The NSG Program - Pixelated Reverie

Tech Design Document

Contents

Figures:

Introduction:

Welcome to the Tech Design Document for the NSG Program - Pixelated Reverie my third year Final Major Project on the BSC Games Technology course at the University for the Creative Arts. This project began Pre-Production in September 2023, followed by Production in January 2024. This project was a passion project of mine which I intend to pitch towards the University's Incubator Studio upon my graduation, as I have a full plan for the game as a complete title. This was a solo project created within Unity, with version 2022.3.0f1 using the High Definition Render Pipeline. This document will contain descriptions for each of the scripts written for this game, explaining their functions within the game and how they tie together. Each script will discuss the variables and functions that make them up. This introductory segment will explain the concept of the game, before beginning the journey into technical jargon.

Game Idea:

The NSG Program - Pixelated Reverie is a hybrid between first-person puzzle games and 2D platforming games. The game has the player explore a digital world made of surreal visuals as they swap between 3D and 2D perspectives and gain new abilities. This game will contain a mix of different ideas which are explored to create interesting puzzle scenarios for the player to explore. The game will be a heavily narrative based experience, exploring a plot of nostalgia as a form of escapism in the face of serious topics, and will be designed to draw the player in with fun mechanics and visuals only to slowly unearth the dark atmosphere beneath the surface. The project this year will all exist within one main environment, a large Cliffside overrun with a giant flower, though the full game idea that will be pitched to the Incubator studio will contain six different areas, which will hopefully be released in different chapters.

Gameplay Segments:

The NSG Program is played within one large environment, which intersects with itself. This environment contains several different puzzles, both in the 3D and 2D spaces, which I separated as different segments during development. Each segment contains a puzzle for the the player to play, with many serving as introductions to mechanics. Detailed below is a list of each segment, in order, and what mechanics are introduced for the unique areas. These mechanics often add to the mechanics previously established in the system and player movement scripts, and are used to give variety to the gameplay. The listing of the mechanics within this segment will also serve as the order that they will be discussed within this document, with each script being examined in order of appearance. So, without further ado, the segments with the game are as follows:

Segment 1 - Introduction To Game:

This first segment serves as the start of the game. The player finds themselves in a enclosed area with a cliff behind them. The gate ahead is blocked by a gate, forcing the player to interact with the first decal crack, which takes them into the first 2D section of gameplay. Since this is the introduction to the game itself, it does not introduce too many unique mechanics past the basic game systems.

Mechanics Established in Segment 1:

Access to Atari Jump - This segment houses a trigger which activates the Atari Jump ability stored in AtartAbilitySave. Script - AtariAbilityTrigger

Decal Text Appear - This segment introduces triggers that cause decal text to fade onto the walls. Script - DecalTextAppear

Destroyable Objects - This Segment houses a 3D object which gets destroyed when the player passes a trigger. Script - DestroyTrigger

Moving Platforms - This Segment introduces the moving platform obstacles found throughout the game. Script - MovingPlatform2D

Segment 2 - Introduction To Inventory:

The second segment of the game introduces the player to the inventory system in the game. This system is complicated enough to have it's own section discussing it, but aside from the Inventory, this area also introduces a new feature in the Projector, which is used to help plan within the 2D space. This area also originally had the player change camera angles to place items down from the Inventory, which ended up being scrapped.

Mechanics Established in Segment 2:

Projectors - This segment introduces the projector items that reveal the 2D platforms within the 3D space, and are used throughout the game. Script - Projector.

Unused Script - This script was cut from the game, but originally worked to change the camera angle so that the player can place down items from their inventory. Script - Examine_Point.

Segment 3 - 2D Rock Puzzle Introduction:

This segment of the game is directly after the player walks through the game's hub area, and introduces the 2D rock ability. This ability is explored further in the section discussing the rock abilities, the script also introduces the use of Timelines for dialogue, though this is also discussed in the Dialogue section of the document. Aside from these, the main thing introduced in this segment are the switch and gate mechanics, which are used throughout the game.

Mechanics Established in Segment 3:

Switches and Gates - the switch and gate system is used throughout the game. As the game progresses the switch ends up being used for more mechanics, such as HoldPlatforms and to change the end position of moving platforms. Scripts - Switch and SwitchTrigger.

Segment 4 - Floor Platforming:

This section follows the introduction of the 3D Rock ability, and requires the player to knock over a vine to change the path in the 2D space. This area introduces two new scripts and mechanics to the game, the first of which being the ability to change the gravity of the scene. This script is needed for the 2D sections to be positioned on the floor, in order to stop the player from falling through the ground. This area also introduces the HoldPlatforms, a variation on the Moving Platforms that only move when the player presses the switch.

Mechanics Established in Segment 4:

Changing Scene Gravity - This area creates a script that can adjust the direction of the scene's gravity so that the player can play on the floor. This is the only scene that requires this however. Script - ChangeGravity.

Hold Platforms - This variety of platform is used throughout the game, and only moves when a corresponding switch has been pressed. Script - HoldPlatforms.

Segment 5 - Boulder Chaser:

The 5th segment of the game has the player platform on a moving boulder, which reveals platforms as it moves. If the player leaves the confines of the boulder then they die and need to restart the challenge. This area mostly used mechanics established in prior areas, but added some more scripts to make the boulder chasing work.

Mechanics Established in Segment 5:

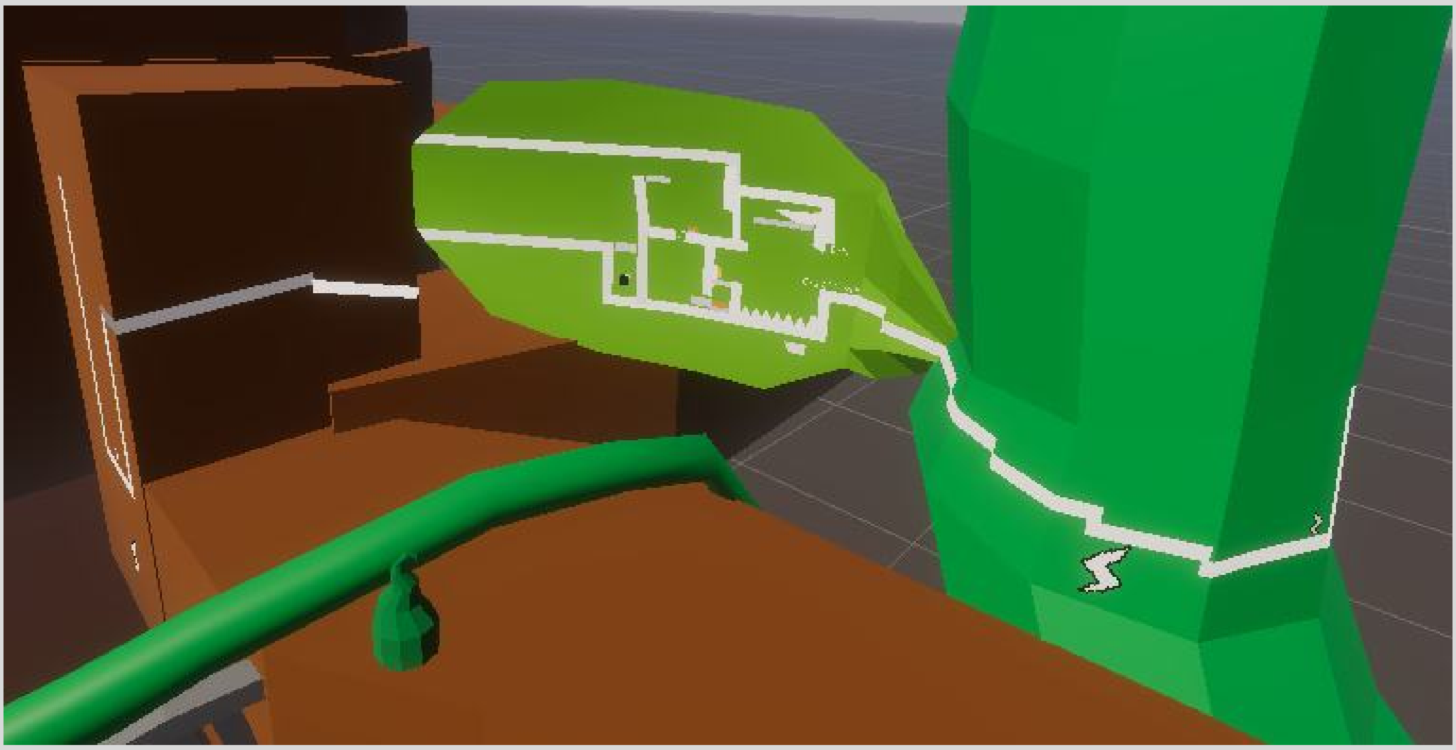
Death When Leaving the Boulder - For the challenge to work, the player needs to die when not keeping up with the boulder, which works well. Script - RockTriggerBool.

Respawning Boulder - An issue that occured when testing out this segment was in regards to the animation of the boulder, which the player needed to wait for before being able to continue the scene, I counteracted this by having the boulder respawn when the player nears it, so that they no longer have to wait. Script - TriggerAnim.

Segment 6 - Leaf Level:

The 6th segment of the game was designed to be a culmination of every mechanic set up in the prior section of the game. As such, this segment doesn't introduce any new mechanics or systems, and was designed more to showcase the level design potential of the mechanics previously added. This segment has the player knock over a leaf with their rock in order to make the path forward, and uses all mechanics from Hold Platforms to switches and gates. I think this works quite well as a culmination, but didn't introduce anything new.

Figure 1 - Segment 6



Segment 7 - Optional Area and Mega Man Introduction:

After the culmination of mechanics that was Segment 6, Segment 7 is mainly there to set up the next 2D Gameplay style, being the Mega Man style of gameplay. Because of this, the segment is filled with new mechanics unique to that style of gameplay, which works quite well. In addition, this segment has an optional area of gameplay that serves to be it's own unique platforming challenge that the player can experience just for fun. This optional area has the player need to use the 3D rock ability to block a leak of water, which then effects the 2D section in turn.

Mechanics Established in Segment 7:

Water Leak - In the optional area, the player needs to block a leak of water in the wall to change the platforms in the 2D area. Script - WaterBlock.

Springs - The optional area introduces springs which increase the height of the player's jump when triggered with. These are used later in the game to bounce off the pollen shot by the Mega Man Character. Script - Spring.

3D Switches - The 7th segment introduces switches in the 3D space, which when pressed open up gates in the 3D area. Script - Switch3D.

2D Vines and Thorns - The introduction to the Mega Man player introduces 2D vines and thorny spikes which the player can grow and shrink using their shot pollen. Scripts - plantgrow and spikevine.

PollenUI - To help the player understand what type of pollen they are shooting, a new bit of UI is set up in this area that shows the current UI to be shot.

Segment 8 - 3D Flower Gun Puzzles:

The eight segment of the game houses two 2D platforming sections, both introducing new elements accessed via the 3D flower gun ability. These puzzles are short and simple require the player needing to shoot vines with their 3D pollen to change the size of the pollen. The first of the 2D sections is a redo of the second segment, with the player increasing the size of the vines using the pollen rather then placing them down as inventory items. The second is a puzzle with springs that change positions based on the size of the vines grown. These springs ended up using the Vines script made for the inventory items, but before that there was a script made for it called SwapSprings, which ended up being obsolete.

Mechanics Established in Segment 8:

Swapping Spring Position - This was done with the Vines script, though had a unused script that was originally used for this. Script - SwapSprings.

Segment 9 - Spring Vine Puzzle:

The ninth section of the game involved the player solving one giant spring puzzle, where the player needs to grow the vines in order to make a path for the spring to hit a switch. This was originally the first instance of this type of puzzle, but proved to be too complicated to be a good introduction, so instead I moved it back and create a simpler version in Segment 8. This area introduces moving platforms in the 3D space, which required new code to the 3D player's movement, which will be explained in this script.

Mechanics Established in Segment 9:

3D Moving Platforms - These 3D Moving platforms are used to help the player see things in different angles, and used the MovingPlatform2D script for this, but needed a new trigger for the 3D player to keep them parented to the object. Script - FPSTrigger.

Segment 10 - Final Segment:

The 10th segment of the game serves as the final one, and is a big finale to the game, incorporating every mechanic previously established, from the Inventory system to 3D buttons and the rock hit ability. To properly incorporate everything however, I needed to create a new system, I needed to create a means for the player to swap between 2D gameplay styles on the fly, so that every mechanic could be properly incorporated.

Mechanics Established in Segment 10:

2D Gameplay Swap - The final area had a trigger that when entered would swap the 2D gameplay style over on the other side of the trigger, doing this required a new script just for it. Script - SwapCharacter.

System Scripts:

This section of the document focuses on scripts that are more in the background, working to keep the game running smoothly This will also include scriptable objects created for use in the game store values across the play experience. Finally, this section will also discuss elements used for debugging that were useful for the overall development cycle. So, let's begin looking at these scripts.

AtariAbilitySave:

The Atari Ability Save script is a scriptable object used to store information on what abilities the player has unlocked in the game. The script is quite short, serving it's purpose as a scriptable object well, being stored as a item within the project files. The script is named this way as it was originally used to store the abilities of the 2D Atari gameplay style, rather then every gameplay style.

AtariAbilitySave - Variables and Awake():

```
[CreateAssetMenu(menuName = "AtariSave", order = 2, fileName = "Atari Save")]
public class AtariAblilitySave : ScriptableObject
{
    //This is the scriptable object which stores the bools which allow the player to use the different abilities.
    //These abilities include the 2D players jump and rock placing, and the 3D players rock throw and flower gun.
    public bool AtariJump, AtariRock, tDRock, FlowerGun;
    void Awake()
    {
        //Each of the bools are set to false upon the game initialy beginning.
        AtariJump = false;
        AtariRock = false;
        tDRock = false;
        FlowerGun = false;
    }
}
```

This script contains only three variables, all of which are bools. They are as follows:

AtariJump (bool) - This bool determines when the player has unlocked the ability to jump as the 2D Atari character.
AtariRock (bool) - This bool determines when the player has unlocked the ability to place rocks as the 2D Atari character.
tDRock (bool) - This bool determines whether the 3D player has unlocked the ability to throw rocks.
FlowerGun (bool) - This bool determines whether the 3D player has unlocked the flower gun ability.

This script, being made to work as a object in the project files, was made to be as simple as possible, with only one function. This function is the void Awake(), which is automatically called at the start of the game. This function stores all the previously mentioned bools to be false, so that the player has to unlock them as the game progresses.

StoreClass and StoreTransform:

These two scripts are placed together, as the two are innately linked, and are quite short. Store Class is a class script, meaning it only exists to be used as a variable in other scripts. This variable created from the class is then used in Store Transform, a scriptable object used for debugging purposes. These scripts were made to make the process of keeping track of the values needed to rotate around corners easier, like a notebook stored as a object in the project files. Without the system created from these scripts, the process of adding corner rotations to the 2D section would be much more annoying.

Store Class:

```
[System.Serializable]
//This is a class used to store the values needed for corner rotations in the 2D space.
public class StoreClass
{
    public string Name;
    public Vector3 Pos;
    public Quaternion Rot;
    public Vector3 Scale;
    public Vector3 Norm;
    public float FOV;
}
```

This script creates a class that incorporates several different variables, which are used to create a list of the values needed for corner rotations:

Name (string) - This string is used to store a name for the collection of variables, so that it is more clear what each element is used for.
Pos (Vector 3) - This Vector 3 is used to store the position data of an object that can be copied later.
Rot (Quaternion) - This Quaternion is used to store the rotation data of an object that can be copied later.
Scale (Vector 3) - This Vector 3 is used to store the scale data of an object that can be copied later.
Norm (Vector 3) - This Vector 3 is used to store the normal of an object that can be copied later.
FOV (float) - This float stores the Frame of View of Camera components that can be copied later.

Store Transform:

```
[CreateAssetMenu(menuName = "TransformSave", order = 3, fileName = "Store Transform")]
public class StoreTranform : ScriptableObject
{
    //This is a scriptable object used to easily copy and paste the Tranforms of objects in the game.
    public StoreClass[] Save;
}
```

This script is only used to create a scriptable object which stores an array of variables under the StoreClass class. This is then used to take note of Transform data of objects and compile them all in one easy to copy list. This is a life saver and something I'm glad to have created, even if it seems very small.

GenreSwap:

This script is one of the most important in the entire game, being the key to swapping between the 2D and 3D gameplay styles. This is done either with the 3D player using a raycast to collide with it, or for 2D player to enter it's trigger. This is done in the game using decal cracks which appear on the walls. So, without further a do, let's look at the script's variables.

GenreSwap - Variables:

```
public class GenreSwap : MonoBehaviour
{
    //This is the script that is used to swap the player between the 2D and 3D states.

    //These GameObjects store the movement for the different gameplay styles.
    [SerializeField] DecalMovement DM;
    [SerializeField] Mega_Man_Movement MMM;

    //This stores the Fade animator when swapping the 2D styles.
    [SerializeField] Animator Fade;
    //These store the variables needed for the different gameplay swaps, such as the markers the player will be moving between, the player and the cracks that are used for the
    transition
    [SerializeField] GameObject[] Markers;
    [SerializeField] GameObject MainPlayer;
    [SerializeField] GameObject Crack1, Crack2, TwoDCrack1, TwoDCrack2, Crack1Trigger, Crack2Trigger, TwoDCrack1Trigger, TwoDCrack2Trigger;
    [SerializeField] GameObject DecalArea;

    //These store the position and normal the 2D player will start at.
    [SerializeField] Vector3 newnormal, newpos;

    //This stores position and rotation the 3D player will start at.
    [SerializeField] Vector3 PlayerPos;
    [SerializeField] Quaternion PlayerRot;

    //This bool is used to check if the crack is swapping between the 2D or 3D gameplay
    [SerializeField] bool TwoDorThreeD;

    //This bool is set to true as the swap is being set up
    [SerializeField] bool SettingUp;

    //This bool is true when the player needs to start in their natural position
    [SerializeField] bool NaturalPos;

    //This stores a Dialogue instance that could play at the start of the 2D segment set up
    [SerializeField] TriggerDialogue TD;

    //This stores the Dialogue_Manager script
    [SerializeField] Dialogue_Manager DI_M;

    //This stores the Press E UI pop up
    [SerializeField] GameObject PressE;

    //This stores the Inventory UI
    [SerializeField] GameObject InventoryItem;

    //This stores the 3D player's CameraRaycast script
    [SerializeField] CameraRaycast CR;

    //This stores the DecalProject projection the script's instance's crack
    [SerializeField] DecalProjector DP;

    //This stores the Cross Hair appearing in the centre screen.
    [SerializeField] GameObject Cursor;

    //If the area needs to change the scene's gravity then it changes it using the SettingGravity script, based on the value of GravityChange.
    [SerializeField] string GravityChange;
    [SerializeField] SettingGravity SG;

    //This stores the positing and rotating that the 2D camera will need to be in at the start of play, if the SetCamera bool is true.
    [SerializeField] bool SetCamera;
    [SerializeField] Vector3 CameraPos;
    [SerializeField] Quaternion CameraRot;
    [SerializeField] GameObject FlatCamera;


    //This stores the audiomanager script
    [SerializeField] audiomanager AM;

    //This stores the 2D rock used in the 2D area
    [SerializeField] GameObject Rock;

    //This stores the PollenUI that appears with the Mega_Man Player
    [SerializeField] PollenUI pollenui;

    //This stores the Projector item that may be projecting the 2D scene
    [SerializeField] Projector P;

    //When this bool is true, it means the 2D player spawned is the Mega_Man.
    [SerializeField] bool StartMegaMan;

    //This stores a voice line that can play before or after the transition
    [SerializeField] VoiceActing VA;
```

This script has a lot of variables, as it stores the information needed when swapping between the two gameplay styles, the variables needed are as follows:

DM and MMM (DecalMovement and Mega_Man_Movement) - These variables are used to store the scripts for the 2D player characters, for either gameplay style.

Fade (Animator) - This stores the animation of the fade that goes on screen whilst the gameplay styles change.

Markers[], MainPlayer, Crack1, Crack2, TwoDCrack1, TwoDCrack2, Crack1Trigger, Crack2Trigger, TwoDCrack1Trigger, TwoDCrack2Trigger, DecalArea (GameObject) - These GameObjects are used when swapping between the 2D and 3D gameplay styles, storing the Markers the player moves between, the 3D Player, the different cracks in both the 3D and 2D space, and the 2D Area itself.

newnormal, newpos (Vector3) - This stores the position and normal that the 2D player is spawned out.

PlayerPos and PlayerRot (Vector3 and Quaternion) - These store the position and rotation that the 3D player should spawn from.

TwoDorThreeD, SettingUp and NaturalPos (bool) - These bools are needed to check how the script is setting up the gameplay. TwoDorThreeD is used to check if the game is swapping to the 2D gameplay style, or 3D style. SettingUp is called when the script begins setting up the gameplay swap and NaturalPos decides whether the player activated should be in their current position or the position values set in the script.

TD, DI_M, PressE, InventoryItem, CR, Cursor (TriggerDialogue, Dialogue_Manager, GameObject, CameraRaycast) - These store different scripts and objects needed for the script. TD stores a dialogue instance that can appear at the start of the 2D space, with DI_M storing the Dialogue_Manager script. PressE and InventroyItem store UI elements, with PressE being the Press E pop-up and InventoryItem storing the Inventory UI. CR stores the CameraRaycast script and Cursor stores the cross hair in the centre of the screen.

DP (DecalProjector) - This stores the DecalProjector of the current 2D crack.

GravityChange and SG (string and SettingGravity) - These store variables used when the 2D space requires a different form of gravity. It changes the direction of gravity by using the string value stored in GravityChange.

SetCamera, CameraPos, CameraRot and FlatCamera (bool, Vector3, Quaternion and GameObject) - These store variables needed to set up the 2D camera in the scene, if it is in a unique position at the start of the section.

AM (audiomanager) - This stores the audiomanager script used on the corner.

Rock (GameObject) - This stores the rock gameObject used in the Atari gameplay styles.

pollenui (PollenUI) - This stores the Pollen UI which appears with the Mega Man Player.

P (Projector) - This stores the Projector which can turn on the 2D Decals.

StartMegaMan (bool) - When this bool is true then the 2D player will be the Mega Man Player.

VA (VoiceActing) - This stores the voice acting clip which can play before or after the transition.

GenreSwap - Start(), Update() and Triggers:

```
//The Start() function stores the value of AM.
private void Start()
{
    AM = GameObject.FindGameObjectWithTag("AControl").GetComponent<audiomanager>();
}

//The Update() function checks if the player has highlighted the 3D crack, and if so waits for them to press E so it can set up the 2D scene
private void Update()
{
    if (TwoDorThreeD == true)
    {
        if (DP.material == CR.Cracks[1])
        {

            if (Input.GetKey(KeyCode.E))
            {
                if (SettingUp == false)
                {
                    PressE.SetActive(false);
                    AM.Crack.SetActive(true);
                    SettingUp = true;
                    switch (TwoDorThreeD)
                    {
                        case true:
                            {

                                StartCoroutine(Setup2D());

                            }
                            break;
                        case false:
                            {
                                    StartCoroutine(Setup3D());

                            }
                            break;
                    }
                }

            }
        }
    }
}

//When entering the Trigger Press E is set to true.
private void OnTriggerEnter(Collider other)
{
    if (TwoDorThreeD == false)
    {
        if (other.gameObject.tag == "Player")
        {
            PressE.SetActive(true);
        }
    }
}
```

```
//This script will allow the player to swap between the 3D or 2D space when pressing E.
    private void OnTriggerStay(Collider other)
    {
        if (TwoDorThreeD == false)
        {
            if (other.gameObject.tag == "Player")
            {
                if (Input.GetKey(KeyCode.E))
                {
                    if (SettingUp == false)
                    {
                        PressE.SetActive(false);
                        SettingUp = true;
                        AM.Crack.SetActive(true);
                        switch (TwoDorThreeD)
                        {
                            case true:
                                {
                                    StartCoroutine(Setup2D());
                                }
                                break;
                            case false:
                                {
                                        StartCoroutine(Setup3D());

                                }
                                break;
                        }
                    }

                }
            }
        }
    }

    //When exiting the trigger PressE is set to false.
    private void OnTriggerExit(Collider other)
    {
        PressE.SetActive(false);
    }
```

The Start() function for this script is used to store the audiomanager, which is stored in AM. The Update() function checks whether the player is highlighting the crack entrance, and if they are and press E, then either the 2D or 3D space will be swap into. When entering or exiting the trigger of the object the Press E UI pop-up will either be activated or deactivated. The OnTriggerStay() function is set up the gameplay transition swap, much like the Update() function. Typically, the Update() function is used for the 3D interactions, and the 2D space is used for the Trigger functions.

GenreSwap - Setup2D():

```
//Sets up the swap from the 3D gameplay to the 2D gameplay.
public IEnumerator Setup2D()
{
    CR.Set2D = true;
    Cursor.SetActive(false);
    DP.material = CR.Cracks[0];
    if(VA != null)
    {
        VA.AddLine();
    }
    Fade.SetBool("Fade", true);
    InventoryItem.SetActive(false);

    if (DM != null)
    {
        DM.LetMove(false);
    }
    if (MMM != null)
    {
        MMM.LetMove(false);
    }

    yield return new WaitForSeconds(0.4f);

    MainPlayer.SetActive(false);
    Crack1.SetActive(false);
    Crack2.SetActive(false);
    DecalArea.SetActive(true);
    if (TwoDCrack1.activeSelf == true)
    {
        TwoDCrack1Trigger.SetActive(true);
    }
    TwoDCrack2Trigger.SetActive(true);

    if (DM != null && StartMegaMan == false)
    {
        DM.gameObject.SetActive(true);
        DM.OriginMark = Markers[0];
        DM.TargetMark = Markers[1];
        DM.SetStart(NaturalPos, newnormal, newpos);
    }

    if (MMM != null && StartMegaMan == true)
    {
        MMM.gameObject.SetActive(true);
        MMM.OriginMark = Markers[0];
        MMM.TargetMark = Markers[1];
        MMM.SetStart(NaturalPos, newnormal, newpos);
    }

    if (SetCamera == true)
    {
        FlatCamera.transform.localPosition = CameraPos;
        FlatCamera.transform.localRotation = CameraRot;
    }

    if (SG != null)
    {
        SG.GravityChange(GravityChange);
        if (DM != null)
        {
            DM.transform.rotation = PlayerRot;
        }
        if (MMM != null)
        {
            MMM.transform.rotation = PlayerRot;
        }
    }
    if(FlatCamera != null)
    {
        FlatCamera.SetActive(true);
    }

    yield return new WaitForSeconds(1f);

    Fade.SetBool("Fade", false);

    if (MMM != null)
    {
        pollenui.gameObject.SetActive(true);
        pollenui.MMM = MMM;
    }
    yield return new WaitForSeconds(0.4f);

    if (TD == null || TD.gameObject.activeSelf == false)
    {
        if (DM != null)
        {
            DM.LetMove(true);
        }

        if (MMM != null)
        {
            MMM.LetMove(true);
        }
    }
    else
    {
        if (DM != null)
        {
            DI_M.DM = DM;
            DI_M.StartDialogue(TD.D);
            TD.gameObject.SetActive(false);
        }

        if (MMM != null)
        {
            DI_M.MMM = MMM;
            DI_M.StartDialogue(TD.D);
            TD.gameObject.SetActive(false);
        }
    }
    SettingUp = false;
    gameObject.SetActive(false);
}
```

The Setup2D() IEnumerator is used to swap from the 3D gameplay style to the 2D gameplay. It starts by having the fade cover the screen, as it deactivates the UI on screen and stops the selected 2D player from being able to move. After a few seconds wait, the script deactivates the 3D player and cracks, and activates the 2D player (whether Atari or Mega Man) and the 2D cracks. If the camera needs to be setup then the SetCamera bool is set to true, and the camera's position is set. Next the script checks if the 2D area needs a different gravity setting, then the SG variable will not be null, and the new gravity will be set. The function ends by activating the Pollen UI if the Mega Man player is playable, and any dialogue if it is needed, before activating the 2D area and removing the fade, so that the player can begin playing. This function is long, but provides everything necessary for a 2D area to work. It was also refined to be workable for either type of 2D Gameplay.

GenreSwap - Setup3D():

```
//Sets up the swap from the 2D gameplay to the 3D gameplay
public IEnumerator Setup3D()
{
    //This functions starts with a fade before deactivating the 2D character and activating the 3D character.
    Fade.SetBool("Fade", true);
    yield return new WaitForSeconds(0.4f);
    MainPlayer.SetActive(true);
    Crack1.SetActive(true);
    Crack2.SetActive(true);
    Crack2Trigger.SetActive(true);
    Crack1Trigger.SetActive(true);
    TwoDCrack1.SetActive(true);
    MainPlayer.transform.position = PlayerPos;
    MainPlayer.transform.rotation = PlayerRot;

    //The function resets every aspects of the 2D players so there are no glitches when entering the 2D area again.
    if (DM != null)
    {
        DM.LetMove(false);
        DM.transform.parent = null;
        DM.gameObject.SetActive(false);
    }
    if(MMM != null)
    {
        MMM.LetMove(false);
        MMM.transform.parent = null;
        MMM.gameObject.SetActive(false);
    }

    if (SG != null)
    {
        SG.GravityChange("Regular");
    }

    if (Rock != null)
    {
        Rock.transform.parent = null;
        Rock.SetActive(false);
    }
    if (MMM != null)
    {
        pollenui.gameObject.SetActive(false);
    }

    if(FlatCamera != null)
    {
        FlatCamera.SetActive(false);
    }

    if(DM != null)
    {
        DM.LeftAble = true;
        DM.RightAble = true;
    }
    if(MMM != null)
    {
        MMM.LeftAble = true;
        MMM.RightAble = true;
    }
    yield return new WaitForSeconds(1f);
    //The function ends by reactivating all the UI and either keeps the 2D area active depending on the Projector.
    CR.Set2D = false;
    Cursor.SetActive(true);
    InventoryItem.SetActive(true);
    Fade.SetBool("Fade", false);
```

```
        if (VA != null)
        {
            VA.AddLine();
        }
        SettingUp = false;
        gameObject.SetActive(false);

        if (P == null)
        {
            DecalArea.SetActive(false);
            gameObject.SetActive(false);
        }
        else if (P != null)
        {
            if (P.Keepon == false)
            {
                DecalArea.SetActive(false);
                gameObject.SetActive(false);
            }
        }
        yield return new WaitForSeconds(0.4f);
```

The Setup3D() IEnumerator does the opposite of the Setup2D() function, being that it deactivates the 2D character and places the 3D character in the spawned location. This function resets every aspect of the 2D player so that there are no glitches when re-entering the 2D space later. It then activates all the UI and either turns off the 2D platforms based on whether the Projector is on. This script works well to tie the gameplay styles together, and is one of the most important in the full script.

CornerTrigger:

The CornerTrigger script is called in the 2D space to set up the 2D player for corner rotations. This is called CornerTrigger as originally it was called via a trigger collision, but eventually adapted to be done via distance calculations and not triggers, as it is more consistent and accurate. This script is designed to store values which are then placed in the 2D Player's rotation set up. Because of this, the script has a lot of variables, which are as follows:

CornerTrigger - Variables:

```
public class CornerTrigger : MonoBehaviour
{
    //This script works to setup the corner rotations for the 2D player

    //This stores Atari Player for the corner
    public DecalMovement DM;
    //This stores the Mega Man Player for the corner
    public Mega_Man_Movement MMM;
    //This sets the Zelda Player for the Corner
    public Zelda_Movement ZM;

    //This decides if the rotation is in the X or Y axis
    public bool XorY;

    //Stores the values used to calculate the corner rotation
    public Vector3[] NormalSave;
    public Vector3 SetNormal;
    public Vector3[] LocationSave;
    public Quaternion[] RotationSave;

    //This stores the different markers for the corner, and the speed at which the platform rotates.
    public GameObject[] Markers;
    public float NewSpeed;

    //These store the rotating values of the Camera if needed.
    public bool CameraB, DoBoolOnce;
    public Vector3[] CLocationSave;
    public Quaternion[] CRotationSave;
    public float[] CFOVSave;

    //This checks the how close the player needs to be before they can rotate.
    public float[] TurnCheck;
```

```
//This stores the distance between theOrigin object and the Target
public float OriginDis;
public float TargetDis;

//This decides if the camera rotation should be done via local or world transforms
public bool LocalR;

//This bool is true when the Atari and Mega Man players are playable in the same 2D space.
[SerializeField] bool TwoObject;

//This stores new rotation values for the 2D Mega Man player if that player is found within the same space as an Atari player.
public Vector3[] SecondNormalSave;
public Vector3[] SecondLocationSave;
public Quaternion[] SecondRotationSave;
```

This script stores several variables used to set up the corner rotation, which are as follows:

DM, MMM and ZM (DecalMovement, Mega_Man_Movement and Zelda_Movement) - These store the different 2D player types, to set them up for the wall rotation.

XorY (bool) - This bool decides if the rotation is in the X or Y axis.

NormalSave[], **SetNormal, LocationSave**[] **and RotationSave**[] (Vector3 and Quaternion) - These store the normal, position and rotation of the player before and after a corner rotation.

Markers[] **and NewSpeed** (GameObject, float) - This sets up the markers used to move on the different walls before and after rotation, and the speed at which the rotation takes place.

CameraB, DoBoolOnce, CLocationSave[], **CRotationSave**[] **and CFOVSave**[] (bool, Vector3, Quaternion, float) - These variables work to create a rotation change if the camera needs to change position around a corner like the way the player does. This only happens if CameraB is true, and will only happen one way if DoBoolOnce is true. Rather then storing a normal, this stores the Frame Of View of the camera.

TurnCheck[], **OriginDis and TargetDis** (float) - These floats are used to calculate the distance the player needs to be from the corner to initiate rotation, with TurnCheck storing the distance on either side of the corner.

LocalR (bool) - When this bool is true it means the changes to the camera are done in the local transform, not the world transform.

TwoObject (bool) - When this bool is true it means both the Atari and Mega Man players are playable in the 2D space, and as such the Mega Man player needs its own set of rotation values.

SecondNormalSave[], **SecondLocationSave**[], **SecondRotationSave[]** (Vector3 and Quaternion) - These arrays store the values for the Mega Man rotation if both the Atari and Mega Man players are playable in the 2D space.

CornerTrigger – Script:

```
    public void SetupRotate()
    {
        //Checks which script it is setting the script for, and then calls that scripts SetUpRotate Function
        if(DM != null)
        {
            if(DM.gameObject.activeSelf == true)
            {
                DM.rotationspeed = NewSpeed;
                DM.SetUpRotate(LocationSave, NormalSave, RotationSave, Markers);

                if (CameraB == true)
                {
                    if (DoBoolOnce == true)
                    {
                        CameraB = false;
                    }
                    DM.SetUpCameraChange(CLocationSave, CRotationSave, CFOVSave, LocalR);
                }
            }
        }
        if(MMM != null)
        {
            if (MMM.gameObject.activeSelf == true)
            {
                MMM.rotationspeed = NewSpeed;
                if(TwoObject == false)
                {
                    MMM.SetUpRotate(LocationSave, NormalSave, RotationSave, Markers);
                }
                else
                {
                    MMM.SetUpRotate(SecondLocationSave, SecondNormalSave, SecondRotationSave, Markers);
                }

                if (CameraB == true)
                {
                    if (DoBoolOnce == true)
                    {
                        CameraB = false;
                    }
                    MMM.SetUpCameraChange(CLocationSave, CRotationSave, CFOVSave);
                }
            }
        }
        else if (ZM != null)
        {
            ZM.rotationspeed = NewSpeed;

            //Exclusive to the Zelda game play, the script checks whether the corner is rotating in the x or y 
axis.
            switch(XorY)
            {
                case true:
                    {
                        ZM.SetUpRotateX(LocationSave, NormalSave, RotationSave, Markers);
                    }
                    break;
                case false:
                    {
                        ZM.SetUpRotateY(LocationSave, NormalSave, RotationSave, Markers);
                    }
                    break;
            }
        }

    }
}
```

The SetupRotate() script is called by the 2D Movement script and is used to automatically assign the variables needed for the corner rotation. It does this based on the type of player interacting with the corner, and strikes that corner's SetUpRotate() functions to set up the movement around the corner. If the camera needs to be rotated, then the 2D player's SetupCameraChange() function. If both the Atari and Mega Man players are playable in the same 2D space then the Mega Man player uses the Second variable storage, rather then the regular type.

EventsCode:

The EventsCode script is used to call specific events within the game, which may not be otherwise possible in the other scripts in the game. It is one long script used to complete several different purposes, though many of the purposes are quite similar. For example, a lot of it is used to add decal tutorials to the scene when the player looks or interact with items.

EventsCode - Script:

```
public class EventsCode : MonoBehaviour

//This script is used to complete several different things within the game.

//This stores the Inventory item vines collectable by the player
[SerializeField] GameObject[] Vines;

//This stores the item placement spots for the inventory vines
[SerializeField] GameObject[] Planters;

//This stores every Tutorial decal that appears thanks to this script.
[SerializeField] DecalTextAppear[] Decals;

//This stores the first projector of the game, which shows the projector tutorial.
[SerializeField] GameObject Projector;

//This bool is set true when the game is played in the build
public bool Build;

//These are voice acting clips which are played under certain conditions in the game.
[SerializeField] VoiceActing WhereamI;

[SerializeField] VoiceActing VineTake;

[SerializeField] VoiceActing TakeThis;
// Start is called before the first frame update
void Start()
{
    //The start() function checks to see if the game is being played in the build or not
    if(Application.isEditor == false)
    {
        Build = true;
    }
    else
    {
        Build = false;
    }
    //The Start() function ends by playing the WhereamI line.
    WhereamI.AddLine();
}

// Update is called once per frame
void Update()
{
    //The Update() function is spent checking to see if the conditions for the tutorial text to appear have been met.

    if (Decals[0] != null)
    {
        if (Projector.gameObject.layer == LayerMask.NameToLayer("ItemSelection"))
        {
            StartCoroutine(Decals[0].Fadein());
            Decals[0] = null;
        }
    }
if (Decals[1] != null)
        {
            foreach (GameObject G in Vines)
            {
                if (G.gameObject.layer == LayerMask.NameToLayer("ItemSelection"))
                {
                    StartCoroutine(Decals[1].Fadein());
                    Decals[1] = null;
                }
            }
        }

        int activeI = 0;
        if (Decals[2] != null)
        {
            foreach(GameObject G in Vines)
            {
                if(G.gameObject.activeSelf == false)
                {
                    activeI++;

                    if (VineTake != null)
                    {
                        VineTake.AddLine();
                    }
                }

                if(activeI >= 2)
                {
                    StartCoroutine(Decals[2].Fadein());
                    Decals[2] = null;

                    if(TakeThis != null)
                    {
                        TakeThis.AddLine();
                    }
                }
            }
        }

        int secondacitveI = 0;
        if (Decals[3] != null)
        {
            foreach (GameObject G in Vines)
            {
                if (G.gameObject.activeSelf == false)
                {
                    secondacitveI++;
                }

                if (secondacitveI >= 3)
                {
                    StartCoroutine(Decals[3].Fadein());
                    Decals[3] = null;
                }
            }
        }

        if (Decals[4] != null)
        {
            foreach (GameObject G in Planters)
            {
                if (G.gameObject.layer == LayerMask.NameToLayer("ItemSelection"))
                {
                    StartCoroutine(Decals[4].Fadein());
                    Decals[4] = null;
                }
            }
        }

        if(Decals[5].FadeCount >= 1 && Decals[6].FadeCount < 1 && Input.GetKeyDown(KeyCode.Mouse1))
        {
            StartCoroutine(Decals[6].Fadein());
            StartCoroutine(Decals[7].Fadein());
        }
    }
}
```

This script has several different variables used within the script, to complete the spefic tasks of the script. These are as follows:

Vines[], **Planters**[] **and Projector** (GameObject) - These GameObject's store the objects found in the second segment of the game, where the Inventory system is introduced. This is done so tutorials appear as the parents interact with these items.

Decals[] (DecalTextAppear) - This array of Decals store the different decal text blocks which appear when the correct conditions are met.

Build (bool) - This bool is used to check if the player is in the Build or the Editor, used for editing purposes.

WhereamI, VineTake, TakeThis (VoiceActing) - These store Voice acting clips played throughout the Editor script's purposes.

This script is made from the Start() and Update() functions, with the Start() function being used to check if the player is in the Build or not. It then plays the WhereamI voice acting clip, so that it always plays at the start of the game. The Update() function is a long series of if statements, waiting to check the conditions for each decal text object to appear have been met. This can be anything from looking at the projector to collecting two of the inventory vines.

Menu:

This script is used to register the buttons within the game's Main Menu. It is so simple that I can explain it in this paragraph, without further study. The script has two public functions: begingame() and endgame() which are called by the Start and Quit buttons in the menu respectively. When begingame() is called, the main scene for the game is called, and when endgame() is called, the game quit's itself. When the player presses the escape key in gameplay they return to this menu.

```
  public class Menu : MonoBehaviour
{
   //This script is used for the buttons in the Main Menu

   //This stores the Buttons of the Main Menu
   [SerializeField] GameObject Button;
   // Start is called before the first frame update
   void Start()
   {
       Button.SetActive(false);
   }

   public void begingame()
   {
       SceneManager.LoadScene(1);
   }

   public void endgame()
   {
       Application.Quit();
   }
}
```

Character Movement and Mechanics

This section of the document will discuss the different scripts used to move the player types around, whether the first person player or the different 2D styles. It will also discuss the mechanics that the player can play with when using these styles. It will start by discussing the 2D movement of the player, in the different gameplay styles, before discussing the mechanics that these styles of gameplay can use. After this it will do the same process for the 3D mechanics and gameplay. With that, lets begin analysing it all.

DecalMovement:

The DecalMovement script is the code used to move the Atari character in the 2D space, which is the fundamental script that was later copied to program the movements of the Atari Player. This script started off relatively simple, and grew more bloated as it progressed. In the actual game version of this, I will work to refine the system so that it doesn't use so much code. Without further ado, lets have a look at the variables used for this script.

DecalMovement - Variables:

```
public class DecalMovement : MonoBehaviour
{
    //This script is the main movement script for the Atari player

    //This stores the Rigidbody of the Atari Player
    public Rigidbody BoxRigid;
    //This is a collection of floats, which store the speed of the player, the distance of the two markers and the speed of the rotation.
    public float Sped, PosDisChecks, PlayDisChecks, rotationspeed;

    public float Jumpheight;

    //This bool checks if the player is currently moving right.
    public bool MoveRight;

    //When this bool is true, it means the camera rotates in the their local positions.
    bool LocalR;

    //These GameObjects are the ones which the player moves between
    public GameObject OriginMark, TargetMark;

    //These store the CornerTrigger Scripts for the two GameObjects stored in OriginMark
    private CornerTrigger OriginCT;
    private CornerTrigger TargetCT;
    //These Vector3 variables automatically update to be the value of the positions of the OriginMark and TargetMark scripts.
    public Vector3 originPos => OriginMark.transform.position;
    public Vector3 targetPos => TargetMark.transform.position;

    //This is a debugging variable used to quickly check the value stored in originPos.
    public Vector3 OriginPosCheck;

    //These bools are used to check if the player can move or rotate. Only one can be true at a time.
    public bool isMove, isRotate;

    //These variables are used for debugging purposes, either to set the normal and location of the player, or to update what the current
normal is.
    public Vector3 SetNormal;
    public Vector3 CurrentNormal;
    public Vector3 SetLocation;

    //This float stores the lerp value at which the platform moves when rotating around corners.
    public float rotationLerp;
    //These Vectors store the start and end values for the player's corner movement, storing the start and end for the Location and Normal.
    public Vector3 MoveLTo, MoveNTo, StartLFrom, StartNFrom;
```

```
//These Quaternion store the start and ending rotation position of the player around corners.
[SerializeField]Quaternion StartRFrom, MoveRTo;

//This bool determines whether the value of the movement around corner needs to be inverted or not.
public bool invertvalue;

//This array stores the markers used to set the next markers after rotating
public GameObject[] Markers;

//These bools are used to check if the player is able to jump or move left or right.
public bool JumpAble, LeftAble, RightAble, CutsceneJump;

//These variables stores values for the camrea around rotations, and the FOV of the camera.
[SerializeField] Vector3 MoveC_LTo, StartC_LFrom;
[SerializeField]Quaternion MoveC_RTo, StartC_RFrom;
[SerializeField] float MoveC_FOVTo, StartC_FOVFrom;

//This bool is true when the camera values change after a corner rotation
[SerializeField] bool CameraChange;

//This stores the camera for the 2D section
public Camera GameCamera;

//This stores the GameObject that will be used if the corner has no script attached
[SerializeField] CornerTrigger NoCT;

//This bool stores whether the rock should be active or not
[SerializeField]bool RockActive;

//This int stores whether the player is moving or rotating
[SerializeField] int MoveorRot;

//This bool determines whether the script should check if they are rotating or moving.
[SerializeField] bool CheckMove;

//This bool checks if the player is setting a rock in the 2D space.
[SerializeField] bool SettingRock;

//This GameObject is the actual rock stored in the script.
[SerializeField] GameObject ActualRock;

//This is a group of the Rock_Trigger script, which are to check whether the player can place a rock
[SerializeField] Rock_Trigger LeftRock, RightRock, CurrentRock;

//This bool is set to true when the player is standing on the rock.
public bool RockOn;

[SerializeField] EventsCode E;

//This checks if the player's rock is inverted with selecting where to place the rock.
[SerializeField] bool Invert;

//This bool checks if the player is moving on the floor.
[SerializeField] bool FLoor;

//This bool is used to check what type of direction the rock needs to be in
[SerializeField] int RockType;

//This is used to check whether abilities are activated in AtariAbilitySave.
[SerializeField]AtariAblilitySave AAS;

//This int is used to set what direction the rock should fall in.
[SerializeField] int rockgravity;

//This stores the death trigger for the player
public DecalDeath DD;

//This stores the audiomanager for the player's movement.
[SerializeField] audiomanager AM;
```

As you can see, this script has a lot of variables (and this is after trimming down the useless ones) so let’s just jump into the analysis.

BoxRigid (Rigidbody) - This variable stores the Rigidbody of the Atari player.

Sped, PosDisChecks, PlayDisChecks, roationspeed (float) - These float variables store the speed of the player, both in normal movement and when rotating around cornrers, and calculate the distance that the two movement marks have between each other, and how the the player is from them all.

Jumpheight (float) - This float stores the height at which the player jumps.

MoveRight (bool) - This bool stores whether the player is currently moving right or not.

LocalR (bool) - This bool, when true, means that any camera changes during rotation happen in the local transform of the camera.

OriginMark and TargetMark (GameObject) - These store GameObjects that the player moves between when moving on the wall.

OriginCT and TargetCT (CornerTrigger) - If the OriginMark and TargetMark GameObjects have the CornerTrigger attached then they will be stored in these GameObjects.

originPos and targetPos (Vector3) - These Vector3 variables update to store the current positions of the OriginMark and TargetMark variables.

OriginPosCheck (Vector3) - This Vector3 is used to show the current value of originPos for debugging purposes.

isMove and isRotate (bool) - These bools are used to check the current movement state of the player, whether they are moving on the wall or rotating along the corners. These cannot be true at the same time.

SetNormal, CurrentNormal and SetLocation (Vector3) - These Vector3 variables store the location and normal the player needs to be placed at during the start of gameplay, and the checks the current normal the player is at during gameplay.

rotationLerp (float) - This float stores the current lerp that the player uses when rotating around corners.

MoveLTo, MoveNTo, MoveLFrom, Move,NFrom (Vector 3) - These Vector3 variables are used to set up the rotation of the player along the corners, checking what position and normal the player needs to start and end at are.

MoveRTo and MoveRFrom (Quaternion) - These variables store where the rotation starts and ends when rotating around a corner.

invertvalue (bool) - This bool checks whether the player's rotation will have inverted values or not.

Markers[] (GameObject) - This array of GameObjects is used to store the markers that the player swaps between when rotating around a corner.

JumpAble, LeftAble, RightAble, CutsceneJump (bool) - These bools are used to check whether the player can jump or move left and right. The Cutscene jump is used to keep the player from jumping during cutscenes.

MoveC_LTo, MoveC_LFrom, MoveC_RTo, MoveC_RFrom, MoveC_FOVTo, StartC_FOVFrom (Vector3, Quaternion and float) - These variables store values for the camera to change around corner rotations, similar to how the player stores values when it rotates. Rather then storing the normal of the camera, it stores the value of its Frame Of View.

CameraChange, GameCamera (bool, Camera) - These variables are used to setup camera rotations. If the CameraChange bool is true then it means the camera will change during the corner rotation, whilst the GameCamera variables stores the camera for the 2D area.

NoCT (CornerTrigger) - This variable is a blank CornerTrigger component on a gameObject and is called when either of the markers don't have any CornerTrigger script attached.

RockActive, MoveorRot, CheckMove, SettingRock, ActualRock (bool, int and GameObject) - These variables house three bools, RockActive, CheckMove and SettingRock, which are each used for the placing rock mechanic. RockActive is used to check if the rock has already been placed or not, CheckMove is used to check if the player is rotating or moving, which is needed to get the rock placing system working and SettingRock is true when the player is actually placing the rock down. MoveOrRot is the value which gets effected when the player is Checking their movement, being set at 0 when the player is moving and at 1 when they are rotating. Finally, ActualRock stores the rock GameObject that can get placed by the player.

LeftRock, RightRock, CurrentRock (Rock_Trigger) - These variables store instances of the Rock_Trigger component, one for the left spawning and one for the right, with CurrentRock storing the side which is currently being used out of the two.

RockOn (bool) - This bool checks to see if the player is currently standing on the rock.

E (EventsCode) - This variable stores the Eventscode Component that is within the game.

Invert and FLoor (bool) - The Invert bool is set when the values of the corner rotation need to be inverted to properly work, whilst the FLoor bool states whether the player is playing on the floor or on the wall.

RockType (int) - This variable stores what type of rock the player is using, and more specifically, what direction it is in.

AAS (AtariAbilitySave) - This variable will store the AtariAbilitySave scriptable object, to check the current values of the abilities the Atari Player can use.

rockgravity, DD and AM (int, DecalDeath and AudioManager) - The DD and AM variables are made to store the DecalDeath needed to kill the Atari Player and the AudioManager component within the general scene. rockgravity is used to determine what direction the rock can fall in.

DecalMovement - Start() and Update():

```
  private void Start()
  {
      AM = GameObject.FindGameObjectWithTag("AControl").GetComponent<audiomanager>();

      //The game starts off by having the player moving,
      and has the script check the current move type of the player for debugging purposes.
      isMove = true;
      isRotate = false;

      CheckMove = true;
 }
```

```
void Update()
{
    //When pressing Escape the game will take the player back to the main menu.
    if (Input.GetKeyDown(KeyCode.Escape))
    {
        Cursor.lockState = CursorLockMode.None;
        Cursor.visible = true;
        SceneManager.LoadScene(0);
    }

    //When pressing Q the script automatically kills the player, so they can restart the current 2D section.
    if (Input.GetKeyDown(KeyCode.Q) && DD != null)
    {
        DD.Die();
    }

    //Checks the current normal of the player.
    CurrentNormal = gameObject.transform.forward;

    //Checks if the Markers have CornerTrigger components
    //If they don't have any attached, then the CT variable will contain NoCT.
    if (OriginMark.GetComponent<CornerTrigger>() != null)
    {
        OriginCT = OriginMark.GetComponent<CornerTrigger>();
    }
    else
    {
        OriginCT = NoCT.GetComponent<CornerTrigger>();
    }
    if (TargetMark.GetComponent<CornerTrigger>() != null)
    {
        TargetCT = TargetMark.GetComponent<CornerTrigger>();
    }
    else
    {
        TargetCT = NoCT.GetComponent<CornerTrigger>();
    }

    //This checks whether the player is rotating or moving and stores it in the MoveorRot int.
    if (CheckMove == true)
    {
        if (isMove == true)
        {
            MoveorRot = 1;
        }
        else if (isRotate == true)
        {
            MoveorRot = 2;
        }
    }

    //The script calls different movement functions based off the value of FLoor
    //This either means they will be moving in the Y-axis on the floor or X-axis on the wall.
    if (FLoor == false)
    {
        RegularGravity();
    }
    else if (FLoor == true)
    {
        FloorGravity();
    }

    //This changes the rotation of the rock if it is the one that is on the floor.
    if (ActualRock.activeSelf == false)
    {
        switch (RockType)
        {
            case 1:
                {
                    ActualRock.transform.rotation = new Quaternion(0, 0.66320771f, 0, 0.748435378f);
                }
                break;
        }
    }

    //Allows the player to place the rock when clicking the left click
    if (Input.GetKeyDown(KeyCode.Mouse1) && AAS.AtariRock == true)
    {
        RockActive = !RockActive;

        //This also works to turn off the rock selection.
        if (RockActive == true)
        {
            RockSet(true);
        }
        else if (RockActive == false)
        {
            RockSet(false);
        }
    }
//Allows the player to place the rock before them.
        else if (isMove == false && isRotate == false && SettingRock == true)
        {
            //This changes the direction of the RockTrigger depending on what direction the player is pressing.
            //If the player can place a rock down the triggers will stay green, if they can't it goes red.
            if (Input.GetKeyDown(KeyCode.A))
            {
                AM.RockDirection.SetActive(true);
                AM.RockDirection.GetComponent<AudioSource>().Play();
                if (Invert == false)
                {
                    LeftRock.gameObject.SetActive(true);
                    RightRock.gameObject.SetActive(false);
                    CurrentRock = LeftRock;
                }
                else
                {
                    LeftRock.gameObject.SetActive(false);
                    RightRock.gameObject.SetActive(true);
                    CurrentRock = RightRock;
                }
            }
            else if (Input.GetKeyDown(KeyCode.D))
            {
                AM.RockDirection.SetActive(true);
                AM.RockDirection.GetComponent<AudioSource>().Play();
                if (Invert == false)
                {
                    LeftRock.gameObject.SetActive(false);
                    RightRock.gameObject.SetActive(true);
                    CurrentRock = RightRock;
                }
                else
                {
                    LeftRock.gameObject.SetActive(true);
                    RightRock.gameObject.SetActive(false);
                    CurrentRock = LeftRock;
                }
            }

            //If the player can place the rock and places enter then the rock ability will be placed.
            if (CurrentRock.CanPlace == true)
            {
                if (Input.GetKeyDown(KeyCode.Mouse0))
                {
                    RockFreeze();
                    AM.RockPlace.SetActive(true);
                    //When placing the rock the script resets all it's Rigidbody values so that it doesn't cause glitches.
                    ActualRock.transform.position = CurrentRock.gameObject.transform.position;
                    ActualRock.GetComponent<Rigidbody>().velocity = Vector3.zero;
                    if (RockType == 0)
                    {
                        ActualRock.transform.rotation = CurrentRock.gameObject.transform.rotation;
                    }
                    ActualRock.SetActive(true);
                    LeftRock.gameObject.SetActive(false);
                    RightRock.gameObject.SetActive(false);
                    RockSet(false);
                }
            }
        }
    }

    //The FixedUpdate calls the gravity calculations for the character
    //So that it stays consistent between the Editor and Build.
    private void FixedUpdate()
    {
        BoxRigid.AddForce(Physics.gravity * BoxRigid.mass, ForceMode.Acceleration);
    }
```

The Start() function works to set up the movement of the Player, having them be set to move along the wall, and to Check the current movement for debugging reasons. THe Update() function is filled with a bunch of if statements, which do different things. THe first two things checked are whether the player is pressing the escape or Q key, with the escape key bringing them back to the main menu, and the Q key killing the 2D player so they quickly go back to the 3D space.

Next, the script checks the current normal of the script, and if the OriginMark and TargetMark GameObjects have CornerTrigger components, if either doesn’t then the NoCT variable will be set in its place. After this, the script checks the current move type of the player for debugging purposes, if the CheckMove bool is true.

After this, the script will call the movement of the player character, based on the value of FLoor. If FLoor is false then it means the player is moving on the Walls, and as such the RegularGravity() function is called. If it is true then the player is moving on the Floors and the FloorGravity() function is called. Both complete the same purpose, just making them dependent on the X or Y axis respectively.

The rest of the Update() function is reserved for the Rock placing mechanic, starting by changing the rotation of the gravity based on the value of the RockType. The script then checks if the player is pressing the Right Click, and will call the RockSet() function based on this, either working to stop or start the rock setting mechanic. Once the RockSetting mechanic period is in place, the script will display either LeftRock or RightRock based on what key the player is pressing, which will check if the rock is able to placed. The script will then set CurrentRock as whichever Rock_Trigger object is active, and if the player presses the Enter key whilst the rock is able to be placed, then the 2D Rock will be set in that position and the Rock setting will end. When placed the rock will reset all it's velocity, to stop any glitches, and the FreezeRock() function will be called to check in what direction the rock should fall in.

Meanwhile, the FixedUpdate() function only has one purpose, to add Gravity to the player, so that the jump isn't too floaty. This is done in the FixedUpdate() so that the gravity is consistent between the Editor and Build.

DecalMovement - RegularGravity():

```
public void RegularGravity()
{
    //Calculates the players distance from the markers they are moving towards.
    OriginPosCheck = originPos;
    PosDisChecks = Vector3.Distance(originPos, targetPos);
    PlayDisChecks = Vector3.Distance(new Vector3(gameObject.transform.position.x, originPos.y, originPos.z),
    originPos);
    //Stores the input the player makes in the horizontal axis.
    float axis = Input.GetAxis("Horizontal");

    //Checks if the player is moving left or right based on the value of axis.
    MoveRight = axis > 0 ? true : false;

    //If the player presses whilst on the ground they will jump.
    if (Input.GetKeyDown(KeyCode.Space) && JumpAble == true && SettingRock == false && CutsceneJump == false)
    {
        AM.Jump.SetActive(true);
        JumpAble = false;
        BoxRigid.AddForce(Vector3.up * Jumpheight, ForceMode.Acceleration);

        //If the player is standing on a rock it will despawn.
        if (RockOn == true)
        {
            ActualRock.SetActive(false);
            RockOn = false;
        }
    }

    //If the player can move then they will move between the two markers on the wall.
    if (isMove == true && isRotate == false)
    {
        Vector3 UpdateTarget = new Vector3(targetPos.x, gameObject.transform.position.y, targetPos.z);
        Vector3 UpdateOrigin = new Vector3(originPos.x, gameObject.transform.position.y, originPos.z);
```

```
            MoveCall(axis, UpdateTarget, UpdateOrigin);

            //Calculates the distance the player is at from the corners, and if close enough causes the player to
rotate around the corner.
            if (Vector3.Distance(gameObject.transform.position, UpdateOrigin) > (Vector3.Distance(UpdateOrigin, UpdateTarget) - TargetCT.TurnCheck[1]))
            {
                if (TargetMark.GetComponent<CornerTrigger>() != null)
                {
                    TargetMark.GetComponent<CornerTrigger>().SetupRotate();
                }
            }
            else if (Vector3.Distance(gameObject.transform.position, UpdateOrigin) < OriginCT.TurnCheck[0])
            {
                if (OriginMark.GetComponent<CornerTrigger>() != null)
                {
                    OriginMark.GetComponent<CornerTrigger>().SetupRotate();
                }
            }
        }
        //If the player is rotatating will check if they can move left or right, before rotating around the corner.
        else if (isMove == false && isRotate == true)
        {
            if (LeftAble == true && RightAble == true)
            {
                CornerRotation(axis);
            }
            else if (LeftAble == false && RightAble == true)
            {
                if (MoveRight == true)
                {
                    CornerRotation(axis);
                }
            }
            else if (LeftAble == true && RightAble == false)
            {
                if (MoveRight == false)
                {
                    CornerRotation(axis);
                }
            }
        }
    }
```

The RegularGravity() function works to move the player along the X axis of the walls, but is practically identical to the FloorGravity() function, expect that function moves the player in the Y axis along the floor. Because of this, I am only going to describe and show one of these functions as showing more would be obsolete.

This function starts by registering the distance between originPos and targetPos, which is saved in the PosDisChecks. Then the game checks the distance between the player in the x and z axis compared to originPos. These are used to calculate when the player should rotate around corners. The function then sets float called axis, which stores the Input of the player in the Horizontal axis, which is used to set the value of MoveRight, to check if the player is moving left or right.

If the player presses space, whilst the Atari Player is able to jump, then the character will jump up. If the player is standing on the rock at the time, then the rock will deactivate. Finally, we get to the big bit, the movement of the player along the walls. If isMove is true, and isRotate is false, then that means the player needs to move between originPos and targetPos. This is done by creating two new Vector3 variables, UpdateTarget and UpdateOrigin, which store the values of targetPos and originPos, with the Y axis removed, so that the player can jump freely without it affecting their movement between the two spaces.

After doing this, the script calls the MoveCall() function, which is used to actually move the player. It then checks the distance of the player compared to the two target points, and will set up the rotation based on this value. Once the player reaches near either of the TargetMark or OriginMark, then the script will check to see if the objects have the CornerTrigger component, and if this is true, then it calls the CornerTrigger's SetupRotate() function. After this, isRotate should be true rather then isMove, and this causes the script to call the CornerRotation() function.

DecalMovement - MoveCall(), SetupRotate() and SetUpCameraChange():

```
void MoveCall(float axis, Vector3 UpdateTarget, Vector3 UpdateOrigin)
{
    //Checks if the player can move left or right
    if (LeftAble == true && RightAble == true)
    {
        gameObject.transform.position = Vector3.MoveTowards(gameObject.transform.position, MoveRight ? UpdateTarget : UpdateOrigin, Mathf.Abs(axis)
    * Time.deltaTime * Sped);
    }
    else if (LeftAble == false && RightAble == true)
    {
        if (MoveRight == true)
        {
            gameObject.transform.position = Vector3.MoveTowards(gameObject.transform.position, MoveRight ? UpdateTarget : UpdateOrigin,
            Mathf.Abs(axis) * Time.deltaTime * Sped);
        }
    }
    else if (LeftAble == true && RightAble == false)
    {
        if (MoveRight == false)
        {
            gameObject.transform.position = Vector3.MoveTowards(gameObject.transform.position, MoveRight ? UpdateTarget : UpdateOrigin,
            Mathf.Abs(axis) * Time.deltaTime * Sped);
        }
    }
}

//Sets the values of the variables used in corner rotation calculation.
public void SetUpRotate(Vector3[] L, Vector3[] N,Quaternion[] Q, GameObject[] M)
{

    float OriginDis = Vector3.Distance(gameObject.transform.position, L[0]);
    float TargetDis = Vector3.Distance(gameObject.transform.position, L[1]);

    bool NearOrigin = OriginDis < TargetDis ? true : false;

    isMove = false;
    isRotate = true;
    rotationLerp = .001f;

    MoveLTo = NearOrigin ? L[1] : L[0];
    MoveNTo = NearOrigin ? N[1] : N[0];

    StartLFrom = NearOrigin ? L[0] : L[1];
    StartNFrom = NearOrigin ? N[0] : N[1];

    MoveRTo = NearOrigin ? Q[1] : Q[0];
    StartRFrom = NearOrigin ? Q[0] : Q[1];

    Markers[0] = M[0];
    Markers[1] = M[1];
    Markers[2] = M[2];

    invertvalue = NearOrigin ? true : false;

}

//If the camera needs to move will set up the variables that will be used.
public void SetUpCameraChange(Vector3[] L, Quaternion[] Q, float[]FOV, bool Localr)
{
    MoveC_LTo = MoveRight ? L[1] : L[0];
    StartC_LFrom = MoveRight ? L[0] : L[1];

    MoveC_RTo = MoveRight ? Q[1] : Q[0];
    StartC_RFrom = MoveRight ? Q[0] : Q[1];

    MoveC_FOVTo = MoveRight ? FOV[1] : FOV[0];
    StartC_FOVFrom = MoveRight ? FOV[0] : FOV[1];

    LocalR = Localr;

    CameraChange = true;
}
```

The MoveCall() function is called by the RegularGravity() and FloorGravity() functions to move the player along the floor of the scene. It does this using a MoveTowards function, moving between UpdateTarget and UpdateOrigin, and the value of axis, which are all set when the function is called. The function won't move the player in the left or right direction is LeftAble or RightAble is false, as that means they have hit a collision. The movement is done this way so that the player doesn't clip through the walls, and stays on track with where they are meant to go.

SetUpRotate(), as the name implies, sets up the player for rotating around the scene's corners. This is called by whatever CornerTrigger the player is close to, and sets the start and end of the rotation based on if the player is near their current OriginPoint. This works to provide the script with all information necessary to effectively rotate, and helps automate the rotation process. If the player is moving left whilst rotating, then invert is set to true, as their movements need to be inverted in order to properly move in that direction. The SetUpCameraChange function does the exact same thing, just for the rotation of the camera.

DecalMovement - CornerRotation() and CameraChangeFunction():

```
void MoveCall(float axis, Vector3 UpdateTarget, Vector3 UpdateOrigin)
{
    //Checks if the player can move left or right
    if (LeftAble == true && RightAble == true)
    {
        gameObject.transform.position = Vector3.MoveTowards(gameObject.transform.position, MoveRight ? UpdateTarget : UpdateOrigin, Mathf.Abs(axis) * Time.deltaTime * Sped);
    }
    else if (LeftAble == false && RightAble == true)
    {
        if (MoveRight == true)
        {
            gameObject.transform.position = Vector3.MoveTowards(gameObject.transform.position, MoveRight ? UpdateTarget : UpdateOrigin, Mathf.Abs(axis) * Time.deltaTime * Sped);
        }
    }
    else if (LeftAble == true && RightAble == false)
    {
        if (MoveRight == false)
        {
            gameObject.transform.position = Vector3.MoveTowards(gameObject.transform.position, MoveRight ? UpdateTarget : UpdateOrigin, Mathf.Abs(axis) * Time.deltaTime * Sped);
        }
    }
}

//Sets the values of the variables used in corner rotation calculation.
public void SetUpRotate(Vector3[] L, Vector3[] N,Quaternion[] Q, GameObject[] M)
{

    float OriginDis = Vector3.Distance(gameObject.transform.position, L[0]);
    float TargetDis = Vector3.Distance(gameObject.transform.position, L[1]);

    bool NearOrigin = OriginDis < TargetDis ? true : false;

    isMove = false;
    isRotate = true;
    rotationLerp = .001f;

    MoveLTo = NearOrigin ? L[1] : L[0];
    MoveNTo = NearOrigin ? N[1] : N[0];

    StartLFrom = NearOrigin ? L[0] : L[1];
    StartNFrom = NearOrigin ? N[0] : N[1];

    MoveRTo = NearOrigin ? Q[1] : Q[0];
    StartRFrom = NearOrigin ? Q[0] : Q[1];

    Markers[0] = M[0];
    Markers[1] = M[1];
    Markers[2] = M[2];

    invertvalue = NearOrigin ? true : false;

}

//If the camera needs to move will set up the variables that will be used.
public void SetUpCameraChange(Vector3[] L, Quaternion[] Q, float[]FOV, bool Localr)
{
    MoveC_LTo = MoveRight ? L[1] : L[0];
    StartC_LFrom = MoveRight ? L[0] : L[1];

    MoveC_RTo = MoveRight ? Q[1] : Q[0];
    StartC_RFrom = MoveRight ? Q[0] : Q[1];

    MoveC_FOVTo = MoveRight ? FOV[1] : FOV[0];
    StartC_FOVFrom = MoveRight ? FOV[0] : FOV[1];

    LocalR = Localr;

    CameraChange = true;
}
```

The CornerRotation() function is used to move the player along the corners of the 2D area.
This only does this for the RegularGravity() function, as the FloorGravity() function does this via a different function called CornerRotationY(), which is the same, save the fact that it used to rotate along the Y axis. This function works by using a lerp for the position of the Player's Position, Rotation and Normal. The value of the lerp is modified by the value of the axis variable, and as such allows the player to move left and right with ease.

Once the player reaches the end of the rotation, either when RotationLerp equals 0 or 1, then the script will set up the player to be able to move along the next wall, setting new values for the Target and Origin Markers. Once this is done it sets isMove back to true, and isRotate to false, before resetting every value set in the SetUpRotate() function. If the camera needs to change during the rotation, then the CameraChangeFunction() is called, which literally does the same thing, just with the camera's saved settings.

DecalMovement - RockFreeze(), SetStart(), LetMove() and RockSet():

```
    //This function sets in what direction the rock should be affected with gravity for, based on the values of rockgravity.
    void RockFreeze()
    {
        switch(rockgravity)
        {
            case 0:
                {
                    ActualRock.gameObject.GetComponent<Rigidbody>().constraints = RigidbodyConstraints.None |
RigidbodyConstraints.FreezeRotation | RigidbodyConstraints.FreezePositionX | RigidbodyConstraints.FreezePositionZ;
                }
                break;
            case 1:
                {
                    ActualRock.gameObject.GetComponent<Rigidbody>().constraints = RigidbodyConstraints.None |
RigidbodyConstraints.FreezeRotation | RigidbodyConstraints.FreezePositionY | RigidbodyConstraints.FreezePositionZ;
                }
                break;
        }
    }

    //This function determines where the 2D player should start, and is called by the GenreSwap script.
    public void SetStart(bool newpos, Vector3 Newnormal, Vector3 NewPos)
    {
        if(newpos == false)
        {
            gameObject.transform.forward = SetNormal;
            gameObject.transform.position = SetLocation;
        }
        else if (newpos == true)
        {
            gameObject.transform.forward = Newnormal;
            gameObject.transform.position = NewPos;
        }
    }

    //Used to set the value of the 2D player's movement
    public void LetMove(bool can)
    {
        if(can == true)
        {
            isMove = true;
            isRotate = false;
        }
        else if(can == false)
        {
            isMove = false;
            isRotate = false;
        }
    }
```

```
//Called to set up the rock ability or deactivate it.
void RockSet(bool Set)
{
    if(Set == true)
    {
        isMove = false;
        isRotate = false;
        CheckMove = false;
        SettingRock = true;
        if(MoveRight == true)
        {
            LeftRock.gameObject.SetActive(true);
            CurrentRock = LeftRock;
        }
        else if(MoveRight == false)
        {
            RightRock.gameObject.SetActive(true);
            CurrentRock = RightRock;
        }
    }
    else if(Set == false)
    {
        SettingRock = false;
        switch (MoveorRot)
        {
            case 1:
                {
                    isMove = true;
                }
                break;
            case 2:
                {
                    isRotate = true;
                }
                break;
        }


        LeftRock.gameObject.SetActive(false);
        LeftRock.CanPlace = true;
        RightRock.gameObject.SetActive(false);
        RightRock.CanPlace = true;
        CurrentRock = null;
        CheckMove = true;
    }
}
```

The remaining functions within this script are all quite small and simple, being used to help generate the mechanics of the game. The first, RockFreeze(), decides what direction the rock should have the affects of gravity on, based on the value of rockgravity. The next function, SetStart() is called by the GenreChange script, and sets the position that the player will be starting at, either with a position set by the GenreChange instance or based on the values set in SetLocation and SetNormal.

The LetMove() function is used to quickly change the values of isMove and isRotate, which can easily be called. The final function within the script is RockSet() which is used to set up the rock placing ability. In this function it will either set up the rock setting, by stopping the player from being able to move before turning SettingRock to true, or it will remove the options, by putting the player back either moving or rotating (based off the value of MoveorRot) and hiding all the Rock_Trigger gameObjects.

Mega_Man_Movement:

The Mega_Man_Movement script is an alternative script used for the Mega Man player in the 2D space. This script uses the same fundamentals established within the DecalMovement script, so as such would be just repeating myself for most of the script. Instead, I will take the time to point out what is new within the script, compared to it's DecalMovement counterpart. So let's start by looking at the new variables:

Mega_Man_Movement - New Variables:

```
public class Mega_Man_Movement : MonoBehaviour
{
//This bool is used to check if the player is shooting regular or evil pollen
    public bool pollenbad;

   //This array stores the sprites of the player as decals
    [SerializeField] GameObject[] Decals;

    //This stores the type of pollen to be shot
    [SerializeField] GameObject[] Pollen;

   //This array stores the spots that the pollen will spawn from.
  //Spawn  stores whichever is currently in use.
    [SerializeField] GameObject[] BulletSpawn;
    [SerializeField] GameObject Spawn;

  //This bool checks if the pollen should be moving left or right.
    private bool PollenMoveRight;

  //This stores the instance of pollen shot in order to add the appropriate values.
    private GameObject TempPollen;

   //This stores the sprites used for the player, swapping between walking and shooting pollen
    [SerializeField] GameObject[] LeftSprites;
    [SerializeField] GameObject[] RightSprites;

    //These variables are used to create a delay between when the player can shoot pollen.
    [SerializeField] bool CanShootPollen;
    [SerializeField] float Pollentimer;

   //This stores the character's animator, so that they can animate the player's movement.
    [SerializeField] Animator Anim;

   //This bool can stop the player from being able to shoot pollen when needed.
    public bool CanShoot;
```

The Mega_Man_Movement script houses almost all the variables that the DecalMovement script used, with the exception of any to do with jumping or placing rocks, as this gameplay style can do neither of those. What the Mega Man player can do however, is shoot pollen bullets. As such, most of the new variables exclusive to this script have to do with that ability, with some also doing with the sprites of the player, as the Mega Man character actually has art associated with them. The new variables are as follows:

pollenbad (bool) - This bool stores whether the player is shooting evil or regular pollen. For context evil pollen is the name given to the type of pollen which shrinks spikes.

Decals[] (GameObject) - This array of GameObjects stores the decal projectors for the player's left and right sprites.

Pollen[] (GameObject) - This array of GameObjects stores the pollen prefabs used to shoot the pollen.

BulletSpawn[] **and Spawn** (GameObject) - The BulletSpawn array stores the different sides that pollen can be shot from, with Spawn storing the currently active spawn point out of those stored in BulletSpawn.

PollenMoveRight (bool) - This bool registers whether the pollen should be moving right or left.

TempPollen (GameObject) - This GameObject stores the current instance of pollen shot.

LeftSprites[] **and RightSprites**[] (GameObject) - These array of GameObjects store the sprites for the player during regular movement and shooting in either direction.

CanShootPollen and Pollentimer (bool, float) - These variables are used to create a cooldown between when the player can shoot pollen, so they don't spam. When CanShootPollen is true then the code uses Pollentimer to wait a few seconds before the player can shoot again.

Anim (Animator) - This stores the Animator used for the Mega Man player's sprites, so that it creates the moving animation.

CanShoot (bool) - This bool can be called by other scripts to stop the player from being able to shoot pollen.

Mega_Man_Movement - Start(), Update(), DirectionMove() and FixedUpdate():

```
private void Start()
{
    Anim.speed = 0;
    isMove = true;
    isRotate = false;
    if (debug == true)
    {
        if (CheckNormal == true)
        {
            SetNormal = gameObject.transform.forward;
        }
        else if (CheckNormal == false)
        {
            SetStart(false, Vector3.zero, Vector3.zero);
        }
    }
}
```

```
//This script functions practically the same as the Decal_Movement script, with some small changes
   void Update()
   {

       if (Input.GetKeyDown(KeyCode.Escape))
       {
           Cursor.lockState = CursorLockMode.None;
           Cursor.visible = true;
           SceneManager.LoadScene(0);
       }

       CurrentNormal = gameObject.transform.forward;
       if (OriginMark.GetComponent<CornerTrigger>() != null)
       {
           OriginCT = OriginMark.GetComponent<CornerTrigger>();
       }
       else
       {
           OriginCT = NoCT.GetComponent<CornerTrigger>();
       }
       if (TargetMark.GetComponent<CornerTrigger>() != null)
       {
           TargetCT = TargetMark.GetComponent<CornerTrigger>();
       }
       else
       {
           TargetCT = NoCT.GetComponent<CornerTrigger>();
       }


       OriginPosCheck = originPos;
       PosDisChecks = Vector3.Distance(originPos, targetPos);
       PlayDisChecks = Vector3.Distance(gameObject.transform.position, originPos);
       float axis = Input.GetAxis("Horizontal");

       if(Input.GetKeyDown(KeyCode.Q))
       {
           DD.Die();
       }

       MoveRight = axis > 0 ? true : false;

       if (axis != 0)
       {
           Decals[0].SetActive(MoveRight);
           Decals[1].SetActive(!MoveRight);

           switch (MoveRight)
           {
               case true:
                   {
                       Anim = RightSprites[0].GetComponent<Animator>();
                       Spawn = BulletSpawn[0];
                       PollenMoveRight = true;
                   }
                   break;
               case false:
                   {
                       Anim = LeftSprites[0].GetComponent<Animator>();
                       Spawn = BulletSpawn[1];
                       PollenMoveRight = false;
                   }
                   break;
           }
       }

       if (axis != 0)
       {
           Anim.speed = 0.5f;
       }
       else
       {
           Anim.speed = 0;
       }
       bool RightorLeft = Decals[0].activeSelf;

       if(Input.GetKeyDown(KeyCode.Mouse1))
       {
           pollenbad = !(pollenbad);
       }

       //When pressing space, rather then jumping the player will shoot a bullet in whatever direction they're facing
       if (Input.GetKeyDown(KeyCode.Mouse0) && CanShoot == true)
       {
           ShootPollen(RightorLeft, pollenbad);
       }
       if (isMove == true && isRotate == false)
       {
           Vector3 UpdateTarget = new Vector3(targetPos.x, gameObject.transform.position.y, targetPos.z);
           Vector3 UpdateOrigin = new Vector3(originPos.x, gameObject.transform.position.y, originPos.z);
           if (LeftAble == true && RightAble == true)
           {
               gameObject.transform.position = DirectionMove(axis, UpdateTarget, UpdateOrigin);
}
        else if (LeftAble == false && RightAble == true)
        {
            if (MoveRight == true)
            {
                gameObject.transform.position = DirectionMove(axis, UpdateTarget, UpdateOrigin);
            }
        }
        else if (LeftAble == true && RightAble == false)
        {
            if (MoveRight == false)
            {
                gameObject.transform.position = DirectionMove(axis, UpdateTarget, UpdateOrigin);
            }
        }

        if (Vector3.Distance(gameObject.transform.position, UpdateOrigin) > (Vector3.Distance(UpdateOrigin, UpdateTarget) - TargetCT.TurnCheck[1]))
        {
            if (TargetMark.GetComponent<CornerTrigger>() != null)
            {
                TargetMark.GetComponent<CornerTrigger>().SetupRotate();
            }
        }
        else if (Vector3.Distance(gameObject.transform.position, UpdateOrigin) < OriginCT.TurnCheck[0])
        {
            if (OriginMark.GetComponent<CornerTrigger>() != null)
            {
                OriginMark.GetComponent<CornerTrigger>().SetupRotate();
            }
        }
    }
    else if (isMove == false && isRotate == true)
    {
        if (LeftAble == true && RightAble == true)
        {
            CornerRotation(axis);
        }
        else if (LeftAble == false && RightAble == true)
        {
            if (MoveRight == true)
            {
                CornerRotation(axis);
            }
        }
        else if (LeftAble == true && RightAble == false)
        {
            if (MoveRight == false)
            {
                CornerRotation(axis);
            }
        }

    }
}

private Vector3 DirectionMove(float axis, Vector3 UpdateTarget, Vector3 UpdateOrigin)
{
    return Vector3.MoveTowards(gameObject.transform.position, MoveRight ? UpdateTarget : UpdateOrigin, Mathf.Abs(axis) * Time.deltaTime * Sped);
}

private void FixedUpdate()
{
    BoxRigid.AddForce(Physics.gravity * BoxRigid.mass, ForceMode.Acceleration);

    if (CanShootPollen == false)
    {
        Pollentimer += 0.1f;
        if(Pollentimer >= 2)
        {
            CanShootPollen = true;
            Pollentimer = 0;
        }
    }
}
```

The Start() function for this script is practically the same as that of DecalMovement, with the only difference being that the function sets the speed of the animation stored in Anim to 0, so that the player isn't animating itself during movement. The Update() function is where things kinda change. Since the Mega Man player is only ever placed on walls, it means that there is only one axis that it can move on, so as such, the movement script is all placed within the Update() function. This movement script is almost identical to that of the DecalMovement script, with the only difference being that the player's movement is set by calling a new function called DirectionMove(). Apart from that, the movement is pretty similar.

The first big difference in the Update() function is when the script checks the value of the axis variable to determine which sprite should be displayed. This is done to make sure the player is faced in the right direction during gameplay, and that the spawn points for the bullets are in the right direction. With this the script also sets the currently showing sprites to be that stored in Anim, which gets a speed of 0.5 whenever the player presses the movement inputs. The reason the player's animation is handled this way is because it's not possible to change the materials of a decal projector in an animation, but it is possible to activate and deactivate projectors in an animation. Because of this, every sprite for the player has to be its own projector.

The final new additions to the Update() function for the Mega Man player is the set up for the Pollen shooting. When pressing the right click button, the game will swap what type of pollen the player will shoot out, inverting the value of the bool upon clicking. When pressing the left click, the function calls ShootPollen function, storing the values for PollenMoveRight and pollenbad into the function, which is used for the calculations of the function. The FixedUpdate(), like with the Atari player, is used to add gravity to the player, but it also used to calculate the time spent waiting when cooling down from shooting pollen, so that the player can't spam, and so the speed is consistent between Editor and Build.

Mega_Man_Movement - PollenAnim() and ShootPollen():

```
IEnumerator PollenAnim(bool MRight)
{
    //This function is used to swap sprites to the shooting sprite when the player shoots pollen.
    //This bases it off the direction is placed in.
    switch(MRight)
    {
        case true:
            {
                RightSprites[0].SetActive(false);
                RightSprites[1].SetActive(true);
                yield return new WaitForSecondsRealtime(0.5f);
                RightSprites[0].SetActive(true);
                RightSprites[1].SetActive(false);

            }
            break;
        case false:
            {
                LeftSprites[0].SetActive(false);
                LeftSprites[1].SetActive(true);
                yield return new WaitForSecondsRealtime(0.5f);
                LeftSprites[0].SetActive(true);
                LeftSprites[1].SetActive(false);
            }
            break;
    }
}

public void ShootPollen(bool RightorLeft, bool PollenBad)
{
    //This script spawns an instance of the pollen to be shot across the stage.
    int pollennum;
    switch (PollenBad)
    {
        case false:
            {
                pollennum = 0;
            }
            break;
        case true:
            {
                pollennum = 1;
            }
            break;
    }
    //The type of pollen shot is based off the value of PollenBad
    switch (CanShootPollen)
    {
        case true:
            {
                //When shooting pollen the function calls PollenAnim to change sprites.
                CanShootPollen = false;
                StartCoroutine(PollenAnim(RightorLeft));
                //Pollen is spawned by instantiating one of the pollen prefabs, and stores the necessary values into the script.
                TempPollen = Instantiate(Pollen[pollennum], Spawn.transform.position, Spawn.transform.rotation);
                TempPollen.GetComponent<Pollen>().MoveRight = PollenMoveRight;
                TempPollen.GetComponent<Pollen>().OriginMark = OriginMark;
                TempPollen.GetComponent<Pollen>().TargetMark = TargetMark;
                TempPollen = null;
            }
            break;
    }
}
```

The PollenAnim() IEnumerator is used to swap the sprite of the player after they shoot pollen, and is called within the ShootPollen() function. This IEnumerator swaps the sprites of the currently displayed decal projector over to the one for the shot sprite, before swapping the sprites back over after a few seconds. The ShootPollen() function is used to, well, shoot the Pollen Bullets. It first checks the type of pollen that needs to be shot, and then instantiates a version of that prefab at the necessary Spawn Point. It then gets everything working by setting the instantiated bullet as TempPollen, and stores the needed values for the calculations found within the Pollen script, before making the value of TempPollen null. This then starts the cooldown for the bullet, which works to stop the player from spamming.

Zelda_Movement:

This script is the final of the three 2D Gameplay styles, but never made it past the prototype stage of development. I plan for this to be in the full release of the game, but for now it is an unused script. Since it's unused, I will summerise it all under all the new elements found in the complete script, as it reuses aspects from DecalMovement again. The main difference is the player can move in the X and Y axis at the same time, meaning it also needs two movement and rotation functions for this, like with the Decal Player. These allow the player to move in both directions at the same time, and aren't as limited as the movement of the Decal Player. The only other new element is that the player is able to swing their sword in the space, which can destroy objects. Anyway, time for a quick look at the script.

Zelda_Movement - Script:

```
public class Zelda_Movement : MonoBehaviour
{
    //This script is the for Zelda Gameplay style, which never made it past the prototyping stages of
    development.
    //This will want to be set in a full game, but is currently unneeded, as such, there will not be many notes.

    //This checks if the player is moving and rotating the X and Y axis.
    public bool isMoveX, isRotateX;
    public bool isMoveY, isRotateY;

    //This stores the decals of the player
    [SerializeField] GameObject[] Decals;

    //This is true when the player swings their sword.
    public bool Sword;

    //This is called when the player has finished swinging their sword.
    private bool SwungSword;

    //This IEnumerator works to swipe the players sword before returning to normal
    IEnumerator SwordSwipe()
    {
        Decals[0].SetActive(false);
        Decals[1].SetActive(true);
        Sword = true;
        yield return new WaitForSeconds(0.5f);
        Sword = false;
        Decals[0].SetActive(true);
        Decals[1].SetActive(false);
        SwungSword = false;
    }

}
```

The Zelda_Movement script only has a few new variables, which are as follows:

isMoveX, isRotateX, isMoveY and isRotateY (bool) - These bools are used for the movement and rotation calculations, with one set for each direction the player can move in, so that they can move in both directions at the same time.

Decals[] (GameObject) - This variable array stores the projectors for the different sprites for the Zelda Player, with the sprites only being no movement and swinging sword.

Sword and SwungSword (bool) - These bools are used to check if the player has swung their sword or not, and are used for the swining sword process.

The only new function in this script, as all the others use elements seen in either DecalMovement or Mega_Man_Movement, is the IEnumerator SwordSwipe() which is used to swing the players sword. This is called in the movement functions when the player presses space. This IEnumerator just completes the process of changing the sprites of the decal and setting Sword to true, before waiting a few seconds and putting everything back to normal. The function sets SwungSword to true when called so the player can't spam the swing, and this gets set back to false when the function is complete.

Rock_Trigger, AddGravity and RockWallCheck:

With the 2D gameplay styles discussed, I will now take the time to discuss the different scripts used to set up the differing mechanics in the 2D space. The first of these are to do with the set up for the 2D rock scripts. The scripts made for this are as follows: Rock_Trigger, which is used for the Rock_Trigger variables in the DecalMovement script which check if the player has space to place a rock; The second is AddGravity, a system setting first created for the means of adding gravity to the rock, this script also has several things that only effect specific objects, such as the rock; Finally, we have RockWallCheck, which is used to have the rock use proper collision when hitting a wall whilst on a moving platform, as that was a glitch the player faced. I will now start explaining each of these scripts in turn to show the reasons for their design.

Rock_Trigger - Script:

```
public class Rock_Trigger : MonoBehaviour
{
    //This script creates triggers which check whether the 2D Rock can be placed in a spot or not.

    //This stores the decal projector of the player
    [SerializeField]DecalProjector DP;
    //This is the material the DecalProjector swaps to when it can be placed
    [SerializeField]Material Good;
    //This is the material the DecalProjetor swaps to when it can't be placed
    [SerializeField]Material Bad;

    //CanPlace is the bool states when the player can place the rock or not.
    public bool CanPlace = true;
// Update is called once per frame
    void Update()
    {
        //The material of the projector is effected by the value of CanPlace
        if(CanPlace == true)
        {
            DP.material = Good;
        }
        else if(CanPlace == false)
        {
            DP.material = Bad;
        }
    }

    //This script is used on the triggers that spawn the rock object, checking whether the rock can be placed.
    private void OnTriggerStay(Collider other)
    {
        if (other.gameObject.tag == "Floor" || other.gameObject.tag == "MoveFloor" || other.gameObject.tag == "Wall")
        {
            CanPlace = false;
        }
    }

    private void OnTriggerExit(Collider other)
    {
        if (other.gameObject.tag == "Floor" || other.gameObject.tag == "MoveFloor" || other.gameObject.tag == "Wall" )
        {
            DP.material = Good;
            CanPlace = true;
        }
    }
}
```

The Rock_Trigger script only has the simple purpose of adding the triggers which check whether the rock can be placed or not. There are two of these attached to the player, one to the left of the player and one to the right. Since the purpose of this script is simple, the script only has a few variables, which are as follows:

DP (DecalProjector) - This stores the projector for the Rock_Trigger object.

Good and Bad (Material) - These Material variables store the different materials the Rock_Trigger script swaps between when checking the trigger, with Good being a green colour and Bad being Red.

CanPlace (bool) - This bool is checked by the DecalMovement script when trying to place the rock, and determines whether the rock has space to be placed.

The material of DP is changed to one of Good or Bad based on the value of CanPlace, with the script setting the materials in the Update() function. The value of CanPlace is changed when something enters the trigger of the instance, or when something exits it. CanPlace is automatically set as true, because then it stops any glitches where the game no longer allows the player to place rocks if their is a collider glitch.

AddGravity - Script:

```
public class AddGravity : MonoBehaviour
{
    //This script is used to add gravity to objects in the game.
    //It has extra functions for when it attached to a 2D Rock

    //This stores the Rigidbody of the object
    [SerializeField] Rigidbody BoxRigid;

    //This stores the EventsCode script
    [SerializeField] EventsCode E;

    //This stores the current position of the rock in the x and z axis
    [SerializeField] float x;
    [SerializeField] float z;
//This checks when the Rock has collided with a wall
    [SerializeField] bool WallHit;

    //When this is true it means this script will also be used to calculate collisions
    [SerializeField] bool CollideCalc;

    //This stores the DecalArea of the current instance of the rock
    [SerializeField] GameObject DecalArea;

    // Update is called once per frame
    void Update()
    {
        //The update function is used to keep the player in the x and z axis positions stored in x and z when it collides with a wall.
        if(WallHit == true && CollideCalc == false)
        {
            gameObject.transform.position = new Vector3(x, gameObject.transform.position.y, z);
        }

    }


    //The Gravity is called in the FixedUpdate to keep it consistent.
    private void FixedUpdate()
    {
        BoxRigid.AddForce(Physics.gravity * BoxRigid.mass, ForceMode.Acceleration);
    }


    //The Collision for this is only used for the 2D rocks.
    private void OnCollisionEnter(Collision collision)
    {
        if(CollideCalc == false)
        {
            //It checks if the rock has collided with a rock, and if it has then it freezes the position of the rock in the x and y axis.
            if (collision.gameObject.tag == "Wall")
            {
                x = gameObject.transform.position.x;
                z = gameObject.transform.position.z;
                WallHit = true;
            }

            //When colliding with the floor the rock no longer is parented to a moveplatform, and instead gets parented to the decalarea
            if (collision.gameObject.tag == "Floor")
            {
                WallHit = false;
                gameObject.transform.parent = DecalArea.transform.parent;
            }
        }
    }
}
```

The AddGravity script is attached to objects in the scene which need to be affected by a consistent level of gravity. This is done in the FixedUpdate() much like how it is in the Player scripts. Every other variable and function in this script is there for functions that only effect 2D rock objects. The variables needed are as follows:

BoxRigid (Rigidbody) – This stores Rigidbody for the object that needs to have gravity affecting it.

E (EventsCode) – This variable stores the Eventscode Component that is within the game.

x and z (float) – These float variables are used to store the position of the Rock when it collides with the Wall in the x and y axis, so that it doesn't move position.

CollideCalc and WallHit (bool) – These bools are used for the 2D Rock calculations, with nothing happening if CollideCalc isn't false. WallHit is true when the rock collides with a wall.

DecalArea (GameObject) – This stores the DecalArea currently used for the rock, and works to parent the rock back to this area after it falls off a moving platform.

The Update() function is used when the rock collides with a wall, and works to freeze it's movement in the x and z positions, so that it doesn't move with the MovingPlatform, whilst letting it fall in the Y axis. The collision of the rock is checked in OnCollisionEnter(), which sets the values of x and z to the rock's current position in the x and z axis when they collide with a Wall. This also sets WallHit as true. When the rock collides with the floor, then it gets parented to the DecalArea and WallHit is set to false. This script is useful to add gravity, and has this needed code to get the collision of the rock working right.

RockWallCheck - Script:

```
public class RockWallCheck : MonoBehaviour
{
    //This checks when the rock has been attached to a MovingPaltofrm and collides with a wall, so it unparents itself.

    [SerializeField] GameObject Rock;

    [SerializeField] bool Left;
    [SerializeField] bool Right;

    [SerializeField] GameObject AttatchedMove;
    [SerializeField] GameObject NormalParent;

    private void OnTriggerEnter(Collider other)
    {
        if (other.gameObject.tag == "Wall")
        {
            if (Rock.transform.parent == AttatchedMove.transform)
            {
                Rock.transform.parent = NormalParent.transform;
            }
        }
    }
}
```

This is a script made specifically for the rock's trigger with walls, like with the functions added to AddGravity. This script is quite simple, so I won't go for a thorough analysis. This script only has one function, being OnTriggerEnter() which checks when the rock collides with a wall, before parenting them back to the regular normal of the rock. I am almost certain this script is mostly obsolete, but just in case, I decided to keep it in the game.

Pollen and SwordDestroy:

Both the Mega Man Player and the Zelda Player have unique abilities that require one piece of unique code each. For the Mega Man player this is the Pollen script, which is attached to the prefabs for the pollen bullets. This script works to move the bullet across the screen after being shot. The Zelda ability on the other hand has the SwordDestroy script which is attached to objects that the Zelda character can destroy when swinging. Anyway, let's start looking at these scripts.

Pollen - Script:

```
public class Pollen : MonoBehaviour
{
    //This script is used for the 2D pollen bullets which are shot by the Mega Man Player.

    //This stores the positions that the bullet will move between
    public GameObject OriginMark, TargetMark;

    //This stores the position the bullet needs to move towards.
    public Vector3 originPos => OriginMark.transform.position;
    public Vector3 targetPos => TargetMark.transform.position;

    //This checks if it is moving right
    public bool MoveRight;

    //This checks the speed of the object
    public float Speed;

    //This checks if the bullet is moving up.
    public bool MoveUp;
void FixedUpdate()
    {
        //This script is attatched to the pollen bullet and allows them to constatly move across the walls, before deleting themselves
        Vector3 UpdateTarget;
        Vector3 UpdateOrigin;

        //The values of UpdateTarget and UpdateOrigin depened on whether the bullet is moving up or sideways.
        if (MoveUp == false)
        {
            UpdateTarget = new Vector3(targetPos.x, gameObject.transform.position.y, targetPos.z);
            UpdateOrigin = new Vector3(originPos.x, gameObject.transform.position.y, originPos.z);
        }
        else
        {
            UpdateTarget = new Vector3(targetPos.x, targetPos.y, gameObject.transform.position.z);
            UpdateOrigin = new Vector3(originPos.x, originPos.y, gameObject.transform.position.z);
        }

        //The object moves the same way as the Decal Player
        gameObject.transform.localPosition = Vector3.MoveTowards(gameObject.transform.localPosition, MoveRight ? UpdateTarget : UpdateOrigin, Speed * Time.deltaTime);

        //When reaching the end of its movement, the bullet destroys itself
        if(gameObject.transform.position == UpdateTarget || gameObject.transform.position == UpdateOrigin)
        {
            Destroy(gameObject);
        }
    }

    private void OnCollisionEnter(Collision collision)
    {
        //When colliding with a wall or floor the pollen destroys itself.
        if(collision.gameObject.tag == "Wall" || collision.gameObject.tag == "Floor")
        {
            Destroy(gameObject);
        }
    }
}
```

The Pollen script is used to move the pollen bullets across the 2D scene much like how the 2D players move. This script has several variables:

OriginMark, TargetMark, originPos, targetPos, MoveRight, Speed (GameObject, Vector3, bool, float) - These variables are stored together as the all serve the same purposes as the variables of the same name in DecalMovement.

MoveUp (bool) - This bool checks if the bullet is meant to be moving upwards, rather then sideways.

The main function of this script is the FixedUpdate() which automatically moves the pollen across the scene based on a MoveTowards function. This function moves between UpdateTarget and UpdateOrigin in the direction it was shot from by the Mega Man Player. The values of UpdateTarget and UpdateOrigin are updated constantly, and depending on teh value of MoveUp may actually move the bullet up the Y-axis, rather then along the X. If the pollen reaches the location of one of the target Vector3 variables, or it collides with the wall or floor, then the bullet gets destroyed.

SwordDestroy - Script:

```
public class SwordDestroy : MonoBehaviour
{
    [SerializeField] Zelda_Movement Zm;

    //This script is attatched to objects which can be destroyed by the sword object, destroying them if the sword enters there collision
    private void OnCollisionEnter(Collision collision)
    {
        if(Zm.Sword == true)
        {
            Destroy(gameObject);
        }
    }

    private void OnCollisionStay(Collision collision)
    {
        if (Zm.Sword == true)
        {
            Destroy(gameObject);
        }
    }
}
```

This script is incredibly small and simple. It works as a Trigger attached to objects which can be destroyed by the Zelda Player's sword. This is done via OnTriggerEnter() and Stay() functions, which check when the sword has been swung, so that the object can be destroyed.

DecalTrigger and DecalDeath:

The final bit of code needed to set up the movement of the 2D characters are DecalTrigger and DecalDeath. DecalTrigger is used to get the collisions to effectively work within the scene, as I found using only the regular box collider caused issue with collisions. The DecalTrigger script is assigned to three trigger around the 2D players, one on the left, one on the right and one below. These are used to check when the player is colliding with the wall or floor, and whether they are being blocked from moving or jumping. DecalDeath is a simple script used to create triggers that kill the 2D player, taking them out the 2D space and getting them to retry. This is then used to also reset the player in the current 2D section.

DecalTrigger - Script:

```
public class DecalTrigger : MonoBehaviour
{
    //This script can work on both the Atari Player and the Mega Man Player
    [SerializeField] AtariAblilitySave AAS;

    [SerializeField] DecalMovement DM;

    [SerializeField] Mega_Man_Movement MMM;

    //These bools set where the trigger are set.
    [SerializeField]bool Jump, Left, Right;

    //This is true when the player is standing on a moving platform.
    [SerializeField] bool OnMoveFloor;

    //This script is used on the three triggers surrouding the 2D player and will check whether they are colliding with a wall or the floor.

    //If this is attatched to the right or left trigger this will cause the player to stop moving in that direction, whilst the bottom trigger will check if the player can jump.

    private void OnTriggerStay(Collider other)
    {
        if (Jump == true)
        {
            if (other.gameObject.tag == "Floor" || other.gameObject.tag == "MoveFloor" || other.gameObject.tag == "Rock")
            {
                if (AAS.AtariJump == true && DM != null)
                {
                    DM.JumpAble = true;
                }

                if(other.gameObject.tag == "Rock" && DM != null)
                {
                    DM.RockOn = true;
                }

                if(other.gameObject.tag == "MoveFloor")
                {
                    OnMoveFloor = true;
                    if(DM != null)
                    {
                        DM.gameObject.transform.parent = other.gameObject.transform;
                    }
                    if(MMM != null)
                    {
                        MMM.gameObject.transform.parent = other.gameObject.transform;
                    }

                }
            }
        }
        else if (Left == true)
        {
            if (other.gameObject.tag == "Floor" || other.gameObject.tag == "Wall" || other.gameObject.tag == "Rock" || other.gameObject.tag == "MoveFloor")
            {


                if(DM != null)
                {
                    DM.LeftAble = false;
                }
                if(MMM != null)
                {
                    MMM.LeftAble = false;
                }
            }
        }
else if (Right == true)
        {
            if (other.gameObject.tag == "Floor" || other.gameObject.tag == "Wall" || other.gameObject.tag == "Rock" || other.gameObject.tag == "MoveFloor")
            {
                if (DM != null)
                {
                    DM.RightAble = false;
                }
                if (MMM != null)
                {
                    MMM.RightAble = false;
                }
            }
        }
    }

    private void OnTriggerExit(Collider other)
    {
        if (Jump == true)
        {
            if (other.gameObject.tag == "Floor" || other.gameObject.tag == "MoveFloor" || other.gameObject.tag == "Rock")
            {
                if (AAS.AtariJump == true && DM != null)
                {
                    DM.JumpAble = false;
                }

                if (other.gameObject.tag == "Rock" && DM != null)
                {
                    DM.RockOn = false;
                }

                if (other.gameObject.tag == "MoveFloor")
                {
                    OnMoveFloor = false;
                    if (DM != null)
                    {
                        DM.gameObject.transform.parent = null;
                    }
                    if (MMM != null)
                    {
                        MMM.gameObject.transform.parent = null;
                    }

                }
            }
        }
        else if(Left == true)
        {
            if (other.gameObject.tag == "Floor" || other.gameObject.tag == "Wall" || other.gameObject.tag == "MoveFloor" || other.gameObject.tag == "Rock")
            {
                if (DM != null)
                {
                    DM.LeftAble = true;
                }
                if (MMM != null)
                {
                    MMM.LeftAble = true;
                }
            }
        }
        else if(Right == true)
        {
            if (other.gameObject.tag == "Floor" || other.gameObject.tag == "Wall" || other.gameObject.tag == "MoveFloor" || other.gameObject.tag == "Rock")
            {
                if (DM != null)
                {
                    DM.RightAble = true;
                }
                if (MMM != null)
                {
                    MMM.RightAble = true;
                }
            }
        }
    }
}
```

Whilst large, the DecalTrigger script actually completes very simple tasks. The script has several variables, which are as follows:

AAS, DM and MMM (AtariAbilitySave, DecalMovement and Mega_Man_Movement) - These variables store instances of the scripts listed, and are used to check the collision of these objects.

Jump, Left, Right, OnMoveFloor (bool) - These bools are used to check which trigger the instance of the script is. Left checks the left side, Right the right, Jump checks the floor and OnMoveFloor is set to true when the player stands on a moving platform.

This script only has two large functions, OnTriggerStay() and OnTriggerExit(). OnTriggerStay(). These functions check what is colliding with the different trigger types, and sets the appropriate variables as true or false. The variables all get set false when exiting a trigger, but only specific variables can be changed per trigger type. Each type of trigger, whether left, right or jump checks to see if it is assigned to the Mega_Man_Player of the DecalMovement script, and sets the right variables based on that. The Left and Right triggers will set the LeftAble and RightAble variables of the scripts as false when they collide with an object, thus stopping the players from walking into walls. The Jump trigger is the one that is unique, checking if the player is on the ground, and therefore if the Atari Player can jump. This trigger also checks when the player is standing on a moving platform, and parents them to the platform, so that they move with the platform as it goes. This system works well to easily, and concisely, have the correct collision for the 2D players.

DecalDeath – Script:

```
public class DecalDeath : MonoBehaviour
{
    [SerializeField]GenreSwap GS;

    //When entering the trigger the player will be kicked back to the 3D section of gameplay.
    //This script can be called by other scripts to quickly leave the 2D space.
    private void OnTriggerEnter(Collider other)
    {
        if(other.gameObject.tag == "Player")
        {
            Die();
        }
    }

    public void Die()
    {
        StartCoroutine(GS.Setup3D());
    }
}
```

This script is incredibly short, but incredibly important. It is a trigger, which when entered, kicks the player back to the 3D space, allowing me to effectively provide a form of death during the platforming segments. This is done via the Die() function, which is called when the player enters the trigger, or when the player presses the Q key. This function calls the Setup3D function found in the GenreSwap script stored to the variable GS.

FPS_Movement:

With the 2D Movement types, and the mechanics that come along with them, discussed it is now time to bring our attention over to the movement and mechanics for the 3D sections of gameplay. This will start with a discussion on the FPS_Movement script, which is used to get the 3D player moving. This works quite well to get effective 3D movement, and really ties the 3D space together. So, let’s start by having a look at the variables used in this script:

FPS_Movement - Variables:

```
public class FPS_Movement : MonoBehaviour
{
    //This script is used to program the movement of the 3D character

    //This is the speed the player walks at
    public float MoveSpeed;

    //This stores the Transform of a empty object, which is used to make sure the player is moving forward relative to their rotated positio
    public Transform Orientation;

    //This is used to change the drag of the player's rigidbody, which affects the Rigidbody's speed.
    public float Drag;

    //These store the inputs of the player in the Horizontal and Vertical axis.
    float HInput;
    float VInput;

    //This stores the direction the player should be moving towards
    Vector3 moveDirection;

    //This stores the scripts Rigidbody
    Rigidbody myRigidbody;

    //This checks if the player has opened the inventory menu
    public bool openInventory;

    //This checks if the player can move
    public bool CanMove;

    //This stores the Inventory script
    [SerializeField] Inventory I;

    //This stores the camera's raycast
    [SerializeField] GameObject CR;

    //This checks if the player has pressed the I key
    public bool PressI;

    //This stores the AudioManager
    [SerializeField] audiomanager AM;

    //This stores the velocity the player moves at in the X and z axis
    [SerializeField] float xVel;
    [SerializeField] float zVel;

    //This checks if the player is moving or not
    public bool moving;
```

This script has several variables all needed for it to work, which are as follows:

MoveSpeed and Drag (float) - The MoveSpeed float stores the speed that the player moves by, whilst Drag stores the amount of drag that should effect the Player Movement.

Orientation (Transform) - This stores the transform of a child to the 3D player, which is used to check where the current forward of the player is. This Transform is important in making sure the player always moves forward.

HInput and VInput (float) - These float variables take in the input of the player in the Horizontal and Vertical axis respectively.

moveDirection (Vector3) - This stores the direction that the player should move towards.

myRigidbody (Rigidbody) - This stores the Rigidbody of the 3D player.

openInventory and CanMove (bool) – These bools check whether the player is opening the inventory, or whether they can currently move.

I and CR (Inventory and GameObject) – I stores the Inventory script, whilst CR stores the GameObject that produces the Camera's Raycast.

PressI (bool) – This checks if the player has pressed the I key.

AM (AudioManager) – This stores the scene's AudioManager.

xVol and zVol (float) – These floats store the velocity of the player in the x and z axis.

moving (bool) – This bool checks if the player is moving, and is used to play the footstep audio.

FPS_Movement – Start(), Update() and FixedUpdate():

```
// Start is called before the first frame update
void Start()
{
    //This sets up the Player in the scene, attaching values to variables and making sure the player can move.

    AM = GameObject.FindGameObjectWithTag("AControl").GetComponent<audiomanager>();
    myRigidbody = GetComponent<Rigidbody>();
    myRigidbody.freezeRotation = true;

    CanMove = true;

    PressI = true;
}

// Update is called once per frame
void Update()
{
    //The Update() function sets up the speed max for the player and the movement inputs.
    FPSInput();
    SpeedContral();

    myRigidbody.drag = Drag;

    //Checks if the player is entering the inventory or closing it.
    if(PressI == true)
    {
        if (Input.GetKeyDown(KeyCode.I))
        {
            openInventory = !(openInventory);
            if (openInventory == true)
            {
                Cursor.lockState = CursorLockMode.None;
                Cursor.visible = true;
                CanMove = false;
                CR.SetActive(false);
                I.OpenInventoryB(true);
            }
            else if (openInventory == false)
            {
                PressI = false;
                I.OpenInventoryB(false);
            }
        }
    }
//This checks if the player is moving, and then players the footsteps audio when they are.
       if(Input.GetButton("Horizontal") || Input.GetButton("Vertical"))
       {
           AM.Walking = true;
           moving = true;
       }
       else
       {
           AM.Walking = false;
           moving = false;
       }

       //When pressing Escape, the player gets brought back to the MainMenu
       if(Input.GetKeyDown(KeyCode.Escape))
       {
           Cursor.lockState = CursorLockMode.None;
           Cursor.visible = true;
           SceneManager.LoadScene(0);
       }
   }

   private void FixedUpdate()
   {
       //If the player can move, then the Movement functions are called in the FixedUpdate()
       if(CanMove == true)
       {
           PlayerMovement();
       }
   }
```

The Start() function is used to initiate the 3D movement, setting the values of Variables such as AM or myRigidbody. It also sets it so the rotations of myRigidbody are frozen, so that it doesn't mess with the Camera Rotations. It then sets the player so they can move. The Update() function is first used to set up the values used for movement in the FPSInput() function. It then also sets the max speed the player should move at using the SpeedContral() function. After this it adds drag to the player's rigidbody, so that they have some realsitic form of speed.

Next, the script checks if the player has pressed the I key. If they have, then they stop being able to move and the Inventory opens. If the inventory is already open then they go back to being able to move. After this, the Update() function checks whether they are pressing the movement keys, and if they are then the script plays the footstep audio used by the player. The final thing the Update() checks is whether the player has pressed the Escape key. If the player has, then they get booted back to the Main Menu. The Movement function for this script, PlayerMovement(), is called within the FixedUpdate() function, so that the movement speed is consistent between Editor and Build.

FPS_Movement - FPSInput(), SpeedContral() and PlayerMovement():

```
private void FPSInput()
{
    //Registers the player input
    HInput = Input.GetAxisRaw("Horizontal");
    VInput = Input.GetAxisRaw("Vertical");
}

private void PlayerMovement()
{
    //Calculates the player's movement
    moveDirection = Orientation.forward * VInput + Orientation.right * HInput;

    myRigidbody.AddForce(moveDirection.normalized * MoveSpeed, ForceMode.Force);
}
```

```
private void SpeedContral()
{
    //Checks and maintains the speed of the player in movement.
    Vector3 SetVelocity = new Vector3(myRigidbody.velocity.x, 0f, myRigidbody.velocity.z);

    if(SetVelocity.magnitude > MoveSpeed)
    {
        Vector3 LimitVelocity = SetVelocity.normalized * MoveSpeed;
        myRigidbody.velocity = new Vector3(LimitVelocity.x, myRigidbody.velocity.y, LimitVelocity.z);
    }
}
```

The FPSInput() function is used to store the values of HInput and VInput into the values of the Input in the Horizontal and Vertical axis. This function is used to update the values each frame. The PlayerMovement() function is used for the actual movement of the player, with the function first figuring out what direction is forward for the player, before adding force to the Player's rigidbody in the direction of the object's normal. Finally, SpeedContral() is a function used to limit the velocity cap for the player, making sure that when moving the player doesn't exceed the max velocity, so the player can't move too fast.

FPS_Camera and CameraRaycast:

Whilst the FPS_Movement script works to have the player move in the 3D space, it is only half the story of what was programmed to move the player in the 3D space. The FPS_Camera script works to move the camera of the Player with the mouse, and uses this to effect the orientation of the script. The CameraRaycast is attached to a child of the FPS_Camera, and produces a raycast which allows the player to interact with items and objects in the scene.

FPS_Camera - Script:

```
public class FPS_Camera : MonoBehaviour
{
    //This works to rotate the First Person Character's camera view and to move the player in the direction the camera is facing.

    //These store the sensitivety of the mouse in the X and Y axis.
    public float XSensitivity;
    public float YSensitivity;

    //This stores the Transform that affects the forward orientation of the player's movement.
    public Transform Orientation;

    //These store the current rotation of the camera.
    float xRotation;
    float yRotation;

    //This stores the player's FPS_Movement script.
    [SerializeField] FPS_Movement FPSM;

    // Start is called before the first frame update
    void Start()
    {
        //The Start() function removes the mouse from the screen.
        Cursor.lockState = CursorLockMode.Locked;
        Cursor.visible = false;
    }

    // Update is called once per frame
    void Update()
    {
        //This script allows the FPS camera to rotate around a locked range.
        if(FPSM.CanMove == true)
        {
            float mouseX = Input.GetAxisRaw("Mouse X") * Time.deltaTime * XSensitivity;
            float mouseY = Input.GetAxisRaw("Mouse Y") * Time.deltaTime * YSensitivity;

            yRotation += mouseX;
            xRotation -= mouseY;

            //Makes it so you can't look up or down more then 90 degrees
            xRotation = Mathf.Clamp(xRotation, -90f, 90f);

            // Sets up the rotation of the camera and its orientation
            transform.rotation = Quaternion.Euler(xRotation, yRotation, 0);
            Orientation.rotation = Quaternion.Euler(0, yRotation, 0);
        }
    }
}
```

The FPS_Camera script has several variables needed to calculate the rotation of the camera, they are as follows:

XSensitivity, YSensitivity, xRotation and yRotation (float) - These floats are used to calculate the rotation of the camera. XSensitivity and YSensitivity are used to calculate the input of the mouse in the X and Y axis, which are then used to calculate the current rotation of the camera in those axis.

Orientation (Transform) - This stores the Transform variable used to effect the direction the player moves towards.

FPSM (FPS_Movement) - This stores the FPS_Movement script that the 3D player uses.

The Start() function for this script is used to lock in the mouse, stopping it from appearing within the scene. The rest of the code for this all takes place in the Update() function, which takes in the input from the mouse in the X and Y axis multiplied by the Sensitivity floats. This value is then added or removed from the xRotation and yRotation float, with the value of xRotation being clamped so that the player can only rotate by 90 degrees in either direction. With this done, the script rotates the camera by these values, and the Orientation transform gets rotated only along the Y-axis. For, when rotating in the Y axis it means the object is rotating left and right, and the X-axis is rotating up and down.

CameraRaycast - Variables:

```
public class CameraRaycast : MonoBehaviour
{
    //This script produces a raycast from the 3D player which allows them to interact with objects in the scene.

    //This stores the range of the raycast
    [SerializeField] float range = 100f;

    //This stores the Layers that the Raycast can interact with
    [SerializeField] LayerMask LM;
    //This stores the 3D Player's camera
    public Camera GameCamera;

    //This stores the item hit by the raycast
    [SerializeField] GameObject InventoryItem;
    //This stores the hit items default layer type
    [SerializeField] int LayerDefault;

    //This shows the Press E UI when highlighting an interactable object.
    [SerializeField] GameObject PressE;

    //This stores the diffrent CrossHair icons
    public GameObject[] CrossHair;

    //This stores the regular and highlighted materials for the cracks.
    public Material[] Cracks;

    //This sets the type of object highlighted.
    private int Type;

    //This is used to store highlighted DecalProjectors
    private DecalProjector DP;

    //This checks if the player is 2D.
    public bool Set2D;

    //This stores the start and end point of the raycast for Debugging purposes.
    [SerializeField] GameObject EndMarker;
    public GameObject StartMarker;
```

The CameraRaycast script has several variables all needed to generate the script. They are as follows:

range (float) - This float stores the range of the raycast.

LM (LayerMask) - This stores the Layers that the raycast can collide with.

GameCamera (Camera) - This stores the 3D player's camera.

InventoryItem (GameObject) - This is used to store the GameObject collided with by the raycast.

LayerDefault (int) - This stores the index value of the layer of the item stored in InventoryItem before it gets changed by the raycast.

PressE (GameObject) - This stores the PressE UI which appear on screen to let the player know they can interact with an item.

CrossHair[] (GameObject) - This stores the different Cross Hair icons that the script can swap between.

Cracks[] (Material) - This stores the different materials that the decal cracks can swap to project.

Type (int) - This stores the type of object the collided object is.

DP (DecalProjector) - This is used to store the collided with Decal Crack's Decal Projector.

Set2D (bool) - This checks if the player is in 2D gameplay or not.

EndMarker and StartMarker (GameObject) - This stores the ending and starting positions of the raycast, for debugging purposes.

CameraRaycast - Start() and Update():

```
    // Start is called before the first frame update
    void Start()
    {
        //This start by setting the default cross hair icon.
        SetCrossHair(true, false, false, false, false);
    }

    // Update is called once per frame
    void Update()
    {
        //This casts a raycatst, which calls the functions of collided objects, based on the type of object
hit.
        EndMarker.transform.localPosition = new Vector3(0, 0, (range / 10));
        RaycastHit hit;
if(Set2D == false)
        {
            if (Physics.Raycast(GameCamera.transform.position, GameCamera.transform.forward, out hit, range, LM))
            {
                if (hit.collider.tag == "InventoryItem")
                {
                    HitInventory(hit);
                }
                else if (hit.collider.tag == "Crack")
                {
                    Type = 2;
                    PressE.SetActive(true);
                    DP = hit.collider.gameObject.GetComponent<DecalProjector>();
                    DP.material = Cracks[1];
                }
                else
                {
                    EndHighlight();
                }
            }
            else
            {
                //if the raycast stops colliding with a object, then it calls EndHighlight.
                EndHighlight();
            }

            //This shows the length of the raycast in the Editor.
            Debug.DrawRay(GameCamera.transform.position, GameCamera.transform.forward * range, Color.green);
        }
    }
```

The Start() function works to set the Default CrossHair to be the one present, before beginning the Update() function. The Update() function works to cast a raycast out, which the game registers the collisions of (when the player isn't in the 2D gameplay mode). When colliding with an object, the script will check if it has either the "InventoryItem" or "Crack" tags. If it has the former, then the HitInventory() function is called, with the hit item being used for the calculations. If it hits the latter then that means it is colliding with a decal crack, in which it will then swap the material of so that the crack projection is highlighted. If it is hitting neither of these, or is not hitting anything, then the EndHighlight() function is called to stop any previous collisions. Finally, the Update() function create a debug line in the Editor to show the length of the raycast.

CameraRaycast - HitInventory():

```
//This function checks what item has been hit by the raycast, and uses this to allow the player to interact with scripts attached to those items.
//The crosshair of the scene changes icon depending on the type of interaction.
//These items get highlighted via the CheckItem() function.
void HitInventory(RaycastHit hit)
{
    Type = 1;
    if (hit.collider.gameObject.GetComponent<ItemPickup>() != null)
    {
        PressE.SetActive(true);
        CheckItem(hit, 1);

        SetCrossHair(false, true, false, false, false);

        if (InventoryItem != null)
        {
            InventoryItem.GetComponent<ItemPickup>().EnterRay();
        }
    }
    else if (hit.collider.gameObject.GetComponent<mouseclick>() != null)
    {
        PressE.SetActive(true);
        CheckItem(hit, 2);

        SetCrossHair(false, false, true, false, false);
if (InventoryItem != null)
        {
            InventoryItem.GetComponent<mouseclick>().Place();
        }
    }
    else if (hit.collider.gameObject.GetComponent<Projector>() != null)
    {
        PressE.SetActive(true);
        CheckItem(hit, 2);
    }
    else if (hit.collider.gameObject.GetComponent<Notes>() != null)
    {
        PressE.SetActive(true);
        CheckItem(hit, 2);
        SetCrossHair(true, false, false, false, false);

        if (InventoryItem != null)
        {
            InventoryItem.GetComponent<Notes>().PickUpNote();
        }
    }
    else if(hit.collider.gameObject.GetComponent<RockHit>() != null || hit.collider.gameObject.GetComponent<WaterBlock>() != null)
    {
        CheckItem(hit, 2);
        SetCrossHair(false, false, false, true, false);
    }
    else if(hit.collider.gameObject.GetComponent<GiveRockAbility3D>() != null)
    {
        PressE.SetActive(true);
        CheckItem(hit, 2);
        SetCrossHair(true, false, false, false, false);

        if (InventoryItem != null)
        {
            InventoryItem.GetComponent<GiveRockAbility3D>().TakeAbility();
        }
    }
    else if(hit.collider.gameObject.GetComponent<TeleportPlant>() != null)
    {
        if (hit.collider.gameObject.GetComponent<TeleportPlant>().Grown == true)
        {
            PressE.SetActive(true);
            CheckItem(hit, 2);
            SetCrossHair(true, false, false, false, false);
            if (InventoryItem != null)
            {
                InventoryItem.GetComponent<TeleportPlant>().PlantTeleport();
            }
        }
        else if(hit.collider.gameObject.GetComponent<TeleportPlant>().Grown == false)
        {
            CheckItem(hit, 2);
            SetCrossHair(false, false, false, false, true);
        }
    }
    else if (hit.collider.gameObject.GetComponent<StartTimeline>() != null)
    {
        PressE.SetActive(true);
        CheckItem(hit, 2);
        SetCrossHair(true, false, false, false, false);
        if(InventoryItem != null)
        {
            InventoryItem.GetComponent<StartTimeline>().CallTimeline();
        }
    }
}
```

The HitInventory() function is called when the raycast collides with an item that is under the “InventoryItem” tag, and is used to call scripts attached to the collided object, as well to highlight the item hit. The script does this by checking the components attached to the hit item, and calls functions from the items for the player to interact with. This is one of those functions that unfortunately has to be large in order for it to work, but is actually quite simple in practise. The script highlights the hit items using the CheckItem() function, and changes the icon of the cross hair using SetCrossHair() depending on the item hit.

CameraRaycast – SetCrossHair(), EndHighlight() and CheckItem():

```
    //This function is used to set the icon of the cross hair depending on what the raycast hits.
    private void SetCrossHair(bool one, bool two, bool three, bool four, bool five)
    {
        CrossHair[0].SetActive(one);
        CrossHair[1].SetActive(two);
        CrossHair[2].SetActive(three);
        CrossHair[3].SetActive(four);
        CrossHair[4].SetActive(five);
    }

    //This function is called when the Raycast should stop hitting objects, and works to reset everything back to the ways they were before being hit.
    void EndHighlight()
    {
        switch (Type)
        {
            case 0:
                {
                    if (InventoryItem != null)
                    {
                        InventoryItem.layer = LayerDefault;

                        SetCrossHair(true, false, false, false, false);
                    }
                    if (DP != null)
                    {
                        DP.material = Cracks[0];
                        DP = null;
                    }
                    PressE.SetActive(false);
                    Type = 0;
                }
                break;
            case 1:
                {
                    if (InventoryItem != null)
                    {
                        InventoryItem.layer = LayerDefault;

                        SetCrossHair(true, false, false, false, false);
                    }
                    InventoryItem = null;
                    LayerDefault = 0;
                    PressE.SetActive(false);
                    Type = 0;
                }
                break;
            case 2:
                {
                    DP.material = Cracks[0];
                    DP = null;
                    PressE.SetActive(false);
                    Type = 0;
                }
                break;
        }
    }

    //This function works to highlight the object hit by the raycast
    //If the InventoryItem is null, then this stores the hit item as it.
    //If the InventoryItem is not null, then the function returns the old InventoryItem to normal, before storing the new collision as the
InventoryItem.
    void CheckItem(RaycastHit Hit, int LNumber)
    {

        if(InventoryItem == null)
        {
            InventoryItem = Hit.collider.gameObject;
            LayerDefault = InventoryItem.layer;
            InventoryItem.gameObject.layer = LayerMask.NameToLayer("ItemSelection");
        }
        else if (Hit.collider.gameObject != InventoryItem)
        {
            InventoryItem.layer = LayerDefault;

            InventoryItem = Hit.collider.gameObject;
            LayerDefault = InventoryItem.layer;
            InventoryItem.gameObject.layer = LayerMask.NameToLayer("ItemSelection");
        }
    }
```

The SetCrossHair() function is used to quickly change the icon displayed on the cross hair, and is done by storing bools when it is called, which are then set onto each of the CrossHair icons. EndHighlight() is called when the raycast stops colliding with anything, and works to reset everything collided with by the raycast back to normal. This takes in the value of Type, so that it can tell what type of item has been collided with, and can therefore only reset the necessary elements. The CheckItem() function is used to add highlights to the collided item. It does this by setting the collided item as InventoryItem, and changes the items layer to the “ItemSelection” layer, which automatically adds a highlight around it. If the InventoryItem variable is null then this is all it does, but if it has already got a value when the raycast collides, then the function works to reset the old value before storing the variable with the new collision.

RockThrow and RockHit:

With the code for the 3D movement explained, I will now work to explain how the 3D mechanics function. The first of these 3D mechanics is a 3D version of the rock ability, which allows the player to throw out rocks which can knock over obstacles. This is done via two scripts, RockThrow - which is the script which actually tosses out a rock, and RockHit - which is attached to objects which are affected by the hit of the rock. So, let’s start by looking at RockThrow.

RockThrow - Variables:

```
public class RockThrow : MonoBehaviour
{
    //This script is used to throw out 3D rocks.

    //This stores the CameraRaycast script
    [SerializeField] CameraRaycast CR;

    //These store the force at which the rock is thrown forward and up.
    [SerializeField] float throwforwardforce;
    [SerializeField] float throwupforce;

    //This stores the Rock prefab that spawns the rock and the current rock spawned.
    [SerializeField] GameObject RockIntance;
    [SerializeField] GameObject CurrentRock;

    //These are used to create a cooldown for the rock after the player throws the rock.
    [SerializeField] float cooldownmax;
    [SerializeField] float currentcooldown;
    [SerializeField] bool shot;

    //This displays the UI Timer that shows the cooldown
    [SerializeField] TextMeshProUGUI Timer;
    //This displays the icon that appears when the player can throw the rock
    [SerializeField] GameObject CanUseSymbol;

    //This stores the AtariAbilitySave scriptable object.
    [SerializeField] AtariAblilitySave AAS;

    //This stores the audiomanager script
    [SerializeField] audiomanager AM;

    //These are audio clips that get played when the player uses the ability.
    [SerializeField] VoiceActing KnockThingsOver;
    [SerializeField] VoiceActing Oww;

    //This tells the script if the player can throw the rock.
    public bool CanRock;
```

The variables used for this script are as follows:

CR (CameraRaycast) - This variable stores the CameraRaycast gameobject.

throwforwardforce and throwupforce (float) - This stores the force at which the rock is thrown both forward and upwards.

RockInstance and CurrentRock (GameObject) - This stores the prefab used to spawn the rock, and the current instance of the rock.

cooldownmax, currentcooldown and shot (float and bool) - the float variables are used to track the cooldown of the rock throw, being triggered to start after shot is set true.

Timer and CanUseSymbol (TextMeshProUGUI and GameObject) - These variables are used for the UI cooldown for the rock, with the Timer displaying the time left before they can throw again, and the CanUseSymbol appearing when they can throw it again.

AAS and AM (AtariAbilitySave and audiomanager) - This stores the AtariAbilitySave and audiomanager scripts.

KnockThingsOver and Oww (VoiceActing) - These variables store some Voice Acting clips that play when the player throws a rock.

CanRock (bool) - This checks whether the player can shoot the rock or not.

RockThrow - Script:

```
//This sets up the Audio Manager at the Start
private void Start()
{
    AM = GameObject.FindGameObjectWithTag("AControl").GetComponent<audiomanager>();
}
// Update is called once per frame

private void Update()
{
    //The Update() function checks whether the player is able to shoot rocks.
    if (AAS.tDRock == true && CanRock == true)
    {
        if (Input.GetKeyDown(KeyCode.Mouse0) && shot == false)
        {
            Throw();
        }
        else if (Input.GetKeyDown(KeyCode.Mouse0) && shot == true && Oww != null)
        {
            //If the player spams the left click then the Oww click plays.
            Oww.AddLine();
        }
    }
}
void FixedUpdate()
{
    //The Timer is called in the FixedUpdate() so its consistent.
    CooldownUI();
    if (AAS.tDRock == true && CanRock == true)
    {
        if (shot == true)
        {
            currentcooldown += 0.1f;
            if (currentcooldown >= cooldownmax)
            {
                shot = false;
                currentcooldown = 0;
            }
        }
    }
}
```

```
    //The throw function deletes the current rock in the scene, before spawning a new one in, which is from the StartMarker stored in the Camera
Raycast.
    void Throw()
    {
        AM.RockThrow.SetActive(true);
        if(KnockThingsOver != null)
        {
            KnockThingsOver.AddLine();
        }
        if (CurrentRock != null)
        {
            Destroy(CurrentRock);
        }
        GameObject Rock = Instantiate(RockIntance, CR.StartMarker.transform.position, CR.GameCamera.transform.rotation);

        Rock.transform.localScale = RockIntance.transform.localScale;

        Vector3 ThrowForce = CR.GameCamera.transform.forward * throwforwardforce + transform.up * throwupforce;

        Rock.GetComponent<Rigidbody>().AddForce(ThrowForce, ForceMode.Impulse);

        CurrentRock = Rock;

        shot = true;
    }

    void CooldownUI()
    {
        //This function is used to affect the Cooldown UI.
        CanUseSymbol.SetActive(!shot);
        Timer.text = "" + Mathf.Round(currentcooldown * 10f) * 0.1f;
    }
```

The Start() function for RockThrow is used to set up the value of AM. The Update() function is where the player can throw the rock, assuming the ability is unlocked and they aren't in the cooldown period. If the player spams the throw button then the Oww Voice Clip plays, to tell them to wait for the cooldown to end. The FixedUpdate() is used to calculate the cooldown, and is used to call the CooldownUI() function, which is used to affect the Cooldown UI. The Throw() function is where the rock gets spawned and thrown. If a rock has already been stored in the scene, then it will be deleted, and a new rock will be spawned and thrown forward from the CamerRaycast's StartMarker. After this the cooldown begins again, before the player is able to throw another rock.

RockHit – Variables:

```
public class RockHit : MonoBehaviour
{
    //This script is used on objects which can be knocked over by the 3D Rock

    //This stores the instances animator
    [SerializeField] Animator ObjectAnim;
    //This checks if the object has been hit
    [SerializeField] bool BeenHit;
    //Some knocked over objects need to change the 2D platforms near them, and these one have this bool true.
    [SerializeField] bool Change2D;
    //This stores the different start cracks swapped between after hitting the object.
    [SerializeField] GameObject[] Cracks;
    //This stores the new Death trigger for the area
    [SerializeField] DecalDeath Death;
    //This stores the 2D area's DecalMovement script
    [SerializeField] DecalMovement DM;
    //This stores the scene's camera
    [SerializeField] Camera SCamera;

    //When this is true it means that the object's collider should turn to a trigger after being hit.
    [SerializeField] bool Triggerit;

    //This shows the object is a 3D switch
    [SerializeField] Switch3D SthreeD;

    //This stores a voice acting line that will be played after the player hits the platform
    [SerializeField] VoiceActing Line;

    //This bool is true when the 2D player is true, and is used to make sure the object doesn't get knocked over whilst the player is 2D.
    [SerializeField] bool Got2D;
```

The variables for this script are as follows:

ObjectAnim (Animator) - This stores the animator for the knocked over object.

BeenHit and Change2D (bool) - The BeenHit bool is used to check whether the object has already been knocked over, whilst Change2D is used to check whether the object will change the 2D space after being knocked over.

Cracks[], Death, DM and SCamera (GameObject, DecalDeath, DecalMovement and Camera) - These variables store elements which are changed in the 2D space after the player knocks over the object.

Triggerit (bool) - If this bool is true, it means the objects collider will become a trigger after being hit.

StreeD (Switch3D) - This stores the switch3D script for if the object is instance of that script.

Line (VoiceActing) - This stores a voice acting line which plays after the object is hit.

Got2D (bool) - This bool checks whether the player is currently in the 2D gameplay.

RockHit - Script:

```
//When hit by a rock the CollisionEnter() function causes the FallingDown() function to be called.
    //If it is a 3D switch then it calls the Switch3D script's HitSwitch() function.
    private void OnCollisionEnter(Collision collision)
    {
        if(DM == null)
        {
            Got2D = false;
        }
        else
        {
            Got2D = DM.gameObject.activeSelf;
        }
        if(collision.gameObject.tag == "Rock")
        {
            if(BeenHit == false && Got2D == false)
            {
                BeenHit = true;
                if(SthreeD == null)
                {
                    Fallingdown();
                }
                else
                {
                    gameObject.tag = "Untagged";
                    SthreeD.HitSwitch();
                }
            }
        }
    }
```

```
//This function plays the knocked over items animation, and if needed will change the 2D areas design.
   public void Fallingdown()
   {
       if(Line != false)
       {
           Line.AddLine();
       }
       gameObject.tag = "Untagged";
       if (Change2D == true)
       {
           ChangeTrigger();
       }
       ObjectAnim.SetBool("Hit", true);
       if(Triggerit == true)
       {
           gameObject.GetComponent<BoxCollider>().isTrigger = true;
       }
   }

   //This function changes the 2D area that is assigned to the knocked over item.
   void ChangeTrigger()
   {
       Cracks[0].SetActive(false);
       Cracks[1].SetActive(true);

       DM.DD = Death;
       DM.GameCamera = SCamera;
   }
}
```

The OnCollisionEnter() function is used to check when the rock has collided with the object. First, it checks whether the player is in the 2D gameplay or not, as the script will not continue if the player is 2D. When hit, the function checks whether the object is a 3D switch or not. If it is, then the HitSwitch function from the Switch3D script is called. If it is not, then the Fallingdown() function is called. This Fallingdown function works to activate the animator for the object to produce the animation of the object falling over. This function will also call the ChangeTrigger() function if the object will change the 2D area, and can change the collision into a trigger, but only if the associated bools are true. The ChangeTrigger() function works to swap things in the 2D space over to the new design, so that the change in the objects position have consequences in the 2D space.

FlowerShoot, pollencollision and TeleportPlant:

The second ability unlocked in the 3D space is the Flower Gun, which allows the player the ability to shoot pollen bullets that can be used to grow plants and hit objects. This is done in a script named FlowerShoot - which I will not actually be showing here, as it is practically identical to the RockThrow script, with the only difference being that it is shooting a pollen bullet, which has no upward arc, and has a shorter cooldown. Since those are the only differences, I will instead analyse the other scripts relating to this ability. pollencollision is a script attached to the pollen bullets shot, and is used to keep the bullet flying forward and check the collision around the object. TeleportPlant is the script attached to objects that can be shot by the pollen bullets, and can grow vines and plants which allow the player to teleport across the stage. These scripts are the real core to the Flower Gun ability working well, and will be my key focus when looking into this.

pollencollision - Script:

```
public class pollencollision : MonoBehaviour
{
    //This script is used to check the collision of the pullet bullets spawned by the 3D Flower gun.

    //These variables are used to give a time limit before the bullets delete themselves.
    public float currenttime;
    [SerializeField] float maxtime = 50f;

    //The FixedUpdate() calls the calculation of the timer for this bullet.
    private void FixedUpdate()
    {
        addTime();
    }

    //When hitting an object the bullet will destroy itself whilst checking what type of object it hid.
    private void OnCollisionEnter(Collision collision)
    {
        if(collision.gameObject.tag == "InventoryItem")
        {
            if (collision.gameObject.GetComponent<TeleportPlant>() != null)
            {
                if (collision.gameObject.GetComponent<TeleportPlant>().Grown == false)
                {
                    collision.gameObject.GetComponent<TeleportPlant>().GrowPlant();
                }
            }
            if(collision.gameObject.GetComponent<Switch3D>() != null)
            {
                collision.gameObject.GetComponent<Switch3D>().HitSwitch();
            }
            Destroy(gameObject);
        }
        else
        {
            Destroy(gameObject);
        }
    }

    //This function is used to delete the bullet instance after a certain amount of time.
    public void addTime()
    {
        if (currenttime >= maxtime)
        {
            Destroy(gameObject);
        }
        else
        {
            currenttime += 0.1f;
        }
    }
}
```

The pollencollision script works to call the functions of objects that the bullet collides with. There are only two varaibles, which both work to time how long the bullet has been there, which are:

currenttime and maxtime (float) - When currenttime equals more then maxtime the bullet instance is destroyed. This is done so the scene isn't overloaded by bullets.

The addTime() function is used to calculate the time the bullet has been spawned for, and is called from the FixedUpdate() function to keep it consistent. The main bit of code for this script is found in the OnCollisionEnter() function, which activates the functions of objects hit by the bullet, much like how HitInventory() works in the CameraRaycast script - and after doing this the instance of the bullet deletes itself.

TeleportPlant - Script:

```
public class TeleportPlant : MonoBehaviour
{
    //This script is placed on objects that are affected by the 3D pollen bullets

    //This stores the different objects that get swapped by the bullet
    [SerializeField] GameObject[] SwapObject;

    //This bool is set to true if the object is a Teleport plant, and makes it so the plant cannot grow again once hit.
    public bool Grown;

    //These store the 3D player and Fade UI, for when the player teleports
    [SerializeField] GameObject Player;
    [SerializeField] Animator Fade;

    //These store the new position and rotation of the player after telelporting.
    [SerializeField] Vector3 NewPos;
    [SerializeField] Quaternion NewRot;

    //This bool is set to true whilst the player is teleporting
    [SerializeField] bool Teleporting;

    //This bool, when true, means it is a vine which can be grown using the same script.
    [SerializeField] bool Vine;

    //This function is called when either a sprout or vine gets hit by the pollen bullet.
    public void GrowPlant()
    {
        //If it is a sprout, then that means the object will grow into a teleport plant, which allows the player to teleport between areas.
        if(Vine == false)
        {
            SwapObject[0].SetActive(true);
            SwapObject[1].SetActive(true);
            SwapObject[2].SetActive(false);
            gameObject.SetActive(false);
        }
        else if(Vine == true)
        {
            //If it is a vine, then that means the object will swap to a different vine size.
            SwapObject[0].SetActive(true);
            gameObject.SetActive(false);
        }
    }

    //If the player interacts with the TeleportPlant, then they can be teleported.
    public void PlantTeleport()
    {
        if (Input.GetKey(KeyCode.E) && Teleporting == false)
        {
            Teleporting = true;
            StartCoroutine(TeleportPlayer());
        }
    }

    //This IEnumerator works to warp the player to a new location.
    IEnumerator TeleportPlayer()
    {
        Fade.SetBool("Fade", true);
        yield return new WaitForSecondsRealtime(1f);
        Player.transform.position = NewPos;
        Player.transform.rotation = NewRot;
        Fade.SetBool("Fade", false);
        Teleporting = false;
    }
}
```

This script is used on plants that can be affected by the pollen bullet, and was originally only made for plants that teleport the player, but eventually adapted to be able to affect other types of plants aswell. The variables for this script are as follows:

SwapObject[] (GameObject) - This array of objects works to swap the objects of the plant once shot by the pollen.

Grown (bool) - This bool works to check if the vine has already been grown.

Player and Fade (GameObject and Animator) – These store the 3D Player and the Fade UI for when the player teleports.

NewPos and NewRot (Vector3 and Quaternion) – These store the values of the location the player warps to with the teleport plant.

Teleporting and Vine (bool) – The Teleporting bool is used to stop the player from teleporting again whilst they are currently teleporting. The Vine bool, when true, means that the object is a vine which changes size after being shot.

This script is made of several functions, two of which can be called by the player. The first, GrowPlant() can be called by the player's pollen bullet, and grows whatever plant it hits. If the object is not a vine, that that means it is a sprout for a teleport plant, and deactivates itself to instead activate the grown version of itself. If it is a Vine then that means all it does is deactivate itself to activate a different sized vine.

The PlantTeleport() function can be called by the CameraRaycast, and when interacted with can teleport the player over to a different location. This is done via the TeleportPlayer() IEnumerator, which adds the fade UI to the screen before moving the player to the location stored in NewPos and NewRot.

GiveRockAbility3D and SwapThrow:

With the 3D abilities described, there needed to be scripts to actually activate these abilities and allow the player to swap between them. The GiveRockAbility3D script was originally made to only activate the 3D Rock ability, but was eventually changed to also be able to give the 3D Flower ability. Once the player has both abilities, the SwapThrow script then allows the player to swap between their throw types by pressing the escape key. This section will explain each of these scripts and how they work to help provide the abilities.

```
public class GiveRockAbility3D : MonoBehaviour
{
    //This script works to activate either the 3D Rock or 3D Flower abilities.

    //This stores the Transform of the object that this script is attached to
    [SerializeField] Transform T;
    //This stores the AtariAbilitySave script
    [SerializeField] AtariAblilitySave AAS;

    //This stores the 3D Rock abilities UI.
    [SerializeField] GameObject RockUI;

    //This stores some tutorial text which fades on screen after the player unlocks the ability.
    [SerializeField] DecalTextAppear DTA;

    //This stores what ability gets awakened.
    [SerializeField] bool ItemType;

    //This stores a voice acting clip that plays after the player unlocks the ability.
    [SerializeField] VoiceActing GainAbility;

    // Start is called before the first frame update
    void Start()
    {
        //The start() funtion works to set up the itemtype, and to see what type of UI needs to be shown
        if(ItemType == false)
        {
            RockUI.SetActive(false);
        }
        T = gameObject.GetComponent<Transform>();
    }
// Update is called once per frame
    void Update()
    {
        //The Update() Function is used to make the object that activates the ability spin around.
        T.Rotate(0, 0.3f, 0);
    }

    public void TakeAbility()
    {
        //When the item is picked up by the player it will activate whichever ability the instance is there to awake.
        if (Input.GetKey(KeyCode.E))
        {
            switch(ItemType)
            {
                case false:
                    {
                        AAS.tDRock = true;
                        RockUI.SetActive(true);
                        DTA.SetOn = true;
                        if(GainAbility != null)
                        {
                            GainAbility.AddLine();
                        }
                        Destroy(gameObject);
                    }
                    break;
                case true:
                    {
                        AAS.FlowerGun = true;
                        DTA.SetOn = true;
                        Destroy(gameObject);
                    }
                    break;
            }

        }
    }
}
```

This script has several variables set to help it activate the 3D abilities, being:

T (Transform) – This stores the Transform of the gameObject.

AAS (AtariAbilitySave) – This stores the AtariAbilitySave scriptable object, of which the abilities are able to be activated in.

RockUI (GameObject) – This stores the UI for the rock ability which activates when the player unlocks the 3D rock ability.

DTA (DecalTextAppear) – This stores a decal tutorial which fades on screen when the player picks up the ability.

ItemType (bool) – This bool, when false means that the ability gained is the 3D rock ability, and when true is the 3D Flower Gun.

GainAbility (VoiceActing) – This stores a voice clip that plays when the player unlocks the ability.

The Start() and Update() functions for this script both have simple purposes. The Start() function works to make sure the RockUI is deactivated at the start of the game, whilst the Update() function works to make the instance of this script rotate on the spot, just to make it more enticing to collect by the player.

The main function for this script is TakeAbility(), which can be called by the CameraRaycast script, and is what gives the player their abilities. This function checks what ability needs to be given, and then gives the needed ability. This works to be quite easy to use for multiple abilities, and easily allows the player to unlock new abilities.

SwapThrow - Script:

```
public class SwapThrow : MonoBehaviour
{
    //This script is used for the player to swap between the 3D Rock and 3D Flower abilities.

    //This stores the AtariAbilitySave scriptable objects, so that the script can check when the two 3D abilities are unlocked.
    [SerializeField] AtariAblilitySave AAS;

    //This bool is used to check if the rock is selected or not
    [SerializeField] bool Rock;
    [SerializeField] RockThrow RT;
    [SerializeField] FlowerShoot FS;

    //This stores the UI for the two 3D abilties, and the 3D FlowerGun to activate and deactivate.
    [SerializeField] GameObject RockUI;
    [SerializeField] GameObject FlowerUI;
    [SerializeField] GameObject FlowerGun;

    // Start is called before the first frame update
    void Start()
    {
        //The start function sets things up so the Rock is the first one playable by the player
        Rock = true;
        FS.CanShoot = false;
        RT.CanRock = true;
        FlowerGun.SetActive(false);
        FlowerUI.SetActive(false);
    }

    // Update is called once per frame
    void Update()
    {
        //The Update() function checks when the player presses the Right click
        if(Input.GetKeyDown(KeyCode.Mouse1) && AAS.FlowerGun == true)
        {
            //When pressed the current active ability is swapped.
            Rock = !(Rock);

            switch(Rock)
            {
                //When Rock is true, then the Rock ability is activated.
                case true:
                    {
                        RT.CanRock = true;
                        RockUI.SetActive(true);
                        FS.CanShoot = false;
                        FlowerUI.SetActive(false);
                        FlowerGun.SetActive(false);
                    }
                    break;
                //When Rock is false, then the Flower Gun ability is activated.
                case false:
                    {
                        FS.CanShoot = true;
                        FlowerUI.SetActive(true);
                        FlowerGun.SetActive(true);
                        RT.CanRock = false;
                        RockUI.SetActive(false);

                    }
                    break;
            }
        }
    }
}
```

The SwapThrow script is used to swap between the 3D flower and 3D Rock ability. The variables are as follows:

AAS (AtariAbilitySave) - This stores the AtariAbilitySave scriptable object, which is used to check when both abilities are active.

Rock, RT and FS (bool, RockThrow and FlowerShoot) - The Rock bool is used to swap between the two abilities, which are stored in RT and FS.

RockUI, FlowerUI and FlowerGun (GameObject) - The UI GameObjects are used to store the UI for the different abilities, whilst FlowerGun is used to store the 3D FlowerGun model that appears when the player uses the gun.

The Start() function is used to initiate things so that the Rock ability is always the first active. The actual swapping of the ability is done within the Update() function, where the value of Rock is inverted when the player presses the Right Click. The current ability is then activated based on the value of rock. When it’s true the 3D rock ability is the one active, with the appropriate UI shown. When it’s false then things swap over to the 3D Flower ability, with the UI and 3D model displaying.

Dialogue, Timelines and Voice Acting:

This section of the document is here to discuss elements of the game which are needed to help tell the narrative, from the Dialogue system to the Notes, Cutscenes and Voice Acting. These all use similar scripts to each other, and can be easily compared, with the scripts for each type tying into each other well. So, let's start by discussing the code for the Dialogue System.

Dialogue and TriggerDialogue:

These two scripts are small but important in setting up the Dialogue system. The Dialogue script creates a class that is used to create a variable in TriggerDialogue, which stores every line of Dialogue in a sequence. Both of these scripts are incredibly tiny, and only consist of one function, if that. So, let's look at them.

Dialogue - Script:

```
[System.Serializable]
public class Dialogue
{
    //This class stores the three variables used for dialogue:
    Name, which stores the name of who's speaking;
    Sentances which stores the dialogue said;
    Tag, which stores the tag used to trigger events at the end of the dialogue
    public string[] Name;

    public string[] Sentances;

    public string tag;
}
```

This script creates a class which, stores three variables, which are:

Name[] (string) - This stores an array of different names of the character speaking in Dialogue sequences.

Sentances[] (string) - This stores an array of the lines said in the Dialogue sequence.

tag (string) - This stores a tag that can cause different effects at the end of Dialogue sequences.

TriggerDialogue - Script:

```
public class TriggerDialogue : MonoBehaviour
{
    //This script is used to attatch a dialogue variable to a gameobject
    public Dialogue D;
    [SerializeField] Dialogue_Manager DI_M;
    [SerializeField] TimelineManager TM;
public void CallD()
    {
        if(DI_M != null)
        {
            DI_M.StartDialogue(D);
            TM.pause = true;
            TM.Dialogue = true;

        }
    }
}
```

This script is used to trigger Dialogue sequences, and has three simple variables:

D (Dialogue) - This stores the values used for the Dialogue System.

DI_M and TM (Dialogue_Manager and TimelineManager) - These variables are used to store the Dialogue_Manager and TimelineManager scripts.

This script only has one function, CallD(), which is called by other scripts when they want to start a Dialogue sequence. This script does what it says, it calls the Dialogue Manager and plays the Dialogue stored in D. This script also works to pause the Timeline that may be playing a cutscene, until the Dialogue sequence is complete.

Dialogue_Manager:

With the two preliminary scripts discussed, this section of the script will be about the main Dialogue_Manager script, which takes the information called by the TriggerDialogue script, and creates a Dialogue sequence out of it. So let's jump in, and look at the script's variables.

Dialogue_Manager - Variables:

```
public class Dialogue_Manager : MonoBehaviour
{
    //This script is used to create Dialogue sequences within the game.

    //This script stores Queues of each Sentance and Name listed in the Dialogue sequence.
    private Queue<string> Sentance;
    private Queue<string> Names;
    public string TextTag;

    //These store the aspects of the text UI on screen, such as the text itself and the animator for the text for it to slide on screen.
    [SerializeField]TextMeshProUGUI Dialogue;
    public Animator Anim;
    //This stores whatever 2D player is active during the Dialogue sequence.
    public DecalMovement DM;
    public Mega_Man_Movement MMM;

    //This bool is set to true whilst the game is in a Dialogue sequence.
    public bool inD;

    //This stores a GameObject that has DecalText which fade onto the scene.
    [SerializeField] GameObject FadeMove;

    //This stores the audiomanager to create a satisfying sound when the player clicks between lines.
    [SerializeField] audiomanager AM;
```

The variables for this script are as follows:

Sentance<>, Names<> and TextTag (string) - These string queue's store the Sentances and Names provided by the called Dialogue variable, and the TextTag string which stores the Tag for the dialogue sequence.

Sentance<>, Names<> and TextTag (string) - These string queue's store the Sentances and Names provided by the called Dialogue variable, and the TextTag string which stores the Tag for the dialogue sequence.

Dialogue and Anim (TextMeshProUGUI and Animator) - These store elements of displaying the text, and the animator for the text, so that the Dialogue box fades onto screen.

DM and MMM (Decal_Movement and Mega_Man_Movement) - This stores the 2D Player type in the scene the Dialogue sequence.

inD (bool) - This bool is set to true when the player is in a Dialogue sequence.

FadeMove (GameObject) - This stores the GameObject of decal text that can fade on screen.

AM (audiomanager) - This stores the audiomanager script.

Dialogue_Manager - Script:

```
// Start is called before the first frame update
void Start()
{
    AM = GameObject.FindGameObjectWithTag("AControl").GetComponent<audiomanager>();

    //The start function starts by setting up the Sentance and Names queue
    Sentance = new Queue<string>();
    Names = new Queue<string>();
}

// Update is called once per frame
void Update()
{
    //The Update() function works to call the DisplayNextSentance() function when the player left clicks
    if(inD == true && Input.GetKeyDown(KeyCode.Mouse0))
    {
        DisplayNextSentance();
    }
}

public void StartDialogue(Dialogue D)
{
    //This function is called to set up the dialogue, setting the Names and Sentances into queues, which is used in later functions
    Sentance.Clear();
    Names.Clear();

    Anim.SetBool("Open", true);

    //The function freezes the movement of the 2D Player during the function
    if(DM != null)
    {
        DM.isMove = false;
        DM.CutsceneJump = true;
    }

    if(MMM != null)
    {
        MMM.isMove = false;
        MMM.CanShoot = false;
    }

    TextTag = D.tag;

    inD = true;

    //The function uses foreach loops to add to the Sentance and Names queues.
    foreach(string S in D.Sentances)
    {
        Sentance.Enqueue(S);
    }
    foreach(string N in D.Name)
    {
        Names.Enqueue(N);
    }
//The function ends by typing the first sentance in the DisplayNextSentance() function
        DisplayNextSentance();
    }

    public void DisplayNextSentance()
    {
        AM.Dialogue.SetActive(true);
        AM.Dialogue.GetComponent<AudioSource>().Play();
        //This function works to begin typing the next line of dialogue
        //The function checks if there are any sentances left to type, and if not EndDialogue is called.
        if (Sentance.Count == 0)
        {
            EndDialogue();
        }
        else
        {
            //The function stores the next line of dialogue and stops all Coroutines, before calling the Type Sentance Coroutine
            string sentance = Sentance.Dequeue();
            string name = Names.Dequeue();
            StopAllCoroutines();
            StartCoroutine(TypeSentance(name, sentance));
        }
    }

    IEnumerator TypeSentance(string N, string S)
    {
        //This works to type the current sentance out char by char.
        Dialogue.text = N + ": ";

        foreach(char letter in S.ToCharArray())
        {
            Dialogue.text += letter;
            yield return new WaitForSeconds(0.03f);
        }
    }

    void EndDialogue()
    {
        //This function ends the dialogue scene, and will trigger events based on the instance's Tag.
        Anim.SetBool("Open", false);

        //This function completes different functions after the dialogue sequence is complete, before setting the player movement back to normal
        if(TextTag == "F")
        {
            DM.isMove = true;
            DM.CutsceneJump = false;
            DM = null;
            inD = false;
            StartCoroutine(FadeMove.GetComponent<DecalTextAppear>().Fadein());

        }
        if(TextTag == "S")
        {
            DM.isMove = true;
            DM.CutsceneJump = false;
            inD = false;
            DM = null;
        }
        if(TextTag == "M")
        {
            MMM.isMove = true;
            MMM.CanShoot = true;
            MMM = null;
        }
        TextTag = null;

        inD = false;
        return;
    }
}
```

This script is made of several different functions which all tie together. The Start() function is used to set up the Sentance and Names queues. The Update() function is used to type the next sentence of the Dialogue. The StartDialogue() function is called by other scripts when setting up the Dialogue system, which pauses the 2D player's movement to start the Dialogue sequence. It then stores each Name and Sentence listen in the given Dialogue class into Names and Sentences, using the foreach loop. The setup ends by calling the DisplayNextSentance() function to display the first line of dialogue.

The DisplayNextSentance() function checks whether their are any sentences left in the Dialogue sequence, and if not, the function calls the EndDialogue() function. If there are any sentences left, the function takes the next line of Dialogue and displays it using the TypeSentance() IEnumerator, which types the sentence out letter by letter. Finally, the EndDialogue() function moves the Dialogue UI off screen, and gives the 2D player movement again. It also completes any functions called by the Tag of the Dialogue sequences, to cause different effects, such as having Decal Text appear when the dialogue is complete.

TimelineManager and StartTimeline:

With the Dialogue system explained, I will now turn my attention to the Timeline system. Timelines are used in my game to create cutscenes, which involve several different Dialogue sequences called after each other. This is done within the TimelineManager script, which works to create effective cutscenes out of the timeline. These Timelines are called from the StartTimeline script, which is attached to areas where Timelines can start. So, let's have a look at these scripts.

TimelineManager - Script:

```
public class TimelineManager : MonoBehaviour
{
    //This script is used to manage timelines in the game to create cutscenes.

    //This stores the Playable Director which contains the Timeline sequence.
    public PlayableDirector PlayD;
    //These bools are used to check if the timeline is paused, playing or in a Dialogue sequence.
    public bool play;
    public bool pause;
    public bool Dialogue;

    //This stores the Dialogue_Mangager script.
    [SerializeField] Dialogue_Manager DI_M;


    private void Update()
    {
        //The update() function checks whether the play, pause or Dialogue bools are true, and calls the appropriate functions based on this.
        if(play == true)
        {
            play = false;
            Play();
        }

        if(pause == true)
        {
            pause = false;
            Pause();
        }

        if(Dialogue == true)
        {
            //When in Dialogue, the Timeline will not continue until the current Dialogue sequenece is complete.
            if(DI_M.inD == false)
            {
                Dialogue = false;
                play = true;
            }
        }
    }
    //This function plays the Timeline
    public void Play()
    {
        PlayD.Play();
    }

    //This function pauses the Timeline.
    public void Pause()
    {
        PlayD.Pause();
    }
}
```

The TimelineManager script is short and effective, using a few variables to get things set up. These variables are as follows:

PlayD (PlayableDirector) - This stores the Playable Director which plays the Timeline.

play, pause, Dialogue (bool) - These bools are used to check if the Timeline sequence is playing, paused or in a Dialogue sequence.

DI_M (Dialogue_Manager) - This stores the Dialogue_Manager script.

The Update() function checks the current values of play, pause and Dialogue. If play is true, then the Play() function is called, which plays the Timeline. If pause is true then the Pause() function is called, which pauses the Timeline. When Dialogue is true, the Timeline will remain paused until the current Dialogue sequence is complete. This simple function works well to create cutscenes which flow perfectly, regardless of the length of the sequence.

StartTimeline - Script:

```
public class StartTimeline : MonoBehaviour
{
    //This script is used to start a Timeline sequence

    //This stores the PressE UI Pop-up
    [SerializeField] GameObject PressE;

    //This stores the TimelineManager script for the cutscene called
    [SerializeField] TimelineManager TM;

    //This bool is true whilst the Timeline is playing
    [SerializeField] bool playing;

    //This int stores if the player has been talked to already
    [SerializeField] int talked;

    //This stores the 2D players movement
    [SerializeField] DecalMovement DM;

    //This is used to call the first Dialogue sequence of the Timeline
    [SerializeField] Dialogue D;
    [SerializeField] Dialogue_Manager DIM;

    //This stores the Decal Text that appears because of the Timeline.
    [SerializeField] DecalTextAppear Decal;

    //CallTimeline() is called when the player presses E whilst triggering with the script
    //This function plays the Timeline.
    public void CallTimeline()
    {
        if (Input.GetKey(KeyCode.E) && playing == false)
        {
            PressE.SetActive(false);
            playing = true;
            TM.Play();
            gameObject.tag = "Untagged";
        }
    }
    private void OnTriggerEnter(Collider other)
    {
        if(playing == false)
        {
            PressE.SetActive(true);
        }
    }

    private void OnTriggerStay(Collider other)
    {
        if(other.gameObject.tag == "Player")
        {
            if(Input.GetKey(KeyCode.E) && playing == false)
            {
                PressE.SetActive(false);
                playing = true;
                //This checks if the player has already experienced the Timeline sequence.
                if(talked == 0)
                {
                    talked = 1;
                    TM.play = true;
                    DM.isMove = false;
                    DM.CutsceneJump = true;
                }
            }
        }
    }

    private void OnTriggerExit(Collider other)
    {
        if(other.tag == "Player")
        {
            PressE.SetActive(false);
        }
    }

    //This function ends the timeline, and is called at the end of the timeline sequence.
    public void EndTimeline()
    {
        DM.CutsceneJump = false;
        DM.isMove = true;
        StartCoroutine(Decal.Fadein());
    }
```

This script is used to call the Timeline, and has several variables for this:

PressE and TM (GameObject, TimelineManager) - This stores the Press E UI pop-up and the TimelineManager script.

playing and talked (bool and int) - The playing bool is set to true when the player is in a Timeline sequence and the talked int stores whether the player has already been through the Timeline before.

DM, D and DIM (DecalMovement, Dialogue and Dialogue_Manager) - These sets of Ds store the Atari Player's movement so it can stop moving at the start of the timeline, the first set of Dialogue for the Timeline which is then called into the Dialogue_Manager script.

Decal (DecalTextAppear) - This stores a piece of decal text which appears during the Timeline.

This script is made to call the selected Timeline when the CallTimeline() function is called by the player pressing the E key within the script's trigger. When calling the timeline, the script will check if the player has already interacted with it once, in which case it will create a new timeline for the situation. The only other function of note is EndTimeline() which is called from within the Timeline itself to get the player back to being able to move.

Notes and NoteParent:

The next system to be discussed here are the note objects that are collectable to the player. These objects are found in instances of the Notes script, which when collected display a note on screen, through the NoteParent script. Using this allows me to easily add extra narrative and lore to the game, without much worry. So, lets look at the first of these scripts.

Notes – Script:

```
public class Notes : MonoBehaviour
{
    //This script is used when calling Notes to be displayed to the player

    //These strings store the text and name of the Note
    [SerializeField] string NoteText;
    [SerializeField] string FileName;

    //This stores the NoteParent script
    [SerializeField] NoteParent NP;


    //When this bool is true it means the note is located in the 2D space.
    [SerializeField]bool flat;

    //This stores the PressE Pop-up
    [SerializeField] GameObject PressE;

    //This stores the gameobject that triggers the note.
    [SerializeField] GameObject P;


    //This function is called by the camera raycast to display a note UI.
    public void PickUpNote()
    {
        if(Input.GetKey(KeyCode.E))
        {
            NP.NoteText.text = NoteText;
            NP.FileName.text = FileName;
            NP.Set(false, null);
            Destroy(gameObject);
        }
    }

    //This function is allows the player to display the note when pressing E within the 2D space's trigger.
    private void OnTriggerStay(Collider other)
    {
        if (flat == true)
        {
            if (other.gameObject.tag == "Player")
            {
                PressE.SetActive(true);
                P = other.gameObject;
                if (Input.GetKey(KeyCode.E))
                {
                    P.gameObject.GetComponent<DecalMovement>().isMove = false;
                    NP.NoteText.text = NoteText;
                    NP.FileName.text = FileName;
                    NP.Set(true, P);
                    PressE.SetActive(false);
                    Destroy(gameObject);
                }
            }
        }
    }

    //When exiting the trigger the value of P and PressE are reset.
    private void OnTriggerExit(Collider other)
    {
        if (other.gameObject.tag == "Player")
        {
            PressE.SetActive(false);
            P = null;
        }
    }
```

This script is used for instances of the notes which the player can collect. It has several variables which are sent to the NotesParent script to call the Note item. These variables are:

NoteText and FileName (string) - These store the text of the note and the Name of it's file.

NP (NoteParent) - This stores the NoteParent script, which these values are sent to.

flat (bool) - This bool is set to true when the note is collected in the 2D space.

PressE (GameObject) - This is the Press E pop up that appears when the player triggers with the note.

P (GameObject) - This stores the instance of the 2D player that interacts with the note.

This script works to provide the values of it's instance to the NoteParent script when the player interacts with it, whether this is done via the PickUpNote() function (which is called by the CameraRaycast) or the Trigger collision, either way, this gameObject will get the text it has stored to appear on screen before deleting the gameObject.

NoteParent - Script:

```
public class NoteParent : MonoBehaviour
{
    //This script is used as the parent for the Note objects the player can pick up.

    //This stores the text and title for the note item
    public TextMeshProUGUI NoteText;
    public TextMeshProUGUI FileName;

    //This bool is true when the note should be set on screen.
    public bool SetNote;

    //This stores the animator for the Note UO
    [SerializeField] Animator Anim;
    //This stores the First Person movement
    [SerializeField] FPS_Movement FPSM;

    //This bool is true if the note displayed was taken from the 2D space.
    [SerializeField] bool flat;
    //This stores the instnace of the player that interacts with the note
    [SerializeField] GameObject P;

    //This stores the audiomanager script.
    [SerializeField] audiomanager AM;

    //The start function sets up the audiomanager.
    private void Start()
    {
        AM = GameObject.FindGameObjectWithTag("AControl").GetComponent<audiomanager>();
    }
private void Update()
    {
        //if the note is set on screen and the played left clicks then the note will leave the screen.
        if(SetNote == true)
        {
            if(Input.GetKey(KeyCode.Mouse0))
            {
                AM.Dialogue.SetActive(true);
                AM.Dialogue.GetComponent<AudioSource>().Play();
                Anim.SetBool("EnterScreen", false);
                SetNote = false;
                //Once the note leaves this function provides movement back to the player that picked it up.
                if(flat == false)
                {
                    FPSM.CanMove = true;
                }
                else
                {
                    P.gameObject.GetComponent<DecalMovement>().isMove = true;
                    P = null;
                }
            }
        }
    }

    //This function is called by the Notes script, and displays the picked up note on screen.
    public void Set(bool f, GameObject g)
    {
        if(g != null)
        {
            P = g;
        }
        flat = f;
        if(flat == false)
        {
            FPSM.CanMove = false;
        }
        Anim.SetBool("EnterScreen", true);
        SetNote = true;
    }

}
```

This NoteParent script takes the values stored by the Notes instance and uses it to present the Note UI. The variables are as follows:

NoteText and FileName (TextMeshProUGUI) - These are the text components that display the text stored in the Notes instance.

SetNote (bool) - This bool is set to true when the note has been displayed on screen.

Anim and FPSM (Animator and FPS_Movement) - This stores the animator of the Notes UI that animates onto screen. FPSM is used to store the First Person player's script.

flat, P and AM (bool, GameObject and audiomanager) - The flat and P variables are set based on the Notes instance, and store if the note was collected in the 2D space. AM also stores the audiomanager, like many other scripts.

The NoteParent scripts sets a note onto the screen when the Set function is called by the Notes script, which displays the Note UI on screen with the text stored in the Notes instance. When the note is on screen the Update() function will remove the Note from screen after the player left clicks next. It then grants movement back to the player, whether 2D or 3D.

VoiceActing and VoiceActingManager:

The next system to discuss is that of adding Voice Acting to the game. This was done via two scripts, VoiceActing which adds a voice clip to a list stored in the VoiceActingManager script, which actually plays the audio out loud. These scripts are quite simple, so let's get right to it.

VoiceActing - Script:

```
public class VoiceActing : MonoBehaviour
{
    //This script is used to play Voice lines in the scene

    //This variables stores the VoiceActingManager script
    [SerializeField] VoiceActingManager VAM;
    //This stores the string of the sentance said by the voice clip played
    [SerializeField] string sentance;
    //This stores the actual voice line played.
    [SerializeField] AudioClip Line;

    //This bool is set to true when the voice clip is triggered.
    [SerializeField] bool triggered;

    //These bools decide whether the voice clip should skip to the newest clip, or for the clip to not delete itself.
    [SerializeField] bool Skip;
    [SerializeField] bool Keep;

    private void Start()
    {
        VAM = GameObject.FindGameObjectWithTag("AControl").GetComponent<VoiceActingManager>();
    }
    //When entering the trigger the script calls the AddLine() function
    private void OnTriggerEnter(Collider other)
    {
        if(triggered == false)
        {
            triggered = true;
            AddLine();
        }
    }

    //This function can be called by triggers or other scripts, and adds the Voice Clip to the list in the VoiceActingManager script.
    public void AddLine()
    {
        VAM.AddLine(Line, sentance, Skip);
        if(Keep == false)
        {
            Destroy(gameObject);
        }
    }
}
```

The VoiceActing script works to add audio clips to the list stored in VoiceActingManager, and has the following variables:

VAM (VoiceActingManager) - This stores the VoiceActingManager that plays all the clips.

sentance and Line (string and AudioClip) - These variables store the voice clip that plays, and a text version of what is said is stored in sentance.

triggered (bool) - This bool is used to check when the player has entered the script's trigger.

Skip and Keep (bool) - These bools are used to set values of the voice clip. If Skip is true it means the clip is played straight away, and is Keep is true it means the instance does not get deleted after being played.

This script works by adding the information of the instance to the VoiceActingManager when the AddLine() function is called, whether in trigger or from another script. This function adds the audio clip stored in line, and all necessary information, into the lists stored in VoiceActingManager. If the Keep variable is false, then the instance gets deleted.

VoiceActingManager - Script:

```
public class VoiceActingManager : MonoBehaviour
{
    //This stores List of the AudioClips that play for the Voice Acting and the text that the character is saying.
    public List<AudioClip> Lines;
    public List<string> Sentance;

    //This stores the AudioSource that plays the Voice clips
    [SerializeField] AudioSource AS;

    //This bool is set true when the audio clip is playing
    [SerializeField] bool Playing;

    //This is the Text for the Text for the Voice Acting
    [SerializeField] TextMeshProUGUI Dialogue;

    //If required, a voice clip can skip right to the newest line.
    [SerializeField] bool Skip;

    //The FixedUpdate is used to check when an voice clip has ended, and whether a new line needs to be played.
    private void FixedUpdate()
    {
        if(Playing == true)
        {
            if(AS.isPlaying != true || Skip == true)
            {
                if (Lines.Count > 0)
                {
                    AS.clip = Lines[0];
                    Dialogue.text = Sentance[0];

                    Lines.RemoveAt(0);
                    Sentance.RemoveAt(0);
                    AS.Play();
                }
                else
                {
                    Dialogue.text = null;
                    AS.clip = null;
                    Playing = false;
                }
                Skip = false;
            }
        }
    }

    //This function is called by the VoiceActing script, and adds the set Voice clip to the list of voice clips.
    public void AddLine(AudioClip AC, string S, bool skip)
    {
        //When the voice clip is needed to skip then the audio list skips straight to the newly added voice line.
        switch(skip)
        {
            case true:
                {
                    Lines.Clear();
                    Sentance.Clear();
                    Lines.Add(AC);
                    Sentance.Add(S);
                    Playing = true;
                    Skip = true;
                }
                break;
            case false:
                {
                    Lines.Add(AC);
                    Sentance.Add(S);
                    Playing = true;
                }
                break;
        }
    }
}
```

The VoiceActingManager script is used to play out any audio clips of voice acting stored by the VoiceActing script. It takes the variables given to it and use them to affect the variables of this script, which are:

Lines<> and Sentance<> (AudioClip and string) - These are lists which store every audioclip and piece of text that needs to be presented by the script.

AS (AudioSource) - This is the audio source which plays the Voice clips stored in Lines.

Playing (bool) - This bool is set to true when the audio clips are being played.

Dialogue (TextMeshProUGUI) - This is a Text component which displays the dialogue said by the voice clip.

Skip (bool) - When this bool is true the added audio clip is skipped to straight away.

This script works when audio clips in the List are played. These lists get added to in the AddLine() function, which adds the values taken from Voice Acting instances to the list, for them to be played. If Skip is true, then the lists are cleared, and the most recent audio clip is immediately played. The FixedUpdate() then checks to see when the currently playing audio clip has finished playing, and deletes it and plays the next clip. If there are no more clips to play, then Playing is set to false.

RandomWait and audiomanager:

The final scripts to be discussed in this section of the document are those used to manage the final bits of audio in the game. The first is a small script called RandomWait, which plays random voice clips when the player stands in one spot for too long, whilst the other is used to manage the general audio of the game, such as sound effects and background music. So let's start looking at them.

RandomWait - Script:

```
public class RandomWait : MonoBehaviour
{
    //This function is used to play a random Voice clip when the player stands still for too long.

    //This takes the 3D player's script to check how long they have been standing still for.
    [SerializeField] FPS_Movement FPSM;

    //This stores and array of VoiceActing which are randomely chosen from
    [SerializeField] VoiceActing[] VA;
    //This stores the timings of the wait.
    [SerializeField] float waitmax;
    [SerializeField] float currentwait;


    //The FixedUpdate() waits for the player to stand still long enough before playing one of the voice clips at random.
    private void FixedUpdate()
    {
        if(FPSM.moving == false && FPSM.gameObject.activeSelf == true)
        {
            if (currentwait >= waitmax)
            {
                currentwait = 0;
                int arraynum = Random.Range(0, 2);
                VA[arraynum].AddLine();
            }
            else
            {
                currentwait += 0.1f;
            }
        }
        else
        {
            currentwait = 0;
        }
    }
}
```

This script has a few variables to get the random voice clip system working correctly, which are:

FPSM (FPS_Movement) - This is the 3D Player's movement and is used to check the time that the player has been standing still for.

VA[] (VoiceActing) - This stores an array of the Voice Acting clips that can be played when waiting a while.

waitmax and currentwait (float) - These store the current time the player has been standing still for, and the max time spent waiting.

These variables are then used in the FixedUpdate() function, which plays the voice clip after not moving for a good while, and only whilst the 3D player is active. This is all this script does, and it works effectively, randomly selecting a voice clip to play after waiting too long.

audiomanager - Script:

```
public class audiomanager : MonoBehaviour
{
    //This script is used to manage the audio of the game.

    //This is the audiosource for the background music
    [SerializeField] AudioSource Music;
    //This is an array of songs that the game swaps between after a song ends
    [SerializeField] AudioClip[] Songs;
    //This stores the titles of all these songs.
    [SerializeField] string[] SongTitle;
    //This stores the current number of song played, and the total song count
    [SerializeField] int SongNumber;
    [SerializeField] int TotalSongs;

    //This is the name of the song credit which displays on screen when a new song plays
    [SerializeField] Animator NameDisplayer;
    [SerializeField] TextMeshProUGUI NameText;
```

```
 //This bool checks if the player is moving
    public bool Walking;
    //These are the sound effects played by the game.
    public GameObject Grab;
    public GameObject WalkingObject;
    public GameObject Place;
    public GameObject RockPlace;
    public GameObject RockDirection;
    public GameObject RockThrow;
    public GameObject Jump;
    public GameObject Crack;
    public GameObject Dialogue;
    public GameObject Projector;

    //This bool is true when it is time to swap to a new song.
    [SerializeField] bool swapsong;

    // Start is called before the first frame update
    void Start()
    {
        //The start function stores the total count of the songs to play before playing the first song
        TotalSongs = Songs.Length - 1;
        ChangeSong(0);
    }

    // Update is called once per frame
    void Update()
    {
        //The Update() function waits for the current song to stop playing befor eit swaps to a new song, or if
the player presses P
        if(Music.gameObject.activeSelf != false)
        {
            if(Input.GetKeyDown(KeyCode.P))
            {
                swapsong = true;
            }
            if (!Music.isPlaying || swapsong == true)
            {
                swapsong = false;
                ChangeSong(SongNumber += 1);
            }
        }
        //The footsteps sound effect is effected by if the player is walking or not.
        WalkingObject.SetActive(Walking);

        //Every other audio clip is activated by other scripts that want to play the song.
        if(Grab.activeSelf == true && !Grab.GetComponent<AudioSource>().isPlaying)
        {
            Grab.SetActive(false);
        }

        if(Place.activeSelf == true && !Place.GetComponent<AudioSource>().isPlaying)
        {
            Place.SetActive(false);
        }

        if (RockPlace.activeSelf == true && !RockPlace.GetComponent<AudioSource>().isPlaying)
        {
            RockPlace.SetActive(false);
        }

        if (RockDirection.activeSelf == true && !RockDirection.GetComponent<AudioSource>().isPlaying)
        {
            RockDirection.SetActive(false);
        }

        if (RockThrow.activeSelf == true && !RockThrow.GetComponent<AudioSource>().isPlaying)
        {
            RockThrow.SetActive(false);
        }
        if (Jump.activeSelf == true && !Jump.GetComponent<AudioSource>().isPlaying)
        {
            Jump.SetActive(false);
        }

        if (Crack.activeSelf == true && !Crack.GetComponent<AudioSource>().isPlaying)
        {
            Crack.SetActive(false);
        }

        if (Projector.activeSelf == true && !Projector.GetComponent<AudioSource>().isPlaying)
        {
            Projector.SetActive(false);
        }

        if (Dialogue.activeSelf == true && !Dialogue.GetComponent<AudioSource>().isPlaying)
        {
            Dialogue.SetActive(false);
        }
    }

    //This function is called to swap the currently played song
    public void ChangeSong(int Number)
    {
        Music.Pause();
        if(Number > TotalSongs)
        {
            SongNumber = 0;
        }
        else
        {
            SongNumber = Number;
        }
        Music.clip = Songs[SongNumber];
        Music.Play();
        StartCoroutine(SongNameShow());
    }

    //This has a UI popup appear showing the credit behind the playing song.
    public IEnumerator SongNameShow()
    {
        NameText.text = SongTitle[SongNumber];
        NameDisplayer.SetBool("Show", true);
        yield return new WaitForSeconds(5f);
        NameDisplayer.SetBool("Show", false);
    }
}
```

This is the script used to manage the audio within the scene, and as such has several audio clips attached to it as a variable. These variables are as follows:

Music (AudioSource) - This is the Audiosource that plays the game's music.

Songs[] **and SongTitle**[] (AudioClip and string) - These are two arrays, the first stores the song files for the game, and the second stores the title and composer for the songs.

SongNumber and TotalSongs (int) - These int store the current array number of the Songs array is playing, and calculates the total length of the array.

NameDisplayer and NameText (Animator and NameText) - This stores the animator used to move the text showing the title of the song onto the screen.

Walking (bool) - This bool stores if the 3D player is currently moving or not, and is used to activate the footsteps sound effects.

Grab, WalkingObject, Place, RockPlace, RockDirection, RockThrow, Jump, Crack, Dialogue and Projector (GameObjects) - These gameobjects all store the audiosources for sound effects in the game.

swapsong (bool) - This bool forcibly swaps over the song playing.

The start() function for this script is used to calculate the total amount of songs to swap between, before it calls the ChangeSong() function to play the first song. The Update() function is then used to check whether the current song has stopped playing (or if the player has pressed the P key) so that the script can swap to a new song, using the ChangeSong() function. The Update() function then keeps the footsteps sound effect active based on if the 3D player is moving or not. The rest of the Update() function is used for the other sound effects, each of which only play when called by the appropriate script.

The ChangeSong() function works to change the song currently playing, pausing the music currently playing beforehand. The song will either swap to the next in the array, or, if the current song is the last in the list, will go back to the first song in the list, and will play the song. When a song plays, the SongNameShow() IEnumerator plays, which has some text move on screen to show the credit behind the currently playing song. This script works well to effectively manage the non-voice acting related audio in the game, and is very effective.

Inventory and UI:

This section of the document is reserved for the game's Inventory System, which is a large system incorporating several scripts to get things working. This system involves large pieces of UI that appear on screen to show the items the player is holding, so as such this section of the document will also work to explain the remaining pieces of UI in the game not discussed in any of the other sections. So, let's start looking at these scripts.

ItemClass and Inventory:

Each Inventory item collected by the player is stored as an ItemClass variable, which is stored within the Inventory script. The Inventory script allows the player to swap between items stored in the Inventory, and is called to open the Inventory. So, let's look at these scripts.

ItemClass - Script:

```
[System.Serializable]
public class ItemClass
{
    //This creates a class to store the variables stored in inventory items
    public string ItemName;
    public Sprite ItemIcon;
    public GameObject ItemGameobject;

}
```

This script creates a new Variable class which is used to store the values of the items picked up by the player. The variables it stores within the class are as follows:

ItemName (string) - This stores the name of the item stored.

ItemIcon (Sprite) - This stores the icon that appears in the inventory items when this item is held.

ItemGameobject (GameObject) - This stores the in game item that is picked up by the player.

This class works well to easily allow for storage of the information crucial for the Inventory to work. The Inventory system is very specific about what names items are stored as, as the name is used to check if the item can be placed in item placing spots. This is used in the game for some puzzles, and gives a good flow to the gameplay.

Inventory - Variables, Start() and Update():

```
public class Inventory : MonoBehaviour
{
    //This script is used to store the items of the inventory that the player is holding

    //This stores a List of every item in the inventory
    public List<ItemClass> Items;

    //This stores the size of the list of Inventory Items
    public int InventoryCount;

    //This stores which element of the Items List is currently selected
    public int CurrentInventory;

    //This stores the Animation for the Current item icon
    [SerializeField] GameObject AnimObject;

    //This stores the ButtonLocation script
    [SerializeField] ButtonLocation BL;

    //This stores the Button prefab used to open the menu
    [SerializeField] GameObject Button;

    // Start is called before the first frame update
    void Start()
    {
        //this sets the icon of the current item to be that which is held currently by the player
        AnimObject.GetComponent<InventoryElement>().BubbleItem = Items[CurrentInventory];
    }

    // Update is called once per frame
    void Update()
    {
        //This script is used to store the inventory and what item is currently being held
        AnimObject.GetComponent<InventoryElement>().BubbleItem = Items[CurrentInventory];
        InventoryCount = Items.Count;

        //The script takes in the mouse scroll data to quickly swap between inventory items
        Vector2 MouseScroll = Input.mouseScrollDelta;
        float MouseScrolly = MouseScroll.y;

        //The player can either press the square brackets or the mouse scroll to swap between inventory items
        if (Input.GetKeyDown(KeyCode.LeftBracket) || MouseScrolly < 0)
        {
            ChangeNumber(false);
        }
        else if (Input.GetKeyDown(KeyCode.RightBracket) || MouseScrolly > 0)
        {
            ChangeNumber(true);
        }
    }
```

The Inventory item takes in the information of items picked up by the player, and displays them on screen via the inventory menu. The variables used for this are as follows:

Items<> (ItemClass) - This List of the ItemClass variable stores every item picked by the player. It will always have the "NoItem" variable stored within it, for when the player isn't holding anything.

InventoryCount and CurrentInventory (int) - These ints store the max size of the inventory, as well as the value for the current inventory item held by the player.

AnimObject (GameObject) - This stores the GameObject for the Inventory icon showing the currently held item. This originally animated into the UI which is why its called AnimObject.

BL (ButtonLocation) – This stores the ButtonLocation script, which is used to open up the Inventory menu.

Button (GameObject) - This stores the Button prefab used to create the items in the Inventory Menu when it's opened.

The Start() function works to set the value of the item shown in AnimObject as the first in the list. The value of the item shown in AnimObject gets updated each frame in the Update() function, which sets it to the current inventory item held by the player. The game takes in the input of the player, using either the square brackets or the mouse scroll, to go up or down the list of inventory items, by calling the ChangeNumber() function.

Inventory – OpenInventoryB() and ChangeNumber():

```
    //When the inventory is opened by the 3D player the script makes a Inventory Item button for each element stored in the inventory, and adds it to an list stored in the
ButtonLocation script.
    public void OpenInventoryB(bool open)
    {
        switch(open)
        {
            case true:
                {
                    int Count = 0;
                    foreach (ItemClass I in Items)
                    {
                        BL.Buttons.Add(Instantiate(Button, BL.DefaultPos.GetComponent<RectTransform>().position, BL.DefaultPos.GetComponent<RectTransform>().rotation));
                        BL.Buttons[Count].GetComponent<RectTransform>().SetParent(BL.gameObject.GetComponent<RectTransform>(), true);
                        BL.Buttons[Count].GetComponent<RectTransform>().localScale = BL.InventoryItem.GetComponent<RectTransform>().localScale;
                        BL.Buttons[Count].gameObject.GetComponent<ButtonInstance>().BubbleItem = Items[Count];
                        BL.Buttons[Count].gameObject.GetComponent<ButtonInstance>().InventoryNumber = Count;
                        Count += 1;
                    }
                    BL.Move = 1;
                }
                break;
            case false:
                {

                    //When this function is called to close the inventory the ButtonLocation script is called to close the inventory
                    BL.Move = 4;
                }
                break;
        }

    }


    void ChangeNumber(bool Up)
    {
        //This function is used to add or decrease the value of the inventory, so the player can swap between items
        switch(Up)
        {
            case true:
            {
                    if(CurrentInventory == (InventoryCount - 1))
                    {
                        CurrentInventory = 0;
                    }
                    else
                    {
                        CurrentInventory += 1;
                    }
            }
            break;
            case false:
            {
                    if (CurrentInventory == 0)
                    {
                        CurrentInventory = (InventoryCount - 1);
                    }
                    else
                    {
                        CurrentInventory -= 1;
                    }
                }
            break;
        }
```

The OpenInventoryB() function is called by the 3D player in order to open the inventory. Initially, this function opened a huge menu where the player slowly swapped from each item in the list, but this was updated to instead open a menu of buttons the player can easily click to select what item they wish to hold. This is now done by instantiating instances of the prefab stored in Button, with each displaying and storing the values of each item stored in the inventory, which get added to the List of buttons in the ButtonLocation script. If the player presses to close the inventory, then the script sets the ButtonLocation script to the settings to close the menu. The ChangeNumber() function is called to change what item in the inventory is currently being held. When the player moves the list up it will either add 1 to the value of CurrentInventory or, if the player is at the end of the list already, the script will go back to the start of the list. When going down the list the opposite of this is done, with the value of CurrentInventory either being subtracted by 1 or is sent to the max count of the inventory. This Inventory system has been well refined to create a simple and easy to use inventory menu, which I am very proud of.

ButtonInstance, InventoryElement and ButtonLocation:

These scripts are all important for displaying the information stored in the Inventory to the player, each working to keep the game flowing well. The ButtonInstance and InventoryElement scripts both work to display information stored in the Inventory to the player on icons in the UI, with the ButtonInstance working to display it in buttons in the Inventory menu, whilst InventoryElement is used to display it on the Inventory Icon that displays throughout gameplay. The Inventory Menu itself is opened when the Inventory script calls the ButtonLocation script, which creates a list of buttons that the player can press to hold a different item. So, let's start looking at these scripts.

ButtonInstance – Script:

```
public class ButtonInstance : MonoBehaviour
{
    //This is used to store values to the instances of the Inventory Buttons, giving the player the stored item when clicked on

    //These are used to display the information stored in the ItemClass variable for the object.
    public ItemClass BubbleItem;
    public Image Icon;
    public TextMeshProUGUI Name;

    //These store the Iventory and ButtonLocation scripts
    [SerializeField] Inventory I;
    [SerializeField] ButtonLocation Bl;

    //This stores what value of the Inventory list that this item is at.
    public int InventoryNumber;

    // Start is called before the first frame update
    void Start()
    {
        //The Start() function attaches the Inventory and ButtonLocation script to their variables.
        I = GameObject.FindGameObjectWithTag("Inventory").GetComponent<Inventory>();
        Bl = GameObject.FindGameObjectWithTag("Button").GetComponent<ButtonLocation>();
    }

    // Update is called once per frame
    void Update()
    {
        //The Update() function works to display the values stored in the variables to the button
        if (I.Items[InventoryNumber].ItemName != "No Item")
        {
            Icon.gameObject.SetActive(true);
            Icon.sprite = I.Items[InventoryNumber].ItemIcon;
        }
else
        {
            //If the Item is "No Item" then that means no icon is displayed
            Icon.gameObject.SetActive(false);
        }
        Name.text = I.Items[InventoryNumber].ItemName;
    }

    public void ClickInventory()
    {
        //When the player clicks the button, the value of the Inventory script's CurrentInventory is set to the InventoryNumber stored in the instance.
        I.CurrentInventory = InventoryNumber;
    }
}
```

The ButtonInstance script works to store the data from each item in the Inventory to a button that can be pressed by the player in the Inventory Menu, with the data for each instance being stored by the Inventory script when calling the Inventory Menu. The variables for this script are:

BubbleItem, Icon, Name (ItemClass, Image, TextMeshProUGUI) - These variables work to display all the information stored within the ItemClass variable attached to this instance.

I and BL (Inventory and ButtonLocation) - These variables store the Inventory and ButtonLocation scripts respectively.

InventoryNumber (int) - This stores what number of the Inventory item list in the Inventory script this item is representing is.

This script works to set display the icon and name associated with the stored Inventory item onto the button's design. If the item held is the "No Item", then no icon will be displayed. The item this instance of the script then serves as a button, which when pressed sets the current item held by the inventory to be whatever value of the ItemClass item stored on the button currently is in the Inventory script. This works well to easily make interactable buttons within the Inventory menu.

InventoryElement - Script:

```
public class InventoryElement : MonoBehaviour
{
    //This script is used to store the information stored in the currently held Inventory Item

    //This stores the data displayed by the currently held inventory item
    public ItemClass BubbleItem;
    public Image Icon;
    public TextMeshProUGUI Name;

    //This stores the Inventory script.
    [SerializeField] Inventory I;

    // Update is called once per frame
    void Update()
    {
        //This script works to present the icon and data for the currently held inventory item
        if (I.Items[I.CurrentInventory].ItemName != "No Item")
        {
            Icon.gameObject.SetActive(true);
            Icon.sprite = I.Items[I.CurrentInventory].ItemIcon;
        }
        else
        {
            Icon.gameObject.SetActive(false);
        }
        Name.text = I.Items[I.CurrentInventory].ItemName;

    }
}
```

The InventoryElement is designed to fulfil a purpose almost identical to that of the ButtonInstance script, though instead of showing an item assigned to it by the Inventoy script, the script works to show the item that is currently held by the player, based off the value of the CurrentInventory int. This script only works to show this item during gameplay, when the player isn't in the Inventory menu, and is updated as the game progresses. This script also doesn't work as a button, so it only serves to be looked at, not interacted with.

ButtonLocation - Variables, Start() and Update():

```
 public class ButtonLocation : MonoBehaviour
{
    //This script is used to store the positions and list for each button presented in the Inventory Menu when it's open.

    //This stores a List of each button in the menu
    public List<GameObject> Buttons;
    //This stores the position that each button needs to move towards
    public GameObject[] ButtonPos;
    //This is used to call how the buttons should move
    public int Move;
    //This stores the number of buttons present in the List.
    [SerializeField] int BCount;
    //This stores the speed at which the buttons should move
    [SerializeField] float Movespeed;
    //This stores the defualt position the buttons should move back to
    public GameObject DefaultPos;

    //This stores the 3D Player
    [SerializeField] FPS_Movement FPSM;

    //This stores the InventoryItem that appears on screen when the Inventory menu isn't open
    public GameObject InventoryItem;

    //This stores the Text that displays telling the player how to close the menu
    [SerializeField] GameObject Close;

    //This stores the CrossHairs in the game
    [SerializeField] GameObject CR;

    // Start is called before the first frame update
    void Start()
    {
        //The start function starts by making sure the Inventory text is not active.
        Close.SetActive(false);
    }

    // Update is called once per frame
    void FixedUpdate()
    {
        //The FixedUpdate() is used to check where in the movement process the menu is at
        switch (Move)
        {
            case 1:
                {
                    //When Move is 1 it means the menu was just opened, it works to count each item stored in the list
                    InventoryItem.SetActive(false);
                    Close.SetActive(true);
                    BCount = 0;
                    foreach (GameObject G in Buttons)
                    {
                        BCount += 1;
                    }
                    Move = 2;
                }
                break;
            case 2:
                {
                    //When Move is 2 it means the buttons are moving over the scene
                    MoveButtons(BCount);
                }
                break;
            case 4:
                {
                    //When Move is 4 it means the buttons are moving back to close the menu
                    MoveButtonsBack(BCount);
                }
                break;
            case 5:
                {
                    //When Move is 5 it sets up the game to go back to movement, with the menu being reset and closed
                    Cursor.lockState = CursorLockMode.Locked;
                    Cursor.visible = false;
                    FPSM.CanMove = true;
                    CR.SetActive(true);
                if (Buttons.Count > 0)
                {
                    foreach (GameObject G in Buttons)
                    {
                        Destroy(G);
                    }
                    Buttons.Clear();
                }
                InventoryItem.SetActive(true);
                Close.SetActive(false);
                Move = 0;
                FPSM.PressI = true;

            }
            break;
    }
}
```

The ButtonLocation script is used to move UI buttons for the Inventory menu on and off the screen. This script, has the following variables, which are called by the Inventory script:

Buttons<> (GameObject) - This list stores each of the UI buttons used for the Inventory menu.

ButtonPos[] (GameObject) - This array of GameObjects store the positions that the items in the Buttons list need to move towards.

Move and BCount (int) - The Move int is used to register what stage of movement the buttons are at, whilst BCount stores the number of buttons listed in the Buttons list.

Movespeed (float) - This stores the speed at which the buttons should move on screen.

DefaultPos (GameObject) - This stores the default positions that the buttons move back to when the Inventory menu is closed.

InventoryItem, Close, CR (GameObject) - These store the the Inventory icon, Inventory Menu text, and game's cross hair in GameObjects to activated and deactivated as the menu opens and closes.

The FixedUpdate() is where the script takes in the value of the Move int, and uses that to determine what phase of opening the Inventory menu currently is at. When Move is 1, that means the menu is being set up, and the function works to count how many buttons are stored in the Buttons list, putting this value into the BCount variable. It then changes the value of Move to 2. When Move is 2, that means the buttons are moving into their places on screen, and the MoveButtons() function is called. When the value of move is 4 that means the buttons need to move back to their starting position so the menu can close, using the MoveButtonsBack() function. When Move is 5, that means the menu is closing, and the script sets the game back so the player can continue moving, and clearing the values of the Buttons list, so that it is ready for the Inventory to be open again.

ButtonLocation – MoveButtons() and MoveButtonsBack():

```
    void MoveButtons(int bCount)
    {
        //This function moves each button towards their position one at a time. When the furthest button reaches its position it means the movement is complete
        for (int i = 0; i < bCount; i++)
        {
            if (i > 0)
            {
                Buttons[i].GetComponent<RectTransform>().localPosition = Vector3.MoveTowards(Buttons[i].GetComponent<RectTransform>().localPosition, ButtonPos[i].GetComponent<RectTransform>().localPosition, Movespeed);
            }
        }

        int value = bCount -= 1;
        if (Buttons[value].GetComponent<RectTransform>().localPosition == ButtonPos[value].GetComponent<RectTransform>().localPosition || Buttons.Count == 1)
        {
            Move = 3;
        }
    }

    void MoveButtonsBack(int bCount)
    {
        //This function moves each button back towards the start when the menu is being closed, when the furtherst button reaches the start position it means the menu has been closed.
        for (int i = 0; i < bCount; i++)
        {
            if (i > 0)
            {
                Buttons[i].GetComponent<RectTransform>().localPosition = Vector3.MoveTowards(Buttons[i].GetComponent<RectTransform>().localPosition, DefaultPos.GetComponent<RectTransform>().localPosition, Movespeed);
            }
        }

        int value = bCount -= 1;
        if (Buttons[value].GetComponent<RectTransform>().localPosition == DefaultPos.GetComponent<RectTransform>().localPosition || Buttons.Count == 1)
        {
            Move = 5;
        }
    }
}
```

The MoveButtons() and MoveButtonsBack() functions achieve practically the same thing, just in opposite ways. MoveButtons() works to move the UI Buttons for the Inventory Menu into place, registering that they have completed their job when the further button reaches their position. The MoveButtonsBack() function does the opposite thing, moving all the UI Buttons back to the start position when the Inventory menu needs to close, registering it as complete when the further button reaches the start point, at which it is registered that the menu is closed. This script works to make a really simple and efficient menu for the Inventory, and makes gameplay work really smoothly.

ItemPickup, mouseclick and Vine:

With the Inventory system explained, I will now explain how the player is able to add items to their inventory, using the ItemPickup script to add items to the list stored in the Inventory script. These inventory items are then able to be placed using the mouseclick script, which allows the player to place items down at certain spots, assuming the player is placing an appropriate item. Finally, the Vine script works to have the place inventory items effect the 2D space near where they were placed. These scripts are quite important, so lets start looking at them.

ItemPickup – Script:

```
public class ItemPickup : MonoBehaviour
{
    //This script is used to pick up items to add them to the player's inventory

    //This stores the ItemClass variable fo the item picekd up
    [SerializeField] ItemClass Item;

    //This stores the Inventory script
    [SerializeField] Inventory I;
    //This stores the Inventory object in the scene
    [SerializeField] GameObject Object;

    //This stores the PressE pop up
    [SerializeField] GameObject PressE;
    //This stores the audio manager script.
    [SerializeField] audiomanager AM;
```

```
// Start is called before the first frame update
    void Start()
    {
        AM = GameObject.FindGameObjectWithTag("AControl").GetComponent<audiomanager>();
    }

    //This script is used to pick up items from the scene and add them to the inventory
    public void EnterRay()
    {
        //When the raycast collides with the item it allows the player to add the item to their inventory by pressing E, which removes the item from the scene
        and adds it to their inventory.
            PressE.SetActive(true);
            Debug.Log("TriggeredWithItem");
            if(Input.GetKey(KeyCode.E))
            {
                if(Object.activeSelf == true)
                {
                    AM.Grab.SetActive(true);
                    AM.Grab.GetComponent<AudioSource>().Play();
                    I.Items.Add(Item);
                    Object.SetActive(false);
                    PressE.SetActive(false);
                }
            }
    }
```

This script is small, and works to add whatever ItemClass variable is stored to it into the list in the Inventory script. The scripts variables are as follows:

Item (ItemClass) - This stores the Item which the player picks up when interacting with this item.

I and Object (Inventory and GameObject) - These store the Inventory script and the GameObject in the scene which is picked up by this script.

PressE and AM (GameObject and audiomanager) - This stores the Press E pop-up and the audiomanager script.

The function of this script is called by the EnterRay() function, which is called when the Camera's Raycast collides with this script. This function, when the player presses E, adds the stored Inventory item to the Inventory scripts Items list, before removing the gameobject from the scene. It is short and effective, easily allowing for new inventory items to be placed in the scene.

mouseclick - Variables, Start() and Place():

```
public class mouseclick : MonoBehaviour
{
    //This script is used to place items stored in the Inventory down into the scene.

    //These store the object and name of the item that can be placed.
    [SerializeField] GameObject[] AppearObjects;
    [SerializeField] ItemClass[] AppearNames;

    //This stores the ItemClass variable for the item placed in the scene.
    [SerializeField] ItemClass TempItem;
    //this checks if an item has already been placed in the area.
    [SerializeField] bool Clicked;
    //This bool is set to true when an item is being placed or removed, so the player can't
    [SerializeField] bool DoEvent;

    //These store the scripts needed for placing these items down
    [SerializeField] Inventory I;
    [SerializeField] audiomanager AM;
    [SerializeField] InventoryElement IB;

    //This audio clip plays if the player trys to place items when they aren't holding anything
    [SerializeField] VoiceActing Nothingtoplace;

    // Start is called before the first frame update
    void Start()
    {
        //The start function sets things up so none of the AppearItems are active during the start.
        AM = GameObject.FindGameObjectWithTag("AControl").GetComponent<audiomanager>();
        foreach (GameObject I in AppearObjects)
        {
            I.SetActive(false);
        }
}

    public void Place()
    {
        //This function is called by the camera Raycast, and allows the player to either add or remove an inventory item in the space.
        if (Input.GetKeyDown(KeyCode.E))
        {
            AM.Place.SetActive(true);
            AM.Place.GetComponent<AudioSource>().Play();
                if (Clicked == false && DoEvent == false)
                {
                    GiveVine();
                }
                else if (Clicked == true && DoEvent == false)
                {
                    TakeVine();
                }
        }
    }
```

This script is used to place items held by the player down into the scene. It originally did this by having the player actually click to place things via the mouse, but this was eventually changed to be done via the camera raycast instead. the variables for this script are as follows:

AppearObject[] **and AppearNames**[] (GameObject and ItemClass) - These arrays are used to store the items that can be placed by the player, as well as the names that items placed need to have in order to be placed.

TempItem (ItemClass) - This stores the ItemClass value for the item placed in the spot.

Clicked and DoEvent (bool) - The clicked bool is use to check if an item has already been placed in the spot or not, and the DoEvent bool is called when an item is being placed or removed, so the player can't spam the function.

I, AM, IB (Inventory, audiomanager and InventoryElement) - these variables store the scripts needed for the item to be placed in the spot.

Nothingtoplace (VoiceActing) - This voice clip is played when the player tries to place an item down with nothing in their inventory.

The Start() function is used to set up the placed items, making sure each is deactivated at the start of the game. Items can be placed and removed via the Place() function, which is called by the CameraRaycast when it collides with the placement point. When the player presses E whilst the spot is looked at, the player can add their inventory item down or take back a placed inventory item. This is done via the GiveVine() and TakeVine() functions, which add and remove items to the spot respectively.

mouseclick - GiveVine(), TakeVine() and DeleteItem():

```
void GiveVine()
{
    //This function checks what item the player is trying to place compared to the items that can be placed.
    int Count = 0;
    bool Delete = false;
    if (I.Items.Count > 1)
    {
        foreach (GameObject I in AppearObjects)
        {
            if (IB.BubbleItem.ItemName == AppearNames[Count].ItemName)
            {
                //If the player can place the item then it will added to the scene
                I.SetActive(true);
                TempItem = AppearNames[Count];
                Clicked = !(Clicked);
                DoEvent = true;
                Delete = true;
            }
else
            {
                I.SetActive(false);
            }
            Count += 1;
        }

        if (Delete == true)
        {
            //When placing an item it is removed from the player's inventory
            I.CurrentInventory = 0;
            DeleteItem();
        }
        DoEvent = false;
    }
    else
    {
        Nothingtoplace.AddLine();
    }
}

void TakeVine()
{
    //This function is used to add a vine back to the player's inventory
        DoEvent = true;
        AppearObjects[0].SetActive(false);
        AppearObjects[1].SetActive(false);
        AppearObjects[2].SetActive(false);

        I.Items.Add(TempItem);
        TempItem = null;
        Clicked = false;
        DoEvent = false;
}

void DeleteItem()
{
    //This function is used when adding a vine to the scene, removing it from the inventory list
    int i = 0;
    int Deletei = 0;
    bool Delete = false;
    foreach(ItemClass IT in I.Items)
    {
        if(IT.ItemName == TempItem.ItemName)
        {
            Deletei = i;
            Delete = true;
        }
        i += 1;
    }

    if (Delete == true)
    {
        I.Items.Remove(I.Items[Deletei]);
    }
}
```

The GiveVine() function checks to see if the item held by the player is able to be placed down. This is done via a Foreach loop which checks each item stored in AppearObjects and AppearNames to see if it matched the held item. If it does, the item gets removed from the player's inventory, using the DeleteItem() function, and adds the GameObject associated with it in AppearObjects into the 3D space. The Inventory item gets removed from the list via DeleteItem() which removes the Inventory item from the Items list in the Inventory script, but before removing it, the script saves the placed item in TempItem, for when the player picks it back up again.

TakeVine() is called when the player interacts with a spot which already has an item placed within it, and adds the item stored in TempItem back into the Inventory script's Items list, whilst also removing the item from the 3D space. This script is designed to easily create new spots that can place specific items, without needing to change the script to do so. Each function is named after Vines, as for a good while the only instance of the Inventory script was used to place vines down.

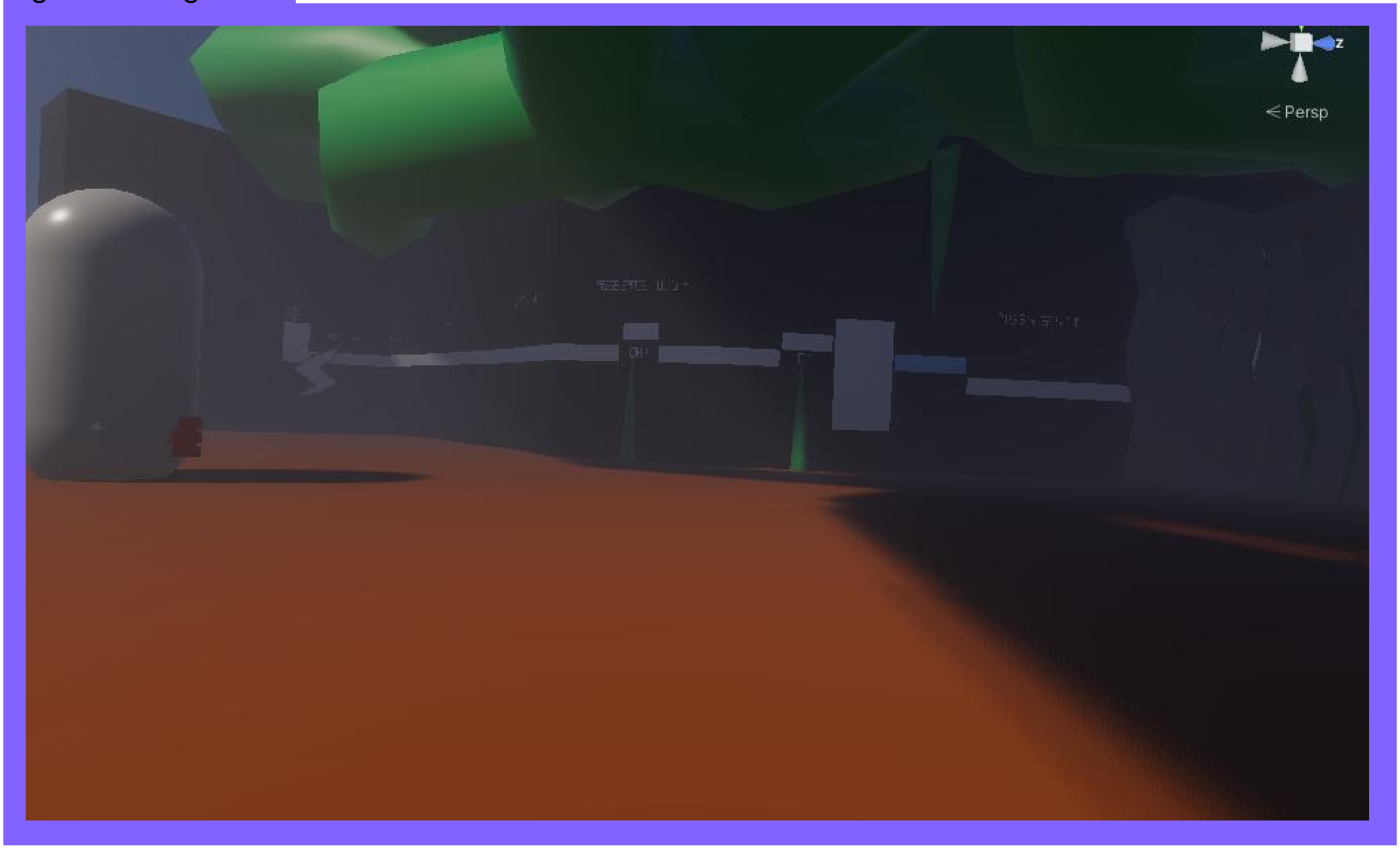
Vine - Script:

The vine script is the final one needed to manage the inventory system, and is so simple I can explain it in this one paragraph. The script checks to see if an item placed by the player is active, and if it is it will also activate new platforms in the 2D space. If the 3D object associated with the script isn't there, then it means no platform will appear in the 2D space. This script is incredibly simple, but is the key piece of code which allows placed items to affect the 2D space. It's named Vine as it was originally only used to place vines down in the second segment.

Segment 1 - Introduction to Game:

The first segment of the game serves as a general introduction to the game, giving the player a space to become antiquated with the 2D and 3D gameplay styles. This area introduces a few simple mechanics and scripts which will be explained in this segment. The scripts discussed in this section are as follows: AtariAbilityTrigger, DestroyTrigger, DecalTrigger and MovingPlatform2D .

Figure 2 - Segment 1



AtariAbilityTrigger and DestroyTrigger:

I am linking these two scripts in one segment as they are both incredibly simple scripts used for only one or two objects in the game that only really serve to check when the player has triggered with them. The AtariAbilityTrigger script activates the Atari Jump ability that's stored in AtariAbilitySave. The DestroyTrigger script destroys a gameobject stored in the scene when triggered, and is only really used for this starting area.

AtariAbilityTrigger - Variables:

```
public class AtariAbilityTrigger : MonoBehaviour
{
    [SerializeField] AtariAblilitySave AAS;
    [SerializeField] CornerTrigger CT;
}
```

This script only has two variables, and is used to trigger the Atari jump ability so that the player can jump. This script also sets it so the camera doesn't change position at the first corner of the first area. The variables are as follows:

AAS (AtariAbilitySave) - The Variable that stores the AtariAbilitySave scriptable object.

CT (CornerTrigger) - The variable that stores the values of the first Corner Trigger of the first decal area.

AtariAbilityTrigger - OnTriggerEnter():

```
//When Entering the trigger the game will allow the Atari player to jump
private void OnTriggerEnter(Collider other)
{
    if (other.gameObject.tag == "Player")
    {
            //This script also sets it so the camera in the first area no longer moves around in scale.
            CT.CameraB = false;
            AAS.AtariJump = true;
            Destroy(gameObject);

    }
}
```

After triggering with the player, this script activates the jump before deleting itself.

DestroyTrigger:

```
public class DestroyTrigger : MonoBehaviour
{
    [SerializeField] GameObject DestroyGObject;
    //This script destroys a object when the player enters its trigger.
    private void OnTriggerEnter(Collider other)
    {
        if (other.gameObject.tag == "Player")
        {
            if (DestroyGObject != null)
            {
                Destroy(DestroyGObject);
                Destroy(gameObject);
            }
        }
    }
}
```

This script only has one variable, being:

DestroyGObject (GameObject) - This variable stores the gameObject that will be destroyed on trigger.

On trigger this script will check to see if the object has been destroyed yet, and if it hasn't it will destroy the object, before the script destroys itself.

DecalTextAppear:

This area is the first to incorporate the bits of Decal text that fade into the scene when triggered, which is seen throughout the game. These are used to show the inner monologue of the protagonist as well as to explain tutorials for new mechanics, whilst keeping things contextual within the 3D space.

DecalTextAppear - Variables and Functions:

```
public class DecalTextAppear : MonoBehaviour
{
    [SerializeField] DecalProjector DP;

    public float FadeCount;
    public bool Triggered;

    public bool SetOn;

    // Start is called before the first frame update
    void Start()
    {
        DP.fadeFactor = 0;
    }

    // Update is called once per frame
    void Update()
    {
        FadeCount = DP.fadeFactor;

        if(SetOn == true)
        {
            SetOn = false;
            StartCoroutine(Fadein());
        }
    }
    //Causes a decal to fade in when entering the trigger.
    private void OnTriggerEnter(Collider other)
    {
        if (other.gameObject.tag == "Player" || other.gameObject.tag == "Pollen" || other.gameObject.tag == "EvilPollen")
        {
            if(Triggered == false)
            {
                StartCoroutine(Fadein());
            }
```

This script has several variables which are used to effectively fade in the Decals in the scene. The variables are as follows:

DP (DecalProjector) - This variable stores the projector for the specific decal that fades onto the scene.

FadeCount (float) - This variable works to store the current value of DP's fadefactor, used for debugging purposes within the Editor.

Triggered (bool) - This bool is used to check whether the player has already triggered the collider for the Decal Text appear.

SetOn (bool) - This bool is used to check if the decal has been called to fade on in the Update() function.

The Start() function for this script is simple and only does one thing, being to set the value of the DecalProjector's FadeFactor to 0, so that the decal start off invisible. The Update() and OnTriggerEnter() function both serve the same purposes - to start the fading process by calling the FadeIn() IEnumator. The difference is how these functions are incorporated. The TriggerEnter() function is used to call it when the 2D player or the 2D pollen enters the trigger. Meanwhile the Update() function calls the IEnumerator within other scripts, such as the EventsCode script.

DecalTextAppear - Fadein():

```
//This IEnumerator is called to fade the Decal Projection on screen.
public IEnumerator Fadein()
{
    Triggered = true;
    //This will keep looping until the projection is fully faded onto the scene.
    if(DP.fadeFactor <= 1)
    {
        DP.fadeFactor += 0.01f;
    }
    yield return new WaitForSeconds(0.01f);

    if(DP.fadeFactor >= 1)
    {
        DP.fadeFactor = 1f;
    }
    else
    {
        StartCoroutine(Fadein());
    }
}
```

This IEnumerator works as a make-shift loop continuously increases the value of the DP's fadefactor, until it is fully visible in the scene. This loop is created in a series of if statements, which check the value of DP's fadefactor. If the fadefactor is less then 1, then it will add 0.01 to the total, before calling the IEnumerator again. Once the value of the fadefactor reaches or exceeds 1, then the IEnumerator is complete, and no longer gets called. This works well to have the decals fade on screen, and can easily have instances of gameObjects within the scene.

MovingPlatform2D:

This area is the first to have the Moving Platforms, which are seen throughout the game, and have a HoldPlatform variant. As the name implies these platforms were only meant to be used within the 2D space, but eventually the design of the game shifted to incorporate them in the 3D space, so the code was adjusted to allow for this. The script was also adjusted later to allow for the end position of the platform to be changed by other scripts and buttons.

MovingPlatform2D - Variables:

```
public class MovingPlatform2D : MonoBehaviour
{

    //These variables store the position for the start and end of the movement
    [SerializeField] Vector3 StartPos, EndPos;
    //These variables are used to save the current start and end of the movement in the script.
    [SerializeField] Vector3 StartSave, EndSave;
    //These variables store the speed of the movement and the count of the lerp.
    [SerializeField] float movelerp, movespeed;

    //This stores the time the platform has to store, and the current save of the wait.
    [SerializeField] float WaitTime;
    [SerializeField] float WaitSave;

    [SerializeField] bool CanMove;

    [SerializeField] bool GoneOnce;

    //This bool when set shows that the platform is a 3D platform.
    public bool Platform3D;
    //This stores the Start of the platform for when the platform's position changes.
    [SerializeField] Vector3 originalstart;

    //This stores the Invisible Walls of 3D object.
    [SerializeField] GameObject Inviswalls;

    [SerializeField] bool Changed;
```

This script has several variables, which are as follows:

StartPos and EndPos (Vector3) - These Vector3s store the starting and ending positions of the platform, and is updated for when the platform moves back and forth.

StartSave and EndSave (Vector3) - These Vector3s save the original starting and ending positions of the platform.

movelerp and movespeed (float) - These floats store the current value of the movement lerp and the speed that the platform moves at.

WaitTime and WaitSave (float) - These floats store the max time that the platform should wait between movements and the float that stores the current time the platform has been waiting for.

CanMove and GoneOnce (bool) - The first of these bools needs to be true for the platform to begin moving, whilst the second is used to register when the platform should begin moving back the way it came.

Platform3D (bool) - This bool, when set, means that the instance of the script is for a platform in the 3D moving space.

originalstart (Vector3) - This Vector3 stores the original starting position of the gameObject, and is used for the 3D platform.

Inviswalls (GameObject) - This variable stores the invisible walls that are used for the 3D platform to make sure the player doesn't fall off the edges.

Changed (bool) - This bool is set to true when the end position of the platform gets changed in the ChangeEnd() function.

MovingPlatform2D - Start(), FixedUpdate() and Update():

```
// Start is called before the first frame update
void Start()
{
    //The start function sets up the movement of the platform
    StartSave = StartPos;
    EndSave = EndPos;
    gameObject.transform.localPosition = StartPos;

    originalstart = StartPos;

}

// Update is called once per frame
void FixedUpdate()
{
    //The movement for the platform is called in the FixedUpdate() so that it is consistent within the Build.
        if (CanMove == true)
        {
            Movement();
        }
}

private void Update()
{
    //The Update() function checks whether it is a 3D platform,
    and if it is, it works to either keep the 3D platforms invisible walls.
    if(Platform3D == true)
    {
        Inviswalls.SetActive(gameObject.transform.localPosition != originalstart);
    }
}
```

The Start() function is used to set up the position of the platform, to make sure it is positioned at the location stored in StartPos. This function also sets up the values of StartSave and EndSave, to be the values stored in StartPos and EndPos. It then sets the value of the originalstart variable to be that of StartPos, which is used in later functions. The Movement for the platform (appropriately named Movement()) is called from within the FixedUpdate() function, so that the speed of the platform's movement is consistent within the Editor and Build.

The Update() function is only needed for the 3D alternatives to the platforms. This function is used to either activate or deactivate the invisible walls surrounding the 3D platform, activating them when the platform is not at the starting position, stored in the orignalstart variable. This works to make sure the player can't fall off the platform whilst its in the air.

MovingPlatform2D - OnCollisionStay(), OnCollisionExit() and ChangeEnd():

```
//The script parents the 2D rock to the platform when it enters the trigger,
so there are no glitches with collision.
private void OnCollisionStay(Collision collision)
{
    if(collision.gameObject.tag == "Rock")
    {
        collision.gameObject.transform.parent = gameObject.transform;
    }
}

private void OnCollisionExit(Collision collision)
{
    if (collision.gameObject.tag == "Rock")
    {
        collision.gameObject.transform.parent = null;
    }
}

//This function is called by other scripts to change the end position of the moving platform.
public void ChangeEnd(Vector3 NewEnd)
{
    if(Changed == false)
    {
        Changed = true;
        StartPos = originalstart;
        EndPos = NewEnd;
        EndSave = NewEnd;

        //The speed of the platform is decreased so that it isn't too fast.
        movespeed = 0.2f;
    }
}
```

The OnTriggerStay() and OnTriggerExit() functions of this script are used to parent and de-parent the 2D Rock objects to the platform. These also originally were used to parent the player themselves to the platforms, though this was later changed to be done in the DecalTriggers script. The ChangeEnd() function is a function called by external scripts, and changes the end position of the platform. This function only changes the values once, no longer working once the changed bool is set to true. This function changes the values of EndPos and EndSave to the new values stored in the NewEnd Vector3, which is set within the function. This causes the platform to teleport back to the start position, but I chose to not waste dev time on fixing this very small issue. The speed of the platform is also decreased, as if it remains the same speed as the platform regularly, then the platform will move too fast.

MovingPlatform2D - Movement():

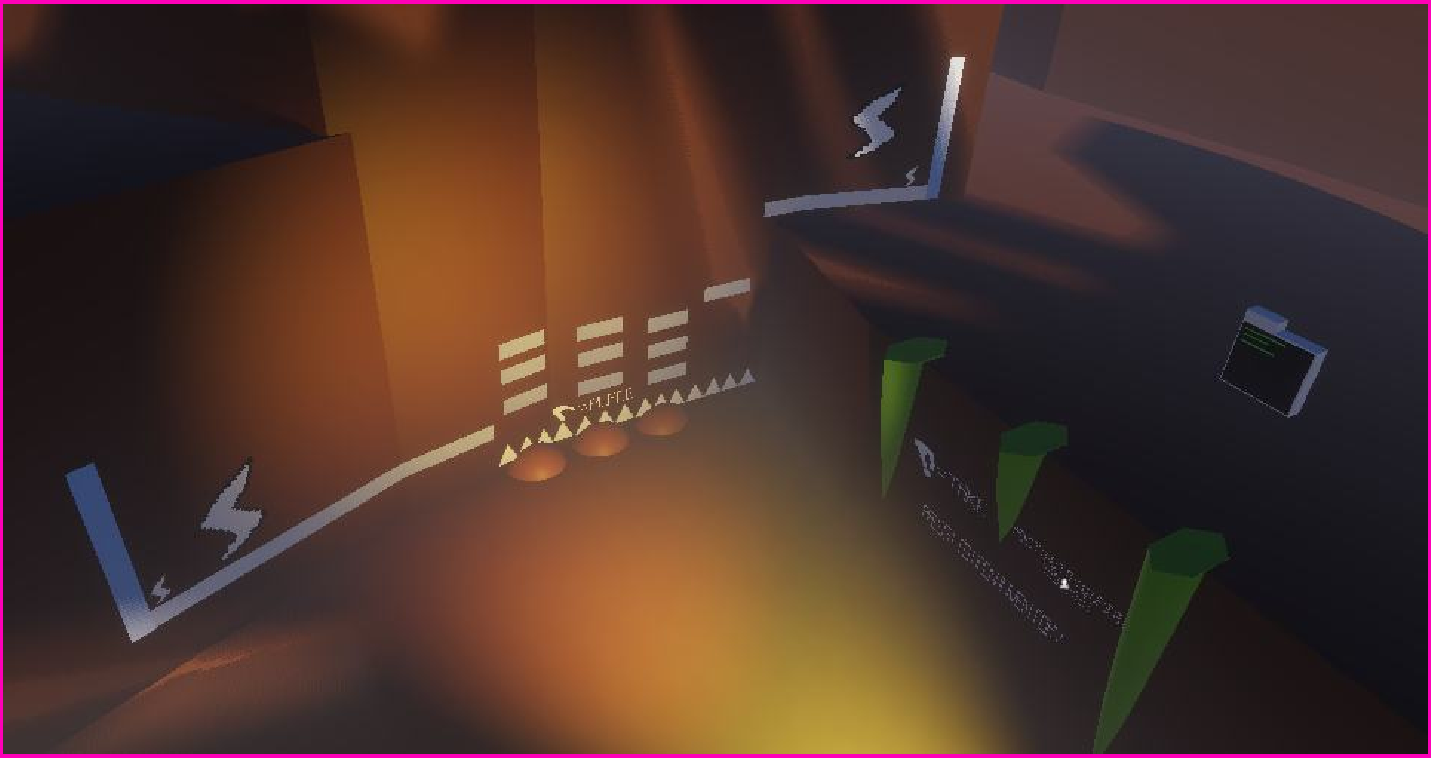
```
//This script is used to create the moving platform, having them move between two locations in a loop
public void Movement()
{
    //This function checks if the platform has already reached the end of the movement
    if(GoneOnce == false)
    {
        movelerp = Mathf.Clamp(movelerp + 1 * Time.deltaTime * movespeed, 0, 1);

        gameObject.transform.localPosition = Vector3.Lerp(StartPos, EndPos, movelerp);
    }

    //This checks the value of MoveLerp, and checks whether the
    platform needs to wait after reaching the end position.
    if (movelerp >= 1)
    {
        if(GoneOnce == false)
        {
            GoneOnce = true;
            if (StartPos == StartSave)
            {
                StartPos = EndSave;
                EndPos = StartSave;
            }
            else if (StartPos != StartSave)
            {
                StartPos = StartSave;
                EndPos = EndSave;
            }
        }

        if(GoneOnce == true && WaitTime <= WaitSave)
        {
            WaitTime += 0.1f;
        }
        else if(GoneOnce == true && WaitTime >= WaitSave)
        {
            WaitTime = 0f;
            movelerp = 0.1f;
            GoneOnce = false;
        }
    }
}
```

The Movement() function is - unsurprisingly - the most important function in the script, and is called within the FixedUpdate(). This function moves the position of the platform via a movement lerp between StartPos and EndPos. This only occurs when GoneOnce is false. The function then checks the value of the moveLerp, checking when it has reached or exceeded the value of 1. When this is true, the script then will swap over the values of the StartPos and EndPos, so that they are the opposite of what they were previously saved as. During this, GoneOnce is set to true so that the platform stops moving. GoneOnce will return to being set as false once the waiting period has been completed, if there needs to be any waiting. The waiting works by increasing the WaitTime by 0.1 consistently until the value exceeds or equals that stored in WaitSave. Once the values reach this point, the function resets for a new movement Lerp. This script works well to properly and easily add Moving Platforms to the game.

Segment 2 - Introduction To Inventory:

The second segment of the game introduces the player to the inventory system in the game. This system is complicated enough to have it's own section discussing it, but aside from the Inventory, this area also introduces a new feature in the Projector, which is used to help plan within the 2D space. This area also originally had the player change camera angles to place items down from the Inventory, which ended up being scrapped.

Figure 3 - Segment 2



Projector:

The Projector is an item that appears throughout the game, and is used to help the player plan with how to handle the 2D platforms and space. It is used for puzzles throughout the game, and is easily placed throughout the game.

Projector - Variables:

```
public class Projector : MonoBehaviour
{
    //The bool for the variables for the projector, being to check when the projector is on and the 2D area that appears.
    [SerializeField] bool On;
    [SerializeField] GameObject OnItem;

    [SerializeField] bool PressedE;

    [SerializeField] GameObject ProjectorLight;

    [SerializeField] audiomanager AM;

    //The audio lines for the projector when it's turned on and off.
    [SerializeField] VoiceActing[] OnandOff;

    //When this bool is set to true, the 2D area will stay on even if the player is swapping between styles.
    public bool Keepon;
```

This script has several scripts all needed to keep it running. The variables are as follows:

On (bool) – This bool registers whether the projector is on or off.

OnItem (GameObject) – This is the 2D section that the Projector reveals when it is on.

PressedE (bool) – This bool checks when the player has pressed the E key or not.

ProjectorLight (GameObject) – This is the light that appears

AM (audiomanager) – This variable stores the audiomanager script.

OnandOff[] (Voice Acting) – This is an array that stores different voice lines for when the player turns the projector on and off.

Keepon (bool) – This bool is set so that the decals don't deactivate when the player leaves the 2D area.

Projector – Code:

```
void Start()
{
    AM = GameObject.FindGameObjectWithTag("AControl").GetComponent<audiomanager>();
}

// Update is called once per frame
void Update()
{
    //When the projector is on, then the light and the 2D area appear.
    ProjectorLight.SetActive(On);
    //This checks if the 2D area is deactivated, and if it is then the values for On and Keepon are false.
    if(OnItem.activeSelf == false)
    {
        On = false;
        Keepon = false;
    }

    //The effects of the projector can be set by the player when they are looking at the projector.
    //The script checks to see if the Projector is highlighted, based on whether the layer of the object is ItemSelected.
    if (gameObject.layer == LayerMask.NameToLayer("ItemSelection"))
    {
        //If the player presses E whilst the projector is highlighted then the script will change the value of the On bool.
        if (Input.GetKeyDown(KeyCode.E) && PressedE == false)
        {
            AM.Projector.SetActive(true);
            AM.Projector.GetComponent<AudioSource>().Play();
            On = !(On);
            PressedE = true;
            switch (On)
            {
                case true:
                    {
                        OnItem.SetActive(true);
                        Keepon = true;
                        OnandOff[0].AddLine();
                    }
                    break;
                case false:
                    {
                        OnItem.SetActive(false);
                        Keepon = false;
                        OnandOff[1].AddLine();
                    }
                    break;
            }
        }
        if(Input.GetKeyUp(KeyCode.E))
        {
            PressedE = false;
        }
    }
    else
    {
        PressedE = false;
    }
}
```

The Start() function is only used to assign the value of the AM variable using FindGameObjectWithTag. The rest of the code is all shown in Update() function. The first thing the script does is activate the light of the projector based on the value of On. It then checks to see if the 2D area is currently active, and if not sets the values of On and Keepon to false, so that the projector isn't on constantly after the player beats the 2D section.

The function then checks to see if the Projector is highlighted, which when it is it will allow the player to press E to turn it on or off. When E is pressed the script will invert the value of On, before checking the value in a switch statement to either deactivate the 2D area or activate it.

Examine_Point:

This script ended up being unused in the game. Originally when placing items the player needed to change cameras to a different angle and place the items down with their mouse click. Since the script is unused I won't go into too much detail, but I'd like to show the script and give a vague idea of what it about.

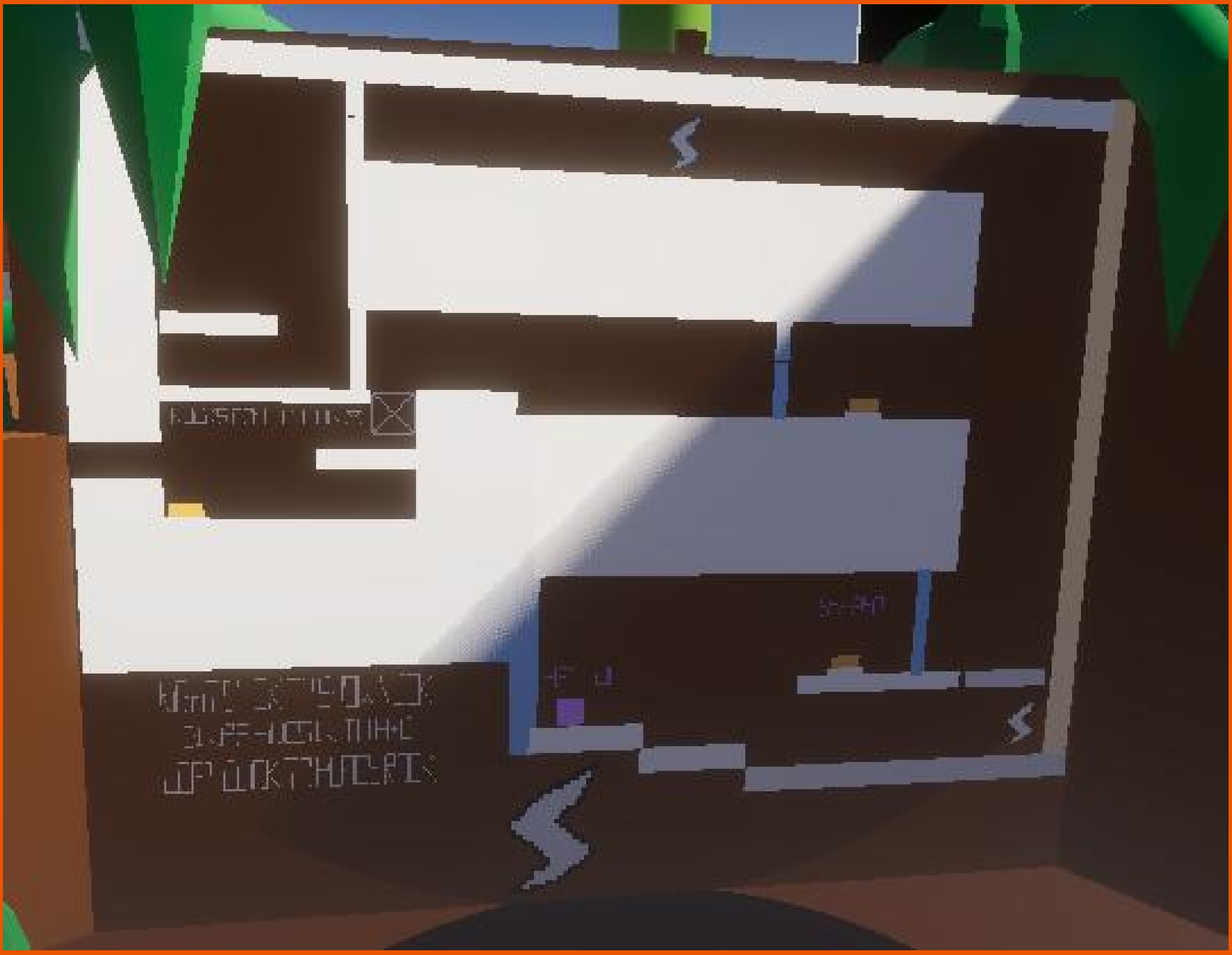
```
public class Examine_Point : MonoBehaviour
{
    [SerializeField] GameObject ExamineObject;

    [SerializeField] GameObject PressE;

    [SerializeField] bool EnteredArea;

    [SerializeField] Camera AreaCamera;

    [SerializeField] Camera MainCamera;

    [SerializeField] FPS_Movement FPSM;

    [SerializeField] GameObject PlayerCylinder;

    [SerializeField] GameObject ExamineUI;

    // Start is called before the first frame update
    void Start()
    {
        AreaCamera.gameObject.SetActive(false);
        ExamineUI.SetActive(false);
        //ExamineObject.SetActive(false);
    }

    // Update is called once per frame
    void Update()
    {
        //Causes elements to occur based on whether the player is looking at the examination area.
        if (EnteredArea == true)
        {
            if (Input.GetKey(KeyCode.Escape))
            {
                if (FPSM.openInventory == false)
                {
                    gameObject.GetComponent<BoxCollider>().enabled = false;
                    EnteredArea = false;
                    AreaLeave();
                }
            }
if (FPSM.openInventory == true)
            {
                ExamineUI.SetActive(false);
            }
            else
            {
                ExamineUI.SetActive(true);
            }
        }
        else
        {
            ExamineUI.SetActive(false);
        }

    }


    private void OnTriggerEnter(Collider other)
    {
        if(other.gameObject.tag == "Player")
        {
            PressE.SetActive(true);
        }
    }
    //Sets up the area examination when in the area's trigger.
    private void OnTriggerStay(Collider other)
    {
        if (other.gameObject.tag == "Player")
        {
            Debug.Log("TriggeredWithItem");
            if (Input.GetKey(KeyCode.E))
            {
                if(EnteredArea == false)
                {
                    gameObject.GetComponent<BoxCollider>().enabled = false;
                    EnteredArea = true;
                    PressE.SetActive(false);
                    AreaLook();
                }
            }

        }
    }

    private void OnTriggerExit(Collider other)
    {
        if(other.gameObject.tag == "Player")
        {
            PressE.SetActive(false);
        }
    }

    //This function sets up the variables of the Examination area and freezes the player's movement.
    In addition it makes the mouse visible on screen again.
    void AreaLook()
    {
        ExamineUI.SetActive(true);
        PlayerCylinder.SetActive(false);
        ExamineObject.SetActive(true);
        MainCamera.gameObject.SetActive(false);
        AreaCamera.gameObject.SetActive(true);
        Cursor.lockState = CursorLockMode.None;
        Cursor.visible = true;
        FPSM.GetComponent<Rigidbody>().useGravity = false;
        FPSM.GetComponent<CapsuleCollider>().enabled = false;
        FPSM.CanMove = false;
        FPSM.Examining = true;
    }

    //This function closes the area examination and brings the player back to regular gameplay
    void AreaLeave()
    {
        ExamineUI.SetActive(false);
        PlayerCylinder.SetActive(true);
        ExamineObject.SetActive(false);
        MainCamera.gameObject.SetActive(true);
        AreaCamera.gameObject.SetActive(false);
        Cursor.lockState = CursorLockMode.Locked;
        Cursor.visible = false;
        FPSM.GetComponent<Rigidbody>().useGravity = true;
        FPSM.GetComponent<CapsuleCollider>().enabled = true;
        FPSM.CanMove = true;
        FPSM.Examining = false;
        gameObject.GetComponent<BoxCollider>().enabled = true;
        EnteredArea = false;
    }
}
```

The general idea of this script is that when the player enters the trigger they can press E to swap camera angles to the area, via the AreaLook() function, which deactivates the player and activates a camera locked to the area. Then when the player presses escape the AreaLeave() function is called, which reactivates the player and removes the camera.

Segment 3 - 2D Rock Puzzle Introduction:

This segment of the game is directly after the player walks through the game's hub area, and introduces the 2D rock ability. This ability is explored further in the section discussing the rock abilities, the script also introduces the use of Timelines for dialogue, though this is also discussed in the Dialogue section of the document. Aside from these, the main thing introduced in this segment are the switch and gate mechanics, which are used throughout the game.

Figure 4 - Segment 3



Switch and SwitchTrigger:

The switch mechanic is used throughout the game, and was initially only there to be used to open gates. As the game progressed, the switch ended up being used for several different purposes, from gates to HoldPlatforms and so forth. The switch mechanic uses two scripts, Switch and SwitchTrigger. SwitchTrigger is a small script attached to the trigger of the switch, which is used to register when the player has stood on the switch or not. This trigger is then registered in the Switch code within the Switch that the trigger will be parented. These switches where created as a prefab to make placement of them easy.

SwitchTrigger:

```
public class SwitchTrigger : MonoBehaviour
{
    //This bool is called in the Switch script to check if the player has hit the trigger.
    public bool Hit;

    //This object stores the object that has triggered with the switch.
    public GameObject TriggerObject;

    //This stores the Switch script attatched to the parent gameobject.
    [SerializeField] Switch S;
    private void Update()
    {
        //The game checks to see if the switch has been hit and whether the TriggerObject
        is null or not active, before resetting the values of Hit and TriggerObject
        //This is done so the script can register when an object is deleted.
        if (Hit == true && (TriggerObject == null || TriggerObject.activeSelf == false))
        {
            Hit = false;
            TriggerObject = null;
        }
    }

    //On Trigger the switch sets Hit to true and stores the triggered object as TriggerObject.
    private void OnTriggerStay(Collider other)
    {
        if (other.gameObject.tag == "Player" || other.gameObject.tag == "Rock" || other.gameObject.tag == "Pollen" || other.gameObject.tag == "EvilPollen")
        {
            Hit = true;
            TriggerObject = other.gameObject;

            //If the switch is made to swap the end position of a moving platform then the Switch scripts HitandChange() function will be called.
            if(S != null)
            {
                if (S.Changed == false && S.ChangeMove)
                {
                    S.HitandChange();
                }
            }
        }
    }

    //On leaving the trigger the values of Hit and TriggerObject are reset.
    private void OnTriggerExit(Collider other)
    {
        if (other.gameObject == TriggerObject)
        {
            Hit = false;
            TriggerObject = null;
        }
    }
}
```

The variables for this script are as follows:

Hit (bool) - This bool is called within the Switch script to check when the trigger has been pressed.

TriggerObject (GameObject) - This GameObject is used to store the current object that has triggered the script.

S (Switch) - This stores the parent objects Switch script.

In the Update() function this script checks whether the object has been triggered, and if so it checks the current value of the GameObject stored in TriggerObject, whether it is null or deactivated. If this is true, then the script registers that the object has been deactivated or deleted, and needs to therefore reset the values of Hit and TriggerObject to false and null. This is done so the script doesn't glitch when objects that were triggering it are deleted without necessarily exiting the Trigger.

The majority of this script's code is found within the OnTriggerStay() function, which if triggered by an appropriate object sets the values of Hit and TriggerObject. This function then checks whether the Switch is used to change the end position of a moving platform, and if this is true then the HitandChange() function from the Switch script is called. When Exiting the trigger the values of Hit and TriggerObject were reset to false and null, allowing the switch to be hit again.

Switch - Variables:

```
public class Switch : MonoBehaviour
{
    //Stores the trigger for the switch
    [SerializeField] SwitchTrigger ST;

    //Registers if the swithc has been hit.
    public bool Pressed;

    //These store the animators for the Gate and Switch
    [SerializeField] Animator GateAnim;

    [SerializeField] Animator SwitchAnim;

    //This value changes the type of switch that the instance is.
    [SerializeField] int SwitchType;

    //This stores potential HoldPlatforms that can be affected by the sript.
    [SerializeField] HoldPlatforms[] HP;

    //These variables store whether the button needs to wait before resetting it's value.
    [SerializeField] bool Waitbool;
    [SerializeField]float waittracker;
    [SerializeField] float WaitMax;

    //This registers if the Gate starts open in the game.
    [SerializeField] bool StartShut;

    //These variables are used for the function when changing the end position of a moving platform.
    [SerializeField] MovingPlatform2D MoveTD;
    [SerializeField] Vector3 NewEnd;

    public bool Changed;
    public bool ChangeMove;

    //These arrays are used to store arrays of Gates that are affected by the switch and whether they start shut.
    [SerializeField] Animator[] GateArray;
    [SerializeField] bool[] StartShutArray;
```

This script has quite a few variables, all needed to properly get it working, they are as follows:

ST (SwitchTrigger) - This variable stores the SwitchTrigger script assigned to a child of the Switch.

Pressed (bool) - This bool is used to mark when the switch has been pressed, used for both debugging purposes and the script itself.

GateAnim and SwitchAnim (Animator) - These animator variables store the animator components for the assigned Gate and the assigned Switch, allowing the gate and switch to move once pressed.

SwitchType (int) - This int stores the type of effect the switch will have when pressed.

HP[] (HoldPlatforms) - This array stores the HoldPlatforms that the switch can effect if it is that version of switch.

Waitbool (bool) - This bool, when true, has the effects of the switch wait a bit before resetting their effects.

waittracker and WaitMax (float) - These floats store the current time that the switch has waited and the max time to wait respectively.

StartShut (bool) - This bool, when true means that the default gate function starts with the gate starting open.

MoveTD (MovingPlatform2D) - This stores the code for any Moving Platforms that will have their end position changed after the switch is pressed.

NewEnd (Vector3) - This Vector3 stores the new position that the moving position will end at.

Changed and ChangeMove (bool) - These bools are used to register if the MovingPlatform needs to be changed, and whether it has changed already.

GateArray[] (Animator) - This stores an array of different gates that can be opened by the switch.

StartShutArray[] (bool) - This array of bools works alongside the array of Gates to mark if they start shut or not.

Switch - Update(), FixedUpdate() and HitandChange():

```
// Update is called once per frame
void Update()
{
    //A switch statement that swaps between the differentely used buttons in the scene
    switch (SwitchType)
    {
        case 0:
            {
                //This calls the default Gate Opening
                SwitchAnim.SetBool("Button", ST.Hit);
                //The Button for Gates
                if (StartShut == false)
                {
                    GateAnim.SetBool("Button", ST.Hit);
                }
                else
                {
                    GateAnim.SetBool("Button", !ST.Hit);
                }
                Pressed = ST.Hit;
            }
            break;
        case 1:
            {
                //The calls the HoldPlatform variaty of switch.
                SwitchAnim.SetBool("Button", ST.Hit);
                foreach (HoldPlatforms hp in HP)
                {
                    hp.Pressed = ST.Hit;
                }
                Pressed = ST.Hit;

            }
            break;
        case 2:
            {
                //The Button opens gate and moves platform
                SwitchAnim.SetBool("Button", ST.Hit);
                GateAnim.SetBool("Button", ST.Hit);

                foreach (HoldPlatforms hp in HP)
                {
                    hp.Pressed = ST.Hit;
                }
                Pressed = ST.Hit;

            }
            break;
        case 3:
            {
                //This button activates as true for a few seconds, before deactivating.
                //This type is used for HoldPlatforms.
                if (ST.Hit == true && Waitbool == false)
                {
                    Waitbool = true;
                    SwitchAnim.SetBool("Button", true);
                }
                if(Waitbool == true)
                {
                    foreach (HoldPlatforms hp in HP)
                    {
                        hp.Pressed = true;
                    }
                    Pressed = true;
                }
                if(Waitbool == true && waittracker >= WaitMax)
                {
                    Waitbool = false;
                    waittracker = 0;

                    SwitchAnim.SetBool("Button", false);

                    foreach (HoldPlatforms hp in HP)
                    {
                        hp.Pressed = false;
                    }
                    Pressed = false;
                }
            }
            break;
```

```
                case 4:
                    {
                        //This button activates as true for a few seconds, before deactivating.
                        //This type is used for Gates
                        if (ST.Hit == true && Waitbool == false)
                        {
                            Waitbool = true;
                            Pressed = true;
                            SwitchAnim.SetBool("Button", true);
                            GateAnim.SetBool("Button", true);
                        }
                        if (Waitbool == true && waittracker >= WaitMax)
                        {
                            Waitbool = false;
                            waittracker = 0;

                            SwitchAnim.SetBool("Button", false);
                            GateAnim.SetBool("Button", false);
                            Pressed = false;
                        }
                    }
                    break;
                case 5:
                    {
                        //This type only animates the switch.
                        SwitchAnim.SetBool("Button", ST.Hit);
                        Pressed = ST.Hit;
                    }
                    break;
                case 6:
                    {
                        //This type open multiple gates at once.
                        SwitchAnim.SetBool("Button", ST.Hit);
                        int i = 0;
                        foreach(Animator A in GateArray)
                        {
                            if (StartShutArray[i] == false)
                            {
                                A.SetBool("Button", ST.Hit);
                            }
                            else if (StartShutArray[i] == true)
                            {
                                A.SetBool("Button", !ST.Hit);
                            }
                            i++;
                        }
                        Pressed = ST.Hit;
                    }
                    break;
            }

        }

        //The wait Timers are called in the FixedUpdate()
        private void FixedUpdate()
        {
            if(Waitbool == true)
            {
                waittracker += 0.1f;
            }
        }

        //This function changes the end position of a Moving Platform.
        //This is stored in a seperate function in order to make sure it only happens once.
        public void HitandChange()
        {
            if(Changed == false)
            {
                Changed = true;
                MoveTD.ChangeEnd(NewEnd);
            }
        }
    }
```

The majority of this script is called within the Update() function, which uses a Switch Statement to check the value of SwitchType to determine what version of function the switch should have, so for this, I will explain each variety of the switch functions in order.

When SwitchType is 0, then the effects of the switch are the default variety, designed to open a singular gate and that's it. This type also checks whether the gate needs to start shut or not, and will invert the values of the gate based on that. When the value is 1, then that means the switch is made to work with the HoldPlatforms, it does this with a for loop that moves each platform selected. When the value is 2 it works to move a HoldPlatform and open a gate at the same time. A 3 value sets it so the HoldPlatforms move, but wait a period of time before moving back to original position. The waiting will be increased within the FixedUpdate().

The 4 value works to keep gates open over a set time, using the same wait system made for the Hold Platforms. When the value is 5, it works to only animate the switch being pressed. Finally, when the value is 6, the switch is used to open multiple gates at once. With all these classified it is easy to add switches with these mechanics, and is also easy to add new functions to the list.

The FixedUpdate() is used to calculate the time spent waiting in a fixed rate that is consistent between the Editor and Build. Finally, the HitandChange() function is called in order to swap the end position of the moving platform assigned to the switch, and is done this way so that the function only occurs once.

GiveRock:

One final script used in this area, is that of GiveRock, which has a public function which is called by the segment's timeline to activate the Rock ability in the 2D area. It is incredibly simple and incredibly short, so doesn't require any real analysis, but regardless it is a script made for the game, and therefore needs to be examined.

GiveRock - Script:

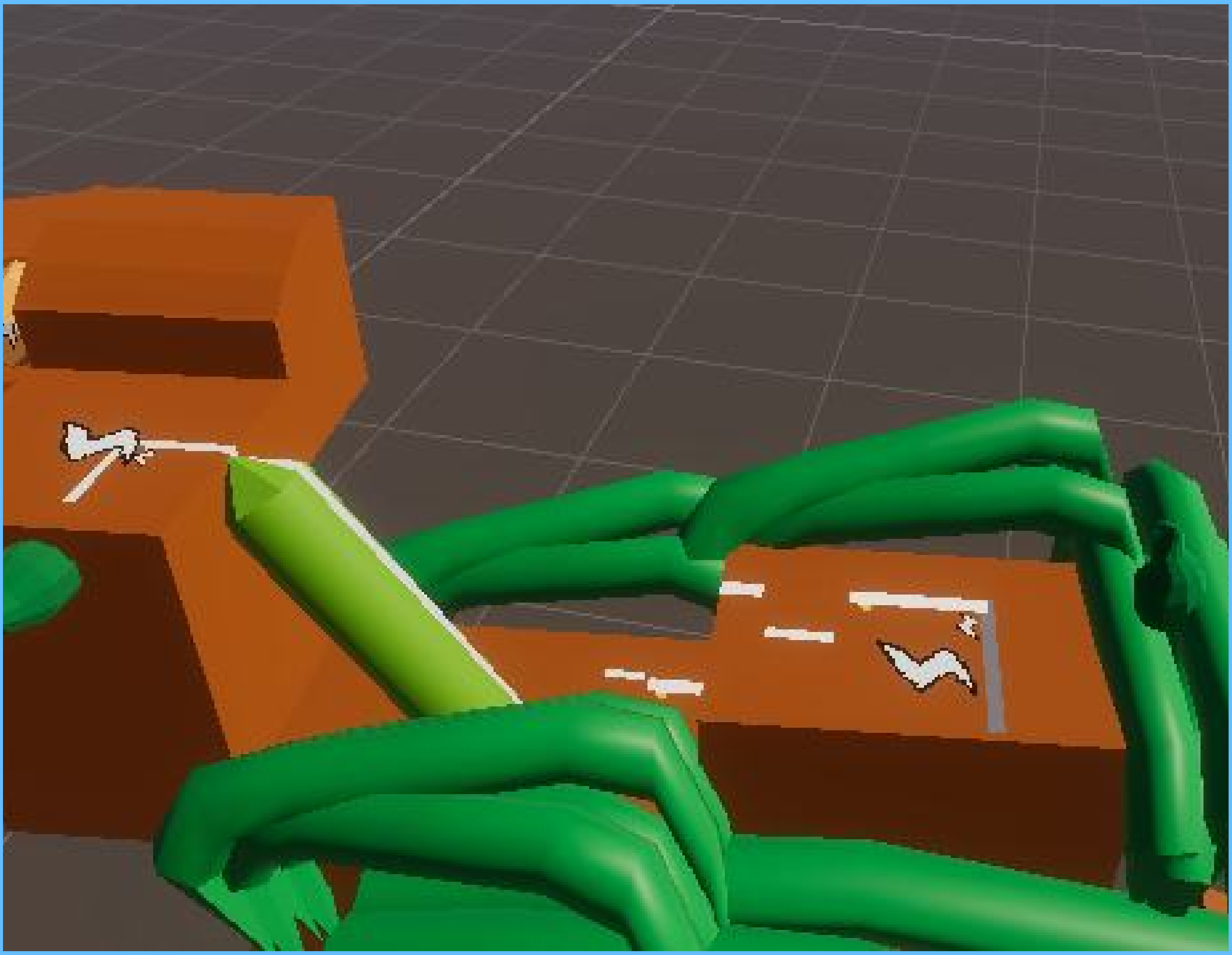
```
public class GiveRock : MonoBehaviour
{
    //This script is called by the Timeline in Segement 3 to activate the player's 2D rock ability.
    [SerializeField] AtariAblilitySave AAS;
    [SerializeField] DecalMovement DM;

    public void GiveRockFunction()
    {
        AAS.AtariRock = true;
        DM.isMove = true;
        DM.CutsceneJump = false;
    }
}
```

This script has one purpose, to activate the AtariRock ability stored in AtartAbilitySave, and to allow the player to move again after the timeline ends. This is done easily via the GiveRockFunction() function which is called by the timeline, and works easily.

Segment 4 - Floor Platforming:

This section follows the introduction of the 3D Rock ability, and requires the player to knock over a vine to change the path in the 2D space. This area introduces two new scripts and mechanics to the game, the first of which being the ability to change the gravity of the scene. This script is needed for the 2D sections to be positioned on the floor, in order to stop the player from falling through the ground. This area also introduces the HoldPlatforms, a variation on the Moving Platforms that only move when the player presses the switch.

Figure 5 - Segment 4



SettingGravity:

This script was created as a general system setting, which ended up only being needed for this one segment. The script is designed to swap the direction of the gravity within the 3D Space, and was used to allow for more unique 2D sections, for example platforms on the floor of the scene, rather then them being locked to the walls of the scene. I believe that in the full release of this game, this script has great potential to allow for greater unique level designs.

SettingGravity - Script:

```
public class SettingGravity : MonoBehaviour
{
    //This script is used to change the gravatational direction of the scene.

    //This Vector3 stores the default values of the gravity.
    [SerializeField] Vector3 RegularPhysics;

    //This GameObject stores the player character
    [SerializeField] GameObject Player;
    // Start is called before the first frame update
    void Start()
    {
        //In the start function the defualt gravity settings are stored in RegularPhysics.
        RegularPhysics = Physics.gravity;
    }

    //This function is called by the GenreChange script to set the gravity for the 2D Space.
    public void GravityChange(string direction)
    {
        //When called this function stores a string, which calls different gravity directions based on the value.
        //The only one that was used in the game was XPos.
        switch(direction)
        {
            case "Zpos":
                {
                    Physics.gravity = new Vector3(0, 0, 9.81f);
                }
                break;
            case "Zneg":
                {
                    Physics.gravity = new Vector3(0, 0, -9.81f);
                }
                break;
            case "Xpos":
                {
                    Physics.gravity = new Vector3(9.81f, 0, 0);
                }
                break;
            case "Xneg":
                {
                    Physics.gravity = new Vector3(-9.81f, 0, 0);
                }
                break;
            case "Regular":
                {
                    Physics.gravity = RegularPhysics;
                }
                break;
        }
    }
}
```

This script only has two variables, which are as follows:

RegularPhysics (Vector3) - This Vector3 stores the regular gravitational value of the scene.

Player (GameObject) - This stores the 3D Player Character.

The value of RegularPhysics is set to be that of the scene's current gravity, which is done within the Start() function. The only other function within this script is the GravityChange() function, which is called by the GenreChange script to change the gravity of the 2D space. This is done using a string variable called by the function called direction, which - dependent on the value - will change the direction that the gravity is pulling with. I find using string variables like this with a Switch Statement works well to efficiently create changes within the scene. In the current version of the game as of the submission, the only direction called is "Xpos", which is used for segment 4 of the game.

HoldPlatforms:

This segment introduces the HoldPlatforms mechanic, platforms which only move whilst an associated switch is pressed. This script is an off-shoot of the MovePlatforms2D script, just with more limitations to how the platform can move. Because of this, this analysis of the script will serve more to explain how the script differs from it's Moving relative.

HoldPlatforms - Variables, Start() and Update():

```
public class HoldPlatforms : MonoBehaviour
{
    //This script is for the variaty of moving platforms in the game which only move when a switch is pressed.

    //These Variables store the Start and Ending Positions of the Platform.
    [SerializeField] Vector3 StartPos, EndPos;
    //these floats store the Lerp of the movement and the speed.
    [SerializeField] float movelerp, movespeed;

    //This bool is called in the Switch script and registers if the Switch has been pressed.
    public bool Pressed;

    // Start is called before the first frame update
    void Start()
    {
        //This sets the position of the platform to be the StartPos
        gameObject.transform.localPosition = StartPos;
    }

    // Update is called once per frame
    void Update()
    {
        //This checks the value of Pressed and calls functions based of it.
        switch (Pressed)
        {
            case true:
                {
                    HeldMovement();
                }
                break;
            case false:
                {
                    ReleasedMovement();
                }
                break;
        }

    }
```

This script has a lot less variables then it's Moving Platform alternative, and this is because it is a lot simpler of a system. The variables are as follows:

StartPos and EndPos (Vector3) - These variables, like with MovingPlatform2D, are used to store the starting and ending positions that the platform needs to be in. Unlike the Moving Platforms, these variables are locked in from the start, and don't get updated as the platform moves.

movelerp and movespeed (float) - These floats serve the same purpose as those in MovingPlatform2D, storing the current value of the lerps progress and the current speed of the platform.

Pressed (bool) - This bool is set within the Switch script, and tells the platform when the associated switch has been pressed.

The Start() function serves to move the platform to the position stored in the StartPos variable. In the Update() function, the script checks the value of the Pressed variable, seeing if the switch connected to it has been pressed. If it is true, then the HeldMovement() function is called, and when it is not pressed, the ReleasedMovement() function is called. These functions serve to move the platform back and forth, rather then being in one Movement() function.

HoldPlatforms - OnCollisionStay() and OnCollisionExit():

```
//Like with the MovingPlatform the script is desigend so that the rock stays parented to it.
private void OnCollisionStay(Collision collision)
{
    if (collision.gameObject.tag == "Rock")
    {
        collision.gameObject.transform.parent = gameObject.transform;
    }
}

private void OnCollisionExit(Collision collision)
{
    if (collision.gameObject.tag == "Rock")
    {
        collision.gameObject.transform.parent = null;
    }
}
```

Like with the MovingPlatform2D script, the OnCollisionStay() and Exit() functions are used to parent the 2D rock to the platform, so that no glitches occur.

HoldPlatforms – HeldMovement() and ReleasedMovement():

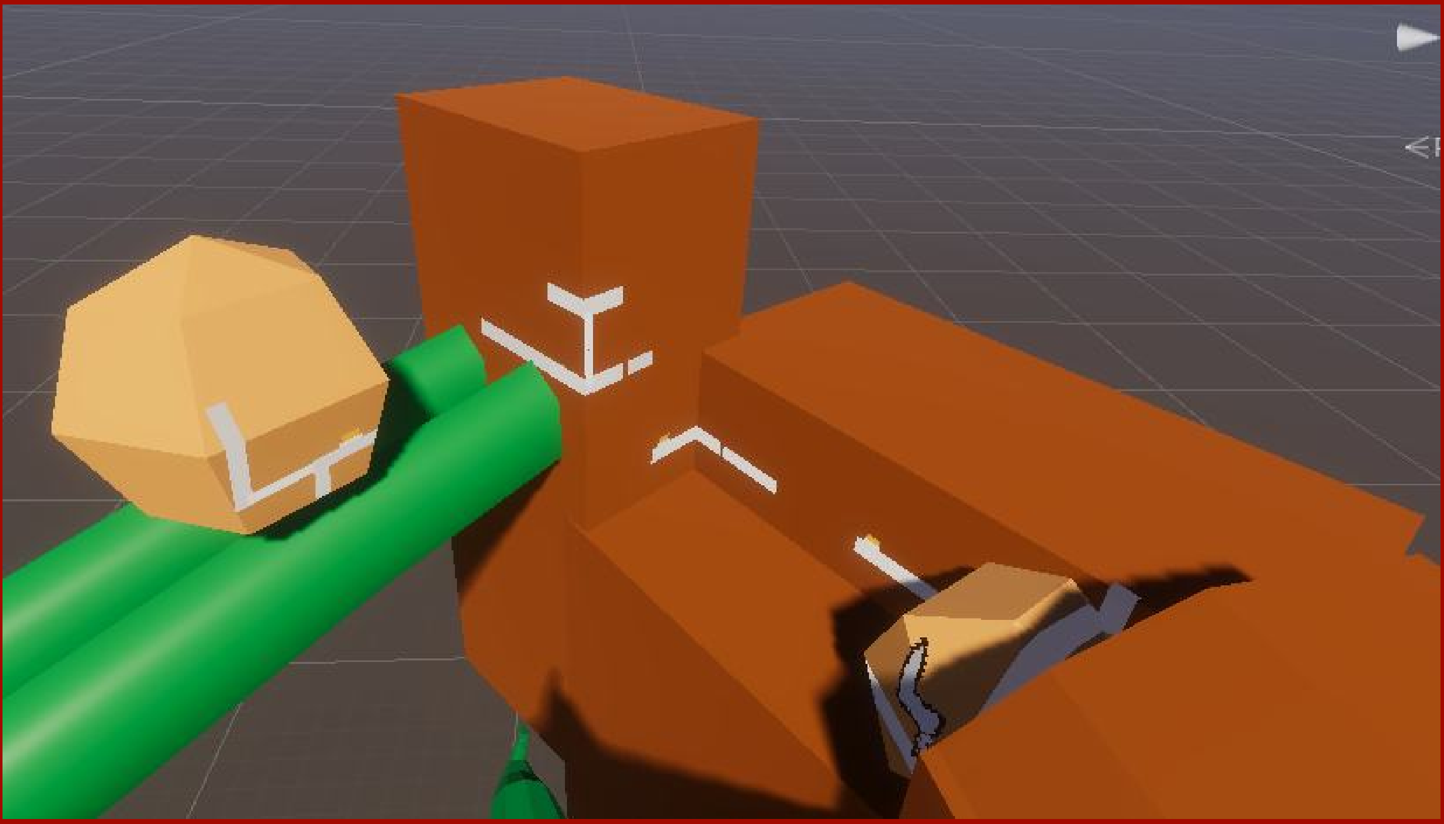
```
//This function is called when the switch is pressed and moves the platform up.
public void HeldMovement()
{
    if(movelerp < 1)
    {
        movelerp = Mathf.Clamp(movelerp + 1 * Time.deltaTime * movespeed, 0, 1);

        gameObject.transform.localPosition = Vector3.Lerp(StartPos, EndPos, movelerp);
    }
}

//This function is called when the switch is released, which moves the platform back to the start position.
public void ReleasedMovement()
{
    if (movelerp > 0)
    {
        movelerp = Mathf.Clamp(movelerp - 1 * Time.deltaTime * movespeed, 0, 1);

        gameObject.transform.localPosition = Vector3.Lerp(StartPos, EndPos, movelerp);
    }
}
```

The HeldMovement() and ReleasedMovement() functions serve to move the platform back and forth when pressed. When the switch is pressed, then HeldMovement() is called, which moves the platform up to the position stored in EndPos. When the switch is released, then ReleasedMovement() is called, which moves the platform back to the position stored in StartPos. This system is very simple and works well to get the platform working.

Segment 5 - Boulder Chaser:

The 5th segment of the game has the player platform on a moving boulder, which reveals platforms as it moves. If the player leaves the confines of the boulder then they die and need to restart the challenge. This area mostly used mechanics established in prior areas, but added some more scripts to make the boulder chasing work.

Figure 6 - Segment 5



RockTriggerBool:

This script is used to check when the player has entered the confines of the boulder, via triggers, so that they get killed when not keeping up with the boulder. This script also is used to open the Gate before the boulder when the boulder nears the gate. This is done by calling the bool within the Boulder's animation.

RockTriggerBool – Variables:

```
public class RockTriggerBool : MonoBehaviour
{
    //This script is used to make the player die after leaving the boulder's confines.

    //This bool is used to affect the value of the gate animators bool
    public bool Set;
    //This stores the death trigger for the boulder
    [SerializeField] DecalDeath DD;

    //This bool checks when the player is in the collision of the boulder
    [SerializeField] bool InCollide;
    //This stores the animator for the Gate
    [SerializeField] Animator Gate;
    //This bool, when true, means that the trigger instance is used to open the gate.
    [SerializeField] bool SetGate;

    //This stores the Atari Player
    [SerializeField] GameObject Player;
```

This script has a few variables, which are as follows:

Set (bool) - This bool is used to change the value of the gate animators bool.

DD (DecalDeath) - This stores the death collider for the boulder to use when killing the player.

InCollide (bool) - This bool is set to true when the player enters the collision of the boulder, and is used to kill the player when they leave the boulder collision.

Gate (Animator) - This is the animator for the Gate that is positioned just before the boulder.

SetGate (bool) - This bool is called in the Animator of the boulder, and registers when the gate should be open or shut.

Player (GameObject) - This stores the decal player for this segment.

RockTriggerBool - Script:

```
private void Update()
{
    //The Update() function is used to open the gate, if the instance is the right trigger for it.
    if(Player != null)
    {
        if(Player.activeSelf == true)
        {
            if (Gate != null && SetGate == true)
            {
                Gate.SetBool("Button", !(Set));
            }
        }
    }
}

//When the player enters the trigger, the script registers it has now collided.
private void OnTriggerEnter(Collider other)
{
       if (other.gameObject.tag == "Player")
       {
           InCollide = true;
       }
}

//When staying the trigger the script registers that the player has collided with the boulder.
private void OnTriggerStay(Collider other)
{
    if (other.gameObject.tag == "Player")
    {
        InCollide = true;
    }
}
```

```
    //If the player exits the trigger of the boulder then they get killed.
    private void OnTriggerExit(Collider other)
    {
        if (other.gameObject.tag == "Player")
        {
            if(InCollide == true)
            {
                DD.Die();
                InCollide = false;
            }
        }
    }
```

The Update() function for this script is used to open and shut the gate animator, with it checking if the instance of the script is meant for the gate to open, based on whether the Player variable is null or not. From there it just checks to see if the gate should be open, and if so it then sets the animation bool of the gate to the inverse of the value of Set.

The script then uses the OnTriggerEnter() and Stay() functions to check if the player is within the confines of the boulder. If the player is, then InCollide is set to true, which is used in the OnTriggerExit() function, which will kill the player upon them leaving the trigger if this variable is true.

TriggerAnim:

The only other script used in this area is that of TriggerAnim, which is used to reset the animation of the Boulder when the player nears the start of the segment, so that they don't need to wait for it respawn. This is done using a Trigger just before the spawning of the boulder, which when enters resets the boulder.

TriggerAnim – Script:

```
public class TriggerAnim : MonoBehaviour
{
    //This script is used to restart the boulders animation so the player doesn't need to wait.

    //This stores the animator for the boulder
    [SerializeField] Animator Anim;
    //This bool stores the value that the animator should be set at.
    [SerializeField] bool set;


    //When triggered this resets the position of the moving boulder.
    private void OnTriggerEnter(Collider other)
    {
        if (other.gameObject.tag == "Player")
        {
            if (Anim != null)
            {
                Anim.SetBool("Restart", set);
            }
        }
    }
}
```

This script only has two variables, which are as follows:

Anim (Animator) - This stores the Animator for the boulder.

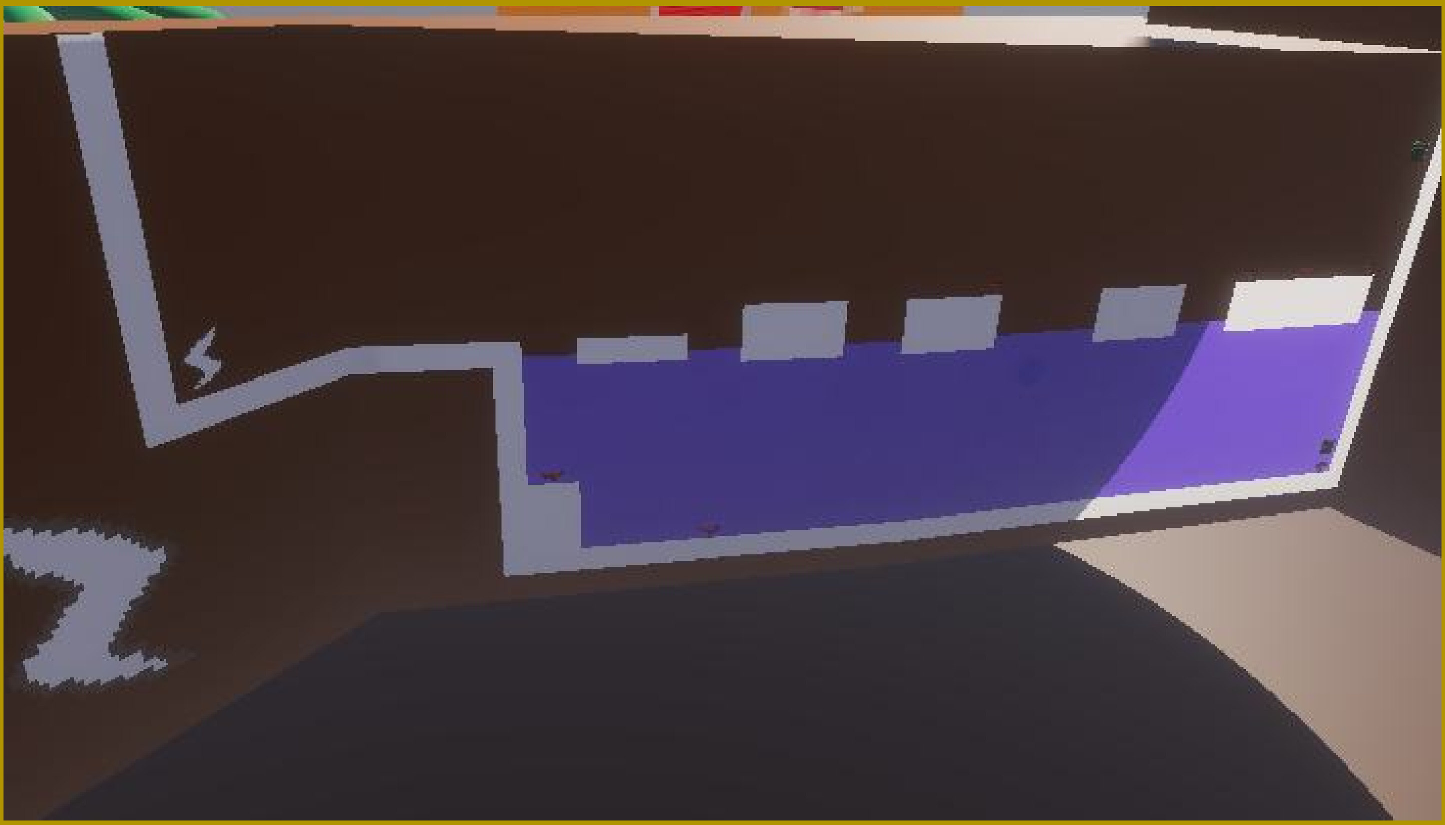
set (bool) - This bool stores what value the Animator of the boulder's bool should be set at.

When entering the trigger of this object, the script restarts the animation, so that the boulder is back at its start location. This is simple and works well to keep the boulder near the player.

Segment 7 - Optional Area and Mega Man Introduction:

After the culmination of mechanics that was Segment 6, Segment 7 is mainly there to set up the next 2D Gameplay style, being the Mega Man style of gameplay. Because of this, the segment is filled with new mechanics unique to that style of gameplay, which works quite well. In addition, this segment has an optional area of gameplay that serves to be it's own unique platforming challenge that the player can experience just for fun. This optional area has the player need to use the 3D rock ability to block a leak of water, which then effects the 2D section in turn. This part of the document will discuss the code presented in the Optional Area first, before examining what was added to the Mega Man Introduction.

Figure 7 - Segment 7, Optional Area



WaterBlock:

The optional area of the game revolves around the mechanic WaterBlock provides, with the player needing to block the water leak with a 3D rock. When this is done the platforms in the 2D area change to reflect if their is any water or not. I like this mechanic and think it creates a good connection between the 2D and 3D spaces.

WaterBlock - Variables:

```
public class WaterBlock : MonoBehaviour
{
    //This script is attatched to the hole squirting water which affects the 2D platforms in the optional area.


    //This stores the different decal platforms that show when the player hits the hole with the rock.
    [SerializeField] GameObject[] RockShow;
    //This Stores the Water particles that appear from the hole in the wall.
    [SerializeField] GameObject Water;
    //This stores the position that the rock will be placed in after colliding with the hole.
    [SerializeField] Vector3 RocPos;
    //This stores the instance of the rock that triggers the hole
    [SerializeField] GameObject RockObject;
    //This bool stores whether the hole has been hit or not.
    [SerializeField] bool BeenHit;
```

This script has five variables needed for the system to work. They are as follows:

RockShow[] (GameObject) - This array of GameObjects store the 2D platforms that are swapped between when the rock blocks the water source.

Water (GameObject) - This GameObject stores the particle system that is used to show the leaking water.

RockPos (Vector3) - This Vector3 stores the position that the rock will be locked at when it blocks the water source.

RockObject (GameObject) - This GameObject stores the instance of the 3D Rock which triggers with the Water Source.

BeenHit (bool) - This bool checks to see if the Rock has blocked the Water Source or not.

WaterBlock - Script:

```
// Update is called once per frame
void Update()
{
    //This checks the whether the object has been hit or not.
    if(RockObject == null && BeenHit == true)
    {
        RockShow[0].SetActive(true);
        RockShow[1].SetActive(false);
        Water.SetActive(true);
        BeenHit = false;
    }
}

//This checks when the rock has collided with the hole in the wall, if it has then it sets the 2D scene up to be the unflooded platforms.
private void OnCollisionEnter(Collision collision)
{
    if (collision.gameObject.tag == "Rock")
    {
        RockObject = collision.gameObject;
        BeenHit = true;
        RockObject.transform.parent = gameObject.transform;
        RockObject.tag = "Untagged";
        RockObject.transform.localPosition = RocPos;
        RockObject.GetComponent<Rigidbody>().velocity = Vector3.zero;
        RockObject.GetComponent<Rigidbody>().constraints = RigidbodyConstraints.FreezeAll;
        RockObject.GetComponent<Rigidbody>().mass = 0;
        Water.SetActive(false);
        RockShow[0].SetActive(false);
        RockShow[1].SetActive(true);
    }
}
```

The functions used within this script are only the Update() and OnCollisionEnter() functions. In the Update() function is used to check whether the 3D Rock blocking the Water Source has been deleted. If this happens, then it swaps the 2D area back to the flooded variety, and adds back the particles. The OnCollisionEnter() function is where the script swaps the 2D area's platforms, changing it over to the unflooded platforms when the 3D Rock enters the collision, it also locks the position of the 3D Rock to be placed directly in the centre of the water source.

Spring:

This spring is a mechanic first introduced in the optional area, which increases the jump height of the player when they enter the spring's trigger. It later got updated to also have the pollen shot by the Mega Man Player deflect off of it, in order to create a chain of bouncess

Spring - Variables:

```
public class Spring : MonoBehaviour
{
    //This script is used to create the Spring Mechanics.
    //Springs can either increase the Atari player's height, or have the Mega Man player's pollen bounce off it.

    //This stores the Decal Player that the Spring will effect
    [SerializeField] DecalMovement DM;

    //This stores the regular jump height of the decal player
    [SerializeField] float RegularJump;
    //This stores the jump height that the player swaps to
    [SerializeField] float SwapJump;

    //These GameObjects store the new starting and ending positions of the pollen shot by the Mega Man player when it hits the spring.
    [SerializeField] GameObject NewEnd;
    [SerializeField] GameObject NewStart;

    //This bool, when true, means that the pollen is bouncing along the Y-axis, rather then the X.
    [SerializeField] bool Up;
```

The spring script has six variables, though not all are needed to be set at once. Half are only needed for the Decal Player, whilst other half are needed for the Mega Man Player's pollen bullets. The variables are as follows:

DM (DecalMovement) - This Variables stores the DecalMovement code needed for the Atari Player that can trigger with the spring.

RegularJump and SwapJump (float) - These float variables store the values needed to change the jump height of the Atari Player. Regular Jump stores the default value of the player's jump, whilst SwapJump stores the value of the jump when the player is colliding with the spring.

NewEnd and NewStart (GameObject) - These GameObjects are used to swap out the current target and start point of the pollen bullets that collide with the spring.

Up (bool) - This bool is used to check whether the pollen bullet bouncing off the spring is moving in the Y-axis or the X-axis.

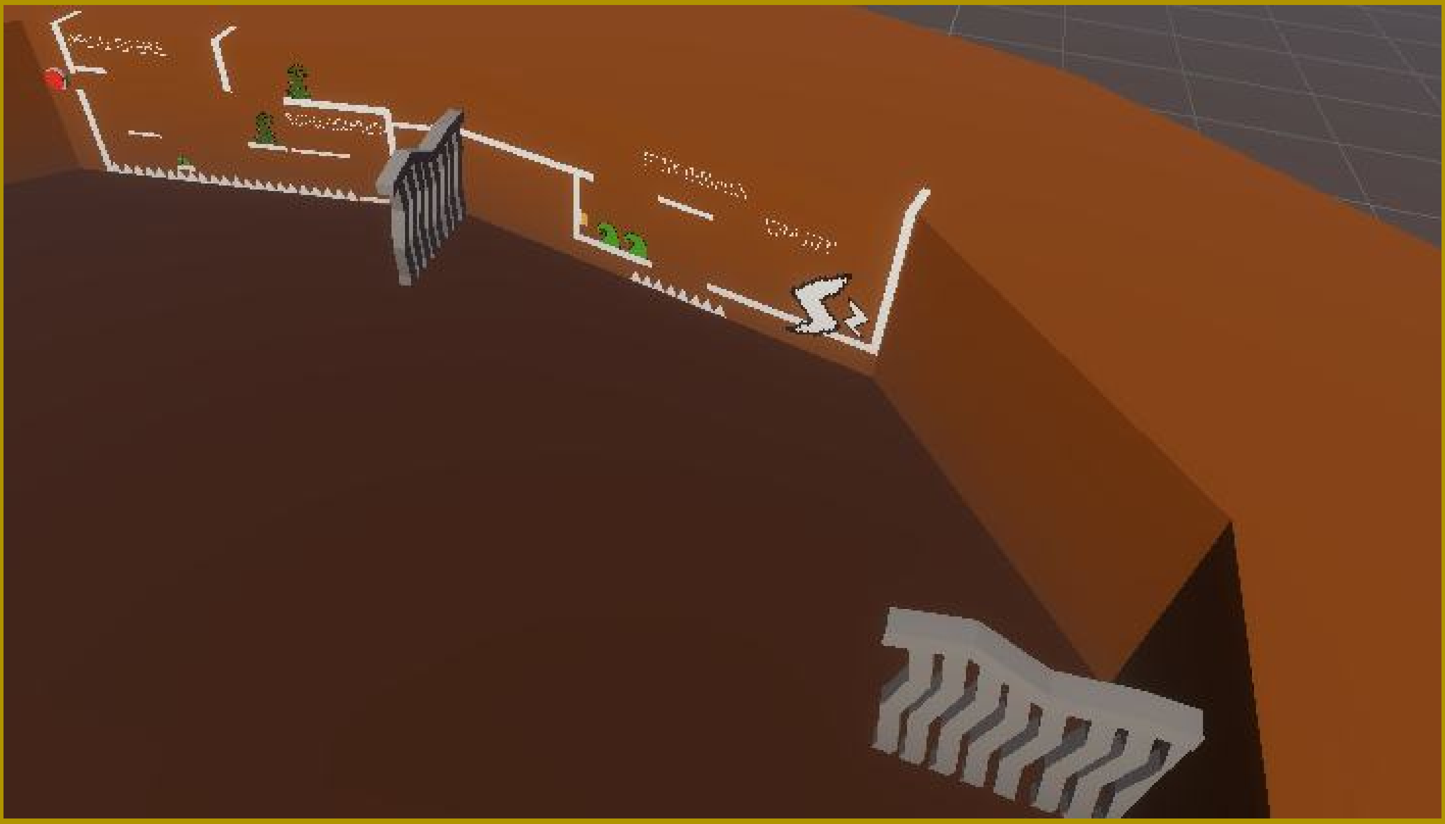
Spring - Script:

```
void Start()
{
    //In the Start() function this script stores the value of the player's regular jump height.
    if(DM != null)
    {
        RegularJump = DM.Jumpheight;
    }
}

private void OnTriggerEnter(Collider other)
{
    //If the player triggers with the spring then their jump height increases
    if(other.gameObject.tag == "Player" && DM != null)
    {
        DM.Jumpheight = SwapJump;
    }

    //If the pollen hits the spring, then the script changes the position that the pollen moves
between
    //This depends on the direction the pollen is moving between.
    if(other.gameObject.tag == "Pollen" || other.gameObject.tag == "EvilPollen")
    {
        other.gameObject.GetComponent<Pollen>().MoveUp = Up;
        switch (other.gameObject.GetComponent<Pollen>().MoveRight)
        {
            case true:
                {
                    other.gameObject.GetComponent<Pollen>().OriginMark = NewStart;
                    other.gameObject.GetComponent<Pollen>().TargetMark = NewEnd;
                }
                break;
            case false:
                {
                    other.gameObject.GetComponent<Pollen>().OriginMark = NewEnd;
                    other.gameObject.GetComponent<Pollen>().TargetMark = NewStart;
                }
                break;
        }
    }
}

//When the player exits the Trigger their jump height returns to normal.
private void OnTriggerExit(Collider other)
{
    if (other.gameObject.tag == "Player" && DM != null)
    {
        DM.Jumpheight = RegularJump;
    }
}
```

The Start() function for the spring is used to store the default value of the Atari Player's jump into the Jumpheight float. When the Atari Player enters the script's Trigger, their jump height gets changed to that stored in SwapJump. The Jumpheight then turns back to normal when exiting the trigger.

If the pollen bullet enters the trigger of the spring, then the positions that the pollen is moving towards gets changed. The means at which teh pollen changes depends on the direction the bullet was shot from. This function also sets the value of the Pollen script's MoveUp bool to be that of this script's Up bool.

Figure 8 - Segment 7, Mega Man Intro



Switch3D:

A mechanic established in this seventh segment are switches found within the 3D world, which were initially designed to open 3D gates in the scene, but later got updated to also be able to open 2D gates. These switches can be hit by 3D rocks and pollen, as well as the 2D pollen shot by the Mega Man player, which is the main goal of the first Mega Man section found in this segment.

Switch3D - Variables:

```
public class Switch3D : MonoBehaviour
{
    //This stores the animator of the Gates in the 3D space
    [SerializeField] Animator GateAnim;
    //This stores the material that the switch changes to when its been hit.
    [SerializeField] Material ActiveGate;

    //This tracks whether the switch has been hit
    [SerializeField] bool BeenHit;
    //This bool tracks whether the switch is used to open a 3D or 2D gate.
    [SerializeField] bool Decal;
```

Switch3D - Script:

```
    //The update function checks to see if the button is used to keep the 2D Gate open
private void Update()
{
    if(Decal == true && BeenHit == true)
    {
        GateAnim.SetBool("Button", true);
    }
}

//When being hit by the 2D pollen, this script calls the HitSwitch() function.
private void OnTriggerEnter(Collider other)
{
    if(other.gameObject.tag == "Pollen" || other.gameObject.tag == "EvilPollen")
    {
        if(BeenHit == false)
        {
            HitSwitch();
        }
    }
}

//The HitSwitch() function is used to open a gate in either the 3D of 2D space, dependent of the value of decal.
public void HitSwitch()
{
    BeenHit = true;
    switch(Decal)
    {
        case false:
            {
                GateAnim.SetBool("Opengate", true);
            }
            break;
        case true:
            {
                GateAnim.SetBool("Button", true);
            }
            break;
    }
    //When a 3D switch is hit, the material of the switch changes to a green.
    gameObject.GetComponent<MeshRenderer>().material = ActiveGate;
}
```

The Switch3D script is used to trigger the HitSwitch() function, either in this script when a 2D pollen bullet collides with it, or by being called in the script used for the 3D rock and 3D pollen, which calls this function. When hit, the HitSwitch() function will open either a 3D or 2D gate, based on the value of the decal bool. The Update() function is then used to keep the 2D gate open once hit, as without this the gate would reset after the player leaves the 2D area, so this fixed that issue.

plantgrow:

The plantgrow script is used for the Vine obstacles that the player needs to hit with their pollen bullets in Mega Man sections. This script works for the regular vines, which grow when shot with normal pollen, and thorny vines which shrivel when shot with evil pollen. Originally this script had the vines change by swapping two objects out for each other, but this led to issues with collision detection, so instead, I made it so the collision and sprites of the specific vine swap, rather then swapping objects.

plantgrow – Variables:

```
public class plantgrow : MonoBehaviour
{
    //This script is used for 2D vines that can be grown by the Mega Man Player's 2D pollen bullets.
    //This also has an alternative for the vines that block the path that can be shrunk by EvilPollen

    //This stores the decal material for the grown vine.
    [SerializeField] Material Plants;
    //These Vector3s store the size and center of the vine's box collider after it is shot.
    [SerializeField] Vector3 BoxSize;
    [SerializeField] Vector3 BoxCenter;

    //This stores the DecalProjector used for the 2D vine.
    [SerializeField] DecalProjector DP;
    //This stores the 2D vine's box collider.
    [SerializeField] BoxCollider BC;
    //These Vector3s store the position and scale that the 2D vine should swap to.
    [SerializeField] Vector3 Scale;
    [SerializeField] Vector3 Pos;
    //This bool when ticked, means that the vine should be shrunk by the player's evil pollen.
    [SerializeField] bool Spikes;

    //This bool is set true after the regular vine is grown.
    public bool Spring;
```

The variables used for this mechanic are as follows:

Plants (Material) - This variable stores the decal material that the decal projector swaps with when hit by the trigger.

BoxSize and BoxCenter (Vector3) - These Vector3 variables store the size and centre positions that the vine's box collider swaps to when hit.

DP (DecalProjector) - This variable stores the projector used to show the vine.

BC (BoxCollider) - This stores the vine's BoxCollider component.

Scale and Pos (Vector3) - These Vector 3 variables store the scale and position of the vine when shot by the pollen bullets.

Spikes (bool) - This bool determines whether the vine type is a regular one needing to be grown, or a spikey one needing to be shrunk.

Spring (bool) - This bool is set to true after the regular vine's grow.

plantgrow – Script:

```
private void OnCollisionEnter(Collision collision)
{
    if (collision.gameObject.tag == "Pollen")
    {
        Destroy(collision.gameObject);

        if(Spikes == false)
        {
            DP.material = Plants;
            BC.center = BoxCenter;
            BC.size = BoxSize;

            gameObject.transform.localPosition = Pos;
            gameObject.transform.localScale = Scale;
            Spring = true;
        }

    }
    else if(collision.gameObject.tag == "EvilPollen")
    {
        Destroy(collision.gameObject);

        if(Spikes == true)
        {
            DP.material = Plants;
            BC.center = BoxCenter;
            BC.size = BoxSize;

            gameObject.transform.localPosition = Pos;
            gameObject.transform.localScale = Scale;
        }
    }
}
```

The only function within plantgrow is the OnCollisionEnter() function. This function checks to see if either the pollen or evil pollen bullets have collided with the vine. If the regular pollen shoots the non-spike vines, then the pollen is destroyed and the vine grows to be taller, and allow the player to move through the vine and walk on the top. The opposite is done when the spikey vines are shot by evil pollen, the vine will shrink and the collider will move away so that the player is no longer blocked by the vine.

spikevine:

The spikevine script was originally programmed to delete the spike vines when the evil pollen collides with them, and was the original one to achieve this purpose, before this was incorporated within plantgrow. This script had to be removed, as deleting the vines, rather then adjusting their collision, causes glitches in the player's collision detection, and it was not worth keeping the script. Below is the code for this script, which is quite short, and doesn't really need any explanation past what I have already said.

```
 public class spikevine : MonoBehaviour
{
    //This script was orignally used to delete the spike vines after being hit by pollen
    //This had to be removed as deleting the object caused issues with the collision
detection.
    private void OnCollisionEnter(Collision collision)
    {
        if(collision.gameObject.tag == "EvilPollen")
        {
            Destroy(collision.gameObject);
            Destroy(gameObject);
        }
        else if(collision.gameObject.tag == "Pollen")
        {
            Destroy(collision.gameObject);
        }
    }
}
```

PollinUI:

As this area introduces the 2D Mega Man player, this also introduced UI for the player to make the gameplay easier to understand. This UI works to swap between two sprites of the different pollen types the Mega Man Player can shoot with. This was a quite simple script that worked well to present the player what pollen type they are using.

PollinUI - Script:

```
public class PollenUI : MonoBehaviour
{
    //This script is used to program UI that shows what type of pollen the Mega Man Player is shooting
    //It is quite simple, activating the UI images based on the value of the Mega Man player's pollenbad bool.
    public Mega_Man_Movement MMM;

    [SerializeField] GameObject[] PollenImage;
    void Update()
    {
        PollenImage[0].SetActive(MMM.pollenbad);
        PollenImage[1].SetActive(!MMM.pollenbad);
    }
}
```

The Pollen UI script only has two variables, being:

MMM (Mega_Man_Movement) - This variable is used to store the Mega_Man_Movement script of the areas Mega Man player.

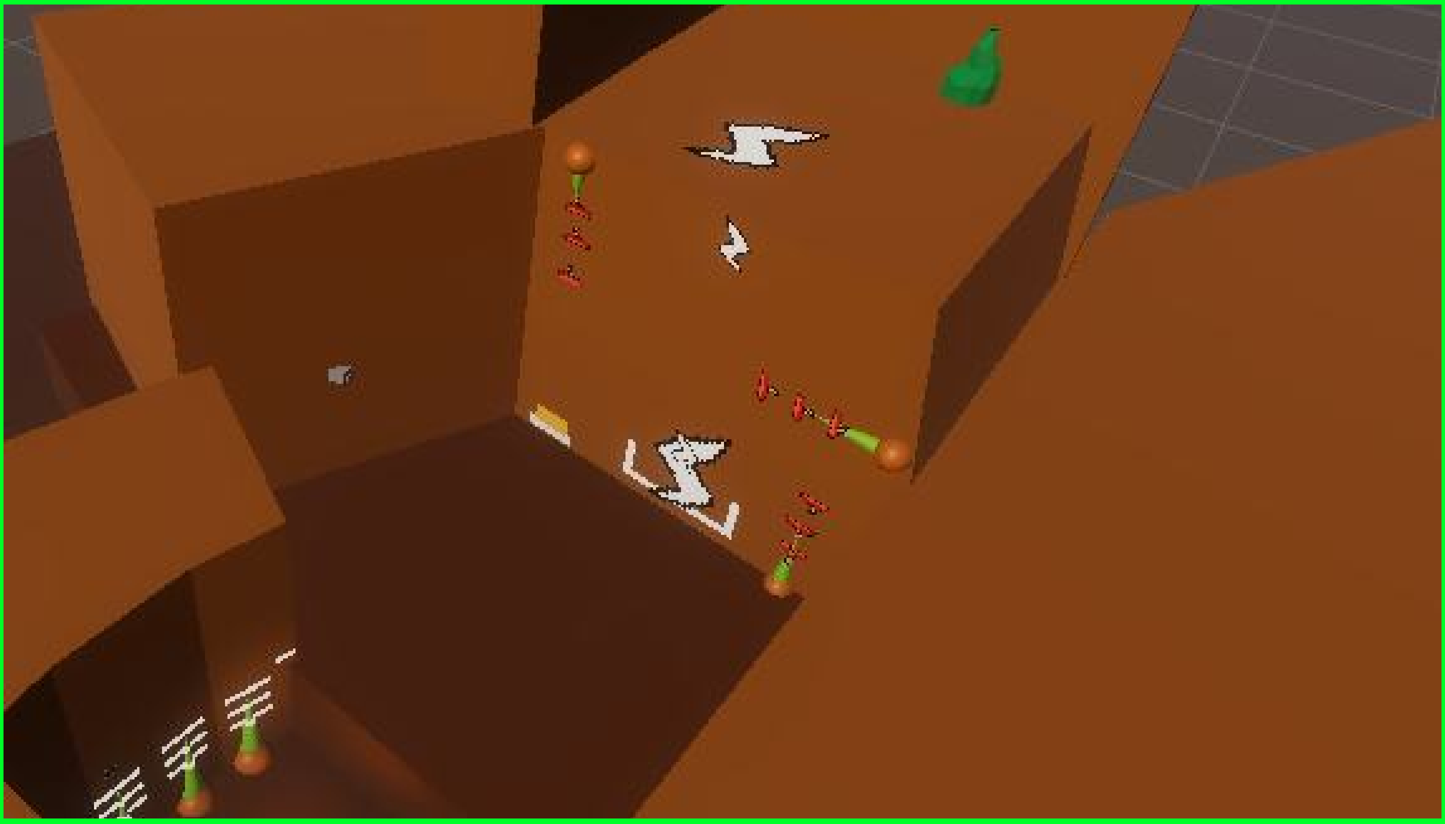
PollenImage[] (GameObject) - This array of GameObjects stores the different UI Image's that appear on the canvas.

The rest of the script takes place in the Update() function, which simply sets the state of the UI Images based on the value of MMM's pollenbad variable, with the regular pollen appearing when it is false, and the evil pollen appearing when it is true. This works well to create UI that clearly shows the pollen type that the player can shoot, and helps to make the gameplay for the Mega Man player feeling more refined.

Segment 8 and 9 - Flower Gun Puzzle and Spring Vine Puzzle:

The eight segment of the game houses two 2D platforming sections, both introducing new elements accessed via the 3D flower gun ability. These puzzles are short and simple require the player needing to shoot vines with their 3D pollen to change the size of the pollen. The first of the 2D sections is a redo of the second segment, with the player increasing the size of the vines using the pollen rather then placing them down as inventory items. The second is a puzzle with springs that change positions based on the size of the vines grown. These springs ended up using the Vines script made for the inventory items, but before that there was a script made for it called SwapSprings, which ended up being obsolete. Segment 8 is followed by Segment 9, which has a larger version of the vine and spring puzzle, which is very large and complicated, and required a 3D moving platform to fully see.

Figure 9 - Segment 8



SwapSprings:

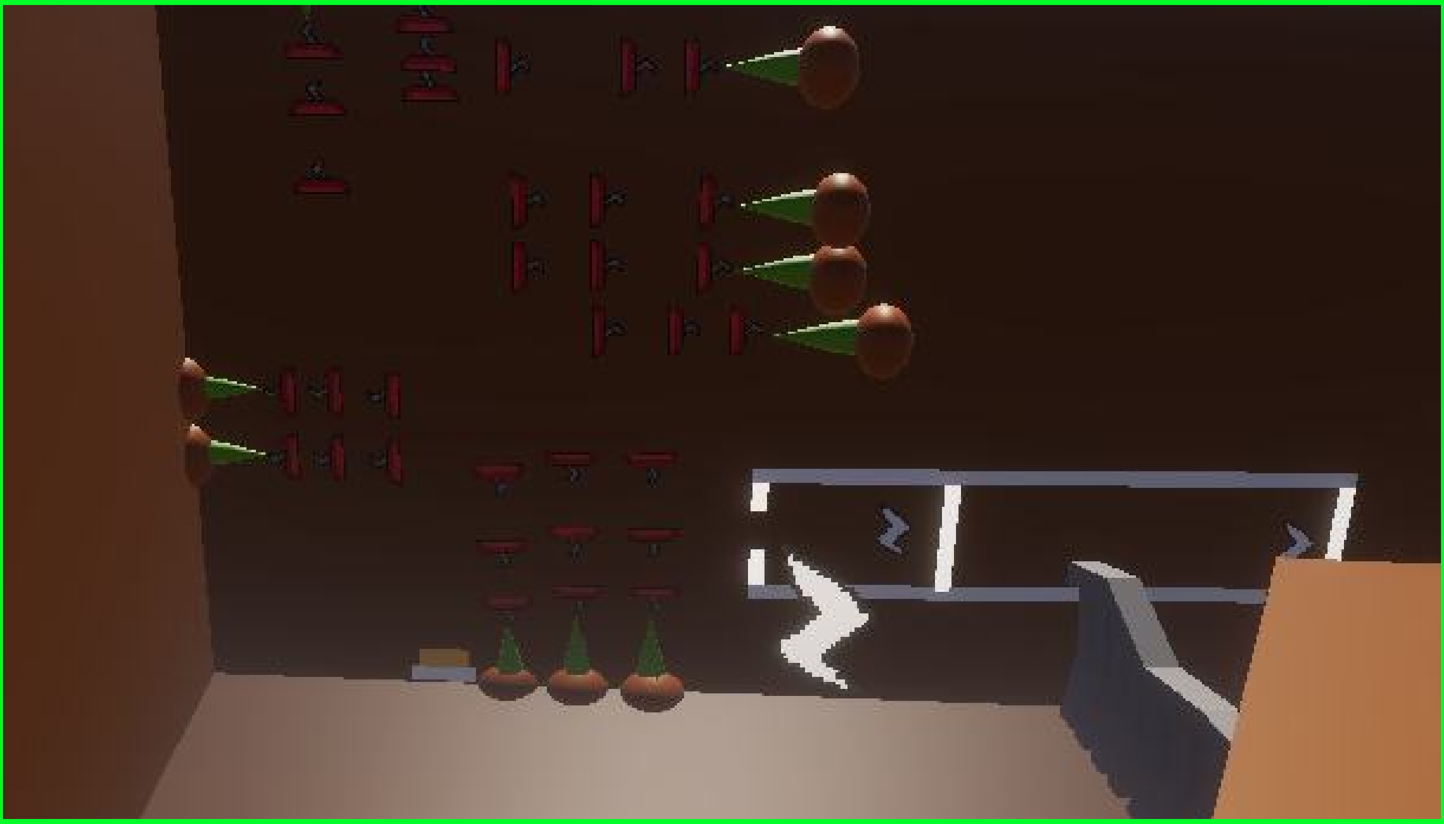
This script was created to have the springs change position based on the size of the vine, though turned out to obsolete within the game, as the purpose was already completed with the Vine script, which was much more efficient and simple. This will show the code for this unused script, and explain the code of it, even if it doesn't end up being used.

SwapSprings - Script:

```
public class SwapSprings : MonoBehaviour
{
    //This script was the orignal way I aimed to swap springs based on the size of the 3D vines
    before I realised I could just use the Vines script made for the inventory puzzle.
    //Because of this, this script ended up being unused.

    [SerializeField] GameObject[] Springs;
    [SerializeField] plantgrow PG;

    //The Update() function activates and deactivates springs based on the size of the vine.
    private void Update()
    {
        switch(PG.Spring)
        {
            case false:
                {
                    Springs[0].SetActive(true);
                    Springs[1].SetActive(false);
                }
                break;
            case true:
                {
                    Springs[1].SetActive(true);
                    Springs[0].SetActive(false);
                }
                break;
        }
    }
}
```

The idea of this script was to check the size of the grown plant, and then change the position of the spring. Another fun fact is that originally I planned to use the plantgrow script for the 3D vines, before I decided to just use the teleportplant script instead.

Figure 10 - Segment 9



FPSTrigger:

The FPSTrigger script is attached to a trigger collision parented to the First Person Player, and is there to parent the player to the 3D moving platforms where applicable. Since this was first needed in the ninth area, I decided to explain the code here. The system is very simple, so I will explain it all in one go.

FPSTrigger - Script:

```
public class FPSTrigger : MonoBehaviour
{
    //This script is attached to the FPS Player, and works to parent them to moving platforms when they are standing on them.
    [SerializeField] GameObject Player;

    private void OnTriggerEnter(Collider other)
    {
        if(other.gameObject.tag == "MoveFloor")
        {
            Player.transform.parent = other.gameObject.transform;
        }
    }

    private void OnTriggerExit(Collider other)
    {
        if (other.gameObject.tag == "MoveFloor")
        {
            Player.transform.parent = null;
        }
    }
}
```

This script only has one variable, which is as follows:

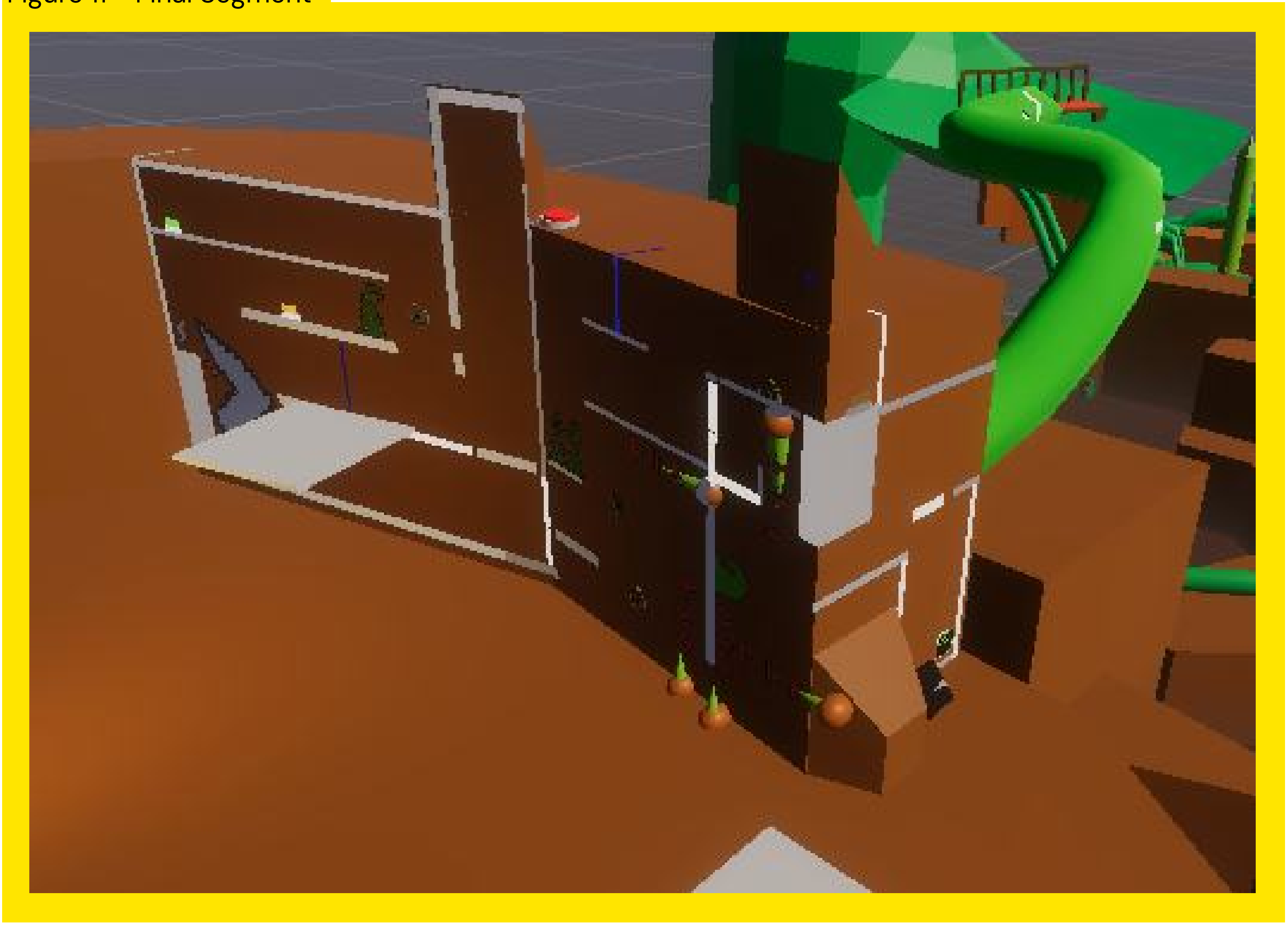
Player (GameObject) - This stores the First Person Player character's gameObject.

The Player variable gets parented to the moving platform when the trigger collides with the 3D Moving Platforms. The trigger then removes this parenting when the player exits the moving platform, and it no longer collides with the trigger.

Segement 10 - Final Segment:

The 10th segment of the game serves as the final one, and is a big finale to the game, incorporating every mechanic previously established, from the Inventory system to 3D buttons and the rock hit ability. To properly incorporate everything however, I needed to create a new system, I needed to create a means for the player to swap between 2D gameplay styles on the fly, so that every mechanic could be properly incorporated.

Figure 11 - Final Segment



SwapCharacter:

This script is the final big mechanic made for the game, and is a trigger, which when collided with, swaps the 2D player type from Atari to Mega Man, and vice versa. This was done using triggers on either side of a decal, which when entered activate the opposite gameplay type onto the other side of the decal. This works to easily teleport the player to either side of the decal, and made the mechanic work very efficiently.

SwapCharacter – Variables:

```
public class SwapCharacter : MonoBehaviour
{
    //This script is used in the final segment of the game to have the player swap between the 2D gameplay styles
    on the fly.

    //This array stores the different gameplay characters
    [SerializeField] GameObject[] Character;
    //This stores the position that the different player types will spawn at when swapping
    [SerializeField] Vector3[] Position;
    //This stores the normal of the different player types after swapping.
    [SerializeField] Vector3[] Normal;
```

This script has three array objects that store the values for the swap. In the arrays, the 0 value is always that of the Atari player's values, and the 1 value is the Mega Man Player. The array Variables are as follows:

Character[] (GameObject) – This array stores the actual GameObjects for the two player characters.

Position[] (Vector3) – This array stores the positions the Player characters get placed in after swapping.

Normal[] (Vector3) – This array stores the Normals that the different Player characters could get spawned at.

SwapCharacter – OnTriggerEnter():

```
    //When entering a trigger the script will swap the player character over to the opposite player
    private void OnTriggerEnter(Collider other)
    {
        //When the Atari Player enters the trigger, it will swap to the Mega Man player.
        if(other.gameObject == Character[0])
        {
            Character[0].SetActive(false);
            Character[1].SetActive(true);
            //When swapping the script automatically updates the positions the player character will be moving
between, so they don't glitch.
            Character[1].GetComponent<Mega_Man_Movement>().OriginMark =
Character[0].GetComponent<DecalMovement>().OriginMark;
            Character[1].GetComponent<Mega_Man_Movement>().TargetMark =
Character[0].GetComponent<DecalMovement>().TargetMark;
            Character[1].transform.position = Position[1];
            Character[1].transform.forward = Normal[1];
        }
        else if(other.gameObject == Character[1])
        {
            //When the Mega Man Player enters the trigger, they will swap to the Atari Player.
            Character[1].SetActive(false);
            Character[0].SetActive(true);
            Character[0].GetComponent<DecalMovement>().OriginMark =
Character[1].GetComponent<Mega_Man_Movement>().OriginMark;
            Character[0].GetComponent<DecalMovement>().TargetMark =
Character[1].GetComponent<Mega_Man_Movement>().TargetMark;
            Character[0].transform.position = Position[0];
            Character[0].transform.forward = Normal[0];
        }
    }
```

The only function used for this script is an OnTriggerEnter() function. This function checks whether it is the Atari or Mega Man player that collides with the player character, and swaps the character to the other side of the portal based on this. In addition, the script updates the OriginMark and TargetMark variables to fit the area it swapped to, so that the character can properly move on the wall without glitching. This worked well and made the mechanic work well. With that, all my central mechanics in the game were programmed, and with that, my game is complete. Thank you for reading threw all my scripts, I hope you didn't die of boredom.