UF UNIVERSITY of FLORIDA
Herbert Wertheim College of Engineering

COP3502

GatorRaider Project

Overview

This project will provide students with practice working with inheritance and polymorphism in programming languages. "GatorRaider" is a game whose play and levels are based on Ms. Pac-Man. In this game, four defender characters (gators) attempt to limit the damage by an attacker character (hunter). Students will work in a team of 2-4 people to develop the behavior for the defending characters. Students will be building a controller for the defending characters. This controller will have the following specification **which students are required to follow**:

Package: `ufl.cs1.controllers`
Class Name: `StudentDefenderController`
Implements: `DefenderController`

Getting the Code Base

This project requires students to work in an existing code base. You will need to download the project from the Git repository on GitHub, using the following URL to clone the repository:

https://github.com/uf-cise-cs1/gatorraider.git

Game Mechanics

In this game, the attacker is attempting to cover as much of the terrain as possible, while the defenders attempt to limit the terrain covered by the attacker. The play occurs in two distinct modes: Normal and Vulnerable.

Normal Mode

In Normal Mode, the attacker tries to cover terrain while avoiding the defenders. If any defender reaches the attacker, the attacker dies and loses a life, starting at the initial location once again.

Vulnerable Mode

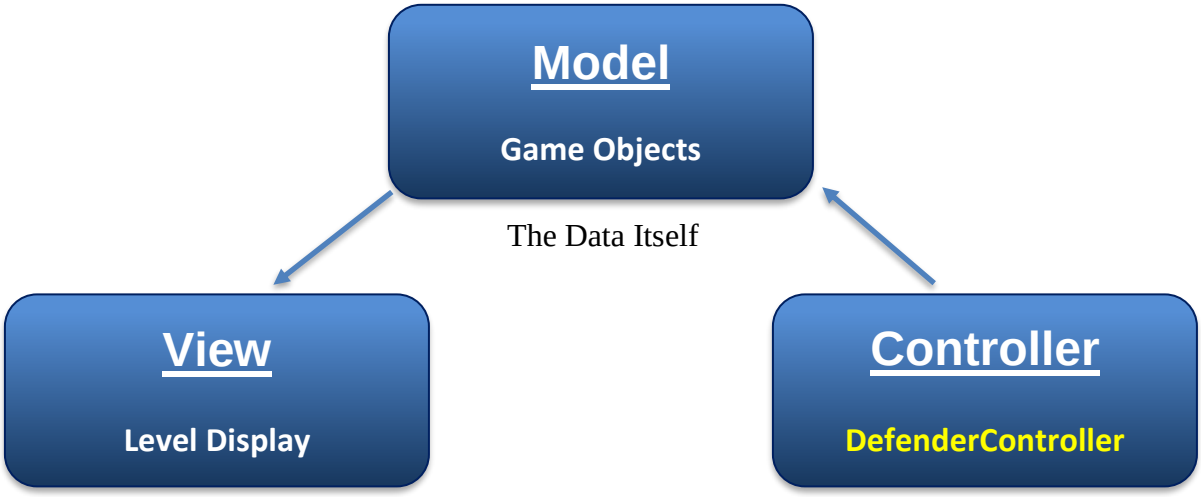
If the attacker reaches a power pill, the mode changes for a limited time to Vulnerable Mode. In Vulnerable Mode, if the attacker reaches any defender, the defender will die and its "soul" will return to the lair, where it will regenerate. Each power pill, once consumed, is gone until the level changes.

Scoring

For every new location the attacker reaches, the attacker will score points. In addition, attacker receives a bonus for reaching the power pills and for killing defenders. The goal of defenders is to limit the score of the attacker.

Project Structure

This code base makes use of the Model-View-Controller (MVC) software pattern:



In this application, the interfaces in the `game.models` package define the format and arrangement of information about the game state. The game start is displayed to the user via the `GameView` class, which draws the game level and game actors to the screen. The behavior of the actors is determined by implementations of the `AttackerController` and `DefenderController`, whose actions change the how the game plays out.

Student Work

Students will work in teams of 2-4 members to develop this project. Students will design, implement, and test the defender controller, and evaluation will include consideration of performance and documentation.

StudentController Interface

Students must implement the **`DefenderController`** interface by providing the following method:

```
public int[] update(Game game, long time)
```

Called to update the agent, given game state game after time milliseconds have passed. This method returns an array of directions, one for each defender, which will be used as the defenders' actions.

General Requirements

The StudentDefenderController submission must meet the following criteria:

1) The controller must use information from the attacker's state, board state, and maze in making decisions.
2) No defender's behavior may be random.
3) Each unique defender's behavior must be encapsulated in at least one method (multiple is also OK).
4) There must be at least as many unique defenders (with unique behaviors) as there are students in the group. If there are four students, all behaviors must be unique; for three, one may be duplicated; etc.

Debugging

There are 7 different configurations to test the project against, the 4 you should be most concerned with are:

To change the defenders used in the configurations go to **`System.game.Exec`** and instantiate your StudentDefenderController, and the pass the object into the configuration you wish to test the defenders against.

1) Example – Visual: this configuration gives a visual demonstration of the game behavior between the attacker your project will be graded against and a set of defenders.
2) Human – Visual: this configuration gives a visual demonstration of the game behavior between yourself controlling the attacker and a set of defenders.
3) Example – Debug: this configuration returns the scores of 5 games played against the attacker your project will be graded against.
4) Example – Scored: this configuration returns the scores of 100 games played against the attacker your project will be graded against. This is the configuration that will be run to get your grade for "Team Performance" (35%).

Game State

Following a summary of the most important classes and methods that are a part of the game state and can be used to determine the next action. NOTE: this is not an exhaustive list; please see source code for full information.

Game Interface

A class implementing the Game interface represents a game state.

List<Node> **getPillList**()
Get a list of all available pills in the current level.

List<Node> **getPowerPillList**()
Get a list of all available power pills in the current level.

boolean **checkPill**(Node location)
Returns true if the location in question holds an available pill.

boolean **checkPowerPill**(Node location)
Returns true if the location in question holds an available power pill.

Attacker **getAttacker**()
Returns a copy of the attacker's actor object.

Defender **getDefender**(int whichDefender)
Returns a copy of the defender's actor object specified (by number).

List<Defender> **getDefenders**()
Retrieves a list of a copy of each defender's actor object.

Game **copy**()
Return a deep copy of this game state object.

Maze **getCurMaze**()
Returns a copy of the current maze information
Random **rng** [static attribute]
Random number generator with fixed seed; this can be used for predictable behavior.

class **Direction** { public static final int **UP, RIGHT, DOWN, LEFT, EMPTY** }
Class containing constants for directions.

Node Interface

A class implementing the Node interface represents a location in the terrain. It has coordinates and neighbors.

int **getX**()

Returns the x-coordinate of this location.

int **getY**()
Returns the y-coordinate of this location.

boolean **isPill**()
Returns true if this location has or had a pill.

boolean **isPowerPill**()
Returns true if this location has or had a power pill.
boolean **isJunction**()
Returns true if this location is a fork (has 3 or more neighbors).

int **getNumNeighbors**()
Returns the number of neighbors this location has.

Node **getNeighbor**(int direction)
Returns the node neighboring this location in the direction specified.

List<Node> **getNeighbors**()
Returns the nodes neighboring this location as a List.

int **getPathDistance**(Node destination)
Returns the distance of the shortest path between this location and the destination.

Maze Interface
A Maze object, representing the terrain, has the following methods:

Node **getInitialAttackerPosition**()
Returns the starting location of the attacker.

Node **getInitialDefendersPosition**()
Returns the starting location of the defenders.

List<Node> **getPillNodes**()
Returns a list of the Node objects where pills are.

List<Node> **getPowerPillNodes**()
Returns a list of the Node objects where power pills are.

List<Node> **getJunctionNodes**()
Returns a list of the Node objects that are forks (have 3 or more neighbors).

Actor Interface
The Actor interface defines methods that are common to all characters (attackers and defenders alike).

Node **getLocation**()
Returns the current location of the actor.

int **getDirection**()
Returns the direction that the actor is currently facing.

List<Node> **getPathTo**(Node destination)

Returns a list of nodes defining a path between the current location and the destination.

`int` **`getNextDir`**`(Node target, boolean approach)`
Returns the next direction the actor should move it to approach (if `approach` is true) or flee (if false) the target.

`Node` **`getTarget`**`(List<Node> targets, boolean nearest)`
Given a list of targets, find the closest (if `nearest` is true) or furthest (if false) from the actor.

`int` **`getReverse`**`()`
Return the direction that is the reverse of where the agent is currently facing.

Attacker Interface

An Attacker object, representing the attacker's actor, provides the following methods:

`List<Integer>` **`getPossibleDirs`**`(boolean canReverse)`
Returns a list of potential directions that the agent could move in. If `canReverse` is true, this can include the direction opposite the one the agent is currently facing.

`List<Node>` **`getPossibleLocations`**`(boolean canReverse)`
Returns a list of potential locations that the agent could move to from its current location. If `canReverse` is true, this can include the location in the direction opposite the one the agent is currently facing.

Defender Interface

A Defender object, representing a defender's actor, provides the following methods:

`boolean` **`isVulnerable`**`()`
Returns true if the defender is in a vulnerable state.

`boolean` **`requiresAction`**`()`
Returns true if the actor currently requires an action to properly continue functioning (due to junction, etc.)

`int` **`getVulnerableTime`**`()`
Returns the amount of time left that the actor will be vulnerable.

`int` **`getLairTime`**`()`
Returns the amount of time remaining that the actor will spend in the lair.

`List<Integer>` **`getPossibleDirs`**`()`
Returns a list of potential directions that the agent could move in, excluding the direction opposite the one the actor is currently facing.

`List<Node>` **`getPossibleLocations`**`()`
Returns a list of potential locations that the agent could move to from its current location, excluding the location in the direction opposite the one the agent is currently facing.

Deliverables

Students will provide the following deliverables upon completion of the project:

1) StudentController.java file (as defined in the specification above)
2) Design and Post-Mortem document including identification of the following:
    a. Diagrams of defender behaviors and methods
    b. Description of individual contributions each member of the team
    c. Description of each individual defender's behavior and strategy (100-300 words each)

d. Description of overall team behavior and strategy (100-300 words)
e. Evaluation of performance of individual defenders and team (100-300 words)
f. Identifications of successes ("what went right") and failures ("what went wrong") (250-500 words)
g. Team reflection (200-300 words)
h. Individual reflection (one per student; 200-300 words)
i. Screenshots of (1) the example tested at 100 rounds; and (2) your controller tested at 100 rounds.
j. The Java version and platform your code was run on for the screenshots.

Grading

Grading of the project will be as follows:

Design & Post-Mortem Document – 30%

This section of the grade will be based on completion and quality of the design and post-mortem document.

Specification Criteria – 35%

This portion of the grade will be evaluated based on student adherence to criteria outlined elsewhere in this document. **NOTE: This does not preclude penalty deductions for faulty submissions or other issues.**

Team Performance – 35%

The remaining portion of the grade will be based on assessment of student team performance against the benchmark agent, **`Devastator`**, on average over 100 tests as compared to the example. On the test platform, **`Devastator`** scores 10,000 points, ***but yours may vary***:

Pct of Example	Grade	Example
100%	100%	10,000
150%	50%	15,000
200%	0%	20,000

Submission

Students will submit a single zip file with a "src" folder and a document in PDF (Acrobat) format.

The path within the zipfile to the source (.java) file should be "src/ufl/cs1/controllers/StudentController.java".