Dish It Out

Artists: Arfa, Nelson
Musicians: Abdi, Shrey
Programmers: Linh, Sheryl

Overview:

A digital card game about cooking ingredients and recipes. Players compete against one another to collect ingredients cards from the market to make dishes from around the world and gain points from doing so. This strategic game allows players to carefully and quickly collect ingredients and create recipes so that they can reach 10 points first to win.

In this 2D game, players will select a cuisine style then collect ingredients accordingly to complete a series of dishes. The first player to collect a certain number of points wins. Points value vary based on the tier the recipes are in. Easier recipes require less ingredients but have fewer points while harder recipes require more ingredients and are valued with higher points. Ingredients will be presented as cards and the UI will resemble a board game where players sit across from each other and be able to observe others' cards/actions. This turn-based strategy card game forces players to strategize which ingredients to get and how that helps them get the recipes they want as their opponent could be competing for the same recipe.

Game Rules:

First, select a region to begin. Two players take turns to play. When it's your turn, you will have three options:

- select 3 different ingredient cards
- select 1 ingredient card to get 2 of the same ingredient
- or purchase a recipe card

To buy a recipe card, you must already have all of the ingredients that the recipe calls for, this will be listed at the bottom left of each recipe card

On the left of each recipe card, you'll find listed the ingredients you need to have in your inventory to be able to buy that recipe.

Each recipe card is worth a certain number of points. Keep an eye out for this on the top right corner of each card.

To win, you must reach a total of 10 points before your opponent does.

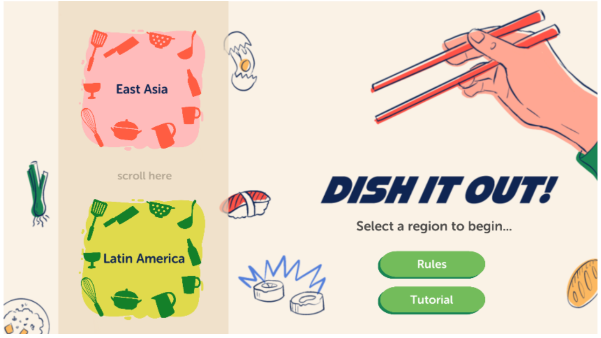
Code Structure:

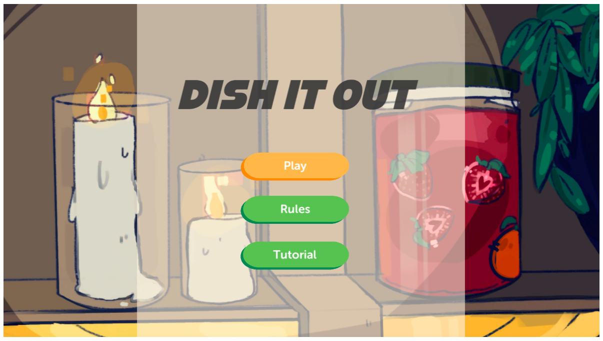
- Title screen scene
    - Region selection
    - Tutorial videos
- Game scene
    - Player turns
    - Shuffle recipe cards
    - Inventory
        - Player score
        - Player ingredients
        - Player recipes
    - Ingredient
        - Add ingredient to inventory function
    - Recipe
        - Purchase recipe
            - Check ingredients needed function
            - Deduct ingredients from inventory function
            - Add recipe to inventory function
    - Recipe Scriptable Objects
        - Properties of each recipe (name, tier, points, ingredients, etc)

Programming Goals:

Title Screen

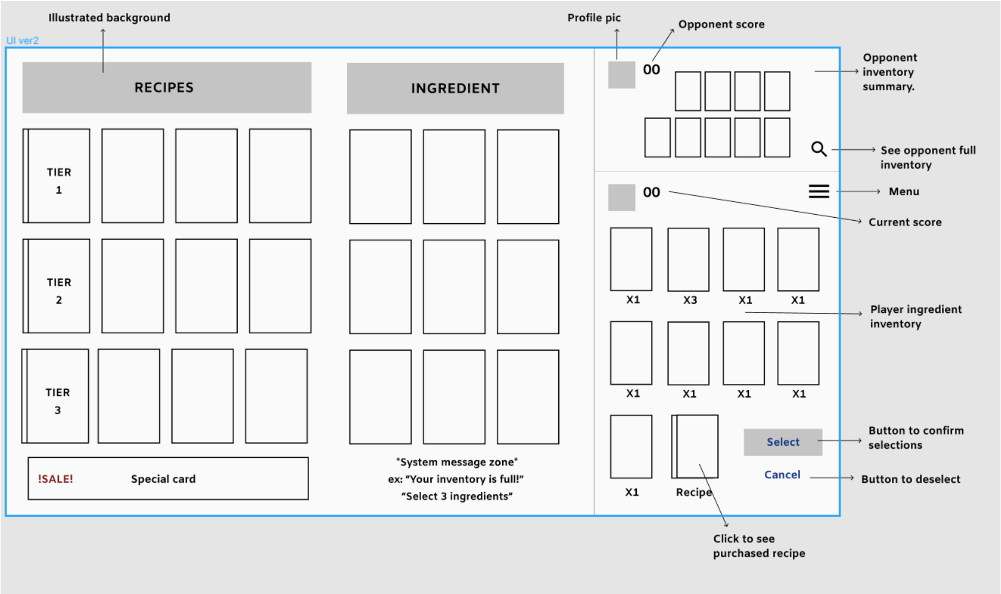
The minimum goal for the title screen was the region select, rules page, and tutorial. With some title screen reiteration, the region select scroll area became an animated panel for players to pick what region they're interested in playing. The minimalist style allowed for creative ways to juice up the title screen with animations. The title screen also had to go through different design iterations. The old format was more geared towards mobile as the scroll area allowed players to swipe through to see the different regions offered, however, with the placeholder art replaced, the screen had to be redesigned with download play in mind.

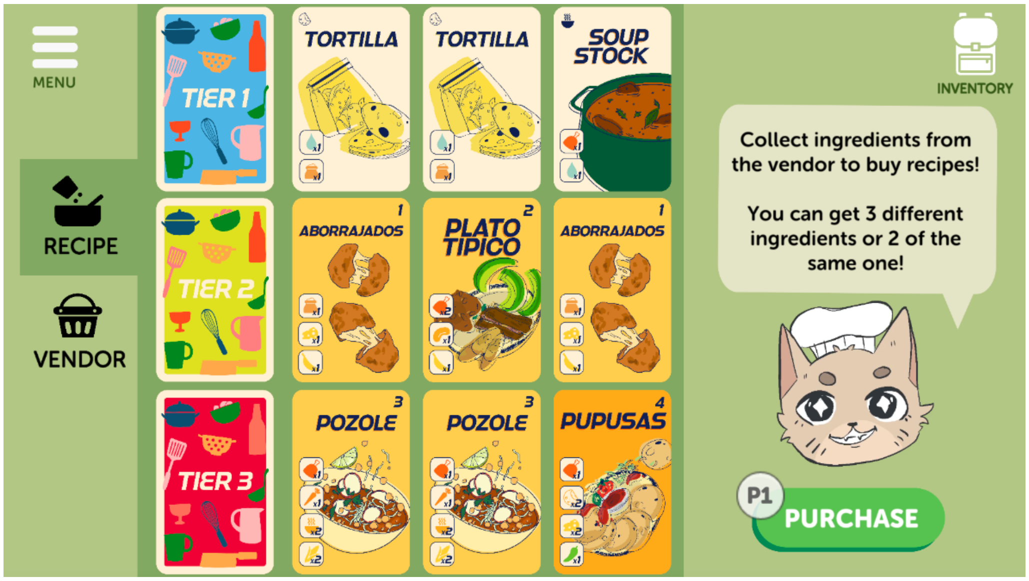




Game Scene

The technical goals for the game were a panel of selectable ingredients, a panel of selectable recipes, an interactive motherly cat guiding the player, an inventory panel, special cards, and a main menu. However, the project was over-scoped and the special card feature had to be cut in order to prevent the quality from dropping. The game also had to go through multiple iterations of UI layout redesign since the game was originally intended to be for mobile. This meant cards had to be big for readability and the UI layout had to be simplified and compartmentalized instead of everything shown all at once which contributed to clutter confusion.



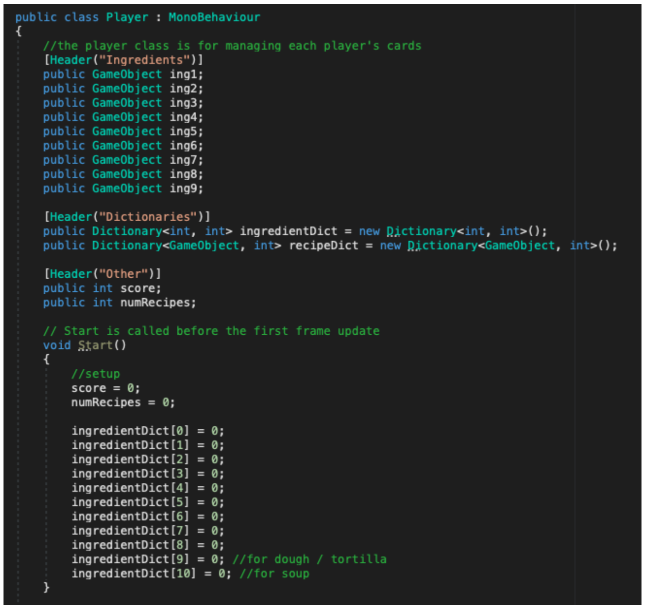


Collect ingredients from the vendor to buy recipes!

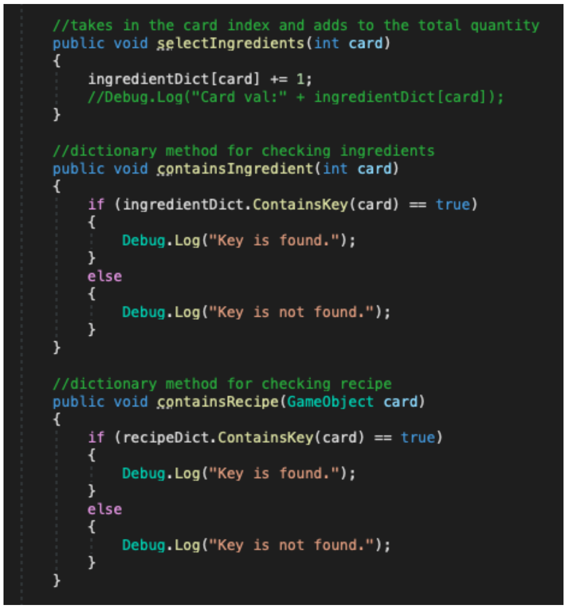
You can get 3 different ingredients or 2 of the same one!

Code Callouts:

Vital scripts consist of inventory, UIController, Recipe, and RecipeInfo. The inventory was managed by two dictionaries: ingredients and recipes. The dictionaries were under the player class which we could make as many instances as needed. Simple dictionary methods were made to manage the inventory. The player class also kept track of scoring and the total number of recipes players have accumulated. This script is not groundbreaking, as it just uses simple dictionary and dictionary methods, but we utilized indices instead of strings to prevent hardcoding ingredients and recipes names. We wanted to reuse and apply this script with other regional dishes so this was the best solution we came up with.

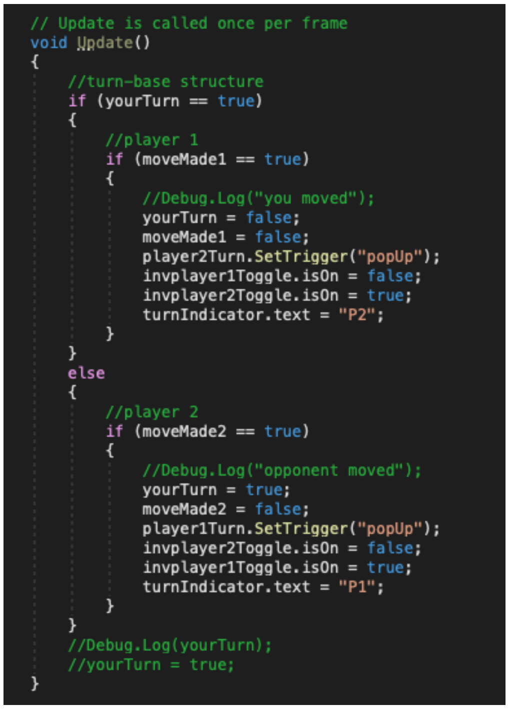


```
public class Player : MonoBehaviour
{
    //the player class is for managing each player's cards
    [Header("Ingredients")]
    public GameObject ing1;
    public GameObject ing2;
    public GameObject ing3;
    public GameObject ing4;
    public GameObject ing5;
    public GameObject ing6;
    public GameObject ing7;
    public GameObject ing8;
    public GameObject ing9;

    [Header("Dictionaries")]
    public Dictionary<int, int> ingredientDict = new Dictionary<int, int>();
    public Dictionary<GameObject, int> recipeDict = new Dictionary<GameObject, int>();

    [Header("Other")]
    public int score;
    public int numRecipes;

    // Start is called before the first frame update
    void Start()
    {
        //setup
        score = 0;
        numRecipes = 0;

        ingredientDict[0] = 0;
        ingredientDict[1] = 0;
        ingredientDict[2] = 0;
        ingredientDict[3] = 0;
        ingredientDict[4] = 0;
        ingredientDict[5] = 0;
        ingredientDict[6] = 0;
        ingredientDict[7] = 0;
        ingredientDict[8] = 0;
        ingredientDict[9] = 0; //for dough / tortilla
        ingredientDict[10] = 0; //for soup
    }
```



```
//takes in the card index and adds to the total quantity
public void selectIngredients(int card)
{
    ingredientDict[card] += 1;
    //Debug.Log("Card val:" + ingredientDict[card]);
}

//dictionary method for checking ingredients
public void containsIngredient(int card)
{
    if (ingredientDict.ContainsKey(card) == true)
    {
        Debug.Log("Key is found.");
    }
    else
    {
        Debug.Log("Key is not found.");
    }
}

//dictionary method for checking recipe
public void containsRecipe(GameObject card)
{
    if (recipeDict.ContainsKey(card) == true)
    {
        Debug.Log("Key is found.");
    }
    else
    {
        Debug.Log("Key is not found.");
    }
}
```

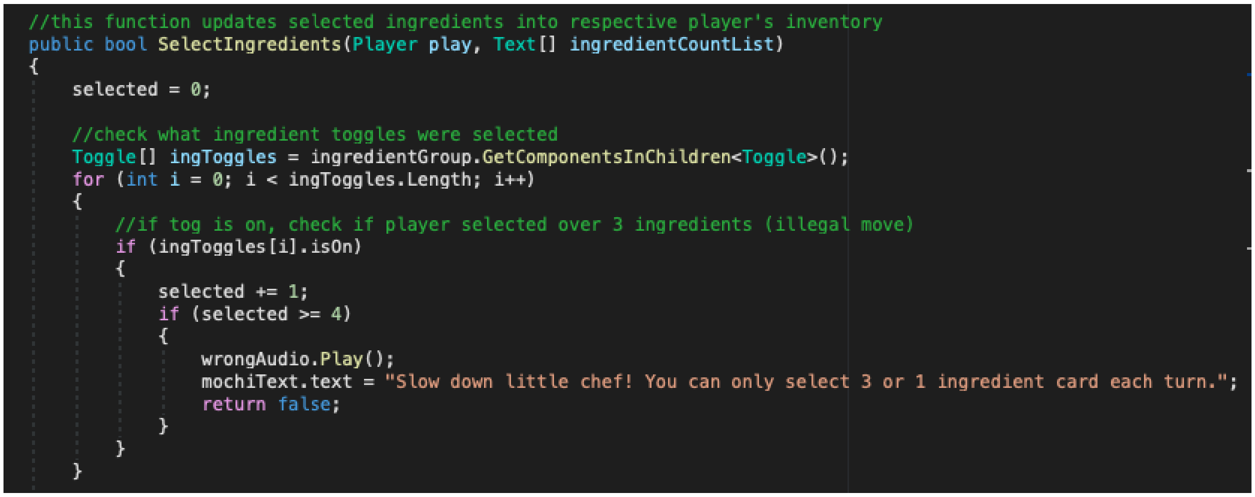
The UIController class is the main skeleton of the game. Because the game is all screen interactions, the UIController takes care of all calls relating to the UI elements. All the cards on-screen are toggles and are part of toggle groups. Since the interactions are from buttons, the turn-based structure is also located in this script. The structure ensures that the player makes a valid move before changing to the other player's turn.



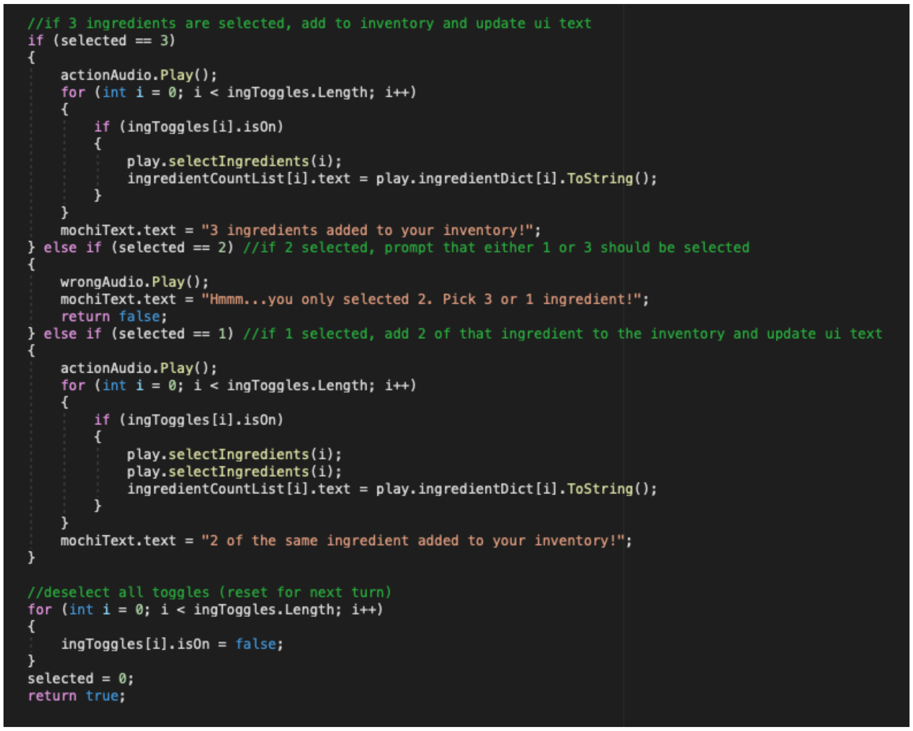
```
// Update is called once per frame
void Update()
{
    //turn-base structure
    if (yourTurn == true)
    {
        //player 1
        if (moveMade1 == true)
        {
            //Debug.Log("you moved");
            yourTurn = false;
            moveMade1 = false;
            player2Turn.SetTrigger("popUp");
            invplayer1Toggle.isOn = false;
            invplayer2Toggle.isOn = true;
            turnIndicator.text = "P2";
        }
    }
    else
    {
        //player 2
        if (moveMade2 == true)
        {
            //Debug.Log("opponent moved");
            yourTurn = true;
            moveMade2 = false;
            player1Turn.SetTrigger("popUp");
            invplayer2Toggle.isOn = false;
            invplayer1Toggle.isOn = true;
            turnIndicator.text = "P1";
        }
    }
    //Debug.Log(yourTurn);
    //yourTurn = true;
}
```

SelectButtonPressed, a UIController function, is called from the onClick of the Select Button in the ingredients panel. It first checks whose turn it is before calling the SelectIngredient function which returns a boolean to denote whether or not it's the ending of the turn. Until moveMade is true, the turn will not change to the next player.

SelectIngredients, a function that returns a bool for whether a valid move was made, first checks what ingredients are selected and validate if it's a legal move. The game rules state that players can pick up 3 different ingredients or 2 of the same ingredients. In terms of programming, the player has to select 3 toggles in which each ingredient will be added to the inventory or 1 toggle in which that one ingredient will be added to the inventory twice. Having over 3 ingredients selected or selecting 2 ingredients toggles are illegal moves.



```
//this function updates selected ingredients into respective player's inventory
public bool SelectIngredients(Player play, Text[] ingredientCountList)
{
    selected = 0;

    //check what ingredient toggles were selected
    Toggle[] ingToggles = ingredientGroup.GetComponentsInChildren<Toggle>();
    for (int i = 0; i < ingToggles.Length; i++)
    {
        //if tog is on, check if player selected over 3 ingredients (illegal move)
        if (ingToggles[i].isOn)
        {
            selected += 1;
            if (selected >= 4)
            {
                wrongAudio.Play();
                mochiText.text = "Slow down little chef! You can only select 3 or 1 ingredient card each turn.";
                return false;
            }
        }
    }
```

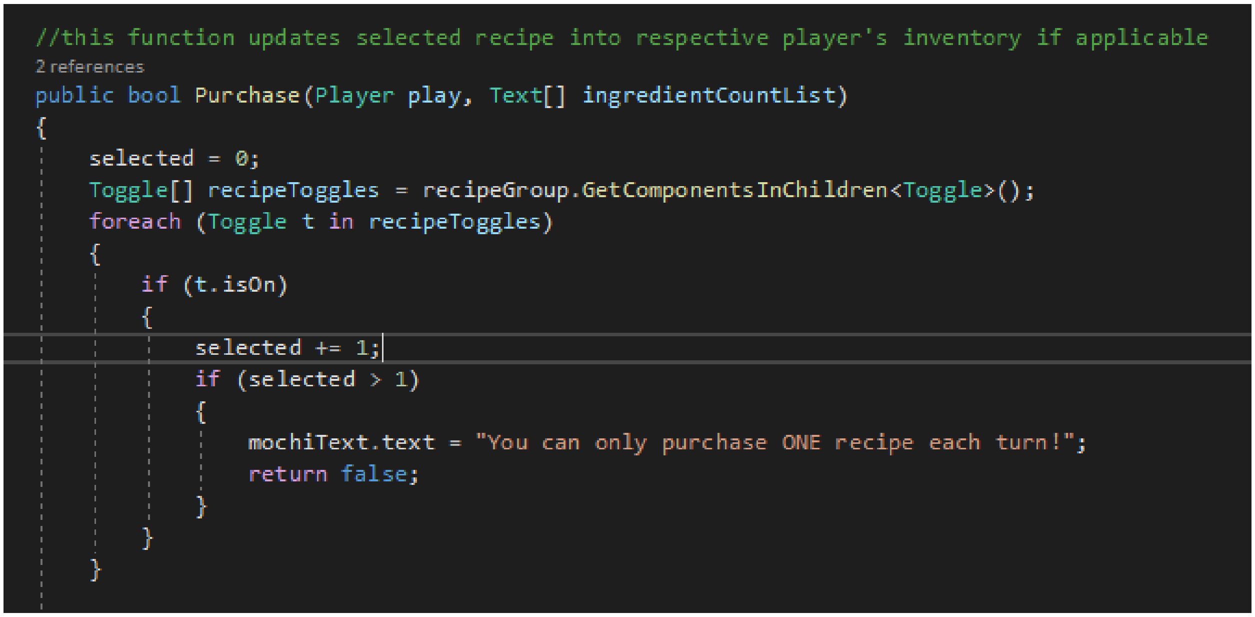


```
//if 3 ingredients are selected, add to inventory and update ui text
if (selected == 3)
{
    actionAudio.Play();
    for (int i = 0; i < ingToggles.Length; i++)
    {
        if (ingToggles[i].isOn)
        {
            play.selectIngredients(i);
            ingredientCountList[i].text = play.ingredientDict[i].ToString();
        }
    }
    mochiText.text = "3 ingredients added to your inventory!";
} else if (selected == 2) //if 2 selected, prompt that either 1 or 3 should be selected
{
    wrongAudio.Play();
    mochiText.text = "Hmmm...you only selected 2. Pick 3 or 1 ingredient!";
    return false;
} else if (selected == 1) //if 1 selected, add 2 of that ingredient to the inventory and update ui text
{
    actionAudio.Play();
    for (int i = 0; i < ingToggles.Length; i++)
    {
        if (ingToggles[i].isOn)
        {
            play.selectIngredients(i);
            play.selectIngredients(i);
            ingredientCountList[i].text = play.ingredientDict[i].ToString();
        }
    }
    mochiText.text = "2 of the same ingredient added to your inventory!";
}

//deselect all toggles (reset for next turn)
for (int i = 0; i < ingToggles.Length; i++)
{
    ingToggles[i].isOn = false;
}
selected = 0;
return true;
```

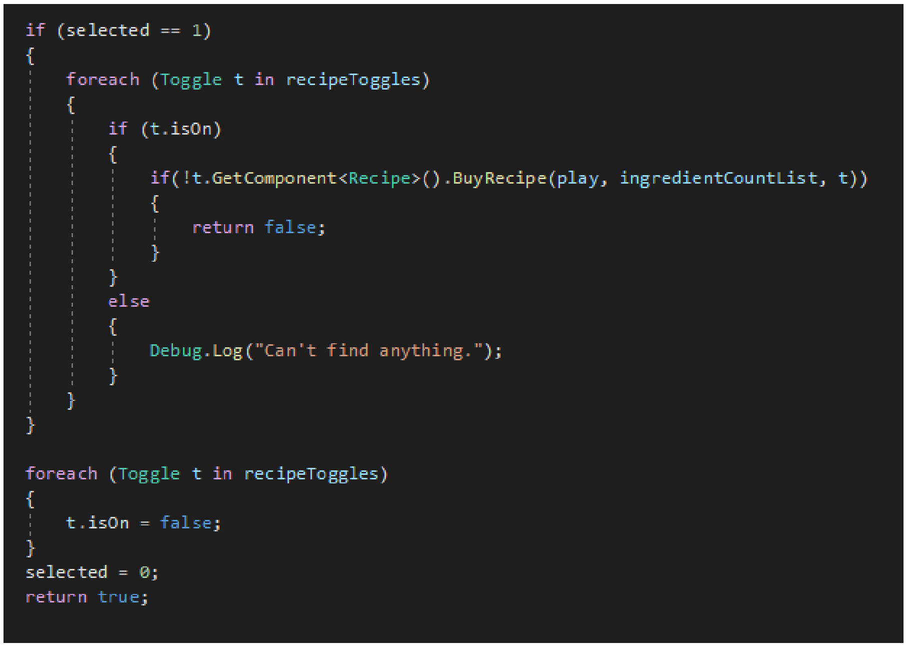
The function will continue to check through other conditions such as 3 selected toggles or 1 selected toggle. 3 unique ingredients or 2 of the same ingredients are added to the inventory respectively. After a legal move is made, the function returns a true and all toggles are deselected and ready for the next player.

The purchase button is set up the same way as the select button where the onClick calls the PurchaseButtonPressed function which runs the Purchase function which returns a boolean denoting if a legal move was made.

The Purchase function takes in the current player inventory, text list, and ingredient count list. It cycles through the recipe toggles and checks which toggle is on. Validation checks are done first to ensure that the player only selected one recipe. If more than one toggle is on, Mochi informs the player that they can only buy one recipe per turn.



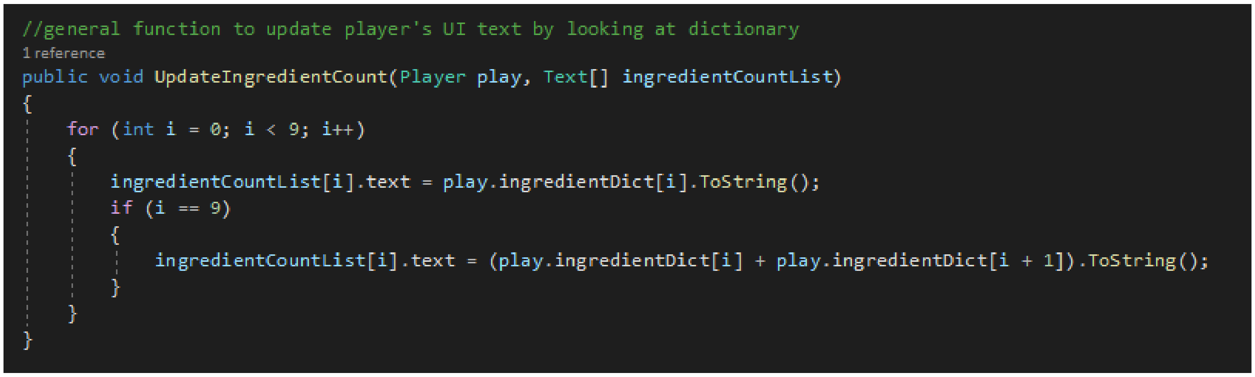
```
//this function updates selected recipe into respective player's inventory if applicable
2 references
public bool Purchase(Player play, Text[] ingredientCountList)
{
    selected = 0;
    Toggle[] recipeToggles = recipeGroup.GetComponentsInChildren<Toggle>();
    foreach (Toggle t in recipeToggles)
    {
        if (t.isOn)
        {
            selected += 1;
            if (selected > 1)
            {
                mochiText.text = "You can only purchase ONE recipe each turn!";
                return false;
            }
        }
    }
```



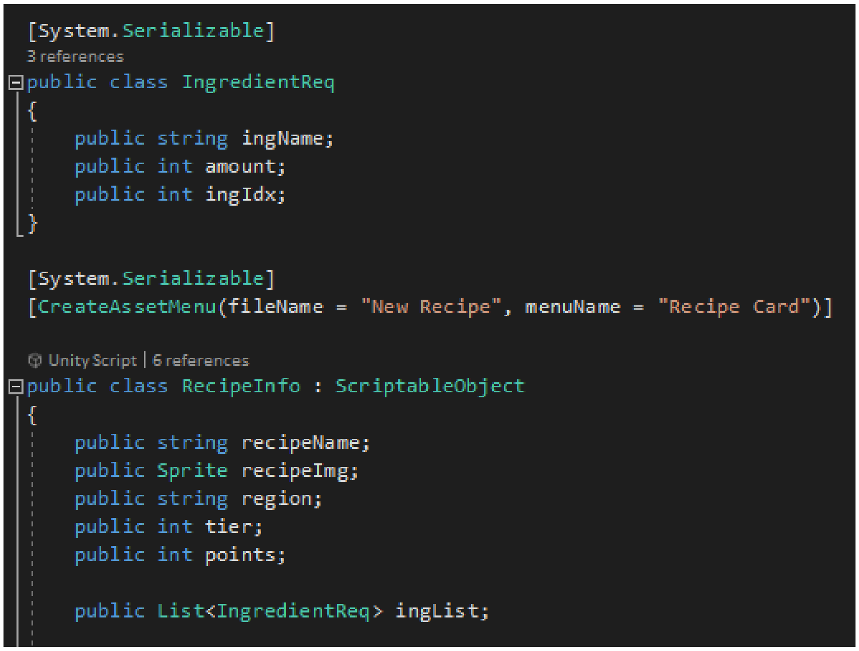
```
if (selected == 1)
{
    foreach (Toggle t in recipeToggles)
    {
        if (t.isOn)
        {
            if(!t.GetComponent<Recipe>().BuyRecipe(play, ingredientCountList, t))
            {
                return false;
            }
        }
        else
        {
            Debug.Log("Can't find anything.");
        }
    }
}

foreach (Toggle t in recipeToggles)
{
    t.isOn = false;
}
selected = 0;
return true;
```

If there is exactly one recipe selected, the recipe script is referenced to check if the player has enough ingredients to buy said recipe. BuyRecipe function from the Recipe class will return a boolean denoting whether the recipe is purchasable or not. If not, the function returns a false and moveMade is not true. If the recipe is purchaseable, BuyRecipe function will deduct the ingredients from the inventory (unless it is a reusable card). Like the previous function, all recipe toggles are deselected before returning true if the checks pass.

All the backend validation needs a function that handles the front end. Players need to see an update of UI elements so that's what the UpdateIngredientsCount function does. It takes in two parameters, player (which has the inventory dictionary), and a list of Text objects. It runs through all 9 ingredients in the dictionary and updates the respective text on the scream. This function can be called from any script anytime UI needs to be updated.

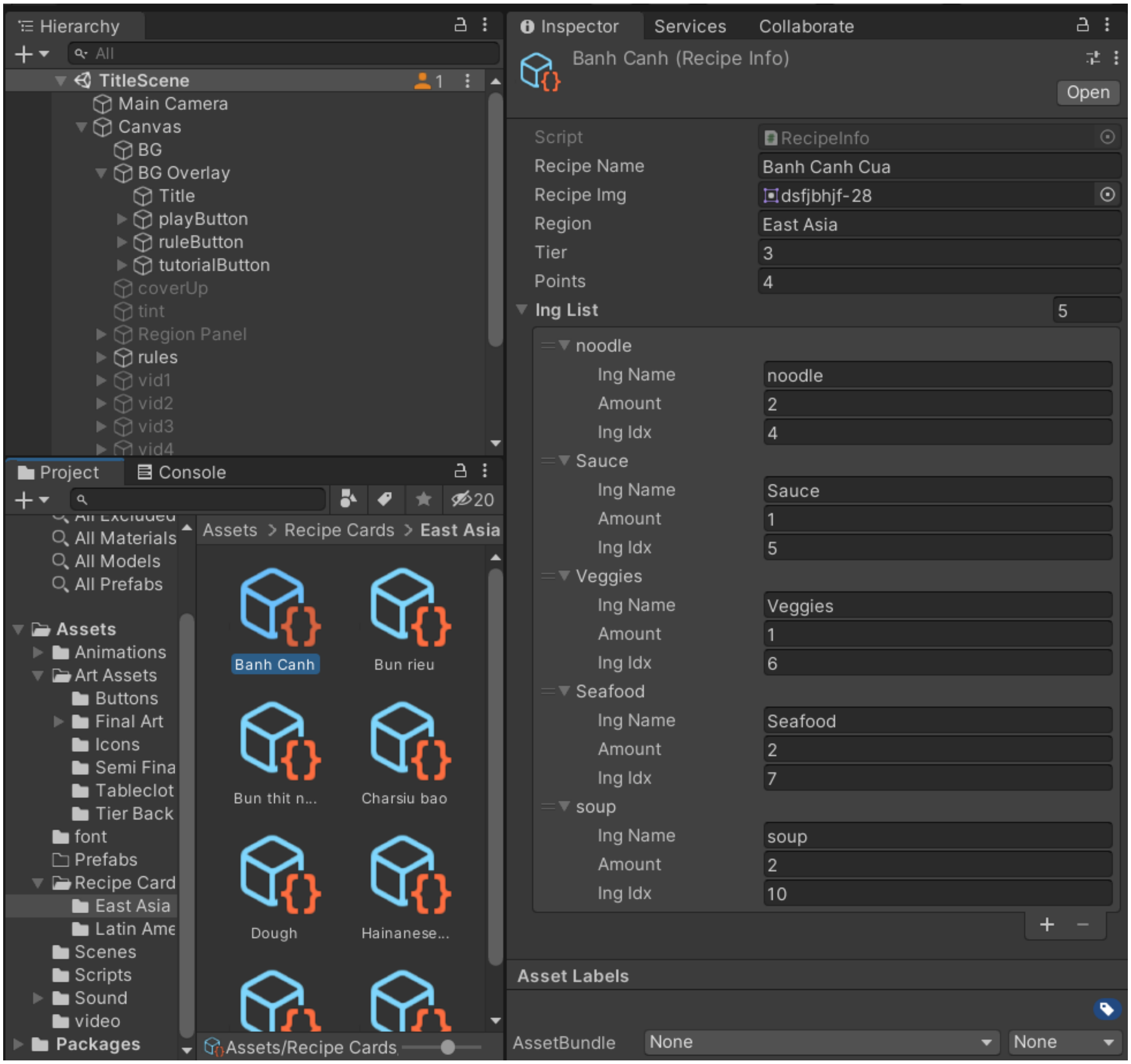


```
//general function to update player's UI text by looking at dictionary
1 reference
public void UpdateIngredientCount(Player play, Text[] ingredientCountList)
{
    for (int i = 0; i < 9; i++)
    {
        ingredientCountList[i].text = play.ingredientDict[i].ToString();
        if (i == 9)
        {
            ingredientCountList[i].text = (play.ingredientDict[i] + play.ingredientDict[i + 1]).ToString();
        }
    }
}
```



```
[System.Serializable]
3 references
public class IngredientReq
{
    public string ingName;
    public int amount;
    public int ingIdx;
}

[System.Serializable]
[CreateAssetMenu(fileName = "New Recipe", menuName = "Recipe Card")]

Unity Script | 6 references
public class RecipeInfo : ScriptableObject
{
    public string recipeName;
    public Sprite recipeImg;
    public string region;
    public int tier;
    public int points;

    public List<IngredientReq> ingList;
```

This game also utilizes scriptable objects. RecipeInfo is a scriptable object that allows us the flexibility of adding as many recipes as we want with the parameters of name, sprite, region, tier, points, and ingredient list. Since all cards have to follow the same format, this scriptable object allows us to make as many instances as needed.



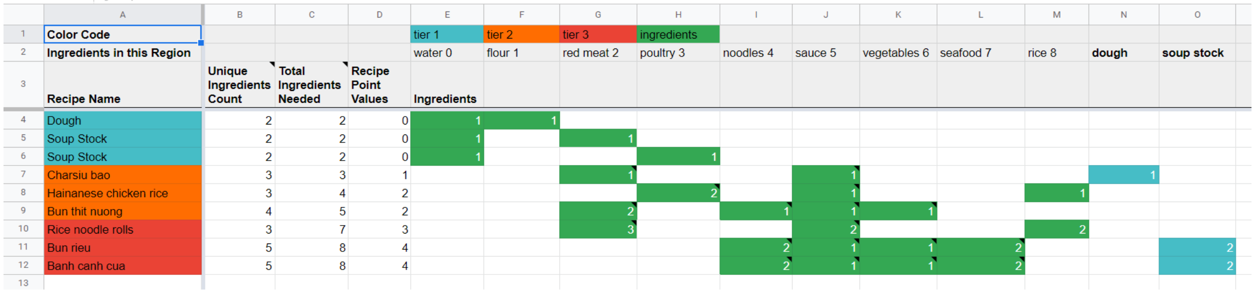
Banh Canh (Recipe Info)

Script	RecipeInfo
Recipe Name	Banh Canh Cua
Recipe Img	dsfjbhjf-28
Region	East Asia
Tier	3
Points	4
Ing List	5
noodle	
Ing Name	noodle
Amount	2
Ing Idx	4
Sauce	
Ing Name	Sauce
Amount	1
Ing Idx	5
Veggies	
Ing Name	Veggies
Amount	1
Ing Idx	6
Seafood	
Ing Name	Seafood
Amount	2
Ing Idx	7
soup	
Ing Name	soup
Amount	2
Ing Idx	10

AssetBundle None None

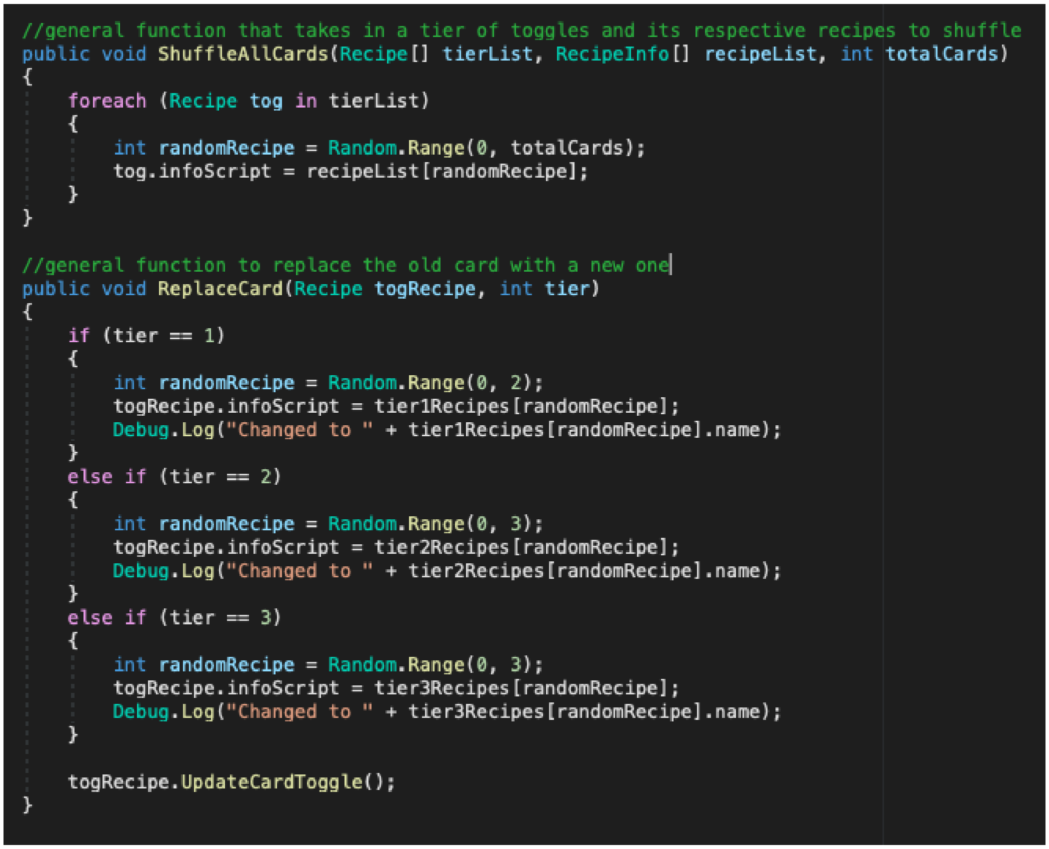
Here is an example of what the scriptable object looks like in the inspector. Anytime information from the recipe is needed, script references could be made to check the ingredient requirements to ensure the players have enough ingredients in their inventory to buy the recipe. Scriptable objects are nice as it acts as a consistent template to add as many recipes as necessary. The shuffle function (later explained), also references this in order to load in the cards and swap out the sprites.

One notable feature is that the strings in the scriptable objects are just for organization. However, it is never used in the scripts itself as the purpose is to avoid hardcoding. Since all regions will have 9 ingredients, the ingredients are assigned an index from 0-9 and referenced number-wise. This allows the toggles to be reused for other regions and are also referenced by numbers.



	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
1	**Color Code**				tier 1	tier 2	tier 3	ingredients							
2	**Ingredients in this Region**				water 0	flour 1	red meat 2	poultry 3	noodles 4	sauce 5	vegetables 6	seafood 7	rice 8	**dough**	**soup stock**
3	**Recipe Name**	**Unique Ingredients Count**	**Total Ingredients Needed**	**Recipe Point Values**	**Ingredients**										
4	Dough	2	2	0	1	1									
5	Soup Stock	2	2	0	1		1								
6	Soup Stock	2	2	0	1			1							
7	Charsiu bao	3	3	1			1			1				1	
8	Hainanese chicken rice	3	4	2				2		1			1		
9	Bun thit nuong	4	5	2			2		1	1	1				
10	Rice noodle rolls	3	7	3			3			2			2		
11	Bun rieu	5	8	4					2	1	1	2			2
12	Banh canh cua	5	8	4					2	1	1	2			2
13															

In order to maintain organization, our team contrived a spreadsheet. Each region gets its own sheet and the format has the color-coded recipes on the y-axis and the numbered ingredients on the x-axis. This was a huge help in referencing the recipe information for the scriptable objects. Our art team also utilized this spreadsheet to ensure the cards and the scriptable objects matched information per recipe.



```
//general function that takes in a tier of toggles and its respective recipes to shuffle
public void ShuffleAllCards(Recipe[] tierList, RecipeInfo[] recipeList, int totalCards)
{
    foreach (Recipe tog in tierList)
    {
        int randomRecipe = Random.Range(0, totalCards);
        tog.infoScript = recipeList[randomRecipe];
    }
}

//general function to replace the old card with a new one
public void ReplaceCard(Recipe togRecipe, int tier)
{
    if (tier == 1)
    {
        int randomRecipe = Random.Range(0, 2);
        togRecipe.infoScript = tier1Recipes[randomRecipe];
        Debug.Log("Changed to " + tier1Recipes[randomRecipe].name);
    }
    else if (tier == 2)
    {
        int randomRecipe = Random.Range(0, 3);
        togRecipe.infoScript = tier2Recipes[randomRecipe];
        Debug.Log("Changed to " + tier2Recipes[randomRecipe].name);
    }
    else if (tier == 3)
    {
        int randomRecipe = Random.Range(0, 3);
        togRecipe.infoScript = tier3Recipes[randomRecipe];
        Debug.Log("Changed to " + tier3Recipes[randomRecipe].name);
    }

    togRecipe.UpdateCardToggle();
}
```

In a real card game, cards are always shuffled and each gameplay is unique. To keep that aspect, we made a shuffle script that shuffles each tier of cards. ShuffleAllCards has 3 parameters takes in a list of tier's toggle recipe script, RecipeInfo list which consists of a list of one tier's recipe scriptable objects, and an integer that indicates the total unique recipes in that tier. ShuffleAllCards is used on start for each tier. ReplaceCard function takes in a recipe toggle and the tier number. When a card is purchased, this function is called by passing in the toggle the card was bought from and randomly gets assigned a new index which refers to a new card scriptable object.