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

Project 1, Semester 2, 2020

Released: 04/09/2020
Due: 18/09/2020 at 20:59 AEST

Overview

Welcome to the first project for SWEN20003! Across Project 1 and 2, you will design and create a simulation of a fictional universe. In this project, you will create the basis of a larger simulation that you will complete 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 (2D graphics, timing, geometric calculations)
- give you experience writing software using an external library (Bagel)

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

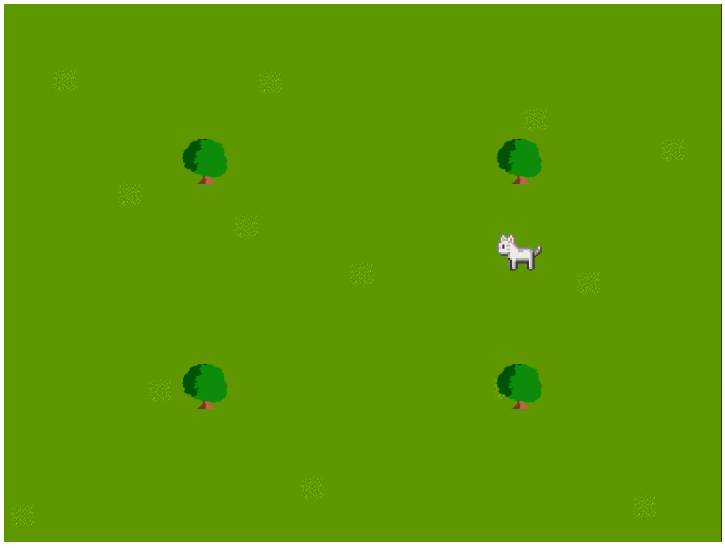

Figure 1: Completed Project 1 Screenshot

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[1]—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.

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.

This simulation is divided into **tiles** that are 64 pixels wide, and 64 pixels high. Simulation elements should always be in one well-defined tile, and a tile may contain more than one element.

The simulation state will only change every 500 milliseconds; this is called a **tick**. For practical reasons, it is not always possible to update exactly at this frequency—you should ensure *at least* 500 milliseconds pass between ticks (but not significantly more than 500).

Hint: you can use the `System.currentTimeMillis()` method to achieve this functionality.

Simulation Elements

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

The 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).

Actors

An **actor** is an object with an associated image that is located at a particular tile, and may perform an action every tick. For now, there are two types of actors:

- **Trees:** a tree is stationary and for now takes no action upon a tick. Its image is located at `res/images/tree.png`.
- **Gatherer:** a gatherer is mobile. At the beginning of the simulation, and every five ticks, it should pick a random direction (left, right, up, or down; diagonal movement is not allowed). Every tick, it should move one tile in its chosen direction. (It may move off the visible screen.) The gatherer's image is located at `res/images/gatherer.png`.

[1] https://people.eng.unimelb.edu.au/mcmurtrye/bagel-doc/

Worlds

The simulation is defined by a **world file**, describing the types and positions of the actors that should appear. For Project 1, you only need to be able to load one world, located at `res/worlds/test.csv`. (You must actually load it—copying and pasting the data, for example, is not allowed.) 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:

```
Tree,256,192
Tree,256,512
Tree,704,192
Tree,704,512
Gatherer,512,384
Gatherer,384,512
```

You must load this world file and create the corresponding actors in your simulation.

Your Code

You are required to submit the following:

- a class named `ShadowLife` that contains a `main` method to run the simulation described above
- at least one other class (the purpose of which is up to you)

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
- Draw a tree on screen
- Draw a gatherer on screen
- Read the world file, and draw the corresponding actors on screen
- Implement the movement logic for the gatherer

Supplied Package

You will be given a package `res.zip`, which contains all of the graphics and other files you need to build the simulation. Here is a brief summary of its contents:

- `res/` – The resources for the simulation.
  - `images/`: – The image files for the simulation.
    - `background.png`: The background image
    - `tree.png`: The image for the tree
    - `gatherer.png`: The image for the gatherer
  - `worlds/`: – The worlds for the simulation.
    - `test.csv` – The testing world file (no others for Project 1)

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[2] 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
    ├── ShadowLife.java
    └── other Java files, including at least one other class
```

On 18/09/2020 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 18/09/2020 at 20:59 AEST will be marked. You **must** make at least 4 commits throughout the de-

[2] https://gitlab.eng.unimelb.edu.au/emcmurtry/emcmurtry-project-1

velopment 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:

- display background and tree graphics
- implemented gatherer movement logic
- 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 E,eanor at `eleanor.mcmurtry@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 **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 Eleanor 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: **1 mark**
  - World is loaded from world file: **1 mark**
  - Trees and gatherers appear correctly: **1 mark**
  - Gatherer movement logic is correct: **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 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**