Assignment 2: Walls, Grid Movement, and Blocking

Due 2/29, 2016 10:00am

1 Overview

In this assignment you will be implementing additional movement logic for the *player object*. Namely, you will be creating walls in the arena and ensuring that the player cannot move through them. Additionally, you will be restricting the movement of the player to the *game grid*, which means the player can only change direction once they're at the *center* of a cell.

2 What has changed

Take a minute before you start to compare the template with your solution to Assignment 1. Don't assume that we've solved things in the exact same way - this may lead to assumptions that will break your code in the future.

The `Entity` class has been given three additional methods: `getGridX`, `getGridY`, and `isSolid`. The first two will retrieve the *grid coordinates* of an entity (which is **distinct** but can be calculated from the *pixel* coordinates of an entity). The last method will tell you if an entity is *blocking* or not (if you can move through it). By default, entities are non-blocking.

3 What you need to do

(1) You will need to add the methods `getGridX` and `getGridY` to the `Player` class. These methods should return the x and y coordinates, respectively, of the player's *center point* on the *game grid*. Each cell in the grid is `Pacman.CELL_SIZE` pixels high and wide. Remember that the player's `x` and `y` coordinates are the player's *upper left* corner.

(2) You will be creating the class `Wall`, which is another type of *Entity*. The UML diagram for the Wall class is given below. Each wall has a `gridX` and `gridY` instance variable. This should be the *game cell* that the wall occupies. You will need to use this to calculate the *pixel coordinates* of the wall, just as you did in Assignment 1. Walls are drawn in the background, thus should have the *back* depth. Walls are also solid, which prevents players and ghosts from moving into the same cell that is occupied by a wall. Notice that there is no `update` method in this class - this is because walls are *static* entities and do not change from frame to frame (it may also be of interest that the `Entity` superclass defines this method to do nothing by default). Walls should be drawn green (`Color.GREEN`). The code will be similar to the Player's `draw` method, but the upper-left corner calculation is a bit different (you cannot simply use `gridX` and `gridY`, as they do not represent pixels).

(3) You will need to make a small change to `EntityBag`. You will need to add a method with the following signature:

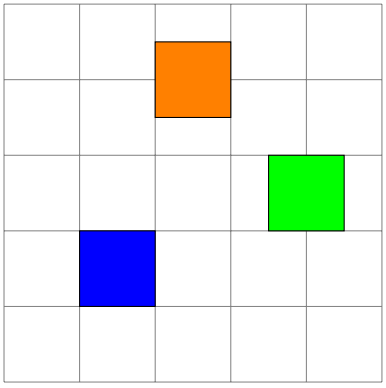
```
public boolean isBlocked(int x, int y)
```

This method should return `true` if any registered **solid** entity with a `gridX` and `gridY` equal to x and y, and `false` if no such entity exists. This should be done with a for-loop passing over each entity in the array (be cautious of the array's *effective size*), and an if-statement against each entity checking the condition above.

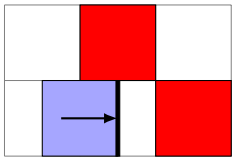
Wall
- gridX : int - gridY : int
+ Wall(x, y : int) + getGridX() : int + getGridY() : int + getDepth() : Depth + isSolid() : boolean + equals(draw : Graphics)

(4) Now the tricky part! You need to ensure that (a) the player's movement is locked to the game grid, and (b) the player's movement will resolve collisions with solid objects.

In order to do (a), we need to ensure that movement will leave *either* the player's `x` or `y` *pixel coordinate* 9is a multiple of the game grid's cell size. If the input to `update` would make this untrue, then the player would not be 'locked to a grid lane', and the player should continue to update according to its most recent direction of movement (this requires an additional instance variable, which the solution calls `facing`). To demonstrate, the blue player could move in any four directions; the green player can move only horizontally; and the orange player can move only vertically (from their current positions).



In order to do (b), you will need to determine *which cell* the user will move into if they continue in this direction. Once you know the coordinates of that cell, the `isBlocked` method of the entity bag can tell you if there is a solid object there (in which case the player must stop moving). You can get an instance of the entity bag through one of the `context` object's accessors.



How do we find the *target cell*? In the diagram below we'd like to say that blue can continue moving to the right until it kisses the bottom-most wall. In order to do that, we'll take the *center point* of the player, and add **half** of the cell width in the direction the player is moving. This will give us at the pixel value of the player's *leading edge* (in this particular case, the player's right side). You can convert this back to grid coordinates by dividing by the cell size. Notice that the player's leading edge will also need to respect wrapping around the arena.

When implementing this, you may notice that collision detection is off by a single pixel when moving left or up (west or north). You may want to fix this by *simply adding one* to the *leading edge* calculation when moving in one of these directions.

Both of these tasks will require modifications to the player's `update` method. You should first try to move according to the player input. If you are not able to for either reason, you should attempt to try again with the player's current facing. If neither directions work, then the player simply does not move.

4 Testing

We have supplied a JUnit test case that tests additional portions of the `Player` class. This test case does **not** test collisions, but does test some cases of grid-locked movement. We have also supplied a working GUI driver that adds a single player instance and some statically placed walls to the game and draws the game grid in the background. **You are responsible for testing your own code.**

5 Submission

Create a *zip archive* of your Eclipse project (including all template files) and upload it to the correct D2L dropbox before the due date. Again, no late work will be accepted.