Star Dash
Final Project Report

21 April 2024

Members:

Lau Rui Han
Qiu Jiasheng, Jason
Goh Jun Yi
Ho Jun Hao

Table of Contents

Overview

Star Dash is a multiplayer racing game that combines the adrenaline rush of Fun Run with the nostalgic charm of the Super Mario Bros. franchise. Players compete against each other in fast-paced races through vibrant and diverse levels filled with obstacles, power-ups, and surprises.

In this racing game, the player who attains the most points wins the race. Throughout the race, players can collect points while competing to reach the end as fast as possible. Reaching the goal faster than others will also award the player more points.

Furthermore, the race features elements such as power-ups and tools. Power-ups are collectibles that increase players' chances of winning, while tools provide alternative forms of navigation through the race. Other elements such as monsters also provide extra difficulty to the race as players need to carefully avoid them or defeat them.

Features and Specifications

Player	Movement: - A joystick is used to move the player left and right. - When the joystick is released, the player will stop moving. - While moving, the player will appear to be running using animations. - The view will follow the player around the game world. Jump: - A jump button is used to move the player vertically upwards. - While the player is jumping, interacting with the joystick will move the player. - Player cannot jump again in mid-air. - After the player jumps, the player will fall back to the ground due to gravity. Hook: - A hook button is used to swing the user in an arc across large space gaps. - Upon firing the hook, the player is unable to move or jump until the swing is completed or the hook is cancelled. - There exists a minimum length that the hook has to extend before the player is allowed to swing in order to provide the minimum ground clearance to prevent the player from colliding with ground whilst swinging. - There is also a maximum length the hook can extend - If the hook fails to be between the minimum and maximum length it will cancel. - If a player collides with an obstacle whilst using the hook, the hook will be cancelled as well. - A hook can only attach to an obstacle Fly: - Can only be activated by attaining the flying power up - While flying a player is unable to move, hook or jump until the power up has ended - If a player collides with anything while flying, the power up will end - To fly higher tap the jump button
Obstacles	- A platform that blocks players from moving through.

	- Entities cannot move through obstacles. - Hooks can only attach to obstacles. - Obstacles have a rectangular bounding area.
Floors and Walls	- Floors and walls are invisible game objects that represent the boundaries of the game world. - The player collides with floors and walls (i.e. Cannot phase through). - Floors and walls have a rectangular bounding area.
Timer	- Updates every fixed time interval to show either how much time has passed or how much time the player has remaining on the game.
Finish Line	- Located at the very end of the map for Race Mode. - Marks the point in which a player has to pass for the game to end. - The bounding area is a rectangle.
Health bar	- The Health bar gives a visual representation of the player's health. - Upon taking damage from monsters, the health bar would decrease
Score Tracker	- The score tracker tracks the current score of the player. - The tracker is able to reflect score changes in accordance to game activity.
Player minimap	- The minimap offers a visual representation of the player's progress and indicates how far they are from reaching the goal. - The minimap displays the progress of all other enemy players to the user.
Stars	- Stars are game objects that contain points and increment the player's score after collection. - After the player comes into contact with the star, the star will disappear and the player's score will increase. - Stars have a circular bounding area.
Power-ups	- Power-ups are game objects that contain some special effect for the player to use. - After the player comes into contact with the power-up box, the power-up box will disappear, and the player will attain a predetermined power-up. - Power-ups have a rectangular bounding area.

Monsters	- Monsters are game entities that move about in the game and can damage the player. - Upon getting hit by the monsters, the player would lose some health and be knocked back. - In order to kill the monster, the player must land on top of the monster (through jumping or other means).
Multiplayer (Local)	- Up to 2 players can play this mode at the same time. - Players play on a split screen with each player's view and controls occupying half of the iPad screen. - Both players race to attain the most points before the game ends. - Certain power-ups exist that players can use to affect the progress of the opposing player. - The first player that reaches the end point will have more completion points, but may not necessarily win the game as the winner is determined by the most total points.
Multiplayer (Online)	- Up to 4 players can play this mode at the same time. - Players can choose to create a lobby and invite their friends or join a lobby that has been created by their friends
Game Mode (Race Mode)	- Game mode where player(s) race to collect as many points as possible before reaching the the end - Players score is being tracked at the top right of the screen - The earlier you pass the finish line, the more points you get - However, where you rank is dependent on your score, not whether you pass the finish line first.
Game Mode (Timed Mode)	- Game mode where player(s) race to collect as many points as possible within 2 minutes. - Scores are being tracked on the top right of the screen

Performance

Requirements

The game should be compatible with all iPads in landscape orientation. The game should run on iOS 15.0 and above.

Runtime Metrics

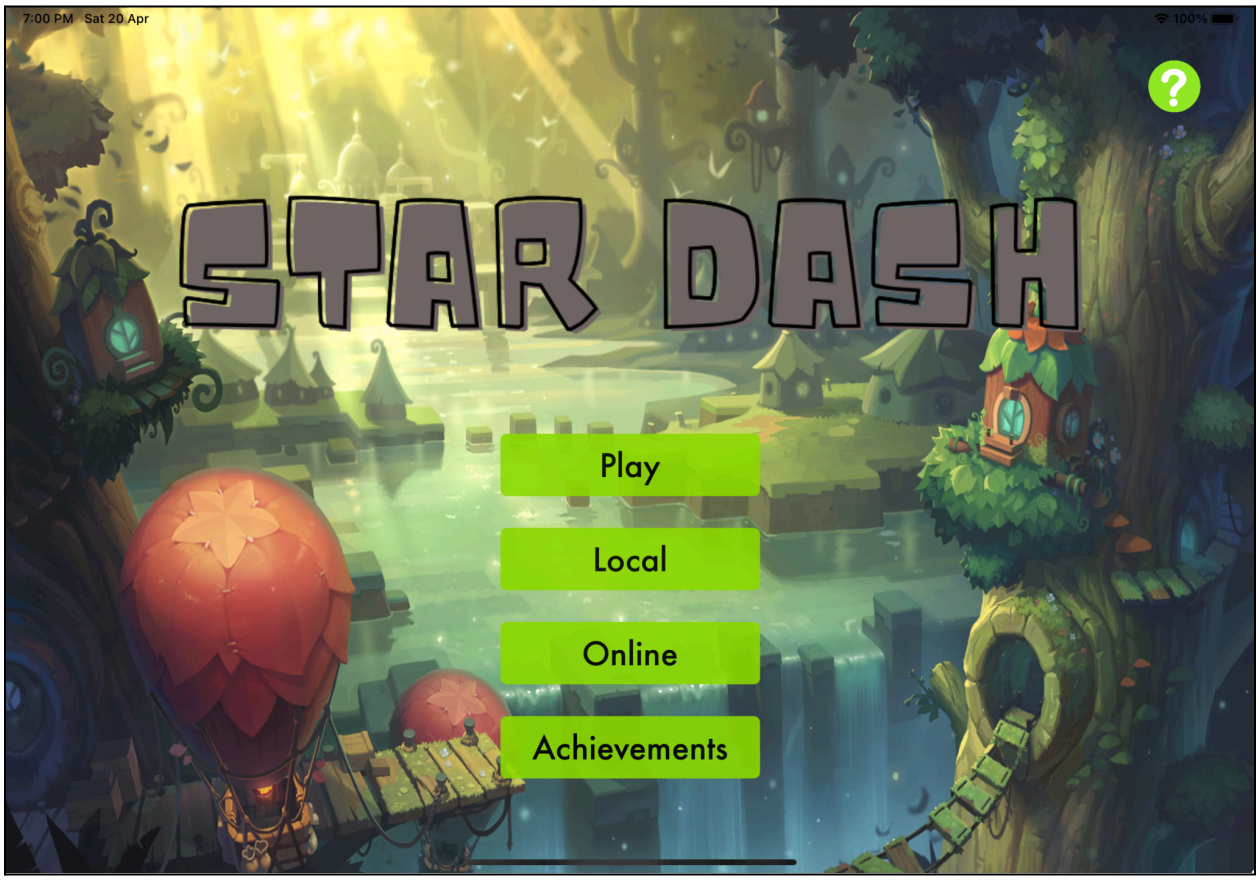
The game should run smoothly without lag. Performance tests were run on both single player and local multiplayer modes on an iPad (8th generation).

For single player mode, the game recorded a 46% CPU usage rate with 171.3 MB memory usage. The game also averaged around 60 frames per second.

For local multiplayer mode, the game recorded a 55% CPU usage rate with 219.6 MB memory usage. The game also averaged around 110 frames per second.

User Manual

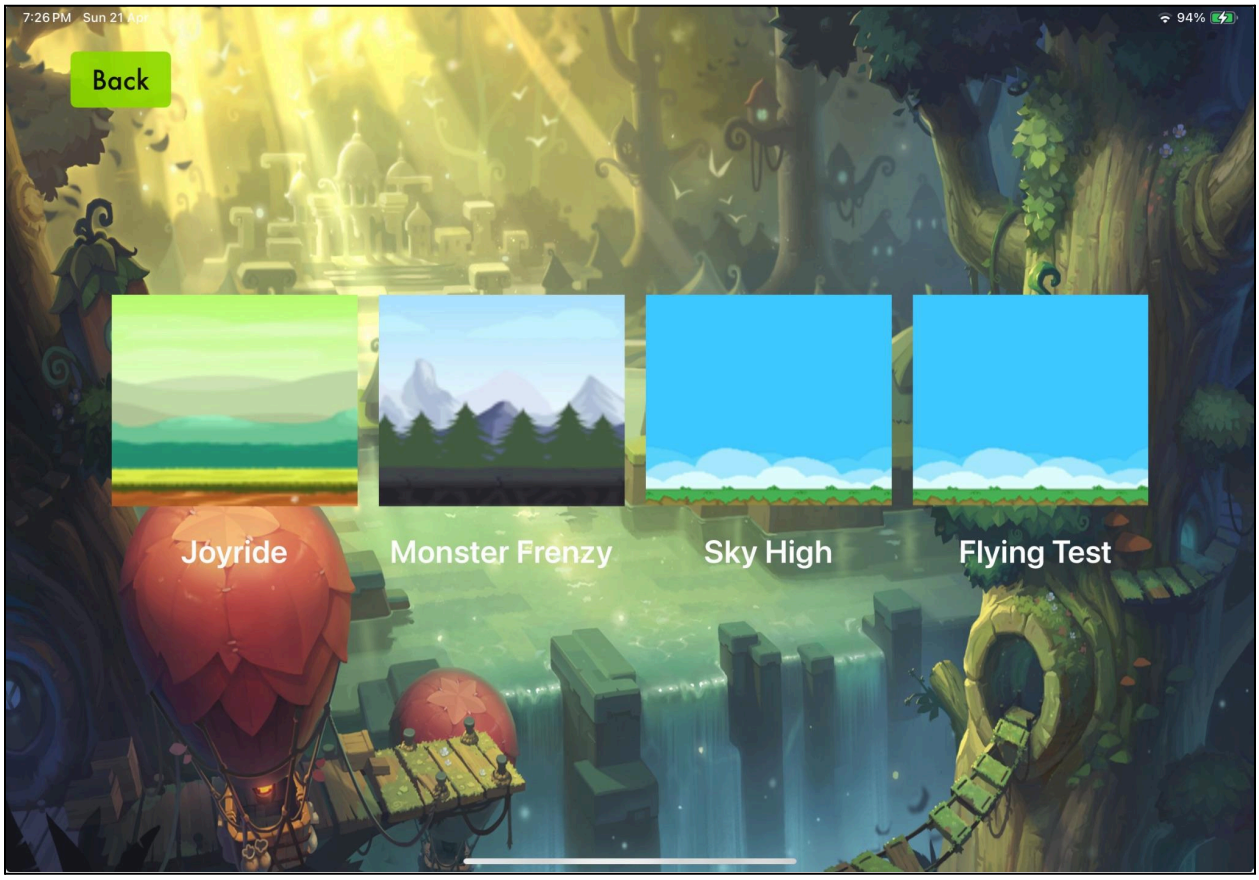
Upon starting the game, you will be met with a screen like this.



Here the player has 5 options:

1. **Play**: Engage in single-player mode
2. **Local**: Participate in local co-op mode
3. **Online**: Participate in online multiplayer mode
4. **Achievements**: View all achievements and track their progress
5. **?** (In the top-right corner): View a simplified user manual

Level Selector



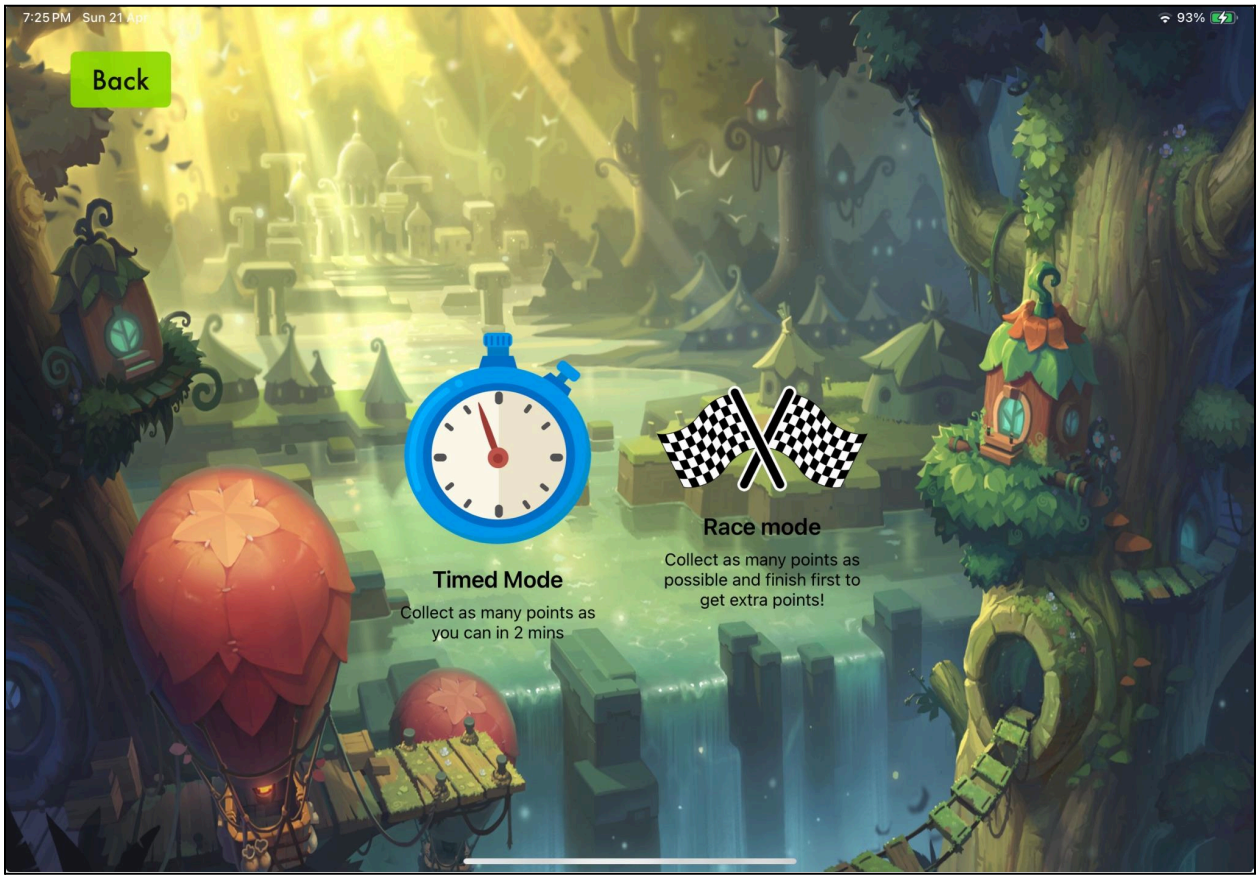
Assuming the user selects **Play** or **Local**, they will be brought to the level selector view.

Here users have an option to pick from 7 preloaded levels:

1. Joyride
2. Monster Frenzy
3. Sky High
4. Flying Test (First power-up is a flying power-up)
5. Speed Boost Test (First power-up is a speed boost power-up)
6. Homing Missile Test (First power-up is a homing missile power-up)
7. Grapple Hook Test
8. Monster Test

Simply tap on any one of the levels to play it. Or if you decide against playing, you can tap the **Back** button located on the top left.

Game Mode Selector



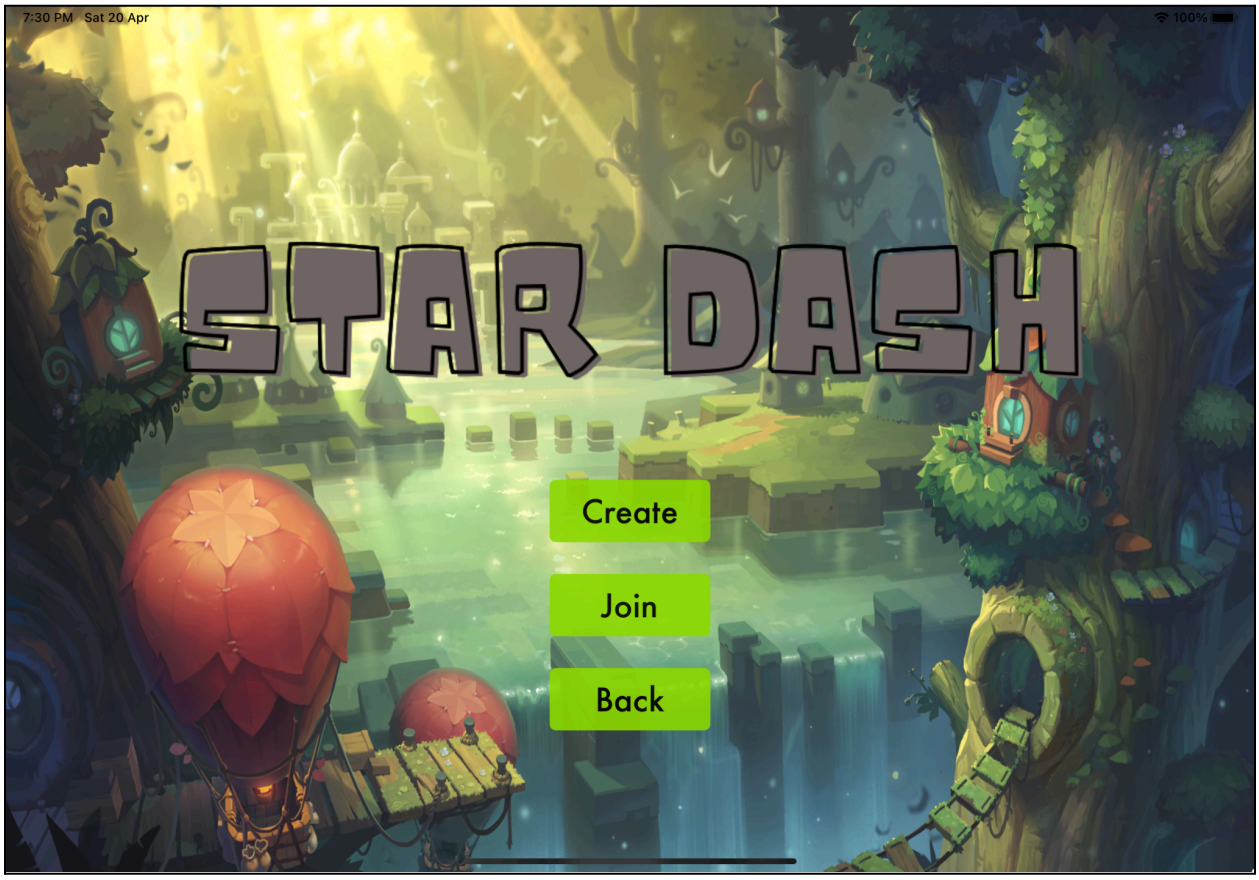
After selecting a level, players will be met with 2 game options:

1. Timed Mode
2. Race Mode

In Timed Mode, player(s) will race to collect as many point as they can within 2 minutes

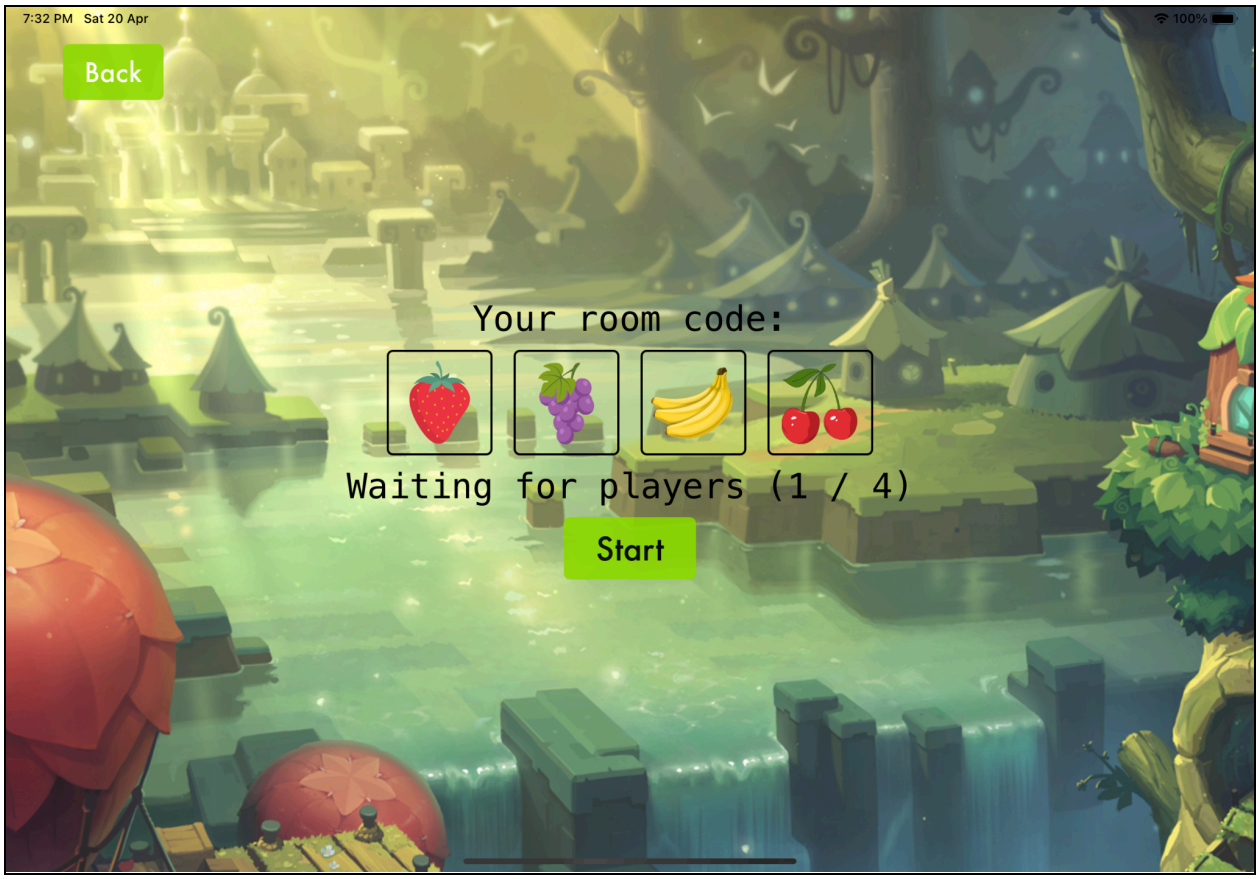
In Race Mode, player(s) will race to collect as many points. Passing the finish line earlier provides more points but does not guarantee a win. The game ends when all players have passed the finish line.

Online multiplayer



Assuming the player selects **Online**, they will be brought to this page where they can choose to either:

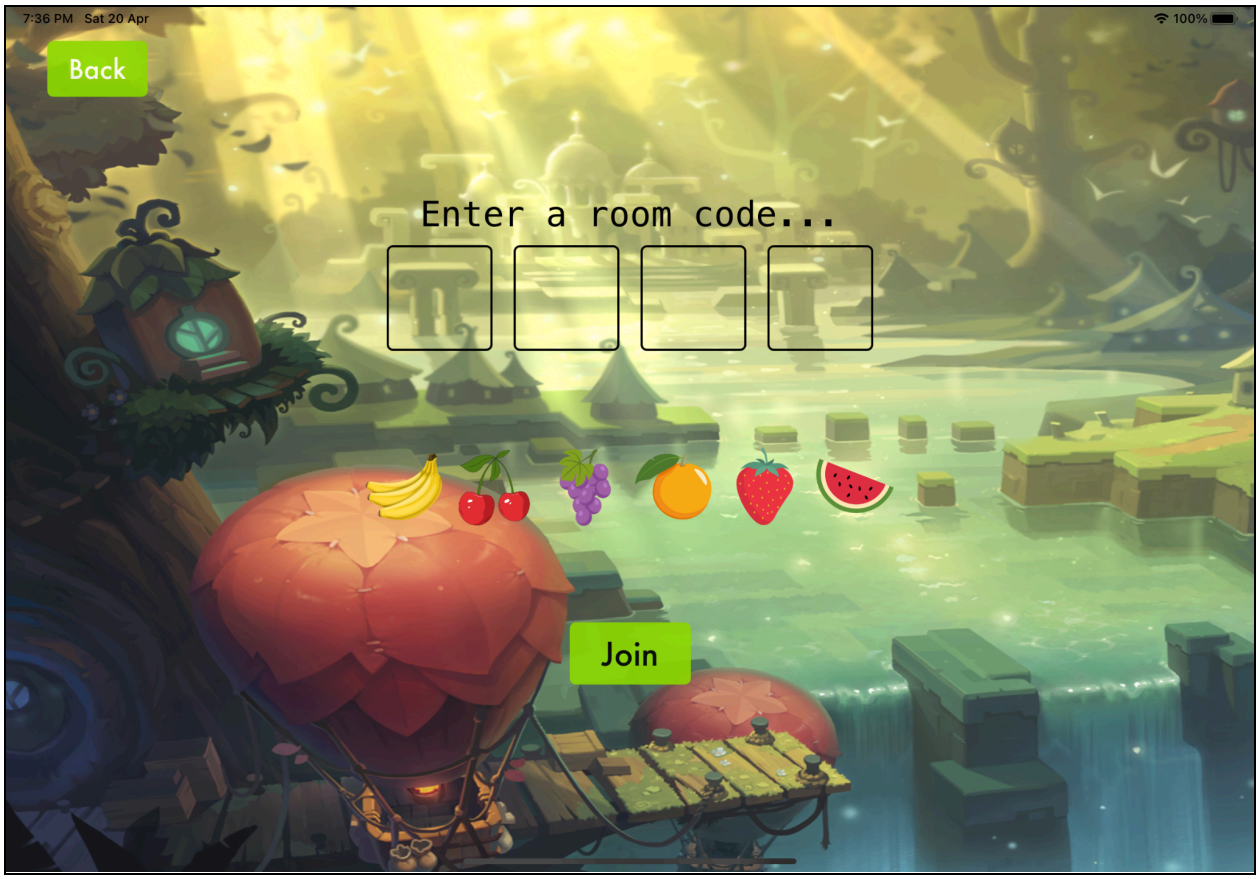
1. Create a lobby to invite their friends to play
2. Join a lobby created by one of their friends



If the user chooses to **Create** a lobby, they will be brought to this screen with the room code. The user will then share the room code with their friends in order for them to join the same lobby as them.

There is a maximum lobby capacity of 4 players. The minimum number of players to play is 1.

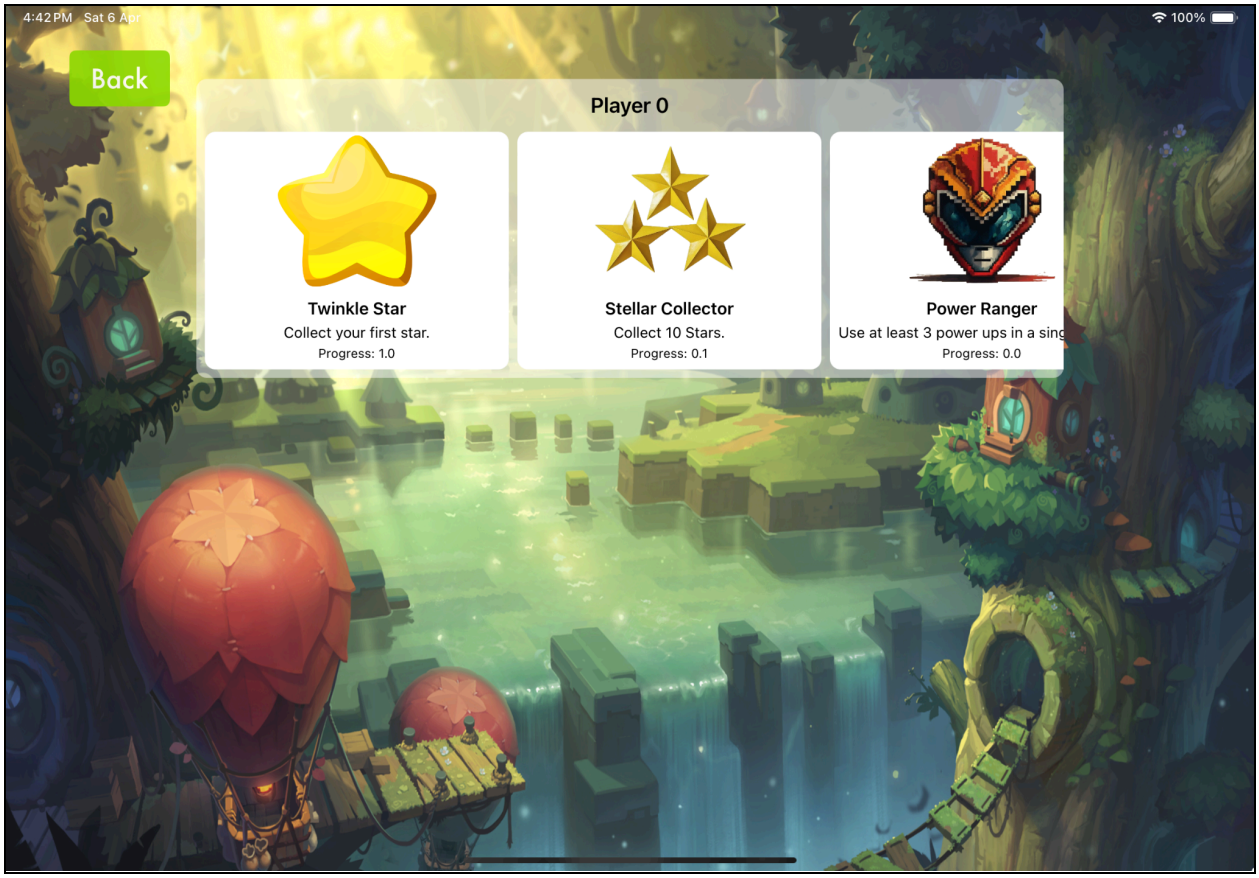
Once everyone is ready, simply press **Start** to begin the game.



If the user decides to **Join**, a lobby, they will be met with this screen, in which they would have to get the room code from the lobby owner.

Achievements

Assuming the user taps on **Achievements**, he will be met with this view if it's their first time playing.

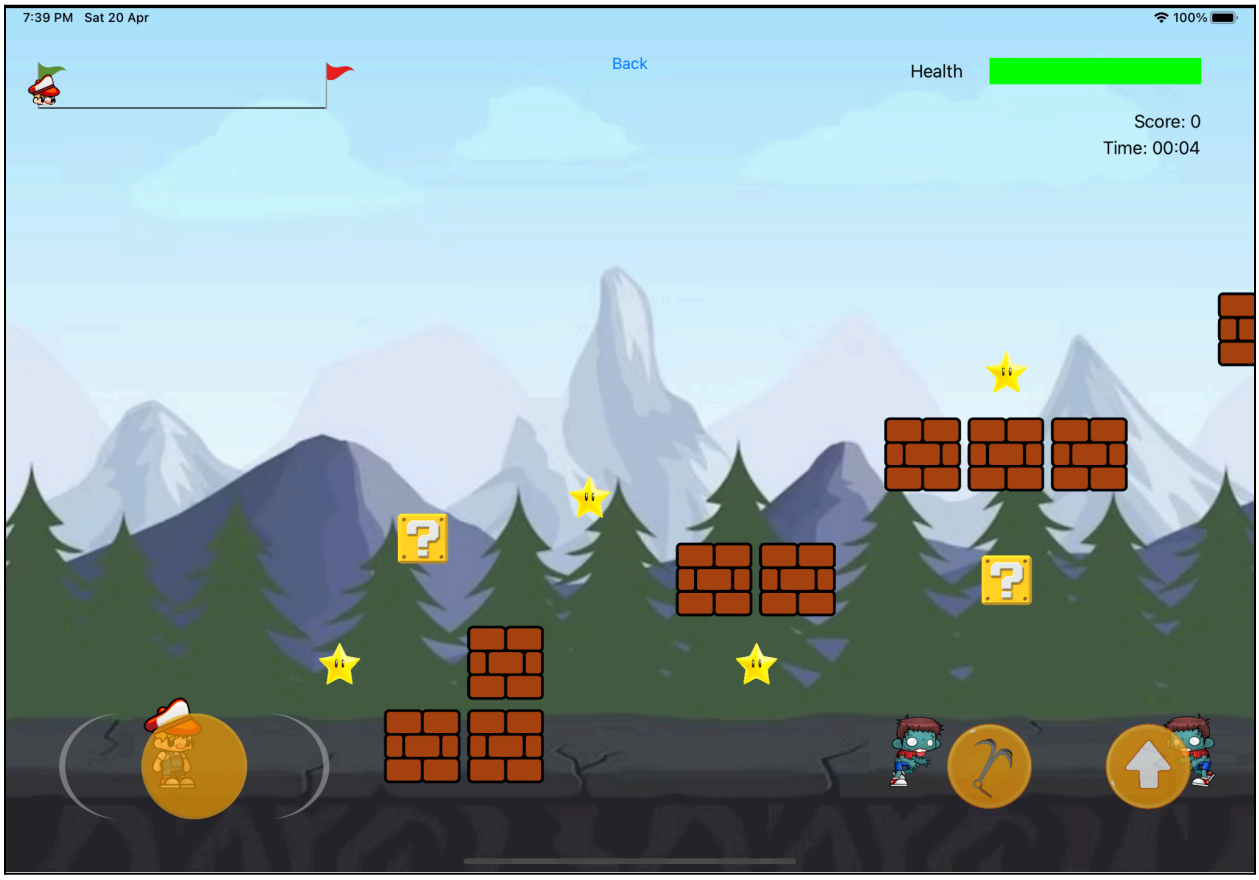


If the player has previously played the game and made progress towards achievements, their profile will be saved. This allows them to view their achievements and track their progress here.

The following table lists all of the achievements available:

Achievement Name	Description
Twinkle Star	Collect a star in any game
Stellar Collector	Collect 10 stars
Power Ranger	Use 3 power-ups in a single game

Player Movement



To move the player left or right, drag the joystick on the bottom-left of the screen in the left or right direction respectively.

To jump, press the up button on the bottom-right of the screen. While the player is jumping, he is unable to jump again until he comes in contact with the floor again.

To use the grappling hook, press the hook button located to the left of the jump button. A hook will be shot out at a 45 degree angle upwards in the direction the player is facing. Once the grappling hook comes in contact with an obstacle, it will retract and swing the player. Note that if the hook is not sufficiently long or if the user collides with an obstacle, the hook will be cancelled.

Collecting Stars

To gain points in the game, you need to collect the stars scattered around the game. You can collect these stars by moving the player into contact with it. Subsequently, your score should increment by a set amount. Your score is reflected at the top-right of the screen.

Power-ups

To attain power-ups in the game, you need to collect the yellow boxes with a question mark. You collect the power-up by moving the player into contact with it. Subsequently, you will gain a predetermined power-up which will give you an advantage in the game.

The following table lists all of the power-ups available:

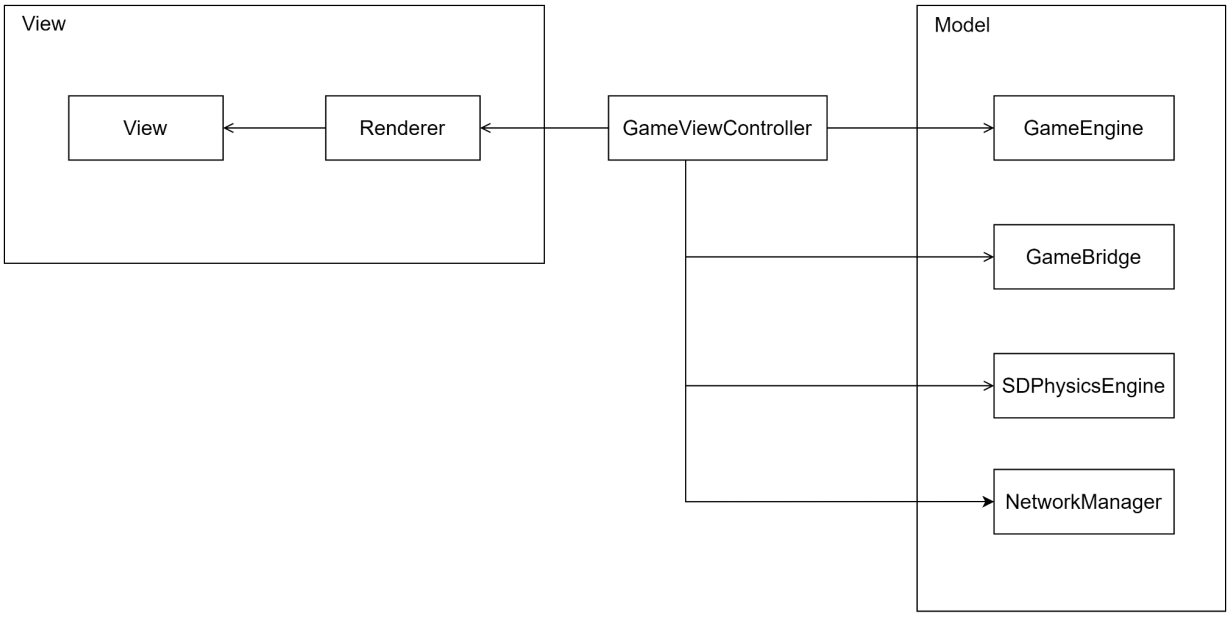
Power-Up	Effect
Speed Boost	- Increases the speed of the user's movement by 1.5 times - Lasts 15 seconds
Homing Missile	- Launches a homing missile at an enemy player in front of the user - The missile will follow the enemy player in front until it collides with the enemy player, killing them - Alternatively, if the missile collides with an obstacle, it will simply destroy itself
Flying	- Turns the user into a plane which flies to the right - Lasts 10 seconds or until the user collides with something - Fly higher by pressing the jump button

Monsters

There are monsters throughout the entire game. These monsters can damage you if you come in contact with them. Upon getting hit by the monster, you will lose some health and experience a knockback. After losing enough health, you will die and experience a 2 second timeout before respawning. You can kill these monsters by jumping and landing on top of them.

Software Design

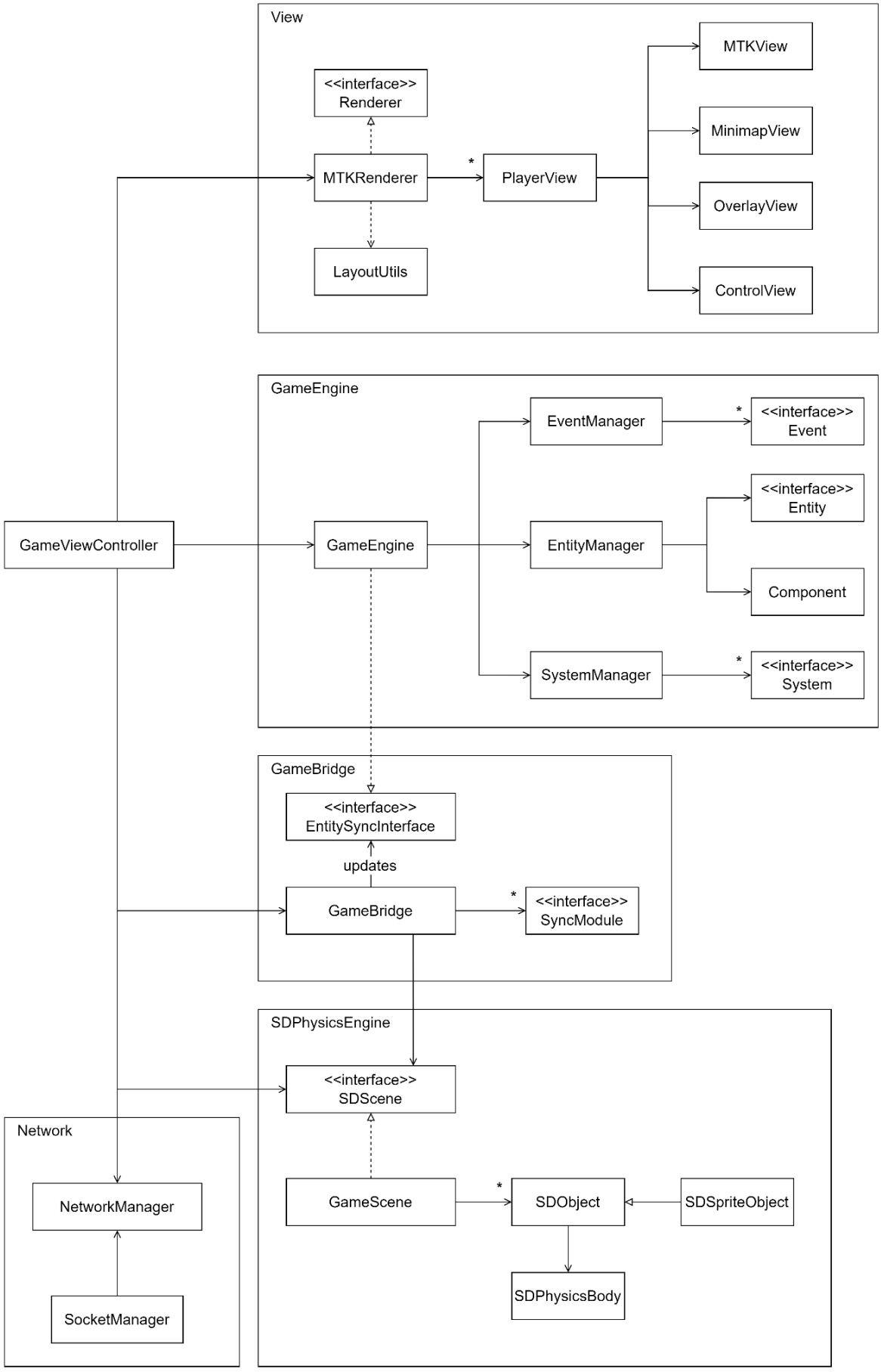
Overall Structure



The overall code structure generally follows a **Model-View-Controller (MVC)** structure, providing clear separation of responsibilities. The view is represented by the *Renderer* and the *View*, while the model is represented by the *SDPhysicsEngine*, *GameBridge*, *GameEngine and Network Manager*

Here is a brief description of each component's responsibilities:

- GameEngine: Game-specific logic
- SDPhysicsEngine: Simulation of physics interactions between objects
- GameBridge: Syncs the objects between the GameEngine and SDPhysicsEngine
- Network Manager: Handles the networking between the client device and server
- Controller: Coordinates the components
- Renderer: Renders the game onto the view
- View: Displays the game and controls to the player and handles user interaction


View
<<interface>>
Renderer
MTKRenderer
*
PlayerView
MTKView
MinimapView
OverlayView
ControlView
LayoutUtils
GameEngine
EventManager
*
<<interface>>
Event
<<interface>>
Entity
GameViewController
GameEngine
EntityManager
Component
SystemManager
*
<<interface>>
System
GameBridge
<<interface>>
EntitySyncInterface
updates
GameBridge
*
<<interface>>
SyncModule
SDPhysicsEngine
<<interface>>
SDScene
Network
NetworkManager
GameScene
*
SDObject
SDSpriteObject
SDPhysicsBody
SocketManager

Game Engine

The Game Engine is responsible for performing game-related logic, such as handling contacts between objects and character movement in game.

Runtime Structure

The Game Engine follows the **Entity-Component-System (ECS)** structure. This design structure is chosen over traditional class-based architecture for its ability to manage complex interactions between entities in games.

In particular, ECS:

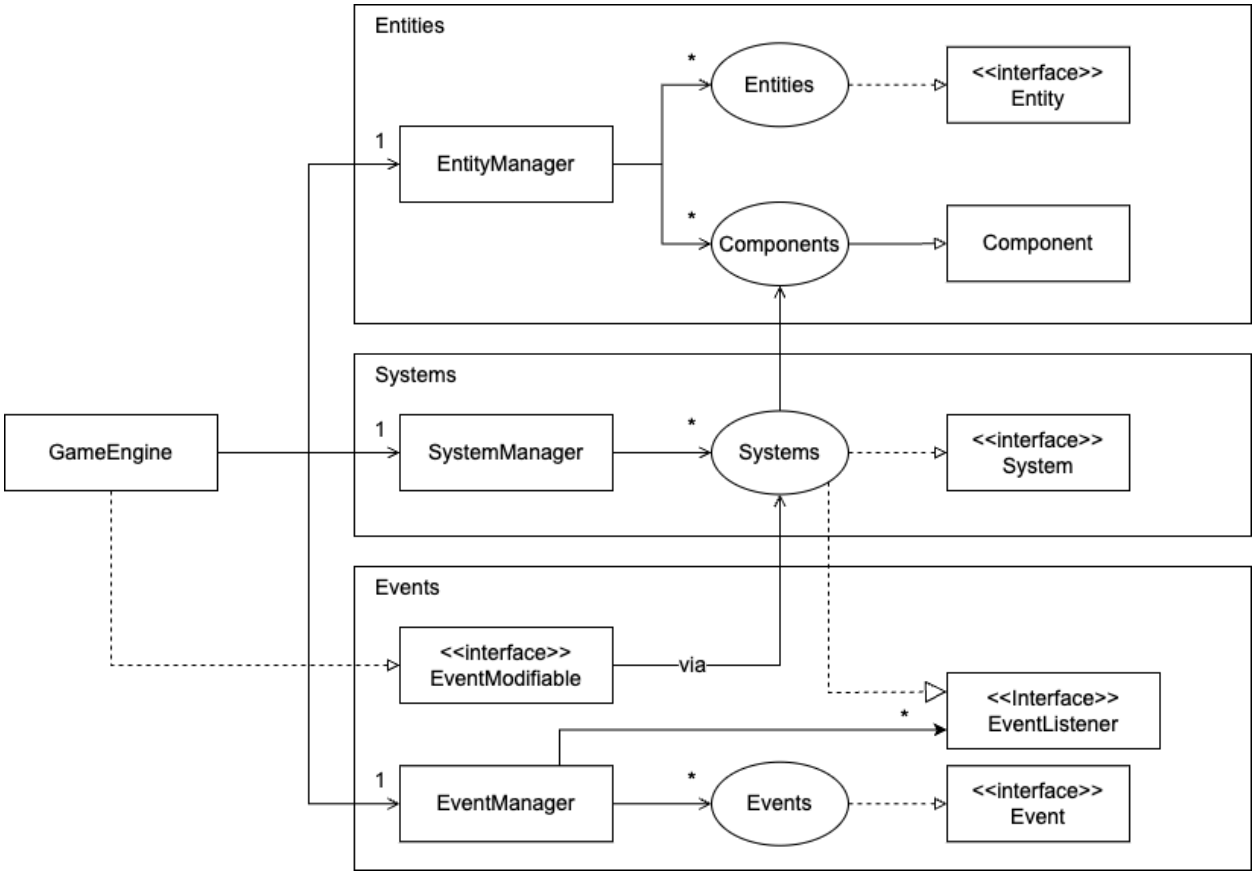
1) Offers enhanced flexibility by enabling components to be dynamically added and removed from entities during gameplay. This allows for more dynamic and customizable gameplay experiences.
2) Enhances scalability by facilitating the addition of new features without requiring extensive modifications to existing code. This makes it easier to extend and evolve the game over time.
3) Promotes a clear separation of concerns by separating the data from the logic of an entity. This separation of responsibility enhances the maintainability and readability of the codebase, making it easier to understand and maintain.

Module Structure

The following table details the ECS's modules and their functions:

Module	**Definition & Purpose**
Entity	Represents a game object.
Component	Represents characteristics of the entity. Contains the data/state of the game object.
System	A process that acts on all the game objects. Contains the logic of how game objects interact.
Events	A signal that is being dispatched to trigger actions or behaviour in the game. Contains data of the event.

The following diagram provides a high-level overview of the GameEngine. To remain concise, the collections of created entity, component, system, and event classes are abstracted in the diagram. For more details and the complete list of classes, you may wish to refer to subsequent sections.



Entities

The *EntityManager* is the entry point for the Entities module, and manages all of the entities and components in the game. It organises these entities and components using maps that associate unique identifiers with their respective entities and components, enabling fast and efficient identification and access.

The Entity Manager has 4 main functions:

1) Add/remove entities and their respective components
2) Fetch entities using entity id
3) Fetch components using component id
4) Identify components that belong to an entity

One important note is that there is no logic inside EntityManager, Entity, and Component. They are solely responsible for maintaining game data.

The following table lists all of the entities created:

Entity	**Purpose**
Player	Represents a player in game
Monster	Represents a monster in game
Collectible	Represents a Star in game
Obstacle	Represents an obstacle block in game
Wall	Represents the invisible wall boundary in game
Floor	Represents the invisible floor boundary in game
GrappleHook	Represents the head of the grappling hook in game
Rope	Represents the rope of the grappling hook in game
FinishLine	Represents a finish line in game
PowerUpBox	Represents the power-up box in game
HomingMissile	Represents a homing missile in game
SpeedBoost	Represents a speed boost power-up in game

The following table lists all of the components created:

Component	**Purpose**
BuffComponent	Represents what effects entity has on
PointsComponent	Represents how many points a entity provides
DeathTimerComponent	Represents how much time remaining an entity remains dead
PlayerComponent	Represents the player number/index
PositionComponent	Represents where an entity is
ScoreComponent	Represents how many points Player entity has

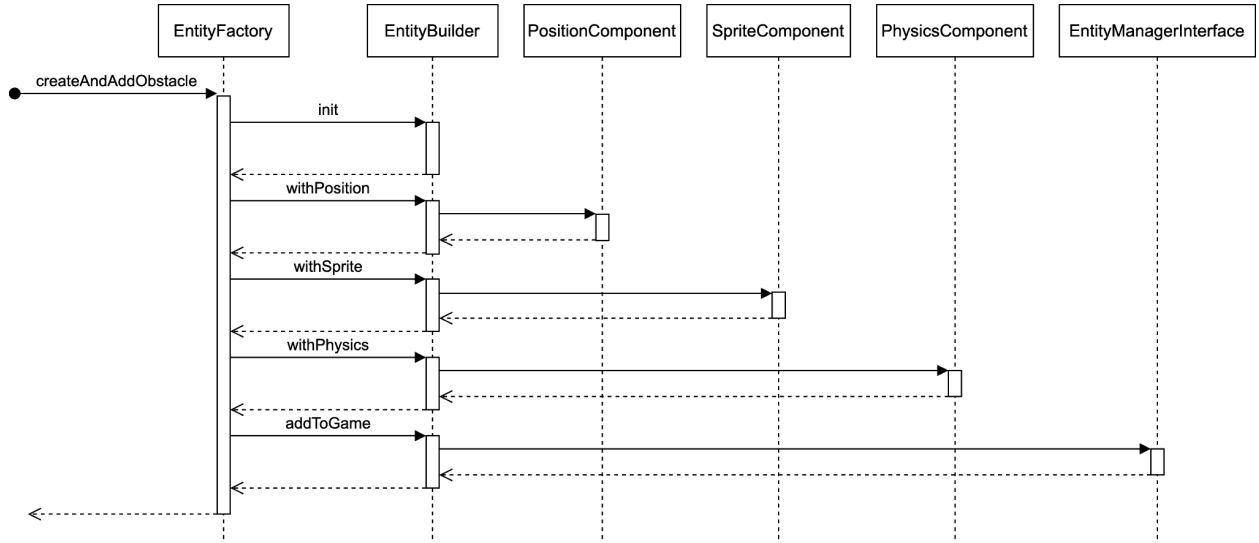
HealthComponent	Represents how much health an entity has
SpriteComponent	Represents what texture and animation an entity should display
PhysicsComponent	Represents the physics of an entity such as speed and force
GrappleHookComponent	Represents hook details such as how much to swing/retract
GrappleHookOwnerComponent	Represents which player entity owns the grapple hook
OwnsRopeComponent	Represents the rope that belongs to a grapple hook
PowerUpComponent	Represents the type of power-up
HomingMissileComponent	Represents the details of a missile such as whether its activated, which player it's locked onto
SpeedBoostComponent	Represents details of a speed boost powerup such as how long it lasts and how much to boost
FlyComponent	Represents whether an entity is currently flying. Also contains how much time the entity can remain flying.
JumpComponent	Represents whether an entity is currently jumping

Creating Entities with Components

The ECS system contains numerous entities, each possessing a unique combination of components. To streamline entity initialization and prevent developers from overlooking certain components during initialization, all entities are instantiated using the *EntityFactory* and *EntityBuilder* classes, which implement the **factory** and **builder design** patterns respectively.

The *EntityFactory* class serves as a centralised hub for creating different types of entities, abstracting the creation logic and reducing code duplication. It utilises the *EntityBuilder* class to facilitate the construction of complex entities by separating the construction process into distinct steps. Developers can use the builder to specify the desired components and their configurations, resulting in more readable and maintainable code.

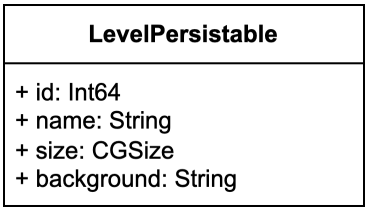
As an example, the following sequence diagram outlines the process to create an *Obstacle* entity:



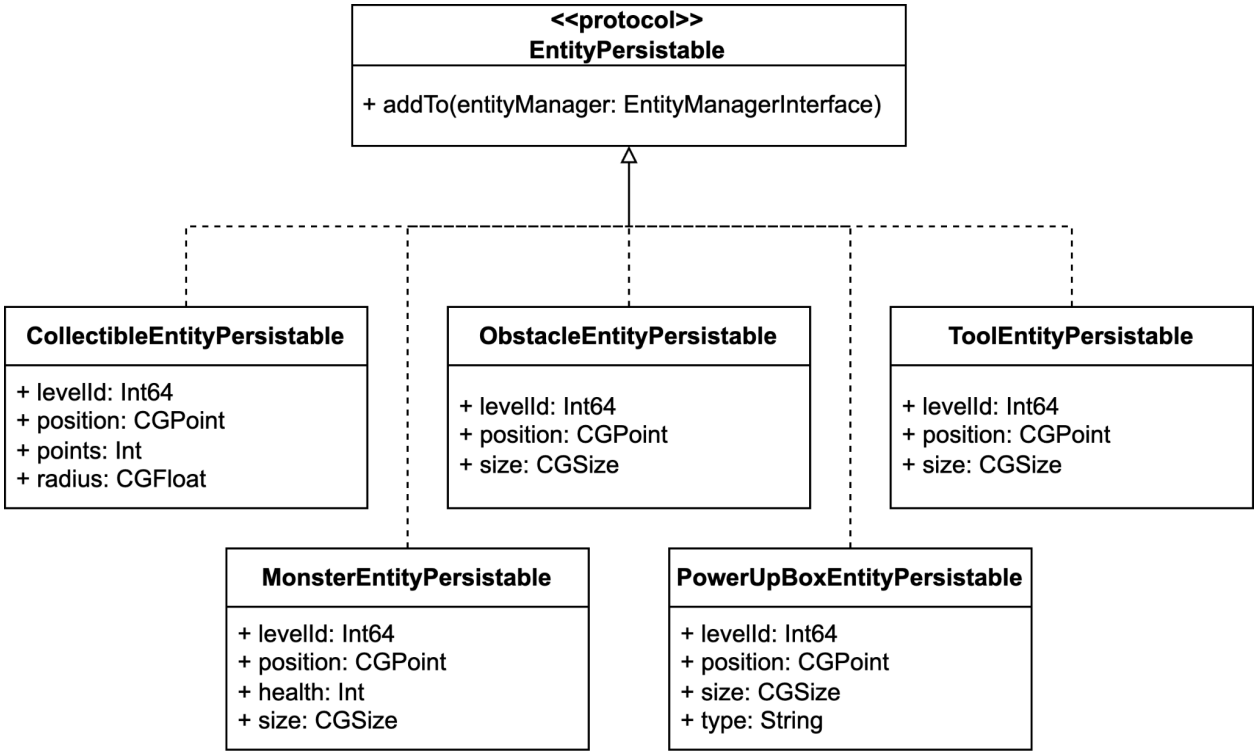
Loading Levels

The levels are predesigned and preloaded into the database from a json file.

The metadata of each level is represented by *LevelPersistable*, as shown in the following class diagram:



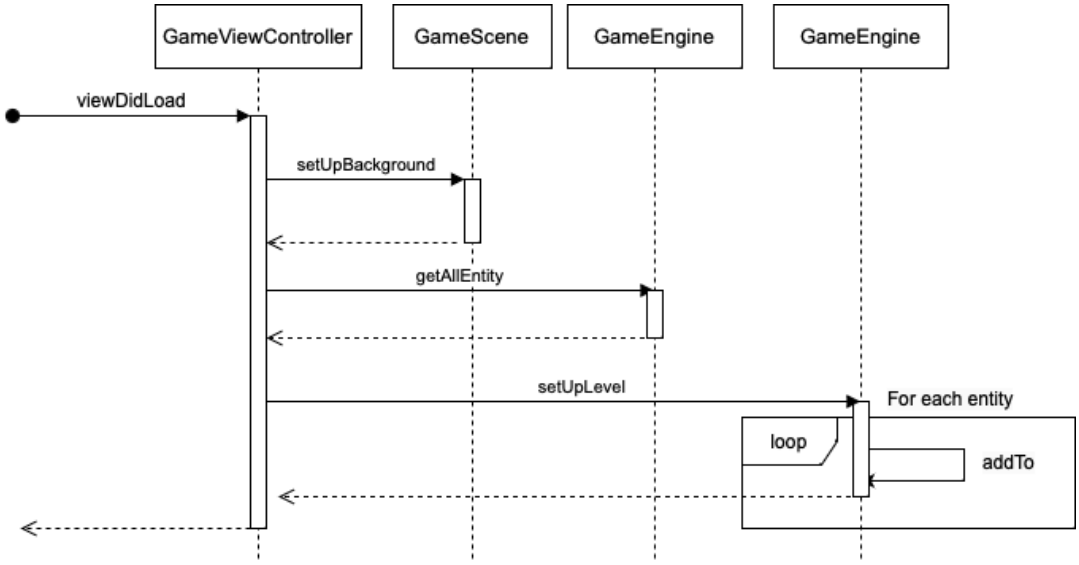
The metadata for each entity is represented by classes that conform to the *EntityPersistable* protocol. The following class diagram illustrates these persistables:



When the level selector is rendered, the metadata of all levels is retrieved from the database and then rendered in the UI.

When the player selects a level to play on, the information of size and background of the level is used to create the *GameScene* and then the *GameViewController* retrieves the entities. Finally, the *GameEngine* loads the level and its entities.

As an example, here is a sequence diagram that details this process:



Systems

The *SystemManager*, which serves as the entry point for the Systems module, contains and manages all the systems that act on and manipulate game objects.

The System Manager has 4 main functions:

1) Add systems into the game
2) Enable/disable certain systems during the game
3) Identify and return a system that is needed
4) Update all the systems at each game tick (if necessary)

Systems do not contain any data. Any default values used are accessed from *Constants*. Systems are solely responsible for any modifications done to entities and components.

There are various systems, each responsible for managing different aspects of the game. Implicitly, we divide systems into 2 groups: Basic and Complex.

Basic systems encompass functions that handle the **logic of accessing and modifying components within their domain**. For instance, the *PositionSystem* includes fundamental functions like updating an entity's position component. These systems typically refrain from directly altering the components they do not manage.

On the other hand, complex systems feature functions with more complicated logic, such as the usage of a grappling hook, which may involve coordinating multiple systems like *PositionSystem* and *PlayerSystem*. These systems are **responsible for managing more elaborate behaviours and interactions** within the application. In complex cases, systems can access other systems or dispatch events through the *EventModifiable* interface.

No matter if it's a basic or complex system, systems can only directly modify entities and components if they are **responsible** for them. Otherwise, they need to get the system responsible for it to make the changes, upholding the single-responsibility principle.

The following table lists the basic systems:

Basic System	**Contains logic to**
GameSoundSystem	Manages the sounds in the game
PlayerSystem	Manage the player component of an entity
BuffSystem	Manage the buff component of an entity
HealthSystem	Manage the health component of an entity
RemovalSystem	Remove entities in the game
ScoreSystem	Manage the score component of an entity
PhysicsSystem	Manage the physics component of an entity such as velocity and size
PositionSystem	Manage the position component of an entity
SpriteSystem	Manage the sprite component of an entity

The following table lists the complex systems:

Complex System	**Contains logic to**
MonsterSystem	Manage how a monster behaves
DeathSystem	Manage how an entity dies
MovementSystem	Manage how an entity moves in game
AttackSystem	Manage what happens when an entity attacks another

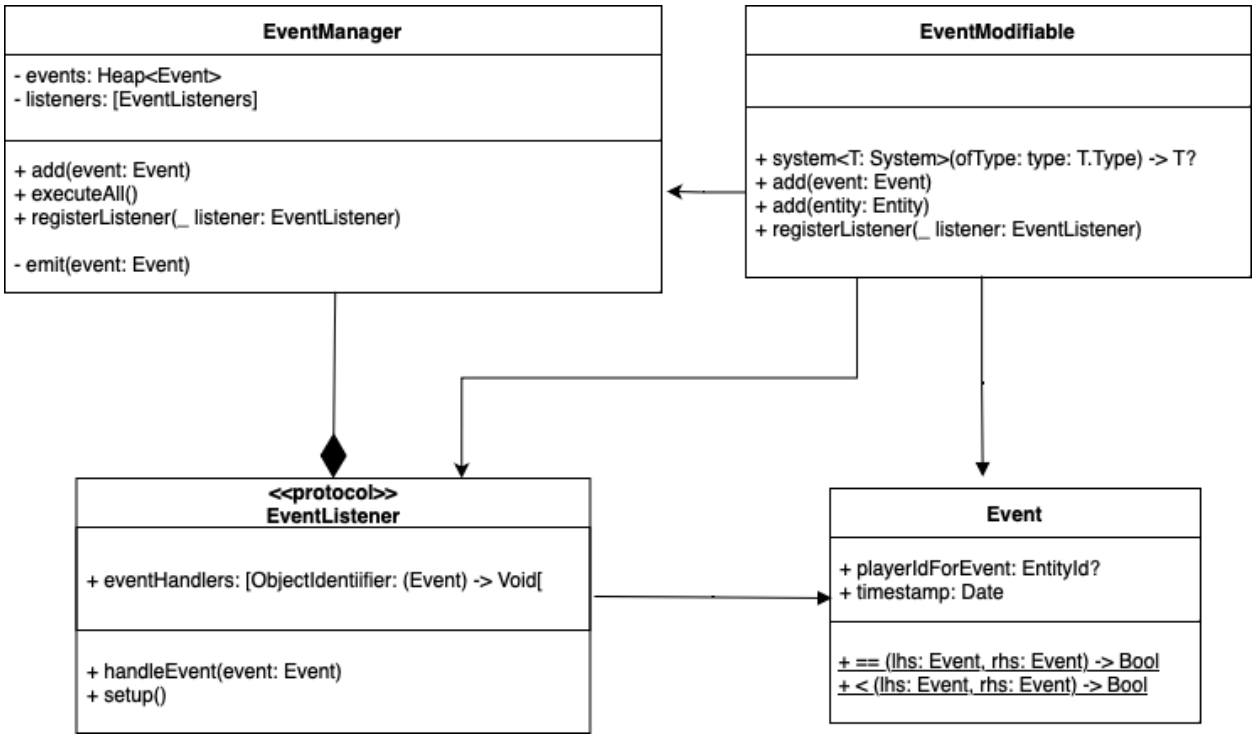
CollisionSystem	Manage what happens when 2 entities collide with each other
FinishSystem	Manages whether a player has completed a game
HomingMissileSystem	Manage how a homing missile behaves
SpeedBoostPowerUp System	Manage how speed boost behaves

All systems inherit the *EventListener* protocol to respond to specific actions or changes within the system. Event listeners are a crucial aspect of our architecture, enabling modules to react dynamically to events. The next section will detail how events are generated and processed within our system.

Events

In the Events module, the *EventManager* receives and manages all the events that have been dispatched in the game. It contains a queue of events that are executed in sequence at each game tick. Future extensions may alter the queue data structure into a priority queue, in the scenario where certain events have higher priority and should be executed first.

The following class diagram outlines the classes involved in handling events:



All events conform to the *Event* protocol. The following table lists all of the events in the game:

Events	
UseGrappleHookEvent	PlayerFloorContactEvent
ShootGrappleHookEvent	PlayerMonsterContactEvent
RetractGrappleHookEvent	PlayerObstacleContactEvent
SwingGrappleHookEvent	GrappleHookObstacleContactEvent

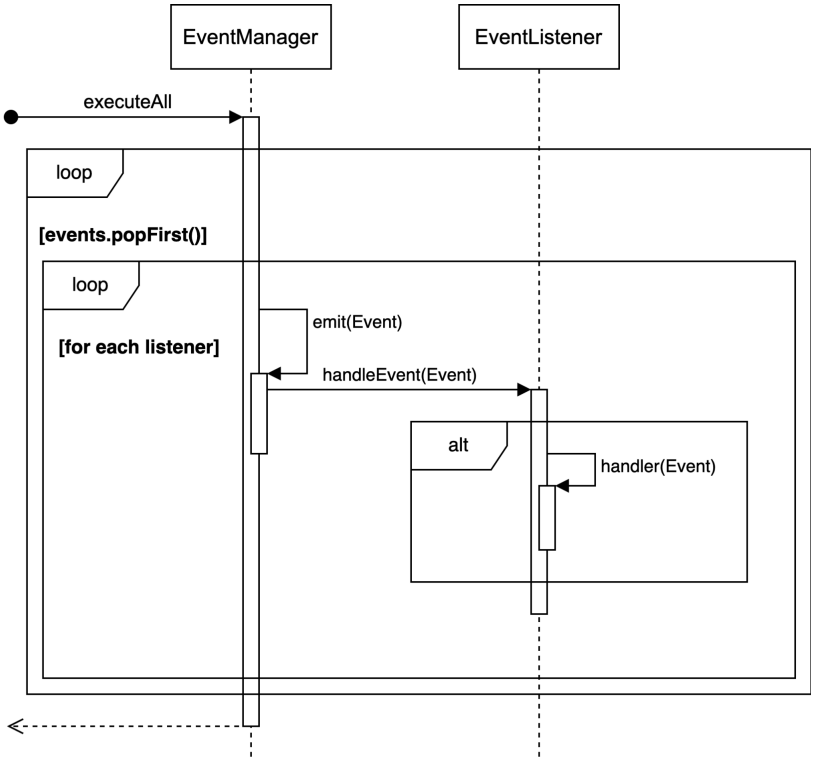
ReleaseGrappleHookEvent	MonsterObstacleContactEvent
PowerUpBoxPlayerEvent	RemoveEvent
MissileHitPlayerEvent	TeleportEvent
MissileBlockedEvent	StopMovingEvent
MonsterAttackPlayerEvent	MoveEvent
PlayerAttackMonsterEvent	JumpEvent
MonsterDeathEvent	RespawnEvent
MonsterMovementReversalEvent	PlayerDeathEvent
PickupCollectibleEvent	StartFlyingEvent
MonsterWallContactEvent	UpdatePositionEvent
DeathEvent	FinishEvent

EventListeners

The *EventManager* in our system utilises the **observer pattern**, allowing EventListeners to register with it. When events are executed or in other words dispatched, they are sent to all registered listeners. Event listeners receive these events through the *handleEvent(...)* method, where they can decide whether the event is relevant to them or not. If it is, they can execute their own logic.

By employing the observer pattern, we ensure a clear separation of concerns. As previously mentioned, systems implement the *EventListener* protocol, ensuring that all logic remains within the systems while still allowing for flexible event handling and customization.

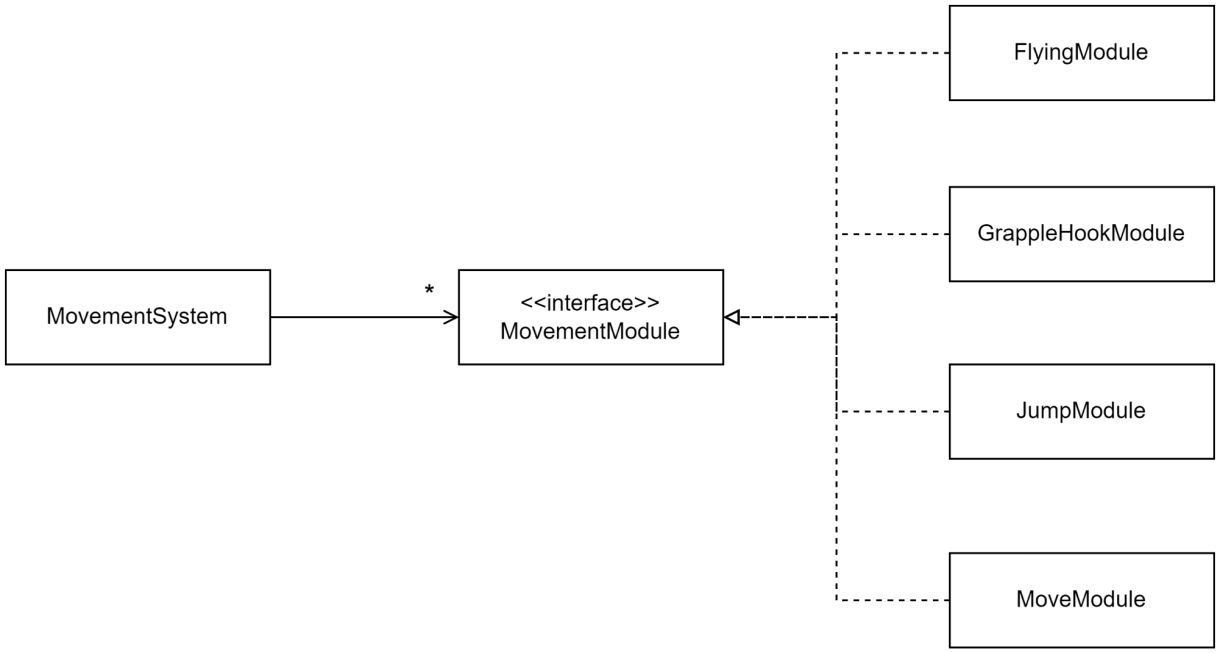
The following sequence diagram illustrates the observer pattern used in handling events:



EventModifiable

GameEngine conforms to the *EventModifiable* protocol, which contains functions to add new Events into the *EventManager*. It grants access to the game's systems, components, and entities. Lastly, it is able to register systems as a listener to the EventManager. EventModifiable is accessible in every system. This helps to facilitate dynamic event handling and system interactions promoting flexible and efficient logic management.

Movement



In Star Dash, there are various ways to traverse the map. Apart from moving and jumping, the player can grapple hook or fly. All logic regarding movement is encapsulated in *MovementSystem* and *MovementModules.*

The following table describes the various *MovementModules* and their corresponding movement types. The order of the *MoveModule* is in decreasing priority, which will be explained below.

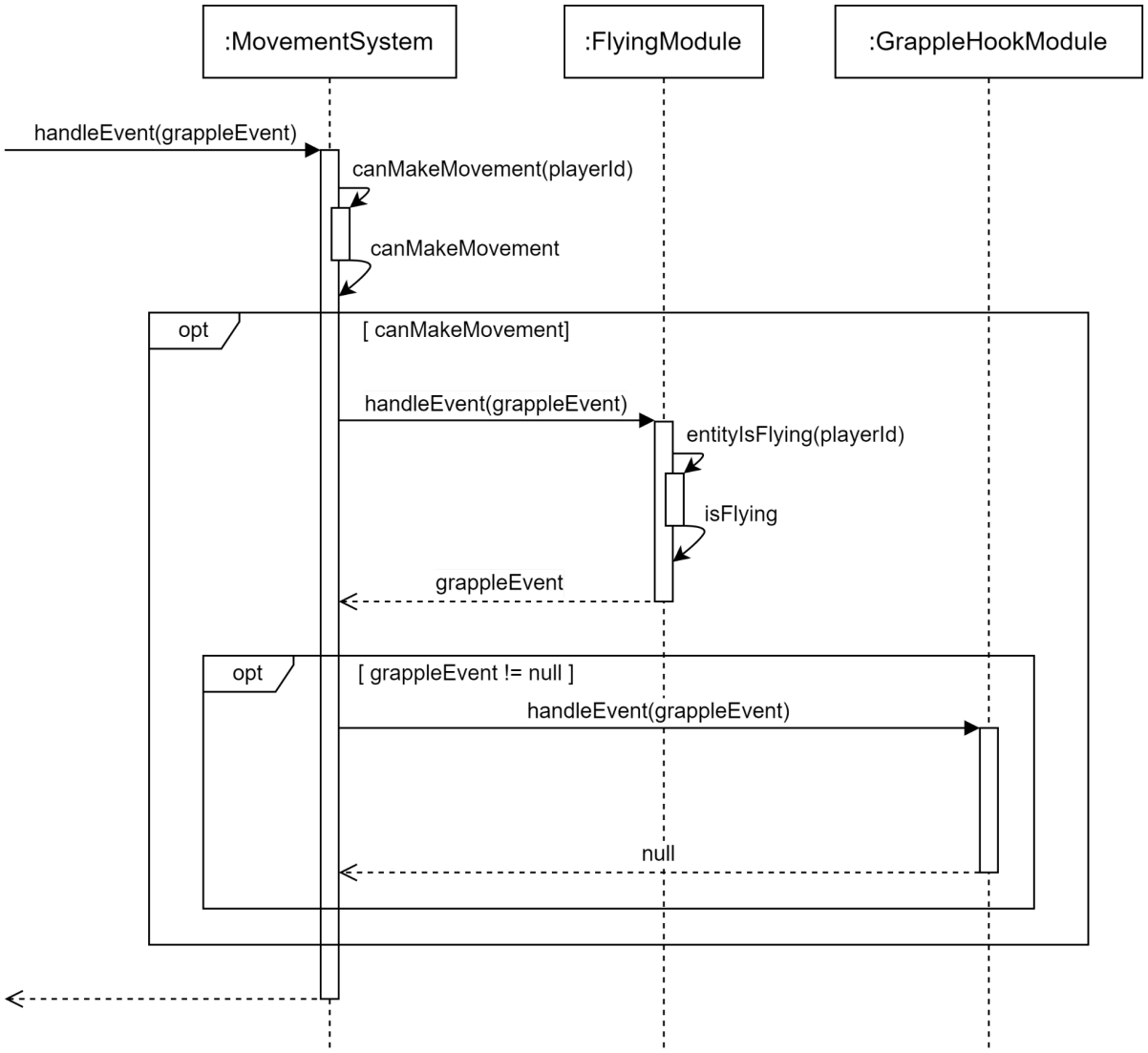
Priority Ranking	MovementModule	Movement Type
4	MoveModule	Running
3	JumpModule	Jumping
2	GrappleHookModule	Grapple Hook
1	FlyingModule	Flying

The design of *MovementSystem* follows the **chain of responsibility** design pattern. All of these *MovementModule* form a chain which follows the priority in the order of the table above. These modules accept events, process them, or pass it down to the next module.

Typically, each of these modules only subscribe to a set of events but they may accept events that other modules subscribed to. For example, the *FlyingModule* subscribes to *StartFlyingEvent* but it may accept *MoveEvent*. This is where the importance of priority comes in. *FlyingModule* can decide whether to drop events it does not subscribe to, in this case, it drops the *MoveEvent* if the player is already flying (to prevent other forms of movement), else it passes it on. Therefore with this structure, every *MovementModule* has strong separation of concerns as it only processes events that they subscribe to and for events that they did not, they would pass it on only if its own movement type is not active (eg. player is not flying).

Furthermore, this design enables dynamic disabling or insertion of *MovementModule* in the middle of the game since every module does not depend on any other modules.

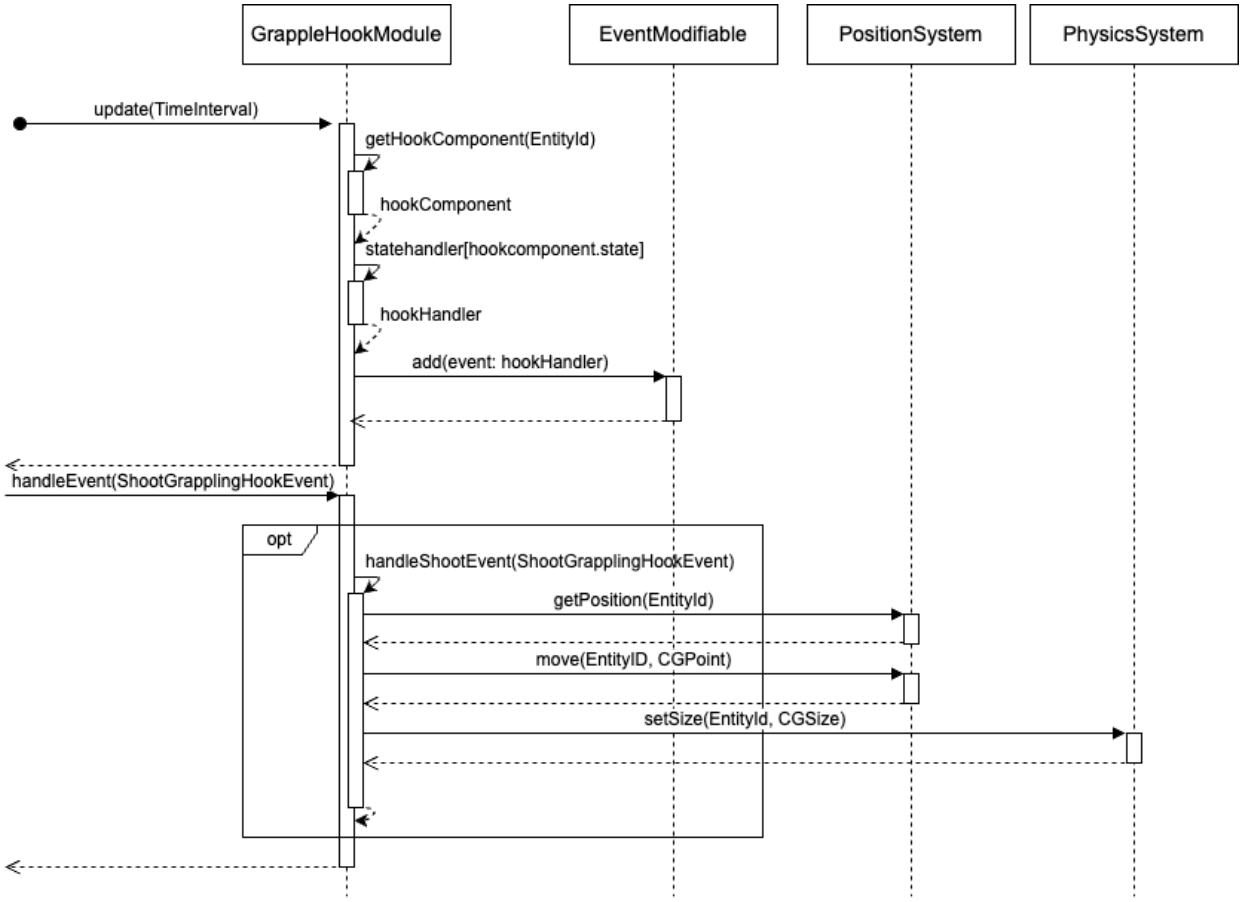
Grapple Hook Event Sequence



The above sequence diagram showcases the situation when the *MovementSystem* is called to handle a grapple hook event when the player is **not flying**. The *MovementSystem* first verifies if the player is capable of making any movement (eg. player is alive). If the player is capable of making movement, the event starts to pass down the various *MovementModules*. As *FlyingModule* has the highest priority, it first receives the event. Since it is not an event it chose to subscribe to, it does not process it and only returns the original event back if the player is not flying (returns null otherwise). If the returned event is not null, the next module, *GrappleHookModule*, receives the

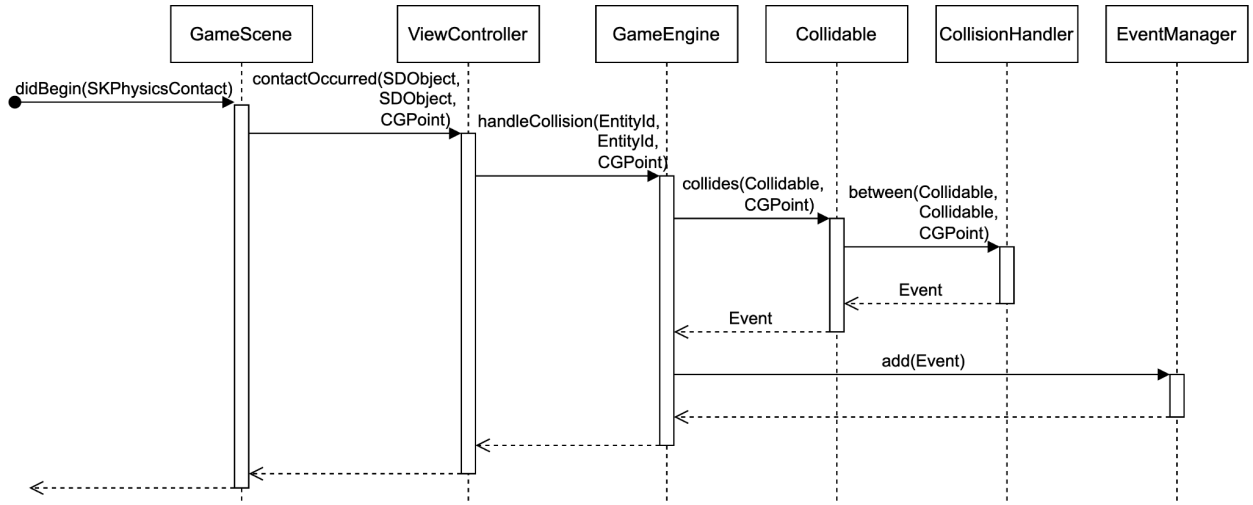
event and finally processes it since it subscribes to it. Finally, it returns null as no other lower priority modules should handle an event related to grapple hook.

Grappling Hook Module



The above sequence diagram gives an example of how the grappling hook module is called at each update to move the player and hook after it has been activated. Each grappling hook has a *GrappleHook* component that stores the state of the hook. There are 4 states: **Shooting**, **Retracting**, **Swinging**, **Releasing**. At each update, the *GrappleHookModule* would determine the state and send the corresponding event out. The event will then be handled accordingly by *GrappleHookModule* when it receives it next time.

Collisions



It is important to note that the *CollisionHandler* is **not responsible for detecting contact or separation**; this task is performed by our *SDPhysicsEngine*. Instead, the *CollisionHandler* in the *GameEngine* focuses on post-collision modifications, such as picking up stars, updating scores, or triggering other game events based on what entities have collided.

Each entity implements the *Collidable* protocol, which employs a **double dispatch pattern**. This pattern enables entities to call the *collides(...)* function between each other without needing knowledge of their specific subclass type of *Entity*. The collision handling logic is centralised within the *CollisionHandler*, which dispatches the appropriate events to manage the collision.

If no events are dispatched, the collision will be resolved by adjusting the player's position to just before contact with the object.

Monsters

Monsters in our game serve as non-player characters (NPCs) tasked with patrolling specific regions. They are a testament to the adaptability of our Entity-Component-System (ECS) architecture.

Specifically, these are the different systems that monster entities share with player entities:

1) HealthSystem: Detects how much health a monster has
2) DeathSystem: Sets the timeout for death before disappearing
3) SpriteSystem: Animates the various monster actions, such as walking and dying
4) AttackSystem: Determines whether monsters should attack a player and how much damage to deal
5) MovementSystem: Moves the monsters

This design choice highlights the versatility and reusability of our ECS implementation, showcasing how various entities can efficiently share and utilise core gameplay systems.

Monsters patrol around an area by moving straight in one direction. When it collides with a player, obstacle or (invisible) patrol wall, there would be a MonsterMovementReversalEvent to change the direction of the monster's movement. Depending on how we want to further extend it, we can also make monsters jump.

Power-Ups

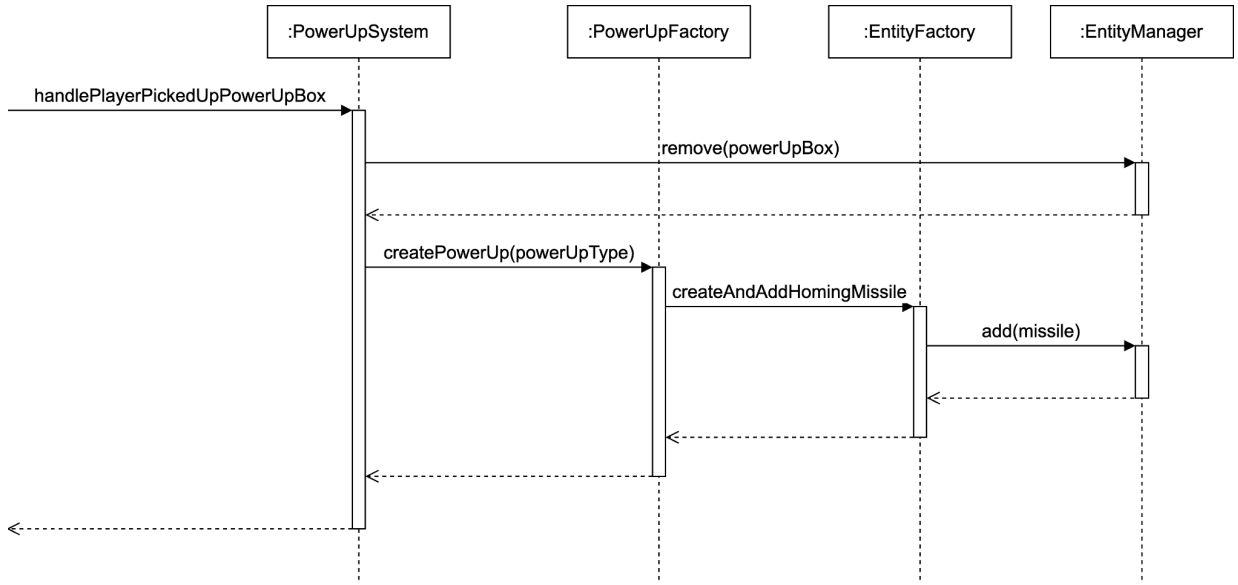
Power-ups in Star Dash refers to buffs/debuffs that players obtain during the games. Currently, only two power-ups are available in the game: Speed boost and homing missile. Furthermore, power-ups are obtained by collecting the power-up boxes found in the map.

The *PowerUpSystem* listens to the *PowerUpBoxPlayerEvent*, where the player picks up the power-up box and assigns a power-up to the player. For each type of power-up, the logic is encapsulated by a single system. This ensures clear separation of concerns between different types of power-ups and extensibility of introducing more power-ups into the game.

The following table depicts the power-up systems corresponding to each power-up:

Power-Up	**System**
Speed boost	SpeedBoostPowerUpSystem
Homing missile	HomingMissileSystem
Flying mode	FlyingModule - MovementSystem

The following sequence diagram describes how a homing missile is launched when the player picks up a power-up box:

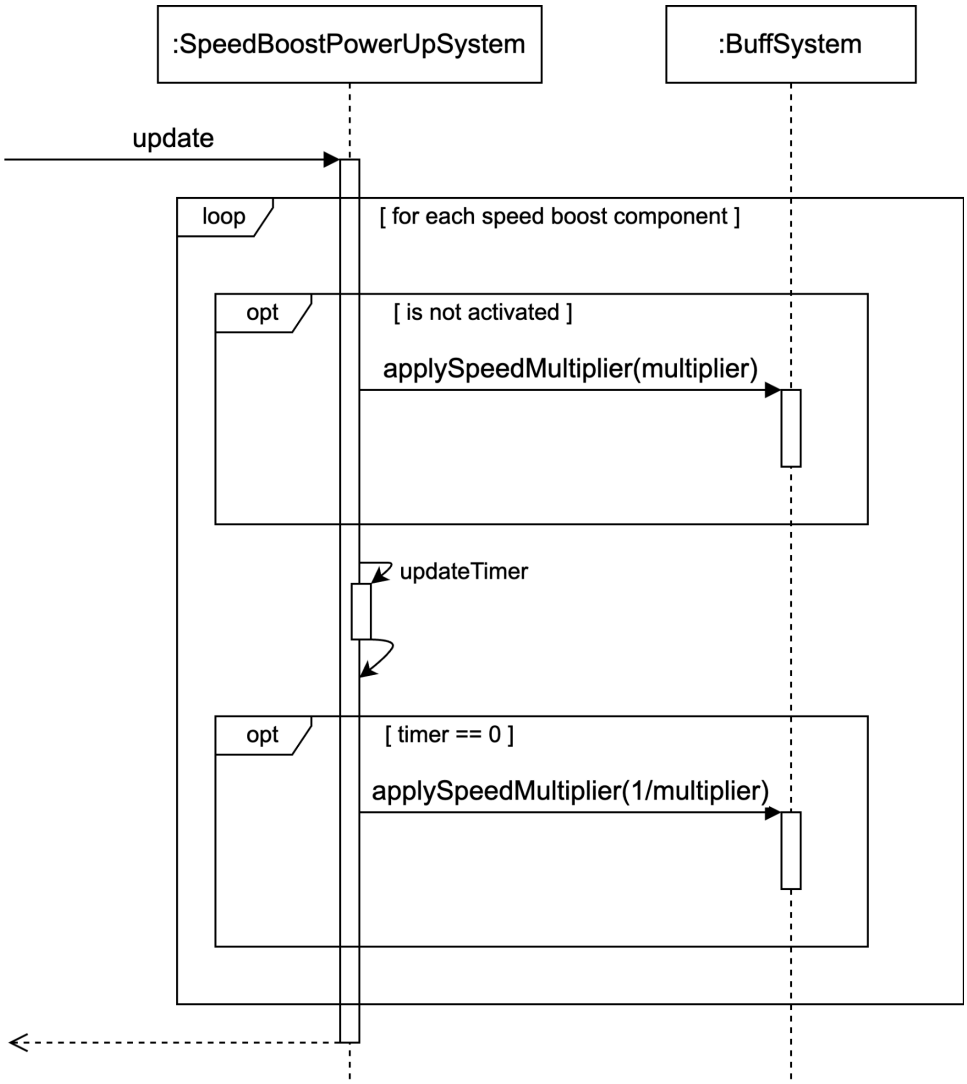


Speed Boost Power-Up

Similarly to the homing missile power-up, once a player picks up a power-up box that contains a speed boost power-up, a *SpeedBoostPowerUp* entity will be added to the game. At every update, *SpeedBoostPowerUpSystem* will activate or deactivate these speed boost power-ups.

When it activates the speed boost multiplier, it applies a movement speed multiplier to the player's *BuffComponent* through the *BuffSystem*. Likewise, it deactivates the speed boost by removing the movement speed multiplier through the *BuffSystem*.

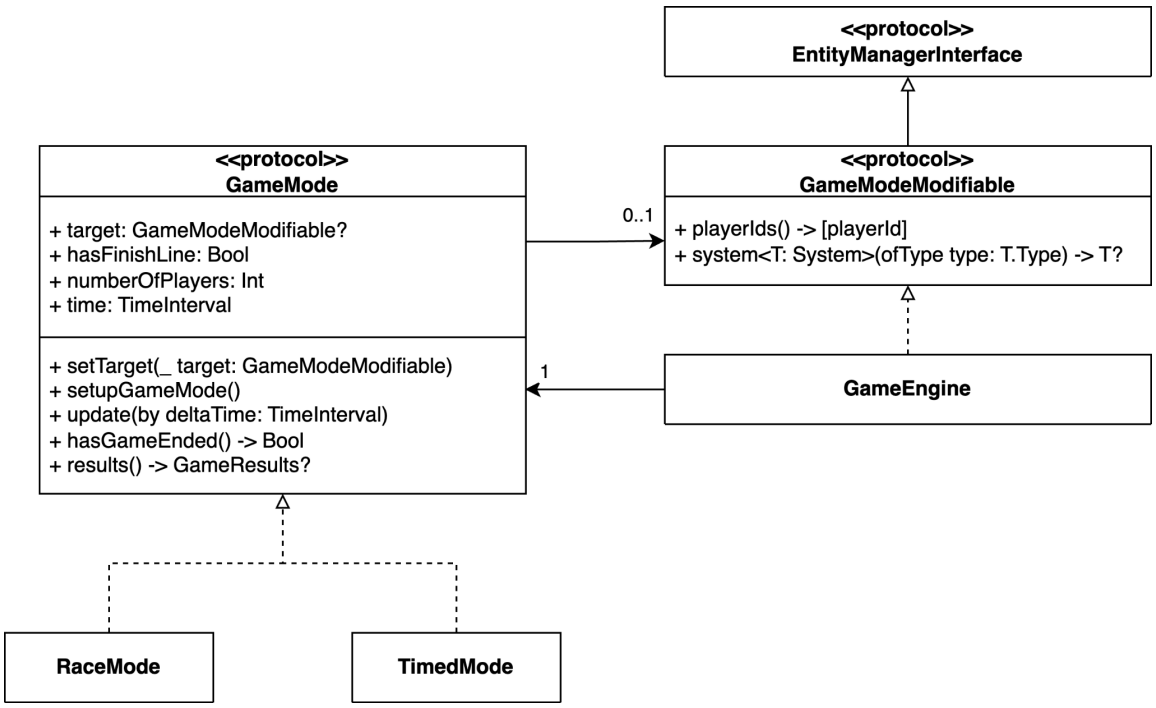
The following sequence diagram illustrates this logic for a speed boost power-up:



Game Modes

The *GameModes* module manages the logic specific to a game mode. Star Dash currently contains two game modes: Race Mode and Timed Mode. Each mode contains different logic to how the game needs to be set up, how the game ends, and etc. For instance, Race Mode ends when all players have crossed the finish line while Timed Mode ends in two minutes. The *GameModes* module is thus extensible to adding more game modes in the future.

The following class diagram outlines the classes and functions involved in handling game modes:



As seen in the above class diagram, *GameEngine* conforms to the *GameModeModifiable* protocol, which allows the following functions:

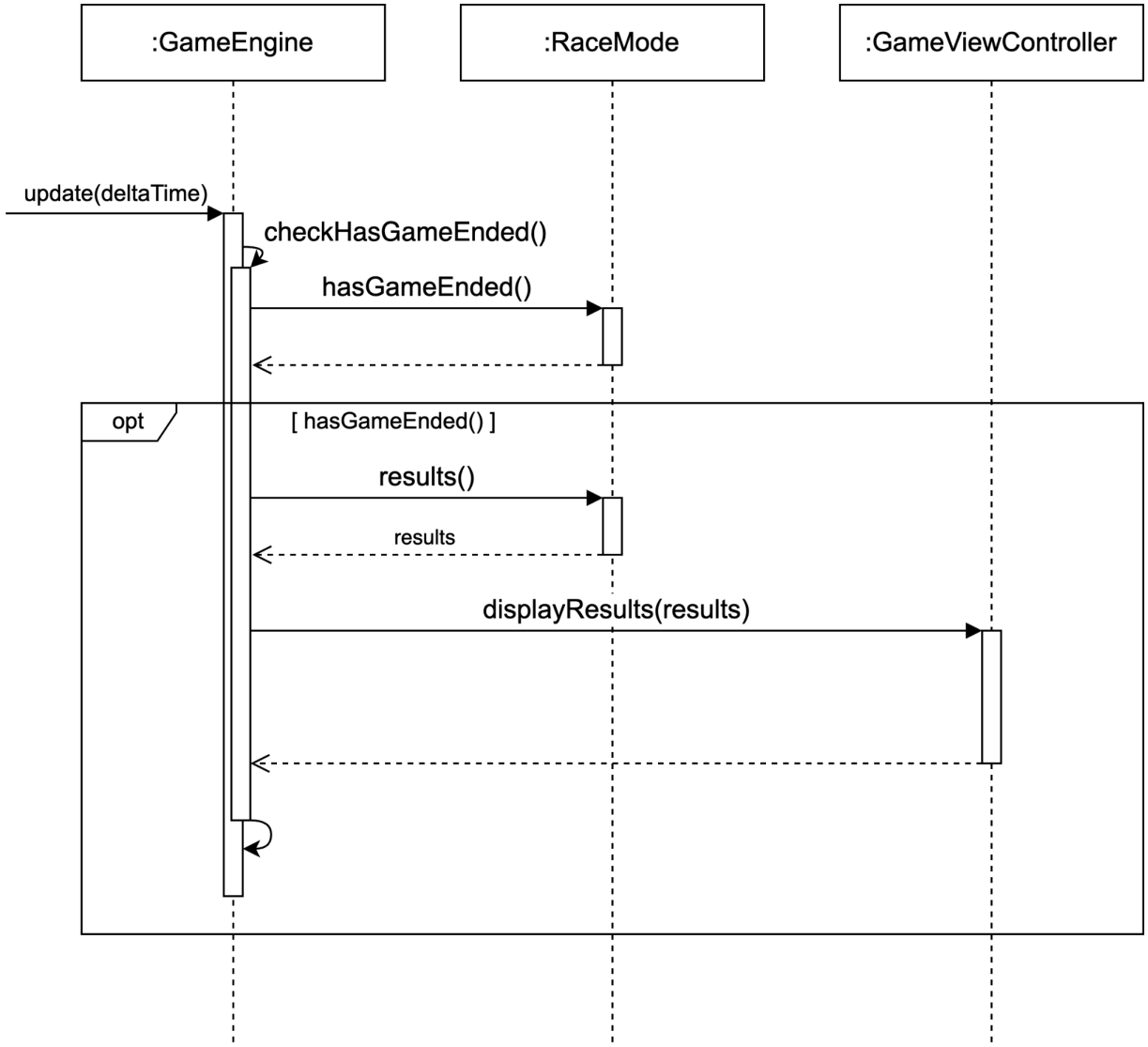
- Add new entities and components into the *EntityManager*
- Access the game's systems
- Fetch the ids of the player entities

Similar to *EventModifiable*, *GameModeModifiable* allows each game mode to access game information and make modifications to the *GameEngine*. By doing so, each game mode can implement logic specific to each game mode.

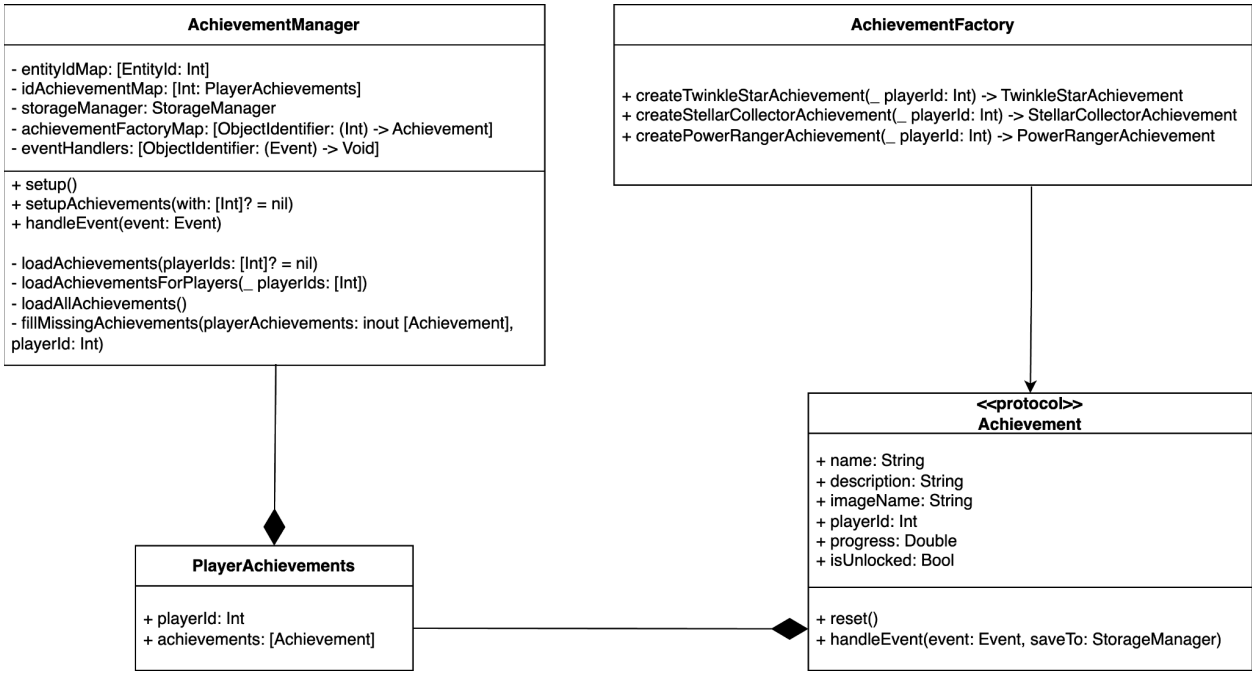
Ending the Game

The *GameModes* module is also in charge of checking whether the game has ended and generating the final results for display. The display of results follows a **delegate** pattern, where the task of displaying the results is handled by the delegate object (i.e. *GameViewController*).

The following sequence diagram shows the game-ending logic for Race Mode:



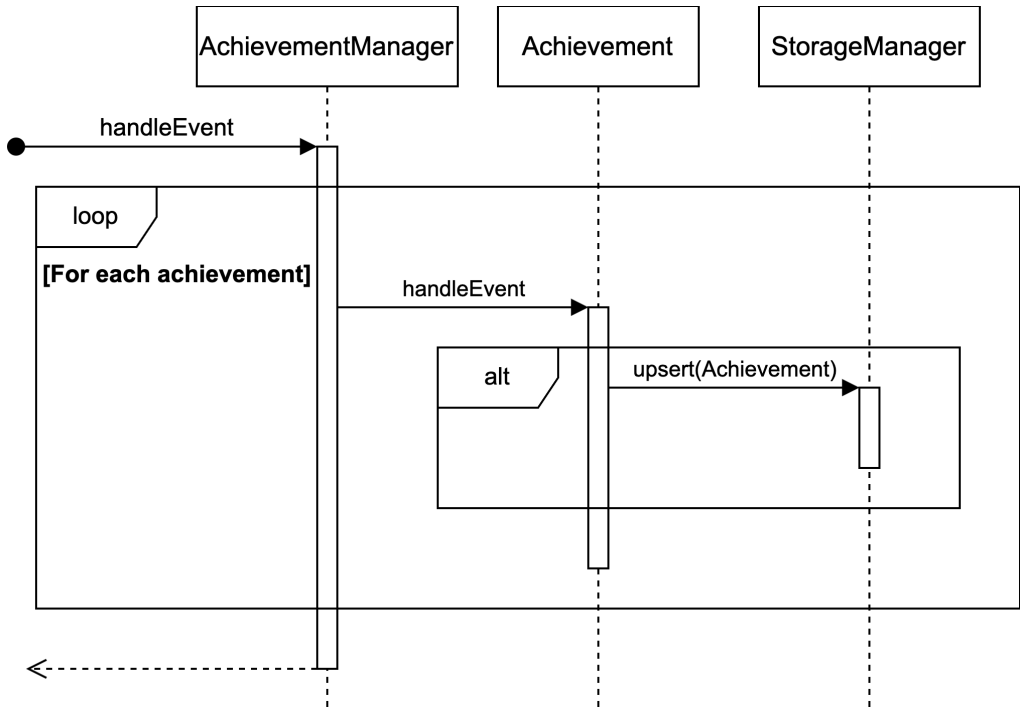
Achievements



The achievement system is developed as a standalone module, separate from the game engine. This separation ensures that the game and achievements are decoupled, allowing us to replace the entire achievement module or add new achievements without impacting the main game and vice versa.

The achievement system utilises the **observer** pattern. It achieves this by inheriting any necessary listeners. For instance, the *AchievementManager* extends the *EventListener* protocol and, based on emitted events, determines if achievements have progressed or been completed. This design allows for flexible integration of gameplay logic into the achievement system.

The following sequence diagram illustrates function class to manage achievements:



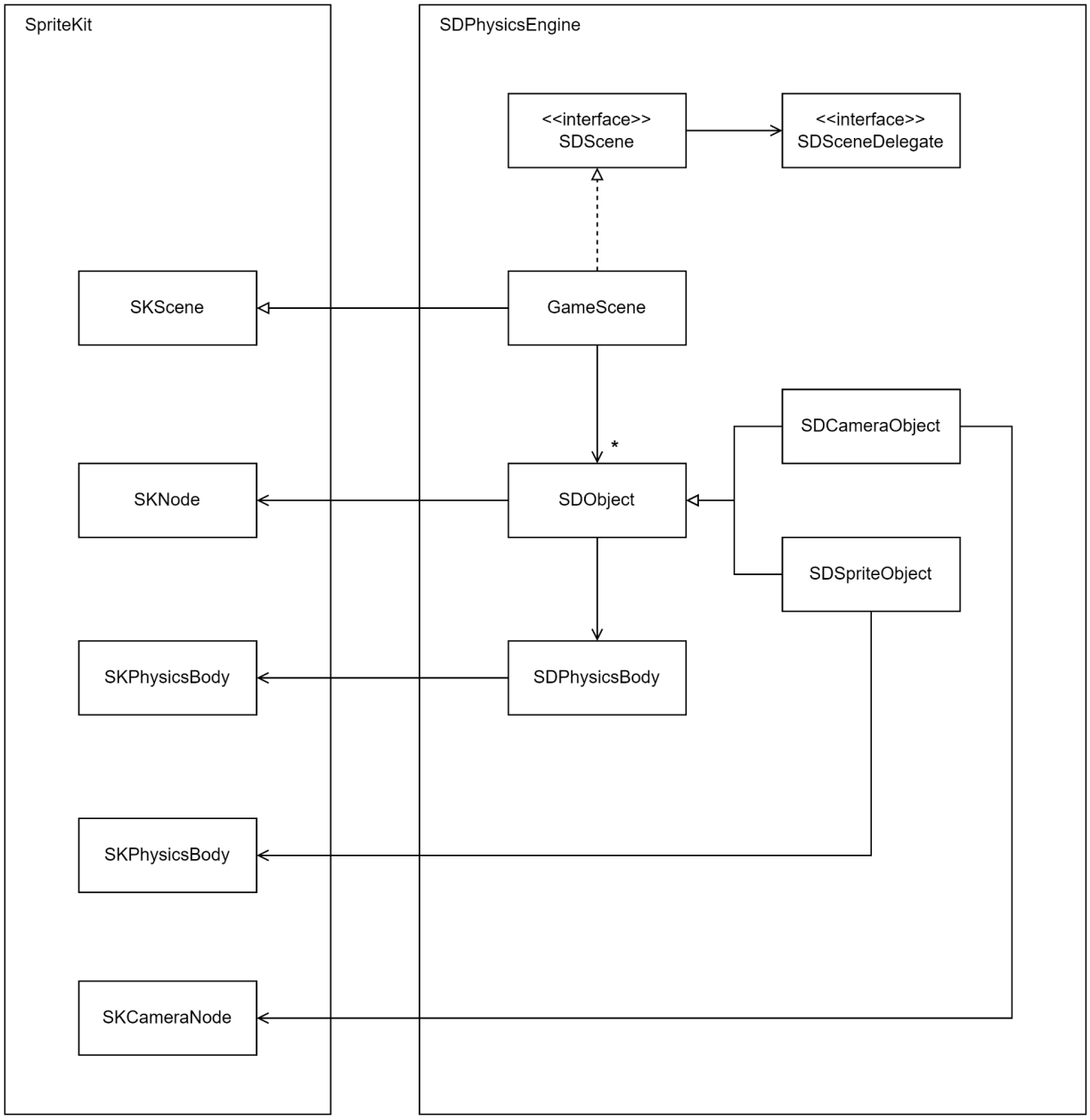
In addition to being decoupled, the system is highly extensible. If new achievements need to be added that are not related to gameplay, such as tracking the number of logins, the AchievementManager can simply extend a new listener and manage the logic accordingly. This approach ensures that the achievement system can easily adapt to future requirements.

All achievements and its progress is saved to the database for persistence.

Currently, the game has 3 achievements, outlined in the following table:

Achievement	**Description**
TwinkleStarAchievement	Collect your first star
StellarCollectorAchievement	Collect 10 stars
PowerRangerAchievement	Use at least 3 power-ups in a single game

SDPhysicsEngine



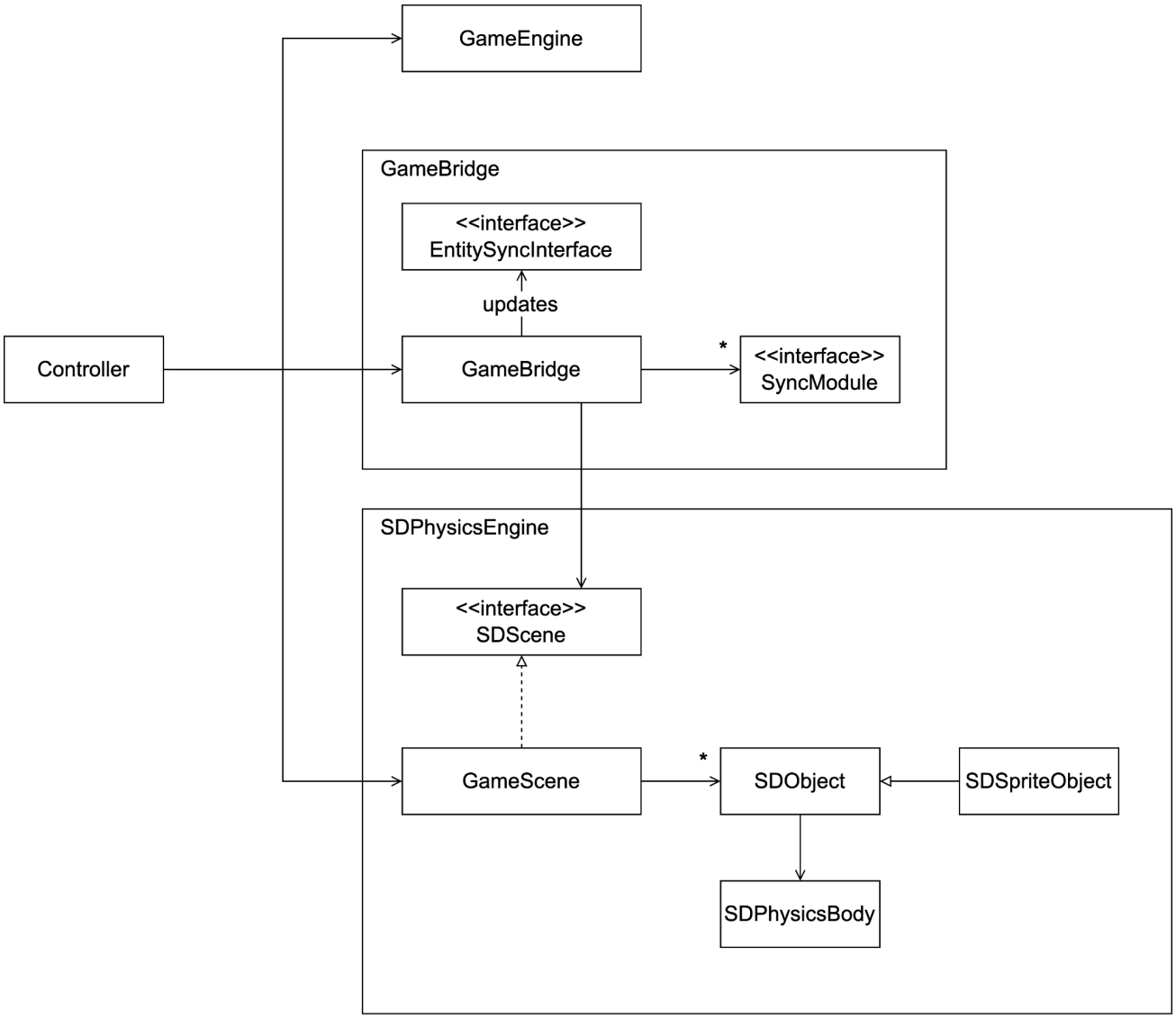
SDPhysicsEngine is a separate package that is responsible for simulating physics interaction between the game objects. All simulation work is powered by **SpriteKit**. Essentially, SDPhysicsEngine is a wrapper around SpriteKit and the codebase only interacts with SpriteKit through the interfaces and wrapped classes of SDPhysicsEngine. Thus, SDPhysicsEngine uses a **facade** design pattern, acting as an interface to the underlying physics engine.

The rationale for creating this package was to take advantage of SpriteKit's physics engine while reducing the coupling between the main codebase and SpriteKit. Defining wrapper classes and interfaces ensures that SDPhysicsEngine is the only package that has immediate dependencies to SpriteKit.

Game Bridge

GameBridge is responsible for synchronisation of states between SDPhysicsEngine and GameEngine. As SDPhysicsEngine maintains its own representations of the game in a physics world and GameEngine has its own set of entities in the game world, GameBridge ensures that the game world and physics world are consistent with each other. Thus, GameBridge acts as a bridge between these 2 sets of objects.

Module Structure



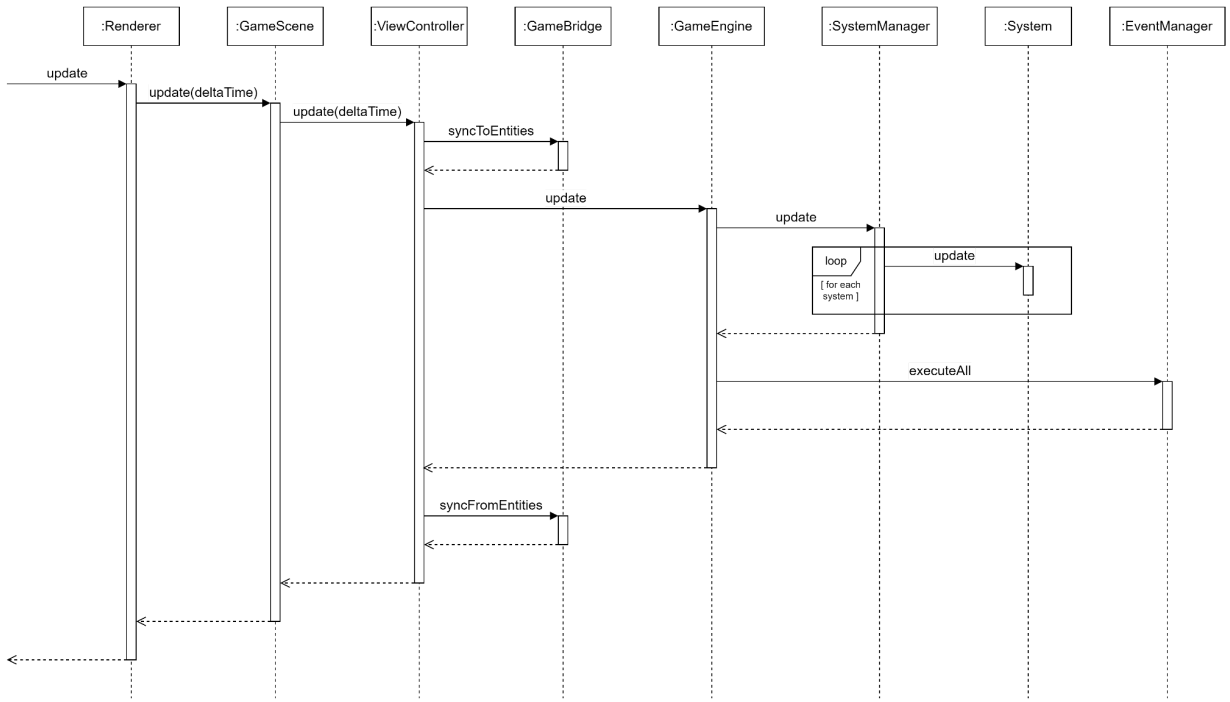
GameBridge maintains maps between GameEngine's entities and SDPhysicsEngine's *GameScene* objects. Furthermore, it holds a collection of *SyncModule*. Each *SyncModule* provides an implementation to sync a specific component of the GameEngine's entities and *SDPhysicsEngine*'s objects. For example, *PositionModule* ensures the objects and entities are consistent in position and rotation while *PhysicsModule* ensures the objects and entities are consistent in velocity and other physical properties.

The following table details the *SyncModule* and their individual functions:

SyncModule	**Function**
ObjectModule	Syncs the position and rotation between *SDObject* and *PositionComponent*
PhysicsModule	Syncs the velocity, mass and other physical properties between *SDPhysicsBody* and *PhysicsComponent*
SpriteModule	Syncs the sprite information between *SDSpriteObject* and *SpriteComponent*

Update Lifecycle with Game Bridge

The following sequence diagram showcases how the *GameBridge* has to sync twice as both *SDPhysicsEngine* and *GameEngine* updates separately.

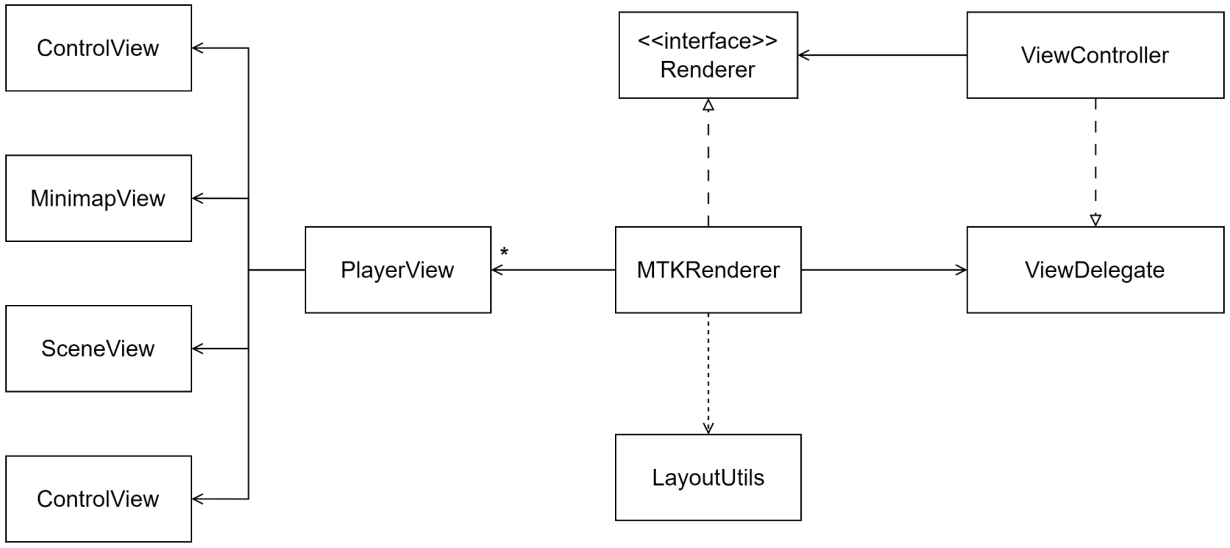


When the *Renderer* receives an update call, it relays this update request to the *GameScene*. *GameScene* would first update the objects in its physics scene, then inform the *ViewController* that the scene has completed updating. The *ViewController* first requests *GameBridge* to sync the *GameScene* objects into the *GameEngine* entities/components.

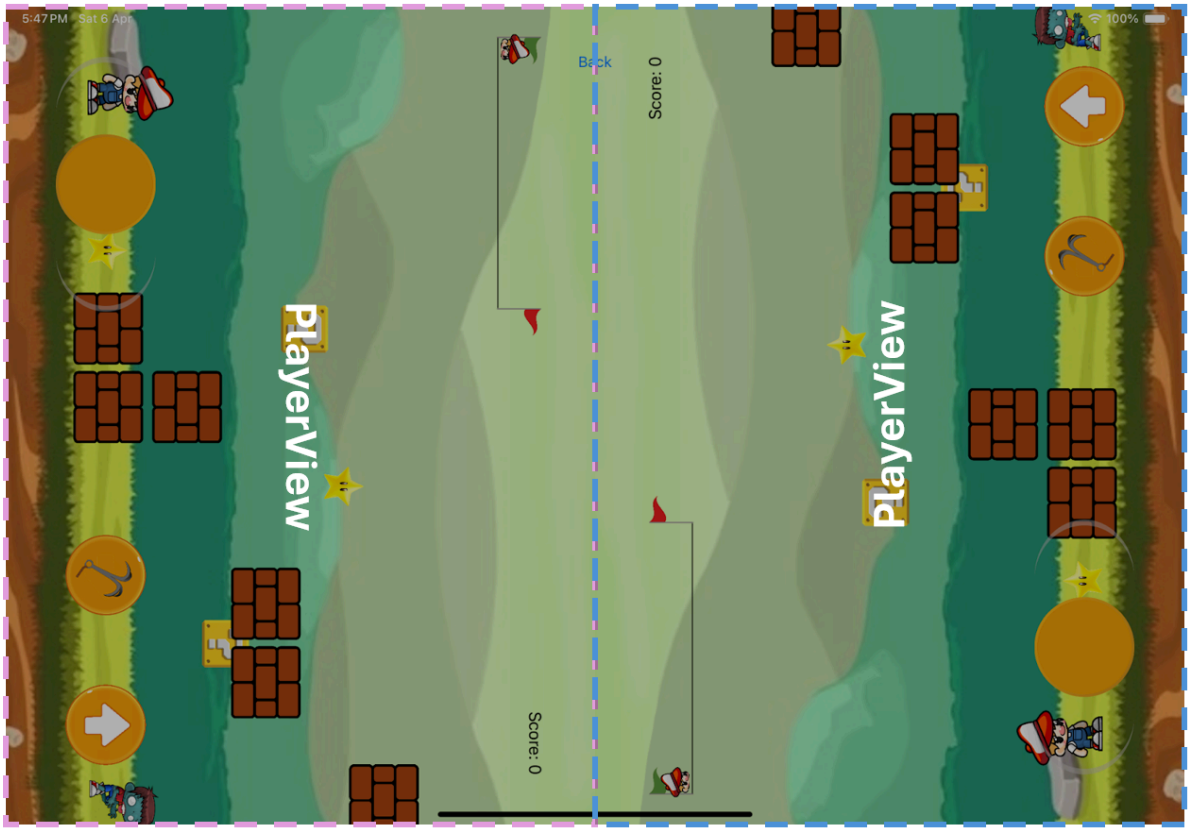
Afterwards, the *GameEngine* will handle the updates of the game. It first updates all of its systems registered in the *SystemManager*, then executes all events in the *EventManager*'s event queue.

Before ending the update cycle, *GameBridge* will sync the *GameEngine* entities/components into the *GameScene* objects. Finally, the *Renderer* will render from the updated game objects.

Rendering



Rendering is coordinated by *MTKRenderer*, which renders the game on to the view. In a multiplayer mode, it is also responsible for coordinating the rendering on multiple views on a single screen.



As seen in the picture above, for each player's perspective, *MTKRenderer* initialises a PlayerView, which is a collection of 4 views:

- MTKView: Displays the game scene
- OverlayView: Displays the game information, eg. score
- MinimapView: Displays the minimap to the user
- ControlView: DIsplays the user input controls, eg. joystick

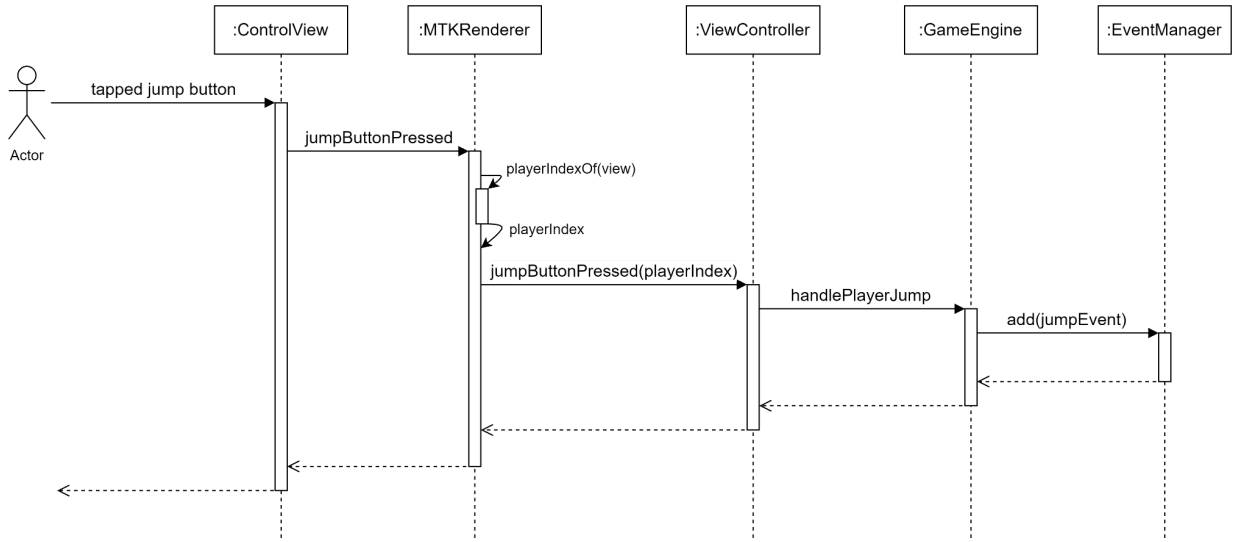
Traditionally, SKView is used to render SpriteKit's SKScene, however, there is a limitation that only one SKView can be presented at any point in time. Therefore, to prepare ourselves for future support of local multiplayer, SKView is not a feasible option and we use MTKView instead.

Local Multiplayer Views

LayoutUtils is what empowers the creation of various layouts for local multiplayer (and singleplayer). When setting up the views for each player's perspective, *LayoutUtils* first splits the given view into subviews for the instantiation of each *PlayerView*. This enables the layout of PlayerView's subviews (etc. controls buttons) to be independent of its location or rotation.

Player Input

The following sequence diagram depicts a situation where the user clicks on a jump button:



When the user clicks on the jump button, the *ControlView* relays this event to the *MTKRenderer* which first determines which player the event belongs to before relaying it to the *ViewController*. Afterwards, the *GameEngine* will insert a *JumpEvent* to the *EventManager*'s event queue. The *JumpEvent* will later be executed by the *MovementSystem* when the *GameEngine* next updates.

Network

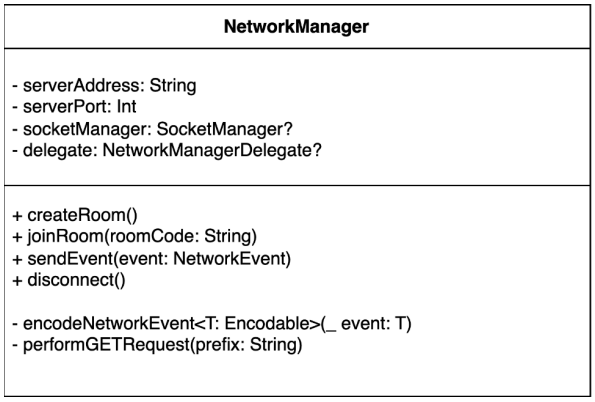
Developed as a standalone module, the network system handles the making of API requests and managing of the WebSocket for the game. This ensures that the game and network are decoupled, allowing us to replace the network module if necessary.

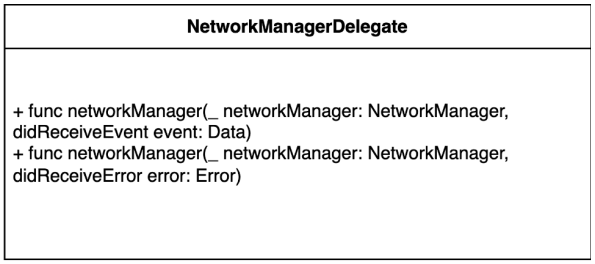
Network Manager

The *NetworkManager* acts as an abstraction layer over the networking used. This ensures that the type of networking used is decoupled from the game, allowing us to swap out the type of networking used easily.

Currently, *NetworkManager* uses a combination of **HTTPS protocol** over **TCP protocol** for API calls and **Websocket** for bidirectional communication with the server.

NetworkManager follows a **delegate** pattern, where events that are received over the network are handled by the delegate.

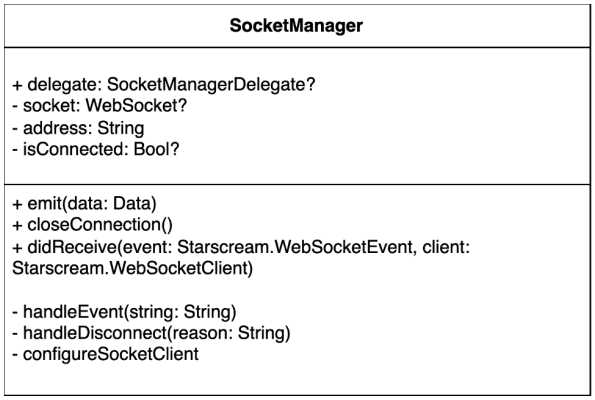


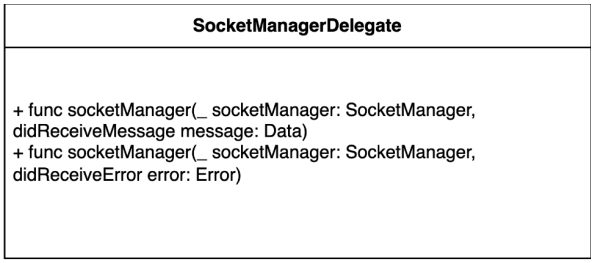


Socket Manager

The *SocketManager* is an abstraction layer over the WebSocket library StarScream. This allows us to replace the library by just changing the implementation in *SocketManager*.

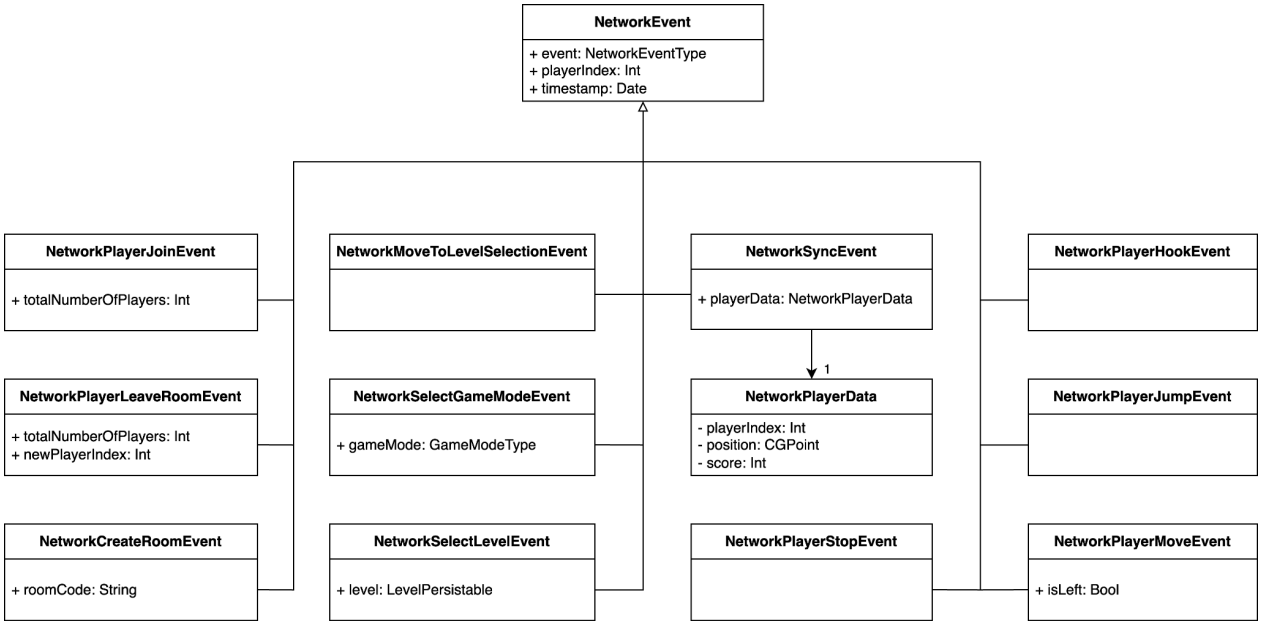
SocketManager follows a **delegate** pattern, where events that are received over the socket are handled by the delegate object.





Network Event

The network events are the events that are sent or received from the WebSocket. All network events inherit from *NetworkEvent* and implement *Codable* to ensure that we can encode when sending events and decode when receiving events.



Server

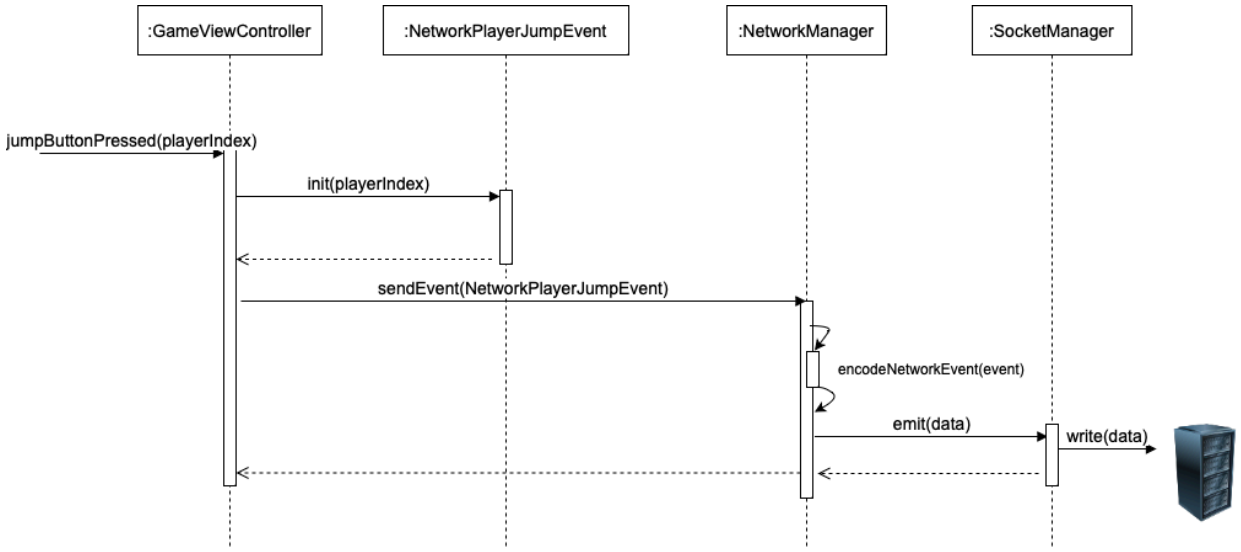
In online multiplayer games, *NetworkEvents* that are received are broadcasted to all connected clients in the room.

The life cycle of the crucial events in an online multiplayer game is explained below.

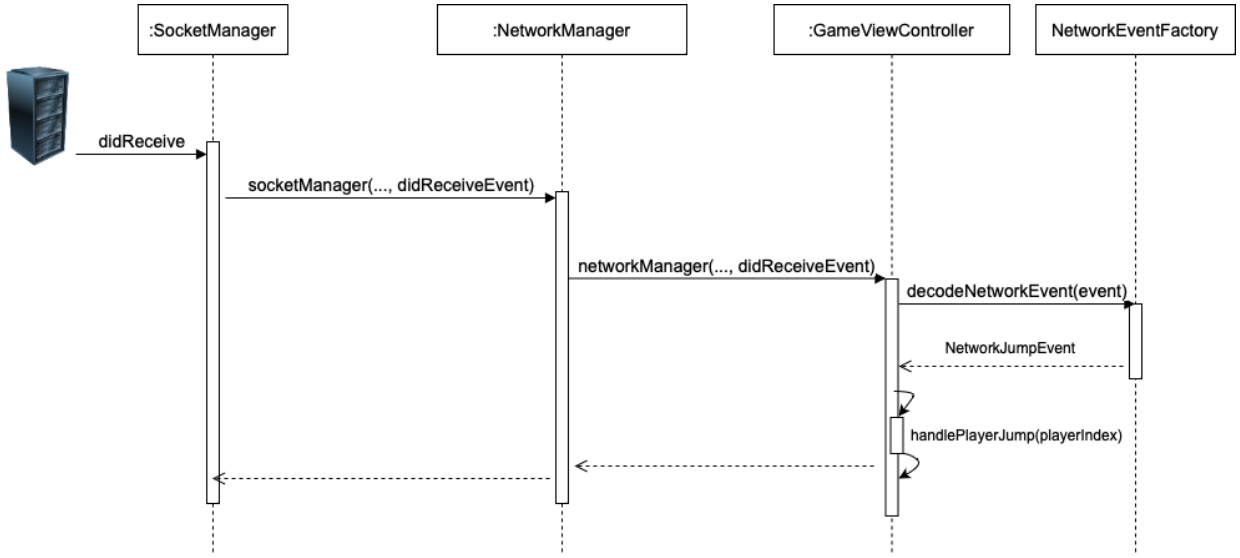
Handling player movement

Player movements are sent to the network instead of being directly handled to reduce the time difference between player movement in each client.

As an example, below is the sequence diagram for sending and receiving of a jump input:



When a jump button is pressed, the *GameViewController* creates a *NetworkPlayerJumpEvent*, then it sends the event through the *NetworkManager*, which then encodes the event, and then emit it through the *SocketManager* which then write the event to the server, which will then be broadcasted to all other connected clients.



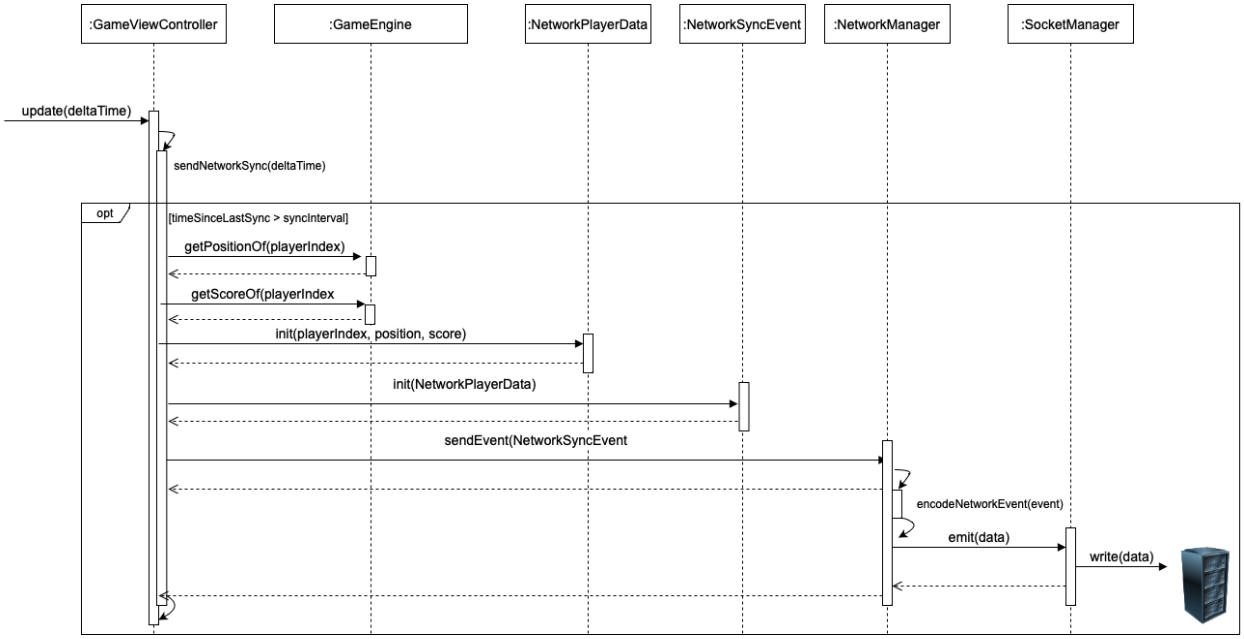
The jump event is received by the client through the *SocketManager* which acts as a delegate to the Websocket. It is then received by the *NetworkManager* acting as a delegate to *SocketManager*, which is then received by the *GameViewController* acting as a delegate to the *NetworkManager*. The *GameViewController* then decodes the received data to *NetworkJumpEvent* using the *NetworkEventFactory*. The *GameViewController* then handles the player jump accordingly.

Keeping players in sync

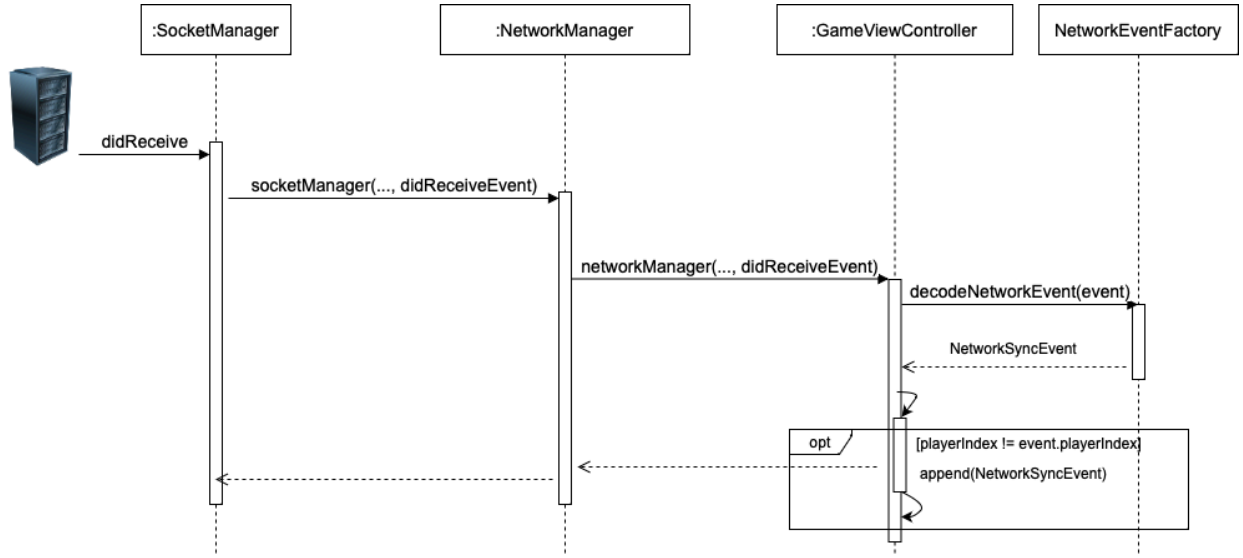
As the server is acting as a broadcaster of events, each client in the room runs the game based on its own game engine, as such the positions of other players on one client will merely be predictions based on the inputs that are received.

This would cause an issue as the input events that are sent could be delayed when being sent to the clients due to indeterministic network traffic. This could cause a slight delay in executing the event, which could result in a difference in position which could trickle down to whether a player hits an obstacle or not. For example, a 10px difference in position of a player in client A and client B means that the player in client A could hit a obstacle and client B could miss the same obstacle.

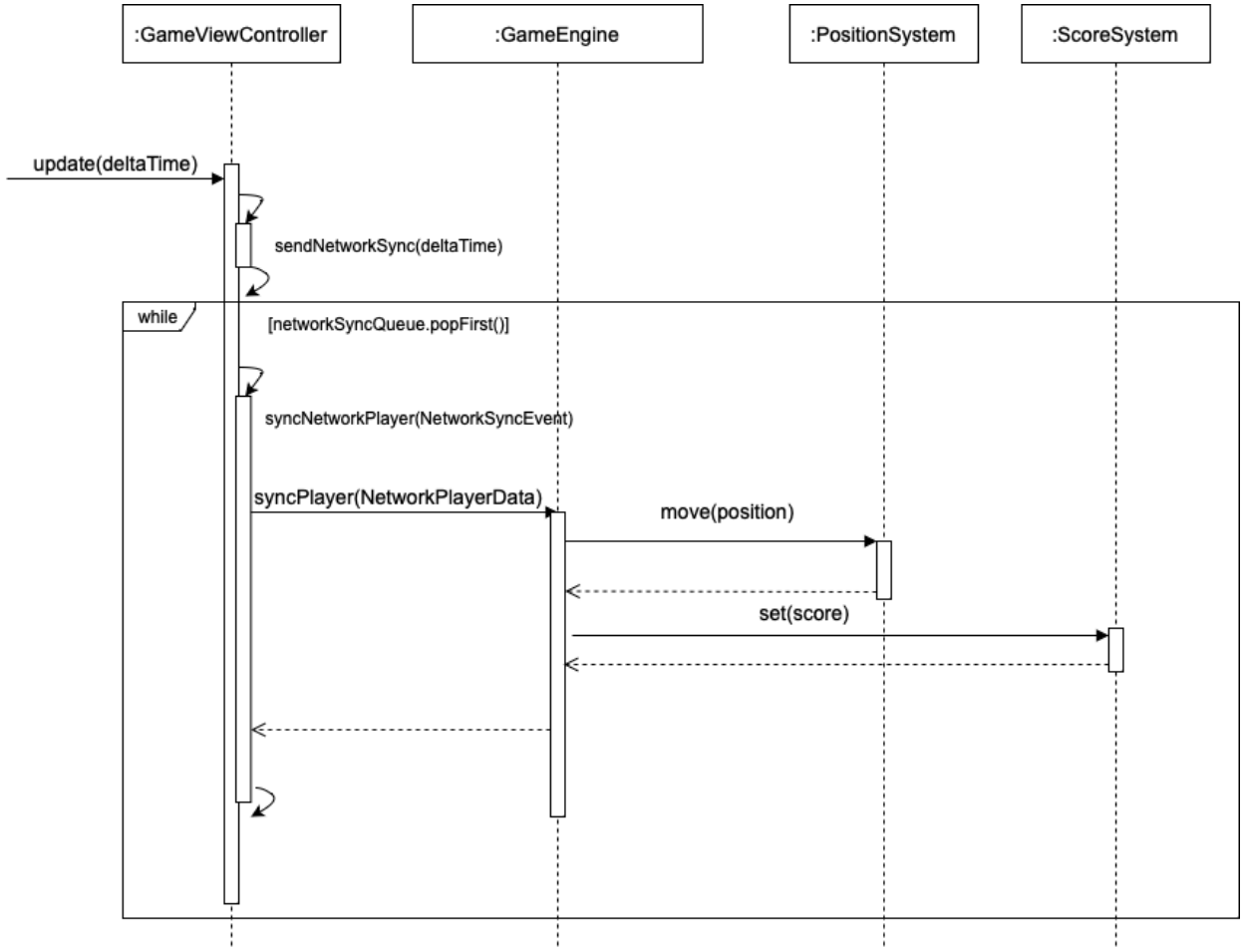
As a solution to this issue, we have the “correct” position of the player be the position of the player on the client that is controlling it. The important data of the player that is controlled by the client-like position and score is sent to the server every 0.3 seconds.



When an update is called on *GameViewController* and the *timeSinceLastSync* is more than *timeSyncInterval*. Then, the position and score of the player is taken from the *GameEngine* and then a *NetworkPlayerData* is created, followed by a *NetworkSyncEvent*, which is then sent through the *Network* Manager which is then sent through the *SocketManager* and then sent to the server which is then broadcasted to the other clients.



When a *NetworkSyncEvent* is received through the *SocketManager* acting as a delegate to the WebSocket, it is then received by the *NetworkManager* acting as a delegate to the *SocketManager*, which is then received by the *GameViewController* acting as a delegate to the *NetworkManager*. The *GameViewController* decodes the event using the *NetworkEventFactory* to *NetworkSyncEvent*. The *GameView* controller then checks if the sync event is from the player controlled by this client; if it is not, then the sync event is added to a SyncEventQueue (not shown on diagram).



During an update cycle on *GameViewController*, it then checks for events in the *NetworkSyncQueue* and loop through the queue, and syncs each player. During the sync of each player. The position and score of that player is updated through the GameEngine.

Using this method of ensuring that the position of each player is synced, we ensure that the player controlled by the player will not lag and ensures a smooth experience for the users. However, the players of the other clients on a client screen could lag when a wrong position is updated. For example, the position of a player A on the screen of

client B could be wrong for a maximum of 0.3s + Time to send to server + Time to send to client before being corrected (Assuming no packet loss).

Testing

We plan to use a mixture of Black-Box Testing as well as White-Box Testing to test the application.

Black-Box Testing

Black-box testing focuses on testing the game's functionality and user experience **without** knowledge of its internal workings.

This approach will be used to test the following sections:

1. Player movement
    a. Player should move in the direction of the joystick when it is moved
    b. Player should stop moving when joystick is released
    c. Player should jump when the jump button is pressed
    d. Player should not be able to jump while in the air
2. Collision Detection
    a. Player should interact correctly with collectible game object and collectibles should disappear after collision
3. Score Mechanics
    a. Score should be correctly updated when collectibles are collected
4. Compatibility
    a. Level should be scaled correctly on different ipad sizes

White-Box Testing

White-box testing focuses on testing the correctness of individual functions and components.

This approach will be used to test the following sections:

1. Unit testing of manager classes
    a. Testing of individual components, like EntityManager, SystemManager, EventManager to ensure correctness in updating data

Reflection

Evaluation

We spend quite a lot of time on ensuring the extensibility of the game objects. Additional behaviour for a new or existing entity will only need component(s) to be attached to the entity. For example, if we want monsters to give points when killed, we just need to add a *PointsComponent* to the *MonsterEntity*.

Lessons Learnt

Decoupling of External Libraries

We used SpriteKit for animation and physics handling, however we recognise that coupling an external library to the code is bad practice and could increase the complexity of maintenance. To resolve this, we created *SDPhysicsEngine*, a wrapper around SpriteKit. By doing so, StarDash has no coupling with SpriteKit. This implies that if we were to change the physics engine used, we will only need to modify SDPhysicsEngine and no change is needed in the codebase of StarDash.

Unrecognised Accrual of Technical Debt Through Incremental Feature Additions

During the development of our game, we initially implemented running and jumping as separate movement systems. As we expanded the game's mechanics, we introduced tools like grappling hooks to enhance traversal speed.

However, when the challenge of integrating flying mechanics arose, we discovered that the existing movement systems were tightly coupled, leading to an unforeseen accrual of technical debt. This realisation prompted a comprehensive redesign of our movement logic to ensure modularity and separation of concerns. We learned to take a bigger point of view as we make modifications or additions to our software. We learned the importance of anticipating future feature expansions and designing systems with flexibility in mind from the outset. Through iterative development and refactoring, we not only addressed the immediate challenge but also established a more adaptable foundation for future enhancements.

Limitations

1. Sync of non player object in online multiplayer
    a. As only the players are sync in online multiplayer, there is a possibility that the position or presence of other objects like monsters could differ from client to client
    b. It will differ in the event that the GameEngine is started at different timing due to a lag in the start of the game triggered by a NetworkEvent
    c. It will also differ in the event that a online player's position is wrong and it interacts with a game object before it's position is corrected by a network sync event
2. Error handling of connection drops
    a. Currently the app only handles connection drops in the lobby, ensuring that the correct number of players is spawned
    b. However if a client's connection is dropped during level selection, game mode selection or during the game, the disconnection is not handled.
    c. This means that there is a possibility that the game will not end if a player is disconnected in Race Mode
    d. This limitation is not due to technical limitation but rather a lack of time in implementing the handling of disconnection and can be solved in future iterations

Appendix

Individual Contributions

Name	Contributions
Lau Rui Han	Entity-Component System, Online Multiplayer, Rendering
Qiu Jiasheng, Jason	Entity-Component System, Game Modes, SDPhysicsEngine
Goh Jun Yi	Rendering, Movement System, Power-Ups, SDPhysicsEngine
Ho Jun Hao	Entity-Component System, Achievements, Movement System

Sprint 1 Task List

Task	Status
Model - Game Engine	Completed
Model - Physics Engine Wrapper	Completed
Rendering for Single-Player	Completed
Basic Collisions	Completed
Run and Jump with Controls	Completed
Collectibles	Completed
Score System	Completed
Game Initialisation from Preloaded levels	Completed

Sprint 2 Task List

Task	Status
Restructure ECS and Events	Completed

Monsters system	Completed
Power-up system	Completed
Sprite animations and textures	Completed
Homescreen	Completed
Level selector	Completed
Achievements	Completed
Grappling hook	Completed
Minimap	Completed
Local multiplayer	Completed
More preloaded levels	Completed

Sprint 3 Task List

Task	**Status**
Clean up unresolved bugs and issues	Completed
Different game modes	Completed
Sound engine	Completed
Additional power-ups	Completed
Online multiplayer	Completed
Redesign movement system	Completed

Unresolved Bugs and Issues

- Cooldown for using tools
    - Expected behaviour: There should be a cooldown to consecutively use a grappling hook, so that it forces players to use the tools strategically.
    - Actual behaviour: There is no cooldown.