Orbital 24 README

Team Name: The Scriptwriters
https://gao327.itch.io/the-scriptwriters

Team Members:
Roderick Kong Zhang (A0286550Y)
Gao Yening (A0272253J)

Proposed Level of Achievement: Apollo 11

Table of Contents

Motivation

Music is an integral yet often underappreciated aspect of our lives. In our increasingly superficial world powered by the brevity and speed of social media and the Internet, Music is becoming increasingly diluted to suit the masses, and taken for granted. To this end, we aim to create an immersive and engaging mobile game to reconnect people with culture and the historical origins of music from various parts of the world, for them to go back to their roots, and gain a deeper understanding and appreciation for music. Additionally, we hope to ignite our users' passion for music creation through our slew of additional features such as an instrument tuner, metronome, beat maker, chord progression creator, and musical track creator. We plan to create a centralised app for music exploration, education, and creation.

Aim

Our goal is to create a musical top-down 2D RPG mobile adventure game with the following characteristics:

- Free exploration
- Maps covering different regions of the parallel universe where users can explore music of different cultures (e.g., Western (European) classical music, Chinese music, Malay music, Indian music, Indonesian music, etc.)
- Quest-guided gameplay
- NPCs
- Interactive items

Brief storyline of the adventure game:
The Main Character (MC) finds themselves inadvertently entering a parallel universe where music is not merely art, but the very essence of life itself. Here, music carries the power to bind communities, convey emotions, and even perform miraculous feats. Yet, this harmonious existence is under threat.

As the MC embarks on a journey to return home, they traverse diverse landscapes, each resonating with their own unique rhythm. Along this journey, the MC uncovers vital clues and forges alliances, all while aiding the inhabitants in safeguarding their musical heritage.

We aim to only cover one or two regions in the adventure game through the course of our Orbital project, given the limited time we have.

User Stories

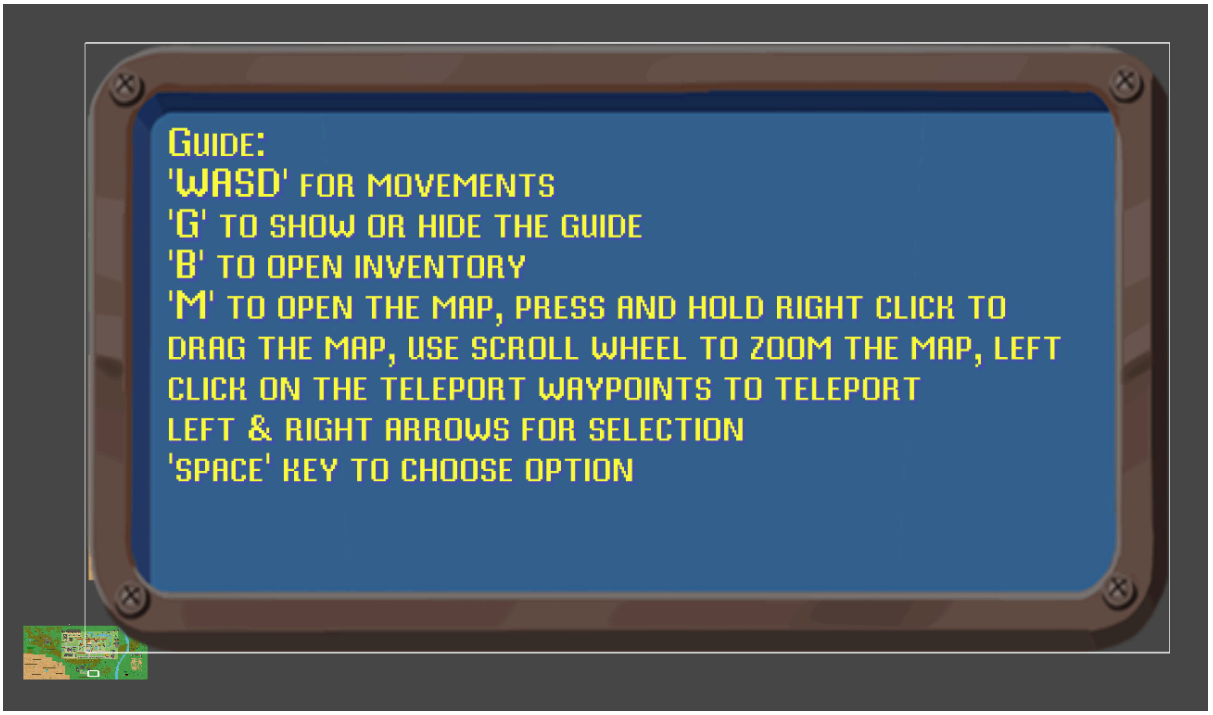
1. As a casual listener, I want to explore diverse music styles through an immersive adventure game, so that I can gain a deeper appreciation for music of various cultures.
2. As an intermediate musician, I want to solve music-related puzzles in an adventure game, so that I can enjoy applying my musical knowledge.
3. As a music enthusiast, I want to interact with an intuitive music creation interface, so that I can easily create music using classical and exotic instruments.
4. As a gamer, I want to solve music-related puzzles, so that I can progress through the game and enjoy the storyline.
5. As an aspiring composer, I want to use the chord progression creator, so that I can develop my musical ideas.
6. As an RPG fan, I want to interact with NPCs that have unique musical backgrounds, so that I can immerse myself in the game's world.
7. As a teacher, I want to recommend this game to my students, so that they can learn about music in an engaging way.
8. As a game developer, I want to integrate background music that changes with regions, so that the game environment feels dynamic and immersive.

Features

1. Tutorial

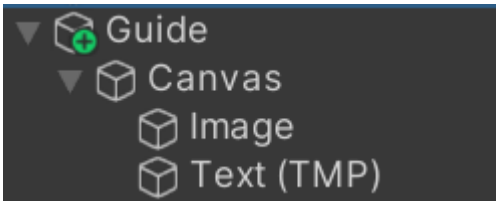
General Concept

For simplicity, the game uses an all-in-one guide for the user to learn how to perform different actions in the game. The guide UI is triggered by the user pressing 'G' button on the keyboard.



Implementation Details

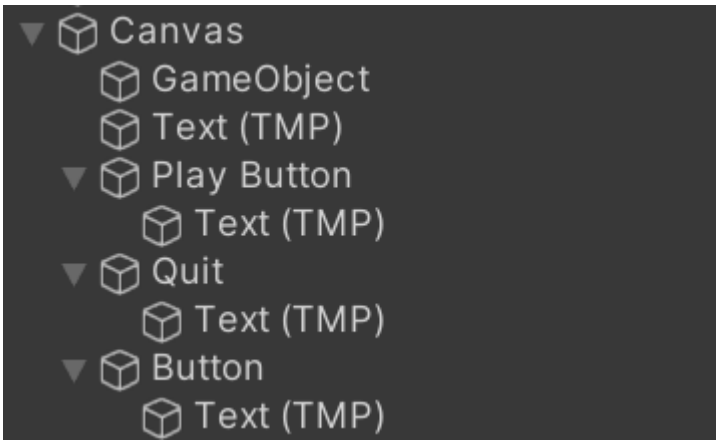
- Implemented using a Canvas with an image as the background and the texts were achieved using textmesh pro.



- Guide_controller script is attached to the Guide gameObject which change the visibility of the canvas.

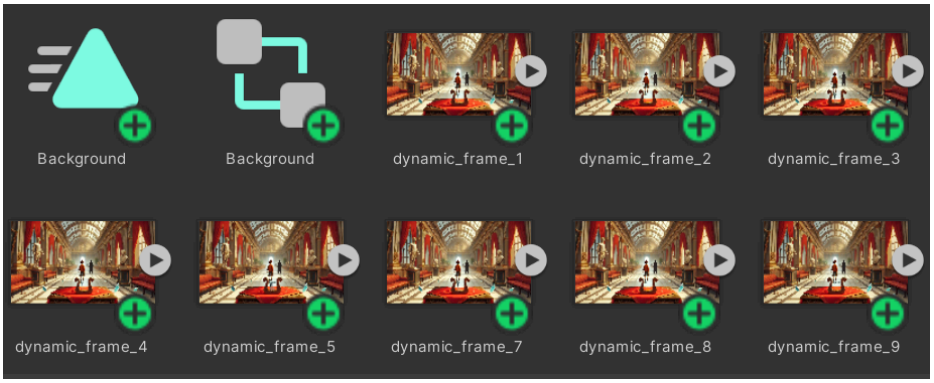
2. Homepage

An animated menu page before player gets into the real game. The homepage is based on UI Canvas gameobject in an empty scene.



Animated Background

The background consists of the main character walking in a museum where a magical lyre was placed. This setting is actually the first scene of the game. The animated background is achieved by creating an empty game object under Canvas. An animator is attached to the game object with animation creating from slight variations of the background picture.



Play and Quit Button

The Buttons in the game are implemented by creating UI → Button - Textmesh Pro GameObject under Canvas. More options such as setting will be included in the future.

Resume

The Resume button allows player to load the status of game they have last saved. (See Save and Load feature below for implementation details).

Save and Load system

General Concept

- This is an important system allowing user to save the game status at various stages of the game and reload the status when they play again. The current implementation is such that the user needs to press F5 to save the game, and click on resume to load back the data.

Current Limitation

- Due to some unresolved issues, player needs to press F9 again to load certain game data. Resume only loads the correct scene. Also, inventory items cannot be saved using the system yet, as the system is still undergoing debugging process.

Implementation Details

The Save and Load system depends on the following:

- SaveData Class: defines a serializable class which stores necessary game data

```
[Serializable]
12 个引用
public class SaveData
{
    public int currentScene;
    public SerializableVector3 playerPosition;
    public Dictionary<string, int> inventory;
    public Dictionary<string, string> inventorySprites; //
    public List<string> storyItems;
    public Dictionary<string, List<string>> conversations;
}
```

- SaveSystem Class: Defined with 2 important functions, SaveGame() and LoadGame(). The former serialise the game data and store locally whereas the latter finds the saved file and deserialise.

```
public void SaveGame(SaveData data)
{
    BinaryFormatter formatter = new BinaryFormatter();
    using (FileStream stream = new FileStream(saveFilePath, FileMode.Create))
    {
        formatter.Serialize(stream, data);
    }
}
```

```
public SaveData LoadGame()
{
    if (File.Exists(saveFilePath))
    {
        BinaryFormatter formatter = new BinaryFormatter();
        using (FileStream stream = new FileStream(saveFilePath, FileMode.Open))
        {
            return (SaveData)formatter.Deserialize(stream);
        }
    }
    else
    {
        Debug.LogError("Save file not found in " + saveFilePath);
        return null;
    }
}
```

- Game Model: A Singleton Class which store all the game data such as MC_Controller (controller for Main Character), InventoryController, etc. Hence, the CreateSaveData() and LoadFromSaveData(SaveData saveData) methods are created under the class.

```
public class GameModel
{
    public MC_Controller MC;
    public DialogController dialog;
    public InputController input;
    public InventoryController inventoryController;
    public MusicController musicController;

    Dictionary<GameObject, HashSet<string>> conversations = new Dictionary<GameObject, HashSet<string>>();
    Dictionary<string, int> inventory = new Dictionary<string, int>();
    Dictionary<string, Sprite> inventorySprites = new Dictionary<string, Sprite>();
    HashSet<string> storyItems = new HashSet<string>();
```

- Game Controller: Implemented as the overall controller which calls function of the above two classes. The code snippets below from the GameController Class shows how saving and loading system functions.

```
public void SaveGame()
{
    SaveData data = model.CreateSaveData();
    saveSystem.SaveGame(data);
    Debug.Log("Game saved.");
}
```

```
public void LoadData()
{
    if (saveSystem.SaveFileExists())
    {
        SaveData data = saveSystem.LoadGame();
        model.LoadFromSaveData(data);
        Debug.Log("Game loaded.");
    }
    else
    {
        Debug.LogError("No save file found.");
    }
}
```

3. Scene Change

Scene Change is an important feature in the game as the MC needs to go to different scenes for different objectives. The feature is implemented by calling the SceneManager.LoadScene() function in UnityEngine.SceneManagement.

4. Start Scene

The story first takes place in a museum where the MC meets a man dressed in black. MC had a conversation with the ManInBlack about the magical lyre. MC is then teleported to the world where the main story takes place.

5. Camera System

The main game uses two cameras - a Main Camera and a UI Camera. The main camera will feature the MC and the world whereas the UI camera will feature UI elements such as message bars. Since they uses the same target display, the images can be superimposed on each other.

Main Camera

In the implementation, Cinemachine is used.

- CinemachineBrain: The CinemachineBrain component is attached to the main camera and manages the blending and transitions between different virtual cameras in the scene, ensuring smooth and cinematic camera movements.
- CinemachineVirtualCamera: CinemachineVirtualCamera components define specific camera shots or views, controlling position, rotation, and behaviour. They can follow or look at targets and are managed by the CinemachineBrain to provide dynamic camera angles.
- CinemachineConfiner: The CinemachineConfiner extension restricts a virtual camera's movement to a specified area, using a 2D collider to define the boundaries. It ensures the camera stays within the intended playable or visible space, enhancing the player's experience by preventing unintended views.

In this case, one virtual camera is created for first and second scene. The confiner takes the boundary of the playing area in each field and the virtual camera will be following the main character (MC). Hence, the MC will be at the middle of the camera until MC reaches the boundary of the playing area.

6. NPCs

Roaming NPC Logic

This type of NPC move autonomously within designated regions. The main logic is setting a radius for NPC movement and allowing it to move in random direction within the radius.

```
Vector2 GetRandomTargetPos()
{
    Vector2 randomDirection = Random.insideUnitCircle.normalized;
    float randomDistance = Random.Range(0, maxDistance);
    Vector2 randomTargetPos = originalPos + randomDirection * randomDistance;

    return randomTargetPos;
}
```

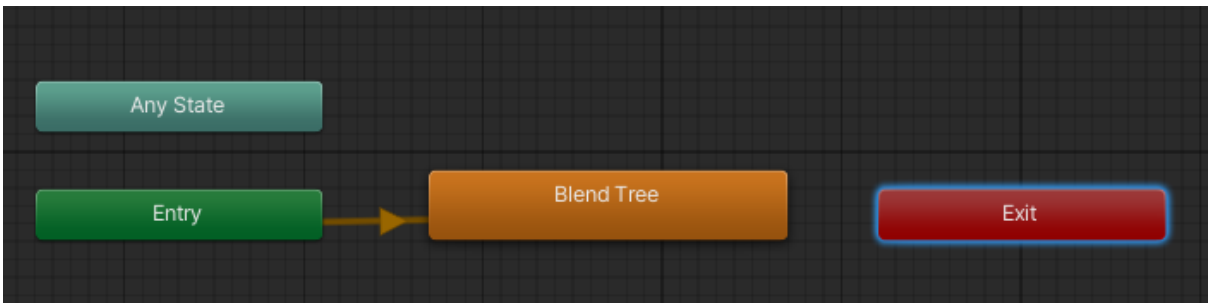
ManInBlack (Quest 1 NPC)

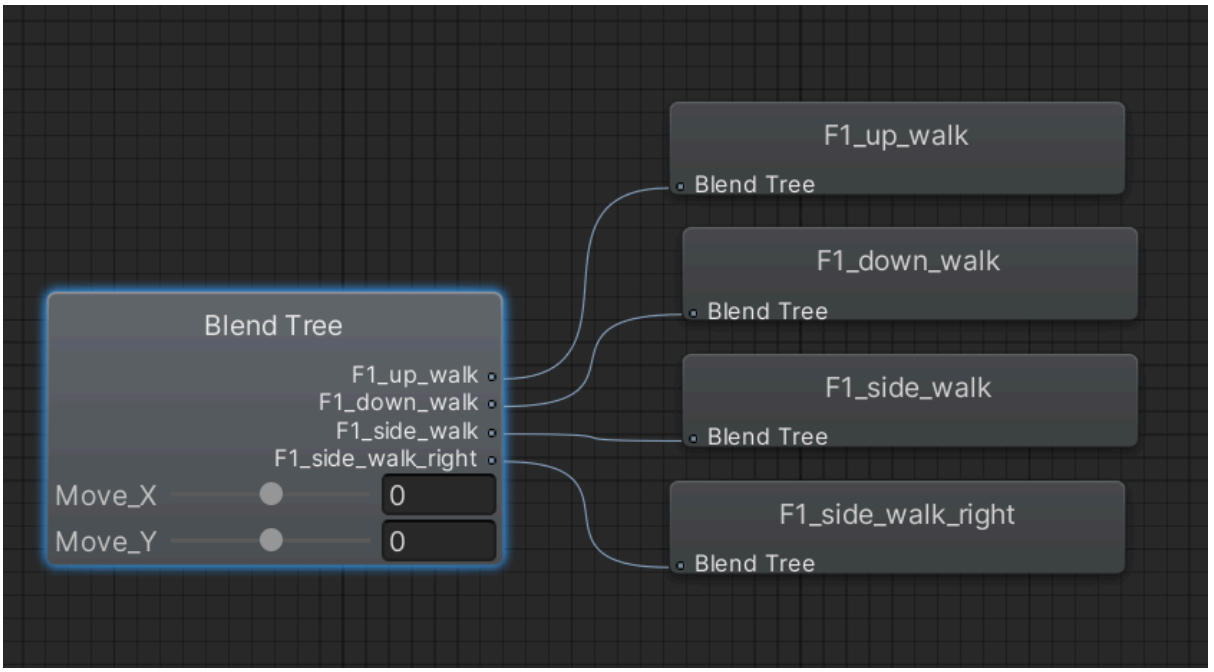
```
public ConversationScript[] conversations;

Quest activeQuest = null;

Quest[] quests;
```

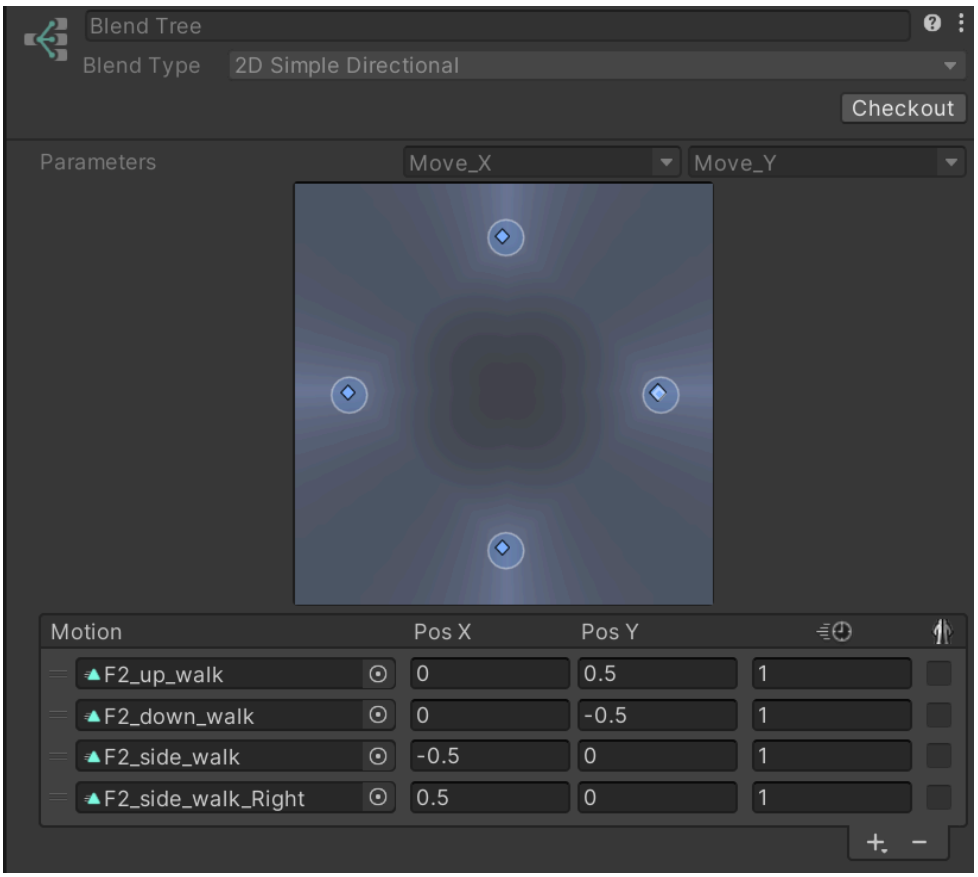
- Quest[] and ConversationScript[] arrays are created to allow quests to be added to a specific NPC, enabling it to have conversations with the MC (Main Character).
- For the ManInBlack NPC, the original concept was that it would only activate when the MC interacts with a specific story item (the inner circle as shown below). Upon activation, the ManInBlack would walk to the MC and engage in a conversation, leading to the start of a quest. In Scene 1, this quest is conversation-based. After the conversation concludes, the ManInBlack would pass an inventory item (the lyre) to the MC. In the latest implementation, this has been simplified: the ManInBlack now waits for the MC at a fixed position.

NPC Animation

- Sprite Credit: https://assetstore.unity.com/packages/2d/characters/3-direction-npc-characters-205182#description by Szadi Art
- Animation clips for NPCs moving up, down, left, and right have been created. An Animator Controller is implemented using a blend tree to determine which animation to use based on the direction the NPC is moving.


Any State
Entry
Blend Tree
Exit


F1_up_walk
Blend Tree
F1_down_walk
Blend Tree
F1_side_walk
Blend Tree
F1_side_walk_right
Blend Tree
Blend Tree
F1_up_walk
F1_down_walk
F1_side_walk
F1_side_walk_right
Move_X
0
Move_Y
0


Blend Tree
Blend Type
2D Simple Directional
Checkout
Parameters
Move_X
Move_Y
Motion
Pos X
Pos Y
F2_up_walk
0
0.5
1
F2_down_walk
0
-0.5
1
F2_side_walk
-0.5
0
1
F2_side_walk_Right
0.5
0
1

7. Conversation

Conversation is implemented by creating a list of Conversation Pieces and a dictionary mapping conversation ID to the Conversation Pieces.

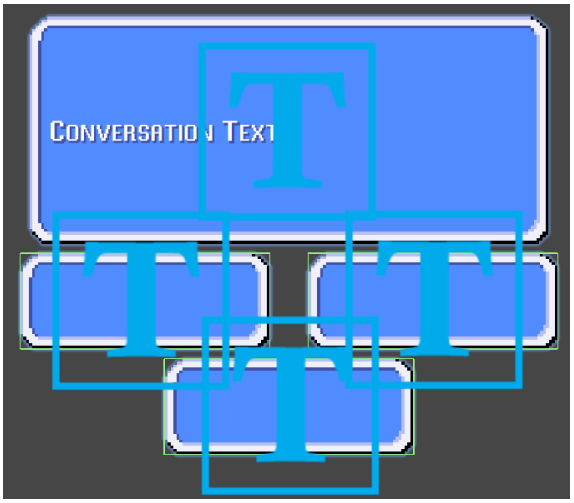
```
Dictionary<string, ConversationPiece> index = new Dictionary<string, ConversationPiece>();
```

Each ConversationPiece includes an options list, which holds multiple Option objects. Each Option object contains: The text for the player's choice, a targetId that links to the next ConversationPiece based on the player's selection.

This structure allows the conversation to branch depending on the player's choices.

Dialog UI

Currently the game uses the prefab Dialog Bar from Creator Kit - RPG. In the future, the sprite may be changed and the buttons will be read input from mouse-clicking.



Similar to the homepage, the dialog bar consists of the dialog panel (upper display), and a number of UI buttons. Texts are added using Textmesh Pro. When a button is pressed, a option handler is triggered to handle the action.

8. Story Item

A Story Item is triggered when the MC (Main Character) reaches a certain area. It is implemented by creating a transparent collider with the "Is Trigger" property enabled, which covers a specific region. When the MC collides with this collider, the OnTriggerEnter2D function in the StoryItem script is called. This function then activates a Creator Kit - RPG prefab MessageBar to display the message. Also, audio specific to the region can be played.

Message Bar

The MessageBar creates a queue to store messages. When called by a story item, it enqueues the message from the story item and displays it. The MessageBar UI is implemented using TextMesh Pro.

9. Inventory

The inventory system in the game allows the MC (Main Character) to collect, manage, and display various items. It is composed of the InventoryController and InventoryItem scripts.

Inventory Item

Inventory items are game objects that the MC can collect. When the MC collides with an item's trigger collider, the item is added to the inventory, a message is displayed using the MessageBar, and the item is deactivated in the scene.

Inventory Controller

- The InventoryController manages the display of collected items in the game's UI. It maintains a list of inventory items and updates the UI when the inventory is toggled using the B key. The items are displayed using a UI element prototype, showing the item's sprite and count.

```
void Update()
{
    if (Input.GetKeyDown(KeyCode.B))
    {
        if (visible)
        {
            sizer.Hide();
            visible = false;
        }
        else
        {
            Refresh();
            sizer.Show();
            visible = true;
        }
    }
}
```

- The controller has drawn reference from the inventory controller prefab in Creator Kit - RPG. The main difference is that the display is triggered by toggling the B (for bag) key.
- The inventory controller is one attribute of the GameModel so due to the following reasons the inventory item is able to stay across scenes:
    - Singleton Pattern: The GameModel is designed as a singleton, meaning only one instance of this class exists throughout the entire game session. This instance is created once and is reused across different scenes.
    - Persistent Data: Since the GameModel instance is not tied to a specific scene, it doesn't get destroyed when the scene changes. The inventory data and other game states are stored in this singleton instance and persist as long as the game is running.
    - Usage Across Scenes: The GameModel instance is accessed and updated by various scripts in different scenes. Any changes to the inventory or other game states are reflected in this single instance, ensuring data consistency.

10. Quest

The quest system allows the MC (Main Character) to undertake and complete quests. Implemented via the Quest script, each quest has associated conversations for in-progress and completed states. When a quest starts, specified game objects are enabled or spawned, and an introductory cutscene may play. Quests require certain items to be completed, and once these requirements are met, the quest is marked as complete. Upon completion, reward items are given to the player, and an optional concluding cutscene is shown. This system ensures a structured flow for starting, progressing, and completing quests within the game.

```
if (Input.GetKeyDown(KeyCode.G))
{
    visible = !visible;
    guideCanvas.gameObject.SetActive(visible);
}
```

11. Main Character

The main character is the controllable character in the game, controlled by MC_Controller.

Sprites are drawn using Aesprite. Animation for the MC is implemented in a similar way as the NPCs (Blended Tree controlled by the direction of movement)

MC Controller Class

- Timer: UnityEngine.Time is used to facilitate attributes updates
- Movement System: Get Axis from input and calculate a target location. Then set the location of the MC to be that target location by fixed update.

```
void FixedUpdate()
{
    if (state == State.Moving)
    {
        MoveState();
    }
}

1 个引用
void MoveState()
{
    Vector2 position = rigidbody2d.position;
    position.x = position.x + speed * horizontal * Time.deltaTime;
    position.y = position.y + speed * vertical * Time.deltaTime;
```

- Power Management: Power Management is a system that is already developed but currently not used in main playing scene. The general Concept is that the MC will have music power and he needs to spend the music power for certain tasks (in future development). He can gain music power through collecting or using certain items. There will also be locations that would cause him to lose music power. In general music power can be seen as HP in this game.

```
public void ChangePower(int amount)
{
    // Check if MC is Invincible.
    if (amount < 0 && isInvincible)
    {
        return;
    }

    if (amount < 0)
    {
        isInvincible = true;
        invincibleTimer = timeInvincible;
    }

    currentPower = Mathf.Clamp(currentPower + amount, 0, maxPower);
    UIPowerBar.instance.SetValue(currentPower / (float)maxPower);
}
```

- As can be seen from the above implementation, ChangePower will be caused by other gameObjects. This will be further elaborated in the damage zone section.
- Timer is again used for calculating the invincible time for the MC. When MC gets damaged, the timer starts and MC will only be damaged again only if MC is still in the damage zone after the invincible time.

12. Overall Scene Design

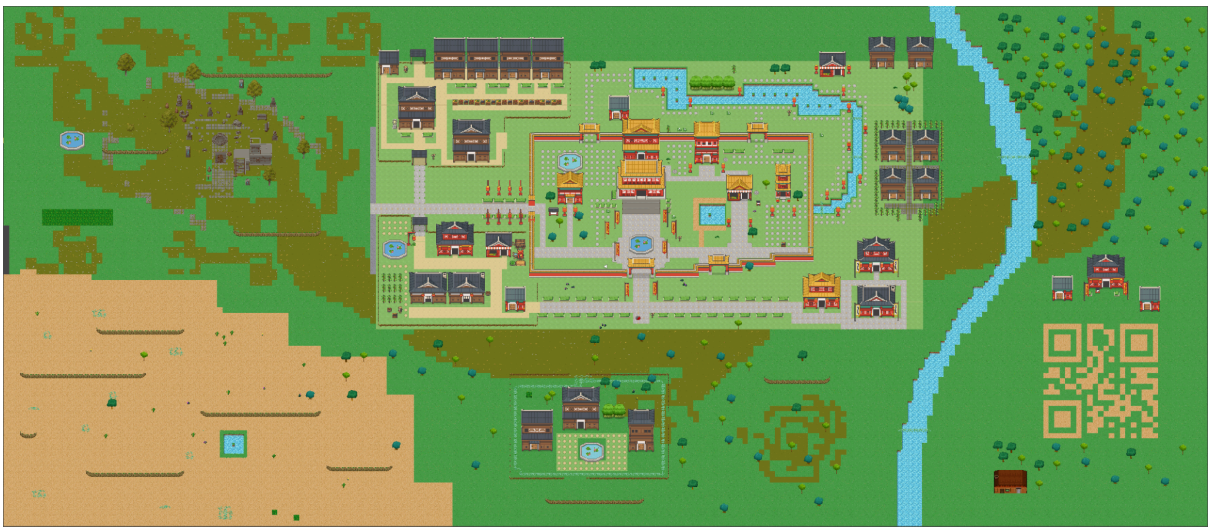
This scene is greatly inspired by Ib. There are some easter eggs in the scene as tributes to Ib.

- Game Flow: MC is able to explore the scene and see the paintings on the wall. When MC collides with the ManInBlack, ManInBlack will start a conversation about the lyre on the table. At the same time, There are two quests attached to the ManInBlack. The first quest is meant to introduce to MC the setting of the main world through conversation. The purpose of the second quest is to give MC inventory item "the lyre" as well as teleporting MC to the second scene. A transparent gameObject (meaning without spriterenderer) is placed near the table which loads the Chinese Scene when MC has a OnTriggerEnter2D collision with it. The gameObject is only enabled when the second quest starts.

13. Scene_Chinese

General Concept

- This scene is first game scene in the main story. The scene is based on elements of Chinese traditional music, hence there are a lot of Chinese influences in the scene design. The music concept this scene will introduce is the Five notes, or the Chinese pentatonic scale, comprising Gong (宫), Shang (商), Jue (角), Zhi (徵), Yu (羽)
- Since the scene is large in scale, the features below are implemented.



Map

General Concept

- 'M' to open the map, press and hold right click to drag the map, use scroll wheel to zoom the map, left click on the teleport waypoints to teleport

Implementation Details

- A Map gameObject is created and under the Map, a UI Canvas which uses Display 1 (the default) is created. Under the Canvas, an Image of the map is added. In the Map gameObject, there is a Map Controller script to control various map functions.
- Press 'M' to open/ close map: Set Map gameObject active/ inactive based on status of KeyCode.M input.

```
if (Input.GetKeyDown(KeyCode.M))
{
    visible = !visible;
    mapCanvas.gameObject.SetActive(visible);

    if (!visible)
    {
        ResetDragState();
    }
}
```

- Handle Zoom: Firstly, the Rect Transform of the image gameObject under the Canvas is set to the middle but not fully stretch so as to allow for room to zoom in and out. Then the game reads Axis input of the mouse scroll wheel, and then change the RectTransform of the map image based on the input reading and a fixed zooming rate.

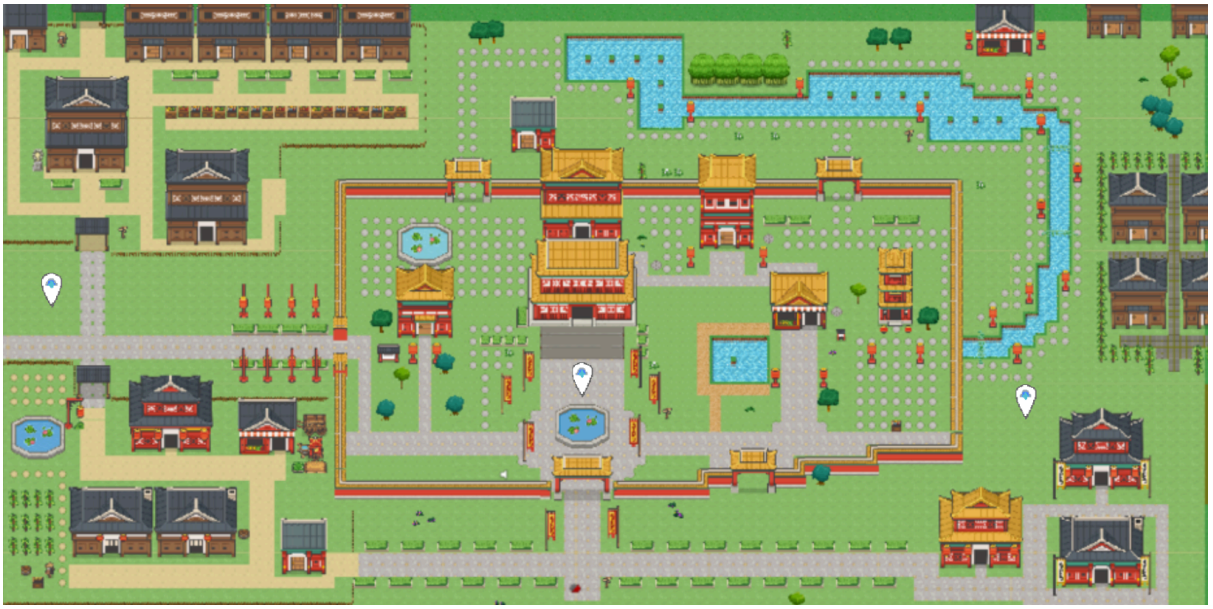
```
void HandleZoom()
{
    float scrollData = Input.GetAxis("Mouse ScrollWheel");
    if (scrollData != 0.0f)
    {
        Vector3 newScale = mapImageRectTransform.localScale * (1 + scrollData * zoomSpeed);

        newScale.x = Mathf.Clamp(newScale.x, initialScale.x * minZoom, initialScale.x * maxZoom);
        newScale.y = Mathf.Clamp(newScale.y, initialScale.y * minZoom, initialScale.y * maxZoom);

        mapImageRectTransform.localScale = newScale;
    }
}
```

- Handle Drag: Player needs to drag the map around so as to navigate to where they want to focus at. It is designed such that player press and hold right click to drag the map. The purpose of doing so is to assign a unique trigger method to the function as left click will be used for teleport function later. This can prevent unwanted behaviour. In the implementation, we need to first check two conditions: right click is pressed down and the mouse is within the map image. When the above conditions are fulfilled, a boolean variable isDragging is set to true. The movement of the mouse will be tracked and the anchorposition of the map image will be transformed based on the change in mouse position, until the right click is lifted up.

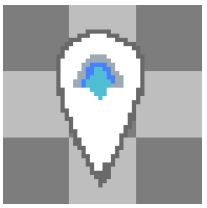
```
void HandleDrag()
{
    // Check if the right mouse button is pressed over the map image before starting the drag
    if (Input.GetMouseButtonDown(1) && RectTransformUtility.RectangleContainsScreenPoint(mapImageRectTransform, Input.mousePosition))
    {
        isDragging = true;
        dragStartPosition = Input.mousePosition;
        Debug.Log("Drag Start Position: " + dragStartPosition);
    }

    // Stop dragging when the right mouse button is released
    if (Input.GetMouseButtonUp(1))
    {
        isDragging = false;
        Debug.Log("Drag End Position: " + Input.mousePosition);
    }

    // Continue dragging while the right mouse button is held down
    if (isDragging)
    {
        Vector3 difference = Input.mousePosition - dragStartPosition;
        mapImageRectTransform.anchoredPosition += new Vector2(difference.x, difference.y);
        dragStartPosition = Input.mousePosition;
        Debug.Log("Dragging: " + difference);
    }
}
```

Teleport Waypoints

Example of waypoints in the central area:



General Concept



- Player can left click on the waypoint to teleport to the corresponding location in the scene.
- The sprite of the waypoints is drawn using Aesprite. I have drawn influence from the Genshin Impact Waypoints.

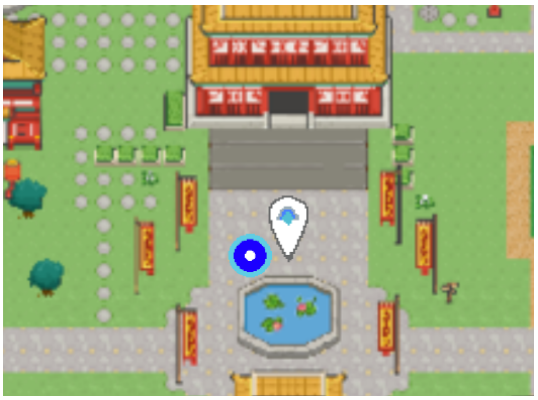
Implementation Details

- The Core of the waypoints is UI button with the sprite above. When clicked it calls functions from a TeleportButtonController Class and a function from the MapController.
- Teleport is implemented using: controller.transform.position = new Vector3(teleportLocation.x, teleportLocation.y, controller.transform.position.z); statement.
- The function in MapController called sets the Map gameObject inactive (which is to close the map) as the waypoint is clicked.
- Dragging uses right, teleporting uses left so as to avoid clashes.

MC_Reference

General Concept

- A reference of MC's real time location on the map image.
- Establish bijection between world coordinate and the map coordinate.



Implementation Details

- Firstly, we need to establish a reference point. Since the map image is set at the centre of the Canvas, we need to make sure the centre of the tilemap (the world created) is set at the centre of the scene. Then we map the world coordinate to the map coordinate based on the ratio between the sizes of the world and the map.

```
Vector2 WorldToMapPosition(Vector3 worldPosition)
{
    // Convert world position to the map's local position
    float mapWidth = 1909.883f;
    float mapHeight = 824.3221f;

    float worldWidth = 220f;
    float worldHeight = 95f;

    // Convert world position to map position
    float x = (worldPosition.x / worldWidth) * mapWidth;
    float y = (worldPosition.y / worldHeight) * mapHeight;

    return new Vector2(x, y);
}
```

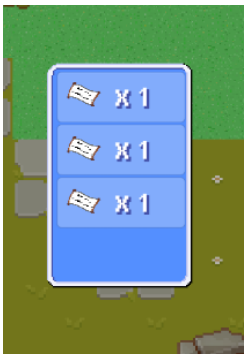
- Then we set the position of the MC_Reference position by markerRectTransform.anchoredPosition = mcPositionOnMap; statement.

Quest

Implementation is similar to the start scene.

Brief play flow of the main quest in Scene_Chinese:

- Player directed to the ruined area of the world.
- Player gets hints by interacting with gameObject.
- Player goes to three runed pillars to collect the following inventory items: Sacred Scroll of Gong/ Shang/ Jue.
- Player goes to a statue. The scrolls are taken by the statue and player receives an inventory item 'essence of five tones'. Also, player will get hint through statue's message, directing player to go to the Altar.
- The Altar consumes the 'essence of five tones' and teleports the MC to the third scene (ancient version of Scene_chinese).



MusicController

Background Music:

Museum (Scene_Start)	Nocturne No.2 in E-flat major Op.9 by Chopin
Main City (Scene_Chinese)	春江花月夜 The Moon over the River on a Spring Night - Chinese Classical
Desert (Scene_Chinese)	On the Vast Earth by Yupeng Chen
Village (Scene_Chinese)	山风拂萝衣 Peaceful Hike by Yupeng Chen
Ruin (Scene_Chinese)	沉睡的往昔 Slumbering Lore by Yupeng Chen

We wanted to compose our own music but did not manage to due to time constraints.

Implementation Details

- Modified upon RPG Creator Kit's Music Controller Prefab. Additional Feature is the music trigger zone feature.
- Dynamic Music Transitions
    - Initialisation
        - Upon starting the game, the MusicController initialises the two AudioSource components and assigns the initial background music to one of them.
        - The initial music track begins playing immediately, setting the ambient tone for the game.
    - Trigger Zones
        - Specific areas in the game world are designated as music trigger zones. These zones are equipped with colliders set as triggers.
        - When the main character (MC) enters one of these zones, the music transition is triggered.
    - Crossfade Mechanism
        - The MusicController handles the crossfade between tracks. When a new AudioClip is assigned, the active AudioSource starts fading out while the other begins fading in with the new clip.
        - The crossfade duration is configurable, allowing for smooth and natural transitions that enhance the gameplay experience.
    - Integration with GameModel
        - The MC_Controller references the GameModel, ensuring that the main character's interactions with the environment, including entering music trigger zones, are tracked.

        - This integration ensures that the correct music is played based on the character's location and actions within the game.

Implemented Features Not Used in Current Scenes (Used in MS1 Proof of Concept)

Power Collectible

- The PowerCollectible component allows the main character (MC) to gain power by collecting items in the game world.
- Key Features:
    - Trigger-Based Collection: Detects collisions with the MC using OnTriggerEnter2D. Increases the MC's power if it's below the maximum limit and then destroys the collectible.
    - Power Management: Utilises MC_Controller to manage the MC's power levels via the ChangePower method.

Damage Zone

- The DamageZone component creates areas where the MC continuously loses power when inside them.
- Key Features:
    - Continuous Damage: Uses OnTriggerStay2D to decrease the MC's power while within the zone.
    - Power Management: Calls ChangePower method in MC_Controller to reduce power.

Future Features

- Minigames
- Scene_Chinese_Ancient

Fully Working Save and Load System	The current version only manages to save the correct scene and correct position. However, it does not successfully save inventory data.
Minigames	Minigames should be a very important feature in our game. There will be a number of music related minigames such as note matching, playing with lyre, etc. Minigames can be in the format of a world quest, such as MC moving to different locations and the controller tracking whether MC has collided with different gameObjects in the correct sequence. Or, it can be on a Canvas with buttons and other UIs. Additionally, the minigame can be an implementation in a different scene.
Scene_Chinese_Ancient	The third scene of the game. MC needs to complete minigames to obtain certain inventory items and return to the present world (second_scene).

Software Design Patterns in the Game

1. Singleton Pattern

The Singleton Pattern ensures that a class has only one instance and provides a global point of access to it. This pattern is implemented in various parts of the game to manage game states and controllers effectively.

Implementations

GameModel

- The GameModel class uses the Singleton Pattern to maintain a single instance of the game's state, including the main character's position, inventory, and conversations.
- This ensures that the game state is consistent and accessible throughout different scenes without multiple instances causing conflicts.

UserInterfaceAudio

- The UserInterfaceAudio class is designed as a singleton to handle audio effects for UI interactions.
- Ensures that only one instance of the audio controller is responsible for playing sounds, avoiding overlapping and conflicting audio clips.

SaveSystem

- The SaveSystem class uses the Singleton Pattern to manage game saving and loading operations.
- Guarantees a single access point for saving and loading game data, ensuring data integrity and consistency.

2. Model-View-Controller (MVC) Pattern

The MVC pattern separates an application into three main components: the Model, the View, and the Controller. This separation helps manage complex applications and promotes organised code.

Implementations

GameModel (Model)

- Represents the game's data and business logic, including player inventory, game state, and conversations.

UI Components (View)

- Classes like InventoryController, DialogController, and PowerCollectible represent the user interface and display the game's state to the player.
- They are responsible for updating the visuals based on the game's current state.

GameController (Controller)

- Acts as an intermediary between the GameModel and the View.
- Handles user input, updates the model, and refreshes the view.
- For example, it manages the game's save and load operations, as well as the transitions between different scenes.

3. Observer Pattern

The Observer Pattern defines a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.

Implementations

Inventory Updates

- The GameModel class holds the inventory data and notifies the InventoryController whenever there is a change in the inventory.
- This allows the UI to reflect the changes in real-time without directly coupling the game logic with the user interface.

UI Refresh

- The InventoryController listens for changes in the GameModel's inventory and updates the UI accordingly.
- This ensures that the inventory display is always in sync with the actual game data.

4. Facade Pattern

The Facade Pattern provides a simplified interface to a complex subsystem. It helps to hide the complexities of the system and provides a single point of access to interact with it.

Implementations

GameController

- The GameController acts as a facade for various game functionalities such as saving/loading the game, transitioning between scenes, and managing the game loop.
- It provides a simple interface for starting a new game, resuming a saved game, and quitting the game, while encapsulating the complex logic behind these operations.

MusicController

- The MusicController simplifies the interaction with the audio system.
- It handles cross-fading between audio clips, playing and stopping music, and adjusting volumes, providing an easy-to-use interface for managing in-game music.

Software Engineering Principles in the Game

1. Single Responsibility Principle (SRP)

The Single Responsibility Principle states that a class should have only one reason to change, meaning that it should have only one job or responsibility.

Implementations

- InventoryController: Manages and displays the player's inventory.
- MC_Controller: Manages the main character's movement and power states.
- MusicController: Manages the background music and transitions.

2. Open-Closed Principle (OCP)

The Open-Closed Principle states that software entities should be open for extension but closed for modification.

Implementations

- Dialog System: New conversation scripts can be added without modifying the existing DialogController.
- Quest System: New quests can be added by creating new Quest objects without changing the quest management logic.

3. Separation of Concerns (SoC)

Separation of Concerns is the principle of separating a computer program into distinct sections, such that each section addresses a separate concern.

Implementations

- UI and Logic Separation: UI elements are separated from game logic. For example, InventoryController handles the UI display, while GameModel manages the inventory data.
- Music Management: MusicController handles music playback and transitions.

4. Don't Repeat Yourself (DRY)

The DRY principle emphasises the importance of reducing repetition within code.

Implementations

- Reusable UI Methods: UserInterfaceAudio provides reusable methods for playing various UI sounds.
- Inventory Management: Methods in GameModel for adding, removing, and checking inventory items are centralised.

Software Testing

Test Item	Test Functionality	Categorization	Expected Outcome
Inventory System	Add Item to Inventory	Unit	Item is added to the inventory and the count is updated.
Inventory System	Remove Item from Inventory	Unit	Item is removed from the inventory and the count is updated.
Inventory System	Display Inventory Items	Integration	Inventory displays all items with correct counts and sprites.
Quest System	Start a Quest	Unit	Quest state changes to active and initial dialog is triggered.
Quest System	Complete a Quest	Unit	Quest state changes to completed and rewards are given to the player.

Quest System	Track Quest Progress	Integration	Quest log updates correctly as player progresses through objectives.
Movement System	Move Character	Unit	Character moves according to player input.
Movement System	Animate Character Movement	Integration	Character's animation changes based on movement direction.
Combat System	Take Damage	Unit	Character's health decreases when taking damage.
Combat System	Deal Damage	Unit	Enemy's health decreases when character deals damage.
Combat System	Character Death	System	Character respawns or game over sequence is triggered upon death.

Dialog System	Trigger Dialog	Unit	Dialog UI appears with correct text when NPC is interacted with.
Dialog System	Display Dialog Options	Integration	Dialog options appear and can be selected by the player.
Save/Load System	Save Game State	System	Game state, including player position and inventory, is saved.
Save/Load System	Load Game State	System	Game state is restored to the saved state.
Music System	Crossfade Music Tracks	Integration	Music crossfades smoothly between tracks.
Music System	Play Sound Effects	Unit	Correct sound effects play for corresponding actions.

User Interface	Display Health Bar	Integration	Health bar updates correctly as player takes damage or heals.
User Interface	Display Power Bar	Integration	Power bar updates correctly as player gains or loses power.
Collision System	Detect Item Pickup	Unit	Items are picked up when character collides with them.
Collision System	Detect Enemy Collision	Unit	Character takes damage when colliding with an enemy.
Teleport System	Teleport Character	Integration	Character is teleported to the correct location when interacting with teleport points.

Map System	Display Map	Integration	Map UI appears and shows the current scene.
Map System	Zoom and Pan Map	Unit	Map can be zoomed and panned correctly.
Map System	Select Teleport Point on Map	System	Character is teleported to the selected point on the map.
Puzzle System	Solve Puzzle	Integration	Puzzle state changes to solved and rewards are given.
Puzzle System	Display Puzzle Hint	Integration	Correct hint is displayed when interacting with a hint object.
Guide System	Display Tutorial Guide	Integration	Tutorial guide appears and can be toggled with a key press.

Unit Testing

Unit testing in our game context involves testing individual components and functionalities in isolation to ensure they perform correctly. For example, unit tests are created to verify that inventory items can be added or removed accurately, character movement responds correctly to player input, and health adjustments are handled appropriately when the character takes damage or heals. These tests are typically automated and use mock objects to simulate interactions, ensuring each component works as expected without dependencies on other parts of the system.

Integration Testing

Integration testing focuses on verifying that different components of the game work together seamlessly. This involves testing interactions between systems, such as ensuring that the inventory UI updates correctly when items are added or removed, dialogues trigger appropriately during NPC interactions, and quests track progress accurately as players complete objectives. Integration tests often simulate real gameplay scenarios to check the communication and data flow between various subsystems, ensuring they function together as intended.

System Testing

System testing is the process of testing the complete game to ensure that all integrated components work together as a whole. This includes verifying that the game can save and load states correctly, music transitions smoothly between scenes, and the map system accurately reflects the player's location and allows for teleportation. System testing involves running the game in various scenarios to check for end-to-end functionality, ensuring that the entire game experience is cohesive, stable, and free from critical bugs. This phase also includes user testing, where feedback from actual players is collected to identify usability issues and areas for improvement.

User Testing

Area of Concern	Bug Description	Solution
Inventory System	Inventory items are not saved correctly across scenes.	Modified the CreateSaveData and LoadFromSaveData methods in GameModel to serialise and deserialize inventory items properly.
Game Save/Load System	Player position is reset to the default position after loading a saved game.	Implemented a teleport function within the LoadFromSaveData method to set the player position after loading the scene.
Game Save/Load System	Inventory is not restored correctly after loading a saved game.	Added detailed logging in the CreateSaveData and LoadFromSaveData methods to ensure inventory items are correctly serialised and deserialized.
Game Save/Load System	MissingReferenceException when accessing destroyed objects during save/load.	Added null checks in the LoadFromSaveData method and updated object references to ensure they exist before accessing them.
UI System	UI elements overlap or do not display correctly.	Adjusted the positioning and layering of UI elements in the DialogLayout and InventoryController scripts to ensure they are displayed correctly.

NPC Interaction System	NPC does not move towards MC or interact correctly.	Refined the NPC movement and interaction scripts in NPCController to ensure they activate and respond as expected, including setting proper flags and conditions.
Dialogue System	Dialogue does not trigger correctly during certain interactions.	Added trigger conditions and refined conversation scripts in the DialogController and NPCController to ensure dialogues are triggered appropriately.
Music Transition System	Music does not transition smoothly when entering new areas.	Enhanced the music transition logic in MusicController to crossfade between audio clips smoothly using the CrossFade method.
Map System	Map zoom and drag functionality does not work as expected.	Refined the map controller script to ensure correct zoom and drag behaviour by adjusting the HandleZoom and HandleDrag methods.
Map System	MC's position on the map does not match actual position in the game world.	Adjusted the conversion logic in MCReference script from game world coordinates to map coordinates to ensure accurate positioning.
Power Collectible	Power collectible does not increase MC's power correctly.	Verified and corrected the power increment logic in the PowerCollectible script to ensure power collectibles affect MC's power as intended.
Damage Zone	Damage zone continuously reduces MC's power without invincibility frames.	Added invincibility frames in the MC_Controller script to prevent continuous power reduction and allow MC recovery time.

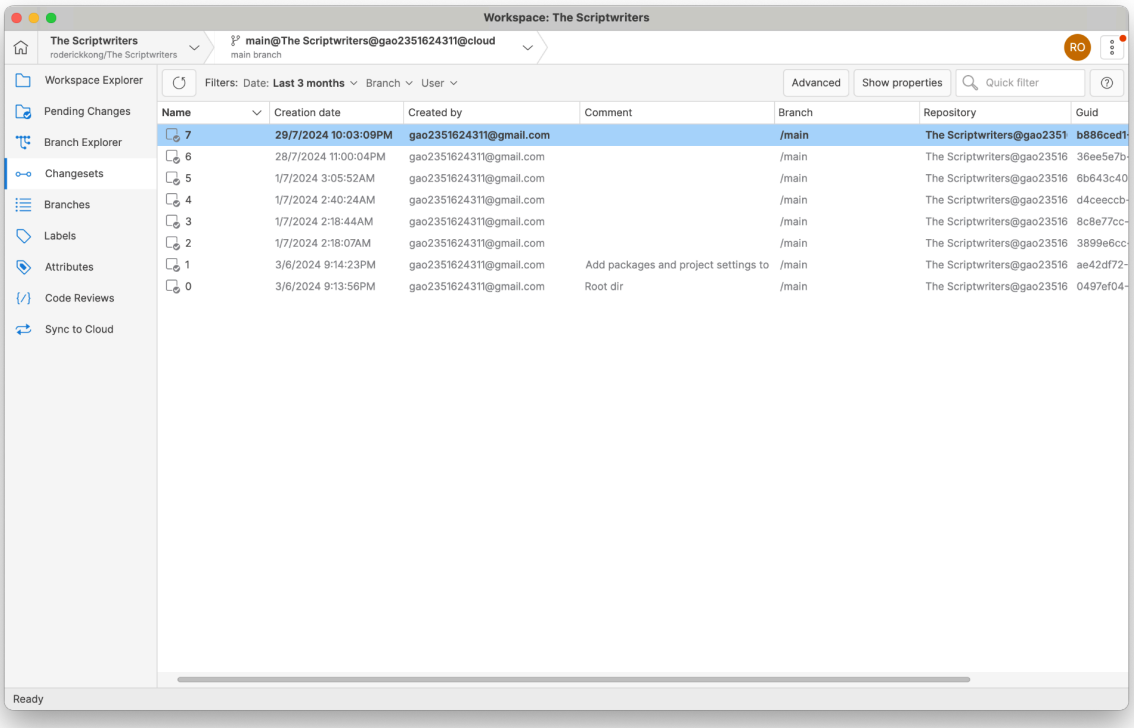
Teleport System	Teleport points do not work consistently or lead to incorrect destinations.	Ensured teleport points are correctly set up and verified destination coordinates in the teleport1 script to ensure consistent teleport behaviour.
Menu System	Quit button does not work as expected in the editor environment.	Added conditionals in the Menu script to handle quit functionality differently in the Unity editor and build environments.
Animation System	MC's animation does not reflect movement direction accurately.	Refined the animation state machine in the MC_Controller script to correctly update animations based on movement direction inputs.
Music Trigger System	Music does not transition when MC enters specific areas with trigger colliders.	Implemented trigger colliders in the SceneChange1 script with associated audio clips and updated the MusicController to handle transitions based on trigger events.

Version Control

Unity Version Control (Previously Plastic SCM)

We started our version control with Unity Version Control primarily because it is much more user-friendly for beginners, allowing us to start sharing and collaborating on our code quickly. Unity Version Control also integrates seamlessly with the Unity Editor, allowing for a more intuitive and streamlined experience when managing changes directly within the development environment, eliminating the need for external tools. It is optimised for managing large binary assets—such as textures, models, and audio files—more efficiently than Git, which can struggle with the size and performance of these assets.

Through Unity Version Control, we constantly updated and merged our code to Unity Cloud whenever we completed a new feature, allowing us to easily note any changes that occurred in our project files or assets. This approach allowed us to collaborate more effectively since team members could work on different features or bug fixes at the same time without causing conflicts. The version control system also provided a safety net by enabling us to revert back to previous versions if any issues arose, ensuring that we could experiment and innovate without the fear of losing critical work. In addition, the review and easy merging of changes also helped maintain project stability and coherence leading to a more organised workflow and smooth development experience.



GitHub, Git Desktop, and Git

However, we eventually moved from Unity Version Control to Git due to various reasons. Git is a widely-used, industry-standard version control system with extensive support and documentation, that is versatile with various types of software projects. Moreover, its distributed nature allows programmers to continue working offline while having full access to the history of their changes. Furthermore, Git's support for pull requests and code reviews improves collaboration and code quality, and its open-source nature makes it a cost-effective solution for version control. Git Large File Storage (LFS) was used to overcome the issue of storing large binary files on the repository.

We used Git to keep track of changes and manage different versions. GitHub acted as the main place to store our project letting us host and share our code well. GitHub Desktop added to this setup with its easy-to-use interface, which made it simple to handle commits, make branches, and combine changes. By using Git's ability to branch and merge, we worked on new features and fixed bugs at the same time without disrupting the main project. The option to make separate branches for different tasks kept our workflow clean and organised, while the merging process helped us smoothly add finished work.