Assignment 1: Basic Movement and Rendering

Due 2/22, 2016 10:00am

1 Overview

In this assigment you will be implementing the logic of the *player object*. The player object will encapsulate the logic of moving and rendering itself. The solution to this assignment adds about twenty-five lines of code.

2 What is supplied

A **driver**, `Pacman`, is supplied which includes the program's `main` method. You **do not** need to understand how this driver works for several weeks, but feel free to poke around. This is the class that handle's the 60fps *game loop* and listens for keyboard events. The `Entity` class is a class for game objects which *update* themselves (move, eat, die, etc) and *draw* themselves. The `EntityBag` class is (for now) a wrapper around an array of entity objects.

The `Player` class is a type of entity, which we will be implementing this week. We will be discussing *inheritance* and *polymorphism* in a following lecture, but for now it is sufficient to know that a `Player` object *is a specific kind of* `Entity`. That is, wherever we want to use an entity object (anything that can update and draw themselves), we can substitute a player.

The `util` package contains two classes which you will need to use (but not modify). The `Depth` enumeration is simply three values: *back*, *middle*, and *front*. These values represent the *draw order* of entities. Entities on a *higher depth* will appear above entities on a *lower depth* at the same location (e.g. ghosts will appear to move *over* pellets). The `Direction` enumeration will represent the *facing* of an entity. Direction instances have some methods that will be useful to you: the `getDeltaX` and `getDeltaY` methods will return the values (-1, 0, or +1) depending on which way you are moving *relative to your current position*. For example, `Direction.WEST` moves right, so `getDeltaX` should return -1, and `getDeltaY` should return 0.

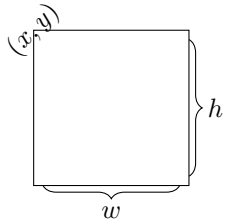
3 What you need to do

Your changes will take place **entirely** in the file `Player.java`. This class represents a player-controlled Pacman which can *move* itself (via the `update` method), and can *render* itself (via the `draw` method). The complete UML diagram for the player class is given below, as well as a detailed description of each method. Several of these methods (the constructor and the methods inherited from the *superclass* `Entity`) have been given as method headings and blank bodies. The methods unlisted in the template file must be written from scratch.

(1) The *constructor* takes a game context object as well as two integers which represent the player's initial position as *grid coordinates*. The accessors for `x` and `y` should return the current *pixel coordinates* of the player (specifically, the player's *upper-left corner*). For example, if the player starts out at grid coordinates $(3, 5)$, then `getX` should initially return 60 and `getY` should initially return 100. Because the grid size may change at some point in the future, you shouldn't hard-code the size, but use the `Pacman.CELL_SIZE` variable instead. The `getDepth` method should return the *front depth*.

Player
- context : GameContext - x : int - y : int
+ Player(context : GameContext, x, y : int) + getX() : int + getY() : int + getDepth() : Depth + update(input : Direction) + equals(draw : Graphics)

(2) The `draw` method should draw a yellow box at the player's location. You can set the *color* of the graphics object with the `setColor` method and pass it `Color.YELLOW` as an argument. You will need to import `java.awt.Color`. Setting the color of a graphics object will change the color of everything you draw in the future. You can draw a colored rectangle by calling the `fillRect` method of the graphics object and giving it the parameters representing: (x, y, *width*, *height*).



(3) The `update` method takes the user's last input as a parameter and should *update* the player's `x` and `y` coordinates. Because this method is called by the driver 60 times a second, you should only move *one pixel* each update. If the player moves off one edge of the screen, they should re-appear on the opposite end. If `input` is `null` (this will be the case before any input from the user), you should do nothing inside this method. **Hint:** The methods in the `Direction` class will be helpful here. You may benefit from making a private helper method to factor out some repeated code.

4 Testing

We have supplied a JUnit test case that tests portions of the `Player` class. We have also supplied a working GUI driver that adds a single player instance to the game and outlines the game area in red. **You are responsible for testing your own code.** Provided JUnit test cases should be treated as additional documentation / specification. Passing the provided JUnit test cases is not a guarantee that you have covered all cases (we will do additional testing by hand).

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. A solution to the homework is presented immediately after the due date in lecture, so no late work is accepted. If there is a last-minute problem uploading to D2L, an on-time submission via email to your lab instructor is accepted (please do not email submissions unless there is a problem with D2L, and use an appropriate subject line so we stay organized).