

THE UNIVERSITY OF MELBOURNE  
SCHOOL OF COMPUTING AND INFORMATION SYSTEMS  
SWEN20003 OBJECT ORIENTED SOFTWARE DEVELOPMENT

## ShadowTaxi

### Project 2, Semester 1, 2024

Released: Wednesday, 4<sup>th</sup> September 2024

Project 2A Due: Tuesday, 17<sup>th</sup> September 2024 at 4:00pm AEST

Project 2B Due: Friday, 11<sup>th</sup> October 2024 at 4:00pm 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 taxi simulation game called *ShadowTaxi* 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](#), including the use of [artificial intelligence](#).

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.

**Extensions & Special Considerations:** From Semester 2, 2024, The Faculty of Engineering and Information Technology has a new process in place for handling Extensions and Special Considerations, linked [here](#). Carefully read through the instructions about the same. This information is also published in Canvas -> Modules -> FEIT Extensions and Special Consideration. **Do not** send emails to the Subject Coordinator without reviewing this process first.

**Late Submissions:** If you submit late (**either** with or without an extension), please complete the Late form in the Projects module on Canvas. For the form, you need to be logged in using your university account. Please **do not** email any of the teaching team regarding late submissions. 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/methods (regardless of privacy)**. **You do not** need to show constructors, getters/setters, dependency, composition or aggregation relationships. Bagel classes can be shown similar to primitive data types. If you so choose, you may show the relationships between classes on a separate page to the

details (attributes and methods) of the classes 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

*“You are a taxi **driver** stuck on an endless road, attempting to survive in this current economic crisis. Move the driver to control a **taxi** in the lanes, pick up **passengers** and drop off the passengers at their **trip end flags** to earn money. Each passenger has a priority, which can increase the driver’s earnings. Collect the **coins**, to increase the passenger’s priority. If you stop past the trip end flag, you lose money! To make things harder, there are **enemy cars and other cars**, who will collide and cause damage to the driver, passenger and the taxi. If the driver or a passenger’s health decreases to below 0, you lose! Can you beat the target score before the time elapses to complete the game?*

The game is an extension to Project 1 and includes the same functionality (*with some slight modifications which are explained later*). The player can press the arrow keys to control a driver, that can drive a taxi. The player has to press the arrow keys to move the taxi left, right or up. The taxi can pick up and accommodate one passenger at a time, by stopping close to them. The player has to drop the passenger off at their respective trip end flag - **stopping past this flag, incurs a penalty on the earnings of that trip**. The taxi can collect coins by colliding with them - collecting coins will increase the priority of the current passenger once. Once dropped off, the trip earnings are added to the total score.

The driver, passengers and the taxi have **health values**. The game also features other cars and enemy cars, that are generated randomly and move independently. If they collide with the taxi, *damage* is inflicted on the taxi. If the taxi’s health reduces to below 0, it becomes permanently damaged and cannot be used. The **user has to move the driver to a new taxi, that will be randomly generated at this stage**. If other cars or enemy cars collide with the driver or passenger when walking, **damage is inflicted on the driver or passenger**. To win, the player needs to beat the target score of 500. If the game runs for more than 15,000 frames, the game ends in a loss. If the driver or a passenger’s health reduces to below 0, it is a loss. Further, if a **new taxi goes off-screen without the driver getting in, this is a loss** (*this will be explained in detail later*). At the end, the current top 5 scores are shown on screen.

## 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!

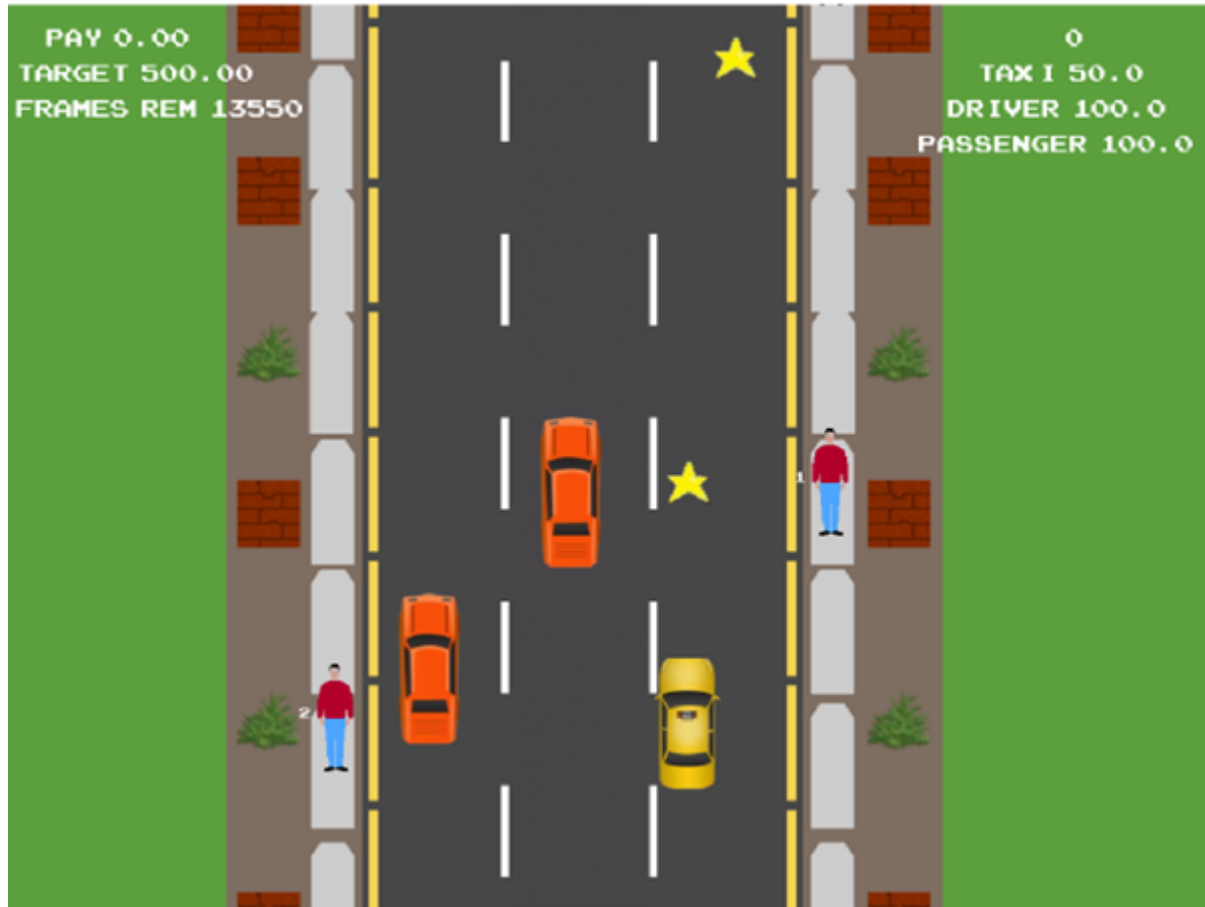


Figure 1: Completed in-game Project 1 screenshot

Note : In the above screenshot, the taxi has not picked up a passenger yet. The actual positions of the entities in the world file we provide you may not be the same as the screenshots shown in this document.

## The Game Engine

The **Basic Academic Game Engine Library** (Bagel) is a game engine that you will use to develop your game. You can find the offline documentation for Bagel on Canvas under the Projects module.

### *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 `ShadowTaxi` is called. It is in this method that you are expected to update the state of the game.

Your code will be marked on **120Hz screens**. The refresh rate is typically 120 times per second (Hz) but some devices might have a lower rate of 60Hz. 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 120Hz. For your convenience, when writing and testing your code, you **may** change these values to make your game playable. If you do change the values, **remember** to change them back to the original specification values before submitting.

## The Game Elements

The default window size should be  $1024 * 768$  pixels. The game consists of a few screens and other different game elements. Below is an outline of these elements you will need to implement.

### *Home Screen*

This is the first screen rendered when the game is started. The background (`backgroundHome.png`) should be rendered on the screen and completely fill up your window during this screen. This has already been implemented for you in the skeleton package.

A title message that reads **SHADOW TAXI** should be rendered with the font provided in the `res` folder (`FS08BITR.ttf`), with a font size 64. The bottom left of the message is as follows: the x-coordinate should be calculated in such a way that the message appears centered horizontally and the y-coordinate should be at 384 pixels.

Below this, an instruction message that reads **PRESS ENTER** should be rendered in the provided font, in size 32. The x-coordinate of the bottom left of the message should be calculated in such a way that the message appears centered horizontally, and the y-coordinate should be at 500 pixels. When the Enter key is pressed, the game changes to the *Player Information Screen*.

**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()` and `Font.getWidth()` methods.

### *Player Information Screen*

The background (`backgroundPlayerInfo.png`) should be rendered on the screen and completely fill up your window during this screen. All messages on this screen are in the provided font, in size 24. The x-coordinate of the bottom left of all the messages should be calculated in such a way that the messages look centered horizontally.

An instruction message that reads `ENTER YOUR NAME` should be rendered at the top. The y-coordinate of the bottom left of this message should be at 200 pixels.

When the user types their name, the entered letters needs to be rendered in **black** in the white box on this screen. When the user presses the Backspace or Delete key, the last letter of the currently rendered name needs to be **removed**. The y-coordinate of the bottom left of this name should be at 380 pixels. You may add any validation on the input name that you want. **Hint:** The `DrawOptions` class in Bagel will help you change the colour of the text. There is also a method given to you in the skeleton that will convert a key press into the corresponding `String` value.

Additionally, an instruction message consisting of 2 lines:

`PRESS ENTER TO START`  
`USE ARROW KEYS TO MOVE`

should be rendered **below** the white box. The y-coordinate of the bottom left of this message should be at 500 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). When the Enter key is pressed, the game changes to the *Game Play Screen*.

### *Game Play Screen*

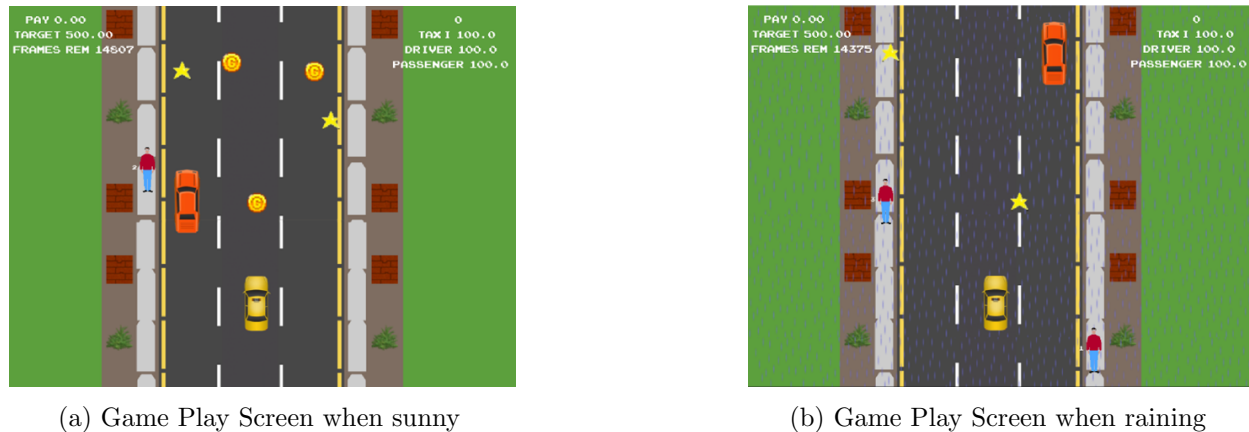
The game now features two types of **weather conditions**, either sunny or raining. A weather condition will be **active from a start frame to an end frame**. A `gameWeather.csv` file has been given to you in the skeleton package, containing weather conditions with corresponding start and end frames. Depending on the current weather condition, the corresponding image must be rendered for the background, i.e. `background.png` for sunny and `backgroundRain.png` for raining conditions.

Similar to Project 1, this screen features two backgrounds stacked **vertically** in the y-axis, to create a scrolling effect during game play. The starting centre coordinates of the first background is (512, 384) and for the second, (512, -384). The second background starts above the first, initially off-screen (*these values are in fact, half the default window width and window height*).

When the player presses the up arrow key, both backgrounds move vertically down by a speed of **5 pixels per frame**. If the current y-coordinate of the centre of one background becomes greater than or equal to 1152, the y-coordinate is set to the (y-coordinate of other background) - `Window.getHeight()`. In other words, the background leaving past the bottom of the window,

moves to above the window (*if done correctly, this will look like a scrolling effect and the taxi moves up on an endless road*).

The game entities and the actual game play will happen during this screen. This behaviour is explained in detail later in the *Game Entities* section.



(a) Game Play Screen when sunny

(b) Game Play Screen when raining

Figure 2: Sample Game Play Screens (note that the taxi has not picked up a passenger yet)

### Game End Screen

When the player wins or loses, the player's name and score (separated by a comma) needs to be written into the `res/scores.csv` file. A method to write to the comma-separated value (CSV) file is given to you in the skeleton package.

For a win or a loss, the game changes to this screen. The background (`backgroundEnd.png`) should be rendered on the screen and completely fill up your window during this screen. All messages on this screen are in the provided font. The x-coordinate of the bottom left of all the messages should be centered horizontally.

On this screen, a message that reads "TOP 5 SCORES -" should be rendered at the **top**, with a font size of 20. The y-coordinate of the bottom left of this message should be at 200 pixels.

Below this, the **top five** scores currently stored in the CSV file, have to be rendered. If there are less than five, all scores will be shown. Each line will be in the format of "*i* - *j*", where *i* is the player name and *j* is the score, given to 2 decimal places. The bottom left of each line should be calculated as follows: the x-coordinate will be centered horizontally and the y-coordinate increases by 40 from 200 pixels (i.e the bottom of the scoring message described above).

A method to read the CSV file and return the Strings, is given to you in the skeleton package (*you will need to choose the top 5 scores before rendering however*).

If the player's score is greater than or equal to 500, this is considered as a win. A message should be rendered, that consists of 2 lines:

CONGRATULATIONS, YOU WON!  
PRESS SPACE TO CONTINUE

If the game runs for more than 15,000 frames without a win happening, this is considered as a loss and the game ends. If the driver's health or a passenger's health is less than or equal to 0, this is also a loss. If a newly generated taxi goes off-screen without the driver getting in, this is a loss (*both of the last two situations will be explained in detail later*). In these cases, the message should be rendered as:

GAME OVER, YOU LOST!  
PRESS SPACE TO CONTINUE

The win/loss message should be rendered, in the font provided, in size 24. The y-coordinate should be at 500 pixels.

If the Space key is pressed, the game returns to the Home Screen and allows the player to play again. 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.



Figure 3: Game Screens

## Properties File

The key values of the game are listed in two **properties files** which are given in the skeleton package. The message coordinates, image filenames and other values are given in the **app.properties** file. The message strings are given in the **message\_en.properties** file. These files **shouldn't** be edited (unless you need to adjust values for any frame rate issues). All properties given in the files should be read-in and **not hard-coded**.

To read a value from one of these properties, a **Properties object** must be created. The **getProperty** method can be called on this object with the required value given as the parameter. For your reference, the skeleton package contains an example of how to read the background image filename, window width and window height values.

## World File

The entities will be defined in a **world file**, describing the type and their position in the window. The world file is located at **res/gameObjects.csv**. A world file is a comma-separated value (CSV) file with rows in one of the following formats:

TAXI or COIN or DRIVER or INVINCIBLE\_POWER, x-coordinate, y-coordinate



or

```
PASSENGER, x-coordinate, y-coordinate, priority, end x-coordinate, y-distance, has-umbrella
```

An example of a world file is shown below:

```
TAXI,500,600
DRIVER,500,600
COIN,450,-316
COIN,309,-2167
PASSENGER,280,-800,3,280,500,0
PASSENGER,280,-1500,1,700,800,1
INVINCIBLE_POWER,625,-7981
```

The given (x, y) coordinates refer to the centre of each image and these coordinates should be used to draw each image. For each passenger, the end x-coordinate is for the end of their trip and the y-distance is the distance travelled vertically during the trip. If the has-umbrella value is 1, the passenger is carrying an umbrella.

You must actually load the file—copying and pasting the data, for example, is not allowed. Marking will be conducted on a hidden **different** CSV file of the same format. You have been given a method to read a CSV file in the skeleton package. **Note:** The values in the example may not be the same as the ones given to you and the order of the entities may change. You can assume that there will always be at least one of each for all the entities. You may also assume that there will be only one driver.

## The Game Entities

The following game entities have an associated image and a starting location (x, y). Remember that all images are drawn from the centre of the image using these coordinates. *Each entity has behaviour that interacts with the other entities, so make sure you read this section fully, before you ask any questions on Ed Discussions!*

### Taxi



Figure 4: taxi.png

The taxi is represented by the image `taxi.png`, shown on the left. It has a radius value of 32. In Project 2, the taxi has two types of movement - with the user's input (when the **driver** is in the taxi) or on its own (without the driver). When the driver is not in the taxi, it will move vertically downwards at a speed of **5 pixels per frame** when the user's *up* arrow key is pressed or held down.

When the driver is in the taxi, it can move on screen in one of three directions (left, right and up) when the corresponding arrow key is pressed. However, for our ease of implementation, we assume the taxi can **only** move horizontally and remains stationary in the



vertical direction (i.e. the other entities will be moving in relation to the player's arrow keys pressed - *this is explained for each entity later*). Hence, when the player's *right* arrow key is pressed or held down, the taxi will move to the *right* by **1 pixel per frame**. When the player's *left* arrow key is pressed or held down, the taxi will move to the *left* by the same speed.

The taxi has a **health value of 100**. This health value is rendered in the format of "TAXI k", where k is the health value. The bottom left corner of this message should be located at (825, 65) and the font size should be 20. During collisions with other cars and enemy cars, the taxi can inflict damage (with a **damage points value of 100**). When colliding with other cars, enemy cars and fireballs, the taxi can also take damage based on the other entity's damage points. A collision happens if the distance between the taxi's current coordinates and the other entities' coordinates is **less than the sum of the radius values of the two colliding entities**.

When the taxi takes damage, **smoke** is rendered using **smoke.png** for 20 frames. It will be rendered at the current coordinates of the taxi and will move vertically *down* by **5 pixels per frame**, when the player's *up* arrow key is pressed or held down. After a collision takes place, the taxi has a **collision timeout of 200 frames**. During this time, the taxi **cannot** take damage from a collision with another car again - *the images may overlap and you can choose the order of rendering as you wish*.

For the **first 10 frames** of this timeout, the taxi and the other collided car will be moved away from each other. If the y-coordinate of the taxi is lower than the y-coordinate of the other car, the taxi is considered as the entity vertically on top, and the other car is the entity vertically bottom. Every frame (for a duration of total 10 frames), the y-coordinate of the entity on top, will decrease by 1 and the y-coordinate of the entity bottom, will increase by 1 (*this will look like the higher collided car moving up on screen and the lower car moving down*).

If the taxi collides with an **invincible power**, it becomes invincible. This effect lasts for **1000 frames** and the taxi won't receive any collision damage during this time. The taxi can **still give damage to other entities** during collisions however.

If the taxi's health is less than or equal to 0, the taxi is permanently damaged and its image shown will change to **taxiDamaged.png**. The driver and passengers currently **in the taxi will be ejected** (*this is explained later for each entity*). A **fire** will also be rendered using **fire.png** for 20 frames. Once again, it will be rendered at the taxi's current coordinates and will move vertically down, as described above for the smoke. If other **entities collide with a permanently damaged taxi, nothing will happen and the entity will pass through**.

Once the taxi is permanently damaged and there is no active taxi on screen, a new taxi will be created **randomly**. Its coordinates are to be calculated as follows: the x-coordinate should be randomly chosen from the **centre values of either the first road lane 1 (360) or the third lane (620)** and the y-coordinate should be randomly chosen in the range of **(200, 400)**. If there is already an entity at the randomly chosen coordinates, you can choose the order of rendering as you wish.

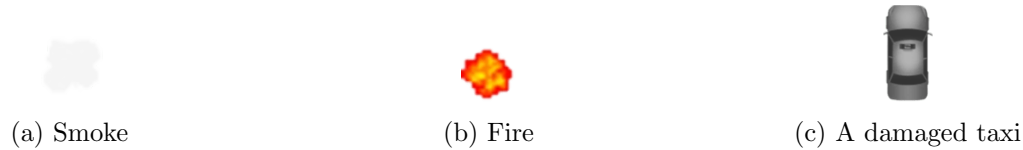


Figure 5: Images related to the taxi's health

Similar to Project 1, a taxi with a driver inside can pick up a passenger. When the taxi picks up a **passenger**, the taxi will have an associated **trip**. The taxi also has an associated total **score**. Once a trip is complete, the earnings of that trip is added to the total score. The trip earnings calculation is explained later in the *Trip* section. The total score is rendered in the top left corner of the screen in the format of "PAY k" where k is the current total score, shown to 2 decimal places. The bottom left corner of this message should be located at (35, 35) and the font size should be 20.



Figure 6: Taxi's score, target &amp; frames rendering

The target score of 500.00, should also be rendered on screen, in the format of "TARGET k", where k is the target score. The bottom left corner of this message should be located at (10, 65) and the font size should be 20.

The remaining number of frames is shown on screen in the format of "FRAMES REM k", where k is the number of frames remaining. The bottom left corner of this message should be located at (10, 95) and the font size is again 20.

## Driver

A driver is an entity shown by `driver.png` and starts the game in the taxi. When the driver is in a taxi, they both move together. This movement was described earlier in the *Taxi* section. If the taxi becomes damaged, the driver will be ejected and the coordinates will be changed to (x - 50, y), where x and y are the driver's coordinates at time of ejection.



Figure 7: driver.png

The driver can get into another taxi once a new one is created. During this stage, the driver can move at a speed of one pixel per frame in one of four directions (up, down, left right) based on the user's key press. The driver is considered as "in the taxi" if the Euclidean distance between the coordinates and the taxi's current coordinates is less than or equal to 10. If this is true, the driver's image will not be rendered - only the taxi image will be shown. The driver's coordinates will be updated whenever the taxi's coordinates are updated.

If the driver walks vertically upwards past the newly generated taxi and the y-coordinate of the taxi becomes greater than or equal to 768, this is considered as a game loss. On screen, this will look like the driver moving upwards and the taxi moving out of the window from the bottom.



Figure 8: blood.png

The driver has a health value of 100. This health value is rendered in the format of "DRIVER k", where k is the health value. The bottom left corner of this message should be located at (800, 95) and the font size

should be 20. When walking, if the driver collides with cars, enemy cars or fireballs, the driver will take damage based on the other entity's damage points. If the driver's health reduces to less than or equal to zero, **blood** is rendered using **blood.png** for 20 frames. It will be rendered at the collision coordinates of the driver and will move vertically *down* by **5 pixels per frame**, when the player's *up* arrow key is pressed or held down.

If the driver collides with an **invincible power**, the driver will be invincible and won't receive any collision damage. This effect will **transfer onto the new taxi that the driver gets into**. If the driver **collides with a coin**, the **priority** value of the passenger with the driver (if any) will **decrease** and the effect will **transfer onto the new taxi**.

Similar to the taxi, if the **driver** collides with a car, enemy car or fireball, the driver has a **collision timeout of 200 frames**. For the first 10 frames, the driver will be moved away from the other collided entity using the same logic as described in the *Taxi* section. The only difference is that **both the x and y coordinates will increase or decrease respectively by 2**.

## Passenger



Figure 9: passenger.png

A passenger is an entity shown by **passenger.png**. When the player's *up* arrow key is pressed or held down, the passenger will move vertically *down* by **5 pixels per frame**.

A passenger has an associated **priority** value of 1, 2 or 3, where 1 is the highest priority. The higher the priority, the higher the trip earnings - *this is explained in detail later*. The priority is rendered **only** when the passenger has not been picked up yet. It is shown as an integer, at the coordinates of  $(x - 30, y)$  where  $x$  and  $y$  are the current coordinates of the passenger. The font size should be 12. If the weather condition is raining and the passenger doesn't have an umbrella, their priority is set to 1.

The **expected earnings** for each passenger before being picked up by the taxi, will be rendered on screen. It is shown to 1 decimal place, at the coordinates of  $(x - 100, y)$  where  $x$  and  $y$  are the current coordinates of the passenger. The font size is 12. The details of how to calculate the earnings are explained later in the *Trip* section.

For an **active taxi to pick up a passenger**, **three** conditions must be fulfilled -

- the taxi must be stopped, i.e. not moving in the horizontal or vertical direction.
- the taxi has no current passenger.
- the taxi must be adjacent to the passenger. This is calculated by calculating the **Euclidean distance** between the two  $(x, y)$  coordinates of the taxi and passenger. If this current distance is **less than or equal to 100**, this is considered as adjacent.

If all 3 conditions are true, the passenger will move towards the taxi. **The passenger's  $(x, y)$  coordinates will increase or decrease respectively, by 1 pixel per frame, until they are equal to the taxi's current coordinates -** *note that this is different to the Project 1 implementation!*

Once the passenger is picked up, the trip will **commence** and the taxi will move based on the

user's input as described in the *Taxi* section. During this movement, the passenger's image will **not** be rendered - only the taxi image will be shown. However, the passenger's coordinates will need to be updated whenever the taxi's coordinates are updated.

If the taxi gets damaged during a trip, the passenger will be ejected and the coordinates will be changed to  $(x - 100, y)$ , where  $x$  and  $y$  are the passenger's coordinates at time of ejection. During this stage, the passenger will **follow** the driver's movements, which are being controlled by the user's key press (i.e. the passenger's coordinates will be increased or decreased by the same values as the driver's).

The passenger has a health value of 100. This health value is rendered in the format of "PASSENGER  $k$ ", where  $k$  is the current health value of the passenger on this trip. If there is no current trip, the value shown will be the minimum health value of all the passengers. The bottom left corner of this message should be located at (775, 125) and the font size should be 20. When walking, if the passenger collides with cars, enemy cars or fireballs, the passenger will take damage based on the other entity's damage points. If the passenger's health reduces to less than or equal to zero, **blood** will be rendered, as described in the *Driver* section. The passenger **cannot** pick up coins or invincible powers.

The passenger also has a collision timeout of 200 frames. For the first 10 frames, the passenger will be moved away from the other collided entity using the same logic as described in the *Driver* section.

When the trip is complete, the passenger will leave the taxi. This happens when the first condition and **either** of the second or third conditions are met -

- the taxi must be stopped, i.e. not moving in the horizontal or vertical direction.
- the taxi's y-coordinate must be less than or equal to the y-coordinate of the **trip end flag** (i.e. the taxi has just moved above the end flag on screen).

**OR**

- the distance between the taxi's coordinates and the trip end flag's coordinates, is less than or equal to the radius of the trip end flag, which has a value of 80 (i.e. the taxi has stopped in the radius of the end flag).

If this is satisfied, the passenger will leave the taxi and move towards the trip end flag. The passenger's  $(x, y)$  coordinates will increase or decrease respectively, by **1 pixel per frame** again, until the coordinates match the trip end flag's coordinates (*for ease, you may consider this motion as two separate movements, in the  $x$  and  $y$  directions*). Once the passenger's coordinates are **equal** to the trip end flag's coordinates, the passenger's image will stay at that position.

## Car

A car is an entity that can move completely on its own. It is represented by one of two images shown below that is randomly chosen at creation. A car will move vertically upwards at a speed in the range of **2 - 5 pixels per frame**, that is randomly fixed at creation.

A car is created **randomly** using the following process - a random integer in the range of (1, 1000) is generated and if this number is divisible by 200, a car is created (a method that performs this functionality is given to you in the skeleton package). The car's x-coordinate is randomly chosen to be the center of one of the three road lanes (360, 480, 620). The y-coordinate is randomly chosen to be either -50 or 768.



(a) otherCar-1.png



(b) otherCar-2.png

Figure 10: Car images

A car has a **health value** of 100. During collisions with any other entity, it can give damage with a points value of 50. When colliding with a taxi, other car, enemy car or fireball, the car can also take damage based on the other entity's damage points. When taking damage, **smoke** will be rendered, similar to the taxi.

Similar to the taxi, it has a **collision timeout** of 200 frames. For the first 10 frames, the car will be moved away from the other collided entity using the same logic as described in the *Taxi* section. Once the collision timeout ends, a new random speed is chosen for the car and it will move vertically upwards again.

If the car collides with a coin or invincible power, the car will continue moving and nothing will happen. If the car's **health** is less than or equal to 0, a **fire** is rendered similar to the taxi and the car's image will stop being rendered.

### Enemy Car

An enemy car is a type of car shown by **enemyCar.png**. It has all the functionality described in the *Car* section. It has a **health** points value of 100 and a **damage** points value of 50. An enemy car is also created randomly - the only difference is that the **number used to check divisibility** is 400.



(a) enemyCar.png



(b) fireball.png

Figure 11: Enemy car and fireball images

An enemy car can also **shoot fireballs**. A fireball is **rendered randomly** at the current coordinates of the enemy car, using the same process used for car creation with a divisibility number of 300. A **fireball will move vertically upwards** at a speed of **7 pixels per frame**. If it collides with a passenger, driver, taxi or other car, it will **inflict a damage points value of 20**. If the fireball collides with an entity or reaches the top of the screen, it will stop being rendered.

### Trip End Flag

A trip end flag is an entity shown by `tripEndFlag.png` and has a radius value of 80. When the player's *up* arrow key is pressed or held down, the flag will move vertically *down* by **5 pixels per frame**. The flag's x-coordinate is the passenger's end x-coordinate, given in the world file. The flag's y-coordinate needs to be calculated using the passenger's starting y-coordinate and y-distance to travel during the trip, both given in the world file (*remember that y-coordinates decrease when moving vertically up on screen!*)



Figure 12: tripEnd-Flag.png

Once a passenger has been picked up (i.e. a trip has started), the flag will be rendered on screen. Once the passenger reaches the trip end flag, it is no longer rendered on screen.

### Invincible Power



Figure 13: invincible-Power.png

The invincible power is an entity shown by `invinciblePower.png`, and has a **radius value of 20**. When the player's *up* arrow key is pressed or held down, the flag will move vertically *down* by **5 pixels per frame**.

When a **taxi or the driver collides** with the power, the **entity** becomes **invincible** to any damage from collisions for **1000 frames**.

### Coin

A coin is an entity shown by `coin.png` and has a radius value of 20. When the player's *up* arrow key is pressed or held down, the coin will move vertically *down* by **5 pixels per frame**.



Figure 14: coin.png

When a coin **collides** with the taxi, the taxi gains the coin power for 500 frames. If the taxi either has a passenger before collision or picks up a passenger whilst the coin power is active, the passenger's current priority value is **decreased** by 1 (i.e. passenger gains higher priority). Once collided with, the coin is no longer rendered on screen.

A taxi can collide with multiple coins during a trip but the passenger's priority will increase **only once**. If the passenger's current priority value is already 1, nothing will happen. The number of frames that the (**most recently collided with**) coin has been active for is rendered on the screen as an integer. The bottom left corner of this message should be located at (900, 35) and the font size should be 20. *This means that every time a new coin is collided with, the rendered frame count "will look like" it has reset.*

**Note** that the **driver** can **also collide** and **gain** the **power of a coin**, as described in the *Driver* section.

### Trip

A trip is not a single entity shown on screen - it is included here to help explain the earnings calculation and message rendering. The earnings are calculated when a trip is **complete** (based

on the conditions described in the *Passenger* section). The earnings are calculated as the **sum** of two fees - distance fee and priority fee.

The **distance fee** is calculated by multiplying the y-distance travelled with the rate per y-distance of 0.1. The **priority fee** is equal to the priority rate based on the passenger's current priority value, as given in the below table - *note that this is different to the Project 1 implementation!*

Priority	Rate
1	50
2	20
3	10

For example :-

- A passenger of initial priority 3, has travelled for 50 pixels. The distance fee would be  $50 \times 0.1 = 5$ . The priority fee would be 10. The total earnings would be  $5 + 10 = 15$ .
- A passenger of initial priority 3, has travelled for 150 pixels and the taxi has collided with one coin. The distance fee would be  $150 \times 0.1 = 15$ . The priority fee would be 20. The total earnings would be  $15 + 20 = 35$ .

A penalty will be applied if **three** conditions are met -

- the taxi is **stopped**, i.e. not moving in the horizontal or vertical direction.
- the taxi's y-coordinate must be **less than** the y-coordinate of the trip end flag (i.e. the taxi has moved beyond the trip end flag on screen).
- the distance between the taxi's coordinates and the trip end flag's coordinates, is **greater than** the radius of the trip end flag.

The penalty is calculated by multiplying the penalty rate of 0.05 with the distance between the trip end flag's y-coordinate and the taxi's y-coordinate when it stopped (when the trip was completed). The penalty will be **subtracted** from the trip earnings. If the earnings becomes negative, it will be set to 0. (*What this implies is that, a "perfect" trip ending will be with the taxi stopping as close as possible to the trip end flag*).

As mentioned earlier in the *Passenger* section, the expected trip earnings are rendered on screen next to each passenger before being picked up - this **will not** include any penalties as the trip has not yet started.

Once the trip **commences**, the trip details are shown on screen in the bottom left. The font sizes for all of these details are 20 and the x-coordinate of bottom left corner of each message should be located at 35. First, a message that reads "CURRENT TRIP -" is rendered with the bottom left y-coordinate at 650. 30 pixels below this, the expected earnings are rendered in the format of "EXP FEE k", where k is the current expected earnings, given to 1 decimal place. 60 pixels below the first trip message, the passenger's current priority value is rendered in the format of "PRIORITY k", where k is the current priority value, as an integer. *Remember that current priority values (and expected earnings) can change if a coin has been collided with!*

Once the trip is **complete**, the first trip message is replaced by one that reads "LAST TRIP -" in

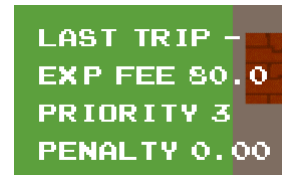


the same position as before. The penalty value is shown 90 pixels below the first trip message, in the format of "PENALTY *k*", where *k* is the penalty value, shown to 2 decimal places. The penalty value is shown even if there is no penalty for that trip. The trip earnings (with any possible penalties) are added to the total score, which will be updated. These last trip details will continue to be rendered until a new trip commences.

A green rectangular box with a dark green border containing white text. The text is arranged in four lines: "CURRENT TRIP -", "EXP FEE 110.0", and "PRIORITY 2".

```
CURRENT TRIP -  
EXP FEE 110.0  
PRIORITY 2
```

(a) Current Trip

A green rectangular box with a dark green border containing white text. The text is arranged in four lines: "LAST TRIP -", "EXP FEE 80.0", "PRIORITY 3", and "PENALTY 0.00".

```
LAST TRIP -  
EXP FEE 80.0  
PRIORITY 3  
PENALTY 0.00
```

(b) Last Trip

Figure 15: Sample Trip Detail Rendering

## Your Code

You must submit a class called `ShadowTaxi` 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. The purpose of this project is to evaluate how well you use OOSD concepts, specifically classes. 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):

- Draw the game play screen with new weather conditions
- Read the world file, then draw the driver, one enemy car and one other car on screen
- Implement movement logic for the 3 entities above
- Implement image rendering, random creation and movement for all the entities in the world file
- Implement the collision logic for the different combinations of entities
- Implement the invincibility power collision, effect on collisions
- Implement new loss detection

## 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 `ShadowTaxi`, `MiscUtils` and `IUtils` classes 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 a new level with different entites or powers). 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 lecturers and tutors. The winning three will have their games showcased on Ed, 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](mailto: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
│   ├── ShadowTaxi.java
│   └── other Java files
└── pom.xml
```

On 11<sup>th</sup> October 2024 at 4:00pm, 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 11<sup>th</sup> October 2024 4:00pm 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 other car's collision behaviour
- refactored code for cleaner design

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

- fesjakhbdjl
- yeah easy finished the enemy cars
- 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.
- 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

From Semester 2, 2024, The Faculty of Engineering and Information Technology has a new process in place for handling Extensions and Special Considerations, linked [here](#). Carefully read through the instructions about the same. This information is also published in Canvas -> Modules -> FEIT Extensions and Special Consideration. **Do not** send emails to the Subject Coordinator without reviewing this process first.

The project is due at **4:00pm sharp** on Tuesday 17<sup>th</sup> September 2024 (Project 2A) and on Friday 11<sup>th</sup> October 2024 (Project 2B). Any submissions received past this time (from 4:00pm 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 the Late form in the Projects module on Canvas. For the form, you need to be logged in using your university account. Please **do not** email any of the teaching team regarding late submissions.

## 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 **11 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: **2.5 marks**
  - Appropriate use of inheritance, interfaces and abstract classes/methods: **2.5 marks**
- Project 2B (feature implementation) is worth **5 marks**.
  - Correct implementation of driver behaviour (including image, movement, health and collisions): **1 mark**
  - Correct implementation of taxi behaviour (including image, movement, health and collisions): **1 mark**
  - Correct implementation of enemy and other car behaviour (including image, movement, health and collisions): **1.5 marks**
  - Correct implementation of invincibility power's behaviour (including image and effects): **1 mark**
  - Correct implementation of screens: **0.5 marks**
- Project 2B (coding style) is worth **6 marks**.
  - Delegation: breaking the code down into appropriate classes and splitting up responsibilities appropriately: **1 mark**
  - Use of methods: avoiding repeated code and overly complex methods: **1 mark**
  - Cohesion: classes are complete units that contain all their data and methods: **1 mark**
  - Coupling: interactions between classes are not overly complex: **1 mark**
  - General code style: appropriate visibility modifiers (use of public & private), avoiding magic numbers, consistent formatting, sufficient commenting: **1 mark**
  - Use of Javadoc documentation: **1 mark**