Pacman

Help Session: 2:30pm Tuesday, November 27th
Interactive Design Checks: December 2nd – 5th
Due Date: 5:00pm Wednesday, December 19th **(NB: there is no late handin!)**

To run the basic demo with all required functionality, type into a shell:
`cs015_runDemo Pacman`
To run the awesome terrific snazzy demos with spectacularly excessive bells and whistles, type:
`cs015_runSnazzyDemo Pacman`

Introduction

Congrats on choosing the best project!!! We are super excited to have you on our team, and YOU should be pumped to join the league of legendary Pacman-ers! You're going to work hard, you may have to sweat a little, maybe shed some tears, but when you're done you'll have made FREAKING PACMAN. While many of you have probably played a version of Pacman before, our requirements differ slightly from the official rules so make sure to read the specifications carefully.

Table of Contents

1. Assignment Specifications

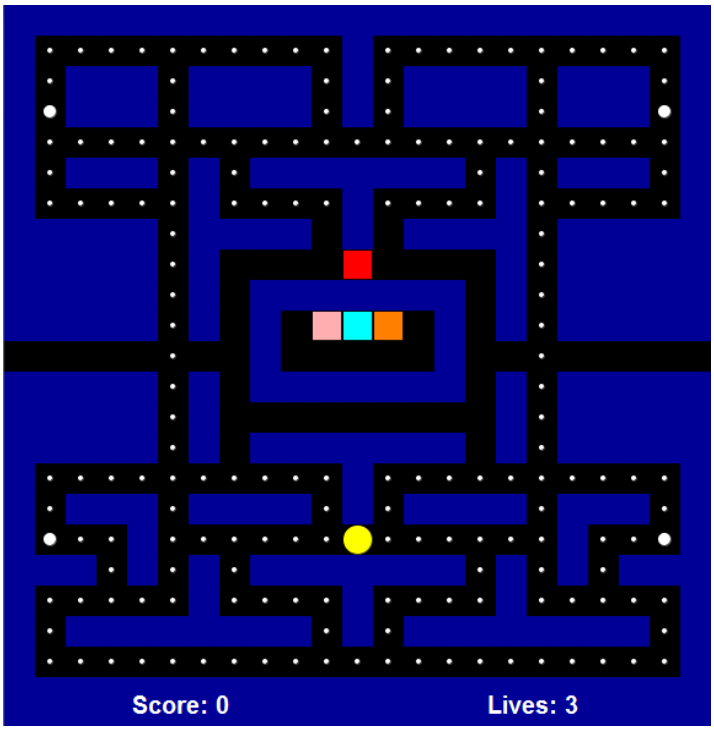
Overview of the game

Pacman is played on a 23x23 grid organized into a maze. The user controls Pacman's direction of movement with the keyboard (see *Controlling Pacman* for more details). The goal is to eat all of the white dots while avoiding the four ghosts (Blinky, Pinky, Inky, and Clyde) who chase after Pacman.

Initial Setup

The board begins with Pacman in his starting location, one ghost outside of the ghost pen, and the rest of the ghosts inside the ghost pen. There are also dots and energizers on the board. Each wall, dot, energizer, ghost, and Pacman himself resides in a single square in the 23x23 grid.

You must also keep track of Pacman's score and remaining lives. This information should be clearly visible on the screen at all times. Pacman will start with 3 lives and 0 points.



Initial state of the board.

Gameplay

If Pacman runs into a ghost, he loses a life and the ghosts and Pacman are all reset to their original starting locations. If Pacman collides with a dot or energizer, he eats the item, which disappears from the board. Eating a dot increases the player's score by 10 points. Energizers increase the player's score by 100 points and send the ghosts into frightened mode for a short time. In frightened mode, the ghosts' behavior changes (more on this in the *Ghost Behavior* section) and Pacman is able to eat them. The ghosts change their color to blue when they are in frightened mode and revert to their original colors when frightened mode ends. When a ghost is eaten, it disappears and reappears in the ghost pen. The ghost pen periodically releases ghosts to the free square just above the pen. Eating a ghost earns the player 200 points.

The ghosts' movement is determined by a breadth-first-search algorithm, which is outlined for you in the *Ghost Behavior* section.

The game ends either when all of the dots and energizers are eaten or when Pacman loses all of his lives. When the game ends, the ghosts and Pacman should all stop moving, and keyboard input should be disabled.

Graphics and Animation

We don't expect any fancy composite shapes or animations for your implementation of Pacman. Just make sure each object is clearly identifiable. The ghosts and Pacman should all move at the same speed. And just like in Tetris, we will accept step-wise animation. Run the demo to see what we mean.

There is a lot of room for creativity (and extra credit!) with the graphics. Feel free to go wild with sprites and fluid animations, **but only after you have finished all the basic requirements**. Run the snazzy demos if you need inspiration.

2. The Map

Hardcoding the entire Pacman map is hard work, and very tedious. Fortunately, we have provided you with a support class `cs015.fnl.PacmanSupport.SupportMap`. This class has a static method `getMap()` which will return a 2D `int[][]` array representing the map. Each element in this array is an `int` ranging from 0 to 5. These 6 possible values each represent different types of squares on the board.

Number	Item in Location	Wall or Free Space	Other Info
0	-	Wall	
1	-	Free	
2	dot	Free	
3	energizer	Free	
4	-	Free	Pacman's starting location
5	-	Free	Ghost's starting location (in the ghost pen)

To call this method, simply type: `cs015.fnl.PacmanSupport.SupportMap.getMap();` The map returned can be indexed by: `map[row][col]`

You must not use this array of integers as your only maze model. Rather, you should use the integer data to generate an object-oriented maze. We suggest that you create an array of maze square objects. Each maze square should know about the elements inside of it. What kind of data structure should store these elements? An array? A vector? What type would it hold? Think polymorphically!

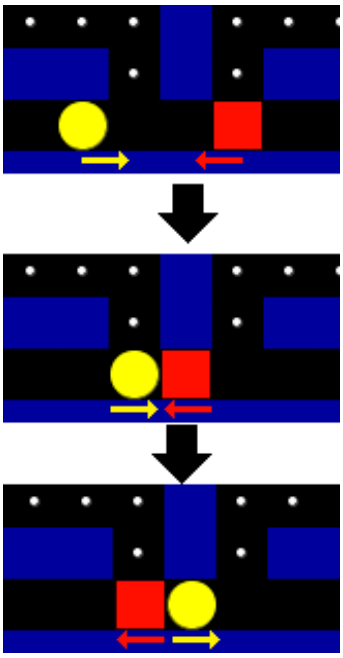
You will notice that there are two squares in the maze where Pacman and the ghosts can move out of the frame. When this happens, they should wrap around to the other side (sounds sort of like Cartoon…). See the demo for a demonstration of how it should work.

3. Collision Detection

Collisions in Pacman are generally simple: a collision occurs if Pacman's location on the grid is the same as another object's location on the grid. But the result of a collision differs greatly depending on what object Pacman collided with. There are many ways to handle collisions in this game, so try to come up with something that is generic and extensible. Here are some questions you need to consider:

- How often will you check for collisions?
- How will you handle multiple collisions? (What if Pacman collides with a dot and a ghost at the same time? A dot and two ghosts?) Hint: it shouldn't matter how many objects Pacman collides with at once. Your design should be able to handle any number of simultaneous collisions.

- What if you wanted to add fruit to the game later? Your design should make adding new objects or changing collision rules as easy as possible.



When detecting collisions, there is a common bug that occurs when Pacman and a ghost are moving towards each other. If the timing is right, it is possible for the two to switch cells at the same time. Since they were never in the same cell, the collision isn't detected and Pacman runs right through the ghost!

There is a simple solution to this bug:

1. Move Pacman
2. Check for collisions
3. Move the ghosts
4. Check for collisions

These four steps should happen one after another in this order.

Note: This is not the only solution. Feel free to come up with your own.

4. Controlling Pacman

As mentioned in the *Gameplay* section, you will use the keyboard to control Pacman's movement. (Refer back to the Tetris handout if you forget how to do this.) If a directional key is pressed, Pacman begins moving in that direction only if he is free to move in that direction. When Pacman starts moving in any direction, he continues moving in that direction until he either runs into a wall or is told to turn in a valid direction. When Pacman runs into a wall, he stops moving.

Make sure you play around with the demo for a better understanding of how controlling Pacman should work. You can use any keys you like for your directional keys, but make sure to document your choice.

5. Directions and Enums

The enumerated type (or enum), like class or interface, is a specific programming pattern used in Java, which bears similarities to both objects and primitive types. Enums are ideal for abstracting groups of constants, such as directions or ghost behavior mode. An enum is defined with a set of constants, conventionally named in all caps. Use the `enum` keyword to define an enum. For example, to define a directional enum type:

```
public enum Direction {
      UP, DOWN, LEFT, RIGHT;
}
```

Enums can also have their own instance variables and methods. For example, you will probably want to include a method in your direction enum that returns the `opposite()` of whichever direction it is called on.

A unique property of enums is that they can be used in a switch statement like ints. To quote the Java guide's example:

```
public double eval(double x, double y) {
      switch(this) {
            case PLUS:   return x + y;
            case MINUS:  return x - y;
            case TIMES:  return x * y;
            case DIVIDE: return x / y;
      }
}
```

This construct may be useful in some of your methods, including the aforementioned `opposite()`.

You cannot instantiate enums with `new`. To get a specific value, you use something like `Direction.LEFT`, or `Direction.DOWN`. For example, if you have a method that takes in a `Direction` as a parameter, then `Direction.LEFT` would be an acceptable value to pass in. All enums have a static `values()` method to return an array of all its values.

The following links are great resources to learn more about enums, with complete examples of how the syntax works. Please use them!
http://download.oracle.com/javase/tutorial/java/javaOO/enum.html
http://download.oracle.com/javase/1,5.0/docs/guide/language/enums.html

6. Ghost Pen

When the ghosts die, they are sent back to the ghost pen and released after a period of time. You should have an object modeling the ghost pen that keeps track of the ghosts currently in the pen. The ghost pen should have its own timer, and at every timer tick it should release a ghost from the ghost pen, if there are any. The ghosts should be released in the same order that they enter the pen. (Hmm...first in, first out. Sound familiar?)

7. Ghost Behavior

The ghosts navigate the board in three different modes: Chase, Scatter, and Frightened. During normal gameplay, the ghosts alternate between Chase mode (chasing after Pacman) and Scatter mode (moving toward the corners of the map). The length of each mode is up to you, but we suggest alternating between 20 seconds of Chase mode and 7 seconds of Scatter mode. These two modes utilize a breadth-first-search in order for the ghosts to reach their target cells as quickly as is possible. Both breadth-first-search and target cells will be detailed in the following sections. Frightened mode is simpler: the ghosts move randomly. The most important aspect of ghost behavior is that a ghost should **never** do a 180-degree turn in any mode. (Note: in the real Pacman, the ghosts turn around every time they switch between modes. It's okay if your ghosts do this, but we aren't requiring it.)

Target Locations

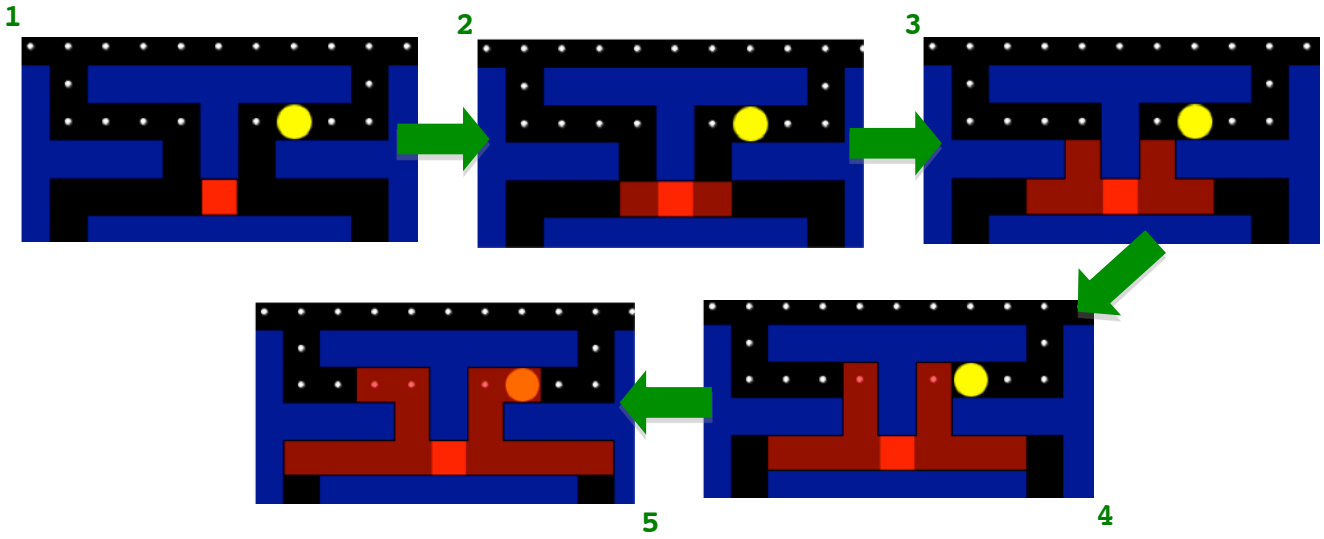
For Chase and Scatter modes, each ghost needs a target location. In Scatter mode, each ghost's target should be a different corner of the board (the targets may be inaccessible). In Chase mode, the ghosts should all have targets that are relative to Pacman's current location. We aren't requiring the complicated targeting system implemented in the original Pacman (see the *Extra Credit* section if you're interested in that), but we are requiring that each ghost has a different target. Here are some suggestions for target locations:

- Pacman's location
- 2 spaces to the right of Pacman's location
- 4 spaces above Pacman's location
- 3 spaces to the left and 1 space down from Pacman's location

Breadth First Search

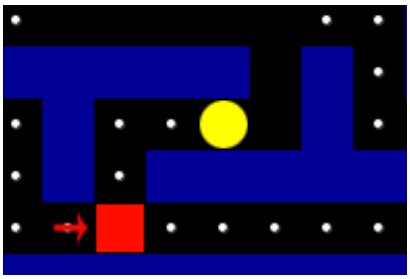
At every step, the ghost needs to know which direction to take in order to get to its target location as fast as possible. You will be implementing a modified Breadth First Search (BFS) algorithm to determine this optimal direction. You'll learn a lot more about breadth first searches and other shortest-path algorithms if you take CS16.

The general idea behind breadth first search is to first search all the squares neighboring you, then expand and search the squares neighboring your neighbors, then expand again, and so on and so forth, until the entire maze has been searched. Note that we need to somehow mark which squares have already been visited, so we don't keep looping around the maze forever.

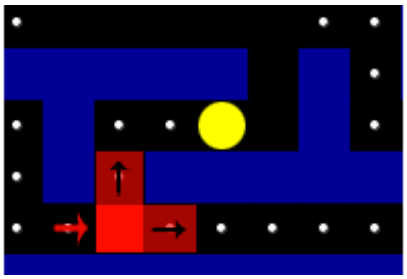


Example of BFS path starting at the ghost's location and searching for Pacman's location, examining all possible neighbors at each step.

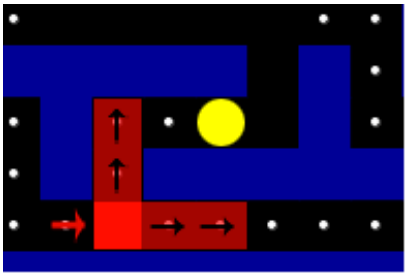
In our implementation of BFS, we will mark squares as "visited" with the **initial** direction the search took to reach that square. That way, when we finally reach our target, it will be conveniently marked with the optimal direction the ghost should take to get to that target as fast as possible!



Here is a simplified simulation of our BFS, with Pacman as our target. The ghost starts out moving right.

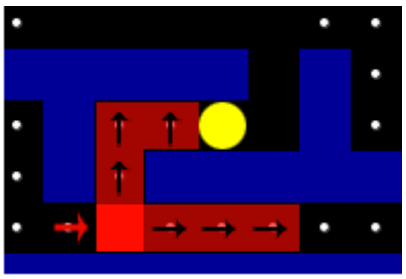


First we find all the directions in which the ghost can move. Remember that ghosts cannot make a 180 degree turn, nor can they turn into walls. In this case, the ghost can either move up or right. We launch our search in each of these valid directions, marking those squares neighboring the ghost with their respective directions.

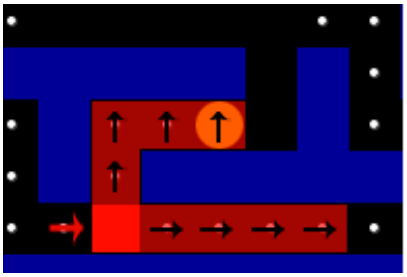


Now, in the spirit of breadth first search, we check the neighbors of the neighbors. The square that was marked as "up" checks all its unmarked neighbors (in this case there is only one) for the target, marking them all with the same direction. Likewise the square that was marked "right" checks all of its unmarked neighbors and marks them with "right" to indicate that they have been visited.

Note that each square technically has four neighbors. But we only check the ones that the ghost could actually move to. That means we don't check squares that are walls. But what about making sure the ghosts never turn 180 degrees? Fortunately, we have already accounted for this rule by never checking a square that has already been visited. Because of this, it is impossible for any trail to double back on itself. Please take time to think about this and understand how this works.



We continue expanding our search, looking for the target. Note that even though the path taken by the initial "up" direction turns to the right, all the squares along the path are still marked "up".

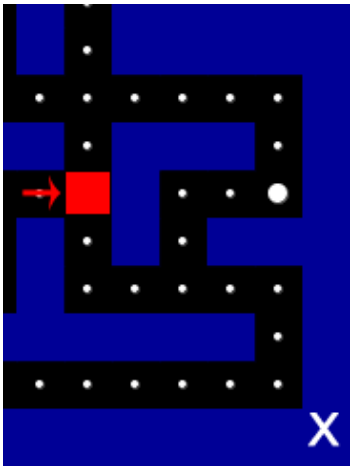


Here, our search has located the target! We've reached the square containing Pacman and marked it with "up", because that's the direction its preceding square was marked with. Now we know that moving up will take the ghost to Pacman fastest. If our target was always Pacman, we could be done with this algorithm now. However, what if the target is not reachable? With our targeting schema, it may be that the target square (e.g. two squares in front of Pacman) is actually a wall, which our current BFS would never reach. Oh no! What then?! There is a solution, but before you continue reading, make sure you have a concrete grasp of how the current BFS simulation works. You should understand that first before we introduce the next level of complexity.

Okay, ready?

Because we need to account for unreachable targets, we can no longer think of this as a breadth first search that ends when the target has been found. Rather, we are going to make a breadth first ***traversal*** of the entire map, looking for the square ***closest*** to the target. The process of visiting our neighbors first, then our neighbors' neighbors, etc. remains exactly the same. We will also use the same technique of marking squares as "visited" with directions.

But here's where it changes. Every time we visit a square, we will compute the distance between that square and the target square. Throughout the entire traversal, we will keep track of the minimum distance between a square and the target, as well as the square that holds that minimum distance. And the end of the traversal, the square holding the minimum distance to the target will be marked with the right direction to take!

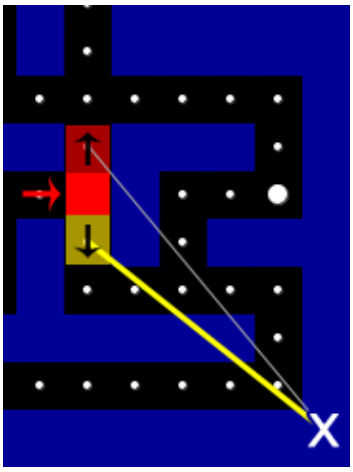


Here is a simulation of our final algorithm, with implementation specifics. Now our target is off of the board, marked by an X.

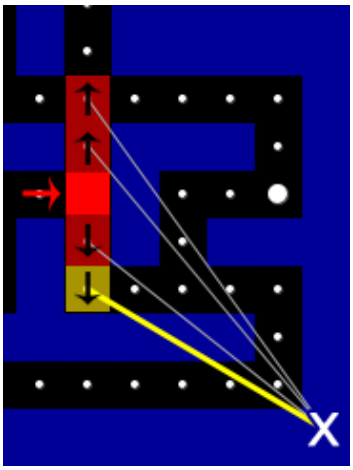
We make a new 2D array of Directions that is the same size as our maze to model the maze during our traversal. The initial value of every cell is null, because they are all unvisited.

We will refer to the squares themselves by their coordinates. `java.awt.Point` conveniently stores an x,y pair, so we can use that class.

Finally, to maintain the order of squares to visit, we will use a queue! Before we begin the traversal, we start by marking the ghost's neighboring squares as visited (in the 2D array) and enqueuing them (by coordinates). During the traversal, each time we dequeue a square to check its distance to the target, we will mark its neighbors as visited and add them to the end of the queue to be checked later. This ensures that we always visit our neighbors before we visit our neighbors' neighbors, and so on. This is the heart and soul of BFS, so make sure you understand how it works!



Now let's start. After visiting the first two squares, we update the minimum distance to be the length of the line in yellow. We also update the minimum distance square, likewise shown in yellow.

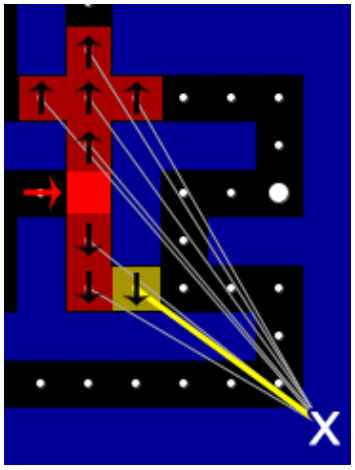


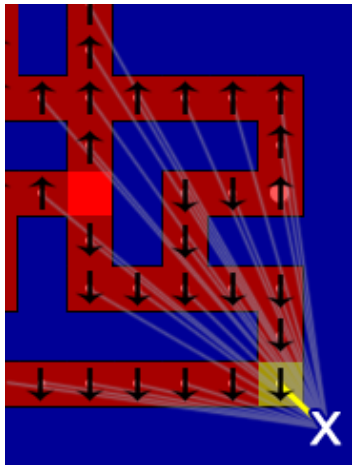
We continue the traversal as before. Remember, to get the next square, we simply dequeue from the queue. At every square, we do the following:

1. Calculate the distance to the target square
2. Update the minimum distance and minimum distance square, if necessary.
3. Find the square's unvisited neighbors, mark them with the current square's direction, and stick them at the end of the queue.

Note that all the squares in the queue have already been marked with a direction. This means that when you dequeue a square, it can tell you what direction to mark its unvisited neighbors with!

Eventually, the entire maze will have been traversed. The 2D array will be fully marked with directions (except for the wall squares), and we will have the absolute minimum distance as well as the coordinates of the minimum distance square. And with those coordinates, we can index into our 2D array to get the direction we need! In this example, we find that the ghost should move down to get to the target fastest!





High Level Pseudocode

```
for each of the ghost's valid neighboring squares:
    add the square's location (java.awt.Point) to the queue and store the square's direction in the
    2D array, indexed by its location
while the queue isn't empty:
    dequeue the next square location
    update the closest distance and the closest point, if necessary
    for each valid neighbor of the current square
        if the neighbor has not been visited, add its location to the queue and store the current
        square's direction in the array, indexed by the neighbor's location
return the direction of the closest point
```

Final note: Even though each ghost has a different target, their breadth first search algorithms are the same. You should not have to repeat the code for breadth first search in more than one place. Think about how to accomplish this.

8. Extra Credit

The following are some ideas for extra credit bells and whistles. ***Only attempt to implement these after you have a fully functional Pacman game***. It is a good idea to hand in a working version of your game before you tackle any extra credit.

- Better turning (If the user presses a directional key and Pacman is NOT able to immediately move in that direction, he will keep going and turn in that direction as soon as he can.)
- Authentic ghost target locations
- Ghosts turn 180 degrees when changing mode
- Pacman moves faster than the ghosts
- Game continues after winning (new levels)
- Fruit
- Sexy graphics

The Pacman Dossier is a complete guide to everything you need to know about Pacman. Refer to it if you are interested in implementing more authentic Pacman features (like ghost targets or fruit), or feel free to make up your own additional features! (See the snazzy demos for examples of some TAs who got a little carried away with the latter.)

http://home.comcast.net/~jpittman2/pacman/pacmandossier.html

Keep in mind that things get very difficult very quickly and time is limited! If you have any questions about how hard something would be to implement, come talk to one of the Pacman TAs. As usual, document any extra credit you implement.

Happy Coding!