COMP390

2023/24

Farming Sim Game

Student Name:	Tianxing Ji
Student ID:	201676480
Supervisor Name:	Dr Tony Tan

DEPARTMENT OF
COMPUTER SCIENCE

University of Liverpool
Liverpool L69 3BX

Acknowledgements

Firstly, I'd like to thank my parents for their support throughout my studies. Meanwhile, thanks also to my sister and her husband for their encourage during my tough times.

Next, I also want to dedicate this to my friends that I met at University of Liverpool.

Then, I also want to appreciate M Studio and Juhani Junkala, who support the game asset pack and music files in this project.

Last but not the least, I want to express my gratitude to my supervisor, Dr. Tony Tan, who supports me and gives me advice on my Final Year Project along with my second marker, Mr. Keith Dures.

UNIVERSITY OF
LIVERPOOL

COMP390

2023/24

Farming Sim Game

DEPARTMENT OF
COMPUTER SCIENCE

University of Liverpool
Liverpool L69 3BX

Abstract

This project aims to recreate similar features of Stardew Valley and then expand the gameplay based on it. Meanwhile, the development of the entire project was done on Unity, which includes the invocation of plugins such as 'Cinemachine' and 'DOTween'. However, the difficulty of recreating similar features was underestimated, therefore the project ended up only completing the main gameplay part, which includes Bag, Seed Growth, Non-Player Characters, Trade System, Animation, Music, and Game Data Storage. Therefore, in the Conclusion, future refinements were brought up, which are mainly expansions to the content of the game rather than gameplay.

Statement of Ethical Compliance

Data Category: A
Participant Category: 0

I confirm that I have followed the ethical guidelines during this project. Further details can be found in the relevant sections of this dissertation.

Table of Contents

Table of Figures

1. Introduction & Background

1.1 Introduction

Generally, games are the most ubiquitous kind of virtual worlds with a lot of positive functions [1]. To be more specific, they can be used to make money, gain reputation and the stuff like that while the most important part is that games are fun [1]. To conclude, games are used to help players or developers to gain mental pleasure through money or fame.

Meanwhile, game types, like game genres, are known as every single but continuous moment of playing the game [1]. For example, the most engaged experience of players, carefully tracking altimeter changes, using joysticks to control the airplane in a virtual world, can be called as game type [1]. However, compared to countless varieties of game type existing and in use in a lot of popular games, the game types, that are concerned mainly, all use some forms of 'interactive fiction' [1], which simulates a virtual world and enables player to gain his/her own experience in this virtual world. As a result, this kind of game type with virtual worlds is different from other non-world games [1], which is called sim game (simulation game).

To be clearer, sim game, which is also called simulation game, include numerous categories of games that enable players to transform life, ecosystems, cities or even the world in the games [1], such as Sid Meier's Civilization with world simulation that allows players to develop a special civilization [2], which uses the similar game type elements after the classic 'city-building' simulation game SimCity [1].

However, apart from this kind of huge sim games where gamers can construct a city or a civilization, there are also some tiny simulation games that allow gamers to be digital farmers. Among them, one of the most popular ones is Stardew Valley, whose developer is encouraged by a traditional farming simulation game - Harvest Moon [3]. Back to the development of this game, Barone, who is the developer and a long – time Harvest Moon fan, thought the gameplay of Harvest Moon is linear and then wanted to make attempts to rich the experience of play this kind of farming simulation game and bring that gameplay into the modern era [3].

Therefore, as a fan of Stardew Valley, Author also wants to do the similar things that take Stardew Valley as the root of this game and extend the gameplay experience to make this type of game more fun. However, as a 4-year development result [3], it is not realistic to extend numerous things. Hence, it would also be accepted if Stardew Valley can be simulated through tutorials online and the aims of basic functions can be achieved.

At the same time, according to the tutorials online [4], Unity and its scripting API, which can give a lot of powerful functions to game developers, is chosen to create the game that simulates Stardew Valley [5], with off-the-shelf game assets from Unity Assets Store [6]. Moreover, though jMonkey Game Engine and Unreal Game Engine can also be used to make this game, considering the adaptability to the development environment of the author, which is Mac OS system, as well as the gap in the number of tutorials and third-party plug-ins, Unity is the most suitable for this project, which is a 2D game.

1.2 Aims and Requirements

The aims of this project are:

- To get used to the code development of a whole game.
- To be able to combine different game elements such as music and game assets.
- To master some necessary functions of a common game engine.

The requirements of this project (the farming sim game will include):

- The function of a digital bag that can store things and tools.
- The function of seed growth process simulation.
- The function of constructing several buildings.
- The function of communicating with non-Player Characters (NPCs).
- The function of a reasonable daily routine for NPCs.
- The function of a trading system that can allow players to sell and buy goods.
- The function of animation triggered at certain points.
- The function of music playback.
- The function of storing and loading the data.

1.3 Glossary

Diagram 1-The Glossary of Terms in Unity

Alpha	Represents the transparency component of a colour, defining its opacity.
Asset	Any project resource used in Unity, such as 3D models, audio files, scripts, etc.
Audio Mixer	Allows developers to implement advanced audio mixing and sound effect controls within a game.
Blend Tree	Used within an Animator Controller to blend multiple animation clips based on parameters.
Canvas	Manages and renders all UI elements such as buttons and text labels.
Collider	A physics component that defines the physical shape of an object for collision detection purposes.
Component	Modular parts attached to Game Objects that give them specific functionality, such as physics or scripting.
Game Object	The fundamental objects in Unity that represent characters, props, and scenery.
Inspector	A panel in Unity that displays and allows editing of the properties of the selected Game Object or asset.
Ray Casting	The process of shooting invisible rays from a point to detect objects along the path of the ray.
Rigid body	A physics component that allows a Game Object to act under the control of physics like gravity and collisions.
Scene	A container where Game Objects are organized and designed, representing different levels or environments in the game.
Script	Code files that add logic and behavior to Game Objects, typically written in C#.
Sprite	A 2D graphic that can be used for static images, animated characters, and other visual elements in a game.
Time.deltaTime	The time in seconds it took to complete the last frame, used to make movement smooth regardless of frame rate.
URP (Universal Render Pipeline)	A highly customizable rendering pipeline that provides enhanced graphics capabilities and efficiency for Unity projects.

2. Design & Implementation

In this part, the overall development process of this game will be discussed and illustrated, including the original design, encountered problems and technical achievements. Meanwhile, because the development method is agile, where the design and implementation are conducted simultaneously, there will be several sub-sections to demonstrate the process of different parts of this game. To be more specific, the whole game development sequence is first for the construction of the basic game structure and related data structure, then for the data processing to realize the functions, and finally for the addition of animation and music that make the game more immersive.

2.1 Main Style Determination

Design:

Figure 1-Screenshot of Stardew Valley Advertisement (From [7])

Explanation:

Since the main purpose of this game is a recreation of some of the features of Stardew Valley as shown in Figure 1, the pixelated style and 2D platforming construction were the first things that were determined. However, as a coder without the experience to create game assets, pixelated style assets for the game are impossible to be created by hand. Thus, the most suitable off-the-shelf game assets should be picked for this game.

Implementation:

Figure 2-Screenshot of Selected Game Assets Pack (From [6])

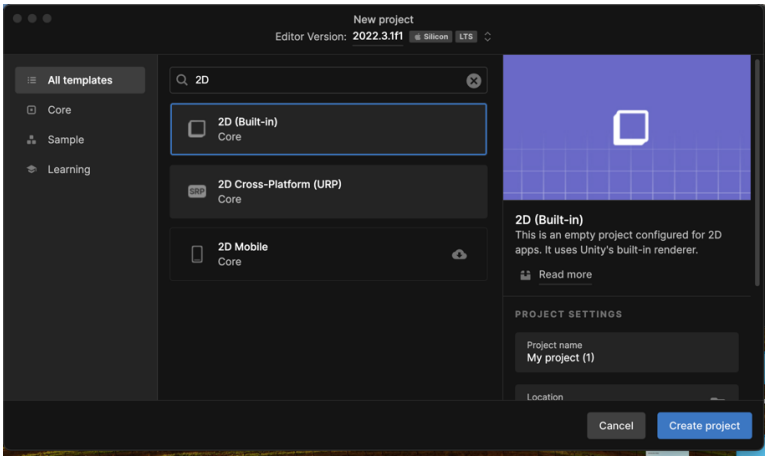


Figure 3-Screenshot of Project Selection

Explanation:
For the pixelated game assets, an assets pack for farm in the Unity Assets Store [6], which is shown in Figure 2, was selected for this project because it includes all the basic elements for a farm game such as animations, crop tools and character components. In other words, the use of this pack reduced the complexity of development process and shifted the center of game development to the code, which made the project more realistic to realize for a coder. Meanwhile, For the 2D platforming construction, 2D (Built-in) template was picked for this project instead of 2D (URP) as shown in Figure 3 because URP is not a necessary component, which can be added later if required. Moreover, the version of Unity Editor is 2022.3.1f1, which was the latest official version that was robust and steady when the project was started.

Problems Encountered:

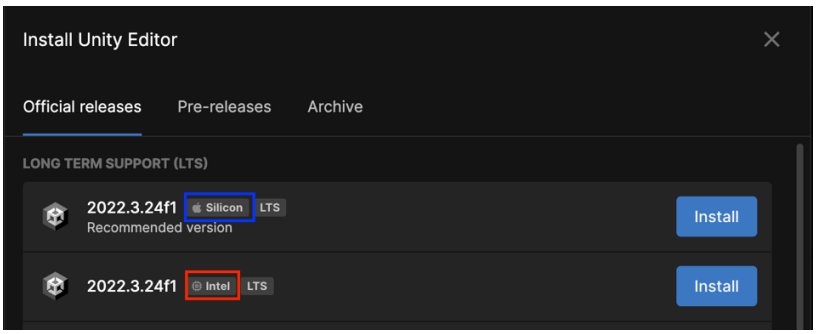


Figure 4-Two Kinds of Versions of Unity Editor

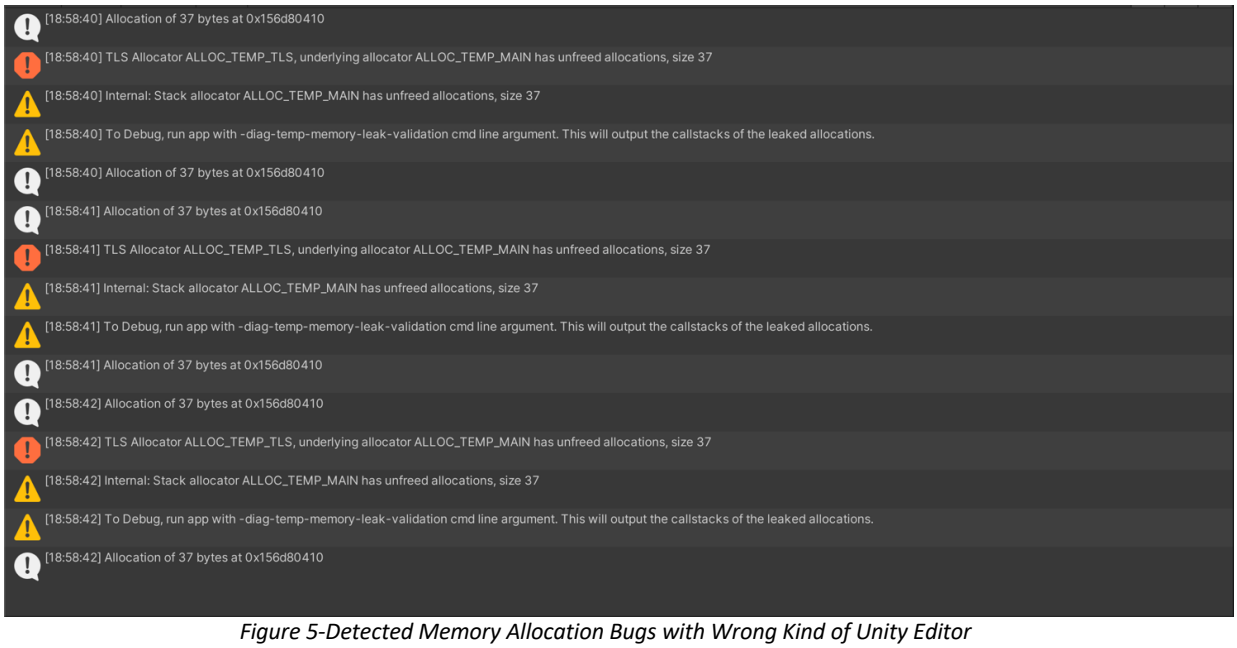


Figure 5-Detected Memory Allocation Bugs with Wrong Kind of Unity Editor

Explanation:
The biggest problem encountered in this section was the wrong choice of Unity Editor kind. Specifically, as shown in Figure 4, the blue box selection ‘Silicon’ represents Apple’s new generation of chips, while the red box selection ‘Intel’ represents Apple’s older generation of chips. Therefore, as shown in Figure 5 illustrating unmatched memory allocations caused by different chips, the wrong choice of Unity Editor type conflicted with the construction of the computer chip, causing the project to crash several times at the beginning of the project.

2.2 Player Creation

Design:

Figure 6-Screenshot of the Final Player Creation

Explanation:

Since the off-the-shelf game assets were applied into this project, the way to put up the components to form the player is shown in Figure 6.

Implementation:

Figure 7-The Components of Player

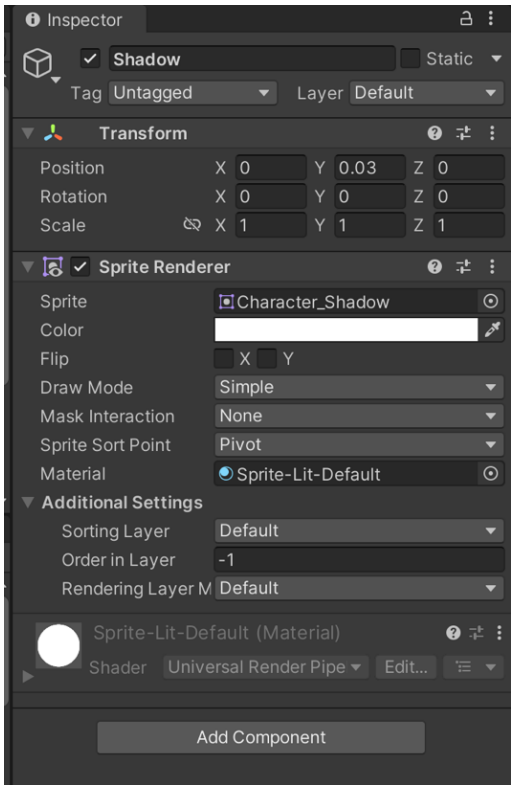


Figure 8-The Inspector of Shadow of Player

Table 1-Order in Layer of Each Component of Player

Components	Order in Layer
Shadow	-1
Body	0
Hair	0
Arm	1
HoldItem	2
Tool	0

Explanation:
Because the property of Unity render order, which is rendering the components from the bottom to the top [8], the components of player were then placed as shown in Figure 7. To be more specific, in this structure of hierarchy, the unity will render from 'Tool' to 'Shadow'. And the reason why 'Tool' and 'Hold Item' components exist in the Player is because the player should have the animation to use tools and hold items, because of which these two components are placed into the player to display animations later. However, rendering order of Unity does not dictate the occlusion relationship between components, so 'Order in Layer' must be set for each component in the Inspector in Figure 8, and the parameters are shown in Table 1. In other words, as shown in Table 1, 'Hold Item' has the highest priority, which will occlude other components when called. Meanwhile, 'Shadow' will hide behind all the components while 'Arm' will still display before the main body components – 'Body', 'Hair', and 'Tool'.

Problems Encountered:
The biggest problem encountered in this section was to understand the occlusion relationship between components of Player, which has been discussed in the Explanation part above.

2.3 Basic Player Movements

Design:

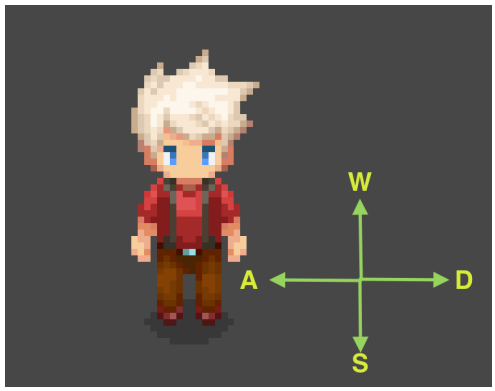


Figure 9-Basic Movements of Player Assigned with Different Keys

Explanation:
The basic movements, which stands for moving up, down, left, and right, were designed to be activated by the keys – W, A, S, and D. Additionally, movement in the oblique direction will be triggered by two keys being pressed at the same time. For example, when the player wants to walk to the upper left, all that is needed is for W and A to be pressed at the same time.

Implementation:

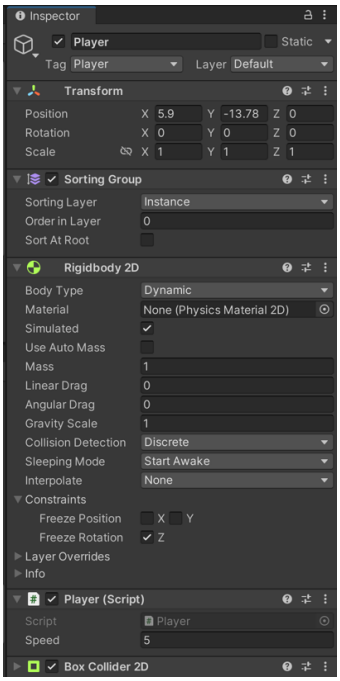


Figure 10-Inspector of Player

```
// set a method to supervise the input on keyboard
private void PlayerInput()
{
    // read the input from keyboard including horizontal and vertical positions
    inputX = Input.GetAxisRaw("Horizontal");
    inputY = Input.GetAxisRaw("Vertical");

    // if X and Y are pressed simultaneously, which means the movement is not straight
    if(inputX != 0 && inputY != 0)
    {
        // set a limited speed
        inputX *= 0.6f;
        inputY *= 0.6f;
    }

    // if the LefShift is pressed, it will make the speed slower
    if (Input.GetKey(KeyCode.LeftShift))
    {
        inputX *= 0.5f;
        inputY *= 0.5f;
    }
    // combine inputX and inputY
    movementInput = new Vector2(inputX, inputY);

    isMoving = movementInput != Vector2.zero;
}

// use the rigidbody to move the player visually
private void Movement()
{
    // In the view of this farming game, instead of adding force, we just change the values of axis value by adding the position change
    // Time.deltaTime is to match different fps
    rigidbodyPlayer.MovePosition(rigidbodyPlayer.position + movementInput * speed * Time.deltaTime);
}
```

Figure 11-Functions in Player (Script) to Implement Basic Movements

Explanation:

To make the player can move, ‘Sorting Group’ and ‘Rigid body 2D’ were added to the player in the inspector, which was shown in Figure 10. To be more specific, the former one is to combine all the components of player such as ‘body’ and ‘hair’ to form a group [9], after which they can move as an object. Meanwhile, the latter one – ‘Rigid body 2D’ is used to simulate the physical elements like speed and mass in a 2D game [10].

Moreover, ‘Player (Script)’ was also added as an element to control the player. To implement the basic movements, two main methods, which are ‘Player Input’ and ‘Movement’, were created.

For ‘Player Input’, the input of user is detected and converted into two values – ‘input X’ and ‘input Y’, which stand for the horizontal and vertical inputs respectively. After that, a new 2D

Vector – ‘movement Input’ will be generated based on the detected two values, which awaits to be used.
For Movement, ‘Rigid body 2D’ is called to move at a given speed and position by using the built-in function – ‘Move Position’ and the calculated variable – ‘movement Input’.

Problems Encountered:

Figure 12-Screenshot of Movements without Freezing Z-axis and Setting 0 Gravity Scale

Explanation:
The biggest problem was forgetting to set Gravity Scale to 0 and freeze rotation, so the player had the same strange movement as shown in Figure 12 – the player was rotating and falling. The reason is that this game is a 2D God’s perspective game where the user can have an overview of the scene, so on the one hand, Gravity Scale should be set to 0 instead of 1 because the gravity scale is by default a vertical downward value with the screen as the reference, whereas this game should have the ground as the reference, and should be vertically inward. Therefore, the gravity does not exist in this perspective, which then should be set to 0.
On the other hand, the player should not rotate, which is more reasonable in this game.
Hence, the rational solution to this problem would be to set the parameters of ‘Rigid body 2D’ to the values shown in Figure 10.

2.4 Basic Map Creation

Design:

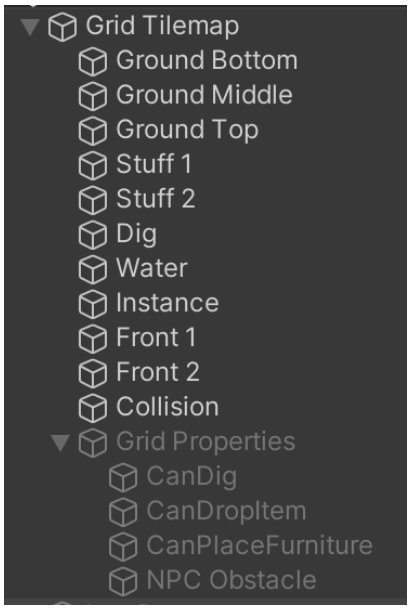


Figure 13-Grid Map Layers and Properties

Explanation:
The use of each layer has been shown in the table below:

Table 2-Explanation for the Purpose of Each Layer

Grid Tile Map Layer	***Purpose of the Layer***
Ground Button	Use brown pixels (soil) to fix the map size
Ground Middle	Use different pixels to decorate the map
Ground Top	Give the map depth with pixels (hill)
Stuff 1	Decorate the map with flowers or other
Stuff 2	Same as above but with different stuff
Dig	Used to show the dug layer
Water	Used to show the watered layer
Instance	The same layer with Player
Front 1	Used to block Player
Front 2	Same as above but with different stuff
Collision	Restrict the movements of Player
Grid Properties	Assign different Properties to facilitate judgement

As shown in Table 2, the map is composed of several different layers, which can largely facilitate the creation and change of maps. Meanwhile, for the later functions such as 'Dig', 'Drop Item' and 'Place Furniture', three properties have been created in 'Grid Properties' to help the judgements to finish the above functions as shown in Figure 13. Moreover, for the reasonable travelling paths of NPCs, another property – 'NPC Obstacle' has been created to help the judgement in A* algorithm to enable the movements of NPCs.

Implementation:

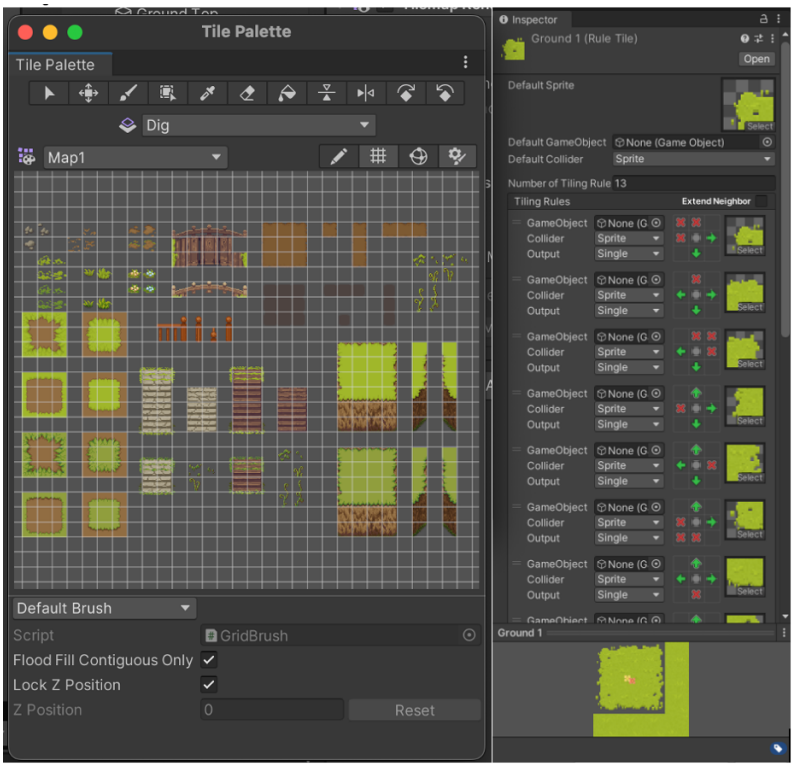


Figure 14-The Unity Palette and The Unity Rule Tile

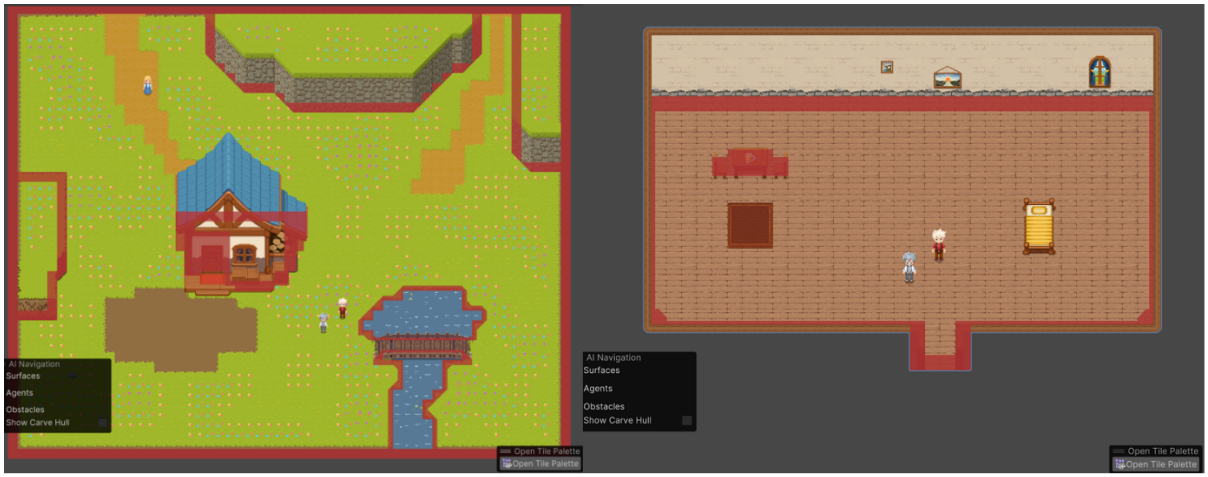


Figure 15-The Two Final Map Drawings

Explanation:

The main way to draw maps is to use the two built-in Unity Tools in Figure 14 – Palette and Rule Tile, where Palette can be used to import mapping materials from the game pack in Figure 2, and then brushes or other tools in Palette can be used to make large-scale maps or small-scale retouches [11]. However, some of the ground material has regular distributions, such as the leftmost pieces of the map in Figure 14. In this case, human drawing tends to disrupt these distributions, thus making the map irregular. Hence, Unity Rule Tile is used to simplify this drawing process. More specifically, by restricting where each piece of map material can appear, a Rule Tile that automatically refines the rules of map material can be got [12], whose drawing result is shown in the bottom right corner of Figure 14.

Meanwhile, there are two maps in this game, as Figure 15 shows. The one on the left of them is what Farm looks like, and the one on the right is what it looks like inside the house. Moreover, there are some red transparent squares in them, which are drawn in Collision Layer. As Table 2 shows, they are used to restrict Player movement, preventing the Player from moving into unexpected places.

Problems Encountered:

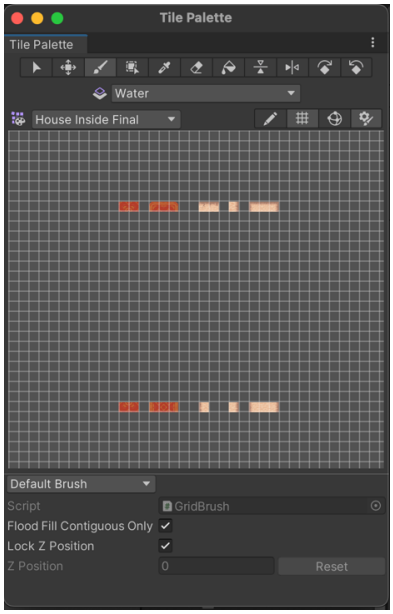


Figure 16-The Unexpected Palette of In-house Material

Explanation:
Although Unity Palette is a powerful tool, it cannot make sure that it can work normally all the time. As shown in Figure 16, it cannot convert the same type of material into a normal one as shown in Figure 14, where all the items in Palette are close to each other. Therefore, it made it harder to map the house. Furthermore, this problem has not been solved so far, even though more material conversions have been tried.
This problem is most likely a problem with the material itself, or a problem with this version of the Unity Editor - 2022.3.1f1. However, since there does not need to be many interactive elements in the house, even If the house is empty, it does not affect the development of the whole game.

2.5 Camera Settings

Design:

Figure 17-Normal Range of Camera

Explanation:
As shown in Figure 17, the normal range of camera, whose position cannot be changed, is just a fixed blue rectangle. In other words, the camera range cannot be changed or moved as Player changes his position. This is not the expected result. The right camera in a game should follow the user, thus the user can observe what happened or will happen to Player and react to the game.

Implementation:

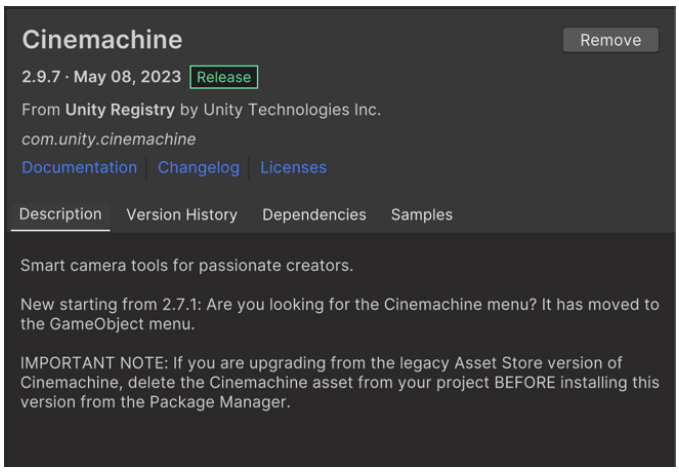


Figure 18-Introduction of Cinemachine (From [13])

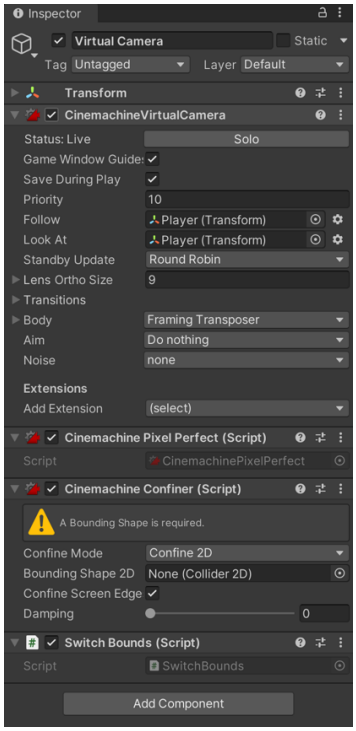


Figure 19-Screenshot of Inspector of Cinemachine Camera

Explanation:

To make the camera actively follow the Player, the easiest way is to add the 'Cinemachine' tool to the Unity Editor [13], then create a 'Virtual Camera' as shown in Figure 18, and in the Component of the 'Cinemachine Virtual Camera', set the 'Follow' and 'Look At' objects to Player. After that, the camera will follow the movement of Player.

Problems Encountered:

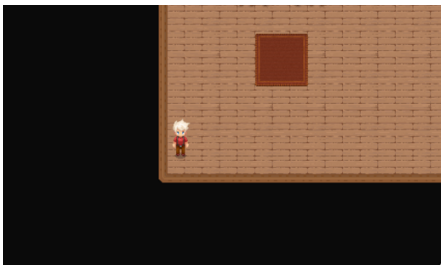


Figure 20-Camera Range Beyond the Boundary

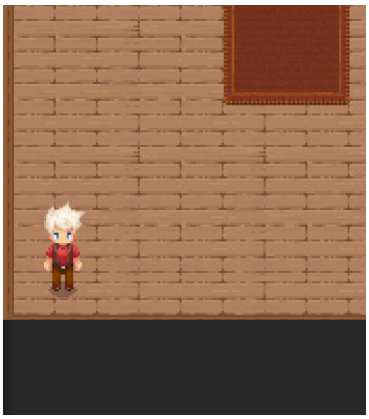


Figure 21-Fixed Camera Range After Setting Cinemachine Confiner

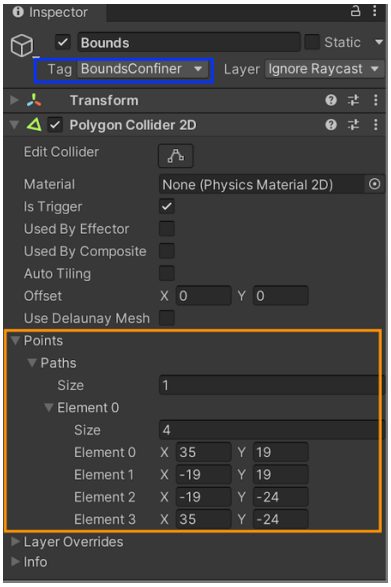


Figure 22-Bounds for Cinemachine Confiner Shape

```
// define a method to find the value of polygan 2D shape of the other shape
private void SwitchConfinerShape()
{
    // tag the bounds shape we need and then use this method to find them
    PolygonCollider2D confinerShape = GameObject.FindGameObjectWithTag("BoundsConfiner").GetComponent<PolygonCollider2D>();

    // before assigning the value to the confiner of original cinemachine, we need to get this component in the code
    CinemachineConfiner confiner = GetComponent<CinemachineConfiner>();

    // assign the value to it
    confiner.m_BoundingShape2D = confinerShape;

    // call this method to clean the plath cache: call this if the bounding shape changes at runtime
    confiner.InvalidatePathCache();
}
```

Figure 23-Code Snippet of Function to Switch Confiner Shape

Explanation:

The following camera shot soon created a new problem, as demonstrated in Figure 20, when the Player was close to the border, the camera shot suddenly exceeded the border, which directly presented a large range of black areas to the Player. Meanwhile, the 'Cinemachine Confiner' in 'Cinemachine' solved this problem [13]. The thing that needs to do is set up the 'Bounds' in the map as shown in Figure 22, specifically using the 'Polygon Collider 2D' plugin to frame the limits of the camera [13]. Then the 'Collider 2D' in 'Bounds' can be assigned to the 'Cinemachine Confiner'.

However, when it comes down to it, Confiner cannot read data across scenes by default, which means that the code (script) is the key to solving the problem. Hence, 'Switch Bounds (Script)' was created and added as a component to the 'Cinemachine Virtual Camera' as shown in Figure 19. Moreover, the most critical method in this code is shown in Figure 23. Specifically, the 'Collider 2D' in the components of the 'Bounds' in the cross-scene needs to be found first (In this case, 'Tag', boxed in blue rectangle in Figure 22, is used for searching), and then assigned to the 'Confiner', where the 'InvalidatePathCache' method in the last step is forced by the Unity Document [14].

Finally, all the problems were solved and the desired result in Figure 21 was obtained.

2.6 Scenery Decorations

Design:

Figure 24-The Design of Scenery Tree

Table 3-The Components of Scenery Tree

Scenery Tree	Component
Top	The position of leaves
Tree_Trunk	The position of trunk

Explanation:
In a scene, something of the scenery decorations is expected. Although they cannot be interacted with, they can add to the aesthetics of the scene. Hence, the Game Pack in Figure 2 has the appropriate material for that. As shown in Figure 24, a Scenery Tree can be created and added to the scene, whose structure is shown in Table 3.
In addition to this, simple Animations can be implemented through the Unity Editor, which will be shown in the Implementation and will be used in many sections below in this project.

Implementation:

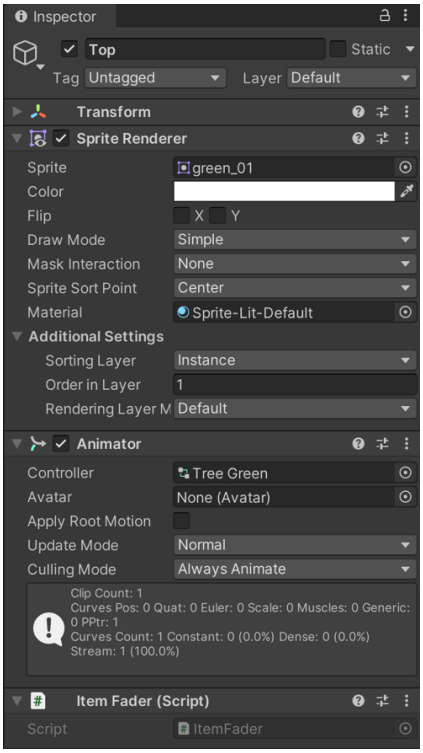


Figure 25-The Inspector of Top of Scenery Tree

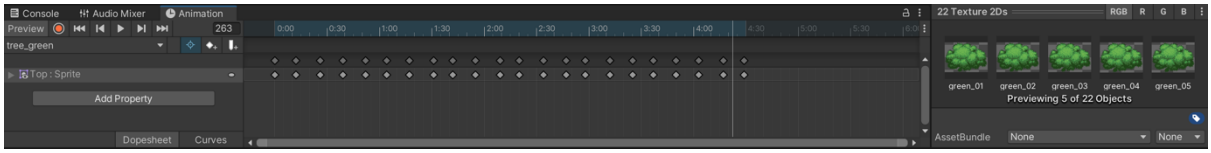


Figure 26-Animation Creation with 22 Objects of Scenery Tree

Explanation:

The process of creating a Scenery Tree is not much different from that of 2.2 Player Creation, especially since there are fewer Layers to analyze, but instead it is simpler. The only different point in this process is adding the 'Animation' component as shown in Figure 25 and create an 'Animator' as shown in Figure 26 [15], where 22 changing pictures of the Scenery Tree are used to simulate the Scenery Tree moving in the wind.

Problems Encountered:

Figure 27-Screenshots of Walking Behind a Tree Before and After Using DOTween

```
using DG.Tweening;
using UnityEngine;

// the Object must have the spriteRenderer
[RequireComponent(typeof(SpriteRenderer))]

public class ItemFader : MonoBehaviour
{
    // get this element
    private SpriteRenderer spriteRenderer;

    // call at the start to assign the value to this variable
    private void Awake()
    {
        spriteRenderer = GetComponent<SpriteRenderer>();
    }

    // to call this method after the situation was finished
    /// <summary>
    /// Back to normal colour gradually
    /// </summary>
    public void FadeIn()
    {
        // we set the color of the target
        Color targetColor = new Color(1, 1, 1, 1);

        // call the DOTween method to change the color with specific time
        spriteRenderer.DOColor(targetColor, Settings.itemFadeDuration);

    }

    // to call this method after the situation was triggered
    /// <summary>
    /// Change to the translucent color
    /// </summary>
    public void FadeOut()
    {
        // we set the color of the target
        Color targetColor = new Color(1, 1, 1, Settings.targetAlpha);

        // call the DOTween method to change the color with specific time
        spriteRenderer.DOColor(targetColor, Settings.itemFadeDuration);
    }

}
```

Figure 28-Code Snippet of Item Fader

Explanation:
However, as shown in the left half of Figure 27, when the Player walked behind the Scenery Tree, he was obscured, which deprived the user of the Player's view, which was not what is expected.
Hence, a method had to be implemented to make the tree transparent. Specifically, the 'Alpha' value of the 'Colour' in the 'Sprite Renderer' in Figure 25 should be adjusted to a value that will reveal the Player, in other words, make the tree transparent.
And this part can be achieved by ‘Item Fade (Script)’, which has been added to the Inspector of Scenery Tree in Figure 25. To be more specific, as shown in Figure 28, after the 'Sprite Renderer' is obtained through ‘Get Component’, two Functions are created to change the 'Alpha' value of the 'Colour'. Alpha' value, where 'Fade Out' is used to change to the expected Alpha Value and 'Fade In' is used to revert to the normal Alpha Value. The process of changing the colours in the process of changing the colours is done quickly by the method provided by 'DOTween' package [16], which is the 'DOColor' method.
After this, when the Player overlaps with the Scenery Tree, which is the collision, the ‘Item Fader’ can be called via code mounted on the Player and the desired effect can be achieved, as shown in the right half of Figure 27.

2.7 Item Data Structure and Generation

Design:

Table 4-The Item Data Structure and Meaning

Variable Name	***Variable Type***	***Variable Meaning***
Item ID	int	The unique number of the item
Name	string	The name of the item
Item Type	ItemType	The type of the item
Item Icon	Sprite	The icon of the item
Item On World Sprite	Sprite	The picture of the item
Item Description	string	The description of the item
Item Use Radius	int	The use range of the item
Can Pick Up	bool	The item can be picked up or not
Can Dropped	bool	The item can be dropped or not
Can Carried	bool	The item can be carried or not
Item Price	int	The purchase price of the item
Sell Percentage	Float (0, 1)	The selling price of the item

Table 5-Item Type and Meaning

Item Type	***Item Meaning***
Seed	The seeds of fruits and vegetables
Commodity	The usual commodities
Furniture	The usual furniture
Hoe Tool	Hoe
Chop Tool	Axe
Break Tool	Pickaxe
Reap Tool	Reap
Water Tool	Water Can

Collect Tool	Basket
Reapable Scenery	Reapable Grass

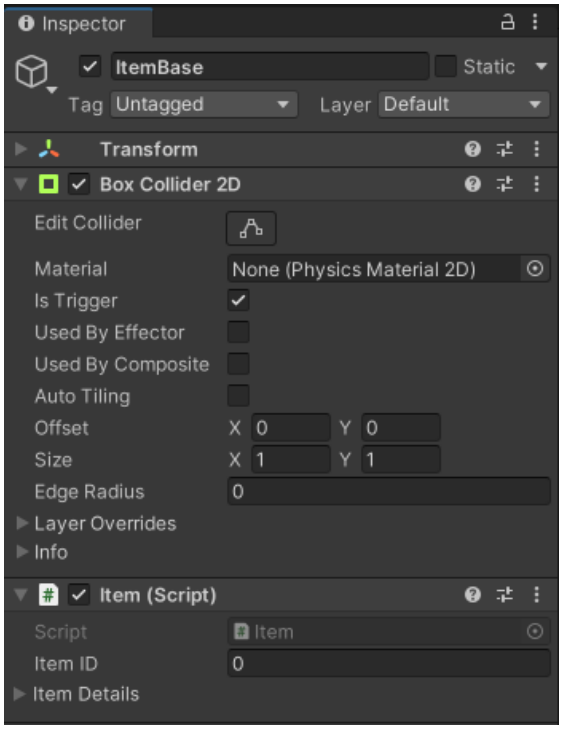


Figure 29-Item Base to Generate Different Items

Explanation:

Because of the game genre, items should be structured and generated at first. As shown in Table 4, it defines the main properties that will be used in this game. To be more specific, it includes identifiers and characteristics such as 'Item ID' and 'Name', as well as more specific attributes like 'Item Type', which can be referred to Table 5 for more details. Meanwhile, it also includes visual representations - 'Item Icon' and 'Item on World Sprite', usage details from 'Item Use Radius' to 'Can Carried', and economic aspects for this sim game to realize resource management – 'Item Price' and 'Sell Percentage'). To conclude, these attributes and types in Table 4 and Table 5 collectively describe how each item is presented, interacted with, and valued within the game, which provide the basis for animated switching, trending systems and stuff like that.

In addition to the creation of the Item data structure, it is also essential to generate the related item in the game world. To finish this, an Item Base is created, as shown in Figure 29, to await being called and then converted into the expected item, which can be realized by a Script.

Implementation:

```
public class ItemDetails
{
    // some specific properties of one item
    public int itemID;
    public string itemName;

    public ItemType itemType;

    public Sprite itemIcon; // to display possible icon for the game object/item
    public Sprite itemOnWorldSprite; // to display them in the world map

    public string itemDescription;
    public int itemUseRadius;

    // some interacting properties
    public bool canPickedup;
    public bool canDropped;
    public bool canCarried;

    // some business properties
    public int itemPrice;
    [Range(0,1)]
    public float sellPercentage;

}
```

Figure 30-The Class of Item Details

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

[CreateAssetMenu(fileName ="ItemDataList_SO", menuName = "Inventory/ItemDataList")] //Through this, we can create a list in Unity Directly
public class ItemDataList_SO : ScriptableObject
{
    public List<ItemDetails> itemDetailsList;
}
```

Figure 31-The Class of Item Data List

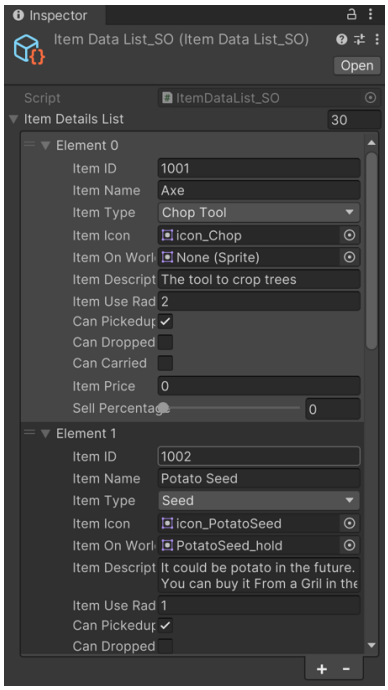


Figure 32-The Inspector of Item Data List

Explanation:

As displayed in Table 4, The class of 'Item Details' can be created, which is the data structure of a single Item. Then, since the data for all items is expected to be stored together, the List of 'Item Data', which is called the class of 'Item Data List', can be created to implement this idea. And because it inherits the properties of a 'Scriptable Object', which is a built-in class in Unity Editor, the 'Item Data List' can be created and treated as an Object in Unity that can be further processed or called. To be more specific, as shown in Figure 32, there are two elements stored into the 'Item Data List' created in Unity Editor, which are Axe and Potato

Seed, showing the standard of ‘Item Details’ Data. (The number of elements is more than two but cannot be showed in this small-size figure.)

Problems Encountered:

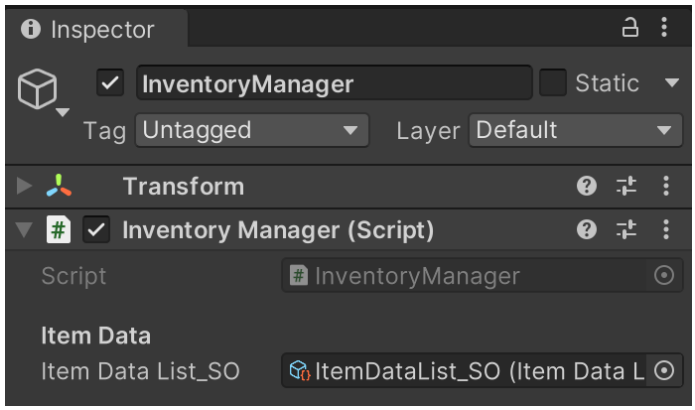


Figure 33-The Inspector of Inventory Manager

```
/// <summary>
/// Return the details information of the object through ID
/// </summary>
/// <param name="ID"></param>
/// <returns></returns>

public ItemDetails GetItemDetails(int ID)
{
    return itemDataList_SO.itemDetailsList.Find(i => i.itemID == ID);
}
```

Figure 34-Code Snippet of Getting Item Details through Item ID in Inventory Manager

```
public void Init(int ID)
{
    itemID = ID;

    // Get the value from Inventory
    itemDetails = InventoryManager.Instance.GetItemDetails(itemID);

    if (itemDetails != null)
    {
        spriteRenderer.sprite = itemDetails.itemOnWorldSprite != null ? itemDetails.itemOnWorldSprite : itemDetails.itemIcon;

        // change the size of the collider with the size of the original object
        Vector2 newSize = new Vector2(spriteRenderer.sprite.bounds.size.x, spriteRenderer.sprite.bounds.size.y);
        coll.size = newSize;
        // Because of pivot problem, we need to add an offset
        coll.offset = new Vector2(0, spriteRenderer.sprite.bounds.center.y);

    }
}
```

Figure 35-Code Snippet of Initialising an Item in Item (Script)

Explanation:

Although it was decided in the Design Part to use code to generate items based on the 'Item Base', one of the most important points was forgotten at the beginning of the design process, and that is that the 'Item Data List ' was not loaded into the game, so there was no way to get the data even with 'Item (Script)'.

However, this problem was solved by a tutorial where the 'Inventory Manager' could be created in a scene (does not disappear) where the Player appears, and the 'Item (Script)' could then be loaded into the game with the 'Item (Script)'. Data List' to remain active for the duration of the game, and through the approach in Figure 34, each Item can be queried by 'Item ID'. Thus, in the 'Item (Script)' that the 'Item Base' carries, 'Init' can be called, which gets the Item by ID It gets all the information about the Item by its ID, and then assigns the value of 'Sprite' (any one of them) to the 'Item Base', and finally, the whole process succeeds in generating expected Items.

2.8 Basic Logic of Picking Up Item

Design:

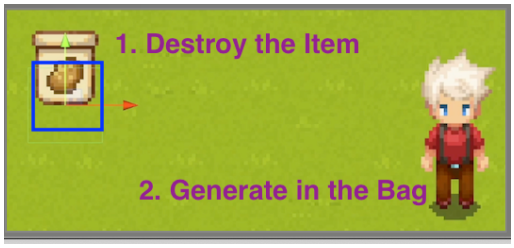


Figure 36-The Logic of Picking Up Item

Explanation:

To focus more on the Bag Logic, the basic logic of picking up Item should be designed first. Because, as the Item can be created based on section 2.7 in the game world, the most possible interaction with it is picking up.

Therefore, there is a basic idea, which is shown in Figure 36. To be more specific, since 'OnTriggerEnter2D' method in the Unity Editor can detect the collision between the Player the Item when the Player enters the range of the blue rectangle [17], this method can be used to destroy the item and pass the details of the item to the related code to generate the item in the Bag, through which a logic about items being picked up is complete.

Implementation:

```
public class ItemPickup : MonoBehaviour
{
    // OnTriggerEnter2D is called when another object enters a trigger collider attached to this object (2D physics only).
    private void OnTriggerEnter2D(Collider2D other)
    {
        // Attempt to retrieve the Item component from the object that entered the trigger.
        Item item = other.GetComponent<Item>();

        // Check if the object that entered the trigger has an Item component.
        if (item != null)
        {
            // Check if the item can be picked up, based on its itemDetails property.
            if (item.itemDetails.canPickedup)
            {
                // If the item can be picked up, add it to the inventory.
                // InventoryManager is a singleton that manages the inventory items.
                InventoryManager.Instance.AddItem(item, true); // The 'true' means the Item should be destoryed

                // Call an event to play a sound effect for item pickup.
                // EventHandler is a class that handles events and might be using the observer pattern.
                EventHandler.CallPlaySoundEvent(SoundName.Pickup);
            }
        }
    }

}
```

Figure 37-Code Snippet of Item Pick Up

```
/// <summary>
/// Add the item to the player's bag (destory the item in the scenery)
/// </summary>
/// <param name="item"></param>
/// <param name="toDestory"></param>
public void AddItem(Item item, bool toDestory)
{
    // if there are the existing objects in the bag
    var index = GetItemIndexInBag(item.itemID);
    AddItemAtIndex(item.itemID, index, 1);

    // Debug.Log(GetItemDetails(item.itemID) + "Name: " + GetItemDetails(item.itemID).itemName);
    if (toDestory)
    {
        Destroy(item.gameObject);
    }

    // TODO: Update UI
    EventHandler.CallUpdateInventoryUI(InventoryLocation.Player, playerBag.itemList);
}
```

Figure 38-Code Snippet of Add Item Called in Item Pick Up

Explanation:
Since the dissertation is written after all the projects are completed, the complete logical code can be displayed. As mentioned in the Design Part, in the code of Figure 37, adding the 'Item Pick Up' script to the Player allows the 'OnTriggerEnter2D' to monitor collisions between the Player and items. Then, it performs the expected logic—removing the item and adding it to the bag. However, in this case, a smart method called 'Add Item' in Figure 38 has been created in the 'Inventory Manager' script, which integrates the logic of removing the item from the scene and adding the item to the bag.

Problems Encountered:

```
Debug.Log(GetItemDetails(item.itemID).itemID + " Name: " + GetItemDetails(item.itemID).itemName);
```

Figure 39-Cope Snippet of Debug Function

Explanation:
The completion of the entire logic code is not complicated. However, there was an issue initially overlooked—whether the details of the item being removed were correct. Therefore, 'Debug. Log' was used here to verify if the data is accurate. Finally, it turned out to be correct. The relevant code is as shown in Figure 39.

2.9 Bag Logic

Design:

Table 6-Data Structure of Bag Item

Variable Name	***Variable Type***	***Variable Meaning***
Item ID	int	The ID of the item
Item Amount	int	The amount of the item

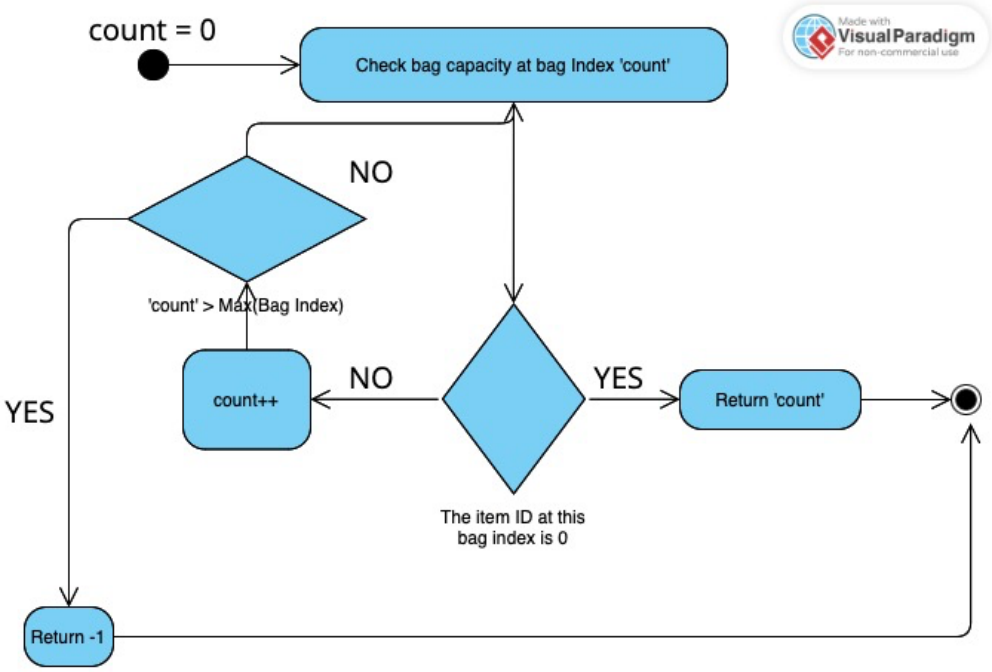


Figure 40-The Logic of Checking Bag Capacity

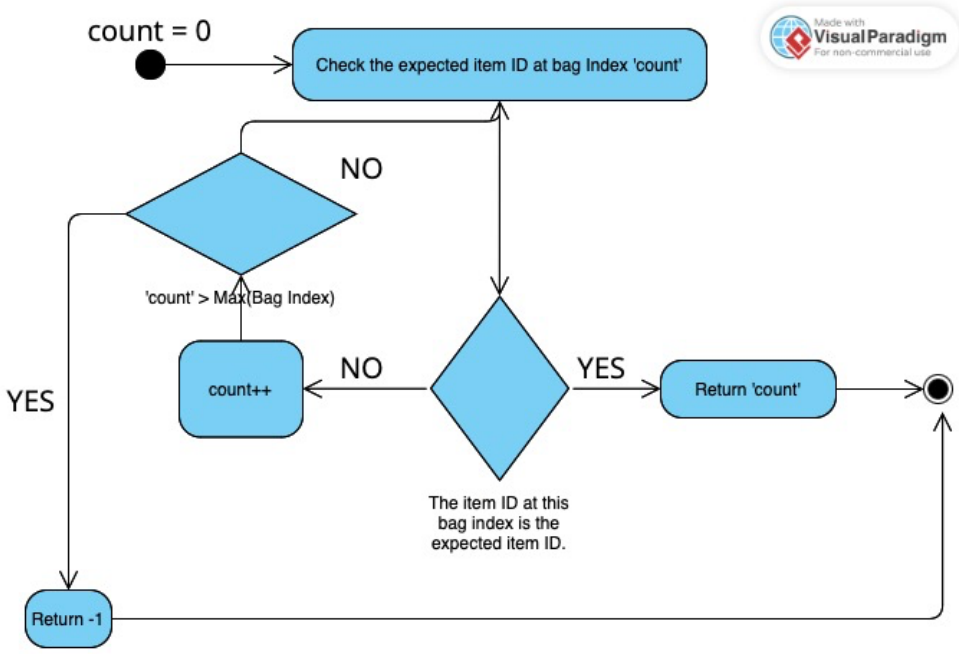


Figure 41-The Logic of Getting Item Index in Bag

Explanation:

As shown in Table 6, 'Item ID' and 'Item Amount' are the only two elements that will be stored into the item of bag. This is because, 'Item ID' can be used to refer to the item in 'Item Data List' and then generate the expected result without the need to store all the details of each item.

Additionally, before adding the items to the bag, two things that need to be checked first are:

- Whether there is enough space for a new item.
- Whether the same item already exists here.

As shown in Figure 40 and Figure 41, they have the similar logic to check the same item position or check the empty position:

1. Initialize Counter (Start with count = 0): It will be used to iterate through the bag indices to record the empty position or the position of the same item.
2. Check Bag Index (Check Bag at Index 'count'): Check the current bag whether it is expected – empty position or position of the same item.
3. ('YES', in Decision Tree) If an expected result is got in step 2, then return the 'count' and end the check; ('NO', in Decision Tree) Otherwise, it will step further into step 4.
4. Increment Counter (count++): If the same item is not found or the slot is not empty, increase the 'count' to move to the next index.
5. ('YES', in Decision Tree) If the 'count' is not larger than the max number of bag index, then loop back to step 2 with the updated 'count' value; ('NO', in Decision Tree) Otherwise, it will directly return '-1' and end the check.

Moreover, the position of items in the bag can change between each other, the logic is as follows:

1. Remember the index of the item that is waiting to change, and the index of the item that is going to be changed.
2. If the Item ID in the target slot is 0, then the item can be assigned to the target slot directly; otherwise, the data should be changed between each other.

Implementation:

```
// if we use the class instead the structure, the value should not be null, which should be tested before continuing the statements of code
[System.Serializable]
public struct InventoryItem
{
    public int itemID;
    public int itemAmount;
}
```

Figure 42-Code Snippet of Inventory Item

```
/// <summary>
/// Find the ID of the existing item in the bag
/// </summary>
/// <param name="ID">ID of the item</param>
/// <returns>-1 if there is not existing item in the bag</returns>
private int GetItemIndexInBag(int ID)
{
    for (int i = 0; i < playerBag.itemList.Count; i++)
    {
        if (playerBag.itemList[i].itemID == ID)
            return i;
    }
    return -1;
}
```

Figure 43-Code Snippet of Getting Item Index in Bag

```
/// <summary>
/// At the certain space, we add the item
/// </summary>
/// <param name="ID">ID of the object</param>
/// <param name="index"> sequence of the item in the bag </param>
/// <param name="amount"> the number of the item picked up </param>
private void AddItemAtIndex(int ID, int index, int amount)
{
    if(index == -1 && CheckBagCapacity()) // The bag does not contain the same item before but has free space
    {
        var item = new InventoryItem { itemID = ID, itemAmount = amount };

        for (int i = 0; i < playerBag.itemList.Count; i++)
        {
            if (playerBag.itemList[i].itemID == 0)
            {
                playerBag.itemList[i] = item;
                break;
            }
        }
    }
    else // The bag contains the same item before
    {
        int currentAmount = playerBag.itemList[index].itemAmount + amount;
        var item = new InventoryItem { itemID = ID, itemAmount = currentAmount };

        playerBag.itemList[index] = item;
    }
}
```

Figure 44-Code Snippet of Adding Item at Index

```
/// <summary>
/// Change two things in bags through dragging
/// </summary>
/// <param name="fromIndex"> the original index of the item we are dragging from </param>
/// <param name="targetIndex"> the target index we are dragging to </param>
public void SwapItem(int fromIndex, int targetIndex)
{
    InventoryItem currentItem = playerBag.itemList[fromIndex];
    InventoryItem targetItem = playerBag.itemList[targetIndex];

    if (targetItem.itemID != 0) // if there is already item in the targetItem bag, we should follow the logic to change two things
    {
        playerBag.itemList[fromIndex] = targetItem;
        playerBag.itemList[targetIndex] = currentItem;
    }
    else // if not
    {
        playerBag.itemList[targetIndex] = currentItem;
        playerBag.itemList[fromIndex] = new InventoryItem();
    }

    EventHandler.CallUpdateInventoryUI(InventoryLocation.Player, playerBag.itemList);
}
```

Figure 45-Code Snippet of In-Bag Changing

Explanation:

As shown in Figure 42, a struct of ‘Inventory Item’ is created for the item data structure in bag, which is not a class. The reason for this will be explained in Problems Part below. And

then like 2.7 section, an Inventory Item Data List is created, which is called ‘Inventory Bag’, to store the items. After assigning it to the ‘Inventory Manager’, more steps can be carried out to handle data.
Moreover, Figure 43 illustrates the code implementation of the activity diagram from Figure 41. However, as discussed in the Design Part, Figure 40 shares a similar logic and thus is not displayed here. Specifically, despite the complexity of the discussion, the final code can be effectively resolved with a simple for loop.
Meanwhile, Figure 44 shows the logic of how to add an item: if there is no identical item in the bag (index = -1) but there is an empty slot, the new item can be stored in that slot. If there is an identical item in the bag, then according to the data structure in Table 6, it is sufficient to just increment the Amount by 1, and then reassign the new Amount of the identical Item back to its original position.
Finally, Figure 45 shows the code for exchanging items in the bag.

Problems Encountered:

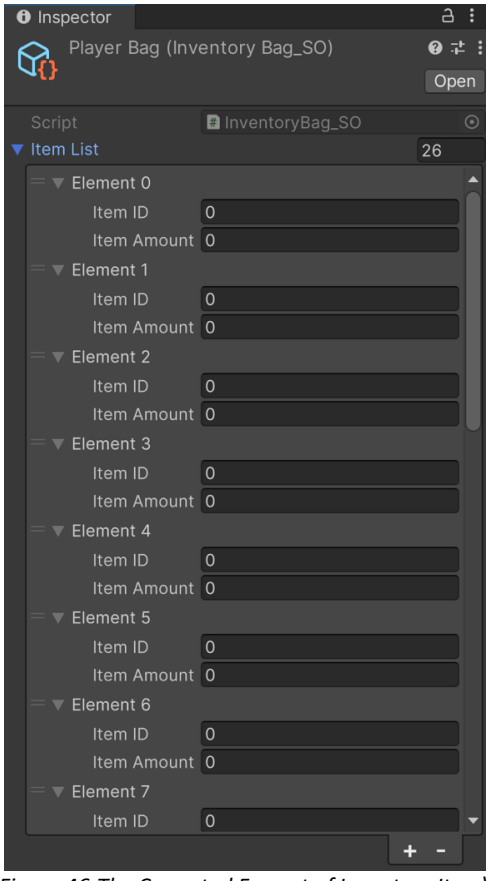


Figure 46-The Corrected Format of Inventory Item

Explanation:
The data structure for Inventory Item was based on ‘Class’ at first, following the method in Section 2.7. However, although the logic is correct, it brought two issues:

1. For ‘Class’, objects need to be instantiated before being assigned values, which increases the complexity of the code that handles the data.
2. Additionally, unlike the data that needs to be defined in advance in the data of Section 2.7, the data in this part for the Bag is guaranteed to exist.

Thus, in this section, ‘Struct’ is used, instead of ‘Class’, to initialize the bag data in each slot by default, as shown in Figure 46.

2.10 Bag UI

Design:

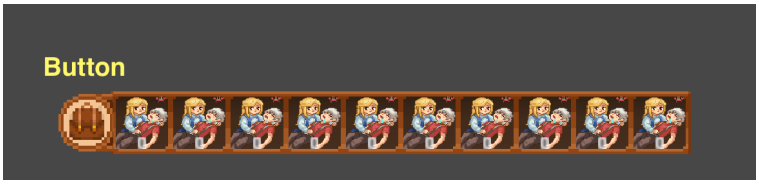


Figure 47-Screenshot of Action Bar UI (Smaller Bag)

Figure 48-Screenshot of Bag UI (Bigger Bag)

Figure 49-Screenshot of Item/Tool Tip UI

Figure 50-Screenshot of Slot UI

Explanation:

For each figure:

1. Figure 50 shows what a Bag Slot looks like: it consists of an icon and a number at the bottom to show the icon of the item in the slot and its amount, respectively.
2. Figure 47 shows the UI for the Action Bar, which is part of the bag: it consists of a button and ten slots that give quick feedback to the player on the use of the item.
3. Figure 48 shows the hidden backpack: it consists of a money symbol and 16 slots. It allows the player to store more of what they want.

4. Figure 48 is the UI for the Item or Tool that displays information: it reads the name, type, description, and selling price of the item to provide the player with enough information to be useful.

Meanwhile, a referred approach for the implementation of item drags and drop swapping is that it can be done by inheriting Unity built-in 'IPointerClickHandler, IBeginDragHandler, IDragHandler, IEndDragHandler' code [18], which will be shown below.

Implementation:

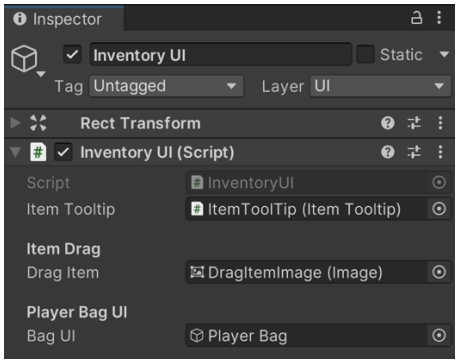


Figure 51-The Inspector of Inventory UI

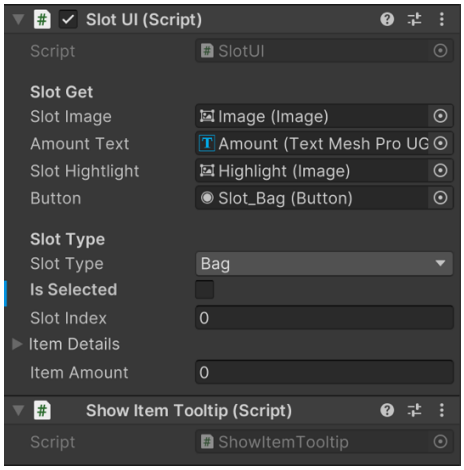


Figure 52-The Inspector of Bag Slot

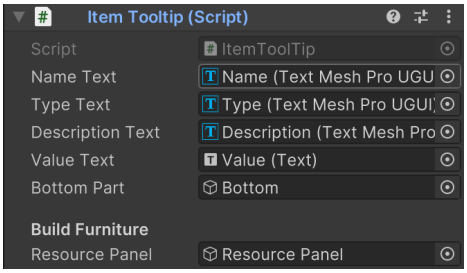


Figure 53-The Inspector of Item/Tool Tip

```
public void OnBeginDrag(PointerEventData eventData)
{
    if(itemAmount != 0) //if ther is an item
    {
        inventoryUI.dragItem.enabled = true; // Enable the item
        inventoryUI.dragItem.sprite = slotImage.sprite; // Assign an sprite according to the slot Image
        inventoryUI.dragItem.SetNativeSize(); // Set Native Size to avoid size problems

        isSelected = true;
        inventoryUI.UpdateSlotHightlight(slotIndex); // High Light the UI
    }
}

public void OnDrag(PointerEventData eventData)
{
    inventoryUI.dragItem.transform.position = Input.mousePosition;
}

public void OnEndDrag(PointerEventData eventData)
{
    inventoryUI.dragItem.enabled = false;
    //Debug.Log(eventData.pointerCurrentRaycast);

    // discard anything that is not related to UI
    if (eventData.pointerCurrentRaycast.gameObject != null)
    {
        if (eventData.pointerCurrentRaycast.gameObject.GetComponent<SlotUI>() == null)
            return;

        // if the UI ray cast is not null, then we should continue
        var targetSlot = eventData.pointerCurrentRaycast.gameObject.GetComponent<SlotUI>();
        int targetIndex = targetSlot.slotIndex;

        // if both are bags type, it mean it's just change between different bags
        if (slotType == SlotType.Bag && targetSlot.slotType == SlotType.Bag)
        {
            InventoryManager.Instance.SwapItem(slotIndex, targetIndex);

        }
    }
}
```

Figure 54-Code Snippet for Item Drag and Drop

Explanation:

For UI creation, the ‘Panel’, ‘Image’, and ‘Button’ that come with Unity can be used. As well as some 'Layout Group' components to eliminate the sorting step. For example, multiple Slot UIs added to a Bag UI can be automatically sorted using a 'Layout Group' as shown in Figure 47 and Figure 48.

Meanwhile, to do data manipulation on the Slot and Item/Tool Tip, such as changing the number at the bottom of the Slot to match the amount of the item, the components for both must be available in Scripts. Hence, as shown in Figures 52 and 53, their data structures and the data needed have been listed. In addition to this, it should be noted that because subsequent item exchanges will exist between Boxes and Bags, ‘Slot Type’ is used to monitor where items are stored shown in the middle part of Figure 52.

Then, in a similar logic to the creation of the 'Inventory Manager' in section 2.7, the 'Inventory UI' in Figure 51 has been created to manage the data for both above and more Inventory related UI later.

Finally, Figure 54 shows a code implementation of a classic item drag and drop. Specifically, after getting the 'Panel' of the drag page and the 'Sprite' of the item being dragged, Unity built-in code detects the mouse ray to get the target data - that is, the 'Slot UI'. The data in the 'Slot UI' can then be used for further processing, for example, its number can be used to determine if this slot is empty, and if it is not, assign its sprite to the preset 'Sprite' of the dragged item. After that, the drag and drop item can then be displayed. Finally, in the similar way, the 'Slot UI' data of the target location can be fetched based on the mouse ray cast, so that it is possible to do either item swapping or adding the amount to the same item.

Problems Encountered:

```
public InventoryLocation Location
{
    get
    {
        return slotType switch
        {
            SlotType.Bag => InventoryLocation.Player,
            SlotType.Box => InventoryLocation.Box,
            _ => InventoryLocation.Player
        };
    }
}
```

Figure 55-Code Snippet of Determining Inventory Location of Item According to Slot Type

```
// know the location and the data of the item
public static event Action<InventoryLocation, List<InventoryItem>> UpdateInventoryUI;
// call the method
public static void CallUpdateInventoryUI(InventoryLocation location, List<InventoryItem> list)
{
    UpdateInventoryUI?.Invoke(location, list);
}
```

Figure 56-Code Snippet of Event of Update Inventory UI

```
// TODO: Update UI
EventHandler.CallUpdateInventoryUI(InventoryLocation.Player, playerBag.itemList);
```

Figure 57-Code Snippet of Calling the Event to Update Inventory UI in Inventory Manager

Explanation:
However, despite having Scripts to manage the data, there is no way for the data from the Bag to be read directly, as the 'Inventory UI' and the 'Inventory Manager' exist in different scenes. Even if the 'Inventory Manager' could be called as an object to get the 'player Bag' in it, this would be difficult to keep the data up to date. Specifically, it is difficult to ensure that data is changed sequentially in the execution of scripts when the data is not only changed in the data itself, but also in the UI. Another possible result is when the data in the 'Inventory Manager' has already been updated by another method, and the UI in the 'Inventory UI', because of being called as an object in the former, is still showing the data as it was before the change. However, real-time updating methods can be created, for example, by keeping the data transferred to each other in every Script that needs to be updated, but this can significantly strain the computer computation.
In this case, the use of the Event method as shown in Figure 56 solves this problem. By creating an Event and executing it with a method – 'Call Event', all Scripts registered for this event can use the data in the 'Call Event' to synchronise their updates. Specifically, in Figure 56, 'Inventory Location' and 'List<Inventory Item>' are used to transfer the data to 'Update Inventory UI'. for UI update, where the definition of 'Inventory Location' can be referred to in Figure 55. Subsequently, 'Call Update Inventory UI' can be used in the 'Inventory Manager' as shown in Figure 57, which is where the 'player Bag' data is managed in the setting, is called to transfer the 'player Bag ' and pass in 'Inventory Location' as Player.

2.11 Movement Animations and Interactive Animations with Items/Tools

Design:

Table 7-Data Structure of Animator Type

Variable Name	***Variable Type***	***Variable Meaning***
Part Type	Part Type	None, Carry, Hoe, Break, Water, Collect, Chop, Reap
Part Name	Part Name	Body, Hair, Arm, Tool

Override Controller	Animator Override Controller	Animator based on a base one

Table 8-The Data Structure of Animator Override

Variable Name	***Variable Type***	***Variable Meaning***
Animators	Animator []	All the animators on Parts of Player
Hold Item	Sprite Renderer	Special case to handle Part Type - Carry
Animator Types	List<Animator Type>	All the expected animators of Structure shown in Table 7
Animator Name Dict	Dictionary<string, Animator>	All the animators on Parts of Player but stored in the Dictionary with the key – Part Name.

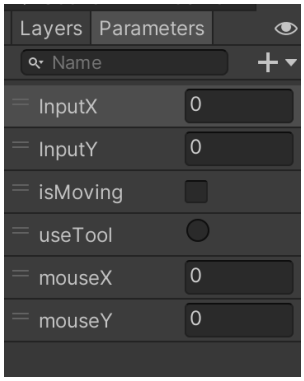


Figure 58-The Parameters of Animations

Explanation:

As shown in Table 7, a structure for storing animations corresponding to character parts is created. Firstly, 'Part Type' is used to record different kinds of animations such as 'Carry' and other tool animations. Secondly, 'Part Name' is used to record body parts for different animations. Finally, since even with different animations, their content is highly repetitive, 'Animator Override Controller' is applied here, which inherits a base walking animation.

At the same time, a script for collaboration must be created, which has the structure shown in Table 8. The 'Animators' are used to read the animators of the Player's parts, such as hair and body, which are later recorded in the 'Animator Name Dict'. Secondly, 'Hold Item' is created here because it needs to read the item's sprite to show what the item is. Then, 'Animator Types' is what Table 7 shows and describes, it reads and records all the parts of the expected animation. Finally, 'Animator Type' finds all the animators in the 'Animator Name Dict' based on its own 'Part Name' parameter', and then make changes to the animation.

Finally, the six parameters that control the animations as in Figure 58 need to be created. The 'Input X' and 'Input Y' are used to record the speed and direction the Player is travelling, which are 'Float' values. Then 'is Moving' and 'use Tool' are used to determine if the Player is moving and using a tool, which are 'Boolean ' value. Finally, 'mouse X' and 'mouse Y' are used to

monitor the position of the mouse when the animation is triggered and change the direction the Player is facing, which are 'Float values.

Implementation:

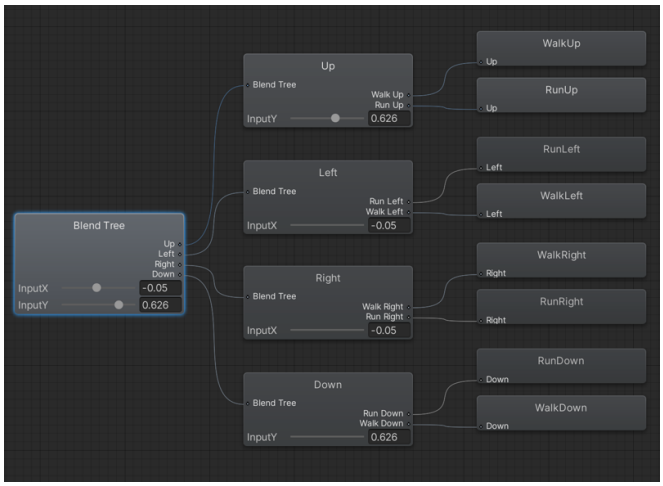


Figure 59-The Blend Tree of Animations

```
private void SwitchAnimation()
{
    foreach (var anim in animators) // loop all the animators of Player parts (like Hair and Body)
    {
        anim.SetBool("isMoving", isMoving); // Set Animation parameters to trigger animations
        anim.SetFloat("mouseX", mouseX);
        anim.SetFloat("mouseY", mouseY);

        if (isMoving)
        {
            anim.SetFloat("InputX", inputX);
            anim.SetFloat("InputY", inputY);
        }
    }
}
```

Figure 60-Code Snippet to Control Animation Parameters

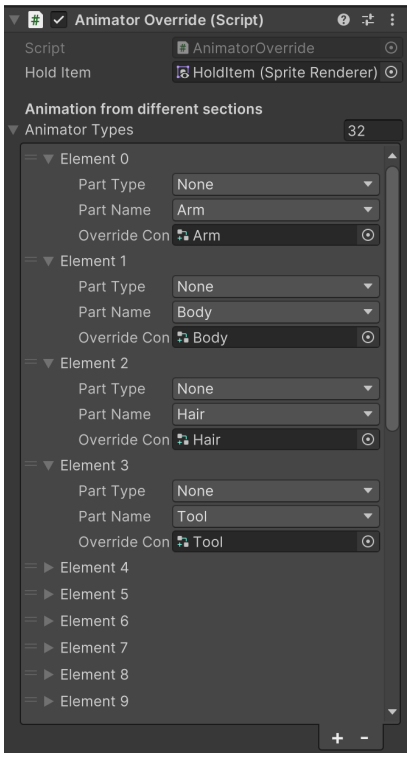


Figure 61-The Inspector of Player with Animator Override (Script)

```
private void SwitchAnimator(PartType partType)
{
    // For each item in animator Types
    foreach (var item in animatorTypes)
    {
        // If the part Type of this item is equal to the expected part Type
        if (item.partType == partType)
        {
            // Search and Run the animatior (animations) with the same name (for the expected part of Player, like Hair, Body and ...)
            animatorNameDict[item.partName.ToString()].runtimeAnimatorController = item.overrideController;
        }
    }
}
```

Figure 62-Code Snippet of Switching Animator according to Part Type

Explanation:
The Blend Tree shown in Figure 59 is a built-in animation management feature of Unity [19]. In other words, the parameters in Figure 58 can be added to the Blend Tree to set the preconditions for animator switching or to set the corresponding parameters, which are managed by the main code snippets in Figure 60 and Figure 62.
Thus, when Player movement is required, 'Set Bool' or 'Set Float' as shown in Figure 60 can be used to activate the parameters set in the corresponding Blend Tree, which activates the animation.
At the same time, since the animation of the Player using the tool needs to be toggled on and off for each part of the animation, it is necessary to store all the animation implementations using the tool in Figure 61 and then control them with the code in Figure 62. The logic has already been elaborated in Design Part, and the code is not complicated to implement.

Problems Encountered:

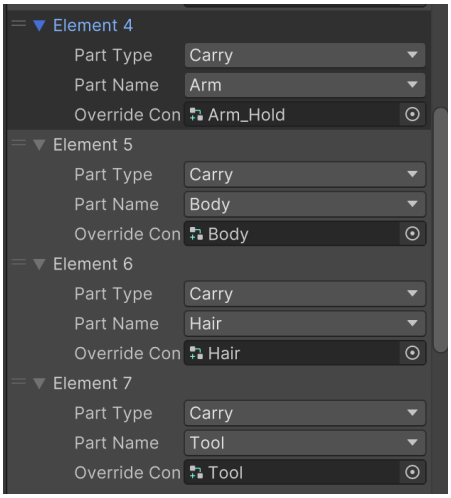


Figure 63-The Animator Types of Carry

Explanation:
Some of the animations using the tool change the animator of all the parts, while others only change the animator of a single part, so not animating all the parts by default can lead to a problem of chaotic animation playback. For example, Figure 63 shows the animation when the Part Type is Carry, and the difference between this and the default animation in Figure 61 is only in the 'Arm' animation. Therefore, in this case, if only the data related to the change of 'Arm' is stored at the beginning, and not the animation of the default case, when some tools only change the animation of 'Body', it is possible that the 'Arm' animation will be changed to the 'Body' animation. 's animation will still be executed, because of which it has not been changed back to its original state. Hence, each 'Part Name' associated with a different 'Part Type', even if it has the same animation, should be logged, so that this problem can be avoided, and the design burden can be reduced.

2.12 Grid Map Information System and Instant Grid Actions

Design:

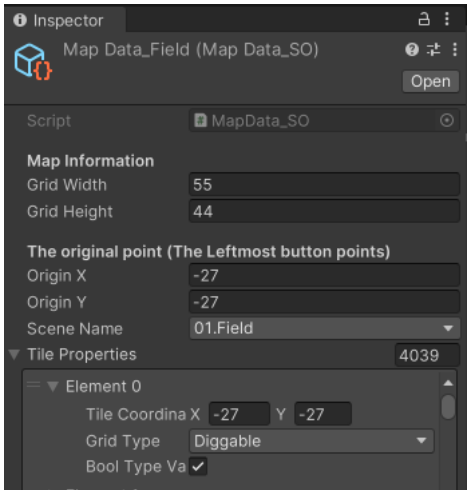


Figure 64-The Inspector of Map Data of Field (Farm)

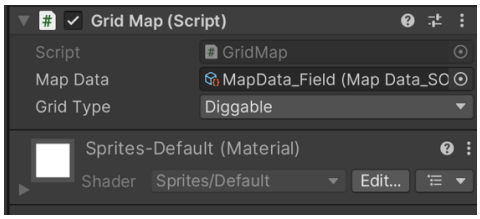


Figure 65-The Variables Needed in Grid Map (Script)

Table 9-The Data Structure of Tile Details

Variable Name	***Variable Type***	***Variable Meaning***
Grid X	int	The X value of Grid Position in Coordinate
Grid Y	int	The Y of value Grid Position in Coordinate
Can Dig	bool	Can dig in this field
Can Drop Item	bool	Can drop item in this field
Can Place Furniture	bool	Can place furniture in this field
Is NPC Obstacle	bool	This field is obstacle for NPCs

Explanation:

First, in Unity, there is a 'Grid Information' script, which informs about each grid and allows other code to read the data related to the position of the grid in local coordinate. Therefore, based on this script, the grid properties mentioned in section 2.4 can be implemented.

Next, Figure 65 shows the 'Grid Map' script, where the type of each grid is labelled inside the map layers of the 4 grid properties in section 2.4, where the types include 'Diggable ', 'Drop Item', 'Place Furniture' and 'NPC Obstacle' which correspond to the four Boolean values in Table 9 respectively. Specifically, the grid that can be diggable can be marked as Diggable, and then the data is stored in Map Data in Figure 64.

Meanwhile, the 'Grid Information' script that comes with Unity only provides the local coordinate position (i.e., the position of the area of all the tagged grids), not the world

coordinate position (i.e., the position of the area of all the tagged grids. Hence, to calculate the position of the world coordinate for each Grid, the length and width of the entire Scene map, as well as the length and width of all the Grids brought together, need to be recorded, as shown in Figure 64. (Since the default Grid size in Unity is 1x1, the size of individual grids does not need to be recorded.)

However, each tag only exists inside the respective layer, for instance, a grid is marked as 'Diggable', but it may also be marked as 'Drop Item'. However, they are all properties of the same grid. To deal with this situation, Table 9 provides a 'Tile Details' data structure, which is used to summaries all the tagged attributes of a grid in a final count.

Finally, based on the knowledge of grid position and grid properties, digging, watering, and NPC routes based on the A* pathfinding algorithm can be implemented.

Implementation:

```
public MapData_SO mapData;
public GridType gridType;
private Tilemap currentTilemap;

private void OnEnable()
{
    if (!Application.IsPlaying(this)) // Only Execute when the application is not in playing
    {
        currentTilemap = GetComponent<Tilemap>(); // Get the informaiton of current tile map

        if (mapData != null) // if the mapData is not null, we should clear it first
            mapData.tileProperties.Clear();
    }
}

private void OnDisable()
{
    if (!Application.IsPlaying(this))// Only Execute when the application is not in playing
    {
        currentTilemap = GetComponent<Tilemap>(); // Get the information of current tile map

        UpdateTileProperties();
f UNITY_EDITOR // execute this code only when it is in Editor mode
        if (mapData != null)
            EditorUtility.SetDirty(mapData); // We save what we drew into the map Data
ndif
    }
}
```

Figure 66-Code Snippet of Grid Map (Script)

```
switch (tileProperty.gridType)
{
    case GridType.Diggable:
        tileDetails.canDig = tileProperty.boolTypeValue;
        break;
    case GridType.DropItem:
        tileDetails.canDropItem = tileProperty.boolTypeValue;
        break;
    case GridType.PlaceFurniture:
        tileDetails.canPlaceFurniture = tileProperty.boolTypeValue;
        break;
    case GridType.NPCObstacle:
        tileDetails.isNPCObstacle = tileProperty.boolTypeValue;
        break;
}
```

Figure 67-Code Snippet of Summarizing Grid Properties according to Grid Type

```
/// <summary>
/// Set dig tile
/// </summary>
/// <param name="tile"></param>
private void SetDigGround(TileDetails tile)
{
    Vector3Int pos = new Vector3Int(tile.gridX, tile.gridY, 0);
    if (digTilemap != null)
        digTilemap.SetTile(pos, digTile);
}
/// <summary>
/// Set water tile
/// </summary>
/// <param name="tile"></param>
private void SetWaterGround(TileDetails tile)
{
    Vector3Int pos = new Vector3Int(tile.gridX, tile.gridY, 0);
    if (waterTilemap != null)
        waterTilemap.SetTile(pos, waterTile);
}
```

Figure 68-Code Snippet of Digging and Watering

Explanation:
Figure 66 shows how a grid can be labelled and converted into data that can be stored in 'Map Data'. Firstly, the 'Map Data' needs to be read here. Secondly, the type of the grid to be labelled needs to be known. For example, whether it is used to mark 'Diggable' or 'Can Drop Item'. Finally, when this marking behavior is end, which is in the 'On Disable' method, the 'Update Tile Properties' method needs to be called, which will help to calculate for every grid element's world coordinate, collating the gird type and adding a 'Boolean' value to prove that it is indeed this grid type.
Then, in Figure 67, the properties on each grid are checked over and the new data in Table 9 is set with 'Boolean' values based on the added 'Boolean' values. In the end, this new data, which is 'Tile Details' is then stored in the new variable, waiting to be called.
Meanwhile, the Digging, Watering and A* pathfinding algorithm can be applied to the 'Grid Map Information System'. In this case, by modelling on the method of building 'Rule Tiles' and plotting them on the map shown in Section 2.4, a similar approach for Digging and Watering is shown in Figure 68: using two 'Rule Tiles' generated in advance, which are 'Rule Tiles', and which are 'Rule Tiles', and which are 'Rule Tiles', and which are 'Rule Tiles'. ', which are 'dig Tile' and 'water Tile' and generating expected tiles in the resulting two map layers, which are 'dig Tile Map' and 'water Tile Map'.
Whereas the A* pathfinding algorithm is a referred approach from YouTube Tutorial [20] and thus is not done in this project, which is placed in the Appendix.

Problems Encountered:

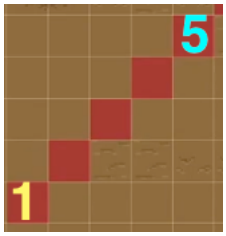


Figure 69-Screenshot of A Algorithm Test*

Explanation:
Despite the integration of the A* pathfinding algorithm, there was no NPC at this stage, so there was no way to test whether it would work successfully on NPCs as well. However, Figure 69 illustrates the results of the A* algorithm from grid 1 to grid 5, which is expected to be used as expected in the NPC section.

2.13 Time System and Light System

Design:

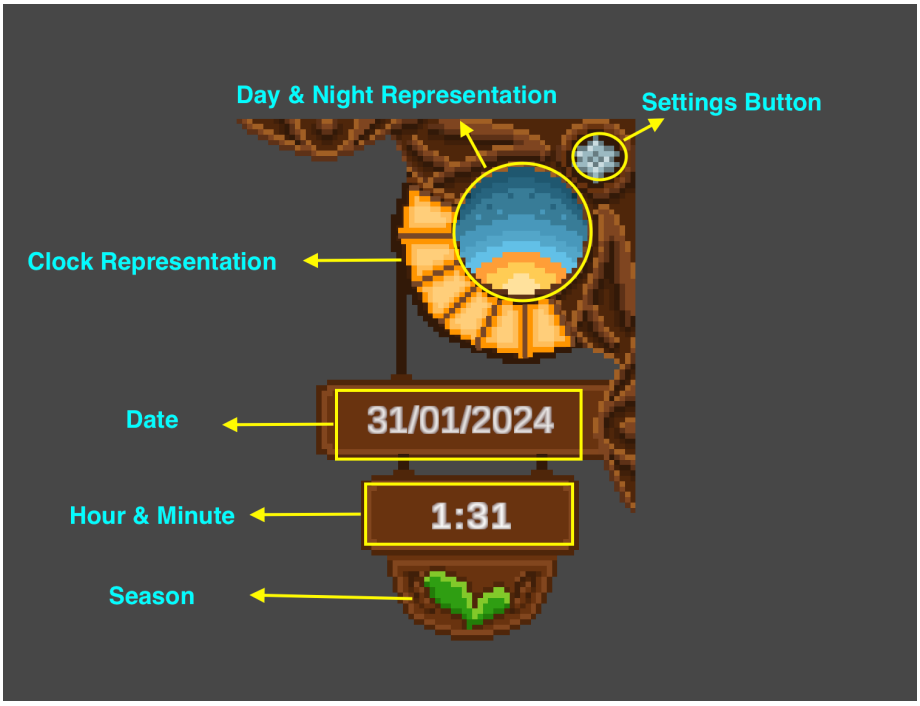


Figure 70-Screenshot of Time UI

Table 10-Baisc Time Settings

Row Number	***Variable Name***	***Variable Type/Value***	***Variable Meaning***
1	Second(s) Threshold	Float: 0.01f	To control the speed of game time. (The smaller the value, the faster the game time)
2	Second Hold	Int: 59	60 seconds in one Minute
3	Minute Hold	Int: 59	60 minutes in one Hour
4	Hour Hold	Int: 23	24 hours in one Day
5	Day Hold	Int: 30	30 days in one Month
6	Month Hold	Int: 12	12 months in one Year
7	Season Hold	Int: 3	3 months in one Season
8	Season	Enum Season: Spring, Summer, Autumn, Winter	4 seasons in one Year

9	Light Shift	Enum Light Shift: Morning, Night	Morning and night in one Day
10	Morning Time	Time Span (Hour, Minute, Second): (5, 0, 0)	The time to get light
11	Nighttime	Time Span (Hour, Minute, Second): (19, 0, 0)	The time to get dark

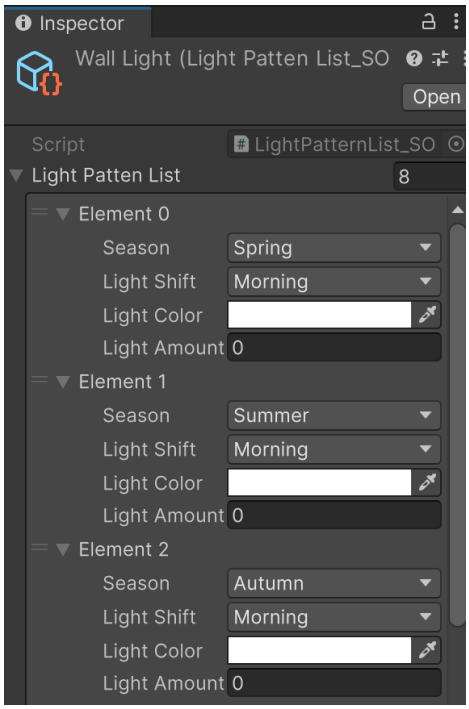


Figure 71-An Example of Light Data List

Explanation:

As designed in Figure 70, this is the 'Game Time UI' in this game, which can be used to tell the player the time within the game. It consists of six components from top to bottom:

1. A ‘Settings Button’, which is used to pause and exit the game, but will be designed in the final section.
2. A circular 'Day & Night Representation' section to show changes in the morning, afternoon, and evening.
3. A 'Date' is used to show the day, month, and year.
4. An 'Hour & Minute' is used to display the current time.
5. A 'Season' icon to show the seasons of the year.

Meanwhile, Table 10 shows all the time-related data and settings in the game. The first of these data is used to calculate the past time in the game, which is the timer in the game. While the second to eighth rows of the variable are seconds, minutes, and other time conversion standards used in real life, which are continued to be used by this game, in line with the player's intuition. Finally, the data from the ninth to eleventh rows are for the day, day, and night in the game.

In addition, for the alternation of day and night, 'Light' is required to be created in the game scenario. It is a component that comes with Unity and can be created and used very quickly

like Panel and Image for UI design. However, some pre-conditions need to be designed, such as the triggering season, time, to meet the actual transition of day/night. Thus, 'Light Data' as shown in Figure 71 was created, which consists of 'Season', 'Light Shift' and the light's Color and Intensity and is stored inside a List of the same type of data like the 'Item Data List' in section 2.7.

Implementation:

```
if (!gameClockPause)
{
    tikTime += Time.deltaTime; // The Time to change frame

    if (tikTime >= Settings.secondThreshold) // If the accumulated time is more than the threshold, we should
    {                                        // clean the accumulated time and call the method to Update Game Time
        tikTime -= Settings.secondThreshold;
        UpdateGameTime();
    }
}
```

Figure 72-Code Snippet of Game Timer according to Game FPS

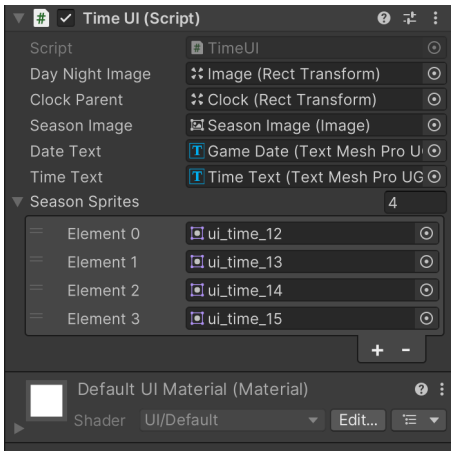


Figure 73-The Attributes of Time UI (Script)

```
                }
                // TODO: Refresh the map and seed grow
                EventHandler.CallGameDayEvent(gameDay, gameSeason);
            }

        }

        EventHandler.CallGameDateEvent(gameHour, gameDay, gameMonth, gameYear, gameSeason);
    }
    EventHandler.CallGameMinuteEvent(gameMinute, gameHour, gameDay, gameSeason);
    // TODO: Change the light
    EventHandler.CallLightShiftChangeEvent(gameSeason, GetCurrentLightShift(), timeDifference);
}
// Debug.Log("Second: " + gameSecond + " Minute: " + gameMinute);
}
```

Figure 74-Code Snippet of Event Handlers in Function of 'Update Game Time'

Explanation:

Figure 72 shows how 'Second Threshold' can be used for in-game timing: here, 'tik Time' is created to add a cumulative count to 'delta Time' is created to accumulate a count of 'delta Time', which is the time spent updating between two frames in a Unity game. Then, 'tik Time' is used to compare it to 'Second Threshold' to update game time. Specifically, if the 'tik Time' has accumulated enough time to be treated as a second, it needs to be reset and the 'Update Game Time' method called to update the time, which is a simulation of real time (60 seconds becomes a minute, 60 minutes becomes an hour, and so on).

At the same time, the management of the 'Time UI' is a rehash of the 'Inventory UI' in section 2.10. Specifically, the 'Time UI (Script)' was created to centrally manage all UI components, and the 'Event Handler' was created again for the time parameters such as hours and days

required by each UI component. for the time parameters required by each UI component, such as hours and days, 'Event Handler' is used again. In other words, when minutes, hours or days are changed, the 'Call Event' can be used to call all the data or UI components that need to be updated because of the change, for which an 'Event' has been registered.
For example, in Figure 74, from top to bottom there are 'Call Game Day Event', 'Call Game Date Event' and 'Call Game Minute Event', where 'Call Light Shift Change Event' is also present. And they are called when the days, hours, minutes, and seconds change respectively, which also passes the data to all the data and UI components that need to be updated based on time.
The special one is 'Call Light Shift Change Event'. It needs to determine if it is daytime or not by the current in-game time and calculate the time difference between the upcoming 'Light Shift', which are done by 'Get Current Light Shift'. After that, these two data are then transmitted to the 'Light' control code along with the 'Season' to update the 'Light', which is the time difference between day and night and between lights such as door lamps.

Problems Encountered:

```
public enum Season
{
    Spring, Summer, Autumn, Winter
}
```

Figure 75-The Data Structure of Season

```
if (monthInSeason == 0) // We need to update the season related data and UI components
{
    monthInSeason = 3; // set back the value of monthInSeason, which is 3 before

    int seasonNumber = (int)gameSeason; // get the int value from enum type, which represents different seasons

    seasonNumber++; // Change to next season

    if (seasonNumber > Settings.seasonHold) // If the season number is over 3, which represent a new year
    {
        seasonNumber = 0; // reset the seasonNumber, which means the season goes back to Spring
        gameYear++; // the year go to the next one
    }

    gameSeason = (Season)seasonNumber; // reset the enum value from int

    if (gameYear > 9999)
    {
        gameYear = 2024; // If it is over the displayed year limitation, back to the year when I finished the FYP.
    }
}
// TODO: Refresh the map and seed grow
```

Figure 76-Code Snippet of Updating Season Data in Update Game Time

Explanation:
One of the problems encountered is how to do type switching within Enum data. Because it is complicated to switch values directly inside the Enum, such as Figure 75's 'Season' data switch. The human mind switches seasons is simple, from spring directly to the next is summer. However, switching data inside the code may require re-assignment of values, in which case it makes the code more complex and less readable.
Thus, it is a question of how not to lose the readability of the code and not to increase the complexity of the code. In practice, however, this problem does not exist because Enum data itself can be forced to convert to int data. Specifically, when 'Season' is created, the 'Spring', 'Summer', 'Autumn' ' and 'Winter' themselves have been assigned values 1, 2, 3 and 4. So, by simply prefixing the 'Season' data with '(int) ' in front of the 'Season' data to do the forced conversion will solve the problem. Then, the forced conversion to the original data structure with '(Season)' is safe to be called by other code that needs it.

2.14 NPC Creation and Movement Schedule

Design:

Figure 77-The Design of NPCs

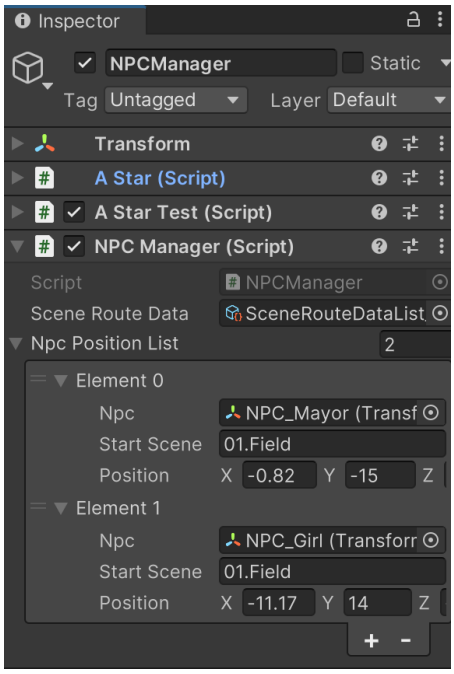


Figure 78-The Inspector of NPC Manager and Script

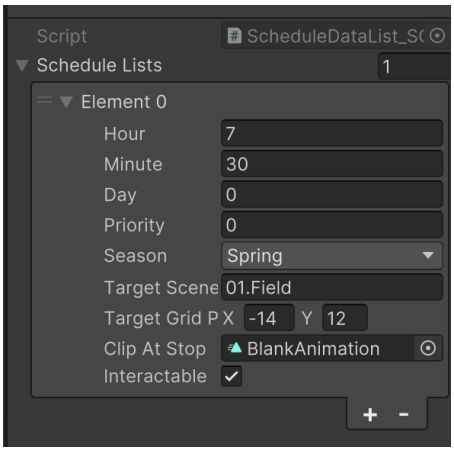


Figure 79-The Schedule Data List of Girl

Table 11-Basic Settings of NPC Movement

Variable Name	***Variable Type***	***Variable Meaning***
Normal Speed	Float	The normal speed of NPCs
Min Speed	Float	The min speed of NPCs
Max Speed	Float	The max speed of NPCs
Dir(ection)	Vector2	The direction of moving
Is Moving	Bool	Whether the NPC is moving or not
Interactable	Bool	Whether the NPC is interactable or not

Table 12-The Properties of Movement Step Calculated by A Star Algorithm

Variable Name	***Variable Type***	***Variable Meaning***
Scene Name	String	Which scene movement step is in
Grid Coordinate	Vector2Int	Coordinate system parameters of the grid for the movement step
Hour	Int	Time at which each movement step arrives
Minute		
Second		

Explanation:
Firstly, Figure 77 shows two NPCs, which share the same design: in addition to the main character, there is a 'Space' key displayed above the character's head, which is used to prompt the player to use this key to interact with the NPC. At the same time, they will also have movement animations as well as animations when they stop.
Figure 78 then shows the components of the 'NPC Manager', which is used to manage the locations of all NPCs and the application of the A* algorithm. The most important of these is the 'NPC Manager (Script)', which, as can be seen from the figure, records the scene and location of the NPC's starting location. Whereas the different but similar 'Schedule Data List', as shown in Figure 79, records the trajectory of the NPC's activity, which is related to the parameters of time, location, animation and whether it can be interacted with.
Therefore, to realize the movement of NPCs, the basic parameters in Table 11 and Table 12 need to be set in advance:

1. The parameters for NPC movement are shown in Table 11. Among them, 'Normal Speed' and 'Dir(ection)' are used to define the movement direction and speed. The 'Min Speed' and 'Max Speed' are used to determine whether the movement time is sufficient or not, which will be discussed in the following Implementation. Secondly, 'Is Moving' and 'Interactable' are used to provide pre-conditions for NPC interaction, which will be discussed in the 'Dialogue and Trade' section.
2. Table 12 shows the parameters of the grid that the NPC needs to pass through during its movement, which is called movement step. after the A Star algorithm is called to calculate the grid that it needs to pass through, the 'Scene Name' and 'Grid' associated with it are used. After the A Star algorithm is called to calculate the grid to pass

through, the associated 'Scene Name' and 'Coordinate' need to be recorded and then executed by the NPC. The time related parameters are used to compare with the in-game time and used to adjust and rationalize the NPC's movement trajectory, such as adjusting the NPC's position when there is enough time.

Implementation:

```
if (schedule.targetScene == currentScene)
{
    AStar.Instance.BuildPath(schedule.targetScene, (Vector2Int)currentGridPosition, schedule.targetGridPosition, movementSteps);
}
```

Figure 80-The Generation of Movement Steps in the Same Scene

```
private void UpdateTimeOnPath() // To record the time for every step
{
    MovementStep previousSetp = null; // create a previous step to determine

    TimeSpan currentGameTime = GameTime; // load the GameTime from Time Manager

    foreach (MovementStep step in movementSteps) // for all the movement steps generated by A star algorithm, we should record time for each step
    {
        if (previousSetp == null) // if this is the first step, then it will be assigned to previous step
            previousSetp = step;

        step.hour = currentGameTime.Hours; // record the current Game Time Calculated by the below code
        step.minute = currentGameTime.Minutes;
        step.second = currentGameTime.Seconds;

        TimeSpan gridMovementStepTime; // record the time it needs to move

        if (MoveInDiagonal(step, previousSetp)) // if moves in Diagonal, it should compute in a different way
            gridMovementStepTime = new TimeSpan(0, 0, (int)(Settings.gridCellDiagonalSize / normalSpeed / Settings.secondThreshold));
        else // otherwise it should compute as normal
            gridMovementStepTime = new TimeSpan(0, 0, (int)(Settings.gridCellSize / normalSpeed / Settings.secondThreshold));

        // Continuous addition to get the next step
        currentGameTime = currentGameTime.Add(gridMovementStepTime);
        // loop
        previousSetp = step;
    }
}
```

Figure 81-Cope Snippet of Updating Time on Path

```
private IEnumerator MoveRoutine(Vector3Int gridPos, TimeSpan stepTime)
{
    npcMove = true;
    nextWorldPosition = GetWorldPosition(gridPos); // Get World Position is used to calculate the center point

    // Still have time to Move
    if (stepTime > GameTime)
    {
        // The time used to move, which is measured by seconds
        float timeToMove = (float)(stepTime.TotalSeconds - GameTime.TotalSeconds);
        // Real move distance
        float distance = Vector3.Distance(transform.position, nextWorldPosition);
        // Max Real move speed
        float speed = Mathf.Max(minSpeed, (distance / timeToMove / Settings.secondThreshold));

        if (speed <= maxSpeed) // if there is still a long time to move, we can adjust the position gradually
        {
            while (Vector3.Distance(transform.position, nextWorldPosition) > Settings.pixelSize) // if the difference between current position and target position is bigger
                                                                                                 // than the pixel size. Then, we have to continue move
            {
                dir = (nextWorldPosition - transform.position).normalized; // get the direction

                Vector2 posOffset = new Vector2(dir.x * speed * Time.fixedDeltaTime, dir.y * speed * Time.fixedDeltaTime); // continuing get the position to move
                rb.MovePosition(rb.position + posOffset); // moves the npc's rigidbody, namely the npc
                yield return new WaitForFixedUpdate(); // wait for next update, which is quite a small time, which only cause a little change
            }
        }
    }

    // if we do not have too much time to move, where the required spped is much more than maxSpeed we set, then just move the npc directly.
    rb.position = nextWorldPosition;
    currentGridPosition = gridPos;
    nextGridPosition = currentGridPosition;

    npcMove = false;
}
```

Figure 82-Code Snippet of Main Logic of Moving

Explanation:

First, Figure 80 shows how 'movement steps' are created: the target scene, current coordinates and target coordinates are passed into the 'Astar' 'Build Path' method in 'Astar', where the 'Scene Name' and 'Grid Coordinate' of the movement steps are processed and added to the where movement steps' 'Scene Name' and 'Grid Coordinate' are processed and added to the movement steps.

However, the time parameters of the movement steps are still not calculated, which is an important judgement factor to measure the speed required by NPCs and set the parameters accordingly. Therefore, the 'Update Time on Path' in Figure 81 needs to be called at the start

along with the end of the 'Build Path' method call to calculate the time parameters of the movement steps. And the basic logic of this method is a For Loop call, which calculates the time needed for each step based on the distance between the grids and the NPC's 'Normal Speed', and then adds it up, so that each movement step will get the corresponding time parameter for each movement step.
Finally, after knowing all the parameters of the movement steps, the NPC will know how and when to move. Nevertheless, Figure 82 shows a more reasonable way on how to deal with movement patterns. Specifically, when the time of the end of the next movement step is known in advance, the speed at which the NPC completes the next movement step can be calculated. Then, if the calculated speed is greater than 'Min Speed' and less than 'Max Speed', it means that the NPC still has enough time to finish. In this case, the NPC will not move at 'Normal Speed', instead it will keep using the new calculated speed and fine-tune the NPC's position with the game's frame-by-frame updates until the calculated speed is greater than the 'Max Speed', which means that when the NPC has no time and is close to the target, the NPC's position will be moved directly to the target positions.

Problems Encountered:
The biggest problem with this section is that it does not finish do 'NPC Movement' across scenes. But since both Player and NPC have similar cross-scene movement logic. Therefore, they will be talked about in the continuous sections, which are section 2.15 and section 2.16.

2.15 Cross-Scene Movement for Player

Design:

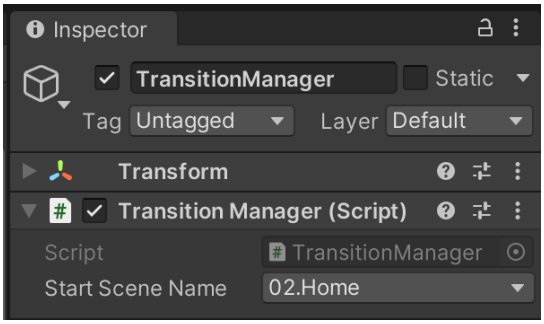


Figure 83-The Inspector of Transition Manger and Its Script

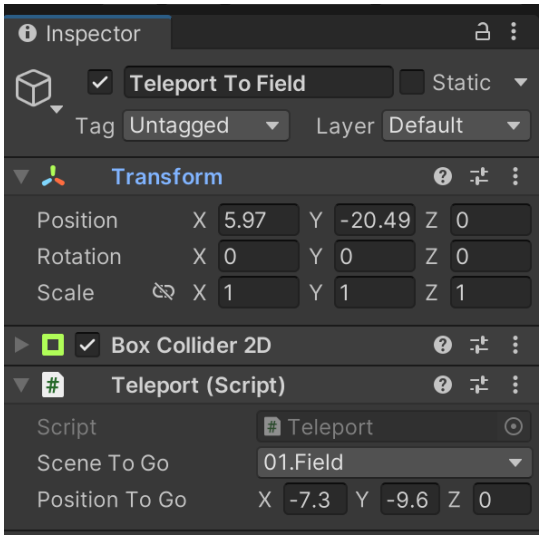


Figure 84-The Inspector of Teleport to Field and Its Script

Figure 85-The Place of Teleport to Field

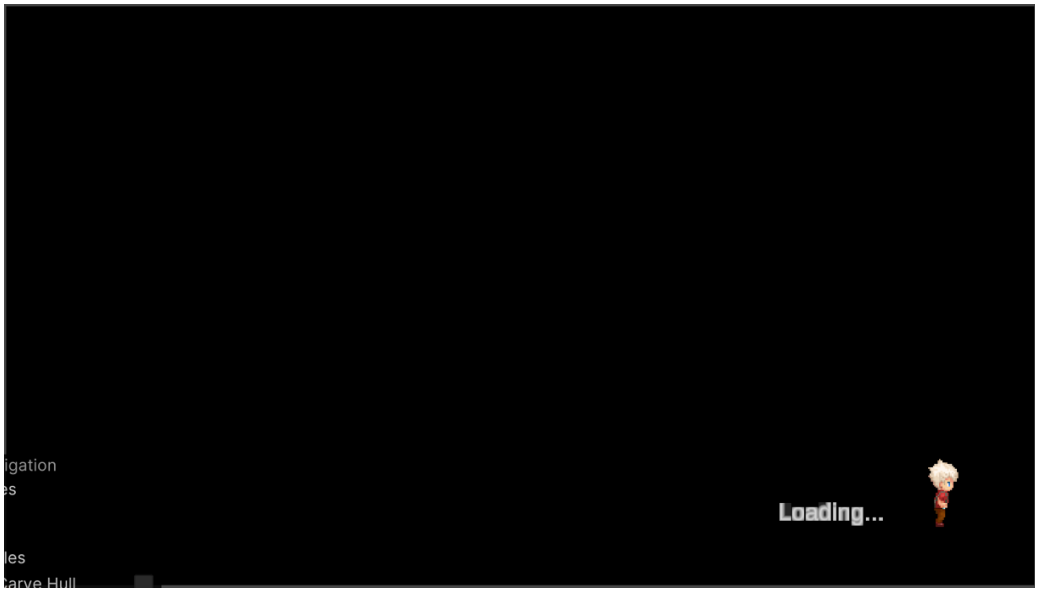


Figure 86-The Design of Fade Panel

Explanation:

Firstly, the 'Transition Manager' in Figure 83 was created to manage scene switching and record the 'Start Scene Name' for switching to a new start of the game. The ability to switch scenes is provided by Unity's 'UnityEngine.SceneManagement', which makes direct calls to the methods that handle 'Scene', which help the Transition Manager' to quickly implement scene switching and allow player targeting.

Meanwhile, the 'Teleport to Field' shown in Figure 84 is the project's trigger for moving from the 'Home' scene to the 'Field' scene. And Figure 85 shows its placement and trigger range, which is marked with a yellow circle. Specifically, as shown in Figure 84, 'Box Collier 2D' is added to this trigger, through which the collision between Player and 'Teleport to Field' can be detected, and then the code for the corresponding scene switch can be triggered in 'Teleport (Script)', which transports the Player to the scene and position shown at the bottom of Figure 84, which is 'Field ' scene at the (-7.3, -9.6, 0) coordinates.

Finally, since a single scene switch is very fast and abrupt, the 'Fade Panel' of Figure 86 was created to buffer the scene switching process. In this case, the 'Canvas Group' component was added to the 'Fade Panel', which makes the 'Fade Panel' can control the text and icons in the bottom right corner of Figure 86. Furthermore, the use of 'Fade Panel' is not complicated, which borrows from the handling of 'Scenery Tree' in 2.6, by changing the ‘Alpha’ value, the UI of 'Fade Panel' can be shown or hidden.

Implementation:

```
/// <summary>
/// Scene Change
/// </summary>
/// <param name="sceneName">The scene we want to go</param>
/// <param name="targetPosition">The default location</param>
/// <returns></returns>
private IEnumerator Transition(string sceneName, Vector3 targetPosition)
{
    EventHandler.CallBeforeSceneUnloadEvent(); // Before Unload the Scene, we should process some data like animator override

    yield return Fade(1); // Call Fade Panel with Alpha value 1 => Get Dark

    yield return SceneManager.UnloadSceneAsync(SceneManager.GetActiveScene()); // Unload the Scene According to the scene name

    yield return LoadSceneSetActive(sceneName); // Load Scene

    // change the position of the player
    EventHandler.CallMoveToPosition(targetPosition); // Call the Player to change the position

    EventHandler.CallAfterSceneLoadedEvent(); // After Loading a new Scene, some data need to be processed again like cinema confiner

    yield return Fade(0); // Call Fade Panel with Alpha value 0 => Get transparent
}

/// <summary>
/// Load the scene and activate it
/// </summary>
/// <param name="sceneName">The name of the scene</param>
/// <returns></returns>
private IEnumerator LoadSceneSetActive(string sceneName)
{
    yield return SceneManager.LoadSceneAsync(sceneName, LoadSceneMode.Additive); // add to the current scene list but not active

    Scene newScene = SceneManager.GetSceneAt(SceneManager.sceneCount - 1); // find the above scene and record it

    SceneManager.SetActiveScene(newScene); // then activate it
}
```

Figure 87-Code Snippet to Change Scene

```
/// <summary>
/// Loading the image during fading to make more smooth game experience
/// </summary>
/// <param name="targetAlpha"> the target alpha to change to (1 is black, 0 is transparent)</param>
/// <returns></returns>
private IEnumerator Fade(float targetAlpha)
{
    isFade = true; // Start to Fade

    fadeCanvasGroup.blocksRaycasts = true; // Stop raycast from the mouse to avoid wrong actions

    float speed = Mathf.Abs(fadeCanvasGroup.alpha - targetAlpha) / Settings.fadeDuration; // Compute the speed to change, fadeDurantion is 0.5f

    while (!Mathf.Approximately(fadeCanvasGroup.alpha, targetAlpha)) // if two alpha values are not similar (the same)
    {
        fadeCanvasGroup.alpha = Mathf.MoveTowards(fadeCanvasGroup.alpha, targetAlpha, speed * Time.deltaTime); // we should change the alpha value from the current
                                                                                                               // alpha value of fade Canvas Group to the target one
                                                                                                               // by using the pseed we calculated and
                                                                                                               // the time spent on frams to frams change in Unity

        yield return null;
    }

    fadeCanvasGroup.blocksRaycasts = false; // Enalbe raycast from the mouse to allow actions

    isFade = false; // stop fade
}
```

Figure 88-Code Snippet of Calling Fade Panel

```
private void onMoveToPosition(Vector3 targetPosition)
{
    transform.position = targetPosition;
}
```

Figure 89-Code Snippet of Calling Player to Change Position

Explanation:

This section makes more applications of the 'Event Handler' discussed in section 2.10. Firstly, when the 'Teleport to Field' shown in Figure 84 is triggered by the Player, then the 'Teleport (Script)' will ' Call Event' and transfers the data from 'Scene to Go' and 'Position to Go' over to another code to perform further actions, which is the code shown in Figure 87.

But before further processing, the 'Fade Panel' related code needs to be introduced. As shown in Figure 88, the 'Fade' method is used to adjust the 'Alpha' value of the 'Fade Panel' such that the 'Fade Panel' to be black or hidden.

After getting the name and location of the scene Player goes to, the 'Transition' method in Figure 87 will start processing, and its main logic is:

1. Before scene unloading, all the data specific to the original scene need to be processed, so 'Call Before Scene Unload Event'.

2. Call the 'Fade' method and set the 'Alpha' value to 1, which is to make the 'Fade Panel' black so that the player can't see the scene transformation process. which is to make the 'Fade Panel' black so that the player can't see the scene change.
3. Then, based on the 'Scene Name' transferred into 'Scene to Go', scenes with the same name will be loaded and activated through the methods provided by 'UnityEngine.SceneManagement', which are integrated into the 'Load Scene Set Active' method at the bottom of Figure 87.
4. Thus, it is time to 'Call Event', which is 'Call Move to Position', to make the 'Player (Script) Event' in 'Player (Script)' to change the Player's position, which is the code in Figure 89.
5. Like step 1, after the scene is loaded, there is also some data that needs to be acquired for the new scene to be processed, so 'Call After Scene Loaded Event'.
6. Finally, call the 'Fade' method and set the 'Alpha' value to 0, which is to make the 'Fade Panel' hidden to allow the player to see the scene after loading.

Hence, the scene switching for Player is implemented.

Problems Encountered:

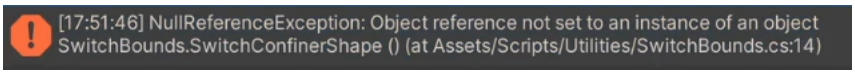


Figure 90-Screenshot of Switch Bounds Bug caused by Scene Changing

```
public class SwitchBounds : MonoBehaviour
{
    // we want to call this method at the start of game
    private void OnEnable()
    {
        EventHandler.AfterSceneLoadedEvent += SwitchConfinerShape;
    }

    private void OnDisable()
    {
        EventHandler.AfterSceneLoadedEvent -= SwitchConfinerShape;
    }

    // define a method to find the value of polygan 2D shape of the other shape
    private void SwitchConfinerShape()
    {
        // tag the bounds shape we need and then use this method to find them
        PolygonCollider2D confinerShape = GameObject.FindGameObjectWithTag("BoundsConfiner").GetComponent<PolygonCollider2D>();

        // before assigning the value to the confiner of original cinemachine, we need to get this component in the code
        CinemachineConfiner confiner = GetComponent<CinemachineConfiner>();

        // assign the value to it
        confiner.m_BoundingShape2D = confinerShape;

        // call this method to clean the plath cache: call this if the bounding shape changes at runtime
        confiner.InvalidatePathCache();
    }
}
```

Figure 91-Code Snippet of Switch Bounds after Integrated with Transition Manager

```
private void OnBeforeSceneUnloadEvent()
{
    holdItem.enabled = false;
    SwitchAnimator(PartType.None);
}
```

Figure 92-Code Snippet of Animator after Integrated with Transition Manager

Explanation:

Despite having been dealt with in the Implementation section, the main problem encountered in this section is that after or after the scene is reloaded, some of the data is not processed to the point where errors are reported frequently.

For example, when the 'Field' scene is unloaded and the 'Home' scene is loaded, the bug hinted at in Figure 90 occurs, which is caused by the scene switching afterwards, this is

because the ' Confiner' required by 'Cinemachine Confiner' is not read, which in the 2.5 section of the design will only be read once at the start of the game. Therefore, this part needs to be modified, as shown in Figure 91, where 'Confiner' needs to be re-read after every scene load. To be specific, the 'Switch Confiner Shape' method needs to be 'After Scene Loaded Event' an 'Event ' and then wait for the 'Call After Scene Loaded Event', which is what is shown in Figure 87.

Another example of 'Before Scene Unload' is 'Animator Override'. Specifically, when the Player has an animation being played before the scene change, it may conflict with the new animation after the scene change. Therefore, parts related to Player animations such as 'Hold Item' and 'Part Type' are cancelled or restored to their default values before the scene is unloaded, as shown in Figure 92.

2.16 Cross-Scene Movement for NPC

Design:

Table 13-The Data Structure of Scene Route

Variable Name	***Variable Type***	***Variable Meaning***
From Scene Name	String	Name of the scene where the route begins
Go to Scene Name	String	Name of the scene where the route ends
Scene Path List	List<Scene Path>	The list of Scene Path

Table 14-The Data Structure of Scene Path

Variable Name	***Variable Type***	***Variable Meaning***
Scene Name	String	Name of the scene the movement step begins
From Grid Cell	Vector2Int	The position of the start movement step
Go to Grid Cell	Vector2Int	The position of the end movement step

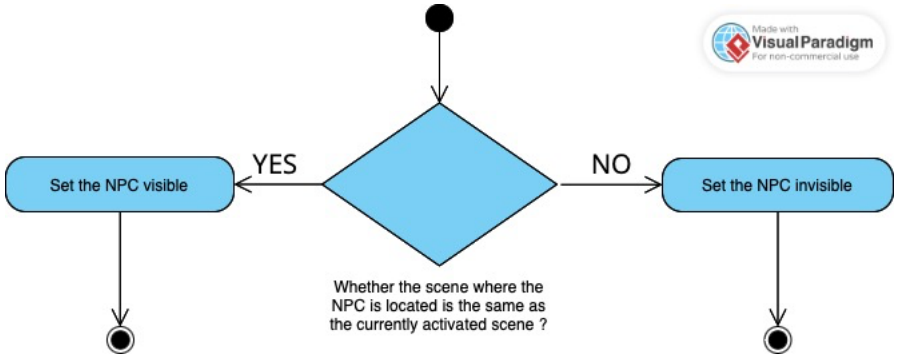


Figure 93-The Logic of NPC Visible Check

Explanation:

The logic for NPC scene switching is very similar to the logic for Player scene switching in part 2.15. Except that NPC scene switching does not require switching scenes. In other words, when the NPC enters another scene, which is not in the same scene as the Player, you only

need to set the NPC to 'Invisible'. In this way, it is possible to simulate the feeling of an NPC crossing scenes, the exact logic is shown by Figure 93.
However, the NPC-related path data still needs to be processed. For example, Table 13 and Table 14 together form the 'Scene Route' which is the trigger for NPC scene switching. Specifically, 'Scene Route' will record the location that the NPC needs to reach when the scene switches, and then generate the NPC's path as in part 2.14.
So, by combining the above two, NPCs can move across scenes.

Implementation:

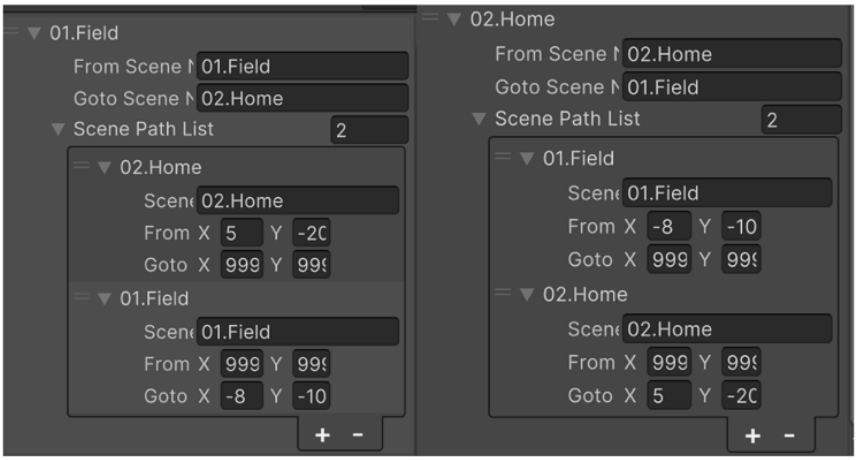


Figure 94-The Two Scene Route Examples

```
SceneRoute sceneRoute = NPCManager.Instance.GetSceneRoute(currentScene, schedule.targetScene);

if(sceneRoute != null)
{
    for(int i = 0; i < sceneRoute.scenePathList.Count; i++)
    {
        Vector2Int fromPos, gotoPos;
        ScenePath path = sceneRoute.scenePathList[i];

        if(path.fromGridCell.x >= Settings.maxGridSize) // if the fromGrid Cell is 99999 bigger than maxGridSize
        {
            fromPos = (Vector2Int)currentGridPosition; // then it means the from position is the current position of NPC
        }
        else // otherwise, it needs to take the from grid cell position in the settings
        {
            fromPos = path.fromGridCell;
        }

        if(path.gotoGridCell.x >= Settings.maxGridSize) // if the gotoGrid Cell is 99999 bigger than maxGridSize
        {
            gotoPos = schedule.targetGridPosition; // then it means it should take the target grid position
        }
        else // if not
        {
            gotoPos = path.gotoGridCell; // it should go the position in settings
        }
        AStar.Instance.BuildPath(path.sceneName, fromPos, gotoPos, movementSteps); // generate the movement steps
    }
}
```

Figure 95-Code Snippet of Generating Cross-Scene Path

```
private void CheckVisiable()
{
    if (currentScene == SceneManager.GetActiveScene().name) // if the current scene of NPC
                                                            // equal to the active scene right now
        SetActiveInScene(); // we set the NPC is visible
    else
        SetInactiveInScene(); // if not, we set the NPC is invisible
}
```

Figure 96-Code Snippet of Checking Visible

Explanation:

Figure 94 shows two Scene Routes, one from 'Field' to 'Home' and the other from 'Home' to Field' and the other is from 'Home' to 'Field'. They have the same logic as Player's Cross-Scene: the trigger location is (5, -20) for 'Home' and (-8, -10) for 'Field'. One difference is that the acquisition of the NPC's current location and target location is different from Player, which constantly reads user input. More specifically, 'Max Grid Size', which is 99999, was created to be used in the code to determine whether to read the current position or target position of the NPC to generate a path. For example, the 'From' of the 'Field' in the lower left corner of Figure 94 has a coordinate system of (99999, 99999), and in the event that such a coordinate, which does not exist in the actual game map, is set out, it represents a path to be read from the current position of the NPC; in contrast, the 'Go to' Position of 'Home' above 'Field' is (99999, 99999), which means that the NPC's current position is (99999, 99999), which means that the target position set in the 'Schedule' of the NPC is read here. Moreover, this whole logic is what the code in Figure 95 implements.
Meanwhile the logic of Figure 93 is implemented in the code in Figure 96, which is the if-else statement judgement.

Problems Encountered:

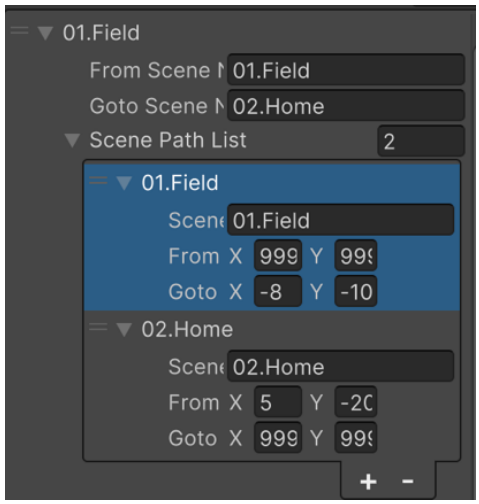


Figure 97-The Wrong Version of Scene Route Data

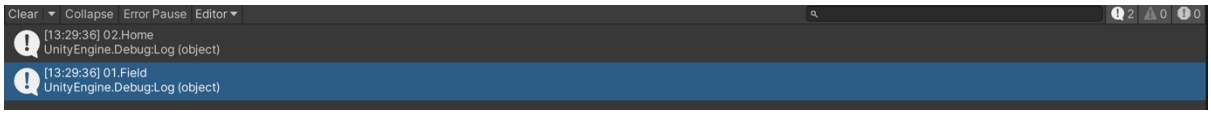


Figure 98-The Debug of the Correct Version of Scene Route

Explanation:
The biggest problem is encountered when filling in the 'Scene Route' data, which means that the way Unity determines the order of the list is directly opposite to the way it is filled in by humans. To be more specific, Figure 97 shows how the 'Scene Route' is filled in at the beginning, because intuitively, the first one to be filled in must come first in the list data. However, this is wrong, and the data goes in and out like a 'Stack' (First In Last Out). The correct way of filling is shown in Figure 94, which is confirmed by the Unity debug report in Figure 98.
Since the order of the List data was not as strict as this section, this problem was never noticed. However, with a little care in filling it out, this problem can be avoided.

2.17 Dialogue Creation

Design:

Figure 99-The UI of Dialogue

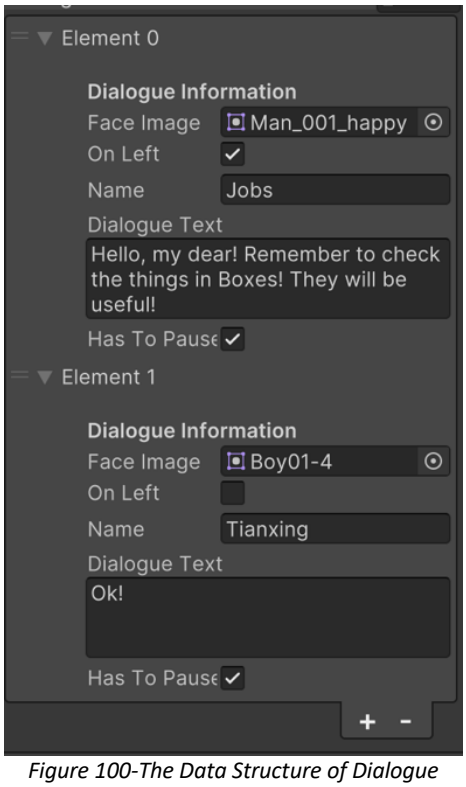


Figure 100-The Data Structure of Dialogue

Explanation:

The dialogue UI is shown in Figure 99. Two of the characters' avatars will be placed on either side of each other to create the feel of the dialogue. The 'Dialogue Text' is then used to display the content of the dialogue, which is accompanied by an 'Instruction' to indicate how to proceed.

Meanwhile, the data structure of the 'Dialogue' is shown in Figure 100, which is used to be processed by the script and transferred to the UI to display the content. The extra 'Has to Pause' is used to briefly stop the dialogue.

Moreover, the method of implementation is not different from the 'Inventory UI' in part 2.10 and the 'Time UI' in part 2.13, both transferring data to the 'UI ' section, and then updating the data for display.

Implementation:

```
private void OnTriggerEnter2D(Collider2D other)
{
    if (other.CompareTag("Player"))
    {
        canTalk = !npc.isMoving && npc.interactable;
    }
}

private void OnTriggerExit2D(Collider2D other)
{
    if (other.CompareTag("Player"))
    {
        canTalk = false;
    }
}
```

Figure 101-Code Snippet of Setting 'is Talking' Boolean Value

```
/// <summary>
/// create dialogue stack
/// </summary>
private void FillDialogueStack()
{
    dailogueStack = new Stack<DialoguePiece>();
    for (int i = dialogueList.Count - 1; i > -1; i--)
    {
        dialogueList[i].isDone = false;
        dailogueStack.Push(dialogueList[i]);
    }
}

private IEnumerator DailogueRoutine()
{
    isTalking = true;
    if (dailogueStack.TryPop(out DialoguePiece result)) // if there is dialogue, pop up
    {
        //Display the Dialogue
        EventHandler.CallShowDialogueEvent(result);
        yield return new WaitUntil(() => result.isDone);
        isTalking = false;
    }
    else // if there is no dialogue
    {
        //Close the Dialogue along with restoration
        EventHandler.CallShowDialogueEvent(null);
        FillDialogueStack();
        isTalking = false;

        if(onFinishEvent != null)
        {
            onFinishEvent.Invoke();
            canTalk = false;
        }
    }
```

Figure 102-Code Snippet of Displaying Dialogue

Explanation:

Figure 102 shows the entire logic of the dialogue. First, if the character is 'can Talking', which is determined by the code in Figure 101, the 'if' of the 'Dialogue Routine' will be executed, which will transfer the data to the 'Dialogue UI' to update the UI data display through 'Call Show Dialogue Event'. Until all the dialogue data is displayed in the game, the 'else' of the 'Dialogue Routine' will be executed, where it will close the 'Dialogue UI ' and 'Fill Dialogue Stack' is called to repopulate the original dialogue data. In addition to this, 'On Finish Event' will also be executed when the dialogue is finished, through which items can be traded.

Problems Encountered:

```
private void OnUpdateGameStateEvent(GameState gameState)
{
    // The value is changed in NPCFunction Script
    switch (gameState)
    {
        case GameState.GamePlay:
            inputDisable = false;
            break;

        case GameState.Pause:
            inputDisable = true;
            break;
    }
}
```

Figure 103-Code Snippet of Controlling the Input in Player (Script)

```
private IEnumerator DailogueRoutine()
{
    isTalking = true;
    if (dailogueStack.TryPop(out DialoguePiece result)) // if there is dialogue, pop up
    {
        //Display the Dialogue
        EventHandler.CallShowDialogueEvent(result);
        EventHandler.CallUpdateGameStateEvent(GameState.Pause);
        yield return new WaitUntil(() => result.isDone);
        isTalking = false;
    }
    else // if there is no dialogue
    {
        //Close the Dialogue along with restoration
        EventHandler.CallShowDialogueEvent(null);
        EventHandler.CallUpdateGameStateEvent(GameState.GamePlay);
        FillDialogueStack();
        isTalking = false;

        if(onFinishEvent != null)
        {
            onFinishEvent.Invoke();
            canTalk = false;
        }
    }
}
```

Figure 104-The Updated Code Snippet of Dialogue Routine

Explanation:

The biggest problem with this section is that the Player can still move when a dialogue has started, which is not expected.

So, in 'Player (Script)' the input of the user is set on the premise. As Figure 103 shows, user input is only detected when the 'Game State' is 'Game Play', whereas if it is 'Pause ', the input of the user is not detected.

Specifically, when the dialogue starts, the updated dialogue code in Figure 104 calls 'Call Update Game State Event', which passes the 'Pause' data to the 'Player (Script)' so that the user's input is not detected until after the dialogue is over, when the 'Gameplay' data is passed into the 'Player (Script)' so that user input is not detected until the dialogue is over, then the 'Gameplay' data will be passed to 'Player (Script)' so that user input is valid again.

2.18 Trade System

Design:

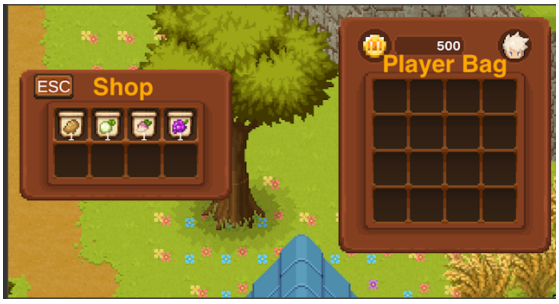


Figure 105-The Design of Trade Instruction

Figure 106-The Design of Trade UI

Explanation:

Figure 105 shows the interface where' Trade' begins: the 'Shop' bag on the left and the 'Player' bag on the right, where items can be dragged and dropped on top of each other to complete the trading process. At the same time, to control the number of trades, the 'Trade UI' in Figure 106 is used to guide the player through the trade process.

Therefore, the most difficult point in this part is the processing of data during the transaction and how to judge whether to buy or sell.

Moreover, for the shop, all its design is built on the data structure and UI design of the previous bag, which can be called the bag of NPCs, only for the 'Money' and the character's 'Icon' being replaced by the 'ESC' icon. At the same time, the updating and displaying of UI data has been repeatedly discussed in several parts, thus it will not be repeated here.

Therefore, the code implementation logic of the transaction is described in detail in this section of Implementation.

Implementation:

```
// if both are bags type, it mean it's just change between different bags
if(slotType == SlotType.Bag && targetSlot.slotType == SlotType.Bag)
{
    InventoryManager.Instance.SwapItem(slotIndex, targetIndex);

}else if(slotType == SlotType.Shop && targetSlot.slotType == SlotType.Bag) // Buy something
{
    EventHandler.CallShowTradeUI(itemDetails, false);
}
else if(slotType == SlotType.Bag && targetSlot.slotType == SlotType.Shop) // Sell something
{
    EventHandler.CallShowTradeUI(itemDetails, true);
}
```

Figure 107-Code Snippet of Item Drag and Drop

```
private void OnCallShowTradeUI(ItemDetails item, bool isSell)
{
    tradeUI.gameObject.SetActive(true);
    tradeUI.SetupTradeUI(item, isSell);
}
```

Figure 108-Code Snippet of Calling Show Trade UI

```
/// <summary>
/// Setup of TradeUI Display
/// </summary>
/// <param name="item"></param>
/// <param name="isSell"></param>
public void SetupTradeUI(ItemDetails item, bool isSell)
{
    this.item = item;
    itemIcon.sprite = item.itemIcon;
    itemName.text = item.itemName;
    isSellTrade = isSell;
    tradeAmount.text = string.Empty;
}

private void TradeItem()
{
    var amount = Convert.ToInt32(tradeAmount.text); // record the input number
    InventoryManager.Instance.TradeItem(item, amount, isSellTrade); // call the 'Inventory Manger' to calculate related data
    CancelTrade(); // close the trade UI
}

private void CancelTrade()
{
    this.gameObject.SetActive(false);
}
```

Figure 109-Code Snippet of Set Up Trade UI

```
/// <summary>
/// Trade Items
/// </summary>
/// <param name="itemDetails"></param>
/// <param name="amount"></param>
/// <param name="isSellTrade"></param>
public void TradeItem(ItemDetails itemDetails, int amount, bool isSellTrade)
{
    int cost = itemDetails.itemPrice * amount;
    // Get the bag index of the item
    int index = GetItemIndexInBag(itemDetails.itemID);

    if (isSellTrade)    // To sell something
    {
        if (playerBag.itemList[index].itemAmount >= amount)
        {
            RemoveItem(itemDetails.itemID, amount);
            // the total price you will get
            cost = (int)(cost * itemDetails.sellPercentage);
            playerMoney += cost;
        }
    }
    else if (playerMoney - cost >= 0)   // To buy something
    {
        if (CheckBagCapacity())
        {
            AddItemAtIndex(itemDetails.itemID, index, amount);
        }
        playerMoney -= cost;
    }
    // Refresh UI
    EventHandler.CallUpdateInventoryUI(InventoryLocation.Player, playerBag.itemList);
}
```

Figure 110-Code Snippet of Trading Item

Explanation:

For the implementation of the trading system, it is not difficult because of the detailed setup of the Bag logic and UI in sections 2.9 and 2.10.

First, based on the already completed item drag and drop exchange system, the purchase and sale of items can be determined. Specifically, after the slot type of 'Shop' is set to 'Shop', when an item is dragged out of 'Shop' and dragged to the 'Bag', then it is a buy; conversely, it is a sell. This is what the code in Figure 107 implements. In addition to this, there is a Boolean value that is also transmitted when 'Call Show Trade UI' is called, which is a judgement on 'is Sell'.

Then, in Figure 108, the 'Trade UI' is called and displayed. The reason this step is performed this way is because 'Trade UI' is a child object of 'Inventory UI', hence it has to be called in 'Inventory UI (Script)'.
Next, all the data related to the 'Item' is transferred to the 'Trade UI (Script)' to be processed, which is shown in Figure 109. Subsequently, when it is detected that the player has entered the relevant number, 'Trade Item' is called to tell the 'Inventory Manager (Script)' to process the data.
Finally, the 'Trade Item' method in the 'Inventory Manager (Script)' calculates and subtracts or adds the amount of money needed to spend on the transaction and how to store or remove the item, after which 'Call Update Inventory UI' is called to update all relevant data and UI displays.

Problems Encountered:

```
public void OpenShop()
{
    isOpen = true;
    EventHandler.CallBaseBagOpenEvent(SlotType.Shop, shopData);
    EventHandler.CallUpdateGameStateEvent(GameState.Pause);
}

public void CloseShop()
{
    isOpen = false;
    EventHandler.CallBaseBagCloseEvent(SlotType.Shop, shopData);
    EventHandler.CallUpdateGameStateEvent(GameState.GamePlay);
}
```

Figure 111-Code Snippet of NPC function to Open or Close Shop

Explanation:
The biggest problem with this part is the same as the 2.17 section, which is that the Player is not expected to be able to move when the shop is open. Hence, in Figure 111, the 'Call Update Game State Event' is similarly called to limit user input.

2.19 Crop System

Design:

Table 15-The New Added Properties in Tile Details

Variable Name	***Variable Type***	***Variable Meaning***
Day Since Dug	int	The day since last dug
Day Since Watered	int	The day since last watered
Seed Item ID	int	The seed item id in this grid (tile)
Growth Days	int	The growth day of this seed in this grid (tile)
Day Since Last Harvest	int	The day since last harvest

Table 16-The Main Data Structure of Crop Details

Variable Name	***Variable Type***	***Variable Meaning***
Seed Item ID	int	The ID of each seed item

Growth Days	Int []	The Growth Days for different stage.
Total Growth Days	int	
Growth Prefabs	Game Object []	The game objects for different growth stages
Growth Sprites	Sprite []	The sprites for different growth stages
Harvest Tool Item ID	Int []	The tool to harvest this item
Require Action Count	Int []	The required tool count to
Produced Item ID	Int []	The item will be produced after successfully harvested
Produced Min Amount	Int []	The min amount of the produced item
Produced Max Amount	Int []	The max amount of the produced item

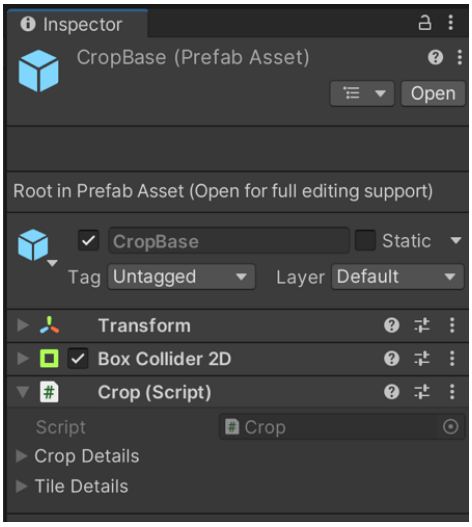


Figure 112-The Inspector of Crop Base

Explanation:

Table 15 shows a few of the new variables added to 'Tile Details' in section 2.12, which are all related to the crop system. Specifically, 'Day Since Dug' and 'Day Since Watered' are used to reset the 'dig Tile' and the 'water Tile', which are shown in section 2.12. The next one, 'Seed Item ID' who is used to store the IDs of the seeds planted inside this grid, which can be used to retrieve the seed library to get the seed information. Then, 'Growth Days' records the number of days the seed has been growing and 'Day Since Last Harvest' records the number of days since the last harvest, which is used to help plants that can grow again. which is used to help determine the growth process of plants that can grow again.

Table 16 shows some of the important data structures in 'Crop Details' which are used to complete the main parameters of the 'Crop System'. The details are described in the table.

In addition, just as the 'Item Base' was created to form all the Items in section 2.7, the 'Crop Base' was created to form all the Crop-related Game Objects.

Implementation:

```
private void OnPlantSeedEvent(int ID, TileDetails tileDetails)
{
    CropDetails currentCrop = GetCropDetails(ID);
    if (currentCrop != null && SeasonAvailable(currentCrop) && tileDetails.seedItemID == -1)    //Used for the first seed planting
    {
        tileDetails.seedItemID = ID;
        tileDetails.growthDays = 0;
        //Display the crop plants
        DisplayCropPlant(tileDetails, currentCrop);
    }
    else if (tileDetails.seedItemID != -1)  //used to refresh the map
    {
        //Display the crop plants
        DisplayCropPlant(tileDetails, currentCrop);
    }
}
```

Figure 113-Code Snippet of On Plant Seed Event

```
/// <summary>
/// Display the map tile
/// </summary>
/// <param name="sceneName">the scene name</param>
private void DisplayMap(string sceneName)
{
    foreach (var tile in tileDetailsDict)
    {
        var key = tile.Key; // get every tile detiles value
        var tileDetails = tile.Value;

        if (key.Contains(sceneName)) // update them
        {
            if (tileDetails.daySinceDug > -1)
                SetDigGround(tileDetails);
            if (tileDetails.daySinceWatered > -1)
                SetWaterGround(tileDetails);
            if(tileDetails.seedItemID > -1)
                EventHandler.CallPlantSeedEvent(tileDetails.seedItemID, tileDetails); // call the method again
        }
    }
}

/// </summary>
```

Figure 114-Code Snippet of Display Map

```
/// <summary>
/// Display the crop plants
/// </summary>
/// <param name="tileDetails">tile map details</param>
/// <param name="cropDetails">seed information</param>
private void DisplayCropPlant(TileDetails tileDetails, CropDetails cropDetails)
{
    //during the growth stages
    int growthStages = cropDetails.growthDays.Length;
    int currentStage = 0;
    int dayCounter = cropDetails.TotalGrowthDays;

    //calculate and determine the current stage
    for (int i = growthStages - 1; i >= 0; i--)
    {
        if (tileDetails.growthDays >= dayCounter)
        {
            currentStage = i;
            break;
        }
        dayCounter -= cropDetails.growthDays[i];
    }

    //access the current item(crop plants) prefabs
    GameObject cropPrefab = cropDetails.growthPrefabs[currentStage];
    Sprite cropSprite = cropDetails.growthSprites[currentStage];

    //should be generated in the center of the grid
    Vector3 pos = new Vector3(tileDetails.gridX + 0.5f, tileDetails.gridY + 0.5f, 0);

    //generate the crop in game world
    GameObject cropInstance = Instantiate(cropPrefab, pos, Quaternion.identity, cropParent);

    //Assign the value we get to them
    cropInstance.GetComponentInChildren<SpriteRenderer>().sprite = cropSprite;
    cropInstance.GetComponent<Crop>().cropDetails = cropDetails;
    cropInstance.GetComponent<Crop>().tileDetails = tileDetails;
}
```

Figure 115-Code Snippet Crop Plant

Explanation:
The three code snippets above show the code implementation of the crop growth process. Firstly, when the seed is planted for the first time, the code under 'if' in Figure 113 is executed, which saves the ID of the seed and starts the 'Growth Day' timer. Then 'Display Crop Plant' will be called, and which will show the initial stage of growth of this seed.
Then, when the number of days in the game changes, the information about the 'Tile Details' such as the 'Growth Day' is updated, which is done through the 'Time System' in section 2.13. 'Time System' is implemented. Subsequently, 'Display Map' in Figure 114 is called to display the new map, which again calls the code in Figure 113 through 'Call Plant Seed Event'. However, this time, since the grid already has the seed ID in it, it will execute the code under 'else', which is 'Display Crop Plant' again.
Although, the role of 'Display Crop Plant' was sketched in the first part, it still needs to be described in detail. First, after getting the current details of the tile and the details of the crop, the code will first sort out how many stages there are, record the current stage and the total number of days it takes to grow. Then the judgement of the growth stage will be made. Finally, when the judgement is over, 'Crop Base' will be called and will be assigned a value using the stage 'Sprite' stored in 'Crop Details', after which it is generated at the position computed from the tile details obtained at the beginning.
Finally, since the construction of the 'Item Base' has already been completed in section 2.7, the generation for 'Produced Item' is simple. To be more specific, it is when the entire harvest action is completed, which is when the ripe crop is bumped to the corresponding 'Require Action Count' by the corresponding 'Harvest Tool Item ID' shown in Table 16. After that, the code will calculate a final 'Produced Amount' by 'Produced Min Amount' and 'Produced Max Amount'. Then, with the 'Produced Item ID' and 'Produced Amount' it is possible to call the code in 'Inventory Manager', which holds the IDs of all Items, loops through the code to generate the 'Produced Item'.

Problems Encountered:

Figure 116-The Design of Crop Tree

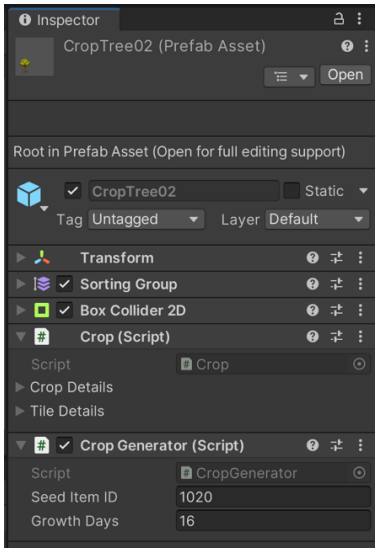


Figure 117-The Inspector of Crop Tree

```
private void GenerateCrop()
{
    Vector3Int cropGridPos = currentGrid.WorldToCell(transform.position);

    if (seedItemID != 0)
    {
        var tile = GridMapManager.Instance.GetTileDetailsOnMousePosition(cropGridPos);

        if (tile == null)
        {
            tile = new TileDetails();
            tile.gridX = cropGridPos.x;
            tile.gridY = cropGridPos.y;
        }

        tile.daySinceWatered = -1;
        tile.seedItemID = seedItemID;
        tile.growthDays = growthDays;

        GridMapManager.Instance.UpdateTileDetails(tile);
    }
}
```

Figure 118-Code Snippet of Generating Crop

Explanation:

In Section 2.6, since the ‘Scenery Tree’ could not be used directly as a cuttable tree, the tree in Figure 116 was created, consisting of a trunk and a top. The process of creating this tree is relatively simple, simply adding its data to the ‘Crop Details’ data list and combing two game objects with sprites. However, to preload these "Crop Tree" in the scene like the "Scenery Tree", the approach shown in Figure 117 and Figure 118 needs to be taken. Specifically, a "Crop Generator (Script)" needs to be created to manage this special case. Meanwhile, the essential code of this script is shown in Figure 118, and the location and details of the tiles must be known in advance and updated in the Grid Map Manager. This is achieved by calling the "Call After Scene Loaded Event" in the "Grid Map Manager". This means that the method will be executed after the scene is loaded.

2.20 Music System

Design:

```
public enum SoundName
{
    none, FootStepSoft, FootStepHard,
    Axe, Pickaxe, Hoe, Reap, Water, Basket,
    Pickup, Plant, TreeFalling, Rustle,
    AmbientCountryside1, AmbientCountryside2, MusicCalm1,
    MusicCalm2, MusicCalm3, MusicCalm4, MusicCalm5, MusicCalm6,
    AmbientIndoor1
}
```

Figure 119-Code Snippet of Sound Name Enum

Table 17-The Data Structure of Scene Sound Item

Variable Name	***Variable Type***	***Variable Meaning***
Scene Name	String	The scene name where the music plays
Ambient	Sound Name	The ambient music
Music	Sound Name	The background music

Table 18-The Data Structure of Sound Details

Variable Name	***Variable Type***	***Variable Meaning***
Sound Name	Sound Name	The Sound Name of the music
Sound Clip	Audio Clip	The music file
Sound Pitch Min	float	The min pitch of the music
Sound Pitch Max	float	The max pitch of the music
Sound Volume	Float [Range (0.1f, 1f)]	The volume of the music

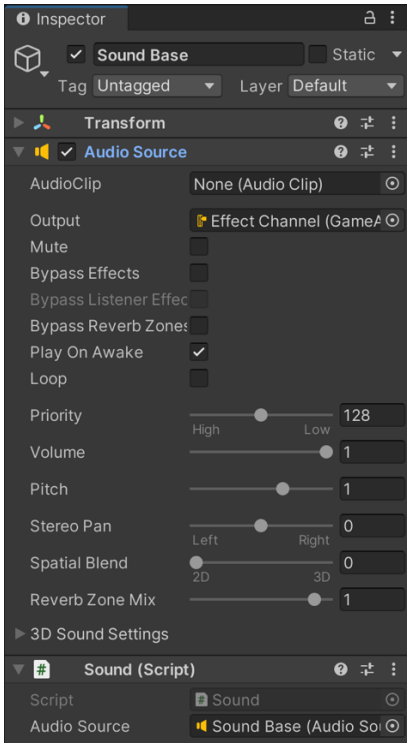


Figure 120-The Inspector of Sound Base

Explanation:
Firstly, Figure 119 shows the Enum value for 'Sound Name', which was originally entered as a separate string, but was changed to this form because it was time consuming and inconvenient.
Secondly, Table 17 shows the data structure of 'Scene Sound Item', which is mainly used to record the background and ambient music needed for each scene.
Then, Table 18 shows the data structure of 'Sound Details', which is like 'Item Details' to record all the required music data such as pitch, volume, and the important music files.
Next, Figure 120 shows the 'Sound Base', which is like the 'Item Base', is used to quickly call the data in 'Sound Details' to implement frequent sound displays such as chopping down trees and knocking on rocks.
Finally, all this music will be centrally managed using the 'Audio Mixer' in Unity for things like volume adjustment. Moreover, all the music files come from free sound website [21].

Implementation:

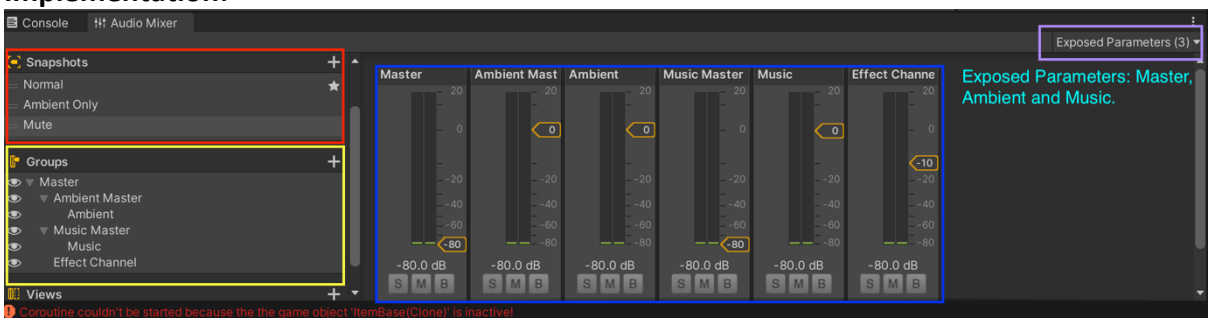


Figure 121-Screenshot of the Final Audio Mixer

```
private IEnumerator PlaySoundRoutine(SoundDetails music, SoundDetails ambient)
{
    if (music != null && ambient != null)
    {
        PlayAmbientClip(ambient, 1f);
        yield return new WaitForSeconds(MusicStartSecond);
        PlayMusicClip(music, musicTransitionSecond);
    }
}

/// <summary>
/// Play the Background Music
/// </summary>
/// <param name="soundDetails"></param>
private void PlayMusicClip(SoundDetails soundDetails, float transitionTime)
{
    audioMixer.SetFloat("MusicVolume", ConvertSoundVolume(soundDetails.soundVolume));
    gameSource.clip = soundDetails.soundClip;
    if (gameSource.isActiveAndEnabled)
        gameSource.Play();

    normalSnapShot.TransitionTo(transitionTime);
}


/// <summary>
/// Play the Amient Music
/// </summary>
/// <param name="soundDetails"></param>
private void PlayAmbientClip(SoundDetails soundDetails, float transitionTime)
{
    audioMixer.SetFloat("AmbientVolume", ConvertSoundVolume(soundDetails.soundVolume));
    ambientSource.clip = soundDetails.soundClip;
    if (ambientSource.isActiveAndEnabled)
        ambientSource.Play();

    ambientSnapShot.TransitionTo(transitionTime);
}
```

Figure 122-Code Snippet of the Logic of Sound Play

```
private void OnPlaySoundEvent(SoundName soundName)
{
    var soundDetails = soundDetailsData.GetSoundDetails(soundName);
    if (soundDetails != null)
        EventHandler.CallInitSoundEffect(soundDetails);
}
```

Figure 123-Code Snippet of Play Sound Event

Explanation:

Firstly, Figure 121 shows what the final 'Audio Mixer' looks like:

- Inside the red border are the 'Snapshots', which record the volume at which the in-game music needs to be played in several preset scenarios. They are 'Normal', 'Ambient Only' and 'Mute'. In more detail, 'Normal' is the volume setting for normal play, 'Ambient Only' is used when the scene is switched, and 'Mute 'Mute' is used when the scene is turned off.
- Inside the yellow border is 'Groups', which represents the relationship of the volume control of the music. Specifically, 'Master' controls the volume of all music. Whereas 'Ambient Master' and 'Music Master' only control their own volume under the subordinate 'Ambient' and 'Music' music volumes. The remaining 'Ambient', 'Music' and 'Effect Channel' record the original volume of the music.
- The blue box shows the range of the volume adjustment and the object to which the volume is to be adjusted.
- Inside the pink box are the 'Exposed Parameters', which are 'Master', 'Ambient' and 'Music'. In other words, they can be adjusted by the relevant scripts within the code, which is the code in Figure 122.

Secondly, after obtaining the 'Scene Name', the 'Ambient' and 'Music' that need to be played can be obtained by searching for a match with the 'Scene Name' stored in the 'Scene Sound Item' shown in Table 17. The 'Scene Name' stored in the 'Scene Sound Item' shown in Table 17 is searched for a match to be obtained. The 'Play Sound Routine' in Figure 122 is then called to play the two pieces of music. Moreover, the 'Play Music Clip' and the 'Play Ambient Clip' together form the 'Play Music Clip', which have similar logic. Therefore, only 'Play Music Clip' will be used as an example to explain the usage: when the sound details of 'Music' are read, its 'Sound Volume' is used to assign a value to 'Music Volume' in 'Audio Mixer', which is used for subsequent volume adjustments. Subsequently, if the 'Game Source' music player is active, the 'Sound Clip' of 'Music' will be read and played. Finally, the music settings for the whole scene should change to the parameters set in advance in 'Snapshot' of 'Normal'.

Finally, Figure 123 shows how to play frequent sound effects such as tree-cutting and stone-knocking: if the music needs to be played while the Player is using the tool, the relevant music can be played via a 'Call Play Sound Event', which accompanies the 'Sound Name' transmission, is obtained. Specifically, the 'On Play Sound Event' in Figure 123 will receive the 'Sound Name' data, and then look for the 'Sound Name' that has the same 'Sound Name'. Name', and then passes this value into Figure 120 using 'Call Init Sound Effect'. After this, the 'Sound Base' with the desired music can then be generated and played in the scene, which will be turned off after playing.

Problems Encountered:

```
private float ConvertSoundVolume(float amount) // Because soundVolune in Settings.cs is from 0 to 1,
                                               // while, in Audio Mixer, it is from -80 to 20
{
    return (amount * 100 - 80);
}
```

Figure 124-Code Snippet of Converting Sound Volume

Explanation:
The problem encountered in this section is that 'Sound Volume' in Table 18 is set to a value between 0 and 1, which does not match the volume range in 'Audio Mixer'. In detail, the volume range in 'Audio Mixer' is from -80 to 20, which is shown in Figure 121. Therefore, 'Sound Volume' needs a method to be converted to the volume range of 'Audio Mixer', which is shown in Figure 124.

2.21 Main Menu

Design:

Figure 125-The Initial Page of Menu UI

Figure 126-The Start Page of Menu UI with Three Save Slots

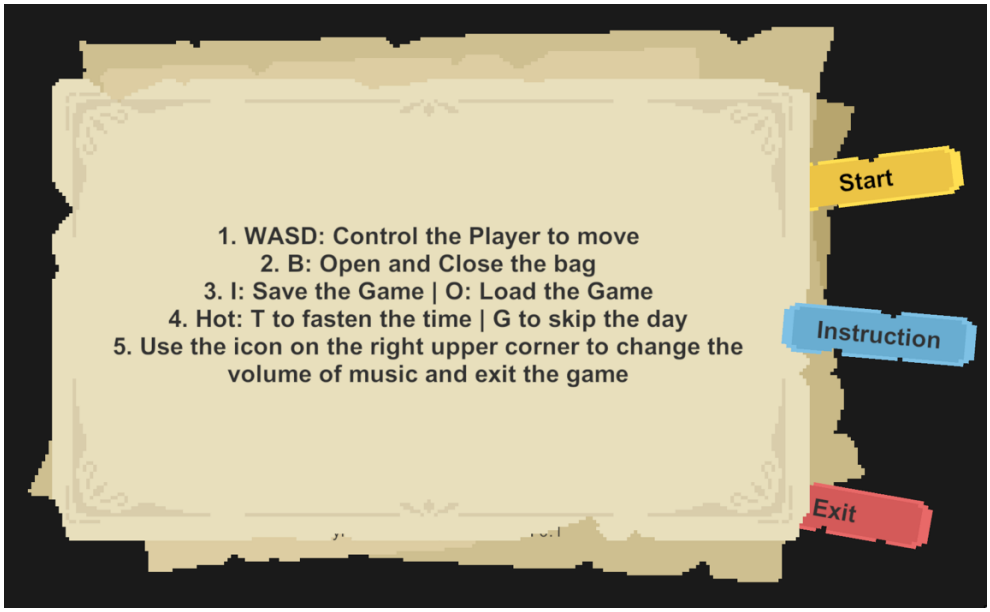


Figure 127-The Instruction Page of Menu UI

Explanation:

Firstly, Figure 125 shows the 'Menu' at the beginning, which consists of a title and three buttons. The title shows the name of the game, while the three buttons 'Start', 'Instruction' and 'Exit' are used to switch between different screens.

Secondly, Figure 126 shows the 'Menu' that appears when the 'Start' button is pressed. It has three separate slots for storing three copies of the game data. Specifically, when there is read game data in the slots, 'Wait for You' will be replaced with the time in the game, and 'Still in the Dream' will be replaced with the location of the character in the game record.

Then, when the 'Instruction' button is pressed, Figure 127's 'Menu' is displayed, which tells the player some basic actions of the game.

Finally, the game will end when the 'Exit' button is pressed.

Implementation:

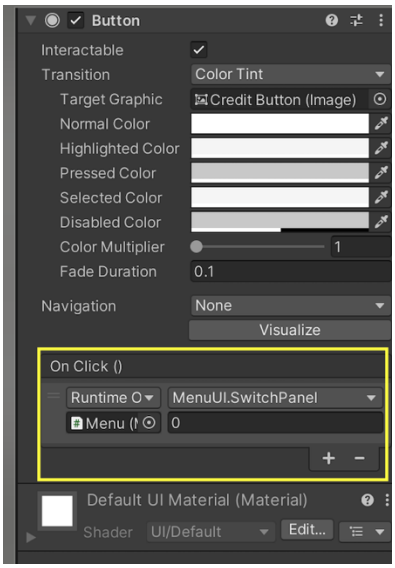


Figure 128-The Information of Unity Button Component

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class MenuUI : MonoBehaviour
{
    public GameObject[] panels;

    public void SwitchPanel(int index)
    {
        for (int i = 0; i < panels.Length; i++)
        {
            if (i == index)
            {
                panels[i].transform.SetAsLastSibling();
            }
        }
    }

    public void ExitGame()
    {
        Application.Quit();
        // Debug.Log("EXIT GAME");
    }
}
```

Figure 129-Code Snippet of Switching Panel UI

```
private void SetupSlotUI()
{
    currentData = SaveLoadManager.Instance.dataSlots[Index];

    if(currentData != null)
    {
        dataTime.text = currentData.DataTime;
        dataScene.text = currentData.DataScene;
    }
    else
    {
        dataTime.text = "Wait For You";
        dataScene.text = "Still in the Dream";
    }
}
```

Figure 130-Code Snippet of Set Up Slot UI for Save Slot

Explanation:

Firstly, important information about 'Button' in Unity is shown in Figure 128. The most important of these is the 'On Click' event in the yellow box, which can be used to activate methods in a particular script. Specifically, to switch the UI from Figure 125 to Figure 127, the three 'Buttons' need to switch the order in which the different 'Menu' pages are displayed in Unity when pressed. Based on this logic, the code for the final implementation is shown in Figure 129: when 'Start' and 'Instruction' are pressed, their own 'Index ' is then passed into the code in Figure 129. The 'Switch Panel' then retrieves the index set in advance within the 'Button', which is shown in the yellow box in Figure 128. Then, when the corresponding 'Menu' page is retrieved and matched, the 'Switch Panel' moves it to the 'Hierarchy' of the entire 'Menu' component. The 'Switch Panel' will then be moved to the bottom of the 'Hierarchy' of the entire 'Menu' component, which will be rendered first and then displayed in Unity. Finally, when the 'Exit' button is pressed, 'Exit Game' in Figure 129 is called and the game is then ended.

In the meantime, the 'Save Slot' in Figure 126 needs to be set separately, which is like the code shown in Figure 130. Specifically, the game data is read into the 'Save Load Manager', which is a referred way of storing games from [22], which will be put into Appendix. Then assign values to 'Data Time Text' and 'Data Scene Text' in 'Save Slot'. If there is no value, then perform the default assignment in 'else'.

Problems Encountered:

Explanation:

Since Unity integrates a lot of UI methods, which are all as concise and effective as shown in 'Implementation', there are no problems being encountered in this section.

2.22 Settings Menu

Design:

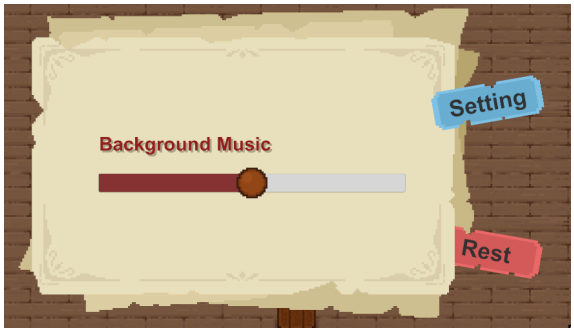


Figure 131-The Setting Page of Settings Menu

Figure 132-The Rest Page of Settings Menu

Explanation:

Figure 131 shows what the 'Settings Menu' looks like when it is first opened, which contains a 'Slider' and two 'Buttons'. Specifically, the 'Slider' is created to control the volume, which is read by the 'Audio Manager' in section 2.20 and then changes the volume. Meanwhile, the 'Setting' button is pressed to show this initial screen, and the 'Rest' button is pressed to show the screen in Figure 132, which contains two new 'Buttons', namely 'Back to Main Menu' and 'Exit Game'. In this case, when the 'Back to Main Menu' button is down, the 'Main Menu' in section 2.21 is expected to be displayed, while the 'Exit Game' button is to close the game directly.

Implementation:

Figure 133-The Settings Button in Time UI

```
private void Awake()
{
    settingsBtn.onClick.AddListener(TogglePausePanel);
    volumeSlider.onValueChanged.AddListener(AudioManager.Instance.SetMasterVolume);
}
```

Figure 134-Code Snippet of Adding Listener of UI Manager

```
private void TogglePausePanel()
{
    bool isOpen = pausePanel.activeInHierarchy;

    if (isOpen)
    {
        pausePanel.SetActive(false);
        Time.timeScale = 1;
    }
    else
    {
        System.GC.Collect(); // GC stands for Garbage Collection, which is used to clear the system garbage to avoid potential crashes
        pausePanel.SetActive(true);
        Time.timeScale = 0;
    }
}
```

Figure 135-Code Snippet of Toggle Settings Menu

```
public void SetMasterVolume(float value)
{
    audioMixer.SetFloat("MasterVolume", (value * 100 - 80));
}
```

Figure 136-Code Snippet of Setting Master Volume

```
public void ReturnMenuCanvas()
{
    Time.timeScale = 1;
    StartCoroutine(BackToMenu());
}

private IEnumerator BackToMenu()
{
    pausePanel.SetActive(false);
    EventHandler.CallEndGameEvent();
    yield return new WaitForSeconds(1f);
    Instantiate(menuPrefab, menuCanvas.transform);
}
```

Figure 137-Code Snippet of Returning to Main Menu

Explanation:

Firstly, the 'Settings Menu' will be activated by the button in Figure 133. But since the logic of 'Switch Panel' and 'Exit Game' is the same as in part 2.21, this part focuses on the volume control and the 'Back to Main Menu' logic and implementation.

Hence, in the code for the 'UI Manager' in Figure 134, add listeners to the 'Settings Button' and 'Volume Slider' to enable the next step when activated. Although the 'Button' can be activated by adding an 'On Click' as in section 2.21, it is added here for both for ease of management.

Subsequently, when 'Settings Button' is activated, the code in Figure 135 is activated. In detail, when the 'Settings Button' is pressed, the 'Toggle Pause Panel' determines whether the 'Settings Menu 'Settings Menu' is opened or not. If it is opened, it closes the 'Settings Menu' and restores the time flow to normal; otherwise, it opens the 'Settings Menu' and pauses the time. The 'System.GC.Collect' is used to clean up the cache and has no real meaning.

At the same time, the code in Figure 136 is activated when the value of 'Slider' changes. Specifically, when the value of 'Slider' is artificially changed, its value is transferred to the 'Set Master Volume' code. 'Set Master Volume' then actively adjusts the volume level in the 'Audio Mixer', as demonstrated in section 2.20.

Finally, the 'Return Menu Canvas' in Figure 137 will be added to the 'On Click' of 'Back to Main Menu'. Specifically, when the button is triggered, the time flow returns to normal. The 'Settings Menu' is then closed. After this, the 'Call End Game Event' is called to allow the rest of the game to process the information about the game closing. In the end, the 'Main Menu' is generated and displayed to the player.

Problems Encountered:

Explanation:

Since Unity integrates a lot of UI methods, which are all as concise and effective as shown in 'Implementation', there are no problems being encountered in this section except for the extensive data processing for 'Call End Game Event' in Figure 137.

3. Testing & Evaluation

Although this thesis includes many different parts of this project in the Design & Implementation section, in fact, all the parts are directly or indirectly fulfilling the requirements in the Introduction section. For example, the creation of 'Item Data' in section 2.7 is also providing a base platform for 'Bag' in sections 2.9 and 2.10. Therefore, in this section, the initial nine requirements of this project are tested, which indirectly tests the corresponding components as well.

3.1 Digital Bag

Test:

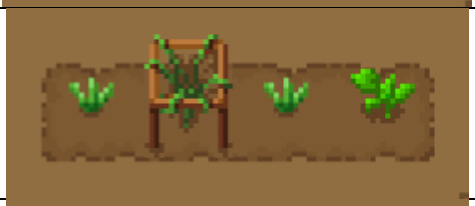
Table 19-Unit Test and Result of Bag System

Test Case	***Explanation***	***Expected Outcome***	***Actual Outcome***
UI Display	Bag UI can display correctly	It should not include any item before storing anything	 
UI Open and Close	Bag UI can be opened and closed	After pressing ‘B’ key, the bag should be opened or closed	Successful
Item Storage	Bag can store items	Bag can store items in each slot	
Item Drag and Drop	Item can be dragged and dropped to change slot	The items in bag can change their slots between each other	
Item Addition	The same items picked up will be added to the existed items in Bag	The amount of the item will be added to the same item in Bag	 The original amount of potatoe seeds is 20  After buying extra 20, there will be 40. (The amount is added to the same item in Bag)
Item Removal	The item can be dropped if wanted	The item that is assigned ‘can Drop’ can be dropped	

3.2 Seed Growth Process

Test:

Table 20-Test of Seed Growth

Stage/ Test Number	*Stage Snapshot*	*Explanation*
1		The state of the seed when it was first planted
2		The state of the seed when it is planted and grows to the first stage
3		The state of the seed when it is planted and grows to the second stage
4		The state of the seed when it is planted and grows to the third stage
5		The state of the seed when it is planted and grows to the fourth stage
6		The state in which the seed is finally harvested

3.3 Constructing Buildings

Test:

In the end it was not realized.

3.4 Communicating with Non-Player Characters

Test:

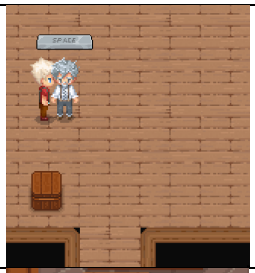
Table 21-Unit Test of Communication

Test Case	***Explanation***	***Expected Outcome***	***Actual Outcome***
UI Display	When the Player get close to NPCs, a UI sign should be displayed	A big 'Space' key should be displayed on the NPC's head	
Dialogue Display	When the communication starts, the dialogue should be displayed	Dialogue UI and Text should be displayed	
Dialogue Continue	When the dialogue starts, it should be continued with only expected key being pressed	Only when the space key is pressed, the dialogue continues	
Player Move	The Player should not move during communication	The Player cannot move	Successful

3.5 Daily Routine for Non-Player Characters

Test:

Table 22-Unit Test of NPC Schedules in a Day

Test Case	***Explanation***	***Expected Outcome***	***Actual Outcome***
Morning, the old man enters the house	The old man should enter the house in the morning	After reaching the expected position, a UI sign should display on the head of the old man	
Morning, the old man goes to the farm	After entering the house, the old man should go back to farm in the morning		

Night, the old man enters the house	At night, the old man should go back to the house		
Morning and Night, the girl stands in the north of farm	The girl should stand in the north of arm in the morning and at night	After reaching the expected position, a UI sign should display on the head of the girl	

3.6 Trade System

Test:

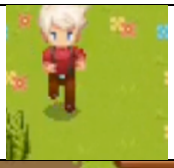
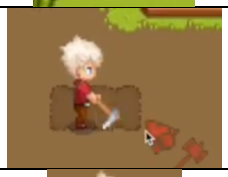
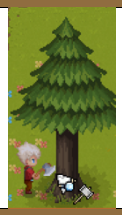
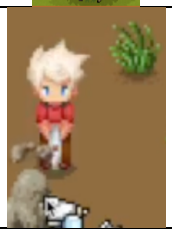
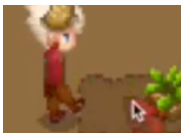
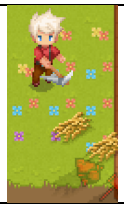
Table 23-Unit Test of Trade System

Test Case	***Explanation***	***Expected Outcome***	***Actual Outcome***
Sell Items	The function of selling things	Get the money and the related items in Bag will be deleted	 Original Money is 500 and the grape seed is 10 per one
Buy Items	The function of buying things	Get the related items and the money will be subtracted	 After selling 100 woods, which can be sold at 5 per one. The money is then 1000

3.7 Animation

Test:

Table 24-Unity Test of Animation

Test Case	***Explanation***	***Expected Outcome***	***Actual Outcome***
Walking	The animation of player movement	The animation of walking should be displayed	
Digging	The animation of digging	The animation of digging should be displayed	
Watering	The animation of watering	The animation of watering should be displayed	
Cutting Down Tree	The animation of cutting down trees	The animation of cutting down trees should be displayed	
Cutting Rock	The animation of cutting rocks	The animation of cutting rocks should be displayed	
Collecting Fruit	The animation of collecting fruits	The animation of collecting fruits should be displayed	
Reaping Grass	The animation of reaping grass	The animation of reaping grass should be displayed	
Tree Falling Down	The animation of falling	The animation of falling should be displayed when the tree is cut down	

3.8 Music Playback

Test:

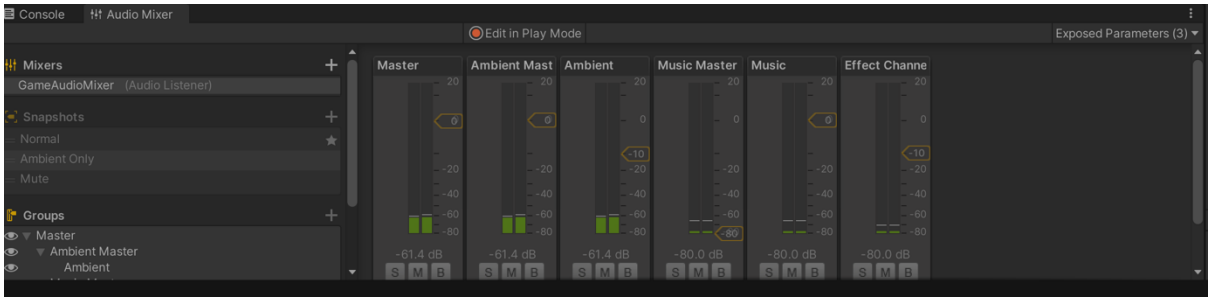


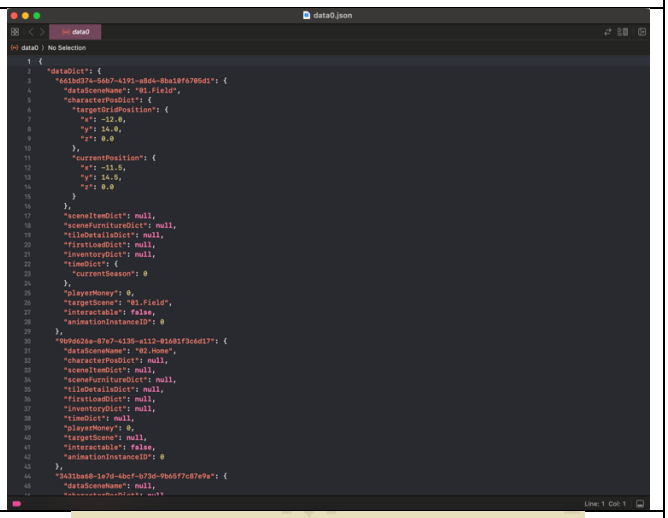
Figure 138-Test of Music Playback

Explanation:
As shown in Figure 138, the music playback works correctly.

3.9 Save & Load Game Data

Test:

Table 25-Unity Test of Game Save & Load Function

Test Case	***Explanation***	***Expected Outcome***	***Actual Outcome***
Save Game	The game data can be saved into a file	There is a file that records the current game state	
Load Game	The game data can be read from a file	The game can start from the previous stage that was saved before	

3.10 Overview Evaluation

From Table 19 to Table 25, together with Figure 138, the realisations of the nine requirements in the Introduction section are shown. Overall, all the completed features were realised as expected, except for the 'Constructing Buildings' section, which there was no time to complete. Specifically, the 'Constructing Buildings' section does not share common parts with the other requirements, it is a completely new system which is so complex that there was no time to complete it.

To summarise, the project still met a significant proportion of the aims and requirements, although some of the features are not completed. But considering the constraints of time and other factors, this project largely was successful.

If there was more time to continue with the project, the 'Constructing Buildings' section would have been expected to be completed, as its completion would have made the game more immersive and colourful.

4. Project Ethics

This project genuinely has no ethnic considerations because this project is still in the beginning stage of a game, which does not need any human participant to test it. Meanwhile, for the game assets and music files got from online, they are under CC0 creative commons license, which means they can be used for any objectives without limitation under copyright or database law [23]. Therefore, Data and Human Participant Categories of this project is A0.

5. Conclusion & Future Work

The purpose of this project is to expand somewhat on the gameplay of the classic farm game Stardew Valley. However, if there is not enough time, it is acceptable to complete a partial reproduction of the gameplay.

Therefore, in this project, the nine requirements in the Implementation were basically implemented through the creation of data structures for the different items, as well as the design of the UI, and finally the implementation of the interaction with code. Specifically, the basic interaction logic is first implemented through the creation of Player, Map and Camera. Then the Bag, Trade, and Farm Crops were realized through the creation of Item Data, including Seeds, Meanwhile, the Time System was built to allow NPCs to have their daily routines and crop growth process revealed. Finally, the Dialogue, Music, and Game Storage are used to give the project the basic look of a game.

However, the game is not simply a functional implementation. In other words, even if most of the aims and requirements are implemented in this project, as stated in Section 3, the final product of this project cannot be considered a good game. This is because it only has most of the features and gameplay that the game has and does not have expansions based on those features and gameplay, which is what the author, as a programmer, lacks.

Hence, the direction of future development of this project should be placed on the expansion of the game's own system, rather than singularly adding more gameplay. For example, more maps can be designed for players to explore freely. Additionally, more crops could also be created for players to interact with, which would require a professional game artist.

To conclude, considering the time constraints, the program was a success. But if it is treated as a game, then it is inadequate. Most of the features are implemented in this project, but to treat it as a game, it should continue to expand the game itself in the future, not more gameplay.

6. BCS Project Criteria & Self-Reflection

1. **An ability to apply practical and analytical skills gained during the degree programme.**

   Answer: this project is a summary of my studies throughout my degree program. Specifically, I have learned from my programme the important theoretical knowledge of computer science such as basic games, software engineering, data structures and the corresponding analytical skills, which were applied in the design, development and finally the methodology illustrated in this dissertation.

2. **Innovation and/or creativity.**

   Answer: although this project is a recreation of similar features of Stardew Valley, it does not mean that there is no need for innovation and creativity. In fact, the creation of the game's data structures, the design of the UI, and the implementation of the code are all innovative, which are the core of this project.

3. **Synthesis of information, ideas, and practices to provide a quality solution together with an evaluation of that solution.**

   Answer: through this project, off-the-shelf game assets, music, and possible existed methods such as Astar algorithm are combined to form a foundation of a game. Then, after updated, tested, and applied knowledge to from similar problems from some platforms, all the resources are finally turned into a basic game, which has been shown in Section 'Design & Implementation.' Additionally, an evaluation of my final year project has been shown in Section 'Testing & Evaluation.'

4. **That your project meets a real need in a wider context.**

   Answer: The primary purpose of my program is for the entertainment of all. But it can still be used for basic education on farm economics. For example, it can be used to learn how to grow different crops in different seasons to maximize financial resources. However, the main purpose is still entertainment.

5&6. An ability to self-manage a significant piece of work and a critical self-evaluation of the process.

Answer: First, for a significant piece of work, I think my greatest personal strength is being proactive. I would not put anything on this project to do on the days before the deadline. Specifically, whether it is Detailed Proposal, Introduction Video, or Final Dissertation, I always finish it at least half a month in advance. Then, I spend the rest of the day polishing my work until I am personally satisfied with it. But that does not mean that the first version of the work is haphazard. On a related note, even the first work is carefully thought out before it is made. For example, for the Introduction Video in this project, even the first, relatively rough version was the result of hours of video recording and editing. Hence, I have always been able to refine my projects to

the point where I am at least satisfied with them, after which I can believe that others will be also satisfied with my work.
Secondly, I am very good at breaking a big project into small, related parts and then completing them one step at a time. As shown in 'Design & Implementation', this project was broken down by me into dozens of different small modules. Only when the previous basic modules are completed will the later, advanced versions of the modules begin. In this way, even though the overall project is very difficult, I rarely experience the despair of not being able to finish a big project as I continue to complete the modules one by one. However, there have been modules that have been incorrectly split up, but they have also, at best, made me feel like they take a lot of time and don't make me feel like I can't do them. Therefore, this is the main thing that keeps my projects making healthy progress.
After that, I am very good at finding relevant information. For some Unity errors and problems, the information can be searched very quickly, and then I can find the solution accordingly.
Meanwhile, I am also very good at finding information. For some Unity errors and problems, the information can be searched very quickly, and then I can find the solution accordingly. For example, in each of the 'Problems Encountered' sections in 'Design & Implementation', I have explained how I solved the problems encountered, which were largely triggered by the discussions of the predecessors in the related game development forum (for discussion). There are, however, some issues that have not been discussed, which have been encountered very little, or perhaps no one has encountered them yet. In this case, the official documentations need to be consulted, even though they are boring and long.
However, I get anxious easily when I do not have a first draft of a project, which brings me the first advantage, but also this one disadvantage as well. Then, because of the anxiety, I would keep thinking about a problem until I figured it out. For example, when I had problems with data transfer in section 2.10, I thought about it for almost a night. I had to come up with a solution on my own because there was very little information on the subject. Although the problem was solved in the end, I did not have a good rest during the night, and the progress of the project was very slow for the next few days. Therefore, I know that this habit of mine can be very bad when it comes to bigger projects, but this project has made me realize this problem and I am making a deliberate effort to change it.
In addition to this, I have the disadvantage that for a period, I will only do one thing. In other words, while I am doing this program, I am putting off almost all my lectures and tutorials in other subjects. Because of this, a lot of classes I need extra time to make up for what I have left behind, which aggravates my academic load, especially when the project still has no progress even under this situation.
To conclude, this program made me realize my strengths and weaknesses. At the same time, even if it did not meet all the aims and requirements, this project is still the work I am most proud of.

References

[1] W. Evans, *2 - Types of virtual worlds*. Oxford, United Kingdom: Chandos Publishing, 2011, doi: 10.1016/B978-1-84334-641-8.50002-3.

[2] B. Edwards. "The History of Civilization." *Game Developer*. Accessed: Oct. 10, 2023. [Online]. Available: https://www.gamedeveloper.com/design/the-history-of-civilization#closemodal

[3] A. Line. "Stardew Valley: Pushing The Boundaries of Farming RPGs." *The Cornell Daily Sun*. Accessed: Oct. 10, 2023. [Online]. Available: https://cornellsun.com/2016/02/23/stardew-valley-pushing-theboundaries-of-farming-rpgs/

[4] Game Dev with JacquelynneHei, *How to make a 2D farming RPG*, (Jan. 24, 2022). Accessed Date: Oct. 11, 2023. [Online Video]. Available: https://www.youtube.com/watch?v=ZPYrdKMDsGI&list=PL4PNgDjMajPN51E5WzEi7cXzJ16BCHZXl&index=1

[5] Unity. "Scripting API." *Unity Documentation*. Accessed: Oct. 11, 2023. [Online]. Available: https://docs.unity3d.com/2022.3/Documentation/ScriptReference/Accessibility.VisionUtility.html

[6] Unity. *Unity Assets Store.* Accessed: Oct. 8, 2023. [Online]. Available: Unity Asset Store - The Best Assets for Game Making

[7] "Stardew Valley." *Stardew Valley*. Accessed: April. 8, 2024. [Online]. Available: https://www.stardewvalley.net

[8] Unity. "2D Sorting." *Unity Documentation*. Accessed: April. 8, 2024. [Online]. Available: https://docs.unity3d.com/Manual/2DSorting.html

[9] Unity. "Sorting Group." *Unity Documentation*. Accessed: April. 8, 2024. [Online]. Available: https://docs.unity3d.com/Manual/class-SortingGroup.html

[10] Unity. "Rigidbody2D." *Unity Documentation*. Accessed: April. 8, 2024. [Online]. Available: https://docs.unity3d.com/ScriptReference/Rigidbody2D.html

[11] Unity. "Creating a Tile Palette." *Unity Documentation*. Accessed: Nov. 12, 2023. [Online]. Available: https://docs.unity3d.com/Manual/Tilemap-Palette.html

[12] Unity Technologies. "Using Rule Tiles." *Unity Learn*. Accessed: Nov. 12, 2023. [Online]. Available: https://learn.unity.com/tutorial/using-rule-tiles

[13] Ethan Bruins, "Cinemachine for 2D: Tips and tricks," Unity Blog, https://blog.unity.com/engine-platform/cinemachine-2d-tips-and-tricks (accessed November 13, 2023).

[14] Unity. "Class CinemachineConfiner." *Scripting API.* Accessed: Nov. 13, 2023. [Online]. Available: https://docs.unity3d.com/Packages/com.unity.cinemachine@2.1/api/Cinemachine.CinemachineConfiner.html

[15] Unity. "Animator component." *Unity Documentation*. Accessed: Nov. 15, 2023. [Online]. Available: https://docs.unity3d.com/Manual/class-Animator.html

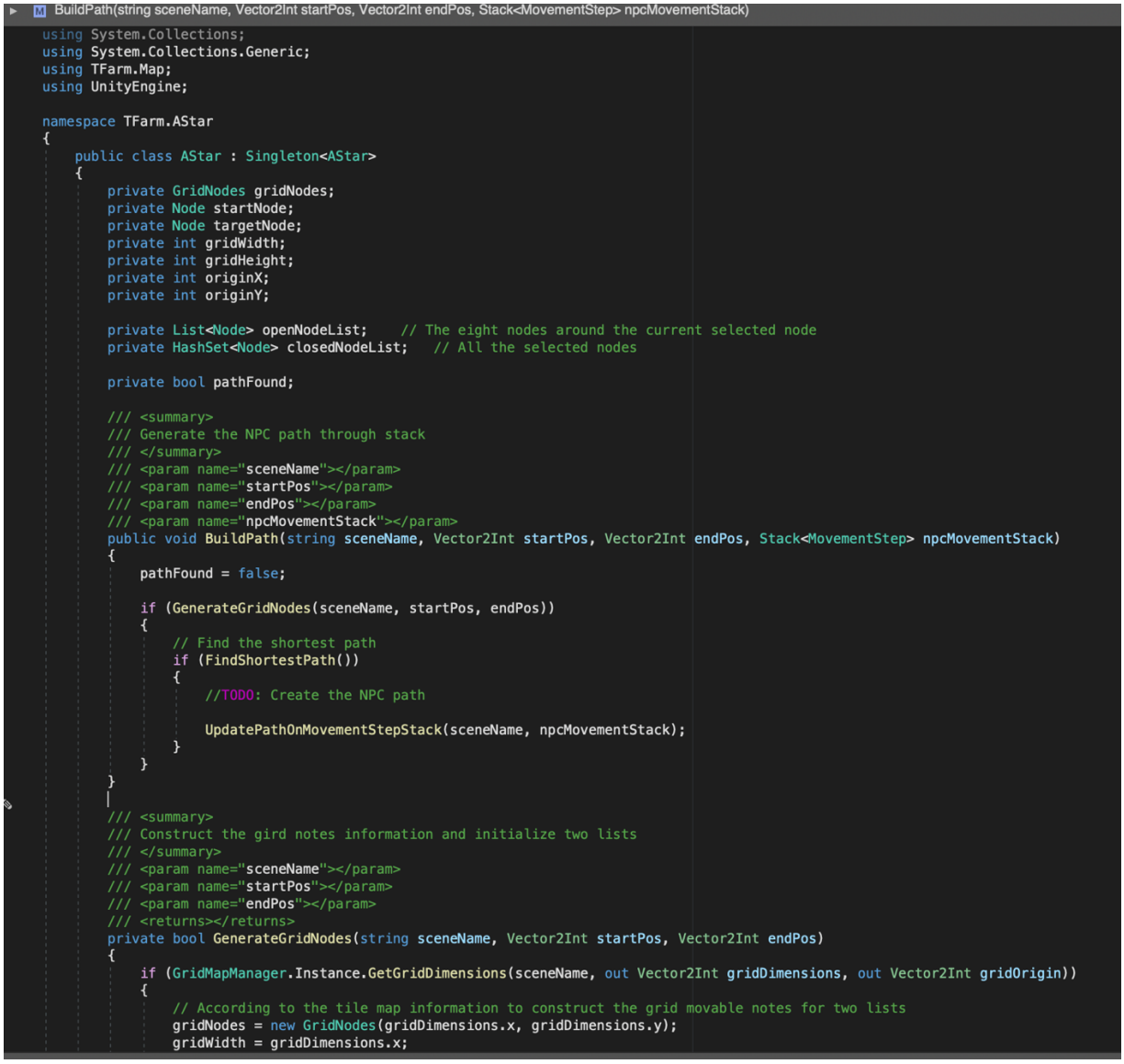
[16] DEMIGIANT. "DOTween (HOTWeen v2)." *DOTween*. Accessed: Nov. 15, 2023. [Online]. Available: https://dotween.demigiant.com

[17] Unity. "MonoBehavior.OnTriggerEnter2D (Collider 2D)." *Unity Documentation*. Accessed: Nov. 17, 2023. [Online]. Available: https://docs.unity3d.com/ScriptReference/MonoBehaviour.OnTriggerEnter2D.html

[18] D. Brown. “Draggable UI In Unity.” *Medium*. Accessed: Dec. 3, 2023. [Online]. Available: https://medium.com/@dnwesdman/draggable-ui-in-unity-21fdcbc77bd1
[19] Unity. “Blend Trees.” *Unity Documentation*. Accessed: Dec. 5, 2023. [Online]. Available: https://docs.unity3d.com/Manual/class-BlendTree.html
[20] Greg Dev Stuff, *Grid Map system in Unity Episode 3 Pathfinding*, (Jun. 9, 2021). Accessed: Jan. 25, 2024. [Online Video]. Available: https://www.youtube.com/watch?v=ajUvzwKbWj8
[21] “Free Sound.” *Free Sound*. Accessed: Mar. 10, 2024. [Online]. Available: https://freesound.org
[22] Shaped by Rain Studios, *How to make a Save & Load System in Unity | 2022*, (Mar. 9, 2022). Accessed: April 6, 2024. [Online Video]. Available: https://www.youtube.com/watch?v=aUi9aijvpgs
[23] “CC0.” *Creative Cmmons*, Accessed: April 19, 2024. [Online]. Available: https://creativecommons.org/public-domain/cc0/

Appendix

The figures below show the essential codes in this project but not developed in this project, which are referred from online tutorials and then updated into this project:



BuildPath(string sceneName, Vector2Int startPos, Vector2Int endPos, Stack<MovementStep> npcMovementStack)

```
using System.Collections;
using System.Collections.Generic;
using TFarm.Map;
using UnityEngine;

namespace TFarm.AStar
{
    public class AStar : Singleton<AStar>
    {
        private GridNodes gridNodes;
        private Node startNode;
        private Node targetNode;
        private int gridWidth;
        private int gridHeight;
        private int originX;
        private int originY;

        private List<Node> openNodeList;    // The eight nodes around the current selected node
        private HashSet<Node> closedNodeList;   // All the selected nodes

        private bool pathFound;

        /// <summary>
        /// Generate the NPC path through stack
        /// </summary>
        /// <param name="sceneName"></param>
        /// <param name="startPos"></param>
        /// <param name="endPos"></param>
        /// <param name="npcMovementStack"></param>
        public void BuildPath(string sceneName, Vector2Int startPos, Vector2Int endPos, Stack<MovementStep> npcMovementStack)
        {
            pathFound = false;

            if (GenerateGridNodes(sceneName, startPos, endPos))
            {
                // Find the shortest path
                if (FindShortestPath())
                {
                    //TODO: Create the NPC path

                    UpdatePathOnMovementStepStack(sceneName, npcMovementStack);
                }
            }
        }

        /// <summary>
        /// Construct the gird notes information and initialize two lists
        /// </summary>
        /// <param name="sceneName"></param>
        /// <param name="startPos"></param>
        /// <param name="endPos"></param>
        /// <returns></returns>
        private bool GenerateGridNodes(string sceneName, Vector2Int startPos, Vector2Int endPos)
        {
            if (GridMapManager.Instance.GetGridDimensions(sceneName, out Vector2Int gridDimensions, out Vector2Int gridOrigin))
            {
                // According to the tile map information to construct the grid movable notes for two lists
                gridNodes = new GridNodes(gridDimensions.x, gridDimensions.y);
                gridWidth = gridDimensions.x;
```

Figure 139-Code Snippet of AStar (Part 1)

```
    /// <summary>
    /// Construct the gird notes information and initialize two lists
    /// </summary>
    /// <param name="sceneName"></param>
    /// <param name="startPos"></param>
    /// <param name="endPos"></param>
    /// <returns></returns>
    private bool GenerateGridNodes(string sceneName, Vector2Int startPos, Vector2Int endPos)
    {
        if (GridMapManager.Instance.GetGridDimensions(sceneName, out Vector2Int gridDimensions, out Vector2Int gridOrigin))
        {
            // According to the tile map information to construct the grid movable notes for two lists
            gridNodes = new GridNodes(gridDimensions.x, gridDimensions.y);
            gridWidth = gridDimensions.x;
            gridHeight = gridDimensions.y;
            originX = gridOrigin.x;
            originY = gridOrigin.y;

            openNodeList = new List<Node>();

            closedNodeList = new HashSet<Node>();
        }
        else
            return false;

        // The gridnotes has values from (0,0), therefore we need to subtract it with the origin node to get the realistic positions nodes information
        startNode = gridNodes.GetGridNode(startPos.x - originX, startPos.y - originY);
        targetNode = gridNodes.GetGridNode(endPos.x - originX, endPos.y - originY);

        for (int x = 0; x < gridWidth; x++)
        {
            for (int y = 0; y < gridHeight; y++)
            {
                Vector3Int tilePos = new Vector3Int(x + originX, y + originY, 0);

                var key = tilePos.x + "x" + tilePos.y + "y" + sceneName;

                TileDetails tile = GridMapManager.Instance.GetTileDetails(key);

                if (tile != null)
                {
                    Node node = gridNodes.GetGridNode(x, y);

                    if (tile.isNPCObstacle)
                        node.isObstacle = true;
                }
            }
        }

        return true;
    }
```

Figure 140-Code Snippet of AStar (Part 2)

```
private bool FindShortestPath()
{
    // Add the start node
    openNodeList.Add(startNode);

    while (openNodeList.Count > 0)
    {   // Sort the node list
        openNodeList.Sort();

        Node closeNode = openNodeList[0];

        openNodeList.RemoveAt(0);
        closedNodeList.Add(closeNode);

        if (closeNode == targetNode)
        {
            pathFound = true;
            break;
        }

        //TODO: Calculate the around eight nodes distance and add them to the open list
        EvaluateNeighbourNodes(closeNode);
    }

    return pathFound;
}

/// <summary>
/// Evaluate the around 8 nodes and get the value
/// </summary>
/// <param name="currentNode"></param>
private void EvaluateNeighbourNodes(Node currentNode)
{
    Vector2Int currentNodePos = currentNode.gridPosition;
    Node validNeighbourNode;

    for (int x = -1; x <= 1; x++)
    {
        for (int y = -1; y <= 1; y++)
        {
            if (x == 0 && y == 0)
                continue;

            validNeighbourNode = GetValidNeighbourNode(currentNodePos.x + x, currentNodePos.y + y);

            if (validNeighbourNode != null)
            {
                if (!openNodeList.Contains(validNeighbourNode))
                {
                    validNeighbourNode.gCost = currentNode.gCost + GetDistance(currentNode, validNeighbourNode);
                    validNeighbourNode.hCost = GetDistance(validNeighbourNode, targetNode);
                    // Parent node
                    validNeighbourNode.parentNode = currentNode;
                    openNodeList.Add(validNeighbourNode);
                }
            }
        }
    }
}
```

Figure 141-Code Snippet of AStar (Part 3)

```
        /// <summary>
        /// Find Valid Nodes, which are not obstacle and not selected
        /// </summary>
        /// <param name="x"></param>
        /// <param name="y"></param>
        /// <returns></returns>
        private Node GetValidNeighbourNode(int x, int y)
        {
            if (x >= gridWidth || y >= gridHeight || x < 0 || y < 0)
                return null;

            Node neighbourNode = gridNodes.GetGridNode(x, y);

            if (neighbourNode.isObstacle || closedNodeList.Contains(neighbourNode))
                return null;
            else
                return neighbourNode;
        }


        /// <summary>
        /// Return the distance value between two nodes
        /// </summary>
        /// <param name="nodeA"></param>
        /// <param name="nodeB"></param>
        /// <returns>14n+10n</returns>
        private int GetDistance(Node nodeA, Node nodeB)
        {
            int xDistance = Mathf.Abs(nodeA.gridPosition.x - nodeB.gridPosition.x);
            int yDistance = Mathf.Abs(nodeA.gridPosition.y - nodeB.gridPosition.y);

            if (xDistance > yDistance)
            {
                return 14 * yDistance + 10 * (xDistance - yDistance);
            }
            return 14 * xDistance + 10 * (yDistance - xDistance);
        }

        /// <summary>
        /// Update the sceneName and grid position (in coordinate) for each step
        /// </summary>
        /// <param name="sceneName"></param>
        /// <param name="npcMovementStep"></param>
        private void UpdatePathOnMovementStepStack(string sceneName, Stack<MovementStep> npcMovementStep)
        {
            Node nextNode = targetNode;

            while(nextNode != null)
            {
                MovementStep newStep = new MovementStep();
                newStep.sceneName = sceneName;
                newStep.gridCoordinate = new Vector2Int(nextNode.gridPosition.x + originX, nextNode.gridPosition.y + originY);
                // push into the stack

                npcMovementStep.Push(newStep);
                nextNode = nextNode.parentNode;
            }
        }
    }
}
```

Figure 142-Code Snippet of AStar (Part 4)

```
using System.Collections;
using System.Collections.Generic;
using System.IO;
using Newtonsoft.Json;
using UnityEngine;

namespace TFarm.Save
{
    public class SaveLoadManager : Singleton<SaveLoadManager>
    {
        private List<ISaveable> saveableList = new List<ISaveable>();

        public List<DataSlot> dataSlots = new List<DataSlot>(new DataSlot[3]);

        private string jsonFolder;
        private int currentDataIndex;

        protected override void Awake()
        {
            base.Awake();
            jsonFolder = "/Users/tevenji/Desktop/FYP Jason Data" + "/SAVE DATA/";
            ReadSaveData();
        }

        private void OnEnable()
        {
            EventHandler.StartNewGameEvent += OnStartNewGameEvent;
            EventHandler.EndGameEvent += OnEndGameEvent;
        }

        private void OnDisable()
        {
            EventHandler.StartNewGameEvent -= OnStartNewGameEvent;
            EventHandler.EndGameEvent -= OnEndGameEvent;
        }


        private void Update()
        {
            if (Input.GetKeyDown(KeyCode.I))
                Save(currentDataIndex);
            if (Input.GetKeyDown(KeyCode.O))
                Load(currentDataIndex);
        }


        private void OnEndGameEvent()
        {
            Save(currentDataIndex);
        }

        private void OnStartNewGameEvent(int index)
        {
            currentDataIndex = index;
        }

        public void RegisterSaveable(ISaveable saveable)
        {
            if (!saveableList.Contains(saveable))
                saveableList.Add(saveable);
        }
```

Figure 143-Code Snippet of Save & Load Manager (Part 1)

```
    private void ReadSaveData()
    {
        if (Directory.Exists(jsonFolder))
        {
            for (int i = 0; i < dataSlots.Count; i++)
            {
                var resultPath = jsonFolder + "data" + i + ".json";
                if (File.Exists(resultPath))
                {
                    var stringData = File.ReadAllText(resultPath);
                    var jsonData = JsonConvert.DeserializeObject<DataSlot>(stringData);
                    dataSlots[i] = jsonData;
                }
            }
        }
    }


    private void Save(int index)
    {
        DataSlot data = new DataSlot();

        foreach (var saveable in saveableList)
        {
            data.dataDict.Add(saveable.GUID, saveable.GenerateSaveData());
        }

        dataSlots[index] = data;

        var resultPath = jsonFolder + "data" + index + ".json";

        var jsonData = JsonConvert.SerializeObject(dataSlots[index], Formatting.Indented); // One line after one line (indented)

        if (!File.Exists(resultPath)) // It exists the same file name file
        {
            Directory.CreateDirectory(jsonFolder);
        }
        Debug.Log("DATA" + index + "SAVED!");
        File.WriteAllText(resultPath, jsonData);
    }

    public void Load(int index)
    {
        currentDataIndex = index;

        var resultPath = jsonFolder + "data" + index + ".json";

        var stringData = File.ReadAllText(resultPath);

        var jsonData = JsonConvert.DeserializeObject<DataSlot>(stringData);

        foreach (var saveable in saveableList)
        {
            saveable.RestoreData(jsonData.dataDict[saveable.GUID]);
        }
    }
}
```

Figure 144-Code Snippet of Save & Load Manager (Part 2)