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

ShadowDimension

Project 2, Semester 2, 2022
Released: Friday, 09/09/2022 at 8:59pm AEST
Project 2A Due: Wednesday, 21/09/2022 at 8:59pm AEST
Project 2B Due: Friday, 14/10/2022 at 8:59pm AEDT

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 fantasy role-playing game called *ShadowDimension* 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 are 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.

Note: If you need an extension for the project, please complete this form. Make sure you explain your situation with some supporting documentation such as a medical certificate, academic adjustment plan, wedding invitation, etc. You will receive an email saying if the extension was approved or if we need more information.

If you submit late (**either** with or without an extension), please complete this form. For both forms, you need to be logged in using your **university** account. Please **do not** email any of the teaching team regarding extensions or late submissions (as you will be redirected to the online forms). All of this is explained again in more detail at the end of this specification.

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 relationship on a separate page 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

*"A dark evil has arrived in your hometown. A group of government scientists have opened a gate in their laboratory to another dimension called the **Over Under**. The Over Under is ruled by **Navec** (an evil creature of immense power) and his henchmen are called **demons**. The scientists thought they could control these creatures but alas they failed and are being held captive in the Over Under. Navec has created **sinkholes** that will destroy the lab and is planning on eventually destroying your world.*

*The **player**'s name is **Fae**, the daughter of one of the scientists. In order to save your father and your town, you need to avoid the sinkholes, find the gate in the lab and defeat Navec & his demons in the Over Under..."*

The game features two levels : *Level 0* is in the lab and *Level 1* is in the Over Under. In Level 0, the player will be able to control *Fae* who has to move around the *walls* and avoid any *sinkholes* that are in the lab. If the player falls into a sinkhole, the player will lose health points. To finish the level, the player has to get to the gate, located in the bottom right of the window. If the player's health reduces to 0, the game ends. You have already implemented Level 0 in Project 1 (the only change required is to the winning message screen which is explained later).

When the player finishes Level 0, Level 1 starts - Fae is now in the Over Under. To win the level and the game, the player must fight and defeat *Navec*. However, the player has to deal with more sinkholes and *demons* too. The player can cause damage to the demons and Navec when the player presses a certain key. Likewise, the demons and Navec can cause damage to the player. The player will also have to move around *trees* (they don't cause any damage like the walls in Level 0).

Level 1 will also give the player the opportunity to change the *timescale* so that the difficulty of the game can be changed. Pressing a certain key will increase/decrease the speed of the demons and Navec (this is explained in more detail later). Like in Level 0, the game will end if the player'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!

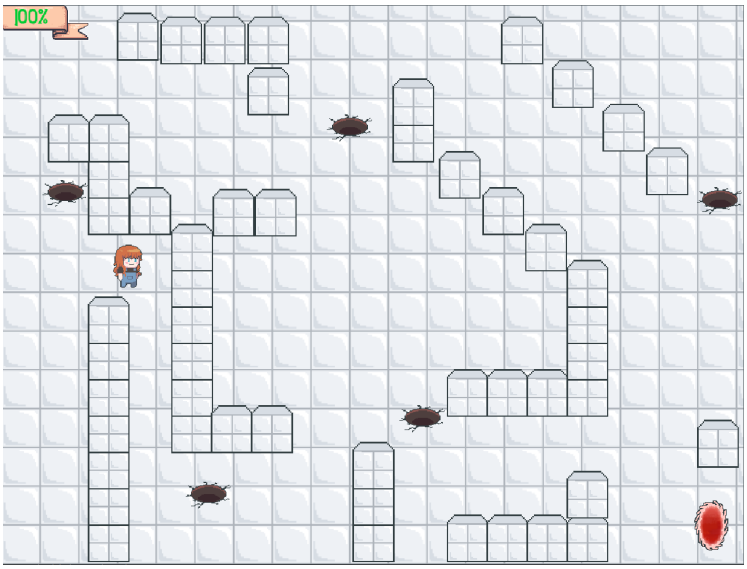

Figure 1: Completed Level 0 Screenshot

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

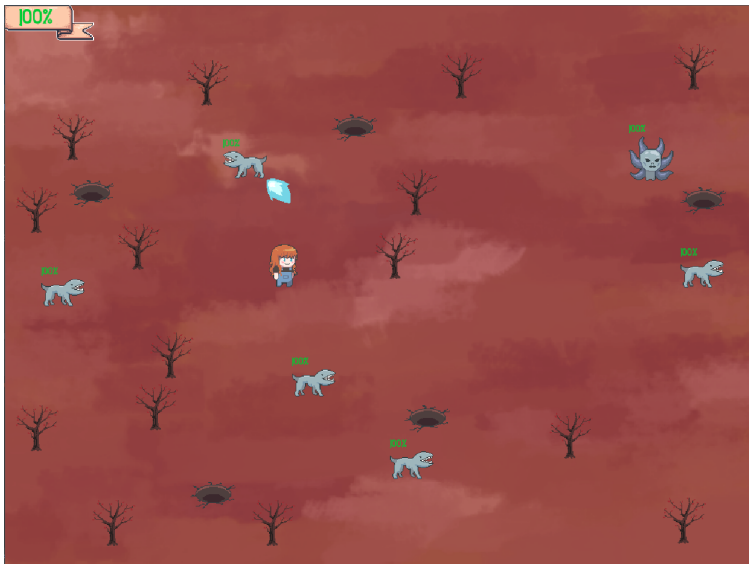

Figure 2: Completed Level 1 Screenshot

The Game Engine

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.

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*. The Bagel `Point` class encapsulates this.

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 `ShadowDimension` 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 player'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 player'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 player 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!

Collisions

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 `75` (unless otherwise specified). All messages should be centered both horizontally and vertically (unless otherwise specified).

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

Level Start

When the game is run, Level 0 should start with a title message that reads `SHADOW DIMENSION` should be rendered in the font provided. The bottom left corner of this message should be located at `(260, 250)`.

Additionally, an instruction message consisting of 2 lines:

```
PRESS SPACE TO START
USE ARROW KEYS TO FIND GATE
```

should be rendered **below** the title message, in the font provided, in size `40`. The bottom left of the first line in the message should be calculated as follows: the x-coordinate should be increased by `90` pixels and the y-coordinate should be increased by `190` pixels.

There must be **adequate spacing** between the 2 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). You can align the lines as you wish. Nothing else, not even the background, should be rendered in the window before the game has begun.

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

```
PRESS SPACE TO START
PRESS A TO ATTACK
DEFEAT NAVEC TO WIN
```

This message should be rendered in the font provided in size `40` and the bottom left of the first line in the message should be located at `(350, 350)`. The spacing and alignment of the lines is

the same as described above. Nothing else, not even the background, should be rendered in the window at this time.

Each level begins once the start key (space bar) is pressed. To help when testing your game, you can 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).

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 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:

```
Player,5,696
Tree,120,680
Demon,50,400
Sinkhole,255,655
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 player is always the first entry in both files, the Level 0 world file will have a maximum of 60 entries and the Level 1 world file will have a maximum of 29 entries.

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 player and the enemies, should not be able to move **outside of this perimeter**. For example, if the player tries to move past the left boundary, they should simply remain at the position they were at when they tried to cross this perimeter until the player is moved 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 player reaches the gate, this is the end of the level. To reach the gate, the player's `x` coordinate must be greater than or equal to 950 and the `y` coordinate must be greater

than or equal to 670. A winning message that reads `LEVEL COMPLETE!` should be rendered as described earlier in the *Level Messages* section. Note that nothing else (not even the background) 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 player defeats Navec (Navec's health reduces to 0 or below), 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 (not even the background) 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 described earlier, the game can only end if the player's health points reduce to 0. A message of `GAME OVER!` should be rendered as described in the *Level Messages* section. Note that nothing else (not even the background) must be displayed in the window at this time, and there's no need to terminate the game window while this is being displayed.

If the player terminates the game window at any point (by pressing the Escape key or by clicking the Exit button), the window will simply close and no message will be shown.

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.

Hint: `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

In our game, the player is represented by Fae. The game should display some information about her current state, as described below.

The player is controlled by the four arrow keys and can move continuously in one of four directions (left, right, up, down) by **2 pixels per frame** whenever an arrow key is **held down**.

The player has **health points**, which is an integer value that determines their current amount of health. The player will always start a level with the maximum number of health points, which is **100**. The player also has damage points, which determines how much damage they inflict on enemies when they overlap; the player starts with **20** damage points. When receiving damage by overlapping with an enemy, the player will lose health points (based on the enemy's own damage

points as described later). If the player's health points reduce to 0, the game ends. The player's health points **do not** become negative.

Moreover, the player can now be in one of three states which determines its behavior when colliding with other entities as well as the image associated with the player. These three states are **IDLE**, **ATTACK** and **INVINCIBLE**.

The player will always start a level in the `IDLE` state, facing the right direction. In this state, nothing happens to any enemies that the player may collide with (though these entities may still impact the player in some way). To render the player in this state, either of the two images below will be used, depending on the direction they are moving to (i.e. when they are moving in the left direction, the image associated should be `faeLeft.png`, and `faeRight.png` for when they're moving to the right).

(a) faeLeft.png

(b) faeRight.png

Figure 3: Player images in IDLE state

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

(a) faeAttackLeft.png

(b) faeAttackRight.png

Figure 4: Player images in ATTACK state

Once the 'A' key is pressed, the player will be in the `ATTACK` state for only **1000 milliseconds**. This means that they 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 they have returned to the `IDLE` state after attacking, there is a cooldown of **2000 milliseconds** that must pass until the player is allowed to attack again. During this cooldown, pressing the 'A' key should do nothing.

If the player takes damage from an enemy (only a demon or Navec, **not** a sinkhole), they go into the `INVINCIBLE` state for **3000 milliseconds**. In this state, any attack on the player will not cause any damage to them. Once this state elapses, the player goes back to its previous state. Note that the `INVINCIBLE` state may **overlap** with the other two states (i.e. the player can attack an enemy whilst being invincible). There are **no separate images** for this state - the player will have the same state images they had before going invincible.

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

Health Bar

The player'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 message should be located at `(20, 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)`.

Figure 5: Health Bar

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.

Enemies

These are entities that can attack the player. But, as described previously, the player can fight back against them, so they can die and disappear from the screen. Note that enemies are allowed to overlap with each other as well as the player during movement.

Demon

Demons feature in Level 1. Demons can be of two types (**passive** or **aggressive**) which is set randomly at creation. Passive demons are *stationary* and aggressive demons can *move* 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 an aggressive demon collides with a sinkhole, tree or reaches the level boundary, it will rebound and move in the **opposite** direction. A demon has 2 states which determine its behavior: **ATTACK** and **INVINCIBLE**.

In the `ATTACK` state, to render the demon, either of the two images shown in Figure 6 will be used. The direction that the passive demon is facing is set randomly at creation. The aggressive demon's image will depend on the direction it is moving to (ie. when the demon is moving in the left direction, the image associated with the demon entity should be `demonLeft.png`, and `demonRight.png` for when it's moving to the right).

In the `ATTACK` state, **both types** of demons have an attack range which is an invisible circle of radius **150 pixels**, which is centred at the **center of the demon's image** (not its top-left coordinate!). If the player enters a demon's attack range, it will shoot fire (rendered with `fire.png`) from one of four points (top-left, bottom-left, top-right or bottom-right of the demon's image). The point is chosen based on how close the player is to each point and this is explained later in the `Fire` section.

The fire will be rendered until the player leaves the demon's attack range. If the player **collides** with the fire, it will cause **10** damage to the player's health. A demon will always start in the `ATTACK` state.

(a) demonLeft.png

(b) fire.png

(c) demonRight.png

Figure 6: Demon images in ATTACK state

If a demon gets attacked by the player, it should go into an `INVINCIBLE` state for **3000 milliseconds**, rendered using the images below to visualize that they're invincible. In this state, any attack by the player will not cause damage to the demon. Once this time elapses, the demon goes back to its default behavior. Note that the `INVINCIBLE` state may overlap with the `ATTACK` state (i.e. the demon can still shoot fire while it's invincible).

(a) demonInvincibleLeft.png

(b) demonInvincibleRight.png

Figure 7: Demon images in INVINCIBLE state

Figure 8: Enemy health

A demon starts with **40** health points and its current health value is displayed as a percentage using the same calculation logic explained above for the player. This health bar is rendered on top of the demon'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 player. If a demon's health points reduce to 0 or less, they will die and disappear from the screen.

Navec

Navec is a special **aggressive** demon that features in Level 1. Navec shoots fire that deals **twice** the damage value of a normal demon and has a maximum health points value **twice** that of a normal demon. His attack range is also bigger than a normal demon (**200** pixels). The fire shot by Navec will be rendered with `navecFire.png`. All other behaviors of Navec, as well as its speed, are the same as a normal demon as explained above.

(a) navecLeft.png (b) navecRight.png (c) navecFire.png (d) navecInvincibleLeft.png (e) navecInvincibleRight.png

Figure 9: Navec images

Fire

Fire is shot by a demon or Navec as described earlier. The fire gets rendered if the player enters an enemy's range and will be rendered from **one of four points** of the demon's image (top-left, bottom-left, top-right or bottom-right). This is chosen based on how close the centre of the player's image is to the centre of the enemy's image (Remember that the given coordinates in the CSV are for the **top-left** of each image).

For our game, the logic is simplified as shown below (X-P stands for x-coordinate of the player's centre, Y-P for y-coordinate of the player's centre, X-E and Y-E are for the enemy's centre) :

- if X-P <= X-E and Y-P <= Y-E, fire should be drawn from **top-left**.
- if X-P <= X-E and Y-P >Y-E, fire should be drawn from **bottom-left**.
- if X-P >X-E and Y-P <= Y-E, fire should be drawn from **top-right**.
- if X-P >X-E and Y-P >Y-E, fire should be drawn from **bottom-right**.

The given fire image should be **rotated** when used from the bottom-left, top-right and bottom-right positions. **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**.

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).

Wall

Figure 10: Wall

A wall is a stationary object that features in Level 0, shown by `wall.png`. The player **shouldn't** be able to overlap with or move through the walls, i.e. the player must move the player around any areas on the level where blocks are being rendered. A wall has no damage points and cannot inflict damage on the player.

Tree

A tree has the image `tree.png` and is only featured in Level 1. Like the wall, the player **shouldn't** be able to overlap with or move through the trees, i.e. the player must move around any areas on the level where trees are being rendered. This rule also applies to any moving enemies that may collide with a tree (they should move in the opposite direction upon collision).

Figure 11: Tree

Sinkhole

A sinkhole is a stationary object that features in both levels and is shown by `sinkhole.png`. Each sinkhole has a damage points value of **30**. The sinkhole should behave similar to a wall and prevent the player from moving through it. If the player falls into (i.e. **collides**) with a sinkhole, it will inflict damage on the player (according to its damage points value). After collision, the sinkhole will disappear from the screen and the player should be able to move through the area it was placed at before.

If any moving enemy collides with a sinkhole, they should move in the opposite direction upon collision and the sinkhole will remain as it was.

Figure 12: Sinkhole

Timescale Controls

The movement speed of the enemies (demons and Navec) in Level 1 assumes a timescale of 0. 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 -3 or above 3. Each change in timescale makes the movement speed of the enemies decrease or increase by **50%**. The effect of a change in the timescale should be reflected immediately within the game.

Note: The player's speed is not affected by the timescale controls.

Entities Summary

Entity	**Image filenames**	**Health points**	**Movement speed**	**Damage points**	**Attack range**	**Collision effect on the player**
Player	faeLeft.png (IDLE & INVINCIBLE, moving left), faeRight.png(IDLE & INVINCIBLE, moving right), faeAttackLeft.png (ATTACK & INVINCIBLE, moving left), faeAttackRight.png (ATTACK & INVINCIBLE, moving right)	100	2	20	-	-
Demon (stationary & aggressive)	demonLeft.png (ATTACK, moving left), demonRight.png (ATTACK, moving right), demonInvincibleLeft.png (INVINCIBLE, moving left), demonInvincibleRight.png (INVINCIBLE, moving right)	40	Random value between 0.2 to 0.7 (for aggressive demons)	10	150	When the player enters its attack range, it shoots fire
Navec	navecLeft.png (ATTACK, moving left), navecRight.png (ATTACK, moving right), navecInvincibleLeft.png (INVINCIBLE, moving left), navecInvincibleRight.png (INVINCIBLE, moving right)	80	Random value between 0.2 to 0.7	20	200	When the player enters its attack range, it fires it shoots fire

Fire	fire.png (if shot by a demon), navecFire.png (if shot by Navec)	-	-	Depends on the demon type	-	Inflicts damage points of the demon that shot it, on the player
Wall	wall.png	-	-	-	-	Prevents the player from moving through it
Tree	tree.png	-	-	-	-	Prevents the player from moving through it
Sinkhole	sinkhole.png	-	-	30	-	Inflicts 30 damage points on the player and disappears

Log

- Every time an entity *inflicts damage* on another entity, 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. When fire inflicts damage on the player, the name of the entity that shot it (either Demon or Navec) should be used as shown below.

  For example:

  `Sinkhole inflicts 30 damage points on Fae. Fae's current health: 70/100`

  `Fae inflicts 20 damage points on Demon. Demon's current health: 20/40`

  `Navec inflicts 20 damage points on Fae. Fae's current health: 50/100`

- When a timescale control is used, a sentence detailing this is printed as follows:

  `Sped up, Speed: x`

```
Slowed down, Speed:  x
```

where `x` is the timescale value after the key has been pressed.

For example:

```
Sped up, Speed:  2
Slowed down, Speed:  0
Slowed down, Speed:  -3
```

Your Code

You must submit a class called `ShadowDimension` 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 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.
- Implement the player's behaviour/logic.
- Implement the tree's behaviour.
- Implement the demons' and Navec's behaviour/logic.
- Implement health bars and log printing.
- Implement the timescale controls.
- Implement level bounds.
- Implement win detection 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 `ShadowDimension` 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 introduce power-up items for Fae). 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 attribute, method and class 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
    ├── ShadowDimension.java
    └── other Java files
```

On 14/10/2022 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 14/10/2022 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 movement logic
- fix the player's collision behaviour
- refactored code for cleaner design

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

- fesjakhbdjl
- yeah easy finished the player

- 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. (Constants are allowed to be public or protected).
- Any constant should be defined as a 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 complete this form. Make sure you explain your situation with some supporting documentation such as a medical certificate, academic adjustment plan, wedding invitation, etc. You will receive an email saying if the extension was approved or if we need more information.

The project is due at **8:59pm sharp** on Wednesday 21/09/2022 (Project 2A) and on Friday 14/10/2022 (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** (*either* with or without an extension), please complete this form. For both forms, you need to be logged in using your **university** account. Please **do not** email any of the teaching team regarding extensions or late submissions (as you will be redirected to the online forms).

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 start screen and entity creation: **0.5 marks**
  - Correct implementation of the player's attack behaviour (including images and states): **2 marks**
  - Correct implementation of demon & Navec's behaviour (including images, states and health points rendering): **2 marks**
  - Correct implementation of fire behaviour (for both demons & Navec): **2 marks**
  - Correct implementation of stationary items' behaviour: **1 mark**
  - Correct implementation of timescale controls: **1 mark**
  - Correct implementation of level transition: **0.5 marks**
  - Correct implementation of bounds, win 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**