BSc (Hons) in Applied Computing (Fintech)
INF1009: Object Oriented Programming

Academic Year 23/24, Trimester 2

Report Proposal

Prepared for:

Professors of INF1009

Prepared by:

Ainsley Chang Qi Jie (2302954)/ 2302954@sit.singaporetech.edu.sg

Jovan Lim Yu Hang (2303397)/ 2303397@sit.singaporetech.edu.sg

Lin Zhenming (2302993)/ 2302993@sit.singaporetech.edu.sg

Varsha Kulkarni (2303153)/ 2303153@sit.singaporetech.edu.sg

Yeow Dao Xing(2303175)/ 2303175@sit.singaporetech.edu.sg

Due by: February 23, 2024

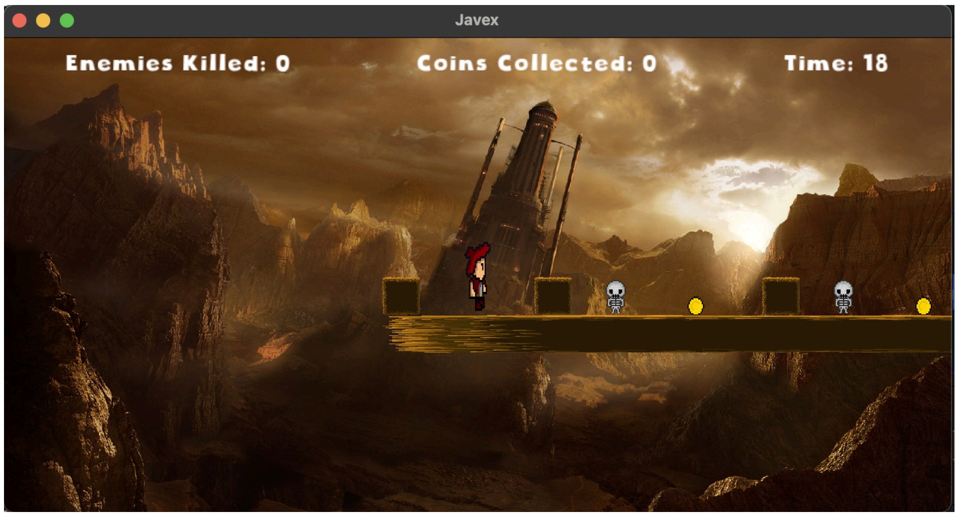
1 Javex, the Game Engine

The creation of a game can be done in a multitude of methods, each with distinct advantages and limitations. The traditional approach of scripting all components in a single file using Functional Programming (FP) lacks efficiency, adaptability and modularity. When flexibility and scalability are the priority, Object-Oriented Programming (OOP) emerges as a superior paradigm, offering a structured framework that facilitates the creation and management of complex game systems.

Introducing Javex, a game engine built around the four cardinal principles of OOP: encapsulation, inheritance, polymorphism, and abstraction. This strategic design ensures that game components are modular, extendable, and, above all, manageable. Encapsulation secures data integrity, inheritance fosters code reuse, polymorphism enhances flexibility, and abstraction simplifies complexity, making Javex an ideal demonstration of OOP best practices.

Central to Javex's design is the Single Responsibility Principle (SRP). Each class within Javex is carefully crafted to perform a unique function, ensuring clarity, ease of debugging, and enhancement. This principle underpins the engine's architecture, leading to a more maintainable and coherent code, thereby streamlining the development process and facilitating future scalability.

Furthermore, Javex embraces the Open/Closed Principle, allowing classes to be extended without altering existing code. This approach not only prevents disruptions to the current game engine's ecosystem but also encourages innovation and continuous improvement. By adhering to this principle, Javex provides a flexible and resilient foundation, enabling developers to build upon the engine while preserving the integrity of its core components.



2 System Design

The Javex game engine is built upon seven foundational manager classes, each designed to streamline the development process and enhance the gameplay experience.These managers work in tandem to create a cohesive and immersive environment, allowing developers to focus on crafting engaging content while ensuring optimal performance and user interaction.

2.1 Simulation Lifecycle Manager

The `SimulationLifecycleManager` acts as the conductor of the Javex engine, initiating and overseeing the game's operational phases. It begins with the `create()` method, which kickstarts the game by initialising essential subsystems: the `SceneManager`, `InputManager`, and `OutputManager`. This ensures a structured and systematic beginning to the game's lifecycle. During gameplay, the `render()` method continuously updates and renders scenes based on user interactions, while the `dispose()` method efficiently handles resource cleanup, preventing memory leaks and ensuring a smooth game closure.

2.2 Scene Manager

The `SceneManager` administers the various scenes or stages of the game, facilitating transitions between menus, gameplay, and other interfaces. It employs a stack-based system to manage scene layers, allowing for dynamic additions or removals. This manager is pivotal for scene lifecycle handling, including initialization, rendering, and disposal, ensuring that each scene correctly reflects the game's current state and user interactions. For our game engine, it promotes reusability by allowing the IDE to add a new scene, extend the Scene.java class and define the outlooks of it. Then, allow the sceneManager to `.push()`, `.pop()` or `.set()` the specific scene at any point of the player's journey.

2.3 Input/Output Manager

For the input/output Manager, Javex would split them into 2 distinct parts that would work independently to be able to control the input from the user and display an output for the user to interact with.

2.3.1 Input Manager

The `InputManager` in Javex offers a flexible framework for handling user inputs, allowing for a customizable gaming experience. It abstracts user actions from specific devices, interpreting inputs as generic commands that can be mapped to any game function. This system enables developers to design versatile control schemes and allows players to tailor their interaction methods. By maintaining a clear separation between input detection and game actions, Javex ensures that controls remain adaptable and intuitive, catering to a wide range of player preferences and input devices.
For our game engine, it promotes scalability by allowing the IDE to allow the game to accept more input devices or input signals, configure it within the InputManager.java class and allow the downstream classes to get the information (in the form of booleans) from it.

2.3.2 Output Manager

Javex's `OutputManager` is responsible for all auditory feedback within the game. It manages sound effects and background music, ensuring that players receive coherent and

timely responses to their actions. The manager allows for dynamic control over sound levels and playback, enriching the gaming atmosphere while providing essential feedback to the player. This careful management of sensory outputs helps to create a more immersive and enjoyable gaming experience, making the game world feel alive and responsive.
For our game engine, it promotes reusability by allowing the IDE to play different music or sound effects in the game, simply call the `outputManager.play(musicPath)` method, where `musicPath` can be defined in each specific scene or specific collisions.

2.4 Player Control Manager

Dedicated to enhancing player interaction, the `PlayerControlManager` interfaces closely with the Input Manager to translate player inputs into character actions. It ensures responsive and intuitive gameplay, allowing players to control their avatars effectively. This manager fine-tunes the player's experience, adjusting character responses and movements to input signals, thereby directly impacting engagement and playability.

2.5 AI Control Manager

The `AiControlManager` imbues non-player characters with intelligence, utilising pathfinding and decision-making algorithms to simulate complex behaviours. Leveraging LibGDX's AI utilities, it allows enemies and NPCs to react dynamically to player actions and game environments. This manager enriches the gaming experience by providing challenging, realistic adversaries and allies, contributing to a lively, unpredictable game world.
For our game engine, it promotes scalability by allowing the IDE to facilitate more AI behaviours in the game, simply define the AI behaviour within the AiControlManager.java class, and pass the intended AI-controlled entity as its argument.

2.6 Collision manager

The `CollisionManager` is crucial for physical interactions within the game, handling collisions between entities to ensure logical and consistent outcomes. By managing how objects interact upon contact, it contributes to the game's realism and adherence to physics. The Collision Manager dictates responses to collisions, from simple contact to complex reactions, ensuring gameplay remains engaging and true to life.
For our game engine, it promotes scalability by allowing the IDE to insert more collisions to be detected and resolved, simply add on to the `.beginContact()` method of the CollisionManager.java class. The game world will then listen to these collisions.

2.7 Entity Manager

The `EntityManager` oversees all game entities, including characters, items, and obstacles. It handles their creation, updates, and rendering, based on defined behaviours and game logic. This manager ensures that each entity interacts correctly with the game world, maintaining state coherence and performance efficiency. It plays a crucial role in populating the game environment with dynamic, interactive elements.
For our game engine, it promotes scalability by allowing the IDE to spawn a new entity of any kind; it can be done within the `initialise()` method of the EntityManager.java class.

3 Object-Oriented Programming (OOP) Principles

Object-Oriented Programming (OOP) principles form the cornerstone of the Javex game engine, ensuring a robust, modular, and scalable architecture. These principles—Encapsulation, Inheritance, Polymorphism, and Abstraction—enable developers to create interactive and complex game elements with enhanced maintainability and code reusability, effectively facilitating a comprehensive and efficient game development process.

3.1 Encapsulation

Encapsulation is the OOP principle that restricts direct access to some of an object's components, which leads to a modular and secure design. In Javex, encapsulation is implemented through access modifiers (`private`, `protected`, `public`), and can be demonstrated in two kinds of instances:

a. Private Attributes and Public Methods: These are commonly seen in the interactions between concrete classes. For example, the `health` attribute in `Player` class is marked private to prevent unauthorised external access and modification. Instead, public methods `getHealth()` and `reduceHealth()` provide controlled access to this attribute, ensuring that health cannot be set to an invalid state by external classes.

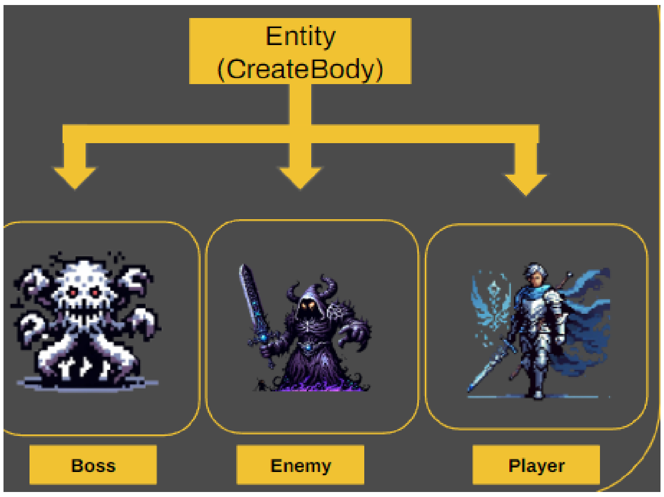
b. Protected Attributes: These are commonly seen in the inheritance of a children class from a parent class. For example, the `Entity` class has protected attributes like `width` and `height`, accessible to any subclass (such as `Player` or `Enemy`) within the same package, facilitating attribute sharing while maintaining control.

3.2 Inheritance

Inheritance is a cornerstone of Javex's architecture, promoting code reusability and logical hierarchy. The engine defines base classes with common functionality from which other classes can inherit, reducing redundancy and enhancing maintainability.

For example, the Scene class serves as a base class for various specific scenes like MenuScene, PlayScene, and EndScene, each inheriting common methods and attributes. This mechanism not only streamlines code updates and maintenance by allowing changes in the Scene class to propagate to all derived classes but also facilitates the consistent implementation of essential scene methods.

Furthermore, inheritance facilitates the polymorphic use of objects. A function expecting a Scene object can operate with instances of its subclasses, allowing for flexible code that can handle different types of scenes uniformly.



3.3 Polymorphism

Polymorphism in Javex manifests primarily through method overriding, allowing classes to implement parent class methods based on their specific requirements. This principle enables Javex to handle various game elements with more flexibility.

Method overriding is extensively used in the scene management system. Each subclass of `Scene`, such as `MenuScene` or `PlayScene`, provides its own implementation of the `handleInput()` method. While the `Scene` class provides a general framework for what every scene should do, the actual content and behaviour of these methods are tailored to the specific needs of each scene, enabling diverse functionalities while maintaining a consistent interface.

This use of polymorphism allows Javex to dynamically change its behaviour during runtime, depending on the current active scene. It eliminates the need for cumbersome conditional statements to determine the scene type, leading to cleaner and more understandable code.

3.4 Abstraction

In Javex, abstraction simplifies the complex reality of user inputs into a streamlined, understandable interface, as demonstrated by the InputManager class. This abstraction allows Javex to capture a wide range of user inputs without cluttering the main game logic with intricate details of each input type.

The InputManager serves as an abstract layer between the physical keypresses and the game's reaction to these inputs. It encapsulates the complexity of detecting and interpreting various forms of inputs, such as keyboard strokes or mouse clicks, by converting them into a set of boolean flags or triggering specific callbacks. This design allows game scenes and entities to interact with user inputs through a simplified, uniform interface, focusing on the "what" rather than the "how" of input processing.

By abstracting input handling, Javex decouples game logic from the specifics of input devices, enhancing the engine's flexibility and portability. Developers can easily modify input behaviours or integrate new input devices without altering the core game mechanics. This level of abstraction fosters a more modular, maintainable codebase and exemplifies how Javex utilises OOP principles to tackle the complexities of game development.

4 Solid Design Principles

The SOLID principles are a set of five design principles aimed at making software designs more understandable, flexible, and maintainable. These principles are crucial for reducing code dependencies, making systems easier to understand, update, and scale. Adherence to these principles ensures that software development follows best practices for object-oriented design, leading to more robust and scalable systems.

4.1 Single Responsibility Principle (SRP)

The Single Responsibility Principle asserts that a class should have one and only one reason to change, signifying that it should have only one job. This principle enhances modularity and facilitates maintenance by ensuring that each class in a system has a single responsibility. In Javex, the InputManager class exemplifies SRP by solely managing user inputs, decoupling this function from other game logic. It handles keyboard events independently, ensuring that input processing is isolated from the game's mechanics. The InputManager class's adherence to SRP is evident through its focused role in updating boolean states based on user interactions, avoiding entanglements with unrelated game functionalities. This focused approach simplifies debugging and enhances the reusability of the class in other contexts, illustrating SRP's benefits in maintaining a clean and understandable codebase. Figure 1 would illustrate as to how a player could be created with different classes each responsible for 1 action of the player.

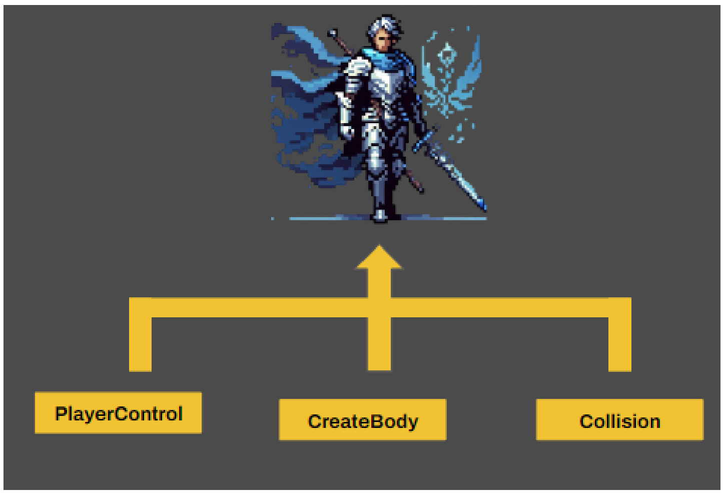


Figure 1: Player Class

4.2 Open/Closed Principle (OCP)

The Open/Closed Principle dictates that software entities should be open for extension but closed for modification. This means that the behaviour of a module can be extended without altering its source code. In Javex, this principle is demonstrated through the Entity class, which serves as a foundation for various game entities like Player, Enemy, and Terrain. For example, new types of entities can be created by extending the Entity class, such as Player and Terrain classes. Both were extended from the entity class with their own specific implementations for creating their physical bodies(‘createBody’). While sharing a common foundation, they also had diverse behaviours and properties among the different entities, and their creation or modification does not affect the Entity class.

This approach allows Javex to incorporate new functionalities seamlessly, illustrating OCP’s role in fostering a flexible and scalable architecture without compromising the integrity of the existing system.

4.3 Liskov Substitution Principle

The Liskov Substitution Principle requires that objects of a superclass shall be replaceable with objects of its subclasses without affecting the correctness of the program. In Javex, LSP can be seen in how different Scene subclasses (like MenuScene, PlayScene, EndScene) are used interchangeably through a common base class reference. This means any Scene can be replaced with its subclass without changing the expected behaviour of the game, ensuring system flexibility and promoting the use of polymorphism effectively. This principle reinforces the architecture's robustness and the interchangeability of its components.

5 Highlights

The Javex game engine showcases its innovation and versatility through the comprehensive design of its AI Control Manager. This feature leverages the LibGDX AI package to its fullest potential, starting with the intricate implementation of SeekBehavior. This behaviour allows non-player characters (NPCs) to dynamically pursue targets within the game, creating a more engaging and challenging environment for players. The AI Control Manager's sophisticated design not only enriches current gameplay but also establishes a solid foundation for integrating diverse AI behaviours. As reflected in Figure 1, 2 different entities would have the steerable adapter method applied to them to allow them to utilise the seek behavior on them to allow the AI control to be applied to the ‘Entity’ to then seek the ‘Target’

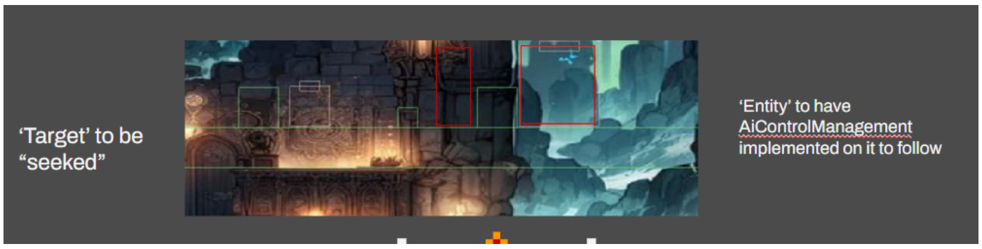

‘Target’ to be “seeked”
‘Entity’ to have AiControlManagement implemented on it to follow

Figure 1:Target and Entity management

The true potential of Javex lies in the modular and extendable structure of its AI Control Manager. Designed for adaptability, it supports the seamless addition of various AI strategies beyond simple seeking, such as Flee, Arrive, and Wander behaviours. This flexibility enhances the game's depth, providing different layers of interaction and complexity. The architecture not only caters to the current gaming landscape but is also poised to accommodate future advancements in AI technology, making it a versatile tool for developers.

The AI Control Manager exemplifies Javex's commitment to delivering a rich and immersive gaming experience. Its potential for expansion and adaptation signifies Javex's approach to game development—one that values depth, player engagement, and technological foresight, ensuring that Javex remains a dynamic and powerful engine for both current and future game developments.

6 Limitations

The Javex game engine, while robust and versatile in many aspects, currently faces a significant limitation due to its static level design. In its present form, Javex offers a singular, unchanging stage for gameplay, with all game objects and scenarios predefined in the EntityManager and rendered in a fixed PlayScene. This design choice, primarily aimed at demonstrating solid Object-Oriented Programming (OOP) principles through distinct manager classes, results in a lack of variety and replayability, as players are confined to a single, unalterable environment.

This limitation stems from the initial architectural decision to focus on demonstrating the integration and functionality of different managerial components rather than on providing diverse gameplay experiences. The EntityManager, responsible for populating the Box2D world, restricts the game to one predetermined set of conditions and challenges. Consequently, while showcasing the engine's capacity to manage game entities effectively, it also bounds the player's experience to a singular, repetitive scenario.

Addressing this constraint, future enhancements to Javex could introduce a more dynamic level design approach. By segregating levels into distinct classes, each with unique attributes, conditions, and AI logic, Javex can significantly expand its gameplay variety. This evolution would allow the engine not only to maintain its OOP integrity and demonstrate the reusability and scalability of its codebase but also to offer game developers and players a richer, more varied gaming experience. Implementing such a modular level system would transform Javex from a demonstration of technical proficiency into a more comprehensive tool for game development, catering to a broader range of creative needs and player preferences.

7 Conclusion

The Javex game engine stands as a testament to meticulous design and engineering, firmly rooted in the principles of Object-Oriented Programming (OOP) and SOLID guidelines. By embracing these foundational concepts, Javex has been crafted not merely as a game engine but as a versatile framework that simplifies the complexities involved in game development. Each component within the engine has been designed with a clear purpose, ensuring adherence to the Single Responsibility Principle, which promotes a cleaner, more organised codebase that is easier to maintain and expand.

The architecture of Javex showcases an exceptional application of OOP principles such as encapsulation, inheritance, and polymorphism, which enhance modularity and facilitate the extension of game features without the need for extensive modifications. This adherence to the Open/Closed Principle ensures that the engine is both robust in its current capabilities and adaptable to future enhancements or varying game requirements.

Moreover, the engine's design emphasises reusability and scalability, enabling developers to build diverse games using Javex as the foundational platform. Whether creating complex game scenes, managing user inputs, or implementing AI behaviours, Javex provides a solid, reusable base that significantly reduces development time and effort. This is particularly evident in its modular manager classes, which abstract away the complexities of specific game functionalities, allowing developers to focus on crafting unique gameplay experiences.

In conclusion, Javex embodies a well-thought-out blend of design principles and practical implementation strategies, making it an ideal starting point for developing engaging and dynamic games. Its commitment to OOP and SOLID principles, combined with its focus on reusability and scalability, positions Javex not just as a game engine but as a comprehensive game development ecosystem. With Javex, designers and developers are equipped with a powerful toolset that streamlines the journey from concept to final product, paving the way for innovative and captivating gaming experiences.

8 Group Contribution

8.1 Ainsley Chang Qi Jie

Ainsley’s main contribution was on creating the input and output managers. For the input manager it was to ensure that whenever a user does an input it would be registered. As for the output manager, it was mainly to ensure that throughout our game there was audio being played. Contributions were also made in rendering the character sprites and working on multiple classes, to ensure everything was rendered correctly. He also worked on the entity manager and ensured that the sprites were loaded only in the playscene so that the game would be versatile and adaptable to future games.

8.2 Jovan Lim Yu Hang

Jovan's contributions was to aid in the creation of the UML Diagram, as well as creating the parent class `Scene`, and other scenes like the menu, play, pause, and end scenes. Design and functionality of the scenes were also thought and worked on. He also developed the `SceneManager` to manage switching between these scenes. In addition, he created the HUDManager and implemented the HUD for the scenes, which displays a HUD that displays stats of the player. All this was done whilst ensuring reuseability and scalability.

8.3 Lin Zhenming

Zhenming was the project lead and his contributions mainly revolved around aligning everyone onto the same page. He consolidated the ideas from teammates and feedback from professors and seniors, converting them into actionable items for the team to act upon. During the development phase, he segregated the workload within the team and reorganised everyone's individual output into the final codebase, ensuring the program runs smoothly from start to end, all while working on his individual assigned task: implementing the Collision Manager. In addition to these, he schedules team meetings for progress check and 1-1 meetings to clear his teammates' obstacles such as getting onboard GitHub for version control and debugging specific portions of codes.

8.4 Varsha Kulkarni

For Varsha, her main contribution was aiding in the creation of the UML diagram, which provided a visual representation of the game's structure. She also worked on the game flow, determining the sequence of events and player progression. Additionally, Varsha was responsible for designing the game map, which involved planning the layout, logic and considering the visual appeal. These tasks collectively helped in shaping the game's structure and gameplay experience.

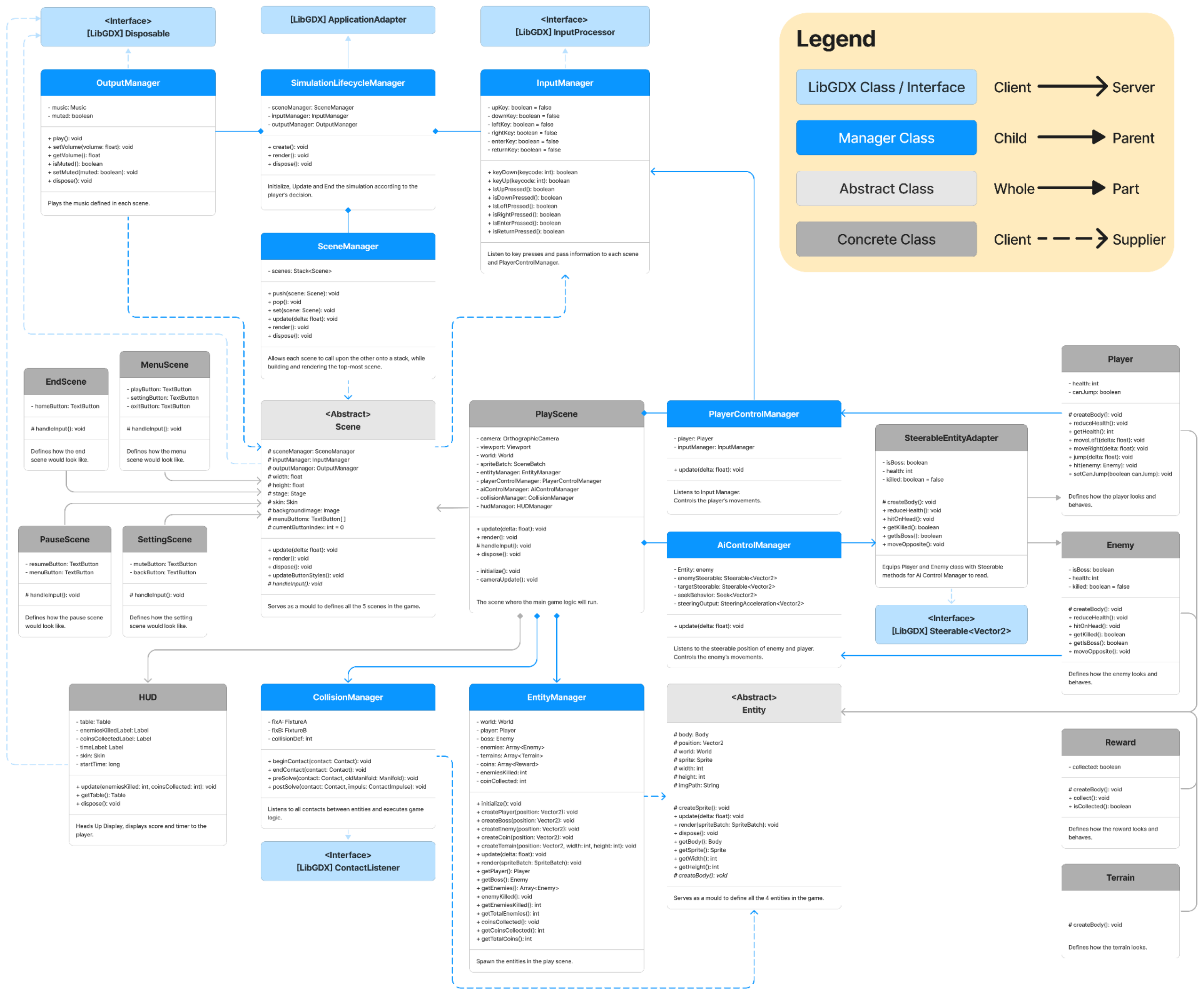
8.5 Yeow Dao Xing

For Dao Xing, his main contribution was the initial introduction of the Entity Management class with emphasis to focus on the enemy class. Dao Xing was able to link the initial enemy with the playscene to act as the basis for the entity class to be able to render into the box2d world to be initialised and allow the linking of the 2 classes. Dao Xing also introduce the Ai element class and formulate the steerable<vector2> to be implemented into the different

9 Classes and their functions

	Class Name	Function
1	SimulationLifecycleManager	Initialize, Update and End the game simulation according to the player's decision.
2	Constants	Contains several constants that are repeatedly used over several classes.
3	InputManager	Listens to key presses and passes information to each scene and PlayerControlManager.
4	OutputManager	Plays the music defined in each scene.
5	SceneManager	Creates a stack for scenes. Allows each scene to call upon the other onto the stack. Builds and renders the top-most scene.
6	Scene <Abstract>	Serves as a mould to define all the 5 scenes in the game.
7	MenuScene	Defines how the menu scene would look like.
8	PauseScene	Defines how the pause scene would look like.
9	SettingScene	Defines how the setting scene would look like.
10	EndScene	Defines how the end scene would look like.
11	PlayScene	The scene where the main game logic will run.
12	HUD	Heads Up Display, displays score and timer to the player.
13	CollisionManager	Listens to all contacts between entities and executes collision logic.
14	PlayerControlManager	Listens to the Input Manager. Controls the player's movements.
15	AiControlManager	Listens to the positions of steerable enemy and player. Controls the enemy's movements.
16	SteerableEntityAdapter	Equips Player and Enemy class with Steerable methods for Ai Control Manager to read.
17	EntityManager	Spawn the entities in the play scene.
18	Entity <Abstract>	Serves as a mould to define all the 4 entities in the game.
19	Player	Defines how the player looks and behaves.
20	Enemy	Defines how the enemy looks and behaves.
21	Reward	Defines how the reward looks and behaves.
22	Terrain	Defines how the terrain looks.

10.0 UML Diagram


Legend
LibGDX Class / Interface
Manager Class
Abstract Class
Concrete Class
Client → Server
Child → Parent
Whole → Part
Client - - → Supplier
<Interface> [LibGDX] Disposable
[LibGDX] ApplicationAdapter
<Interface> [LibGDX] InputProcessor
OutputManager
- music: Music
- muted: boolean
+ play(): void
+ setVolume(volume: float): void
+ getVolume(): float
+ isMuted(): boolean
+ setMuted(muted: boolean): void
+ dispose(): void
Plays the music defined in each scene.
SimulationLifecycleManager
- sceneManager: SceneManager
- inputManager: InputManager
- outputManager: OutputManager
+ create(): void
+ render(): void
+ dispose(): void
Initialize, Update and End the simulation according to the player's decision.
InputManager
- upKey: boolean = false
- downKey: boolean = false
- leftKey: boolean = false
- rightKey: boolean = false
- enterKey: boolean = false
- returnKey: boolean = false
+ keyDown(keycode: int): boolean
+ keyUp(keycode: int): boolean
+ isUpPressed(): boolean
+ isDownPressed(): boolean
+ isLeftPressed(): boolean
+ isRightPressed(): boolean
+ isEnterPressed(): boolean
+ isReturnPressed(): boolean
Listen to key presses and pass information to each scene and PlayerControlManager.
SceneManager
- scenes: Stack<Scene>
+ push(scene: Scene): void
+ pop(): void
+ set(scene: Scene): void
+ update(delta: float): void
+ render(): void
+ dispose(): void
Allows each scene to call upon the other onto a stack, while building and rendering the top-most scene.
EndScene
- homeButton: TextButton
# handleInput(): void
Defines how the end scene would look like.
MenuScene
- playButton: TextButton
- settingButton: TextButton
- exitButton: TextButton
# handleInput(): void
Defines how the menu scene would look like.
<Abstract> Scene
# sceneManager: SceneManager
# inputManager: InputManager
# outputManager: OutputManager
# width: float
# height: float
# stage: Stage
# skin: Skin
# backgroundImage: Image
# menuButtons: TextButton[ ]
# currentButtonIndex: int = 0
+ update(delta: float): void
+ render(): void
+ dispose(): void
+ updateButtonStyles(): void
# handleInput(): void
Serves as a mould to defines all the 5 scenes in the game.
PauseScene
- resumeButton: TextButton
- menuButton: TextButton
# handleInput(): void
Defines how the pause scene would look like.
SettingScene
- muteButton: TextButton
- backButton: TextButton
# handleInput(): void
Defines how the setting scene would look like.
PlayScene
- camera: OrthographicCamera
- viewport: Viewport
- world: World
- spriteBatch: SceneBatch
- entityManager: EntityManager
- playerControlManager: PlayerControlManager
- aiControlManager: AiControlManager
- collisionManager: CollisionManager
- hudManager: HUDManager
+ update(delta: float): void
+ render(): void
# handleInput(): void
+ dispose(): void
- initialize(): void
- cameraUpdate(): void
The scene where the main game logic will run.
PlayerControlManager
- player: Player
- inputManager: InputManager
+ update(delta: float): void
Listens to Input Manager.
Controls the player's movements.
AiControlManager
- Entity: enemy
- enemySteerable: Steerable<Vector2>
- targetSteerable: Steerable<Vector2>
- seekBehavior: Seek<Vector2>
- steeringOutput: SteeringAcceleration<Vector2>
+ update(delta: float): void
Listens to the steerable position of enemy and player.
Controls the enemy's movements.
SteerableEntityAdapter
- isBoss: boolean
- health: int
- killed: boolean = false
# createBody(): void
+ reduceHealth(): void
+ hitOnHead(): void
+ getKilled(): boolean
+ getIsBoss(): boolean
+ moveOpposite(): void
Equips Player and Enemy class with Steerable methods for Ai Control Manager to read.
<Interface> [LibGDX] Steerable<Vector2>
Player
- health: int
- canJump: boolean
# createBody(): void
+ reduceHealth(): void
+ getHealth(): int
+ moveLeft(delta: float): void
+ moveRight(delta: float): void
+ jump(delta: float): void
+ hit(enemy: Enemy): void
+ setCanJump(boolean canJump): void
Defines how the player looks and behaves.
Enemy
- isBoss: boolean
- health: int
- killed: boolean = false
# createBody(): void
+ reduceHealth(): void
+ hitOnHead(): void
+ getKilled(): boolean
+ getIsBoss(): boolean
+ moveOpposite(): void
Defines how the enemy looks and behaves.
HUD
- table: Table
- enemiesKilledLabel: Label
- coinsCollectedLabel: Label
- timeLabel: Label
- skin: Skin
- startTime: long
+ update(enemiesKilled: int, coinsCollected: int): void
+ getTable(): Table
+ dispose(): void
Heads Up Display, displays score and timer to the player.
CollisionManager
- fixA: FixtureA
- fixB: FixtureB
- collisionDef: int
+ beginContact(contact: Contact): void
+ endContact(contact: Contact): void
+ preSolve(contact: Contact, oldManifold: Manifold): void
+ postSolve(contact: Contact, impuls: ContactImpulse): void
Listens to all contacts between entities and executes game logic.
<Interface> [LibGDX] ContactListener
EntityManager
- world: World
- player: Player
- boss: Enemy
- enemies: Array<Enemy>
- terrains: Array<Terrain>
- coins: Array<Reward>
- enemiesKilled: int
- coinCollected: int
+ initialize(): void
+ createPlayer(position: Vector2): void
+ createBoss(position: Vector2): void
+ createEnemy(position: Vector2): void
+ createCoin(position: Vector2): void
+ createTerrain(position: Vector2, width: int, height: int): void
+ update(delta: float): void
+ render(spriteBatch: SpriteBatch): void
+ getPlayer(): Player
+ getBoss(): Enemy
+ getEnemies(): Array<Enemy>
+ enemyKilled(): void
+ getEnemiesKilled(): int
+ getTotalEnemies(): int
+ coinsCollected(): void
+ getCoinsCollected(): int
+ getTotalCoins(): int
Spawn the entities in the play scene.
<Abstract> Entity
# body: Body
# position: Vector2
# world: World
# sprite: Sprite
# width: int
# height: int
# imgPath: String
# createSprite(): void
+ update(delta: float): void
+ render(spriteBatch: SpriteBatch): void
+ dispose(): void
+ getBody(): Body
+ getSprite(): Sprite
+ getWidth(): int
+ getHeight(): int
# createBody(): void
Serves as a mould to define all the 4 entities in the game.
Reward
- collected: boolean
# createBody(): void
+ collect(): void
+ isCollected(): boolean
Defines how the reward looks and behaves.
Terrain
# createBody(): void
Defines how the terrain looks.