CS3217 Software Engineering on Modern Application Platforms

Final Project Report

Cloud Jumpers

Submitted by

Eric Bryan	A0220597B
Hoang Trong Tan	A0219767M
Phillmont Muktar	A0184545X
Sujay R Subramanian	A0180309J

Table of Contents

Project Overview

Cloud Jumpers is an online multiplayer vertical platformer for iPad designed for 1 to 4 players. In general, the aim is to reach the uppermost end platform at the top layer of the game world as fast as possible. To reach the top, there will be many clouds for players to jump on. Along the way, there will be many challenges that cause players to restart from the bottom, such as random falling meteors, or disruptions from other players who activate power-ups.

Features and Specifications

Player	**Movement** - A draggable joystick is used to move the player horizontally. - The speed is proportional to the distance between the centre of the inner stick to the centre of the outer stick. - When releasing the joystick, the player will stop moving in the horizontal direction. **Jump** - A jump button is used to jump a player vertically upwards. - Direction of jump can be adjusted using the draggable joystick. - Pressing the jump button while a player is in the air (either is already jumping or falling) will have no effect. **Store PowerUp** - When a player collide a powerUp, the player will store a powerUp in his inventory queue, which can be dequeued and activated - In multiplayer mode, if two players collide with the same powerUp, the first player that hits first will obtain the power up **Activate PowerUp** - A player can activate the powerUp by tapping on the game play area. PowerUp will be activated in a first-in-first-out order from the inventory queue.
Clouds	- A platform that a character can jump and step on. - A character cannot go through the clouds. - Moving out of the clouds will cause characters to fall. - The bounding area is a rectangle. - Some clouds can move back and forth in the horizontal direction
End Platform	- Located at the very top of the game world.

	- The end goal where a character has to step on as fast as possible. - The bounding area is a rectangle.
Timer	- Updated every 0.1s that shows how long a player has taken to play the game.
Natural Disasters (like meteors)	- At any point during the game, there may be disasters (such as meteors) being activated anywhere on the screen. - A player is to avoid getting hit by this meteor - A player who is hit by the meteor will respawn back to its original starting position
Lobby	- A holding area where up to 4 players can go in to play a multiplayer mode. - A host player can type in different seeds to generate different layout - A host player can choose a game mode in the lobby.
Single player (Time Trials)	- A timer is used as an indication of how well a player has performed for this mode. - A player just has to jump to the end platform as fast as possible, via the intermediate clouds. - Once a player reaches the end platform at the top, the timer will stop and a dashboard will be shown where the player's final timing is shown alongside with the 10 best timings ever recorded.
Multiplayer (Race to the Top)	- Up to 4 players can play this mode at the same time. - The players are to compete to reach the end platform as early as possible. - If any two players are on the same cloud, the first person will respawn back to his original starting position. - PowerUps can be obtained and activated to disturb other players. - Once a player reaches the end platform at the top, he will be brought to a dashboard page, showing his timing, while waiting for other players to finish the game.
Multiplayer (King of the Hill)	- A multiplayer game that utilises a point-based scoring system - The closer a player is to the top, the higher the points he gets per unit time. - The objective is to stay at a higher position for as long as possible. - Two players on the same cloud will respawn the first player back to the original starting position - PowerUps can be obtained and activated to disturb other players - A player who reaches the end platform will attain a God

	role. A God can scroll the whole scene, and activate power-ups to disturb others from reaching the top. - In this mode, two players who reach the top platform will respawn the first player. - The game stops after the timer reaches 120

User Manual

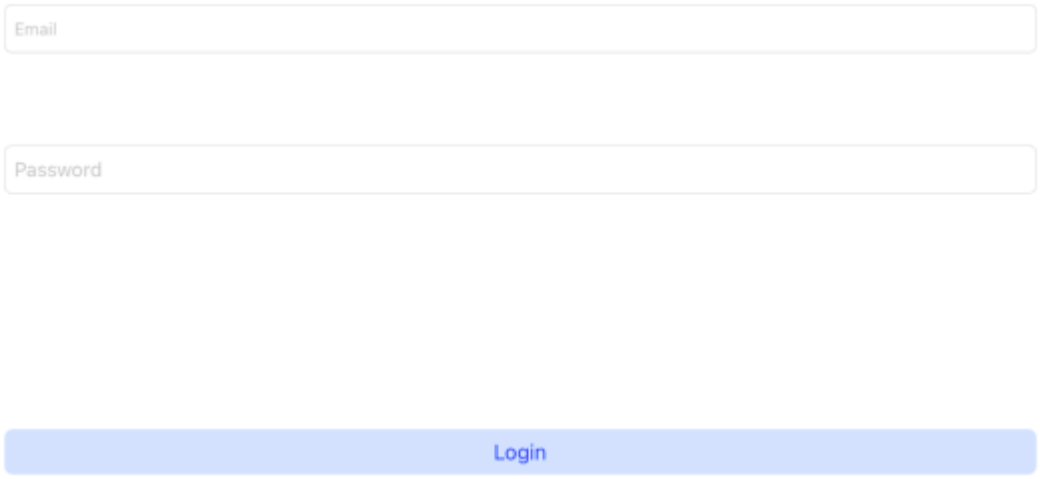
Thank you for choosing to play Cloud Jumpers, a competitive 1-4 player iOS platformer game with powerups, different game modes, kill mechanics, and more!

Cloud Jumpers includes both single-player and multiplayer gameplay. It requires an active internet connection to play.

Entering the game

Upon starting the game, you will arrive at a login page.

If you do not have an account yet, you can press sign up to create a new Cloud Jumpers account. To do so, you will need to provide a valid email address, a strong password as well as your name. Choose your name wisely, as it will appear in-game and within the highscores!



Once you have an account, you can use your credentials, consisting of a valid email and password, to login.

Viewing lobbies

Upon logging in, you will be redirected to the lobby selection screen, which looks like this:

In this screen, you can perform the following actions:

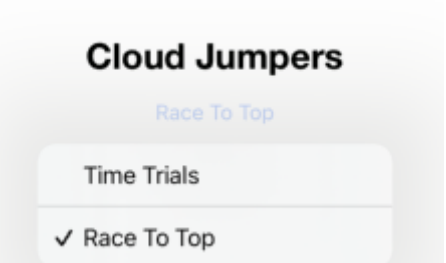
- View your achievements, by tapping on the button on the top left. Doing this will take you to the achievements page.
- Sign out, by clicking on the button on the top right. Doing this will take you back to the login screen.
- Create a new lobby, by tapping on the New Lobby button at the top of the screen. Doing so will take you to a specific lobby's screen.
- Join existing lobbies, by tapping on the cells that appear on the screen.
    - Greyed out cells are indicative of games in progress, and cannot be joined.
    - Green cells are indicative of games that have yet to begin, and are currently accepting new players. Tapping on one of these will cause you to join the corresponding lobby, taking you to the specific lobby's screen.

Game Lobby & Configuration

Upon tapping on a lobby or the create lobby button, you are brought into your lobby, where you can perform the following actions:

1. You can press ready, which will make the background of your player's entry turn green, as shown below. When ready, you will no longer be able to leave the lobby. When all players in the lobby are ready and a sufficient number of players are in the lobby for the chosen gamemode, the game will begin automatically.

2. Change the game mode, provided you are the lobby host and have not indicated that you are ready. You cannot change the game mode if you are in a ready state.

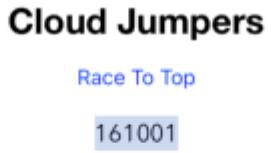


Game modes

There are three game modes available for you to play, as defined below.

Game Mode	Players	Description	Available features
Time Trial	1	Reach the end platform as fast as possible. Ranking will be based on Fastest timings across all players for the seed.	- Random Event (Meteors) - Shadow Player of the top ranked player on the highscores
Race To Top	2 - 4	Be the first to reach the top platform. Rankings are relative positions.	- Random Events - Kill mechanics - Powerups
King of the Hill	2 - 4	Game lasts for 120 seconds. Accumulate points every second, with more points being awarded the closer you are to the top. At the top platform, you have access to an unlocked camera, and randomly generated power ups to prevent others from climbing higher.	- Random Events - Kill mechanics - Powerups - Top platform unlocked camera - Top platform random powerup generations

3. Change the level seed, provided you are the host and are not ready. This is a similar restriction to being able to change the game mode. To do so, tap on the number (highlighted below), and the keyboard should pop up, allowing you to change the game seed.



The game seed will be randomly generated for every lobby. You can specify the seed in order to play previously seen maps again.

4. Pressing the leave button will remove you from the lobby, and move you back to the lobby selection screen.

Navigating through the game world

Controlling the player

To move a player, drag the joystick for horizontal movement. The speed of the joystick is proportional to the distance between the centre of the inner joystick to the centre of the outer joystick. To jump, tap on the jump button. A player cannot jump while in the air. To adjust the jump direction, drag the joystick while the player is jumping.

To obtain a power-up, simply collide with the power-ups you see on the screen. To activate a power-up, simply tap on the screen where you want the power-up to be activated, provided you have at least one in your inventory queue.

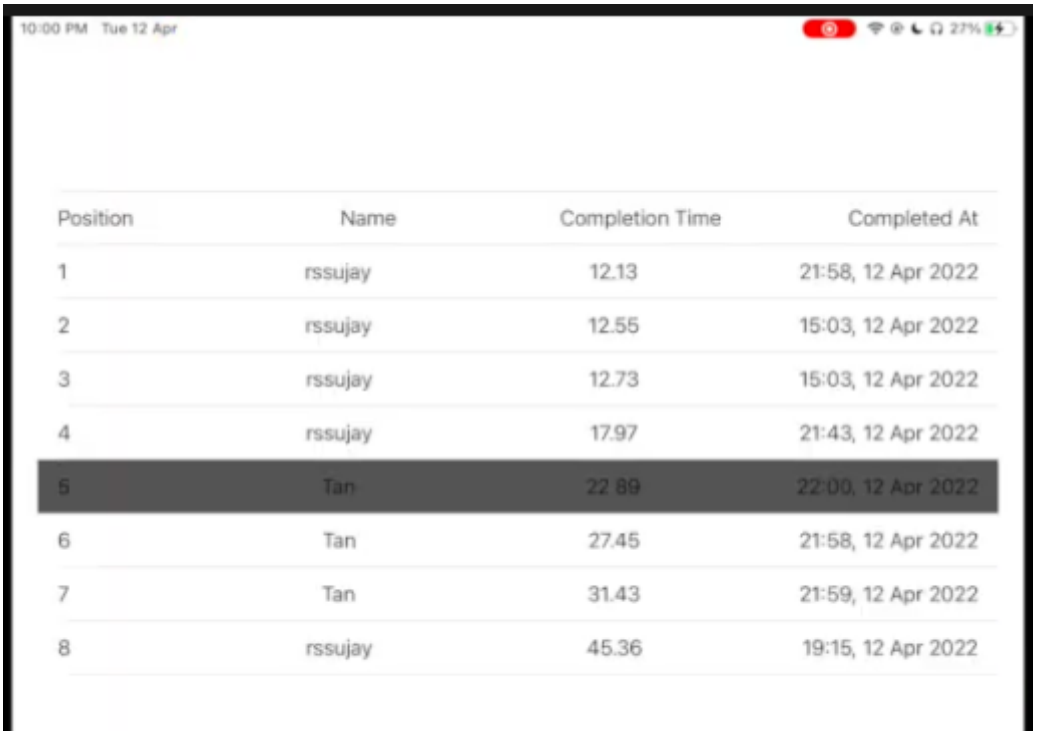
Power-ups

The following power ups are available during gameplay, when playing game modes that support power up spawn.

Power Up	Effect
Freeze	Freeze the movement of a player. Affected players are not able to move or jump for 5 seconds
Confuse	Confuse the affected players. A player who intends to go in one direction will instead go in the opposite direction.
Teleport	A player who activates this power up can teleport to anywhere, depending on the tap location.
Slowmo	Slow down the movement of affected players. Affected players will move/jump with ⅓ of the normal speed/impulse.
Blackout	All players except the activator will see a black screen for 5 seconds.

Highscores

After finishing a game, the highscores page appears. In light mode, it looks similar to what is shown below.



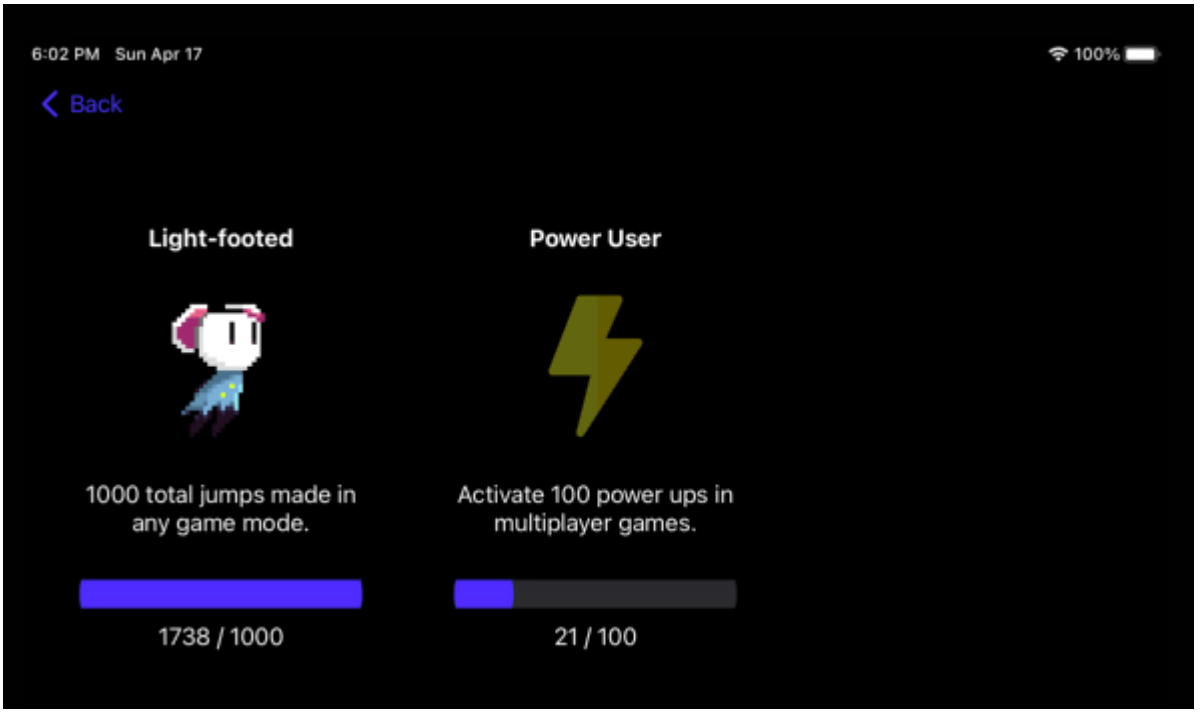
For each game mode, the highscores will look a little different.

After a Time Trial game, you will be able to observe your ranking across all players, for the seed you've just played, within the highscores table.

For a Race To Top game, you may end earlier than other players. As long as you stay on the highscores page, you will get **live updates** on other players' positions as they finish.

For a King of the Hill game, you will get to see everyone's positions almost immediately, as everyone will end their game at the same time.

Achievements



The achievements page tracks your account's progress through games. Achievements you have unlocked appear with a full progress bar and the image fully visible, while achievements yet to be unlocked appear with a faded image and a progress bar indicating how close you are.

You can press the back button to exit the achievements menu.

That's all! Hope you have countless hours of fun playing Cloud Jumpers!

Post-Sprint 2 Specifications

Feature	Status
King of the Hill game mode	Completed
Shadow player	Completed
Moving clouds	Completed
Achievements	Completed
SpriteKit facade	Completed
Indicators for current players in highscore	Completed
Indicators for current players in scene	Completed
Variable lobby names	Completed
Slowmo power-up	Completed
Teleport power-up	Completed
Blackout power-up	Completed

Software Design

Cloud Jumpers adopts two architectures: **Model-View-Controller (MVC)** for UI such as lobbies, authentication, log in, and sign up pages, and **Entity-Component-System (ECS)** for game-related models, lifecycle, rendering, and user input. It uses SpriteKit as its game engine for physics simulation, rendering, and visual effects.

Core Structure

The figure on the following page outlines the relationships between constructs in the game-related models.

> Note that in this documentation, the term "game engine" refers to a module that provides the simulation of core game logic, such as rendering and physics. Cloud Jumpers abstracts these functionalities to a game engine, i.e., SpriteKit, but the constructs have been designed to be modular and applicable to other engines, e.g., SceneKit or RealityKit.

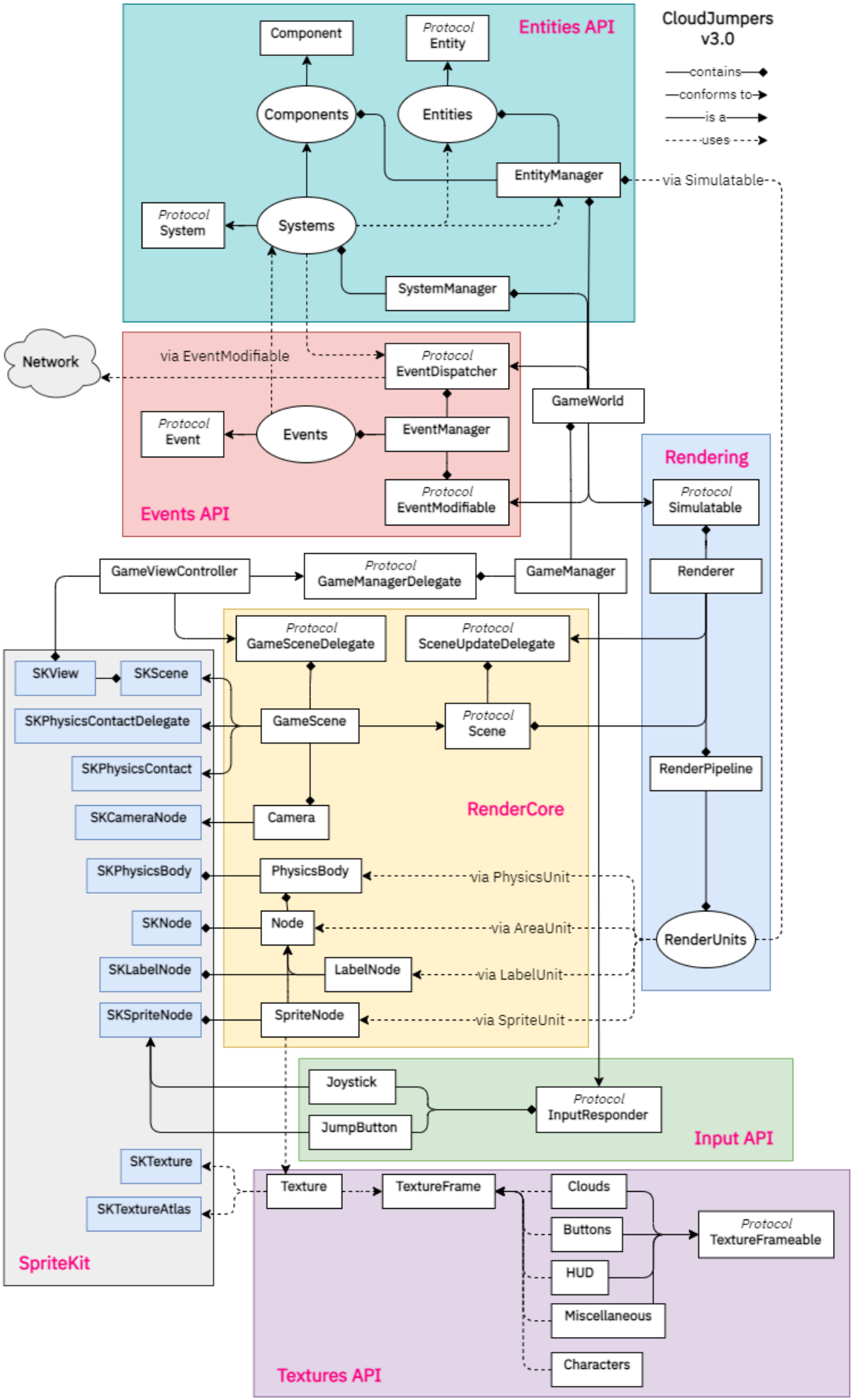
Entities API

This module contains the raw models represented in the game world. It adopts a blend of Adam Martin's ECS and Bilas/GameplayKit's ECS, with the core definition from the former and the OOP adaptability from the latter. This blend means that entities still essentially are methodless and attributeless, but have their own constructs (e.g., `Player`, `Cloud`, etc.) with only one method: `setUpAndAdd(to:)` which contains the logic necessary to initialise the relevant components and add them *to EntityManager*, NOT the entity itself. This method is invoked by EntityManager when an entity is added to it. Following Adam Martin's ECS *purely* would have to define a method for every entity to map an integer ID to an array of components. With our blend of Martin's and Bilas/GameplayKit's ECS, entities are:

- still inherently 'dumb,' i.e., methodless and attributeless (Martin's),
- compartmentalised in their own constructs (Bilas/GameplayKit's), and
- adhering to the open-closed principle (as a corollary).

Therefore, we define our Entities API specifications as follows:

- **Entity** is a class that contains a unique ID and represents anything that appears in the game world, e.g., player, clouds, platform, power-ups, etc.
- **Component** is a class that contains raw data for *strictly one behaviour* that *any entities* can possess, e.g., sprite, physics, camera-anchored, etc.
- **System** is a class that modifies components and adds/removes entities, with(out) accessing data from components of other types.


CloudJumpers
v3.0
contains
conforms to
is a
uses
Entities API
Component
Protocol
Entity
Components
Entities
EntityManager
via Simulatable
Protocol
System
Systems
SystemManager
Network
via EventModifiable
Protocol
EventDispatcher
GameWorld
Protocol
Event
Events
EventManager
Protocol
EventModifiable
Events API
Rendering
Protocol
Simulatable
Renderer
GameViewController
Protocol
GameManagerDelegate
GameManager
Protocol
GameSceneDelegate
Protocol
SceneUpdateDelegate
SKView
SKScene
SKPhysicsContactDelegate
GameScene
Protocol
Scene
SKPhysicsContact
RenderPipeline
SKCameraNode
Camera
RenderCore
SKPhysicsBody
PhysicsBody
via PhysicsUnit
SKNode
Node
via AreaUnit
RenderUnits
SKLabelNode
LabelNode
via LabelUnit
SKSpriteNode
SpriteNode
via SpriteUnit
Joystick
JumpButton
Protocol
InputResponder
Input API
SKTexture
SKTextureAtlas
Texture
TextureFrame
Clouds
Buttons
HUD
Miscellaneous
Characters
Protocol
TextureFrameable
SpriteKit
Textures API

There are two entry points to the Entities API: EntityManager and SystemManager, both of which are managed by GameWorld. EntityManager is used purely to store entities and components, as well as retrieving and removing them. SystemManager stores an array of Systems and extends the ability to update all systems synchronously and access systems to forward external modifiers, e.g., events, to entities and components. This separation allows EntityManager to be a single source of truth of game models, while SystemManager focuses on registering and updating game systems.

On modifications of entities and components

Entities and components are only modifiable by systems. Therefore, any changes to them can only be done by systems that handle the logic for modification. This ensures that the core representations of the game-related models are kept safe by a single authority to ensure there are no mis-synchronisations due to random willy-nilly access—especially important considering they are all reference types. External modifiers can modify entities and components by using methods extended by relevant systems accessible via SystemManager.

Runtime Structure

GameWorld

GameWorld is the class that unites Entities API, Events API, and Rendering. It is designed to be the assembly point for modules that need to access the game models and carry out complex logic, e.g., dispatching events to the network, rendering, accumulating metrics for achievements, etc. These modules are designed to be modular and communicate with GameWorld via their own protocols (think delegates), to which GameWorld can conform.

This separation allows modules to stay modular, i.e., if Events API were to be uninstalled, or be changed with a different data structure, GameWorld, Entities API, and other functionalities will still be intact and maintaining their behaviours. Most importantly, it also allows *these modules to only get access to what they really need without giving full access to EntityManager*, thus ensuring entities and components stay unmodified.

For example, events will need access to EntityManager to be able to add/remove entities. Although systems can add/remove entities, they are not specified to do that, and so it will be awkward to have some system extending a method just to add/remove entities. Note that systems are first designed to *update components*. Instead of giving access to EntityManager, we make events communicate with a target protocol EventModifiable that extends only the functionalities to (1) add/remove entities and (2) access systems.

GameWorld also acts as a barrier between Entities API and these modules. This ensures accesses to the Entities API needs to be concretely defined on a *need-to-access basis*, rather than just the developer's 'intuition' to not directly access Entities API externally.

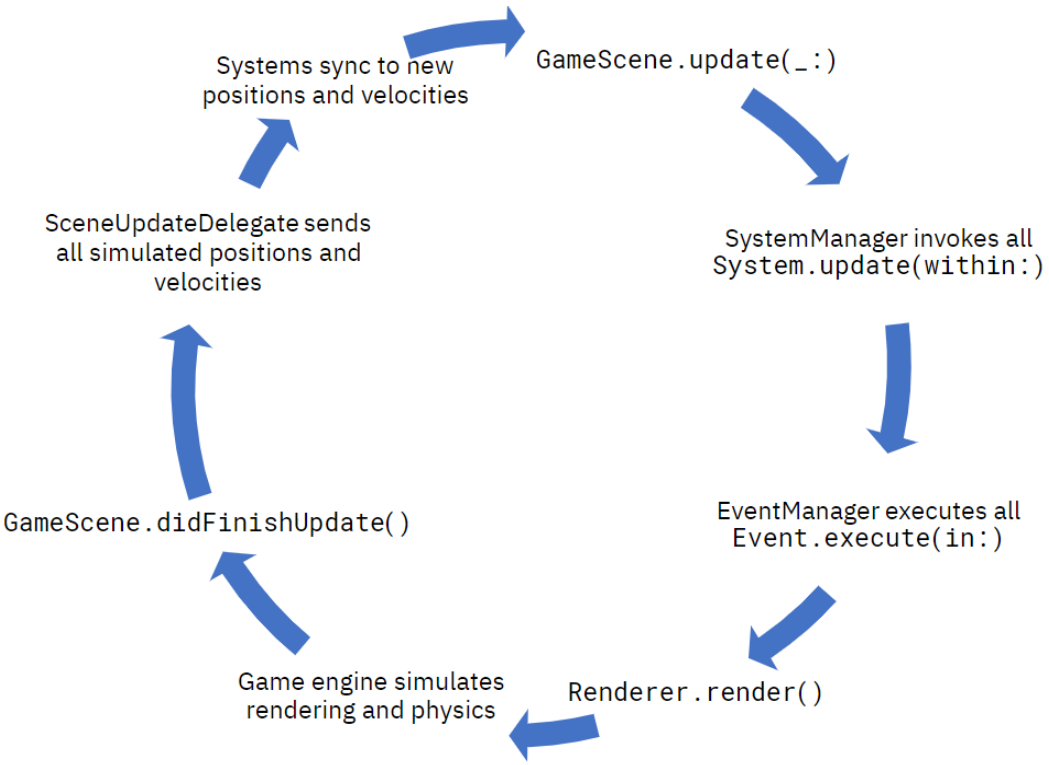
Modules attached to GameWorld and their access levels. GameWorld conforms to the protocols that these modules specify.

Access level	**EventManager**	**Renderer**	**GameRules**	**Achievements**
via	EventModifiable EventDispatcher	Simulatable	RuleModifiable	MetricsProvider
Add entity	Yes		Yes	
Remove entity	Yes			
Get entity from ID		Yes		
Access system	Yes			Yes
Activate system			Yes	
Add event	Yes		Yes	
Add effector				
Dispatch event			Yes	
Add component			Yes	
Check component		Yes	Yes	
Access component		Yes	Yes	

Game update cycle

Our game abstracts the game engine to SpriteKit. The update cycle is outlined as follows:

1. `GameScene.update` → `GameSceneDelegate.update` → `GameManager.update` → `GameWorld.update`
2. SystemManager updates all registered systems and calls their `update(within:)`
3. EventManager executes all enqueued events. Note that EventManager is resolved after SystemManager to ensure that any ad-hoc modifications (by events) are not overwritten by autonomous updates (by systems).
4. Renderer obtains entities from EntityManager via Simulatable and creates, updates, and synchronises entities/components to their respective nodes in the GameScene.
5. `GameScene.didFinishUpdate` → `SceneUpdateDelegate.didFinishUpdate`
6. Renderer forwards the new positions and velocities of all nodes to GameWorld
7. GameWorld invokes PositionSystem's and PhysicsSystem's sync method with the new positions and velocities



Overview of the game update lifecycle from Entities API to GameScene

Systems and events separation

Systems cannot enqueue events, both by design and structure. This restriction was made to prevent infinite modification cycles from happening, e.g. PositionSystem enqueues MoveEvent that calls PositionSystems that enqueues MoveEvent. It was also made since:

- events are meant to signal modifications to entities/components, and systems already have direct access to them, and
- systems are meant to solely update entities/components.

Giving access to events meant that systems can make use of the advanced Events API features, making lifecycle possibly more complex.

Renderer

The renderer is responsible for synchronising entities (models) and nodes (views). With the renderer rendering at every update cycle, it acts as a publisher to ensure entities are translated to views. This allows modifications to the game world to simply happen in the models and the views can stay as ‘dumb’ representations of them.

Cases for rendering synchronisation actions for each entity at every update cycle

Entity	Node	Description	Renderer action
Exists	Does not exist	Entity has not been rendered	Create
Does not exist	Exists	Entity has been removed	Prune
Exists	Exists	Entity has been rendered	Update

Create action will create a node from the entity, depending if it needs a SpriteNode (with SpriteComponent), LabelNode (with LabelComponent), or just a generic Node. Once the node is created, Renderer will cache the node. **Update action** will read from relevant components of an entity and update the attributes of the associated node.

The Renderer runs in *only one linear iteration* for all entities, and in doing so, cross-checks the current entities from EntityManager with its own cache. Matching entities are 'whitelisted,' and at the end of rendering, those that are no longer matching (e.g., does not exist in EntityManager but exists in cache) will be pruned. **Pruning** removes the nodes from the view and uncaches them.

These update actions, pruning, and the one-iteration features are important to ensure that rendering is done as efficiently and fast as possible as it is the last phase of visual feedback.

RenderPipeline and RenderUnits

There can be a plethora of components and the renderer will have to keep up with these very specific ways to create/update them. To accommodate these complexity, Renderer abstracts the specifics of create and update to RenderPipeline.

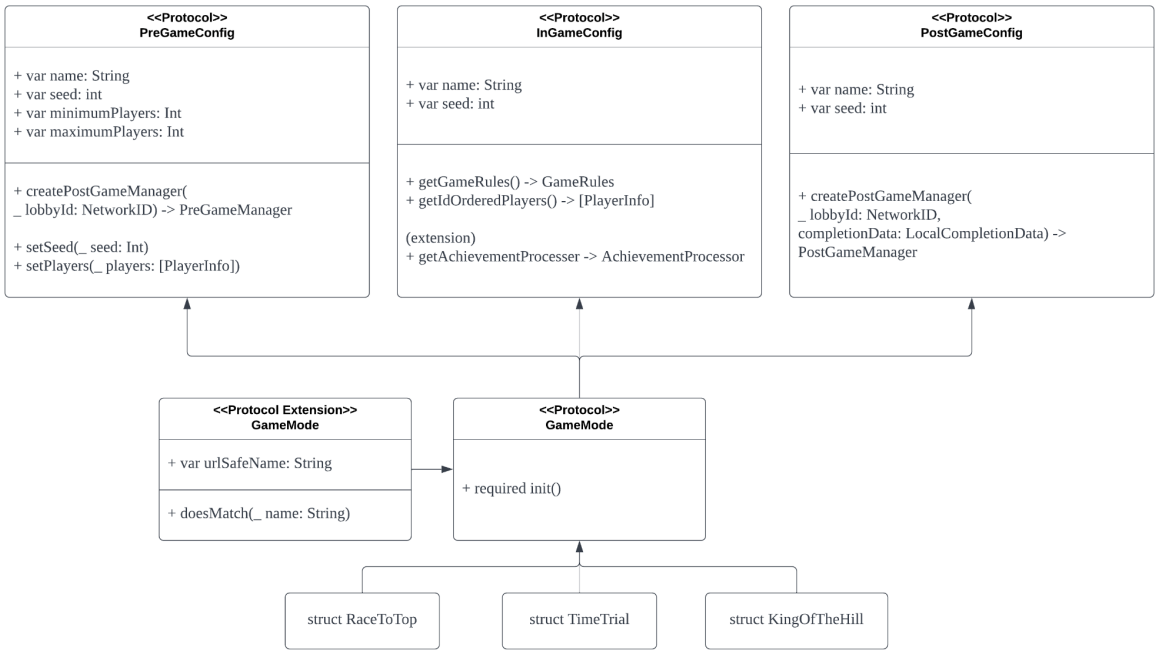
RenderPipeline is essentially similar to SystemManager. It manages an array of RenderUnits, e.g., SpriteUnit, PhysicsUnit, etc., that each holds the specifics of updating a node based on a particular component. A RenderUnit can both create and update a node. Renderer passes an entity to a continuous pipeline of RenderUnits, thereby allowing the respective node to be updated visually.

This concept of RenderUnits allows the rendering logic for specific components to be extensible and modular, i.e., adhering to the open-closed principle. RenderUnits can be detached, attached (in RenderPipeline) and created as needed. Since they run in a pipeline and accepts/rejects only certain conditions for an entity (e.g., SpriteUnit rejects entities without a SpriteComponent), this design allows dynamic rendering to happen without typecasting and if/switch clauses.

Game Lifecycle

A game of Cloud Jumpers varies based on configuration options - game seed, selected game mode as well as a number of runtime-determined factors such as number of players and highscores state. A good architecture is required to allow for extensibility for new game modes to be introduced, with minimal changes to existing game mode specifications.

The GameMode class acts as a complete specification and single source of truth for how a game evolves.



As shown above, partial interfaces allow selectively exposing specific methods, based on the lifecycle stage the device is currently in. The GameMode object is accessed through the appropriate interface, based on the containing class.

In order to facilitate creation, as well as enumeration of all game modes, a modification of the simple Factory pattern is used, together with a visitor-pattern inspired approach of shifting the responsibility to classes in identifying if they should be instantiated.

In the lobby, the GameMode's configuration methods are exposed via the PreGameConfig protocol for changing of parameters, such as the game seed.

In the game, the getter attributes are exposed through the InGameConfig protocol for the game to work with.

After the game has concluded, the PostGameConfig protocol exposes attributes for post game actions, such as submission to highscores.

Pre-& Post-Game Managers

The GameMode exposes methods that create Pre and PostGame managers. The PreGame Managers perform one or more of the following tasks:

- Async network calls
- Preparing game event publishers and subscribers

The PostGame managers perform the following tasks:

- Highscores data submission
- Highscores endpoint polling, to keep the data live and updated
- Preparing data for the Highscores ranking table shown to the user

GameRules

Game Rules represents the set of unique rules of each game mode. It is a special system-hybrid created by the GameMode and passed into the Game Manager. The Game Rules construct responsible for the followings:

- Set up and update mode's specific display labels, e.g. a count-up timer for Time Trial, a count-down timer for King of The Hill
- Decide on modes' specific events: Game Rules have access to the run-time components of the local devices, and is updated intervally (Like a normal Game System)
- Decide end condition based on the current game mode, which will be checked by the Game Manager
- Propagated the local statistics of interest to the Game ViewController via a delegate, to be used by PostGameManager

GameRules have no exposed method for other asynchronous run-time structures to access it, and only perform closed, internal updates.

To facilitate GameRules interaction with the GameWorld, the RuleModifiable delegate is created , which allows GameRules to create/update entities and components as a normal ECS's system.

However, RulesModifiable also allows GameRules to access the Metric System (explained more into detail down below), to gather the statistics of interest. Furthermore, RuleModifiable also allows GameRules to fire local events, which is something that a pure system is explicitly forbidden to do.

Cross-Device Synchronised Start

For multiplayer game modes, it is important that players start together at approximately the same time. As the game is played over the internet it is likely that users will have devices of varying latencies. Additionally, having no synchronised start mechanism gives an unfair advantage to the player who pressed ready last, as their device will be aware of the game start condition prior to other devices.

A LobbySynchronizer module is introduced to address this issue, and works as follows.

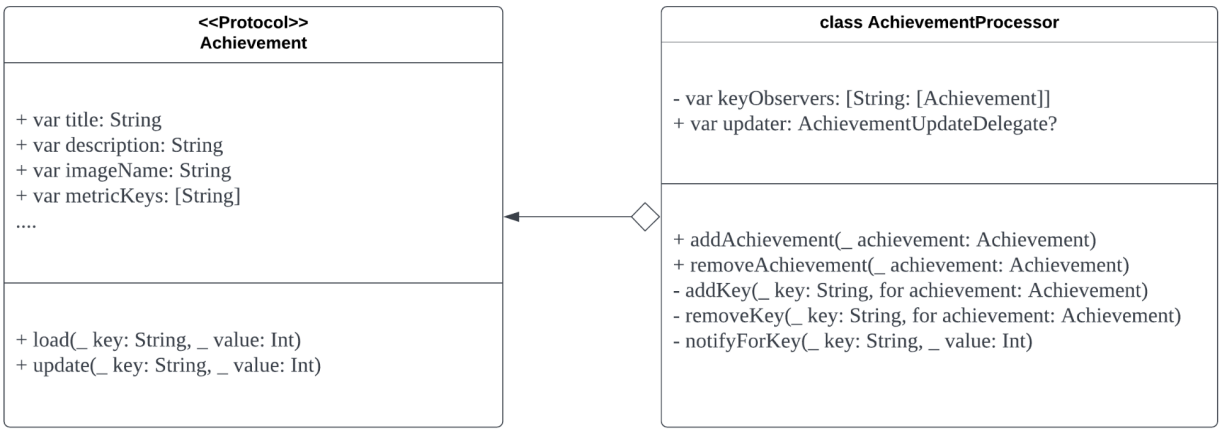
- Each device is informed of a player's last update time (as a timestamp), as registered by the database

- Each device independently sets a start time, using a scheduledTimer, to begin the game at (timestamp + X seconds).

Save for factors such as clock drift, scheduling differences, and disconnections, games start at approximately the same time.

Metrics Provision

As part of the Achievements feature, there exists a need for Achievements to be fed with information from numerous sources on demand, one of which is ingame counters.



However, Achievements should not be aware of or coupled to implementations of sources, or any modifiers along the way. Additionally, they should be notified of changes, possibly without knowing where the information originates from.

This makes the use of the Observer pattern suitable. In our implementation, the Achievements themselves specify keys they are interested in getting notified for, handled by the Achievement Processor's addAchievement method. This allows for independent development of achievements, as well as suppliers of values and keys to the Achievement system.

Lobbies

A lobby plays multiple important roles in a player's gameplay, through facilitating:

- Grouping and management of players
- Dynamic host changes (e.g. failover when player disconnects or exits)
- Synchronised fair start for games
- Lifecycle management

It delegates responsibilities to subordinate classes, a ListenerDelegate, which is designed to listen and call back on receiving updates, as well as an UpdateDelegate, which generates lobby updates such as player-driven actions: (join, leave) and (ready, not ready) and call back.

Dynamic host change

For certain gameplay elements, we constrain only a single player at a point in time to be able to trigger certain events. Players can disconnect, finish, and a mechanism is required for all devices to agree on who is the player allowed to fire events during runtime.

When the current host leaves, either through use of the Game UI or through external triggers e.g. power off, Firebase detects that the device has gone down and performs a database update on its behalf. All other devices generate a deterministic ordering of hosts independently, and try to apply the change via transactions. Once successful, all devices will receive an update on the new hostId.

Module Structure

RenderCore: The game engine facade

In this game, we used SpriteKit extensively, and in prior sprint reports, we have highlighted the main issue with using SpriteKit: **it tries to combine models, views, and controllers together**. It was designed to be a game engine and the whole game software itself, when our project requires only the game engine functionalities of it. As such, we created a facade over it and we call it RenderCore.

Notably, there were different ways to facade SpriteKit in our code. From the class relationships diagram, for example, we see that:

- GameScene ***subclasses*** SKScene, because it has to override the touchesBegan, touchesMoved, touchesEnded, update, and other methods from SKScene.
- Node ***composes*** SKNode and proxies some methods and attributes from SKNode.
- Texture ***creates*** *(uses)* SKTexture so that SpriteNode can send it to SKSpriteNode.

The principle behind our facading, regardless of the ways, was simple: **if we decide to do away with SpriteKit, are the codes *behind* this construct affected?** For example, if GameScene is no longer an SKScene, the rest of the game-related logic will be fine because they all interface with the Scene protocol. The composition-based facades will also be extremely easy to change because one simply needs to change the typealias `*Core` and change the constructors.

Simply put: if we decide to change our game engine, only the methods within RenderCore will need to be modified extensively to adapt to the new game engine.

ContactHandler and SeparationHandler

Note that these are not collision resolutions. They are handled by the game engine. These are modules enabling the *handling* if such collision occurs, i.e., what to do. For example, if a player collides with the top platform, it will cause the game to end.

What we require from the game engine is to let us know when two nodes **contacts** and **separates**. In SKPhysicsContactDelegate, these are fired by the `didBegin(contact:)` and `didEnd(contact:)` methods. Of course, these all are facaded :]

> We define collision as just a general term adopted from SpriteKit. In our system, we define contact as when collision begins, and separation as when collision ends.

To identify which types of entities are colliding, we employ double dispatch, or otherwise mostly referred to as the visitor pattern. Entities conform to the Collidable protocol that extends the methods to signal `collides(with:)` and `separates(from:)`.

The definitions of these methods are provided in *extensions* of the related entities to prevent spamming entities. They all proxy their calls to ContactHandler and SeparationHandler to fire events resulting from contacts/separations.

Double dispatch allows us to determine which method to run depending on the two concrete entity types without the use of if/switch clauses, avoiding cyclomatic complexity. Moreover, it also allows for type safety, since the method resolution is based on concrete types. For these reasons, we chose double dispatch over if/switch clauses and string type-method maps. Although, it does make it complicated when developing as we needed to define all combinations of entity types in ContactHandler and SeparationHandler.

Input API

The Input API forwards the UITouch handlers from GameScene to an InputResponder, to which GameManager can conform. Input API is not a part of the Entities API, and the joystick and jump button are not rendered by Renderer. Moreover, they are subclasses of SKSpriteNode. We decided to make inputs separate from the game-related models as they are a way for the user to modify the game world, not part of the game world itself.

This separation results in the Entities API handling the lifecycle of in-game objects (no outside inputs) and Input API handling the inputs from the user and forwards it to GameManager to be processed. This way, the Entities API need not reach outside to View and stay in Model.

> Note that even though the game-related models are not adopting MVC, we still try to separate modules where possible to ensure modularity. We always ask the question: ***for this abstraction layer, can the game still run without it?*** If yes, then we will maintain the genericity of that module.

This design also allows Joystick, JumpButton or any other inputs in the future to simply capture their own touches. The new Input API allows touchable objects to capture their own touches (per-touch method), resulting in better performance (no need for O(n) linear loop for all touches) and input resolution. These two schools of thought for controlling user interaction were discussed in Apple's SKNode documentation.

Levels API: Procedural content generation

To allow for dynamic generation of levels, the Levels API enables seed-based content generation. There are generators created to generate contents of different aspects of the game. For example, LevelGenerator takes a Blueprint struct containing:

- the dimension of the world,
- the dimension of the content to generate, e.g., cloud, power-up nodes, etc,
- ranges of tolerance, i.e., distance between contents, and
- the integer seed used for random number generation (e.g., 69420),

and returns an array of CGPoints which the client can use to generate their concrete contents, e.g., cloud entity, power-up entity, bomb drop locations, etc. It generates content with a time complexity of O(n) where n is the number of CGPoints (hence, content nodes) generated. In a similar fashion, RandomSpawnGenerator generates a random position and velocity, based on seed and generation info (similar to Blueprint), to spawn disasters.

The Levels API is:

- **deterministic**, as the same seed and blueprint will generate the same content.
- **dimension-agnostic**, as the sizing data is abstracted out with Blueprint,
- **realistic**, as it will ensure that the content generated is playable, e.g., no clouds are unreachable, and visually comprehensible,
- **responsive**, as one can scale the content by changing the blueprint but keeping the same seed; that was why they were kept separate.

The Levels API was developed to provide procedural content generation. This allows for challenging and random-yet-deterministic levels for the players without having to develop

and store specific sets of levels which is a huge boon for development time and app storage. To a larger extent, similar to Minecraft, players can make their own world by providing different seeds and even share them, opening doors for community. In our game, players can also modify the seeds in a lobby before playing, allowing them to challenge themselves with new random levels.

Events API

The Events API is responsible for enabling external modifications to entities and components via system. Its entry point is EventManager which stores all fired events and executes them synchronously based on rank and timestamp on every update cycle. EventManager organises events in a PriorityQueue, and an event's priority is its rank (defined in an enum) and timestamp.

Events follow the command pattern to provide fine-grained modification logic to entities and components via systems. This design pattern allows the modification logic to be compartmentalised in each event's definition, adhering to the open-closed principle, instead of being chucked in functions only to be called in a switch clause, which is not extensible and may invoke cyclomatic complexity—too many decisions made in a function.

EventManager communicates with GameWorld to access entities, components, and systems via EventModifiable, and dispatches remote events via EventDispatcher protocols.

Network events synchronisation

Some events can be resolved locally, e.g., input, but some may need to be dispatched to other devices, e.g., inventory update, positioning, spawning, etc. For these events, remote events are dispatched (not executed) by EventManager to other devices, which will result in the creation of actual events in other devices' EventManager queue. To power this feature, EventManager follows the subscriber/publisher pattern exposed by the network module.

EventManager supports many advanced lifecycle features, such as deferred events, events chaining, event effectors, and events piggybacking.

Deferred events

This feature allows an event to determine when it can be executed by EventManager, and if not yet, its execution is deferred by being re-enqueued for execution in the next update tick. This feature is useful, for example, to remove an entity after some time or leaving GameScene. It is also powerful because the condition for an event's execution can be modular and wide-ranging (not just by time), rather the state of Entities API via systems.

Events chaining

This feature allows events to be chained like Java Stream API. For example,

```
MoveEvent(on: entity, by: displacement)
```

```
    .then(do: JumpEvent(on: entity))
    .then(do: AnimateEvent(on: entity, to: .jumping))
    .then(do: LogEvent("I made it through!"))
```

will move an entity, initiate a jump, switch its animation, and log a message. This way, events that are meant to be executed consecutively can happen regardless of priority, deferment, or other events enqueued in EventManager. Chained events are composed into a BiEvent (think Java's BiFunction).

Event effectors

This feature allows modifications to existing events in the EventManager queue. Effectors essentially transform events, i.e., takes in an event and returns a modified version of that event. Similar to events, effectors apply the command pattern to provide modification and invalidation logic. EventManager clears out invalidated effectors and passes each event through a pipeline of effectors (if any) before executing them. This way, several events can be intercepted, resulting in dynamic behaviours. This feature is extremely useful for power-ups, for example:

- Freeze: NullMoveEffector that takes a MoveEvent and returns a MoveEvent with a zero displacement—essentially no movement.
- Confuse: SwapMoveEffector that takes a MoveEvent and returns a MoveEvent with negative x-displacement—essentially swapping left and right movements.
- Shield: NullRespawnEffector that takes a RespawnEvent and returns a no-op RespawnEvent—essentially shielding the player from being attacked.

And since effectors are applied in series of pipelines, effectors can stack onto each other, resulting in a combined effect, e.g., confuse and shield combined.

Events piggybacking

This feature allows events to supply new events and effectors to the EventManager after its execution. `Event.execute(in:thenSuppliesInto)` passes in an inout Supplier to which events can supply their events, remote events, and effectors. The supplied events will be executed, remote events will be dispatched, and effectors will be applied, all within the same update tick. This feature allows for the creation of contextual events and effectors, i.e., created based on the changes made during the execution of an event. For example, an ActivatePowerUpEvent might need to fire a RemoveEntityEvent to remove the power-up within a certain time after activation.

These advanced lifecycle features were developed to make event generation and execution as flexible and modular as possible to support various modifications to the Entities API in various ways. This is because there are many constructs who will be able to change entities and components, such as the Input API, network modules, power-ups, and disasters.

Especially with power-ups, their effects can be multi-faceted and require access to different parts of the code, e.g.,

- Freeze needs to “hijack” MoveEvent and JumpEvent to fire a zero displacement.
- Confuse needs to “hijack” MoveEvent to fire the negative x-displacement.
- Rocket needs to add impulse in the PhysicsComponent.

Essentially, these features provide multiple ways of modifying the Entities API without providing direct modifications access. This way, the Entities API, Events API, and their clients can stay separated and modular.

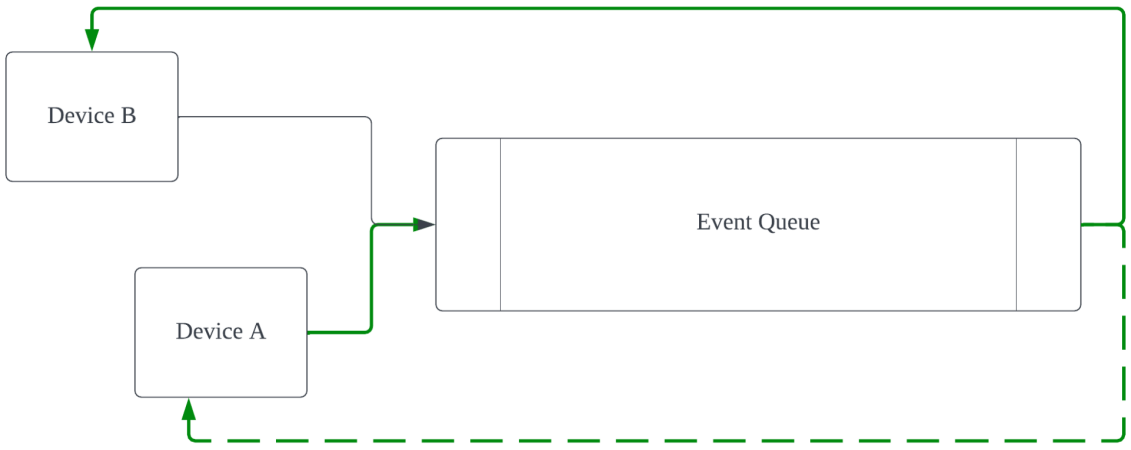
Multiplayer Event Queue Architecture

Supporting multiplayer modes for a real-time game involves accounting for multiple constraints such as latency, reliability, and extensibility.

The event queue architecture acts as a backbone to the multiplayer capabilities of the game, providing the following characteristics that were important for our game:

- N writers and readers
- Support for both regular and unpredictable events in an async but timely manner
- Single source of truth on chronological ordering of events (enqueue timestamp)
- Fairness in performance and latency, as well as lack of reliance on any single device

The event queue is implemented as a FIFO queue on top of Firebase Realtime database. This allows for the establishment of an ordering of game events, as well as a single source of truth that does not pose a significant advantage to any player.

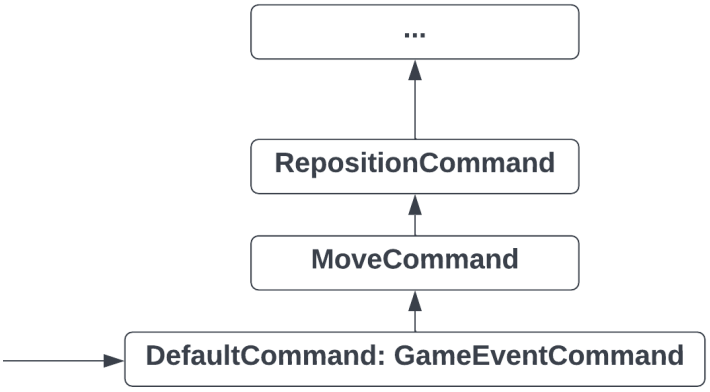


Publishing devices can opt in to receive events for which they are the source on dequeue. For example, if two players are picking up a powerup, each of their intent to pick up a power up is enqueued, and both devices receive the sequence of enqueues in the same

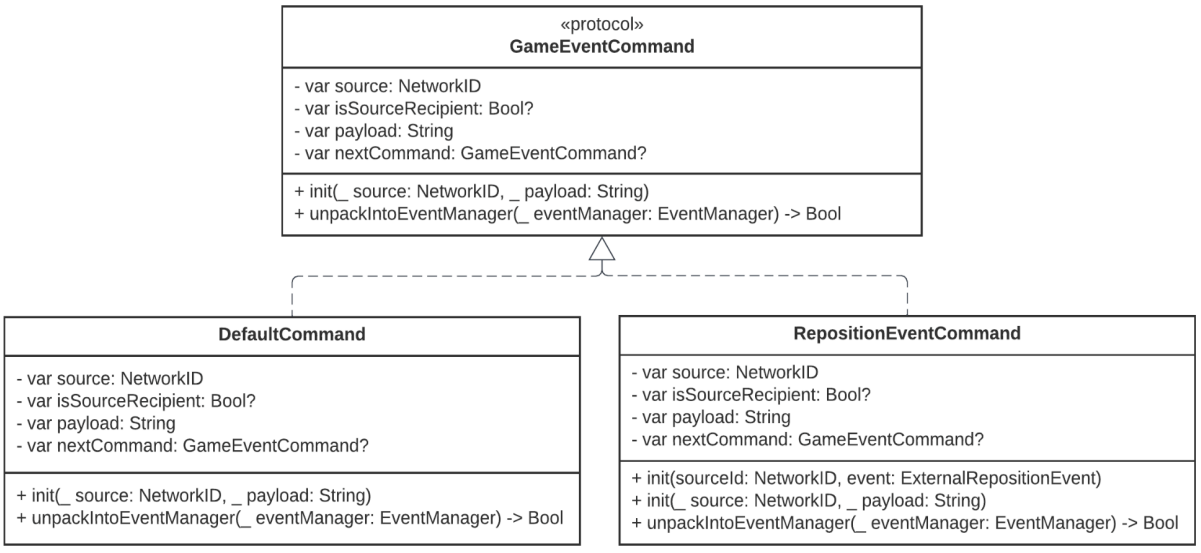
order. Each device then accepts that the powerup belongs to the device that was registered as having enqueued earlier.

Given the nature of the game, there exists a need to support arbitrary game events, which are handled in different ways by the game event manager. The command pattern was used together with the chain-of-responsibility pattern in order to separate the event queue functions from game logic.

Commands wrap RemoteEvents, which are serializable, and their attributes allow them to signal the type of event they should be unpacked into. As object references are not transferred across devices, a chain-of-responsibility pattern is employed on the receiver's side to identify and unpack the command payload into the corresponding event.



The chain of responsibility pattern is used to handle events of varying complexity under a generic network interface



Class diagram for GameEventCommand subtypes, which carry RemoteEvents over the network and convert them into Events at the destination

Each command has two initializers - the first is used to convert underlying RemoteEvents into a payload string. The second constructor is used to recover a command on the receiver's end.

Each underlying game event is required to conform to a Protocol, RemoteEvent, which conforms to Codable and provides an extension function, toJsonString(), that converts data into JSON.

The use of commands as an abstraction layer allows for arbitrary handling of different types of events, without needing the network layer to be aware of what it is transporting.

Furthermore, it remains extensible for purposes other than game events. For example, to support non-game events, we can add a horizontal sibling to GameEventCommand which conforms to a similar protocol, using a general EventManager.

Alternatives that were considered include making of use message queue solutions, such as RabbitMQ or ZeroMQ. The interfaces to Firebase have been delegated to allow for easy substitution of event queue backends. This is particularly important during development, where performance expectations may not be met by a chosen backend.

Extension of Event Queue Architecture: Shadow Player

In our final sprint, an additional feature requested for Cloud Jumpers was a Shadow Player to be included in the single-player Time Trial mode.

At present, players send positional updates outwards every 6 game ticks (~16.67 ms) in a broadcast manner to the event queue, which is then read by all other devices, and processed through updating of their own sprites.

This functionality is enabled for all game modes, including Time Trial. By virtue of having an event-queue based architecture, we are able to naturally extend and replay games of stored chronological game events.

In our implementation, this is supported by a Firebase Emulator, which pulls all commands from a past game prior to the start of the game, and replays them in a Time Trial player's live game using a scheduledTimer that is dynamically set to callback based on timestamp deltas in between the commands.

Authentication

Authentication in our application is backed by a Firebase Auth implementation, which uses a singleton to provide a single source of truth of the user's current authentication status. It is made available through an AuthService.

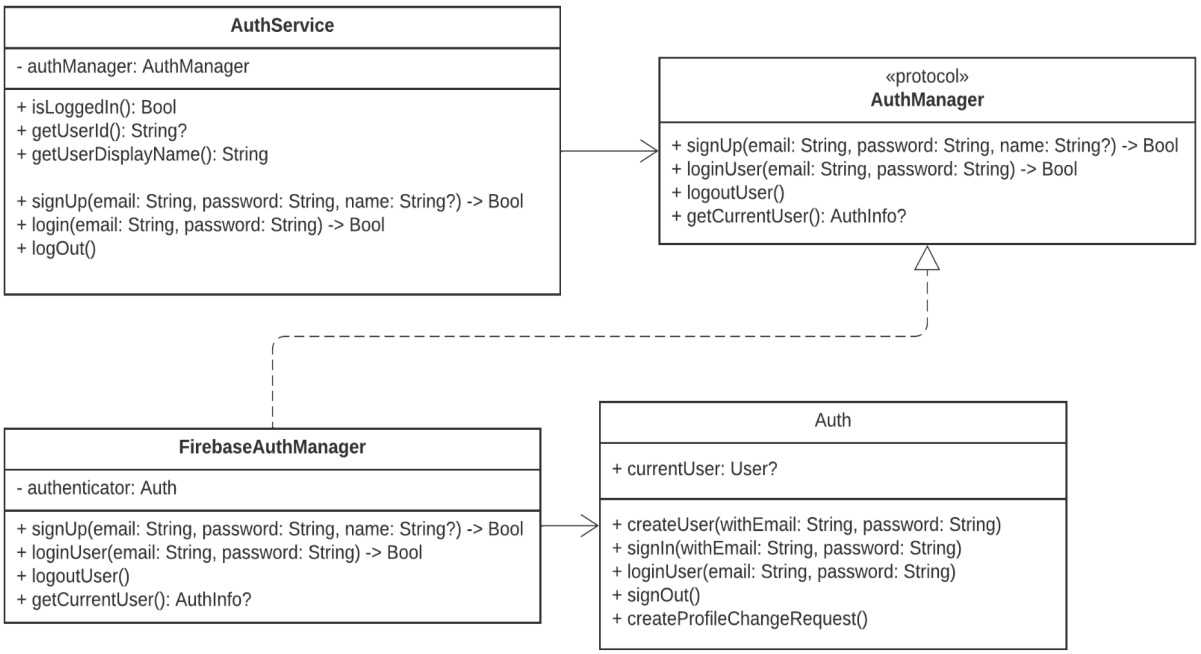


Figure: Authentication functionality provided by a client AuthService, which is supported via the adapter pattern

With this design, we will be able to better support swapping out authentication frameworks in the future if needed.

Customised Rankings

After a game, player positions are displayed to the user as a table, and the columns displayed to the user depend on the game mode. For example, a king of the hill mode might make use of a points-based scoring system, while a classic race game mode mostly makes use of time taken to determine positions.

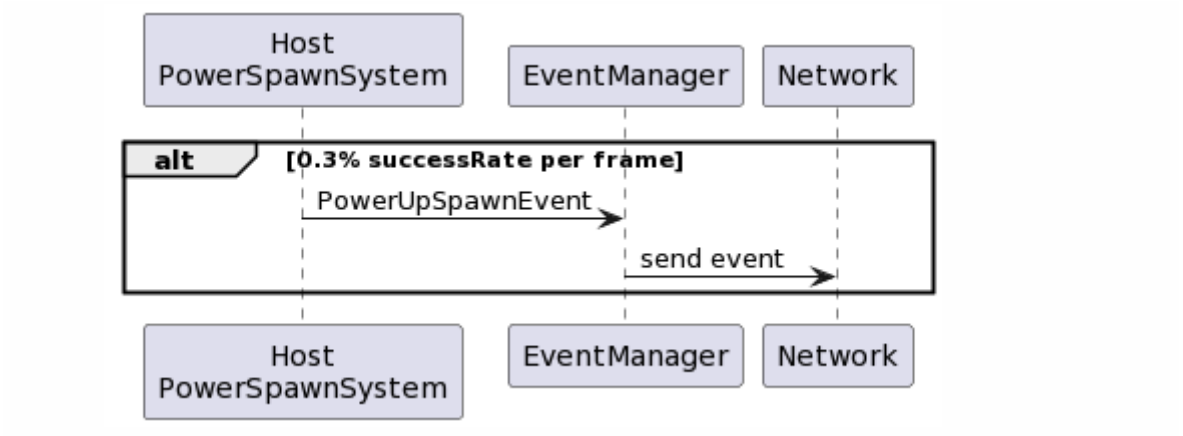
To support a variety of game modes, support for a generic table that supports per-row key-value pairs, which map column names to column value, has been implemented. The rankings are dynamically updated as more people complete the game mode, and this is facilitated through subscription via a delegate that makes network requests.

The current implementation makes use of periodic polling via a REST delegate that performs GET and POST requests periodically to a remote backend server.

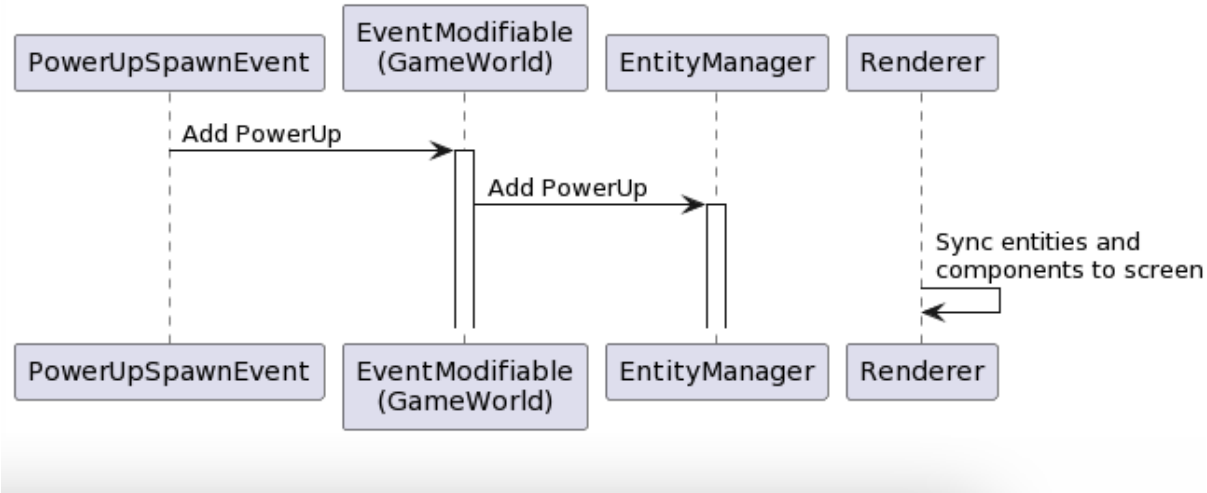
Viable protocol alternatives that could be swapped for through a new delegate would be gRPC, websockets, or other push-friendly protocols, however this was decided to be unnecessary for periodically refreshing a ranking or highscores page.

Power-ups

Power-up creation

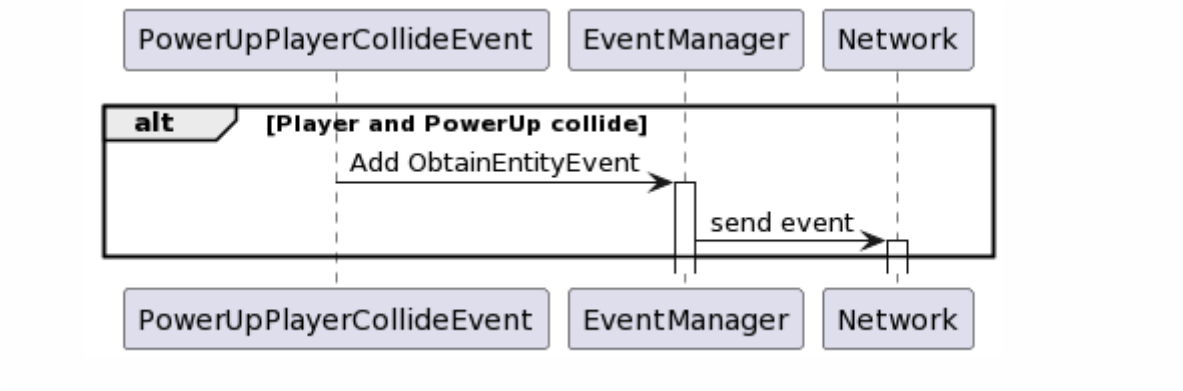


A host player will send a power-up creation Event to the network with a probability of 0.3% per frame, anywhere randomly on the screen.

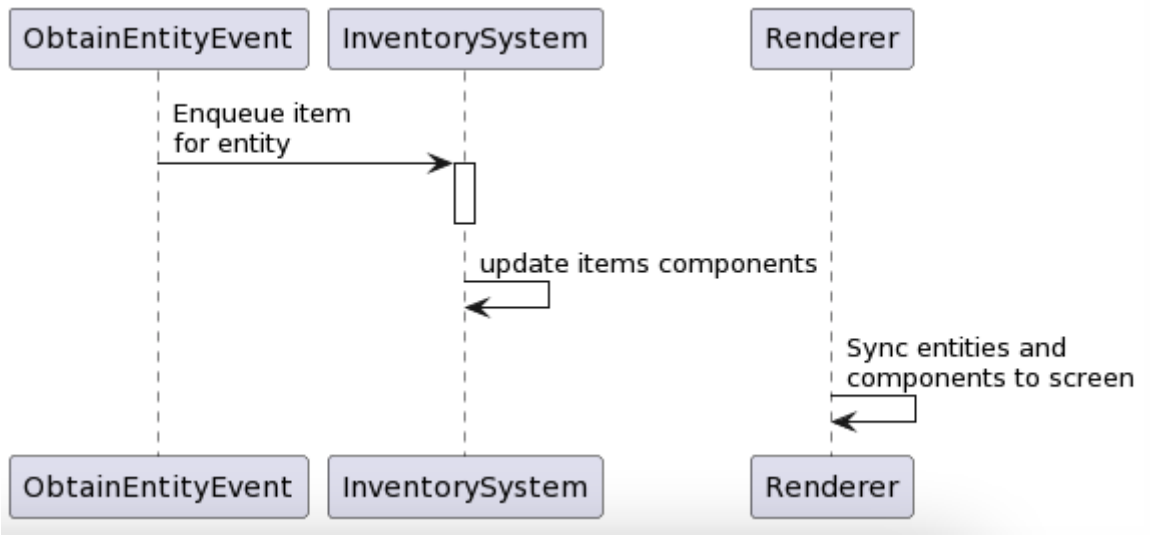


Upon receiving the creation event from the network, all devices add the powerUp Entity to the EntityManger, via an EventModifiable protocol that GameWorld conforms to. The Renderer will then sync with the EntityManager to render the PowerUp to the screen.

Power-up obtain



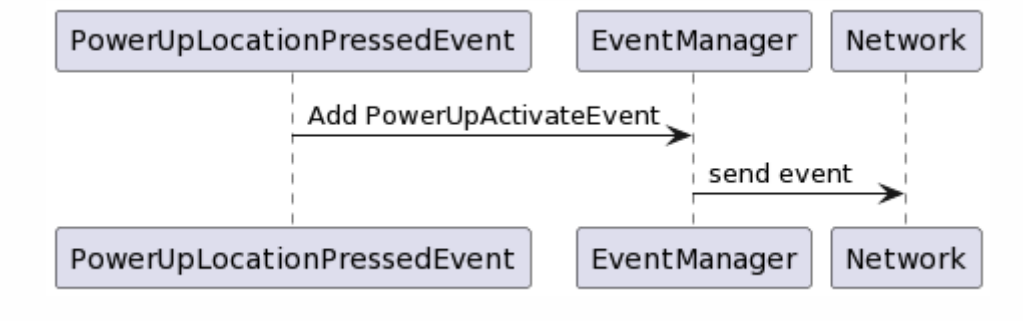
When a player collides with a Power Up, a PowerUpPlayerCollideEvent will be enqueued to the EventManager. Upon dequeued, this event will send an ObtainEntityEvent to the network, with the player and entity obtained specified.



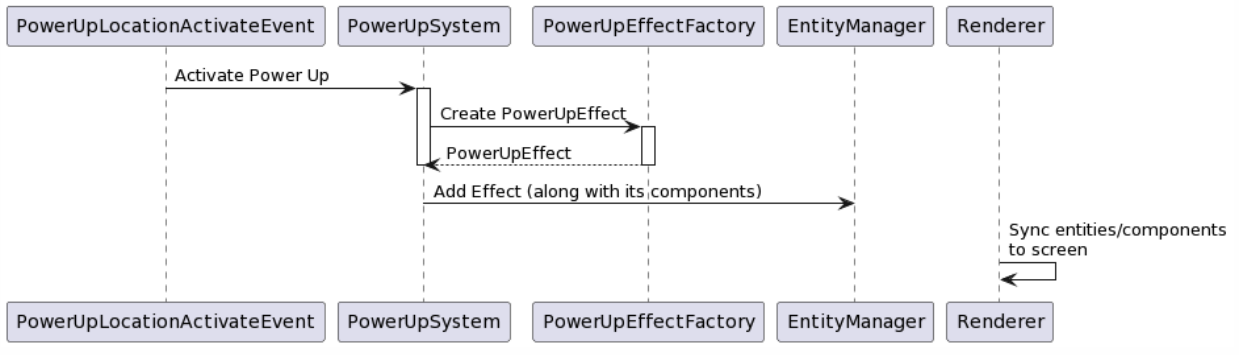
Upon receiving the obtainEntityEvent from the network, all devices will add the entity obtained to the inventory queue of the affected player. InventorySystem will then adjust the components of the entity inside the inventory for Renderer to adjust the view accordingly. Components being adjusted include:

1. positionComponent to change to the inventory queue position
2. spriteComponent to hide the item away so that other players will not be able to take the same PowerUp.

Power-up Activation



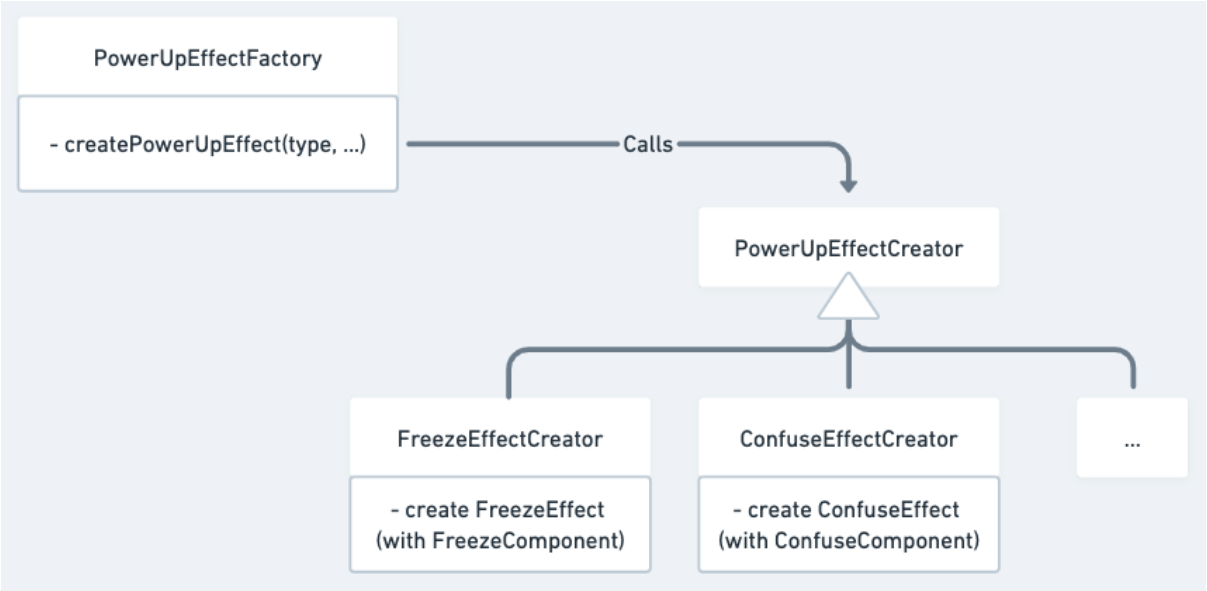
When a player taps on the screen, a PowerUpLocationPressedEvent will be enqueued to the EventManager. Upon dequeued, PowerUpActivateEvent will be sent to the network which will be broadcasted to all devices



Upon receiving the PowerUpActivateEvent, executing this event will call PowerUpSystem to activate the PowerUp. PowerUpSystem will call a PowerUpEffectFactory to create a PowerUpEffect, with a component that indicates the effect of the PowerUp, depending on the type of the PowerUp being activated.

Dedicated Systems are created to handle the logic of the different components. Currently, the Power-up-effect related components and systems are:

1. FreezeComponent, handled by FreezeSystem
2. ConfuseComponent, handled by ConfuseSystem
3. SlowmoComponent, handled by SlowmoSystem
4. TeleportComponent, handled by TeleportSystem
5. BlackoutComponent, handled by BlackoutSystem



To map the powerUpType to the corresponding component, Factory Pattern is used. One major advantage of using this pattern is the conformance to the Open/Closed Principle, where creating a new component/system requires only:

1. adding a new Creator/Factory Class to create the new Power-up Effect
2. adding a new component
3. adding a new system to define the new behaviour of the component.

It does not require any modifications of existing codes.

Disasters

Disaster creation

This part is similar to the creation of Power ups. A host player will send a disaster-creation Event to the network with a probability of 0.5% per frame, anywhere randomly on the screen. Upon receiving the creation event from the network, all devices add the Disaster Entity to the EntityManger, via an EventModifiable protocol that GameWorld conforms to. The Renderer will then sync with the EntityManager to render the PowerUp to the screen.

Disaster Hit

A player hit by a disaster will create a DisasterPlayerCollideEvent, which upon dequeued, will send 2 events to the network:

1. removeEvent, to remove the disaster
2. respawnEvent, to respawn the affected player

Achievements

Achievements are created through a simple factory method, owing to the following requirements:

- Separation of instantiation from usage, as Achievements require network-based actions to remain up to date. This involves setting or instantiation with callbacks and delegates.
- A single place of instantiation, as all achievements will be instantiated together, e.g. to display to the user

To achieve this, an array of achievement concrete types are listed in a factory class, and an init method is made a protocol requirement such that all instances can be instantiated in a generalized manner.

A newly introduced Achievement will only need to be specified within the factory's list in order to be instantiated and used downstream alongside all existing achievements.

For each achievement, network updates are abstracted out to a delegate, which performs update and fetch actions on behalf of the Achievement instance for the specified player.

Latency

In a real-time multiplayer game, a requirement for a smooth gameplay experience is that it is important that in-game latency remains low. However, across the application's various functionalities, there exists some flexibility in latency requirements.

For our lobby, Firebase Realtime Database is used due to its built-in support for transactions, hence facilitating real-time updates while maintaining a consistent lobby state.

For the game, Firebase Realtime Database is once again used due to its ability to maintain 100 ms or faster updates consistently. Again, a broadcast mechanism suits our game's need, as many game events require updating all other players - e.g. a player's position should be sent to all other players promptly, so that their devices can update the respective sprite's position. One issue with using a database as a broker is that enqueued events remain permanently persisted until deletion. To circumvent this, for game modes that do not require event reconstructability, the database has been configured to perform eventual deletion.

For pre and post-game actions, such as keeping the rankings page updated, REST API calls and pollers were used to periodically fetch data in a background thread from a remote backend server, as slower updates would not result in a significant deterioration of user experience.

Testing

Our general approach to testing as of this sprint involves mostly black-box testing, with some components being white-box tested. As we move towards later stages of development, we aim to use a mix of both to argue for correctness in our application.

Deciding on one or the other between white-box and black-box tests will depend on the component(s) involved and the degree of code ownership (as opposed to logic subsumed by a framework).

White-box testing has been used in testing out some individual components. White-box testing is suitable for wrappers over third-party framework calls, ensuring that (i) the transformation and propagation of information through the wrapper is as expected and (ii) the appropriate interface requirements for usage throughout the rest of the application is indeed satisfied by the wrapper. This is because a developer making use of the framework would already be aware of the code, and would be able to use the tests to bridge between the framework and required implementation.

For user flows and gameplay, we plan to continue to use black-box testing instead, as user interaction with the system, as they progress through common flows, is likely to best expose bugs in terms of priority and frequency of occurrence. For the game portion of our application, it would be even harder to test programmatically due to its current dependency on SpriteKit. We plan to test the inputs and outputs to the various game systems instead as a proxy.

Additionally, certain components, such as the lobby system, are unlikely to produce comprehensive test conditions when approached using only black box testing, due to the stringent timing conditions involved when testing different decision paths. For these, we aim to use end-to-end black-box testing through gameplay to argue for general correctness, as well as white box testing, where possible, to test specific scenarios of interest that would likely be especially serious in terms of game integrity.

An additional resource we have access to game logs, by virtue of our event queue architecture. If something feels wrong during a game, we can revisit the sequence of events and determine if there were any issues.

Reflection

Evaluation

Although a lot of the time is used to refactor our code based on the previous meeting, we did enjoy the fruition of our structure as we added new features. Addition of new features mostly require an extension to what we already have, without needing to modify our existing codes. For example, to add a new behaviour to an entity, we only need to create a new component, attach it to the entity, and create a system to define the behaviour of the component. ECS is also a great choice as it makes it very easy to add new behaviours, especially in the context of game-making where a lot of new imaginative behaviours can be possibly added.

Lessons

Facade is useful

Spritekit inherently comes without proper separation between the View and the Model. The View should only handle the rendering to the scene, without knowing any logic, while the Model should only handle the logic. To work around this, our group has implemented a facade pattern to cleanly separate the views and the models. More about this is elaborated on the section on Game Engine Facade above.

Modularity via interface hiding

Oftentimes, we have two classes that interact with each other, but one needs not use all the methods exposed by the other class. These extra methods are utilised by other classes instead, hence it is not possible to make these methods private. Recognizing that there is a violation to the "Interface Segregation Principle", where a class should supposedly be exposed to only interfaces the class needs, our group has found a way to work around this issue. A class is only exposed to a protocol that contains only methods that it requires, while the other class conforms to this protocol. This way, the former class is not exposed to the other unnecessary methods that the latter class provides.

Known issues

Sounds and SpriteKit

SoundManager manages all sounds in our game. It currently uses AVFoundation's AVAudioPlayer to play BGMs and SFXs. We noticed that during gameplay in SKScene, every time a sound is played, the FPS drops from 60 to 50 – 53 fps. The FPS returns back to 60 fps after the sound is done playing. It apparently has been a known issue between SpriteKit and AVAudioPlayer for a while in the developers' community and still appears to be unresolved. We plan to solve this bug by using SKAudioNode, or using a built-in play-audio SKAction, or load balancing AVAudioPlayers, or even playing the audio in the background

thread. Since the issue stems from Apple's SDKs themselves, it was important for us to come up with many strategies to resolve it.

Storyboard and UITableViewCell

As we used Storyboard, we encountered a few commonly known bugs with constraints being broken at runtime in the version our application was tied to. In such situations, we made a decision based on the tradeoff between the severity of the UI issue and the rest of our code. Sometimes, leaving in magic numbers or hardcoded solutions may not bode well for future maintainability.

Appendix 1: Test Cases

General application

Test uninstall / reinstall:

- Upon uninstalling and subsequently reinstalling the application, the player should be able to recover their progress (towards achievements) by logging into the same account as before.

- Upon a player leaving the game through abnormal events (including but not limited to disconnection, device minimization and crashes) that are beyond the natural progression of the game, their game state should not block other users from participating in the game e.g. a player crashes in the lobby while not readied up. They shouldn't remain within the lobby.

User Flow Outside Gameplay

Test Authentication:
When user enter the application for the first time, the page presented should be the Login page, if enter from second time onward after login successfully, the Lobbies screen should be presented

For LoginView:

- Entering the correct email and password, and pressing sign in should log a player in. A successful login message should be displayed briefly following which the player should be redirected to the lobbies selection view

- Entering in an incorrect email and password, and pressing sign in should not allow a player to log in. The player should not be redirected and should be prompted by an incorrect credentials message displayed to them

- Not entering anything and pressing sign in, the result should be similar to entering an incorrect email

- Pressing on the sign up button should redirect players to the sign up view

For SignUpView:

- Entering an incorrectly formatted email should not lead to a successful sign up. An unsuccessful sign up message should be displayed to the player

- Entering an email that has been previously registered should not sign up the player. An unsuccessful signup message should be displayed to the player

- Keying in a weak password with 5 characters or less should not sign up the player. An unsuccessful signup message should be displayed to the player

- Entering a compliant set of { name, email, and password of more than 5 characters } should successfully sign up the player. A successful signup message should be displayed to the player

- No programmatic redirection should occur on this view

Once login (or sign up then login) successfully, you should be inside a LobbiesView that shows the current active lobbies.

- Pressing on the sign out view should sign out the user, and return them to the landing view. Future player navigation, such as by going to the signup view and pressing “Back” for example, should not let them access any view that requires a user to be authenticated

- Clicking on new lobby should redirect the player into the lobby view

- Created lobbies should be visible on the lobby selection page. They should include the name and the gamemode of the lobby, as well as the number of players currently inside the lobby and the maximum number supported

- The lobby selection view should be dynamically updated as the player remains on the view. Additionally, when a player exits the view and re enters through various avenues, the lobbies listed should be current.

- For a game in progress, the corresponding lobby should be greyed out, regardless of the number of players. Tapping on the lobby cell should have no effect.

- For a gamemode where max capacity has been reached, the cell should be greyed out. Tapping on the lobby cell should have no effect.

- For a lobby where all players have been removed (through any means), the lobby entry should immediately disappear and not be tappable.

Once you select an existing lobby or create a new one, you should be transfer to the LobbyView, where the following cases can happen:

- Pressing ready, under a situation where the player becoming ready would not allow the game to begin, should turn the player’s corresponding cell’s background to green across all lobby participants’ devices to indicate their ready status. On the player’s device the leave button should be disabled and cannot be interacted with

- A host leaving through any means, e.g. disconnection, device crash, or through the UI, should close the lobby if the player is the only player in the lobby.

- A non-host player leaving should not change the host of the lobby.

- Pressing ready, under a situation where the player becoming ready would allow the game to begin, should transition the player to the game loading view

- When the number of players in the lobby is lower than the minimum required for the game to begin, the players should remain in the lobby regardless of ready status. The game should not begin.

- Pressing leave, when the player is not ready, should allow the player to exit the lobby selection view. If the player was the host, among the remaining players, the player with the lowest userId (lexicographically) should become the next host

While you are inside the LobbyView you can customise your lobby with mode and selections. The following cases are considered:

- If the user is in single player mode (TimeTrial mode/default mode), they can change game modes by tapping on the gamemode text specified at the top of the view.

- If the user is in a multiplayer lobby and is host, they can also change the game modes

- If user is in single player or is the host in a multiplayer lobby, they can specify the seed of the level to play on by tapping on the seed text below the game mode and input a number

- If you user is not host, the actions specified above will be prevented by unresponsive tap

- Game modes which are incompatible with the current occupancy should not be selectable. An example test would be to start a time trial mode, and have another player join. The host, in an unready state, should not be able to change to time trial until the capacity drops <= the maximum number of players for time trial.

Once all players are ready and the number of players satisfy the game mode requirement, the game will enter the transition state:

- In a multiplayer game mode, players should transition into the game at (close to) the same time

Once the game finishes the transition stage, you will enter GameView, the User GamePlay section will go into details on the test cases for this. Upon landing on the final platform, a player should be sent to post-game view

Ranking interactions in PostGameView for Time Trial mode:

- On arrival at the end game view after a time trial round, the player should be able to see up to the top 10 fastest times for their seed. Players can also see their ranking relative to the seed global ranking, e.g. 16th place if they are not inside the top 10.

- On arrival at the end game view after a race to the top round, the player should be able to see their own prevailing position, as well as others who have finished earlier.

- On arrival at the end game view, if the player is the host, a new host should be created or the lobby should be destroyed, depending on whether there are other players still within the game view.

- Reusing the same seed for a time trial round should retrieve the rankings in an

idempotent manner. For the race to the top mode, the rankings should always begin as empty and fill up with only players within the lobby.

- A player waiting in the post-game view, if they are not in the last position, should be able to observe the rankings table being updated by other players finishing after them.
- The highscores should be updated together with the player's position, e.g. if the player beat the previous third place, their score should appear in third place and the previous 10th place, if present, should be replaced.
- The player should be able to observe a "Lobbies" button below the rankings table. Pressing on the button should take the player back to the lobby selection view. The player should be able to create a new lobby. The player should also be able to join another existing lobby without limitations.

Test Achievements:

Upon clicking on the Achievements button at the top left within the Lobbies view, the player should be redirected to the Achievements page.

- The player should be able to observe a back button. Upon pressing this button, the player will return to the lobby selection view.
- The player should see achievements that they have unlocked with a full progress bar. The player should see achievements that they have unlocked with a bright, opaque image.
- The player should see their true progress numerically represented beyond the achievement requirement, if they have unlocked the achievement.
- The player should see achievements that they have yet to unlock with a less than full progress bar , corresponding to the amount of progress they have made. The player should see achievements that they have yet to unlock having a dark, less than opaque image.
- Once the player has exactly met the achievement requirement, at that point and from that point onwards, the player should be able to see their achievement as unlocked.

Once the game finishes the transition stage, you will enter GameView, the User GamePlay section will go into details on the test cases for this. Upon landing on the final platform, a player should be sent to post-game view.

User Flow for Gameplay

Joystick Control

- As the joystick is dragged horizontally, the player's sprite should move in the direction corresponding to the joystick's displacement from the centre
- The player's sprite should move with a speed proportional to the joystick's displacement from the centre
- When the joystick is released, the player's sprite should no longer move
- When faced with a collision, the player's sprite should not move into the colliding object in spite of further collision-causing joystick inputs
- When the player is in the air, joystick inputs should still allow them to fall towards the left or right, corresponding to the direction of joystick displacement

Jumping Control

- When a player is standing on a platform, pressing the jump button should displace the player vertically upwards
- When the player is falling, pressing the jump button should not have any effect on the player's sprite.
- When the player is jumping, pressing the jump button should not have any effect on the player's sprite.
- When a player is jumping, usage of the joystick should alter the direction of jump

PowerUps Interaction

- When a player hits a power-up on the game area, it should be kept in an inventory queue shown at the bottom of the screen
- When a guest hits a power-up, the power-up should be hidden and non collidable from the scene
- When two players hit the power-ups at the same time, only the first player should get the power-up
- An unclaimed power-ups should disappear after 7 seconds
- Tapping the screen will activate the power-up at the tap location. Power-up effect should correspond to the first power-up in the queue.

- After power-up activation, power-up in the queue should disappear.
- When activating a powerUp, a visual effect should be seen on the tap location
- Players nearby should be affected by the power-ups.
    - Freeze effect should Nullify the movement and jumping of player
    - Confuse effect should reverse the horizontal movement of player
    - Slowmo effect should slow down the movement and jumping of player
    - Teleport effect should teleport the main player to the location of tap
    - Blackout effect should darken the screen of all other players

Random disasters

- On the game area, there should be random meteors appearing at some point of time
- Colliding with the rock of the meteor should trigger a respawn event on the affected player.
- Collision with a meteor should make the meteor disappear
- On a multiplayer mode, a guest who hits the meteor should also cause the meteor to disappear, and the guest to be respawned

Clouds

- A player should be able to jump on and onto a cloud
- If a player is on a moving cloud, the player should not move along with the cloud.
- If a cloud moves pass a player standing on top of the cloud, the player should fall.
- A player standing on a cloud should not be falling; a player falling can land on a cloud, which will arrest their fall
- When two players jump on the cloud, the first player who came first should respawn at the bottom of the screen.

Camera

- At a certain point when a player’s sprite is going up, the camera should scroll down to follow the player

- When the camera is scrolling, other fixed on screen components, such as the joystick, jump button and timers should not be displaced

- When a player's sprite falls, the camera should scroll up to follow the player going down

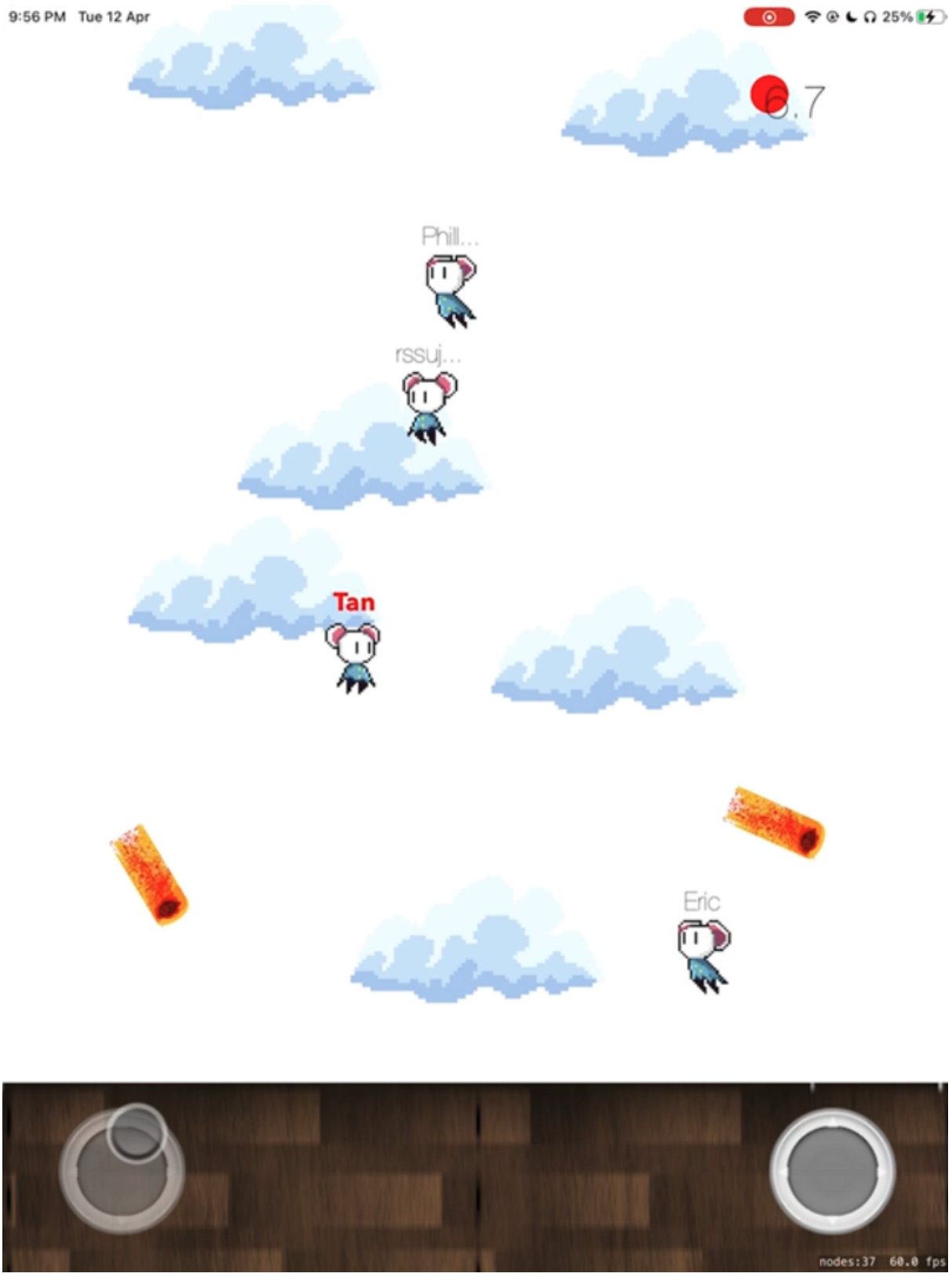
Goal State

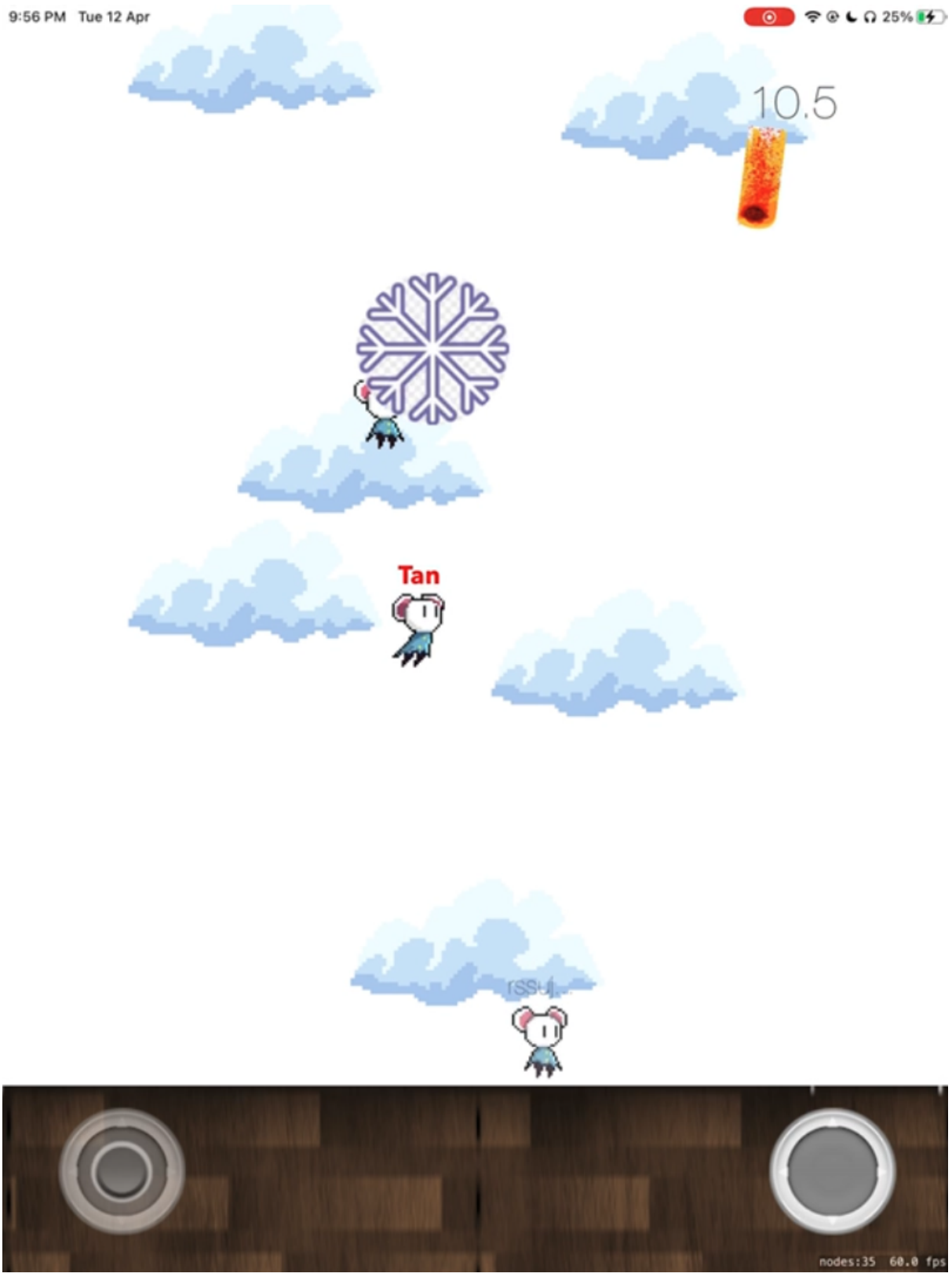
- The goal state should be a yellow end platform that the player's sprite can land on

- Upon landing on the end platform, the player is directed to the post game view

Appendix 2: Screenshots / Video

Gameplay video can be found at:
https://uvents.nus.edu.sg/event/20th-steps/module/CS3217/project/6


9:56 PM Tue 12 Apr
25%
6.7
Phill...
rssuj...
Tan
Eric
nodes:37 60.0 fps


9:56 PM Tue 12 Apr
25%
10.5
Tan
nodes:35 60.0 fps

Appendix 3: Concept Arts

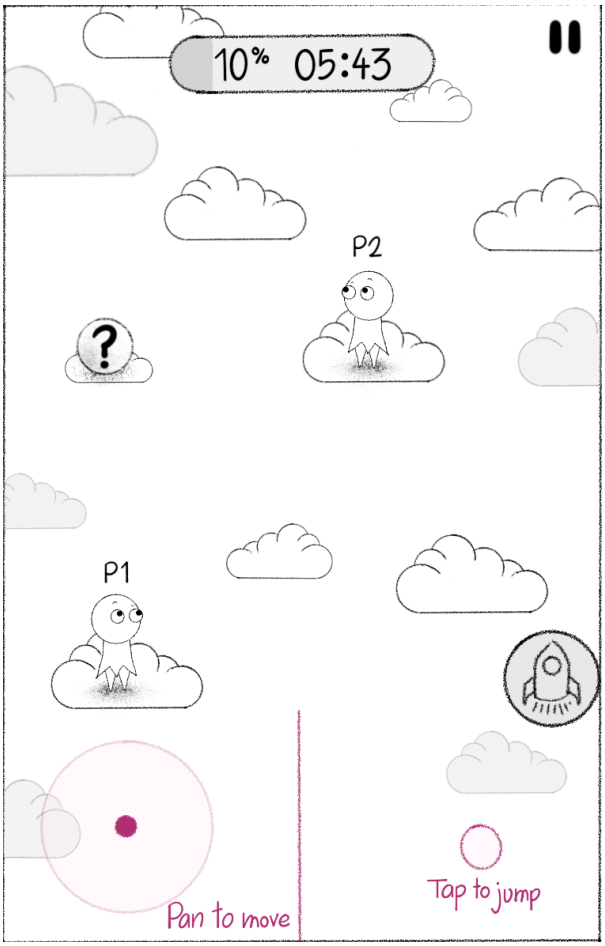


Figure 1: Idle game view

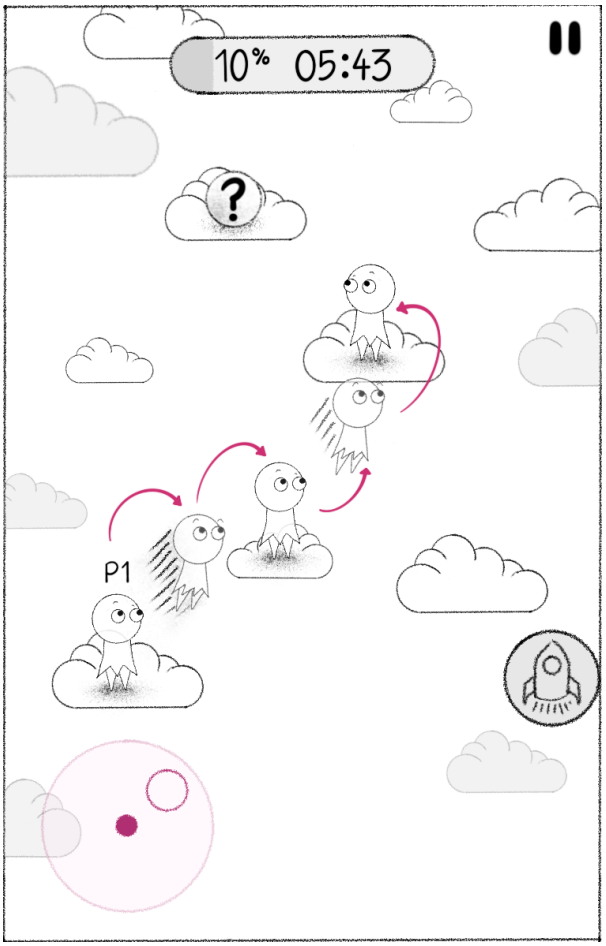


Figure 2: Jumping mechanism