CS420 Technical Report: Potion Crafting Simulator

Team: Daniel Austerman & Ashe Visavale

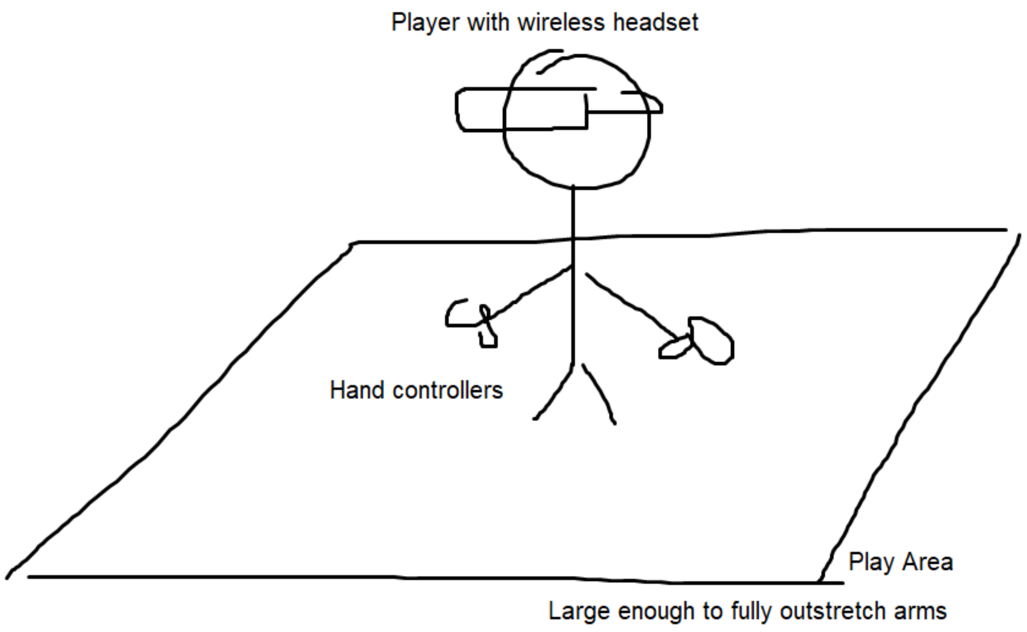
Class: CS420 Virtual Reality, Dr. Turini

Date: 7/23/23

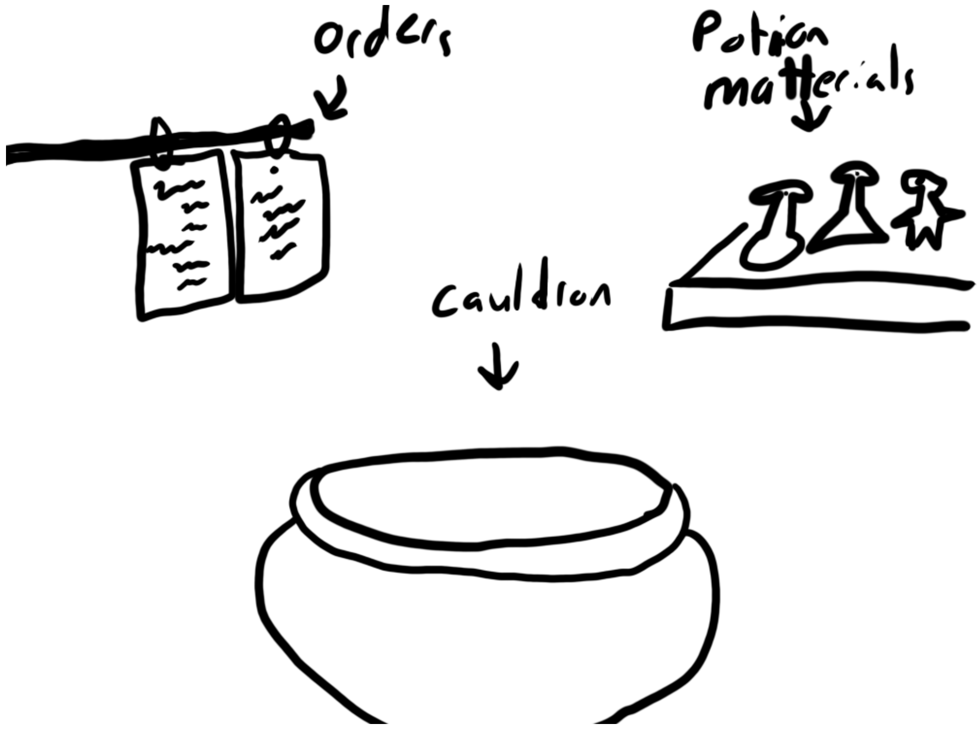
App Concept

The Potion Crafting Simulator is a virtual reality game in which the goal is to craft various different potions using different instructions. Upon starting the game, the player will be prompted with a simple menu in which they can choose to start the game. During gameplay, the player will be prompted with different recipes randomly for making potions, and, in virtual reality, will use hand controllers to grab ingredients, put them into a cauldron, and make the potion. In order to do this, the player must look around in virtual reality to find where instructions are posted, and then look in different locations to find the necessary ingredients. All ingredients will be placed within a reasonable distance so the player can reach them in virtual reality without having to move around a large room. The player will succeed in the game if they craft the correct potion as instructed a certain amount of times (likely determined prior to the beginning of the game), and fail if they make the potion incorrectly a certain amount of times (also likely determined prior to beginning the game). There may also be a time limit put into place for crafting each individual potion. Additional features, if possible, may include an increased number of ingredients and recipes, multiple cauldrons at once for multi-tasking, or level-based structure with feedback on player performance.

VR System Design Diagram


Player with wireless headset
Hand controllers
Play Area
Large enough to fully outstretch arms

In-Game VR View



Technical Features

Hardware Requirements

- Computer (with VR capabilities)
- Wireless VR headset (Communicates with computer)
- Hand controllers (2) (Communicates with headset and computer)

Locomotion

- Direct locomotion in small area
- Play area
  - Slightly larger than player's outstretched arms (possibly 7-8 ft diameter to account for large players)
  - Player will be standing mostly still during play
  - Full 360 degree movement required
  - Ceiling higher than average player's vertical reach
- Hand controllers
  - 1-to-1 motion required
  - At least 1 button on each controller for holding and interacting with objects

Interactions

- Potion ingredients (multiple, exact amount determined by development time)

- Potions
    - Grab and manipulate potions
    - Deliver potions when complete
- Cauldron
    - Put ingredients into cauldron
    - Take potion out of cauldron
    - Stir (if possible with development scope)
- See orders
    - Random selection for an order
    - See all ingredients for a potion order
    - Order goes away when potion is complete
- Menu
    - Determine amount of potions needed for success
    - Determine amount of potions failed before failure
- If possible within development time:
    - Timer
    - Additional cauldrons
    - Additional Stations (for more complex recipes)
    - Additional recipes and ingredients
    - Level structure (as opposed to random)

Development

First, we imported a 3D model of a cauldron to act as the base for our potion crafting system.

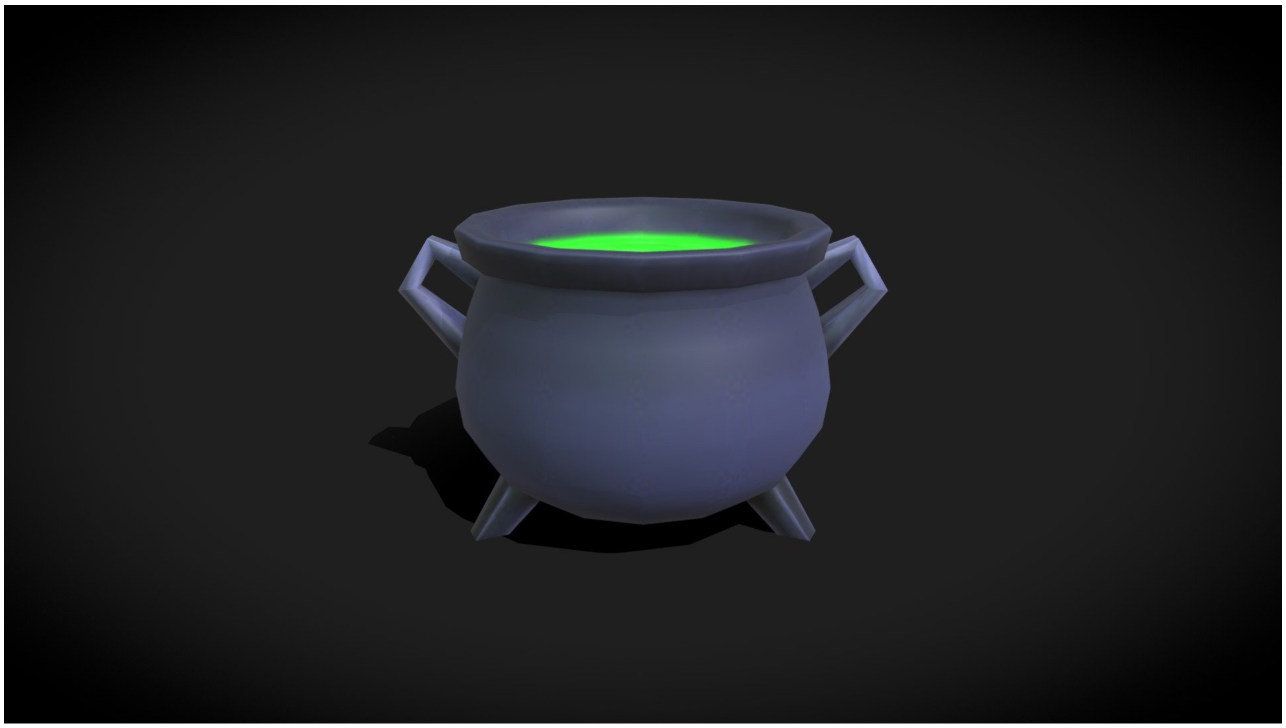


Figure 1: Cauldron preview image.

We aligned it in VR space so the player would be standing just in front of it, while still leaving it reachable.

Next, we began to implement systems in order to handle the delivery of recipe instructions to the player.

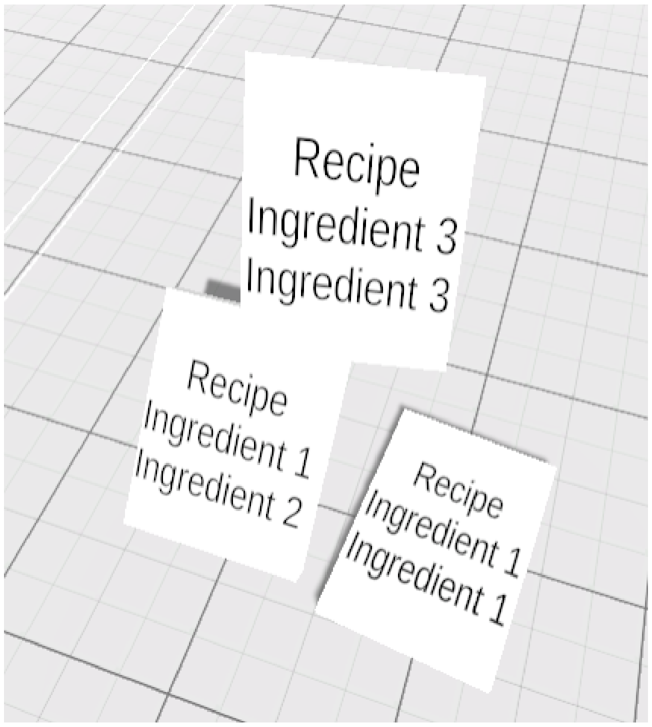
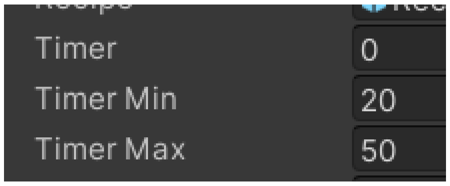


Figure 2: A few recipe cards. Note that recipes can be either two or three ingredients.

In order to spawn the cards, a prefab was made that consisted of a cube in a paper shape with a TextMeshPro child. When spawning a new card first, the script "RecipeSpawnerController" handles a timer which, when expired, spawns a new card.

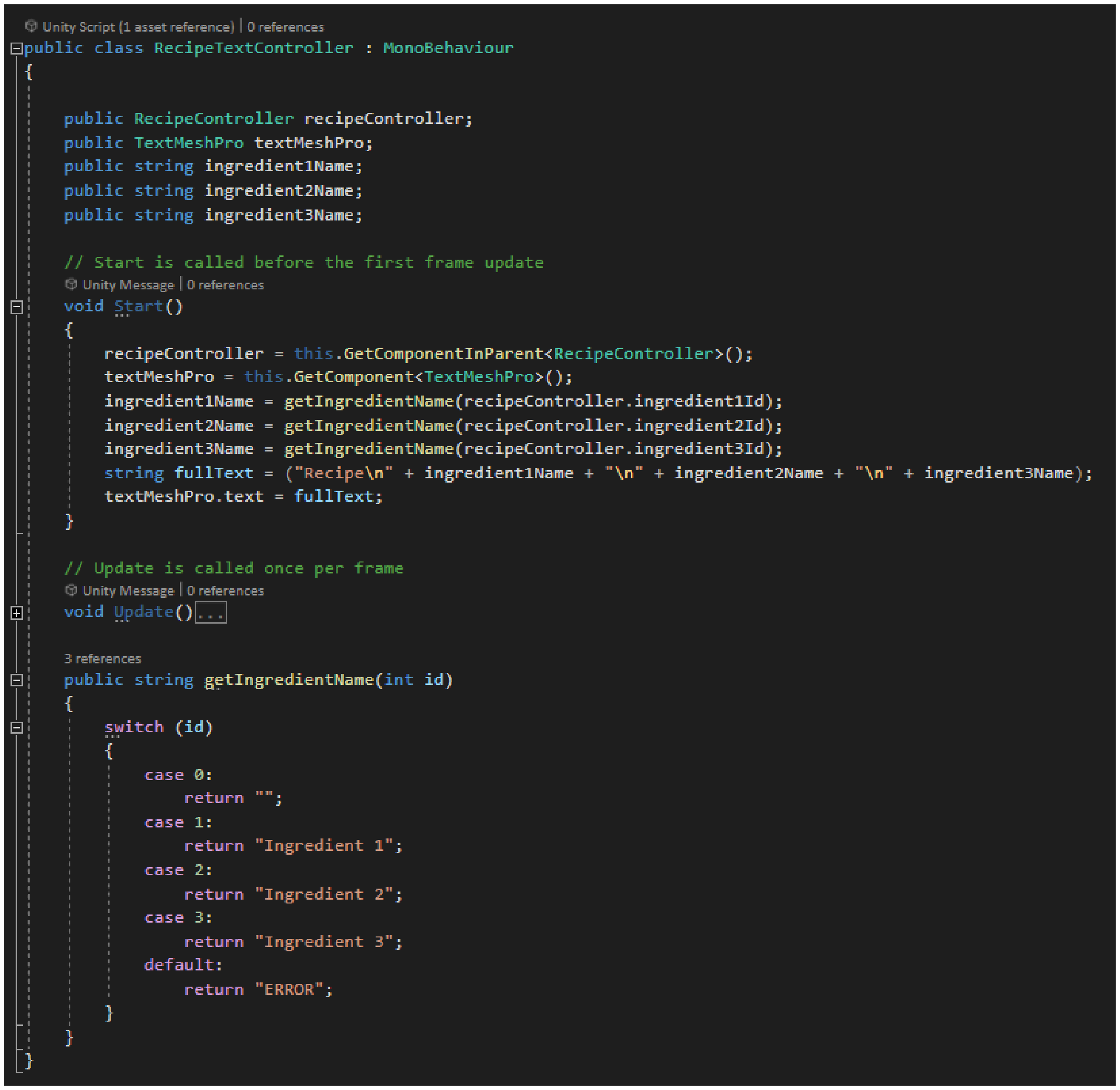


Timer	0
Timer Min	20
Timer Max	50

Figure 3: Timer variables. "Timer" dictates the time it takes to spawn the very first recipe. "Timer Min" designates the minimum amount of time before a new recipe is spawned, while "Timer Max" designates the maximum amount of time before a new recipe is spawned.

Then, when the card is successfully spawned, a script titled "RecipeController" sets the recipe to a random recipe from a specialized list, including a list of ingredients needed. Lastly, the RecipeTextController script takes the ingredient list and displays it on the card as text.

Code 1: RecipeTextController Script.



```
Unity Script (1 asset reference) | 0 references
public class RecipeTextController : MonoBehaviour
{

    public RecipeController recipeController;
    public TextMeshPro textMeshPro;
    public string ingredient1Name;
    public string ingredient2Name;
    public string ingredient3Name;

    // Start is called before the first frame update
    Unity Message | 0 references
    void Start()
    {
        recipeController = this.GetComponentInParent<RecipeController>();
        textMeshPro = this.GetComponent<TextMeshPro>();
        ingredient1Name = getIngredientName(recipeController.ingredient1Id);
        ingredient2Name = getIngredientName(recipeController.ingredient2Id);
        ingredient3Name = getIngredientName(recipeController.ingredient3Id);
        string fullText = ("Recipe\n" + ingredient1Name + "\n" + ingredient2Name + "\n" + ingredient3Name);
        textMeshPro.text = fullText;
    }

    // Update is called once per frame
    Unity Message | 0 references
    void Update()...

    3 references
    public string getIngredientName(int id)
    {
        switch (id)
        {
            case 0:
                return "";
            case 1:
                return "Ingredient 1";
            case 2:
                return "Ingredient 2";
            case 3:
                return "Ingredient 3";
            default:
                return "ERROR";
        }
    }
}
```

After the cards were successfully spawning, we added some ingredients to the world and a spawner script so there would always be ingredients in the world. The ingredients were made grabbable so the player can move them around and add them to the cauldron to make potions. A model of a table was also imported in order to have the ingredients sit on something, making the ingredients easier to grab for the player.

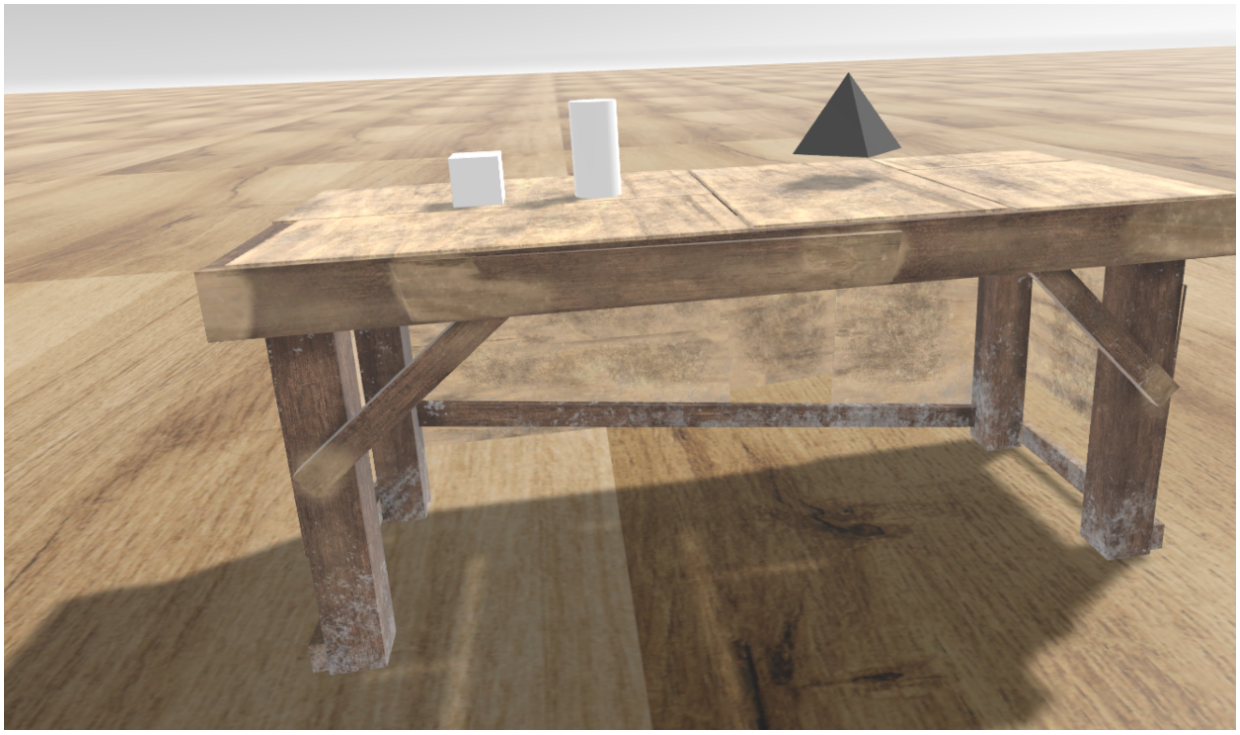
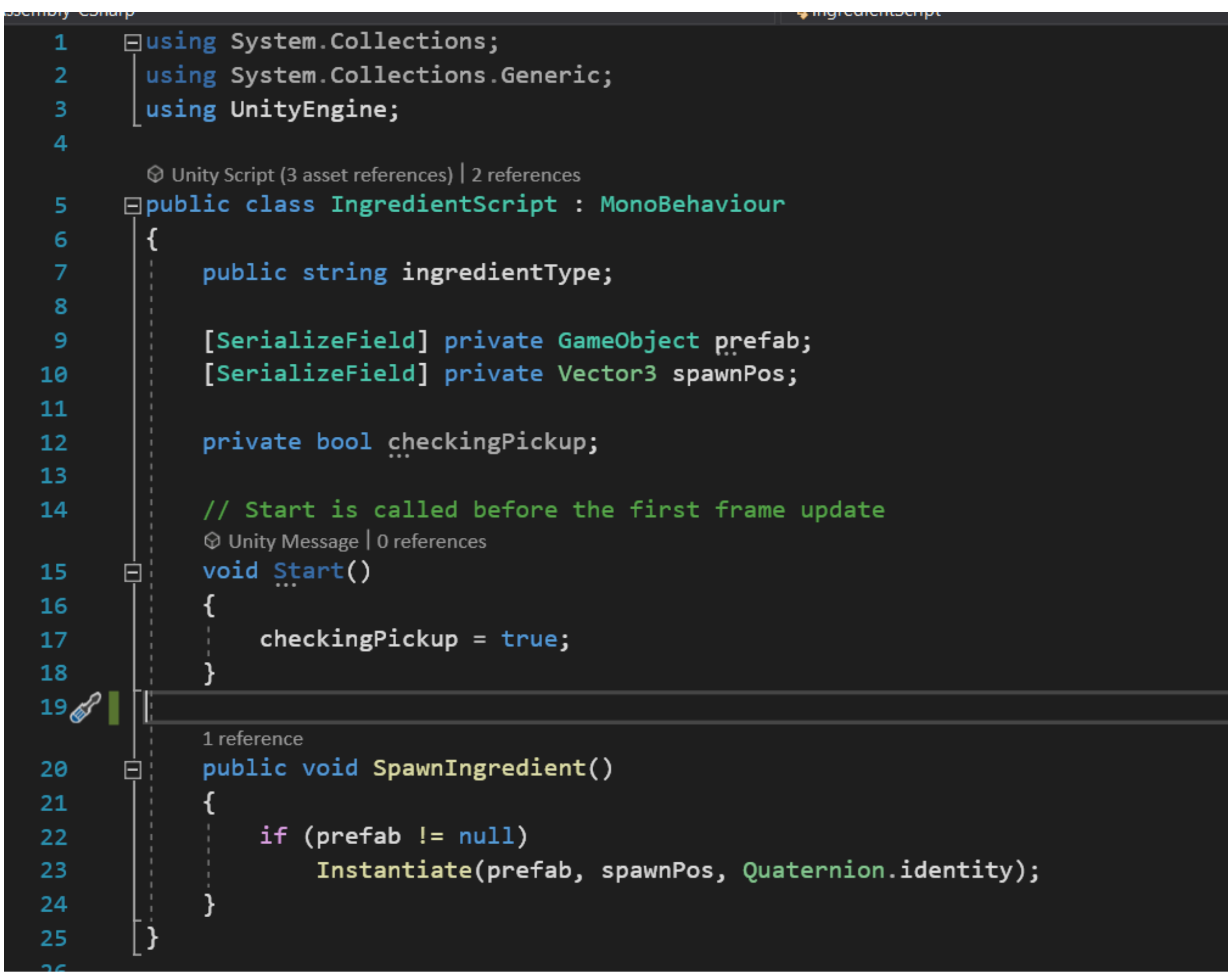


Figure 4: Ingredients

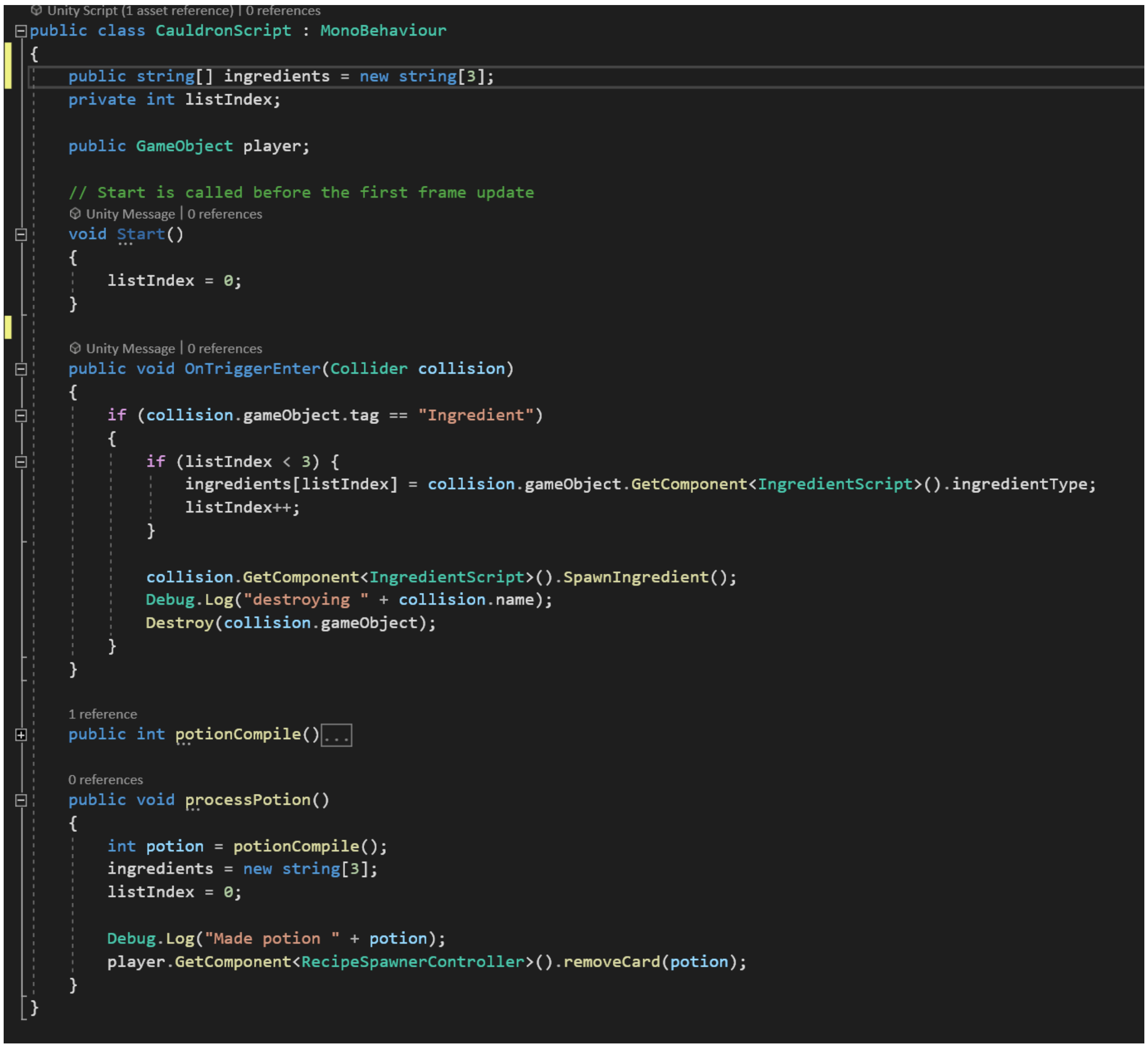
Code 2: Ingredient spawning script.



```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

Unity Script (3 asset references) | 2 references
public class IngredientScript : MonoBehaviour
{
    public string ingredientType;

    [SerializeField] private GameObject prefab;
    [SerializeField] private Vector3 spawnPos;

    private bool checkingPickup;

    // Start is called before the first frame update
    Unity Message | 0 references
    void Start()
    {
        checkingPickup = true;
    }

    1 reference
    public void SpawnIngredient()
    {
        if (prefab != null)
            Instantiate(prefab, spawnPos, Quaternion.identity);
    }
}
```

After the ingredients were successfully implemented, the cauldron was then programmed to take in the ingredients. That is, when the ingredients collided with the cauldron, the ingredient would be deleted from the world and "added" to the cauldron.

Code 3: Cauldron script. Note that a large switch statement has been collapsed for readability.



```
Unity Script (1 asset reference) | 0 references
public class CauldronScript : MonoBehaviour
{
    public string[] ingredients = new string[3];
    private int listIndex;

    public GameObject player;

    // Start is called before the first frame update
    Unity Message | 0 references
    void Start()
    {
        listIndex = 0;
    }

    Unity Message | 0 references
    public void OnTriggerEnter(Collider collision)
    {
        if (collision.gameObject.tag == "Ingredient")
        {
            if (listIndex < 3) {
                ingredients[listIndex] = collision.gameObject.GetComponent<IngredientScript>().ingredientType;
                listIndex++;
            }

            collision.GetComponent<IngredientScript>().SpawnIngredient();
            Debug.Log("destroying " + collision.name);
            Destroy(collision.gameObject);
        }
    }

    1 reference
    public int potionCompile()...

    0 references
    public void processPotion()
    {
        int potion = potionCompile();
        ingredients = new string[3];
        listIndex = 0;

        Debug.Log("Made potion " + potion);
        player.GetComponent<RecipeSpawnerController>().removeCard(potion);
    }
}
```

In order for the cauldron to actually make a potion, a button was added to the side of the cauldron, which, when pressed, calls a method to process the potion. This method looks at the ingredients added to the cauldron, and determines what potion is being crafted.

If the potion being crafted aligns with a recipe card which currently exists in the world, then the recipe will be marked as "complete", deleting the card from the world.

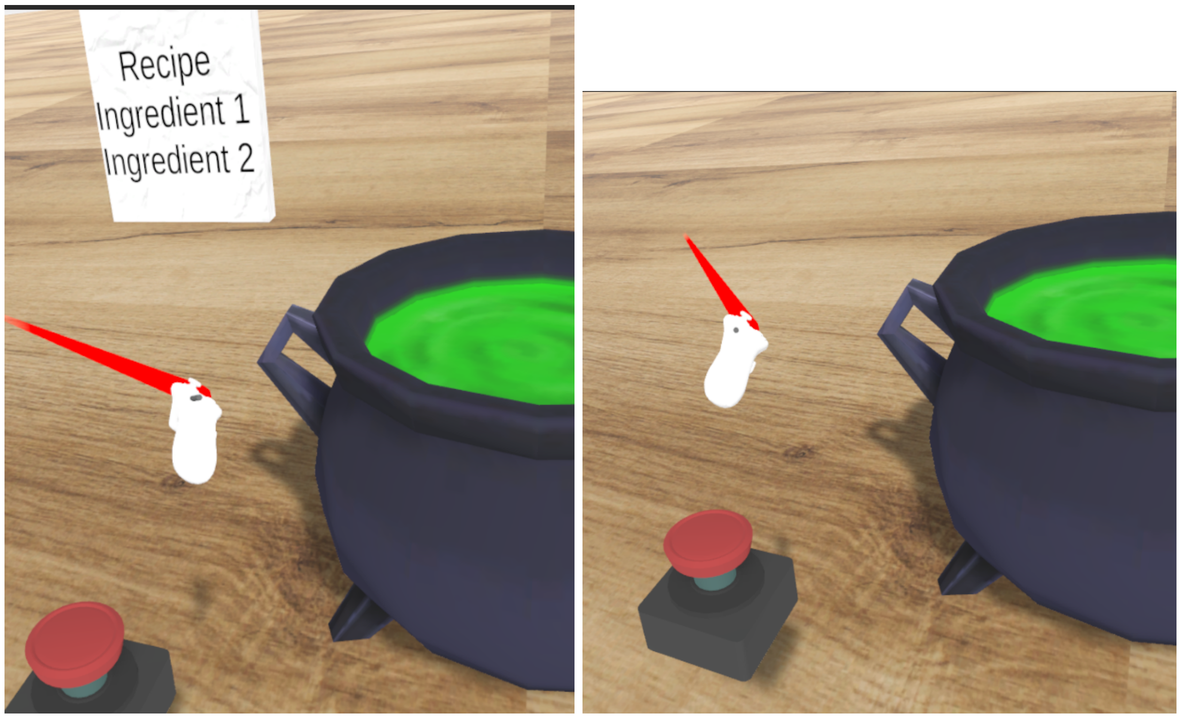
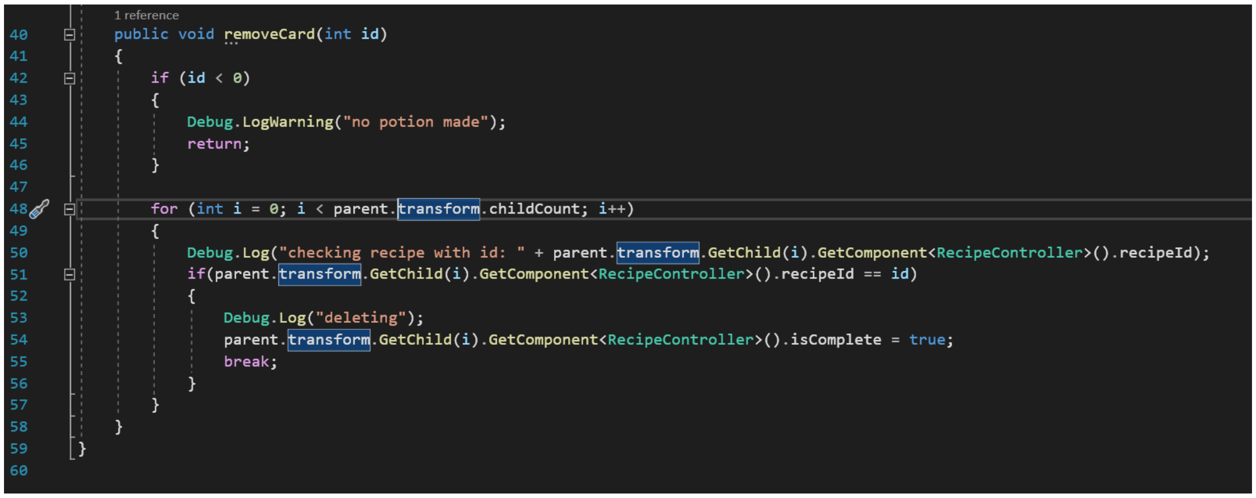


Figure 5: Button with recipe card before and after a potion is made. Note: the button was pressed in between these screenshots.

Code 4: Remove card method, located in RecipeSpawnerController.



```
1 reference
public void removeCard(int id)
{
    if (id < 0)
    {
        Debug.LogWarning("no potion made");
        return;
    }

    for (int i = 0; i < parent.transform.childCount; i++)
    {
        Debug.Log("checking recipe with id: " + parent.transform.GetChild(i).GetComponent<RecipeController>().recipeId);
        if(parent.transform.GetChild(i).GetComponent<RecipeController>().recipeId == id)
        {
            Debug.Log("deleting");
            parent.transform.GetChild(i).GetComponent<RecipeController>().isComplete = true;
            break;
        }
    }
}
}
```

The process of getting recipe cards and making the potion then continues infinitely, until the player wishes to stop. Currently, the plan is to make a system in which the player is tasked with completing a certain amount of potions in a certain time period, but said system will need to be implemented near the end of development as to not interfere with other systems. A full list of planned features can be found below.

Future Plans

- Proper models and names for ingredients
- Additional ingredients
- Hand models instead of controller models
- Refined recipe cards
- Menu system
- Completed potions tracking (number of completed recipes)/ending the game

- Potion failure system