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

ShadowDance

Project 2, Semester 2, 2023
Released: Friday, 8th September 2023 at 4:30pm AEST
Project 2A Due: Wednesday, 20th September 2023 at 4:30pm AEST
Project 2B Due: Friday, 13th October 2023 at 4:30pm 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 music arcade game called *ShadowDance* 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 & late submissions: If you need an extension for the project, please complete the Extension form in the **Projects** module on Canvas. 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 the Late form in the **Projects** module on Canvas. 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. 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

*'The aim is simple : the **player** has to hit the corresponding musical **notes** that appear on screen in different **lanes** on time to score **points**. To win each level, you need to beat a target score. The second level features a **special lane** that has **special notes** such as **bomb**, **speed up**, **slow down** and **double score**. The third level includes **enemies** who try to steal notes from the lanes and a **guardian** who will shoot **projectiles** at these enemies when a key is pressed. Can you beat the target scores and win the game?"*

The game features three levels : *Level 1*, *Level 2* and *Level 3*. In Level 1, the notes will descend from the top vertically in the 4 lanes. The player has to press the corresponding arrow key when the note overlaps with the stationary note symbol at the bottom. The accuracy of how close the note was to the stationary note when the key was pressed, will determine the points given. There will be *hold notes* that require the player to hold down the key. To finish the level, the player needs to beat the target score of 150 when all the notes have fallen. If the player's score is lower, the game ends. You have already implemented Level 1 in Project 1 (the only change required is to the start/end screens which is explained later).

Level 2 features the same game-play as above but the player now has to deal with additional features such as a *special lane* that has *special notes*. Similar to a normal note, if the corresponding key is pressed for the special note, its effect is applied (this is explained in detail later). To win the level, the player must beat a score of 400.

Level 3 is the final level. It includes all of the above as well as extra features such as *enemies* and a *guardian*. An enemy moves horizontally and it will steal notes from nearby lanes by colliding with them. The guardian will shoot projectiles at the nearest enemy when the corresponding key is pressed. To win the level, the player must beat a score of 350.

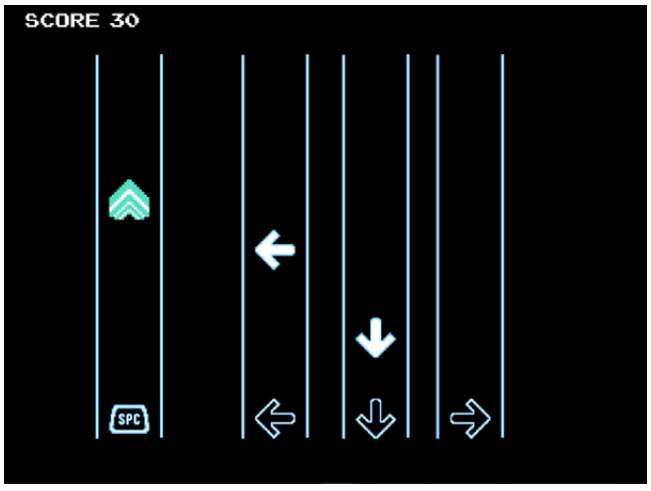
Note that the game does not need to be played progressively. You can choose which level to play from the start screen and also at the end of each level.

An Important Note

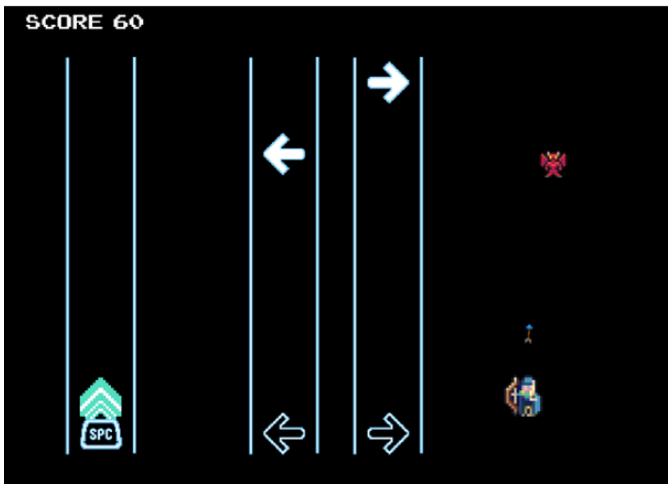
Before you attempt the project or ask any questions about it on the discussion forum, it is crucial that you read through this entire document thoroughly and carefully. We've covered every detail below as best we can without making the document longer than it needs to be. Thus, if there is any detail 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: Start Screen Screenshot



(a) Completed Level 2 Screenshot



(b) Completed Level 3 Screenshot

Figure 2: Level Screenshots

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

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 `ShadowDance` 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 now 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 (these changes are explained later). 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 120Hz screens.**

The Levels

Our game will have three levels, each with elements to implement that are described below.

Window and Background

The background (`background.png`) should be rendered on the screen to completely fill up your window throughout the game (for the start screen and all the levels). The default window size should be `1024 * 768` pixels. The background has already been implemented for you in the skeleton package.

Start Screen

Each level has the same start screen. The screen has a title message that reads `SHADOW DANCE` should be rendered in the font provided in `res` folder (`FSO8BITR.ttf`), in size `64`. The bottom left corner of this message should be located at `(220, 250)`.

Additionally, an instruction message consisting of 4 lines:

```
SELECT LEVELS WITH
NUMBER KEYS

1 2 3
```

should be rendered **below** the title message, in the font provided, in size `24`. The bottom left of the first line in the message should be calculated as follows: the x-coordinate should be increased by `100` pixels and the y-coordinate should be increased by `190` pixels.

There must be **adequate spacing** between the 4 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.

The player chooses which level to play by pressing the corresponding key (1, 2 or 3). Once the level is over, the player will be re-directed back to the start screen. To help when testing your game, you can allow the player to pause the game whilst in a level (i.e. everything in the window will stop moving) by pressing the `Tab` key (this is **not assessed** but will help you when coding!)

World File

The lanes and the notes will be defined in a **world file**, describing the type and their position or time of appearance in the window. The world files for each level are `level1.csv`, `level2.csv` and `level3.csv` correspondingly. (For 60Hz screens, use the corresponding '-60" file, i.e. for Level 1 use `level1-60.csv`). A world file is a comma-separated value (CSV) file with rows in one of the following formats:

```
Lane, type of lane, x-coordinate
            (or)
Type of lane, type of note, frame-number
```

An example of a world file:

```
Lane,Special,200
Lane,Right,742
Down,Normal,776
Down,Hold,1357
Special,DoubleScore,2370
```

The **type of lane** refers to either the arrow key which it corresponds to or if it is a special lane, for example: the first entry in the above example is for the lane corresponding to the special lane. The **type of note** refers to whether the note is normal, a hold note or a special note, and the **frame number** is the frame in which the note starts appearing on screen. For example, the last entry in the above example refers to a special note of the double score type that appears from the 2370th frame onwards in the special lane.

You must actually load the files—copying and pasting the data, for example, is not allowed. You have been provided with extra world files to test on (`test1.csv`, `test2.csv`, `test3.csv`). Marking will be conducted on a hidden **different** CSV file of the same format. **Note:** You can assume that there will always be **only one** special lane in both Levels 2 and 3, and that there will be a **maximum** of four normal lanes in each level. The number of notes in the CSV may vary however.

End Screen

Each level has the same end screen. The end screen has **two** messages - a win/loss message and an instruction message.

When all the notes in the CSV file have fallen, if the player's score is **higher** than the target score for that level, this is considered as a win. Hence, the first message is a winning message that reads `CLEAR!` The x-coordinate of this message should be centered horizontally and the y-coordinate is `300`. The font size to be used is `64`. If the player's score is **less** than the target score of the level, this is a loss. In this case, the message reads `TRY AGAIN`. The coordinates and the font size are the same as previously mentioned. The target scores for each level are `150`, `400` and `350` respectively.

The second message on the end screen is an instruction message that reads `PRESS SPACE TO RETURN TO LEVEL SELECTION`. This should be rendered **below** the first message. The x-coordinate should be centered horizontally and the y-coordinate is `500`. The font size to be used is `24`.

When the player presses the **space key**, the start screen should be rendered again as described in the *Start Screen* section and the player can choose to play again. The player can terminate 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.

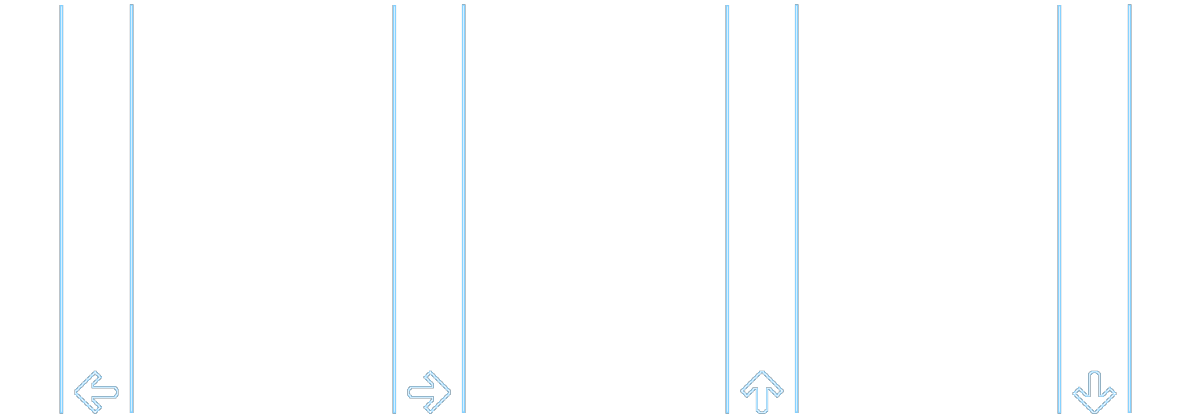
Hint: The `drawString()` method in the Font class uses the given coordinates as the bottom left of the message. So to center the x-coordinate of the message, you will need to calculate the coordinate using the `Window.getWidth()` and `Font.getWidth()` methods.

The Game Entities

The following game entities have an associated image (or multiple!) and a starting location `(x, y)`. Remember that all images are drawn from the **centre** of the image using these coordinates.

Lane

In this game, there are **four** normal lanes for the four arrow keys (left, right, up and down) represented by the images shown on the next page. The `(x, y)` coordinate of the centre of each image is as follows : the x-coordinate is given in the CSV file and the y-coordinate is **384**. The position of the lane stays constant throughout the game.



(a) laneLeft.png (b) laneRight.png (c) laneUp.png (d) laneDown.png

Figure 3: The lane images (note that the size has been reduced to fit on the page)

A lane can have any number of notes and hold notes. Each lane has a stationary note symbol at the bottom which is the target. The player needs to press the corresponding arrow key when the falling note overlaps with the stationary note to score points. The y-coordinate of the centre of the four stationary notes is **657**.

Special Lane

This is a special type of lane shown below that features in Level 2 and 3. Only special notes will fall down this lane. The player needs to press the **space key** when the falling special note overlaps with the stationary symbol at the bottom of the lane, to activate the effect of the special note. Only one special lane will be present in one level. The y-coordinates mentioned in the previous section are the same for special lanes too.

Figure 4: laneSpecial.png

Note


(a) noteLeft.png


(b) noteRight.png


(c) noteUp.png


(d) noteDown.png

Figure 5: The note images

There are **four** types of normal notes for the four arrow keys represented by the above images. Each note will descend vertically from the top of the screen in the corresponding lane, for example: a left note should descend in the left lane and an up note should descend in the up lane. The starting `(x, y)` coordinate of the centre of each note image is as follows : the x-coordinate is the **same** as the x-coordinate of the lane it corresponds to and the y-coordinate is **100**.

Each note moves downwards at a speed of **2 pixels per frame** (for 60Hz screens, increase this value to 4). The frame number from which the note starts being drawn on screen is given in the CSV file. The note will continue to be drawn until either it either leaves the window from the bottom of the screen or the player presses the corresponding key for the note (and a score is calculated as described below).

Note Scoring

The score for each key press is calculated based on the **accuracy** of how close the given note was to the stationary note symbol in the lane when the key was pressed. When the key is pressed, the **absolute distance** in pixels between the y-coordinate of the centre of the falling note and the y-coordinate of the centre of the stationary note is calculated. The method to determine the score from this distance is shown below.

- If distance <= 15, this is a `PERFECT` score and receives `10` points
- If 15 < distance <= 50, this is a `GOOD` score and receives `5` points
- If 50 < distance <= 100, this is a `BAD` score and receives `-1` points
- If 100 < distance <= 200, this is a `MISS` and receives `-5` points.

If the note leaves the window from the bottom of the screen without the corresponding key being pressed, this is considered as a `MISS` too and receives `-5` points.

When a score is calculated, the corresponding **score message** (shown in the list above) must be rendered on screen, roughly centered horizontally and vertically. The font size must be set to `40` and the message must be rendered for `30` frames (`15` frames for 60Hz screens). For example, when the score is perfect, the text rendered must be `PERFECT`.

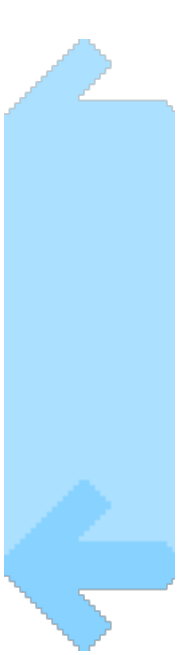
The player's current **total score** must also be rendered on screen. The score is rendered in the top left corner of the screen in the format of `"SCORE k"` where `k` is the current score. The bottom left corner of this message should be located at `(35, 35)` and the font size should be 30.

Figure 6: Score

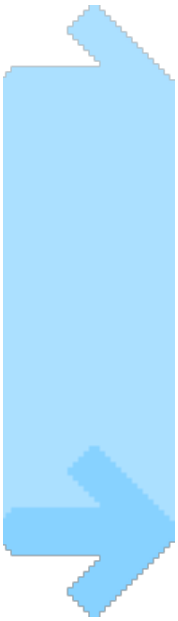
Hold Note

A hold note is a **different type of normal note**, where the player has to **hold down** the corresponding arrow key for the duration in which the falling hold note overlaps with the stationary note in the lane. Once again, there are **four** types of hold notes for the four arrow keys shown in the figure below.

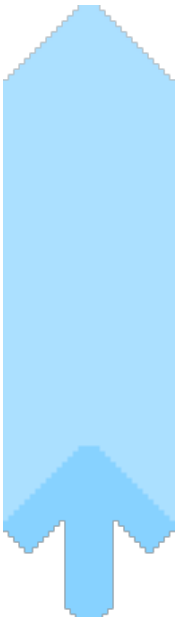
The starting `(x, y)` coordinate of the centre of each image is as follows : the x-coordinate is the **same** as the x-coordinate of the lane it corresponds to and the y-coordinate is **24**. The speed is the **same** as for a normal note and the frame number is also given in the CSV file.



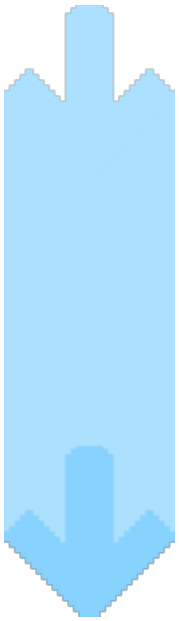
(a) holdNoteLeft.png



(b) holdNoteRight.png



(c) holdNoteUp.png



(d) holdNoteDown.png

Figure 7: The hold note images

Hold Note Scoring

For hold notes, the scoring is calculated in a similar manner but the difference is that **two** scores are calculated - first when the hold is **started** and the second when the hold is **released**. **Hint:** To check if a hold has been released, you will find the `wasReleased()` method in the `Input` class helpful.

When the hold starts, the y-coordinate at the **bottom** of the image of the holding note needs to be used to calculate the distance (instead of the centre of the image like in a normal note). Likewise when the hold is released, the y-coordinate at the **top** of the image needs to be used. **Hint:** To calculate the y-coordinate at the bottom of the hold note image, add `82` to the centre y-coordinate, and for the y-coordinate at the top of the image, subtract `82`.

The method of determining the score from the distance is the same as given in the *Note Scoring* section (bottom of page 8). If the hold note leaves the window from the bottom of the screen without the hold starting, this is considered as a `MISS` and receives `-5` points. If the hold was released at a distance greater than `200` pixels, this is a `MISS` too.

Similar to normal notes, the score messages should also be rendered on screen and the scores from hold notes, need to be added to the total score too.

Special Note

A special note is a special type of normal note that features in Levels 2 and 3. Similar to normal notes, each note has a starting `(x, y)` coordinate, where the x-coordinate is the **same** as the x-coordinate of the lane it corresponds to and the y-coordinate is **100**.

Instead of a score, each special note has an **effect** that activates when the key for the lane it corresponds to is pressed. The distance calculation is simpler than the other notes - if the distance between the y-coordinate of the stationary note (at the bottom of the lane) and the y-coordinate of the special note is <= 50, it is considered as **activated**. When activated, a message is rendered on screen **similar** to the scoring messages for normal and hold notes. The 4 types of special notes are described below.

Double Score Note

This note can only appear in the special lane and is represented by the image on the right. The special effect of this note is that all scores are **doubled** temporarily. The message to be rendered when activated reads `Double Score`. The effect lasts for `480` frames and scoring returns to normal after this elapses.

Figure 8: note2x.png

Speed Up Note

Figure 9: noteSpeedUp.png

This note can only appear in the special lane and is represented by the image on the left. Its special effect is that the speed of all notes is **increased** by **1**. The message to be rendered when activated reads `Speed Up` and `15` points is added to the score. The effect lasts until a slow down note is activated.

Slow Down Note

This note can only appear in the special lane and is represented by the image on the right. The special effect of this note is the speed of all notes is **decreased** by **1**. The message to be rendered when activated reads `Slow Down` and `15` points is added to the score.

Figure 10: noteSlowDown.png

Bomb Note

Figure 11: noteBomb.png

This note can appear in any of the normal lanes or in the special lane. It is represented by the image on the left. The special effect of this note is that all notes in that lane will be **removed** from the screen. The message to be rendered when activated reads `Lane Clear`.

Enemy

The enemy is an entity that appears in Level 3 and is represented by the image on the right. In Level 3, every `600` frames, an enemy must be created. At creation, the enemy's `(x, y)` coordinates must be chosen as follows : the x-coordinate must be **randomly** chosen between `100` and `900`, the y-coordinate must be **randomly** chosen between `100` and `500`. An enemy can move horizontally at a speed of **1 pixel per frame** either in the left direction or right direction, and this must also be chosen randomly at creation.

Figure 12: enemy.png

When the enemy's x-coordinate reaches either `100` or `900`, the enemy must start moving in the opposite direction (i.e. the enemy will reverse direction at these coordinates). The enemy's goal is to steal normal notes from lanes by **colliding** with them. Collisions are detected as follows: if the distance between the centre-coordinates of the enemy image and the centre-coordinates of the note image is <= 104, this is considered as a collision. If collided with, the note will disappear from the screen.

Guardian

Figure 13: guardian.png

The guardian is an entity that appears in Level 3 and is represented by the image on the left. The guardian does not move and remains at `(800, 600)` throughout. When the **left shift** key is pressed, the guardian will fire a **projectile** at the nearest enemy to its location. The nearest enemy is found by finding the enemy with the closest distance calculated once again using the centre of the respective images.

Projectile

A projectile gets created by the guardian when attacking an enemy and is rendered using the image on the right. A projectile moves at a speed of **6 pixels per frame** in the direction of the enemy target which is set at creation. Its image should be rotated in the direction of the enemy.

Figure 14: arrow.png

For example, if the guardian is at the coordinate `(3,3)` and the enemy is at the coordinate `(13,8)`, the projectile should be fired with its image rotated by `0.464 radians`. The diagram below illustrates how this can be calculated (note that the positions given are for explanation and may not reflect in-game positions).

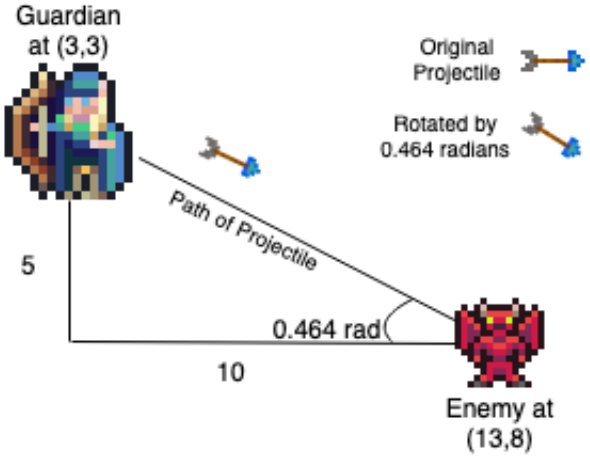


Figure 15: Projectile Explanation

If the projectile collides with the enemy, the enemy will **disappear** from the screen. Collisions are detected as follows: if the distance between the centre-coordinates of the enemy image and the centre-coordinates of the projectile image is <= 62, this is considered as a collision. If the projectile reaches the edges of the window without colliding with an enemy, the projectile needs to stop being updated (rendered).

Hint: To deal with velocities, you many find it useful to use the Vector2 class in Java. 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.

Sound

Note that this section is optional and you will not be assessed on this.

If you want to add sound, you may use the same music demo from Project 1. Two new tracks

(`track2.wav` and `track3.wav`) have been provided in the `res` folder for you to use with Level 2 and 3.

Your Code

You must submit a class called `ShadowDance` 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 new level start screen.
- Implement the Level end screen.
- Read the Level 2 CSV file.
- Implement the special lane and special notes' behaviour/logic.
- Implement Level 2.
- Read the Level 3 CSV file.
- Implement the enemy, guardian and projectile behaviour/logic.
- Implement Level 3.

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 `ShadowDance` 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 a new level with different entities). 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 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
   ├─ ShadowDance.java
   └─ other Java files
```

On 13th October 2023 at 4:30pm, 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 13th October 2023 4:30pm 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 projectile's collision behaviour
- refactored code for cleaner design

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

- fesjakhbdjl
- yeah easy finished the bomb note
- 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 Extension form in the **Projects** module on Canvas. 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 **4:30pm sharp** on Wednesday 20$^{\text{th}}$ September 2023 (Project 2A) and on Friday 13$^{\text{th}}$ October 2023 (Project 2B). Any submissions received past this time (from 4:30pm 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 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 **20** 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 **8 marks**.
  - Correct implementation of start screen and level selection: **0.5 marks**
  - Correct implementation of lanes and special lanes: **0.5 marks**

  - Correct implementation of notes and hold notes: **0.5 marks**
  - Correct implementation of each special note's behaviour (including images, movement and effects): **4 marks**
  - Correct implementation of enemy's behaviour: **1 mark**
  - Correct implementation of guardian and projectiles: **1 mark**
  - Correct implementation of end screen: **0.5 marks**
- 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**