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

ShadowPac

Project 1, Semester 1, 2023
Released: Friday, 31st March 2023 at 4:30pm AEDT
Initial Submission Due: Wednesday, 12th April 2023 at 4:30pm AEST
Project Due: Friday, 21st April 2023 at 4:30pm 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, 2023. Across Project 1 and 2, you will design and create a maze game in Java called *ShadowPac* (an imitation of the original 80's arcade game *Pac-Man*). 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, including the use of **artificial intelligence**.

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)

Note: If you need an extension for the project, please complete the Extension form in the **Projects** module on Canvas. Make sure you explain your situation with some supporting documentation such as a medical certificate, academic adjustment plan, wedding invitation, etc. You will receive an email saying if the extension was approved or if we need more information.

If you submit late (**either** with or without an extension), please complete the Late form in the **Projects** module on Canvas. For both forms, you need to be logged in using your **university** account. Please **do not** email any of the teaching team regarding extensions or late submissions. All of this is explained again in more detail at the end of this specification.

You **must** make at least 5 commits (excluding the Initial Submission commit) throughout the development of the project, and they must have meaningful messages. This is also explained in more detail at the end of the specification.

Game Overview

*"The **player** controls **PacMan**, the infamous yellow main character, through an enclosed maze. To win, move around the walls, eat all the **dots** in the maze and avoid the **ghosts!**"*

The game consists of two levels - Project 1 will only feature the first level. The player will be able to control PacMan who has to move around the walls, eat the dots and avoid the ghosts. The ghosts are stationary (*for Project 1*) and if the player collides with the ghost, the player will lose a life. The player has 3 lives in total. To win, the player needs to eat (collide) with all the dots. If the player loses all 3 lives, the game ends.

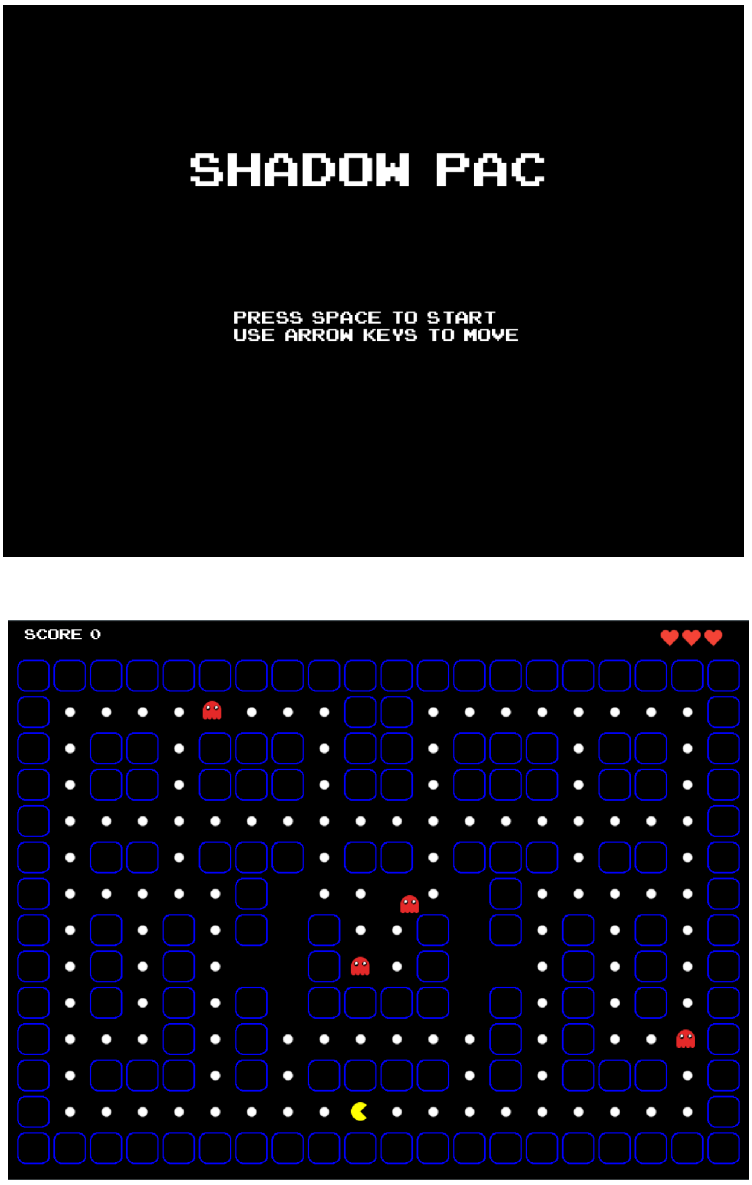


Figure 1: Completed Project 1 Screenshot

The Game Engine

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.

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 `ShadowPac` 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.**

Collisions

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 and that the player meeting a ghost or eating a dot is a collision. Bagel contains the `Rectangle` class to help you.

The Game Elements

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

Window and 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 the `res` folder (`FSO8BITR.ttf`), in size `64` (unless otherwise specified). All messages should be roughly centered both horizontally and vertically (unless otherwise specified).

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

Game Start

When the game is run, a title message that reads `SHADOW PAC` should be rendered in the font provided. The bottom left corner of this message should be located at `(260, 250)`.

Additionally, an instruction message consisting of 2 lines:

```
PRESS SPACE TO START
USE ARROW KEYS TO MOVE
```

should be rendered **below** the title message, in the font provided, in size `24`. The bottom left of the first line in the message should be calculated as follows: the x-coordinate should be increased by `60` pixels and the y-coordinate by `190` pixels.

There must be **adequate spacing** between the 2 lines to ensure readability (you can decide on the value of this spacing yourself, as long as it's not small enough that the text overlaps or too big that it doesn't fit within the screen).

World File

The player, the ghosts, the walls and the dots 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:

```
Player,474,662
Ghost,474,462
Wall,12,50
Dot,74,112
```

You must actually load it—copying and pasting the data, for example, is not allowed. **Note:** You can assume that the player is always the first entry, there are always 4 ghosts, 145 walls and 121 dots, and that this world file always has a maximum of 271 entries.

Win Conditions

Each dot is worth 10 points. Once the player eats all the dots (i.e. reaches the score of 1210), this is considered as a win. A winning message that reads `WELL DONE!` should be rendered as described earlier in the *Messages* section.

Lose Conditions

If there is no win, the game will continue running until it ends. As described earlier, the game can only end if the player's number of lives reduce to 0. A message of `GAME OVER!` should be rendered as described earlier in the *Messages* section. If the player terminates the game window at any point (by pressing the Escape key or by clicking the Exit button), the window will simply close and no message will be shown.

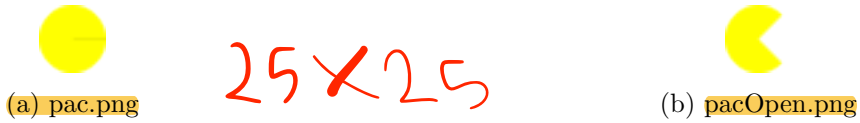
Game Entities

The following game entities have an associated image (or multiple!) and a starting location `(x, y)` on the map which are defined in the CSV file provided to you. Remember that you can assume the provided images are rectangles and make use of the `Rectangle` class in Bagel; the provided `(x, y)` coordinates for a given entity should be the **top left** of each image.

Hint: `Image` has the `drawFromTopLeft` method and `Rectangle` has the `intersects` method for you to use, refer to the Bagel documentation for more info.

The Player

In our game, the player is represented by PacMan. The player is controlled by the arrow keys and can move continuously in one of four directions (left, right, up, down) by **3 pixels per frame** whenever an arrow key is **held down**.



(a) pac.png (b) pacOpen.png

Figure 2: The player's images

The player is represented by the images shown above. Every **15 frames**, the image rendered should switch between the two (i.e. it should look like the player opening and closing its mouth). Initially, the player will start by facing right, as shown above. Based on the direction the player is moving, the image being rendered needs to be **rotated**. For example, if the player is moving down, the images need to be rotated by either 90 degrees clockwise or 270 degrees anti-clockwise, as shown below.

(a) pac.png after rotation (b) pacOpen.png after rotation

Figure 3: The player's images rotated when moving downwards

Hint: The `drawFromTopLeft` method has an overloaded method that takes a `DrawOptions` object as a parameter. `DrawOption` objects have a `setRotation` method, that allow the rotation to be set in radians.

Figure 4: Player's lives

The player has **3 lives**. If the player collides with the ghost, the player will lose a life and its position will be reset to the **starting position**, facing right. If the player loses all 3 lives, the game ends. Each life is represented by `heart.png`. The hearts are rendered in the top right corner of the screen. The first heart should be rendered with the top left coordinate at `(900, 10)`. The x-coordinate of each heart after should be increased by `30`, as shown on the left.

The player has an associated **score**. When the player collides with a dot, the player's score increases by `10` (the points value of the dot). The score is rendered in the top left corner of the screen in the format of `"SCORE k"` where `k` is the current score. The bottom left corner of this message should be located at `(25, 25)` and the font size should be `20`.

Figure 5: Player's score rendering

Ghost

A ghost is a stationary entity, shown by `ghostRed.png`. When the ghost collides with the player, the player's position resets to the **starting position** and the ghost remains as it was.

Figure 6: Ghost

25 x 25

Dot

Figure 7: Dot

A dot is a stationary object, shown by `dot.png`, with a points value of `10`. When the player collides with a dot, the player's score increases by the dot's point value.

26 x 26

Wall

A wall is a stationary object, shown by `wall.png`. The player **shouldn't** be able to overlap with or move through the walls, i.e. the player must move around any areas on the level where walls are being rendered.

Figure 8: Wall

50 x 50

Your Code

You must submit a class called `ShadowPac` 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 title and game instruction messages on screen
- Draw the player on screen
- Draw the walls on screen
- Draw the ghosts on screen
- Draw the dots on screen
- Read the world file, and draw the corresponding objects on screen
- Implement movement/image logic for the player and behaviour when colliding with a wall
- Implement the player's behaviour when colliding with a ghost and render the hearts on screen
- Implement the player's behaviour when colliding with a dot
- Implement win detection and draw winning message on screen
- Implement lose 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 `ShadowPac` 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 (You are not allowed to modify any of the files in this folder).
  - `background0.png`: The image to represent the background.
  - `wall.png`: The image to represent a wall.
  - `pac.png`: One of the images to represent the player.
  - `pacOpen.png`: The other image to represent the player.

  - `ghostRed.png`: The image to represent a ghost.
  - `dot.png`: The image to represent a dot.
  - `heart.png`: The image to represent a heart.
  - `FSO8BITR.ttf`: 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 containing credit for the font and images (you can ignore this file).
- `src/` – The skeleton code for the game.
  - `ShadowPac.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 **Wednesday, 12th April 2023 at 4:30pm**.

1. Clone the `[user-name]-project-1` folder from GitLab.
2. Download the `project-1-skeleton.zip` package from LMS, under Project 1.
3. Unzip it.
4. Move the **contents** of the unzipped folder to the `[user-name]-project-1` folder **in your local machine**.
5. Add, commit and push this change to your remote repository with the commit message `"initial submission"`.
6. Check that your push to Gitlab was successful and to the correct place.

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.

```
username-project-1
├─ res
│  └─ resources used for project 1
└─ src
   ├─ ShadowPac.java
   └─ other Java files
```

On 21st April 2023 at 4:30pm, 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 21st April 2023 4:30pm 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 the player's collision behaviour
- refactored code for cleaner design

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

- fesjakhbdjl
- yeah easy finished the player
- 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 (apart from constants).
- Any constant should be defined as a final static variable (**Note:** for Image and Font objects, use only final). Constants can be public depending on usage. 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 complete the Extension form in the **Projects** module on Canvas. Make sure you explain your situation with some supporting documentation such as a medical certificate, academic adjustment plan, wedding invitation, etc. You will receive an email saying if the extension was approved or if we need more information.

The project is due at **4:30pm sharp**. Any submissions received past this time (from 4:30pm 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** (*either* with or without an extension), please complete the Late form in the **Projects** module on Canvas. For both forms, you need to be logged in using your **university** account. Please **do not** email any of the teaching team regarding extensions or late submissions.

Marks

Project 1 is worth **8** marks out of the total 100 for the subject. Although you may see how inheritance can be used for future extensibility, you are **not required** to use inheritance in this project.

- **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 – **5.5 marks**
  - Starting screen is implemented correctly: (**0.25 marks**)
  - The player's images and movement behaviour is implemented correctly: (**1 mark**)
  - The player's score and lives logic is implemented correctly: (**0.5 marks**)
  - The ghosts and their images are implemented correctly: (**0.5 marks**)

  - Player and ghost collisions (including corresponding actions) implemented correctly: (**1 mark**)
  - Wall and its' collisions with the player is implemented correctly: (**1 mark**)
  - Dot and its' collision with the player is implemented correctly: (**0.75 marks**)
  - Win detection is implemented correctly with winning message rendered: (**0.25 marks**)
  - Loss detection is implemented correctly with game-over message: (**0.25 marks**)
- Code (coding style, documentation, good object-oriented principles) – **2.5 marks**
  - Delegation and Cohesion – breaking the code down into appropriate classes, each being a complete unit that contain all their data: (**1 mark**)
  - Use of Methods – avoiding repeated code and overly long/complex methods: (**0.5 marks**)
  - Code Style – visibility modifiers, consistent indentation, lack of magic numbers, commenting, etc. : (**1 mark**)