FUN FARM

Hồ Việt Bảo Long - 21125071
Thi Hồng Nhựt - 21125053
Trần Minh Khánh - 21125070
Vương Quang Việt Tùng - 21125145

1. INTRODUCTION

In a village, a farmer was living peacefully with his farm and house. One day, he discovered his land was under siege by hordes of mischievous slimes and relentless skeletons. To protect his home, the farmer armed himself with weapons and decided to fight back against the enemies.

2. GAMEPLAY

In this game, players will play as the farmer, who has the following missions:

- Grow the plants to collect products
- Protect the plants from enemies
- Gain money from selling products
- Use money to upgrade the items (weapons, plants, lands, etc)

Enemies of players in this game are skeletons and slimes

- They appear after a certain time
- They attack and kill our plants

HOW TO PLAY?

01.

Try to plant as many plants on your field. You can enlarge your plantable land by purchasing more pieces of land.

02.

Take care of your plants. Use your water reasonably because water is not unlimited.

03.

You have to take care of yourself: Buy food in the village or you'll die.

04.

You have to look out for your enemies: Slimes during the daytime and Skeletons at night.

HOW TO PLAY?

MOVE

Press Arrow keys

WATER

Press W (while there is enough water)

DEFEND

Press D (while there are enough fences)

ATTACK

Press Space

SWITCH WEAPON

Press Tab (if a gun is owned)

3. FEATURES

MAIN FEATURES

- The game has 2 modes: **Survival** and **Creative**
- Interactive menus: Main menu, Pause Menu, etc.
- Control system with keyboard and mouse
- Have background music, sound effects
- Animations
- Minimap
- Plant plants, water them before they get deteriorated, and harvest when they are ready.

GAME MODES

Survival

Creative

No health

No enemies

No village scene

No defense

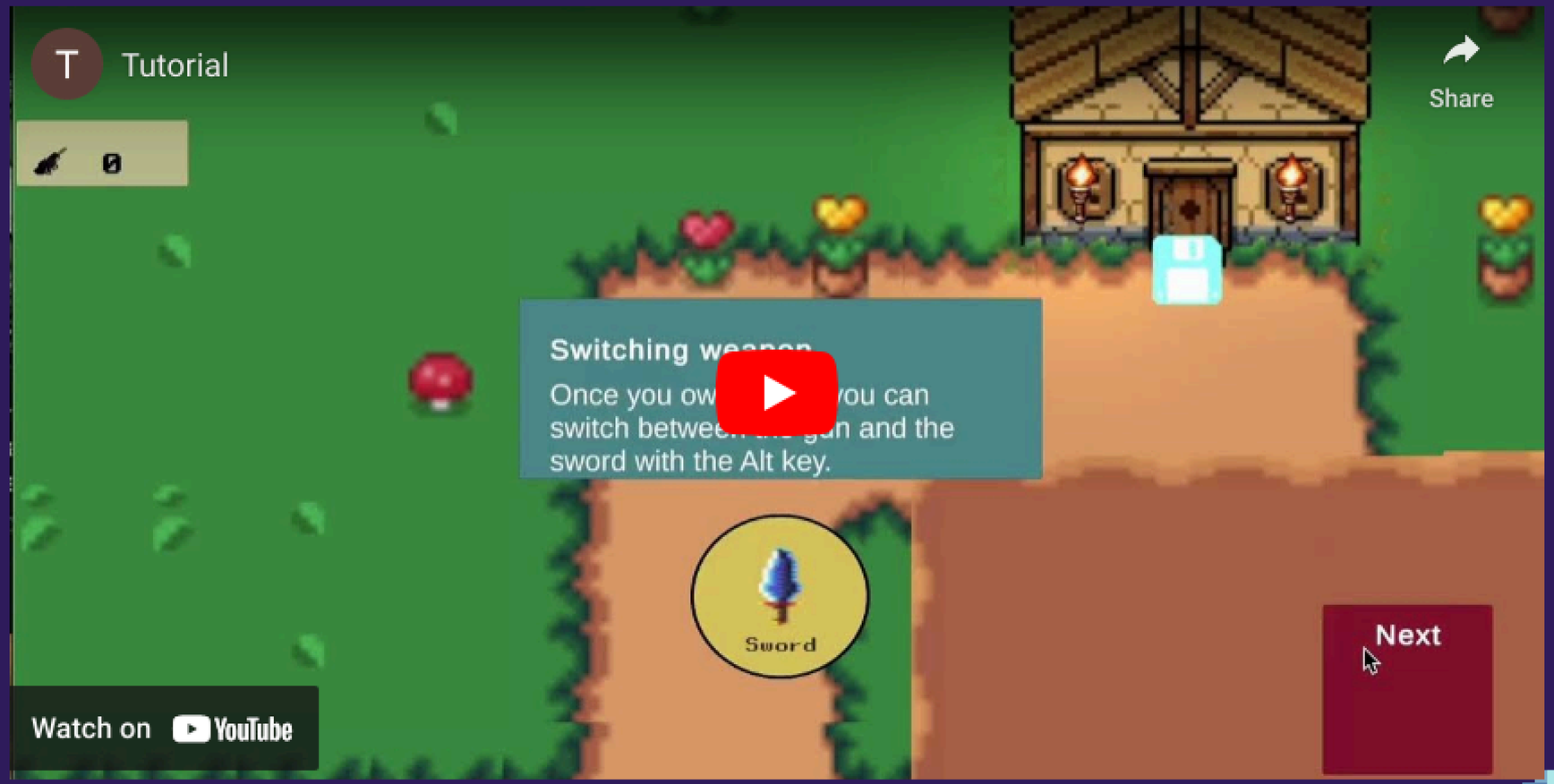
PLANTING

Planting a tree

PLANTING

- There are different types of plants with different deterioration time, leveling up time, purchase price, and harvest price.
- The player should water his plants regularly, otherwise, they will get deteriorated.
- However, there is a catch: Water is not limited, and the player should allocate water to his plants reasonably. Water is refilled every hour.

TUTORIAL SCENE



TUTORIAL SCENE

The game saves whether the player has opened the game before.

If no, a tutorial scene will be shown to introduce the player with basic gameplay.

SERIALIZATION

SERIALIZATION

The game supports saving and loading. Since the game is quite time-consuming, it is necessary for a user to save and load it again when needed.

PLAYER ATTACK - SWORD

PLAYER ATTACK - GUN

PLAYER ATTACK - GUN

Using a gun inflicts more damage to enemies and it is also a feasible way to attack within a long range.

BUILDING FENCES

BUILDING FENCES

Fences are used to prevent enemies from advancing further. Every day, there are 10 fences for a player to place on his land.

BUYING NEW PIECES OF LAND

BUYING NEW PIECES OF LAND

To be able to plant more, it is possible to purchase new pieces of land in the game.

BUYING FOOD

BUYING FOOD

The player gets more and more fatigued throughout the day. When his health reaches 0, he dies. Therefore, food must be provided to recover his health. To buy food, he goes to the village (during the day only) and reaches the market.

SKELETON'S AI

SKELETON'S AI

At night, skeletons appear and attempt to kill the player. They chase the player and move to wherever the player is, even when he is moving. One skeleton is going to sabotage both torches to turn the lights off to make the atmosphere darker.

ENEMIES

Slimes: Generated during daytime and they destroy plants, but only when there is at least one planted plant.

Skeletons: A couple is generated at night and they try to kill the player. A couple of skeletons is only generated when there are no other skeletons on the scene.

WEATHER

The weather in the game can be randomly rainy or sunny.

When rainy, these things happen:

- Deterioration progresses of plants are delayed
- Walking speed is reduced
- The sky is slightly darker

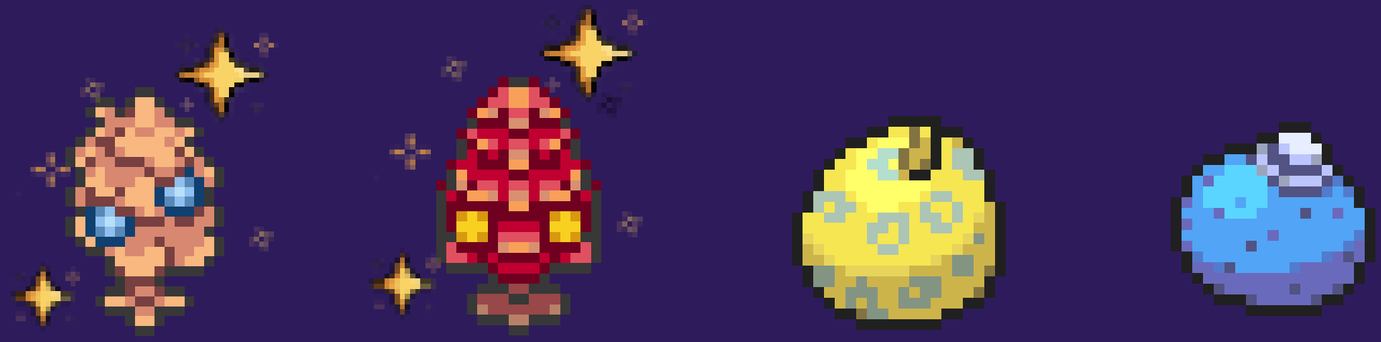
WEATHER

4. GRAPHIC ASSETS

All assets are taken from free assets on itch.io

Main asset pack: **Cute Fantasy Free** (with limited player animations, plants, and weapons)

These assets are taken from https://github.com/msikma/pokesprite



Main asset pack: Berry

5. IMPLEMENTATION

PLAYER MOVING

When idling, the Animator component of the player game object plays the Idle animation according to its current orientation. There are 4 possible orientations: ***UP, DOWN, LEFT, RIGHT***. If the pressed (or held) key is different from the current orientation, a function for switching has to run first before the player starts moving (if the key is being held).

The game also detects when holding a key (instead of just pressing), the player will start moving.

PLANT MANAGER

Planted plants are stored in a dictionary with keys being Vector3Int, indicating positions of plants on the **Plant** tilemap.

The next deteriorating time and **the next leveling up time** are stored and they’re checked in the FixedUpdate to ensure the plants level up or get deteriorated accordingly.

PLANT MANAGER

When the player waters a plant, the Time value for **The next deteriorating time** is added, making the deterioration slower.

FOOTPRINT

Every tile is associated with a footprint tile.

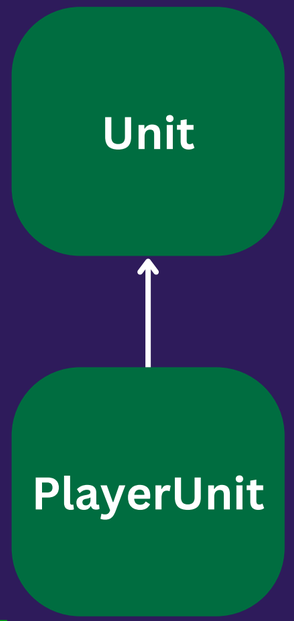
The game saves the current position to **lastPosition when the player arrives at that position**. When the player moves (lastPosition != currentPosition), a new object storing footprint information (position, time) is pushed to a queue.

FOOTPRINT

A footprint disappears after 10 seconds. In **FixedUpdate**, the game tries to dequeue and make the footprint disappear ***as far as possible***, meaning that when the current top item of the queue does not meet that requirement to disappear, it can stop dequeuing (because for a FIFO queue data structure, if the current top item has not been there long enough, later items are also certain to not meet this requirement).

HEALTH

Health is stored in a class called **Unit.** For the player, apart from health, there is information on water amount and money. This is stored in an inherited class called **PlayerUnit**.



SHOOTING

Bullets from the player's gun use **Kinematic** rigidbody to make it movable by only applying a velocity to the rigidbody itself. This behavior is intended.

```
public void Shoot(Vector2 direction)
{
    initialPosition = transform.position;
    this.direction = direction;
    rb.velocity = direction * speed;
}
```

SHOOTING

When **OnTriggerEnter2D** runs (the collider of a bullet triggers) and hits anything with the **Enemies** tag (Skeletons and Slimes), it makes the SpriteRenderer component of the hit object turns **red** ***using Color.Lerp to make it turn red for an amount of time.***

SHOOTING

```
0 references
void Update()
{
    if (isBlinking)
    {
        blinkTimer += Time.deltaTime;

        float blink = Mathf.Abs(Mathf.Sin(Time.time * blinkSpeed));
        Color currentColor = Color.Lerp(originalColor, blinkColor, blink);
        spriteRenderer.color = currentColor;

        if (blinkTimer >= blinkDuration)
        {
            StopBlink();
        }
    }
}

3 references
public void Flash()
{
    isBlinking = true;
    blinkTimer = 0.0f;
}

1 reference
void StopBlink()
{
    isBlinking = false;
    spriteRenderer.color = originalColor;  // Reset to the original color
}
```

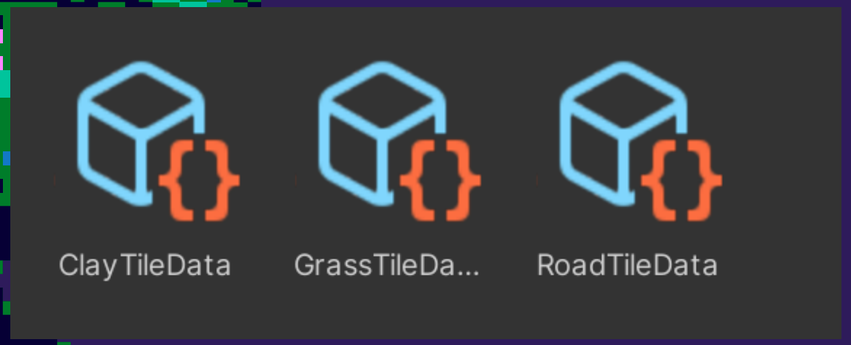
SCRIPTABLE OBJECT

Scriptable Object (SO) is a way to store data across different game objects and game scenes. This is helpful to store stats without the need of a game object. Another helpful use is initializing basic information of an object in the game.

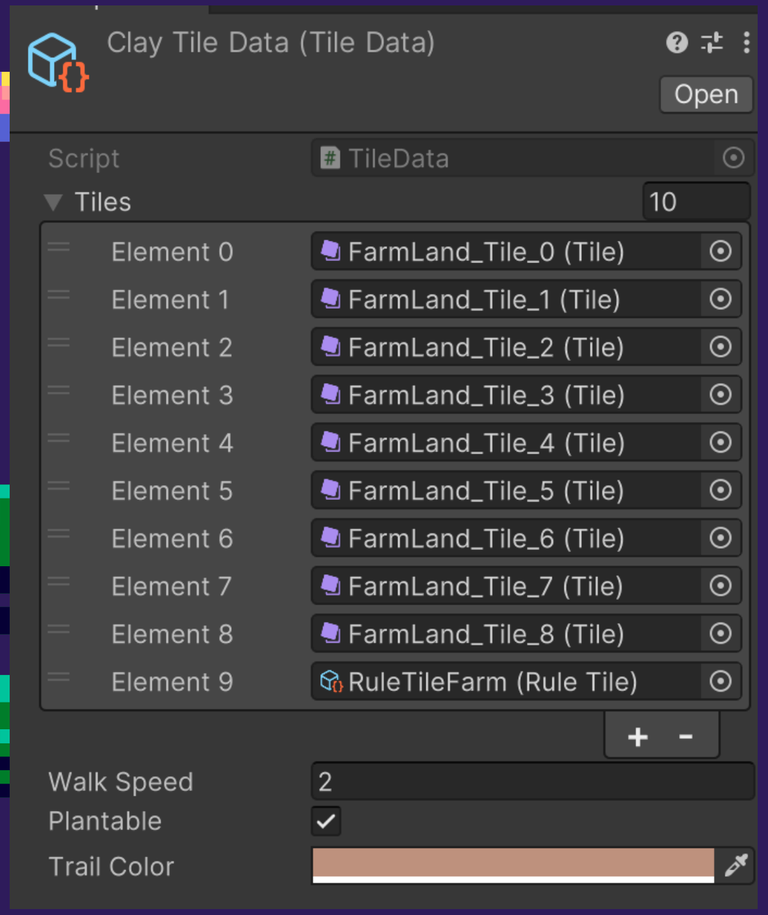
SCRIPTABLE OBJECT

In this game, SOs are used to store information of:

- Terrain tiles
- Plants

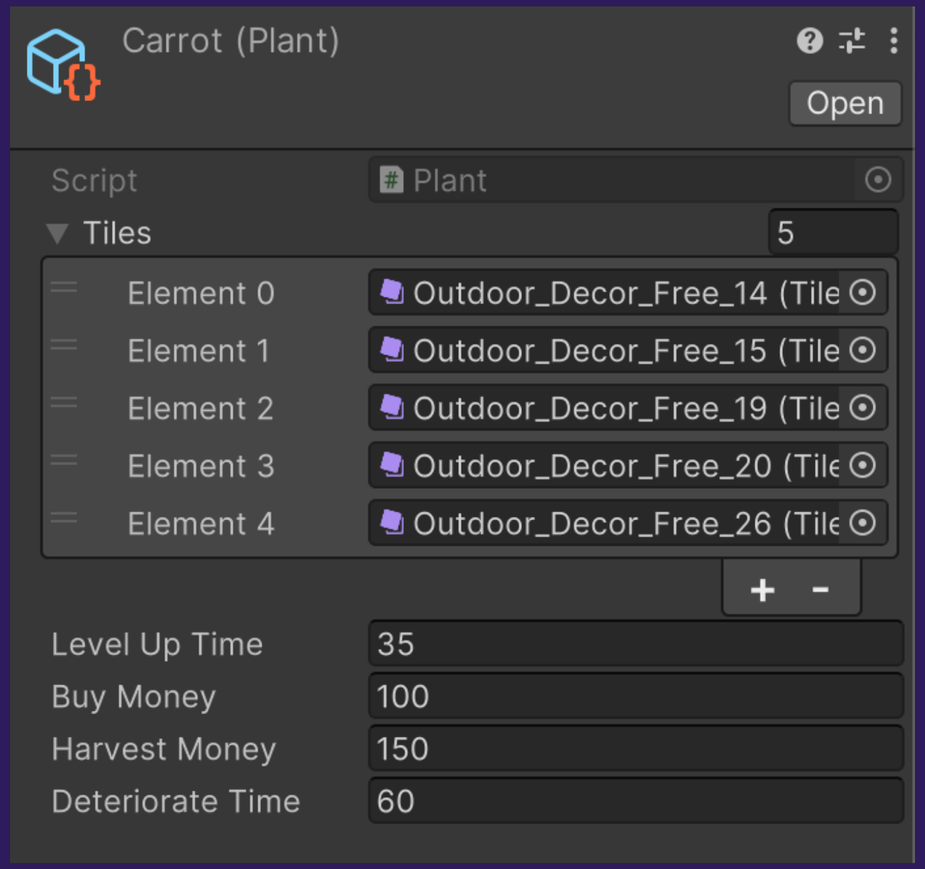


SCRIPTABLE OBJECT



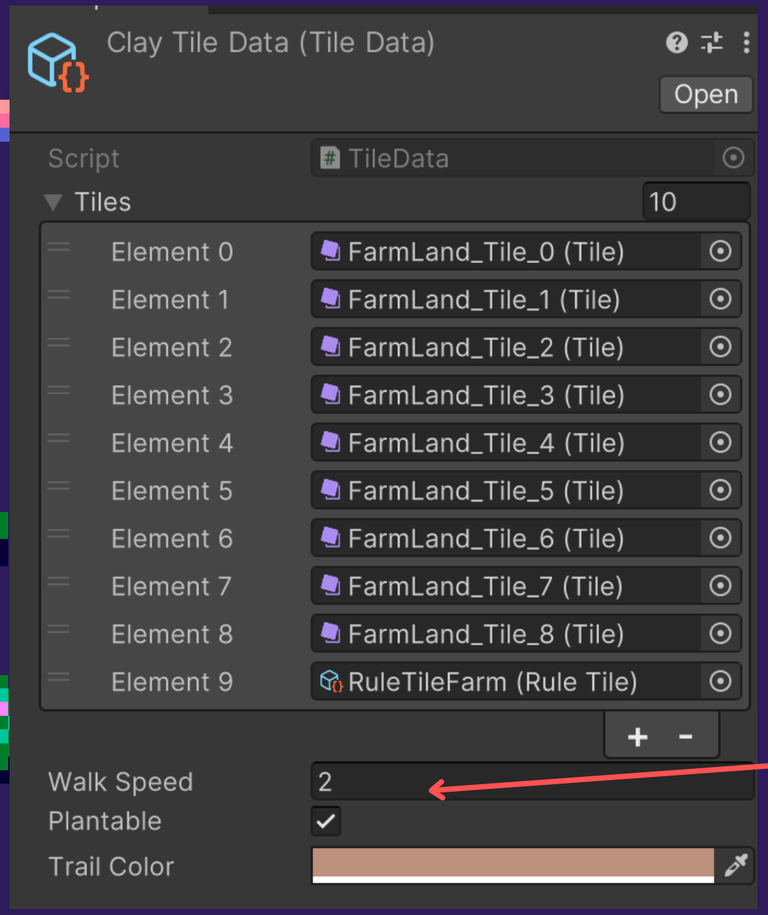
SO is used to store basic information about a specific tile, including walking speed, plantable, and trail color (all are going to be mentioned later).

SCRIPTABLE OBJECT



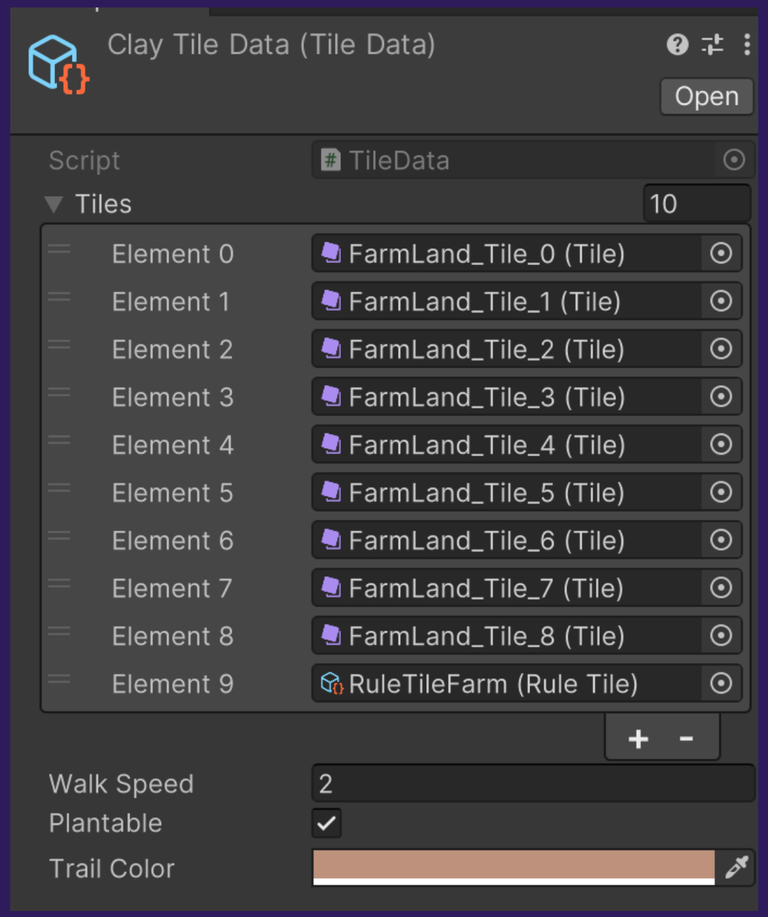
SO for plants

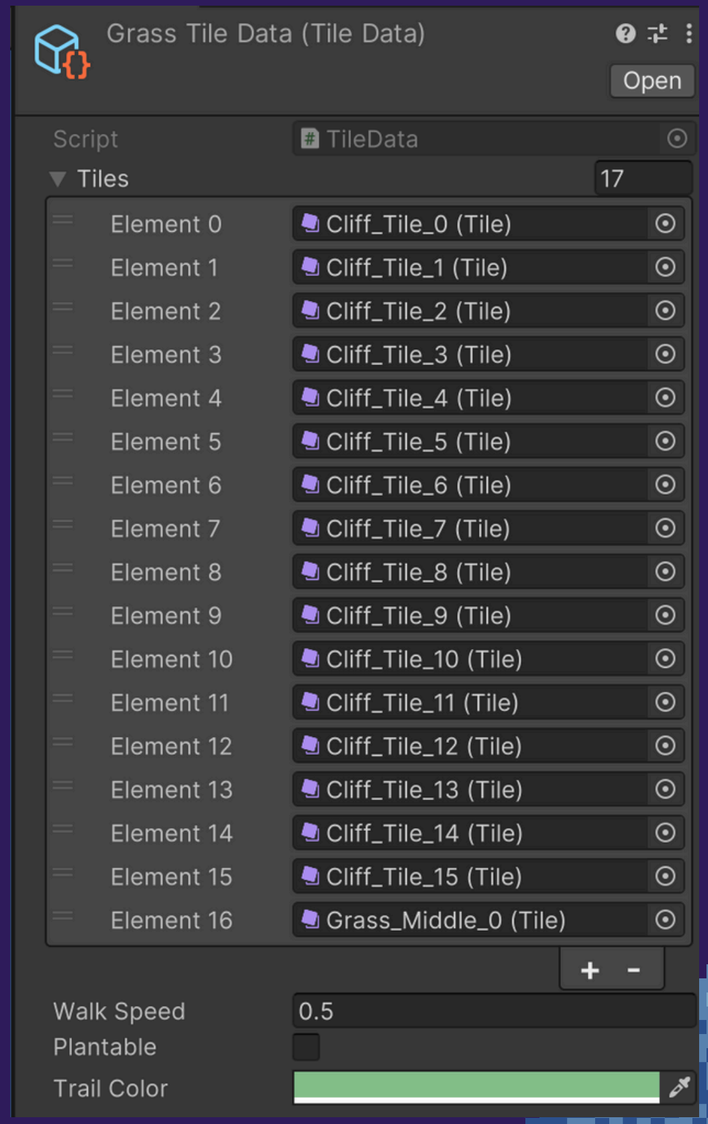
WALKING SPEED BY TERRAIN



This field is used to manipulate walking speed according to different tiles.

WALKING SPEED BY TERRAIN





WALKING SPEED BY TERRAIN

Tiles of a SO are then reassigned as keys to a dictionary, with their values being their respective SOs.

```
1 reference
[SerializeField] private List<TileData> tileDatas;

14 references
private Dictionary<TileBase, TileData> dataFromTiles;
10 references
```

RULE TILE

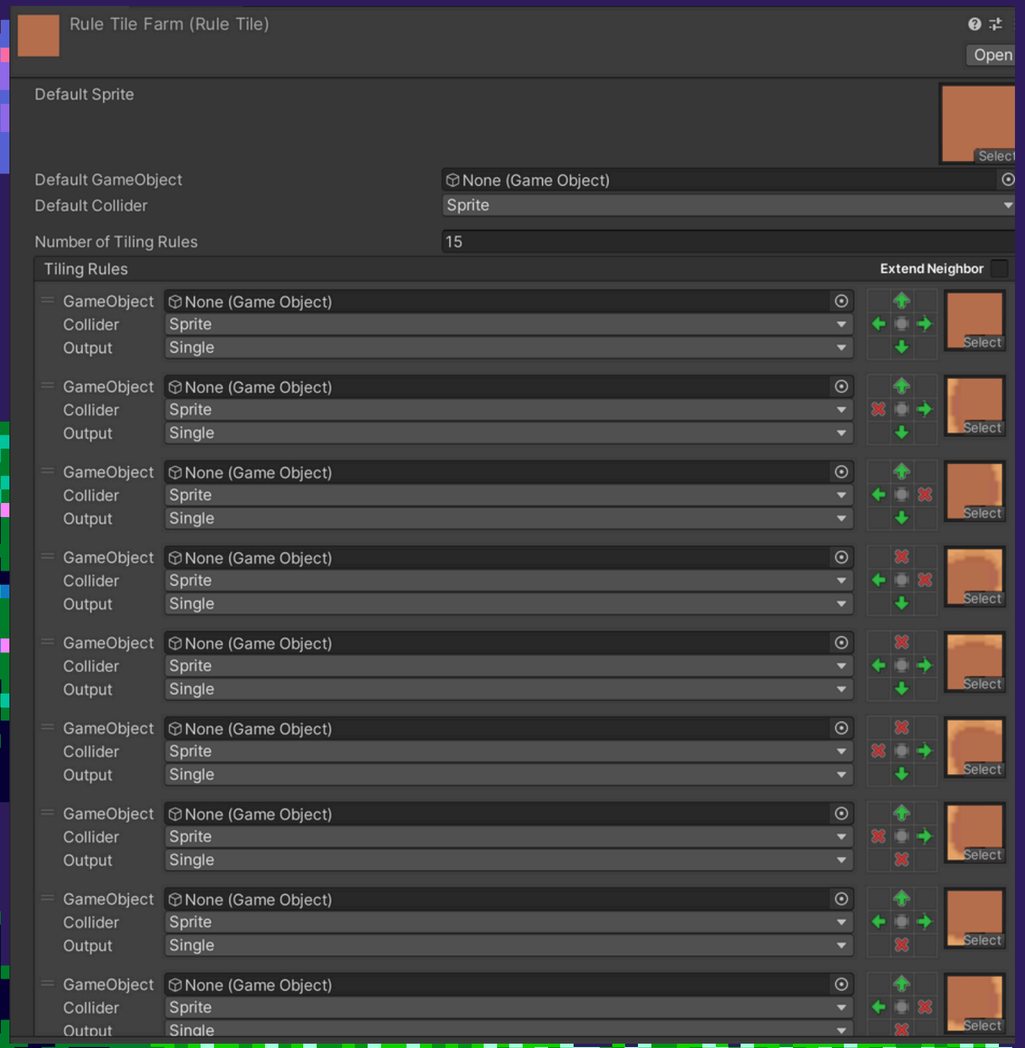
The game consists of a feature called **Land Purchase**. It is noticeable that the land has a layout to make it look realistic.

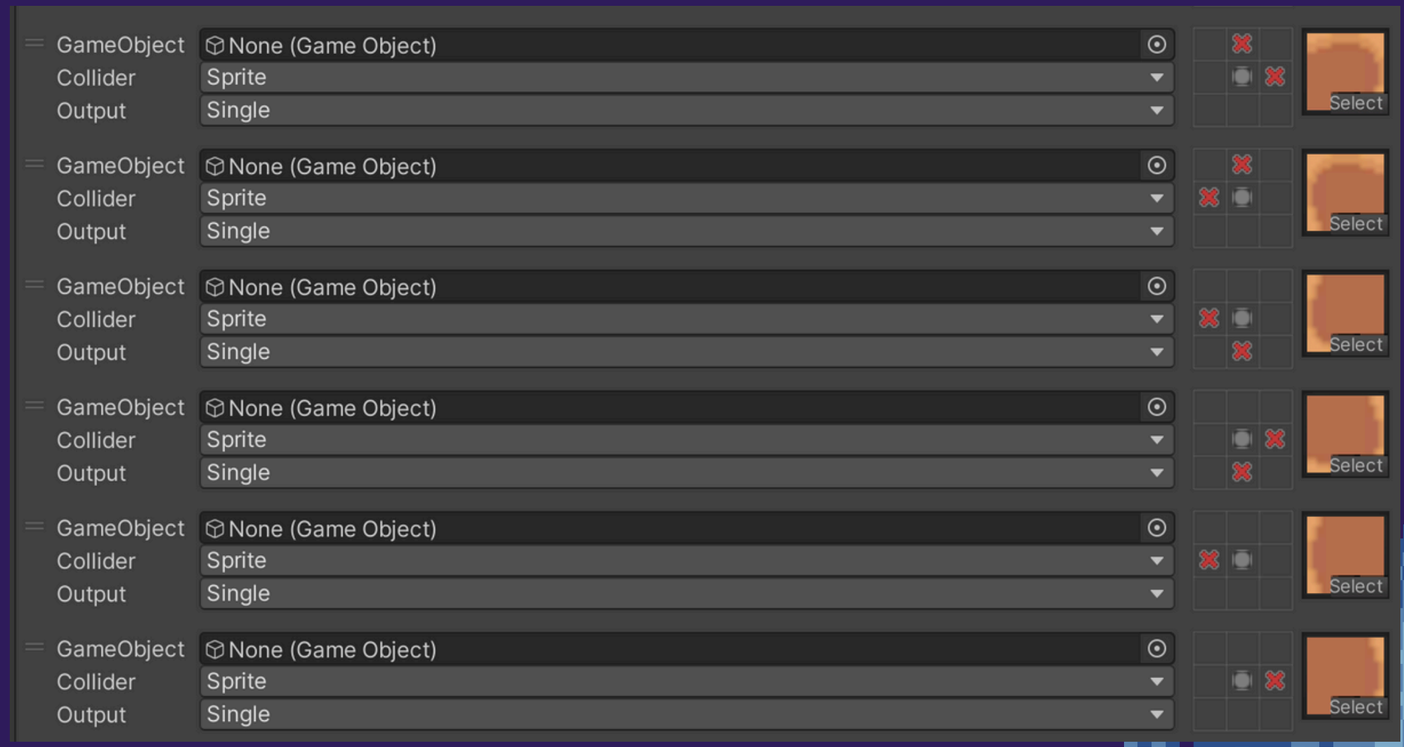
For edge tiles, there should be a curve acting as an edge instead of filling a totally brown edge.

RULE TILE

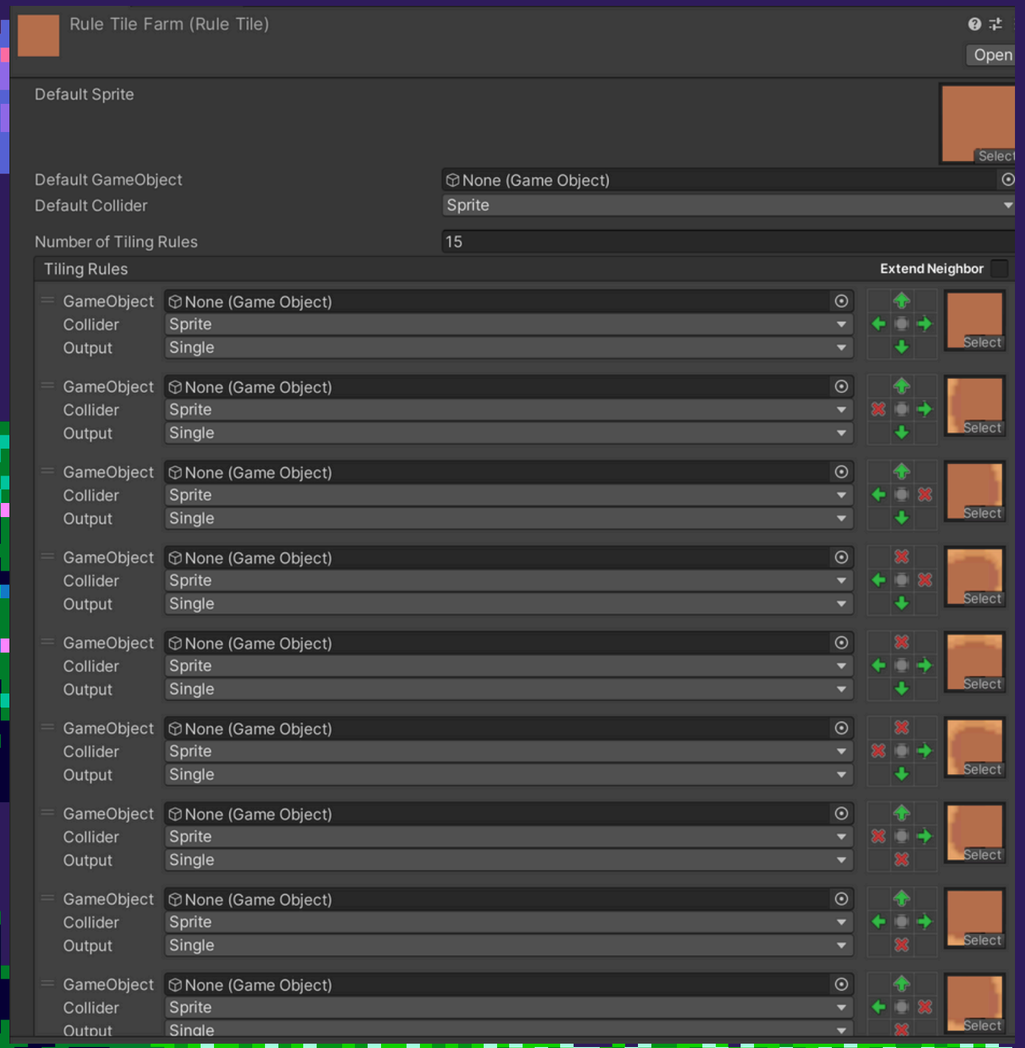
Therefore, **Rule Tile** is applied to ensure that edge tiles must have these “edge” curves.

RULE TILE


Rule Tile Farm (Rule Tile)
Open
Default Sprite
Select
Default GameObject
None (Game Object)
Default Collider
Sprite
Number of Tiling Rules
15
Tiling Rules
Extend Neighbor
GameObject
Collider
Output
Sprite
Single
Select


GameObject
None (Game Object)
Collider
Sprite
Output
Single
Select

RULE TILE



Complete tile when all are surrounded

Border on the left when the left side is unoccupied

Border on the right when the right side is unoccupied

Border on the top right when the right side and top side are unoccupied

Border at the top when the top side is unoccupied

Border on the top left when the left side and top side are unoccupied

Border on the bottom left when the left side and bottom side are unoccupied

Border an the bottom when the bottom side is unoccupied

Border on the bottom right when the right side and bottom side are unoccupied

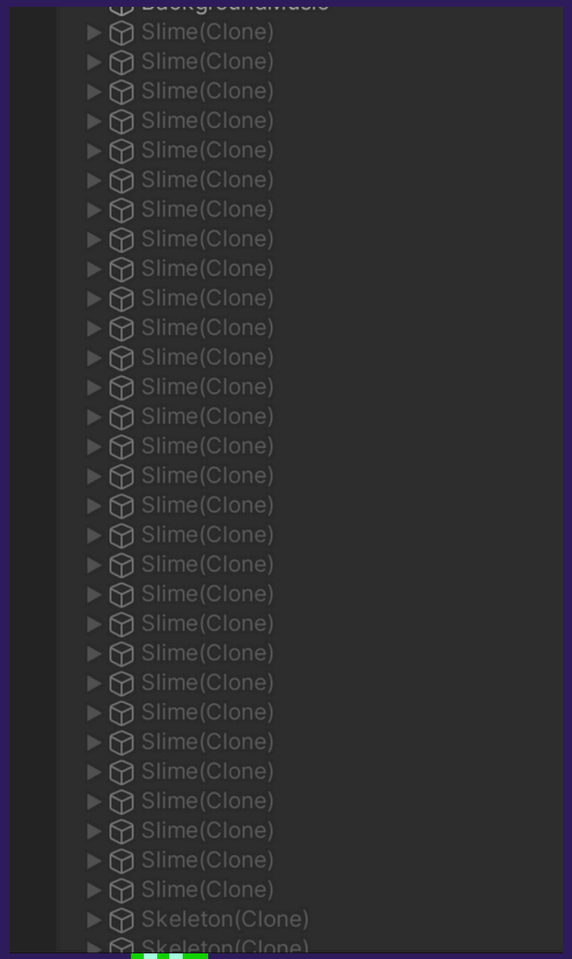
RULE TILE

OBJECT POOLING

Object pooling is a design pattern that can provide performance optimization by reducing the processing power required of the CPU to run repetitive create and destroy calls. Instead, with object pooling, existing GameObjects can be reused over and over.

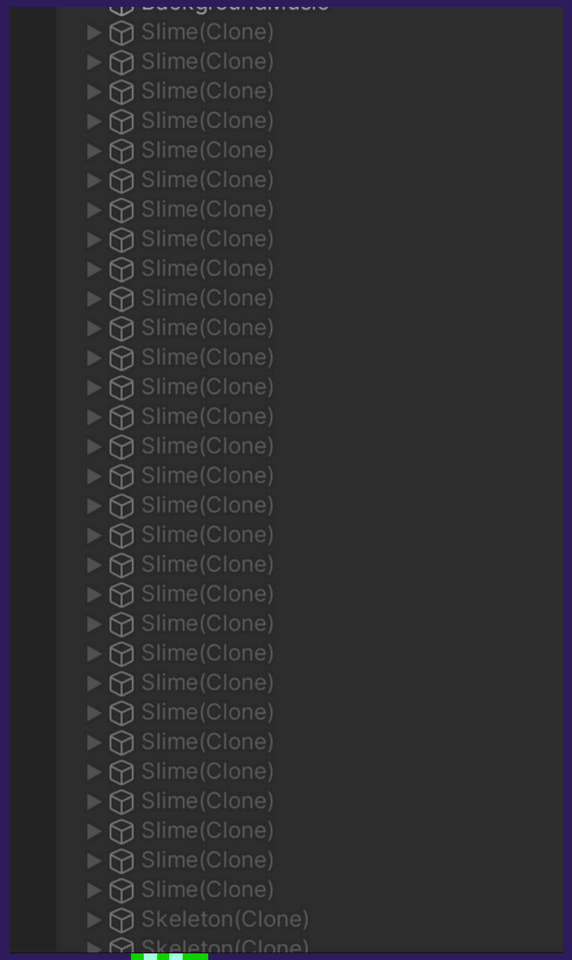
This is useful when there is a GameObject that has to be instantiated repeatedly in the game.

OBJECT POOLING

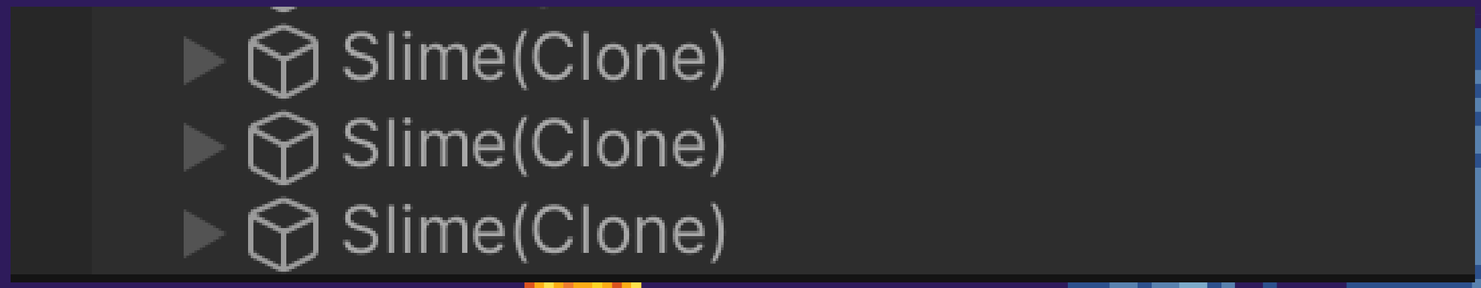


Slimes and Skeletons when they are not ready to spawn

OBJECT POOLING



When slimes are ready to spawn



OBJECT POOLING

Store different queues for different GameObjects.

```
6 references
private static Dictionary<string, Queue<GameObject>> poolDictionary; // Pools corresponding to tags
```

Keep track of spawned objects to put them back when they are no longer in use.

```
5 references
private static List<Pair<GameObject, string>> spawnedObjects;
```

OBJECT POOLING

Object pooling is also used for spawning bullets since bullets can be instantiated many times throughout the game.

DAY/NIGHT CYCLE

To simulate day-night cycle in the game, we used **Global Light 2D** with a color that changes throughout the day.

DAY/NIGHT CYCLE

The intensity of the Global Light also fluctuates throughout the day to match with the time.

This can be achieved by using **Mathf.Lerp**

Light intensity: 0.8 - 1.5

Light intensity: 0.1 - 0.65

DAY/NIGHT CYCLE

Another Spot Light 2D is used to simulate light emitted from a torch.

SHADER

Using shader is a good way to manipulate graphical aspect of a sprite in the game.

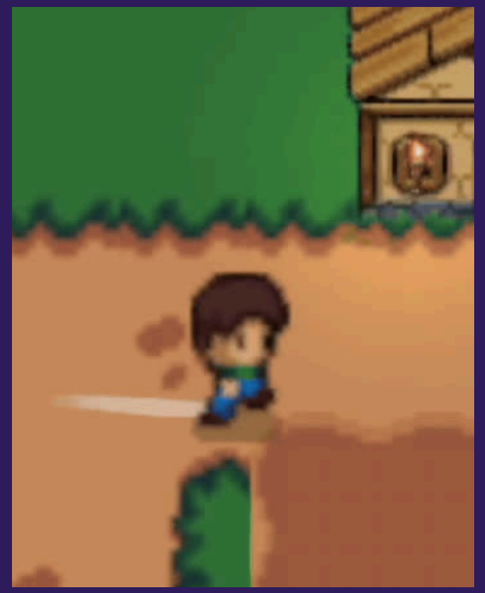
In our game, we've implemented shader to create effect on:

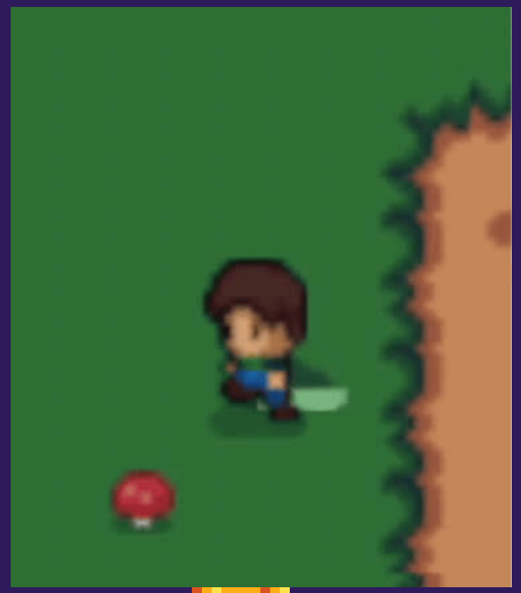
- The Save icon
- Plants when they're ready to be harvested

SHADER

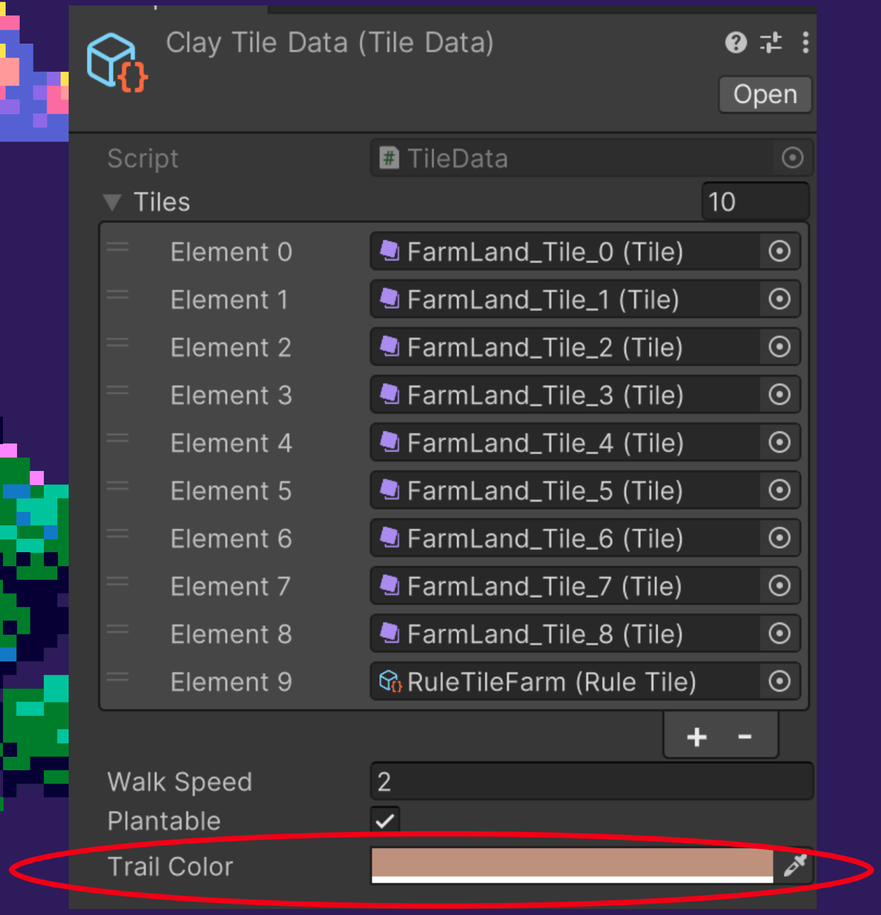
PARTICLE SYSTEM

Particle system is used to implement the dust trail effect when the player moves on dirt or grass in the game.





PARTICLE SYSTEM



The dust trail color is saved in tiles' respective Scriptable Objects. By applying this color to the start color of the dust trail particle system, we were able to make the trail look similar to the terrain.

MINIMAP

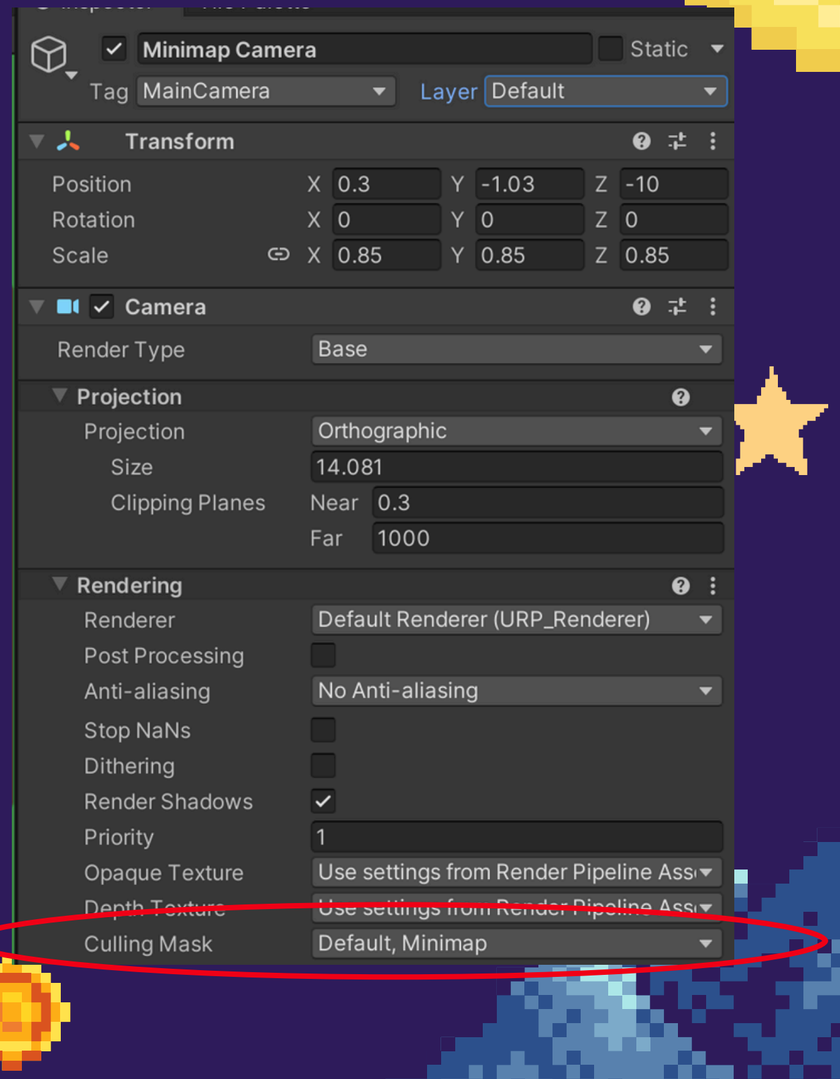
To implement a minimap that allows the player to easily navigate incoming enemies, we have added the following objects to our game:

- Layers
- Indicating arrows
- Another camera

MINIMAP

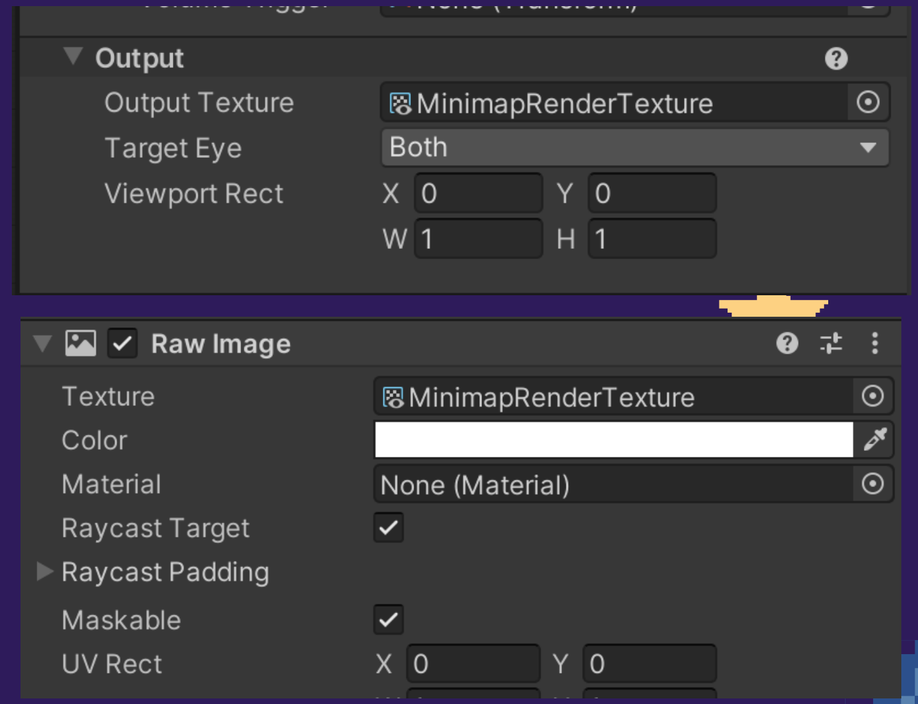
We added a new layer: **Minimap.** This layer contains indicating arrows only since the minimap only shows these arrows and it does not show other things, such as plants.

Next, we created a **secondary Camera** in our game scene. This camera only shows objects from **the Minimap layer**.

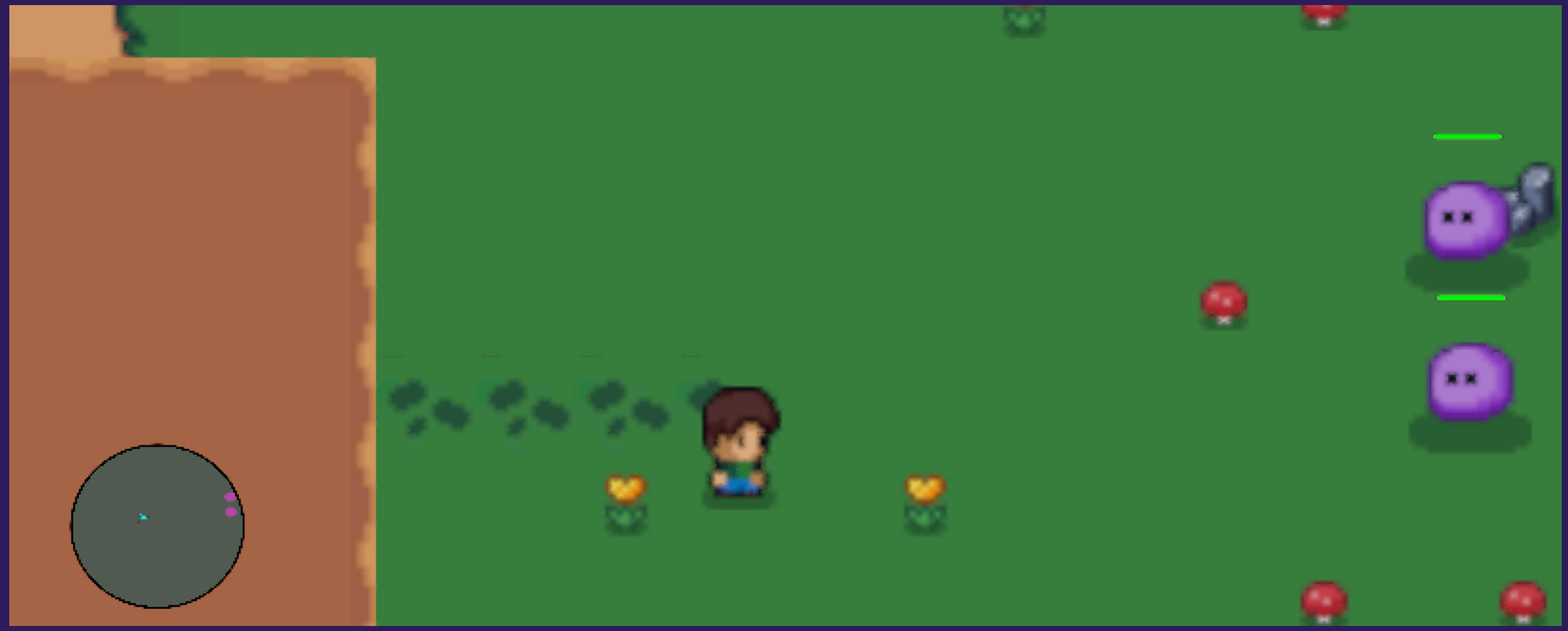


MINIMAP

Next, we created a new **Render Texture** and set the output texture of the minimap camera to this render texture. To show the map on the screen, we added a new Raw Image component and set its texture to the aforementioned render texture.



MINIMAP



SERIALIZATION

To save and load game, we constructed classes that are intended for serialization and only contain neccessary and serializable information.

SERIALIZATION

```
37 references
public class TimeManage : MonoBehaviour
{
    25 references | 30 references | 4 references
    public int currentHour = 6, currentMinute = 0, currentDay = 1;
    1 reference
    public float updateIntervalSeconds;
    6 references
    private Light2D globalLight;

    3 references
    private DateTime lastUpdated;

    4 references | 1 reference
    public TextMeshProUGUI timeText, dayText;

    37 references
    public static TimeManage instance;

    1 reference
    public TimeData Serialize()
    {
        TimeData timeData = new TimeData();
        timeData.currentHour = currentHour;
        timeData.currentMinute = currentMinute;
        timeData.currentDay = currentDay;


        return timeData;
    }
```

```
[Serializable]
8 references
public class TimeData: DataSerialize
{
    2 references
    public int currentHour;
    2 references
    public int currentMinute;
    2 references
    public int currentDay;

    1 reference
    public static TimeData Deserialize(string json)
    {
        return DataSerialize.Deserialize<TimeData>(json);
    }
}
```

SERIALIZATION

Objects of the ***Dictionary*** class are not serializable. A workaround is to create an alias class

SERIALIZATION

```
[Serializable]
9 references
public class SerializableDictionary<TKey, TValue>
{
    [Serializable]
    4 references
    public class SerializableDictionaryEntry
    {
        6 references
        public TKey key;
        6 references
        public TValue value;

        1 reference
        public SerializableDictionaryEntry(TKey key, TValue value)
        {
            this.key = key;
            this.value = value;
        }
    }

    7 references
    public List<SerializableDictionaryEntry> entries = new List<SerializableDictionaryEntry>();

    2 references
    public void FromDictionary(Dictionary<TKey, TValue> dictionary)
    {
        entries.Clear();
        foreach (var kvp in dictionary)
        {
            entries.Add(new SerializableDictionaryEntry(kvp.Key, kvp.Value));
        }
    }
```

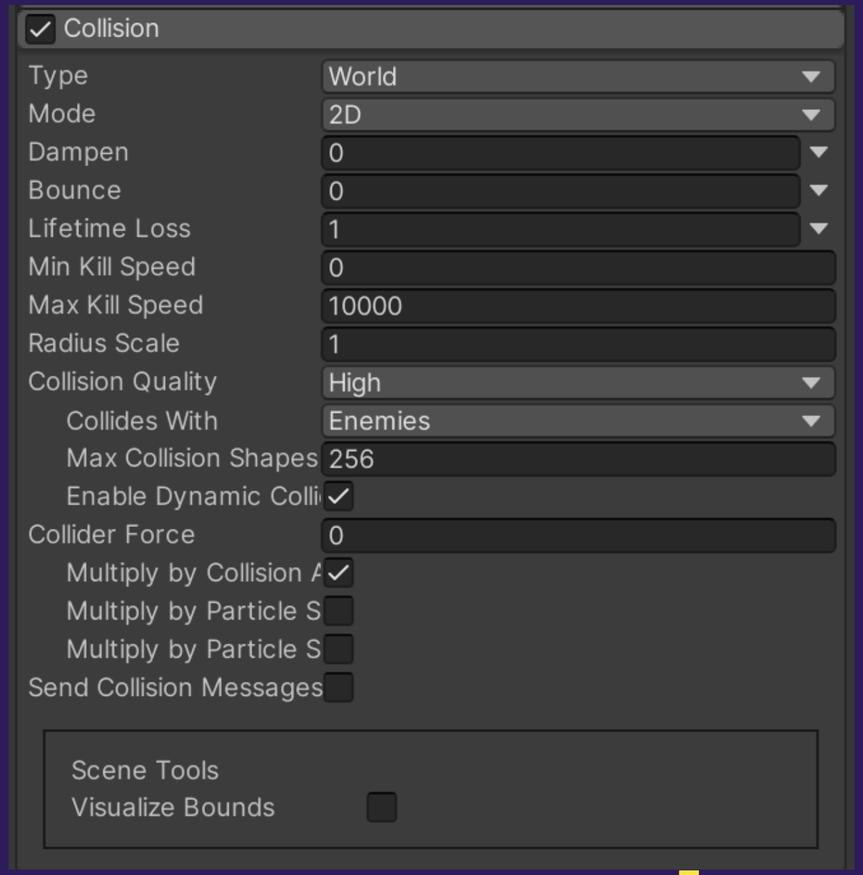
```
    2 references
    public void FromDictionary(Dictionary<TKey, TValue> dictionary)
    {
        entries.Clear();
        foreach (var kvp in dictionary)
        {
            entries.Add(new SerializableDictionaryEntry(kvp.Key, kvp.Value));
        }
    }

    3 references
    public Dictionary<TKey, TValue> ToDictionary()
    {
        Dictionary<TKey, TValue> dictionary = new Dictionary<TKey, TValue>();
        foreach (var entry in entries)
        {
            dictionary[entry.key] = entry.value;
        }
        return dictionary;
    }
}
```

RAIN

The raining effect is created by using **the Particle System**. We call this object **Rain**.

To make the splash effect, we've enable the **Collision feature** (World) of the Particle System (with **Bounce = 0 and Lifetime Loss = 0.1**).

RAIN

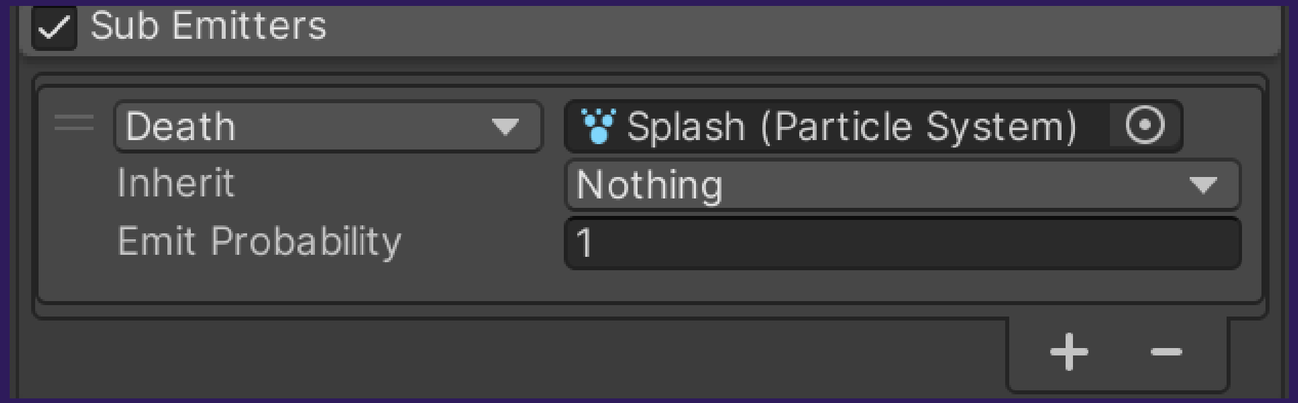

Collision
Type
World
Mode
2D
Dampen
0
Bounce
0
Lifetime Loss
1
Min Kill Speed
0
Max Kill Speed
10000
Radius Scale
1
Collision Quality
High
Collides With
Enemies
Max Collision Shapes
256
Enable Dynamic Colli
Collider Force
0
Multiply by Collision A
Multiply by Particle S
Multiply by Particle S
Send Collision Messages
Scene Tools
Visualize Bounds

RAIN

To make the rain looks more realistic, a splash effect on the ground is also implemented. This is also implemented using **the Particle System**.

The new particle system object is set as the sub-emitter of the aforementioned **Rain** particle system.

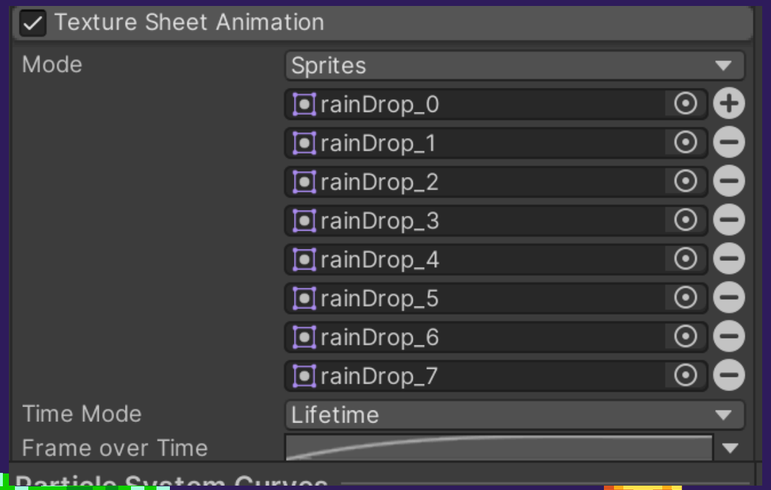
RAIN



When raindrops hit, they die. After this, splashes are shown.

RAIN

The **Splash** particle system use the Texture Sheet Animation component to make it look like a raindrop splashing after hitting.



RAIN

GAME DEMO


FunFarm Demo
Share
FUN FARM
TUTORIAL
QUIT
Tutorial mode
Watch on YouTube


THE END.