GooReat Restaurant

Final Report

By

			
Anthony	Gabriel	Hui Qi	Samuel

Requirements

Overview

Game Name

Gooreat Restaurant - A plate of Goo with a Serving Slime on the Side in Space

Game Concept

Gooreat Restaurant is a (co-op) game that takes place in the galaxy, where the players navigate their spaceship around to cook and spread the love of food all around the galaxy!

Tanye is a slime that has been inspired by food dishes ever since he was just a small blob.
He bought his own spaceship when he grew up and started this fantastic adventure of spreading his food creations all around in the galaxy.

Game Description

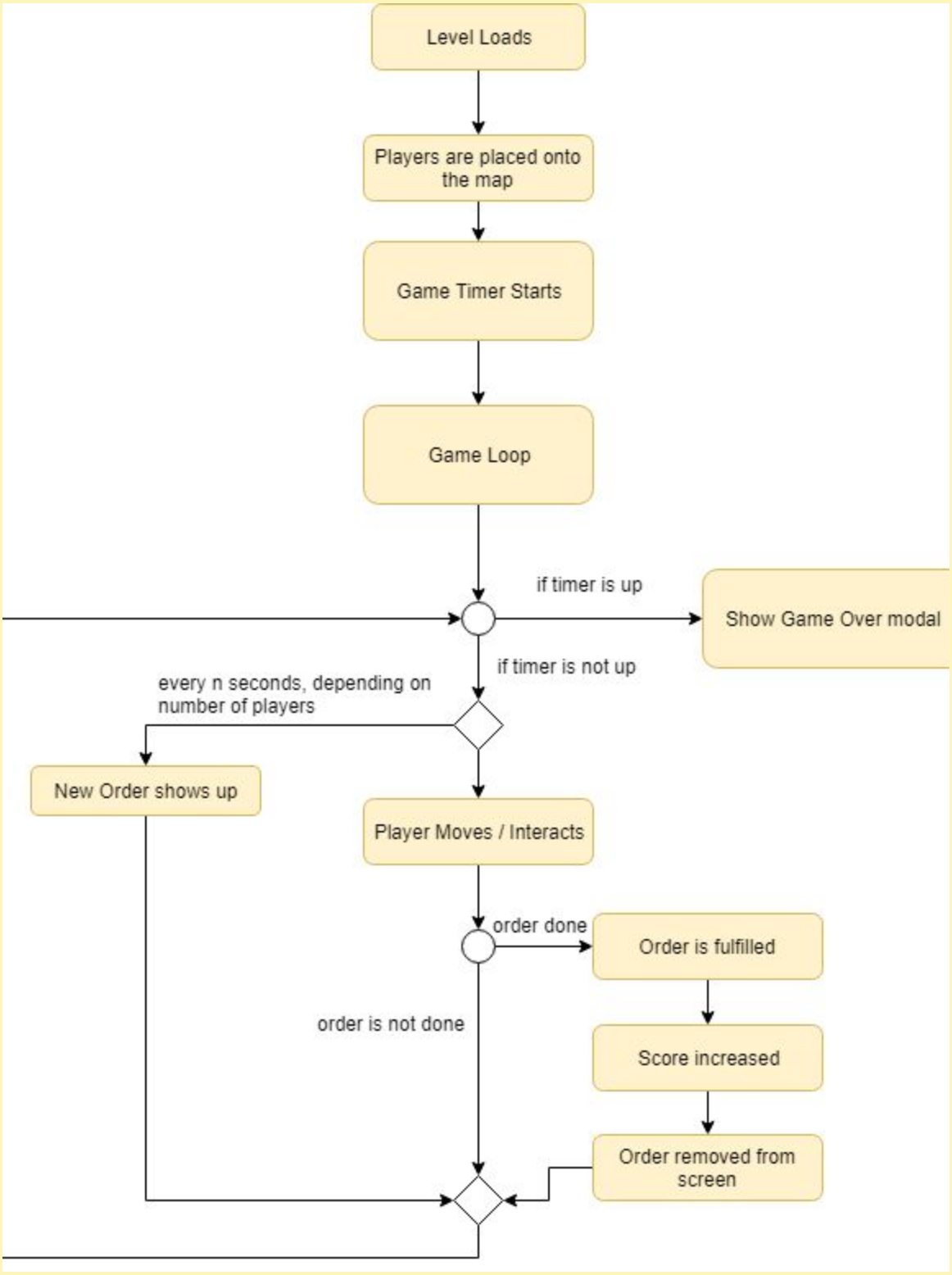
Gooreat Restaurant is a cooking simulation game, whereby players are required to complete the dishes and orders as fast as possible within the time limit.
Players are able to play in single-player mode or multiplayer with their friends.
Play alone to get rid of some boredom, play together with your friends to have a whole new level of excitement!

Game Mechanics

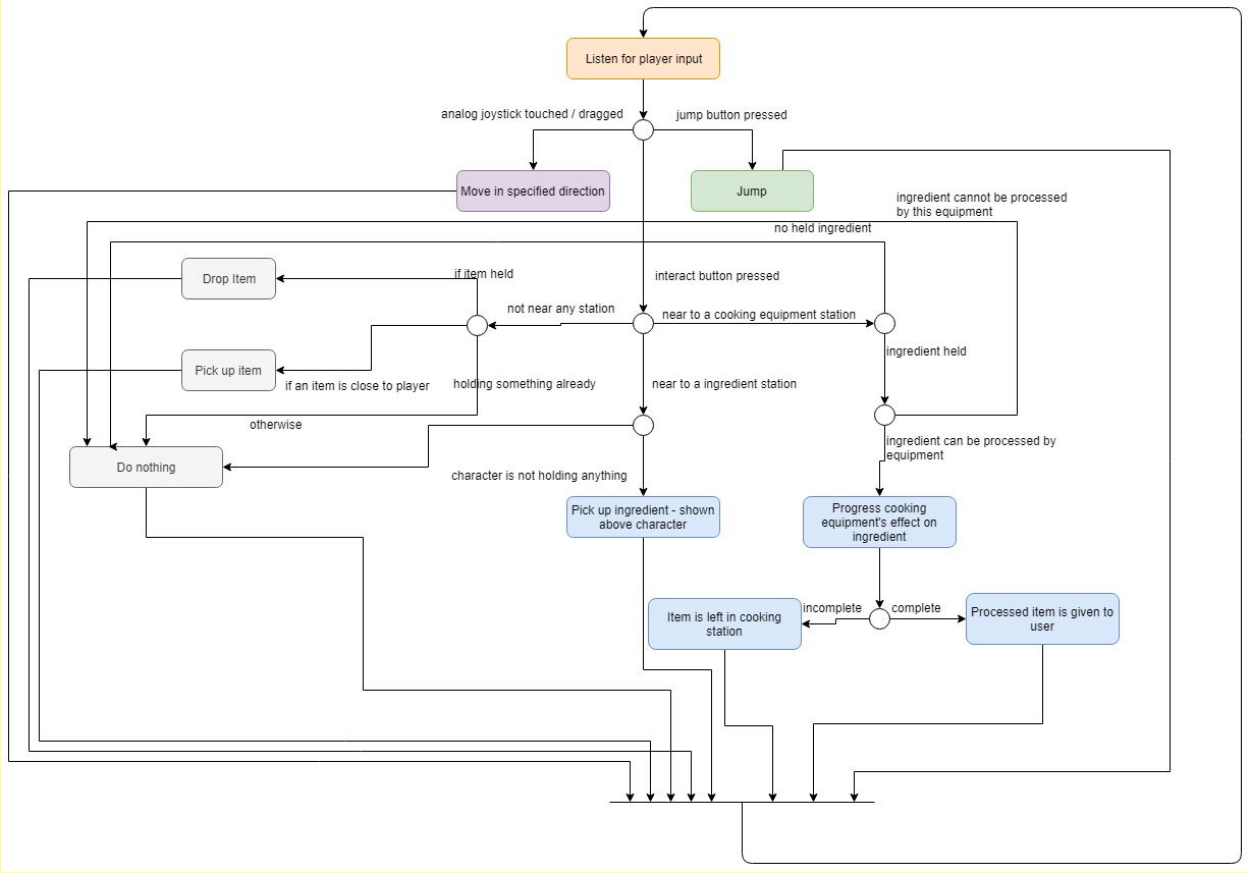
The main mechanic of the game involves moving the player around to make food that customers have ordered. The player moves by controlling an on-screen analog controller, and has to complete a specific set of actions to complete the order, which is shown on the top of the screen.

Game Flow

An overview of the game flow can be seen below.



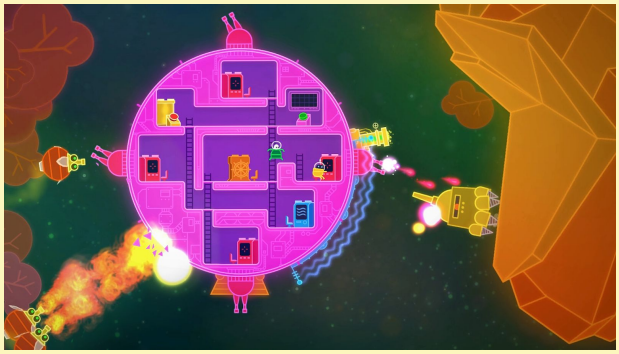
The flow of how the user's actions should be perceived is shown below:



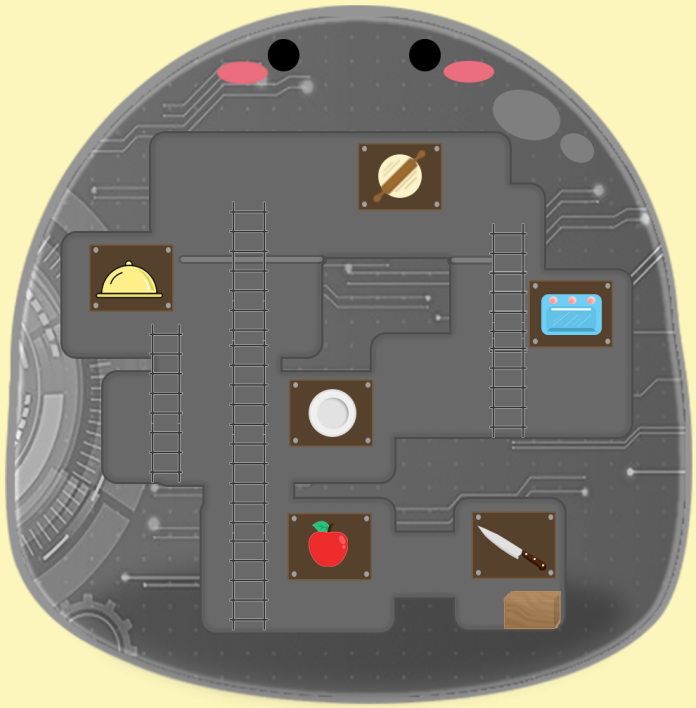
Game Inspiration and Breakdown

The game idea inspiration came from 2 games - *Overcooked* and *Lovers in a Dangerous Spacetime*.

These are some screenshots to refer to for *Overcooked* and *Lovers in a Dangerous Spacetime* respectively.



Our game idea revolves around the idea of having a co-op platformer where achieving the key objective (fulfilling as many orders as possible, as fast as possible) is made simpler as more people join, but at the same time, can lead to more chaotic situations should the players not be able to cooperate well. This game idea is similar to both *Overcooked!* and *Lovers in a Dangerous Spacetime*, where cooperation is rewarded handsomely. The image below depicts a more accurate version of our game level design.

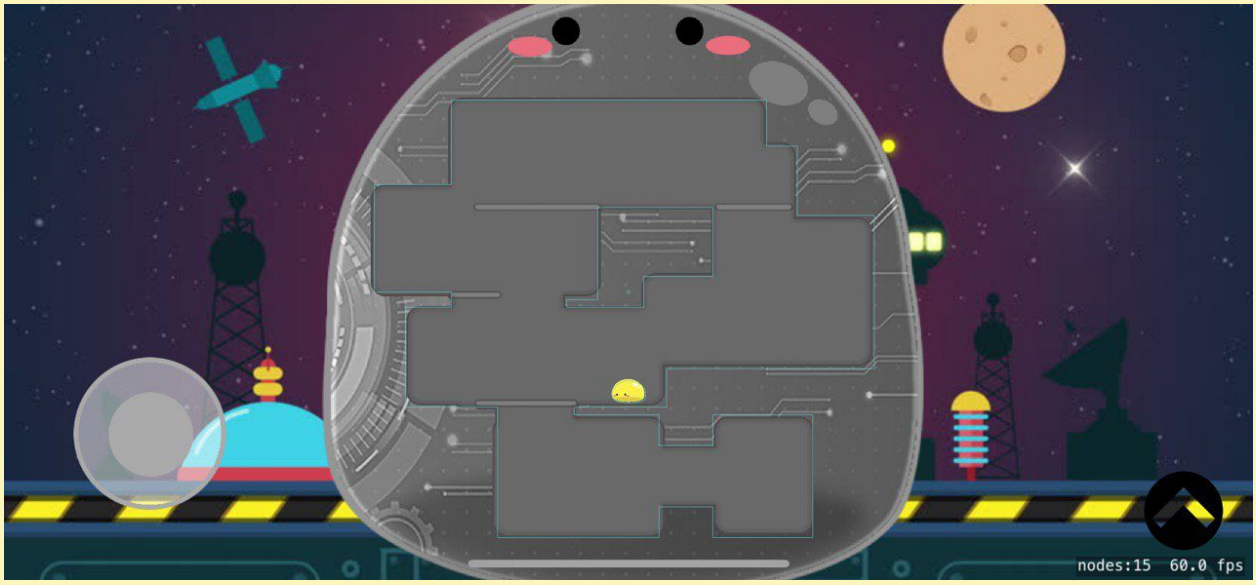


As seen from the level design above, it is visually similar to that of *Lovers in a Dangerous Spacetime*, whereby there are different stations located around.

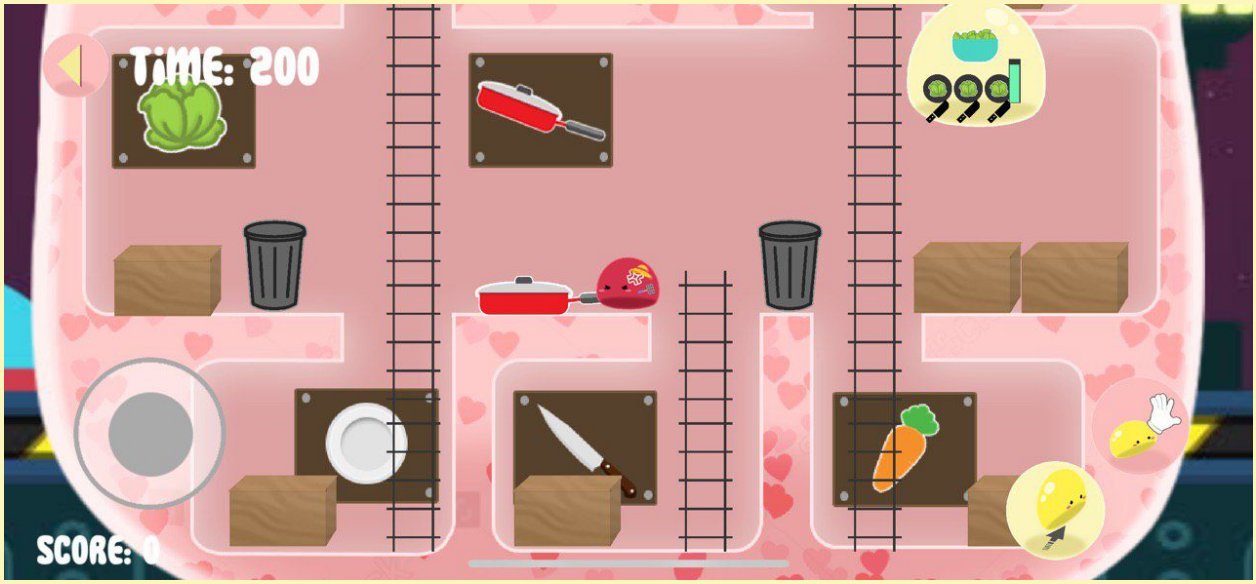
However, just like *Overcooked*, there are different stations whereby players are able to do different actions such as chop and cook etc.

Dynamic Camera

The initial look of the game play was that the one view will be able to see the whole scene.



The new look is a dynamic camera, whereby the camera is now much more zoomed in, and the camera will follow wherever the player.

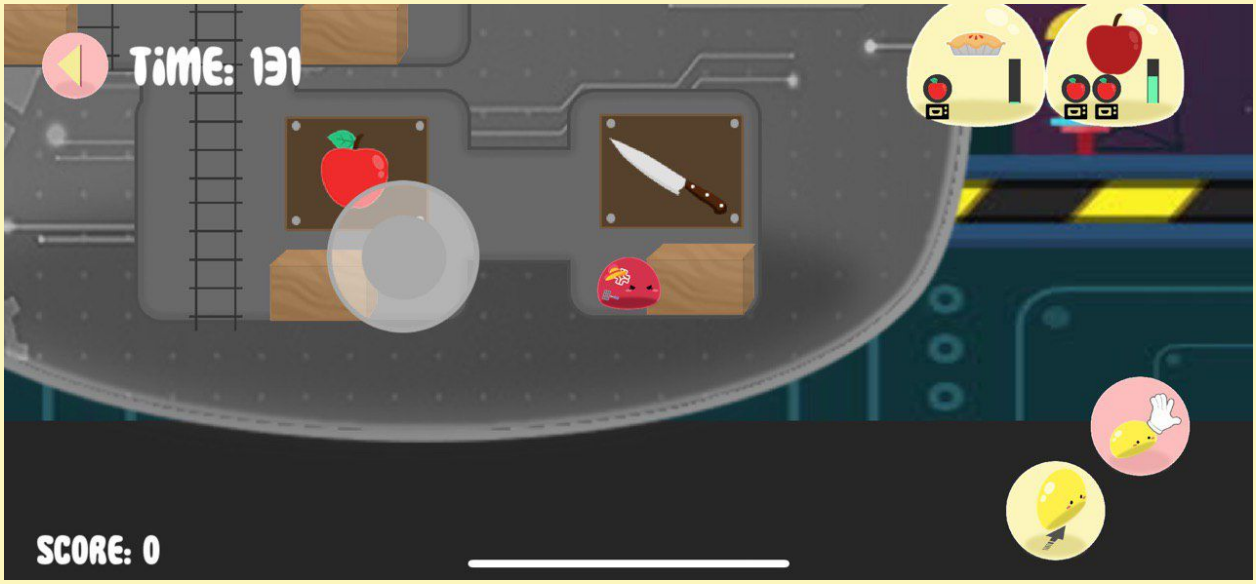


This gives more gamified look and feel, and we are able to expand the whole game field much more since we are not just limited to the small screen.

Directional Pad/Analog Joystick

The Joystick is located at the bottom left of the screen. Upon pressing down on the joystick, nothing will happen until the player holds and drags the joystick to the different direction. Based on the direction that player is dragging, the sprite will move accordingly.

The analog joystick will change position depending on where the player is tapping, as long as the tapping position is still within the left area of the screen.



Jumping button

The jump button will be located at the bottom right of the screen. Upon tapping on the jump button, the sprite will be activated to start jumping around. Holding the button will not have any continuous effect for jumping right now. We will be doing more testing to see whether does this affect the UX.

If both the joystick and jump is being activated at the same time (i.e dragging the joystick and pressing the jump button), players can expect the sprite to be jumping into the direction the player has dragged.

Interact button

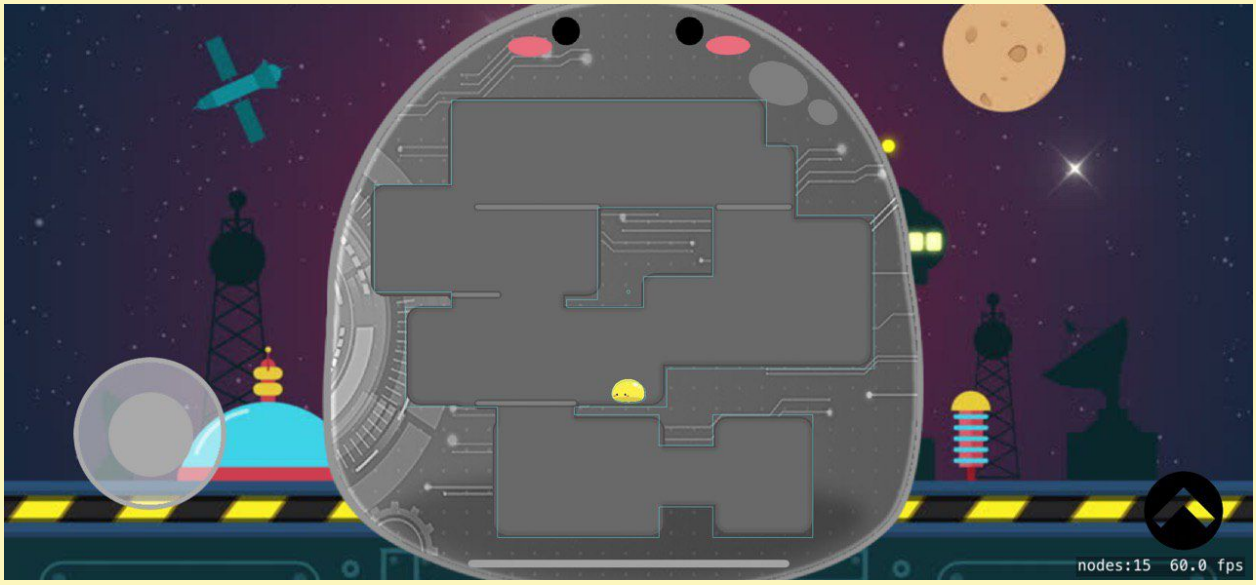
The interact button will be located at the bottom right of the screen, along with the jump button. This interact button allows users to interact with the stations. Upon interacting with the stations, the users are able to pick up objects such as the ingredients and/or plates. Eventually, they will need to pick them up to the serving counter.

Other than picking them up, the same button is being used to do actions such as chopping (basically actions that are related to food processing). Depending on the station, each station will require a certain number of interactions to process the food. For instance, each chopping is 20% completion of the food, so users are required to tap the interact button 5 times in order to complete the chopping of the food.

We do not include an additional button here for processing of food as it is redundant to add one more additional button for that at the moment. However, depending on our user testing and feedback, these buttons may still be updated accordingly to the needs of the users and suitability.

Bounding areas

We will look at a snapshot of the game to explain the bounding areas.



The bounding area/physics body is indicated by the light blue outline on the different areas.

Slime

As seen from the above diagram, the slime is currently enveloped by a collider that is of the same shape and the size itself. However, this is being generated by the code such that the physics body is generated based on the texture image, which can cause a bit more inefficiency as compared to just creating a circle or rectangle physics body.
Though, in XCode, physics body are created at the start of level and it will not change during the rest of the game unless a script is made to change the physics body dynamically, which is very inefficient.
When the game is more structured and implemented, more testing will be done to see if the physics body of the slime affects the efficiency and FPS of the game.

Playing area border

As seen from the above diagram, playing area is being enveloped by the blue border as well. However, in this case, this is not a **physics body**. Before we go on further, let's look at the the physics bodies available stated by the Apple's Developer Guide.

- Dynamic volume: simulates a physical object with volume and mass that can be affected by forces and collisions in the system. Use dynamic volumes to represent in the scene that need to move around and collide with each other.
- Static volume: similar to a dynamic volume, but its velocity is ignored and it is unaffected by forces or collisions. However, because it still has volume, other objects can bounce off it or interact with it. Use static volumes to represent items that take up space in the scene, but that should not be moved by the simulation. For example, you might use static volumes to represent the walls of a maze. While it is useful to think of static and dynamic volumes as distinct entities, in practice these are two different modes you can apply to any volume-based physics body. This can be useful because you can selectively enable or disable effects for a body.
- Edge: An *edge* is a static volume-less body. Edges are never moved by the simulation and their mass doesn't matter. Edges are used to represent negative space within a scene (such as a hollow spot inside another entity) or an uncrossable, invisibly thin boundary. For example, edges are frequently used to represent the boundaries of your scene. The main difference between an edge and a volume is that an edge permits movement inside its own boundaries, while a volume is considered a solid object. If edges are moved through other means, they only interact with volumes, not with other edges.

Thus, the one we are using here for the area is the *edge*, the static volume-less body, as we just need it to be a boundary and a space to contain the sprite, preventing it for exiting the boundary. Hence, we do not use physics body for this case.

Interactable objects

The interactable objects will be items such as the tables and more items if necessary. These interactable objects will be containing a physics body as well for the sprite and these items to be in contact with each other. Currently, the interactable objects that we have are:

1. Tables (stations)
2. Ingredients
3. Ladders
4. Playing area

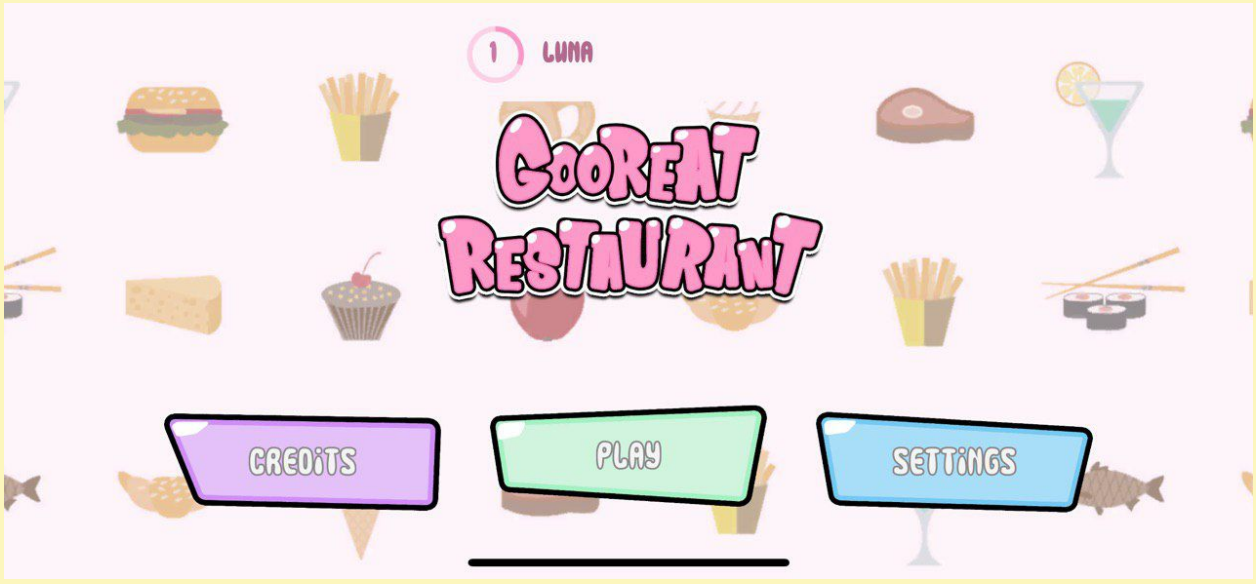
Customization

Players can now customize their name and their slime colour at the start of the game (the first time they have downloaded the game). This serves as an additional feature to our game, but not to the game logic for now.

User Manual

For this application, it can only be run on iOS devices such as the iPhone and iPad.

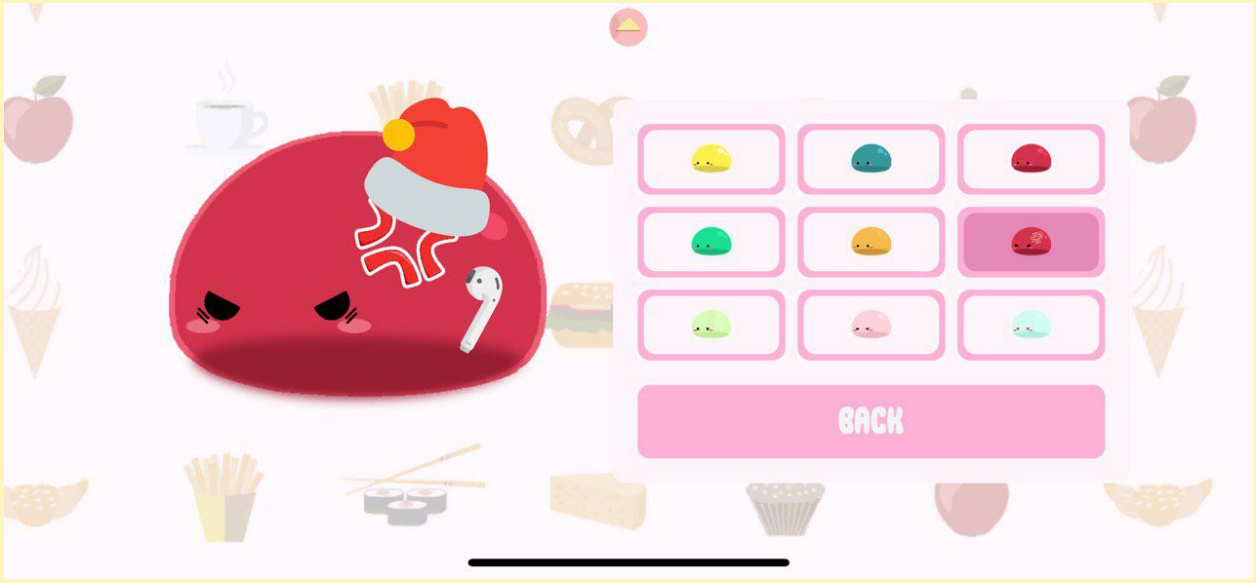
Game Features



The image above shows the main screen, which is also the same screen players will see when they enter the game.

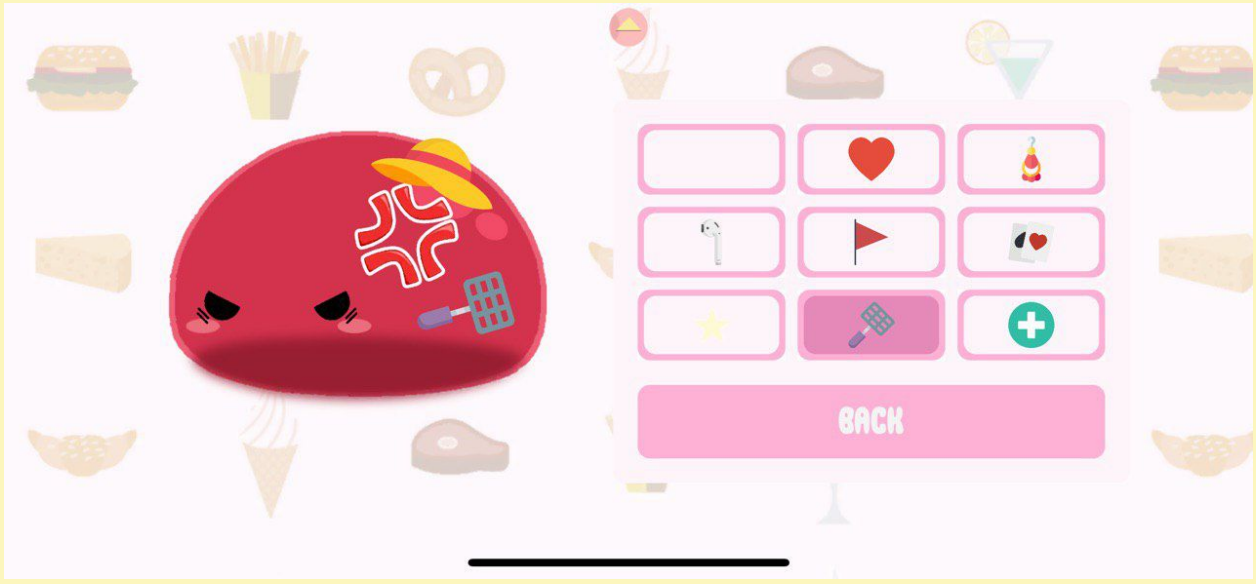
Customization

Players are able to navigate to the customization from the main screen, by clicking on the name (button on the top center of the screen). This allows users to customize the colour of the slime, hat and accessories as seen from the next few images.



Slime customization of colour.

Slime customization of hat.

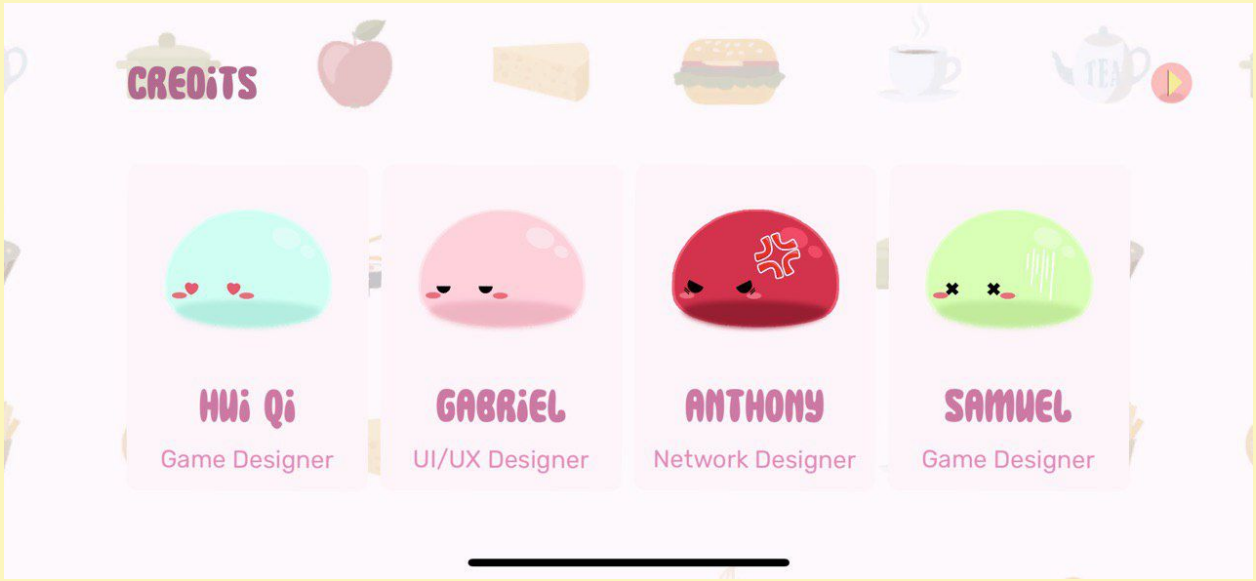


Slime customization of accessory.

Reset Data

Players can choose to reset the data by clicking on the settings button in the main screen. This will reset all the progress, level, experience and the slime's look, as well as all saved scores.

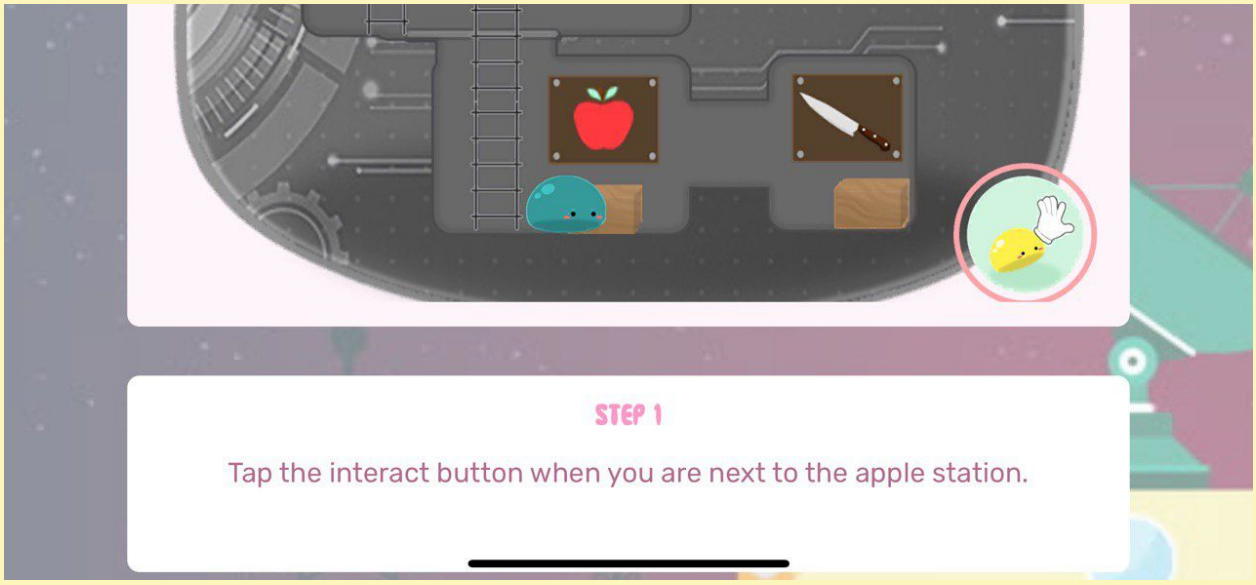
Credits

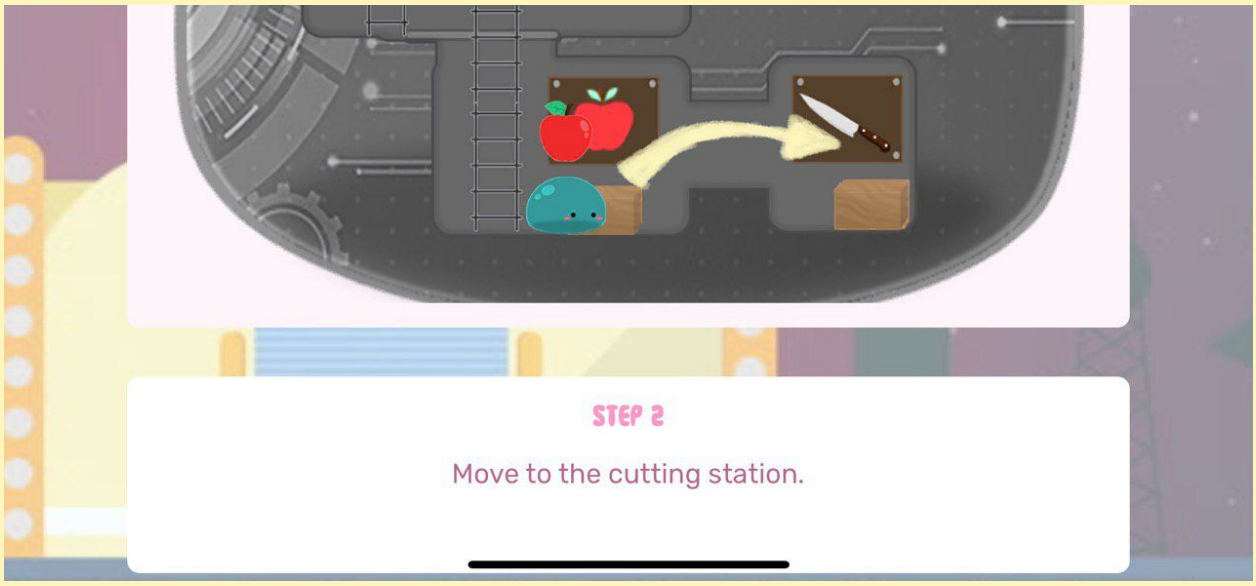


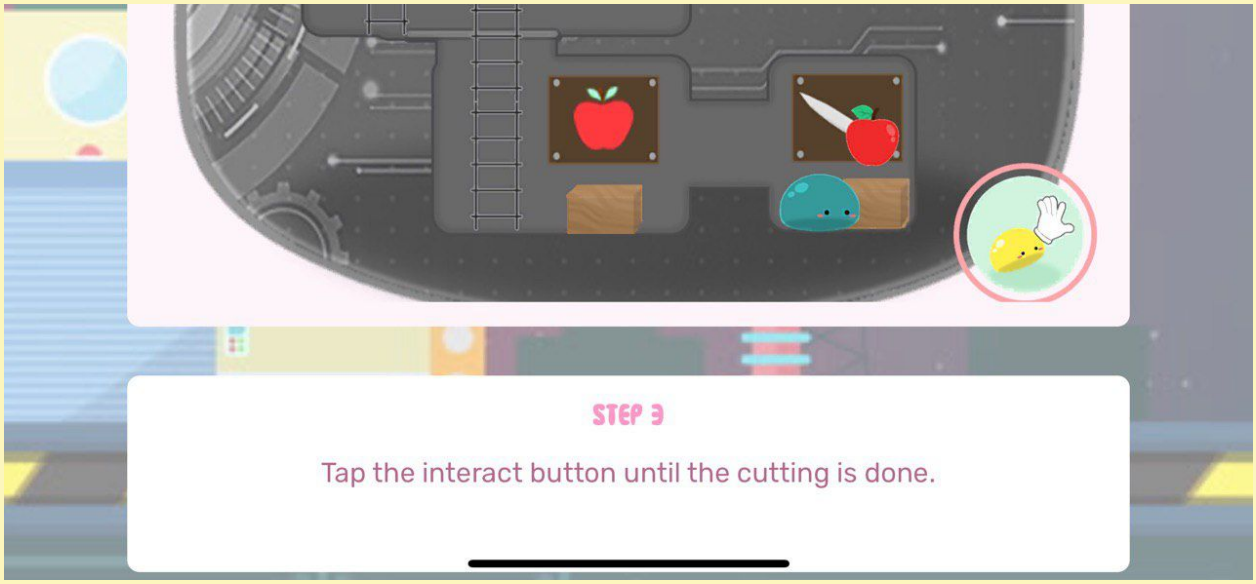
Players can view the credits via the credits button on the main screen.

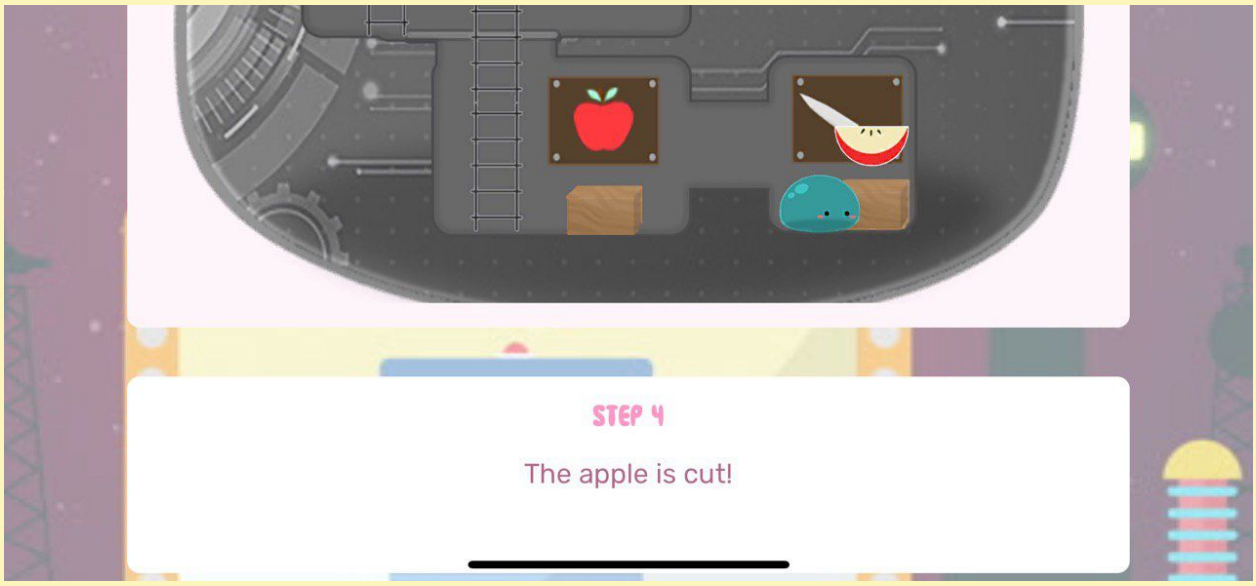
Tutorial

There is a short tutorial before the start of every game, to assist in the players to understand how to navigate and use the interaction button, so they will be able to catch on quickly if it's their first time playing the game.
The images for the tutorial can be seen below:









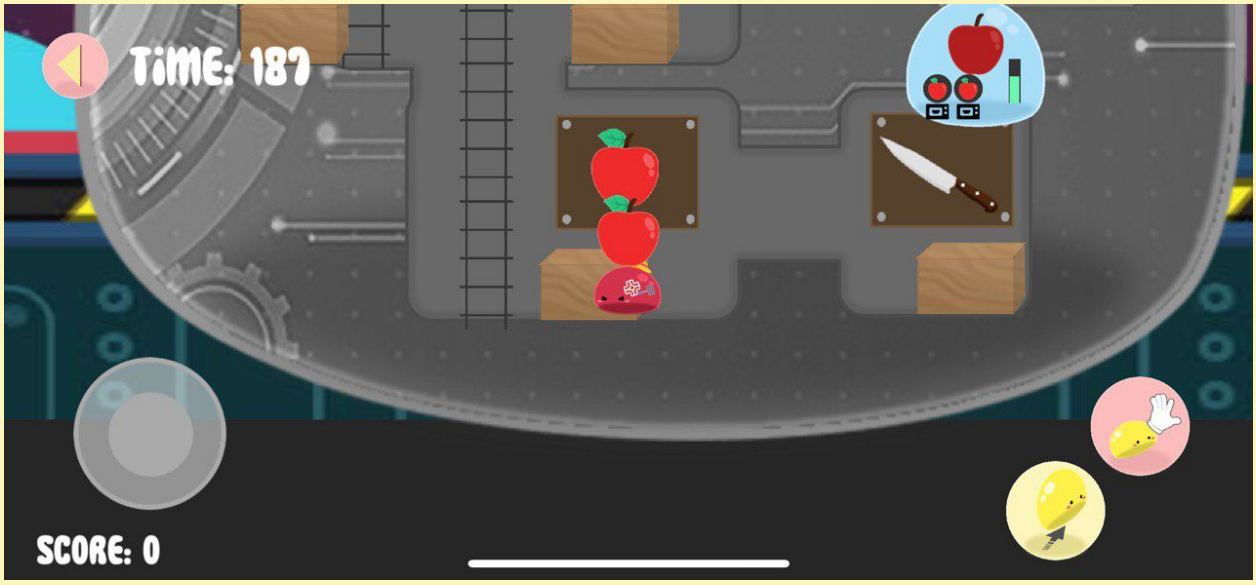
Preparation time for players

There is a short preparation time when the player enters the stage, to get them familiarise with their supports, and to improve the user experience. The preparation time is depicted as below:

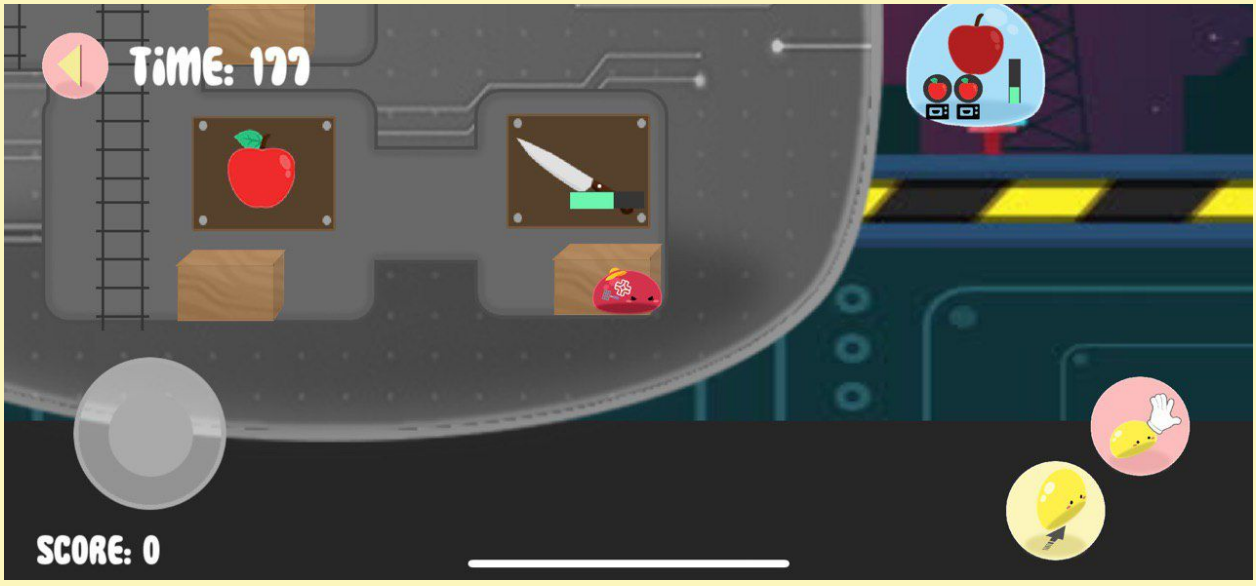
Preparation of food

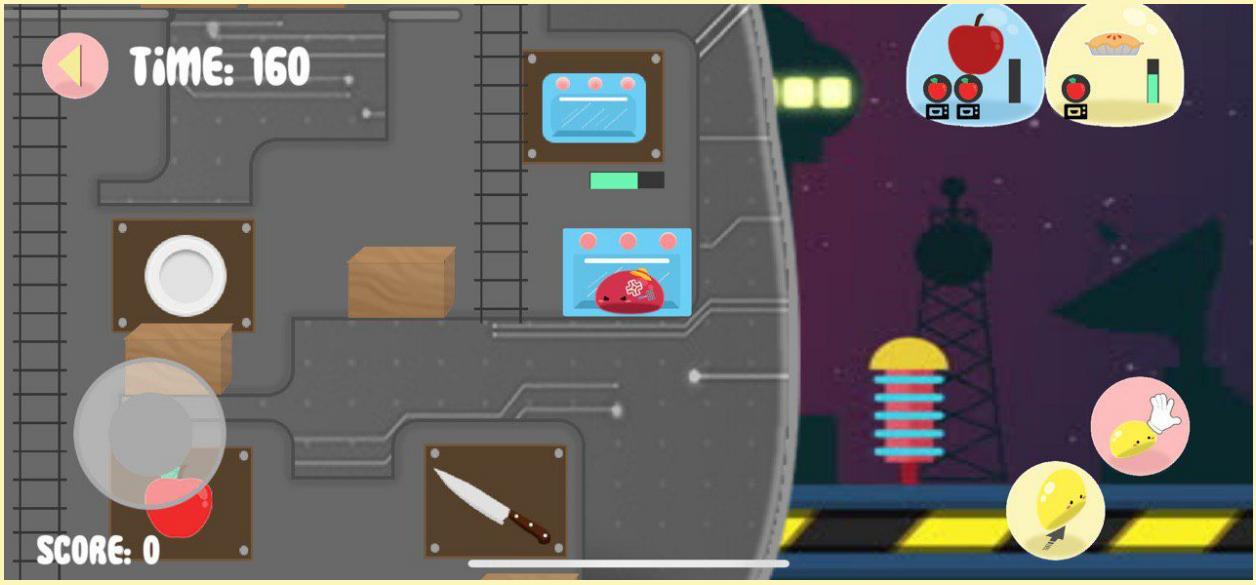
To prepare food, the players have to follow the orders that are currently being displayed at the top, right hand corner of the screen.
As seen in the menu orders, it also displays the ingredients required, and the method of cooking needed for that specific ingredient. The green bar shows the amount time left for the food. The more time that is left, the higher points the player will get when they serve the correct food.

Players will then need to go to the Ingredient Storages to pick up the food:

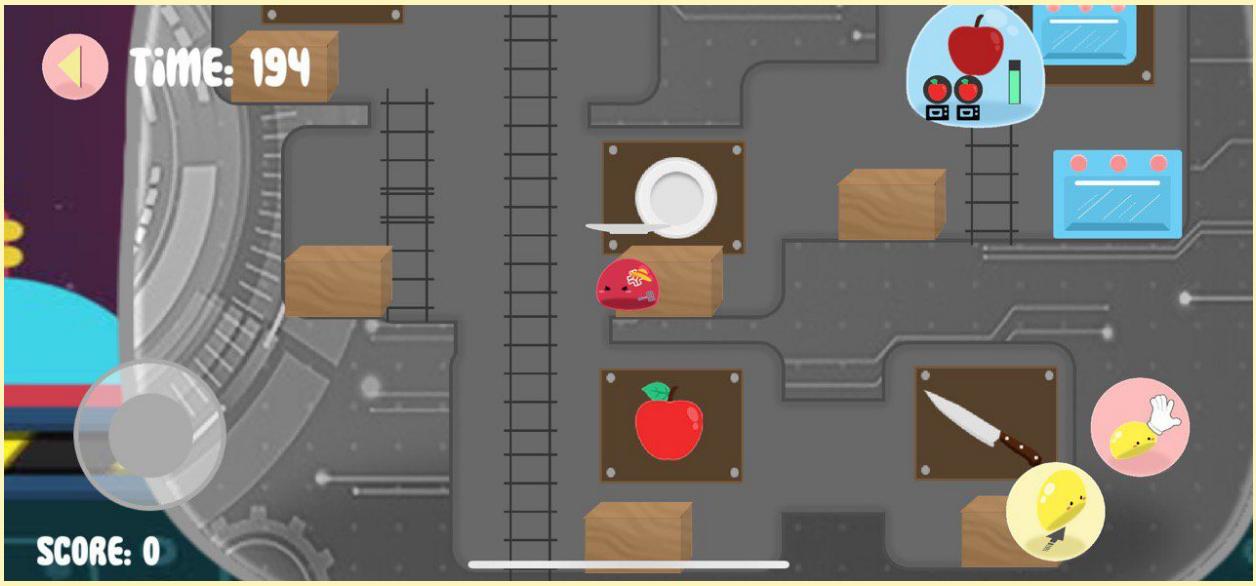


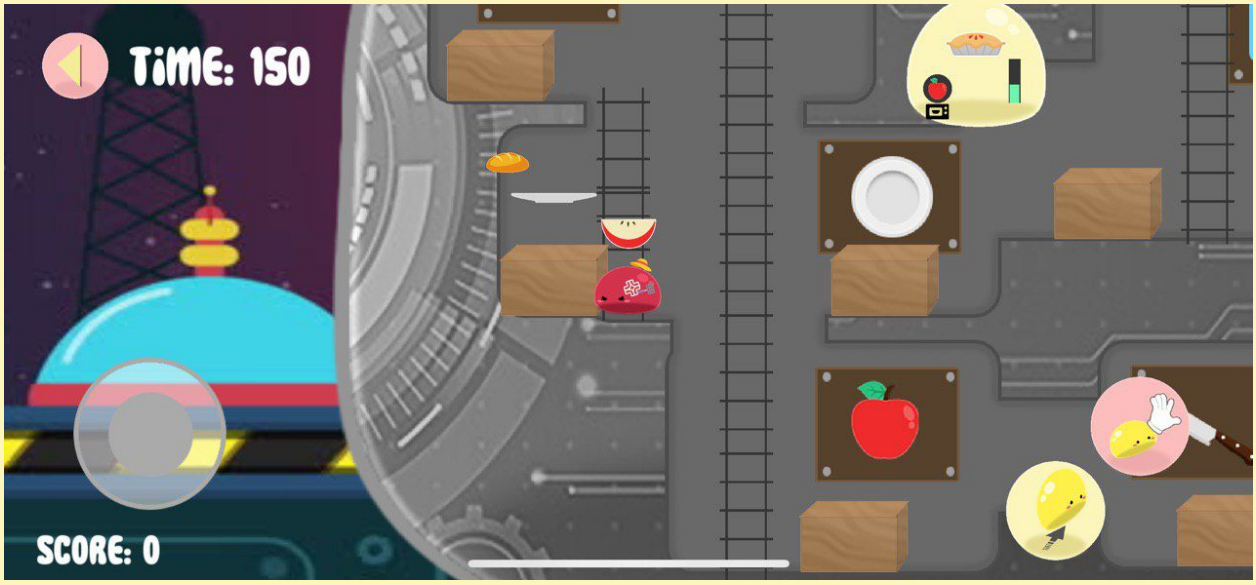
All the food has to be processed by a cooking method, either chopping, baking or even frying. There will be a progress bar to indicate the amount of completion for the processing.
For chopping, players are required to tap the interact button **multiple times** till the ingredient is finished processing.
For other processing methods such as baking or frying, players will have to put the ingredients inside and wait for a couple of seconds before the food is finished.





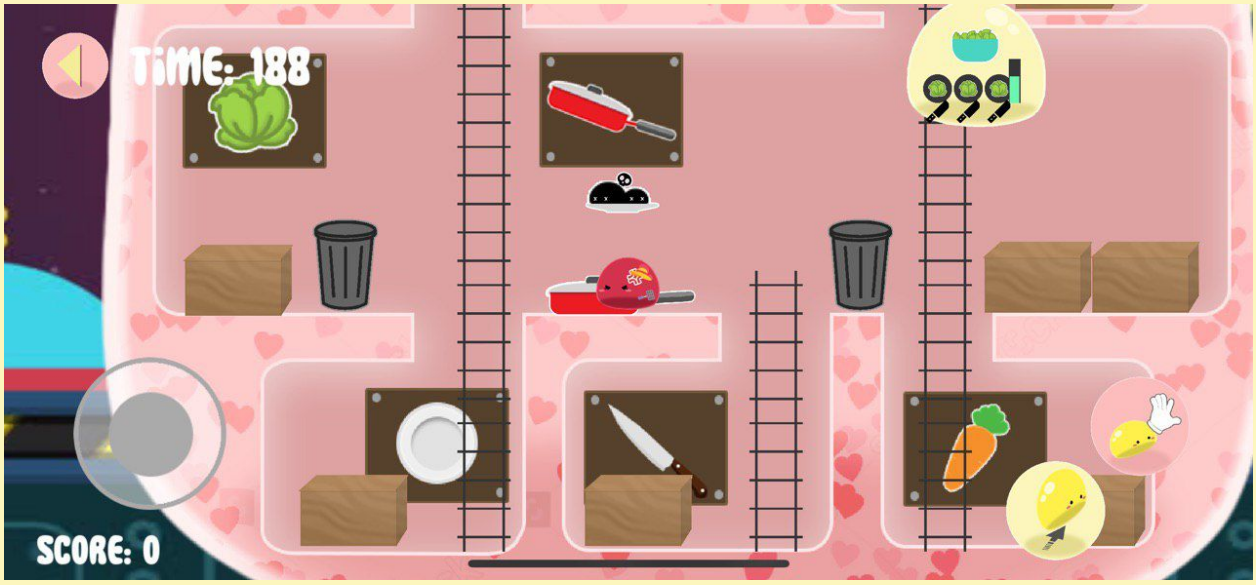
Players will have to put the food ingredients on a plate:



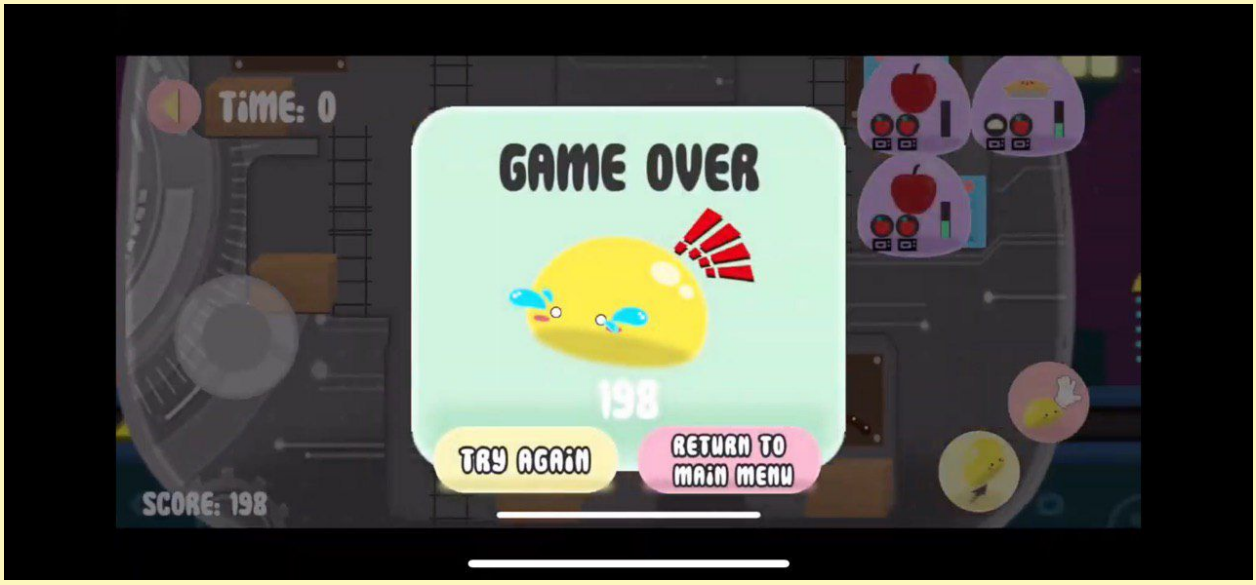


And when the recipe is done, they can go ahead and serve the food!

When the player cooks something that should not be cooked, he will get “junk”, which cannot be used for anything. For example, trying to fry lettuce will result in getting the junk item:



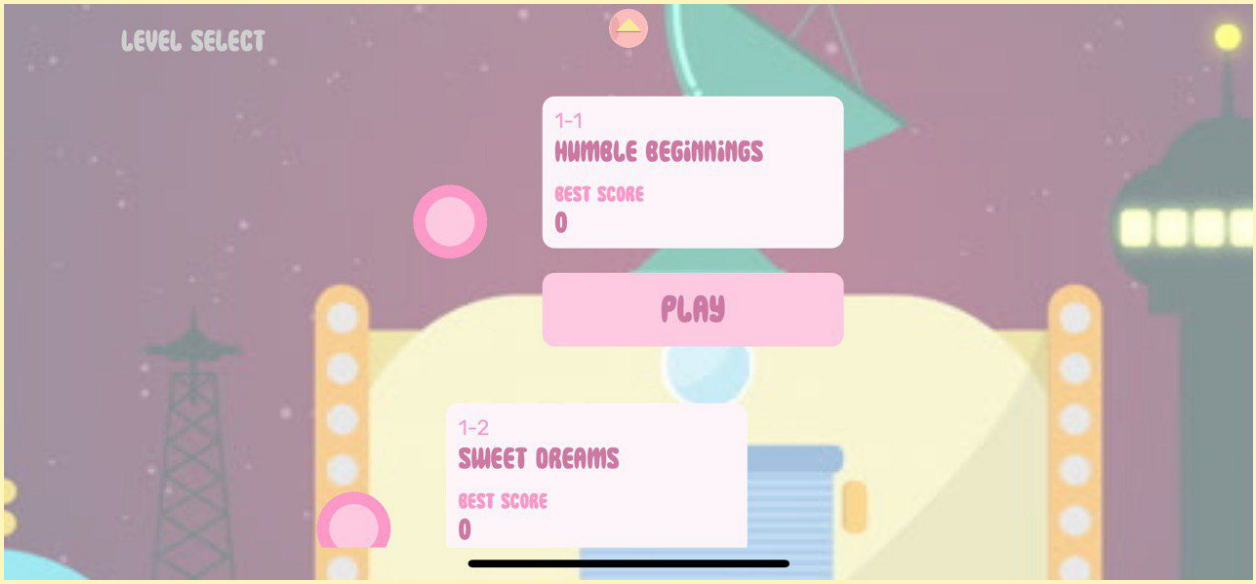
At the end of the game, the game over screen will be displayed, where players can choose to restart the level or return to the main menu.



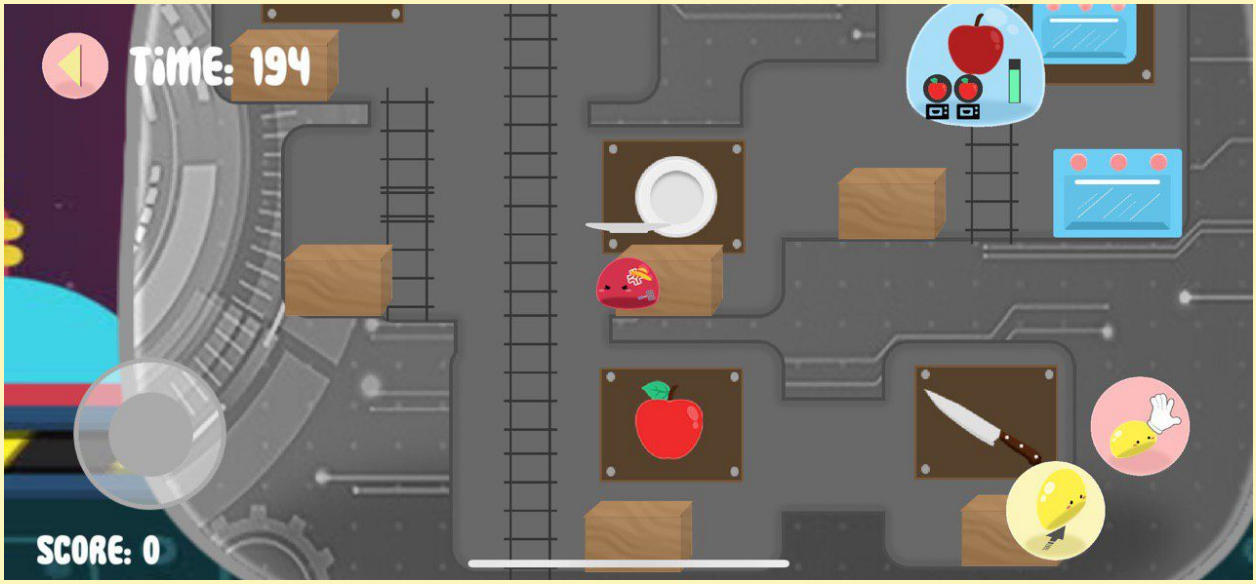
Single Player

After clicking play, there are 2 options for the players - single player or multiplayer mode.

Upon choosing single player, there will be 2 levels prepared for the players.



Level 1

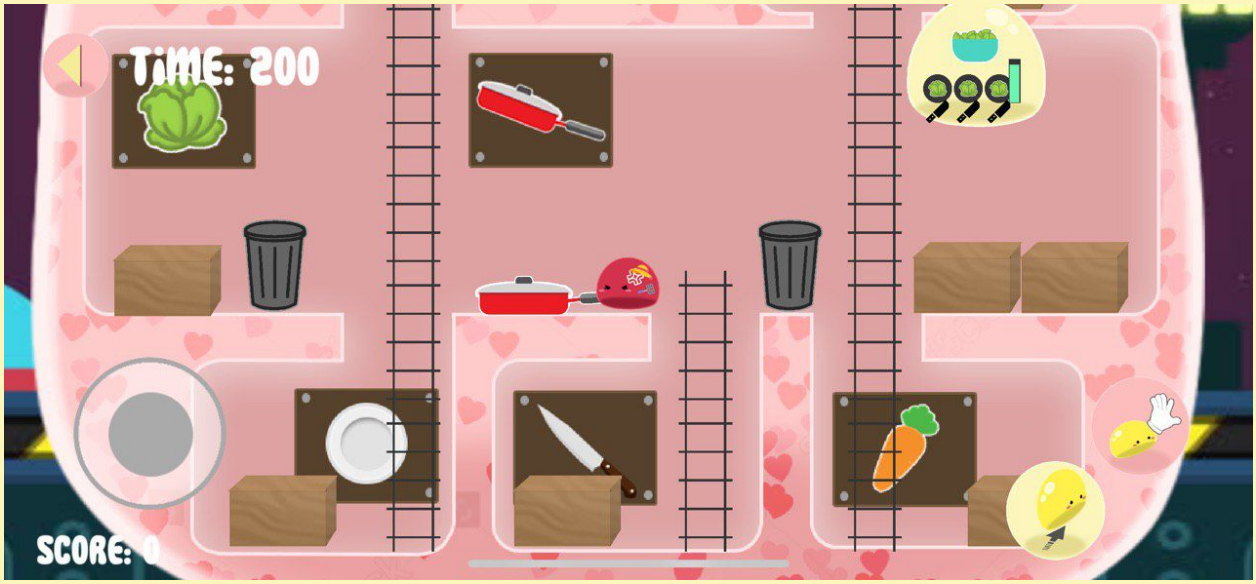


For level 1, there are only **2 recipes:**

Recipe	Ingredients needed	Quantity	Processing method
Apple Pie	Apple	1	Chopping
			Baking
	Dough	1	Baking
Baked Apple	Apple	2	Chopping

			Baking

Level 2

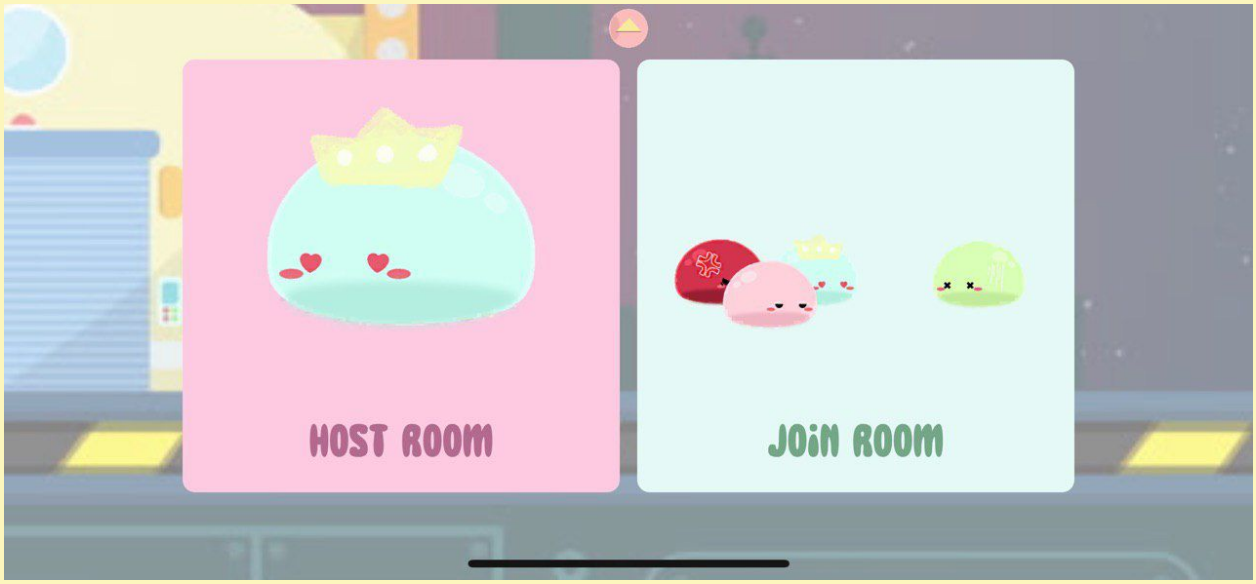


In this level, to differentiate the 2 levels and the mood from the first level, a different spaceship is being used (pink spaceship), and the type of ingredients used are different as well.
For level 2, there are **3 recipes:**

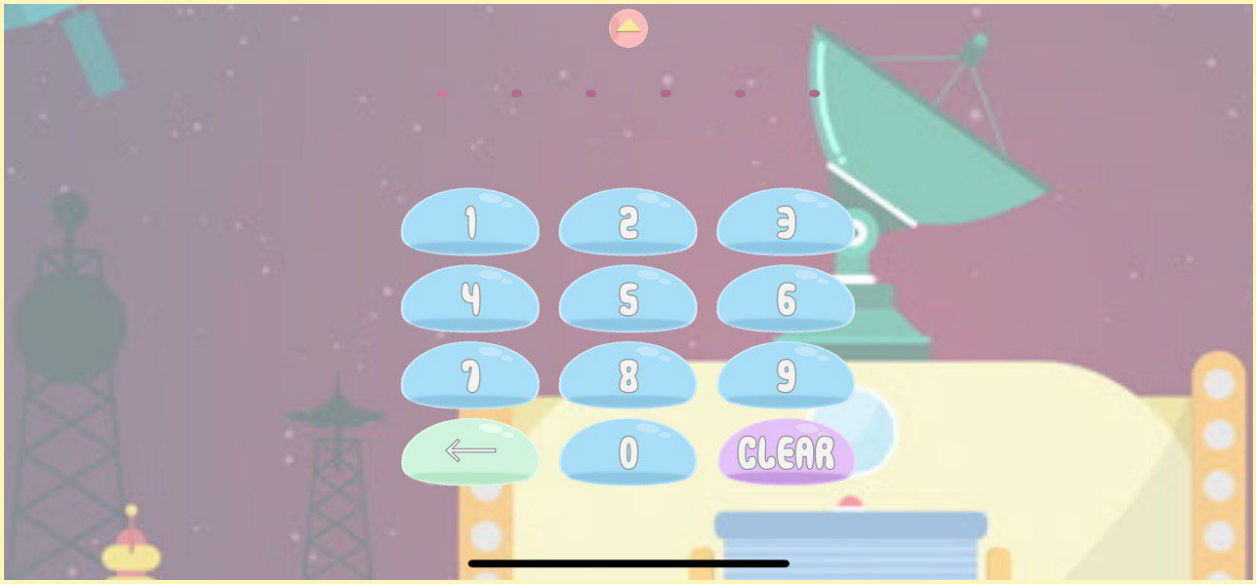
Recipe	**Ingredients needed**	**Quantity**	**Processing method**

Tri Salad	Potato	1	Chopping
	Lettuce	1	Chopping
	Carrot	1	Chopping
Fries	Potato	3	Chopping
			Frying
Lettuce Salad	Lettuce	3	Chopping

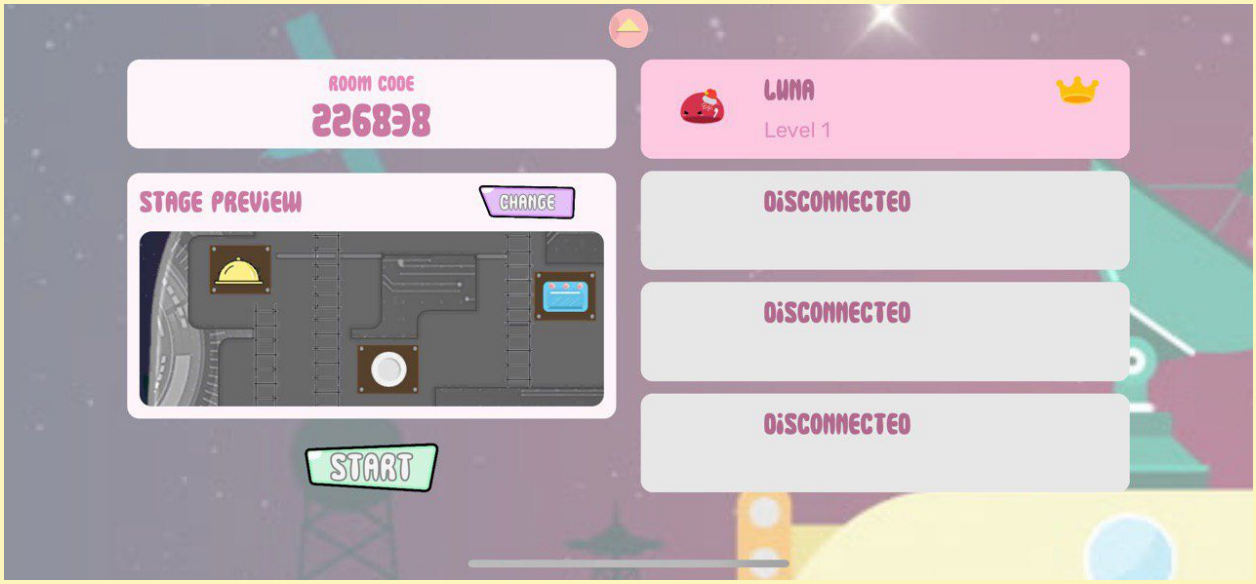
Multiplayer

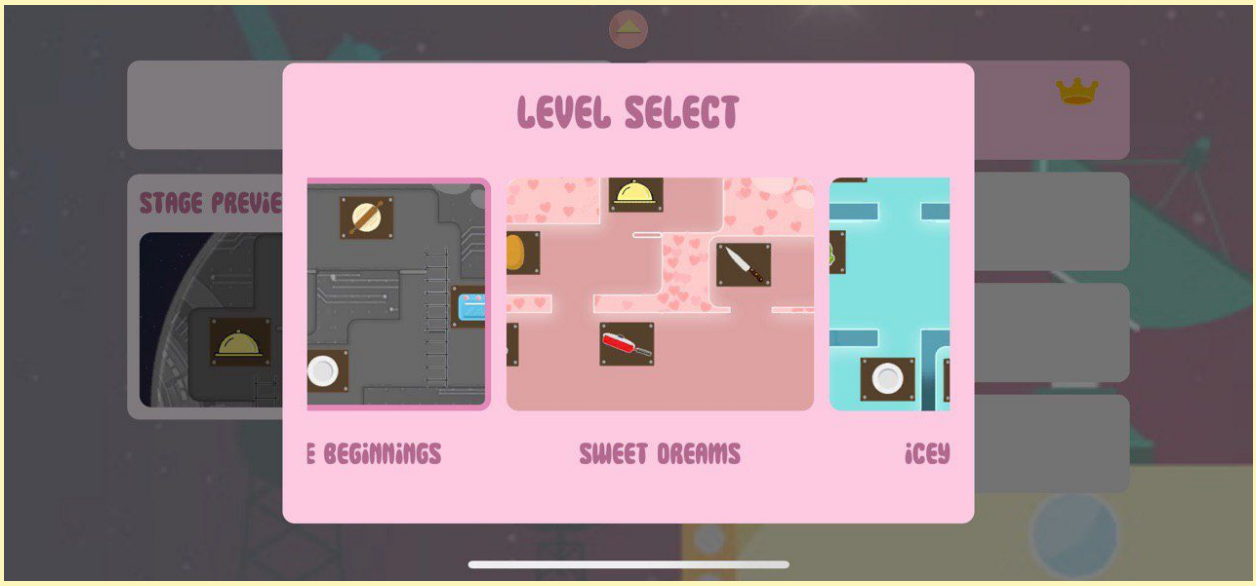


Players can either choose to host a room or join a room in the multiplayer main menu screen.

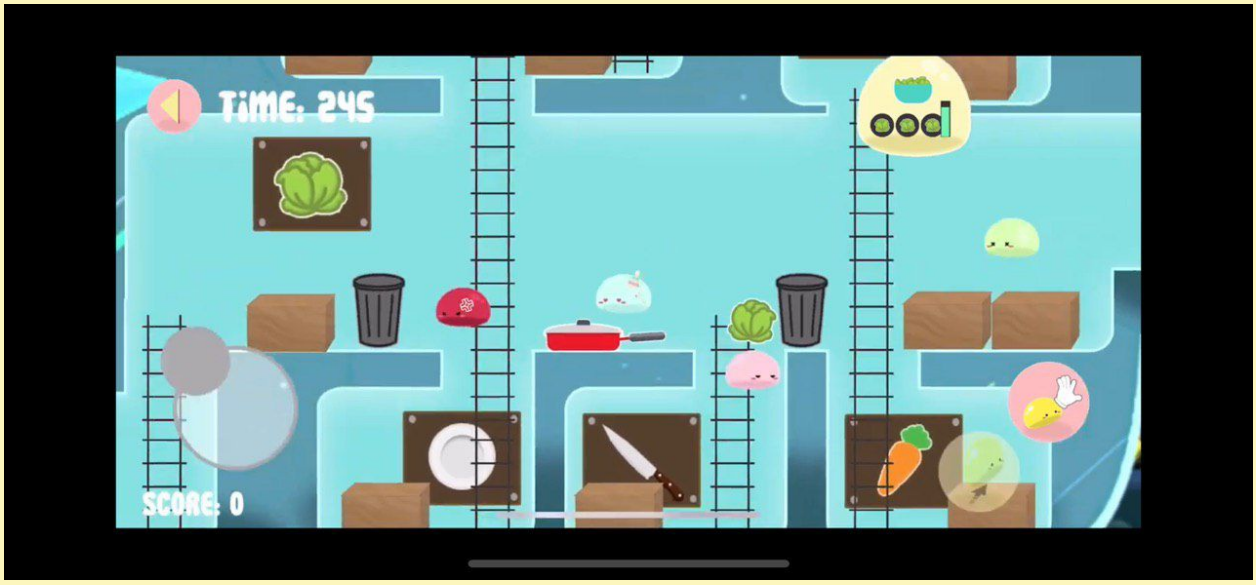


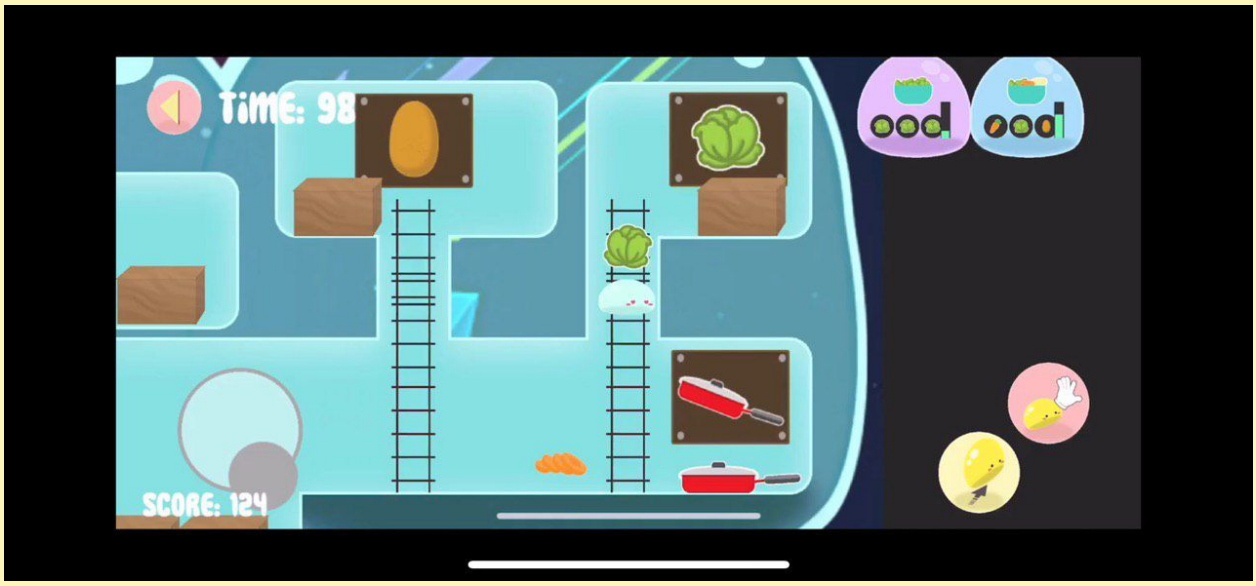
Tapping on joining room should return players to this interface.





There are 3 levels for selection. The first 2 levels are the same as that of the single player stage.
There is 1 more additional level in the multiplayer stage which is not present in single-player mode, where the map is much larger than the other two maps.





For level 3, there are **6 recipes:**

Recipe	Ingredients needed	Quantity	Processing method
Tri Salad	Potato	1	Chopping
	Lettuce	1	Chopping
	Carrot	1	Chopping
Fries	Potato	3	Chopping
			Frying

Lettuce Salad	Lettuce	3	Chopping
Carrot Potato Salad	Carrot	1	Chopping
	Potato	1	Chopping
Lettuce Carrot Salad	Lettuce	1	Chopping
	Carrot	1	Chopping
Lettuce Potato Salad	Lettuce	1	Chopping
	Potato	1	Chopping

Performance

The performance of the application is generally good. It runs at a consistent frame rate of 60 fps with the number of nodes that we have in the game. We tried the game on older devices such as iPhone 5 and 6, and there are no noticeable frame drops.

SKTextureAtlas

According to the Apple's Developer Guide, *SKTextureAtlas is a collection of textures that were either created from an .atlas folder in the app bundle or created at runtime*. ***Texture atlases improve memory usage and rendering performance by reducing draw cells.***

Benefits of using less memory:

- Increases performance: Accessing memory is the second slowest part of the hardware. The more we can reduce our memory usage, the faster our application can run.
- Prevents our application from crashing: If our game ends up to have many textures and images, there's a chance for our application to use up more memory that what the device have, especially for the older mobile devices. iOS will terminate other applications to make up for the difference, but when that fails, the application will thus terminate, appearing like a crash.
- Keep our application running in the background: When memory gets low, iOS terminates the applications running in the background as mentioned in the previous point. If our memory requirements were

being decreased, the chances of iOS terminating our application will be lower when it's in the background, and the application will thus seem to run faster.

Another reason for us to use texture atlases instead of the images and sprites itself will be that it makes our game run faster.

To see the performance benefits, we are able to run the a test whereby we run the application with many sprites without atlases, compared to one that uses atlas to render. Comparing the FPS displayed on the screen, there will be an improvement (increase in value) for the FPS.

Physics Collision

In our game, we are using the physics bodies that's under the SpriteKit's physics engine. For this, we are looking at contact detection, not collision detection. Contact detection merely detects when one physics body overlaps another in space, it doesn't otherwise move or affect the bodies in contact.

Designs

Overview

Use of Third-party Libraries

In this project, we decided to use Firebase, RxSwift, RxCocoa, RxGesture, and SnapKit for the main UI.

For the Game Scene, we are using SpriteKit, AVFoundation and UIKit for now.

Firebase

Firebase provides multiple useful services, but we are only using their anonymous authentication (to handle anonymous identification) and realtime database (to handle storing and synchronising of game states) features.

The most significant feature of Firebase is that it is entirely possible to support our game without having a server using its serverless architecture. Thus, the client immediately communicates with the Firebase realtime database, reducing the number of codebase we have to create and maintain.

Alternatives we considered and why we ended up choosing Firebase:

- Creating our own server which supports real-time data and anonymous authentication service
    - Pros
        - Might have a faster response times than Firebase if implemented correctly (Firebase uses GRPC which is essentially HTTP/2)
        - Flexibility in the the type of data we can store and how we can manipulate the data without the client needing to notify the server.
        - Able to do complex queries unlike Firebase.
    - Cons
        - Need to build an entirely new server which supports websocket connections and anonymous authentication,

which is another codebase we need to support and keep updating as time passes.
- Other realtime framework which does not need us to build a server and database from ground up. (e.g. Prisma)
    - Pros
        - Querying is more flexible than Firebase.
    - Cons
        - Same reason as previously, since most of these frameworks require us to create an intermediate server to communicate with their interface.
        - Most of these realtime frameworks still use HTTP, which probably does not improve the response times.

Rx Libraries

Rx libraries provide us with a easy way to handle view updates and actions from the user. It allows the abstraction of such things and allows for an easier development time.

SnapKit

SnapKit allows us to programmatically create view constraints without hassle, which ultimately helps us save time.

Realm

Realm allows us to be able to easily write and read persistent data. Even though we did not have a lot of data, it was enough to warrant an external library - UserDefaults was hard to use for anything that were not mostly strings, and CoreData was hard to work with.

Of course, we built an abstraction layer over Realm, so if we decide to ditch Realm for some other persistent storage solution, we only need to change the abstraction layer.

UI Routing and View Controller Hierarchy

Even though UIKit provides us with a way to handle routing and view controller hierarchy, we found that it did not fit our use case as a game. A game is expected to have fun and unique transitions, as compared to utility

or productivity applications where sticking to the norm is expected. UIKit's segues are unable to achieve the visual effect that we envisioned, and hence we built a custom routing solution.

This approach allows us to decouple components as much as possible - each view controller's view is in a .xib file. Hence, multiple people working on multiple views are unlikely to lead to conflicts during integration as compared to working on the Storyboard together. Furthermore, since our ViewController class does not have any rules on its hierarchy, a ViewController can compose of other ViewControllers, which is not possible when using UIKit's UIViewController class.

The main issue with implementing our own routing and view controller hierarchy is implementing the lifecycle methods that we needed.

Alternatives we considered and why we ended up choosing this approach:

- UIKit - using the Storyboard, UIViewControllers, and segues.
    - Pros
        - Built into the standard library, easy to get started with.
        - Storyboard provides a cohesive view of how each screen relates to another screen, and what segues there are at a glance.
        - Easy to configure and modify.
        - Stays true to the view controller lifecycle without additional configuration.
    - Cons
        - Interface Builder is basically a glorified XML builder, the XML that it produces is hard to read. When multiple work on the same storyboard, merge conflicts are bound to occur, and the only way to fix the conflicts is by trying to understand the XML that is produced. We foresee that this will cause a huge problem in the future.
        - Storyboard segues have limited customizability - they are unable to provide us with the visual effect that we wanted.
        - Depending on how the segues are called, it is possible for view controllers to not get deinitialized as it is still on the routing stack, causing memory leaks.
- SpriteKit - using a SpriteScene, nodes, and cameras
    - Pros

        - Built into the standard library, easy to get started with.
        - SpriteScene has impressive support for keyframe animations, with visual feedback.
        - Many built-in effects like camera movement and lighting.
        - Easy to configure and modify.
    - Cons
        - SpriteKit's interface builder do not have IBDesignables like UIKit, making it tiresome to configure colours and text styles manually.
        - Building all routes in a single SpriteScene means that all view elements are loaded at the start, which is a waste of memory.

SpriteKit

SpriteKit is being used for the physics simulation of the game world and for the rendering of the sprites into the game scene.

SpriteKit is being chosen as it is the most maintained library for physics in Swift currently as compared to the other alternatives that we have. In addition, SpriteKit leverages Metal to achieve **high-performance rendering**, while offering a simple programming interface to make it easy to create games and other graphics-intensive apps.

SpriteKit also offers the options of running animations, allowing us to add in more effects into our visual elements and gracefully transiting between scenes.

AVFoundation

We use AVFoundation as our audio subsystem for playing background music and any form of sound effects. It provides us a platform that allows us to edit any form of audio or video clips easily if required.

UIKit

This is being used for user interfaces. The UIKit framework provides the required infrastructure for the iOS apps. It also handles the event for things such as multi touch, which is what we need for our application as well (when the character needs to move around and jump/interact at the same time).

Module Structure

Model Class Diagram (found below)

Current module structure:

Stage

We use Stage class (a subclass of SKScene) as the base of the game, where this stage class will presented by the view controller, thus utilizing the SpriteKit and its engine for the game. This will be used to render most of the Game Objects that are defined as SKSpriteNode.

The Stage class should not contain much logic due to some reasons as per considered below as per referred from: https://developer.apple.com/library/archive/documentation/GraphicsAnimation/Conceptual/SpriteKit_PG/DesigningGameswithSpriteKit/DesigningGameswithSpriteKit.html#//apple_ref/doc/uid/TP40013043-CH7-SW1

1. We don't directly add all the nodes to the Scene directly. This is to allow more flexibility over the control of nodes. For instance, looking at the diagram for our module structure, we can edit the structure of the Slime SKSpriteNode alone only without affecting everything else (e.g. Ladder), such as applying any rotation or scaling to the sprite itself.
2. We use the same type of drawing mode for all the rendering, as this makes the rendering faster and much more efficient, hence the performance will not affected as much as possible.
3. As seen in the diagram, most of the logic is done outside of the Stage class, which is what we are trying to achieve. This is for us to encapsulate the different behaviours, thus we created the different nodes and subclasses to handle the logic.
    a. This allows our scene to adapt more easily.
    b. Our classes and logic are not tied specifically to the scene itself, but working on behalf of the scene. Hence, our methods need not touch any SpriteKit and/or SceneKit content and we are still able to implement the Game Logic.

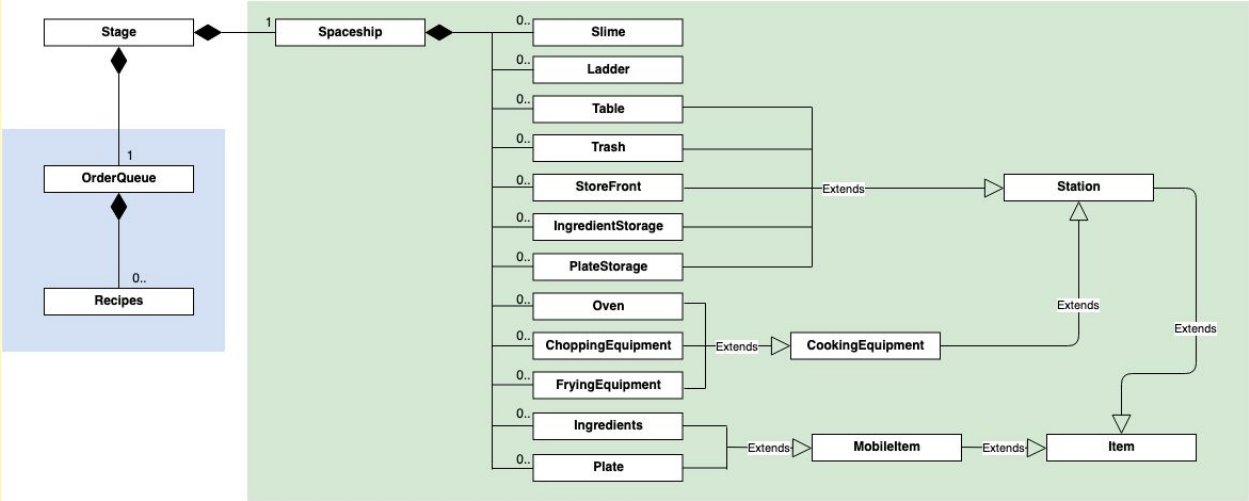
Stage consists of a spaceship, a SKSpriteNode, where most stuffs that are required for the game are inside, and a OrderQueue, a class that maintains all

the recipes that the players are asked to make and generate new orders based on that recipes.

We made Station as a superclass that have many classes subclass it. Although it look like that is very similar to a protocol, we use **superclass** instead because there are several repeated stuffs in init part where this class subclass and utilize init and properties of SKSpriteNode, and since protocol cannot subclass SKSpriteNode (of course), we make it into a superclass such that only a minimal init is done for its children classes.
How we organize our structure for the classes in terms of rendering is similar to that of a node tree. We would want to include the node as a part of the node tree as it will have a chance of being rendered later on (for e.g. the ingredients the character is carrying around), the node contains actions (the logic in the classes) that are required for the gameplay itself.

As a result, this makes the whole rendering of stage much more easier as our intention was to handle the rendering of the stages to be within 1 function - generating level. By doing so, when we wish to implement **more stages** and **different types of level design**, we are able to do it much more easily as everything has be accounted for within **generateLevel**, whereby the individual data such as positioning are accounted for in the Spaceship class and/or the individual classes.

In this game, each player will control one slime each and will go and interact with various stations to produce the food that is equivalent to the recipe that is listed in the OrderQueue.



Item has two main functions, which are:

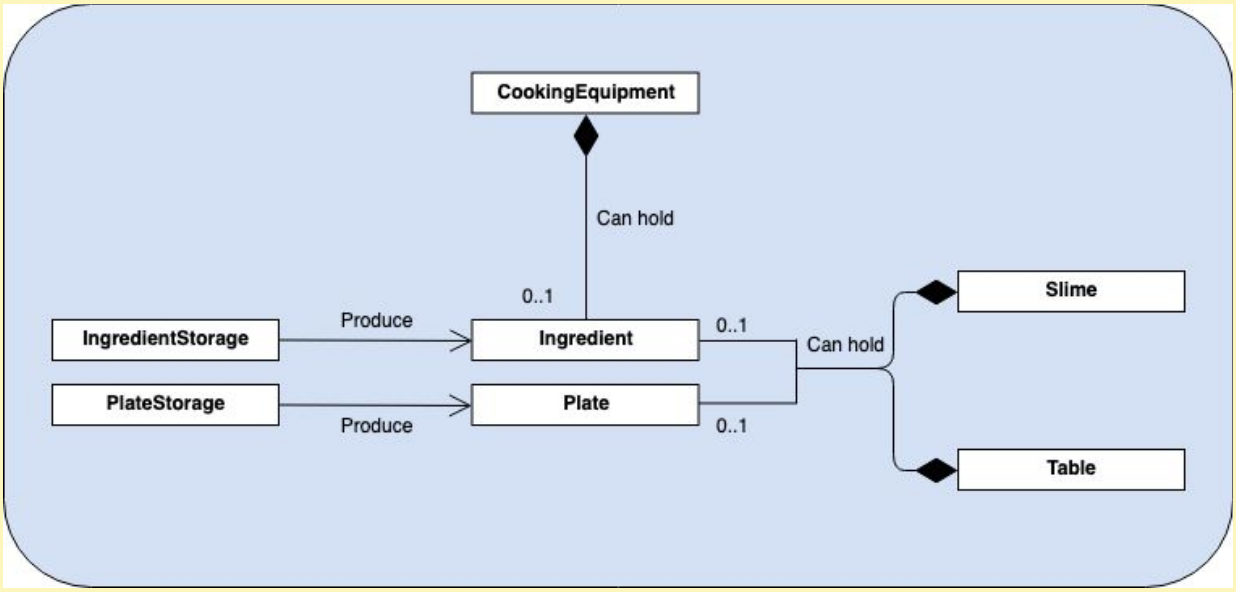
```swift
// Check whether the item is able to be processed by this station
// Parameters:
//      - item: the item that is queried, can be nil, where there is no item passed to this
// Return value:
//      true if the the item is able to be processed by this station, false otherwise
func ableToInteract(withItem item: Item?) -> Bool {
    return false
}

// Process an item
// Parameters:
//      - item: the item that will be processed, can be nil when there is no item passed to
// Return value:
//      Optional Item, the item that has been processed (or nil if the processing does not p
func interact(withItem item: Item?) -> Item? {
    return nil
}
```

- ableToInteract(_ item: Item?) -> Bool
  This function will take an item (it is intended for the slime to offer the item it carries to this function) and check whether this item can process this particular item. It is possible that some items is intended to process stuff when the item is nil (for example, when taking a mobileitem or to cook ingredients).
- interact(_ item: Item?) -> Item?
  This function will take an item (possibly nil) and process the item and return the processed item (possibly nil). It is possible that this function take nil and return something or take something and return nil.

These two general function will be overridden by all its subclasses, including stations, which are depending on what each Station do when they are interacted by the slime, and mobileitems, such as plate or ingredients, for example:

- Table will give item in it if there is no item given and there is something on the table, will take the item if there is no item in the table and there is an item given, or add ingredients into plate if there is a plate in top of it and the item given is an ingredient.
- Trash will take any item given to it and will discard it
- Storefront will only take Plate and serve the food inside the plate
- Ingredient and Plate storage will give Ingredient of a given type and Plate, respectively
- Cooking equipment will take ingredient, process it and return it back (or nil if the ingredient is still processed)
- Plate can take nil and return itself (take plate action) or take Ingredient to add it to the food inside the plate
- Plate can take nil and return itself (take ingredient action) or take Plate to add itself to the food inside the taken Plate and then returning the taken Plate back



Cooking Equipment

Next we are going to discuss about the most important thing of the game, which is the Ingredient, Food, CookingEquipment, and Recipe interaction. In

this case, we use two different types of **enums**, which are *IngredientType* and *CookingType*.

```
enum CookingType: String, Codable {
    case baking
    case chopping
    case frying
}
```

```
enum IngredientType: String, Codable {
    case junk
    case apple
    case dough
    case potato
    case lettuce
    case carrot
}
```

The reason of why we decided to use enums are due to the following:

1. Easiness purposes in designing the levels, whereby the raw values of the enums (string) of the ingredients in the recipes can be passed in a plist without changing in the code
    a. Meaning, the code itself will not have many hardcoded values in the code itself, all we need to do is edit the value through plist (which is similar to how we pass in data through JSON or even Excel files), which makes our code a lot cleaner and we just need to parse the data that is being passed in
    b. We are able to edit the level or even add new levels easily. All we need to do is add in the level details through the plist, generate the level we desire via the level name in the code and we are able to generate any level that we desire.
2. We are able to extend our data inside enums and edit it easily based on the game and/or level design. For instance, we can add a new ingredient into the Ingredients enum class much more easily. This is extensible to our design.
   For instance, based off the Ingredient Type, all we need to do is just add in one more new ingredient *case rice* and we just need to key in the value of rice in our data (plist) that is being parsed in to our recipe.

Ingredient class has several important properties and methods:

- type: IngredientType
  Type of the ingredient
- processed: [CookingType]
  The array of CookingType enum that this ingredient has been processed with
- currentProcessing: CookingType?

CookingType that this ingredient is currently processed with (or nil if this ingredient is not in the middle of processing)

- processingProgress: Double
  The progress of the current processing method. The processing will finish when this progress reached 100.0
- cook(by method: CookingType, withProgress progress: Double)
  Increase the progress of this processing (of course with all the checking). Then if it reached 100.0, the progress is resetted, the currentProcessing become nil (it finished the progress) while its value is appended in the processed array.

Another interesting thing we implemented for this Ingredient is that we override the hash variable and isEqual function of this ingredient class, such that two ingredients object will be considered equal and hashed to the same place when the two ingredients have the same type and exact same processed orders.

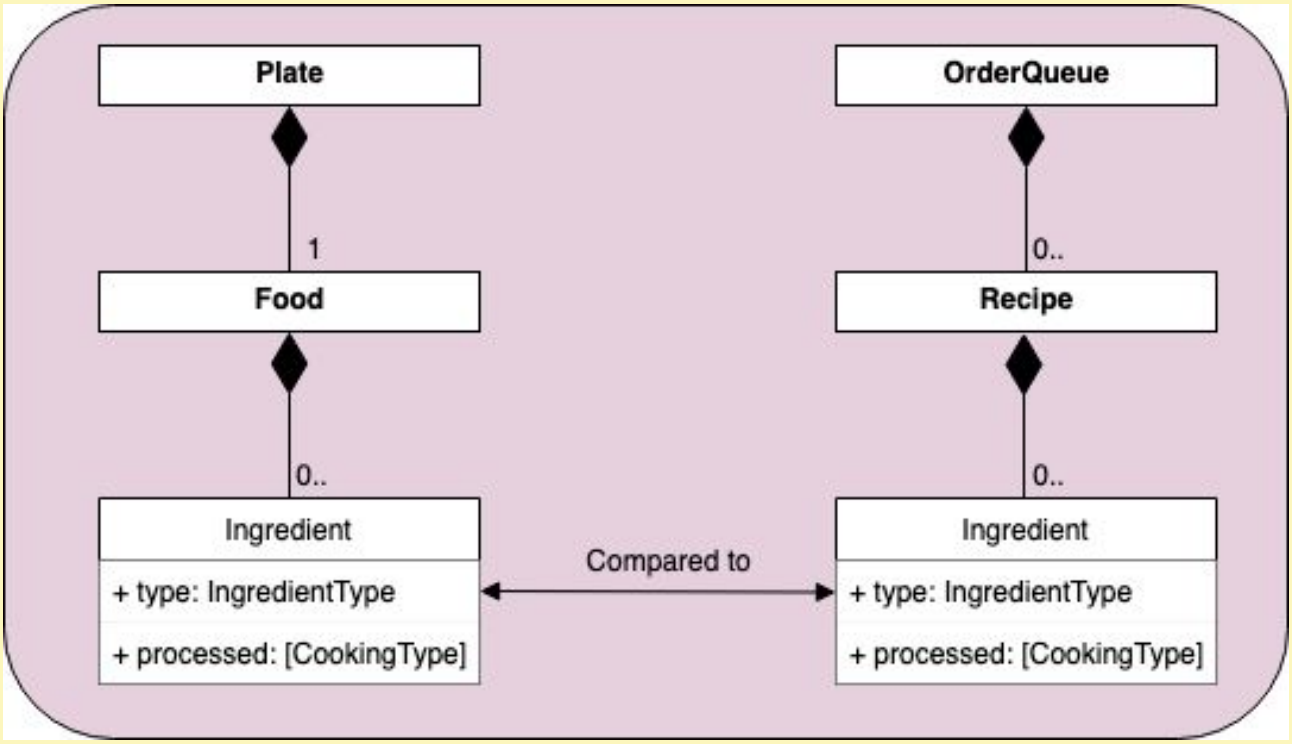
CookingEquipment is a superclass (the reason why this is not a protocol is the same why Station is not a protocol) that is subclassed to several cooking equipment, such as Oven, ChoppingEquipment, and FryingEquipment, where each subclass has different ingredient types that is allowed to be processed with that equipment, and different type of CookingType.

By default, CookingEquipment have two different types of processing, one is automatic (set by timer) and manual (need to tap interact button). Both type of processing will increase the progress bar of the cooking of the ingredient inside it. And the value of the increased can be changed in the subclasses, such that we can have equipment that only process ingredient by tapping (automatic == 0%, and manual != 0%, for example ChoppingEquipment) or only by automatic timer (automatic != 0% and manual == 0%, for example FryingEquipment and Oven), or by combination of both (but not yet used by any of its subclasses).

```
// Automatic processing, called by timer
@objc func automaticProcessing() {
    continueProcessing(withProgress: 0.0)
}

// Manual processing, called from interact
func manualProcessing() {
    continueProcessing(withProgress: 100.0)
}
```

Food is a simple class that is contained in a plate, and it contains a dictionary of ingredient as a key and integer as the value, to note how many ingredients is there each. Since, we modified isequal and the hash for ingredient, this dictionary will be compared to the recipe ingredient list in the OrderQueue.



Order Queue

OrderQueue is a class to maintain orders in form of Recipe object. It has several important properties and methods, which are:

- possibleRecipes: Set<RecipeTemplate>
  All the possible recipe template ordered in the stage via this particular orderQueue
- recipeOrdered: [Recipe]
  The current recipe ordered
- addRandomOrder()
  Find one random recipe from the possible recipe, regenerate (will explain about this in the Recipe description below), and add it into the recipeOrdered array
- completeOrder(withFood food: Food) -> Bool

When a plate is being served, the food content will be send to this function, and it will compare to the recipeOrdered element's ingredient list (Recipe ingredientsNeeded variable). If it is found, will return true, else if it is wrong serving, will return false

Recipe is a class that contained in the OrderQueue. It is initialized with an array of compulsory ingredients and an array of optional ingredients with probability for each optional ingredient (so that same recipe can require different ingredients depending on probability). This class has an important property:

- ingredientsNeeded: [Ingredient:Int]
  The dictionary of ingredient object as key and int as value, similar to the Food, and will be compared against the

Recipe is created by RecipeTemplate, where it is stored in the OrderQueue. So when the OrderQueue need to generate a random recipe, a recipe template will be randomly selected from a set of RecipeTemplate, and a recipe from the template is generated (and thus, the probability for optional ingredients will be rerolled every time the recipe is created).

Networking

This networking section only concerns the game's multiplayer features. Firstly, the multiplayer structure uses a host-player relationship inside a room. The host is not the only one controlling and observing the data flow inside the firebase database. The other players (other than the host) will also be responsible for some of the queries and writes inside the database, but are limited in scope.

The multiplayer feature of this game can be divided into two phases: the room phase and the game phase. In each of the phases, the host and the players have different roles and responsibilities in terms of writing into the database. However, for both phases, the host and the other players will be observing the same sets of data. Here is a list of the roles that both the host and the other players are responsible for in the room phase:

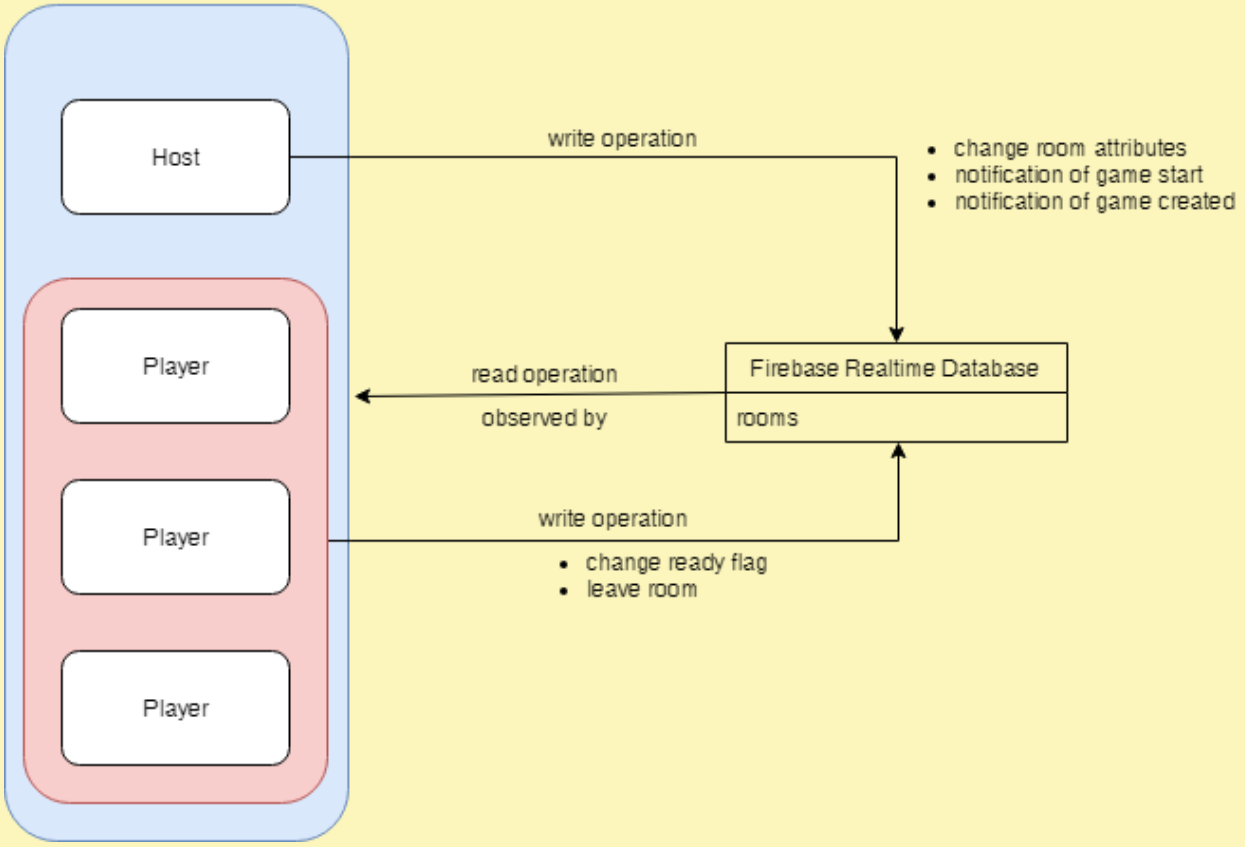
Role	Responsibilities
Host	• Starting the game • Creating the game in the database • Changing the room name • Changing the room map • Kicking out people inside the room • Closing the room • Notifying other players that game is starting • Notifying other players that the game has been created and that they should move to the game instance from the room instance
Player	• Change their own respective ready flags • Leave room
Host and player	• Observe room states all at once for changes in data

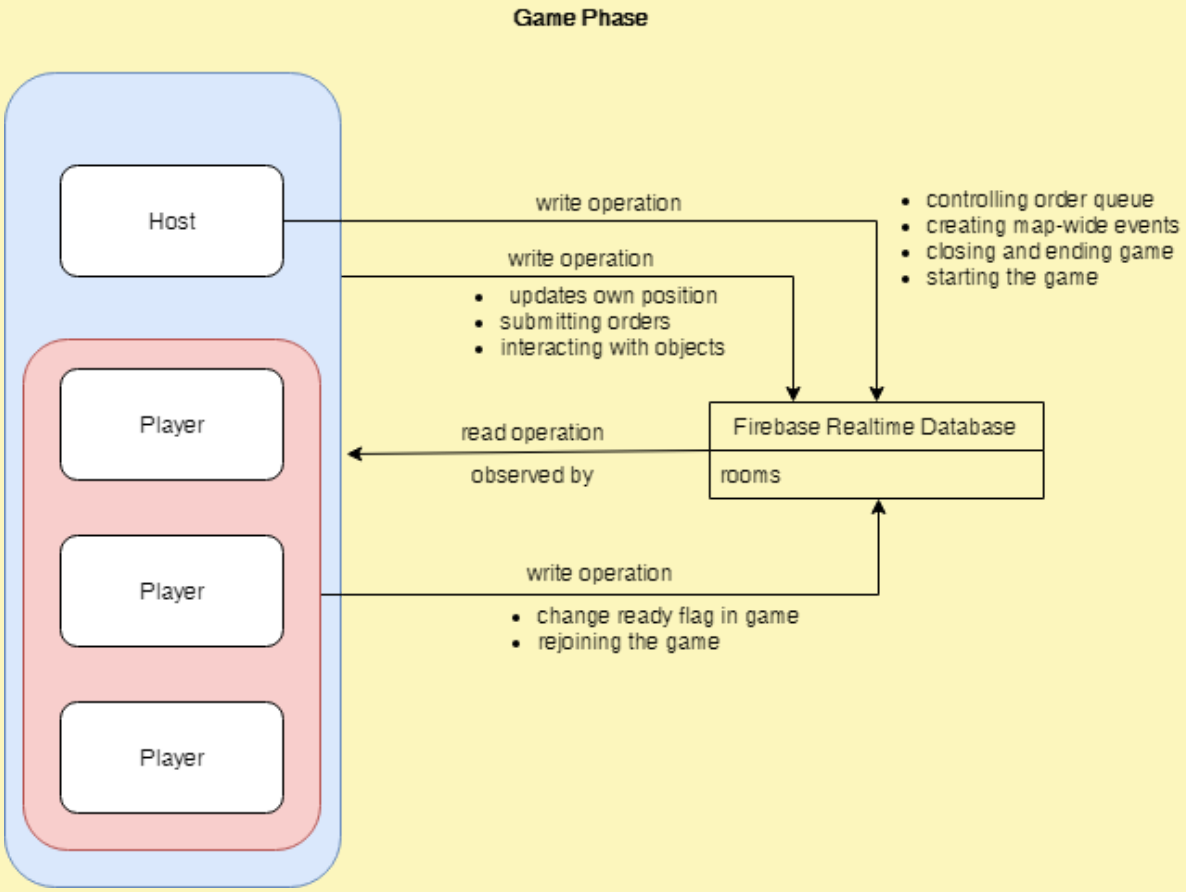
On the other hand, here are the list of the roles that these two types of players are responsible for in the room phase. Note that this structure is so that the database is the one source of truth for all the players involved in the multiplayer game:

Type	Responsibilities
Host	• Starting the game when every player is ready and inside the game view controller • Ending the game • Queueing new order every set intervals • Deciding whether a submitted order is correct or not • Making map-wide changes/events which requires precise timing intervals • Controlling game timer • Closing the game • Sending out game notifications
Player	• Rejoining the game after disconnect • Changing state to disconnected after connection is lost to the database • Setting flag to ready for game start
Host and player	• Updates own location inside the map • Observes each player's movements except for own • Observes objects' states inside the map • Observes the states of all the stations inside the maps • Updates interactions to other objects and reflects it inside the database • Submitting orders • Updates notifications for important events

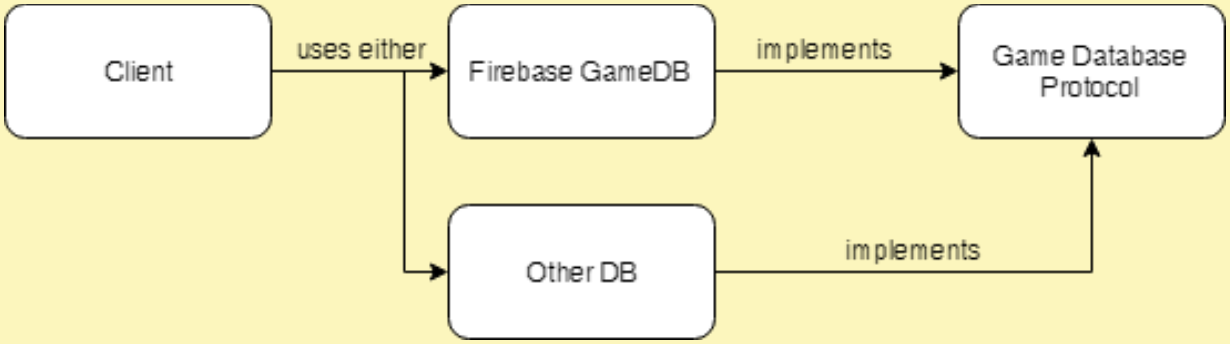
A diagram of these interactions can be shown below.

Room Phase


Host
Player
Player
Player
write operation
change room attributes
notification of game start
notification of game created
read operation
observed by
Firebase Realtime Database
rooms
write operation
change ready flag
leave room



As we are not sure whether Firebase is the best choice moving forward, we decided to create another layer of abstraction for both Firebase Realtime Database and Firebase Auth. This abstraction is in the form of a protocol which contains methods which are specific to the game mechanics/features such as joining a room and observing a room's state.



This layer of abstraction, a protocol named GameDatabase, is made such that when Firebase is swapped out in the future, only the implementation inside this abstraction class is changed but the other modules in the application which uses these abstraction class' methods will not be affected at all.

The protocol functions to query or update the database follow one specific rule. Every request must have a completion block and an error block. These two blocks must not occur together at any point of time. When an error occurs, it means that the database query or update fails vice versa. This makes it such that errors can be traced and handled properly without giving fraudulent content to the database. Thus, this ensures atomicity of the database while giving us a safer way of solving bugs.

Here are a list of all the methods, methods with self-explanatory names will not be described:

Room phase:

- joinRoom()
- observeRoomState(): listens for changes in the database and returns a closure which is fired on data change
- updateRoomName()
- createRoom()
- closeRoom(): this method also deletes the id reference in the database, allowing another person to claim this id
- leaveRoom()
- startGame(): also creates the game reference in the database
- createGame(): creates a reference of the game in the database so that it can be observed

Game phase:

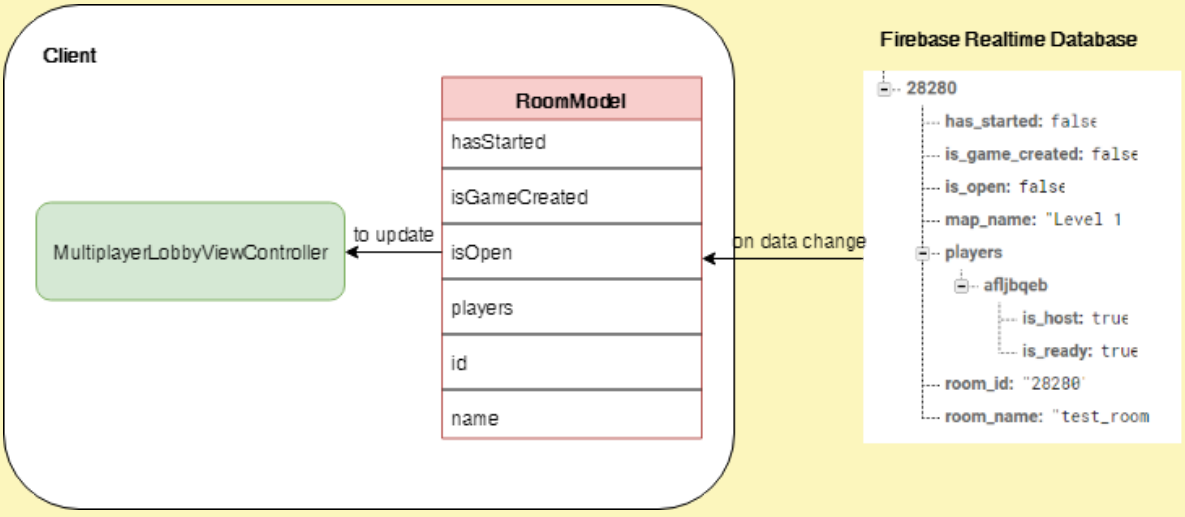
- observeGameState(): similar to observeRoomState but for the Game scene itself
- queueOrder()
- submitOrder()
- removeOrder()
- dropItem()
- pickUpItem()
- interactWith()
- sendNotification()

- addStageItem()
- removeStageItem()
- updateStageItem()
- addScore()

Others:

- observeRejoinGame(): listens for whether the user has any game which can be rejoined
- rejoinGame()

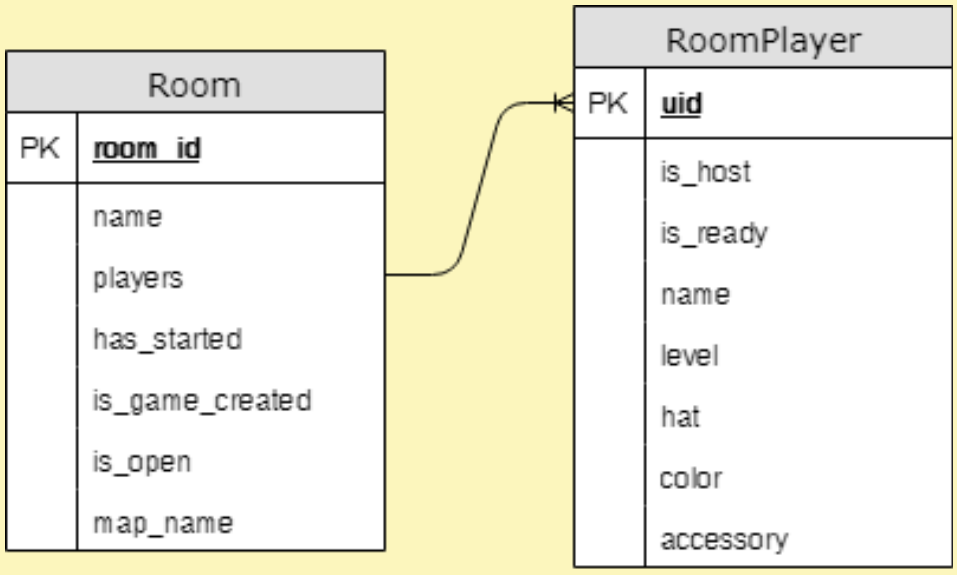
Therefore, to support these features alongside Firebase Realtime Database. Some additional models need created to make the data sent from Firebase to the client more modular (these are similar to factory functions). These models contain the game states which are stored in the database but do not necessarily need to be stored in the actual game/room scene models themselves. These models are created from utility functions inside the Firebase database implementation. This is generally a good practice when using Firebase in a native application. The diagram below shows an example of how this is going to work.

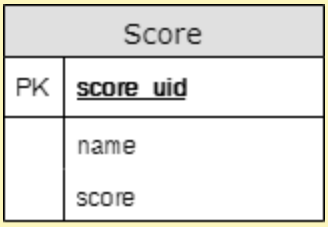


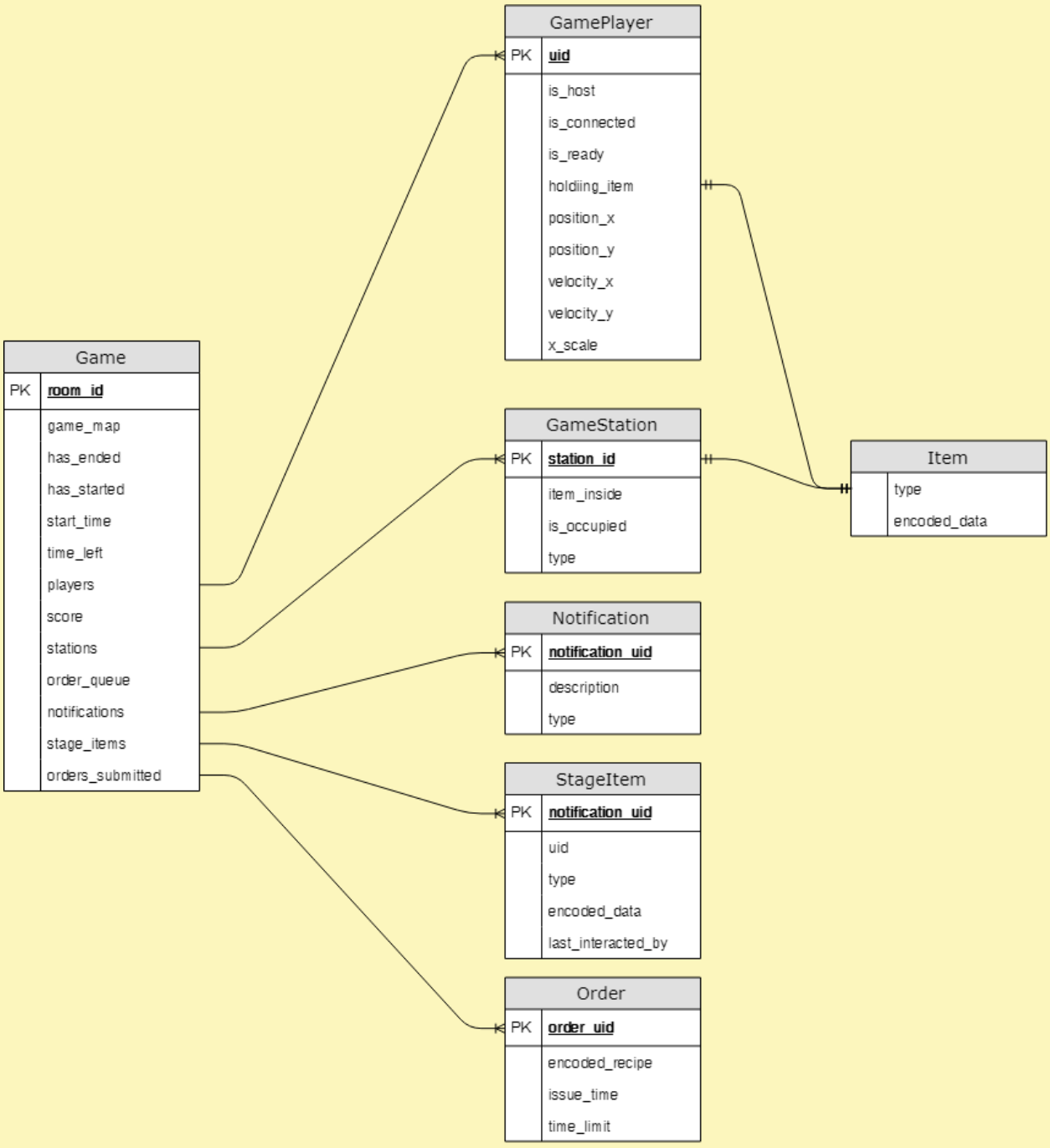
These additional models are in the form of a RoomModel, GameModel, and (probably) PlayerModel since these are the three objects which will be constantly undergoing change when connected to the Firebase database. The objects also provide a type safe environment for us to work with inside XCode When there is a change in the states stored in Firebase, the JSON

provided will be cast onto these models for use by the other modules in the application through a simple closure.

There are no new models provided for the game phase as of yet, but relevant classes/models will be created for the multiplayer mode if need be depending on how the single player game logic is implemented. Another interesting challenge is how to generate the map objects inside the database itself, considering that each map has different layouts, different stations, and different recipes altogether. The relationship diagram is illustrated below.







As previously discussed that the level data will be stored in a plist, and that the before the game starts the host has to create the game reference first in the database, we can have the host pre-populate some of the map data required in the database reference and notify the other players that the game has been created.

After the game has been created, a boolean flag is changed and the other players are able to move to the game instance with the decided map data. The logic for all of the game features will be handled locally, but the updating of the states of the stations, locations, and objects inside the particular game instance will be done by each and every player (both host and other players). This is what we feel will be the best approach to implement a safe multiplayer environment using Firebase.

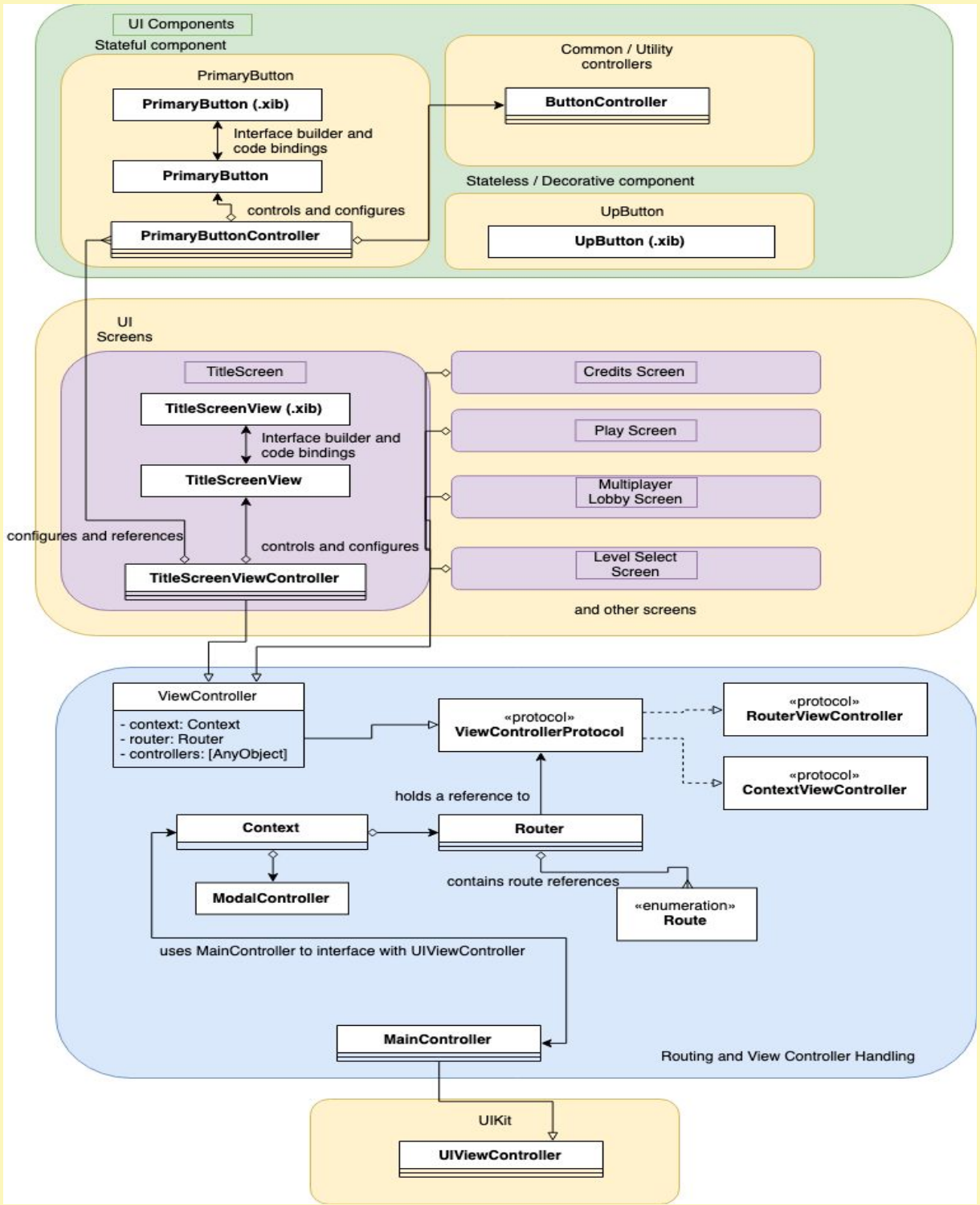
Several interesting things we did to make sure we have a very stable database, we used Firebase's transactions. These transactions prevent race conditions when users update the same references in the database. This transaction works in such a way that a request will be queued and will only fire when there are no on-going processes on that reference.

Here are the list of actions that requires transactions to be made:

- Updating the item a player is holding
- Taking item from a station or from the ground
- Putting items into an already put item on the ground or inside the station
- Adding score to current score

Lastly, for smoothing of other player's movements, we have employed two strategies to smooth the other player's movements. Firstly is to store and update the velocity of the other players. This makes sure that in the intervals between updates the other players do not just stay still and skip to the next location. Next, we implemented a tween in-between updates to animate the movement of the other players. We have found that this method is very effective in animating the movements of the other players on ladders.

UI - using the MVC pattern, and abstracting View Controllers

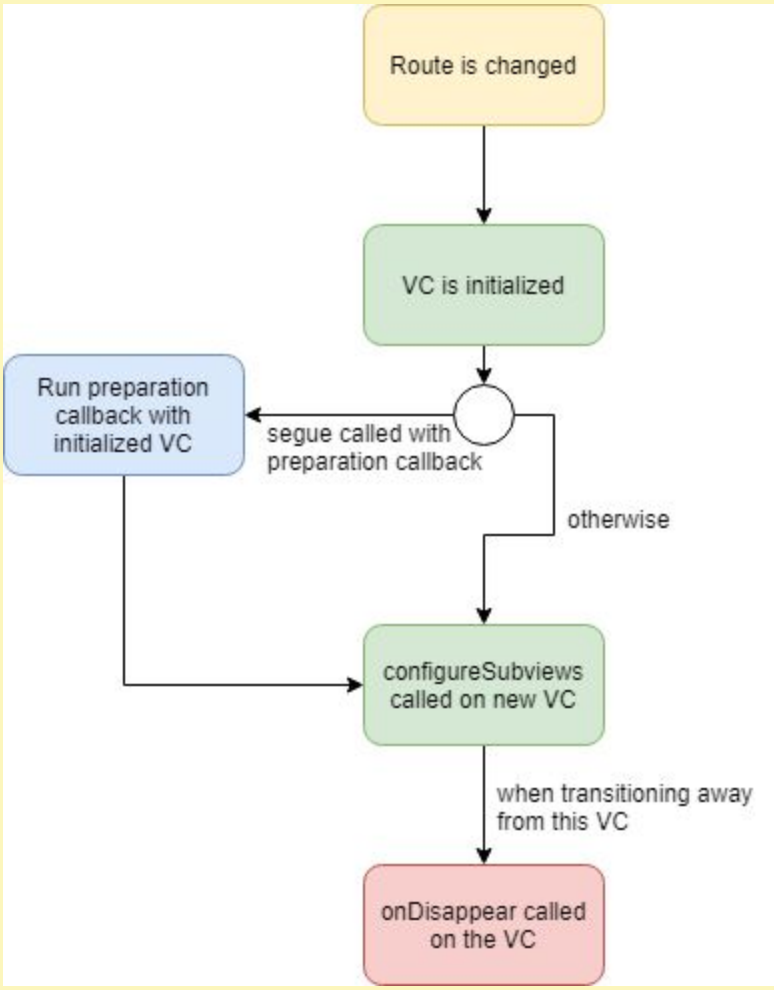


We have a MainController which interfaces with a UIViewController - all other view controllers inherit from a base ViewController class which provides some lifecycle and utility functions. Each screen has 3 essential parts - the .xib file, the interfacing Cocoa Touch class, and a controller. The controllers then further initialize any UI components if necessary. In this case, the TitleScreen has a few buttons that when pushed, should lead to other screens. The configuration for this is done in the TitleScreenViewController and PrimaryButtonController.

The MainController then controls how the transitions are handled, and does the transitions independently of what each ViewController does. The MainController also creates a Context object, which is passed into all ViewControllers instantiated so that context-based methods, such as displaying an alert or modal can be done within any ViewController. It also allows us to store global objects which we used sparingly - dependency injection between controllers is used whenever possible; only objects that are shared between multiple controllers such as the local user's data is stored in the Context.

We utilized the MVC pattern here heavily - the Model is any data structure that the controllers interact with. The controllers handle all user gesture events and decides how to modify the model as such. The controller also modifies the view as necessary, using Rx as a medium to do so whenever convenient. The View is made out of both the .xib file and the related Cocoa Touch class, which only contains the IBOutlets for the controller to interact with.

We believe that this way of doing things allows for great extensibility - we can compose controllers in any way we like, resulting in loose coupling.



Our implemented system has a simplified version of UIKit's lifecycle methods. Although this constrained us a little, we realised that this was all we needed for our case.

Testing

Testing Strategy

UI Testing

The testing strategy for UI testing is going to be manual black-box testing. The UI for the game may change drastically in between versions, and UI tests are hard to describe and code. (Should we test whether tapping a button results in an alert that says "Feature is currently unimplemented."? The alert

text may change frequently. Also, when the feature is eventually implemented, such a test is rendered useless) It may make sense to implement UI tests once the application development and user flow is more stable, but until that is achieved, adding automated UI tests only serve to slow down progress on the actual project, while not adding much overall value.

The manual testing that will be done will require sharp eyes which pay attention to detail. Some examples are as follows.

- When a transition segue is triggered, do the transitioning views have jarring split-second transforms?
- Do the transitions look cohesive?
- When a person who has never used the application starts it up, does the main flow of actions jump out at him? Does he have to look out for unusual objects to achieve his purpose?
- Are buttons placed in the right places? Does placing the menu button here cause the player to randomly pause the game due to a slip of the fingers?

Game Scene testing

The testing strategy for the Game Scene will be mainly manual black-box testing. This is due to the reason that many things that are happening in the game scene will be best represented through visual cues and pop-ups in the game itself.

The game logic itself will be a different set of testing from the Game Scene testing. This is just purely what the user can see from the game scene.

Some examples are as follows:

- What happens when the joystick is being held down and dragged around? What expected behaviour will the slime have?
- Does the amount of drag to the joystick correspond to the distance the slime is moving? Will it feel natural?
- Upon tapping the jump button, what is the expected behaviour of the slime?
- Is the positioning of the joystick and the action buttons are of a bad positioning? Are the interact and jump button too close to each other, causing the players to have a wrong action resulted?

- What happens when the player tries to interact with an interactable object? What happens when the player tries to interact with a non-interactable object?
- How does the player navigate around?
- Will there be any sound effects when certain actions are being executed?
- If the players tap on anywhere on the screen but not the joystick and action buttons, what is going to happen?

Game Scene Logic testing

As per mentioned above, the game scene testing itself will be different with the logic.

What we are looking at here is what is going on beneath the product, the game logic itself. For instance, what is going on when a certain action is being done and called. This will consist of both black-box and glass-box testing.

Some examples are as follows:

- What happens when a certain ingredient is being placed into the pot and starts cooking?
    - Should expect a visual cue such as countdown bar to signify how long does it take to cook?
    - If there is no ingredient, shouldn't expect the pot to start cooking itself and there won't be any visual cues and should expect a null value in the pot.
- When the player serves a dish, should expect points to increase, and the first item on the menu list that matches the dish served to disappear.
    - What happens if the wrong dish is served?
    - What happens if the plate is empty?
    - What happens if the dish is not being served?

Networking testing

To test our networking component, we are intending to have both black box and glass box testing.

Some examples are as follows:

- What happens when the player taps on Multiplayer mode?
    - Should expect a room to be created if there are no rooms available?
    - Should expect the player to automatically join a room?
    - Should expect the player to choose a room and join?
    - For all the points above, what kind of feedback/visual cues will the player receive?
- What happens during the game?
    - Is the database properly updated when a user interacts with an object?
    - Is the game going to properly be closed after the timer ends?
    - Is the game going to be started at the same time across all devices?
    - Is the game running on synchronised timings?
- What happens when two players attempt to do the same action together?
    - Does taking an item from a station together cause problems such as duplication?
    - Does taking an item from the ground cause duplication problems?
    - Does putting an item into a plate on the ground or into a station cause unnecessary problems such as items disappearing?
- If the player loses connection, will he be kicked out of the room immediately, or will there be an attempt or a chance to reconnect?
    - Any feedback/visual cues?
    - Should expect a reconnecting status?

Test Results

UI Testing

The UI Testing has been done mostly by allowing other people who have not tried the application out to play around with it. There were no major issues found and most of the navigation was rather intuitive.

We also made use of the opportunity at the 14th STePs showcase to see how different demographics use the application, and most of them had no issues.

Stress Testing

Making use of the 14th STePs showcase, we had the opportunity to stress test our application under the following conditions:

- Intermittent and slow Internet connections
- Multiple multiplayer games running at the same time
- Long periods of application usage

There were a few issues found, such as memory leaks that would cause the application to stutter as more time went on, but for the most part, things worked fine and most games were carried to completion without a problem.

Reflection

Evaluation

Name	Evaluation
 Anthony	Failures: Integration of multiplayer features into the previously single player game is very difficult. While initially my strategy was to integrate the multiplayer after every feature in the single player features are implemented, this is not a good strategy and led to a few implementation problems on the days leading to STePS. On hindsight, we should have planned the entire architecture with multiplayer already in mind. Some of the implementations can sometimes be considered as a shortcut. This is shown in how the handling of race conditions manifested in the actual game, where the objects can be shown to be duplicated for a short amount of time before automatic accounting takes place. The single player features should take into account all the multiplayer implementations in mind. I should have indicated this at the start, but I developed the multiplayer implementation in isolation knowing that it will pose a problem for us in the future. Successes: However, despite this problems, I feel that the implementation of our networking and multiplayer system is stable and very dependable, as shown in STePS, where even four people are able to play side by side without any interruptions. Furthermore, the host-player relationship designed for multiplayer has gone through several revisions, where the host's responsibilities increases over time and it has been a success in handling the flow of the multiplayer games. The

	system that we built ensures minimum chances of other players cheating during the game itself. In the end, the code is structured in such a way that there is very loose coupling between Firebase and how the network is implemented. This is thanks to the protocol that was created so any implementation of database should conform to that protocol. Thus, if we decide to swap out Firebase in the future, we will not change any of the implementations inside the game itself. For the game itself, we are proud of how we handled the movement of the other players inside the game. Since the host does not need to keep updating his own position in relation to the database, the host should not experience any form of lag. However, the other players' movements have been smoothed to account for latency so that they are shown to move as seamlessly as the host.
 Gabriel	Successes: The introduced routing / view controller hierarchy replacement worked very well for our use case, deallocating memory whenever possible. There was an issue for our use case with UIKit's routing other than aesthetic purposes that we found out later - if we pushed view controllers onto the routing stack, the memory was not cleared unless you actively cleared it, which meant that your stack was no longer a stack. I managed to create a method of dependency injection between view controllers without that complication, and it solved some of our memory leaks. It is also rather simple to replace a UIViewController with our ViewController, making the transition between libraries seamless. Another success would be the loose coupling of the system that we made. Most of us worked on different parts of the code at the same time, and there was little trouble during integration. I believe that this is due to how we chose to build our system: in a modularized manner, such that it was easy to integrate with other parts. Failures: Folder Organization - while dumping all the controllers inside a Controllers folder and views inside a Views folder

	works well for small projects, it becomes unwieldy as the project becomes bigger. I did not enforce duck-typing earlier, and navigating the project became more arduous over time. Routing - although most of the routing was a success, I failed to create a generalized routing system that could perform any sort of transition. This was largely due to an oversight that we only needed one router, and as such I chose to skimp on generality to save time. Further on in the project, I found that I required nested routers within, and ended up creating specialized routers and transitions for them, which was not ideal. Performance Issues - Due to the nature of controllers, we have to store them while they are active, and discard them once the controller becomes irrelevant. However, it is easy to commit the mistake of forgetting to do so, resulting in a strong cyclic dependency between the stale view controller and the controller, which is a cause of memory leaks. Since there are no ways to enforce this, manual checking is required.
 Hui Qi	Success: ● Being able to implement most of the features with the current structure that our group has discussed. When we were trying to implement additional features for the new level, we were able to implement most of the features quite easily with the current structure, which is a good thing as this proves that most of our current structure works. Failures: ● Personally, i would want to have created the assets much earlier and created more levels in terms of product. However, this module was focused more on the software engineering aspect. Hence, this will bring lesser attention away from the product itself. I took more time trying to stay within the structure we have, more time to improve the coding style and standard.

 Samuel	Success: ● Decoupling between classes, and even functions inside each classes inside the models, thus making adding features much easier. For example: ○ Enabling ingredients and plate to be dropped and taken from floor; this idea only proposed two days before steps, and within half a day this functionality (including for multiplayer - coordinating with Anthony) already implemented successfully with minimal changes. ○ When putting the ingredients into plate correctly, the sprite changes into the recipe picture; this idea only proposed the morning of the day of the steps, and I (and Anthony, for multiplayer coordination) was able to implement it successfully with minimal changes ● “Prepare for the future” implementation, where some parts of the code are not utilized fully yet. For example, optional Ingredients in recipe (so that same template can produce multiple different recipes in the order by probability). Although sometimes this is a bit overengineer that make some implementation difficult since need to consider this. Failures: ● A bit slow in implementing the classes, where the single player only finished one week before steps, causing chaotic multiplayer implementation and coordination. ● Since I did not have the idea what the network is going to do, I was designing the classes for single player implementation only. I should have coordinate with Anthony for networking much earlier so I can design the class better and easier to integrate with the network.

Lessons

Name	**Lessons Learnt**

 Anthony	I have learnt that the most important thing to do in a team is to communicate. This communication includes how we design our code and how we will implement old or new features. Most of the time that I do during the last few days leading up to STePS is to communicate with Samuel, who implemented the single player game logic so that I do not accidentally break any game flow and implement the multiplayer features properly. As I also worked on the multiplayer UI flow, communication with Gabriel was also essential and some multiplayer game flow which is unique to the game mode itself requiring unique flow, in this case I asked Hui Qi for help. The multiplayer room system and the game multiplayer system would not be as polished without proper communication amongst every one of us. I feel that I have learnt a lot from both solving the problems that arise from this and from looking at how the others work. The tenacity of my team members have allowed me to push what I have been able to implement in this project. Overall, this project has taught me a lot of technical and soft skills required to perform well in a team.
 Gabriel	I would probably have fleshed out the routing idea a lot more thoroughly before any work began. It was easily the core of all the UI work and a sloppy first job made it tough to extend its functionality generally. Also, I would have taken the entire routing part into its own library, as it rightly should be. This would allow for less coupling of code, and make the overall code base cleaner. I also learnt that proper communication and planning can make integration less painful. I had not worked in many projects with other people before, and as such did not know what to expect when working with other people on the same project. However, communication with the other team members, as well as reasonably good project planning, allowed us to work in tandem without many issues. We constantly updated others on our own progress so no double work would be done. As much as possible, we built our code to be easily used by other people, so understanding the code that other people wrote did not

	pose too much of a problem.
 Hui Qi	I was able to learn quite a lot of things from this module, learning how to have a cleaner and proper structure for the whole project. This actually allows me to learn better on how to create a structure from scratch, and how important the code structure and architecture is. For future projects, I am able to better structure the entire code. Due to the nature of the platform (XCode) that we are using, some things didn't work as well as expected, so it took a little more time and some restructuring here and there was needed. I didn't really like how some things are being done on my end due to limitations of SpriteKit. In Game Development on other platforms such as Unity, the structure is a lot easier to be controlled and certain things that are related to rendering are much more flexible.
 Samuel	The first lesson that I have learnt in this project is how to design the code nicely from scratch. Splitting the project into different small parts will help a lot in designing and coding process. In fact, I just realized I can refactor a part of my code better and make it cleaner a day before this report is written (and luckily I am able to refactor it fast and bugless, hopefully). The next lesson that I have learnt is that the importance of communication. As mentioned in the failures above, I should have communicate earlier with Anthony to design such that the implementation of the game model is more multiplayer-friendly. Lastly, I also realized that to make a good product, a good team, where everyone supports each other, is needed. Although usually in a team, people do different parts, communication is still needed in some of the areas, especially during the integration, where a lot of communication and discussion usually happening. If the discussion happened is supportive and constructive, everyone will be more motivated in doing things and perform better.

Known Bugs and Limitations

Some of the bugs mentioned are already fixed, or we have found solutions to fix those bugs.

Bugs

XCode Crashes when applying Texture to Color Sprite

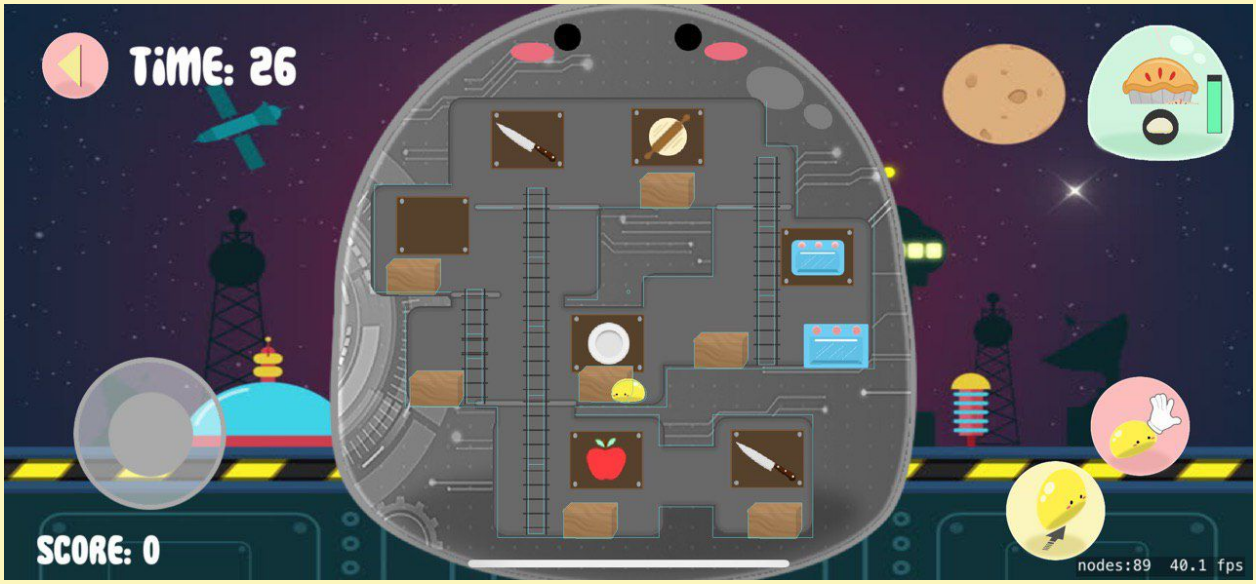
This bug does not come from our code, but it's an existing XCode bug which can be found on the Apple Developer Forums.
When the user tries to bring in a Coloured Sprite into the Scene, and attempts to change the texture, XCode may crash without any reason. Apparently, this bug has been going on for a few years but it has yet to be fixed. Hence, the only way for us to apply a texture to the color sprite and apply physics body is through code only. Due to this bug, we are unable to use other approach other than manually coding it out and applying the sprite and the physics body to it, even though it is more cumbersome.

Weird physics body behaviour when using an edge loop

When there is a gap in between the edge loop that is being formed, the physics body of the character (slime) will fall through it and or there will be a jerking effect.

As seen from the image above, when the character attempts to walk on the circled parts, the slime will either fall through the rectangular parts or the

character will jerk/fly out of place. This will thus cause undesirable effects to the movement of the character.
After some time, we realized that the falling through and jerking effect will only happen if we use edge loop to form those areas. Hence, we have to change the way we draw the border and the blocked areas to prevent this from happening. The new physics border can be seen from the image below.



This will thus cause the slime to work properly now.

Objects are seen to be duplicated for a short amount of time

Taking an object at the same time in multiplayer will cause the item to be duplicated momentarily in the screen. This will then correct itself due to automatic accounting by transactions implemented. This can be solved by going lower level into the implementation of the interact function in single player and handling the race conditions before the command to take item fires.

Limitations

Flexibility of creating a physics body vs edge body

As mentioned in the Performance point earlier in the proposal, there are different type of physics body used in XCode.
To create a physics body based off the texture is easier, as there are methods to create a body from the texture - *init(texture: SKTexture, size: CGFloat)* or *init(texture: SKTexture, alphaThreshold: Float, size: CGSize)*. Hence, it is easy to create this physics body.

However, there is no such function if we were to create an edge body. The edge body can be easily created if it's the regular quadrilateral shape, meaning a square/rectangle. Once the number of points goes above 4, it will be more tedious and cumbersome to create an edge body. The way to create edge body will be to plot from point to point and match it up with the edge loop.
Currently, there is no better way in XCode to do up an edge body easily by reading off the texture, hence we have to make do with it and make sure not to come up with a complicated shape or design.

Appendix

Test cases

UI Tests

- Title Screen
    - Tapping the Play button should lead to the Play Mode screen if the user has created a character, or the character creation screen if he has not.
    - Tapping the Credits button should lead to the Credits screen.
    - Tapping the Settings button should lead to the Settings screen.
    - If the user has created a character, there should be an overview box on the top of the screen. Tapping on this box should lead to the character customization scene.
- Character Creation Screen
    - Tapping on the input box should open up a modalized input field for the user to type in their character's name. When the modalized input field is closed, the input value should be copied to the real input field.
    - Tapping on the left and right arrows of the colour selection should change the slime's colour to one of 9 different base colours and expressions.
    - When the OK button is pressed, the character should be created and saved in the user's persistent data. The game should then segue to the Play Mode screen.
- Character Customization Screen
    - On the left should be a view of the character's current appearance.
    - On the right should be a brief overview of the character (name, level, exp bar), as well as 3 buttons which show what cosmetics the character is currently wearing.
    - Tapping on a cosmetic button should transition the box to a selection screen where the user is able to pick other cosmetics to put on.

        - Changing the cosmetic should immediately be saved to the user's persistent memory. Restarting the game when a cosmetic has been changed should reflect that change.
- Credits Screen
    - Tapping on the back button should lead to the Title screen.
- Settings Screen
    - Tapping on the back button should lead to the Title screen.
- Play Mode screen
    - Tapping on the up caret button should lead to the Title Screen
    - Tapping on the Single-player Mode button should lead to the Level Selection screen.
    - Tapping on the Multiplayer Mode button should lead to the Host/Join room screen.
- Host/Join room Screen
    - Tapping on the Up caret button should lead to the Play Mode screen
    - Tapping on the Join Room button should lead to a screen where you can enter the room code.
    - Tapping on the Host Room button should lead to the Multiplayer Lobby screen, with a randomly generated room code.
- Join Room screen
    - Pressing the up caret button should lead to the Host/Join room screen
    - The user should be able to enter a 6-digit numerical room code using the on-screen number pad.
    - Pressing the backspace button should delete 1 character off the room code entry
    - Pressing the clear button should clear all numbers off the screen.
    - When the 6 digits are all filled up
        - If the code is valid, the scene should transition to the Multiplayer Lobby screen with that room code.
        - If the code is invalid, an alert should show up, alerting the user that the room code is invalid.
        - If the room game has started, an alert should also show up, telling the user that the game has started for that particular room
        - If the room is full, the user should be alerted that the room is already full
- Multiplayer Lobby screen

    - The screen should show the lobby's room code.
    - The screen should show the players who are connected in the lobby, their names and their levels, as well as their character portrait.
    - The screen should show a Start button only to the host of the room which starts the game with the players in the room.
    - The screen should show a level preview to all players of the room.
    - The screen should show a 'Change' button to only the host, which allows him to change the stage that the room plays in.
        - A modal should be opened when the change button is pressed, allowing the host to select the map from the available multiplayer maps.
        - When a level is selected, the modal should close and the map should be updated for all players.
    - The screen should not display any players who have since disconnected from the room.
    - Pressing the back button should alert the user confirming their action:
        - If the user confirms the exit, the user will be taken to the multiplayer menu screen, if the user is host, the room will be closed
        - If the user does not confirm the exit, the alert dialog will close
    - If the host leaves the room either by leaving the screen or disconnecting, all players should be disconnected from that room.
- Level Select Screen
    - Scrolling through the list of levels should snap to the closest level.
    - The snapped level component should be at the center of the screen.
    - Tapping play should lead to the Loading screen.
    - The best score should be shown for each level.
- Loading Screen
    - A relevant tutorial should be shown for the stage.
    - Tapping anywhere on the screen should advance the tutorial.
    - Once all tutorial screens are shown, it should segue to the game screen.

Game Scene Test

- Analog Joystick
    - Touching the joystick should have no effect to the slime (sprite)
    - Touching and dragging the joystick **for a very small distance** towards the any direction should see the slime moving **slowly** towards the desired direction
    - Touching and dragging the joystick **for a very small distance** the up direction should have no effect to the slime (sprite)
    - Touching and dragging the joystick **for a very small distance** towards down direction should have no effect to the slime (sprite)
    - Touching and dragging the joystick **for a large distance** towards the any direction should see the slime moving **faster** towards the desired direction
    - Touching and dragging the joystick **for a very large distance** the up direction should have no effect to the slime (sprite)
    - Touching and dragging the joystick **for a very large distance** towards down direction should have no effect to the slime (sprite)
- Jump button
    - Tapping on the jump button once should expect the slime to move upwards
    - Holding on the jump button should expect the slime to move upwards only once, and it should stop moving thereafter
    - Tapping on the jump button multiple times should expect the slime to move upwards multiple times
- Interact button
    - Tapping on the interact button when the slime is not any of the station should not have anything happening
    - Tapping on the interact button when the slime is near the **serve station**
        - Without a plate: Nothing should happen
        - With a plate:
            - Recipe matched with one of the orders in the order queue should expect the score to increase
                - A new order should pop out accordingly
            - Recipe does not match with the orders in the order queue should expect the score to decrease
    - Tapping on the interact button when the slime is near the ingredient stations

        - If the slime is not carrying anything, should expect the ingredient to appear depending on the station on top of the slim
        - If the slime is carrying something already, should not expect any overriding of object.
    - Tapping on the interact button when the slime is near the cooking equipment stations (e.g. oven, chopping etc.)
        - If the slime is carrying an ingredient
            - If the ingredient is appropriate/can be processed at the food station, should expect the the ingredient to disappear from the top of the slime's head and will appear on top/in the cooking equipment station.
            - If the player taps multiple times afterwards, should expect the newly processed ingredient to appear on top of the slime's head
        - If the slime is not carrying an ingredient
            - Nothing should happen
- Back Button
    - Tapping on the back button should have a pop up screen/prefab that will ask the users whether they want to go back to the main menu or resume game, and the game will be paused
        - Tapping on going back to the main menu should expect the users to go back to the previous screen
        - Tapping on the resume game should expect the menu prefab to close and the game shall continue
    - In multiplayer, there is no pop-up and the user should immediately be redirected to the multiplayer menu screen. If the user is host and presses this button, the other players will be disconnected from the game as well.
- Test collisions
    - Collision between slime and the boundary of the play area
        - If the slime moves towards the boundary of the play area via the joystick, the slime should be blocked and will be unable to pass through
- For multiplayer only:
    - All the above interact tests should be valid when a player interacts with an object and is observed on another player's screen
    - Notification system:

- The notification system is only for multiplayer mode. Single player mode should not see any of these notifications
- A new notification should pop up when a new order has arrived
- A new notification should pop up when an order times out
- A new notification should pop up when any user submits an order and it is correct
- A new notification should pop up when any user submits an order and it is incorrect
- A new notification should pop up when the timer reaches 60 seconds
- A new notification should pop up when the timer reaches 10 seconds

Detailed Schedule + Task list

We will be referring to the previous schedule and task list that we have submitted previously, whereby the ones that are completed are mark with green.

The table below will also show the detailed breakdown of the tasks that each person is supposed to do.

Date(s)	Expected Implementation	Person in charge
10 Mar	Submission of Project Proposal 1	All
11 Mar ~ 16 Mar	The team will be doing a mock-up of game prototype and the projected code structure that will be mentioned in the later part of this proposal (point 7). Targeted date to have a rough mock-up/prototype of the game to be on 15 March. The game design and art will also be in progress during this period of time to allow the team to have a rough feel of the game.	All
17 Mar	Completion of game overview, balance and design plans	All
18 Mar ~ 23 Mar	Incrementally improving the prototype. The design/theme of the game should be put into place at this point of time. The team will spend the time over the weekend to finish up the progress report 1 that is to be submitted on 24 March.	All
24 Mar	Completion of boilerplate code (Models, Controllers, etc.) preliminary design document submission (Progress report 1)	All
25 Mar ~	ORIGINAL: Improving on the game	All

30 Mar	balancing and conducting UAT. Review of code structure if needed at this point of time. <u>CURRENT:</u> Implementation of networking and game mechanics based off the game design using the code architecture that the team has come up with. The specific details are shown as below.	
	Networking ● Integrating the networking abstraction to the application. ● Finalisation of models and states required for Room and Game scenes in the Firebase realtime database ● Finish up all the required methods for the networking abstraction class. ● Finish up the anonymous authentication process and integrate into the game.	Anthony
	UI Implementation ● General utility UI components should be made. ○ Modals, alerts, buttons, icon buttons ● UI elements should be able to interact with underlying user data to write and read persistent data. ● Dependency injection between view controllers should be made possible without the overuse of global variables / context.	Gabriel
	Game Design/Logic Implementation ● Implement all the logic in the game (since the stage started until the end game condition), including order generation,	Hui Qi & Anthony & Samuel

	cooking process, and serving process. • Implement multiplayer system, where there is interaction between the model to the network and to make the network is the single source of truth. • Playtesting to further balancing the design of the game and updating the game design if needed	
	Game UI Refinements/Drawings/Assets • Game Assets needed for the first level • Increase the resolution of the assets	Hui Qi
	Testing • Manual testing should be done.	All
31 Mar	Completion of beta product.	All
1 Apr ~ 6 Apr	Play-testing and improving/implementation of any additional game features. The team will spend the time over the weekend to finish up the progress report that is to be submitted on 7 April. Designing of Marketing Collaterals will begin	All
	Networking • Improvement of connection latency • Integrating networking into game • Implementing game networking models and processes • Improving efficiency	Anthony

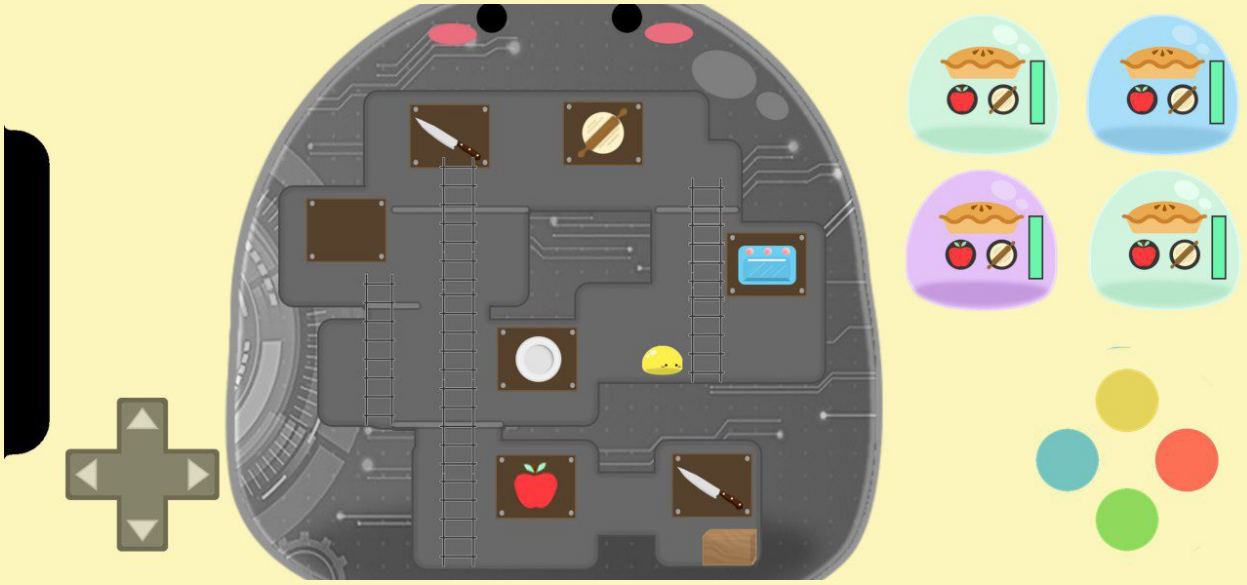
	UI Implementation ● Refinement of UI ● Patching up buggy areas	Gabriel
	Game Design/Logic Implementation ● The code should be extensible enough for us to create more stages if needed, or to refine the logic and clean up the code structure	Hui Qi & Samuel
	Testing ● Manual testing should be done	All
	Designing of Marketing Collaterals ● Marketing collaterals for STePS such as poster, shirts etc. if needed	Hui Qi
7 Apr	Progress Report 2	All
7 Apr ~ 16 Apr	The team will be conducting intensive play-testing during this period of time, in order to get the best game balance as much as possible, and to sieve out any possible errors and bugs. Completion of testing phase and buffer to touch up or fix certain parts of the game features, completion of application. Bug fixes and minor adjustments will be done during this period of time. Marketing collaterals and designing of the poster for STePS will also be completed and printed out. **NEW ADDITIONAL DETAILS BREAKDOWN:** ● Intense testing of networking ● Intense play-testing of the levels that we have implemented ● Implementation of sound effects ● Clean up the rest of the game	All

	logic ○ The basic game logic have been completed, now we are left we cleaning up of UI and UX to match up with the game logic. We are mainly left with cleaning up since most of the functions and logic are already implemented. Things to clean up are as follow: ■ Fixing of game flow for the different buttons such as replay or returning to main menu ■ Refinement/Refactoring of code along the way if needed (This will be updated after we have the meeting with the Professor, to see whether our current architecture/model works well) ■ Fixing the game flow for users while navigating throughout the game ■ Debugging if there are any bugs reported/found during playtesting and implementation ■ Fix UIViewController display for the UI layout (Menu) ■ Additional UI needed for certain areas for the gameflow/game scene needs to be implemented ○ Adding in new level for	

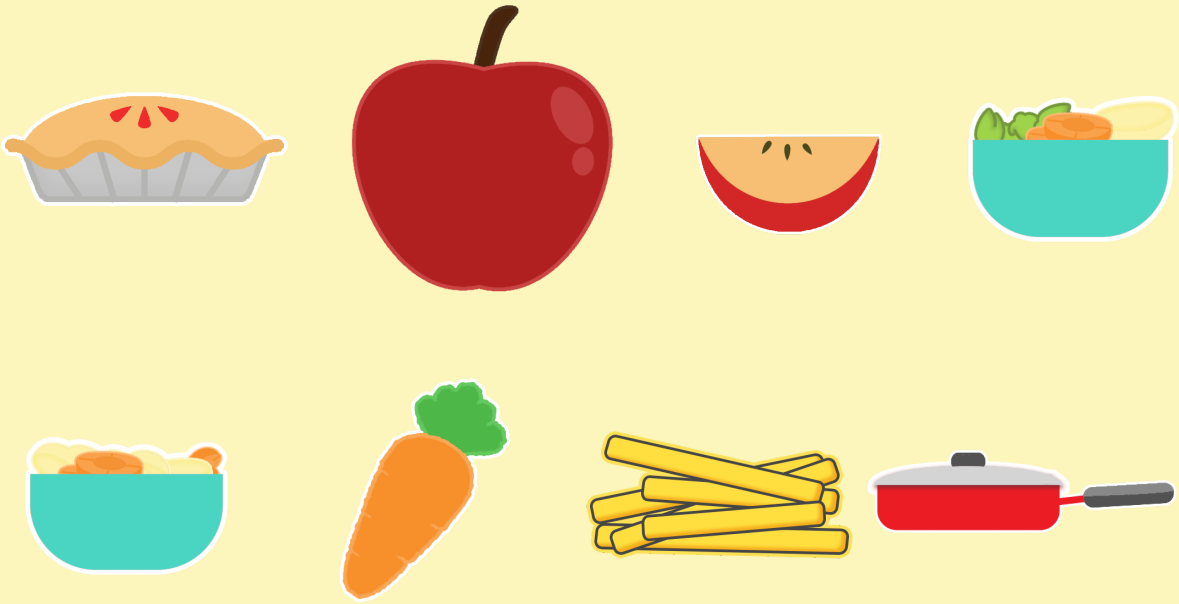
	multiplayer (new level design needed) ○ Optimization (depending on how long is the loading time of the game on different devices), and the FPS rate ● Working on Marketing collaterals required for STePS (e.g. Poster)	
17 Apr	STePS	All
18 Apr ~ 20 Apr	The team will be spending the time to complete the final project report. The team will also start preparing for the final project presentation.	All
21 Apr	Final project report submission	All
22 Apr ~ 27 Apr	Final project presentation	All

GUI sketches/screenshot

Mockups



Final Report progress

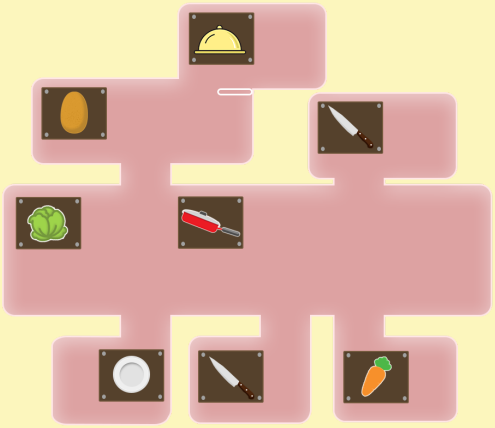


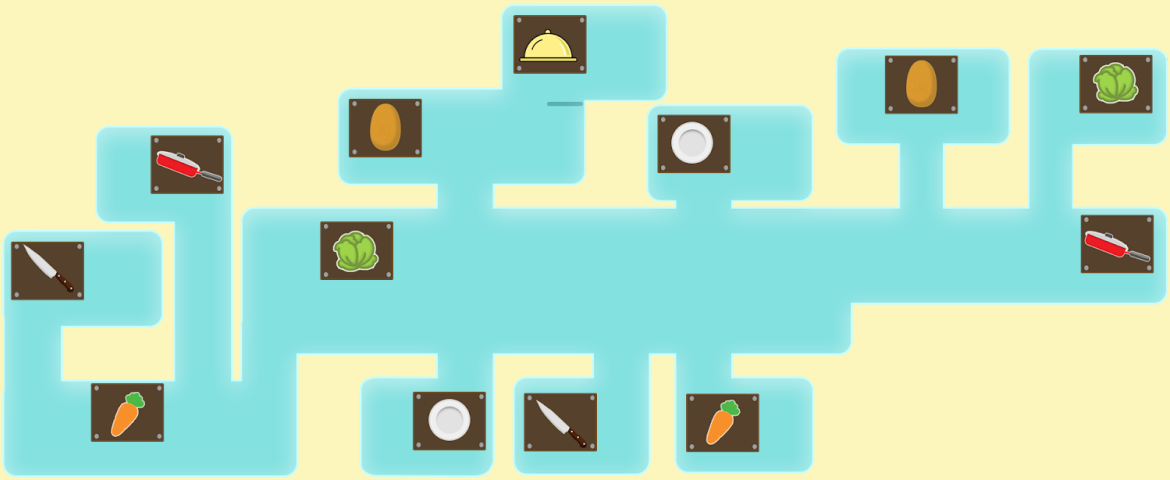




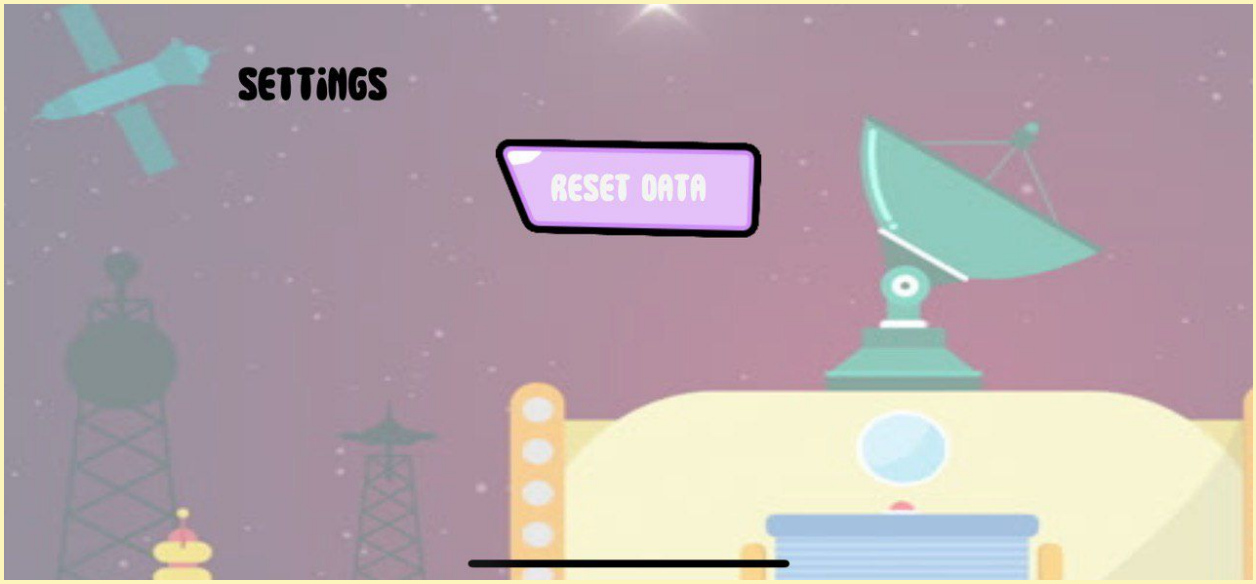


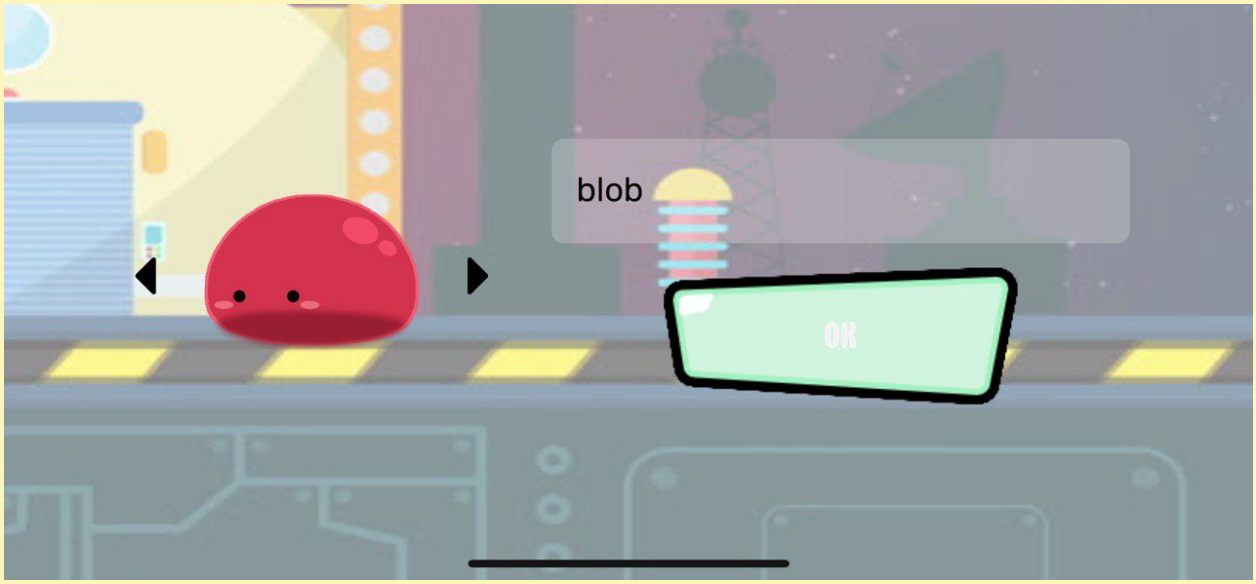


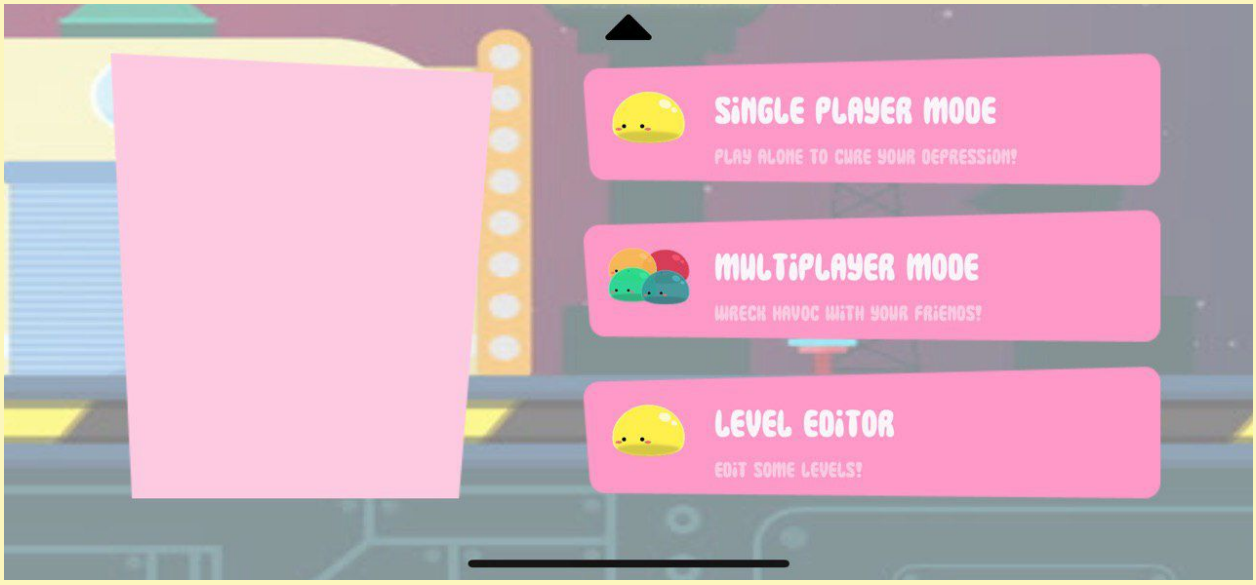


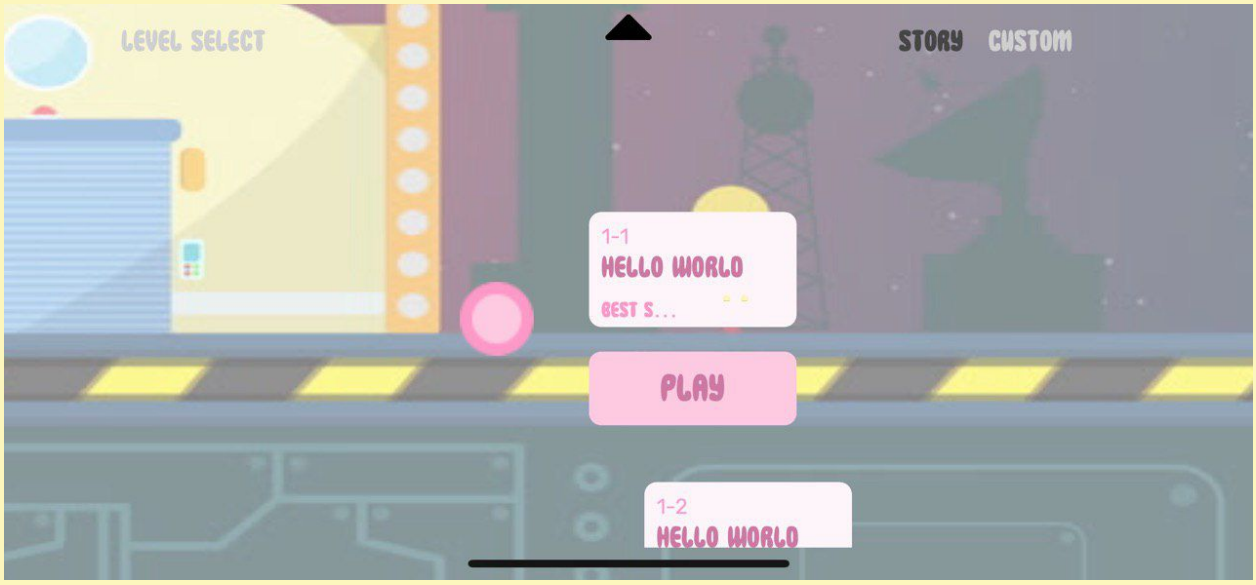


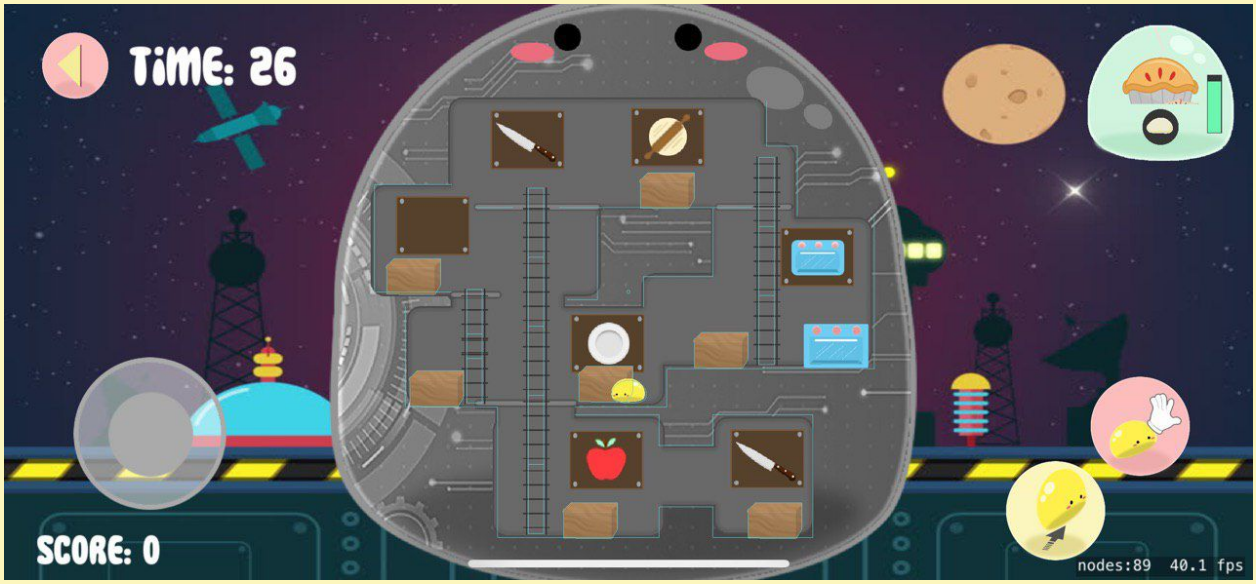
Progress Report 2 progress


SETTINGS
RESET DATA

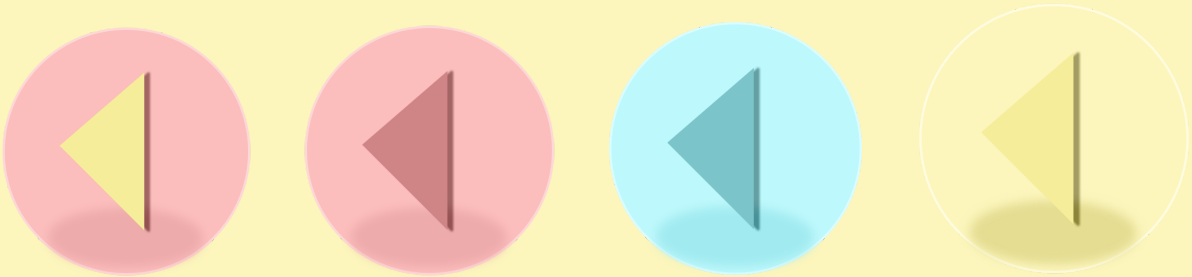

blob
OK


SINGLE PLAYER MODE
PLAY ALONE TO CURE YOUR DEPRESSION!
MULTIPLAYER MODE
WRECK HAVOC WITH YOUR FRIENDS!
LEVEL EDITOR
EDIT SOME LEVELS!

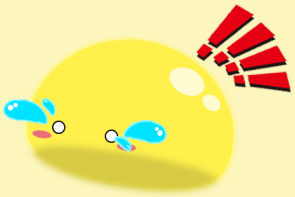

LEVEL SELECT
STORY
CUSTOM
1-1
HELLO WORLD
BEST S...
PLAY
1-2
HELLO WORLD

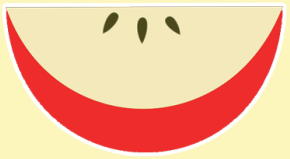

TIME: 26
SCORE: 0
nodes:89 40.1 fps


GAME OVER
TRY AGAIN
RETURN TO MAIN MENU



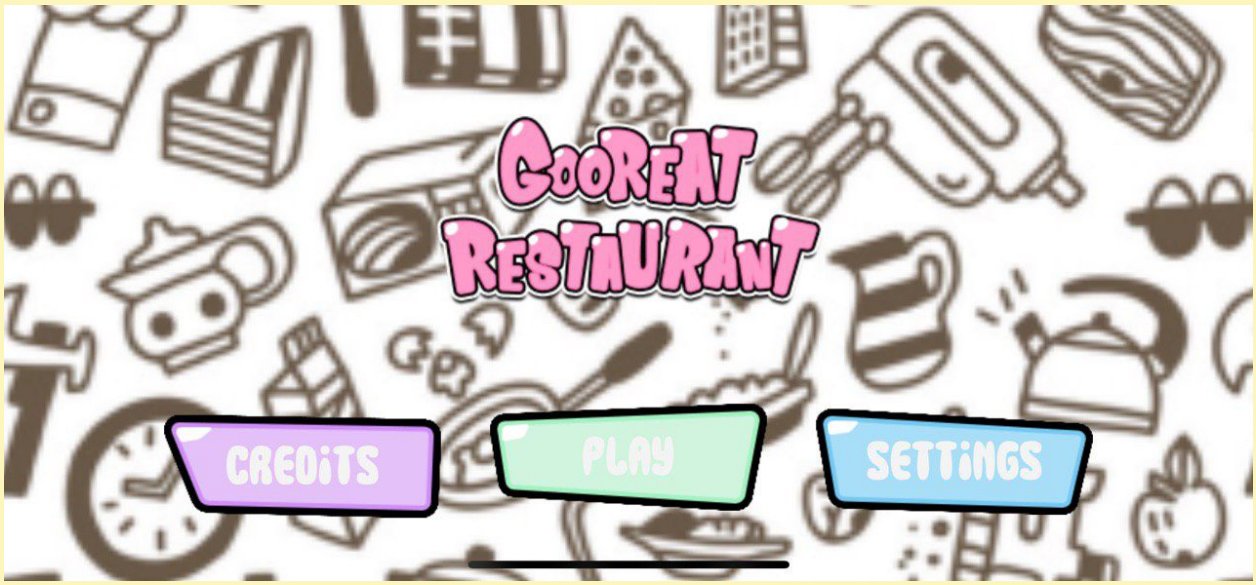
RETURN TO MAIN MENU

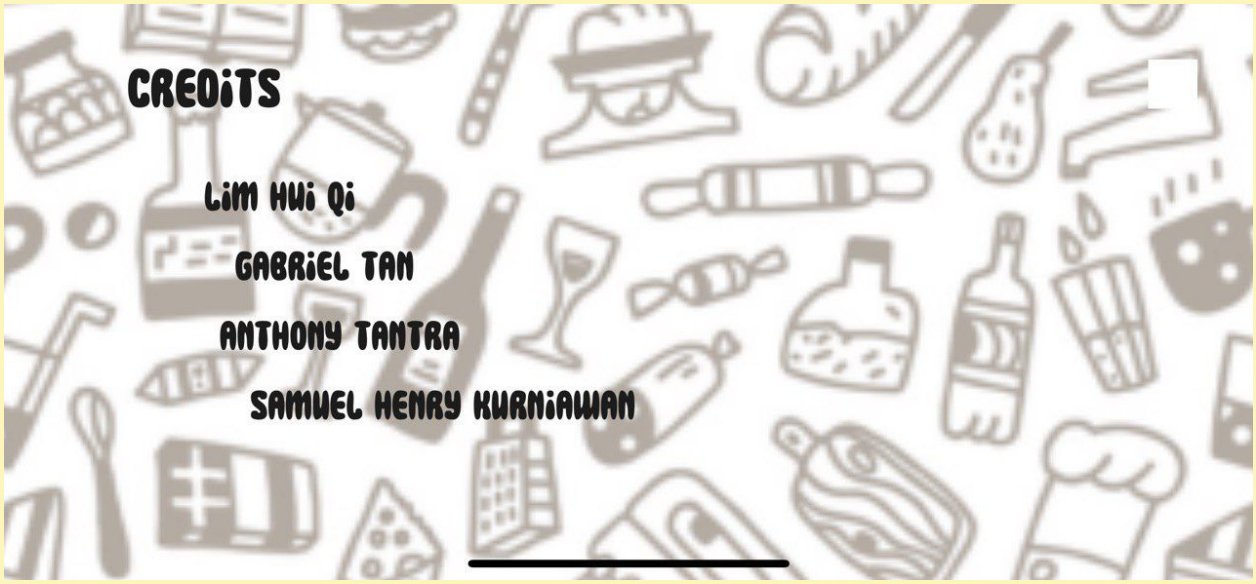


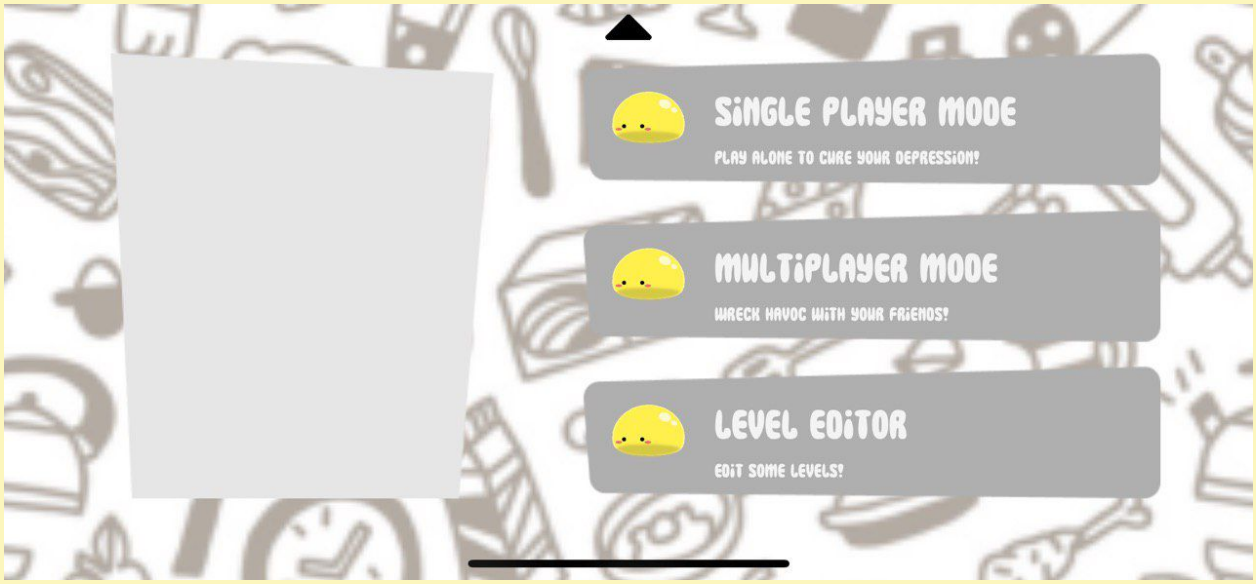


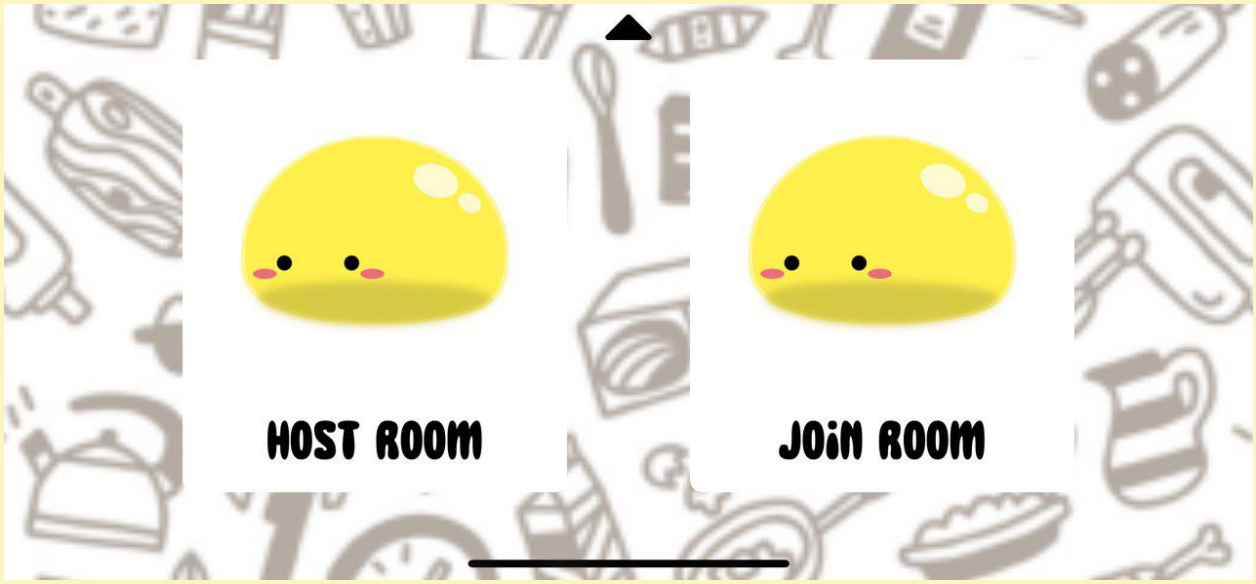
TRY AGAIN

Previous submitted progress



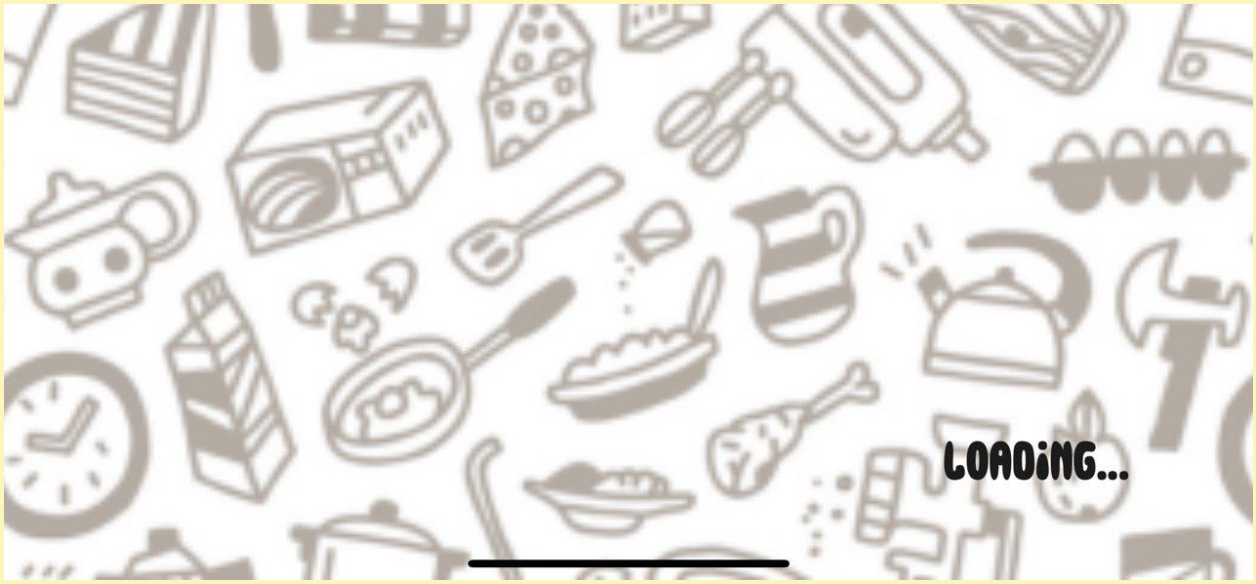


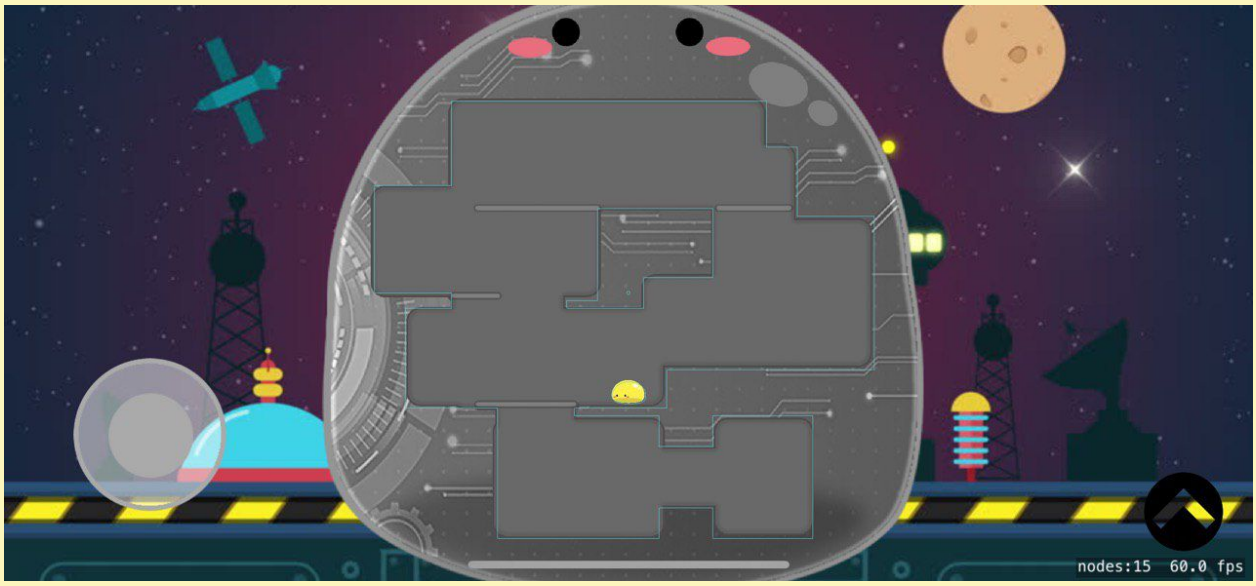

SINGLE PLAYER MODE
PLAY ALONE TO CURE YOUR DEPRESSION!
MULTIPLAYER MODE
WRECK HAVOC WITH YOUR FRIENDS!
LEVEL EDITOR
EDIT SOME LEVELS!


HOST ROOM
JOIN ROOM

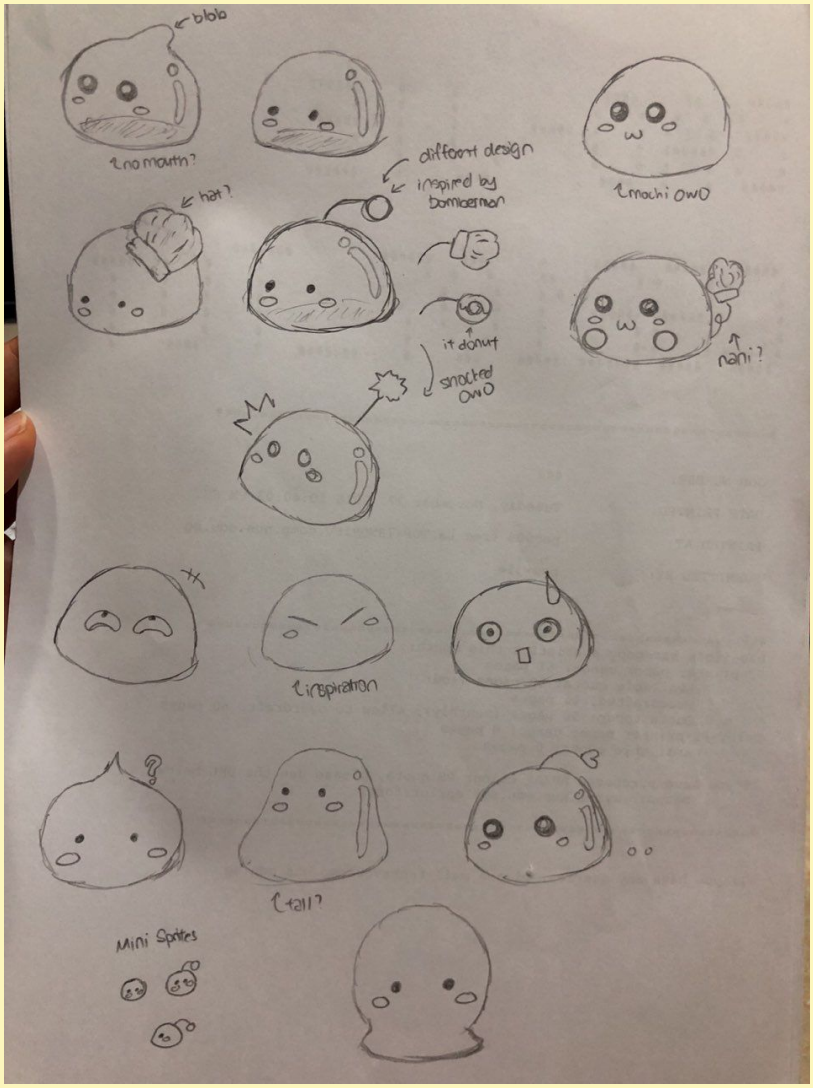

UH-OH!
This feature is coming soon(TM).
OK

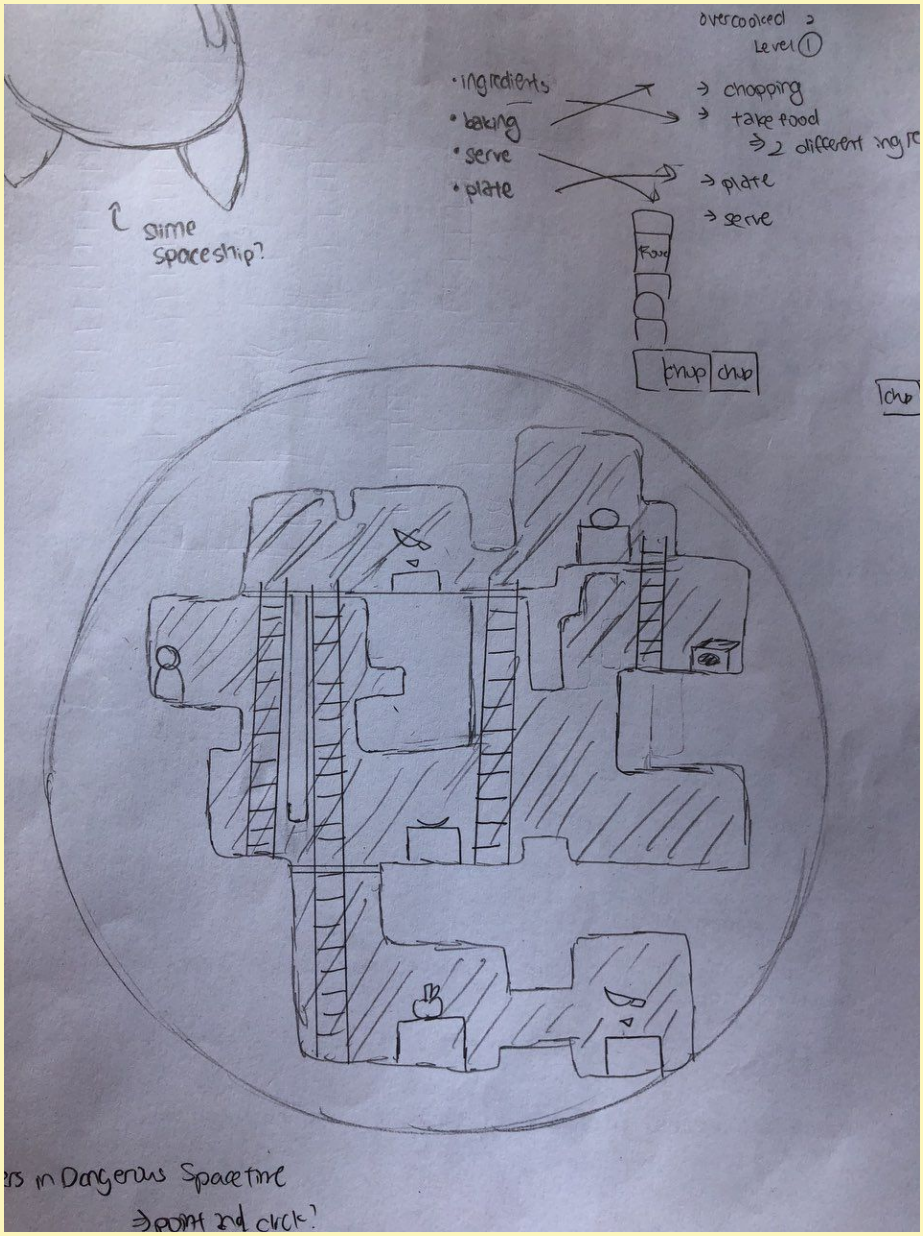

LEVEL SELECT
STORY
1-1
HELLO WORLD
BEST SCORE
100
RATING
PLAY
1-2
HELLO WORLD
BEST SCORE

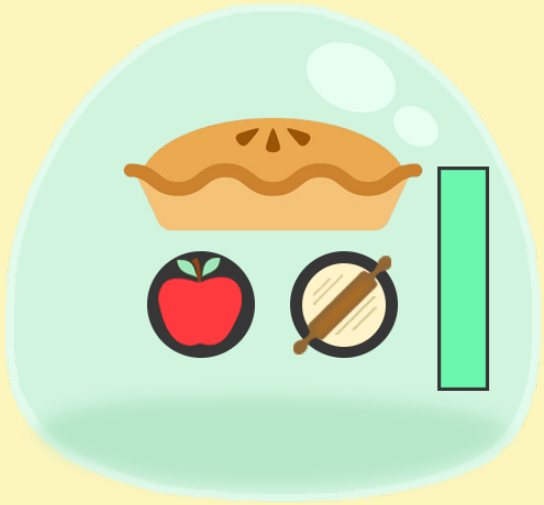

LOADiNG...

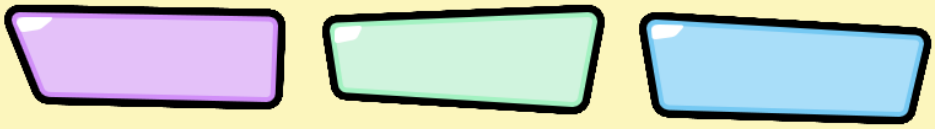

nodes:15 60.0 fps

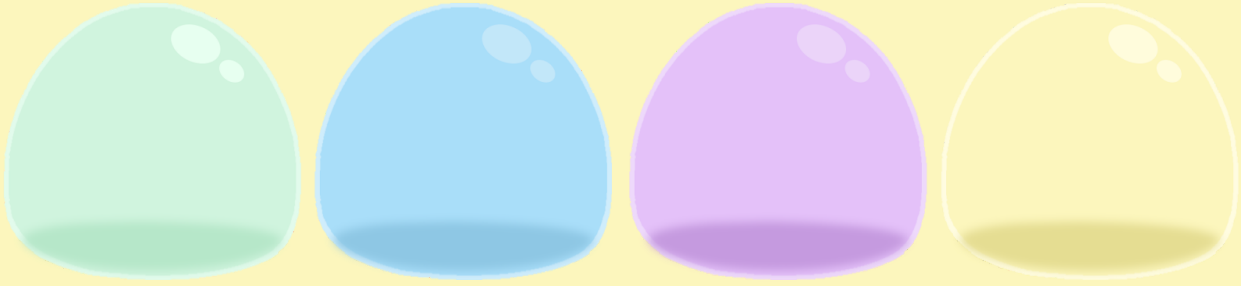
Art assets/mockups

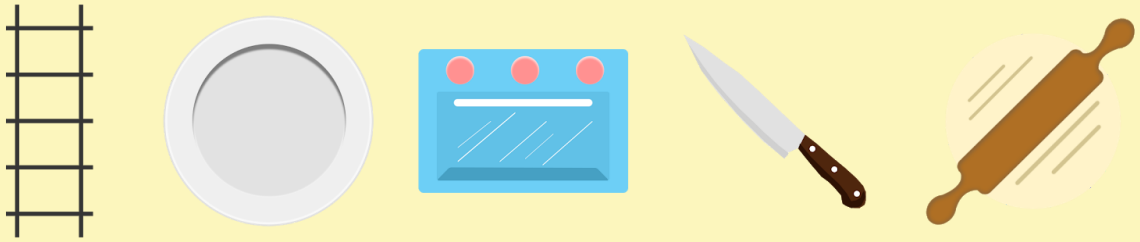



slime spaceship?
overcooked
Level 1
• ingredients
• baking
• serve
• plate
→ chopping
→ take food
→ plate
→ serve
chop
chop
chop
Dangerous Spacetime









Issues still unresolved

- UICollectionView display layout for the Menu Prefab
    - Displaying the menu orders (includes animation)
    - Displaying the ingredients needed for the recipe

Specifications

NIL