The University of Melbourne
School of Computing and Information Systems
SWEN20003 Object Oriented Software Development

Project 2, Semester 2, 2018

Released: Saturday 8th of August
Project 2A Due: Saturday 22nd of September, 11:59pm
Project 2B Due: Friday 12th of October, 11:59pm

Overview

In this project, you will create a graphical arcade game in the Java programming language, continuing from your work in Project 1.[1] The graphics will be handled using the Slick graphics library.

This is an **individual** project. You can discuss it with other students, but all submitted code must be your own work. You can use any platform and tools you like to develop the game, but we recommend using the Eclipse IDE, since that is what we are supporting in class.

You will not be required to design any aspect of the game itself; this document should provide all necessary information about how the game works. You will, however, be required to design the classes for your software solution before you implement it.

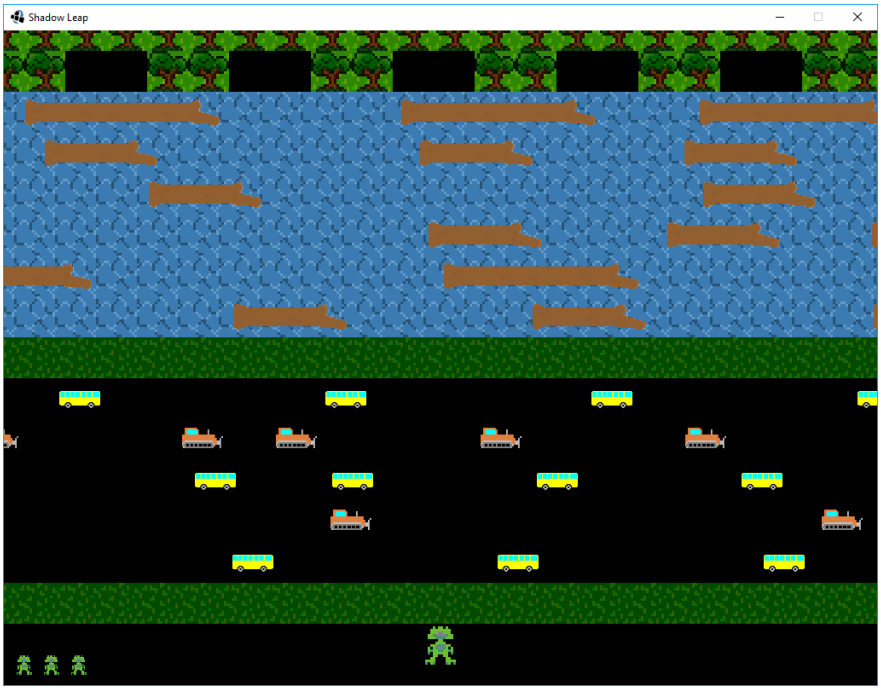
There are two parts to this project, with different submission dates.

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 in a separate document to the class members, but you **must** use correct UML notation. **Please submit in PDF format 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 only there to encourage you to think about object-oriented principles before you start programming.

[1] We will be providing a full working solution for Project 1 at a later date. You are welcome to use all or part of it, with appropriate attribution.

Shadow Leap



Game overview

Shadow Leap is an arcade game where the player must correctly time their leaps to guide the frog to the other side of the road and river, while avoiding falling into the river or getting run over by cars. The game is divided into vehicles, tiles, and rideable objects.

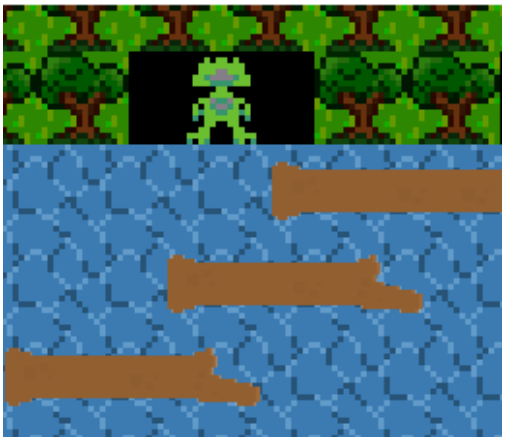
Below I will outline the different game elements you need to implement.

The player

The player behaves similarly to Project 1. A new addition is *solid* tiles; the player should not be able to move into these tiles. The player also now has *lives*. The player starts with three lives, and when the player makes contact with a hazard such as a vehicle or water, the player loses a life, reducing the lives counter by one and resetting the player's position to the initial position. If the player loses a life when there are zero remaining, the game should end.

The lives remaining should be displayed in the bottom-left using the image `lives.png`; the first icon should be drawn at the position (24,744), and each additional icon should be drawn 32 pixels to the right of the previous.

The empty tiles at the top of the screen are your goal. When the player reaches these tiles, the space should be filled with a frog sprite centred in the gap.



When this occurs, the player should reset to its initial position. When all holes are filled, the game should proceed to the next *level* as detailed below. If the player attempts to fill a hole that has already been filled, the player should instead lose a life.

The levels

The data for the objects in the game is stored in a comma-separated value format. Each line corresponds to either a *tile* or a moving *object*. A tile is described by a name, an x coordinate, and a y coordinate. An example is provided below:

```
water,0,96
```

In this case, a water tile should be created at the position (0,96). A moving object is described by a name, an x coordinate, a y coordinate, and a boolean value that indicates if the object should move to the right. An example is provided below:

```
bus,48,432,false
```

In this case, a bus object should be created at the position (48,432), and it should move towards the left.

When a level is loaded, each line of the CSV file should be interpreted in this way, and then the player should be created at the position (512,720).

The first level is named `0.lvl`, and the second level is named `1.lvl`. If the next level is reached after the second level, the game should end.

The moving objects

Moving objects move at a constant speed across the screen, either left or right. All moving objects have the property that when they are completely off screen on one side, they should reappear at the other side of the screen, so that they are just barely not visible.

The moving objects are divided into *vehicles* and *rideable objects*. There are four kinds of vehicles. Unless otherwise indicated, the player should lose a lief when they make contact with a vehicle.

- The bus: 
  The bus behaves as in Project 1. It moves at a rate of 0.15 pixels per millisecond.

- The racecar: 
  The racecar behaves the same way as the bus, except it moves at a rate of 0.5 pixels per millisecond.

- The bike: 
  The bike moves at a rate of 0.2 pixels per millisecond. When the bike reaches an x coordinate of either 24 or 1000, the bike should reverse direction.

- The bulldozer: 
  The bulldozer moves at a rate of 0.05 pixels per millisecond. The bulldozer is a *solid* object; instead of causing the player to lose a life, the bulldozer should instead push the player in its direction of travel at a a rate of 0.05 pixels per millisecond. If the bulldozer were to push the player off the screen, the player should lose a life instead.

There are three kinds of rideable objects; when the player is on top of these, they should not lose a life for making contact with the water. The player should also move along with the rideable object at the same speed as the object. When the rideable object moves off the screen, the player should remain at the edge of the screen until the rideable object disappears, at which point the player should lose a life.

- The log: 
  The log moves at a rate of 0.1 pixels per millisecond.

- The long log: 
  The long log moves at a rate of 0.07 pixels per millisecond.

- The turtles: 
  The turtles move at a rate of 0.085 pixels per millisecond. Every 7 seconds, the turtles should disappear underwater and not be displayed on screen; when they are not visible, they do not protect the player from the water. After 2 seconds underwater, they should resurface and be visible again.

The extra life

The last component of the game is the extra life object: 

At the beginning of the game, a random number of seconds between 25 and 35 should be chosen. When this time has passed, a random log or long log object should be chosen, and an extra life object should be created at the same position as the chosen log object. This process should then repeat.

The extra life object moves along with the log object it is on top of; when the log disappears from one side and reappears at the other side, the extra life should follow the log. When the player makes contact with the extra life object, it should be destroyed and the player should gain a life.

The last feature of the extra life object is that it should move around on its log. It should first attempt to move right, one tile every 2 seconds. When it can no longer move to the right, it should move to the left at the same rate, and so on. After 14 seconds the extra life object should disappear.

Implementation checklist

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

1. Vehicles are loaded and can hit the player (1)
2. All vehicles move correctly (0.5)
3. The bulldozer pushes the player (0.5)
4. The player moves along with the rideable objects (1)
5. The frog hole correctly shows when it has been reached (0.5)
6. The level completes when all holes have been reached and the next level begins (0.5)
7. The number of lives is correctly displayed (0.5)
8. The player loses a life when they make contact with a hazard (0.5)
9. The game ends when lives run out (0.5)
10. The turtles follow the correct timing of sinking and resurfacing (0.5)
11. The extra life object is generated with the correct timing (1)
12. The extra life object moves correctly and gives an extra life (1)

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 obstacles, variety of levels, 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 far too much time on your hands, we will hold a competition for the best game extension or modification, judged by the lecturer and tutor. 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!

The supplied package

You will be given a package, `oosd-project2-package.zip`, which contains all of the graphics and data you need to create the game.

Submission and marking

Submission will be through the LMS. Please submit your entire project folder, including resources and all code. Note that all public methods, attributes, and classes should have Javadoc comments, as explained in later lectures. You are not required to actually generate the Javadoc.

Good Coding Style

Good coding style is a contentious issue; however, we will be marking your code based on the following criteria:

- You should *not* go back and comment your code after the fact. You should be commenting as you go.
- You should be taking care to ensure proper use of visibility modifiers. Unless you have a very good reason for it, all instance variables should be private.
- Any constant should be defined as a static final variable. Don't use magic numbers!
- Think about whether your code makes assumptions about things that are likely to change for Project 2.

- Make sure each class makes sense as a cohesive whole. A class should contain all of the data and methods relevant to its purpose.

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**. As soon as midnight falls, a project will be considered late unless an extension has been granted. There will be no exceptions. There is a penalty of 2 marks for the first day a project is submitted late, plus 1 mark per additional day. If you submit late, you **must** email Eleanor with your student ID and number 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 approach to association (1 mark)
  - Appropriate use of inheritance (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. (1 mark)
    - Use of documentation (javadocs) (1 mark)