Ant vs. SomeBees

Programming Project

1 Introduction

In this project, you will implement a Tower Defense called *Ants Vs. SomeBees*. As the ant queen, you populate your colony with the bravest ants you can muster. Your ants must protect their queen from the evil bees that invade your territory. Irritate the bees enough by throwing leaves at them, and they will be vanquished. Fail to pester the airborne intruders adequately, and your queen will succumb to the bees' wrath.

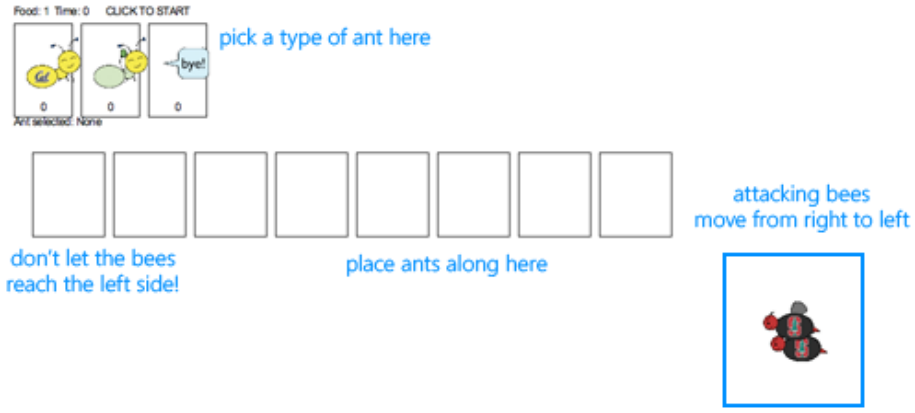
This project is inspired by an existing assignment from John DeNero, Tom Magrino and Eric Tzeng (UC Berkeley), which was inspirated by the PopCap Games' Plants Vs. Zombie®. The current version was adapted from the Python programming language to Scala. It it provides you with less pre-written code, but gives you much more freedom about how to organize your code.

1.1 Due date and grading

You are requested form teams for two people for this project, with (at most) one group of three people. Nobody is allowed to work alone. You have turn your code on Friday December 2. at noon. There will not be any public defense for this project, but you will have to send your report **and a 30-seconds long screencast** demoing your application (kazam is a nice application ot take screencasts) before Sunday December 4. night. No late submission will be accepted.

1.2 Game Concepts

Here is a screenshot of the *Ants vs. SomeBees* that we will build in this project. This game is turn-based, which means that time is split in periods of 3 seconds, during which each entity does one thing. The player controls the ants and must prevent the bees from invading the nest by placing ants in each cells of the corridor. The game ends either when a bee reaches the ant queen on the left (you lose), or the entire bee flotilla has been vanquished (you win).



During a turn, new bees may enter the ant colony and the player can place new ants (provided that the food stock permits). Finally, all insects (ants and bees) take individual actions: ants throw leaves at bees, and bees sting ants.

2 Provided Template

First download the archive file and unpack it. It contains this document (`Ants.pdf`), a code template that you should reuse (in directory `src/`), some images to use in your game (in `gfx/`) and various files to compile and edit your project (`build.sbt`, `Makefile`, `ants.geany`). The template code uses Scala Swing for the graphics, which jar file is also provided. Any possible improvement to this template is warmly welcome.

You are free to edit and compile your code in the way that you see fit. I personally used `geany` to edit the code, and `make` to compile and execute it, but YMMV. In other projects, I'm a happy user of Eclipse (the Programming Caterpillar), but wanted to experiment with geany.

As you will eventually realize, it is very difficult to share code with your teammate without the appropriate tools. You are perfectly free to use anything you see fit, such as USB key, email attachments or pigeons, but you probably want to use a SVN or Git.

Take some time to explore the provided template. As you can see, its organization is somehow similar to the simple game engine used during the RetroGames python project, a while ago.

The game is composed of a main `DemoApp` object containing the keyboard and mouse handlers and a method `onTick` that is called 50 times per second. This method is intended to contain the main game logic. In this small example, it spawns new bee sprites from time to time. The game contains a set of `Sprites`, which are game elements depicted on screen. They are also updated independently 50 times per second, and disappear when they go out of the board.

2.1 Public fields and methods of the Game object

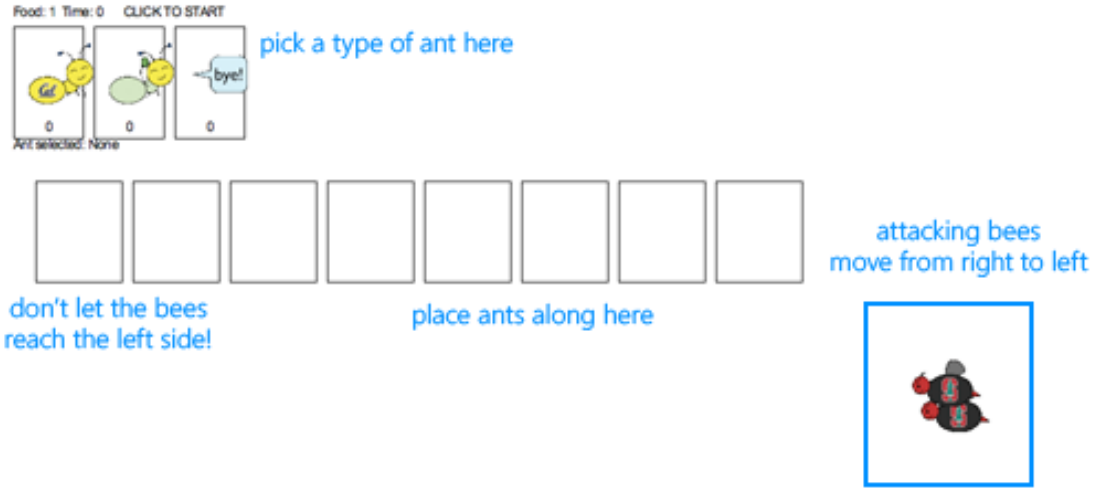
- **ticks** `:Int` Age of the application (incremented 50 times a second)
- **sprites** `:List[Sprite]` List containing all the in-game sprites, in case you want to iterate over them all.
- **addObject** `(sp:Sprite)` Method adding the provided sprite to the set.
- **delObject** `(sp:Sprite)` Method removing that sprite from the set.
- **size** `()` Returns the current size of the window.
- **setSize** `(width: Int, height: Int)` Changes the window size.
- **onTick** `()` Should contain the global game logic; Called 50 times a second.
- **onKeyPress** `(keyCode: Value)` Keyboard handler (possible values online).
- **onMouseMove** `(x:Int, y:Int)` What to do when the mouse moves.
- **repaint** `()` Trigger a redraw of the game window.

2.2 Public fields and methods of the Sprite class

- **(x,y)** `:(Int, Int)` Current position
- **(dx,dy)** `:(Int, Int)` Current speed, applied to the position on each tick.
- **age** `:Int` Current age of the sprite
- **size** `:Dimension` Defines the `size.width` and `size.height`.
- **onTick** `()` The game logic of this sprite; Called 50 times a second.
- **onClick** `()` What to do when the sprite is clicked.
- **isOob** `(maxX:Int, maxY:Int):Boolean` Returns true if the object gets out of bound. The engine removes objects that do.

- **isInside** `(px: Int, py:Int):Boolean` = Returns true if the provided coordinates are within the sprite boundaries. The engine use it to determine which sprites are clicked.

3 Game Basics



Let's have a closer look at the screenshot of the game we should do. The colony is on top. This is where you select the kind of ant that you want to place on your board. The colony cell containing the word "Bye" is particular: when you select this cell and click on one of your existing ant, it is removed from the game (food is not refunded). Under the colony, you can see one tunnel of eight places, in which you can place ants (at most one ant per place). On the right, you can see the hive, containing 2 bees that will soon enter your tunnel.

3.1 Proposed code organization

You probably want to declare a `Sprite` sub-class for the colony cells on top, and adequately override its `onClick()` method. `Place` should be another subclass of `Sprite`, representing each cell of the tunnel. You also want to declare a class `Insect`, that will be the ancestor of `Bee` and of all `Ant` classes.

This project consists in implementing several kind of ants. Our goal is not to make a nice game to play (and the original *Plants vs. Zombies* may well remain more pleasant). Instead, our objective is to experience all OOP concepts seen during the lectures.

3.2 Game turns

Each turn lasts three seconds, during which each insect entity do one move:

- **Bee** either sting any ant that blocks its path or move to the next *Place* if not blocked;
- **Harvester Ants** add one food to the colony;
- **Thrower Ants** throw a leaf at a bee located in the same place.

The player can place new ants on the places at any moment (provided that sufficient food is available). No place can contain more than one ant at the same time (and may also contain several

Bees).

4 Implementing the Game

4.1 Warm up

Harvester (Cost: 2; Armor: 1)

Thrower (Cost: 3; Armor: 1)

At first, food should not be an issue: you can create as many ants as you wish until you fill all existing Places. For that, click on the one cell of the Colony to select it, and then click on the tunnel place in which you want to add an ant of the selected type.

The game is decomposed in several turns (of three seconds each). The player can add new ants at any time during the game. At each turn, the following actions occur:

- A bee enters in the right-most place of the tunnel, unless all bees entered the game already.
- Each ant moves, depending on its type. The `HarversterAnt` adds one Food to the colony (useless for now, but soon crucial). If a bee is in the same Place, the `ThrowerAnt` removes one Armor to the bee (that has initially 3 Armors). An insect with 0 Armor is killed.
- Each bee moves.
  - If there is no ant in its place, it moves to the next place.
  - If there is an ant in its place, it reduce its Armor by one.

If all bees are defeated, the player wins. If one bee passes the left-most place, the player loses.

4.2 A playable game

Implement the food logic, where you cannot place a new ant if you don't have enough food.

Short Thrower (Cost: 3; Armor: 1)

Long Thrower (Cost: 3; Armor: 1)

Now, modify your HaversterAnt so that it shots at the nearest bee on its right. Well, it makes the game a bit too easy, so we will introduce two new kind of ants. The short-range thrower can only shot bees that are at most two entrances away while the long-range thrower can only shot bees that are at least three entrances away.

4.3 Graphical Refinements

The insects should move smoothly from one place to another at the end of each turn, and the leaves shot by throwers should be animated too. The player should still be able to place ants at any point of time (provided that there is enough food).

5 Water and Fire

Fire Ant (Cost: 5; Armor: 1)

Scuba Thrower (Cost: 5; Armor: 1)

5.1 Adding a Fire Ant

Implement the *FireAnt*. A *FireAnt* has a special *reduceArmor* method: when the *FireAnt*'s armor reaches zero or lower, it will reduce the armor of all *Bees* in the same *Place* as the FireAnt by its damage attribute (defaults to 3).

5.2 Real game

To make things more interesting, make three tunnels (one under the other). When a bee enters the board, it picks a tunnel randomly. A full game is now composed of several waves of bees, arriving at predetermined turns.

5.3 Adding water to the game

Our tunnels are still a bit boring as is, because all places are the same (but the hive). We are thus going to create a new type of Place called Water.

Only an ant that is *watersafe* can be deployed to a *Water* place. In order to determine whether an Insect is *watersafe*, add a new attribute to the Insect class named *watersafe* that is *false* by default. Since bees can fly, make their watersafe attribute true, overriding the default.

Now, implement the *addInsect* method for *Water*. First call *Place.addInsect* to add the insect, regardless of whether it is *watersafe*. Then, if the insect is not *watersafe*, reduce the insect's armor to 0 by invoking *reduceArmor*. Do not copy and paste code. Try to use methods that have already been defined and make use of inheritance to reuse the functionality of the *Place* class.

5.4 Adding a Scuba ant

Currently there are no ants that can be placed on Water. Implement the *ScubaThrower*, which is a subclass of *ThrowerAnt* that is more costly and *watersafe*, but otherwise identical to its base class.

6 More units

Wall Ant (Cost: 4; Armor: 4)

Ninja (Cost: 6; Armor: 1)

Hungry Ant (Cost: 4; Armor: 1)

6.1 Wall Ant

We are going to add some protection to our glorious *AntColony* by implementing the *WallAnt*, which is an ant that does nothing each turn (already the default action of the *Ant* class), but presents a large armor value.

6.2 Ninja Ant

Implement the *NinjaAnt*, which damages all Bees that pass by, but is never seen. It cannot be attacked by a *Bee* because it is hidden, nor does it block the path of a *Bee* that flies by. To implement this behavior, add a new attribute *blocksPath* to *Ant* that is *true* by default and *false* for *NinjaAnt*.

6.3 HungryAnt

We will now implement the new offensive unit called the *HungryAnt*, which will eat a random *Bee* from its place, instantly killing the Bee. After eating a Bee, it must spend 3 turns digesting before eating again.

7 God saves the Queen

Bodyguard (Cost: 4; Armor: 2)

Queen (Cost: 6; Armor: 2)

7.1 Bodyguard Ant

A *BodyguardAnt* differs from a normal *Ant* because it can occupy the same *Place* as another ant. When a *BodyguardAnt* is added to the same *Place* as another ant, it shields the other ant and protects it from damage. Attacks should damage the *BodyguardAnt* first and only hurt the protected ant after the *BodyguardAnt* has perished.

A *BodyguardAnt* has a field *ant* that stores the ant contained within the bodyguard. It should start off as *null*, indicating that no ant is currently being protected. Give *BodyguardAnt* a *containAnt* method that adds the Ant passed as a parameter inti that field.

Now, change your program so that a BodyguardAnt and another Ant can simultaneously occupy the same Place:

- Add an attribute to *Ant* indicating whether it is a container. Only *BodyguardAnt* are containers.
- Add a method to *ant* indicating whether the receiver can contain the parameter: If and only if the receiver is an empty container and the paramter is not a container.
- Change the *addInsect* method accordingly.

7.2 QueenAnt

The queen is a waterproof *ScubaThrower* that inspires her fellow ants through her bravery. Whenever the queen throws a leaf, she also doubles the damage of all other ants in the same tunnel with her, including any ants protected by a bodyguard. Once any ant's damage has doubled, it cannot be doubled again.

With great power comes great responsibility. The Queen is governed by three special rules:

- If a bee ever enters the place occupied by the queen, then the bees immediately win the game. The game ends even if the queen is protected by a bodyguard. The bees also win if any bee reaches the end of a tunnel where the queen normally would reside.
- There can be only one true queen. Any queen beyond the first one is an impostor and should die immediately (its armor reduced to 0) upon taking its first action, without doubling any ant's damage or throwing anything
- The true (first) queen cannot be removed. Attempts to remove the queen should have no effect (but should not cause an error).

8 Removing all mutable variables (optional)

Your program is nicely organized so far, but it is still based on variables and destructive updates. Once your project is working, rework it to use only values and non-destructive updates.

We provide you with three examples to guide you on this path. First, this short set of blog posts discuss the programming of purely functional Pac-Man. Then, here comes a functional tetris that is completely variable-free. And finally, this other presentation presents the organization of a rather large game using the functional approach with Scala. This is probably larger than what you will need for your project, but that's still very interesting.

9 Concluding remarks

In your report, explain the OOP concepts that each implemented ant puts into practice. It may well be the case that some OOP concepts are not well covered by any of the implemented ants. In this case, define a new kind of ant (or bee) that would demonstrate the missing concepts. Implementing these new insects is optional.

Your report should also contain the inheritance tree(s) of all implemented classes. PlantUML is a nice solution to include UML in a LaTeX document, but other solutions exist.

You must absolutely cleanup and comment your code before turning it in. You should strive to make your code as pleasant to read as possible.

Programs must be written for people to read, and only incidentally for machines to execute. (Harold Abelson).