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

Project 1, Semester 1, 2021

Released: 2/4/2021
Due: 23/4/2021 at 20:59 AEST

Overview

Welcome to the first project for SWEN20003! In this project, you will design and simulate **ShadowTreasure**, a treasure hunt game, continuously in Projects 1 and 2. In this project (Project 1), you will create the graphical interface and implement the interactions between the characters in the game. These will form the basis for the game development in Project 2.

This is an **individual project**. You may discuss it with other students, but all of the implementation must be your own work. You may use any platform and tools you wish to develop the project, but we officially support IntelliJ IDEA for Java development.

The purpose of this project is to:

- give you experience working with an object-oriented programming language (Java)
- introduce fundamental game programming concepts (e.g., 2D graphics, geometric calculations)
- give you experience writing software using an external library (Bagel)

Bagel Concepts

The **B**asic **A**cademic **G**raphical **E**ngine **L**ibrary (Bagel) is a game engine that you will use to develop your simulation. You can find the documentation for Bagel here—please check this documentation first if you need clarification on how to use the library.

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, and additionally allows floating-point positions to be represented. **Note that, in the rest of the context, we omit the entity "pixel" of any metric amount.** For example, "the step size is 10" means the step size is 10 pixel.[1]

Many times per second, the program's logic is updated, the screen is cleared to a blank state, and all of the graphics are **rendered** again. Each of these steps is called a **frame**. Every time a frame is to be rendered, the `update` method of `AbstractGame` is called. It is in this method that you are expected to update the state of the simulations.

[1] This means, when you write java code, you don't need to worry about the entity, just set the value to 10, e.g., `STEP_SIZE = 10`.

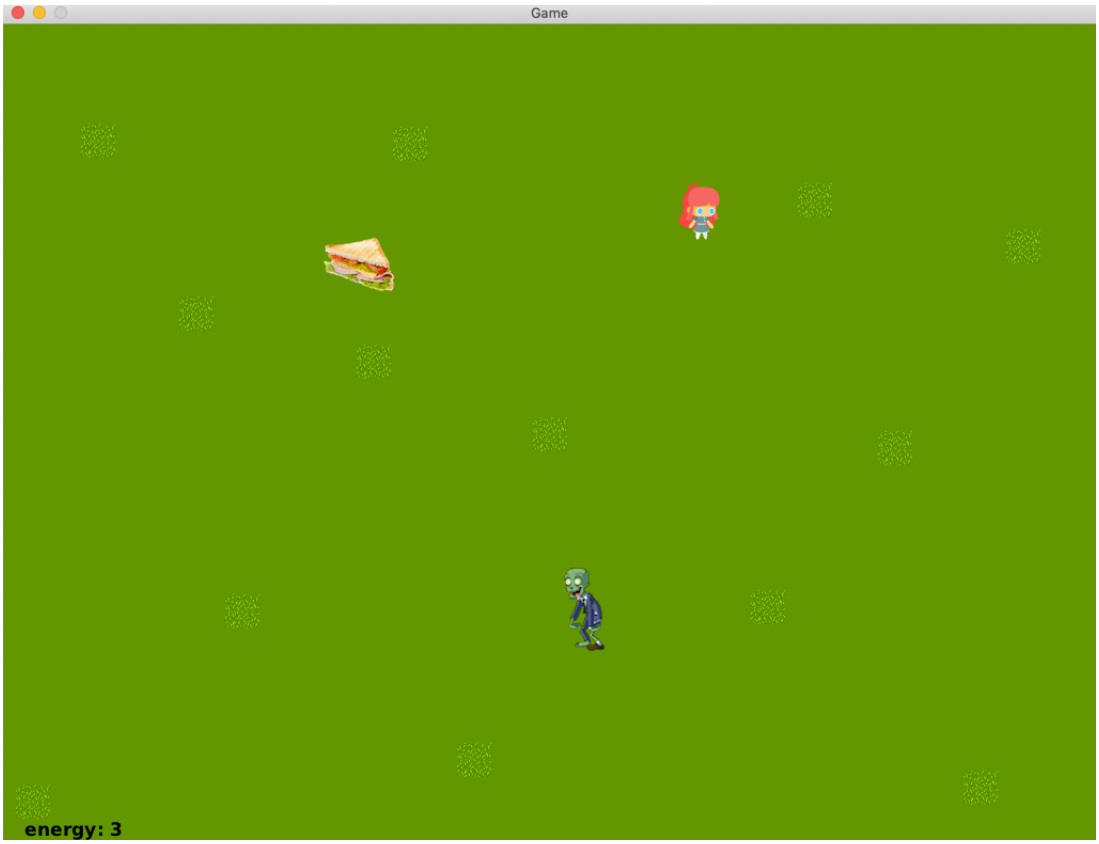

Figure 1: An Example of Project 1 Screenshot (note that the position of player, sandwich and zombie might be different in your implementation).

The simulation state must change only every 10 frames; this is called a **tick**. That is, counting the number of frames from 1 (at the beginning of the game), the updates of all objects (including status/variable value changes and movements) should only happen in the frame number that is an integer multiple of 10, e.g., 10, 20, 30, ...

Figure 1 shows a screenshot from the simulation after completing Project 1.

Treasure Hunt Game

The Player enters a tomb to search for a treasure box. The tomb is filled with zombies, that the Player needs to fight before reaching the treasure. While the fighting is energy-consuming, the tomb is also filled with the nutritious foods to let the Player regain strength.

In Project 1, you will implement three entities: the Player; a zombie; and a sandwich; and the basic interactions between them. There is **only one player** in the treasure hunt game. For Project 1, there is one zombie and one sandwich. The character and function descriptions of the Player, zombie and sandwich are outlined below.

(a) Zombie

(b) Sandwich

(c) Player

Figure 2: Images of Characters in ShadowTreasure

Background

There is a static background that should be rendered below all of the other elements, and will not change. The image for the background is located at `res/images/background.png`. It should be rendered so that its top-left is at the coordinate (0, 0).

Entities

The Player (described below), can interact with other entities in the tomb. In Project 1, you need to implement two types of stationary entities (in addition to the Player): a zombie and a sandwich.

- **Zombie:** a stationary entity in the tomb. Its image is located at `res/images/Zombie.png`. See Figure 2(a).
- **Sandwich:** a stationary entity that provides nutrition to the Player. Its image is located at `res/images/Sandwich.png`. See Figure 2(b).

Player

The Player is an entity with an associated image `res/images/player.png`. See Figure 2(c). The Player has the following attributes:

- *Energy*: an `int` attribute that indicates the energy level of the Player. The energy level is updated when the Player interacts with the entities (zombie and sandwich), according to Algorithm 1. The energy level should be displayed in black at position $(20, 760)$ in the form of

  ```
  energy: [energy level]
  ```

  in size 20 and font DejaVuSans-Bol (use the file `DejaVuSans-Bold.ttf` provided in the res.zip file). See Figure 1.
- *Step size*: The Player has step size 10 indicating the distance she moves when she moves one step.

Algorithm 1: Interaction Logic and Energy Update of Player in **Each Tick**

if *the Player meets a zombie* **then**
  the Player is frighted by the zombie and her energy level is reduced by 3. The treasure hunt game will be terminated.[2]
else if *the Player meets a sandwich* **then**
  the Player eats the sandwich, which increases her energy level by 5. The sandwich will disappear from the screen.
endif
if *the Player's energy level* ≥ 3 **then**
  the Player moves towards the zombie by one step.
else
  the Player moves towards sandwich by one step.
endif

Initial Positions

The initial positions of the entities are determined by a **environment file**, located at `res/environment.csv`. This is a comma-separated value (CSV) file with rows in the following format for zombie and sandwich:

```
[Type], x-coordinate, y-coordinate
```

where the [Type] is Zombie or Sandwich respectively and the x-coordinate and y-coordinate show the location of the entity.

A row in the file for the Player will have the format:

```
Player, x-coordinate, y-coordinate, energy level
```

where x-coordinate and y-coordinate show the initial location of the Player and the last column, energy level, shows the starting energy level of the Player. The environment file we provided in the **zip file** has the following content:

```
Player,650,100,2
Zombie,300,200
Sandwich,500,400
```

This means that you should render the Player, a zombie and a sandwich at position $(650, 100)$, $(300, 200)$ and $(500, 400)$, respectively, at the beginning of the game. The Player's initial energy level is 2.

You must actually load the environment file—copying and pasting the data, for example, is not allowed. You must load this environment file and create the corresponding entities in your simulation. When we testing your code, we may use a different environment file.

Algorithm 1: Interactions between the Player and other entities and energy update

The Player has strong radar that collects the information (type and position of other entities) in the game. She should interact and set/change the direction of her movement in each tick according to Algorithm 1. See the following notes for Algorithm 1.

- *Definition of 'meet'*: In lines 1 and 3, the Player is considered to **meet** to an entity (zombie or sandwich), if the Euclidean distance between the Player and the entity is **(strictly) less than 50**. Recall that Euclidean distance between (x_1, y_1) and (x_2, y_2) is

$$\text{dist} = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}.$$

- *One-step move*: Refer to the solutions to **workshop questions in Week 5** for how to make the Player move by one step in lines 7 and 9. Note, the direction should be normalized so that the length of each move is 10.[3] All related information is provided in the solutions to Week 5 workshop.

Output

It is required that you trace and output the Player's movement. You need to print to the standard output (stdout), i.e., the screen/console, the position the Player moves to and her energy level in each tick, including the initial position and energy level (as readfrom the environment file), in the form of

```
x-coordinate, y-coordinate, energy level
```

in one line for each tick.

Use the method with the following signature:

```
public static void printInfo(double x, double y, int e)
```

in the class `ShadowTreasure`, in the skeleton code provided (described later), to print the position of the Player.

Note: You should make sure you write to stdout correctly in each tick. We will read the stdout when we check the functionality of your code.

Your Code

You are required to submit the following:

- a class named `ShadowTreasure` that contains a `main` method to run the simulation described above

[2] In project2, you will be asked to make the Player fight with the zombie. But, you don't need implement this part in project 1.

[3] When you look into it, apply some tolerance: 10 ± 0.1. This is also the tolerance we will apply when we test your code.

- and all other classes you have designed and implemented to make the game work.

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 help you get started, here is a checklist of the required functionality, with a suggested order of implementation:

- Draw the background on screen.
- At the beginning of the game, load the environment file and draw the Player, a zombie and a sandwich at the designated position on screen.
- Implement Algorithm 1.

Supplied Package

You will be given a package `project-1.zip`, which contains: (1) Skeleton code for the `ShadowTeasure.java` class in the `res` folder to help you get started - this class also has the method to print the Player information as described above; (2) All of the graphics and other files you need to build the game in the `res` folder; (3) Input and output files for two test scenarios in the `test` folder; and (4) `pom.xml` file required for Maven. Below is a more detailed description of its contents:

- `src/` – The folder for the code.
  - `ShadowTreasure` – Skeleton code for a game.
- `test/`
  - `test1/` – the sample video in Canvas shows the game simulation for test1
    - `environment.csv`: The environment file, the same file as in `res/IO/environment.csv`.
    - `output.csv`: The output that is expected to display on the screen after you load `environment.csv` and run `ShadowTreasure`.
  - `test2/`
    - `environment.csv`: The environment file.
    - `output.csv`: The output that is expected to display on the screen when you load `environment.csv` and run `ShadowTreasure`.
- `res/` – The resources for the simulation.
  - `images/` – The image files for the simulation.

    - `background.png`: The background image
    - `zombie.png`: The image for the zombie
    - `sandwich.png`: The image for the sandwich
  - `IO/` – The input/output files for treasure hunt game.
    - `environment.csv`: The environment file.
  - `font/` – The font files for the simulation.
    - `DejaVuSans-Bold.ttf`: The fond file.

Make sure you **do not change anything in sub-directories `images/` and `font/`**: We may load `res/` in your submission when testing your code.

Note: To get started with the project, clone the [user-name]-project-1 project from GitLab, copy the contents of this zip file to [user-name]-project-1 folder in your machine and add, commit and push the initial code. Then start implementing the project (adding code to meet the requirements of the specification). Please remember to add, commit and push as you add code regularly as you progress, adding meaningful comments for your commits.

Testing: Test your code by running the game `ShadowTreasure` with at least the two `environment.csv` files provided in `test1` and `test2` sub-folders under the `test` folder in the supplied package, and then compare the output with the `output.csv` in the respective folders.

Note: Please do additional testing as needed. When marking, we will be testing your code with other input files.

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 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. An example repository has been set up here showing an ideal repository structure. At the **bare minimum** you are expected to follow the following structure. You **can** create more files/directories in your repository.

```
username-project-1
├── res
│   └── resources used for project
└── src
    ├── ShadowTreasure.java
    └── other Java files, including at least one other class
```

On 23/4/2021 at 21:00 AEST, 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 23/04/2021 at 20:59 AEST will be marked. You **must** make at least 4 commits 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:

- displayed background and load the Player, zombie and sandwich graphics correctly
- implemented the Player's movement logic (Algorithm 1) correctly
- refactored code for cleaner design

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

- fesjakhbdjl
- i'm hungry
- 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 `static final` variable. Don't use magic numbers!

  - A string value that is **used once** and is self-descriptive (e.g. a file path) does not need to be defined as a constant.

- 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 Ni Ding at `ni.ding@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 Gitlab as usual; please do however email Ni Ding once you have submitted your project.

The project is due at **20:59 AEST sharp**. Any submissions received past this time (from 21:00 AEST 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 Ni Ding 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 rendered correctly; Player, zombie and sandwich are rendered correctly according to the environment file: **1 mark**
  - Player movement and interaction logic (see Algorithm 1) is correct: **3 marks**
    - Correct interaction with zombie and sandwich, e.g., turning direction, etc., when she meets another entity: **1 mark**;
    - energy level is updated correctly: **0.5 mark**;
    - Move towards correct direction, when she does not meet another entity: **1 mark**;
    - Correct step size: **0.5 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 long/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, etc.: **1 mark**