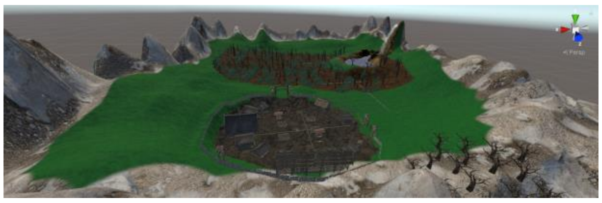
III. METHODOLOGY

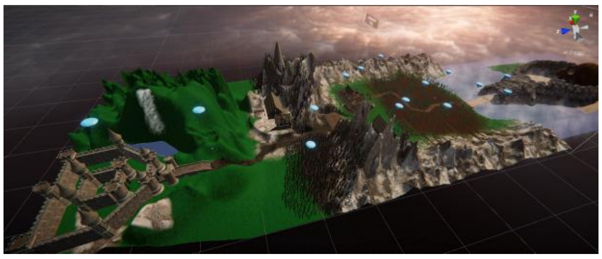
1. Scene Management

1.1 Terrain:

To start with. We looked up how to make a map. After searching for a bit, we discovered the terrain and researched how to create and design one. The terrain is the same as a plane but equipped with a lot more options. It has brushes to raise or lower parts of the terrain and paint textures onto the terrain. With these features having a terrain will give us a bigger range for editing and designing a map. We searched for tutorials and watched people edit terrains. As a result, we found two packages that helped us immensely in designing our map. The first is the "Terrain Sample Asset Pack" on Unity's store. The second was "Terrain tools" . It allows us to have an easier time switching between terrain textures, painting textures, and even blending the textures for a more realistic look. With these tools in hand, we started designing the map, looked for free textures to use on "PolyHaven'' and designed our scene. This was the first design of our scene:

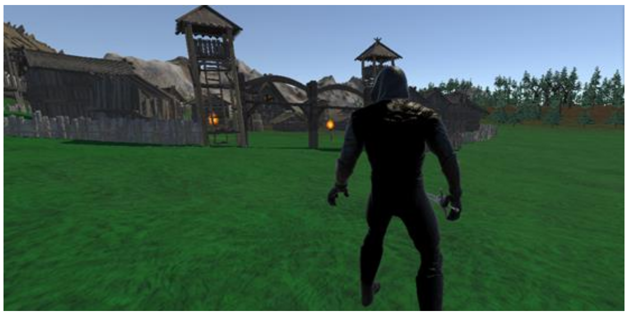


Later on, we redesigned it and added more parts to the map. This is the final scene below



1.2 Lighting:

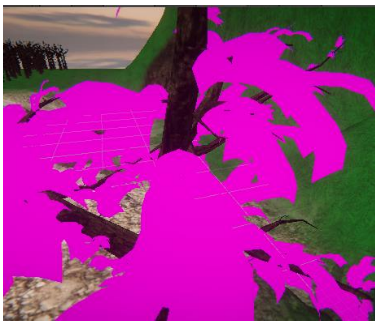
We want to have a simulation of realistic lights in our scene which makes the scene feel realistic. We implemented a directional light which lights up our scene and has 23 spotlights which act as fire illuminations. As we can see here, the scene does not look visually appealing like colors, and the lights do not look realistic at all.



To get over this problem, we need to bake the lights in the scene and add some light Probes in it.

1.3 Shader Graph:

Our aim in this game is to make a realistic looking open world game. In order to do so and to incorporate various topics learned, using a shader graph to simulate real water seemed as a viable option. The troubles faced with this technique was the need to switch the render pipeline to URP instead of the standard pipeline. Unfortunately, this change drastically affected our scene not only by turning most of the materials pink, but it also affected some assets and their functionality. Furthermore, our scene had incorporated high quality trees in the forest area and changing to URP caused the leaves of the trees turn into a paper-like shape, not standard leaves, on top of being pink as seen below:

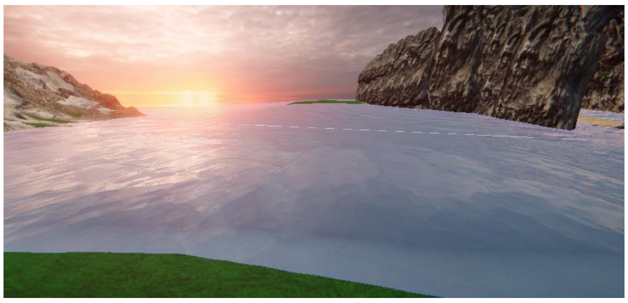


This had occurred due to the fact that the Prefab was using a custom shader that was incompatible with URP. Luckily, we were able to gain the realistic shape and appearance of the leaves when changing the shader to "*Universal Render Pipeline/Lit*". On the downside, we had lost features the asset was providing which was wind effect and so on.

Moving on to the water shader graph. To simulate realistic water there were some features that had to be taken into consideration:

1. Reflection based on Camera and Light.
2. Depth of water.
3. Wave direction and speed.

In the end this is the look we ended up achieving. Following examples others have tried it did the job.

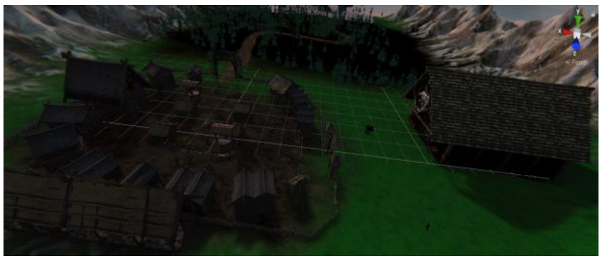


1.4. Building/Settlements:

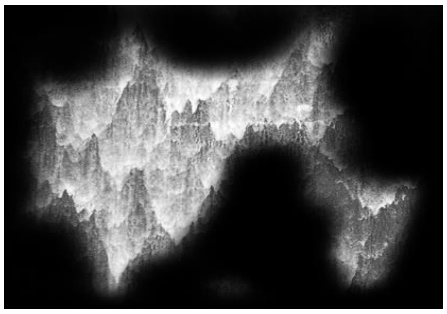
We decided to build a village where the player can interact with some NPC. There are 16 buildings and other props that make the place look habitable. Another building outside the village is a lounge where the NPCS and the player can have fun and listen to music, and there is a forest at the end of it there is a cave that is surrounded by water. We made these scenes to make the player explore the scene and have fun. However, these models had a lot of vertices and triangles. So they gave us some issues since they require more processing power CPU and GPU to render.

1.5. Particle Systems:

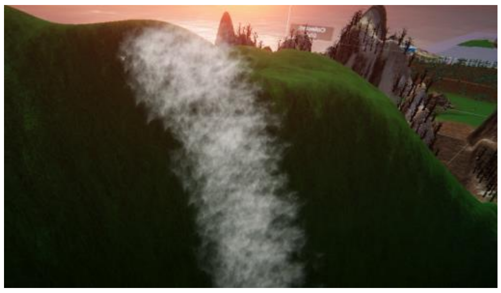
In our game we used the particle system to give a realistic effect of fire, fog and a waterfall.
In order to replicate real fire a realistic texture was required. To make it more unique we created our own texture using a simple brush on *Procreate* (which is an iPad app) to give the effect of each the fire and smoke particles and water for the waterfall.



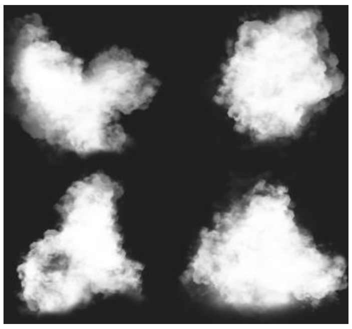
Firstly, let's address how the waterfall was designed. Below here was the texture created using an image and erasing the sides keeping a faded edge.



The point of all of this is that when in Unity, to be able to make the particles flow more realistically, we would need to choose the shader "*Legacy Shaders/ Particles/ Additive-Multiply*" this shader helps in creating a glow effects when the particles are added on top of the other like how water gets brighter when more of it flows together. As we can see here this was the final result for the waterfall which originates at the top of the mountain with the gravity modifier set to 10 to make it fall downwards.



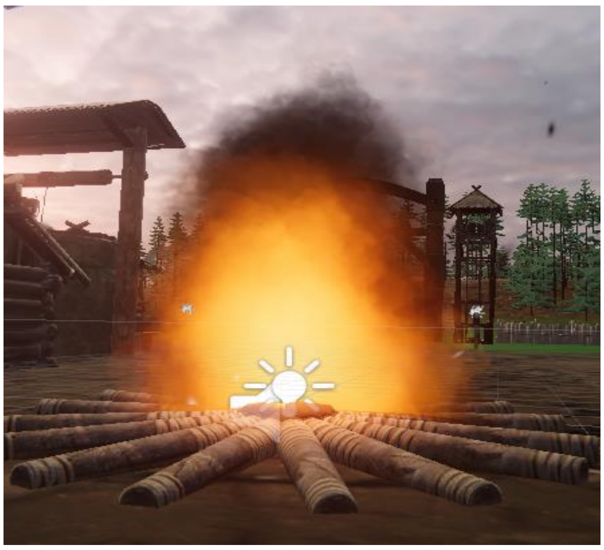
Secondly the fire. This was slightly more complicated than the waterfall to complete. To begin with, the textures manually created was sectioned off into 4 parts as seen here:



The point was to create options for the particle system to alternate through so that each particle generated would be a random one chosen from these 4 textures to create a more life-like transition of the smoke and flames. For the fire itself the color was set to change over the lifetime from bright yellow to red.

Both the smoke and fire had the same texture, but the only difference was that the material created to be used for the smoke was set to the shader "*Legacy Shaders/ Particles/ Alpha Blended*" which is used to display the dark color since the fire material used the

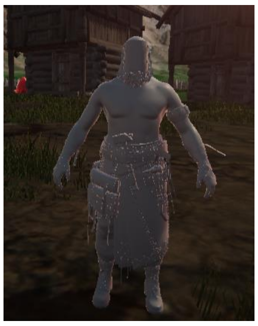
additive shader that doesn't show the dark colors due to the transparency effect. To add on, for the fire a light object was added to the particle system to give the effect on the area surrounding the fire that it emits light.
Finally, a third particle system was added to create the sparks, which was simple as no custom texture was made. It was made to delay as in real fire, the sparks start a few seconds after the fire begins, and for the Renderer Mode, the "*Stretched Billboard*" was used to mimic long flecks of sparks.
For all the particle systems they were made to rotate, or noise was added to create a sense of realism with air and other elements affecting the fire.
Below is the final result of the fire:

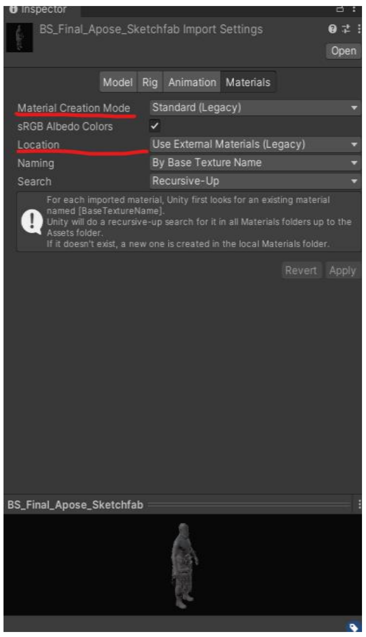


Lastly for the fog a similar approach was taken to the waterfall, which was a custom texture: and material using the additive shader to glow when many are on top of one another. Also, a rotation was applied, and the difference from the other particle systems was the *Prewarm* option was ticked so that when the game starts, the particles would be already displayed with no delay.

1.6 Animation & 3D Models:

For characters, we looked for free 3d models online. We got most of our 3d models from "Sketchfab.com", a website where people upload their 3D models, and some are free to download. We also used Mixamo for some of the 3d models, a website where developers can find rigged characters and it has a wide variety of animations free for anyone to download. The first problem was that some of the 3D models did not have the materials working, resulting in the 3D model being all white without any materials.





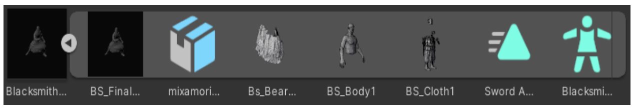
BS_Final_Apose_Sketchfab Import Settings

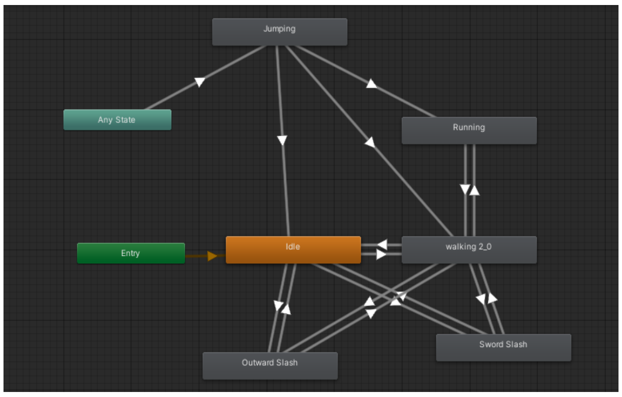
Material Creation Mode: Standard (Legacy)
sRGB Albedo Colors: ✓
Location: Use External Materials (Legacy)
Naming: By Base Texture Name
Search: Recursive-Up

For each imported material, Unity first looks for an existing material named [BaseTextureName].
Unity will do a recursive-up search for it in all Materials folders up to the Assets folder.
If it doesn't exist, a new one is created in the local Materials folder.

After some research, we found two solutions to the problem. One was to change the Material Creation mode and location in the materials section of the 3D model's FBX to Legacy. This lets the FBX locate the textures downloaded with it, not those in the actual FBX. The second solution was to create the material using the textures we got with the 3D model.

As for animations, we uploaded the 3D models we got on Mixamo. It gave us a wide variety of animation and it is where we got all of our animations. Mixamo also has an auto-rigging function when anyone uploads a 3D model without a skeleton. After getting the 3D model and the animation we simply created an animator controller, extracted the animation from the FBX, and then added it to the animator controller. Finally, the only step left is to write the logic for the character to do the animations at the right time.

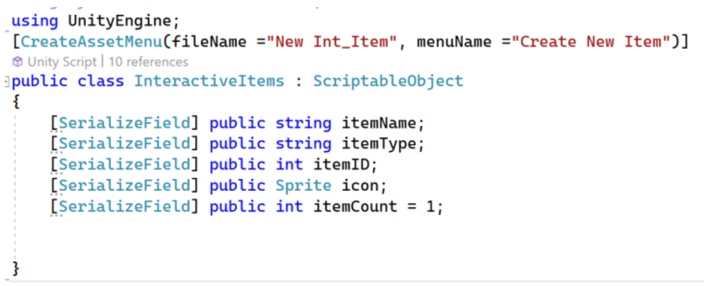




1.7 UI/ Inventory System:

To create an interactive game, we introduced an inventory system where the player goes around the scene and collects these items which in our game are crystals.

To implement this system a *Scriptable* object which is a class in Unity that you can use to make multiple instances of a custom set of attributes of your choosing. For example, in this game we created a class called *Interactive Items* that held the following attributes:

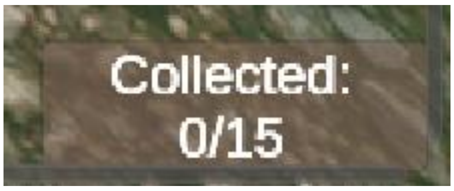


```
using UnityEngine;
[CreateAssetMenu(fileName ="New Int_Item", menuName ="Create New Item")]
Unity Script | 10 references
public class InteractiveItems : ScriptableObject
{
    [SerializeField] public string itemName;
    [SerializeField] public string itemType;
    [SerializeField] public int itemID;
    [SerializeField] public Sprite icon;
    [SerializeField] public int itemCount = 1;


}
```

In this way for every crystal added to the scene we could give it a custom name, icon etc. to be used in the inventory system itself that will be displayed to the user.

A small manual animation was applied to the crystals where when the mouse was hovering over them the crystal would move up slightly to indicate to the user that they should collect this item.

An Inventory bar was set to the side to be less intrusive to the player and would only appear when a crystal is collected as well as expanding to the size of how many slots were in the inventory bar. This was achieved by using a *"Vertical Grid Layout"*.



Collected:
0/15

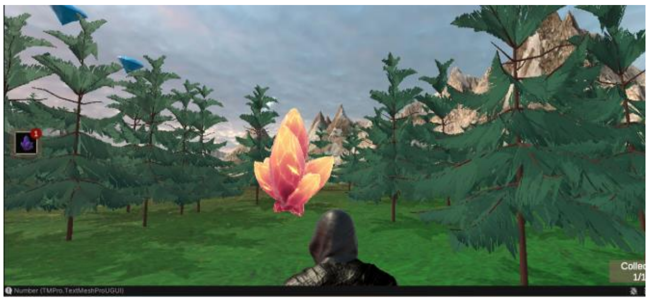
This box was added to the corner to help the player know how much they have left till the level is completed.

The inventory system algorithm works in the following way which is enacted once the player pressed on the collectible/ interactive item:

```
Item pressed{
                    Item_list.add(colle
       cted_item);

       Destroy collectible GameObject;

       If (collected_item already
       exist)
              Increment value of item
              by one;
Create new inventory slot using
      collected_item image from
the Scriptable Object attached
to it;

Item_count++;

If (item_count == total)
      Display level complete
      message and fireworks;

      Restart scene;
}
```

This script is added to an empty GameObject which then acts as a manager for this UI inventory system.

<u>UI final look:</u>



1.8 <u>Postprocessing:</u>

We want to make some effects in our scene, like color adjustments in our scene and some effects when we enter the building in the scene.
The initial step to do is to create Three post-processing volumes, namely the "Global Volume," and "Box Volume Storage1", "Box Volume Storage2" These volumes were placed under a parent object named "LightEffects." The "Global Volume" was intended to cover the entire map, while the "Box Volume Storage" targeted a specific area inside the building named "Storage" and the other one was put inside a building called "Longe".

We Need To Adjust Some Post-Processing Profiles:

To achieve the desired cinematic view / realistic view and visual enhancements, adjustments were made to the post-processing profiles of both volumes. The "Global Volume" profile included tone-mapping adjustments to control the overall brightness and contrast, lift gamma gain for color correction, and color adjustments for fine-tuning the colors in the scene. A vignette effect was added to create a gradual darkening around the edges of the screen, enhancing the cinematic feel.

Implementing Bloom and Depth of Field:

The post-processing profile for the "Global Volume" incorporated the bloom effect to create a realistic glow around bright objects, adding a touch of visual flair. Depth of field was also configured to control the focus distance and simulate camera lens blurring, drawing attention to specific points of interest in the scene.

Specialized Effect for "Box Volume Storage" And for the "Longe":

The "Box Volume Storage" profile was designed to provide a unique visual effect for the specific area inside the building. Adjustments were made to the bloom effect and color adjustments to create a distinct atmosphere that fits the interior environment.

White Balance, Chromatic Aberration, and Motion Blur:

To further enhance the cinematic experience, the post-processing profiles were fine-tuned with white balance adjustments to control the overall color temperature. Chromatic aberration was added to simulate a subtle color separation effect often observed in real-world cameras. Motion blur was also implemented to create movement and dynamism in the visuals during fast-paced gameplay.