Cauldron Chaos

Christoph Daxerer*

Software Engineering ba. VZ FH-Oberösterreich Hagenberg

ABSTRACT

For the V1_VIREIL Virtual Reality course this paper describes the design and impelemntation of a potion mixing game for the HTC Vive.

1 INTRODUCTION

This paper

2 DESIGN

When designing a VR game it is crucial to understand the inherent nature of the medium. By addressing the strengths and weaknesses of VR already in the design phase, we can avoid many of the pitfalls that plague VR games.

2.1 Possibilities are Restrictions

The HTC Vive features a vast set of interactions. Controllers, buttons, touchpads, all fully tracked. Although this is a great strength, it also means that the player has a lot of freedom.

Often, freedom leads to confusion. To counteract this the decision was made not to limit the player, but to engineer towards their intuition and expectations. Leading to the following design decisions:

- **everything** the player expects to be interactable should be interactable
- controls should be intuitive and minimal
- the player should be able to move around freely

2.2 Core Gameplay Loop

The game is quite simple to learn. Throw ingredients into the cauldron in the correct order to create the desired potion. One player is the alchemist (in VR) and the other is the assistant (on the computer).

By navigating through a brewing graph the assistant can find out which ingredients are needed and in which order. The alchemist can then search for the ingredients in the room and throw them into the cauldron.

2.3 Ingredient System

Every ingredient is made up of *parts*. Each part has a *material* and an *amount*. Together they make up the composition of the ingredient.

Materials follow a hierarchy. An example provides figure 1. This means that 50 Oil isn't just 50 Oil, but also 50 Liquid and 50 Base. Important to note is that consuming a material also consumes the same amount of all its parent materials. An intended side effect of this is that consuming from a material higher up in the hierarchy decreases the root material, while maintaining the material of all its children.

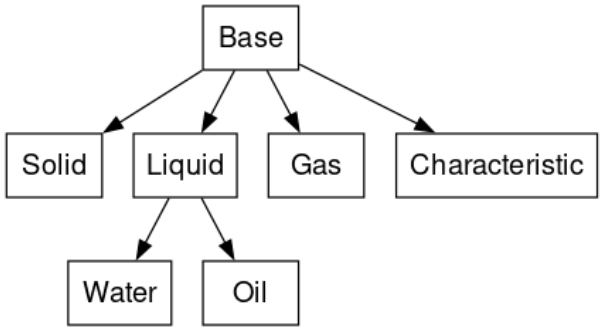

Figure 1: An example of the material hierarchy.

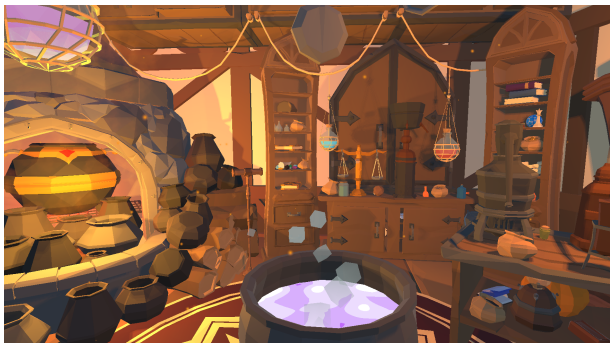

Figure 2: The initial view for the alchemist.

2.4 Engineering for Intuition

The alchemist is immediately thrown into the game. Instinctively, the first thing they will do is **orientate themselves**. This is why the game starts with the alchemist standing in the middle of the room as seen in figure 2.

After 5 seconds an **audible and visual cue** delivers the first task. After communicating with the assistant a new problem promptly arises for the alchemist: "What are ingredients?".

Intuitively, the alchemist will try to pick up the first thing his brain classifies as "pick-up-able". Because almost everything in the room is an ingredient this will work.

Having the object in his hand the alchemist will most likely either throw it into the cauldron or come to the next problem: "How do I know what to throw into the cauldron?".

While working out what materials are needed, the alchemist needs to scout the house for ingredients. This is where his curiosity will lead him to try and interact with the **Ingredient Inspect Station**.

*e-mail: christoph.daxerer@chello.at

3 Implementation

States and ingredient materials are implemented as Unity's `ScriptableObjects`. Figure 3 shows the class diagram of the most important classes.

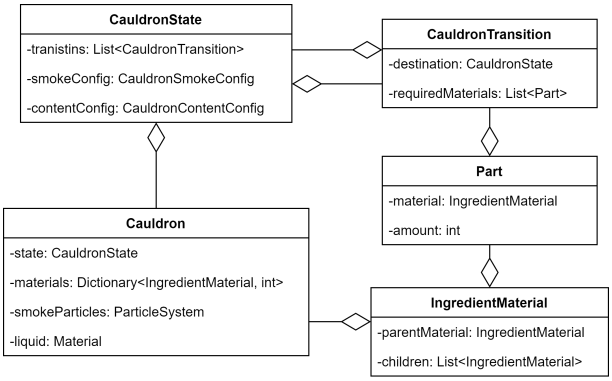

Figure 3: Class diagram of the most important classes.

To keep track of the materials, states and transitions, they are tracked in a .dot file. By using a simple .dot parser and some custom editor scripts, the graphs are set up in Unity automatically.

3.1 Dot Parser

The .dot parser is kept quite simple and is based mainly on the three regular expressions shown in listing 1.

```
(?:\n *)(\w+)\s*(?:\[(.*?)\])?; //nodes
(\w+)\s*->\s*(\w+)\s*(?:\[(.*?)\])?; //edges
(\w+)\s*=\s*(.*?); //attributes
```

Listing 1: Regex Patterns

The parser then returns a simple graph data structure.

3.2 Ingredient Materials

Ingredient materials have a name, a parent material and a list of child materials. This allows navigating the material hierarchy in both directions.

3.3 Cauldron States

The cauldron state graph is a directed graph with states as nodes and transitions as edges. Each state holds:

- A list of `CauldronTransitions` that lead to other states
- A `CauldronContentConfig` that holds configuration for the cauldron content shader
- A `CauldronSmokeConfig` that holds configuration for the cauldron smoke particle system

These three components are all just `Serializable` classes as they are not reused anywhere else outside their state.

3.4 Cauldron Transitions

The transitions are defined by a set of required parts (materials and amounts) and a destination state. Using a simple function that checks if the cauldron contains the required parts, the parent state greedily selects the first transition that is valid.

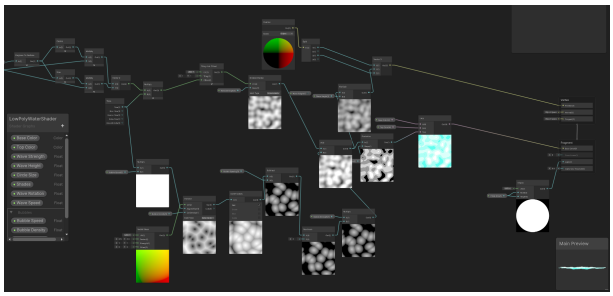

Figure 4: The cauldron content shader.

3.5 Cauldron

The cauldron is in essence just a map of materials to amounts. Whenever an ingredient is thrown into the cauldron the amount of the materials is increased according to the parts and the material hierarchy. The cauldron also has a current state that is checked for transitions whenever an ingredient is added.

The cauldron implements the `IObservable` interface and notifies all observers whenever the state changes.

The cauldron also provides functions that apply `CauldronContentConfig` and `CauldronSmokeConfig` to the shader and particle system.

3.6 Cauldron Content Shader

For the shader a combination of different noise textures along with posterization, lerp and vertex displacement is used. Figure 4 shows the shader graph for the cauldron content shader. The waves are created using a gradient noise texture and the bubbles are created using an inverted voronoi noise texture.

Various parameters exposed by the shader are used to configure the shader at runtime.

Vertex Displacement is done "before" posterization to avoid jumping vertices and get a smoother effect. The posterization gives the shader a more cartoony look fitting the low poly style of the game.

3.7 Ingredients

Ingredients are `MonoBehaviours` that have a name and a list of parts. The script requires a `Throwable` component from the SteamVR plugin.

3.8 Ingredient Inspect Station

To allow the alchemist to inspect ingredients and their composition the `IngredientInspectStation` is used. Using a trigger area the station detects when an ingredient is placed on it. Displaying the ingredient's name and composition is done via a `TextMeshPro` component. To animate the text a simple coroutine is used.

3.9 Audio System

To make playing audio clips in 3D space easier, a simple audio manager and helper class is used. The audio manager has a pool of `AudioSources` that can be used to play audio clips. The helper class provides a simple interface to play audio clips at a given position.

4 Stumbling Blocks

4.1 SteamVR Don't Destroy On Load

The SteamVR player prefab comes standard with the *DontDestroyOnLoad* flag set. Together with default Throwable setting of not restoring the parent on release, this leads to the duplication of the player and everything he held at some point when reloaing the scene.

This was fixed by setting the *Do Not Destroy* flag to false in `Player -> SteamVRObjects -> [SteamVR]`.

ACKNOWLEDGMENTS

The author wishes to thank Schmidt Alexander, Schachinger Julian and Unterberger Bernhard for testing and providing valuable feedback.