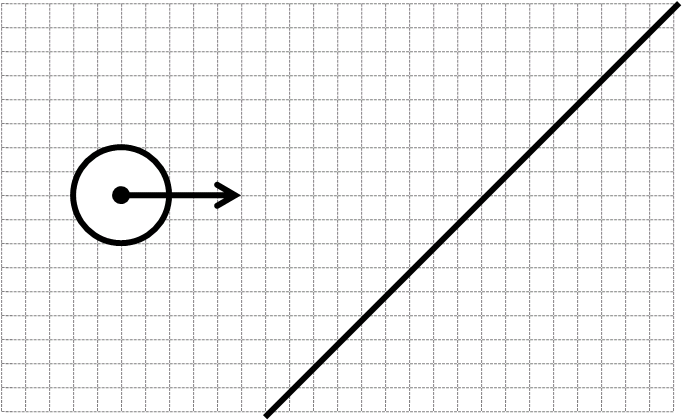
In order to help you get a feel for the relationship you just derived, carry out the following three specific calculations for finding the tangential and normal directions.

1. Let the wall be horizontal, so .
   1. What is the tangential direction (unit vector)?
   2. What is the normal direction (unit vector)?
   3. What is the inclination angle of the wall?
2. Let the lengths of the two sides be equal, .
   1. What is the tangential direction (unit vector)?
   2. What is the normal direction (unit vector)?
   3. What is the inclination angle of the wall?
3. Let the lengths of the two sides be related by .
   1. What is the tangential direction (unit vector)?
   2. What is the normal direction (unit vector)?
   3. What is the inclination angle of the wall?

*Reflection Off a Tilted Wall*

Now, you’ll put all of this together to figure out what the final velocity of a ball that was initially moving horizontally with a speed and reflects off of a wall tilted at 45˚. You’ll be able to use the results of (2)a., above.

1. Write the velocity. Use list notation.
2. Write the tangential direction. You may copy from (1)a. in the previous section.
3. Find the tangential component. To find the tangential component of the velocity, multiply each component of the initial velocity by the corresponding component of the tangential direction, then add them. *This is called the projection of the velocity on the wall’s direction*.
4. Write the normal direction. You may copy it from (2)b. in the previous section.
5. Find the normal component. To find the normal component of the velocity, multiply each component of the initial velocity by the corresponding component of the tangential direction, then add them. *This is called the projection of the velocity on the wall’s normal direction*.
6. Reverse the normal component. Multiply each component from (5) by -1.
7. Find the final velocity. Multiply the initial tangential component by the tangent direction and the final normal component by the normal direction. Add the tangential and normal velocity vectors component-wise. The resulting vector is the final velocity.



In-Class Assignment 15 – *Angled Wall II: Slanted Tunnel*

In two dimensions, you have a new problem: what if the object that you interact with is not directed along a cardinal direction like it was in the *Rolling Ball* exercise? You need the ball to interact with the wall as if the wall’s direction were a cardinal direction. To do this, we rotate the coordinate system, resolve the collision, and rotate it back to its original coordinate system.

*Physics Goals*: Normalization, distances, reflections

*Programming Goals*:

*Instructions:*

1. Find the normal and tangent directions to the wall..
   1. Find the wall’s tangent vector. In the find\_tangent() method of the Wall class, scale the unnormalized slope vector to be unitary.
   2. Find the wall’s normal vector. In the find\_normal() method of the Wall class, use the tangent to find the unit normal.

If this has been done correctly, you should have two slanted walls on the field.

1. Determine if the ball hits a wall. Finding whether a ball hits a horizontal or vertical wall is fairly straightforward. So is finding whether it hits another ball. But determining if it hits a slanted wall is more complicated.
   1. Find the distance from the wall to an object. This distance is technically the distance from the central line of the wall to the center of the object. Use the normal and tangent vectors to do this in the distance\_to() method of the Wall class.
   2. Determine if that distance is close enough to hit. Use the distance method and the sizes of the wall and the ball to determine if the ball hits the wall in the wall’s is\_hit\_by() method.
   3. Call the is\_hit\_by() method for each wall in the engine() function. Put it in a print statement to check if it’s working.
2. Resolve the ball’s collisions with the wall.
   1. Create a method to resolve the collision. In the Ball class, work up a method using the tangent and normal vectors of the wall to reverse the velocity component perpendicular to the wall and leave the parallel component unchanged. Place it in the bounces\_off() stub.
   2. Call the bounces\_off() method from the engine when the ball hits the wall.

In-Class Assignment 16 – *Galton’s Board*

Galton’s board is a famous example of a sensitive dependence upon initial conditions in mechanical phenomena (deterministic chaos). Depending upon slight differences in the initial position of a ball that is incident on a triangle of pegs, the ball will fall into any one of a number of bins at the bottom of the triangle. This is what you’ll simulate today.

I have included the code for the gravity, the walls, and the bin counting. You will have to determine the direction of a reflection of a round object (the ball) off of another round object (the pin) and create a large number of pins by using for-loops for their creation, execution, and visualization. For loops are used routinely in programming to

*Physics Goals*: Normals and tangents

*Programming Goals*: List Construction, for-loops.

1. Draw the pegs. I have included some posts at the edges of the walls. These posts will keep the .
2. Draw the posts using a for-loop. A for-loop over a list will do the same (indented) thing for all of its members. In the template, this is done for the list walls (simple) as well as the list scores (more complicated). When the statement says

for wall in walls:

wall.draw(…)

Python will draw each wall in sequence. Just like an if-statement, Python will do everything that is indented under the for-statement, but in this case rather than doing it conditionally, it will do it for each and every element wall in the list walls.

1. Create a single peg. You can get a good idea about how well your code works by creating a single peg and seeing how well it works when the ball collides with it. So, here you will create a single working peg at the mouth of the chute that is dropping the balls.
   1. Create a single Post object. Use the Post class to create a single post at the mouth of the chute. The point should be 25 m to the right of the edge of the board and 7 m below it.
   2. Add it to the pins list. Some other pegs are part of the background (they’re used to keep the ball from entering the wall from its edge), and you’re going to want to do everything with them that you do with the pegs on the triangle. You should now see a single peg where you expect it to be: in front of the mouth of the chute. To add the post to the list use the append() method of the posts list.
   3. Find the normal and tangential directions for the reflection. The normal direction of a reflection of a round object off of a round obstacle is in the direction of the radius vector from one object to another. Do this calculation in the Post class. Knowing the normal direction, you should be able to find the tangential direction.
   4. Create a distance function. The peg will need a distance function between the ball and the peg. Use the distance\_to() method in the Post class.
   5. Activate the reflection. In the engine() function, check to see if there are any reflections on the pegs, and if so, make them reflect using the ball’s bouces\_off() method. You will have to use a for-loop. Both the walls list and the posts list are combined into an obstacles list when sent to the engine() function, so you’re going to activate the walls and the pegs simultaneously.
2. Create a triangular array of posts. To create a Galton’s board, you will have to make a large number of identical pegs. These pegs are arranged in a grid, each line having one more peg than the previous line, each peg being 4 m left or right of its neighbors, and each line being 4 m below the previous line. You will do this in what is called a nested loop, one loop inside of the other.
   1. Given the description above, in terms of the variables i and j, write the x- and y-position of the object that is in column i of row j:

x(i,j) =

y(i,j) =

* 1. Create a function that you will call in the initialization sequence in main() that will create a line of the correct number of pegs in a given row. A good way to do this is to create a list of numbers using the range() statement to create a list to iterate through in the for-loop cycling through j. You don’t need to name the peg, you can just add it to the posts list by defining it in the append() method after calculating its position.

You should type the range statement with a few numbers in it into the command line to get a feel for what it does. If you define the row number i and use that in your position assignment, then it will be easier to convert this into a nested loop. When you’re done with this, you should see a single row of posts on the screen that reflect the ball if it hits.

* 1. Nest the loops by cycling the code above through 8 rows.

In-Class Assignment 17 – *Kiss Shot*

Pool is a common example for collisions. It’s a common game that most people are familiar with and its round balls move in two dimensions and allow for simple collision detection and resolution. In this exercise, we use a simplified pool simulation to demonstrate the physics of collisions in multiple dimensions.

*Physics Goals*: Collisions in two dimensions

*Programming Goals*:

1. Make the cue ball move. Before the cue ball can hit the object ball, it has to start moving. Use the keyboard to start it moving.
   1. The control function is hidden in an if-statement. When the go variable is True, the simulation will run. In the event loop, change to go variable to true when the spacebar is pressed. A useful list of PyGame key constants can be found at:

<http://thepythongamebook.com/en:glossary:p:pygame:keycodes>

1. Make one ball hit the other. The trick to this is rotating your coordinate system so that the normal direction acts the way that the collisions did in one dimension and the tangential direction does not change. This is exactly the same trick as we used for reflections off of a wall and off of pegs, only using the collision code.
   1. Make the ball detect a collision with another ball.
   2. Resolve the collision.
      1. Tilt the axes to the normal-tangential plane. This involves finding the normal and tangential components of the velocity of each ball.
      2. Perform a 1D collision on the normal components of the two objects. This can be done exactly as in the *Colliding Wheels* assignment.
      3. Recombine vectorally to find the resultant velocities.
   3. Use one ball’s collision detector to determine if there has been a collision in the engine() and then use the same ball’s collision resolver to find the final velocities of the balls. If your algorithm is correct, the ball will go to the pocket.
2. Make the pocket catch the balls. One of the most important things in pool is detecting that a ball has entered into a pocket, which you will do in this step.
   1. Create a method in the Pocket class that detects if the ball’s center is in the hole.
   2. Create a method in the Ball class that places the ball in the center of the pocket and makes it stop moving.
   3. Use the first method to activate the second method in the engine().

In-Class Quiz – Sliding Rectangle

In this quiz, you will draw a sliding rectangle on the screen. It will have a flashing arrow in its center, pointing in the direction of motion. As is usual in this course, the object’s properties are specified in SI units and only scaled to fit the screen in its draw() method. The scaling constant has already been chosen for you.

The method for drawing the arrow has already been created for you, you will just have to manipulate it. You can find useful specifications in the docstrings of the various methods of the Mover class.

Each activity will be scored on two criteria, equally weighted: (1) is the effect correct? (2) is the procedure the kind that we use in class?

Work from the template found on Blackboard, replace the work “template” if the quiz file with your last name, write your name in the “Author” comment in the code, complete the following steps, and e-mail your work to [games.rantschler@gmail.com](mailto:games.rantschler@gmail.com).

Questions:

1. Draw a rectangle on the screen. The object already knows its physical location on the field and its physical size, you don’t have to change them. You have to use them and the scaling factor to draw the rectangle in the right place on the screen. Remember, the location of the object is its *center*.
2. Animate the rectangle. The objects already has a velocity in SI base units (m/s). You will have to make it move by correctly writing an update() function.
3. Draw the arrow. Draw an arrow in the center of the rectangle. The draw\_arrow() method of the Mover class has already been written for you. You will just have to figure out how to use it.
4. Make the arrow point the correct way. Change your arrow call so that it is drawn pointing in the direction of motion based on the velocity of the object.
5. Flash the arrow. Make the arrow cycle through a series of colors as the object moves. Use at least three colors.

Extra Credit:

Make the program print the number of times that the rectangle wraps around the screen.