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

Project 2, 2020

Released: Friday 8th May 11:59pm
Project 2A due: Friday 22nd of May, 11:59pm
Project 2B due: Saturday 13th of June, 4:59pm

Overview

In this project, you will create a graphical game in the Java programming language, continuing from your work in Project 1. We will provide a full working solution for Project 1; you are welcome to use all or part of it, provided you add a comment explaining where you found the code at the top of each file that uses the sample code.

This is an **individual** project. You can discuss it with other students, but all submitted code must be your own work.

There are two parts to this project, with different submission dates.

The first task, **Project 2A**, requires that you produce a class design demonstrating how you plan to implement the game. This should be submitted in the form of a UML diagram showing all the classes you plan to implement, the relationships (e.g. inheritance and associations) between them, and their attributes, as well as their primary public methods. (Constructors, getters, and setters need not be explicitly included.) If you so choose, you may show the relationships separately to the class members in the interests of neatness, but you must use correct UML notation. **Please submit as PDF only.**

The second task, **Project 2B**, is to complete the implementation of the game as described in the rest of this specification. You do not have to follow your class design; it is there to encourage you to think about object-oriented principles before you start programming. Indeed, it is expected that some aspects of your design will need to change when you start programming.

Shadow Defend

Figure 1: Shadow Defend Gameplay

Shadow Defend is a tower defence game where the player must attempt to defend the map from enemies using towers. As such, there are two main components of the game: **slicers** (the enemy), and **towers**.

Towers are manually purchased by the player and can be placed on a non-blocked area of the map. There are different types of towers, each with different characteristics, but the overall goal of a tower is to prevent slicers from exiting the map.

Slicers are the main antagonists of the game. They spawn at varying points during a **wave**. Like towers, there are different types of slicers, each with different characteristics, but the overall goal of a slicer is to reach the 'exit' of the map.

When the game begins, the first level is loaded and rendered, the buy panel and status panels are rendered, and $500 starting cash is awarded to the player as shown in Figure 1. The player can set up their first defensive tower (if they want) and then when they're ready, they can press 'S' to initiate the first wave of enemies.

Waves

A wave consists of a number of *wave events*. The waves for every map is the same, and they are specified inside `waves.txt`:

waves.txt (example - your waves.txt will differ)

```
1,spawn,20,slicer,1000
1,delay,2000
2,spawn,35,slicer,500
3,spawn,25,slicer,500
3,spawn,18,superslicer,1000
...
```

Each line of the file specifies a **wave event**. There are two types of wave events:

Spawn Event

A spawn event comes in the form:

```
<wave number>,spawn,<number to spawn>,<enemy type>,<spawn delay in milliseconds>
```

For example: `1,spawn,20,slicer,1000` represents a spawn event for **wave 1** that spawns **20** enemy **slicer**s with a delay of **1000**ms between each consecutive spawn. In total, this wave event should take 19000ms to complete (since the first slicer is spawned as soon as the wave event is started). The next wave event begins immediately after the previous wave event is completed.

Delay Event

A delay event comes in the form:

```
<wave number>,delay,<delay in milliseconds>
```

For example: `1,delay,1234` represents a delay event for **wave 1** that simply waits (delays) for **1234ms** before moving onto the next wave event.

Wave Logic

A wave begins when the 'S' key is pressed. The wave events are processed in order until there are no wave events left to process. When there are no events to process for the current wave, and there are no enemies left on the map, the wave is considered finished and the player is to be awarded $\$150 + wave \times \$100$ where *wave* is the wave number that just finished. For example, if wave 6 just ended, the player would be granted \$750. If there is a wave in progress when the 'S' key is pressed - nothing should happen.

There will be no missing waves (i.e. a gap between two consecutive wave numbers). There is no limit on the number of waves that may be in a map. For a given wave, there may be zero or more spawn and delay events, but there will always be at least one of either.

Wave Event Validity and Modifications

You are free to modify the `waves.txt` file to suit your testing and development process. We will be marking with our own `waves.txt` (following the proper format with valid input). You can assume all wave events in waves.txt are valid and can be processed (i.e. valid enemy types, sequential wave numbers, positive integers for all numeric fields). The provided `waves.txt` file will differ from that used in the demonstration video and in testing.

Panels

There are two main panels in the game: the **buy panel** and the **status panel**.

Buy Panel

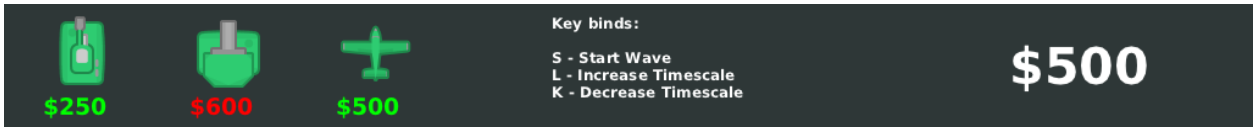

Figure 2: Buy panel

The buy panel contains the towers available for purchase (*purchase items*), a list of key binds, and the money the player currently has. The background for the buy panel is provided for you in the images folder. The first purchase item should be offset 64px horizontally (from the left of the window). There should be a 120px gap between the center of each purchase item, and the center of the purchase items should be drawn 10px above the center of the buy panel. The top left coordinate of the money indicator should be rendered 200px to the left of the right edge of the window, and 65px from the top of the window. The buy panel will always be rendered at the top of the screen.

The cost of the purchase item should be rendered below its image. The costs for each purchase item are detailed in Table 1. If the player has enough funds to buy the purchase item, it should be rendered in green - otherwise in red.

Purchase Items

Purchase Item	Price
Tank	$250
SuperTank	$600
Airplane	$500

Table 1: Purchase Items

When a panel item is left clicked (and the player has the funds to purchase it), a copy of the panel item image should be rendered at the user's cursor so that they can have a visual indicator of where the tower is to be placed. This is shown in Figure 3. If the tower cannot be placed where the user's

Figure 3: Click and Place

cursor is, then the visual indicator should not be rendered when hovering over that spot. A tower cannot be placed on a coordinate if the center of the tower to place intersects with a panel, the bounding box of another tower, or with a blocked tile. Clicking again while a panel item is selected (and the cursor is over a valid place on the map) should create a new tower at the chosen location and deduct the cost of the tower from the player's money. Clicking while the cursor is over an invalid place on the map should not do anything, and it should leave the purchase item as selected. Right clicking while a purchase item is selected should de-select the tower.

Key Binds

In the middle of the buy panel, a list of key binds should be rendered. This is just an informational feature that will help the player understand their capabilities. The key binds for the game are described in Table 2.

Key	Action
S	Start wave
L	Increase timescale by 1
K	Decrease timescale by 1

Table 2: Key binds

Status Panel

Figure 4: Status Panel

The status panel shows the state of the game at any given time. The state of the game can be described by: the current wave (either the wave currently in progress or the wave we are waiting to start), the current time scale, current "status", and the number of lives left. The player initially starts with 25 lives. The current status can be be one of:

- Winner!

- Placing
- Wave In Progress
- Awaiting Start

These statuses are provided in order of priority. For example, if the player is *placing* a tower during a *wave in progress*, the status would be "placing".

The status panel is rendered at the bottom of the screen, and the text for the status panel should be positioned as shown in Figure 4. The precise location does not matter as long as it is roughly similar and all text stays within the window bounds. If the time scale is greater than 1 it should be rendered green; otherwise, it should be rendered white.

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
- Polylines

A polyline is a connected sequence of line segments that are described by a list of `Points`. Figure 5 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.

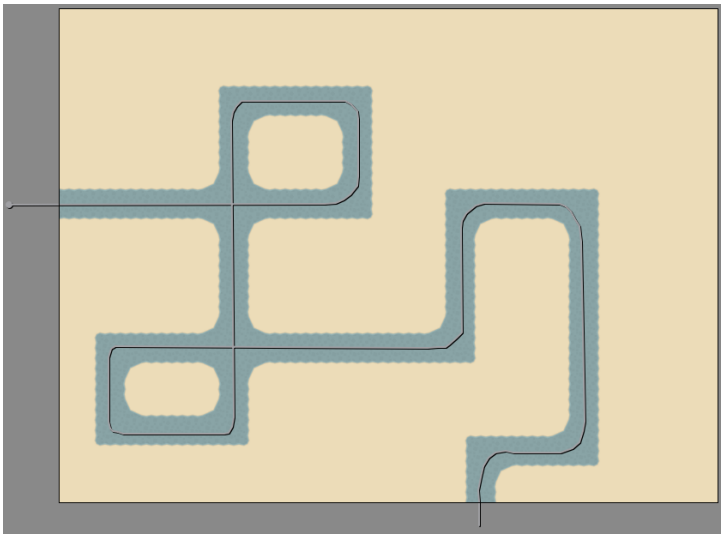

Figure 5: Polyline visualisation, the polyline starts just outside the left border, and terminates just outside the bottom border

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

Unlike Project 1, each tile may or may not contain the *blocked* map property. We carry over the same polyline logic from Project 1. The TMX format (and subsequently Bagel) supports multiple polylines, but the maps we supply will only contain one.

Timescale Controls

The delays and movement rates in this project specification assume a timescale 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 should increase by 1 (if possible). When the K key is pressed, the timescale should decrease by 1 (if possible). The timescale should not go below 1 or above 5. If we have some delay in our game with a specified base value of 5 seconds and a movement rate of some object that is 1px/frame and the 'L' key is pressed, the delay should be decreased to 2.5 seconds, and the movement rate of the object should be increased to 2px/frame. If it is pressed again the delay should be decreased to 1.67 seconds and the movement rate increased to 3px/frame.

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

The Slicers

Slicers are the base enemy of the game. 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. When a slicer successfully exits the map without being eliminated, the player loses lives equivalent to the penalty associated with the slicer. When the number of lives a player has reaches zero (or lower), the game should exit.

A slicer starts off with a specific amount of **health**, a **speed**, a **reward** and a **penalty**. When the health of a slicer reaches 0 (or lower), the slicer is considered "eliminated" and should be removed from the game, and the value of its reward added to the player's money. Any extra damage inflicted upon a slicer **is not** carried over to its child slicers (if any).

Regular Slicer

Figure 6: Regular slicer

A regular slicer moves at a rate of 2px/frame, has a health of 1 unit, a reward of $2, and a penalty of 1 life.

Figure 7: Super slicer

Super Slicer

A super slicer moves at 3/4 the rate of a regular slicer, has the same health as a regular slicer, has a reward of $15, and has a penalty equivalent to the sum of its child slicers' penalties.

When a super slicer is eliminated, it spawns two regular slicers at the location it was eliminated.

Mega Slicer

Figure 8: Mega slicer

A mega slicer moves at the same rate as a super slicer, has double the health of a super slicer, has a reward of $10, and has a penalty equivalent to the sum of its child slicers' penalties.

When a mega slicer is eliminated, it spawns two super slicers at the location it was eliminated.

Apex Slicer

Figure 9: Apex slicer

An apex slicer moves at the half the rate of a mega slicer, has 25 times the health of a *regular slicer*, has a reward of $150, and a penalty equivalent to the sum of its child slicers' penalties.

When an apex slicer is eliminated, it spawns four mega slicers at the location it was eliminated.

The Towers

Towers allow the player to defend the map from enemies. There are two main types of towers, active towers and passive towers. An active tower can deal damage to an enemy within its **radius**. It typically does this by **facing the enemy** and then launching a projectile at them. When this projectile intersects with the enemy, **damage** is applied to the enemy through the reduction of the enemy's health and the projectile is removed from the game. Each active tower has a **projectile cooldown** associated with it. Once a projectile is fired, the tower enters a 'cooldown' period where it cannot attack any enemies. After the cooldown period has elapsed, the tower is allowed to launch

another projectile. If there are multiple enemies within the tower's area of effect, you are free to choose an enemy to target however you want. Although we will specify the area of effect of active towers in terms of its radius (which implies a circular region), you are allowed to consider this area a rectangular region to make things simpler.

Tank

Figure 10: Tank

A tank is an active tower that has an effect radius of 100px, does 1 unit of damage per projectile, and has a projectile cooldown of 1000ms. Its image is `tank.png`, and its projectile's image is `tank_projectile.png`.

Super Tank

Figure 11: Super Tank

A super tank is an active tower that has an effect radius of 150px, does three times as much damage as a regular tank, and has a projectile cooldown of 500ms. Its image is `supertank.png`, and its projectile's image is `supertank_projectile.png`.

Airplane

Figure 12: Airplane and Explosive

The airplane is a passive tower. When placed, the airplane (`airsupport.png`) will spawn outside of the map and will fly at a rate of 3px/frame either horizontally or vertically across the map, dropping an explosive (`explosive.png`) on the map at the plane's current coordinate every 0–3 seconds. The drop time should be chosen randomly from 1 to 2 seconds (inclusive). The drop time is chosen again after each drop. The explosive has an effect radius of 200px and detonates after 2 seconds of it being dropped. When an explosive detonates, it deals 500 damage to every enemy in its effect radius and is subsequently removed from the game.

The direction (horizontally or vertically) that the plane will travel when placed will alternate. The first plane that is placed will travel horizontally, the second will travel vertically, the third will travel horzontally, etc. If the airplane is to travel horizontally, it will be spawned outside the left edge of the window with the same y coordinate as where it was placed. The plane will then move straight to the right until it leaves the right edge of the window. If the airplane is to travel vertically, it will be spawned outside the top edge of the window with the same x coordinate as where it was placed. The plane will then move straight downwards until it leaves the bottom edge of the window.

The airplane is only removed from the game once all its dropped explosives have detonated and it has left the window. The plane cannot drop any explosives when it is outside the map.

Projectile Logic

Every **projectile** starts at the center of the attacker, and moves at a constant **speed** of 10px/frame towards a **target enemy**. If the projectile is destined for a particular enemy, the enemy may be in motion — so the projectile should always move towards the updated location of the enemy. A projectile 'hits' an enemy if the bounding box of the projectile intersects with the bounding box of the target.

Levels

The game should be able to support multiple levels. For the purposes of assessment, you can assume there will only be 2 levels in the game, however you should design your solution in a way that can easily support increasing this limit. You will be provided with two map files (`1.tmx` and `2.tmx`). When a level is complete (i.e. all waves have finished) - the game state is completely reset and the next level is loaded. When a level is loaded, the respective map file will be loaded (i.e. level 2 will load `2.tmx`). This does **not** mean the game closes and reopens - the window must stay open during the reset of the game state and loading of the next level. When all levels are completed, the game must show the status "Winner!". The first level is specified by `1.tmx`, second by `2.tmx`, etc. The game begins on level 1.

Bagel

The **B**asic **A**cademic **G**raphical **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, drawing images to the screen, and drawing text 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 of values (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

For Project 2B, 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

This project may seem daunting. As there are a lot of things you need to implement, we have provided a feature checklist, ordered roughly in the order we think you should implement them in, together with the marks each feature is worth:

- The map, buy panel and status panel are loaded appropriately onto the screen (1 mark)
- Key binds (including timescale functionality) (1 mark)
- Wave spawning (3 marks)
- Different types of slicers (2 marks)
- Purchasing and placing towers (2 marks)
- Buy panel and status panel (1 mark)
- Airplane tower (3 marks)
- Tower attacking (including projectile logic) (2 marks)
- Super/Mega/Apex slicers spawn children on elimination (1 mark)
- Players lives are tracked and when player loses all of them (1 mark)

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 resources for the game.
  - `fonts/`: – The font files for the game
  - `images/`: – The image files for the game
  - `levels/`: – The levels for the game

The **.tsx** and **.png** files within the `levels/` folder 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.

Customisation

Optional: we want to encourage creativity with this project. We have tried to outline every aspect of the game design here, but if you wish, you may customise any part of the game, including the graphics, types of slicers, towers, resources, game mechanics, etc. You can also add entirely new features. However, to be eligible for full marks, you must implement all of the features in the above implementation checklist.

For those of you with too much time on your hands, we will hold a competition for the best game extension or modification, judged by the lecturer and tutors. The winning three will be demonstrated at the final lecture, and there will be a prize for our favourite. Past modifications have included drastically increasing the scope of the game, implementing jokes and creative game design, adding polish to the game, and even introducing networked gameplay.

If you would like to enter the competition, please email the head tutor, Rohyl Joshi, at `rohyl.joshi@unimelb.edu.au` with your username and a short description of the modifications you came up with. I can't wait to see what you've done!

If you like, you may submit a minimal version of the game to be assessed, and email a second extended version to Rohyl. Extensions submitted this way may use any libraries you like, not just Bagel and the Java standard library.

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 for both Project 2A and Project 2B will take place through GitLab. You are to submit to your `<username>-project-2` 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. For Project 2A submission, you are only expected to have the `Project-2A.pdf` file in your repository. You can have other files/folders in your repository. For Project 2B you are expected to submit your regular project directory, especially the `res` and `src` folders. You can leave your Project-2A.pdf as is.

username-project-2
- Project-2A.pdf
- res
  - *resources used for project 2B*
- src
  - ShadowDefend.java
  - *other Java files*

On the deadline for both Project 2A and Project 2B, 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 the deadline will be marked. You **must** make at least 5 commits throughout the development of the Project 2A and 2B (you only really need 1 commit for Project 2A), 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 Tower rotation logic
- fix projectile launch logic
- refactored code for cleaner design
- uploaded Project 2A design PDF

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 GitLab as usual; please do however email Rohyl once you have submitted your project.

The project is due at 4:59pm. Any submissions received past this time (from 5:00pm 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 Rohyl so that we can ensure your late submission is marked correctly.

Marks

Project 2 is worth **35** marks out of the total 100 for the subject. A full mark breakdown appears on the next page.

- Project 2A is worth 10 marks.
  - Correct UML notation for methods (2 marks)
  - Correct UML notation for attributes (2 marks)
  - Correct UML notation for associations (2 marks)
  - Good breakdown into classes (2 mark)
  - Sensible use of methods and attributes (1 mark)
  - Appropriate use of inheritance, interfaces, and `abstract` (1 mark)
- Project 2B is worth 25 marks.
  - Features implemented correctly: 17 marks (see Implementation checklist for details)
  - Coding style, documentation, and good object-oriented principles: 8 marks
    - Delegation: breaking the code down into appropriate classes (2 marks)
    - Use of methods: avoiding repeated code and overly complex methods (1 mark)
    - Cohesion: classes are complete units that contain all their data (1 mark)
    - Coupling: interactions between classes are not overly complex (1 mark)
    - General code style: visibility modifiers, magic numbers, commenting etc. (2 marks)
    - Use of documentation (javadocs) (1 mark)