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

ShadowPirate

Project 2, Semester 1, 2022
Released: Thursday, 14/04/2022 at 8:59pm AEST
Project 2A Due: Monday, 02/05/2022 at 8:59pm AEST
Project 2B Due: Friday, 20/05/2022 at 8:59pm AEST

Please read the complete specification before starting on the project, because there are important instructions through to the end!

Overview

In this project, you will create a role-playing game in the Java programming language, continuing from your work in Project 1. We will provide a full working solution for Project 1; you **may** 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 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.

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.

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. You **do not** need to show constructors, getters/setters, dependency, composition or aggregation relationships. If you so choose, you may show the relationships separately to the class members in the interest of neatness, but you must use correct UML notation. Please submit as a **PDF file** only on Canvas.

The second task, **Project 2B**, is to complete the implementation of the game as described in the rest of this specification. You **do not** need to strictly follow your class design from Project 2A; you will likely find ways to improve the design as you implement it. Submission will be via GitLab and you must make **at least 5 commits** throughout your project.

Game Overview

"The notorious pirate Blackbeard has stolen the Crown Jewels and is now hiding on the mysterious Tortuga Island. You are a sailor in the Royal Navy's crew that has been instructed by the King to recover the jewels. Unfortunately your crew was ambushed, everyone has been captured and held prisoner on Blackbeard's ship, The Flying Dutchman near the island. Your mission, should you be brave enough to accept it, is to escape the ship, fight the pirates on the island and recover the jewels..."

The game features two levels : *Level 0* is on the ship and *Level 1* is on the island. *Blackbeard* has done some voodoo magic and Level 0 has changed from Project 1! The *sailor* looks different, can now move continuously (which means the player can move around the level by holding down the arrow keys), can perform attacks when the player presses a certain key, and he no longer falls off the edge of the ship (because he learned to not walk off edges!). The *blocks* look different too and no longer cause any damage to the sailor; they simply stop the sailor from overlapping with the blocks' location.

Level 0 will now feature a new challenge for the sailor - deadly *pirates*! Pirates are live entities that move in certain directions and have health points associated with them. They can shoot projectiles at the sailor when the sailor comes close. If a projectile hits the sailor, he will lose health points, but the sailor can still fight back. Details on this are given in the Sailor and Pirate sections. To win the level, the sailor must reach the exit, located by the ladder (in the bottom right). If the sailor's health reduces to 0 or less, the game ends.

These changes to Level 0 are additive - you should be able to implement these changes on top of either the sample solution or your existing code from Project 1, but you will be required to update the code to reflect your class design from Project 2A - e.g. use proper inheritance, association etc.

In Level 1, the sailor has escaped the ship and arrived on Blackbeard's island. To win the level and the game, the sailor must get to the *treasure.* However, the sailor has to deal with *bombs* on the island which explode (causing damage) if the sailor collides with them. In addition to the bombs, the sailor will encounter items on the island that can help him. These items (*potion*, *elixir* and *sword*) will have different effects on the sailor when collided with and the item will disappear as they collide (i.e. the item gets picked up). And of course, Level 1 will feature even more pirates, including *Blackbeard* himself! The game will end if the sailor's health reduces to 0 or less.

An Important Note

Before you attempt the project or ask any questions about it on the discussion forum, it is crucial that you read through this entire document thoroughly and carefully. We've covered every detail below as best we can without making the document longer than it needs to be. Thus, if there is any detail about the game you feel was unclear, try referring back to this project spec first, as it can be easy to miss some things in a document of this size. And if your question is more to do on **how** a feature should be implemented, first ask yourself: *'How can I implement this in a way that both **satisfies the description** given, and helps make the game **easy and fun to play**?'* More often than not, the answer you come up with will be the answer we would give you!

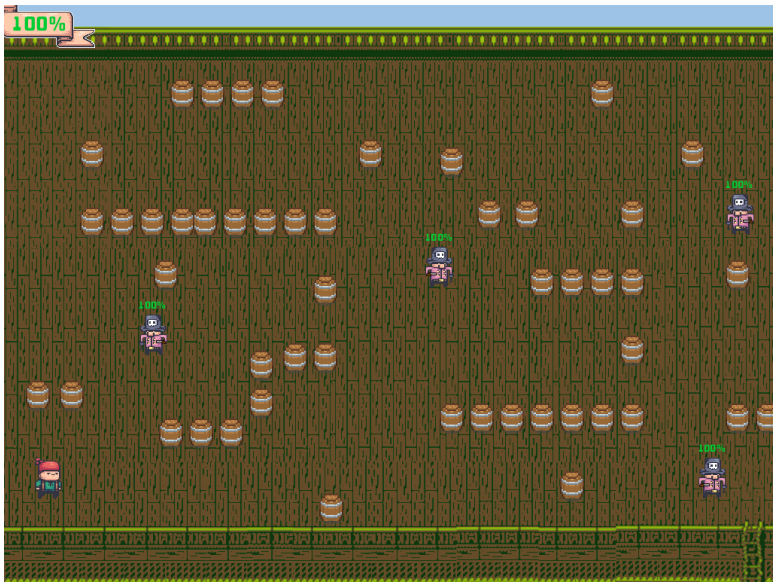


Figure 1: Completed Level 0 Screenshot

Note : the actual positions of the entities in the levels we provide you may not be the same as in these screenshots.

Figure 2: Completed Level 1 Screenshot

The Game Engine

When working with Bagel, it's useful to understand how images are rendered onto the game window in each frame and how these frames are updated between each render. 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*.

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 `ShadowPirate` 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.**

It is highly recommended that you store this refresh rate as a **constant** in your code, which can be used to calculate timers and cooldowns. For example, let's say we want the sailor's attack cooldown to be 500 milliseconds. To check whether it's been 500ms since the last attack, we can have a counter variable that gets incremented by 1 in each update() call, and by dividing this counter value with the number of frames per millisecond, we can calculate the exact number of seconds it's been. Thus, if it's been 30 frames since the sailor's last attack, and the refresh rate is 60 frames per second, we can calculate that it has been $30/\frac{60}{1000} = 500$ milliseconds since that last attack, so the sailor should be able to attack again in the next frame. Note that this is purely an example, and there are different ways to implement the cooldowns, so we'll allow some room for error in these calculations. As long as the game still plays as intended, it's okay if you miss some cooldowns by a couple milliseconds!

Collision

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. Bagel contains the `Rectangle` class to help you.

The Levels

Our game will have two levels, each with messages that would be rendered at the start and end of the level.

Window and Background

In Level 0, the background (`background0.png`) should be rendered on the screen to completely fill up your window throughout the game. In Level 1, the image `background1.png` should be used for the background. The default window size should be `1024 * 768` pixels.

Level Messages

All messages should be rendered with the font provided in `res` folder, in size `55`. The bottom left of all the messages (unless otherwise specified) should be rendered at `(x, 402)` where `x` is the coordinate such that the message is centered horizontally.

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

Level Start

When the game is run, Level 0 should start with an instruction message consisting of 3 lines:

```
PRESS SPACE TO START
PRESS S TO ATTACK
USE ARROW KEYS TO FIND LADDER
```

This instruction message should be rendered as described above, on top of the background and in the font provided. There must also be **adequate spacing** between the 3 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). Nothing else, not even the level's background, should be rendered in the window beyond this message before the level has begun.

At the start of Level 1, the following instruction message with these 3 lines should be shown:

```
PRESS SPACE TO START
PRESS S TO ATTACK
FIND THE TREASURE
```

The spacing between the lines is the same as described above.

Each level begins once the start key is pressed. To help when testing your game, you should allow the user to skip ahead to the Level 1 start screen by pressing the key 'W' (this is not assessed but will help you when coding, especially when working on Level 1).

Level Bounds

This is a rectangular perimeter that represents the edges of the level, which will be provided in the level's CSV file (`TopLeft` for the top-left (x, y) coordinate of the perimeter, `BottomRight` for the

bottom-right). You can assume that all entities provided will have a starting location within this perimeter. Moving entities, like the sailor and the enemies, should not be able to move **outside of this perimeter**. For example, if a sailor tries to move past the left boundary, instead of an immediate Game Over like in Project 1, the sailor should simply remain at the position it was at when it tried to cross this left edge until the player moves the sailor in a different direction.

For enemies, they should simply start moving in the opposite direction if they collide with the level bounds (e.g. if they were moving up when they reached the top edge, they should start moving down after the collision).

Win Conditions

For Level 0, once the sailor reaches the ladder, this is the end of the level. To reach the ladder, the sailor's x coordinate must be greater than or equal to 990 and the sailor's y coordinate must be greater than 630. A winning message that reads `LEVEL COMPLETE!` should be rendered as described in the Level Messages section. Note that nothing else must be displayed in the window at this time and this message should be rendered for **3 seconds** before displaying the start screen for Level 1.

In Level 1, once the sailor reaches (collides) with the treasure, this is considered a win. A winning message that reads `CONGRATULATIONS!` should be rendered as described in the Level Messages section. Once again, nothing else must be displayed in the window at this time, and there's no need to terminate the game window while this is being displayed.

Lose Conditions

On either level, while there is no win, the game will continue running until it ends. As will be described below, the game can only end if the sailor's health points reduce to 0 or the player terminates the game window themselves. A message of `GAME OVER` should be rendered as described in the Level Messages section. Note that nothing else must be displayed in the window at this time, and there's no need to terminate the game window while this is being displayed.

World File

All the actors will be defined in a **world file**, describing the types and their positions in the window. The world file for Level 0 is `level0.csv` and Level 1 is `level1.csv`. Both world files will now contain the level bounds at the end of each file. 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:

```
Sailor,130,670
Pirate,203,410
Block,225,50
Blackbeard,860,180
Treasure,975,50
```

```
TopLeft,0,20
BottomRight,1000,640
```

You must actually load both files—copying and pasting the data, for example, is not allowed. **Note:** You can assume that the sailor is always the first entry in both files. Also assume that a Block entry in the world file for level 1 always refers to a **Bomb**.

The Game Entities

All game entities have an associated image (or multiple!) and a starting location `(x, y)` on the map which are defined in the CSV files 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. Drawing images from the top left will make it easier for you to make use of Bagel's `Rectangle` class to check for image overlaps since you can only initialise rectangles at a top-left position. (`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

The entity that our game's players will control is the heroic sailor, and the game should also display some information about our sailor's current state, as described below.

Sailor

The sailor is controlled by the four arrow keys and can move continuously in one of four directions (left, right, up, down) by **1 pixel per frame** whenever an arrow key is **held down**.

Hint: instead of checking whether a key input `wasPressed` like in Project 1, you should be checking whether the key input `isDown`)

The sailor has health points, which is an integer value that determines their current amount of health. The sailor will always start a level with the maximum number of health points, which is **100**. It also has damage points, which determines how much damage it inflicts on enemies when they overlap; the sailor starts with **15** damage points. When receiving damage by overlapping with a hostile entity, the sailor will lose health points (based on the hostile entity's own damage points as described later). If the sailor's health points reduce to 0, the game ends.

Moreover, the sailor can now be in either of two states which determines its behavior when colliding with other entities as well as the image associated with the sailor entity. These two states are **IDLE** and **ATTACK**.

The sailor will always start a level in the `IDLE` state, facing the right direction. In this state, nothing happens to any enemies that the sailor may collide with (though these entities may still impact the sailor in some way). To render the sailor in this state, either of the two images below

will be used, depending on the direction the sailor is moving to (ie. when the sailor is moving in the left direction, the image associated with the sailor entity should be `sailorLeft.png`, and `sailorRight.png` for when it's moving to the right).

(a) sailorLeft.png

(b) sailorRight.png

Figure 3: Sailor images in IDLE state

Pressing the 'S' key will place the sailor in the `ATTACK` state (no other event should trigger this). In this state, the sailor can still move around, but colliding with any enemies will inflict the sailor's full damage points to those entities (i.e. the entities' health points will be reduced by the damage points of the sailor). This implies that the sailor can damage multiple enemies at once, depending on how many enemies it overlapped with while in the `ATTACK` state. To show that the sailor is in its `ATTACK` state, we'll use the images below for rendering. Again, the image used will depend on the direction the sailor is moving to (it may still change direction while in the `ATTACK` state). Also note that you're free to decide whether the sailor is rendered over or under the enemy when they overlap.

(a) sailorHitLeft.png

(b) sailorHitRight.png

Figure 4: Sailor images in ATTACK state

Once the 'S' key is pressed, the sailor will be in the `ATTACK` state for only **1000 milliseconds**. This means that it will return to the `IDLE` state only after this time period has passed, so moving in a different direction should not cancel out the `ATTACK` state. After the sailor has returned to the `IDLE` state after attacking, there is a cooldown of **2000 milliseconds** that must pass until the sailor is allowed to attack again. During this cooldown, pressing the 'S' key should do nothing.

Note: the sailor should have the same functionality in both Level 0 and 1.

Health Bar

The sailor's current health points value is displayed on screen as a percentage of the maximum health points. This can be calculated as $\frac{CurrentHealthPoints}{MaximumHealthPoints}$. For example, if the player's current Health Points is 15 and their maximum is 20, the percentage is 75%. This percentage is rendered in the **top left** corner of the screen in the format of `k%`, where `k` is rounded to the nearest integer. The bottom left corner of this

Figure 5: Health Bar

message should be located at `(10, 25)` and the font size should be 30.
Initally, the colour of this message should be green `(0, 0.8, 0.2)`. When the percentage is below 65%, the colour should be orange `(0.9, 0.6, 0)` and when it is below 35%, the colour should be red `(1, 0, 0)`.

Hint: Refer to the DrawOptions class in Bagel and how it relates to the `drawString()` method in the Font class, to understand how to use the RGB colours.

Inventory

Figure 6: Inventory

When the sailor picks up an item (described in the Items section below), its corresponding item icon image is rendered in the top left of the window, below the sailor's current health and will stay on screen until the end of the game. Since there are 3 items that can be picked up during the level, you must show the icons for each item that has been picked up so far **grouped vertically**, with the icon for the **first** item picked up at the **top** and that of the **last** one picked up at the **bottom**. You must not show the icons for items that haven't been picked up. For example, if the sailor has picked up items in the order: Sword, Potion, Elixir, then the icons should be shown in Figure 6.

The exact positions of these icons are for you to decide, as long as they're located below the sailor's health and have **adequate spacing** between them. You only need to show these icons for Level 1 as Level 0 will have no items.

Enemies

These are entities that can move throughout the level and attack the player. But, as described previously, the player can fight back against them, so they can die and disappear from the screen. There will be only two types of enemies, each with its own difficulty. Note that enemies are allowed to overlap with each other as well as the player during movement.

Pirate

Pirates feature in both levels. A pirate moves in one of four directions (left, right, up and down), randomly selected upon creation, at a **random speed between 0.2 to 0.7** pixels per frame. If it collides with a block, bomb or reaches the level boundary, it will rebound and move in the **opposite** direction. A pirate has 3 states which determine its behavior and image: **READY TO ATTACK**, **COOLDOWN** and **INVINCIBLE**.

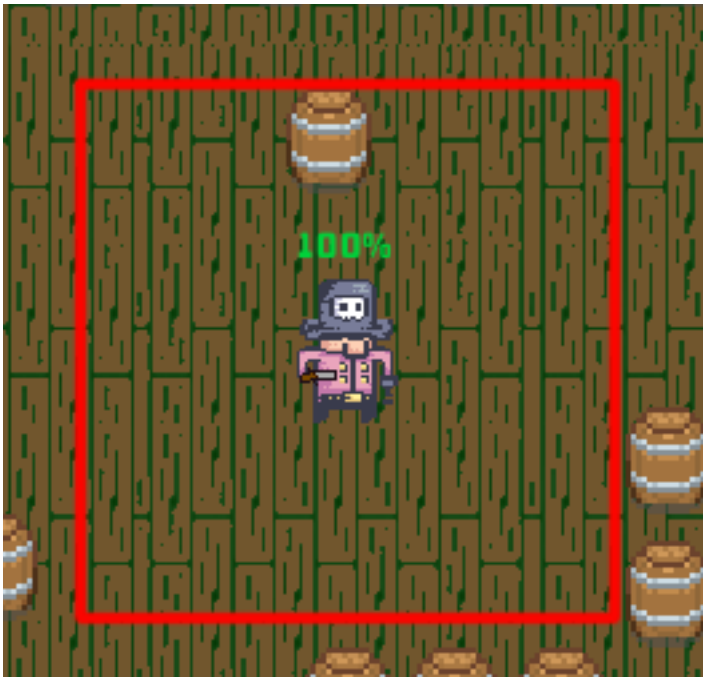
In the `READY TO ATTACK` state, the pirate moves as described above. To render the pirate in this state, either of the two images below will be used, depending on the direction the pirate is moving to (ie. when the pirate is moving in the left direction, the image associated with the pirate entity should be `pirateLeft.png`, and `pirateRight.png` for when it's moving to the right).

A pirate also has an attack range in this state, which is basically an invisible square of size **100 pixels** with the **center of the pirate's image** (not its top-left coordinate!) precisely at the **center of this square**. If the sailor enters (collides with) a pirate's attack range, it will attack the sailor by firing a projectile at them (rendered with `pirateProjectile.png`) that travels **in a straight line** from the pirate's location to the location of the player **when the projectile was fired**. This means that the player should be able to easily dodge fired projectiles by moving around it. If the projectile collides with the sailor, it will cause 10 damage to the sailor's health. Details on the projectiles are given in the Projectile section.

(a) pirateLeft.png

(b) pirateProjectile.png

(c) pirateRight.png



(d) Attack range placement (the square should be invisible, the border shown above is only for visualization)

Figure 7: Pirate images in READY TO ATTACK and COOLDOWN state

Once the pirate fires one projectile, it immediately goes into the `COOLDOWN` state for **3000 milliseconds**. In this state, the pirate cannot fire a projectile and can only move around. Once the time elapses, the pirate goes back into the `READY TO ATTACK` state.

If a pirate gets attacked by the sailor, the pirate should go into an `INVINCIBLE` state for **1500 milliseconds**, rendered using the images below to visualize that they're invincible. In this state,

any attack by the sailor will not cause damage to the pirate. Once this time elapses, the pirate goes back to its default behavior. Note that the `INVINCIBLE` state may overlap with the other two states (eg. the pirate may still be able to shoot projectiles at the sailor while it's invincible).

(a) pirateInvincibleLeft.png

(b) pirateInvincibleRight.png

Figure 8: Pirate images in INVINCIBLE state

Figure 9: Enemy health

A pirate starts with **45** health points and its current health value is displayed as a percentage using the same calculation logic explained above for the sailor. This health bar is rendered on top of the pirate's image at `(x, y - 6)` where `(x, y)` is the top left of the image, as shown here. The font size should be 15 and the color logic is the same as explained above for the sailor. If the pirate's health points reduce to 0 or less, they will die and disappear from the screen.

Blackbeard

Blackbeard is a special pirate whose projectiles deal damage value **twice** that of a normal pirate and a maximum health points value **twice** that of a normal pirate. His attack range is also twice as big as a normal pirate, and his cooldown period is half of the pirate's (1500 milliseconds). The projectile fired by Blackbeard will be rendered with `blackbeardProjectile.png`. All other behaviors of Blackbeard, as well as its speed, are the same as a normal pirate as explained above.

(a) blackbeardLeft.png

(b) blackbeardRight.png

(c) blackbeardProjectile.png

(d) blackbeardInvincibleLeft.png

(e) blackbeardInvincibleRight.png

Figure 10: Blackbeard images

Projectile

A projectile is fired by a pirate or Blackbeard as described above, and travels at a speed of **0.4 pixels per frame** if shot by a regular pirate, and **0.8 pixels** if by Blackbeard. Its image should be rotated depending on the direction it was fired at. For example, if a pirate is at the coordinate

`(3, 3)` when the sailor enters its range at coordinate `(13, 8)`, it should fire a projectile with its image rotated by `0.464 radians`. The diagram below illustrates how this can be calculated.

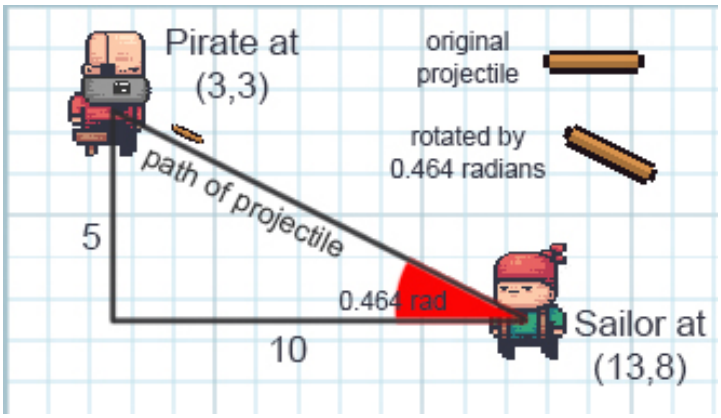


Figure 11: Projectile Explanation

Hint: to rotate an Image in Bagel, you can include a `DrawOptions` parameter in the `draw()` method. A DrawOptions instance allows you to specify detailed options for drawing images, and has a `setRotation()` method for setting the rotation value, measured in **radians**. You may need to use one of the trigonometric functions from Java's inbuilt Math class to calculate this value. Feel free to round up the value you calculate as needed, as long as the image's final rotation looks correct.

The game should stop rendering and updating a projectile only if the projectile has either **collided with the sailor** or **reached the level boundary** (not the edges of the game window). Due to whitespace in the projectile images, you should use the projectile's **position** instead of its image rectangle to check whether it has reached the sailor. Basically, a projectile's collision with the sailor should look **natural**; how this is calculated is for you to decide (eg. if the distance from the projectile's coordinate to the sailor's coordinate is below a certain threshold, the projectile can be considered to have hit the sailor.) A projectile should not be able to change direction or rotation once it has been fired. It's important that the game stops updating a projectile once it's disappeared for performance reasons.

Stationary Entities

These are entities placed throughout the level that do not move, at locations specified by the level CSV file. These may apply some effect on the moving entities that collide with them, and may need to disappear at some point (i.e. the game should stop rendering and updating them).

Block

A block has the image `block.png` and is only featured in Level 0. Unlike Project 1, when the sailor touches the block, the block **does not do any damage** to the sailor. However, the sailor still shouldn't be able to overlap

with or move through the blocks, i.e. the player must move the sailor around any areas on the level where blocks are being rendered. This rule also applies to any moving enemies that may collide with the block (they should move in the opposite direction upon collision).

Bomb

A bomb has the image `bomb.png` and features in Level 1. Each bomb has a damage point value of **10**. When the sailor collides with the bomb:

- the bomb's image is changed to `explosion.png`,
- its full damage point value is inflicted on the sailor **once** (similar to the blocks in Project 1) and

- it disappears from the screen after **500 milliseconds** has passed since the sailor first touched the bomb.

While it's still on the screen, the bomb should still behave similar to a block and prevent the sailor from moving through it. But once it's disappeared, the sailor and any moving enemies should be able to move through the area it was placed at before. The bomb **should not explode** when an enemy collides with it, nor should it inflict any damage on the enemy. It should instead simply behave as a block would for these entities.

Treasure

The treasure, has the image `treasure.png` and is the end goal of Level 1. When the sailor collides with the treasure, the player **wins the game**. It does not need to behave like a block, so it's okay if the sailor or enemies can move through it. You also don't need to show it disappearing after the sailor has collided with it as the winning screen should be rendered immediately after.

Items

These are stationary entities that can be picked up by the player by colliding with them, with their icon then shown in the player's inventory. Each can boost the player's attributes in some way, and their effects should overlap if multiple have been picked up. Picking up these items will cause them to disappear from the level. Note that enemies should be able to move through these items (without causing them to disappear), unlike blocks and bombs (whether or not the enemy is rendered under or over the item if this happens is up to you).

- ***Potion***
  A potion, with an image `potion.png`, is an item that will increase the sailor's current health points value by 25 but not beyond the sailor's current maximum health points value.

- ***Elixir***
  An elixir, with an image `elixir.png`, is an item that will **permanently** increase the sailor's **maximum** health points value by 35 and increase the current health points to the new maximum value.

- ***Sword***
  A sword, with an image `sword.png`, will **permanently** increase the sailor's damage points value by 15.

Entities Summary

Entity	Image filenames	Health points	Movement speed	Damage points	Attack range	Collision effect on Sailor
Sailor (moving)	sailorLeft.png (IDLE, moving left), sailorRight.png(IDLE, moving right), sailorHitLeft.png (ATTACK, moving left), sailorHitRight.png (ATTACK, moving right)	100	2	15	-	-
Pirate (moving)	pirateLeft.png (not INVINCIBLE, moving left), pirateRight.png (not INVINCIBLE, moving right), pirateInvincibleLeft.png (INVINCIBLE, moving left), pirateInvincibleRight.png (INVINCIBLE, moving right)	45	Random value between 0.2 to 0.7	10	100	When sailor collides with its attack range, it fires its projectile
Blackbeard (moving)	blackbeardLeft.png (not INVINCIBLE, moving left), blackbeardRight.png (not INVINCIBLE, moving right), blackbeardInvincibleLeft.png (INVINCIBLE, moving left), blackbeardInvincibleRight.png (INVINCIBLE, moving right)	90	Random value between 0.2 to 0.7	20	200	When sailor collides with its attack range, it fires its projectile

Projectile (moving)	pirateProjectile.png (if shot by regular pirate), blackbeardProjectile.png (if shot by Blackbeard)	-	Depends on the pirate type	Depends on the pirate type	-	Inflicts damage points of the pirate that shot it on sailor
Sword	sword.png, swordIcon.png (for inventory)	-	-	-	-	Increases sailor damage points by 15
Potion	potion.png, potionIcon.png (for inventory)	-	-	-	-	Increases current sailor health points by 25
Elixir	elixir.png, elixirIcon.png (for inventory)	-	-	-	-	Increases sailor's max health points by 35 and sets current health to this value
Block	block.png	-	-	-	-	Prevents sailor from moving through it
Bomb	bomb.png (when not touched by sailor), explosion.png (in exploding state)	-	-	10	-	Inflicts its full damage points on sailor, behaves like Block while exploding
Treasure	treasure.png	-	-	-	-	Ends the game with a victory screen

Log

Every time an entity *inflicts damage* on another entity or when the sailor picks up an item, a sentence detailing this is printed on the **command line**, in the following format.

- Entity A inflicting damage on entity B:

  `A inflicts x damage points on B. B's current health: y/z`

  where `x` is the amount of damage, `y` is the health points value after the damage was inflicted and `z` is the maximum health points value.

  For example:

  `Bomb inflicts 10 damage points on Sailor. Sailor's current health: 90/100`

  `Sailor inflicts 15 damage points on Pirate. Pirate's current health: 30/45`

- Sailor colliding with item A:

  `Sailor finds A. Sailor's current health: y/z`

  where `A` is either a *Potion* or a *Elixir*, `y` is the health points value after the item's corresponding point increase was done and `z` is the maximum health points value.

  For example:

  `Sailor finds Potion. Sailor's current health: 95/100`

  `Sailor finds Elixir. Sailor's current health: 135/135`

- Sailor colliding with item A:

  `Sailor finds A. Sailor's damage points increased to y`

  where `A` is a *Sword*, `y` is the damage points value after being increased by the sword.

  For example:

  `Sailor finds Sword. Sailor's damage points increased to 30`

  `Sailor finds Sword. Sailor's damage points increased to 45`

Your Code

You must submit a class called `ShadowPirate` 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 (in addition to the features in Project 1):

- Implement the changes to Level 0.
- Implement the end of Level 0 screen.
- Implement the Level 1 start screen.
- Render the Level 1 background and place the game entities read from the CSV file, on the island.
- Implement the sailor behaviour/logic.
- Implement the pirate and Blackbeard behaviour/logic.
- Implement the behaviour/logic of each item as detailed in the *Game Entities* section.
- Implement health bars, inventory, and log printing.
- Implement level bounds.
- Implement the treasure behaviour and end of Level 1 screen.
- Implement lose detection for Level 1.

Supplied Package and Getting Started

You will be given a package called `project-2-skeleton.zip` that contains the following: (1) Skeleton code for the `ShadowPirate` 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. You should use this template exactly how you did for Project 1, that is:

1. Unzip it.
2. Move the **content** of the unzipped folder to the local copy of your [username]-project-2] repository.
3. Push to Gitlab.
4. Check that your push to Gitlab was successful and to the correct place.
5. Launch the template from IntelliJ and begin coding.
6. Commit and push your code regularly.

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 actors, behaviour of actors, etc. (For example, an easy extension could be to

make the pirates move randomly on their own). You can also add entirely new features. For your customisation, you **may** use additional libraries (other than Bagel and the Java standard library).

However, to be eligible for full marks, you **must** implement all of the features in the above implementation checklist. Please submit the version **without** your customisation to [username]-project-2 repository, and save your customised version locally or push it to a new branch on your Project 2 repository.

For those of you with far 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 have their games shown at the final lecture, and there will be a prize for our favourite. Past modifications have included drastically increasing the scope of the game, adding jokes and adding polish to the game, and even introducing networked gameplay.

If you would like to enter the competition, please email the head tutor, Tharun Dharmawickrema at `dharmawickre@unimelb.edu.au` with your username, a short description of the modifications you came up with and your game (either a link to the other branch of your repository or a .zip file). You can email Tharun with your completed customised game anytime before **Week 12**. Note that customisation does **not** add bonus marks to your project, this is completely for fun. We can't wait to see what you come up with!

Submission and Marking

Project 2A

Please submit a **.pdf** file of your UML diagram for Project 2A via the Project 2A tab in the Assignments section on Canvas.

Project 2B - 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.
- For full marks, **every** public class and method must have a short, descriptive Javadoc comment (which will be covered later in the semester).

Submission will take place through GitLab. You are to submit to your `<username>-project-2` 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-2
├── res
│   └── resources used for project 2
└── src
    ├── ShadowPirate.java
    └── other Java files
```

On 20/05/2022 at 9:00pm, 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 20/05/2022 8:59pm 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 Blackbeard logic
- fix sailor collision behaviour
- refactored code for cleaner design

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

- fesjakhbdjl
- yeah easy finished the sailor stuff
- 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 final static variable (**Note:** for Image and Font objects, use only final). 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 Andrew at `andrew.valentine@unimelb.edu.au` explaining your situation with some supporting documentation (medical certificate, academic adjustment plan, wedding invitation, etc.). If an extension has been granted, you may submit via LMS (for Project 2A) and GitLab (for Project 2B) as usual; please do however email Andrew once you have submitted your project with an extension.

The project is due at **8:59pm sharp** on Monday 02/05/2022 (Project 2A) and on Friday 20/05/2022 (Project 2B). Any submissions received past this time (from 9:00pm 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 Andrew so that we can ensure your late submission is marked correctly.

Marks

Project 2 is worth **22** marks out of the total 100 for the subject. You are **not required** to use any particular features of Java. For example, you may decide not to use any interfaces or generic classes. You will be marked based on the **effective and appropriate** use of the various object-oriented principles and tools you have learnt throughout the subject.

- Project 2A is worth **8 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: **1 mark**
  - Appropriate use of inheritance, interfaces and abstract classes/methods: **1 mark**
- Project 2B (feature implementation) is worth **10 marks**.
  - Correct implementation of entity reading and creation: **1 mark**
  - Correct implementation of sailor attack behaviour: **2 marks**
  - Correct implementation of pirate attack behaviour and health points rendering: **2 marks**
  - Correct implementation of items behaviour: **2 marks**
  - Correct implementation of level transition: **2 marks**
  - Correct implementation of winning and end of game logic: **1 mark**
- Coding Style is worth **4 marks**.
  - Delegation: breaking the code down into appropriate classes: **0.5 marks**

- Use of methods: avoiding repeated code and overly complex methods: **0.5 marks**
- Cohesion: classes are complete units that contain all their data: **0.5 marks**
- Coupling: interactions between classes are not overly complex: **0.5 marks**
- General code style: visibility modifiers, magic numbers, commenting etc.: **1 mark**
- Use of Javadoc documentation: **1 mark**