Physics

Due: Wednesday, February 20 @ 11:59pm (Scala Only)

Updates and Pitfalls

- You may add more functionality to the 4 Scala classes from step 3 of the project structure to complete the objectives, but do not use this functionality in your testing. Your tests are ran against complete submissions on the server including these 4 classes, complying with the project structure, but your added functionality will not be present
- TestDetectCollision now onlys tests your own code and the 3 failing submissions, not my correct code

Objective

Gain experience with Object-Oriented Programming and modifying the state of objects through references

Description

Simulate the physics of objects in a world with respect to velocity, location, gravity, and collisions with walls. Our simulated world will have a subset of the laws of physics found in the real world. In our world only the following laws apply:

- The word is 3-dimensional with dimensions labelled x, y, and z
- The world has a constant acceleration due to gravity in the negative z direction (positive z is up). This acceleration will be stored as a positive number
- The world contains a list of objects each with a 3d location vector and 3d velocity vector
- The world contains a list of boundaries that cannot be crossed by the objects. Boundaries do not move and extend infinitely in the z dimension
    - Boundaries are defined by their two end points
- The ground is the plane at z=0. No objects can fall below the ground
- An object on the ground will have its velocity in the z direction set to zero

Project Structure

1. Create a new project in IntelliJ
2. In the src folder create a package named physics

3. In the physics package create Scala classes (To clear some confusion these classes are given below.)
    a. **PhysicsVector**: With a constructor that takes 3 doubles (x, y, z) and stores them in state variables
    b. **Boundary**: With a constructor that takes 2 PhysicsVectors and stores them in state variables. These vectors represent the end points of the boundary
    c. **PhysicalObject**: With a constructor that takes 2 PhysicsVectors and stores them in state variables named “location” and “velocity” in this order (location is the first parameter, velocity is the second). These variables will be accessed during grading so the names must be exact
    d. **World**: With a constructor that takes a Double representing the acceleration due to gravity of the world, a public state variable named objects of type List[PhysicalObject] (Not in the constructor), and a public state variable named boundaries of type List[Boundary] (Not in the Constructor). The objects and boundaries will be accessed during grading so the names must be exact (gravity is not accessed by the grader)
4. In the physics package create a Scala object named Physics. This is where you will write all your code for the assignment. The 4 classes are already finished, though you may add methods to them if it will help you complete the objectives
5. In the src folder create a package named tests

```
package physics

class Boundary(val start:PhysicsVector, val end: PhysicsVector) {

}
```

```
package physics

class PhysicalObject(val location: PhysicsVector, val velocity: PhysicsVector) {

}
```

```
package physics

class PhysicsVector(var x: Double, var y: Double, var z: Double) {

}
```

```
package physics

class World(val gravity: Double) {

 var objects: List[PhysicalObject] = List()
 var boundaries: List[Boundary] = List()

}
```

Testing Objectives (45 points)

Testing Objective 1

In the Physics object write a method named computePotentialLocation that takes a PhysicalObject and a Double (delta time) as parameters and returns a PhysicsVector. The returned vector is the location of the object after delta time has passed based on its velocity if there are no boundaries in the way. This method should not allow objects to fall through the ground so if the z position would be negative it should be 0.0 instead. You are not modifying the object's location in this method since we still need to check for collisions with boundaries to see if this move is valid.

In the tests package write a test suite named TestComputeLocation that tests this method.

Testing Objective 2

In the Physics object write a method named updateVelocity that takes a PhysicalObject, a World, and a Double (delta time) as parameters and returns Unit. This method will update the object's velocity based on the acceleration due to gravity of the input world. This acceleration will be a positive value during testing (magnitude only). If the object's z location is 0.0 (on the ground) and its z velocity is negative (falling) then its z velocity should be set to 0.0 to reflect that the falling object has landed on the ground.

In the tests package write a test suite named TestUpdateVelocity that tests this method.

Testing Objective 3

In the Physics object write a method named detectCollision that takes a PhysicalObject, a PhysicsVector (potential location for the object), and a Boundary as parameters and returns a Boolean. This method is called when an object is attempting to move to a new location and returns false if that move would collide with the Boundary. Assume that boundaries cannot be crossed regardless of height (ie. ignore the z dimension for this method).

Hint: This method is determining if 2 line segments intersect in a 2d space.

In the tests package write a test suite named TestDetectCollision that tests this method.

Primary Objective (35 points)

In the Physics object white a method named updateWorld with parameters of type World then Double (delta time) and returns Unit. This method updates all the objects in the world

based on all the laws of physics in this assignment and the Double represents the amount of time the world should be advanced.

In this method, each object's velocity should be updated *before* its location is updated. As an implication of this, if an object reaches the ground it's z velocity will not be set to 0.0 until the next time this method is called.

If an object collides with a boundary during an update that object will not move at all in the x/y direction, but can still move in the z direction (ie. Objects don't get as close to the wall as they can before stopping and instead don't move at all)