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

Project 1, 2020

Released: 24/04/2020
Due: Friday, 08/05/2020 at 11:59pm AEST

Overview

Welcome to the first project for SWEN20003! Across Project 1 and 2, you will design and create a tower defense (TD) video game. Tower defense is type of strategy game where your goal as the player is (typically) to defend your territory from attackers. In this project, you will create the basis of a larger tower defense 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. 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)

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

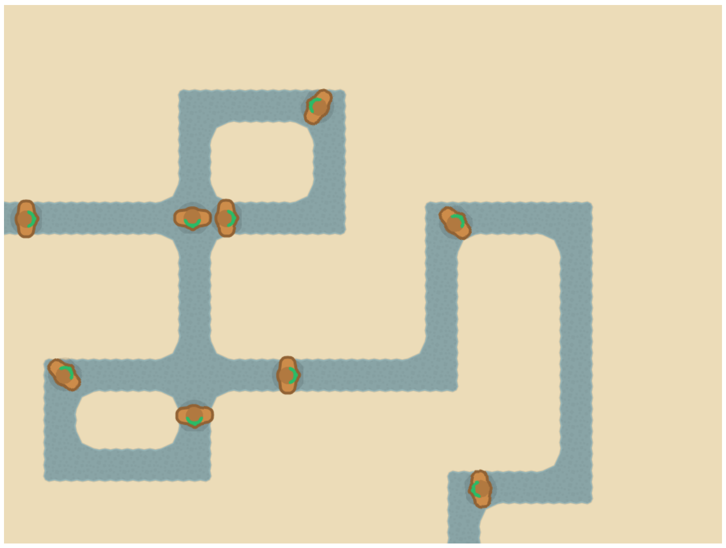

Figure 1: Completed Project 1 Screenshot

Game Elements

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

The Map

When the game is run, the map should be rendered to the screen. The map files for this project will be supplied in the TMX format[1]. Bagel has ***rudimentary*** functionality to parse and render a tiled map, so you do not need to worry about the specifics regarding the map format. The map contains two main pieces of metadata:

- Blocked tiles
- Polyline

A polyline is a connected sequence of line segments. A polyline is described by a list of `Points` that make up the line. Figure 2 shows the polylines associated with the given map. The corners are not smooth curves, but a number of small line segments connected to give the impression of one.

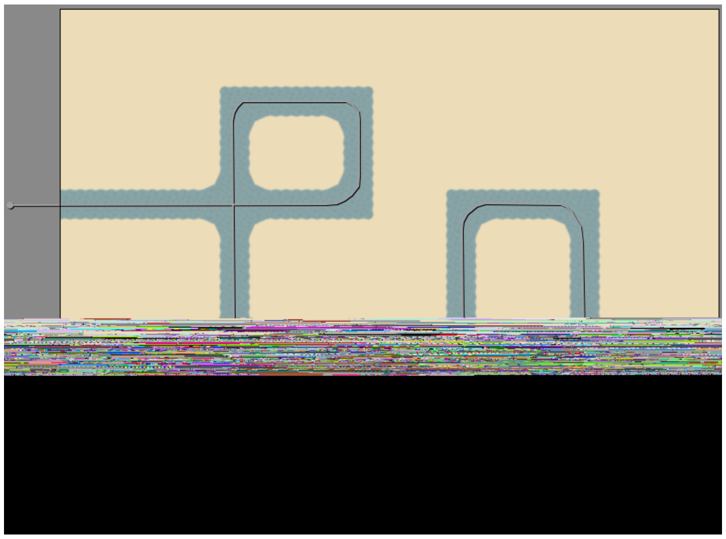

Figure 2: Polyline visualisation, the polyline starts just outside the left border, and terminates just outside the bottom border

For this project, the only metadata that you need to be concerned about is the polyline associated with the map. The TMX format (and subsequently Bagel) supports multiple polylines, but the maps we supply will only contain one. The `getAllPolylines` method of `TiledMap` returns a `List<List<Point>>`. `Lists` are covered later in the semester, so here is example code showing ***one way*** to iterate through the list and get the `Points`:

```
// Iterates through each point in the polyline
for (Point point : map.getAllPolylines().get(0)){
    // Access each Point of the polyline here
}
```

[1]The TMX map format is used by Tiled to represent a tile-based map.

The Slicers

Slicers are the most basic enemy of the game. When the 'S' key is pressed, a new **wave** begins. A wave consists of the spawning (creation) of multiple slicers. There should be a spawn delay of 5 seconds between two consecutive slicers. For this project, there will be only one wave, and that wave will consist of 5 slicers.

The goal of the slicer is simple: navigate through the enemy territory and get to the other end. When a slicer is spawned, it is spawned at the start of the polyline described by the current map. A slicer moves through the enemy territory by traversing the polyline at a rate of 1 px/frame.

When all slicers from the wave have reached the other side of the enemy territory, the map is considered finished. For this project, when a map is finished the game window should close.

If there is a wave in progress when the 'S' key is pressed nothing should happen.

Timescale Controls

The provided timings (5 second delay between slicers, and a slicer movement rate of 1px/frame) assume a timescale multiplier of 1. In our complete tower defense game, we often will want to speed things up so that we don't have to sit through waves that will take a long time to complete.

When the 'L' key is pressed, the timescale multiplier should increase by 1. When the 'K' key is pressed, the timescale multiplier should decrease by 1 (if possible). The timescale multiplier should not go below 1. For example, when the 'L' key is pressed once during a wave, the spawn delay between slicers should be decreased to 2.5 seconds, and the movement rate of slicers should be increased to 2px/frame.

The effect of a change in the timescale multiplier should be reflected immediately within the game.

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. The documentation will contain answers to most questions about map loading and rendering, accessing the polyline(s) in a map, and drawing images to the screen.

Graphics Concepts Overview

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.

60 times per second, the program's logic is updated, the screen is 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 is called. It is in this method that you are expected to update the state of the game.

Often, we would like to draw ***moving*** objects. These are represented by a **velocity**: a pair (also called a **vector**) (v_x, v_y) that represents how far the object should move each frame. Calculating the new position can be done via ***vector addition*** of the position and velocity; Bagel contains the `Vector2` class to facilitate this. You are not required to use this class; it is merely provided for convenience. The **magnitude** (or **length**) of a vector (x, y) can be found via the formula $\sqrt{x^2 + y^2}$.

Your Code

You must submit a class called `ShadowDefend` 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 map on screen
- Draw a slicer on screen
- Make slicer spawn at beginning of the map's polyline
- Implement logic to make slicer traverse entirety of polyline
- Rotate slicer to face its direction of travel
- Spawn multiple slicers with a time delay
- Implement timescale functionality ('L' and 'K' keys)
- Implement wave starting functionality ('S' key)

Supplied Package

You will be given a package `res.zip`, which contains all of the graphics and other files you need to build the game. You can use these in any way that you want. Here is a brief summary of its contents:

- `res/` – The images for the game.
  - `images/`: – The image files for the game
    - `slicer.png`: The image to represent a slicer
  - `levels/`: – The levels for the game
    - `1.tmx` – The map file
    - `sheet.tsx` – Tile Set XML file to support the map file
    - `sheet.png` – Tilesheet image to support the TSX file

The files **sheet.tsx** and **sheet.png** exist to provide data for the map file. You do not need to understand the internals or interact with these files at all, Bagel will automatically handle them when loading in the map.

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/folders in your repository.

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

On 09/05/2020 at 12:00am, 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 08/05/2020 11:59pm will be marked. You **must** make at least 5 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:

- implemented polyline traversal logic
- fix slicer orientation 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!
- 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 Rohyl at `rohyl.joshi@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 Rohyl once you have submitted your project.

The project is due at **11:59pm sharp**. Any submissions received past this time (from 12:00am 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 Rohyl so that we can ensure your late submission is marked correctly.

Marks

Project 1 is worth **15** marks out of the total 100 for the subject.

- Features implemented correctly – **10 marks**
  - Map is rendered correctly: **2 marks**
  - Slicers spawn with appropriate time delay: **2 marks**
  - Slicers traverse polyline appropriately: **2 marks**
  - Slicers orient themselves correctly: **2 marks**
  - Game ends when all slicers escape (reach the end of the polyline): **2 marks**
- Code (coding style, documentation, good object-oriented principles) – **5 marks**
  - Delegation – breaking the code down into appropriate classes (**2 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**)