SCHOOL OF INFOCOMM TECHNOLOGY

INF1009 Object-Oriented Programming

GROUP ASSIGNMENT 1

AY2022/2023 TRIMESTER 2

172 Ang Mo Kio Avenue 8 Singapore 567739

Team P2 Group 9

Student Name & E-mail	**Nik Mohammad Farhan Bin Azmi** 2201237@sit.singaporetech.edu.sg
	Mohammad Afiq 2200546@sit.singaporetech.edu.sg
	Tham Weng Cher 2203544@sit.singaporetech.edu.sg
	Ng Yan Qing 2201506@sit.singaporetech.edu.sg
	Esther Chew Hui Qing 2201990@sit.singaporetech.edu.sg

Object-Oriented Programming: Team Project Part 2

ESTHER CHEW HUI QING (2201990), MOHAMMAD AFIQ (2200546), NG YAN QING (2201506),
NIK MOHAMMAD FARHAN BIN AZMI (2201237), THAM WENG CHER (2203544)

ICT, Singapore Institute of Technology

***Abstract*— This paper provides a comprehensive analysis of the overall system design of a game and game engine, focusing on the purpose and functionality of the different components and layers. The report also examines the implementation of design patterns in the engine and the game, highlighting their impact on performance and scalability. Additionally, the report discusses the improvements made to the game engine since part 1 of the project, with emphasis on the object-oriented principles applied to both the engine and the game. Limitations with the game and game engine are also discussed, and recommendations for future development are made. Moreover, the report highlights several design innovations within the game, including unique features and gameplay mechanics that enhance the user experience.**

***Keywords* — LibGDX, Java, game engine, design patterns, architecture patterns, layers, components, packages, game development, object-oriented programming, OOP, abstraction, inheritance, encapsulation, composition, interfaces, generics**

I. Introduction

The objective of this assignment is to create an entertaining game for children utilising Java's LibGDX game library. In this report, the team will detail the system design of the game, including enhancements and updates made to the engine from the previous project phase. Additionally, the team will explain the object-oriented programming principles and design patterns implemented in the game. Furthermore, the team will emphasise the educational features of the game, aimed at piquing children's interest in space exploration.

II. Overall System Design

The game and game engine have been created using the Java programming language and the LibGDX game library. The system design for both the game and the game engine is crucial in ensuring a successful game. This section will discuss the overall system design, including the purpose of every layer and component, as well as design patterns implemented. Additionally, the improvements made to the game engine since the first project phase will also be highlighted.

A. Overview and Purpose of Components and Layers

The system is made up of two layers - the game engine layer and the game layer. In LibGDX, this was implemented using packages. As illustrated in Fig. 1, the game engine layer is a package that contains manager components, such as behaviour, collision, entity, input, lifecycle, screen, and sound managers. Fig 1 also illustrates the game layer, containing several components including contextual game components, game logic, and screens.

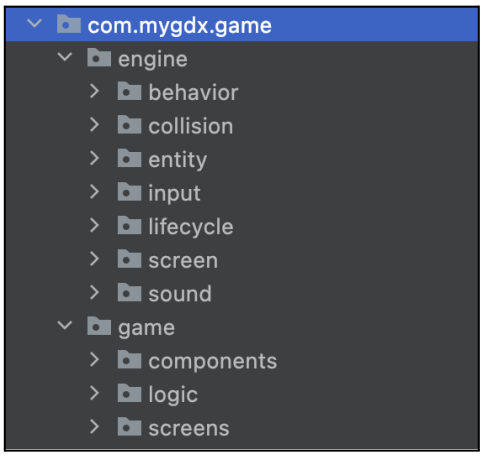

Fig 1. Game and Engine layering.

1. Game Engine Layer

The first layer of the overall system is the game engine. The game engine layer (engine package) consists of several managerial components pertaining to the game engine. The components (capitalised) are as follows:

a. Behaviour: The Behaviour component is an essential component of the game engine layer, specifically designed to encapsulate the interface related to the behaviour of non-player entities. The Behaviour package includes the BehaviourManager interface, which defines the methods governing the actions of non-player entities. These methods, such as moveUFO() and dropPlanet(), help regulate the behaviour of these entities within the game.

b. Collision: The Collision component is responsible for encapsulating the classes or interfaces related to managing collisions between entities and the "collidable" entities themselves. This package includes the CollisionManager interface, which contains method definitions for detecting collisions between entities, as well as the CollidableEntity class that implements the CollisionManager interface.

c. Entity: The Entity component encapsulates the Entity abstract class and the EntityManager class, used for managing the state of the various entities displayed to the user in the game.

d. Input: The Input component encapsulates the input manager of the game engine, in charge of handling user inputs. In the case of this game engine, the Input component contains the CustomInputProcessor class, which implements the InputProcessor interface, allowing the class to define its own implementation of user input methods such as keyDown().

e. Lifecycle: The Lifecycle package encapsulates the Main class, which is the class responsible for handling the overall lifecycle of the game.

f. Screen: The screen package contains the ScreenManager class, in charge of creation and managing the states of all screens.

g. Sound: This component contains the class that manages the sounds such as in-game music and sound effects.

2. Game Layer

The game layer, which forms the second layer of the system, is where the actual game is designed and implemented. Unlike the game engine layer, this layer uses the tools provided by the game engine to incorporate additional functionality related to the game, including game logic and the various elements that make up the game world. The game layer consists of the following components:

1. Components: Contains the various classes representing the different game entities that make up the game world, such as objects and backgrounds. It also includes non-game related components such as the buttons used in the game and menus.

2. Logic: Contains the class that pertains to the game's logic and behaviour, such as movement and interactions between entities in-game.

3. Screens: Contains classes pertaining to the various screens used in the game such as main menu, gameplay and game over screens.

B. Implementation of Design Patterns

The team has implemented two software design patterns into the game engine, namely, the Singleton and Factory creational design patterns.

The Singleton design pattern has been employed in the Main class of the game engine to ensure that only one instance of the class can exist throughout the application. This design pattern restricts the instantiation of a class to a single instance, and the Main class serves as the Singleton class in this case. The constructor of the Main class has been given the private access modifier to prevent the instantiation of the class from outside the class. Instead, the getInstance() method has been used to obtain an instance of the Main class, which checks if an instance already exists. If an instance does not exist, a new instance is created. This approach ensures that only one instance of the Main class can exist, which in turn prevents the creation of multiple lifecycle managers and other managers, which could consequently lead to unexpected behaviour in the game engine. The code snippet illustrating the Singleton design pattern is shown in Fig 2.

```
1 usage   Farhan Azmi
private Main() { }

/*
    A static method to get the instance. If instance is null (No Main instance present),
    create a new Main instance.
*/
1 usage   Farhan Azmi
public static Main getInstance() {
    if (instance == null) {
        instance = new Main();
    }
    return instance;
}
```

Fig 2. Singleton design pattern applied to Main class.

Moreover, the EntityManager class utilises the Factory creational design pattern to create enemy entities in the game. The pattern is implemented through the spawnEnemy() method, which can create either "Asteroid" or "UFO" entities based on the passed in entityType (string) parameter. If the entityType is "Asteroid," the method creates an Asteroid object and adds it to an ArrayList of Asteroid entities. Similarly, if entityType is "UFO," one or more UFO objects are created and added to an ArrayList of UFO entities. The CollidableEntity class is utilised in both cases to encapsulate the created objects, enabling additional functionality like collision detection. Figure 3 shows a code snippet that illustrates the application of this design pattern.

```
public ArrayList spawnEnemy(String entityType) {
    if (entityType.equals("Asteroid")) {
        Random random = new Random();
        int[] possibleX = {250, 250, 300, 350, 400, 450, 650};
        int chance = random.nextInt(possibleX.length);
        Asteroid asteroid = new Asteroid( imgPath: "asteroid.png");
        CollidableEntity<Asteroid> asteroidEntity = new CollidableEntity<Asteroid>(
                possibleX[chance],
                 y: 800,
                asteroid,  width: 64,  height: 64);
        asteroids.add(asteroidEntity);
        return asteroids;
    } else if (entityType.equals("UFO")) {
        Random random = new Random();
        int max = 5;
        ArrayList<Integer> possibleX = new ArrayList<Integer>();
        Integer[] elementsToAdd = {100, 200, 300, 400, 500, 600, 700};
        possibleX.addAll(Arrays.asList(elementsToAdd));
        int numUFO = random.nextInt(max) + 1;
        for (int i = 0; i < numUFO; i++) {
            UFO ufoObject = new UFO( imgPath: "alien.png");
            int x = possibleX.get(random.nextInt(possibleX.size()));
            if (possibleX.contains(x)) {
                int index = possibleX.indexOf(x);
                possibleX.remove(index);
            }
            int y = 460; // Put beyond the screen first
            CollidableEntity<UFO> ufoEntity = new CollidableEntity<UFO>(
                    x,
                    y,
                    ufoObject,  width: 64,  height: 64);
            UFOs.add(ufoEntity);
        }
        return UFOs;
    }
    return null;
```

Fig 3. Code snippet illustrating the use of factory pattern

There are several reasons for utilising the factory pattern for the spawnEnemy() method:

1. Encapsulation: The factory method encapsulates the object creation logic, hiding it from the client code. This allows for a cleaner, more maintainable codebase.

2. Flexibility: By using the factory method, it is possible to change the underlying implementation of the objects being created without changing the client code. This allows for greater flexibility in the design of the system, allowing dynamic instantiation of objects based on certain conditions, such as user input, which makes it more flexible than traditional object creation methods.

3. Extensibility: The factory method design pattern allows for easy extension of the system, as new object types can be added by creating new factory classes that implement the factory interface. In the context of the spawnEnemy() method, the method is able to easily spawn more enemy entities into the game by simply calling the method, be it an “asteroid” entity or a “UFO” entity.

4. Decoupling: The factory method design pattern decouples the client code from the actual implementation of the objects being created, which can help reduce dependencies and make the codebase more modular.

III. Improvements Made To Game Engine

The team has incorporated several improvements and additions into the game engine since the first phase of the project. The modifications and additions are listed in this section.

1. Sound Manager (SoundManager class)

The implementation of a sound manager class allows for the efficient handling of audio resources within the game, to enhance user experience and immersion in the game through the addition of background music and special effects sounds. The SoundManager class contains the methods such as *playMusic* which takes a *ScreenType* parameter to play background music based on the screen displayed and *stopMusic* which stops any currently playing music to make sure only one background music is playing at a time.

2. Screen Manager (ScreenManager class)

The ScreenManager class underwent significant changes resulting in a more efficient approach to screen management. Specifically, the ScreenManager class now fully owns the Screen classes, unlike before where the Main (Lifecycle Manager) class owned them. As a result, the Main class only needs to instantiate the ScreenManager class and call its instantiateScreen() method to create the screens. The updated code snippet depicting the new ownership of screens (under the instantiateScreens() method) is illustrated in Fig 4.

```
public void instantiateScreens() {
    // Create screens
    mainMenuScreen = new MainMenuScreen(this.game,  screenManager: this);
    pauseScreen = new PauseScreen(this.game,  screenManager: this);
    gameOverScreen = new GameOverScreen(this.game);
    scoreboardScreen = new ScoreboardScreen(this.game);
    controlScreen = new ControlScreen(this.game,  imgPath: "controls.jpg");
    gameScreen = new GameScreen(this.game);
```

Fig 4. instantiateScreens() method creates the screens in ScreenManager class.

Furthermore, the ScreenManager class now offers an expanded range of features that enable the creation of engaging "Storyboards" within the game. These Storyboards include cutscenes that heighten the player's overall experience and enhance the game's narrative elements. The newly added method, generateStoryboards(), requires two parameters: an ArrayList of image paths (in string format) and a string specifying the type of storyboard to be generated. This feature allows developers to generate specific types of screens such as pre-game cutscenes or in-game fact screens. The implementation of the generateStoryboards() method is illustrated in Fig. 5.

```
public ArrayList<StoryboardScreen> generateStoryboards(ArrayList<String> imgPaths, String type) {
    ArrayList<StoryboardScreen> storyboards = new ArrayList<>();
    for (int i = 0; i < imgPaths.size(); i++) {
        StoryboardScreen storyboard = new StoryboardScreen(this.game, imgPaths.get(i), type);
        storyboard.setCurrent(i);
        storyboards.add(storyboard);
    }
    return storyboards;
}
```

Fig 5. generateStoryboards() method

3. Collision Manager (CollisionManager interface)

The Collision Manager retains its role as a CollisionManager interface, as in the first phase of the project. However, instead of having a method declaration that specifically checks for collision between the player entity and a falling object entity, the CollisionManager interface now includes only one method declaration, checkCollision(), which checks collision between a player and another entity. This method takes two parameters, both of which must be instances of CollidableEntity (with the first having a bound of Player), and returns a boolean value indicating whether a collision has occurred (true) or not (false). In Fig. 6, the code snippet for the CollisionManager interface during the first phase of the project can be seen. In contrast, Fig. 7 displays the modified CollisionManager interface code snippet.

```
public interface CollisionManager<P, F, E> {
    // This class contains the methods to which the entities behave when a collision occurs.
    void limitPlayerMovement(E screenWidth, E screenHeight);
    boolean checkFallingObjectCollision(P player, F fallingObject);
}
```

Fig. 6. CollisionManager interface in Project Part 1

```
public interface CollisionManager {
    4 usages  1 implementation  Farhan Azmi
    boolean checkCollision(CollidableEntity<Player> player, CollidableEntity other);
}
```

Fig. 7. Updated CollisionManager interface

4. Behaviour Manager (BehaviourManager interface)

The Behaviour Manager has been expanded to provide additional functionality for non-player entities, building upon the first phase of the project where it remained an interface with only one method declaration, moveFallingObject(), allowing movement for non-player entities. The interface has now been augmented to incorporate new non-player entity behaviours. These include enemy UFOs firing at the player through the fireWeapon() method, UFO movement through the moveUFO() method, and the ability for Asteroid and Planet entities to drop through the dropPlanet() and dropAsteroid() methods respectively.

IV. UML Diagrams of Game Engine and Game

The complete UML diagram for the Game Engine and Game can be seen in the Appendix section of this report.

V. Object-Oriented Principles Adopted

This section covers the object-oriented programming (OOP) principles that have been implemented in the Game Engine and Game, with the team providing detailed explanations and elaboration.

A. Use of OOP principles in the Game Engine

The game engine utilises several OOP principles, and they are as follows:

1. Inheritance

The concept of inheritance is used in many components within the game engine, including the Main class that extends from the Game class (LibGDX library). This can be seen in Fig 8. Using inheritance allows the team to create a new class that is a modified version of an existing class. In this case, the Main class inherits from the Game class, which means that it gets all of the methods and properties defined in the Game class, including the create() and render() methods. This allows the reuse of

existing code and avoids duplicating it in every class that needs it.

Additionally, using inheritance makes the code more modular and easier to maintain. By separating out the common functionality into a base class like Game, the team is able to avoid duplicating code across multiple classes and ensure that any changes or bug fixes to that code are reflected in all of the classes that inherit from it.

```
public class Main extends Game {
```

Fig 8. The Main class inherits from the Game class (LibGDX library).

2. Encapsulation

The concept of encapsulation is applied to various components in the game engine. In the case of the ScreenManager class, encapsulation is used to protect the ScreenManager class's internal state by making the instance variables (storyboardImgPath, planetVisitImgPath, storyboards, visitPlanetStoryboards, mainMenuScreen, pauseScreen, gameOverScreen, scoreboardScreen, controlScreen, gameScreen) **private**. This means that the variables cannot be accessed directly from outside the ScreenManager class.

3. Abstraction

The concept of abstraction is an OOP principle that allows for hiding the implementation details of a class from external classes and provides a simplified interface for a user to interact with the class. For example, the ScreenManager class contains getter and setter methods for its private variables such as the getPauseScreen() and setPauseScreen() allowing external classes to access and modify the state of these variables without knowing the details of how the fields are implemented.

Abstraction is also achieved through the use of interfaces in various managers, including the BehaviourManager and the CollisionManager. The CollisionManager interface defines a method called checkCollision(), which must be implemented by any class that implements this interface. The responsibility of implementing the method lies with the class that implements the interface, enabling a higher level of abstraction in the code.

4. Generics

Generics are employed in the EntityManager class of the game engine. This class comprises various ArrayLists that contain CollidableEntity, a generic type that can be bound to either Player, Asteroid, or UFO types. The reason for the use of generics in the EntityManager ensures type safety, as the collections (ArrayLists) can only contain instances of the specified generic type, reducing the errors that can occur when working with collections of different types.

```
public class EntityManager {
    2 usages
    private ArrayList<CollidableEntity<Player>> players;
    7 usages
    private ArrayList<CollidableEntity<Asteroid>> asteroids;
    4 usages
    private ArrayList<CollidableEntity<UFO>> UFOs;
```

Fig 9. EntityManager holds collections of generics with specified bounds (Player, Asteroid or UFO)

5. Association (Composition)

Composition (strong association) is used in several components of the game engine. For example, the Main class has several game managerial instances such as the ScreenManager, EntityManager and SoundManager. Using composition allows the Main class to delegate responsibilities (such as managing screens and entities) to other classes while maintaining a high level of abstraction. This makes the code more maintainable, easier to understand and modify since each class is responsible for their respective tasks. The code snippet illustrating the use of composition in the Main class is illustrated in Fig. 10.

```
public class Main extends Game {
    11 usages
    private ScreenManager screenManager;
    4 usages
    private SoundManager soundManager;
    42 usages
    public EntityManager entityManager;
```

Fig 10. Illustrating composition in the Main class.

Another example of the use of composition in the game engine is exhibited in the ScreenManager class. The ScreenManager class is responsible for the management of the screens, and as such has ownership over the individual Screen classes. The use of composition promotes flexibility in screen management and simplifies the codebase by isolating screen-specific logic into this class. The code snippet showing the use of composition in the ScreenManager class is illustrated in Fig. 11.

```
public class ScreenManager {
    8 usages
    private ArrayList<String> storyboardImgPath;
    11 usages
    private ArrayList<String> planetVisitImgPath;
    3 usages
    private ArrayList<StoryboardScreen> storyboards;
    3 usages
    private ArrayList<StoryboardScreen> visitPlanetStoryboards;
    3 usages
    private MainMenuScreen mainMenuScreen;
    3 usages
    private PauseScreen pauseScreen;
    3 usages
    private GameOverScreen gameOverScreen;
    3 usages
    private ScoreboardScreen scoreboardScreen;
    3 usages
    private ControlScreen controlScreen;
    3 usages
    private GameScreen gameScreen;
```

Fig 11. Illustrating composition in the ScreenManager class.

B. Use of OOP principles in the Game

In addition to the game engine, the game also employs various OOP principles, which are listed below:

1. Implementation of interfaces

In the game layer, there are several classes that implement interfaces defined in the game engine layer. One example is the Asteroid class, which is an in-game entity and implements the BehaviourManager interface. This enables it to implement behaviours related to the Asteroid class, such as the dropAsteroid() method. A simplified UML diagram illustrating the relations between the Asteroid and the BehaviourManager interface is shown in Fig 12.

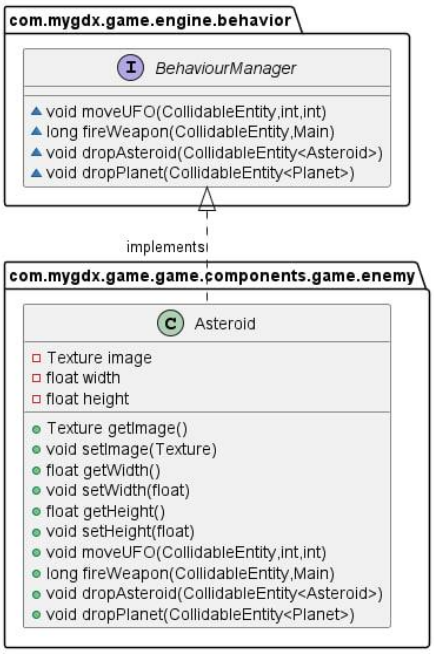

Fig 12. UML diagram illustrating the Asteroid implementing the BehaviourManager interface

2. Encapsulation

Encapsulation is a crucial concept used in the game layer as well. Take, for instance, the Player class, which encapsulates variables such as the player's width, height, current and maximum health, and texture, using the **private** access modifier. By doing so, the internal state of the variables remains secure and can only be accessed by the class itself.

3. Abstraction

In the game layer, abstraction is also accomplished by setting class variables as private and providing access and modification through getter and setter methods. This approach enables the internal workings of the class to be hidden and

allows for more control over how the variables are accessed and modified. By using getter and setter methods, the class can validate and control the values that are being assigned to its variables, providing an additional layer of abstraction to the code.

4. Association (Composition)

In the game layer, composition is utilised in various classes, particularly those related to in-game entities such as Player, UFO, Asteroid, and Planet. Each of these entities possesses a Texture instance, which is essential for displaying the entity on the screen. This exemplifies the application of composition in the game layer.

VI. Reflections on Limitations with Engine and Game

A. Limitations to Game Engine

1. Only supports mouse/keyboard inputs

Currently, the game only supports mouse and keyboard inputs for gameplay, which limits the compatibility with devices that do not support these input methods. This could make the game inaccessible to children who may not have access to a computer with these input devices or prefer to play on a different platform such as a mobile device that has a touchscreen interface. It is noted that additional input processors can be added (such as ones that permit touch inputs) to achieve this.

2. Simplification of BehaviourManager interface

To further simplify the BehaviourManager interface, it is proposed to merge the dropAsteroid() and dropPlanet() methods into a single method called dropEntity(), as they exhibit similar behaviours. This approach can reduce code duplication and make the code more efficient. Therefore, the updated BehaviourManager should only contain three methods: moveUFO(), fireWeapon(), and dropEntity().

B. Limitations to Game

1. Only supports single player

The game currently only supports single player game mode. To provide a more engaging and interactive experience, implementing a multiplayer game mode can also be taken into consideration for future plans. In order to do so, the game engine can be expanded to handle the increased number of players, along with their respective data.

2. Lack of preferences or settings

The game currently lacks the functionality for players to adjust important settings such as brightness, audio, and control settings that can significantly impact their experience. Providing players with the ability to customise these settings can greatly enhance the game's accessibility and appeal to a wider audience of children.

3. Possible improvements for gaining children's interest in Space Exploration

The current objective of the game is to foster children's interest in space exploration. To achieve this, the game puts the player in the role of an astronaut and tasks them with navigating through space while also taking care of their ship and avoiding hazards such as alien attacks and asteroid collisions. As the player progresses, they encounter more planets within the solar system and are provided with interesting facts about each planet through a narrative-driven approach when they successfully land on the planet. However, there is room for further immersion in the game's environment by enhancing the planet exploration experience. This could be achieved through modifications such as changing the background or implementing more challenging ship controls, taking into account environmental factors such as strong winds on some planets.

VII. Design Innovations Within The Game

A. Health Bar

In the game, a "health bar" is displayed, allowing the player to be aware of the health of the ship's hull. The length and colour of the bar determines the ship's hull integrity. If the player incurs damage to the ship by either getting shot by the UFO's lasers or colliding with an asteroid, the health bar decreases in length and changes colour to orange or red should it fall below the 66% and 33% threshold respectively. The health bar is drawn onto the screen using the ShapeRenderer library. The health bar can be seen on the top left-hand corner of Fig. 13.

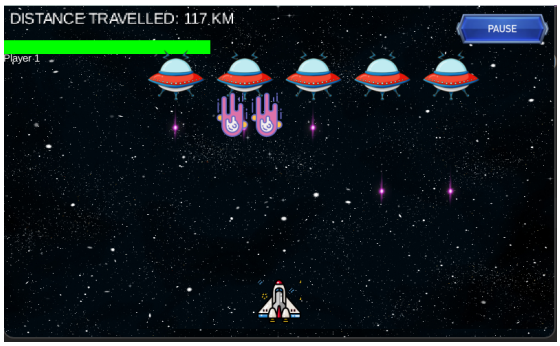

Fig 13. HealthBar

B. *Scoreboard*

The Scoreboard is a feature that allows the player to monitor important information such as the game's start date and time, total distance travelled in kilometres, and the number of defeated alien entities. The recorded scores are conveniently stored in an XML file generated by the Libgdx Class Preferences, which is automatically saved in the user's home directory. For Windows users, the file can be found in %UserProfile%/.prefs, while for Mac users, it is located in ~/.prefs/. The illustration of the Scoreboard in-game is shown in Fig 14.

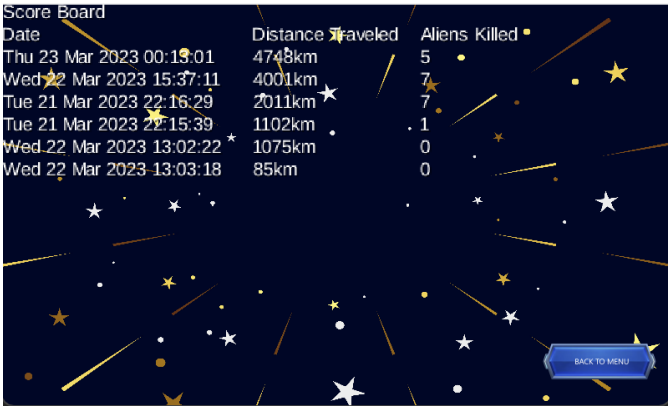

Fig 14. ScoreboardScreen

C. *UFO Spawning Behaviour*

When the UFO enters the game, it does not simply appear on the screen. Rather, it descends from the top of the screen, providing the player with a sense of immersion and making it appear as though the UFO arrived from beyond the bounds of space.

The implementation of this behaviour is possible due to the moveUFO() method, which calls the moveDown() method, which shifts the UFO's position down only if it is above 330 vertical pixels. The moveUFO() method is called in the GamePlay class, under the **game.logic** package.

VIII. Team Contribution

The team contribution table outlining the contribution of each team member is listed in Table I.

Table I

Team Member	Team Contribution
ESTHER CHEW HUI QING	1. SoundManager class
MOHAMMAD AFIQ	1. Health bar implementation 2. Implementation of in-game logic (GamePlay class) 3. Behaviour of Asteroid entity
NG YAN QING	1. ScoreboardScreen class
NIK MOHAMMAD FARHAN BIN AZMI	1. Game Engine modifications 2. Behaviour of UFO and Planet entity 3. Implementation of in-game logic (GamePlay class)
THAM WENG CHER	1. Game engine modifications

IX. Conclusion

In conclusion, this report has provided an overview of the design and implementation details of a game engine and game built using it. Through the use of object-oriented principles and design patterns, the team was able to create a flexible and extensible game engine, and utilise this game engine to develop a game that could entice children to be interested in the domain of space exploration. In addition, the improvements made to the engine since part 1 of the project have allowed the engine to achieve better code reusability and scalability, while design innovations within the game have improved the overall gameplay experience. However, there

are still some limitations with the game and game engine, such as the lack of support for certain features and the need for further optimization. In future work, the team plans to address these limitations and continue to improve the engine and game. Overall, this project has provided valuable insights into the development of game engines, the applications of the object-oriented programming paradigm in the context of software development, as well as the importance of good software design principles in this field.