TECHNICAL DESIGN DOCUMENT

Cult of the Merger

Abstract

Summary of your game

Will Bennett

Table of Contents

Content

Design Patterns

Prototype

Prototyping can be boiled down to creating similar objects of each other. This approach was taken in creating the pawn system with a building class being able to spawn all the different objects into the scene. The idea used in the project is to create a base pawn object and then attach different component to change its functionality.

This is achieved through the use of two scriptable objects to store data about each pawn. The core data storage scriptable object is called Object Data. This class will be expanded in the scriptable object section of this document.

Update Method

Unity inherently has an update method; this method is called every frame (Unity, N.D.). It has different versions of an update method that updates at different rates. These are used to update the positions of the pawns when they are being dragged.

Game loop

Game loop allows gametime to be decoupled from the user's processor speed. This goes well with the update method pattern. Most of this game will be decoupled from the user except from the generation of mana which will be generated by different in time when the user logged off and back onto the game.

Singleton

Game manager is the key class that makes use of the singleton pattern. This means that there should only be one instance of the game manager as it will persist between scenes.

```
public class GameManager : MonoBehaviour
{
    #region  Singleton
    private static GameManager _instance;

    public static GameManager Instance
    {
        get
        {
            if (_instance == null)
            {
                Debug.Log("GameManager is null");
            }

            return _instance;
        }
    }

    private void Awake()
    {
        if(_instance)
            Destroy(gameObject);
        else
            _instance = this;
        DontDestroyOnLoad(this);

    }
    #endregion
```

Figure 1 : Screenshot of the singleton being used in the game manager class

Event System

Event system allows for processing events that objects in the scene have sent. This can help with decoupling classes. Unity comes with a built-in event system which the most common use is to manage the inputs of the user (Unity, N.D.), this allows for other event systems to be built using a similar system. In this project it is being used to allow components to talk to other components an example of this is when a pawn levels up and gets their stats upgrades. This gets communicated between the merge component and the mana component. This is due to not all pawns owning a mana component such as the buildings. Therefore, subscribing listeners to this event allows the merge component to be flexible with the use of other classes. To limit the reach of an event being fired each listener has an id linked to its components. Through the use of this objects do not need to directly reference each other, this follows the design pattern on Observer.

Code snippets

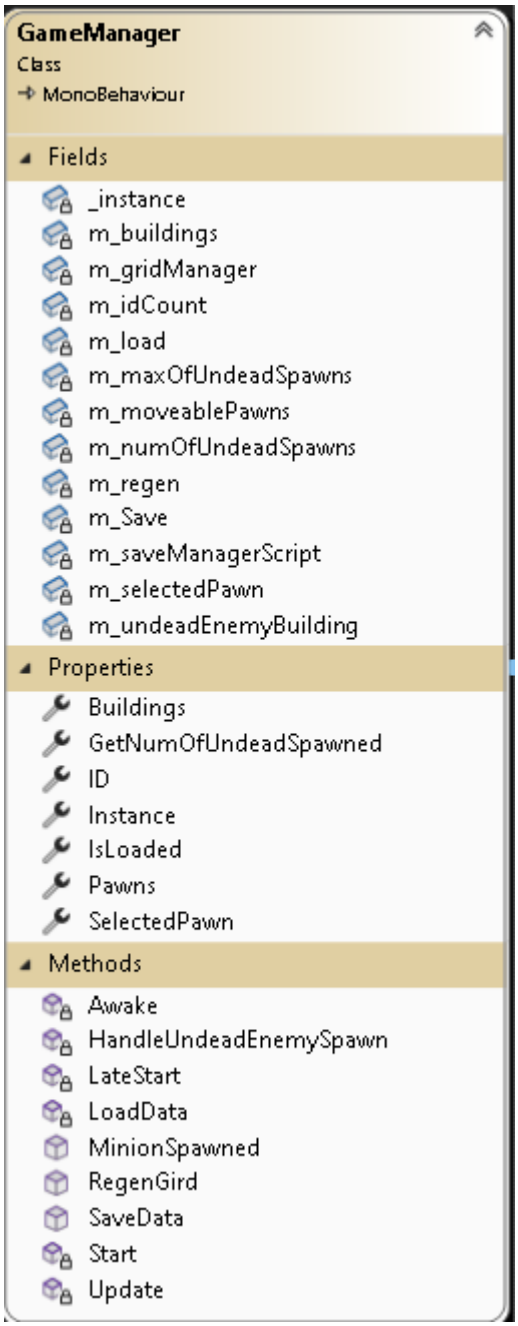
```
using System;
using System.Collections;
using System.Collections.Generic;
using System.Data;
using UnityEngine;

public class GameEvents : MonoBehaviour
{
   public static GameEvents m_current;

   public void Awake()
   {
      m_current = this;
   }
   public event Action<int> onMinionLevelUp;

   public void MinionLevelUp(int id)
   {
      if (onMinionLevelUp!=null)
      {
         onMinionLevelUp(id);
      }
   }
}
```

Figure 2: Screenshot showing event system

Game Manager

The game manager helps control the functionality of the game; it makes use of the singleton pattern which allows only one version of it to exist. This means a single manager will persist between scenes, fortunately this game only has one scene, but a game manager is still needed. This manager handles loading and saving data through the save manager script and storing key pawns in a list. This is done within two lists, one which contains all of the buildings in the scene to update their grid lists and another to store all of the moveable pawns which will be sent to the save manager.

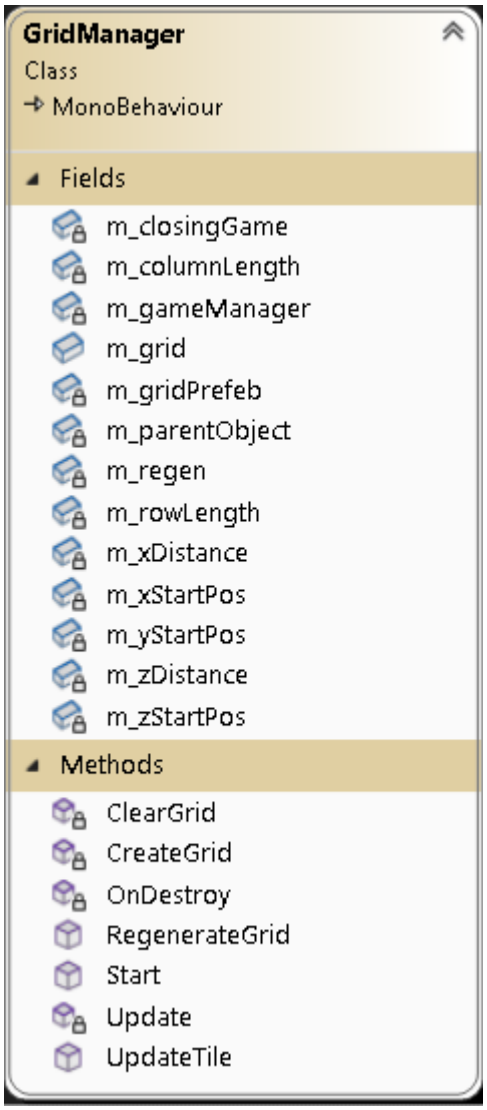
UML



Grid Manager

The grid manager is an extension of the game manager which allows it to spawn the grid on the game start and helps manage updating the tiles with the data that the game manager stores.

UML



Saving

Saving is done via a script attached to the game manager. There are key times when the game saves due to the unity event OnApplicationClose() does not get called every time an app closes due to the application not fully closing despite the app being forced to close. To get around this issue the save manager will save at two key parts, when the application loses focus by using OnApplicationFocus(), and when pawn's movement script is interacted. Saving being bound to when the application loses focus will allow the game to be saved when the game closes or if the player exits the game. Meanwhile saving when the pawn's movement script is interacted with means all new positions of pawns will be saved, this can help just in case OnApplicationFocus() does not get called.

All of this is triggered by the game manager to stop duplicate saving from occurring. As stated, earlier pawns with movement scripts are saved, these are saved to player prefs in a certain way as seen in Figure 3. These are then loaded using non interactable buildings, using their load pawn function. The strings they are saved as are separated by "_" and put into a list, an if statement checks for their pawnType and sends it to the corresponding building.

```
1 reference
public void SavePawns(List<PawnMovement> pawns)
{
    //DeletePrefs();
    for (int i = 0; i < pawns.Count; i++)
    {
        string key = "pawn"+ i.ToString();
        string data;

        //save where it is on the grid
        data = pawns[i].GetHomeTileNum.ToString() + "_";
        //save what type of pawn it is
        data += pawns[i].GetPawnMerge.GetPawnObjectType.ToString()+"_";
        //save what index it is
        data += pawns[i].GetPawnMerge.GetPawnDataIndex.ToString() + "_";
        //save waht level it is
        data += pawns[i].GetPawnMerge.GetPawnLevels.ToString() + "_";
        //save what minion type it is
        data += pawns[i].GetPawnMerge.GetMinionType.ToString() + "_";

        //save what type of item it is
        data += pawns[i].GetPawnMerge.GetItemType.ToString() + "_";
        //save what type of reward it is
        data += pawns[i].GetPawnMerge.GetRewardType.ToString() + "_";
        //save what type of enemy it is
        data += pawns[i].GetPawnMerge.GetEnemyType.ToString() + "_";
        //save what type of store it is
        data += pawns[i].GetPawnMerge.GetStoreType.ToString() + "_";

        //m_Data.Add( data);
        //save to player prefs
        PlayerPrefs.SetString(key, data);
    }
    PlayerPrefs.SetInt("PawnNum", pawns.Count);
    SaveStats();
    PlayerPrefs.Save();
    Debug.Log("save data");
}
```

Figure 3: How pawns are saved

```
public void LoadPawns()
{
    //load from player prefs
    Debug.Log("load data");
    for (int i = 0;i<PlayerPrefs.GetInt("PawnNum");i++)
    {
        m_dataList.Add(new ListWrapper());
        string key = "pawn" + i.ToString();
        m_dataList[i].m_data = PlayerPrefs.GetString(key).Split("_").ToList();
        //m_dataList[i].m_data = m_Data[i].Split("_").ToList();
    }

    for (int i = 0; i < m_dataList.Count; i++)
    {
        if (m_dataList[i][1]=="Minions")
        {
            m_graveMinionBuilding.LoadPawn(int.Parse(m_dataList[i][0]), int.Parse(m_dataList[i][2]), int.Parse(m_dataList[i][3]),
                m_dataList[i][4], m_dataList[i][5], m_dataList[i][6], m_dataList[i][7], m_dataList[i][8]);
        }
        if(m_dataList[i][1] == "Building")
        {
            if(m_dataList[i][4] == "Undead")
            {
                m_graveBuilding.LoadPawn(int.Parse(m_dataList[i][0]), int.Parse(m_dataList[i][2]), int.Parse(m_dataList[i][3]),
                m_dataList[i][4], m_dataList[i][5], m_dataList[i][6], m_dataList[i][7], m_dataList[i][8]);
            }
            else if(m_dataList[i][4] == "Plant")
            {
                m_lifeBuilding.LoadPawn(int.Parse(m_dataList[i][0]), int.Parse(m_dataList[i][2]), int.Parse(m_dataList[i][3]),
                m_dataList[i][4], m_dataList[i][5], m_dataList[i][6], m_dataList[i][7], m_dataList[i][8]);
            }
            else if (m_dataList[i][4] == "Demon")
            {
                m_hellBuilding.LoadPawn(int.Parse(m_dataList[i][0]), int.Parse(m_dataList[i][2]), int.Parse(m_dataList[i][3]),
                m_dataList[i][4], m_dataList[i][5], m_dataList[i][6], m_dataList[i][7], m_dataList[i][8]);
            }
        }
```

Figure 4: How pawns are loaded

Pawns

Pawns are the components that make up the gameplay elements. These can come in different iterations by using prefab variants to give them different components. These variants will use the prototype pattern to have all of their data related to their spawning be pre-set in a scriptable object.

Components

Components in Unity are functional pieces of every GameObject. This means each component should contain properties to define the behaviour of said GameObject (Unity, 2023). By default, each GameObject has components attached to them such as a transform, but more can be added such as a rigidbody. This allows the GameObject to have physics be applied to it.

Bringing this back to Cult of the Merger, the purpose of component-based programming is to create reusable code that can be applied to GameObjects throughout the game. A great example of this is pawns. In engine there are a few scripts that are attached to pawns but allow for each of them to use them in different ways depending on which type of pawn it is.

Interactable

Interactable is attached to any object that needs to be tapped or dragged around the scene. This class handles the inputs from the input manager and acts as a bridge to the other pawn components.

Functions

HandleTouchDown

If the bool m_isInteractable is set to true and the object touched is a building this function will call its tapped function.

HandleHoldInput

This function handles the setting up variables for the dragging function. These variables are its starting positions and setting the movement script of the pawn as it being held. It uses the camera's world position to the screen to set the pointer position.

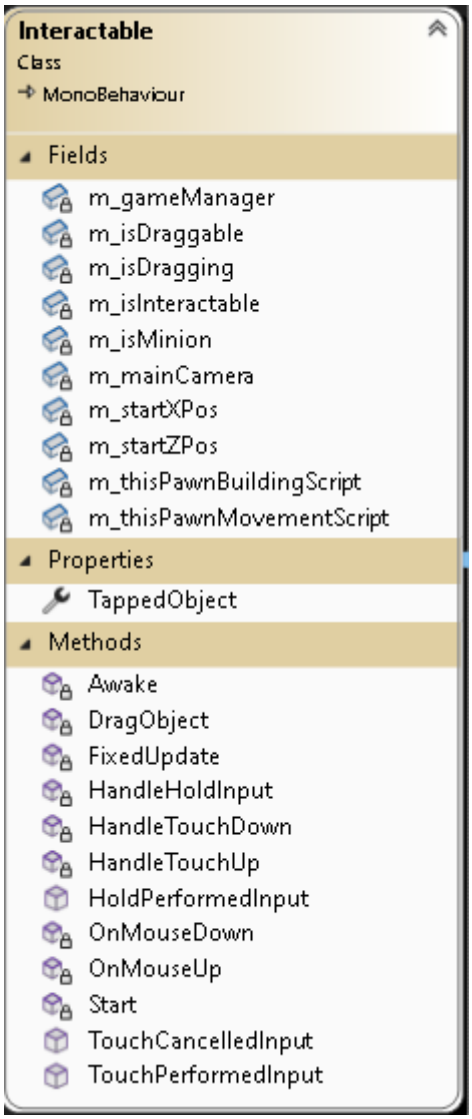
HandleTouchUp

This handles the player letting go of an object and sets dragging to false, and calls dropped on the movement script.

DragObject

This handles physically moving the object around in the scene.

UML



Movement

Movement in this game is done through dragging the pawns to either move them to a different space or to initiate merging. This script is having core focuses of storing the pawns home tile and its number, and to detect when the pawn is being dropped or being moved.

Home tile is a way for a pawn to remember where it is coming from. This is initially assigned when the pawn in spawned if there is space on the grid. The home tile is important as if the pawn is moved

to an occupied tile, it will be moved back to its home tile when released. When the pawn is dropped a raycast is shot to update the tile via the grid manager.

Functions

GetHomeTile

This is a property that allows other classes to have access to the pawn's home tile.

ClearHomeTile

This is a public function that sets the pawns current home tile taken state to false. It is called whenever the pawn is moved or is destroyed such as in merging with another pawn.

Dropped

This is the main function of the class and it collects the data related to dropping a pawn onto a tile. It uses a raycast to detect if the tile below is free and if so it updates the grid manager to update its list of tiles. It also calls other classes functions such as AttemptDropMerge or AttemptSacrifice to check if it is able to merge with a pawn below or if it can be sacrificed. The dropped pawn is then moved back to its home tile.

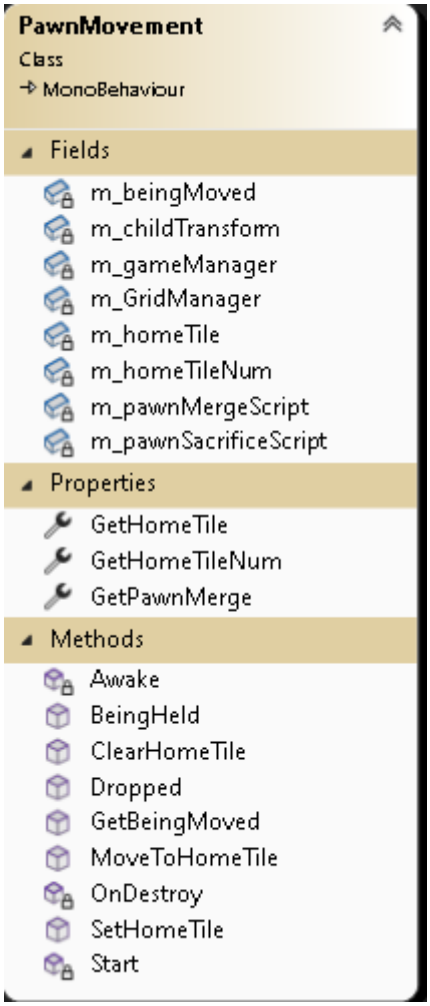
SetHomeTile

This public function sets the current pawn's m_homeTile to a new location and moves it there.

MoveToHomeTile

This function moves the pawn to the pawn's stored m_homeTile location. It then calls the GameManagers save function.

UML



Building

This component handles spawning everything in the scene from items to minions. It achieves this spawning the pawns related to their data stored on the Object data scriptable object.

Functions

AssignObjectData

AssignObjectData takes all of the data from ObjectData and ItemData and transfers them into lists dedicated. These lists will then be used later when setting their typing and getting specific data related to the pawns.

PopulateGrid

This populates the grid reference that each building holds, this can allow them to pass relevant location information to the pawns they spawn. This includes checking if a tile is free and setting a minion's home tile.

Tapped

The main purpose of this function is to handle touch input from the interactable script. This will cause the building to spawn a pawn on a free tile. If the building has a limited amount of uses, then it will decrement here.

LoadPawn

LoadPawn is used by the save manager to load pre-existing pawns, it does this by passing SpawnPawn data.

SpawnPawn

Spawn pawn instanciate the pawns depending on their typing. This is done through checks against the data stored and separated from the object data scriptable object. This data is then compared against the data created in PawnDefinitions, specifically the struct called MPawnObjects which holds the different types of pawns.

SetPawnTypes

There is a range of pawn types that can be chosen from SpawnPawn function, these include buildings, minions, items, enemies, rewards, and mana stores. Some of these pawns are linked to spawning depending on chances such as items from rewards. This will then get a reference to the items name from the buildings chance scriptable object. This name is then compared to the relevant item stored in m_itemData and then spawned.

Costs

This function handles the cost related side of spawning pawns. This is an important class since the progression of the game is linked to resource collection.

UML

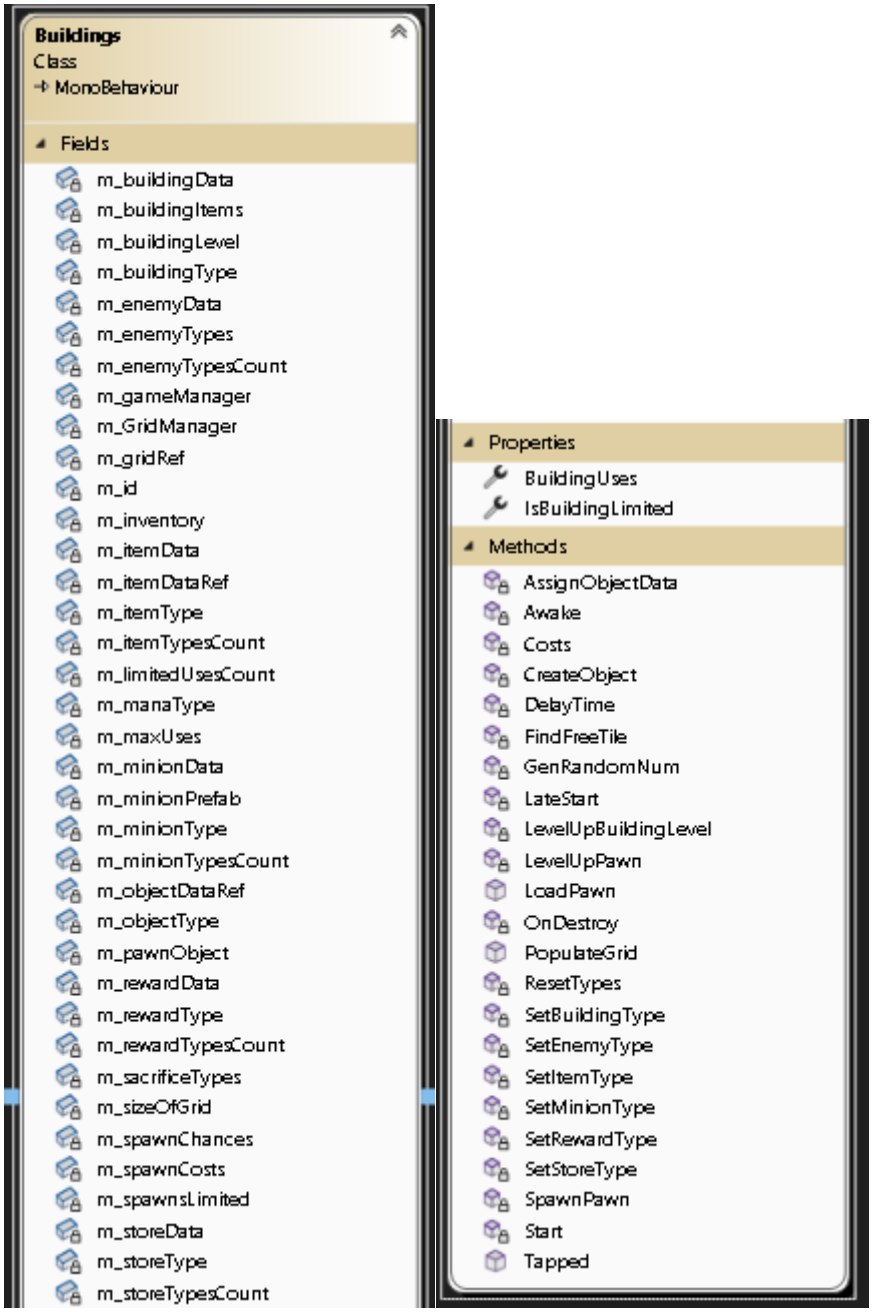


Figure 5: Second half of Building UML

Figure 6: First half of Buildings UML

Pawn Merging

Merging is an important part of the core gameplay loop as this is how the player interacts with the world. This means quite a few pawns need to access this component. The point of this component is to allow two pawns to combine if certain conditions are met.

Functions

ChangePawnVisual

When two pawns are merged the visual representation of the pawn needs to be changed, this function destroys the old version and instantiates the next one. This is also called at the start of the class to set the pawn's visual aspect.

Merge

This is the main function of the class; it handles the functionality of checking if both of the pawns are of the same type and level. If they are it will change their visual and updates its home tile and level. This is also used to fight enemies by taking the minion's damage stat.

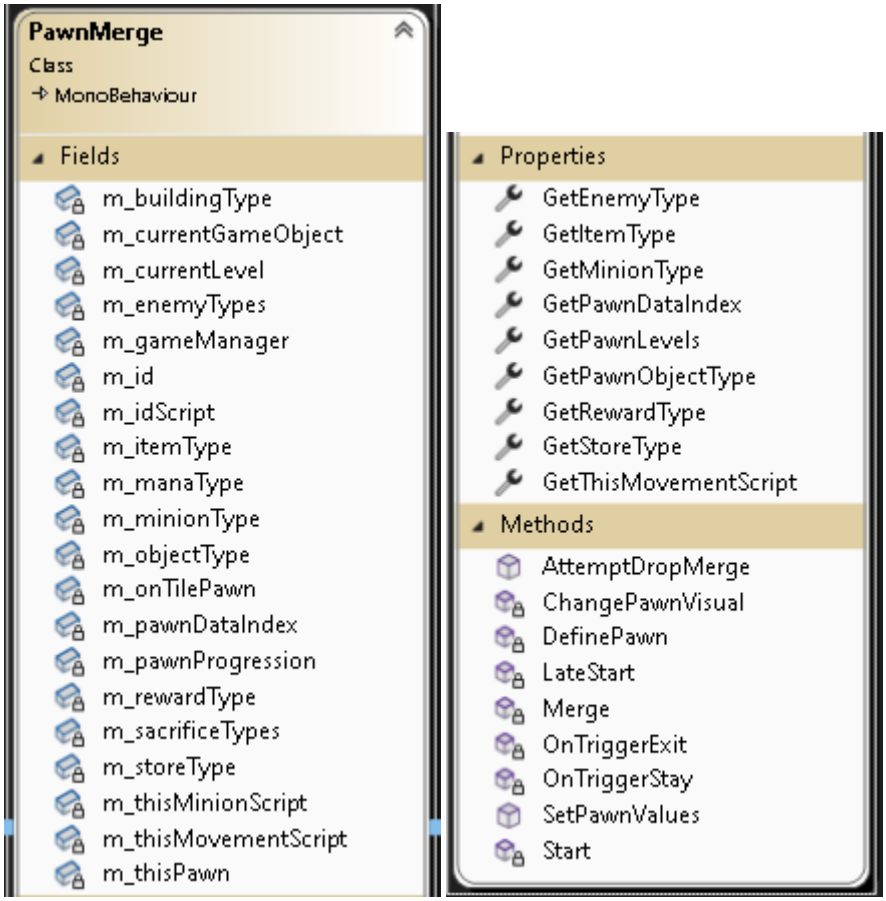
AttemptDropMerge

This is a public function that calls merge, it will only be able to call merge if certain criteria are met which is if the m_onTilePawn has a reference. This function is called by PawnMovement script when the pawn is dropped.

OnTrigger Events

These events are called by the pawns collider when it collides with other objects. These set and clear the collided objects to the variable m_onTilePawn. These two events are OnTriggerStay and OnTriggerExit.

UML



Mana

This class handles the mana generation of pawns. It listens to events which is called on a timer by the inventory class. Mana is an important part of the game as it is part of the core gameplay loop since it cost mana to spawn minions.

Events

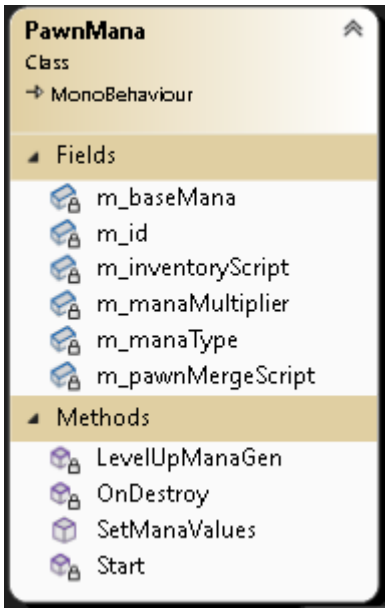
- OnPawnLevelUp

Functions

LevelUpManaGen

This function is called when the event this class is listening to is called. This will increase the pawn's base mana its generating by its multiplier. These values are then sent to the inventory.

UML



Sacrifice

Sacrificing is another important part of the gameplay loop as it one of the ways the player can upgrade their cult.

Events

- onPawnLevelUp
- PawnSacrifice

Functions

LevelUpValues

This increases the pawn's sacrificial value by its multiplier. This gives the player an incentive to level up their minions before they sacrifice them. This function is called by the class listening to the event onPawnLevelUp

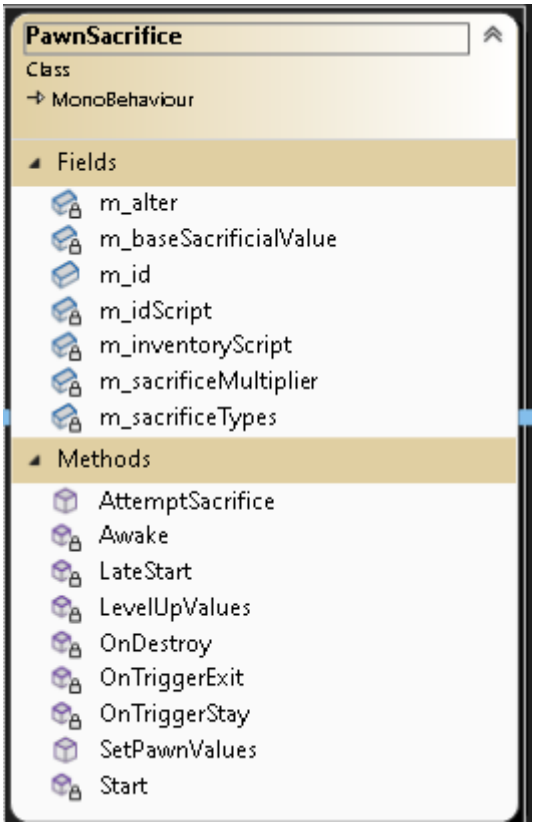
AttemptSacrifice

This function is runs the event PawnSacrifice and sends over its variables to attempt to sacrifice itself.

OnTrigger Events

This class makes use of the game objects collider to run the events called OnTriggerStay and OnTriggerExit. These events assign the object they are colliding with to m_alter if it has the script AlterSacrifce attached. OnTriggerStay sets m_alter to null.

UML



Store

Mana stores act as a way to increase the total amount of mana the player can have at one time.

Event

- onPawnLevelUp

Functions

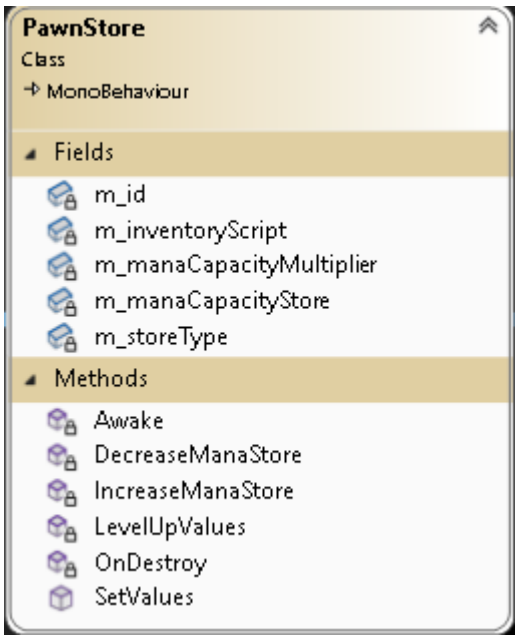
LevelUpValues

This increases the capacity of the pawn by its multiplier. It listens to the event called OnPawnLevelUp. Before increasing it takes away its value form the inventory script.

Changing Mana Store Values

This encompasses two functions called IncreaseManaStore and DecreaseManaStore. These increase/decrease the value stored in the inventory script.

UML



Enemy

Enemies are an important part of the gameplay loop since their loot drops is the only way the player can have access to runes that the player can spend on buying buildings.

Functions

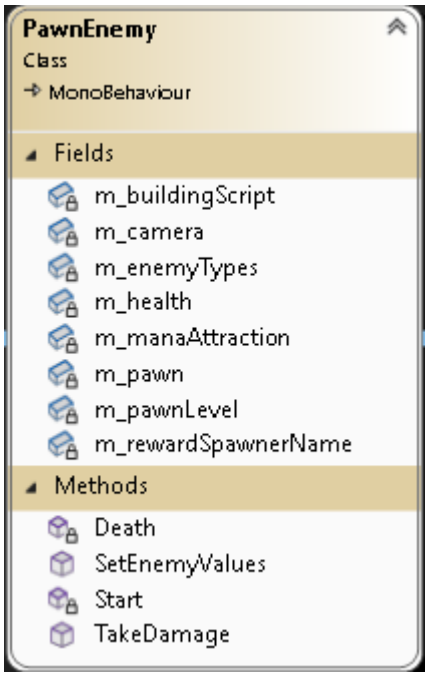
TakeDamage

This public function is called within the Merge component's merge function. It gets passed in a float that is applied to the enemy's health. If the health reaches zero, the enemy dies.

Death()

This is called by the TakeDamage function, when the pawn dies it calls its building script to spawn a chest as a reward for killing it.

UML



Minion

This class handles all specific data about a minion, an example of this is storing data related to damage. If the minion's level is below 2 then it will not be able to do damage since it is a crafting component.

Events

- onPawnLevelUp

Functions

LevelUpValues

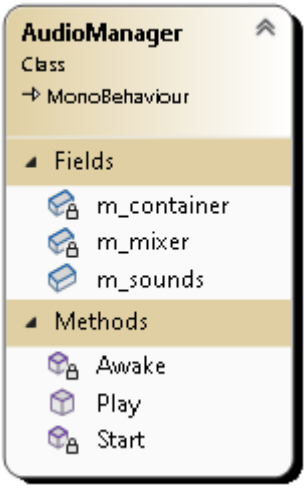
This levels up the damage value by its modifier. This function is run from the event called onPawnLevelUp.

UML

Audio Manager

This contains and plays all of the sounds used in the game, it loads all of the data stored in sounds and allocates the data to clips. These clips are then linked by name and can be played by calling Play() and passing in the name of the clip.

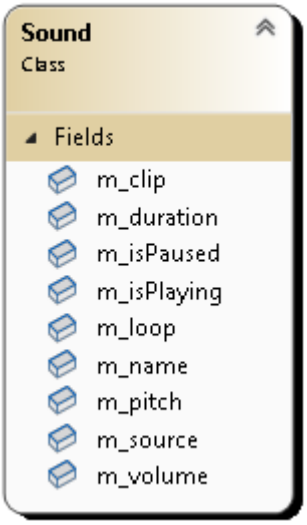
UML



Sound

This acts as a container for the data related to the sound clip.

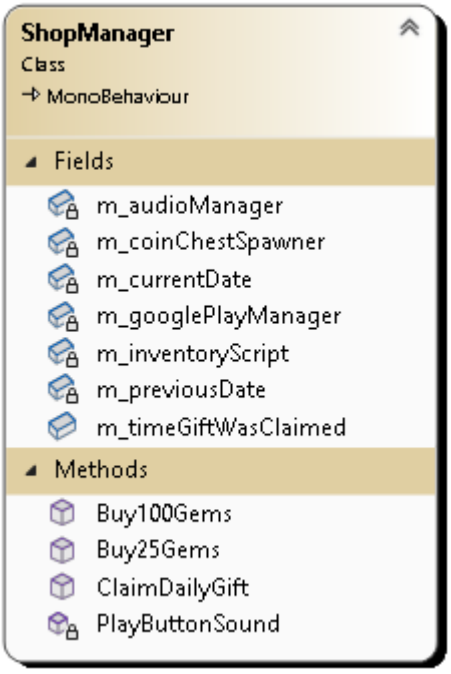
UML



Shop

The shop is currently set up using dummy logic to simulate the mechanics behind it. The player will be able to buy gems with this dummy logic. These gems can help them progress through the game without having to spend time waiting for mana or trying to get certain loot. To reward players for playing the game, the shop will also have coins as an option to buy items, but this will be limited to certain items. Through the use of a time system, daily free loot can be used to get the player to log in everyday to aide in progressing their cult.

UML



Inventory

This class handles all interactions with storing data related to mana, gems, cult value, and coins. It then updates the UI to represent these changes.

Functions

GenerateMana

This function is called on a loop over a delay as to not update the UI too many times, this will help the performance of the game. When called the capacity is checked against its store and if it is smaller, it will generate mana using the related property.

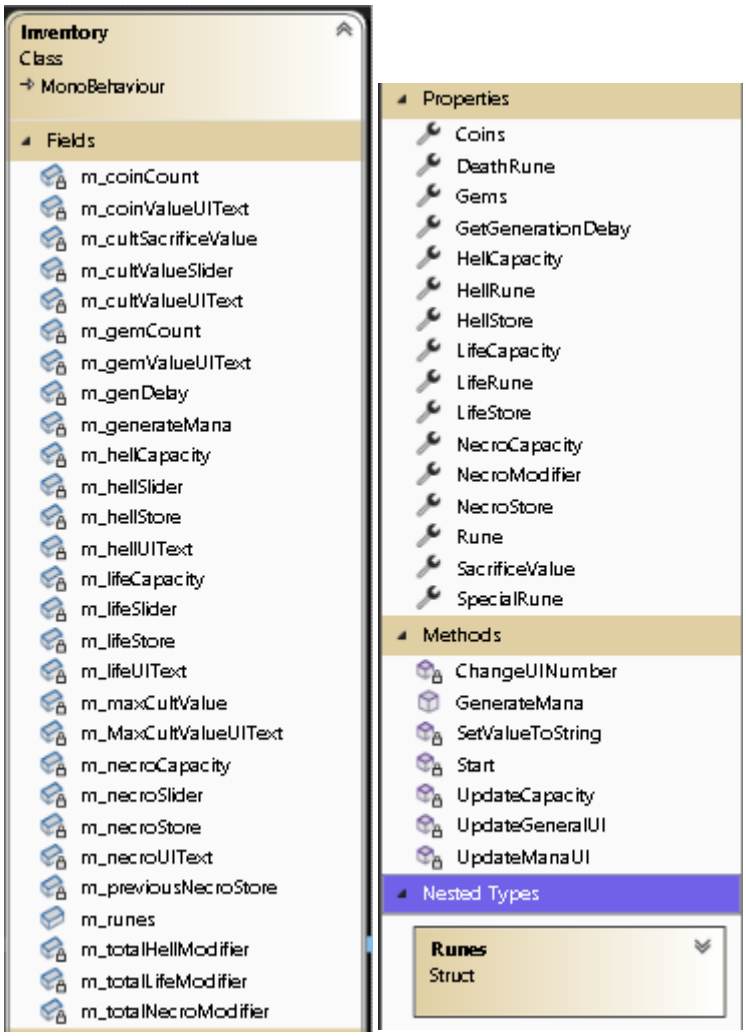
UpdateManaUI

Similar to GenerateMana, this gets called on a delayed loop to increase the game's performance. This gets the updated mana numbers and updates their respective UI elements in the canvas.

UpdateGeneralUI()

This is called whenever specific values are updated using the class's properties. These can be access by other classes, but this allows the UI to be updated only when the value has actually changed.

UML



Pawn Definitions

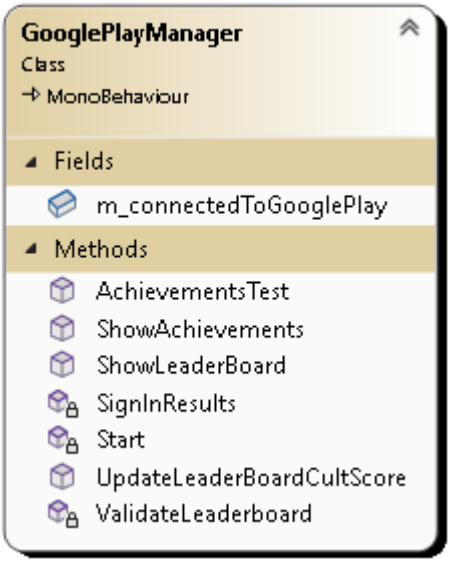
PawnDefinitions is a struct that contains a wide range of enums that are used in the scriptable objects and throughout all of the scripts to compare and set pawn types.

UML

GooglePlayManager

This manager handles all outputs to GooglePlay, this includes completion of achievements, high score values, as well as showing tables dictating the data stored for these.

UML



Scriptable Objects

Scriptable objects are a data container implementation used in Unity. These can allow the reduction of copies of values as one scriptable object can store unchanging values such as the mana type of each minion. Scriptable objects work slightly different to components as they must be attached to each game object through assets.

Here is an example of this being created in the engine.

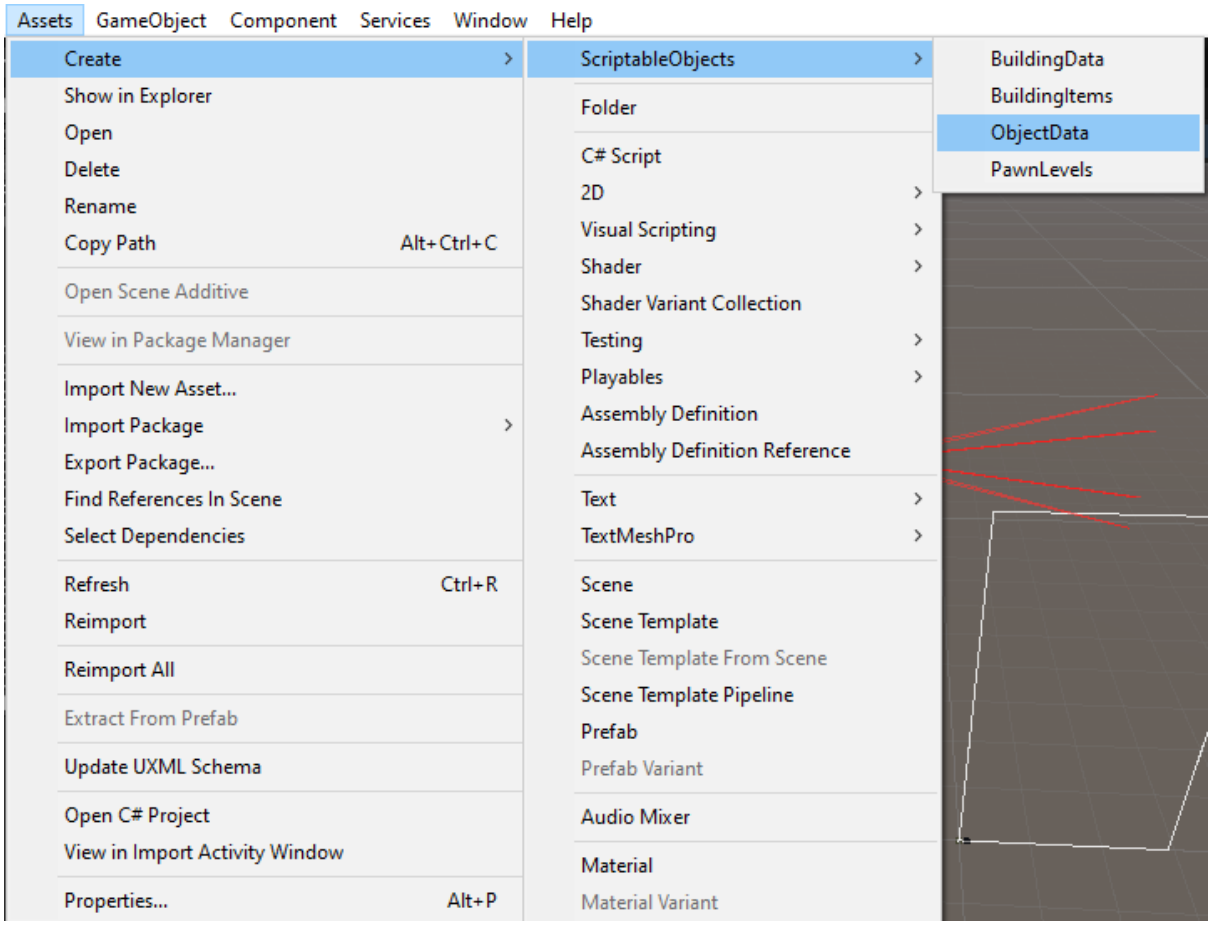
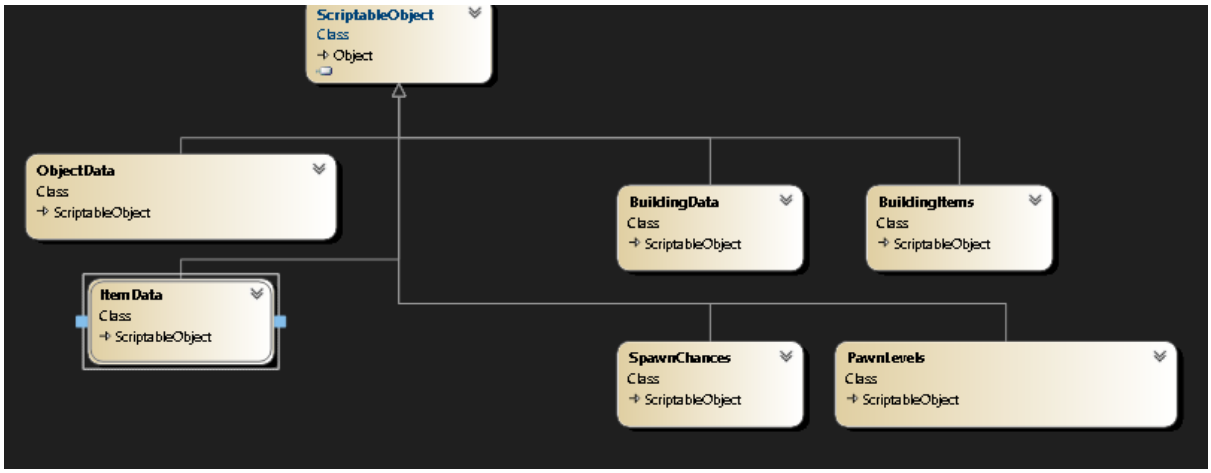


Figure 7: Screenshot of Unity showing how to create custom scriptable objects.



Object Data

Object data is the main scriptable object in the game, this stores data about all of the pawns. These are held in lists depicting the relevant data that can be refenced. This helps performance due to all buildings don't have to have their own version of the data.

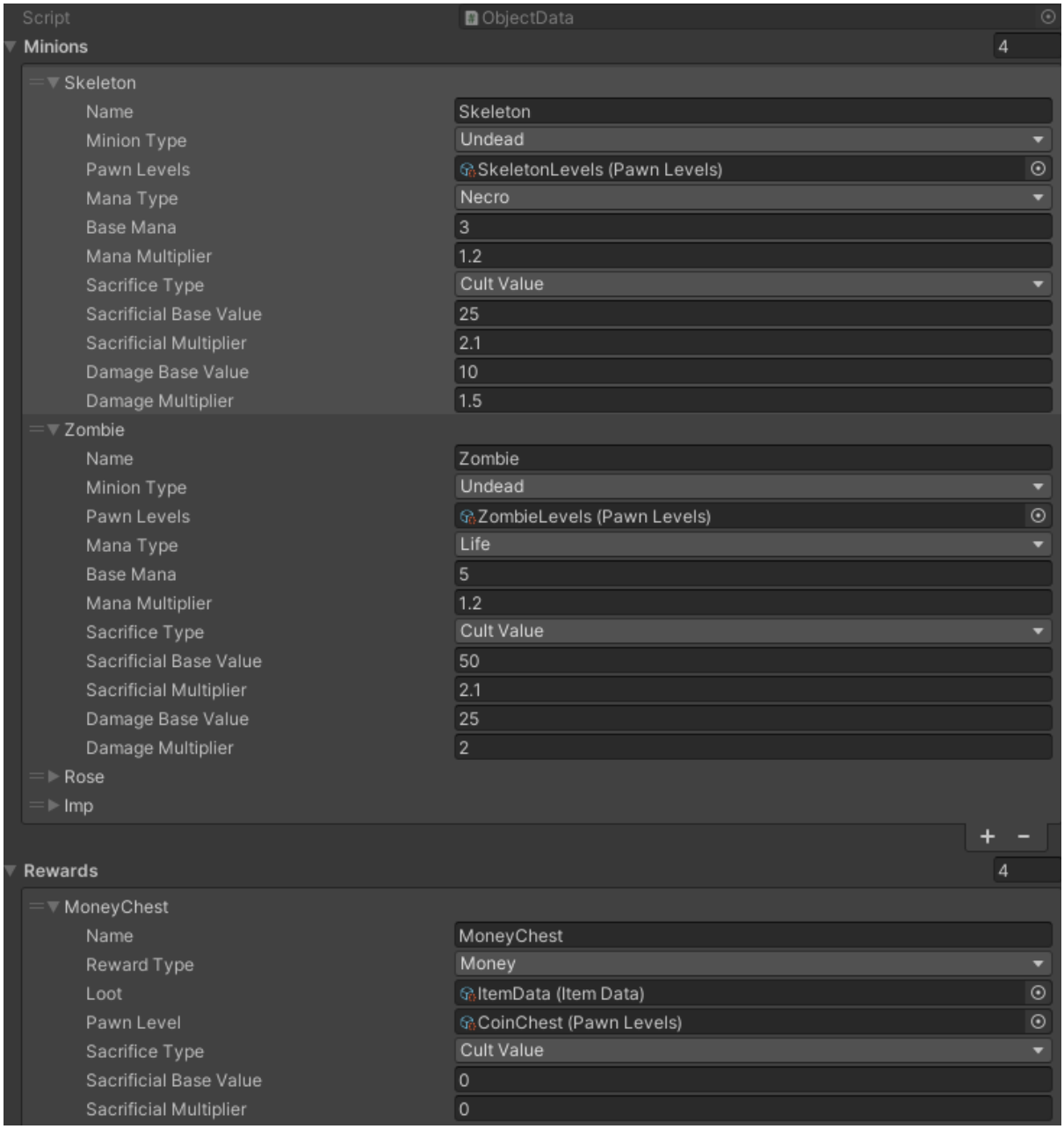
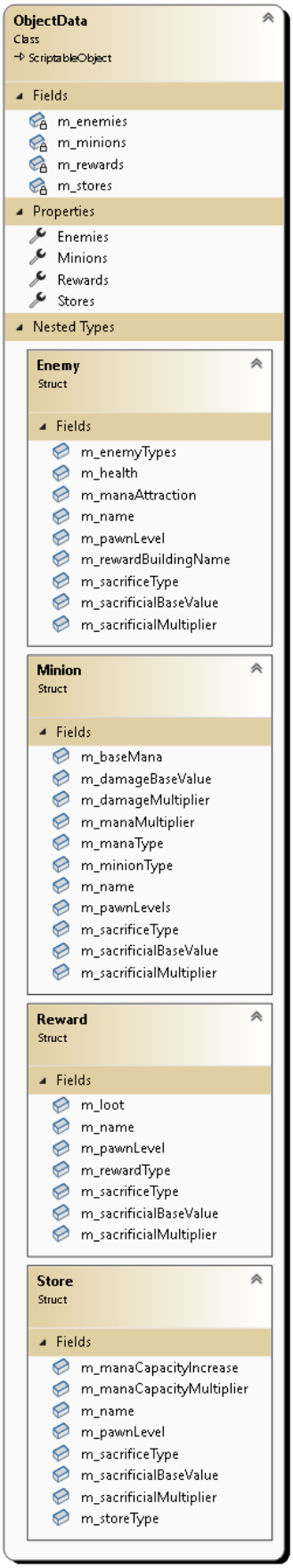


Figure 8: Screenshot of the implementation of object data in engine

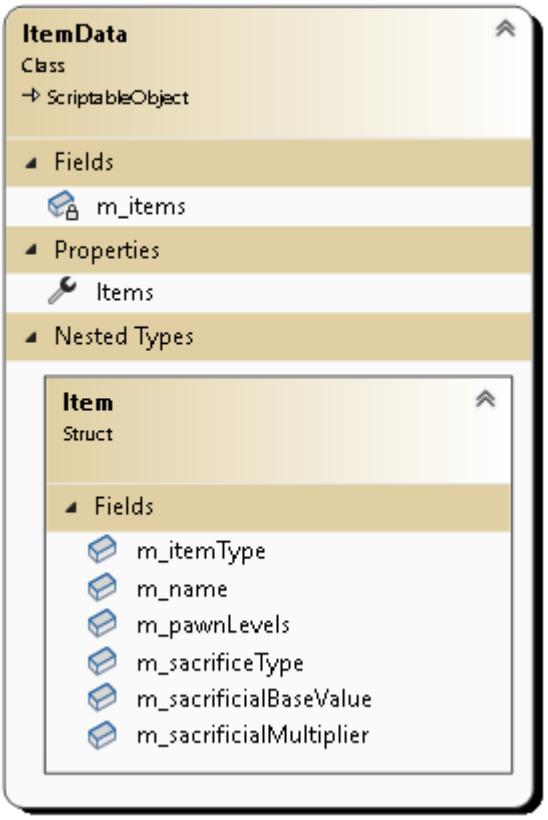
UML


ObjectData
Class
ScriptableObject
Fields
m_enemies
m_minions
m_rewards
m_stores
Properties
Enemies
Minions
Rewards
Stores
Nested Types
Enemy
Struct
Fields
m_enemyTypes
m_health
m_manaAttraction
m_name
m_pawnLevel
m_rewardBuildingName
m_sacrificeType
m_sacrificialBaseValue
m_sacrificialMultiplier
Minion
Struct
Fields
m_baseMana
m_damageBaseValue
m_damageMultiplier
m_manaMultiplier
m_manaType
m_minionType
m_name
m_pawnLevels
m_sacrificeType
m_sacrificialBaseValue
m_sacrificialMultiplier
Reward
Struct
Fields
m_loot
m_name
m_pawnLevel
m_rewardType
m_sacrificeType
m_sacrificialBaseValue
m_sacrificialMultiplier
Store
Struct
Fields
m_manaCapacityIncrease
m_manaCapacityMultiplier
m_name
m_pawnLevel
m_sacrificeType
m_sacrificialBaseValue
m_sacrificialMultiplier
m_storeType

Item Data

Item data is a similar scriptable object that serves the same purpose as object data but instead it holds data related to just items. This allows entries within Object Data to reference items. This is used for the reward system.

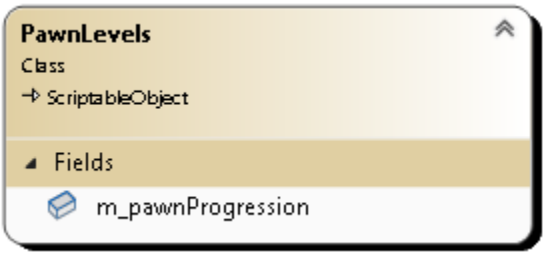
UML



Pawn Levels

Pawn levels contains all progression that a pawn would make when merging. This progression is a list of prefabs of the objects that will be used when changing the visual representation of the pawns. Not all the pawns use multiple elements but instead just one index. To progress through the indexing the level of the pawn is used to get a specific index of the pawn's level in its linked pawn level data.

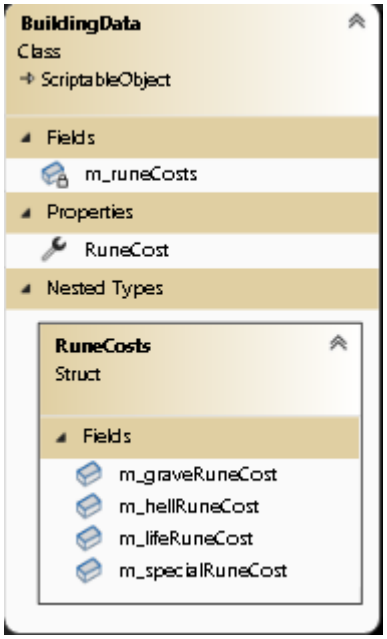
UML



Building Data

This scriptable object stores data about the costs that building use. This is mainly used by menu buildings since they will be spawning minion spawners and mana storage.

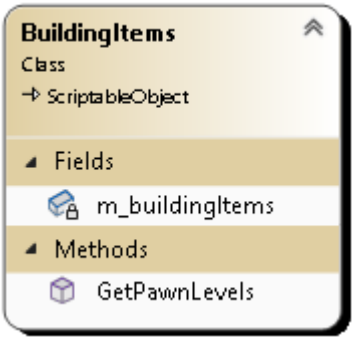
UML



Building Items

This scriptable object holds the data related to the building's visual progression of its levels.

UML



Controls

To allow flexibility in the game's development the project uses Unity's new input system to design the input controls. This is done by creating touch control actions and binding events to them. These actions are then handled by the InputManager script.

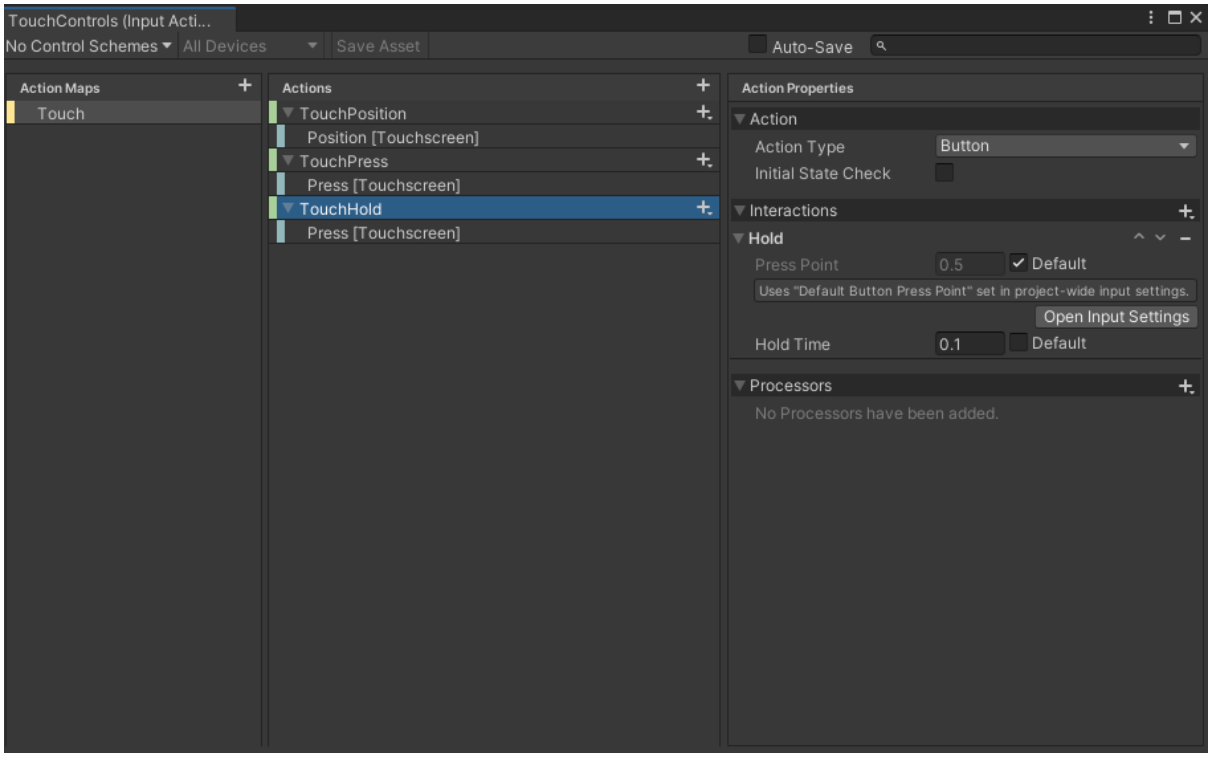


Figure 9: Touch control actions

Events

- m_touchPressAction
- m_touchHoldAction

Functions

GetTouchPosition

This used InputAction to get where the player has pressed on the screen, this is used by the Touch functions to process the input further.

TouchPressed

This sends a raycast to an object's location using the camera's position related to the press to get the interactable script of a pawn, this will then call the TouchPerformedInput of the class and pass it the position of the touch.

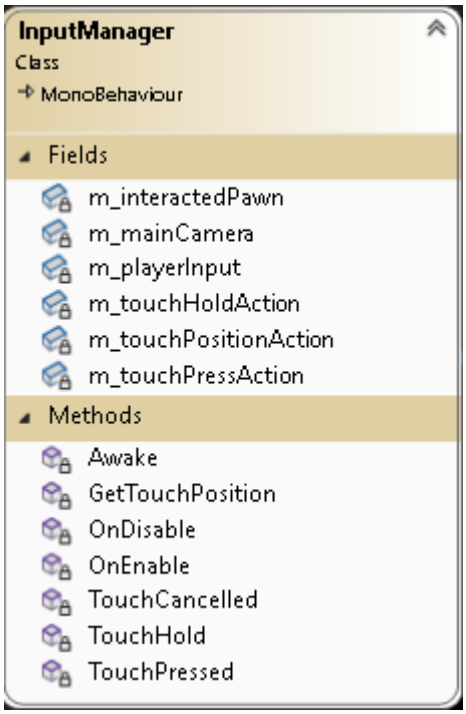
TouchHold

Similar to TouchPressed, except this function handles holding action and will call the function HoldPerformedInput on the interactable script of the collided pawn

TouchCancelled

This clears the current stored pawn in m_interactedPawn and then calls TouchCancelledInput on the interacted pawn.

UML



Offline Progression

Since the game's genre is an idle game, a lot of the progression happens offline. These offline sessions allow the player to gain materials, specifically generating mana to progress further when they are back online. This means that an offline progression mechanic is integral for the game. This is achieved through the SaveManager which uses the saved modifiers to increase the mana that would have been generated if the application were closed. This is done through checking how many seconds had passed from the application's last save point.

Analytics

Analytics for this project was passed onto Google Play Games. This is useful due to its ability to give a wide range of data that can be easily added to other areas of the game. Additionally using Google's services can allow for more features to be easily implemented such as a leader board or achievement.

Figures

Bibliography

Unity, 2023. *Introduction to components.* [Online]
Available at: https://docs.unity3d.com/Manual/Components.html
[Accessed 30 March 2023].

Unity, N.D.. *Event System.* [Online]
Available at: https://docs.unity3d.com/560/Documentation/Manual/EventSystem.html
[Accessed May 2023].

Unity, N.D.. *MonoBehaviour.Update.* [Online]
Available at: https://docs.unity3d.com/ScriptReference/MonoBehaviour.Update.html
[Accessed May 2023].