CS3217 Final Report

Adam Chew, Foo Guo Wei, John Phua, Tay Yu Jia

i. Requirements

Overview

Hive2D is a real-time multiplayer strategy game where players must develop their colony while managing their economy, and compete with one another for dominance in the world. Each player starts with a central hive and builds nodes to gather resources, expand their colony, and launch attacks on their opponents.

Features and Specifications

Features

- Lobby system
    - Players can create or join games through lobbies
    - Players can choose a display name before creating/joining the lobby
    - Players can share their room code to invite other players to join
    - Players can change lobby settings while inside lobby
- Game
    - Players can tap the screen to build a node
    - Players can pinch the screen to zoom in and out
    - Players can pan the screen to move the map
    - Players can build multiple resource nodes that gather different resources
    - Players can upgrade nodes with resources
    - Players can see health bar of nodes when combat is in progress
    - Players can leave game
    - Players can see their resources accumulated
    - **Players can build combat nodes that have different attacking behavior**
    - **Players can see projectiles fired when combat nodes attack**
    - **Terrain tiles can have different effects and restrict the ability to build a node on it**
    - **Fog of war (players can only see within a certain radius of their own nodes)**
    - **Minimap (players can see a bird's eye view of the entire game map)**

Specifications

- Lobby system
    - A lobby is initialized with 1 player as host, a generated human-readable room code and default lobby settings
    - A lobby can allow up to 4 players
    - A minimum of 2 players are required for a game to start
    - Only the host should be allowed to start game
    - Players should be removed from lobby when disconnected or closed application

- Networking
    - Lobby setting updates should be propagated to all players in lobby
    - Game actions should be propagated to all players in game
        - No actions are lost, actions from different players are ordered by time and delivered to all other players
- Game
    - A game is initialized with players and settings from the lobby
    - Game should initialize players with a Hive node each
    - Game should be updated by actions from all other players in each iteration of the game loop
    - Game should allow players to add nodes to their colony
    - Game view should be zoom-able and pannable
    - Game should rely on logical coordinates
    - New nodes can only be built within a certain radius of existing nodes
    - New nodes cannot collide with existing nodes
    - Game should support multiple types of resource gathering nodes and combat nodes
    - Resource Nodes can be upgraded to improve abilities
    - Combat nodes are able to attack enemy nodes that are in range by reducing their health
    - Nodes that have zero health should be destroyed
    - **Combat nodes can have different attacking behavior**
    - **Terrain tiles can have different effects**

User Manual

Refer to ‘User Manual’ in Appendix.

ii. Design

Overview

Top-level Organization

Hive2D is organized into the following packages - Networking, Authentication, Lobby, Game.

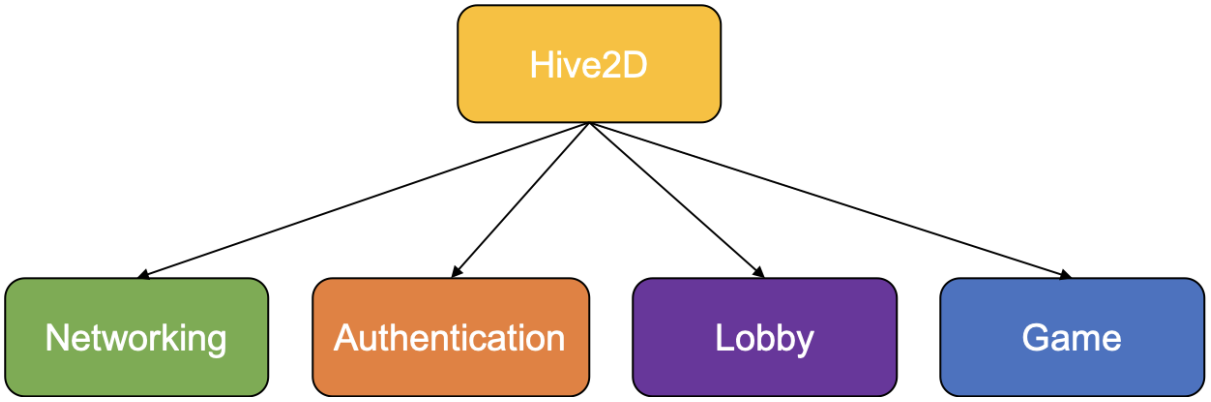


Figure 1: Top-level organization

Third-party modules

- Firebase

Used for simple anonymous authentication and setting up lobbies between devices. Provides service guarantees & behaviour that reduces work needed on the backend.

- RabbitMQ

Used as a cloud message broker between devices, syncing game actions between players. Provides ordering guarantees for actions across devices.

- NVActivityIndicatorView

Used for UI/UX.

Module Structure

Networking Package

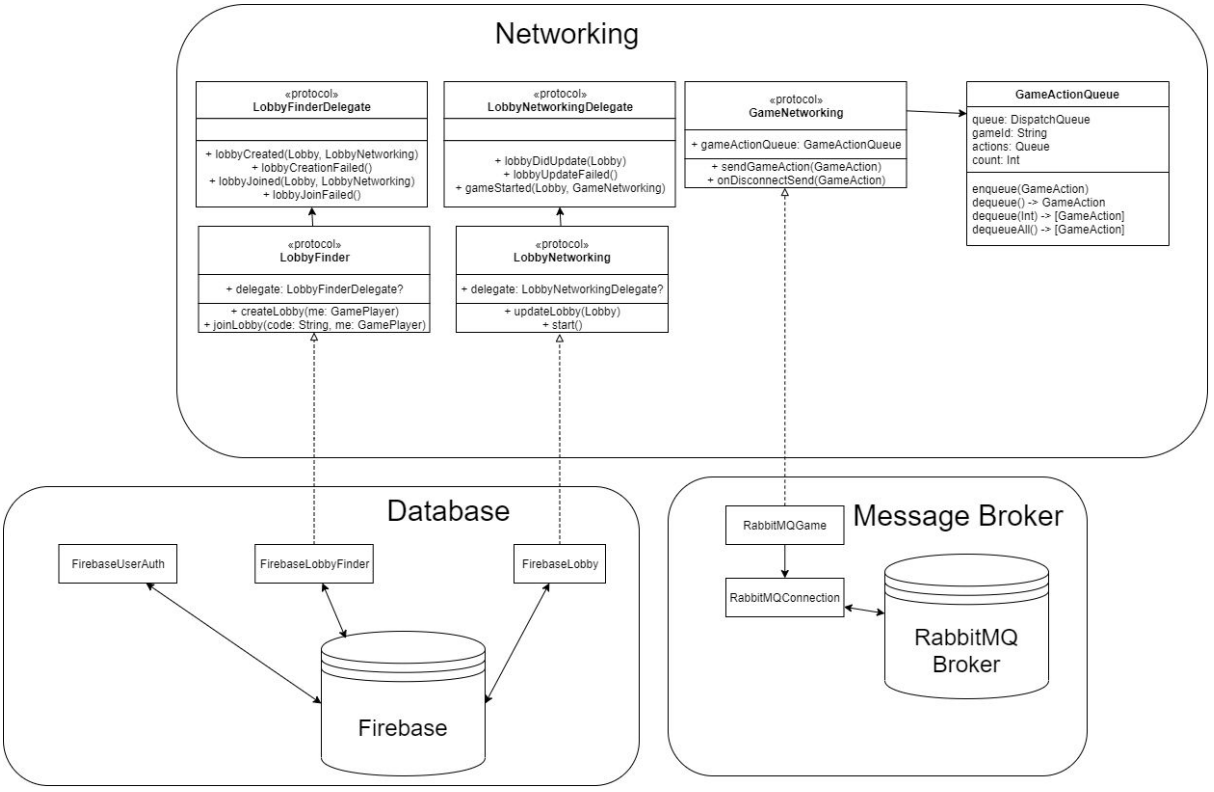


Figure 2: Class diagram for Networking

The *Networking* package handles syncing of data across different devices. This is used in Lobby and Game packages to sync lobby data and game messages respectively. This package makes use of the **Delegation** pattern extensively, as networking receives messages asynchronously. The package also follows the **Facade** pattern to decouple details of the underlying database from the rest of the application. As a result, the underlying services can be changed without affecting the rest of the application.

Firebase works well for real-time lobby state synchronisation between players while RabbitMQ was added to guarantee common ordering of game actions across players during the game.

Authentication Package

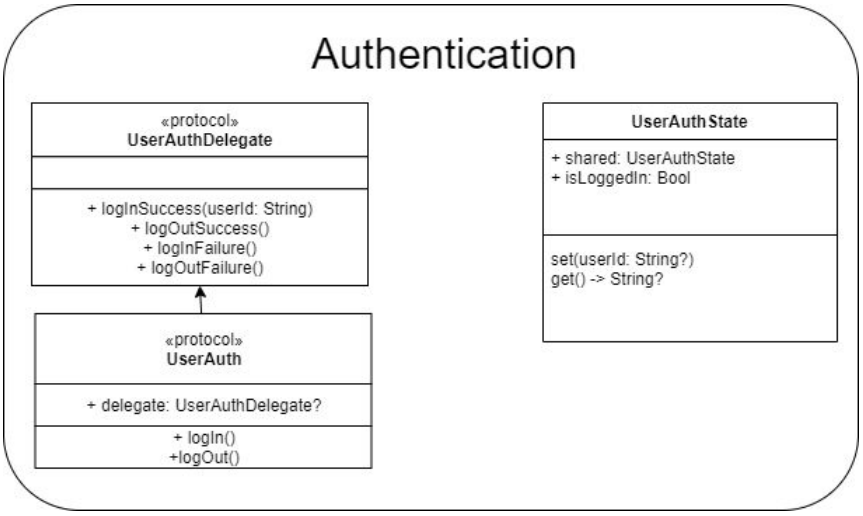


Figure 3: Class diagram for Authentication

The *Authentication* package creates a unique identity for the device for use in other packages. It currently allows users to play anonymously without the need of creating an account. UserAuthState is implemented using a **Singleton** pattern. We find this acceptable as user context is shared across the whole application, and there should only be a single instance of it at runtime. This allows for easy access to the user context without having to pass it around.

Lobby Package

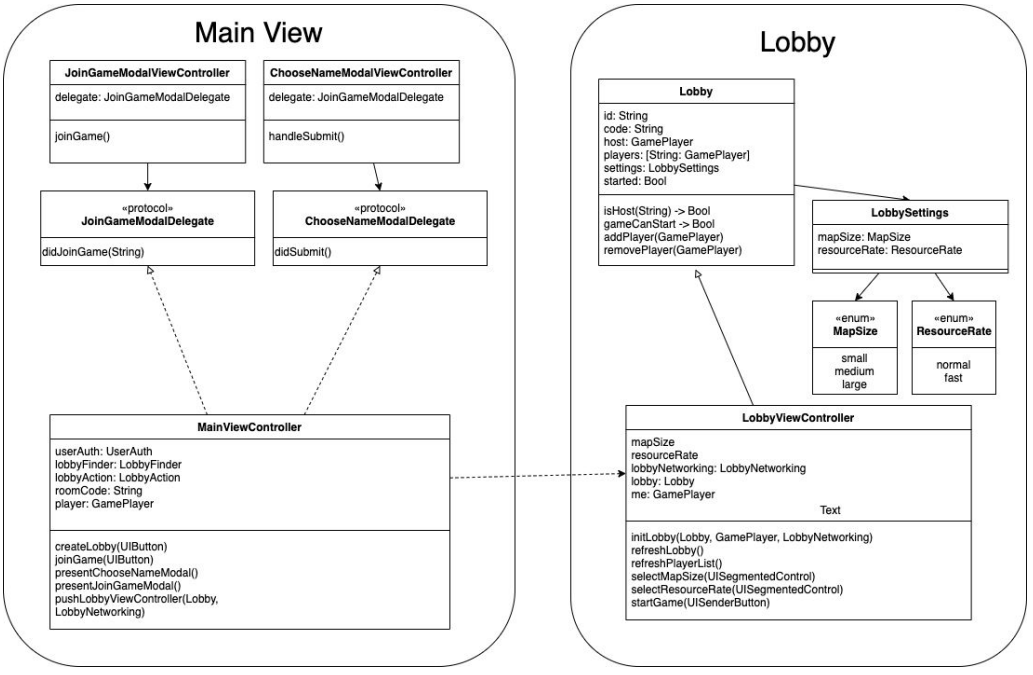


Figure 4: Class diagram for Lobby and Main View

The *Lobby* package handles the matchmaking of players before a game is started.

Game Package

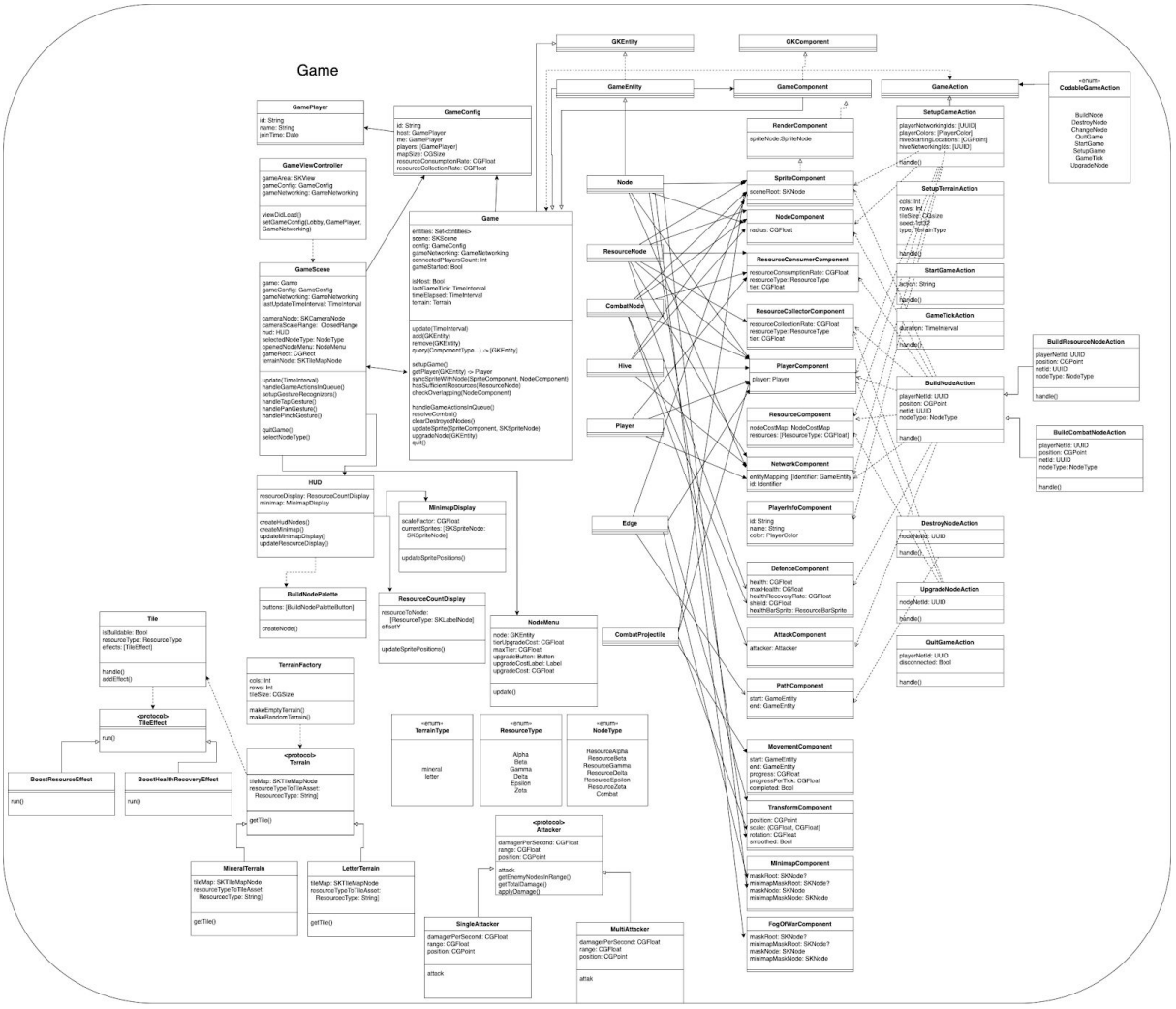


Figure 5: Class diagram for Game

The *Game* package handles all logic and display for the game. It follows an **Entity-Component-Action** architecture (described in detail below), which is our adaptation of the popular Entity-Component-System architecture but implemented with GameplayKit and streamlined for our game. Our core game data types are all subclasses of GameEntity and GameComponent. Each GameComponent encapsulates one functionality — containing use-case-specific logic and data at both the entity and game level, the latter implemented as static functions and variables. GameEntities are templates that provide a specialised initializer defining the minimal set of components required to form that entity.

Entity-Component-Action Architecture

Motivation: Entity-Component-System

This is the “vanilla” architecture which espouses the idea of entities as IDs, components as data and systems as logic. The main benefits of this is that the architecture is composition-based and data-driven. This makes evolving game design easy.

Underlying Engine: GameplayKit (Entity-Component)

Apple’s GameplayKit came with its own version of ECS engine, comprising entities, components, and component-systems. This architecture merges systems and components, and thus removes the focus on being data-driven. Instead, it focuses simply on using composition to organize code and functionality. Components should encapsulate one functionality, and attaching a component to an entity should endow it with that functionality (with minimal changes elsewhere to the code). We chose to implement our game on top of this engine, as that saves us the effort of having to roll our own.

Game, GameEntity and GameComponent

As core game logic is now distributed amongst the components, there needs to be an orchestrator that brings them all together into a cohesive unit. For this, we introduce the concept of a Game, which acts as the (minimal) core of our architecture — managing the game entities and serving as the interface with the networking component. To integrate this into the GameplayKit engine, we created GameEntity and GameComponents, which are subclasses of GKEntity and GKComponent respectively but extended with didAddtoGame and willRemoveFromGame callbacks.

GameActions

Hive2D is a multiplayer game, which means game state synchronization of our engine is a core feature. We’ve decided to adopt a heavy-client approach where the server is (literally) just a message broker synchronizing a log of game events. Two clients with the same log should produce the exact same game state.

GameActions are our constructs that represent both the important game events that require network synchronization, as well as the systems that manage the handling logic. It leverages the **Command** pattern to distribute the logic for different actions, such as game setup, synchronization, etc. Sometimes, related actions can hint towards Object-Oriented behavior; one such example is the BuildNodeAction for both Combat and Resource nodes, which possess mostly similar logic. As such, it was implemented using protocol and extensions (to simulate an Abstract Class), where BuildNodeAction provides the default method `handle(Game)` and its

subclasses are responsible for subclass-specific functionality like creating the specific type of node.
GameActions also serve as the bridge between Game and Networking to advance and synchronize game state across player devices. To ensure that **networking concerns are kept separate**, we have chosen to serialize GameEntities together with GameActions when sending over the network; therefore, relying on our serialization factories to link the correct entities after decoding network messages. Furthermore, the serialization factories need only interact with the NetworkComponent (instead of the Game), and any entity with a NetworkComponent is automatically serializable, due to our design choice of encapsulating functionality in components.

Application flow

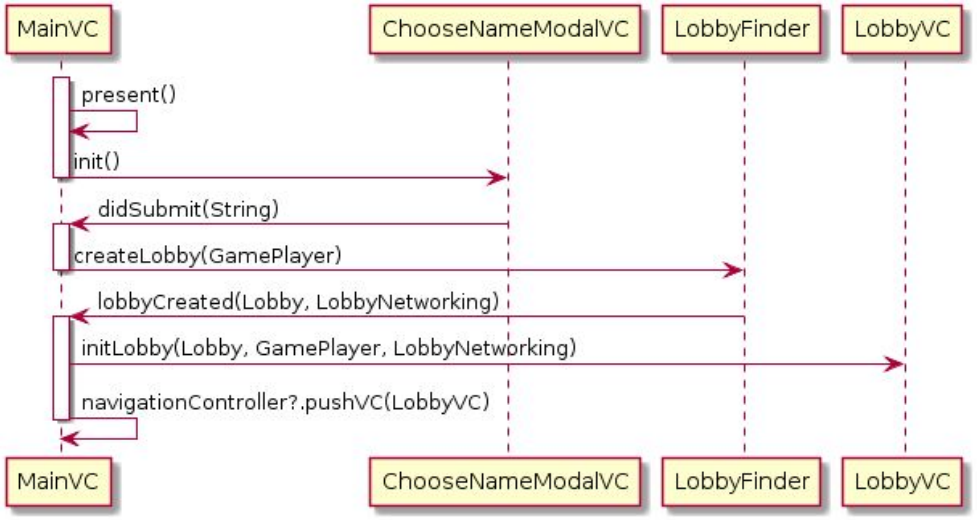


Figure 6: Sequence diagram when player creates a lobby

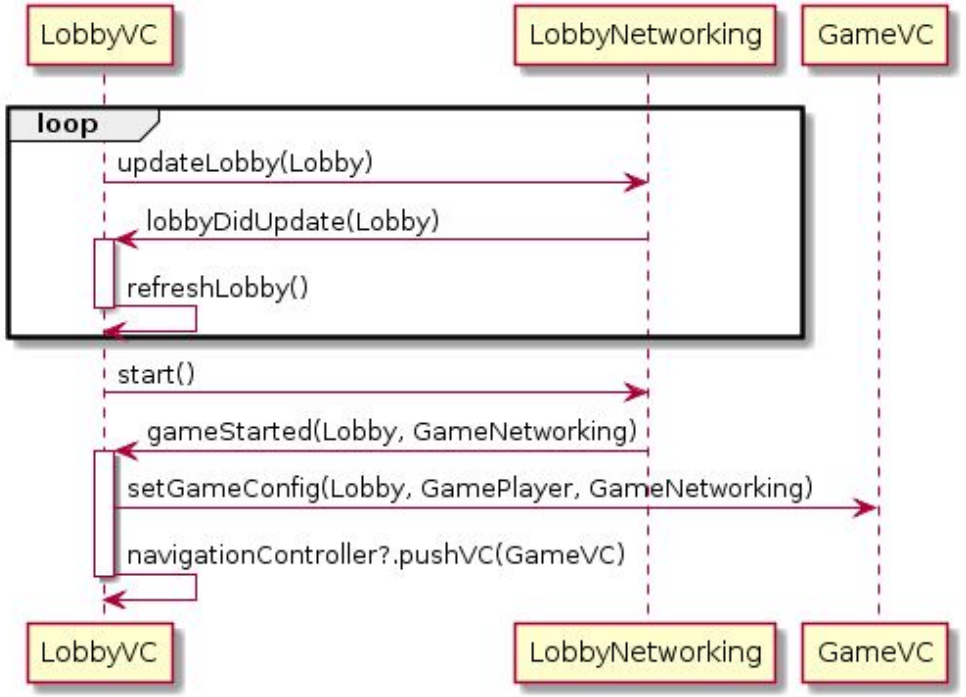


Figure 7: Sequence diagram for lobby

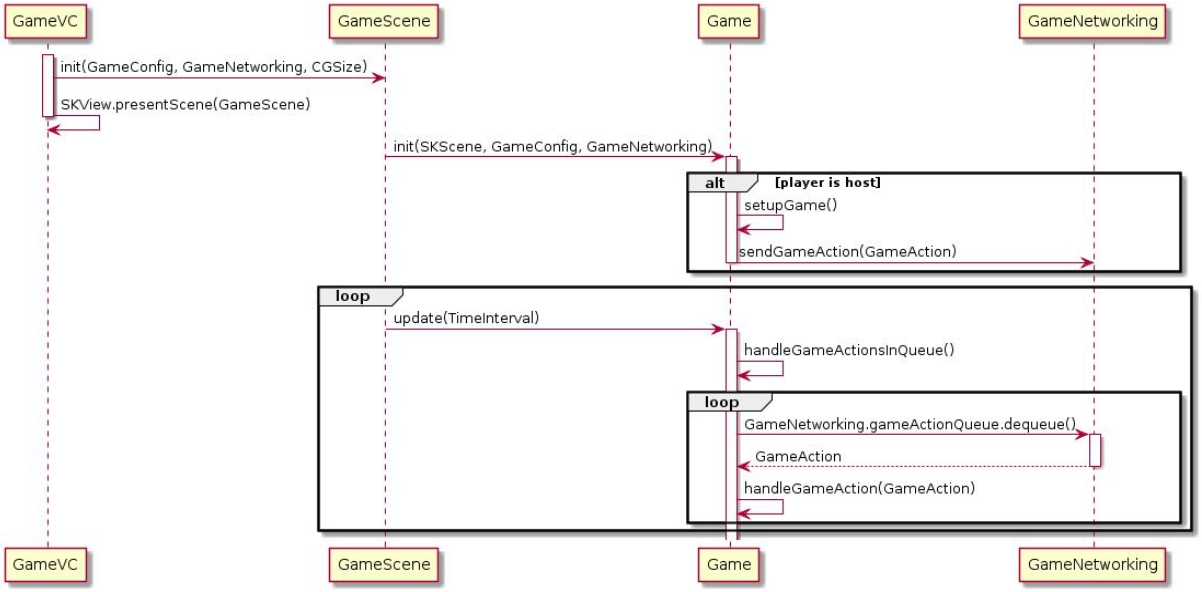


Figure 8: Sequence diagram for game loop

Design Considerations

1. **Adapting GameplayKit with Game Integration**

The main issue we faced here was that some of our functionality relied on both entity-level and game-level information. For example, a sprite needs to be added to a scene or root node to be rendered — which means that the component handling the rendering logic needs access to every entity's sprite. However, each entity should also have a sprite component, representing the entity-specific sprite to render. The dilemma lies in how to organize these two components.

Option 1: GKComponentSystem

We considered using the component-system construct to handle game-level logic, leaving components to handle entity-level logic. This is quite a clean separation of concerns. However, it required changing from per entity updates to per component updates — basically, every component type needs to have a corresponding component-system, regardless of whether it has to work with game-level information. Furthermore, the Game class would have to be updated whenever new component types are created or removed, since it manages the component-systems.

Option 2: Move game-level logic into Game

The second option we considered was to move logic at the game-level into the Game class. This made sense because those are, after all, logic and data that are not entity specific, and Game is already a manager class. However, this meant that functionality logic is now in two places, and as the feature set grows the Game is only going to get bigger with more variables and functions. It also goes against the Entity-Component philosophy of organizing functionality in one place.

Option 3: Integrate Game into GKEntity and GKComponent; All logic in components (Implemented)

The third option, which we eventually implemented, was to keep all related game functionality in one place — the component. Game-level information will be stored as static variables on the component, while entity-level information and logic are implemented as class variables and functions. The component relies on didAddToGame and willRemoveFromGame callbacks to manage their own game-level data. The advantage of this approach is a small Game which only needs to loop through all components to call the delegate functions whenever an entity is added or removed.

2. **Changing GameAction to protocol instead of enum to leverage on Command Pattern**

Option 1: Using enum for GameActions, handling GameActions in Game

In Sprint 1, we had a series of handlers in Game to handle the different actions, with GameAction being an enum containing the different types of actions. This provided

good separation between the Networking and Game layer, but Game was bloated with handlers and was hard to maintain when action behaviour was changed.

Option 2: Make GameAction protocol, use **Command** Pattern for different action types (Implemented)
We flipped the dependency between Game and GameAction, with all GameActions implementing a protocol with the method `handle(game:)`. This allowed us to split the logic for handling each action into the respective actions, greatly simplifying the action handling logic within the Game. Even with GameAction being a protocol, separation between Networking and Game layer was maintained by using an enum class as an Adapter.

3. Synchronising game steps across players

In our design, we have decided to pass actions across players instead of partial or full game states. This reduces the size of data exchanged between devices and allows us to rely on the devices to handle Game logic. However, this created issues with game state synchronisation as each device was updating resources and resolving combat using their internal game loop. For example, a player could be gaining resources from a node that they have just built, but the build node action was delayed to other players and the resources are not credited on the other devices.

Solution: Host sends game step updates as an action regularly (Implemented)
Game updates should be synchronised between players. This was accomplished by having the host send regular `GameTickAction`s, which updates resources and resolves combat nodes when handled.

4. Logical Game Coordinates and the SKCameraNode

We implemented our game world as a fixed sized square, offering players three different map sizes to choose from. This meant that our node sizes and other game related constants only need to be tuned to one set of logical coordinates, without having to worry about how they are being presented.

Originally, we intended to allow for our views to be presentable to any screen size and any aspect ratio — after all, we just needed to scale the sprites accordingly. This was a bad idea as there were many presentational components that had to be laid out, and changing the screen size / aspect ratio necessitated a change in those layouts. For example: our game has a heads-up-display component that stays on the player's screen all the time, and the offsets, font sizes and placements optimized for a particular screen size simply does not work for another one with a different aspect ratio.

Hence, we decided to also fix our viewport dimensions at 1024 * 768, which is the most common iPad landscape aspect ratio, and design our game for this aspect ratio only. In the future, if better support for other aspect ratios is required, a different set of layouts can be crafted for those, and the user will be able to switch accordingly.

Next, we had to figure out how to map the viewport to the actual world view that the player is supposed to have. For this, we turned to the SKCameraNode, which allows us to scale and pan our display view around the scene by giving the node a position and scale. As part of game design, we also limited the size of the world view that the player can have (i.e. zoom in/out limits). The following figure illustrates how all these come together (the red area is the actual world view, and the whole world will be scaled such that the red area fills the viewport):

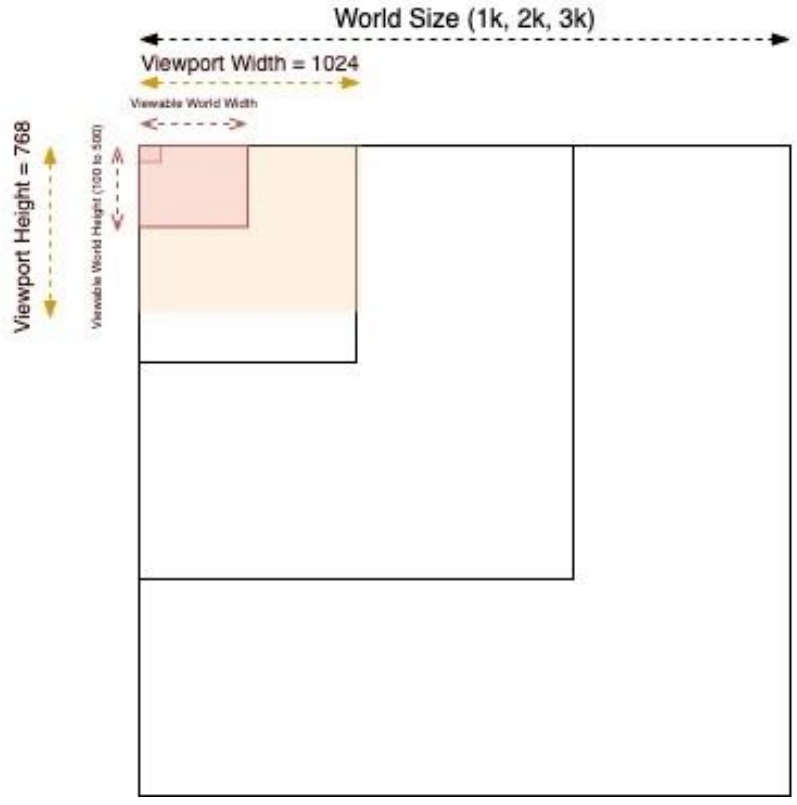


Figure 9: The View and the World

5. **Different attacking behavior for combat nodes**

Option 1: Store an attackType enum in AttackComponent

This is the simplest approach that fulfills the requirements but it introduces code bloat because of the huge if-else/switch cases required to choose the attacking behavior. This makes it quite unmaintainable in the future, as many variants of attack behaviors could be introduced.

Option 2: Use an AttackFactory that generates closures to encapsulate attacking behavior

AttackComponent can be initialized with an AttackFactory that returns a function with signature `([Node]) -> void`. This function can determine how the enemy nodes will be attacked (e.g. single/multi target), what is the range of the combat node, etc. This is a pretty neat implementation that is easily extensible. However, it does not allow for reusable logic as each function is separate from another.

Option 3: Create an Attacker protocol and have AttackComponent use it to resolve attacking behavior (Implemented)

AttackComponent can be initialized with an XXXAttacker that conforms to the Attacker protocol. The Attacker protocol specifies properties of an Attacker (e.g. damagePerSecond, range, position) and leaves the implementation of attacking behavior to the implementer. Using extensions, it is also possible to provide reusable logic to all conforming types of Attacker. Hence, when introducing new Attacker types, only the minimal requirement of specifying which enemy nodes are targeted is required.

6. **Terrain generation**

Option 1: Use constructors for instantiating custom terrains

The most straightforward way to create a terrain is to use its constructor. While this has the least overhead and is the fastest way to add new terrains to the game, it entails creating a switch case to check for the right terrain type and call its constructor, which occurs in the game setup logic.

Option 2: Create a factory with shared terrain properties and terrain generation methods (implemented)

This method adds the overhead of having another class manage the creation of terrain, which means also one more file to change when the game designer wants to add a new type of terrain to the game. However, the benefit we get comes from shifting the switch case from the game logic into the factory, which makes the game logic easier to follow and less cluttered.

7. **Terrain tile effects**

Option 1: Create tiles with a custom effect by overriding the effect handler method of base tile protocol/class

This is the simplest way to support easy extension of the base tile class for tiles with custom effects. However, it does not allow for easy extension of existing tile effects, because overriding the effect handler results in not being able to reuse the code for those effects.

Option 2: Create TileEffect protocol and add custom tile effects to the Tile object (implemented)

In this approach, custom tile effects are created as classes that implement the TileEffect protocol and can be added to the list of tile effects of a Tile object. This allows for code reuse of existing tile effects as one can simply add the existing tile effect PLUS any other tile effects (existing or not) to create a new tile effect that is a combination of all the added effects. Note that the game designer has the choice to either create a new tile effect by creating an entire new class, or mix and match the existing tile effects, or a combination of both, thus allowing ultimate flexibility. As the game grows bigger to incorporate more elements and customization, terrain tile effects scale easily by allowing a complex mix of many individual effects.

iii. Testing

We plan to write unit tests and integration tests. For components that use networking, we intend to use stubs that implement the appropriate networking protocols.

Performance tests will be conducted to verify that the game runs smoothly at 60 FPS.

Detailed testing strategy can be found in the Appendix.

iv. Known Bugs

When creating/joining a lobby for the first time, the lobby might sometimes be stuck loading (>30 seconds) as the RabbitMQ client fails to establish a connection to the broker. This is likely due to some configuration issue with the cloud RabbitMQ service we used (CloudAMQP).

- **Resolution**: Restart application and try again

v. Reflections

Evaluation

- Easy to add or change behaviour by adding/modifying only the relevant components — all relevant logic are in the same place
- Easy to tweak game parameters through configuration (all constants in one place)
- Networking is abstracted from the rest of the game, easy to replace (we changed from Firebase to CloudAMQP)

- Easy to add new terrain effects through composition of individual effects

Lessons

- It is tricky to rely on 3rd party services for correctness guarantees. In this project, we tried to remove the need of a backend server by using cloud services/platforms like Firebase and CloudAMQP (cloud RabbitMQ broker). However, these services did not provide for certain requirements needed for the application. Specifically, Firebase could not ensure common ordering of GameActions across players, but RabbitMQ will cause late players to lose prior GameActions. As a result, we had to compensate with less than elegant workarounds on the application to meet the requirements. Ideally, a lot of the session management and synchronisation logic would be housed on a backend server, which can provide the required guarantees in a much cleaner fashion.
- We spent quite a while trying to reconcile GameplayKit with vanilla ECS, and how to design the game interactions the “ideal” way. Along the way we realised that the more we tried to separate the components and systems to eliminate cross-component interaction, the less useful engines like SpriteKit and GameplayKit become (e.g. in SpriteKit the physics engine is inherently coupled to the sprite). Eventually we accepted that it is not helpful to try and achieve ECS-style separation of concerns while using GameplayKit and SpriteKit, and embraced the less restrictive philosophy of “components as functionality units” to design our game.

vi. Appendix

Testing Plan

Unit Tests

GameActionQueue (already implemented)

- When used serially, should function exactly like a FIFO queue
- When actions are enqueued in parallel, all actions should appear in the queue (no actions are lost)

Game

- When handling setup game action, should add Player and Hive entities to the system corresponding to the game config
- When handling start game action, should increment the connected players count by 1,
    - And if count is equal to the player count in config, should set flag for game started to true
- When handling build node action, should check if node to be added can be added
    - If node is overlapping any other node, should return without doing anything
    - If player executing the action has insufficient resources, should return without doing anything
    - Else, should add the node to collection of entities
- When handling destroy node action, should destroy the node corresponding to the entity containing the same network id
- When handling quit game action, should delete all the player's entities
- When handling change node action, should check if node can be upgraded
    - If player executing the action has insufficient resources, should return without doing anything
    - Else, should change the node

LobbyFinder

- Creating a lobby should return an appropriate lobby instance with the player as host
    - A 4-digit code should be generated for the lobby that other players can use
- Joining a lobby using an appropriate code should add the player to the given lobby and return the appropriate lobby instance
- Attempting to join a lobby with an invalid code should return failure
- Attempting to join a lobby that has already started should return failure

LobbyNetworking

- When a player joins the room, all existing players should be updated on it

- When a player changes the settings of the game, all players should be updated on it
- When host starts the game, all players should be notified of the start event exactly once
- Players that are disconnected/quit the game should be removed from any lobby they were in

GameNetworking

- Actions sent should be propagated to all other players in the game
- Actions received should be added to the GameActionQueue asynchronously

UserAuth

- On login success, should populate the UserAuthState singleton appropriately
- On logout success, should nullify the UserAuthState singleton

Integration Tests

Main view

- When pressing the 'Create Lobby' button, a modal (ChooseNameModalViewController) should be presented asking the player to type their name.
- When pressing the 'Join Game' button, a modal (JoinGameModalViewController) should be presented asking the player to type a room code.
- After typing the room code and confirming, a modal (ChooseNameModalViewController) should be presented asking the player to type their name.
- After typing their name and confirming, the LobbyViewController should be presented, showing the player that they are in a lobby.
- When pressing the 'Settings' button, a modal (SettingsModalViewController) should be presented.

Lobby

- When a player creates a lobby, their name should be at the top of the player list.
- When a player joins an existing lobby, their name should be added to the player list below the existing players.
- When toggling the options in settings, the SegmentedControl should update based on where the player presses.
- If the player is the host of the lobby, the start button should be visible. Else, the start button is not visible.
- If there are insufficient players (< 2) in the lobby, the start button should be disabled for the host.

Game

- Tapping on the screen should create a node for the player.
- Panning the screen should move the map.

- Pinching the screen should zoom in and out of the map with a minimum xScale of 0.5 and maximum xScale of 1.5.

Contributions

Name	Contributions

Adam Chew (Project Manager)	- Implement “ECS” architecture
	- Setup Github Action
	- Implement edge linking between nodes
	- Implement node upgrading
	- Implement UI elements for
	  - Edge linking
	  - Node upgrading
	- Implement terrain

Foo Guo Wei	- Implement game logic - Connect game presentation with game logic - Implement minimap - Implement fog-of-war

John Phua	- Implement networking package
	  - Link up lobby updates
	  - Link up game actions across players
	- Implement anonymous authentication to identify players
	- Refine GitHub Action configuration
	- Implement rendering for
	  - Different node types
	  - Different players
	  - Health bars for nodes
	- Handle networking for leaving game

Tay Yu Jia	- Implement models and logic for the main screen and lobby
	- Implement game features
	  - Combat resolution
	  - Game actions
	  - Serialization of entities for game actions
	- Implement rendering for
	  - Different node types
	  - Display resource count for player

Project Schedule

Sprint	Tasks	Assignee

1	Create entities	Adam
	Create components	Adam
	Implement resolvers	Guo Wei
	Implement game loop	Guo Wei
	Implement gestures in game	Yu Jia
	Sync lobby updates through networking	John
	Provide a thread-safe buffer to sync game actions across players	John
	Implement anonymous authentication for player identification in application	John
	Create models and view controller for lobby	Yu Jia
	Implement view controllers for main screen	Yu Jia
	Set up GitHub Actions CI	Adam & John
2	Implement game tick	Yu Jia
	Implement logical coordinates (sync with zoom and pan of map)	Guo Wei
	Implement leave game functionality	John
	Create entities and components for Combat resolution	Yu Jia
	Implement resolvers for Combat resolution	Yu Jia
	Implement distance restriction for building own nodes	Adam
	Implement upgrading of nodes	Adam
	Rendering: Differentiate different player nodes	John
	Rendering: Differentiate different node types	John
	Rendering: Select between different types of node	Yu Jia
	Rendering: Display links between nodes	Adam
	Rendering: Display HP for nodes	John
	Rendering: Display resource count for player	Yu Jia

	Implement minimap	Guo Wei
3	Implement fog-of-war	Guo Wei
	Add different attacking behavior for combat nodes	Yu Jia
	Switch to dedicated message queue for networking	John
	Fire projectiles when attacking	John
	Terrain generation	Adam
	Add custom effects for terrain tiles	Adam
	Refactor Game + ECS	Guo Wei
	Refactor NetworkComponent	Guo Wei
	Make Minimap part of EC-System	Guo Wei
	Refactor BuildNodeAction	Yu Jia
	Implement custom decoding of entities in GameActions	Yu Jia

GUI sketches/screenshot

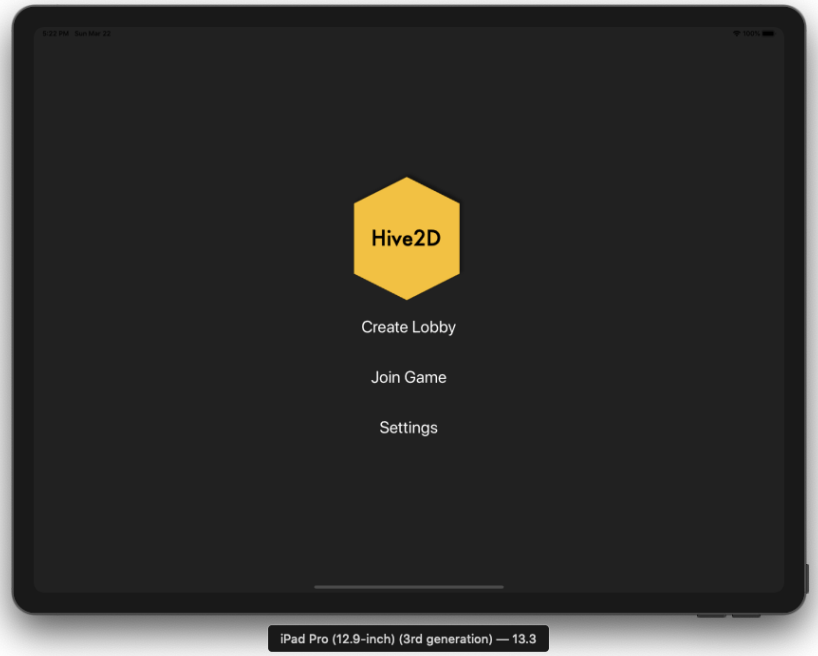


Figure 9: Hive2D main screen

Figure 10: Hive2D lobby

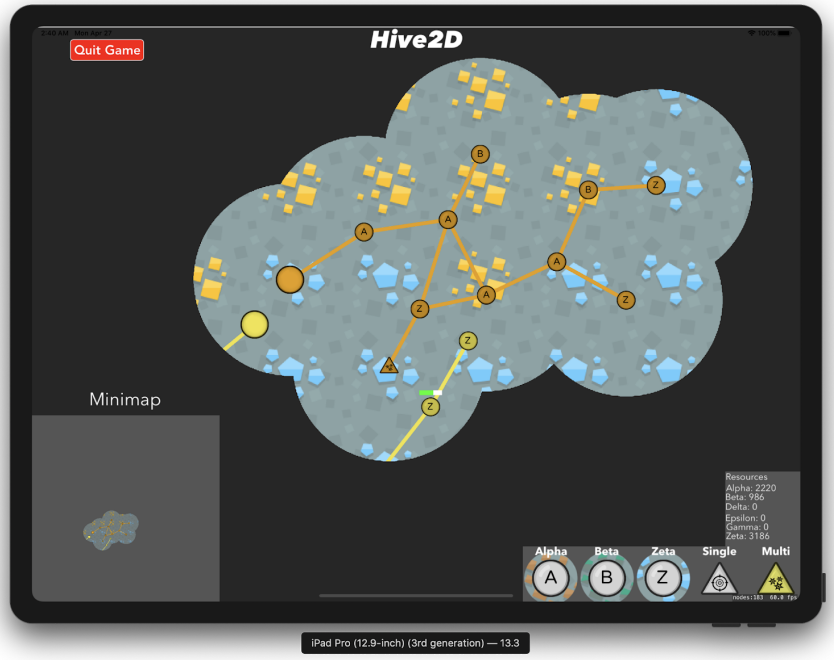


Figure 11: Hive2D game

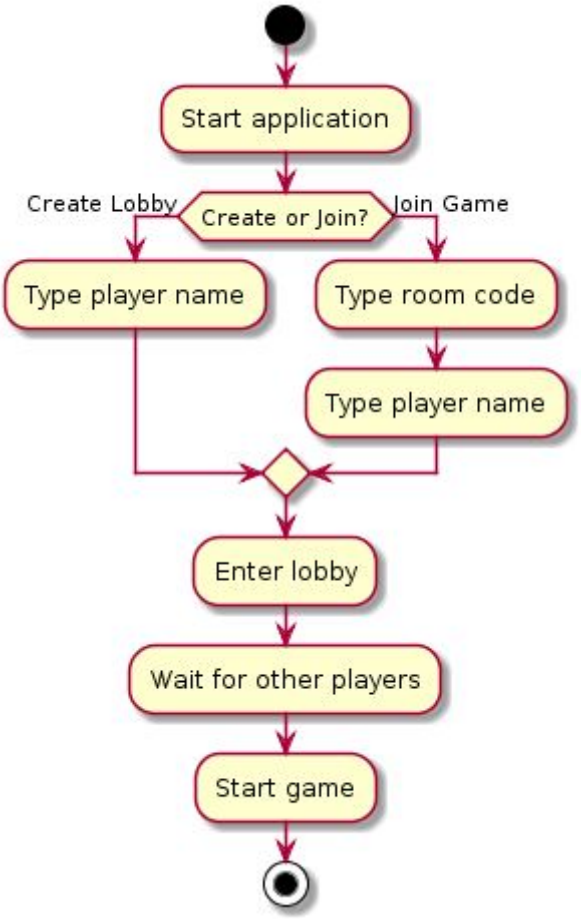
User Manual

Version 1.0.0

Device requirements

- iOS 13.2 (iPad only)

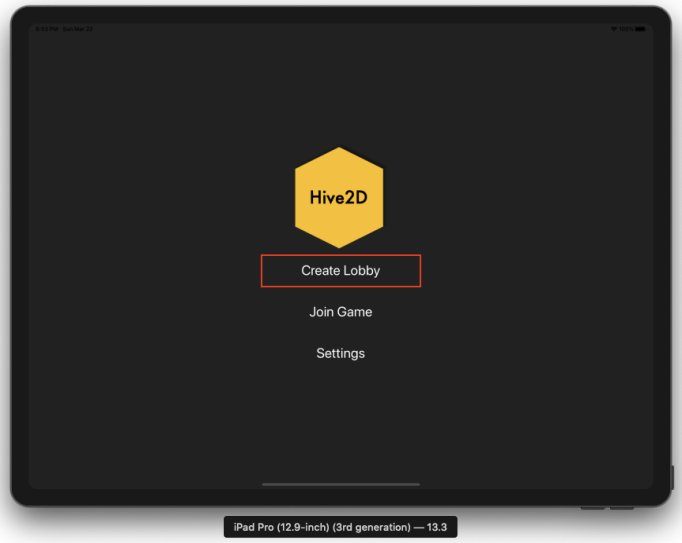
User flow for application



To start playing Hive2D, you will have to first match with other players by creating your own lobby or joining others' lobby. Please follow the instructions, accompanied with screenshots, to ensure that you can enjoy the game to its fullest!

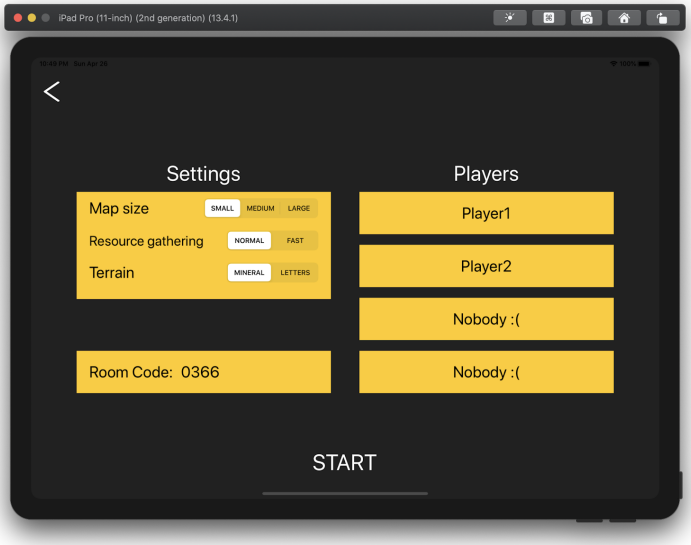
i. Creating a Lobby

1. On the main screen, press the ‘Create Lobby’ button.



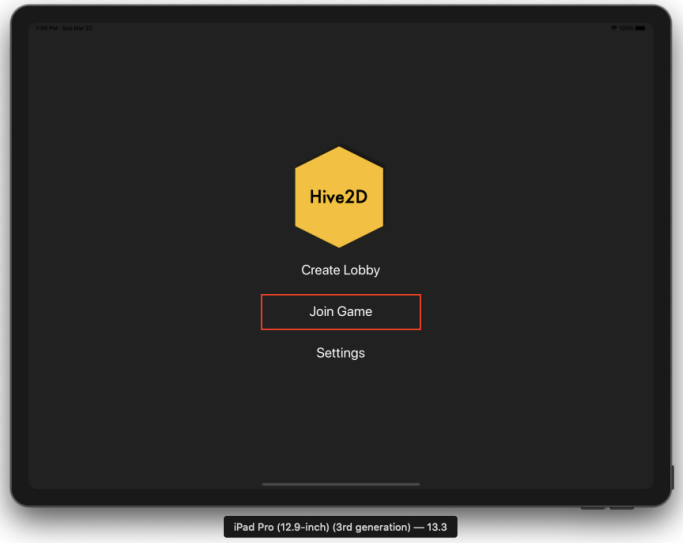
2. Type in a preferred display name for your player.

3. You should see the following screen. To invite other players, share with them the room code. Also, you may toggle settings for map size, resource gathering rate, and terrain. As the host, you are the only player who can see the start button to start the game.



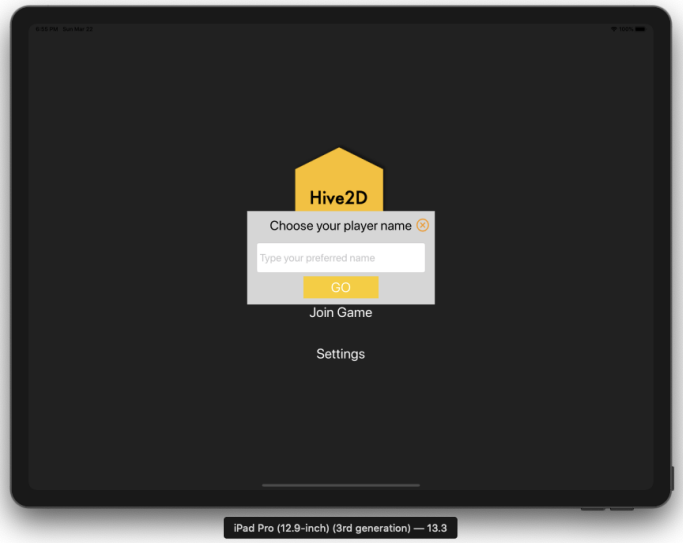
ii. Joining a Lobby

1. On the main screen, press the ‘Join Game’ button.



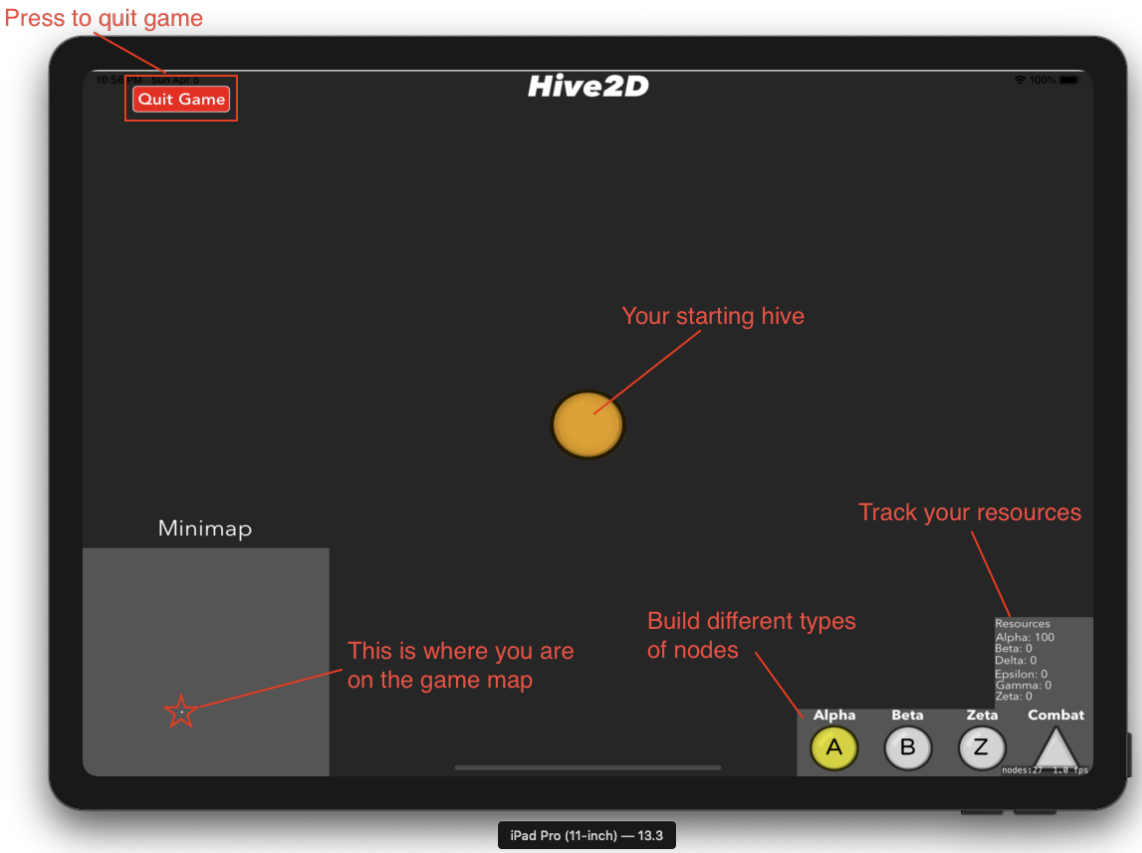
2. Fill in the room code here.

3. Type in a preferred display name for your player.



4. You should see the following screen.

iii. Understanding the game interface



iv. Building your very first node!

1. Tap on a point near your hive to build a node.

As you may realise, there is a restriction on how far you may build nodes. This is to ensure that your nodes are always connected to your hive so they can send resources back to you.

There are different types of nodes that you can build -- ResourceNodes and CombatNodes. ResourceNodes help you to gather resources and they are specified by the letter shown on them. For example, the first node built in the screenshot is an Alpha ResourceNode that generates Alpha resources. *Hint: Building resource generating nodes on their special terrain tiles will also boost the collection rate!* On the other hand, CombatNodes are your main assault forces for taking down enemies. They are capable of attacking enemy nodes within a certain range; when the health of nodes drop to zero, they get destroyed so be careful!

Below are the types of nodes that our game currently supports!

Node	Alpha ResourceNode	Beta ResourceNode	Zeta ResourceNode	Single Target CombatNode	Multi Target CombatNode
Asset					

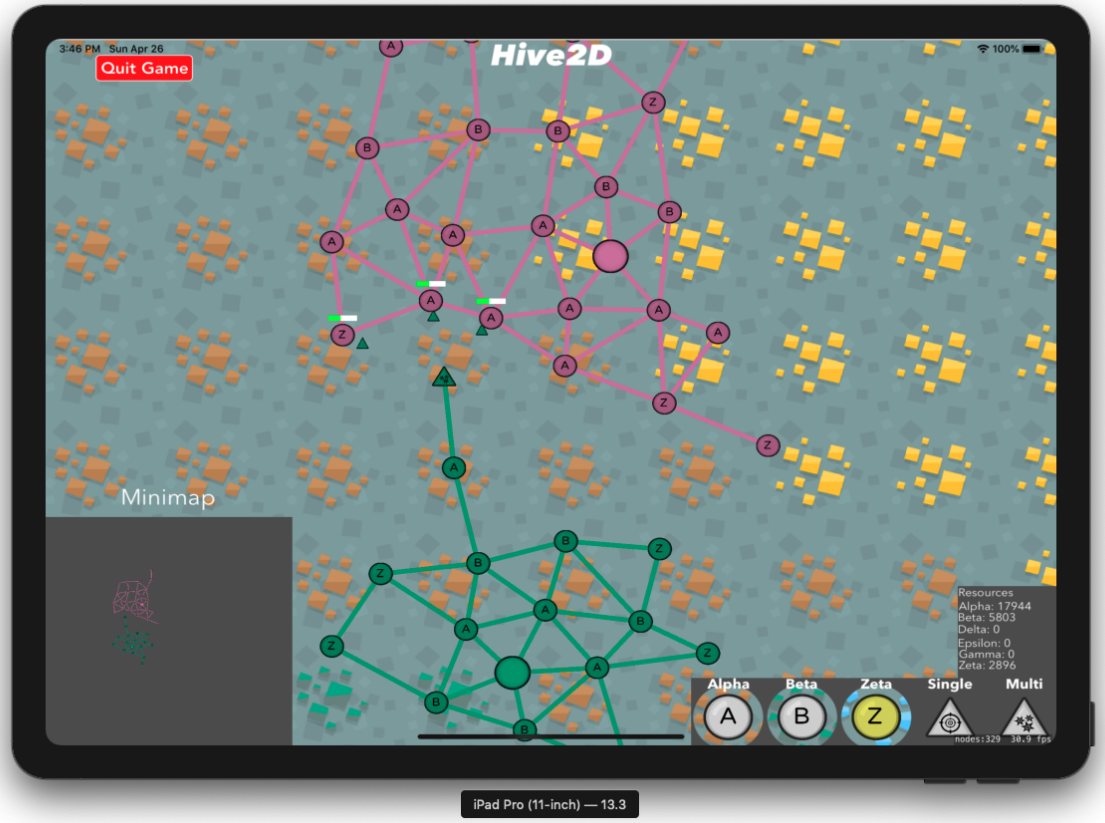
Each type of node also requires different costs to build them. This is depicted in the cost matrix shown in the table below.

Node Type	Resources required		
	Alpha	Beta	Zeta
Alpha ResourceNode	30	0	0
Beta ResourceNode	50	0	0
Zeta ResourceNode	30	30	0
Single Target CombatNode	0	0	50
Multi Target CombatNode	20	30	40

v. Upgrade your node to improve abilities

You can also upgrade a resource node by clicking on it and clicking ‘Upgrade’. The cost of the upgrade is displayed before you decide to upgrade a node. Every node can be upgraded a limited number of times. Each time a node is upgraded, it receives a boost in its resource collection rate, allowing you to farm resources more quickly and build more nodes in the long run.

v. Engage in combat with opponents



In the screenshot above, green nodes refer to you and pink refer to an opponent. By placing a CombatNode near to your opponent's nodes, you will inflict damage onto them by firing projectiles (the health of a node is shown by the green bar). After the health drops to zero, the node is considered destroyed and will disappear.

vi. Terrain effects

The game currently supports 2 types of terrain, each with different tile sets and varying tile effects. Building resource nodes on certain tiles on the terrain will grant bonuses according to those tiles' effects. For example, for the terrain Mineral, building an Alpha node (which generate Alpha resources) on a Copper tile will boost the resource collection rate of that node. On the other hand, building any node on Silver or Gold tiles will boost its health regeneration, regardless of whether it matches the tile's resource type.

Mineral Terrain				
Tile	ResourceType	Buildable	Tile Effects	
			Boost Resource Collection rate 2x if node matches tile resource type	Boost Health Regen 2x
Copper	Alpha	✓	✓	✗
Iron	Beta	✓	✓	✗
Ruby	Delta	✓	✗	✗
Silver	Epsilon	✓	✗	✓
Gold	Gamma	✓	✓	✓
Diamond	Zeta	✓	✓	✗
Lava	-	✗	✗	✗

And for terrain Letter:

Letter Terrain				
Tile	ResourceType	Buildable	Tile Effects	
			Boost Resource Collection rate 2x if node matches tile resource type	Boost Health Regen 2x
A	Alpha	✓	✓	✗

B	Beta	✓	✓	✗
D	Delta	✓	✗	✗
E	Epsilon	✓	✗	✓
G	Gamma	✓	✓	✓
Z	Zeta	✓	✓	✗
Wall	-	✗	✗	✗

vii. Zooming in/out of the map

Pinch inwards to zoom out and outwards to zoom in. This should change the size of nodes shown accordingly.

viii. Panning the map

Slide your fingers in a panning motion across the screen to move the map.

ix. Quit the game

Press the quit game button at the top left corner and you should return to the main screen.