CIS 201 – Computer Science I
Laboratory Assignment 11

Introduction

In this lab you will work with classes to implement a Tic-Tac-Toe game in FANG. This game will use components with “state” – that is, components that contain information about themselves in relation to the game.

Strategy

Consider how you might make a simple version of Tic-Tac-Toe: your `Game` creates and positions nine sprites on the screen. Each sprite corresponds to one of the nine squares in the game. Then, in `advance`, you would check for mouse clicks, determining if any of the sprites was clicked on. If one is clicked *and* it is empty, you would update the sprite with the current player’s symbol and change whose turn it is. Along the way, you would check for winning and draws (Cat’s game).

Think about how your `advance` would look. Assuming the sprites are in an `ArrayList`, there would be a big loop with many nested `if` statements in it to check for a winner or a draw. But this code would be too complex to hold in your mind, all at once.

Programmers respond to this sort of complexity by *abstration*: that is, hiding the details of some rule or some component by writing methods with descriptive names that express a solution to the problem, and then by filling in the details of these methods at another time. Abstraction can also be achieved by breaking responsibility for the solution to the problem across multiple components.

The “simple” approach given above is difficult because the Tic-Tac-Toe game would be responsible for handling both game-level details *and* square-level details. A *separation of responsibility* is possible if we build “smart” squares that can handle the square-level details and then let the game handle the game-level details. This is a standard technique in object-oriented design, the creation of objects that “know what to do”.

Checkpoint 1

Write down what activities a Tic-Tac-Toe game must support. Divide up your list into game-level and square-level activities. Ask yourselves the following questions:

- **What should a square know (properties) and do (methods)?**
- **What should the game know and do that a square doesn’t?**

Show us your work when you are done.

Desigining smart squares

We will assume that each *square* is responsible for:

- Advancing the state of the square:
  - Checking if the square has been clicked.
  - If the square has been clicked, checking whether it is legal to have clicked on the square (it is not legal if the square is not empty)
  - If clicking on the square is legal, updating the current state of the square (`X` or `O`) based on the current player, and reporting that an update has been made.
- Reporting on the current state of the square (blank, `X`, or `O`).

The *game* is then responsible for

- Setting up the game: creating the squares, scaling and positioning them (in `setup`).
- Advancing every square every frame, and changing the player if the square has been updated.
- Showing the game status (whose turn it is, etc.)
- Reporting on the current player (“X”, “O”).
- Ending a turn.
  - Checking if the current player has won.
  - Checking if there is a cat’s game.
  - Otherwise, change which player’s turn it is.

This lab builds two cooperating classes, `TicTacToe` and `TicTacToeSquare`, each fulfilling one of these sets of requirements.

Note *how* the two classes cooperate: A `TicTacToe` object will need references to all of the squares so that it can call `advance` for each of them; A `TicTacToeSquare` object needs a reference to the game of which it is a part to be able to check for mouse clicks and to determine whose turn it is, when it needs to be updated. This means that, like the `OneDie` class, squares will be constructed with a reference to the game object.

Create a suitable directory (`Lab11`) in which you will do your work.

Start editing the `TicTacToe` and `TicTacToeSquare` classes. The first class should extend `Game`, and the second should extend `CompositeSprite`. You will be building this game one piece at a time. First, you will get the squares to draw in the right places. Then you will make the squares aware of mouse clicks. And finally you will make the game check for end-of-game conditions.

`TicTacToeSquare` should create a rectangle sprite of a somewhat dark (non-black) color. You will add the "X" and "O" sprites later. This means writing a *constructor* for the class; there is no need for anything else right now.

You will need to decide what *parameters* the constructor requires. Since a `Game` is required for getting colors (and, eventually, checking for mouse clicks and determining whose turn it is), you should pass in the game when constructing a square. You will need a field of type `Game` in the `TicTacToeSquare` class to refer to the game, perhaps something like this:

```
    private Game theGame;
```

In the `setup` method in the `TicTacToe` class, you will call the constructor with something like this:

```
    TicTacToeSquare someSquare = new TicTacToeSquare(this);
```

The reference to the `Game` makes it possible to get colors, to get mouse clicks, and to check whose turn it is.

Your `setup` method in `TicTacToe` should construct nine `TicTacToeSquare` objects and put them in an `ArrayList`. The squares should be distributed in a 3-by-3 grid centered at the top of the game canvas. (You will need some of the other canvas real estate to show the player status and other information.) Each of the objects should be 0.20 screens square (use the `setScale` method). The centers of the squares in the rows and columns should be 0.25 apart – at *x*-coordinates 0.25, 0.50, and 0.75 for the columns, and at *y*-coordinates 0.15, 0.40, and 0.65 for the rows. Add them to the `ArrayList` so that the top left square is the first to be added, the top middle square is next, and so forth, so that the bottom right is last.

You should also need to create a field variable `turn` to keep track of which player's turn it is. Use zero (0) for "X", one (1) for "O". Finally you will need , a `StringSprite` field variable to hold a status message, scale to 0.10, position at the bottom-center of screen at 0.9 down – this is the the reason the board is so high.

Checkpoint 2

Show us when you have the nine somewhat dark boxes located properly on the screen.

Status message

Create methods to update the status message. You will need to be able to tell the user whose turn it is, who won, and that the game is a draw. Modify your `setup` program so that it displays that player 0 (“X”) is the first to play.

Square contents

A `TicTacToeSquare` object needs a field `content` to track its own contents – that is, a blank, “X”, or “O”. You should use -1, 0, and 1 to correspond to blank, “X”, and “O”, respectively. Initially, make the content equal to -1 for empty.

Implement `isEmpty` (a public, Boolean method) in your `TicTacToeSquare` class and a getter method for the content.

To display the content, add a `StringSprite` field to your `TicTacToeSquare` class. The scale should be 1.0, and it should initially contain the empty string. Set its color to some light color and make sure the sprite is added to the composite sprite. When you set the content of your square, you will set the text to either `X` or `O`.

For testing purposes, set the string to something like # for testing, and verify that your squares will all have big #s displaying on the screen when you run your game.

Checkpoint 3

Show us your work at this point. Be sure to set the string to an empty string before you continue to the next step.

Advancing each square

Create an `advance` method in `TicTacToeSquare` that will be called from `TicTacToe`’s `advance` method. In the `advance` method for a `TicTacToeSquare`, do the following:

```
if ((this square is empty) &&
    (the mouse has been clicked) &&
    (this sprite intersects the mouse click)) {
      update content with the value of the current player
      update appearance of StringSprite
      call theGame.finishTurn()
}
```

Now, in the `TicTacToe` `advance` method, call the `advance` method on every square on every frame. Also, write a stub for the method `finishTurn`. (This method should be public void with no parameters).

At this point, you should have running code that starts out with all blank squares and that turns each square to an `X` when you click on it.

Next you should implement `finishTurn` in the `TicTacToe` class: it takes no parameters and just changes whose turn it is. If turn was 0, make it 1 and if it was 1, make it 0.

Checkpoint 4

At this point your program should change empty squares alternately to `X`s and `O`s until all the squares are filled. Show us your work.

Check for a win

Modify `finishTurn` to something like the following:

```
if (winner()) {
  // handle win: display a winning message and wait
} else if (catsGame()) {
  // handle cat’s game: display an appropriate cat’s game message and wait
} else {
  // change whose turn it is as before
}
```

Next you need to write the two methods `winner` and `catsGame`. First, write stubs for them: each will have `public` visibility and will return a `boolean`. In your stubs, you should return `false`.

Checkpoint 5

With these changes, your program should still behave exactly as before. Show us your work.

Checking for a cat’s game and a winner

To implement your (currently stubbed out) `catsGame` method, you should look at all the squares and return `false` if any of them is empty. Otherwise, return `true`. Use the `isEmpty` method in the `TicTacToeSquare` class to check for being empty.

The `winner` method is a touch messier. You will need to check for all eight possible ways that a game can be won. Given the layout of the squares in the `ArrayList`, the possible combinations are rows $(0,1,2)$, $(3,4,5)$, and $(6,7,8)$. Similarly you can find the possible column and diagonal combinations.

Write a method `check` that takes three integer parameters and checks to see if the current `turn` matches the contents of the squares at the given `ArrayList` positions, using the `getContents` method. If all of them match, return `true`, otherwise return `false`.

Now your `winner` method can simply call the `check` method with the parameters $(0,1,2)$ and so forth, and should return `true` if any of these calls returns `true`, and should return `false` otherwise. If

Checkpoint 6

Show us your work at this point.

The end game

Finally, set up your program so that the game can start over. Add a boolean field variable `waiting` (a boolean variable is sometimes called a *flag*) which is set to `false` in `setup`. Then, at the beginning of `advance`, if `waiting` is `true`, check for the player pressing the space bar. If they do, call `startOver()` to restart the game. Otherwise, simply return from the `advance` method.

You should set `waiting` to be true in case either the game is won or if it's a cat's game. Make this change in your `advance` method.

Checkpoint 7

Show us your completed program.