The University of Melbourne
Department of Computer Science and Software Engineering
SWEN20003 Object Oriented Software Development

Project 2, Semester 2, 2019

Released: Friday 13th September
Project 2A due: Friday 27th of September, 11:59pm
Project 2B due: Friday 18th of October, 11:59pm
Updated Thursday 19th of September

Overview

In this project, you will create a graphical arcade game in the Java programming language, continuing from your work in Project 1. We will provide a full working solution for Project 1; you are welcome to use all or part of it, provided you add a comment explaining where you found the code at the top of each file that uses the sample code.

This is an **individual** project. You can discuss it with other students, but all submitted code must be your own work.

There are two parts to this project, with different submission dates.

The first task, **Project 2A**, requires that you produce a class design demonstrating how you plan to implement the game. This should be submitted in the form of a UML diagram showing all the classes you plan to implement, the relationships (e.g. inheritance and associations) between them, and their attributes, as well as their primary public methods. (Methods such as getters and setters need not be explicitly included.) If you so choose, you may show the relationships separately to the class members in the interests of neatness, but you must use correct UML notation. **Please submit as PDF only.**

The second task, **Project 2B**, is to complete the implementation of the game as described in the rest of this specification. You do not have to follow your class design; it is there to encourage you to think about object-oriented principles before you start programming. Indeed, it is expected that some aspects of your design will need to change when you start programming.

Shadow Bounce

Game overview

Shadow Bounce is an arcade game where the player must attempt to clear the game board of red pegs with a limited number of shots. Once the board is cleared, the player can progress to the next board, and so on until all boards are cleared or the player runs out of shots, whereupon the game should end. A *turn* begins when the board is first loaded, and ends when all balls from the turn have fallen off the bottom of the screen. When the turn ends, a new one begins.

The player begins with 20 shots. Each turn after the first, the number of shots should decrease by 1. When there are no more shots left, the game should end.

The game can be divided into three main types of objects: *pegs*, which can be destroyed to advance to the next level; *balls*, which are used to destroy pegs; and a *powerup*, which can be activated by striking it with the ball to get a bonus.

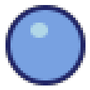
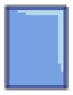
Boards are loaded from comma-separated value (`.csv`) files, numbered `0.csv` (the first board) to `4.csv` (the final board). **You will not be tested on any other boards. Boards 1 through 4 will be released at a later date**. Each line of these files is in the following format:

```
type,x,y
```

where `type` is the type of peg to be created, `x` is an integer representing the x-coordinate to create the peg at, and `y` is an integer representing the y-coordinate to create the peg at. The pegs should be created with their **centre** at these coordinates.

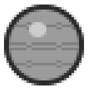
The pegs

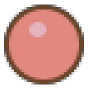
Pegs come in three shapes. The shape will be specified along with the type; e.g. `blue_horizontal_peg`, `grey_vertical_peg`. If it is not specified, the shape should be normal.

- **Normal pegs**: 
  These are the usual circular pegs from Project 1.
- **Horizontal pegs**:
  these pegs are a horizontal rectangle shape.
- **Vertical pegs**: 
  these pegs are a vertical rectangle shape.

They come in four types. Blue and grey pegs are specified in the board files; the others are created after the board is loaded.

- **Blue pegs**: these pegs have no special behaviour.

- **Grey pegs**: 
  these pegs behave as the blue pegs, but cannot be destroyed.

- **Red pegs**: 
  at the start of each board, one fifth (rounded down) of the blue pegs should become red instead. When all red pegs are destroyed, the game should advance to the next board.

- **Green pegs**:
  at the beginning of each turn, a random blue peg should become green. If the green peg is destroyed, two balls of the striking ball's type should be created at the green peg's position, with an initial velocity of 10 pixels per second diagonally upwards and to the left and right respectively. At the end of each turn, if the green peg was not destroyed, it should become blue again.

The balls

There are two types of ball the player has access to. In Project 2, balls should **bounce** off pegs they strike. If the ball strikes the peg from the top or bottom, it should reverse its vertical direction. Similarly, if the ball strikes the peg from the left or right, it should reverse its horizontal direction. The `Rectangle` class in Bagel contains a method to help you with this. As in Project 1, the circular pegs may be treated as though they were square.

- **The normal ball**:
  this ball has no special behaviour. It is created in the same way as Project 1, and moves with the same gravity. **Note:** If your screen is too small to display the window fully, you may reduce the size of the window (by calling `AbstractGame`'s constructor). The ball should then be created with an x value of half the window width, and a y value of 32. All pegs must remain visible.

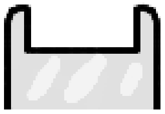
- **The fire ball**:
  the fire ball behaves as the normal ball, except when it strikes a peg, all pegs within 70 pixels of the struck `peg's centre` are destroyed. When the turn finishes, the ball returns to normal.

The powerup

At the start of each turn, with a 1/10 chance a powerup should be created at a random position on the screen. The other 9/10 of the time, no powerup should be created. The powerup should choose a random position on the screen and move towards it at a speed of 3 pixels per frame. When the powerup is within 5 pixels of its destination, it should choose another random position. If the ball strikes the powerup, the powerup is activated and destroyed.

- **Fire Ball**: when this powerup is activated, the player's ball should be replaced with the fire ball.

The bucket



The bucket begins at the coordinate (512, 744). **Note:** As with the ball, if you need to reduce the window size, you can; the bucket should begin with an x value of half the window's width, and a y value equal to the window's height minus 24.

The bucket moves left at a rate of 4 pixels per frame. When any part of the bucket reaches the left or right sides of the screen, it should reverse direction.

If a ball leaves the bottom of the screen while making contact with the bucket, the player should get an additional shot.

Implementation checklist

This project may seem daunting. As there are a lot of things you need to implement, we have provided a feature checklist, ordered roughly in the order we think you should implement them in, together with the marks each feature is worth:

1. The board is loaded and visible on screen (1 mark)
2. Grey pegs cannot be destroyed (0.5 marks)
3. Pegs are randomly chosen to be red when the board is loaded (0.5 marks)
4. The game advances to the next board when all red pegs are cleared (1 mark)
5. The game ends when all boards are cleared or the player runs out of shots (0.5 marks)
6. A peg is randomly chosen to be green pegs each turn (0.5 marks)
7. When the green peg is destroyed, the extra balls appear and move correctly (1 mark)
8. Powerup appears and moves as described (1 mark)
9. Powerup causes fire ball (0.5 marks)
10. Fire balls destroy multiple pegs (0.5 marks)
11. The bucket functions correctly (1 mark)

Customisation

Optional: we want to encourage creativity with this project. We have tried to outline every aspect of the game design here, but if you wish, you may customise any part of the game, including the graphics, types of units, buildings, resources, game mechanics, etc. You can also add entirely new features. However, to be eligible for full marks, you must implement all of the features in the above implementation checklist.

For those of you with too much time on your hands, we will hold a competition for the best game extension or modification, judged by the lecturer and tutors. The winning three will be demonstrated at the final lecture, and there will be a prize for our favourite. Past modifications have included drastically increasing the scope of the game, implementing jokes and creative game design, adding polish to the game, and even introducing networked gameplay.

If you would like to enter the competition, please email the head tutor, Eleanor McMurtry, at `mcmurtrye@unimelb.edu.au` with your username and a short description of the modifications you came up with. I can't wait to see what you've done!

If you like, you may submit a minimal version of the game to be assessed, and email a second extended version to Eleanor. Extensions submitted this way may use any libraries you like, not just Bagel and the Java standard library.

The supplied package

You will be given a package, `oosd-project2-package.zip`, which contains all of the graphics and other files you need to build the game. You can use these in any way you like.

Submission and marking

Technical requirements

- The program must be written in the Java programming language.
- The program must not depend upon any libraries other than the Java standard library, the Bagel library, and Bagel's dependencies.
- The program must compile fully without errors.
- Every **public** method, attribute, and class must have Javadoc comments, as explained in later lectures.

Submission will take place through the LMS. Please zip your project folder in its entirety, and submit this `.zip` file. **Do not submit a .rar, .7z, .tar.gz, or any other type of compressed folder.**

Ensure all your code is contained in this folder.

Extensions and late submissions

If you need an extension for the project, please email Eleanor at `mcmurtrye@unimelb.edu.au` explaining your situation with some supporting documentation (medical certificate, academic adjustment plan, wedding invitation). If an extension has been granted, you may submit via the LMS as usual; please do however email Eleanor once you have submitted your project.

The project is due at **11:59pm sharp**. Any submissions received past this time (from 12:00am onwards) will be considered late unless an extension has been granted. There will be no exceptions. There is a penalty of 1 mark per day or part thereof for a late submission. If you submit late, you **must** email Eleanor with your student ID so that we can ensure your late submission is marked correctly.

Marks

Project 2 is worth **22** marks out of the total 100 for the subject.

- Project 2A is worth 6 marks.
  - Correct UML notation for methods (1 mark)
  - Correct UML notation for attributes (1 mark)
  - Correct UML notation for associations (1 mark)
  - Good breakdown into classes (1 mark)
  - Sensible use of methods and attributes (1 mark)
  - Appropriate use of inheritance, interfaces, and `abstract` (1 mark)
- Project 2B is worth 16 marks.
  - Features implemented correctly: 8 marks (see Implementation checklist for details)
  - Coding style, documentation, and good object-oriented principles: 8 marks
    - Delegation: breaking the code down into appropriate classes (2 marks)
    - Use of methods: avoiding repeated code and overly complex methods (1 mark)
    - Cohesion: classes are complete units that contain all their data (1 mark)
    - Coupling: interactions between classes are not overly complex (1 mark)
    - General code style: visibility modifiers, magic numbers, commenting etc. (2 marks)
    - Use of documentation (javadocs) (1 mark)