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

ShadowPirate

Project 1, Semester 1, 2022
Released: Friday, 01/04/2022 at 8:59pm AEDT
Initial Submission Due: Thursday, 07/04/2022 at 8:59pm AEST
Project Due: Thursday, 14/04/2022 at 8:59pm AEST

Please read the complete specification before starting on the project, because there are important instructions through to the end!

Overview

Welcome to the first project for SWEN20003, Semester 1, 2022. Across Project 1 and 2, you will design and create a role-playing game called *ShadowPirate* in Java. In this project, you will create the first level of the full game that you will complete in Project 2B. This is an **individual project**. You may discuss it with other students, but all of the implementation must be your **own work**. By submitting the project you declare that you understand the University's policy on academic integrity and aware of consequences of any infringement.

You may use any platform and tools you wish to develop the game, but we recommend using IntelliJ IDEA for Java development as this is what we will support in class.

The purpose of this project is to:

- Give you experience working with an object-oriented programming language (Java),
- Introduce simple game programming concepts (2D graphics, input, simple calculations)
- Give you experience working with a simple external library (Bagel)

Game Overview

"The notorious pirate Blackbeard has stolen the Crown Jewels and is now hiding on the mysterious Tortuga Island. You are a sailor in the Royal Navy's crew that has been instructed by the King to recover the jewels. Unfortunately your crew was ambushed, everyone has been captured and held prisoner on Blackbeard's ship, The Flying Dutchman near the island. Your mission, should you be brave enough to accept it, is to escape the ship, fight the pirates on the island and recover the jewels..."

Project 1 will only feature the first level - escaping the ship. The player will be able to control the sailor who has to move around the blocks on the deck of the ship. To win, the sailor must get to the exit, located by the ladder (in the bottom right). The blocks do not move but if the sailor collides with a block, they will lose health points. If the sailor's health reduces to 0 or they

move off the sides of the ship and fall into the sea, the game ends *(a sailor who can't swim - how ironic, I know).*

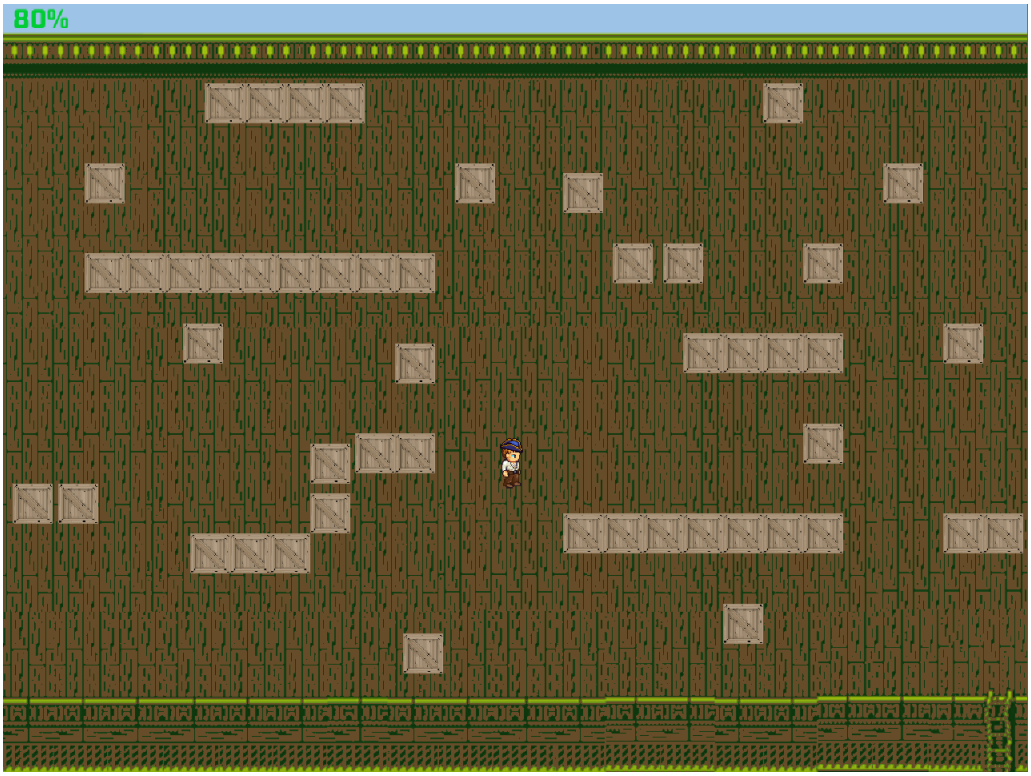


Figure 1: Completed Project 1 Screenshot

Bagel

The **B**asic **A**cademic **G**ame **E**ngine **L**ibrary (Bagel) is a game engine that you will use to develop your game. You can find the documentation for Bagel here.

Graphics Concepts Revision

Coordinates

Every coordinate on the screen is described by an (x, y) pair. $(0, 0)$ represents the top-left of the screen, and coordinates increase towards the bottom-right. Each of these coordinates is called a *pixel*. The Bagel `Point` class encapsulates this.

Frames

Bagel will refresh the program's logic at the same refresh rate as your monitor. Each time, the screen will be cleared to a blank state and all of the graphics are drawn again. Each of these steps is called a **frame**. Every time a frame is to be rendered, the `update()` method in `ShadowPirate` is called. It is in this method that you are expected to update the state of the game.

The refresh rate is typically `60` times per second (Hz) but newer devices might have a higher rate. In this case, when your game is running, it may look different to the demo videos as the constant values in this specification have been chosen for a refresh rate of 60Hz. For your convenience, when writing and testing your code, you **may** either change these values to make your game playable or lower your monitor's refresh rate to 60Hz. If you do change the values, **remember** to change them back to the original specification values before submitting, as your code will be **marked on 60Hz screens.**

Collision

It is sometimes useful to be able to tell when two images are *overlapping*. This is called **collision detection** and can get quite complex. For this game, you can assume images are rectangles. Bagel contains the `Rectangle` class to help you.

Game Elements

Below is an outline of the different game elements you will need to implement.

The Background

The background (`background0.png`) should be rendered on the screen and completely fill up your window throughout the game. The default window size should be `1024 * 768` pixels. The background has already been implemented for you in the skeleton package.

Messages

All messages should be rendered with the font provided in `res` folder, in size `55`. The bottom left of all the messages (unless otherwise specified) should be rendered at `(x, 402)` where `x` is the coordinate such that the message is centered horizontally.

Hint: The `drawString()` method in the Font class uses the given coordinates as the bottom left of the message. So to center the message horizontally, you will need to calculate the x coordinate using the `Window.getWidth() and Font.getWidth()` methods.

Start of Game

When the game is run, an instruction message that reads `PRESS SPACE TO START` should be rendered as described above, on top of the background and in the font provided. Additionally, a message that reads `USE ARROW KEYS TO FIND LADDER` should be rendered 70 pixels below the instruction message. Note that nothing else should be in the window at this time. The game will run when the space key is pressed.

World

The sailor and the blocks will be defined in a **world file**, describing the type and their position in the window. The world file is located at `res/level0.csv`. A world file is a comma-separated value (CSV) file with rows in the following format:

```
Type, x-coordinate, y-coordinate
```

An example of a world file:

```
Sailor,30,660
Block,30,500
Block,75,500
Block,207,550
```

You must actually load it—copying and pasting the data, for example, is not allowed. **Note:** You can assume that the sailor is always the first entry in the file and that this world file will always have a maximum of 50 entries.

Attacks

When there is a **collision** between the images of objects, an object may inflict damage onto the other. This is called an **attack**. This will be described further in the sections below. In this project, you will need to detect any collisions between the sailor and the blocks. Remember that you can assume the provided images are rectangles and make use of the `Rectangle` class in Bagel.

Sailor

The sailor is controlled by the arrow keys and can move in one of four directions (left, right, up, down) by 20 pixels each time an arrow key is pressed. When moving left, `sailorLeft.png` should be rendered and when moving right, `sailorRight.png` should be rendered. Initally, the sailor will start by facing right. The sailor has a **Damage Points** attribute with a value of 25. This is the maximum damage they can cause in an attack. The sailor cannot attack a block but a block can attack the sailor.

The sailor also has a **Health Points** attribute, which is an integer value that determines their current level of health. The maximum health points value for the sailor is 100. When attacked by a block, the sailor will lose health points (based on the block's damage points value) and move to its' original position before collision (it will look like the sailor is bouncing off the block). If the sailor's health points reduce to 0, the game ends.

The current health points value is displayed as a percentage on screen. This can be calculated as $\frac{CurrentHealthPoints}{MaximumHealthPoints}$. For example, if the player's current Health Points is 15 and their maximum is 20, the percentage is 75%. This percentage is rendered in the top left corner of the screen in the format of `k%` where `k` is rounded to the nearest integer. The bottom left corner of this message should be located at `(10, 25)` and the font size should be 30. Initally, the colour of this message should be green `(0, 0.8, 0.2)`. When the percentage is below 65%, the colour should be orange `(0.9, 0.6, 0)` and when it is below 35%, the colour should be red `(1, 0, 0)`.

Hint: Refer to the DrawOptions class in Bagel and how it relates to the `drawString()` method in the Font class, to understand how to use the RGB colours.

Block

The block is a stationary object, shown by `block.png`. Each block has a `Damage Points` value of 10 and can attack the sailor. This means that if the sailor and a block collide, the sailor will lose 10 health points.

Log

Every time the block inflicts damage on the sailor, a sentence detailing this is printed on the **command line**, in the following format.

```
Block inflicts 10 damage points on Sailor.  Sailor's current health:  y/z
```

where `y` is the health points value after the damage was inflicted and `z` is the maximum health points value.

For example:

```
Block inflicts 10 damage points on Sailor.  Sailor's current health:  90/100
```

Out-of-Bound Detection

If the sailor leaves the window from either the **left** border or the **right** border, it is considered out-of-bound. If the sailor moves off the **top** edge (above 60 pixels) or the **bottom** edge (below 670 pixels) of the ship deck shown in the background, it is also considered out-of-bound. Once an out-of-bound is detected, the game should end.

Win Detection

Once the sailor reaches the ladder, this is considered as a win. To reach the ladder, the sailor's `x` coordinate must be greater than or equal to 990 and the sailor's `y` coordinate must be greater than 630. A winning message that reads `CONGRATULATIONS!` should be rendered as described earlier in the *Messages* section. Note that nothing else must be displayed in the window at this time.

End of Game

If there is no win, the game will continue running until it ends. As described earlier, the game can only end if the sailor's health points reduce to 0 or there is an out-of-bound detected. A message of `GAME OVER` should be rendered as described earlier in the *Messages* section. Note that nothing else must be displayed in the window at this time.

Your Code

You must submit a class called `ShadowPirate` that contains a `main` method that runs the game as prescribed above. You may choose to create as many additional classes as you see fit, keeping in mind the principles of object oriented design discussed so far in the subject. You will be assessed based on your code running correctly, as well as the effective use of Java concepts. As always in software engineering, appropriate comments and variables/method/class names are important.

Implementation Checklist

To get you started, here is a checklist of the game features, with a suggested order for implementing them:

- Draw the game instruction messages on screen
- Draw a sailor on screen
- Draw a block on screen
- Read the world file, and draw the corresponding objects on screen
- Implement movement logic for the sailor
- Implement the sailor's health points attribute logic and render the percentage on screen
- Implement the block's attack logic (collision detection) and sailor's corresponding actions
- Implement win detection and draw winning message on screen
- Implement lose detection and draw losing message on screen
- Implement out-of-bound detection and draw losing message on screen

Supplied Package

You will be given a package called `project-1-skeleton.zip` that contains the following: (1) Skeleton code for the `ShadowPirate` class to help you get started, stored in the `src` folder. (2) All graphics and fonts that you need to build the game, stored in the `res` folder. (3). The `pom.xml` file required for Maven. Here is a more detailed description:

- `res/` – The graphics and font for the game.
  - `background0.png`: The image to represent the background.
  - `block.png`: The image to represent the block.
  - `sailorLeft.png`: The image to represent the sailor when moving left.
  - `sailorRight.png`: The image to represent the sailor when moving right.
  - `wheaton.otf`: The font to be used throughout this game.
  - `level0.csv`: The world file for the first level (there is only one world file for this Project).
  - `credit.txt`: The file giving credit to the creators of the images (you can ignore this file).
- `src/` – The skeleton code for the game.
  - `ShadowPirate.java`: The skeleton code that contains entry point to the Game, a `readCSV()` method and an `update()` method that draws the background.
- `pom.xml`: File required to set up Maven dependencies.

Submission and Marking

Initial Submission

To ensure you start the project with a correct set-up of your local and remote repository, you must complete this Initial Submission procedure on or before **Thursday, 07/04/2022 at 08:59pm**.

1. Clone the `[user-name]-project-1` folder from GitLab.
2. Download the `project-1-skeleton.zip` package from LMS, under Project 1.
3. Copy the contents of `project-1-skeleton.zip` to the `[user-name]-project-1` folder **in your local machine**.
4. Add, commit and push this change to your remote repository with the commit message `"initial submission"`.

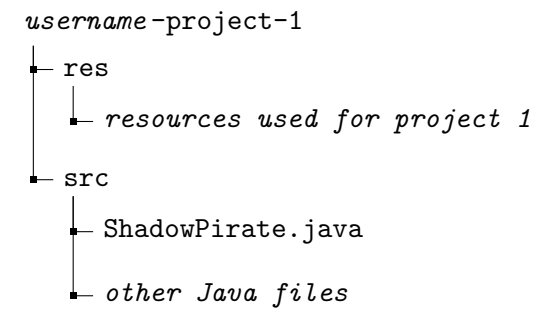
After completing this, you can start implementing the project by adding code to meet the requirements of this specification. Please remember to add, commit and push your code regularly with meaningful commit messages as you progress.

You **must** complete the Initial Submission following the above instructions by the due date. Not doing this will incur a **penalty of 3 marks** for the project. It is best to do the Initial Submission before starting your project, so you can make regular commits and push to Gitlab since the very start. However, if you start working on your project locally before completing Initial Submission, that is fine too, just make sure you move all of the contents from your project folder to `[user-name]-project-1` in your local machine.

Technical requirements

- The program must be written in the Java programming language.
- Comments and class names must be in English **only**.
- The program must not depend upon any libraries other than the Java standard library and the Bagel library (as well as Bagel's dependencies).
- The program must compile fully without errors.

Submission will take place through GitLab. You are to submit to your `<username>-project-1` repository. At the **bare minimum** you are expected to follow the structure below. You **can** create more files/directories in your repository if you want.



On 14/04/2022 at 9:00pm, your latest commit will automatically be harvested from GitLab.

Commits

You are free to push to your repository post-deadline, but only the latest commit on or before 14/04/2022 8:59pm will be marked. You **must** make at least 5 commits (excluding the Initial Submission commit) throughout the development of the project, and they must have meaningful messages (commit messages must match the code in the commit). If commits are anomalous (e.g. commit message does not match the code, commits with a large amount of code within two commits which are not far apart in time) you risk penalization.

Examples of **good, meaningful** commit messages:

- implemented movement logic
- fix sailor collision behaviour
- refactored code for cleaner design

Examples of **bad, unhelpful** commit messages:

- fesjakhbdjl
- yeah easy finished the sailor stuff
- fixed thingzZZZ

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. *(Yes, we can tell).*
- 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 final static variable (**Note:** for Image and Font objects, use only final). Don't use magic numbers!
- Think about whether your code is written to be easily extensible via appropriate use of classes.

- Make sure each class makes sense as a cohesive whole. A class should have a single well-defined purpose, and should contain all the data it needs to fulfil this purpose.

Extensions and late submissions

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

The project is due at **8:59pm sharp**. Any submissions received past this time (from 9:00pm onwards) will be considered late unless an extension has been granted. There will be no exceptions. There is a penalty of 1 mark for a late project, plus an additional 1 mark per 24 hours. If you submit late, you **must** email Andrew 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.

- **NOTE:** Not completing the Initial Submission (described in the Submission and Marking section here) before beginning your project will result in a **3 mark penalty**!
- Features implemented correctly – **6.5 marks**
  - All objects are rendered at given coordinates: (**0.5 marks**)
  - Sailor's movement behaviour is implemented correctly: (**1 mark**)
  - Block's attack behaviour is implemented correctly: (**1 mark**)
  - Sailor's health points logic and percentage is implemented correctly: (**1 mark**)
  - Win detection is implemented correctly with winning message rendered: (**1 mark**)
  - Loss detection is implemented correctly with game-over message: (**1 mark**)
  - Out-of-Bound detection is implemented correctly with game-over message: (**1 mark**)
- Code (coding style, documentation, good object-oriented principles) – **1.5 marks**
  - Delegation and Cohesion – breaking the code down into appropriate classes, each being a complete unit that contain all their data: (**0.5 mark**)
  - Use of Methods – avoiding repeated code and overly long/complex methods: (**0.5 mark**)
  - Code Style – visibility modifiers, consistent indentation, lack of magic numbers, commenting, etc. : (**0.5 mark**)