Chapter 1

User Experience and Interaction

The game is separated into three main scenes, the starting scene, the creator scene and the play scene. Each scene has its own purpose and corresponding functionality.

1.1 Starting Scene

When the player launches the game he is immersed in the starting scene which serves as a home or main menu area. Inside the game the user can use the left controller joystick to move and the left controller joystick to look while the triggers are used for user interface elements such as buttons. When navigating in the starting area the player can find the doors that will transfer them to either the creator scene, the play scene or exit the game. When the create world door is approached, a user interface will be displayed with a prompt to the player about creating a world.

When the player is in front of the load world door a user interface will be displayed asking the player what the desired world to be loaded is based on all created worlds that are in the user's files. After a world is selected the game will load the selected world in the play scene so that the players can play the story.

Exiting the game requires a bit of further exploration. After the exit game wall is found the player has to follow the path to the exit game door where a user interface will be displayed asking the player if they want to exit the game.

Figure 1.1: Left: Create World Door, Right: Load World Door

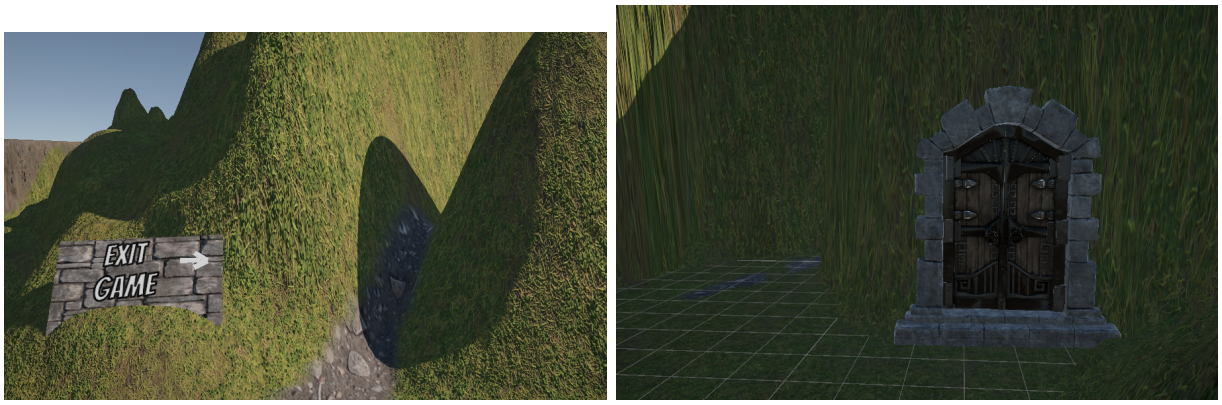

Figure 1.2: Left: Exit Game wall, Right: Exit Game door

1.2 World Creator

1.2.1 Gameplay

The purpose of the world creator is for one or more players to use the available tools and build a world of their liking in a collaborative manner. When entering the create world scene the tutorial interface shows up explaining how creator mode works to the player. In the world creator the player can move up and down in space by using the right controller joystick. In the scene only a blank terrain is visible to the player on which actions can be applied that form and give colour to the terrain while objects can also be placed on it. The objects that are used can be edited by using the object editor interface that shows up when a summoned object is activated.

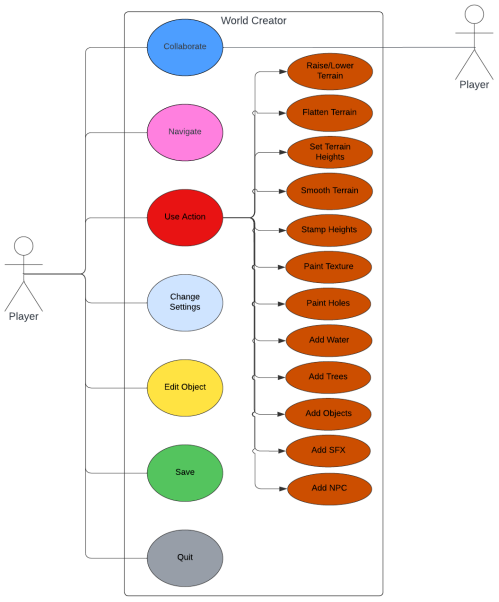

Figure 1.3: World Creator Use Case Diagram

1.2.2 Actions

In world creator mode the player has a tool palette which is essentially a user interface where they can switch between tool actions for terrain manipulation and object placement. When the player holds the left controller grab button the tool palette will be showed. In the outer area of the palette all the selectors for the various tools are placed. These selectors can be activated with the right controller trigger button. There are also settings that are used globally by the terrain tool brush while some actions also have their own specific settings and extra selectors, which can be selected again by using the right controller trigger.

The terrain tool brush settings are:

- Brush size: affects the area size of actions on the terrain
- Brush strength: affects the intensity of actions on the terrain
- Height: sets the height for the set heights action
- Tree density: sets the desired tree density for painting trees on the terrain

- Circular brush: toggles between rectangular and circular brush shape

The available actions of the world creator are:

Raise/Lower Terrain

As the name suggests this action raises or lowers the terrain. If the raise/lower toggle of the raise/lower action is set to true the action will raise the terrain, otherwise the terrain will be lowered.

Figure 1.4: Raise/Lower Terrain

Flatten Terrain

The flatten terrain action sets the heights of the terrain to the lowest point for area equal to the specified brush size.

Set Terrain Heights

The set heights action raises or lowers the terrain height to the specified height slider value.

Smooth Terrain

The smooth terrain action smooths terrain heights in area depending on the brush size.

Stamp Heights

The stamp heights action, when selected, will display four 2D heightmaps that are available for "printing" on the terrain, the slope, canyon, hill, erosion and cracked earth

heightmaps. When the action is triggered on the terrain the selected heightmap will be printed with size equal to the brush size.

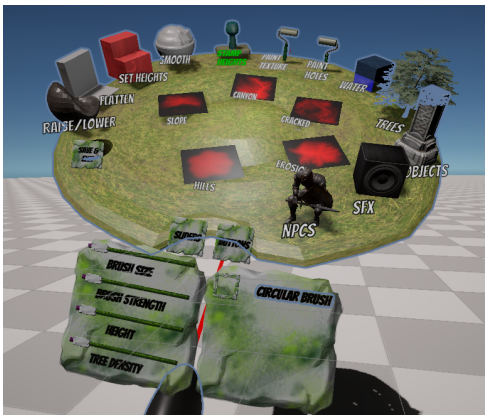

Figure 1.5: Stamp Heights

Paint Texture

The paint texture action allows the player to plaint the terrain given the 4 available textures: grass, sand, black sand and pebbles. The painetd area will have size equal to the brush size.

Figure 1.6: Paint Texture

Paint Holes

The paint holes action creates holes on the terrain when the hole toggle is set to true,

while when the toggle is set to false, existing holes can be turned to terrain again.

Figure 1.7: Paint Holes

Add Water

When the add water action is selected, water cubes, spheres, boxes and capsules will appear, each shape having three different available shaders. After selecting one of these available objects the player can then place them on the terrain.

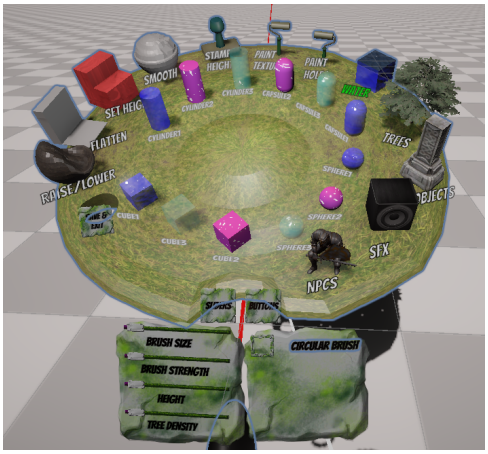

Figure 1.8: Add Water

Add Trees

The add trees action when selected will display four tree types, bare, small, medium and tall, that the player can select to place on the terrain. For the add trees action there

are the remove trees and the mass paint/remove trees toggles. When the mass paint remove trees toggle is set to true trees, will be added in an area determined by the brush size, otherwise a single tree will be placed at the point of activation. The remove trees toggle, when set to true will enable tree removal in an area determined by brush size when the mass paint/remove trees is set to true or removal of a single tree at position close to action activation.

Figure 1.9: Add Trees

Add Objects

The add object action when selected will display several object selectors for the available objects along with two buttons for navigating the available object pages, the previous and next page buttons. The player can place an object on the terrain after selecting the corresponding object selector and using the right controller trigger on the desired terrain point.

Add SFX

The add SFX action when selected will display several SFX selectors for all available SFX. The player can place an SFX object on the terrain after selecting the corresponding SFX selector and using the right controller trigger on the desired terrain point. The SFX renderer active toggle will show the mesh of the SFX object box on all clients when set to true, while when set to false the mesh renderers of SFX objects will stop being displayed.

Add NPCs

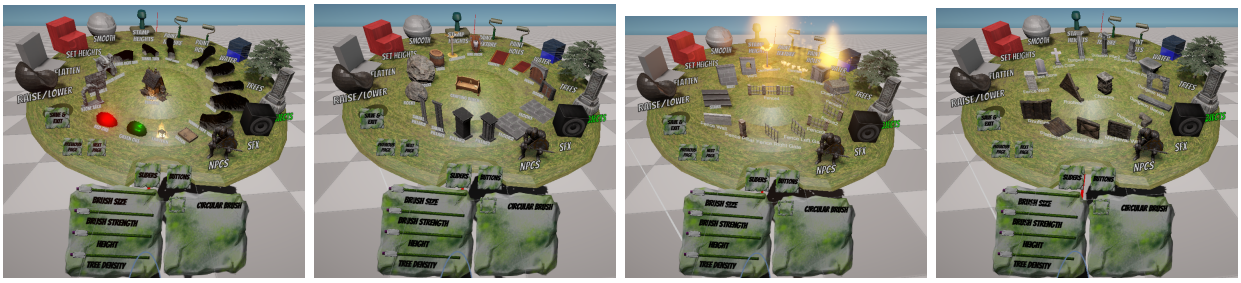

Figure 1.10: Add Object(From left to right: Page 1, page 2, page 3, page 4)

Figure 1.11: Add SFX

The add NPC action when selected will display two available NPCs, the ghoul and the shade. The ghoul NPC is a simple enemy while the shade NPC has a more complex behaviour. The player can place an NPC on the terrain after selecting the corresponding NPC selector and using the right controller trigger on the desired terrain point. The enemy agents active toggle will enable the navigation mesh agents on enemies on all clients when set to true, while when set to false the navigation mesh agents will be disabled for all enemies on all clients.

1.2.3 Object Editor

After any water, object, SFX or NPC is placed on the terrain, the player can use any controller trigger button while targeting that object to open the object editor interface and activate the object editor or close it and deactivate the editor if it was previously

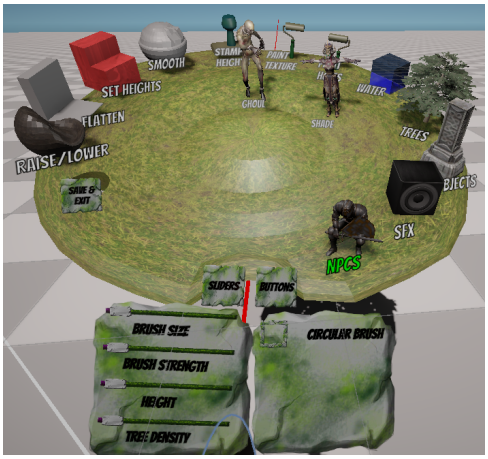

Figure 1.12: Add NPC

activated. There are four options given: move, rotate, scale and delete.

- Move: hold a grab button while aiming at the object to move it around
- Rotate: hold grab with a selected object to rotate it according to the controller's rotation
- Scale: hold both grab buttons while aiming at an object and bring the controllers close to scale down or move the controllers away from each other to scale up
- Delete: show the delete interface prompting the player if they really wish to delete the object

1.2.4 World Saving

When the world is shaped according to the players' desire, each player can press the save button located on the left side of the palette to save the world locally in their file system so that the created world can be available for loading in the play scene. Once every player has saved the world they can hold the left controller secondary (Y) button to quit the world creator scene.

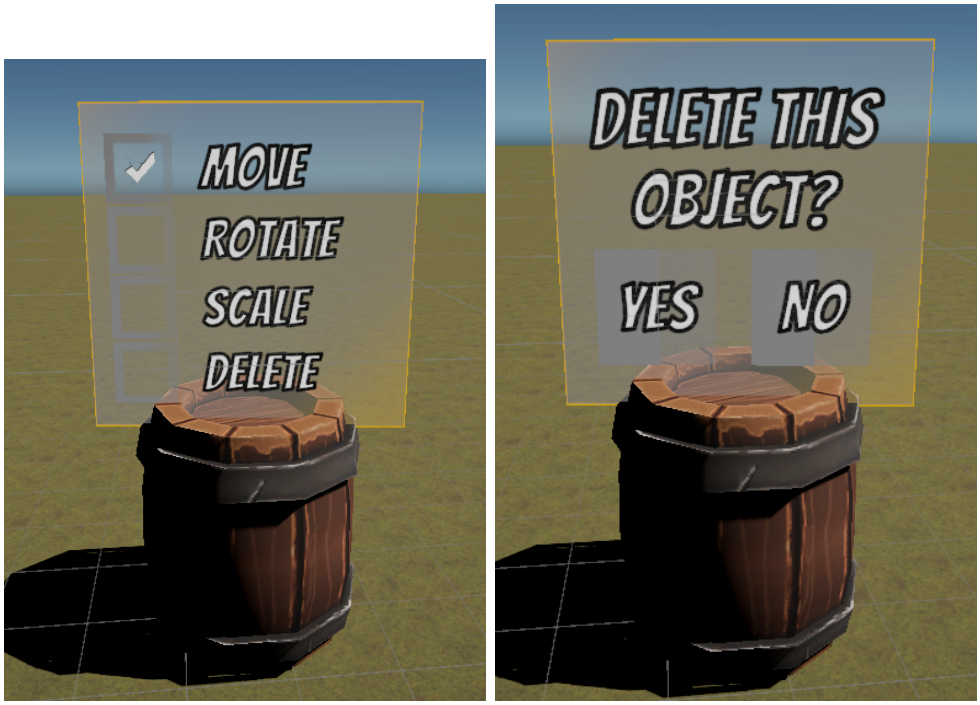

Figure 1.13: Left: Object Editor Interface, Right: Delete object option

1.3 Play Mode

After a world is created and saved the player can return to the starting scene and load the world in play mode. When a player enters play mode they are assigned one of two roles, game master or player. The first player to connect in play mode gets the game master role while any other players that connect after that get the player role.

1.3.1 Game Master

The game master should ideally have participated in the world creation process so that they know what the purpose of that specific world is and have knowledge about how to guide the player in the game and help them complete the desired challenge. This can be achieved by the game master narrating the environment and any information relevant to the task to the player and applying gameplay mechanics to the world depending on the player's actions. The means that allow the game master to act in that way are the use of the terrain tool, the item spawner and the audio mixer.

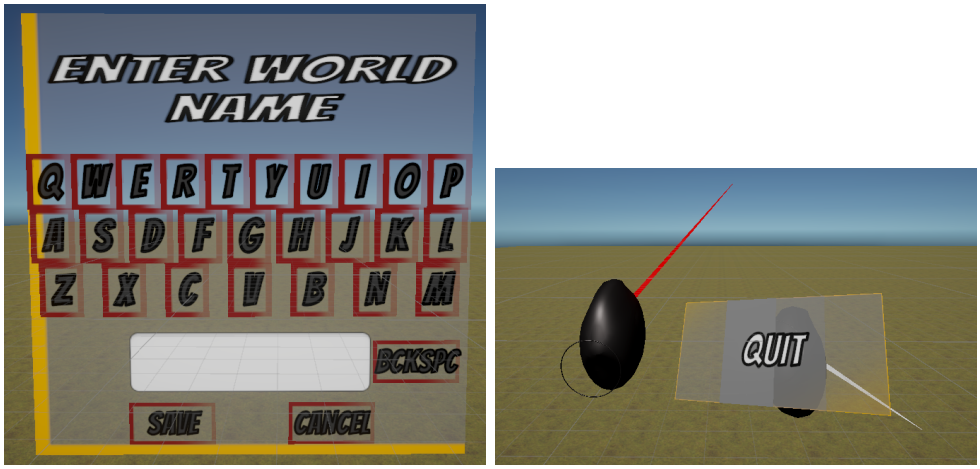

Figure 1.14: Left: Save World canvas, Right: Quit Button

Terrain Tool

The terrain tool provides the same functionality as in the world creator mode and allows the game master to manipulate the world in real time while the player is playing the game. They can change the terrain heights, textures and holes and add trees, water and objects if they decide that should be the outcome of actions of the player as the game progresses.

Item Spawner

The item spawner allows the game master to give the player crucial items for progressing the game further when certain conditions are met, for example when the player defeats a certain enemy or solves a puzzle.

Audio Mixer

The audio mixer lets the game master fine tune the voice and SFX channels of the master audio. By using the audio mixer the game master can optimize the volume of the voice chat and SFX objects independently, allowing for audio transitions and narration style changes.

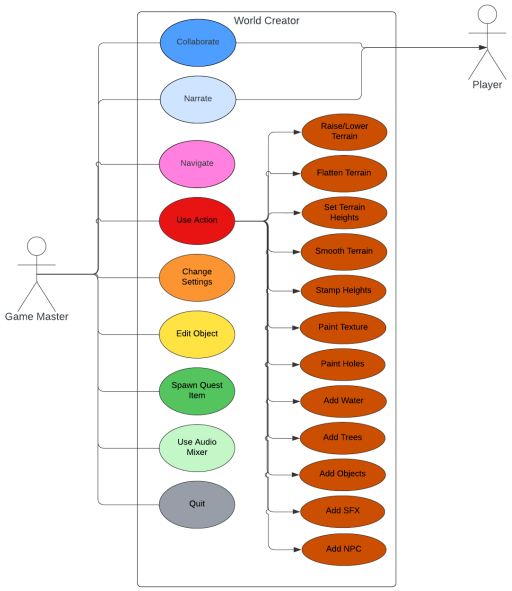

Figure 1.15: Game Master Use Case Diagram

1.3.2 Player

When the player connects in the play room the play mode tutorial will be displayed explaining to the player the ways to navigate and act in the game under the guidance of the game master.

Character

Since this is a role playing collaboration game, the player plays the role of a character who possesses magical powers. The character is of the wizard class, has a health amount and five core attributes and a character level equal to one when the game starts.

The five core attributes are:

- Constitution: represents the character's physical resilience and is equal to 3
- Strength: represents the character's physical strength and is equal to 2
- Intellect: represents the character's magical prowess and is equal to 4
- Stamina: represents the character's endurance and is equal to 2

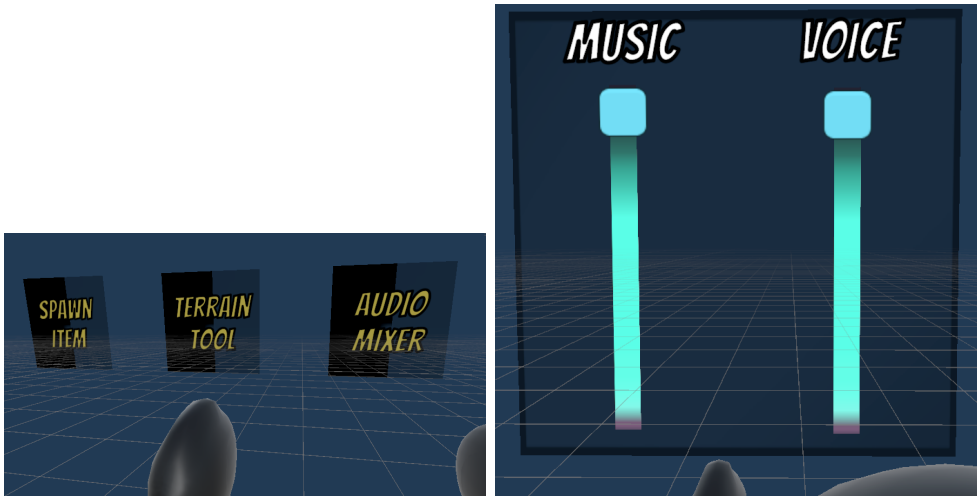

Figure 1.16: Left: Game master action selection interface, Right: Audio mixer interface

- Talent: represents the character's ability to successfully complete tasks that demand precision in detail and is equal to 3

Combat

When navigating the world, the player may encounter enemies who will try to attack. The player can engage in combat with their wizard skills, the fireball projectile attack at character level one and the call lighting AOE (Area of Effect) at character level two. The fireball can be thrown by holding the right controller trigger while moving the right controller forward. The level attribute can increase when the character reaches certain amounts of experience, which is gained from defeating enemies in combat. At character level two, by holding both grab buttons, a green circle will appear if they aim at the terrain with their head's forward direction. If an enemy is inside that area then the circle's color will turn red. Finally, if the player lowers their hands while aiming, the call lightning spell will be cast and enemies inside the area will be damaged.

When the player takes damage their health is reduced depending on the enemy's attack strength and possible visual effects are displayed on the player's vision.

There are three possible visual effects that can be applied in combat:

- Wounded effect: the player's peripheral vision is filled with a red colored animation

Figure 1.17: Player Use Case Diagram

for a short amount of time after taking damage or continuously if the player's current health is below 30% of their maximum health

- Blinded: the player's vision is blurred for a short amount of time after receiving the blinded condition from an enemy attack
- Poisoned: the player's peripheral vision is filled with a green colored animation for a short amount of time after receiving the poisoned condition from an enemy attack

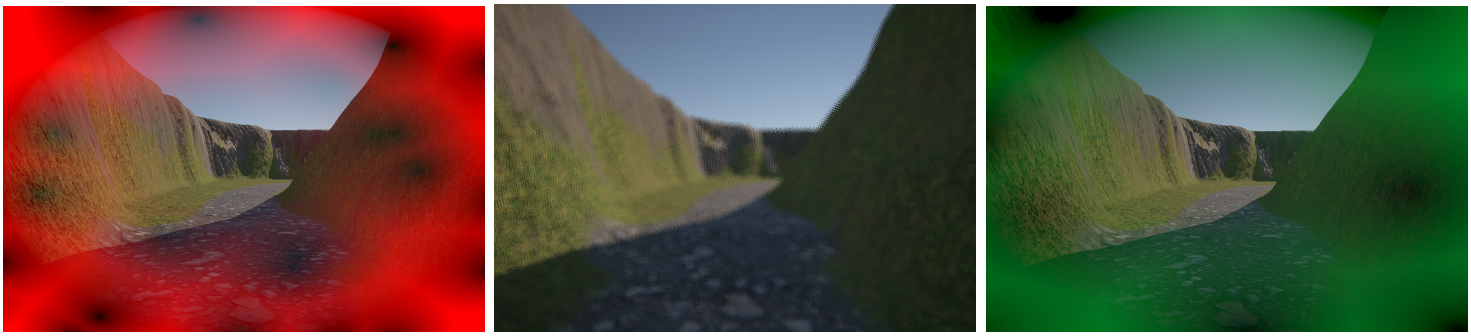

Figure 1.18: Left: Wounded effect, Middle: Blinded, Right: Poisoned

Inventory

The character also possesses an inventory which can hold items that can be used in the game for various reasons. Items can be obtained from the game master or by using the crafting bench.

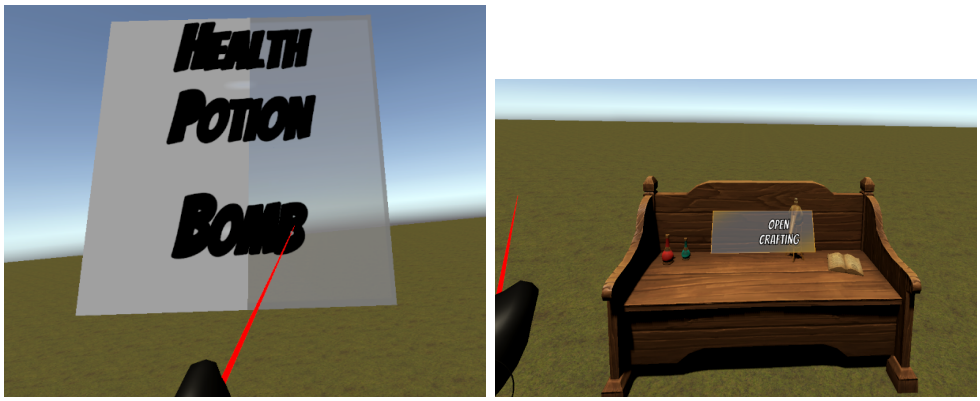

Figure 1.19: Left: Character inventory interface, Right: Crafting bench

1.4 Level Design

In order to fully demonstrate play mode in action, a level has been designed with focus on the intended collaborative aspect of the application and the utilization of the character system. Also, the level is designed with non linearity of gameplay in mind, so that the player can complete the story in different ways depending on their play style. The level is divided in four areas, each one having their own encounter and goal, while the story is inspired from traditional role-playing games.

1.4.1 Story

When the player enters the world, they find themselves in a forest south of an abandoned village. The village has become abandoned since powerful evil magic has come to the land, bringing plague and terror to its inhabitants. North of the village lie dwarven remnants of old times, said to hold valuable materials and treasures. East of the village there is the graveyard where the villagers used to pay respects to their deceased loved ones; however, at recent times the dead have awakened and roam the land, terrorizing and killing innocents in their path. The hill to the northeast has completely fallen to

darkness, beneath the black clouds, where the source of this evil rests; the powerful necromancer that raised the dead and brought plagues, lives in the old castle on the hill.

1.4.2 Purpose

The purpose of this level is for the player to navigate the world with the guidance of the game master and make their way to the northeastern part of the map where the final boss is, with the goal of defeating it. This can be achieved by following two different approaches. The first approach is for the player to first head to the graveyard and find the two undead enemies where they must defeat them. After defeating the two enemies, the player will be awarded with a key that opens the door to the castle where they must go to next. After reaching the castle, the player must enter and face the boss in combat and defeat it in order to restore peace in the land.

The alternative approach is for the player to head north and find the dwarven remnants, where they will find the crafting bench. When the crafting bench is activated, the crafting interface will appear and the player will be able to craft a health potion and a bomb. The bomb can be used at the castle doors to destroy them so that the player can then face the final boss and defeat it to complete the game.

The difference between the two approaches is that by following the first approach and killing the two undead enemies, the player will earn a character level and unlock the second skill which will in turn help them to beat the final boss more easily. In contrast, by going with the second approach the player will not have to face the first two enemies but beating the final boss will be harder since the player will have only one spell in their arsenal.

The purpose of the game master is to narrate the story to the player and help them navigate the world, giving them the necessary information so that they can choose which approach to follow. Furthermore, besides narration, the game master also has the responsibility of applying simple game mechanics such as dropping the castle door key for the player to pick up after the first two enemies are defeated.

1.4.3 Encounters

Ghoul

The first two enemies the player will encounter are very simple enemies that just chase and attack. Once the player is inside the enemy's range of detection or the enemy is damaged by the player, the enemy will begin chasing the player and try to attack if they reach close enough. The ghoul has only one melee attack which deals a small amount of damage and is easy to defeat.

Nightshade

The second enemy is more complicated and when defeated the player wins the game. When the player is close enough, the enemy will start moving, executing one of three attack patterns or a backward movement. The three possible attack patterns are based on two basic attacks, one projectile and one AOE attack. The projectile attack will apply the blind condition to the player when hit while the AOE attack will apply the poison condition. The first attack pattern first uses a backward movement and the projectile attack follows, the second pattern starts with the enemy running left for a short amount of time and if the player is visible to it the projectile attack is cast, and then the same sequence is repeated three more times with the enemy running right and left. For the third attack pattern the enemy floats upward for a short time and then casts the AOE attack and floats down again.

Chapter 2

Implementation

2.1 Introduction

For the implementation process, the first step was to install the Unity editor version 2022.3.13f1 and create a new project using the 3D URP template offered in Unity Hub. Also the Addressables package is imported which is necessary for asset management.

2.1.1 XR Setup

Then the XR Interaction Toolkit was imported from the package manager and the following settings were set up in the project settings:

- In the XR Plug-in Management tab the OpenXR plug-in provider was enabled
- In the OpenXR tab under XR Plug-in Management, render mode was set to Multipass and the Valve Index and Oculus Touch Controller profiles were added.

2.1.2 Networking

The networking of this thesis is implemented using the Normcore third party networking solution by Normal. Normcore consists of a series of layers, each one built on the layer below, providing abstraction that makes implementing networking easier for a plethora of features [**?**].

The layers that Normcore consists of are :

- Realtime: This is the API that bridges the Unity scene to Normcore's datastore.

- Room + Datastore API: The Room + Datastore API manages the connection to a room server and the synchronization of raw state in the application. It is unaware of the Unity scene or any Unity objects.
- Transport: The lowest level is our transport API. It is responsible for getting messages between point A and point B.

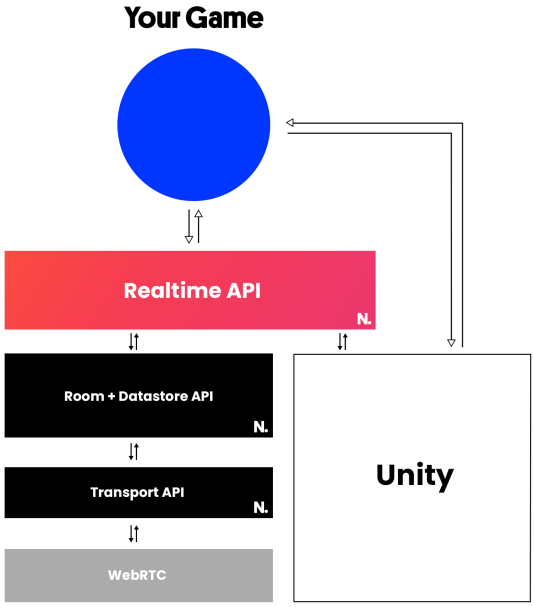

Figure 2.1: Normcore Architecture

Rooms in Normcore are used to separate groups of players. They're most commonly used to host a single match for a game or a persistent space for a productivity app. Players who join the same room name will be automatically connected to the same room. All state is synchronized using the room's datastore. If an object is moved in the world, its position changes in the datastore. The datastore will automatically detect any changes and notify all clients connected to the room so they can update their world to match.

Normcore uses an MVC (Model, View, Controller) based architecture in order to help establish a clear separation of concerns for what handles the networking code.

The datastore holds a collection of RealtimeModel objects and ensures that they're kept in sync between clients. In Unity, each GameObject represents the visual state of

your app, so we consider it to be the View. And finally, the RealtimeComponent scripts act as the controller.

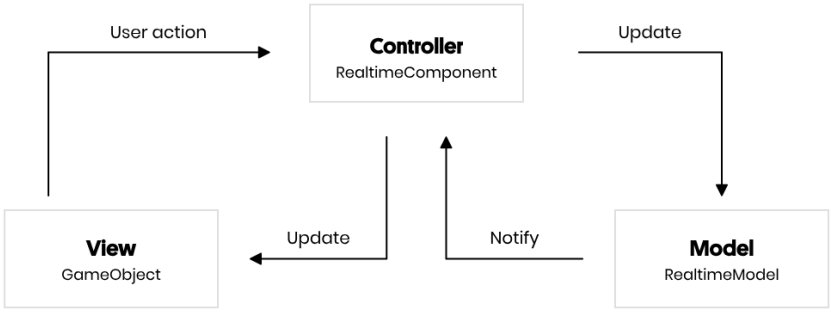

Figure 2.2: Model/View/Controller Architecture

The Realtime API is the layer that will be used for synchronizing states in the game. The Realtime API is the API that Normcore uses for all real-time synchronization in Unity. It's the layer that synchronizes all objects in the Unity scene to the Normcore datastore [**?**].

The lower-level Room + Datastore API manages the connection to the room server and the datastore. Realtime is a layer built on top of that. Realtime manages the Room + Datastore API and makes it easier to synchronize the state of the scene with the datastore.

Normcore uses realtime components to synchronize objects in a scene. It includes a few pre-built components and also includes a rich API for creating custom data synchronization scripts.

2.1.3 Pre Scene and XR Origin

When the application launches the pre scene is loaded which contains all the game objects that need to stay persistent during all scene loading and unloading, even for the starting scene. This is done in order to ensure that objects such as the XR Origin are not instantiated more than once after returning to the starting scene from the create or play scenes. All objects in this scene that need to stay persistent between scenes are called with the DontDestroyOnLoad function.

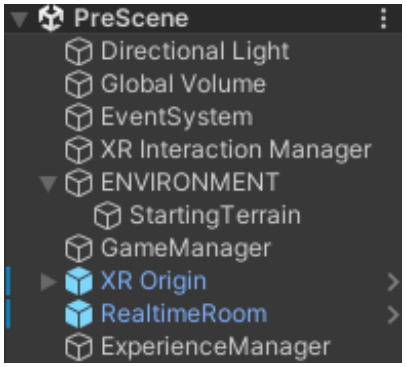

Figure 2.3: Pre Scene Hierarchy

The event system is used to listen to events for handling logic during runtime and the XR Interaction Manager manages interactions between the XR player and the virtual environment.

The Game Manager object has the Game Manager, Game Addressables Manager and Load World scripts attached to it. The GameManager script implements the singleton design pattern and its purpose is to hold variables that need to be easily accessible in any script. Also in the GameManager script, any worlds that have been created and saved in creator mode are read to be ready for loading.

The GameAddressablesManager script is responsible for loading and instantiating addressable assets that are grouped in categories and are accessed from the script by using asset references.

The GameAddressablesManager script also implements functions that return asset game objects to other scripts when specific assets are selected in UIs in the game.

The LoadWorld script implements the logic for loading created worlds when entering play mode and a detailed explanation will be presented in the corresponding section.

The RealtimeRoom game object has the Realtime script attached to it which implements the Realtime API functionality, the Realtime Avatar Manager script which manages avatars that use realtime components and the CustomRealtimePrefabLoadDelegate script which implements functions for loading addressable assets that use realtime components.

Experience Manager is a game object that has the ExperienceManager script attached to it which listens to player experience related events in play mode and invokes the related event delegates.

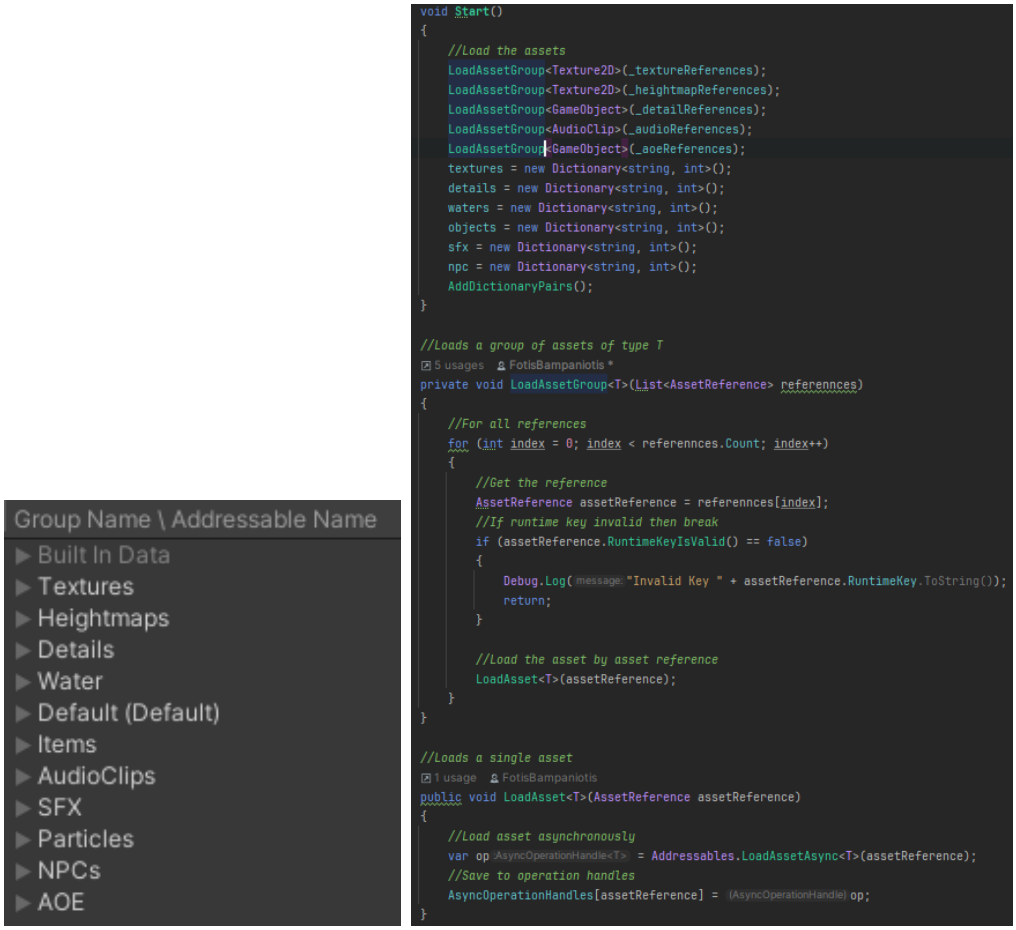

Figure 2.4: Left: Addressables Groups, Right: Game Addressables Manager Asset Loading

The XR Origin game object is essentially the XR player. It is the parent of the XR camera and left and right hand controllers and has attached scripts that manage player input, movement and rigidbody.

The XR Origin movement uses Unity's new input system which uses input actions as the mediator between the user's controller's actions and the actions that are executed in the game. The PlayerMovement script listens to changes of the left and right controller joysticks and applies movement and rotation depending on the offset from the joysticks' axis' start. The left controller joystick controls the movement of the XR Origin while the right controller joystick controls the rotation.

```
public GameObject InstantiateAsset(AssetReference assetReference, Vector3 position, Quaternion rotation, string n)
{
    GameObject obj = null;
    if (assetReference != null)
    {
        //Instantiate asynchronously
        void OnCompleted(AsyncOperationHandle<GameObject> asyncOperationHandle)
        {
            if (n == "Details")
            {
                //Save instantiated aseet to the appropriate instance list
                saveToInstances(assetReference, position, asyncOperationHandle.Result, instances: "Details");
            }
            //Set the target terrain as the parent of the instantiated asset
            asyncOperationHandle.Result.gameObject.transform.SetParent(GameManager.Instance.ActiveMode == GameManager.GameMode.Create
                ? ttmn.TargetTerrain.transform
                : GameManager.Instance.PlayTerrain.transform);
            //Save to the assetReference dictionary
            assetReferences[asyncOperationHandle.Result] = assetReference;
            var notify = asyncOperationHandle.Result.AddComponent<NotifyOnDestroy>();
            notify.Destroyed += Remove;
            notify.AssetReference = assetReference;
            obj = asyncOperationHandle.Result.gameObject;
            if (n == "AOE")
            {
                //Assign aoe object
                if (GameManager.Instance.Player.GetComponent<Wizard>())
                {
                    GameManager.Instance.Player.GetComponent<Wizard>()._aoeArea = obj;
                    obj.transform.SetParent(GameManager.Instance.Player.transform);
                    obj.GetComponent<CallLightning>().Inputs = GameManager.Instance.Player.GetComponent<Character>().Inputs;
                    obj.GetComponent<CallLightning>().Character = GameManager.Instance.Player.GetComponent<Character>();
                }
            }
        }

        assetReference.InstantiateAsync(position, rotation).Completed += OnCompleted;
    }
    return obj;
}
Frequently called  16 usages  new *
public GameObject RealtimeInstantiateAsset(AssetReference assetReference, Vector3 position, Quaternion rotation, string instances)
{
    GameObject res = Realtime.Instantiate(assetReference.RuntimeKey as string, position, rotation, Realtime.InstantiateOptions.defaults);
    //Save instantiated aseet to the appropriate instance list
    if (instances != null)
    {
        saveToInstances(assetReference, position, res, instances);
        res.GetComponent<ObjectEditor>().Refer = assetReference;
    }
    return res;
}
```

Figure 2.5: Game Addressables Manager Asset Instantiation

2.1.4 Starting Area

The starting scene was first created by adding a terrain. A starting terrain data asset was created that was used in the StartingTerrain script to initialize the terrain correctly upon program start.

The starting scene contains three door game objects, two are for entering creator and play mode and the other one is for exiting the application.

The world creator door is a parent object that has three children, two door objects and a UI canvas. The door has a box collider and a WorldCreatorDoor script responsible for detecting players in front of the door and enabling the UI canvas. After the UI is enabled, when the player presses the create button, the script loads the Create scene, unloads the

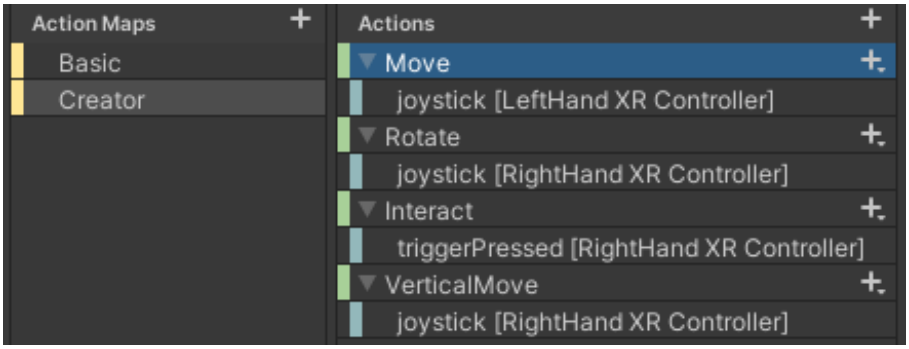

Figure 2.6: The action map used in creator mode

```
public class StartingTerrain : MonoBehaviour
{
    [SerializeField] TerrainData tdata;  Serializable
    // Start is called before the first frame update
    Event function  FotisBampaniotis
    void Start()
    {
        gameObject.GetComponent<Terrain>().terrainData = tdata;
        GetComponent<AudioSource>().Play();
    }
}
```

Figure 2.7: Starting Terrain Script

starting scene and connects to the realtime room with room name "CreatorRoom".

The load world door behaves similarly with the world creator door with the difference that the UI prompts the user to select the world to be loaded followed by a list of buttons, each with the name of a created world. When the user selects a world, the play scene is loaded, the starting scene unloaded and the realtime room "PlayRoom" is called to be connected to.

2.2 World Creator

In the creator scene, when loaded, the terrain's CreatorTerrain script will create and initialize the terrain data with the desired settings. The terrain also has an XR Simple Interactable component that allows the player to interact with it. After the Realtime component of the Realtime Room game object connects to the Create Room, the Realtime Avatar Manager will instantiate the player avatar. The player avatar is controlled by the XR Origin and it holds all the scripts that implement the terrain tool's functionality.

```
public void LoadCreateWorldScene()
{
    transform.GetComponent<Collider>().enabled = false;
    GameManager.Instance.loadingScreen.SetActive(true);
    var op AsyncOperation = SceneManager.LoadSceneAsync("CreatorScene", LoadSceneMode.Additive);
    op.completed += (operation) =>
    {
        _gm.ActiveMode = GameManager.GameMode.Create;
        SceneManager.UnloadSceneAsync("StartingScene");
        player.GetComponent<Rigidbody>().useGravity = false;
        player.GetComponent<InputManager>().enabled = true;
        player.GetComponent<PlayerMovement>().InitCreatorActionMap();
        var realtime = _gm._realtime.GetComponent<Realtime>();
        realtime.Connect( roomName: "CreatorRoom");
        realtime.didConnectToRoom += OnConnectedToCreatorRoom;
    };
}
2 usages  FotisBampaniotis *
private void OnConnectedToCreatorRoom(Realtime realtime)
{
    GameAddressablesManager.Instance.Ttmn = FindObjectOfType<TerrainToolManagerNetwork>();
    realtime.didConnectToRoom -= OnConnectedToCreatorRoom;
    _gm.Player =  GameObject.Find("VR Player2(Clone)");
    if(_gm.Player.GetComponent<RealtimeView>().isOwnedLocallyInHierarchy)
        GameManager.Instance.Player.GetComponent<TutorialManager>().creatorInfoPanel.SetActive(true);
    EnableTerrainTool();
    GameManager.Instance.loadingScreen.SetActive(false);
}
```

Figure 2.8: World Creator Door Script (Door shown in Figure 4.1)

Furthermore, when the creator scene is entered, the tutorial manager component is enabled which enables the tutorial interface. The tutorial manager script implements the functions for navigating the tutorial interface pages, enabling or disabling the show next and previous info buttons and the close tutorial function which disables the interface.

The architecture of the terrain tool has been built around the Terrain Tool Manager Network component which manages functionality of all the components related to the terrain tool.

2.2.1 Components

TerrainToolManagerNetwork

The terrain tool manager network holds references for all other terrain tool components and variables necessary for the various tools to work correctly and it also has a TerrainModificationAction enumeration field that indicates the currently selected terrain tool. In its update function it handles the display of the quit room UI and the tool's UI

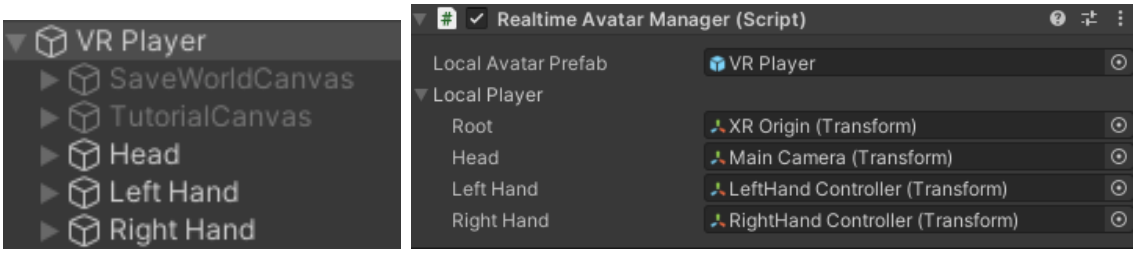

Figure 2.9: Left: VR Player Game Object Hierarchy, Right: Realtime Avatar Manager Component

in play mode and if the player avatar is owned locally by the client it allows the execution of the LocalUpdate function which handles the routing of the terrain tool's actions. In the local update the palette UI is showed if the player holds the left controller grab button. Each frame a ray is cast from the right controller's position towards the forward direction.

If the raycast hits an object with a Terrain component attached to it, then if the trigger button is pressed that same frame, the currently selected modification action is executed. Some modification actions can be executed once for each trigger press while others can be executed several times per frame when the trigger button is being held. The add trees, water, object, sfx and npc actions execute once for each trigger press to avoid duplicate instantiations and improve performance. The raise/lower terrain, flatten terrain, set terrain heights, smooth terrain heights, paint texture and paint holes actions will be executed for each 0.05 seconds that pass in each frame. This ensures that the functionality of these actions is smooth and continuous without the user having to use the trigger consecutively.

If the raycast hits an object with an ObjectEditor component attached to it and the trigger button is pressed, if the source of the trigger is a creator player or the Game Master in play mode, then the ObjectEditor UI for that object is activated if it was deactivated and vice versa and the object enters or exits edit mode accordingly. When an object enters edit mode, if the move edit action is active, the track position attribute of the XR Grab Interactable component is set to true while when the object exits edit mode, the same attribute is set to false.

The Terrain Tool Manager Network also implements functions for retrieving terrain data such as heightmaps and size which are useful for action executions and functions for setting indexing variables in the Terrain Tool Brush Network component.

Finally, the component implements crucial functions for creating model instances of realtime components that are needed for synchronizing action executions across clients.

TerrainToolBrushNetwork

The Terrain Tool Brush Network component holds information needed for all actions such as the world position of the raycast hit mentioned in the previous component, brush position on the terrain, brush size, the actual brush size that can be used when taking into consideration terrain edges and all other action variables that can be changed by the user for the various tool actions. In this component's variables, selected asset indices are also saved along with their corresponding asset references, for example a water game object and its index in the water references list that will be used by the add water tool.

The functions implemented in this component take care of initializing variables such as brush size and shape to the desired values and translating world position to terrain position and calculating action brush safe size.

```
//Calculate the start position of the brush on the terrain
Frequently called  5 usages  FotisBampaniotis *
public Vector2Int GetBrushPosition(Vector3 worldPosition, int brushWidth, int brushHeight)
{
    var resolution = getCorrectResolution();
    var terrainPosition = Ttmn.WorldToTerrainPosition(worldPosition);
    return new Vector2Int( x: (int)Mathf.Clamp( value: terrainPosition.x - brushWidth / 2.0f, min: 0.0f, max: resolution),
        y: (int)Mathf.Clamp( value: terrainPosition.z - brushHeight / 2.0f, min: 0.0f, max: resolution));
}

//Calculate the brush safe size for action execution
Frequently called  4 usages  FotisBampaniotis
public Vector2Int GetSafeBrushSize(int brushX, int brushY, int brushWidth, int brushHeight)
{
    var resolution = getCorrectResolution();
    while (resolution - (brushX + brushWidth) < 0) brushWidth--;
    while (resolution - (brushY + brushHeight) < 0 ) brushHeight--;
    return new Vector2Int( x: brushWidth, y: brushHeight);
}
```

Figure 2.10: Terrain Tool Brush Network Script Functions

TerrainToolUINetwork

The terrain tool UI network script implements all the functions needed for the UI elements like buttons, toggles and sliders for showing and hiding UI canvases and setting variable values across all of the terrain tool functions. It also implements the functions that set the various assets for the terrain tools in the terrain tool brush manager.

TerrainToolAddressablesManagerNetwork

The Terrain Tool Addressables Manager Network script implements the functions that set the indices of the selected asset references in the terrain tool brush manager network.

TerrainToolNetwork and HeightmapSync

The terrain tool network and heightmap sync scripts implement the functionality of actions related to the terrain heightmap. First of all, the heightmap sync script is derived from the RealtimeModel class HeightmapSyncModel which holds the RealtimeArray RealtimeProperty of type HeightmapActionModel. The HeightmapActionModel is a RealtimeModel class containing all the necessary (Realtime)Properties for synchronizing heightmap actions across all clients.

```
[RealtimeModel]
12 usages  FotisBampaniotis
public partial class HeightmapActionModel
{
    [RealtimeProperty(1, reliable: true)]
    private string _modificationAction;
    [RealtimeProperty(2, reliable: true)]
    private float _xpos;
    [RealtimeProperty(3, reliable: true)]
    private float _ypos;
    [RealtimeProperty(4, reliable: true)]
    private int _brushSize;
    [RealtimeProperty(5, reliable: true)]
    private float _height;
    [RealtimeProperty(6, reliable: true)]
    private float _strength;
    [RealtimeProperty(7, reliable: true)]
    private int _stampIndex;
    [RealtimeProperty(8, reliable: true)]
    private bool _raise;
    [RealtimeProperty(9, reliable: true)]
    private bool _circular;
}
```

```
[RealtimeModel]
3 usages  FotisBampaniotis
public partial class HeightmapSyncModel
{
    [RealtimeProperty(1, reliable: true)]
    private RealtimeArray<HeightmapActionModel> _actions;
}
```

Figure 2.11: Left: Heightmap Action Model, Right: Heightmap Sync Model

When a player executes a heightmap related action the terrain tool manager network will create a heightmap action model with the desired property values and add it to the HeightmapSyncModel actions RealtimeArray.

When an action model is added to the RealtimeArray all clients are notified and the HeightmapActionAdded function in the HeightmapSync script is called. A new terrain

```
private TerrainToolBrushNetwork CreateActionBrush(HeightmapActionModel action)
{
    var networkedBrush = gameObject.AddComponent<TerrainToolBrushNetwork>();
    networkedBrush.worldPos.x = action.xpos;
    networkedBrush.worldPos.y = action.ypos;
    networkedBrush.brushSize.x = action.brushSize;
    networkedBrush.brushSize.y = action.brushSize;
    networkedBrush.scaledSize = action.brushSize * Ttmn.Brush.getCorrectResolution() / 100;
    networkedBrush.circularBrush = action.circular;
    return networkedBrush;
}
```

Figure 2.12: Create Action Brush

```
switch (modificationAction)
{
    case TerrainModificationAction.Raise_Lower:
        hsync.RegisterHeightmapAction(CreateHeightmapActionModel(point));
        break;

    case TerrainModificationAction.Flatten:
        hsync.RegisterHeightmapAction(CreateHeightmapActionModel(point));
        break;

    case TerrainModificationAction.SetHeights:
        hsync.RegisterHeightmapAction(CreateHeightmapActionModel(point));
        break;

    case TerrainModificationAction.SmoothTerrain:
        hsync.RegisterHeightmapAction(CreateHeightmapActionModel(point));
        break;
```

```
private HeightmapActionModel CreateHeightmapActionModel(Vector3 worldPosition)
{
    HeightmapActionModel heightmapActionModel = new HeightmapActionModel();
    heightmapActionModel.modificationAction = stringifyAction(modificationAction);
    heightmapActionModel.xpos = worldPosition.x;
    heightmapActionModel.ypos = worldPosition.z;
    heightmapActionModel.brushSize = Brush.brushSize.x;
    if (modificationAction == TerrainToolManagerNetwork.TerrainModificationAction.SetHeights)
        heightmapActionModel.height = Brush.fixedHeight;
    if (modificationAction == TerrainToolManagerNetwork.TerrainModificationAction.SetHeights
        || modificationAction == TerrainToolManagerNetwork.TerrainModificationAction.Raise_Lower
        || modificationAction == TerrainToolManagerNetwork.TerrainModificationAction.StampHeigths)
        heightmapActionModel.strength = Brush.strength;
    if (modificationAction == TerrainToolManagerNetwork.TerrainModificationAction.StampHeigths)
        heightmapActionModel.stampIndex = Brush.HeightmapAssetIndex;
    if (modificationAction == TerrainToolManagerNetwork.TerrainModificationAction.Raise_Lower)
        heightmapActionModel.raise = Brush.isRaise;
    heightmapActionModel.circular = Brush.circularBrush;

    return heightmapActionModel;
}
```

Figure 2.13: Left: Register Heightmap Action, Right: Create Heightmap Action Model

tool brush network component is created and all the variable values from the action model are passed to the corresponding ones in the new tool brush component. Also a new terrain tool network component is added to the game object, both new creations taking place so that the existing component's variables are not changed.

After the new brush is created the appropriate action function in the new terrain tool network component is called. The terrain heightmap is actually a two dimensional float array with dimensions equal to the terrain heightmap resolution of the terrain data. Each cell's float value represents the actual height of the terrain at that point.

In the terrain tool network script the RaiseLowerTerrain function locates the heightmap area that should be affected considering the brush position and size and all float values in that area in the array are increased or decreased by the amount of the brush strength, depending on the raise/lower boolean variable value. The flatten terrain function sets

all heights in the brush area to zero, the set terrain heights function sets the heights according to the brush height variable value. The smooth heights function applies a Gaussian filter with a kernel size of 5 inside the brush area and the stamp heights reads the height values of a texture heightmap the user selected and add them to the brush area multiplied by brush strength. After an action is executed, the created terrain brush and terrain tool network components are destroyed.

TerrainToolMeshNetwork and SplatmapSync

The terrain tool mesh network and splatmap sync scripts implement the functionality of actions related to the terrain splatmap(alphamap texture) and hole map. Similar to the previous set of scripts, the splatmap sync script is derived from the RealtimeModel class SplatmapSyncModel which holds the RealtimeArray RealtimeProperty of type SplatmapActionModel. The SplatmapActionModel is a RealtimeModel class containing all the necessary (Realtime)Properties for synchronizing splatmap and hole map actions across all clients.

```
[RealtimeModel]
12 usages  FotisBampaniotis
public partial class SplatmapActionModel
{
    [RealtimeProperty(1, reliable: true)]
    private string _modificationAction;
    [RealtimeProperty(2, reliable: true)]
    private float _xpos;
    [RealtimeProperty(3, reliable: true)]
    private float _ypos;
    [RealtimeProperty(4, reliable: true)]
    private int _brushSize;
    [RealtimeProperty(5, reliable: true)]
    private int _textureIndex;
    [RealtimeProperty(6, reliable: true)]
    private bool _surface;
    [RealtimeProperty(7, reliable: true)]
    private bool _circular;
}
```

```
[RealtimeModel]
3 usages  FotisBampaniotis
public partial class SplatmapSyncModel
{
    [RealtimeProperty(1, reliable: true)]
    private RealtimeArray<SplatmapActionModel> _actions;
}
```

Figure 2.14: Left: Splatmap Action Model, Right: Splatmap Sync Model

When a player executes a splatmap or hole map related action the terrain tool manager network will create a splatmap action model with the desired property values and add it to the SplatmapSyncModel actions RealtimeArray.

When an action model is added to the RealtimeArray all clients are notified and the SplatmapActionAdded function in the SplatmapSync script is called. Again, new terrain tool brush network and terrain tool mesh network components are created and initialized.

```
case TerrainModificationAction.PaintHoles:
    ssync.registerSplatmapAction(CreateSplatmapActionModel(point));
    break;

case TerrainModificationAction.PaintTexture:
    ssync.registerSplatmapAction(CreateSplatmapActionModel(point));
    break;
```

```
private SplatmapActionModel CreateSplatmapActionModel(Vector3 worldPosition)
{
    SplatmapActionModel splatmapActionModel = new SplatmapActionModel();
    splatmapActionModel.modificationAction = stringifyAction(modificationAction);
    splatmapActionModel.xpos = worldPosition.x;
    splatmapActionModel.ypos = worldPosition.z;
    splatmapActionModel.brushSize = Brush.brushSize.x;
    if (modificationAction == TerrainToolManagerNetwork.TerrainModificationAction.PaintTexture)
        splatmapActionModel.textureIndex = Brush.TextureAssetIndex;
    if (modificationAction == TerrainToolManagerNetwork.TerrainModificationAction.PaintHoles)
        splatmapActionModel.surface = Brush.isSurface;

    splatmapActionModel.circular = Brush.circularBrush;
    return splatmapActionModel;
}
```

Figure 2.15: Left: Register Splatmap Action, Right: Create Splatmap Action Model

```
private void SplatmapActionAdded(RealtimeArray<SplatmapActionModel> actions, SplatmapActionModel action, bool remote)
{
    if (!action.isOwnedLocallyInHierarchy)
    {
        TerrainToolBrushNetwork actionBrush = CreateActionBrush(action);
        TerrainToolMeshNetwork terrainMeshTool = gameObject.AddComponent<TerrainToolMeshNetwork>();
        terrainMeshTool.Ttmn = Ttmn;
        actionBrush.Ttmn = Ttmn;
        if (action.modificationAction == "PaintTexture")
        {
            actionBrush.TextureAssetIndex = action.textureIndex;
            actionBrush.texture = actionBrush.Ttmn.SelectTexture(actionBrush.TextureAssetIndex, referenceType: "Texture");
            terrainMeshTool.getPixelsFromTexture(actionBrush);
            terrainMeshTool.paintTexture(actionBrush);
        }
        else if (action.modificationAction == "PaintHoles")
        {
            terrainMeshTool.Ttmn.modificationAction = TerrainToolManagerNetwork.TerrainModificationAction.PaintHoles;
            actionBrush.isSurface = action.surface;
            StartCoroutine(routine:terrainMeshTool.paintHoles(actionBrush));
        }
    }
}
```

Figure 2.16: Splatmap Action Added function

The terrain splatmap is a three dimensional float array with the first two dimensions being the width and height of the array and the third dimension being the number of terrain texture layers on the terrain. The hole map is two dimensional boolean array with each cell representing if the corresponding point of the terrain is terrain or hole.

In the terrain tool mesh network script the PaintTexture function sets the mixing weight of each splatmap at each x,y coordinate to the A component value of the selected texture pixel(a pixel has the RGBA(red, green, blue and alpha) properties with the a component representing the transparency). The PaintHoles function sets the hole map value at each point of the brush area to true or false depending on the terrain tool brush's isSurface variable value.

TerrainDetailToolNetwork, DetailSync and TerrainToolFirebase

The terrain detail tool network and detail sync scripts implement the functionality of actions related to trees and details. The detail sync script is derived from the RealtimeModel class DetailSyncModel which holds the RealtimeArray RealtimeProperty of type DetailActionModel. The DetailActionModel is a RealtimeModel class containing all the necessary (Realtime)Properties for synchronizing detail related actions across all clients.

```
[RealtimeModel]
12 usages  FotisBampaniotis
public partial class DetailActionModel
{
    [RealtimeProperty(1, reliable: true)]
    private int _treeIndex;
    [RealtimeProperty(2, reliable: true)]
    private bool _remove;
}
```

```
[RealtimeModel]
3 usages  FotisBampaniotis
public partial class DetailSyncModel
{
    [RealtimeProperty(1, reliable: true)]
    private RealtimeArray<DetailActionModel> _actions;
}
```

Figure 2.17: Left: Detail Action Model, Right: Detail Sync Model

When a player executes a detail action, a paint or remove details function from the terrain detail tool network script is executed to calculate the amount of details to instantiate in the appropriate area. For the AddDetail and MassPaintDetails functions, TreeInfo objects are created and added in a list. For the RemoveDetail and MassRemoveDetails, a TreeBounds object is created for storing the upper and lower bounds of terrain coordinates to remove details from. After the TreeInfo list or the TreeBounds object is completed, it is sent for serialization in the Terrain Tool Firebase script.

For the Terrain Tool Firebase script to work, a Google Firebase project has been set up so that the Firebase storage can be used. For the serialization process the NewtonSoft Json package in Unity has been used which offers out of the box serialization and deserialization and many other useful features. In the Serialize and Upload TreeInstances and Serialize and Upload TreeBounds functions, after serialization is completed, the serialized objects are save to the user's local files and uploaded to the Firebase storage.

Back to the Add/Remove Details functions, after the serialization and upload of the corresponding files, a detail action model is created and added to the DetailSyncModel's

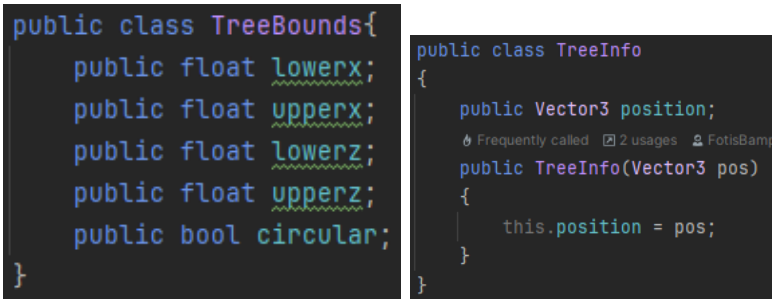

Figure 2.18: Left: Tree Bounds class, Right: Tree Info class

actions RealtimeArray Realtime Property and all other clients are notified.

Again, new terrain brush and terrain detail tool objects are created. In the Remove Details Networked function, all trees inside the TreeBounds are removed and in the Paint Details Networked function a tree is instantiated for each TreeInfo object in the TreeInfos list. After an action is completed the created terrain brush and terrain detail tool objects are destroyed.

TerrainWaterToolNetwork

The terrain water tool network implements the function called from the terrain tool manager network that instantiates water objects. For the water objects shader the simple water shader URP asset has been used from the Unity asset store [**?**].

TerrainObjectToolNetwork

The terrain object tool network implements the function called from the terrain tool manager network that instantiates object objects.

TerrainSfxToolNetwork and SfxRendererSync

The terrain sfx tool network implements the function called from the terrain tool manager network that instantiates sfx objects. The SfxRendererSync is derived from the RealtimeModel class SfxRendererModel which holds the boolean RealtimeProperty rendererActive. When a user changes the terrain tool sfx renderers active setting all clients will be notified and enable or disable all sfx objects' MeshRenderer component to match

```
public void SerializeandUploadTreeInstances(List<TreeInfo> treeInstances)
{
    string json = JsonConvert.SerializeObject(treeInstances);

    // Save the JSON data to a file
    string path = Application.dataPath + "/detailActionInstances" + (dSync.GetModelCount() + 1) + ".json";
    File.WriteAllText(path, contents: json);
    Debug.Log( message: "treeinstances converted to JSON and saved to " + path);
    SaveToStorage( fileReference: "/detailActionInstances" + (dSync.GetModelCount() + 1) + ".json"
        , filePath: Application.dataPath + "/detailActionInstances" + (dSync.GetModelCount() + 1) + ".json");
}

Frequently called  2 usages  FotisBampaniotis *
public void SerializeandUploadTreeBounds(TreeBounds treeBounds)
{
    string json = JsonConvert.SerializeObject(treeBounds);

    // Save the JSON data to a file
    string path = Application.dataPath + "/detailActionBounds" + (dSync.GetModelCount() + 1) + ".json";
    File.WriteAllText(path, contents: json);
    Debug.Log( message: "treeBOUNDS converted to JSON and saved to " + path);
    SaveToStorage( fileReference: "/detailActionBounds" + (dSync.GetModelCount() + 1) + ".json"
        , filePath: Application.dataPath + "/detailActionBounds" + (dSync.GetModelCount() + 1) + ".json");
}
```

Figure 2.19: Serialization and Upload to Firebase storage

the SfxRendererModel's rendererActive boolean value.

TerrainNpcToolNetwork and EnemyAgentSync

The terrain npc tool network implements the function called from the terrain tool manager network that instantiates npc objects. The EnemyAgentSync is derived from the RealtimeModel class EnemyAgentModel which holds the boolean RealtimeProperty agentActive. When a user changes the terrain tool enemy agent active setting all clients will be notified and enable or disable all npc objects' NavMeshAgent component to match the EnemyAgentModel's agentActive boolean value.

SaveWorld

The SaveWorld script implements the function that saves a created world in the user's local files. First, a check is made to see if a folder named "Worlds" exists in the user's application folder and if it doesn't, one is created. Then a folder named as the selected

```
private IEnumerator ExecuteDetailAction(DetailActionModel action)
{
    TerrainToolBrushNetwork actionBrush = gameObject.AddComponent<TerrainToolBrushNetwork>();
    TerrainDetailToolNetwork actionTool = gameObject.AddComponent<TerrainDetailToolNetwork>();
    actionBrush.Ttmn = ttmn;
    actionTool.Ttmn = ttmn;
    actionTool.Ttamn = ttmn.Ttamn;

    if (action.remove)
    {
        ttf.Read = false;
        actionTool.TreeBounds = new TreeBounds();
        ttf.StartCoroutine(routine: ttf.DownloadandDeserializeTreeBounds(model.actions.Count, actionTool));
        yield return new WaitUntil(() => ttf.Read == true);
        actionTool.RemoveDetailsNetworked(actionTool.TreeBounds);
    }
    else
    {
        actionTool.TreeInfos  = new List<TreeInfo>();
        ttf.Read = false;
        ttf.StartCoroutine(routine: ttf.DownloadandDeserializeTreeInstances(model.actions.Count, actionTool));
        yield return new WaitUntil(() => ttf.Read == true);
        actionBrush.TreeAssetIndex = action.treeIndex;
        actionBrush.treePrototype = new TreePrototype();
        actionBrush.treePrototype.prefab = GameAddressablesManager.Instance.addressableObjectSelected(action.treeIndex
            , actionBrush.Ttmn.Ttamn.DetailReferences); // GameObject
        actionTool.PaintDetailsNetworked(actionTool.TreeInfos, action.treeIndex, actionBrush);
    }
    Destroy(actionBrush);
    Destroy(actionTool);
}
```

Figure 2.20: Execute Detail Action on notified clients

world name is created inside the worlds folder.

For the actual saving, the terrain heightmap, alphamap, terrain layers and hole map as well as all instantiated details and water, object, sfx, and npc objects are serialized in Json files using the JsonConvert.SerializeObject() function. Finally, the serialized files are saved inside the folder with the selected world name.

Quit Room

The quit room components disconnects the player from the current realtime room and loads the starting scene. When the starting scene is loaded the creator scene is unloaded.

```
public void AddWater(Vector3 worldPosition)
{
    Spawn(worldPosition);
}
Frequently called  1 usage  FotisBampaniotis *
private void Spawn(Vector3 pos)
{
    if (ttmn.Brush.WaterAssetIndex >= 0)
    {
        AssetReference currRef = ttamn.Realtime.GetComponent<CustomRealtimePrefabLoadDelegate>().WaterReferences[ttmn.Brush.WaterAssetIndex];
        GameAddressablesManager.Instance.RealtimeInstantiateAsset(currRef, pos, rotation: Quaternion.Euler( x: 0,  y: 0,  z: 0), instances: "Water");
    }
```

Figure 2.21: Spawn Water function (for objects shown in Figure 4.8)

```
case TerrainModificationAction.AddWater:
    Twtn.AddWater(point);
    break;
case TerrainModificationAction.AddObject:
    Totn.addObject(point);
    break;
case TerrainModificationAction.AddSfx:
    Tstn.addSfx(point);
    break;
case TerrainModificationAction.AddNpc:
    Tntn.AddNpc(point);
    break;
```

Figure 2.22: Add function calls for realtime assets

2.2.2 Object Editor

In creator mode, instantiated objects with an XR Grab Interactable component attached to them can be edited by activating the object editor. When an object enters edit mode, if the move edit action is selected, the XR Grab Interactable component's trackPosition attribute is set to false and the trackPosition to true. Also its collider's isTrigger property is set to false so that it can be grabbed and moved.

When the edit mode changes to rotate, the interactable's trackRotation is set to true and if the grab button of the right controller is being hold, the object's rotation will align with the rotation of the right controller.

When the scale edit mode is activated both trackPosition and trackRotation of the object's interactable component are set to false. The scale mode utilizes a scaling boolean property which is initially false. When initiating scaling by holding both controllers' grab buttons, the scaling boolean property will be set to true and the distance between the two controllers will be saved in the startingScalingDistance property. While scaling, meaning for each frame that both grab buttons are held down, the scale factor of the object is

```
public void AddObject(Vector3 worldPosition)
{
    Spawn(worldPosition);
}
Frequently called  1 usage  FotisBampaniotis *
private void Spawn(Vector3 pos)
{
    if (ttmn.Brush.ObjectAssetIndex >= 0)
    {
        AssetReference currRef = ttamn.Realtime.GetComponent<CustomRealtimePrefabLoadDelegate>().ObjectReferences[ttmn.Brush.ObjectAssetIndex];
        GameAddressablesManager.Instance.RealtimeInstantiateAsset(currRef, pos, rotation: Quaternion.Euler( x: 0,  y: 0,  z: 0), instances: "Object");
    }
}
```

Figure 2.23: Spawn Object function (for objects shown in Figure 4.10)

```
partial class Layer
{
    public string layerName;
    public Vector2 tileSize;
    public Vector2 tileOffset;
}
```

```
partial class DetailObject
{
    public string name;
    public Vector3 position;
}
```

```
partial class RealtimeObject
{
    public string name;
    public Vector3 position;
    public Quaternion rotation;
    public Vector3 scale;
}
```

Figure 2.24: Helper classes for serializing objects, Left: Layer Partial Class, Middle: Detail Partial Class, Right: Object Partial Class

calculated by subtracting the starting scaling distance from the current distance between the two controllers. Then, the scale factor is added to the local scale of the object after ensuring that the sum is greater than zero.

For the scale action, the scale balancer component is implemented, which manages the scale of the edit interface so that it is not scaled while scaling the object.

Each time a move, rotate or scale action is initiated, the Object Editor component's occupy function is called. If the state of the object is free and not occupied, ownership of the realtime components of the object is requested and the occupied state is applied. While in occupied state the ownership of the realtime components can not be requested. After a move, rotate or scale action is completed, the object's state is set to free and realtime component ownerships are cleared.

The final edit action available is delete object. When the delete action is selected, a prompt will be given to the player asking if they are sure to delete the object. If they choose yes the object is deleted otherwise the move, rotate, scale and delete options are displayed again.

```
void Update()
{
    transform.localRotation = Quaternion.Inverse(transform.parent.localRotation) * lastParentRotation * transform.localRotation;
    lastParentRotation = transform.parent.localRotation;
    // Calculate the opposite direction
    Vector3 oppositeDirection = -GameManager.Instance.origin.transform.position + transform.position;
    transform.LookAt( worldPosition: transform.position + oppositeDirection);

    if (parent.CompareTag("Enemy"))
    {
        b = parent.transform.GetChild(4).GetComponent<SkinnedMeshRenderer>().bounds;
    }
    else
    {
        b = parent.GetComponent<MeshRenderer>().bounds;
    }

    transform.localScale = new Vector3( x: FixeScale / parent.transform.localScale.x,
        y: FixeScale / parent.transform.localScale.y, z: FixeScale / parent.transform.localScale.z);

    if (parent.transform.childCount>=2 && parent.transform.GetChild(1).name == "Attach")
    {
            var p: Vector3 = parent.transform.GetChild(1).transform.position;
            transform.position = p + new Vector3( x: 0, y: b.max.y - b.min.y, z: 0);
    }
    else
    {
        transform.position = Vector3.up + b.max + new Vector3( x: 0, y: 1, z: 0);
    }
}
```

Figure 2.25: Scale balancer (for the interface shown in Figure 4.13)

2.2.3 User Interface

Palette Action Selectors

The terrain tool interface consists of objects for selecting the desired terrain tool modification action. Each action selector object has an XR Simple Interactable component along with a script responsible for changing the active modification action, its visibility and enabling possible asset selectors.

Palette Asset Selectors

When an action that can select from a range of assets to use, such as the paint texture action, has been selected the corresponding asset selectors will be displayed on the user interface. Each asset selector object has an XR Simple Interactable component and an asset selector script. In the interactable component's activate event, the select asset function is set as a listener with the corresponding asset index in that asset's asset references dictionary.

```
public class PaintTextureSelector : MonoBehaviour
{
    [SerializeField] private GameObject palette;  Changed in 3 assets
    [SerializeField] private GameObject textureAssetSelectors;  Changed in 3 assets
    1 asset usage  FotisBampaniotis
    public void OnHover()
    {
        transform.localScale = transform.localScale + new Vector3( x: 1f,  y: 1f,  z: 1f);
    }
    1 asset usage  FotisBampaniotis
    public void OnHoverExit()
    {
        transform.localScale = transform.localScale - new Vector3( x: 1f,  y: 1f,  z: 1f);
    }
    1 asset usage  FotisBampaniotis *
    public void OnActivation()
    {
        transform.localScale += new Vector3( x: 1f,  y: 1f,  z: 1f);
        palette.GetComponent<Palette>().Origin.GetComponent<TerrainToolManagerNetwork>().modificationAction
            = TerrainToolManagerNetwork.TerrainModificationAction.PaintTexture;
        textureAssetSelectors.SetActive(true);
        palette.transform.GetChild(6).GetChild(0).GetChild(5).GetComponent<TextMeshProUGUI>().color = Color.green ;
    }
    1 asset usage  FotisBampaniotis
    public void OnDeactivation()
    {
        transform.localScale = transform.localScale - new Vector3( x: 1f,  y: 1f,  z: 1f);
    }
    Event function  new *
    private void Update()
    {
        if (transform.localScale.x < .7f || transform.localScale.y < .7f || transform.localScale.z < .7f)
            transform.localScale = new Vector3( x: 0.9003983f,  y: 0.7979493f,  z: 0.9003977f);
    }
}
```

Figure 2.26: Paint Texture Selector (for selecting the action shown in Figure 4.6)

2.3 Character and Combat Systems

The character and combat systems implemented are interdependent and provide basic role playing functionality to the game. The character system consists of three main components, the character, the experience manager and the inventory. The combat system consists of the wizard class, five spell components and the enemy npc behaviour components.

2.3.1 Character

The character component has all the properties needed for the implemented gameplay mechanics to function, like character class, max and current health, experience, level, stats, active conditions, etc. It also implements the IDamageReceiver and IConditionRe-

```
public class TextureAssetSelector : MonoBehaviour
{
    public GameObject origin;  Changed in 2 assets
    private TerrainToolManagerNetwork ttmn;
    Event function  FotisBampaniotis
    void Start()
    {
        ttmn = origin.GetComponent<TerrainToolManagerNetwork>();
    }
    4 asset usages  FotisBampaniotis *
    public void SelectTextureAsset(int assetIndex)
    {
        ttmn.Brush.texture = ttmn.SelectTexture(assetIndex, referenceType: "Texture");
        ttmn.Ttamn.setTextureIndex(assetIndex);
        ttmn.Tmtn.GetPixelsFromTexture();
        SelectInfoText();
    }
    1 usage  FotisBampaniotis
    private void SelectInfoText()
    {
        for(int i = 0; i < transform.parent.childCount; i++)
        {
            transform.parent.GetChild(i).GetChild(0).GetChild(0).GetComponent<TextMeshProUGUI>().color = Color.white;
        }
        transform.GetChild(0).GetChild(0).GetComponent<TextMeshProUGUI>().color = Color.green;
    }
}
```

Figure 2.27: Texture Asset Selector (for selecting the texture objects shown in Figure 4.6)

ceiver interfaces.

Receive Damage

The receive damage function implements the ReceiveDamage funtion of the IDamageReceiver interface and takes a damage integer and a source transform as its parameters. This function decreases the character's current health by the amount of damage if the difference is greater than zero, otherwise it sets it to zero. Additionally, the wound condition is activated for a second if the component is owned by the local realtime client.

Receive Condition

The ReceiveCondition function takes a condition type and a float type parameter and starts the ActivateCondition co-routine if the component is owned by the local realtime client.

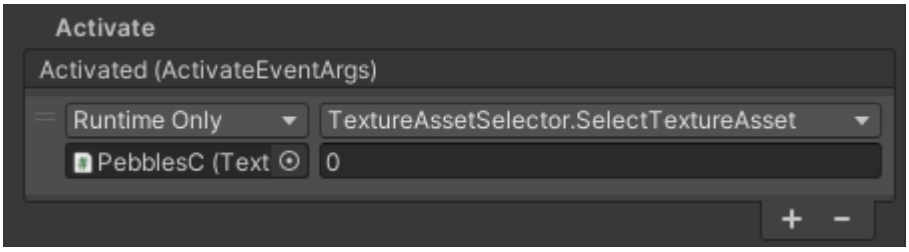

Figure 2.28: Texture Asset Activate Event

Activate Condition

The ActivateCondition function takes a condition type and a float type parameter and activates a condition for a specified duration depending on the input parameters. The implementation of the conditions as full screen effects has been achieved by adding renderer features to the universal renderer data of the universal render pipeline.

When a condition is activated the corresponding renderer feature is activated for the specified duration.For the wounded and poisoned conditions, the FadeFx co-routine is started after the effect's duration ends. The wound and poison full screen conditions are implemented with the use of shader graphs. For the implementation of the full screen blind condition the Fast Mobile Post Processing unity asset from the asset store has been used which offers various effects.

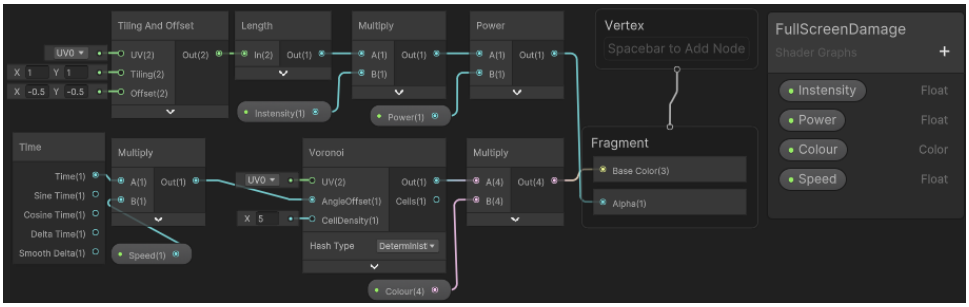

Figure 2.29: Full Screen Damage Condition Shader Graph (shown in Figure 4.18)

Fade Fx

In the Fade Fx co-routine, the effect's intensity variable is decreased gradually for

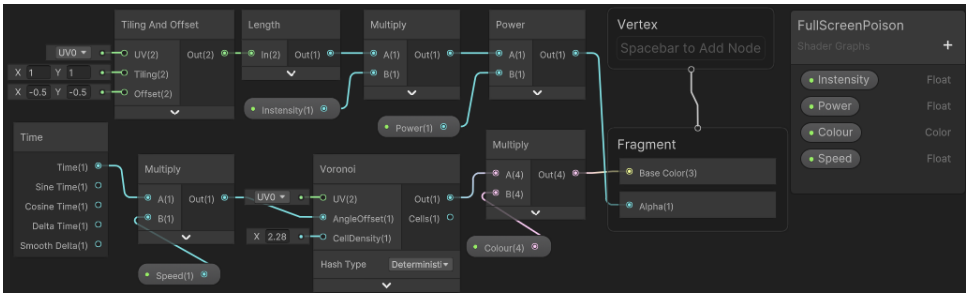

Figure 2.30: Full Screen Poison Condition Shader Graph (shown in Figure 4.18)

half a second in that effect's shader graph before it is disabled.

Get Experience

When the character component is enabled the Get Experience function is subscribed as the experience manager's GainExp event. This function is invoked when the GainExp event is fired in the Add Experience function of the experience manager component. The Get Experience function has a gainedExperience integer parameter which is added to the character's experience.

Level Up

If the experience gained by the character is equal or greater than the character's current level experience cap, the character's level is increased and the experience cap for the new level is doubled by the previous experience cap. Furthermore, the character's inventory capacity and max health increase.

Inventory

The character inventory is implemented by a list of strings in the Inventory component. The component is responsible for initializing the capacity and adding a few items at start and in the update function the inventory interface canvas is enabled if the left controller's primary button is pressed. The inventory interface includes buttons for each item present in the inventory list.

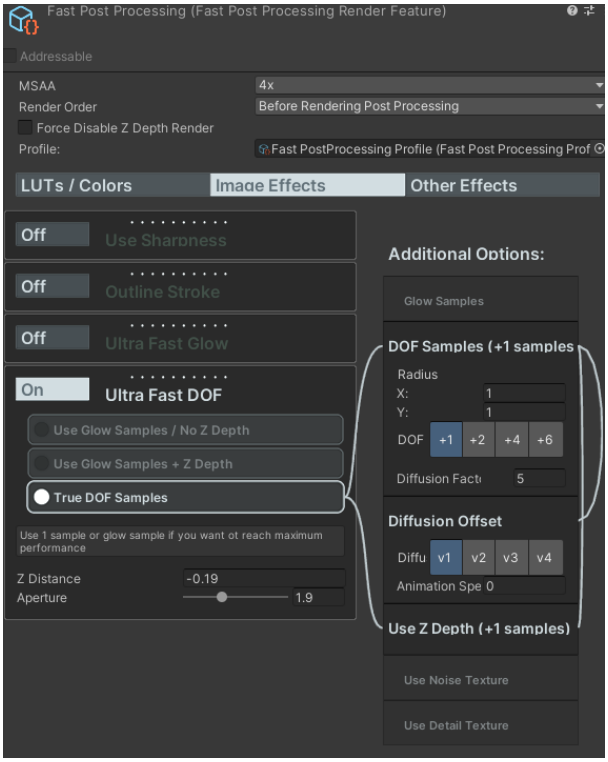

Figure 2.31: Full Screen Blind Asset Settings (shown in Figure 4.18)

Furthermore, the component implements the functions for using the health potion and bomb items. The health potion item simply increases the character's current health by 10 while when the bomb item button is pressed, the waiting for position boolean property is set to true, a message to the player is enabled asking them to select the instantiation position of the item and if the right controller trigger is pressed, the item is instantiated.

2.3.2 Combat Components

Wizard Spells

The wizard class implements the logic for casting the two available player spells. In the start of the component's update function a check is made to ensure that the component is owned by the local client so that remote clients' actions will be blocked.

The Fireball function is implemented and called in the update method. In the fireball function if the right trigger button is being held and the fireball is not on cooldown,

```
public void AddExperience(int amount)
{
    GainExp?.Invoke(amount);
}
```

```
private void GetExperience(int gainedExperience)
{
    _experience += gainedExperience;
    if (_experience >= _experienceNext)
    {
        LevelUp();
    }
}
```

Figure 2.32: Left: Add Experience Function, Right: Get Experience Function

the controller's device velocity will be read. If the dot product between the device's velocity and forward direction vector is greater that .1, the fireball game object will be instantiated at the controller's position and its cooldown will be activated.

The fireball game object has the fireball component attached to it which implements the IDamageApplier interface. In its start function the terminate co-routine is called which destroys the game object after 4 + character level seconds. In the update function the fireball game object is moved forward and if the objects collides an enemy, the apply damage function is called which in turn calls the enemy's receive damage function with the fireball's damage and owner properties as its parameters.

In the update function, if the character's level is greater than one, the TargetAoe function will be called which is the targeting mechanic for the call lightning spell. If both grip buttons are being held and the spell is not on cooldown, a ray will be cast from the player's head in the forward direction and the aoe area indicator will be instantiated at the raycast's hit point.

The aoe area indicator game object has the call lightning component attached to it which inherits the AoeSpell class and implements the IDamageApplier interface. The aoe spell class implements the on trigger enter and exit functions which detect enemies that are inside the area and changes the area color from green to red and vice versa on enter and exit. In the call lightning script the targeting mechanic is executed for every frame that the grab buttons are held down. While the player is targeting the are is moved to the raycast's hit point if the hit distance is less than or equal to 10 from the ray origin. Finally, the velocity of the controllers are read and if the dot product between the velocity of each controller and its down direction is greater than 0.4 the spell is cast.

When the spell is cast, the call lightning particle game object is instantiated and if any enemies are inside of its aoe area the apply damage function is called which in turn calls each enemy's receive damage function with the call lightning's damage and owner

properties as its parameters.

2.3.3 Enemies

Both enemies implemented in the game consist of components that define their behaviour and components that synchronize their state across all clients. An enemy component has been implemented, that both enemy base classes components inherit from. Also a scriptable game object has been created for each enemy to save the necessary stats that are accessed in play mode.

The enemy component hold properties that are common for all enemies, such as max and current health, enemy type, agent, animator and more. The component implements the IDamageReceiver interface and in its update function a check is made on its health to determine if the enemy should be dead or not. If the enemy's health is zero or lower the death function is called which disables the collider and rigidbody components and calls the Experience Manager's Add Experience function which notifies all event listeners.

The receive damage function updates the enemy's health via the corresponding synchronization component of each enemy and changes any idle states if needed. Additionally, an EnemyUI component is implemented which handles the health update on the enemy's user interface.

Ghoul

The ghoul base class that inherits from the enemy class is responsible for enabling the behaviour and UI components in play mode.

The GhoulSync realtime component is derived from the GhoulModel RealtimeModel component which has three RealtimeProperty fields, one for the target, one for the health and one for the state. Each of these realtime properties has a didChangeEvent which is very useful for notifying clients of value changes.

In the GhoulSync component there are implemented functions for setting the model's properties values as well as functions for handling value changes when the didChangeEvent of a property is fired. The UpdateGhoulHealth function is called when the HealthDidChange event is fired and sets the health variable value of the ghoul's local component to be equal to the model's health value. The UpdateGhoulState and UpdateGhoulTarget functions synchronize the ghoul's state and target to match the ones of the model.

The behaviour of the ghoul enemy is simple, having two active states, an idle one and the death state. In the idle state a Physics.OverlapSphere function is used to detect players in a 10 unit radius and if a target is detected, the model's target property is set and the state is changed to Chase.

In the chase state the animator walk state is enabled and the ghoul is moved towards the target and if the distance between them is less than or equal to two, the state is changed to Attack if it is not on cooldown. In the Attack state the distance between the ghoul and the target is checked again and if it is less than or equal to two, the animator attack state is enabled.

In the animator's attack state there are two events added, one in the middle and one in the end. The middle event calls the attack function which checks the distance again and if the check passes the attack is set on cooldown and the receive damage function on the target is called. The end event sets the ghoul state back to chase.

Finally, if the ghoul's health is zero or less, the death state is entered, the animator death state is enabled and the behaviour component is disabled.

Boss Enemy

The necromancer is the boss enemy and its base class that inherits from the enemy class is responsible for enabling the behaviour and UI components in play mode. The necromancer's synchronization components follow the exact same architecture as the ghoul ones with the minor difference that instead of a state property the NecromancerModel has a transition property.

```
[RealtimeModel]
7 usages
public partial class NecromancerModel
{
    [RealtimeProperty(1, reliable: true, createDidChangeEvent: true)]
    private string _target;
    [RealtimeProperty(2, reliable: true, createDidChangeEvent: true)]
    private int _health;
    [RealtimeProperty(3, reliable: true, createDidChangeEvent: true)]
    private string _transition;
}
```

Figure 2.33: Necromancer Model

The necromancer's behaviour is more complicated with its states consisting of patterns

with common elements so it is implemented with nested state machines. The state machine implementation that is used is taken from the web [**?**].

The NecromancerBehaviour component has properties related to the necromancer base class, animator, states, transitions and it implements the master state machine. In the start method the state machine object is created along with the state objects needed, the possible transitions from each state with the desired transition conditions are declared and the starting state of the state machine is set to idle.

```
_stateMachine = new StateMachine();
idle = new NecromancerIdle(necromancerB: this);
attack1 = new NecromancerAttack1(necromancerB: this);
attack2 = new NecromancerAttack2(necromancerB: this);
attackAoe = new NecromancerAoe1(this);
attackAoe2 = new NecromancerAoe2(this);
backwalk = new NecromancerBackwalk(necromancerB: this);

//TRANSITIONS

At(from: idle,  to: backwalk,ToBackWalk);
At(from: idle,  to: attack1,ToAttack1);
At(from: idle,  to: attack2,ToAttack2);
At(from: idle,  to: attackAoe,ToAttackAoe);

At(from: backwalk,  to: idle,ToIdle);
At(from: backwalk,  to: attack1,ToAttack1);
At(from: backwalk,  to: attack2,ToAttack2);
At(from: backwalk,  to: attackAoe,ToAttackAoe);

At(from: attack1,  to: idle,ToIdle);
At(from: attack1,  to: backwalk,ToBackWalk);
At(from: attack1,  to: attack2,ToAttack2);
At(from: attack1,  to: attackAoe,ToAttackAoe);

At(from: attack2,  to: idle,ToIdle);
At(from: attack2,  to: backwalk,ToBackWalk);
At(from: attack2,  to: attack1,ToAttack1);
At(from: attack2,  to: attackAoe,ToAttackAoe);

At(from: attackAoe,  to: idle,ToIdle);
At(from: attackAoe,  to: backwalk,ToBackWalk);
At(from: attackAoe,  to: attack1,ToAttack1);
At(from: attackAoe,  to: attack2,ToAttack2);

//START STATE
_stateMachine.SetState(idle);
_state = NecromancerState.Idle;
```

```
private bool ToIdle()
{
    if(_transition == NecromancerBehaviour.Transitions.Idle)
        _state = NecromancerState.Idle;
    return _transition == NecromancerBehaviour.Transitions.Idle;
}
Frequently called  4 usages
private bool ToAttack1()
{
    if(_transition == NecromancerBehaviour.Transitions.Attack1)
        _state = NecromancerState.Attack1;
    return _transition == NecromancerBehaviour.Transitions.Attack1;
}
Frequently called  4 usages
private bool ToAttack2()
{
    if(_transition == NecromancerBehaviour.Transitions.Attack2)
        _state = NecromancerState.Attack2;
    return _transition == NecromancerBehaviour.Transitions.Attack2;
}
Frequently called  4 usages
private bool ToAttackAoe()
{
    if(_transition == NecromancerBehaviour.Transitions.AttackAoe)
        _state = NecromancerState.AttackAoe1;
    return _transition == NecromancerBehaviour.Transitions.AttackAoe;
}
Frequently called  4 usages
private bool ToBackWalk()
{
    if(_transition == NecromancerBehaviour.Transitions.Backwalk)
        _state = NecromancerState.Backwalk;
    return _transition == NecromancerBehaviour.Transitions.Backwalk;
}
```

Figure 2.34: Left: State Machine Initialization, Right: State Machine Transition Conditions

In the NecromancerBehaviour the SetTransition function can only be call by the owner of the necromancer's realtime component. It takes the available states to transition to

into consideration and comes up with a random transition between them.

All states implement the IState interface which has the Tick, OnEnter and OnExit functions. The idle state enables the idle animation on enter and casts the Physics.OverlapSphere function with a radius of 15 units in the Tick function. If a player is detected, the target is set and the SetTransition function is called.

The states that implement a nested state machine are the attack1, attack2 and attackAoe states. The attack1 state first enters the backwalk state and the projectile attack state after that. The attack2 state executes a loop starting from the run state and transitioning into the projectile attack state four times. When the forth loop is completed, the SetTransition function is called. The Aoe1 state enters the float state and after the enemy floats upwards, the Aoe attack state is entered. When the Aoe attack state is exited, the float state is entered once more and the enemy floats downwards before the SetTransition function is called.

The float state on enter disables the enemy agent component and enables the float animation. Based on the iteration property which is incremented on enter, the gravity of the enemy is disabled, the kinematic rigidbody property enabled and a high position is selected to move to if it is the first iteration, or a low position is selected if it is the second. In the tick method the enemy moves to the selected position and when the movement is completed, if it is the second iteration, the SetTransition method is called. On exit the agent and gravity are enabled again, the kinematic property is disabled and the iteration is set to 0.

When entering the backwalk state, the backwalk animation is enabled, the agent enabled and the agent set destination is called with a position to the back of the enemy. In the tick method if the time passed in this state is more than a second and if the state was activated in the master state machine, the SetTransition function is called.

When entering the run state the run animation and the agent are enabled and the set destination method is called to the left or right. In the tick method if the attack2 state machine is active, the enemy runs for 1.5 seconds if the target is in sight or for 3 seconds if it isn't and then the SetTransition function is called.

When entering the projectile attack state, the projectile attack animation is enabled and the agent is stopped. The projectile attack animation has two events added to it, one in the middle and one in the end. The middle event calls the Fire Projectile function in the necromancer behaviour component which instantiates the projectile particle game object and increments the attack2count variable if the realtime is owned locally. If the

attack1 state is active the cooldown is enabled otherwise the fire projectile is called four times after the cooldown is enabled and the SetTransition function called. The end event signals that the projectile animation attack is over so that any transitions in the nested state machine can take place.

When entering the aoe attack state the aoe attack animation is enabled which also has two events added to it, one in the middle and one in the end. The middle event calls the Cast Aoe function of the necromancer behaviour component which instantiates the aoe attack particle game object if the realtime is owned locally.

The projectile attack game object has the NectoricTouch component attached to it which moves it in the forward direction in the update method. Upon collision with the player the apply damage function is called which calls the receive damage function of the character and the apply condition function. The apply condition function calls the receive condition function of the character with the blind condition and a duration of 2 as its parameters.

The aoe attack game object has the PoisonBath component attached to it which starts the self destruct co-routine at start, hence destroying it after four seconds. In the update method the DealDamage function is called every second which calls the apply damage for any players that are colliding with it. The apply damage function calls the receive damage function of the character and the apply condition function. The apply condition function calls the receive condition function of the character with the poison condition and a duration of 3 as its parameters.

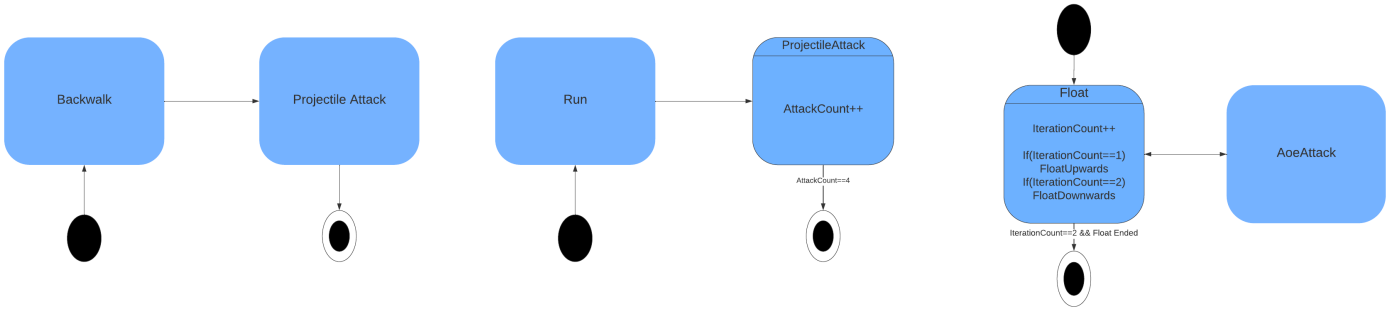

Figure 2.35: Left: Attack1 State Machine Diagram, Middle: Attack2 State Machine Diagram, Right:AttackAoe State Machine Diagram

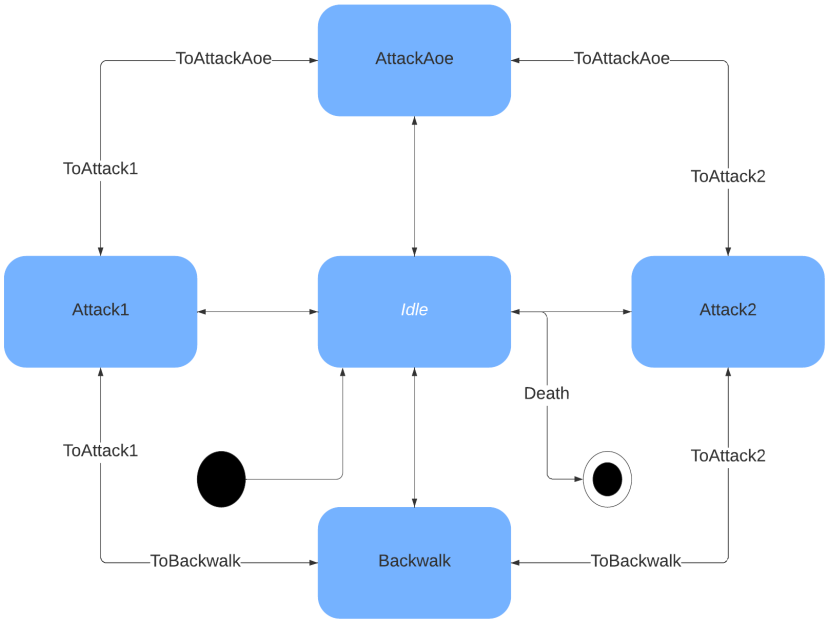

Figure 2.36: Necromancer State Machine Diagram

2.4 Play Mode

When entering play mode, the play scene is loaded, then the starting scene is unloaded. Then the connection request to the realtime room named "PlayRoom" is made and the OnConnectedToPlayRoom function is subscribed as a listener to the realtime didConnectToRoom event.

2.4.1 World Loading

When the client connects to the play room successfully the OnConnectedToPlayRoom is executed and the avatarCreated event of the realtime avatar manager component is subscribed to with the AvatarCreated function implemented. Subsequently, the created world folder in the user's application files is accessed. Then the heightmap, terrainLayers, alphamap, holes and details json files are deserialized into appropriate object types and used to replicate the terrain data and instantiated details of the created world.

The next frame after the OnConnectedToPlayRoom is done, the player avatar is created and is assigned a role based on its realitime client ID. If the avatar's ID is

zero(if it is the first avatar created in the room), the Game Master role is assigned to it and the game master components that are attached to it are enabled. If the avatar is owned locally the XR Origin's rigidbody use gravity property is set to false and the LoadRealtimeObjects function is called. The LoadRealtimeObjects function reads the water, object, sfx and enemy files in the created world folder and instantiates them in the realtime room if they are not already present. Finally, after the objects have been instantiated, the terrain's nav mesh surface is baked.

If the created avatar's realtime ID in greater than zero, the Player role is given to it, the character and inventory components are enabled and the play mode tutorial interface is enabled.

2.4.2 Game Master

The game master component implements the functions for selecting the different game master actions. In the update function the game master action select canvas is enabled if the left controller primary button is held down.

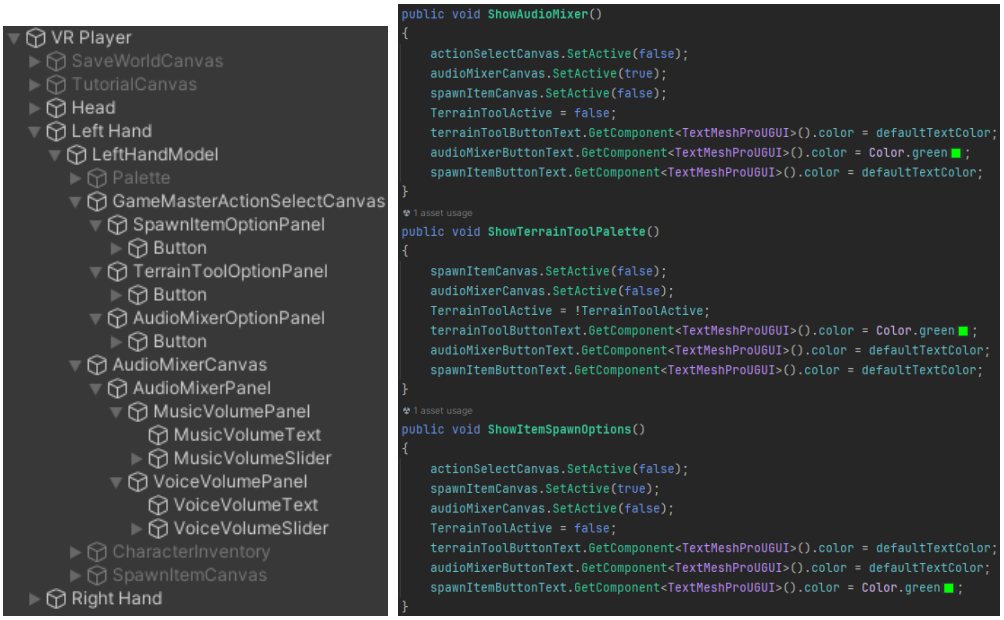

Figure 2.37: Left: Game Master objects in avatar hierarchy, Right: Game Master action selections code (for UI shown in Figure 4.16)

In the Game Master item spawner component, after the "Door Key" item has been

selected in the spawn item interface, the wait for spawn position variable is set to true. In the update method if the wait for position is true, a ray is cast when the trigger button is pressed and the item is instantiated at the raycast hit point.

The Game Master audio mixer gives the game master the ability to mix the voice and music channels of the master audio of all clients. The game master audio mixer component inherits from the Realtime component of type Mixer Volume Model which is a Realtime Model class with the voice and music volume float realtime properties.

```
[RealtimeModel]
6 usages
public partial class MixerVolumeModel
{
    [RealtimeProperty(1, reliable: false, createDidChangeEvent: true)]
    private float _voiceVolume;
    [RealtimeProperty(2, reliable: false, createDidChangeEvent: true)]
    private float _musicVolume;
}
```

```
private void VoiceVolumeChanged(MixerVolumeModel model, float voiceVolume)
{
    mixer.SetFloat(name: "voiceVolume", model.voiceVolume);
}
2 usages
private void MusicVolumeChanged(MixerVolumeModel model, float musicVolume)
{
    mixer.SetFloat(name: "musicVolume", model.musicVolume);
}
1 asset usage
public void VoiceVolumeSliderChanged()
{
    model.voiceVolume = voiceSlider.value;
}
1 asset usage
public void MusicVolumeSliderChanged()
{
    model.musicVolume = musicSlider.value;
}
```

Figure 2.38: Left: Mixer Volume Realtime Model, Right: Game Master audio mixer component methods (for UI shown in Figure 4.16)

The Game Master audio mixer implements the functions for setting the model's music and voice volume properties. Furthermore, it implements the functions that handle the value changes of the model's properties when the didChange event is fired. In order to be able to change the master mixer channel's volumes from script, a float parameter needs to be created for each of the channels that is then exposed from the Unity editor.

2.4.3 Gameplay

A few more components have been built to implement the extra gameplay mechanics that are needed.

Audio Emitter

The audio emitter component is attached to the sfx objects and contains an audio

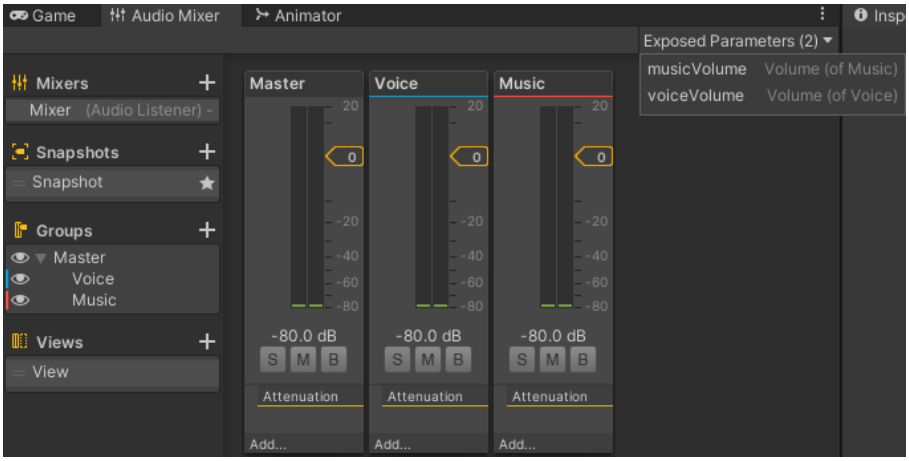

Figure 2.39: Audio mixer with exposed parameters

source property which is the reference to the audio source component that handles audio play back. The audio source component has an audio clip property which is the actual audio file played to the selected output channel which is the music channel for all sfx objects.

The audio emitter script implements the on trigger enter and exit functions start and stop audio playback when the player enters or exits that object's trigger collider. The audio tracks used as audio clips for the sfx objects are composed by Quest Master [**?**].

Crafting Bench

The crafting bench game object offers crafting features with the attached crafting bench component attached to it that is implemented. In the update function the Physics.OverlapSphere method is called and if there is a player in range two items needed for crafting are added to the character's inventory the first time and the open crafting interface is enabled. If the player activates the object the crafting interface is enabled, displaying a crafting panel for each item that is available for crafting. The item crafting panel is instantiated with an ingredient panel for each ingredient needed to craft that item and a result item panel which indicates the craft result.

Bomb

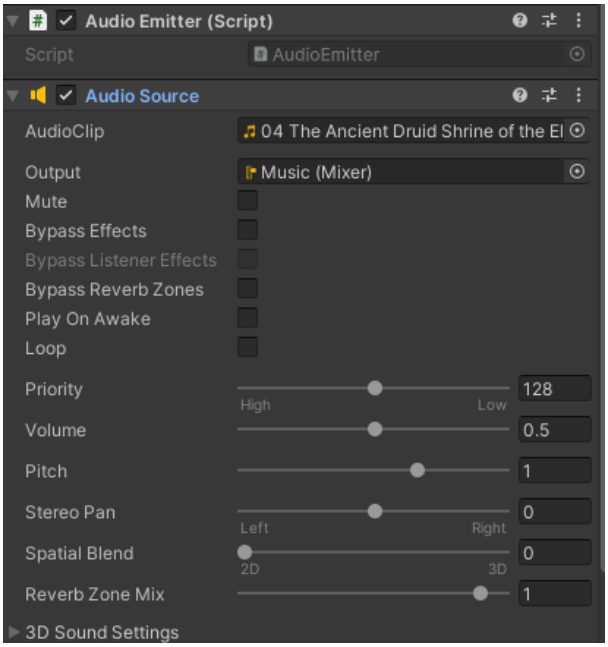

Figure 2.40: The audio emitter and audio source components attached to the sfx object (for objects shown in Figure 4.11)

The bomb object is used to destroy door objects and the bomb component implements this functionality. When the bomb object is instantiated, the explode co-routine is started and after five seconds, any door objects that are nearby are destroyed. When the bomb explodes, the explosion particle system is instantiated and the bomb game object is destroyed.

Doors

There are three types of doors that are implemented, left opening, right opening and double doors with the three corresponding components functioning in a similar way. A boolean property named open along with closed and open rotation properties exist in each component. When the object is activated the open property is changed and the door starts rotating towards the desired rotation until it reaches it.

The double doors however can only be opened if the player has acquired the door key from the Game Master.

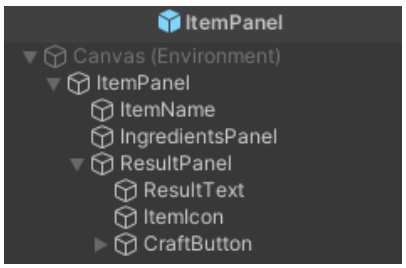

Figure 2.41: Craftable item interface panel (shown in Figure 4.19)

```
private void InitBombCraftable()
{
    GameObject craftableItem = Instantiate(itemPanelPrefab, craftPanel.transform);
    craftableItem.name = "Bomb";
    craftableItem.transform.GetChild(0).GetComponent<TextMeshProUGUI>().text = "Bomb";
    GameObject craftableIng = Instantiate(ingredientIconPrefab, craftableItem.transform.GetChild(1).transform);
    craftableIng.GetComponent<UnityEngine.UI.Image>().sprite = icons[2];
    GameObject craftableIng2 = Instantiate(ingredientIconPrefab, craftableItem.transform.GetChild(1).transform);
    craftableIng2.GetComponent<UnityEngine.UI.Image>().sprite = icons[1];
    craftableItem.transform.GetChild(2).GetChild(1).GetComponent<UnityEngine.UI.Image>().sprite = icons[3];
    craftableItem.transform.position =
        new Vector3(craftableItem.transform.position.x, craftableItem.transform.position.y, z: 0);
    craftableItem.transform.GetChild(2).GetChild(2).GetComponent<Button>().onClick.AddListener(CraftBomb);
}
1 usage
private void CraftBomb()
{
    _character.Inventory.CharacterInventory.Remove(item: "Black Pearl");
    _character.Inventory.CharacterInventory.Remove(item: "Rope");
    _character.Inventory.CharacterInventory.Add(item: "Bomb");
    GameObject itmBtn = Instantiate(itemButtonPrefab, parent: _character.Inventory.InventoryCanvas.transform.GetChild(0));
    itmBtn.transform.GetChild(0).GetComponent<TextMeshProUGUI>().text = "Bomb";
    itmBtn.GetComponent<Button>().onClick.AddListener(call: () => _character.Inventory.UseBomb());
    RemoveItemPanel(n: "Bomb");
}
2 usages
private void RemoveItemPanel(string n)
{
    for (int i = 0; i < craftPanel.transform.childCount; i++)
    {
        if (craftPanel.transform.GetChild(i).name == n)
            Destroy(craftPanel.transform.GetChild(i).gameObject);
    }
}
```

Figure 2.42: Crafting implementation (shown in Figure 4.19)

```
void Start()
{
    _loader = GameManager.Instance._realtime.GetComponent<CustomRealtimePrefabLoadDelegate>();
    StartCoroutine(routine: Explode());
}

Frequently called  1 usage
private IEnumerator Explode()
{
    yield return new WaitForSeconds(5f);
    var inArea :Collider[] = Physics.OverlapSphere(transform.position, radius: 10, layerMask: 1 << 16);
    if (inArea.Length > 0)
    {
        foreach (var coll :Collider in inArea)
        {
            Debug.Log(message: "LOOKING FOR DOOR COMPONENT");
            if (coll.gameObject.TryGetComponent<OpeningDoubleDoor>(out OpeningDoubleDoor dd)
                || coll.gameObject.TryGetComponent<OpeningLeftDoor>(out OpeningLeftDoor ld)
                || coll.gameObject.TryGetComponent<OpeningRightDoor>(out OpeningRightDoor rd))
            {
                Debug.Log(message: "FOUND DOOR COMPONENT");
                Realtime.Destroy(coll.gameObject);
            }
        }
    }
    GameAddressablesManager.Instance.RealtimeInstantiateAsset(_loader.Particles[5], transform.position, Quaternion.identity, instances: null);
    Realtime.Destroy(gameObject);
}
```

Figure 2.43: Bomb implementation

```
    if (_open && door.transform.rotation != _openRotation) {
        door.transform.rotation = Quaternion.RotateTowards(from: door.transform.rotation, to: _openRotation, maxDegreesDelta: rotationSpeed * Time.deltaTime);
        if (Quaternion.Angle(a: door.transform.rotation, b: _openRotation) < 0.1f) {
            door.transform.rotation = _openRotation;
            if (!_navMeshBuilt) {
                // Rebuild NavMesh when door is fully open
                GameManager.Instance.PlayTerrain.GetComponent<NavMeshSurface>().BuildNavMesh();
                _navMeshBuilt = true;
            }
        }
    }
    else if(!_open  && door.transform.rotation != _closedRotation)
    {
        door.transform.rotation = Quaternion.RotateTowards(from: door.transform.rotation, to: _closedRotation, maxDegreesDelta: rotationSpeed * Time.deltaTime);
        if (Quaternion.Angle(a: door.transform.rotation, b: _closedRotation) < 0.1f) {
            door.transform.rotation = _closedRotation;
            if (!_navMeshBuilt) {
                // Rebuild NavMesh when door is fully open
                GameManager.Instance.PlayTerrain.GetComponent<NavMeshSurface>().BuildNavMesh();
                _navMeshBuilt = true;
            }
        }
    }
}

2 asset usages
public void Open() {
    GetComponent<RealtimeView>().RequestOwnership();
    door.GetComponent<RealtimeView>().RequestOwnership();
    door.GetComponent<RealtimeTransform>().RequestOwnership();
    _open = !_open;
    if (_open)
        _actionPrompt.transform.GetChild(0).GetChild(0).GetComponent<TextMeshProUGUI>().text = "Close Door";
    else
        _actionPrompt.transform.GetChild(0).GetChild(0).GetComponent<TextMeshProUGUI>().text = "Open Door";
    _navMeshBuilt = false;
}
```

Figure 2.44: Door rotation