

THE UNIVERSITY OF MELBOURNE  
DEPARTMENT OF COMPUTER SCIENCE AND SOFTWARE ENGINEERING  
SWEN20003 OBJECT ORIENTED SOFTWARE DEVELOPMENT

## Project 1, Semester 2, 2018

Released: Saturday 25th of August, 5:00pm

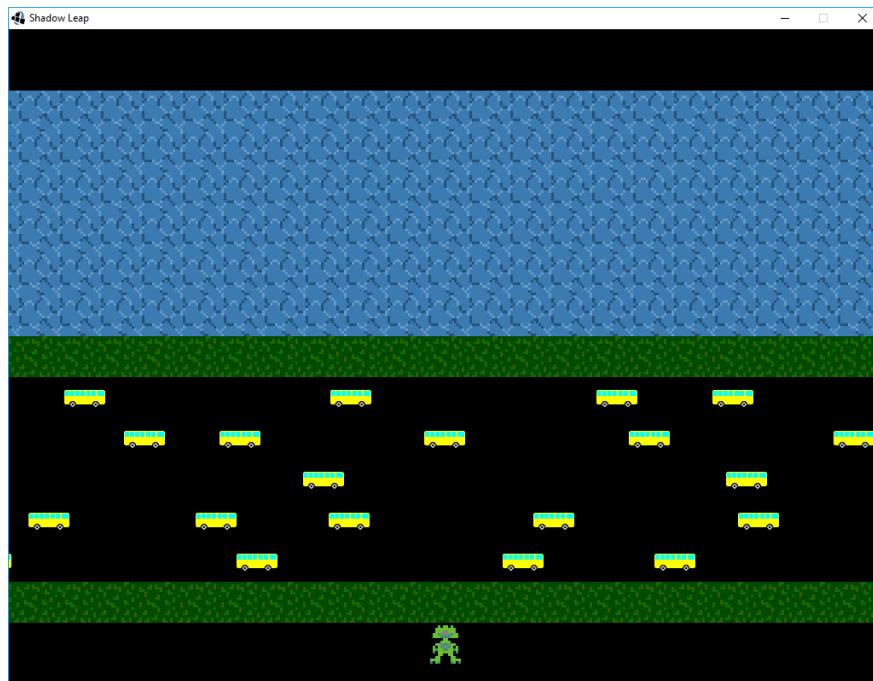
Due: Saturday 8th of September, 11:59pm

Updated 29th of August

### Overview

Welcome to the first project for SWEN20003! We will be using the Slick library for Java, which can be found at <http://slick.ninjacave.com/>. Week 4's workshop introduces Slick, so refer to the tutorial sheet if you have trouble getting started. This is an **individual** project. You may discuss it with other students, but all of the implementation must be your own work.

Project 2 will extend and build on Project 1 to create a complete game, so it's important to write your submission so that it can be easily extended. Below is a screenshot of the game after completing Project 1.



This early version of **Shadow Leap** includes a controllable frog character that can jump across the road while avoiding the cars travelling on the road. If the frog is hit by a car, or the frog jumps into the water, the game ends.

## Slick concepts

This section aims to clarify some common mistaken assumptions students make about the Slick engine, as well as to outline some important concepts.

A Slick game works according to the **game loop**. Dozens of times each second, a **frame** of the game is processed. In a frame, the following happens:

1. The game is **updated** by calling the `update()` method. The number of milliseconds since the last frame is passed as the argument `delta`; this value can be used to make sure objects move at the same speed no matter how fast the game is running.
2. The game is **rendered** by calling the `render()` method. To do this, the entire screen is cleared so it displays only black; that way, no images from the previous frame can be seen. Images can **only** be drawn inside this method.


The number of frames that are processed each second is called the **frames per second**, or FPS. Different computers will likely have a different value for FPS. Therefore, it is important to make sure your `update()` method works the same way regardless of this value.

In Slick, positions are given as pairs of  $(x, y)$  coordinates called **pixels**. Note that  $(0, 0)$  is the top-left of the window; the  $y$  direction is therefore the opposite of what you may be used to from mathematics studies. Keep in mind that while only integer locations can be rendered, it may make sense to store positions as floating-point values. For the purposes of this document, positions are given **from the centre of the sprite**.


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

## The tiles

The game is **tile-based**, meaning the game is divided into discrete squares, 48x48 pixels in size. The first thing you need to implement is some tiles to add colour and description to the game. These come in two varieties:

- **The grass tile:** 

This tile is just for decoration to indicate the beginning and end of the road. It can be found at `assets/grass.png`.

- **The water tile:** 

This tile is a hazard! For now, the frog cannot swim in the rapid river, so the game should exit when the frog makes contact with the water. It can be found at `assets/water.png`.

These tiles should be placed in a precise location; for now, this will be hard-coded into your program in the following manner:

- Water tiles should fill the screen horizontally, and be located from the range of y-coordinates 336 to 48.
- Grass tiles should fill the screen horizontally, and be located at the y-coordinates 672 and 384.

## The player

The main character of the game is the **player**. The frog representing the player is controlled with the keyboard, allowing the player to move around the screen. The relevant image can be found under `assets/frog.png`.

The player's position (and in fact, all of the sprites' positions) on-screen should be stored as an  $(x, y)$  coordinate, using a floating-point data type to avoid rounding errors. The player should start at the position (512, 720). When the left, right, up, or down arrow keys are pressed, the player should move precisely one tile in that direction – that is, 48 pixels.

The player should never be able to move off the screen.

## The obstacles

The obstacles the frog needs to avoid take the form of yellow buses, found under `assets/bus.png`. These have simple behaviour: they move either left or right at a rate of 0.15 pixels per millisecond, and once they are off-screen, they re-appear on the opposite side of the screen. When the player makes contact with these obstacles, the game should end.

Buses are created with different separations between them to make the game interesting. They are also created with different "offsets", to ensure their patterns vary. To create the buses, you should start at the offset, then create one bus per separation distance until you are no longer on the screen.

- Buses at y-location 432 should have a separation distance of 6.5 tiles, and an offset of 48 pixels.
- Buses at y-location 480 should have a separation distance of 5 tiles, and an offset of 0 pixels.
- Buses at y-location 528 should have a separation distance of 12 tiles, and an offset of 64 pixels.
- Buses at y-location 576 should have a separation distance of 5 tiles, and an offset of 128 pixels.
- Buses at y-location 624 should have a separation distance of 6.5 tiles, and an offset of 250 pixels.

The top row of buses should move left. The following row should move right, and the rows should continue alternating in this way.

## Deciding when sprites make contact

For this game, you will need to determine when two sprites are “touching” one another. In general, this is called **collision detection**, and can be an extremely difficult problem.

For our game, we will be using a simplified approach called **bounding boxes**. We will draw an invisible box around each sprite, and if two boxes intersect, we will say the sprites they belong to are **in contact** with one another. To this end, a class performing the necessary calculations has been provided at `utilities/BoundingBox.java`. In particular, it has a constructor that creates a box from an image for you. You may use this class without attribution.

**This is not an algorithms subject.** Any solution will be accepted, regardless of whether it has a poor asymptotic complexity. Think about whether the scale is large enough to make asymptotic runtime an important factor.

## Your code

Your code should consist of at least these three classes:

- **App** – The outer layer of the game. Inherits from Slick’s `BasicGame` class. Starts up the game, handles the `update` and `render` methods, and passes them along to `World`.
- **World** – Represents everything in the game, including the background and all sprites.
- **Sprite** – Represents a sprite and handles rendering its image as well as updating relevant data.

You will likely find that creating more classes will make the project easier. This decision is up to you as a software engineer! Make sure your code follows object-oriented principles like abstraction and encapsulation.

## 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:

1. Render the tiles in the correct locations
2. Render the player on-screen
3. Allow moving the player from place to place
4. Load the buses and have them move correctly
5. End the game when the player makes contact with the water and buses

## The supplied package

You will be given a package, `oosd-project1-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 want. Here is a brief summary of its contents:

- `src/` – The supplied source code.
  - `App.java` – A complete class which starts up the game and handles input and rendering.
  - `World.java` – A file with stub methods for you to fill in.
  - `Sprite.java` – A file with stub methods for you to fill in.
  - `utilities/` – A folder containing classes to assist you.
    - \* `BoundingBox.java` – A complete class containing bounding box logic.
- `assets/` – The images for the game.
  - `frog.png` – The image for the player frog.
  - `credit.txt` – The source for each of the images included.
  - `bus.png` – The image for the bus obstacle.
  - `water.png` – The image for the water tile.
  - `grass.png` – The image for the grass tile.

## 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 and the Slick library.
- The program must compile without errors.

Submission will take place through the LMS. Please zip your `src/` folder inside your Eclipse project **in its entirety**, and submit this `.zip` file. **Make sure your code works with the `assets/` folder as provided.** Ensure all your code is contained in this folder. We will provide a link on the LMS to the appropriate submission page closer to the due date.

### 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](mailto: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 1 is worth **8** marks out of the total 100 for the subject.

- Features implemented correctly – **4 marks**
  - Tiles are drawn correctly – **1 mark**
  - Player moves correctly – **1 mark**
  - Buses are initialised and move correctly – **1 mark**
  - The game ends if the player makes contact with buses or water – **1 mark**
- Code (coding style, documentation, good object-oriented principles) – **4 marks**
  - Delegation – breaking the code down into appropriate classes (**1 mark**)
  - Use of methods – avoiding repeated code and overly complex methods (**1 mark**)
  - Cohesion – classes are complete units that contain all their data (**1 mark**)
  - Code style – visibility modifiers, consistent indentation, lack of magic numbers, commenting (**1 mark**)