SINGAPORE INSTITUTE OF TECHNOLOGY

OBJECT-ORIENTED PROGRAMMING TEAM PROJECT REPORT (P12 TEAM 8)

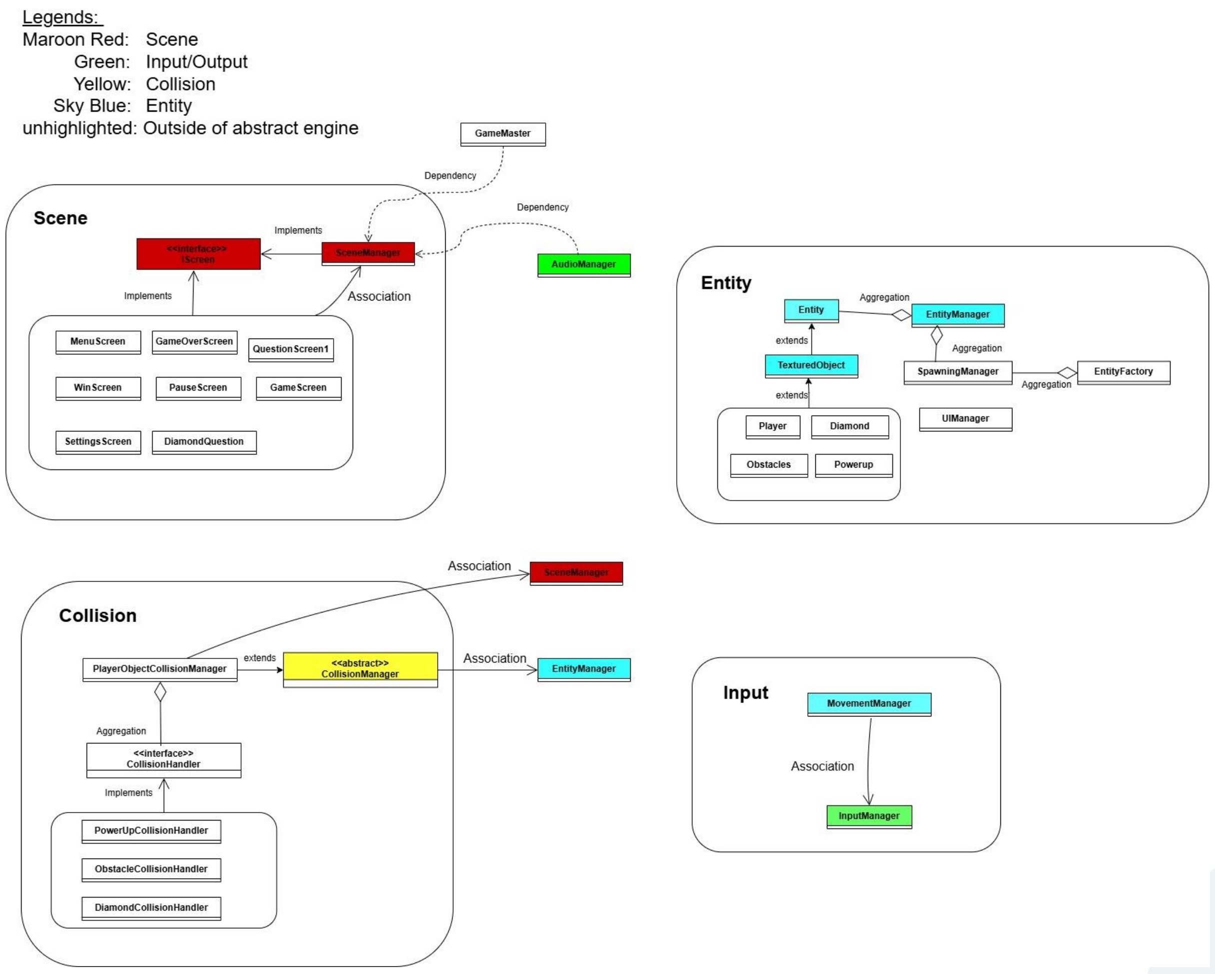
MUHAMMAD WAFIYUDDIN BIN ABDUL RAHMAN	*******
PAE XIANG SHENG	*******
LIM QI ZHEN	*******
TEO WEN TIAN BRENDAN	*******
MUHAMMAD SAAD BIN HADI	*******

Content Page

I. Introduction

Ninja Scam Dash is an exciting 2D game that combines fast-paced free-running action with an important educational twist. In this game, players control a ninja navigating through challenging obstacles while learning about scam prevention. Along the way, they can collect power-ups to enhance their abilities and help them win the game. This project is built upon the game engine developed in Project Part 1, using Object-Oriented Programming principles to ensure scalability and maintainability. The aim is to engage players in an entertaining yet informative experience while promoting awareness of scam prevention.

II. Completed UML Diagram



Overview of the completed UML diagram

The above diagram is our completed UML diagram which shows the overall relation between the different entities. The in-detail UML diagram with the variables and methods used in the specific classes will be in the appendix section of the report.

III. Improved abstract Game Engine

From the feedback that was given to us on our abstract game engine, we applied the following changes to the abstract layer.

Collision Management:

The collision manager from part 1 has been refactored to follow better object-oriented design principles. The application specific logic has been removed from the original collision manager, and it now has been made abstract to improve flexibility and reusability.

Instead of a single collision manager to handle all the collisions in the game, there are now several different dedicated collision handlers based on the current objects in the game, Powerup, obstacle and diamond. Each handler implements an interface called CollisionHandler which defines a handleCollision() method that these handlers must implement, which has object-specific logic in them. This ensures that each game object type has its own distinct logic for handling player-object collisions.

To manage these collision handlers, a new dedicated collision manager that extends the abstract original collision manager, called PlayerObjectCollisionManager is created. This manager registers the several different collision handlers for specific game objects in a hashmap, using the class of the object being collided as the key. When collisions are detected in the game, this manager calls upon the specific handler and calls the handleCollision method specified in its own collision handler.

These changes improve the scalability and flexibility of collision handling.

Scene Management:

Previously, the screens used the libgdx screen interface where there were unimplemented methods used. Currently, having the IScreen interface ensures that the methods defined are implemented only in the sub classes that need them. Having unimplemented methods violates interface segregation principle (ISP) as the classes are forced to have methods that are not needed in their current functionality. Therefore, with the IScreen added, the class implements methods when it is needed and the code is more focused.

Input Management:

The Input Manager is responsible for handling all player controls in a flexible and centralized way. Instead of hardcoding specific key presses throughout the game, it uses a mapping system where actions like “move left” or “move up” are linked to specific keys (for example, ‘A’ or ‘W’). This makes it much easier to manage, change, and personalize controls without modifying multiple parts of the code.

At the start of the game, the Input Manager sets up default key bindings using the common WASD keys. These bindings are stored in a map so that any part of the game can easily ask, for example, “Is the moveRight key being pressed right now?” This system also allows for real-time changes, which is especially useful in the Settings screen where players can customize which keys perform which actions. When the player selects an action to rebind, the Input Manager waits for the next key press and then updates the mapping accordingly.

The Movement Manager works closely with the Input Manager to move the player. Every frame, it checks if a directional key is being pressed and tells the player character to move in that direction. This design separates the input logic from the movement logic, making it easier to manage and modify in the future.

Across the rest of the game, the Input Manager is passed to all screens — such as the Menu, Game, Pause, and Settings screens — through their constructors. This ensures consistency in how input is handled no matter which part of the game the player is in. For example, even when the game is paused or a new screen is shown, the Input Manager still holds the correct key settings.

Overall, this input system is simple, efficient, and designed to be user-friendly. It gives players control over how they want to play, and it gives developers a single point of control for all input-related features. It is easily expandable in the future to support gamepads, or even save custom key configurations for the next time the game is launched.

Audio Management:

The AudioManager is responsible for handling background music and sound effects of the game engine. According to the feedback received, the following changes were made to improve the implementation of Audiomanager.

The current AudioManager now plays audio by filePath instead of by index previously. This is done by maintaining the Hashmap to store sound files, and when playSound(filePath) is called, it checks if the sound is already loaded in the Hashmap, and loads and stores the sound if not already. This prevents reloading of the same sound multiple times, making handling of audio more efficient.

The ObstacleCollisionHandler retrieves AudioManager through

SceneManager, without instantiating AudioManager directly. When a collision occurs between player and obstacles, a sound effect is played. Since AudioManager is not directly accessed via CollisionHandler, this prevents unnecessary dependencies between the two, while allowing for real-time auditory feedback for collision.

Design Pattern - Singleton:

At the abstract layer, the AudioManager now follows a Singleton design pattern, by using a getInstance() method. This ensures that there is only one instance that is managing the audio playback across the game. This implementation helps prevent redundant sound management and excessive memory usage.

The SceneManager explicitly receives the AudioManager instance by Dependency Injection, as suggested by the feedback given. This makes accessing audio functionalities easier, while ensuring that there is no redundant instantiation of audio. Implementing AudioManager through this method also prevents circular dependency, which was an issue in Part 1.

IV. Application of OOP and SOLID Principles

The section covers a summary of the principles adopted in our abstract layer and its respective justification.

OOP principles:

Encapsulation:

1. AudioManager:
    a. SoundMap and playingSounds are kept private, preventing direct modification from external classes.
    b. Other classes interact with the AudioManager through methods like playSound(), stopSound().

Abstraction:

1. AudioManager:
    a. The logic of audio management is contained within AudioManager and other classes interact with it through simple methods.

SOLID principles:

<u>Single Responsibility Principle (SRP):</u>

1. AudioManager:
    a. AudioManager is only responsible for managing audio (loading, playing, stopping and disposing of sounds).
2. InputManager:
    a. Only manages the input mapping and key press detection, without handling movement or the UI.

<u>Open/Closed Principle (OCP):</u>

1. AudioManager:
    a. Handles only one job which is audio playback and sound management

<u>Interface Segregation Principle (ISP):</u>

1. IScreen Interface:
    a. Having a custom Screen interface, allows implementations of the essential methods making the overall code to be more focused. This allows each screen to implement only what it needs.

<u>Dependency Inversion Principle (DIP):</u>

1. AudioManager:
   SceneManager receives AudioManager by dependency injection, which then provides it to ObstacleCollisionHandler, instead of directly creating an instance.

V. Game Implementation

In our simulation layer, we incorporated the following design patterns: Factory Method and Strategy Method. Additionally, we introduced new classes across the game management areas as elaborated below.

1. New Entity Classes:

<u>UIManager:</u>
Since our game displays and updates the number of diamonds and the player's health represented by the heart, the UIManager is added to manage the display of the player's health and the diamond entities during gameplay.

SpawningManager:
Our game also spawns moving entities at a random position within the set game boundary. With a spawning Manager, it takes care of the rate at which the entities are spawning.

Entities(Diamond,Obstacle,Power-ups):
The mentioned entities are needed in our simulation layer mainly to interact with the player entity. These entities are also part of the game features that will be elaborated in section VII of the report.

EntityFactory:
EntityFactory is responsible for generating the different entities that are needed by the SpawningManager for the gameplay. This will be further elaborated in subsection 4 of this section on implementing Factory Method.

2. New Scene Classes:

QuestionScreen1:
As our game aims to educate users on scam prevention through questions, we have added a screen to display a multiple choice question for users to select. These questions will appear when the player collides with a scroll and that extra health if the correct answer has been selected.

DiamondQuestionScreen:
Similarly, this screen presents players with more challenging scam prevention questions, which will appear when the player collides with a diamond.

PauseScreen:
The PauseScreen allows players to pause and resume the game anytime they want, allowing them to control their pace for a more flexible learning experience.

WinScreen and GameOverScreen:
An indication to players that they have won or lost the game. They will be able to play the game again by navigating back to the main menu.

3. New Collision Classes:

PlayerObjectCollisionManager:
This new manager registers the new object-specific collision handlers, and calls the appropriate handler depending on what collisions are detected in game.

CollisionHandler:

This class is a new interface that is created to allow the object specific collision handlers to inherit from. The only function available in this interface is handleCollision() that takes in two parameters, Player and Entity.

<u>ObstacleCollisionHandler:</u>
This Obstacle-specific collision handler defines the logic in the handleCollision() function that determines what happens in the game when the Player collides with an Obstacle. If triggered, it reduces the player's health, plays a hit sound effect, and removes the obstacle from the game.

<u>DiamondCollisionHandler:</u>
This Diamond-specific collision handler defines the logic in the handleCollision() function that determines what happens in the game when the Player collides with a Diamond. If triggered, it prompts the user with a question, and if they answer it correctly, they get the diamond. Else, the player loses a health point.

<u>PowerUpCollisionHandler:</u>
The PowerUp-specific collision handler defines the logic in the handleCollision() function that determines what happens in the game when the Player collides with a PowerUp. If triggered, it prompts the user with a question, and if they answer it correctly, they gain one health point. Else, the player does not get any benefits.

4. Design Patterns at Game Layer:

a. Factory Method:

Our game spawns many entities such as the power ups and obstacles during runtime. To ensure efficient and flexible entity creation at runtime, a factory method is implemented.

<u>Implementation:</u>
The entity creation is in their respective entity classes where their random X and Y positions are defined in their respective functions, which will then return an entity object. These functions are called in the EntityFactory class where it will produce the entity as requested by the spawningManager. Once the entities are created, they are then added into the EntityManager to manage the entity in the game layer. Lastly for the entities to spawn randomly and consistently in the game screen, the instance of the spawning manager must be declared and the update() method must be called.

With the implementation of Factory method, the following OOP and SOLID principles were adhered to.

<u>OOP Principles:</u>

1. Encapsulation:
   The entity creation process is hidden from the SpawningManager, which only interacts with the EntityFactory and uses it to receive entities to spawn during the gameplay. Hence, the OOP principle of encapsulation is used.

<u>SOLID Principles:</u>

1. Single Responsibility Principle(SRP):
   Each class has one responsibility where the Spawning Manager is responsible for spawning entities and that the EntityFactory is responsible for producing the entity requested by the spawning Manager. The creation of the entities is handled by their own entity classes. Hence, the single responsibility principle is not violated.

2. Open/Closed principle(OCP):
   If there are more entities to spawn, changes can be made easily by adding on to the EntityFactory and defining the positions of where the entity will spawn in the individual classes. Since changes will only be made to the simulation layer, the open/closed principle is not violated.

b. Strategy Pattern:

In this game, the Strategy Pattern is applied to manage how different types of game object collisions are handled. This design pattern allows for creation of different collision handling behaviors, encapsulating each one within its own class, and enabling them to be interchangeable during gameplay depending on the type of object involved in the collision.

<u>Strategy Interface: CollisionHandler</u>

This interface defines a method for all collision-handling behaviors, ensuring that each concrete strategy class implements a standard method for handling collisions. This promotes consistency and allows the context to interact with any handler in a fixed way.

Concrete Strategies: Diamond,PowerUp,Obstacle CollisonHandlers

Each collision handler represents a specific strategy for handling collisions with a particular type of game object: (as previously mentioned in **Section 3: New Collision Classes of Part V**)

These concrete strategies encapsulate their respective logic independently, allowing each type of collision to be handled in isolation from the others.

Context: PlayerObstacleCollisionManager

The PlayerObstacleCollisionManager acts as the context within the Strategy Pattern. It maintains references to the various collision strategies and is responsible for selecting and applying the appropriate one based on the game object involved in a collision. Instead of embedding all the collision logic directly in the manager, insteads calls the handleCollison() method in the specific collision handler. This separation of concerns makes the collision system more modular and easier to extend or modify.

By implementing the Strategy Pattern, the game's collision handling becomes more flexible and maintainable. New collision types can be supported simply by creating new handler classes that implement the shared interface, without modifying existing logic. This approach adheres to principles such as Open/Closed and Single Responsibility Principle, making the system more robust and scalable as the game evolves.

OOP Principles:

1. Encapsulation:
   Each collision handler encapsulates its own logic for handling collisions, as it contains the specific logic that is unique to the type of collision it is responsible for.

2. Polymorphism:

   Game objects don't need to know how collisions are handled, as they share a common function that needs to be implemented, handleCollision() that can provide their own specific behaviours.

This method can then be overridden by each subclass to provide their own specific collision behaviours.

<u>SOLID Principles:</u>

1. Single Responsibility Principle (SRP):

Each handler is focused only on a single type of collision. The only thing each handler changes or maintains is the behaviour for that specific object collision type. In addition, the manager has one responsibility: managing the collision detection by calling the appropriate handler for the specific game object.

2. Single Responsibility Principle (SRP):

The system is designed to easily allow for new collision handlers to be added with minimal changes to the existing PlayerObjectCollisionManager. For instance, when new entity types are introduced to the game, it is possible to create a new handler for that type and register it in the manager, without altering the core collision manager code.

VI. Innovative Feature in Game Layer

1. Educational Collision-Based Power-Ups

One of the most creative mechanics is how collision is tied with collectables (like scrolls and diamonds) to educational quiz interactions. When the player collects a power-up or diamond, they aren't just awarded instantly. Instead, a question screen will appear where they must answer a scam-awareness question correctly to earn the benefit.

PowerUp → leads to a basic question; rewards health if answered correctly

Diamond → triggers a harder question; rewards a diamond if correct, damages the player if wrong

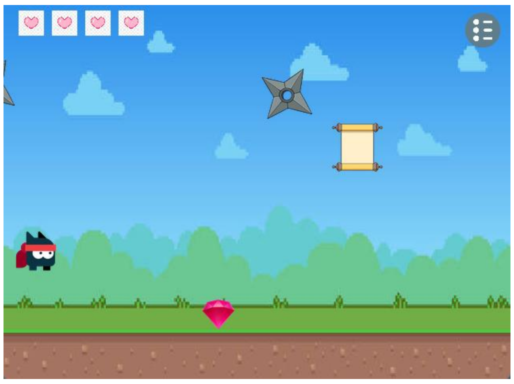
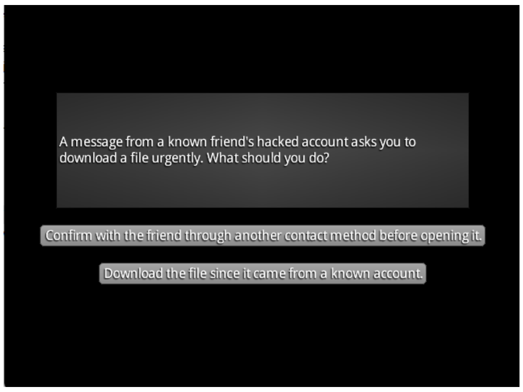


Figure 1:Ninja Clashes with power-ups


A message from a known friend's hacked account asks you to download a file urgently. What should you do?

Confirm with the friend through another contact method before opening it.
Download the file since it came from a known account.

Figure 2: Scam question pop-up

This mechanic not only adds an extra layer of challenge but also promotes learning through gameplay, making the game both fun and informative.

2. Win Condition Based on Smart Collection

Instead of having a traditional score or time-based victory, our game uses a “collect 3 diamonds to win” system , but the twist is that players must also pass a mini-quiz after each diamond collection. This adds depth and purpose to item collection and avoids passive play. Players must actively earn each win condition through understanding.

3. Fully customizable controls

Our settings screen that allows players to rebind their movement keys is a great usability feature. Our game offers this flexibility to cater to different player preferences and accessibility needs.

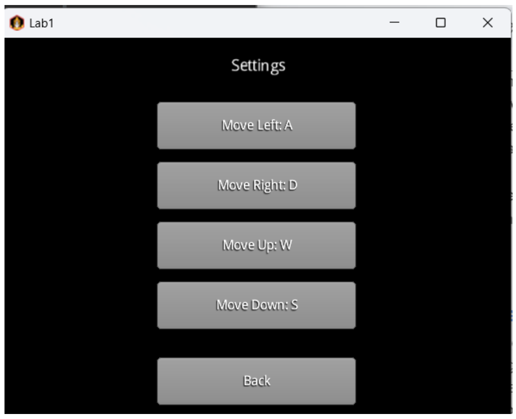


Figure 3: Key Binding Screen

4. Screen Transition System with SceneManager

The use of a custom SceneManager class to handle seamless transitions between transitions of screens. The screen only implements the method from an interface while the SceneManager manages which screen to display, ensuring a clean and organized game state.

5. Health + Diamond UI Display

Through UIManager, our game clearly displays real-time visual feedback with hearts for health and diamonds for progress. This visual cue, placed at intuitive screen locations, keeps players informed without overwhelming them.

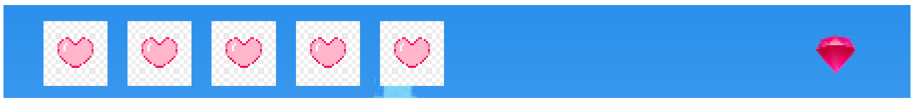


Figure 4: Player at full health and a Diamond

VII. Game limitations

1. Limited Audio Control

Explanation: Currently players are unable to adjust background music or sound effects during the game. For instance, players can't change the volume, switch music tracks, or mute the audio. In other words, players have limited control over their in-game audio experience.
Impact: This can be a problem if a player finds the background music too loud or too repetitive. It also limits the potential for adding a more personalised audio experience, like switching to different tracks during different stages of the game.

2. Lack of Dynamic Difficulty Adjustment

Explanation: The game doesn't automatically adjust its difficulty as the player progresses. This means the game challenge stays the same, regardless of how well the player is doing.
Impact: As players improve, the game may become too easy for them, or if they're struggling, it could remain frustratingly hard. A dynamic difficulty system could make the game feel more responsive to individual players' abilities.

VIII. Reflection of GAI teammate

Pros:

- **Efficient Communication:** The team can interact in individual chat rooms, allowing for focused discussions. While the AI teammate does not analyze the context of the conversations, it remains a valuable resource that we can turn to for specific questions related to the project while having discussions amongst ourselves.

- **Specific Context Suggestions:** The AI teammate has access to this specific project's context data, which enables it to offer relevant and insightful game design suggestions. This makes its responses more tailored to our needs, enhancing the brainstorming process.

Cons:

- **Unreliable Output:** One of the most common issues is that, at times, when the team asks a question, the AI fails to provide any output. This sometimes disrupts the workflow and makes it difficult to rely on the tool consistently.

- **Inconsistent Response Times:** The AI's speed in generating output is quite inconsistent. While it's quick on some occasions, there are times when the responses either take too long or fail to load altogether, leaving us uncertain whether the system is still processing or has encountered an error.

Overall, the GAI teammate proved to be a useful tool for our project. Despite some technical hiccups, with further development and optimization, we believe that it will become a useful resource for students, enhancing both collaboration and learning.

IX. Team Contributions

DONE BY	TASK DESCRIPTION
MUHAMMAD WAFIYUDDIN BIN ABDUL RAHMAN	- Improved on InputManager - Introduced entities on the game UI for better illustration
PAE XIANG SHENG	- Improved AudioManager to use a string path - Implemented Singleton on AudioManager
LIM QI ZHEN	- Implemented factory method to better manage the game entitles - Added a pause screen feature - Added a customized screen interface for sceneManager
TEO WEN TIAN BRENDAN	- Improved the overall Collision system - Introduced new collision handlers along with a new manager to improve management of collisions between objects
MUHAMMAD SAAD BIN HADI	- Introduced new PowerUp, Diamond Entity classes - Added game logic based on these new classes

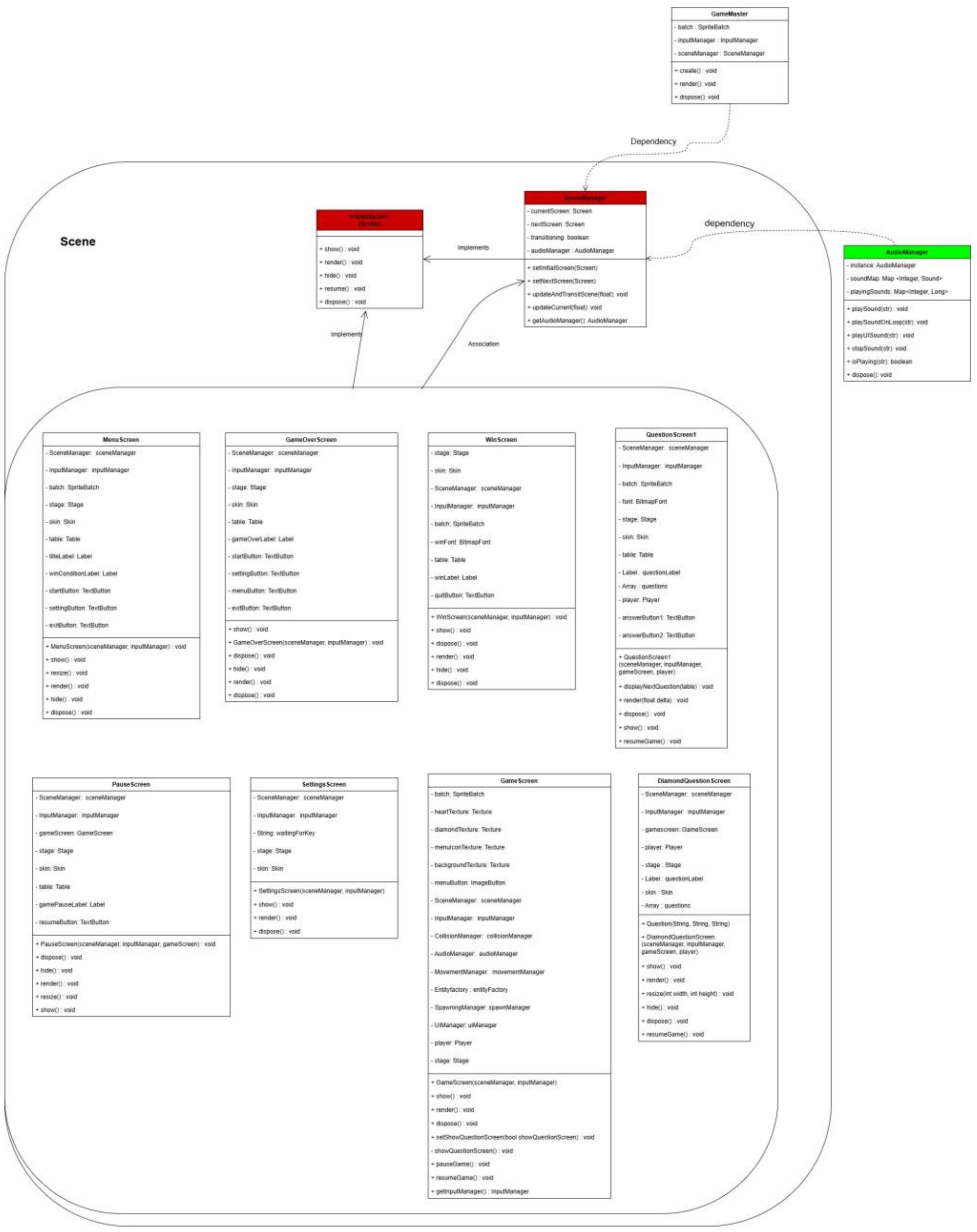
X. Appendix

The diagrams below showcased our UML diagram on a more detailed level, where all the variables and methods are shown. The highlighted entities are part of the abstract layer while the unhighlighted ones are part of the game layer.

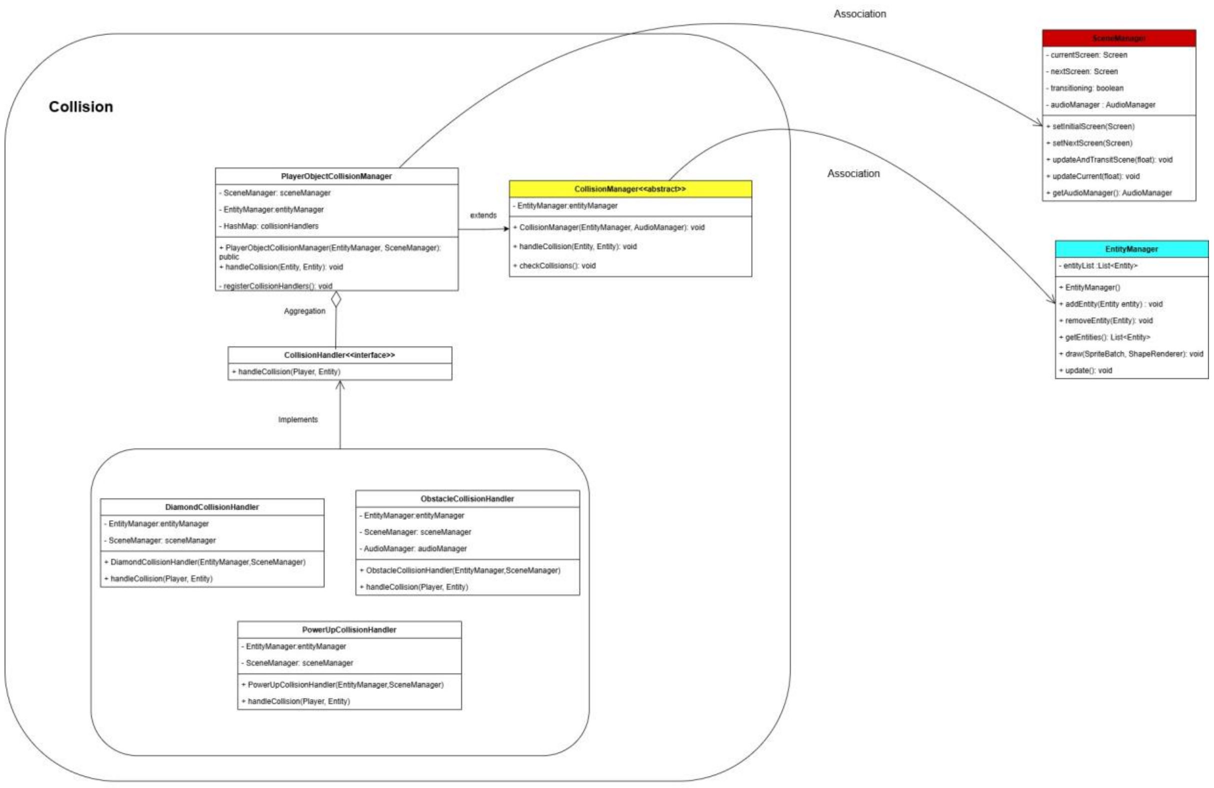
Legends:

Maroon Red:	Scene
Green:	Input/Output
Yellow:	Collision
Sky Blue:	Entity
unhighlighted:	Outside of abstract engine

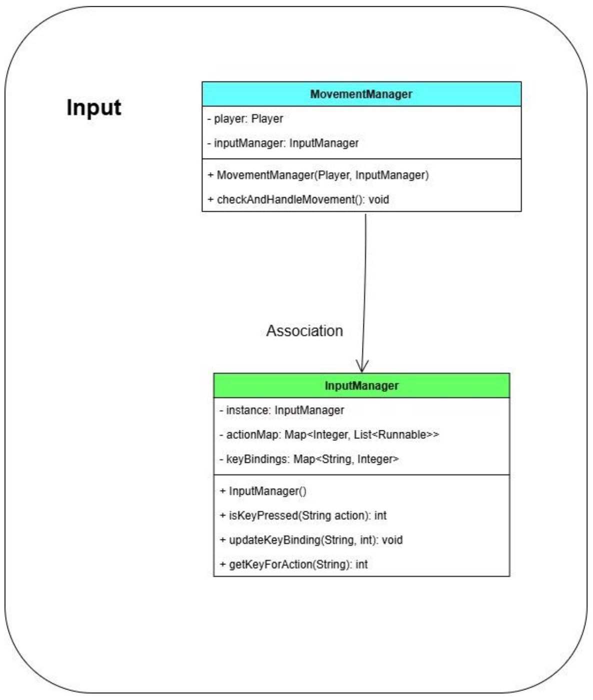
Scene:


GameMaster
- batch : SpriteBatch
- inputManager : InputManager
- sceneManager : SceneManager
+ create() : void
+ render(): void
+ dispose(): void
Dependency
Scene
<<Interface>>
IScreen
+ show() : void
+ render() : void
+ hide() : void
+ resume() : void
+ dispose() : void
Implements
SceneManager
- currentScreen: Screen
- nextScreen: Screen
- transitioning: boolean
- audioManager : AudioManager
+ setInitialScreen(Screen)
+ setNextScreen(Screen)
+ updateAndTransitScene(float): void
+ updateCurrent(float): void
+ getAudioManager(): AudioManager
dependency
AudioManager
- instance: AudioManager
- soundMap: Map <Integer, Sound>
- playingSounds: Map<Integer, Long>
+ playSound(str) : void
+ playSoundOnLoop(str): void
+ playUISound(str) : void
+ stopSound(str): void
+ isPlaying(str): boolean
+ dispose(): void
Implements
Association
MenuScreen
- SceneManager: sceneManager
- InputManager: inputManager
- batch: SpriteBatch
- stage: Stage
- skin: Skin
- table: Table
- titleLabel: Label
- winConditionLabel: Label
- startButton: TextButton
- settingButton: TextButton
- exitButton: TextButton
+ MenuScreen(sceneManager, inputManager) : void
+ show() : void
+ resize() : void
+ render() : void
+ hide() : void
+ dispose() : void
GameOverScreen
- SceneManager: sceneManager
- InputManager: inputManager
- stage: Stage
- skin: Skin
- table: Table
- gameOverLabel: Label
- startButton: TextButton
- settingButton: TextButton
- menuButton: TextButton
- exitButton: TextButton
+ show() : void
+ GameOverScreen(sceneManager, inputManager) : void
+ dispose() : void
+ hide() : void
+ render() : void
+ dispose() : void
WinScreen
- stage: Stage
- skin: Skin
- SceneManager: sceneManager
- InputManager: inputManager
- batch: SpriteBatch
- winFont: BitmapFont
- table: Table
- winLabel: Label
- quitButton: TextButton
+ WinScreen(sceneManager, inputManager) : void
+ show() : void
+ dispose() : void
+ render() : void
+ hide() : void
+ dispose() : void
QuestionScreen1
- SceneManager: sceneManager
- InputManager: inputManager
- batch: SpriteBatch
- font: BitmapFont
- stage: Stage
- skin: Skin
- table: Table
- Label : questionLabel
- Array : questions
- player: Player
- answerButton1: TextButton
- answerButton2: TextButton
+ QuestionScreen1 (sceneManager, inputManager, gameScreen, player)
+ displayNextQuestion(table) : void
+ render(float delta) : void
+ dispose() : void
+ show() : void
+ resumeGame() : void
PauseScreen
- SceneManager: sceneManager
- InputManager: inputManager
- gameScreen: GameScreen
- stage: Stage
- skin: Skin
- table: Table
- gamePauseLabel: Label
- resumeButton: TextButton
+ PauseScreen(sceneManager, inputManager, gameScreen) : void
+ dispose() : void
+ hide() : void
+ render() : void
+ resize() : void
+ show() : void
SettingsScreen
- SceneManager: sceneManager
- InputManager: inputManager
- String: waitingForKey
- stage: Stage
- skin: Skin
+ SettingsScreen(sceneManager, inputManager)
+ show() : void
+ render() : void
+ dispose() : void
GameScreen
- batch: SpriteBatch
- heartTexture: Texture
- diamondTexture: Texture
- menuIconTexture: Texture
- backgroundTexture: Texture
- menuButton: ImageButton
- SceneManager: sceneManager
- InputManager: inputManager
- CollisionManager: collisionManager
- AudioManager: audioManager
- MovementManager: movementManager
- Entityfactory : entityFactory
- SpawningManager: spawnManager
- UIManager: uiManager
- player: Player
- stage: Stage
+ GameScreen(sceneManager, inputManager)
+ show() : void
+ render() : void
+ dispose() : void
+ setShowQuestionScreen(bool showQuestionScreen) : void
- showQuestionScreen() : void
+ pauseGame() : void
+ resumeGame() : void
+ getInputManager() : InputManager
DiamondQuestionScreen
- SceneManager: sceneManager
- InputManager: inputManager
- gamescreen: GameScreen
- player: Player
- stage : Stage
- Label : questionLabel
- skin : Skin
- Array : questions
+ Question(String, String, String)
+ DiamondQuestionScreen (sceneManager, inputManager, gameScreen, player)
+ show() : void
+ render() : void
+ resize(int width, int height) : void
+ hide() : void
+ dispose() : void
+ resumeGame() : void

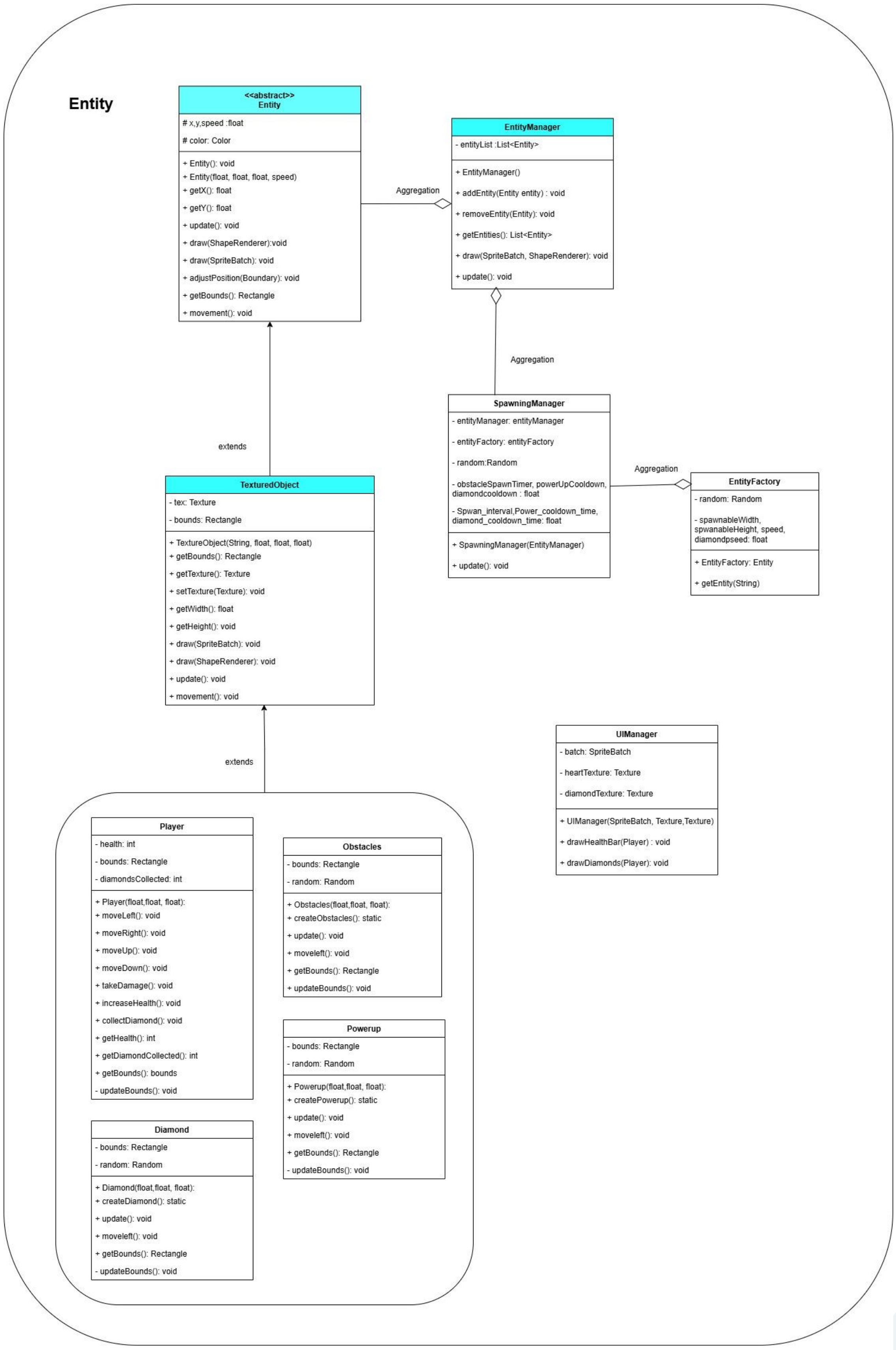
Collision:


Association
SceneManager
- currentScreen: Screen
- nextScreen: Screen
- transitioning: boolean
- audioManager : AudioManager
+ setInitialScreen(Screen)
+ setNextScreen(Screen)
+ updateAndTransitScene(float): void
+ updateCurrent(float): void
+ getAudioManager(): AudioManager
Collision
Association
PlayerObjectCollisionManager
- SceneManager: sceneManager
- EntityManager:entityManager
- HashMap: collisionHandlers
+ PlayerObjectCollisionManager(EntityManager, SceneManager): public
+ handleCollision(Entity, Entity): void
- registerCollisionHandlers(): void
extends
CollisionManager<<abstract>>
- EntityManager:entityManager
+ CollisionManager(EntityManager, AudioManager): void
+ handleCollision(Entity, Entity): void
+ checkCollisions(): void
EntityManager
- entityList :List<Entity>
+ EntityManager()
+ addEntity(Entity entity) : void
+ removeEntity(Entity): void
+ getEntities(): List<Entity>
+ draw(SpriteBatch, ShapeRenderer): void
+ update(): void
Aggregation
CollisionHandler<<interface>>
+ handleCollision(Player, Entity)
Implements
DiamondCollisionHandler
- EntityManager:entityManager
- SceneManager: sceneManager
+ DiamondCollisionHandler(EntityManager,SceneManager)
+ handleCollision(Player, Entity)
ObstacleCollisionHandler
- EntityManager:entityManager
- SceneManager: sceneManager
- AudioManager: audioManager
+ ObstacleCollisionHandler(EntityManager,SceneManager)
+ handleCollision(Player, Entity)
PowerUpCollisionHandler
- EntityManager:entityManager
- SceneManager: sceneManager
+ PowerUpCollisionHandler(EntityManager,SceneManager)
+ handleCollision(Player, Entity)

Input:


Input
MovementManager
- player: Player
- inputManager: InputManager
+ MovementManager(Player, InputManager)
+ checkAndHandleMovement(): void
Association
InputManager
- instance: InputManager
- actionMap: Map<Integer, List<Runnable>>
- keyBindings: Map<String, Integer>
+ InputManager()
+ isKeyPressed(String action): int
+ updateKeyBinding(String, int): void
+ getKeyForAction(String): int

Entity:


Entity
<<abstract>>
Entity
# x,y,speed :float
# color: Color
+ Entity(): void
+ Entity(float, float, float, speed)
+ getX(): float
+ getY(): float
+ update(): void
+ draw(ShapeRenderer):void
+ draw(SpriteBatch): void
+ adjustPosition(Boundary): void
+ getBounds(): Rectangle
+ movement(): void
Aggregation
EntityManager
- entityList :List<Entity>
+ EntityManager()
+ addEntity(Entity entity) : void
+ removeEntity(Entity): void
+ getEntities(): List<Entity>
+ draw(SpriteBatch, ShapeRenderer): void
+ update(): void
Aggregation
extends
SpawningManager
- entityManager: entityManager
- entityFactory: entityFactory
- random:Random
- obstacleSpawnTimer, powerUpCooldown, diamondcooldown : float
- Spwan_interval,Power_cooldown_time, diamond_cooldown_time: float
+ SpawningManager(EntityManager)
+ update(): void
Aggregation
EntityFactory
- random: Random
- spawnableWidth, spwanableHeight, speed, diamondpseed: float
+ EntityFactory: Entity
+ getEntity(String)
TexturedObject
- tex: Texture
- bounds: Rectangle
+ TextureObject(String, float, float, float)
+ getBounds(): Rectangle
+ getTexture(): Texture
+ setTexture(Texture): void
+ getWidth(): float
+ getHeight(): void
+ draw(SpriteBatch): void
+ draw(ShapeRenderer): void
+ update(): void
+ movement(): void
extends
UIManager
- batch: SpriteBatch
- heartTexture: Texture
- diamondTexture: Texture
+ UIManager(SpriteBatch, Texture,Texture)
+ drawHealthBar(Player) : void
+ drawDiamonds(Player): void
Player
- health: int
- bounds: Rectangle
- diamondsCollected: int
+ Player(float,float, float):
+ moveLeft(): void
+ moveRight(): void
+ moveUp(): void
+ moveDown(): void
+ takeDamage(): void
+ increaseHealth(): void
+ collectDiamond(): void
+ getHealth(): int
+ getDiamondCollected(): int
+ getBounds(): bounds
- updateBounds(): void
Obstacles
- bounds: Rectangle
- random: Random
+ Obstacles(float,float, float):
+ createObstacles(): static
+ update(): void
+ moveleft(): void
+ getBounds(): Rectangle
+ updateBounds(): void
Powerup
- bounds: Rectangle
- random: Random
+ Powerup(float,float, float):
+ createPowerup(): static
+ update(): void
+ moveleft(): void
+ getBounds(): Rectangle
- updateBounds(): void
Diamond
- bounds: Rectangle
- random: Random
+ Diamond(float,float, float):
+ createDiamond(): static
+ update(): void
+ moveleft(): void
+ getBounds(): Rectangle
- updateBounds(): void