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

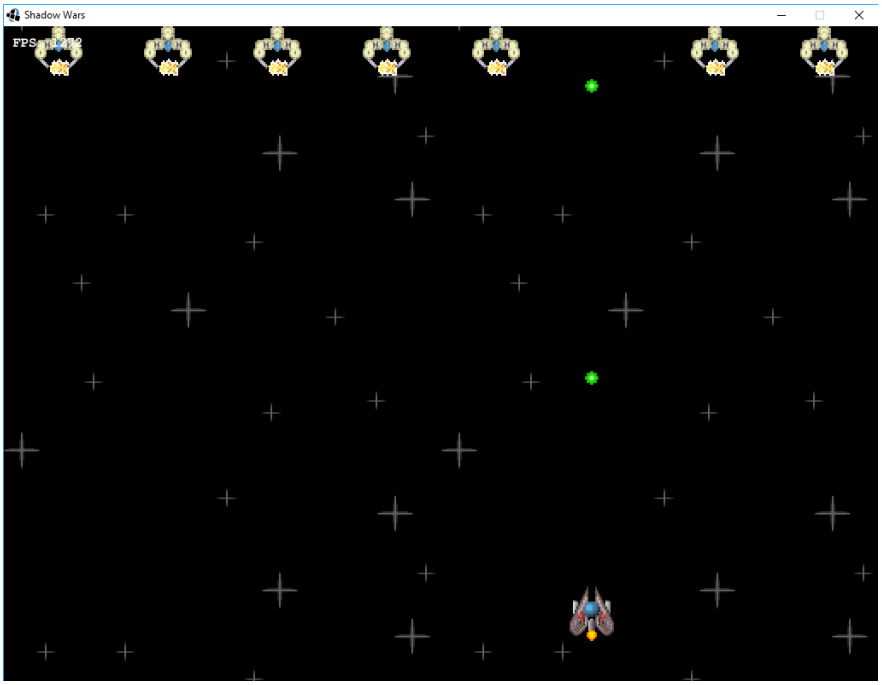
Project 1, 2018

Released: Friday 31st of March, 7:00pm
Due: Friday 13th of April, 11:59pm

Overview

Welcome to the much-anticipated 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 Wars** includes a single player spaceship that can be moved around the screen with the arrow keys. The spacebar fires a laser shot at the enemy droids, and if the shot connects, the droids are destroyed. Similarly, if the player is unfortunate enough to run into the droids, 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.

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

The background

The game takes place in space, so it's important you add some scrolling stars to communicate this to the player. The image you should use for this can be found in the file `res/space.png`. However, the image does not cover the whole screen, so your code should **tile** the image across the whole screen. Figure 1 shows each individual section of the background outlined in red.

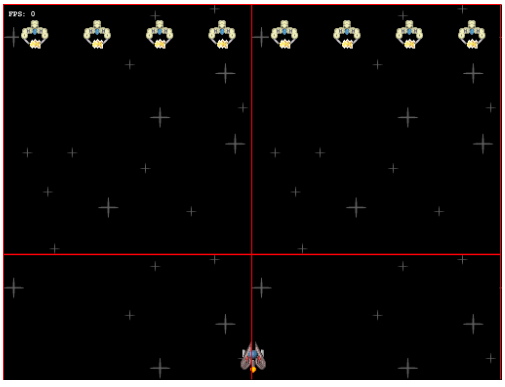


Figure 1: An example of how the tiling of the background may look.

The background image should be rendered multiple times each frame at the appropriate positions. The background should also **scroll**; each **millisecond**, the background should move 0.2 pixels downwards. Note that the image should still be tiled when it moves! An example of a moved tiling is provided in Figure 2.

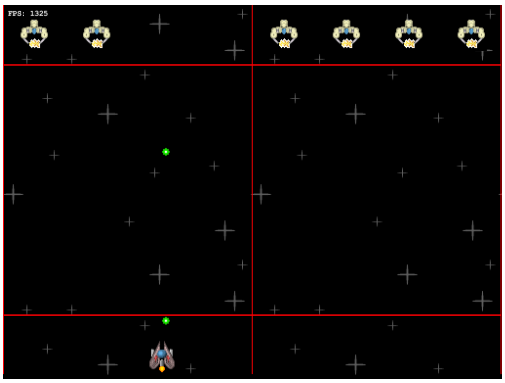


Figure 2: An example of how the tiling changes when the background scrolls.

Below are special types of objects we will call **sprites**. A sprite is an object that stores an image that should be rendered each frame. A sprite can also be **updated**, with the specific behaviour depending on the type of sprite.

The player

Our noble hero fighting against the swarm of enemies is the **player**. The ship representing the player is controlled with the keyboard, allowing the player to move around the screen. The relevant image can be found under `res/spaceship.png`.

The player's position (and in fact, all of the sprites' positons) 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 $(480, 688)$. When the left, right, up, or down arrow keys are pressed, the player should move at a rate of 0.5 pixels per millisecond in the appropriate direction.

The player should never be able to move off the screen; it should stop moving when the player tries.

Every time the spacebar is pressed, the player should create a **laser** at the centre of its image.

The laser

The laser shot is a small green laser that travels through space towards the enemies. It is represented by the image file located at `res/shot.png`. Every millisecond, the laser shot should move 3 pixels upwards on the screen. When it makes contact with an **enemy** object, both objects should be removed from the game and no longer rendered or updated.

The enemy droid

The enemy droid should do nothing other than stand in its place. It is represented by the image file located at `res/basic-enemy.png`. When it makes contact with the player, the game should exit, as the player has been destroyed. There should be eight enemies in total. The first should be located at $(64, 32)$, and each enemy should be separated by 128 pixels (measured from their centres).

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 background and tile it correctly
2. Make the background scroll while still tiling correctly
3. Create a player object that can move around
4. Create enemy objects
5. Make the enemy objects end the game when the player makes contact
6. Allow the player to fire laser shots
7. Make the laser shots destroy enemies

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.
- `res/` – The images for the game.
  - `basic-enemy.png` – The image for the enemy droid.
  - `credit.txt` – The source for each of the images included.
  - `shot.png` – The image for the player's laser shot.
  - `space.png` – The star background.
  - `spaceship.png` – The image for the player ship.

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 zipped folder. **Make sure your code works with the `res/` 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` 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**
  - Background tiles (and scrolls?) across the screen – **1 mark**
  - Player moves correctly – **1 mark**
  - Player can fire shots that move correctly – **1 mark**
  - Shots destroy enemies, and the game ends if the player makes contact with enemies – **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**)