CS420 Technical Report: Potion Crafting Simulator

Team: Daniel Austerman & Ashe Visavale

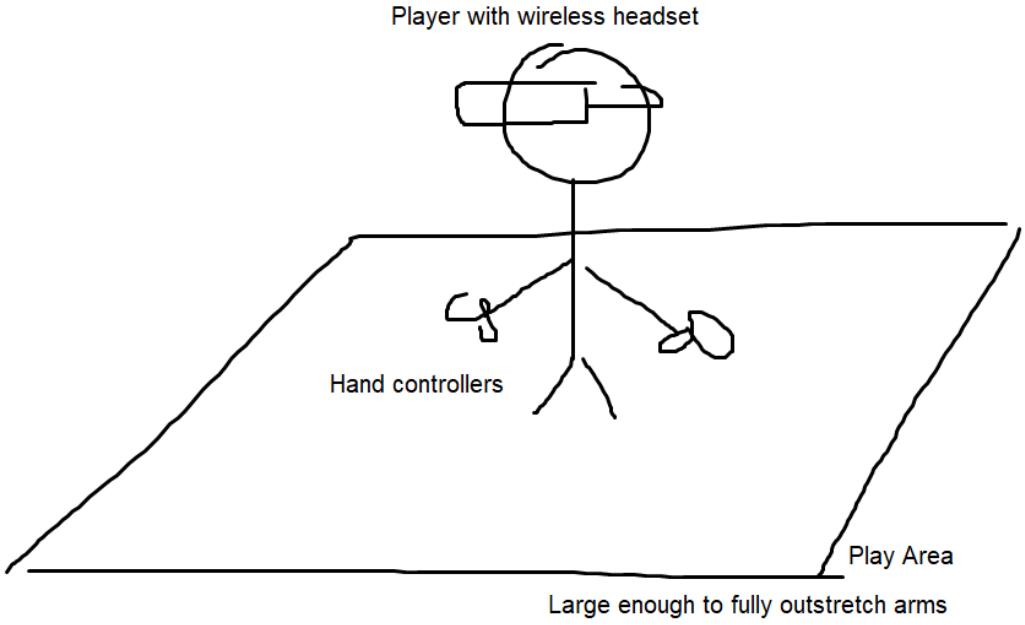
Class: CS420 Virtual Reality, Dr. Turini

Date: 9/23/23

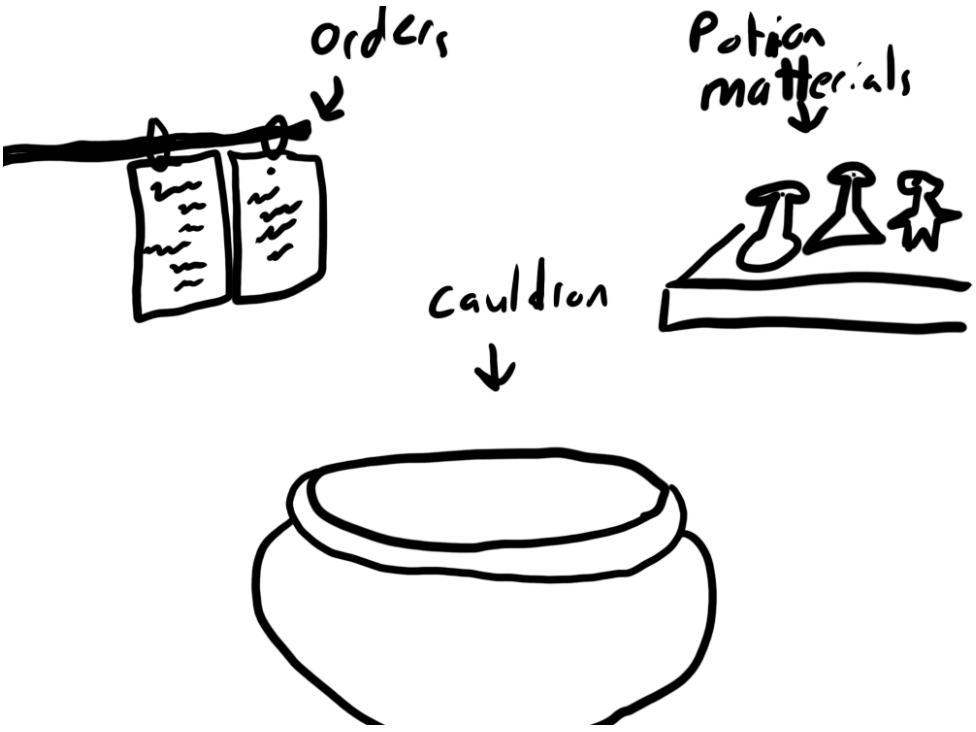
App Concept

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



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.

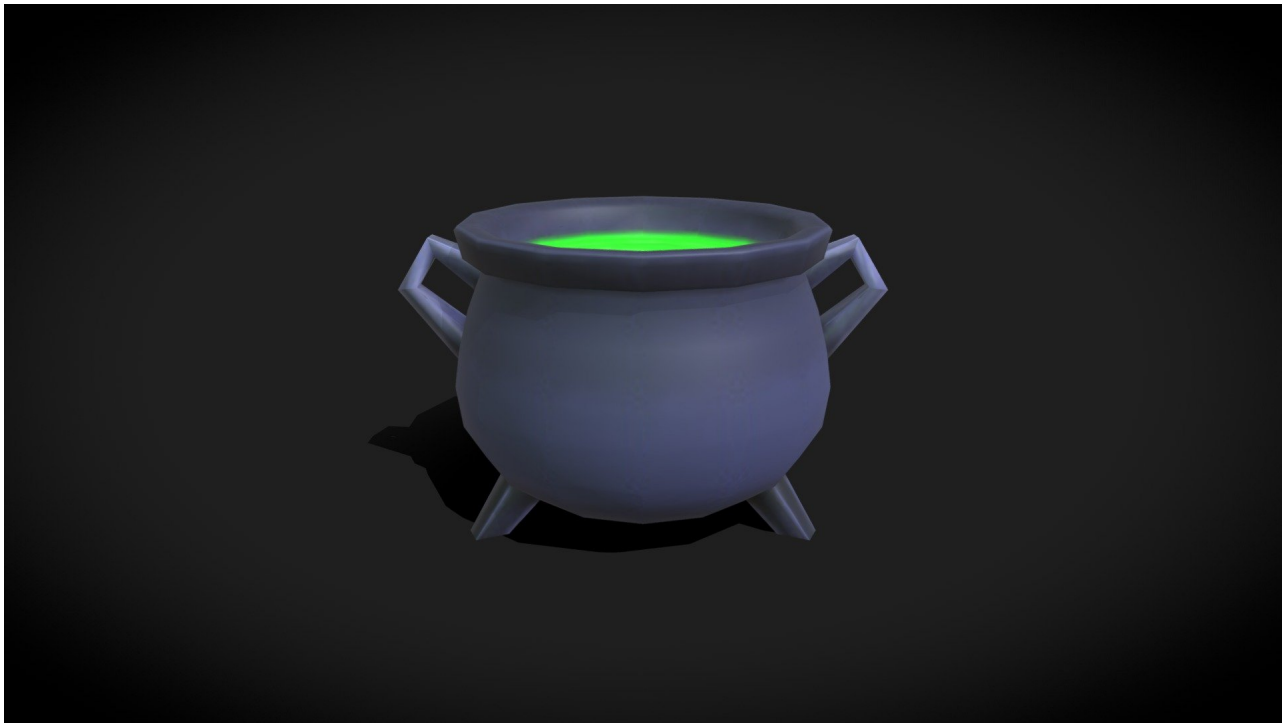


Figure 1: Cauldron preview image.

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.

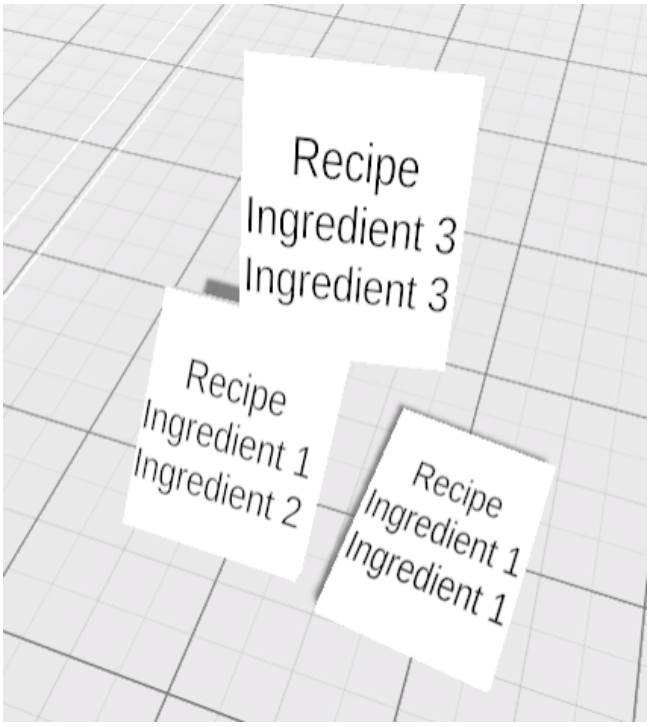


Figure 2: A few recipe cards. Note that recipes can be either two or three ingredients.

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.

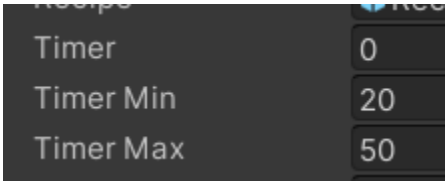


Figure 3: Timer variables. "Timer" dictates the time it takes to spawn the very first recipe. "Timer Min" designates the minimum amount of time before a new recipe is spawned, while "Timer Max" designates the maximum amount of time before a new recipe is spawned.

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.

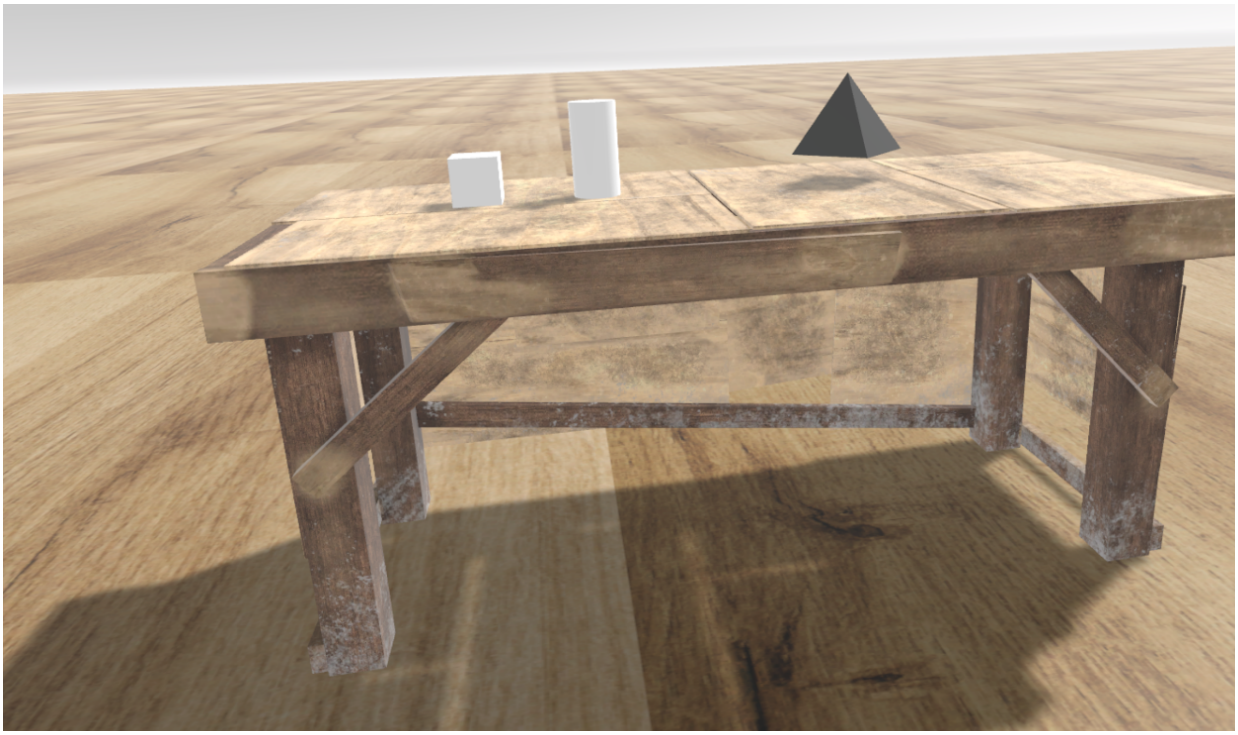


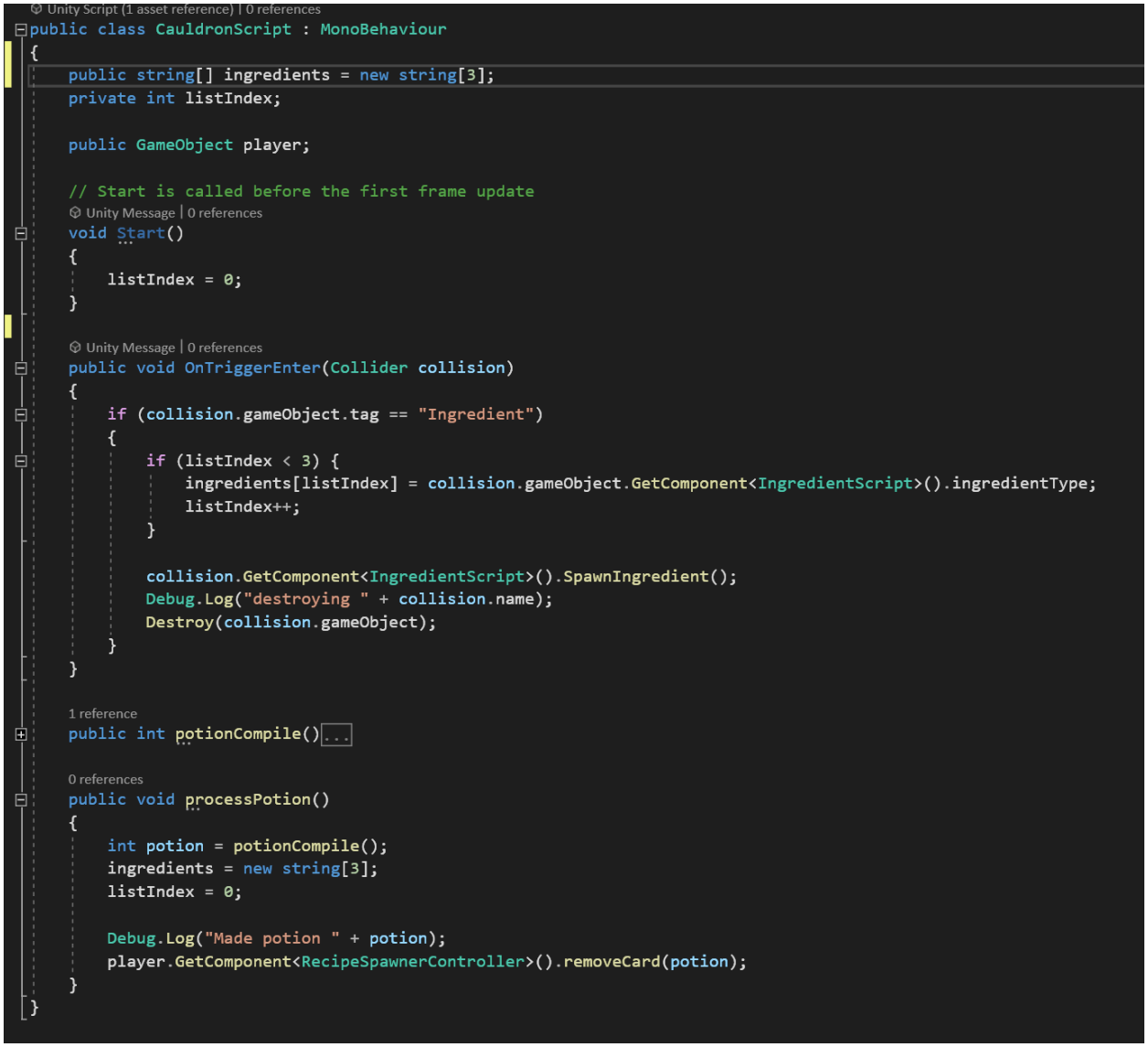
Figure 4: Ingredients

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.

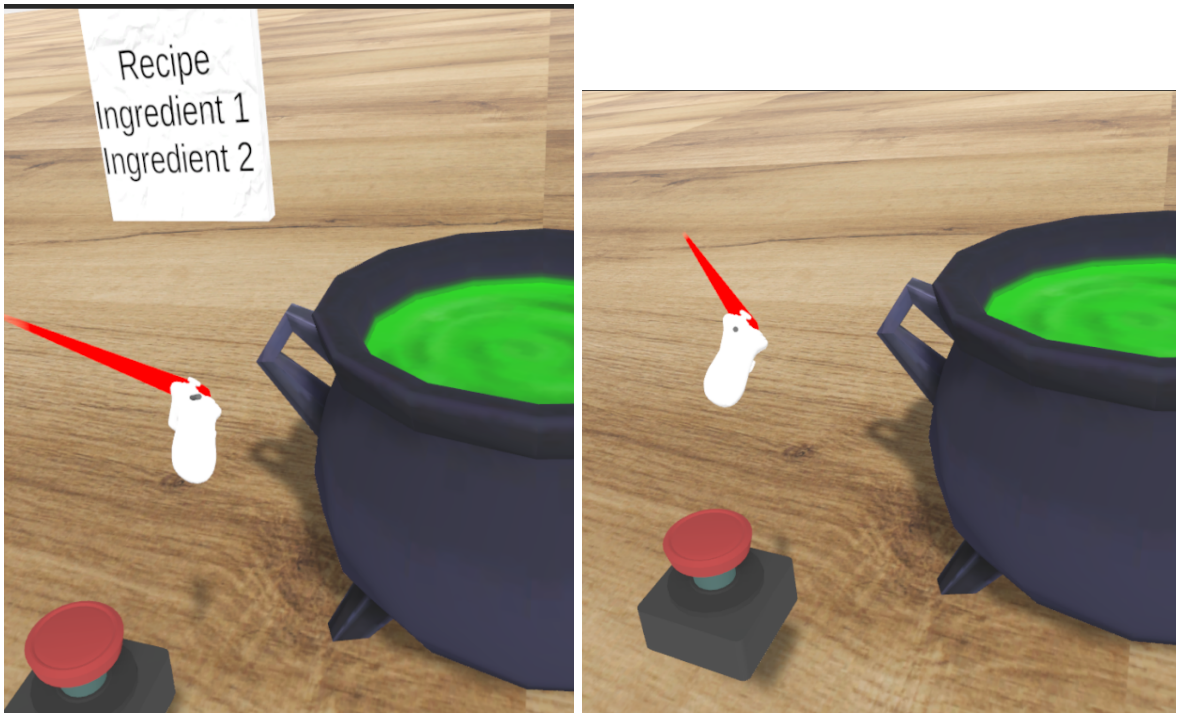


Figure 5: Button with recipe card before and after a potion is made. Note: the button was pressed in between these screenshots.

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;
        }
    }
}
}
```

SUBMISSION 3 CONTENT BEGINS HERE

Next, the VR environment itself was developed. In order to represent an appropriate setting for the game, we decided to simulate being inside of a cave. This cave was represented by a sphere object with a stone texture to create the illusion of being in a cave. In order to preserve the illusion, the cave is large enough to cover well over the entire play area. Exiting the play area will freeze the camera and not allow the player to leave the cave.

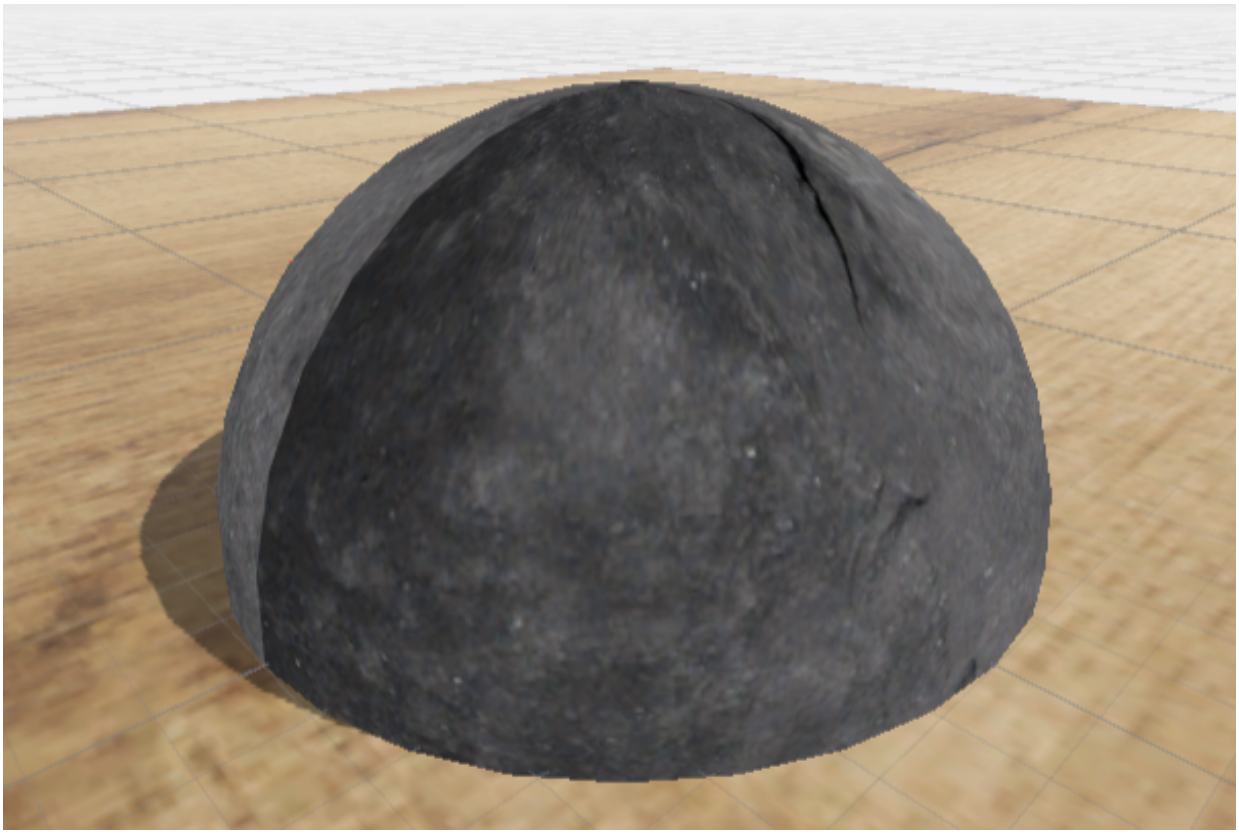


Figure 6: The exterior of the cave sphere. Note that this exterior will never be seen during normal gameplay.

In order to ensure that the cave texture appeared on the inside of the sphere, a shader was developed in order to place the texture on both the inside and outside of the sphere.

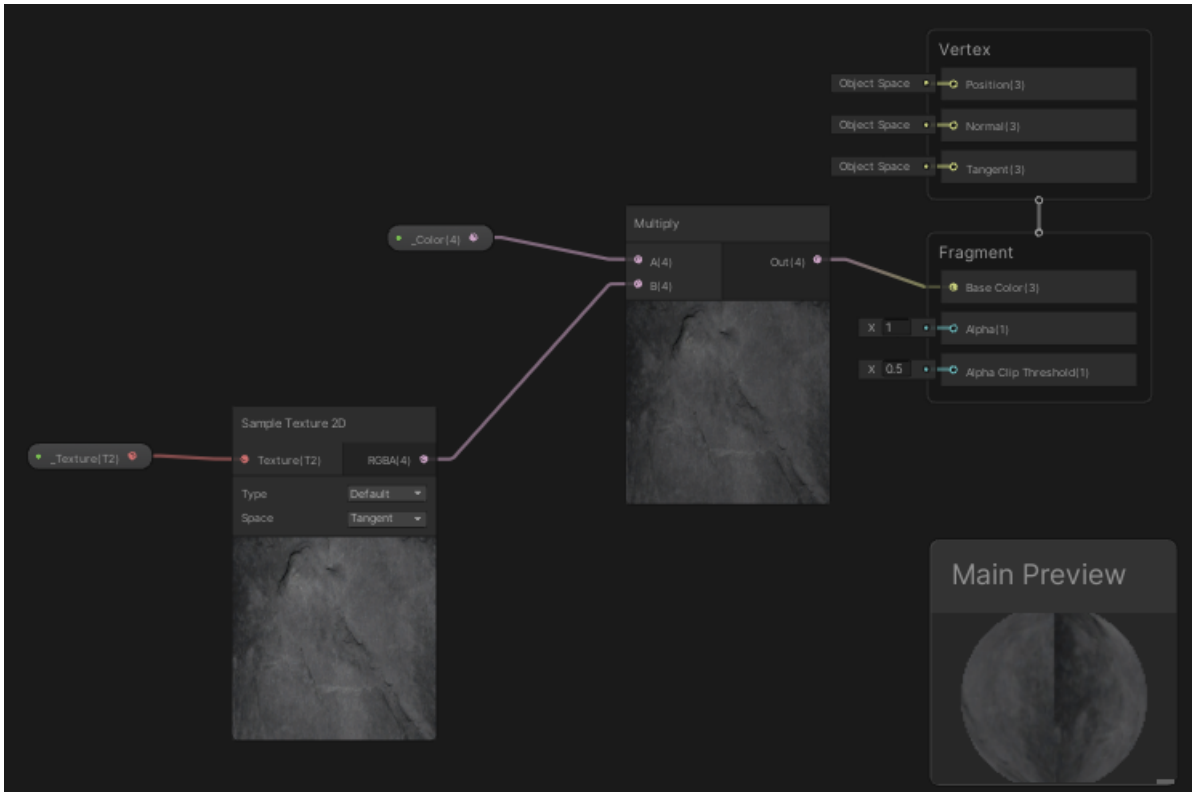


Figure 7: Shader graph for cave texture.

After the VR environment was completed, it was then time to refine the game. First, the ingredients were decided, and the placeholders were replaced. The final ingredient list consists of frogs, leaves, feathers, elixirs, and flowers.

Figure 8: Ingredients with final models.

Since the total number of ingredients increased, additional recipes were also added. Additionally, the text on the recipe cards were replaced with the actual names of the ingredients, rather than the previous placeholder names of “ingredient 1”, “ingredient 2”, etc.

At this point, sounds were also implemented. There is a “splash” sound effect which plays when an ingredient is inserted into the cauldron, a “confirm” sound effect that plays when a potion is successfully crafted, and a “deny” sound effect when a potion is made unsuccessfully. The “deny” sound effect also acts as the potion failure system. The sound effects were implemented through basic code triggers.

After the sound effect system was added, background music was also implemented. This background music is copyright-free and plays on loop during the game.

In order to spruce up the visuals of the game, the controller models were also replaced with hands. Unfortunately, there was significant difficulty in getting the hands to animate with the grab button, so they remain static throughout the game.

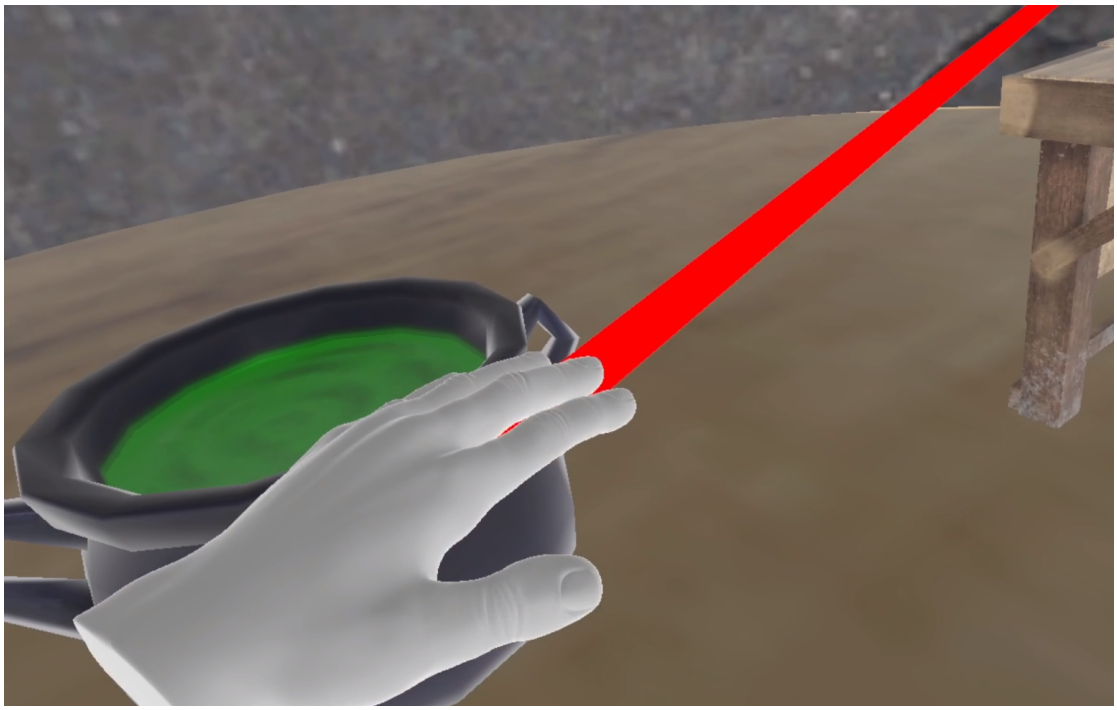
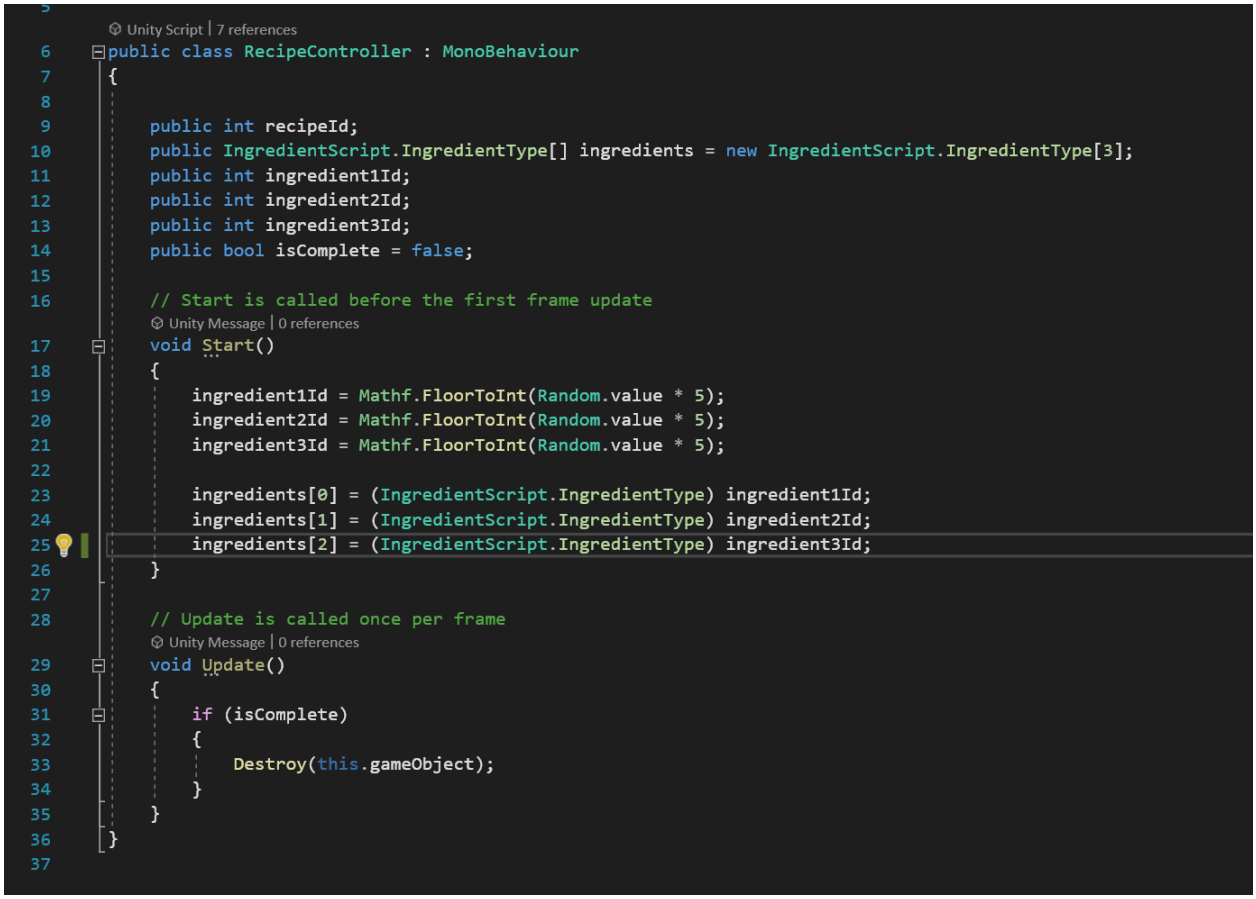


Figure 9: The hand present in-game.

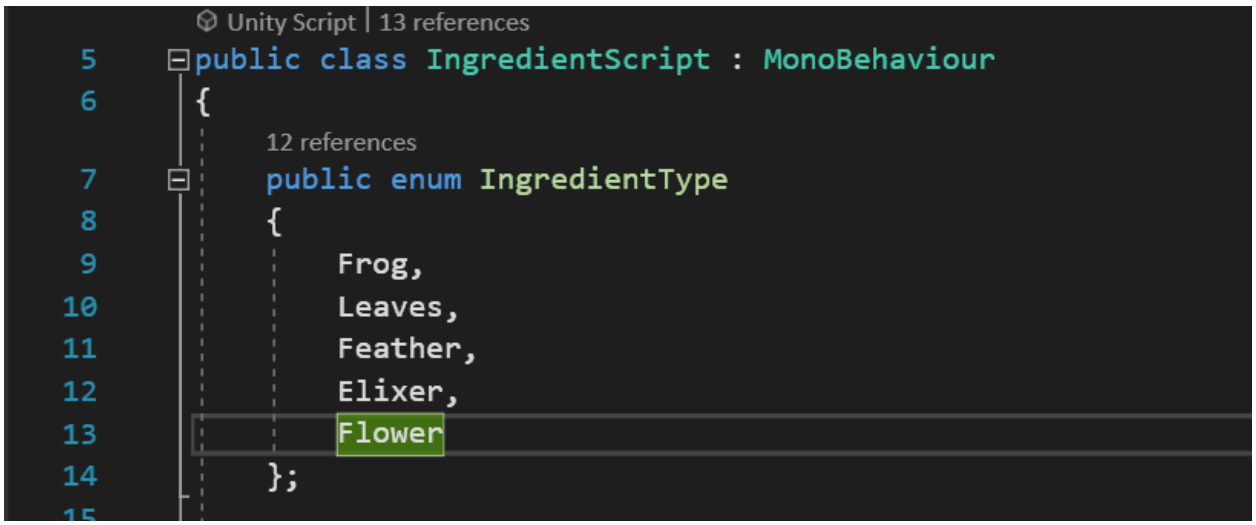
With the game now looking presentable, the next logical step was to refine the process of generating recipes. It was decided that recipes should be randomly generated rather than predetermined in order to allow for greater diversity when playing the game.

Code 5: New recipe generation script with random ingredients.



```
Unity Script | 7 references
public class RecipeController : MonoBehaviour
{

    public int recipeId;
    public IngredientScript.IngredientType[] ingredients = new IngredientScript.IngredientType[3];
    public int ingredient1Id;
    public int ingredient2Id;
    public int ingredient3Id;
    public bool isComplete = false;

    // Start is called before the first frame update
    Unity Message | 0 references
    void Start()
    {
        ingredient1Id = Mathf.FloorToInt(Random.value * 5);
        ingredient2Id = Mathf.FloorToInt(Random.value * 5);
        ingredient3Id = Mathf.FloorToInt(Random.value * 5);

        ingredients[0] = (IngredientScript.IngredientType) ingredient1Id;
        ingredients[1] = (IngredientScript.IngredientType) ingredient2Id;
        ingredients[2] = (IngredientScript.IngredientType) ingredient3Id;
    }

    // Update is called once per frame
    Unity Message | 0 references
    void Update()
    {
        if (isComplete)
        {
            Destroy(this.gameObject);
        }
    }
}
```

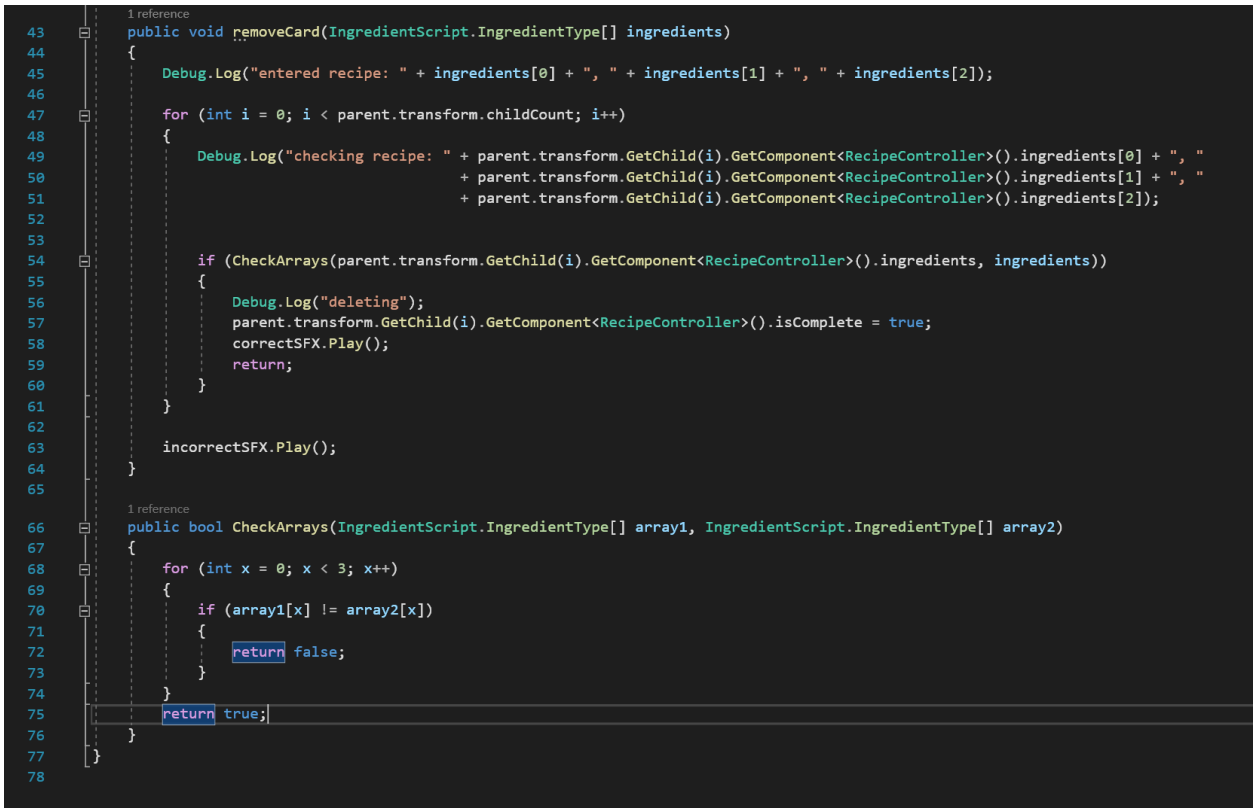
Code 6: Ingredient names in code.



```
Unity Script | 13 references
public class IngredientScript : MonoBehaviour
{
    12 references
    public enum IngredientType
    {
        Frog,
        Leaves,
        Feather,
        Elixer,
        Flower
    };
```

Additionally, in order to add further depth, it was determined that recipes need to have their ingredients added in order. In order to implement this, the script which handles the checking of recipes was overhauled using a for loop.

Code 7: New recipe verification check



```
1 reference
public void removeCard(IngredientScript.IngredientType[] ingredients)
{
    Debug.Log("entered recipe: " + ingredients[0] + ", " + ingredients[1] + ", " + ingredients[2]);

    for (int i = 0; i < parent.transform.childCount; i++)
    {
        Debug.Log("checking recipe: " + parent.transform.GetChild(i).GetComponent<RecipeController>().ingredients[0] + ", "
                                      + parent.transform.GetChild(i).GetComponent<RecipeController>().ingredients[1] + ", "
                                      + parent.transform.GetChild(i).GetComponent<RecipeController>().ingredients[2]);


        if (CheckArrays(parent.transform.GetChild(i).GetComponent<RecipeController>().ingredients, ingredients))
        {
            Debug.Log("deleting");
            parent.transform.GetChild(i).GetComponent<RecipeController>().isComplete = true;
            correctSFX.Play();
            return;
        }
    }

    incorrectSFX.Play();
}

1 reference
public bool CheckArrays(IngredientScript.IngredientType[] array1, IngredientScript.IngredientType[] array2)
{
    for (int x = 0; x < 3; x++)
    {
        if (array1[x] != array2[x])
        {
            return false;
        }
    }
    return true;
}
}
```

This code also still uses a return statement in order to prevent all identical recipes from being deleted off of a single craft.

In total, this code refinement also allows for potential further expansion of the ingredient system if desired, but under the scope of this project, the ingredient list was kept to five for simplicity.

Finally, the cauldron was updated to ensure success in all situations, including making the code future-proof for potential expansion.

Code 8: Refined cauldron script

```
Unity Script | 0 references
public class CauldronScript : MonoBehaviour
{
    public IngredientScript.IngredientType[] ingredients = new IngredientScript.IngredientType[3];
    private int listIndex;

    public GameObject player;

    public AudioSource splashSoundEffect;

    // 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>().ingredient;
                listIndex++;
            }

            splashSoundEffect.Play();

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

    0 references
    public void processPotion()
    {
        player.GetComponent<RecipeSpawnerController>().removeCard(ingredients);

        ingredients = new IngredientScript.IngredientType[3];
        listIndex = 0;

    }
}
```

After the refinement of all systems, there was a notable amount of required debugging, explained in the “testing and evaluation” section. The above code reflects the final product and not code which contains extreme bugs.

The process of getting recipe cards and making the potion then continues infinitely, until the player wishes to stop.

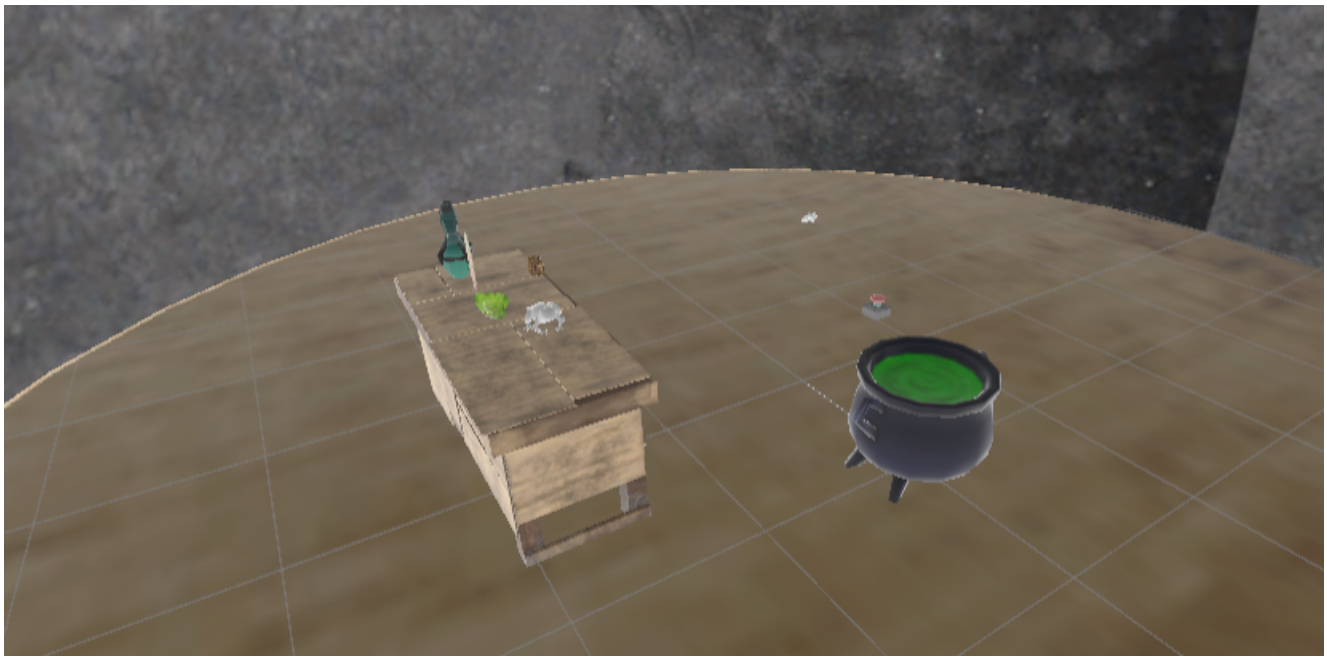


Figure 10: Final game environment before play is started, sans hand models.

This concluded the development time. Unfortunately, there were a few small features which could not be implemented successfully, and a few features which were out of the scope of the development time.

The most notable feature which was cut was a menu system. The menu system was attempted at multiple points during development, but caused major issues with core game systems. Additionally, the menu was planned to be implemented before the tallying of potions, as the menu would determine how many potions would need to be crafted for the game to be cleared. As is probably obvious, since the menu system failed, the tallying of potions was not implemented.

As was mentioned before, the code is built to support any number of ingredients for future expansion. Additionally, in the future, the greatest thing to implement would be timer-based functionality. For example, each recipe could be on a timer before it despawns and is considered a failure, or the cauldron could take some time in order to cook up a potion. These timers could create a sense of urgency in the game that is currently not present, but was beyond the scope of this project. Lastly, we believe it would be a nice addition to make the cauldron stirrable, but the implementation of such a feature would certainly add a large amount of development time.

Testing and Evaluation

The main bug that would occur during development was the gravity of objects being disabled.

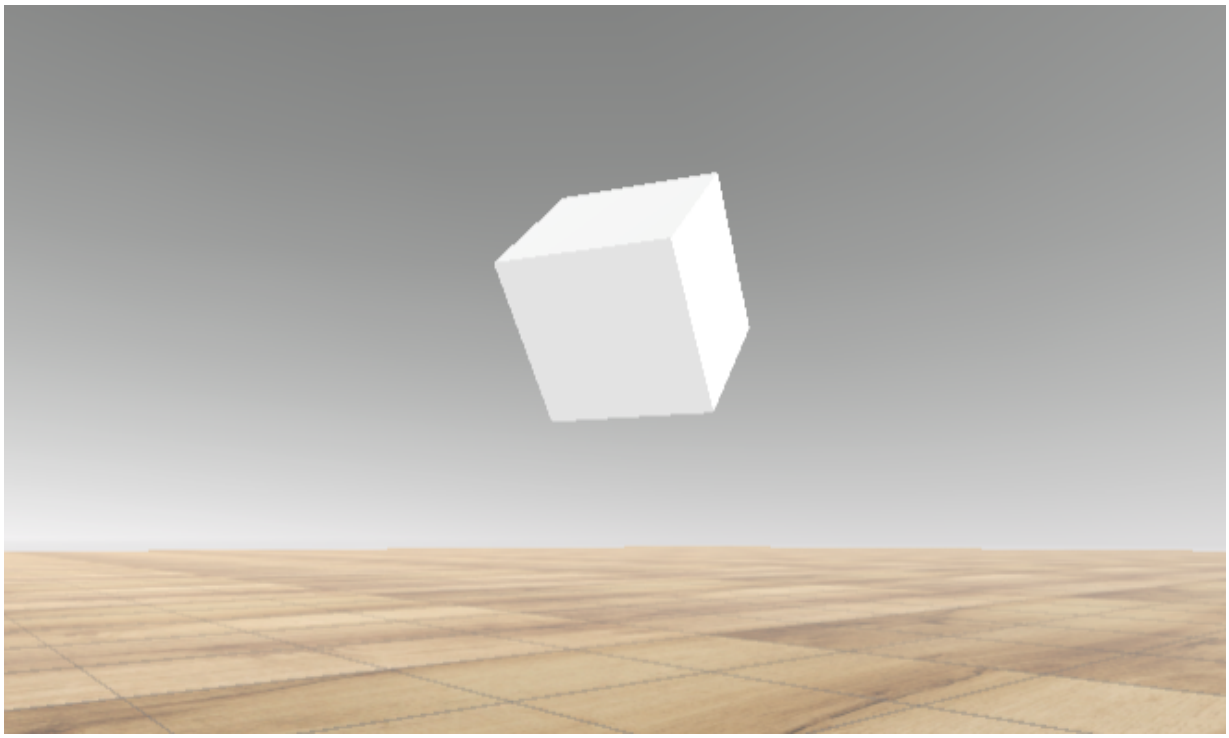


Figure 1: Floating cube (placeholder ingredient object)

This would frequently occur after an ingredient was added to the cauldron for the first time. Once an object's gravity was disabled, it was usually useless and could no longer effectively be added to the cauldron or interact with the game world. There were varieties of reasons why this occurred, but the most common was a disabling of components such as the rigidbody when a new object was spawned in. This error was corrected in code by ensuring all objects were spawned identically and in the prefabs by ensuring every prefab had the same components enabled by default.

Throughout testing, recipe cards would frequently pile up in excess quantities when the game was left running (normally while testing other aspects of the game).

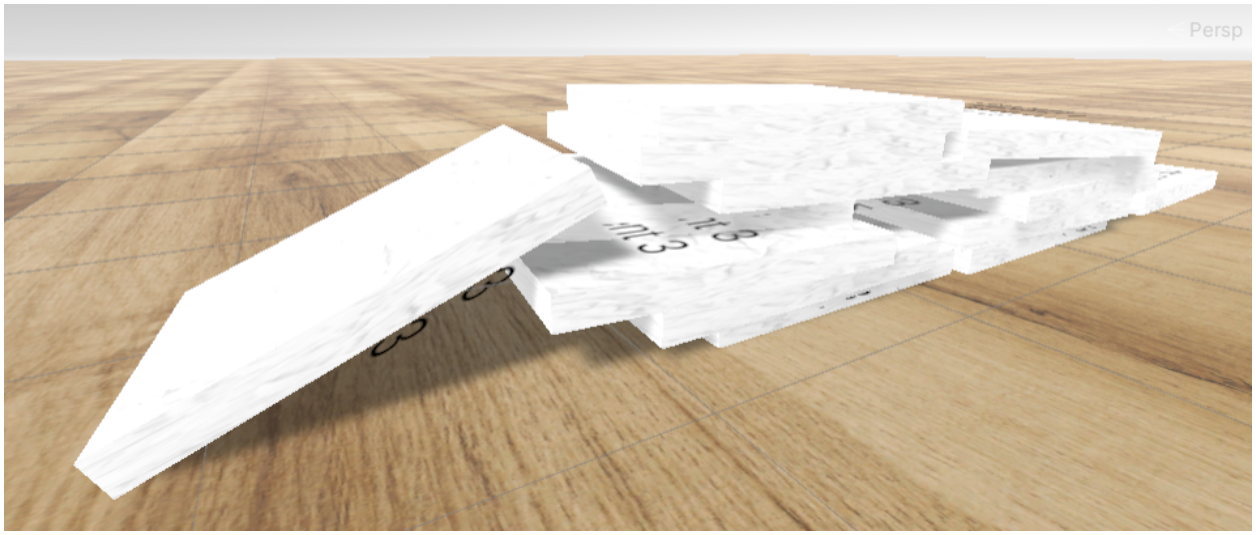


Figure 2: A pile of recipe cards, the quantity of which steadily increased over time.

This actually ended up being a good stress test for the game logic, as it was vital that completing one recipe did not delete all identical recipe cards from the stack. The intended game mechanic is that one completed potion correlates to one card, like completing an order at a restaurant.

The next notable bug during testing came with generating ingredient cards with more than three ingredients.

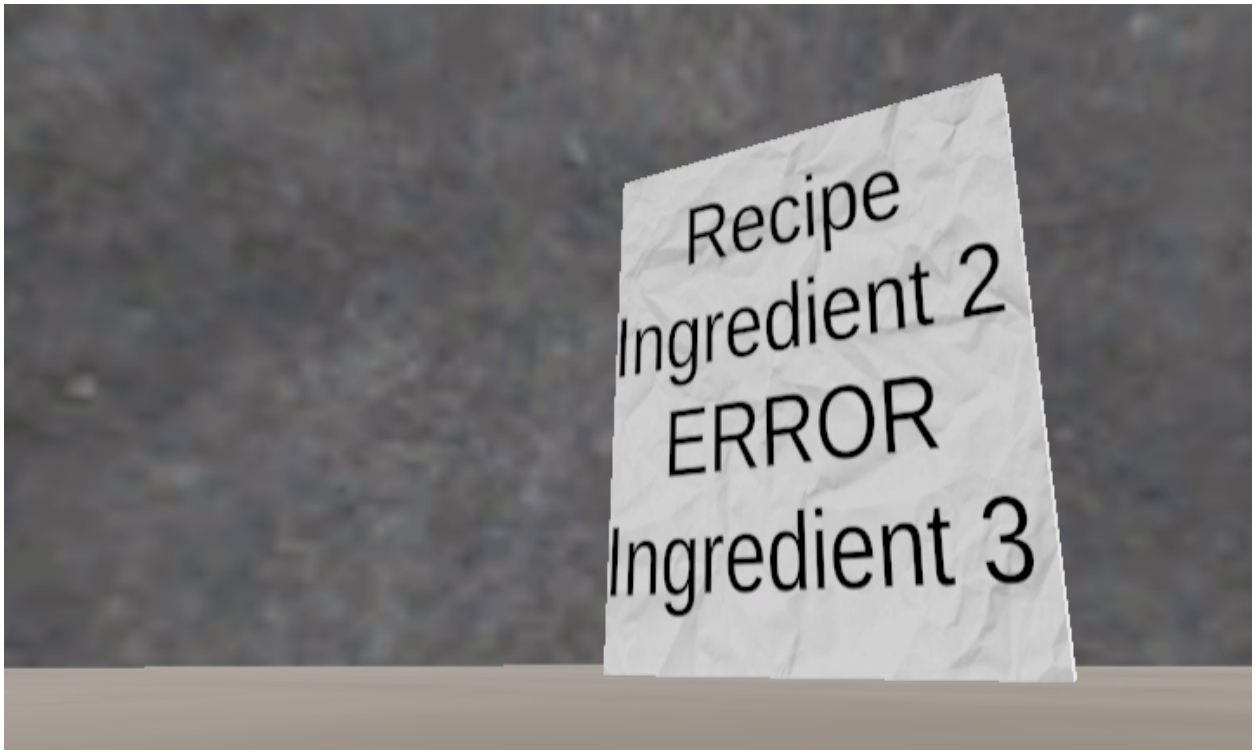


Figure 3: Testing ingredient card with an error.

This bug was present due to insufficient support for all ingredients. In order for the bug to be squashed, each ingredient had to be separately evaluated when it came to generation, but once this evaluation was completed, the bug was squashed quickly, and recipe card generation worked as intended.

Once the VR environment was decided, important testing was done with the edge of the play area. First, the play area was edge tested so the player could not leave the cave. Second, it was imperative that ingredients at any area of play were still grabbable, so ingredients were placed at the edge of the cave in order to test.

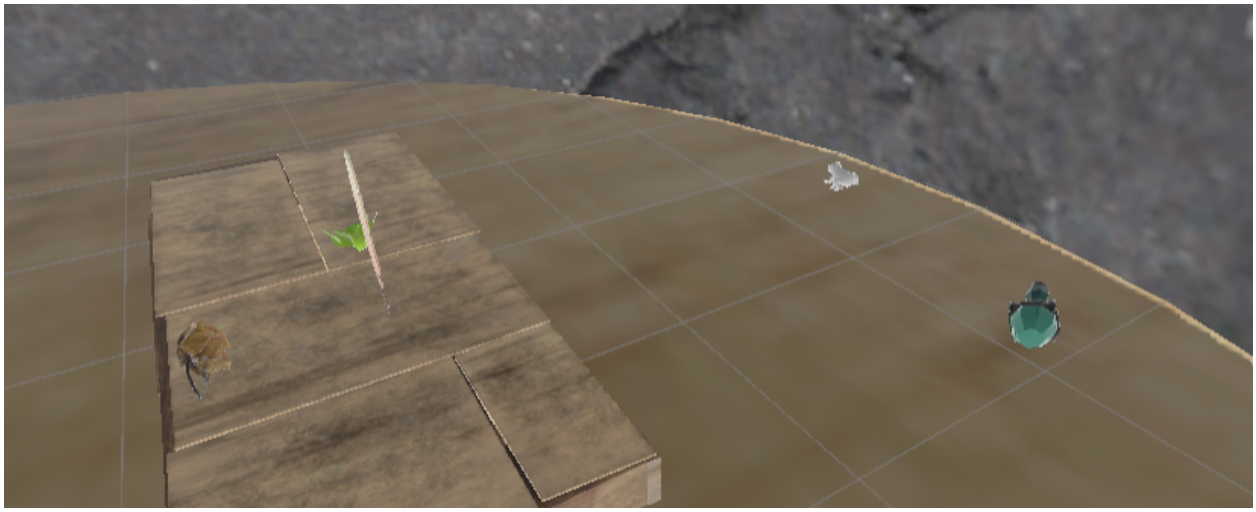


Figure 4: Some ingredients placed near the edge of the VR environment.

Thankfully, with the grabbing system that was already implemented, objects can be grabbed at any point in the cave, meaning no changes had to be made to prevent bugs.

Once all systems were in place, the next logical evaluation to complete was continuous testing of the game to ensure no random bugs would appear. During this testing, the only issues which occurred were minor. Oddly enough, during the demonstration video, intended physics resulted in a curious situation. The elixir ingredient was grabbed, but due to a unique angle, it collided with a feather ingredient while being grabbed, knocking the feather away.

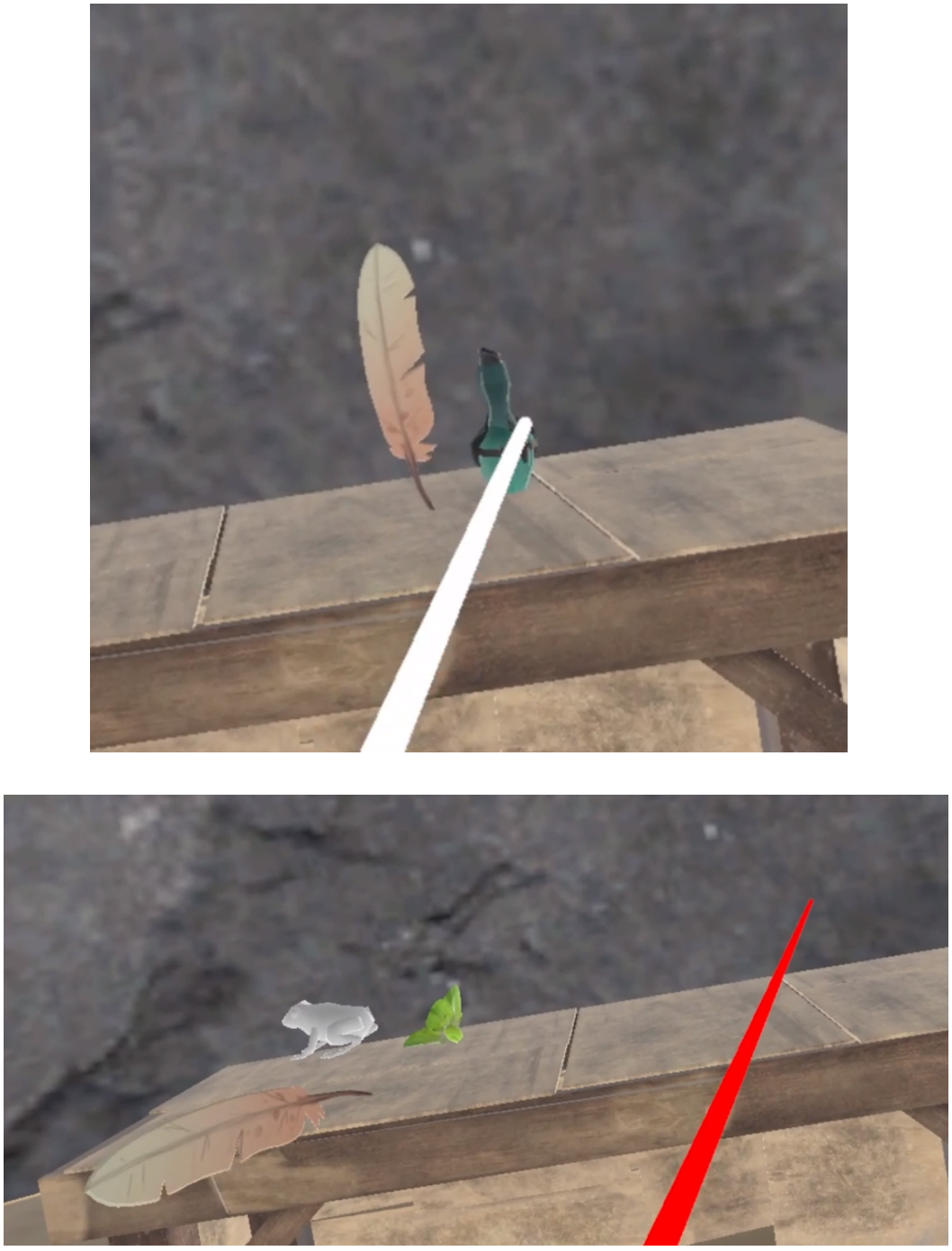


Figure 5: The feather situation described above.

This feather then proceeded to fly directly into the cauldron, adding it to the current potion. This behavior is not a bug, and we decided to keep this in the demonstration video to show how

managing ingredients when in hectic situations may result in mistakes, which adds to the fun of the game.

No major changes occurred as a result of the final batch of testing, as no major bugs appeared. After the minor issues were resolved (which involved changing very few lines of code), the project was declared complete.

References and Appendices

Models

Cauldron model:

https://sketchfab.com/3d-models/cauldron-6a7432eef5864d3bafa1911efb6b2c86#download

Elixir model:

https://sketchfab.com/3d-models/elixir-bd5e6fa3686e4017bcd2285ffe02a648

Feather model:

https://sketchfab.com/3d-models/day-1-feather-771e62e4f43d4c3c81f7d35a8884578c

Plant model:

https://sketchfab.com/3d-models/lettuce-1dfd949d61ae4d378d7c65571746f693

Rose model:

https://sketchfab.com/3d-models/dried-yellow-rose-829c589920cd4991bd563f383956ec4a

Table model:

https://sketchfab.com/3d-models/wooden-table-8a5b41d6445c4f1fbefb2e4abfeebb0d

Toad model:

https://sketchfab.com/3d-models/le-toad-3425b8e4aad348c490f50d12933861b8

Textures

Paper texture:

https://www.pexels.com/search/paper%20texture/

Stone texture:

https://www.pexels.com/photo/gray-rock-8892/

Wood texture:

https://www.freepik.com/free-photo/natural-wooden-background_5505940.htm#query=floor%20texture&position=14&from_view=keyword&track=ais">Image

Sounds

Background music:

Scanglobe- Extramundane 2 from https://freemusicarchive.org/genre/Ambient

Confirm sound:

https://freesound.org/people/JustInvoke/sounds/446114/

Deny sound:

https://freesound.org/people/philRacoIndie/sounds/551543/

Splash sound:

https://pixabay.com/sound-effects/splash-6213/