Assignment 5: Centipede, Part 2

Goals: Explore the complexity of multi-stage games containing an abundance of data.

Instructions

Where last we left our intrepid gnomish garden defender, we were building a level editor in which a player could establish the initial terrain of the level and the initial placement of the gnome. Once the player was satisfied with the level and pressed 'S', the game ended. This time, the game will enter a new phase and play can really begin. In this phase, the initial centipede will spawn at the top left of the screen and zig-zags its way from top to bottom, with some additional details. The game ends when the centipede reaches the bottom of the screen and collides with the gnome or the far wall, or the gnome defeats the centipede.

Note: you are *strongly discouraged* from using mutation on this assignment. The `funworld` library we are using does not expect mutation to occur, and you may get bizarre or not-easily-reproducible bugs as a result.

Note: you are *encouraged* to use generics if and where you think they will help. (In particular, you are welcome to use `IList<T>` instead of `ILoWhatever`, and to use function objects wherever they may help.)

Due Dates:

- Monday, February 17th, 9:00 pm

Submission format

You may find it overwhelming to keep all of your code within one file. You are permitted to split your code into multiple files, if it helps. Do *not* use any subdirectories or packages, as that will make things needlessly confusing. If you are more comfortable submitting all your code in a single file, though, that is also fine.

You will need to submit (1) all your code, and (2) a screenshot of your game after it's been playing for a little while. To submit these things together, you'll need to zip them together:

- On Windows, open Explorer, and select all the files you want to submit. Right click, and select "Send to > Compressed (Zipped) folter".
- On Mac, open Finder, and select all the files you want to submit. Right click, and select "Compress items"
- On Linux, open Nautilus, and select all the files you want to submit. Right click, and select "Compress..." Enter a filename, select the ".zip" option, and press Enter

5.1 Requirements

New partners

You are working with a new homework partner, which means you each have an implementation of part 1 of this game. You must combine your two existing assignments: choose the best parts of each, choose the most interesting examples/test cases from each, and if your designs were markedly different, explain in a comment at the top what those differences were and why you chose the version you did. The code you write on this assignment is almost entirely new (it still deals with gnomes and centipedes and garden terrain), so any discrepancies you had on part 1 should not affect building part 2 now.

New Phase, New Class

The two different phases of the game operate with two very different behaviors. One cares about clicks and one does not, one advances with time and one does not, one has an ending condition and one does not, etc.

Since these two phases are clearly related (they're part of the same game) but clearly do not share the same functionality, your program should be designed with different `World` classes: one for each phase. You must choose whether they each extend the `World` class directly, or whether there is enough common functionality that they should extend some new abstract class you define that in turn extends the `World` class.

At the top of your file, leave a comment briefly describing which design choice you made and why.

Depending on your design choices below, you might come up with additional `World` subclasses. If you do so, be sure that the purpose statements for each class explain what each kind of `World` is being used for.

The gnome

- The gnome can move left or right, but cannot "cycle around" from one side of the screen to the other.
- The gnome cannot move up or down.
- The player wins when all segments of the centipede are destroyed. The player loses the game when the centipede(s) reach the bottom row of the board where the gnome is and collides with either the gnome or the far wall (i.e. the right wall if the centipede is moving right; or left wall if it's moving left).
- The speed of the gnome is faster than the speed of the centipede(s).
- The gnome can shoot a dart at the centipede(s), and only one dart can be onscreen at any given time. If a dart has been fired, a new dart cannot be fired until the old one goes off-screen, or collides with a dandelion or a centipede segment and disappears. The speed of a dart is faster than the speed of a centipede.

- The player shoots a dart by pressing the spacebar. This will trigger a key event with `" "` as the key-information string.

The centipede(s)

- The game starts with a single centipede containing 10 segments, though you should allow this number to be easily customized via a constructor argument to your new `World` class. Each segment is one game-cell in diameter. The centipede starts with just its head visible at the top-left corner of the screen moving right. When it reaches a wall, it turns downward, moves down one row, and then turns again and keeps going in the opposite direction (i.e. if was moving left, it now is moving right, and vice versa).
- While we don't mandate a specific artistic vision for your game, there should be some clear visual indication of where the head of a centipede is and if it is moving left or right.
- When a centipede collides with a dandelion, it drops down and reverses direction just as if it hit the walls of the game. For example, if the centipede's head is in cell (5, 10) and moving right, and there's a dandelion in cell (6, 10), then the head turns down toward cell (5, 9), and then turns again towards cell (4, 9).
- If there is a second dandelion directly in the centipede's path as it drops down a row (i.e. in the example above, in cell (5, 9); also, if there's a dandelion next to the wall that the centipede just hit), that dandelion should be ignored. (In other words, you can't "trap" the centipede in a blockade of flowers.)
- > This is an 80's-style video game, not a biology lesson...

  When a segment of a centipede is hit by a dart, that segment of the centipede immediately turns into a dandelion. The centipede has now been split into two independent parts. The front of the centipede continues in its normal motion; the back of the centipede immediately collides with the new dandelion, and drops down and reverses direction as a result.
- *Extra credit:* Centipedes react to collisions with each other just like collisions with dandelions: If one centipede is about to collide with another, it drops down and reverses direction.
- The game is won when all segments of the initial centipede have been turned into dandelions.

Motion and Darts

- The movable game items (i.e. player, the centipedes, and the darts) are all intended to move *smoothly* in this game, meaning they do not move at a rate of 1 cell per tick. You will need to figure out how to translate between the idea of discrete grid cells and the actual positions of the movable game items.

  However, although a dart moves faster than a centipede, it must never "move past" a centipede or dandelion and miss them by mistake. Therefore, its speed must be less than 1

cell per tick.

- Since players and centipedes can move to positions that are not exactly centered within a cell, we have to decide where and what exactly a dart can hit. When a player fires a dart, you should determine which cell the player is closest to (i.e. round the player's position to the nearest cell center) and fire the dart vertically from that cell center. Likewise, if a centipede is currently partway between two cells, you should figure out which segment of the centipede is closest to the dart's horizontal position, and convert that segment to a dandelion. Darts can hit targets when they're vertically within the same row as each other.
- If a dart hits a dandelion, both the dart and the dandelion disappear, and that dandelion cell converts to pebbles.

Tick-tock On The Clock

> This style of programming is called a fluent interface.

As you may have noticed, a lot has to happen in one tick (centipedes have to move, collisions need to be handled, darts may be fired, etc.). To make this manageable, we recommend you design methods that use the current (`this`) world and produce a new world, where **one** of those steps have been taken, and a new world is returned. You can then chain those together to in your `onTick` method. Using this methodology, it should look something like this (with much better naming, of course):

```
// advance the world by one tick: do step one, then step two, then step three,
// and then step four
public MyGardenDefenseClass onTick() {
   return this.doStepOne()
       .doStepTwo()
       .doStepThree()
       .doStepFour();
}
```

An added benefit of designing this collection of methods is, of course, the ability to test each step of your code independently.

As a note of caution, beware that you're calling dependent operations in the right order. For example, the order in which you choose to handle centipede/player collisions and to handle centipede/dart collisions could easily affect the results of the game... Work through a wish-list, and where there are logical dependencies between steps of the game (i.e. "*this* has to happen before *that*, in case *something* happens first...") be sure to explain them in comments.

Constructor, constructor, everywhere, nor any class to create

As this game involves quite a lot of fields, and we just suggested you write a lot of methods that return a new world object, it would certainly be a good idea to have some convenience constructors lying around (see Lecture 10). Here is a pattern you may find useful to replicate:

```
abstract class AFoo {
```

```
  int bar;
  boolean baz

  AFoo(int bar, boolean baz) {
    this.bar = bar;
    this.baz = baz;
  }
}

class MyFoo extends AFoo {
  String abc;
  Color cde;
  Posn efg;

  // standard constructor
  MyFoo(int bar, boolean baz, String abc, Color cde, Posn efg) {
    super(bar, baz);
    this.abc = abc;
    this.cde = cde;
    this.efg = efg;
  }

  // copy past
  MyFoo(MyFoo past) {
    this(past.bar, past.baz, past.abc, past.cde, past.efg);
  }

  // copy past, except for abc
  MyFoo(MyFoo past, String abc) {
    this(past);
    this.abc = abc;
  }

  // return a new MyFoo with abc set to ""
  MyFoo clearAbcString() {
    return new MyFoo(this, "");
  }
}
```

The first `MyFoo` constructor is our standard constructor. The second essentially copies a `MyFoo` and produces a new one. The third constructor produces a new `MyFoo` exactly the same as the past one, except for the `abc` field, which is initalized to be whatever string is passed in.

This last constructor makes a method like `clearAbcString` very easy to write, as the entire object doesn't have to pulled apart to just update one field in the new `MyFoo`.

To IList or ILo?

It is up to you whether or not you want to use the `IList<T>` interface we have recently covered or the `ILo*` pattern from earlier in the course. The former has the advantage of being

able to write and re-use abstractions like `foldr`, `map`, `filter`, etc. easily, while the latter has the advantage of not having to write many function objects.

Yikes, This Sure Is A Lot!

Yes, it is, but don't worry, you can do it. A big part of being able to succeed with this assignment is to keep track of what data represents what, how to organize it, and what you will need to do with it. As always, sticking to the design recipe and fleshing out your wishlist in advance is the best way to go about handling daunting programs. To help you out, we've *started* an informal wishlist of tasks your program must be able to do:

- Determine when a single game piece (e.g. darts) has come into contact with any of a list of game pieces (e.g. centipede segments)
- Move a list of game pieces from their current position to their next positions
- Remove game pieces that should be destroyed for whatever reason

All of these directly imply that your program also needs to do the following:

- Determine when a single game piece has come into contact with another game piece
- Move a single game piece from its current position to its next position
- Determine when a game piece should be destroyed

To begin, we recommend fleshing out this wishlist and storing it somewhere you can easily access, and determing which classes each task should belong to. Then, you can add the method stubs to each of the classes, and write examples and tests before you begin to program. Give each method its own test function, and comment them out. As you implement each method, you can uncomment that method's test function, and that way you can test the whole program as you go.

Suggestions

While you are welcome to tweak the game to create gameplay you enjoy (so long as you adhere to all of the above specifications as well as the ones from the previous homework), the constants below will generate reasonable gameplay:

- Tile width and height: `40` pixels
- Tick rate: `1.0 / 28.0` seconds per frame. **NOTE:** you must specify a positive tick rate at the very beginning when you call `bigBang`, even if your initial world doesn't need an `onTick` handler—or else when you switch over to the worlds that do need it, the clock won't tick at all.
- Number of ticks it takes for a centipede to move across a tile: `10`
- Number of ticks it takes for the player to move across a tile: `7`
- Number of ticks it takes for a dart to move across a tile: `4`

Start early, test thoroughly, and come to office hours if you need any help. Testing this assignment is tricky, since there are so many parts. We will stipulate that you *do not need to test* "standard" list methods like map or fold or length. (That does not give you license to create obscure or game-specific list methods and expect to not have to test those!) We will also stipulate that you do not have to test your graphics thoroughly. You do need to submit a screenshot of your game, and graders will run your game to see that it looks like the screenshot.

To test your game, some hints:

- Exhaustive thoroughness of testing in this assignment is infeasible and boring. Instead, come up with a testing *plan* of *interesting* scenarios you want to test. Write this plan as a bullet-point list in a comment above your examples class. Then, as you design those test cases, update the comment with the name of the test method that does so. Even if you don't completely test every single interesting scenario, the comment documents both what you think is interesting and how far along you've gotten in testing.

- You hopefully created several convenience constructors, to make it easy to build a new world state that's *almost the same* as the current one. Since each method you design should produce an individual change or two, these constructors may be helpful in creating the expected values for your test cases.

- You hopefully structured your `onTick` handler as a chain of several smaller methods, each of which has a narrowly focused purpose statement. This should make testing much easier, since each method can be tested in isolation.

- You can call the `onKey*` and `onMouse*` methods yourself, passing in key or mouse information, and confirm that they produce new, slightly-different world states as expected.

- Finally, recall our discussions of producing *minimal* interesting test cases. For instance, what is the simplest world state that could test whether your centipede will collide with dandelions and turn correctly? Do you need a 10-segment centipede to test that, or would a 2-segment centipede suffice?