Unity Framework MobileApp TDD

1. [Ads Manager](#_z1x1ggnyl80a)
2. [Gesture Detector](#_rgha5s9wg54p)
3. [Sound System](#_7dqbxiua9if1)
4. [Save System](#_6jmd25yg76sb)
5. [Pooling System](#_dgorn9qxa04m)

Ads Manager

# Introduction:

The "AdsManager" class is responsible for managing Ads in the game. It uses the Singleton pattern to ensure that there is only one instance of itself present in the game at any given time.It has two variable for the Android and IOS id and two array for contain all the Rewarded Buttons and the Interstitial Buttons.

The Rewarded Buttons use the RewardedAdsButton class, while the Interstitials use InterstitialAdsButton

# Monobehaviour:

In its “Awake” method,set both arrays with all useful objects and initializes them thanks to the function "InitializeAds".

# Methods:

After initializing the Ads you will have two possible messages.

After this phase the other two classes are used to start and load the advertisements.

# 

# Method list:

## AdsInizializer method list:

* “InitializeAds":

Get Game id and check the variable \_testMode and initialize Ads.

* “OnInitializationComplete”:  
  If the initialization is successful, load the ads.
* “OnInitializationFailed”:  
  if the initialization is failed, print error message

## InterstitialAdsButton and RewardedAdsButton

* “LoadAd”:  
  Load content to the Ad Unit.
* “ShowAd”  
  Show the loaded content in the Ad Unit.
* “OnUnityAdsAdLoaded”:  
  Optionally execute code if the Ad Unit successfully loads content.
* “OnUnityAdsFailedToLoad”:  
  Optionally execute code if the Ad Unit fails to load, such as attempting to try again.
* “OnUnityAdsShowFailure”:  
  Optionally execute code if the Ad Unit fails to show, such as loading another ad.
* “OnUnityAdsShowStart”:  
  Execute the function content when advertising starts.
* “OnUnityAdsShowClick”:  
  Execute the content of the function when you click the button to start the advertisement.
* “OnUnityAdsShowComplete”:  
  Run the content of the function when advertising is over

Gesture Detector

# Introduction:

The "GestureDetector" class allows you to understand what type of input you use. It uses the Singleton pattern to ensure that there is only one instance of itself present in the game at any given time. In this script we have an enumeration, a class not Monobehaviour and a class Monobehaviour. In the enum there are all possible types of Gesture, the class not Monobehaviour contains all the main variables and instead in the class Monobehaviour there are all the useful functions

# Gesture class:

The Gesture class is a simple container class that stores the start position, end position, current gesture type, and a list of Touch objects. The class also provides setters and getters for these properties, and a default constructor that sets default values for these properties.

# Gesture type:

The GestureType enumeration defines the different types of gestures that the system can detect.

# Gesture Detector:

The GestureDetector class is a singleton class that extends the MonoBehaviour class and is responsible for detecting and tracking gestures. It has a public Gesture object that stores the current gesture information, a minDistance variable that is used to determine the minimum distance required for a swipe gesture, and several other private variables that are used to track the state of the system.

# Monobehaviour:

The “Awake” method of the GestureDetector class is used to initialize the minDistance variable based on the height of the screen. The “Update” method is called every frame, it calls the “VerifyInput” method, which is responsible for detecting and tracking gestures.

# 

# Method list:

* “VerifyInput”:  
  it uses a switch statement to determine the number of touches currently present, and calls the appropriate method based on the number of touches.
* “ZeroTouch”:  
  It is called when there are no touches present, it sets the standard values for the Gesture object.
* “OneTouch”:  
  It is called when there is only one touch present, it uses a switch statement to determine the phase of the touch, and calls the appropriate method based on the phase. It is also called the "CheckTapCounter" method.
* “CheckTapCounter”:  
  It is used to check if the current gesture is a tap or a double tap, by keeping track of the last gesture and the time of the last gesture, by using a coroutine.
* “TwoTouch”:  
  It is called when there are two touches present, it is used to detect pinch in and out gestures by comparing the distance between the two touches.
* “AssignTouchList”:  
  It is used to assign the touch list to the gesture object.

SoundSystem

# Introduction:

The "SoundManager" class is responsible for managing sound in the game. It uses the Singleton pattern to ensure that there is only one instance of itself present in the game at any given time. It has various variables and properties to handle background scene music, sounds and effects, and the volumes of these sounds. It also has four lists variables that determinates multiple sounds can be played simultaneously.

# Monobehaviour:

In its "Awake" method, the class sets itself up based on the desired sound settings and loads the volume values saved in the "PlayerPrefs". In its "Update" method, it checks for a new scene, verifying if there is a sound scene or not and checks if music is playing

# Methods:

The class has several methods for muting and unmuting sounds, as well as for saving volume values to the "PlayerPrefs". It also has several functions for playing sounds and music, both once and in a loop adding and removing AudioSource component. In addition, it has the ability to adjust the volumes of sounds and scene background music.

# Methods List :

* "SetVolumesByPlayerPrefs":

loads volume values from the "PlayerPrefs"

* "MuteSounds":

mutes main volume and saves the previous value to the "PlayerPrefs"

* "MuteEffect":

mutes effects volume and saves the previous value to the "PlayerPrefs"

* "MuteSceneSound":

mutes scene background music volume and saves the previous value to the "PlayerPrefs"

* "UnMuteSounds":

restores main volume to its previous value

* "UnMuteEffect":

restores effects volume to its previous value

* "UnMuteSceneSound":

restores scene background music volume to its previous value

* "SaveVolumesIntoPlayerPrefs":

saves current main, effects, and scene background music volumes to the "PlayerPrefs"

* "PlayMusic":

plays a given music clip

* "PlayEffect":

plays a given sound effect clip

* "PlayMusicLoop":

plays a given music clip in a loop

* "PlayEffectLoop":

plays a given sound effect clip in a loop

* "StopMusic":

stops all music from playing

* "StopEffect":

stops all sound effects from playing

* "StopMusicLoop":

stops all looping music from playing

* "StopEffectLoop":

stops all looping sound effects from playing

* "CheckNewScene":

checks for a new scene and plays the appropriate background music

* "CheckMusicPlaying":

checks if any music is currently playing

* "SetVolumesByEditor":

sets the main, effects, and scene background music volumes based on editor settings

SaveSystem

# SaveSystem Class Introduction:

# The "SaveSystem" class uses the Singleton pattern to ensure that there is only one instance of itself present in the game at any given time. It has several methods for saving and loading data using a binary file format, checking for the existence of a directory, and getting the full path of a file based on the file name and directory. A “SaveClassParent” that has a variable, named “data” of T type where T is a class derived from “DataClassParent”, can use the following methods of this singleton:

# Methods:

* "Save":

serializes the given object to a binary file in the specified directory

* "Load":

deserializes the given binary file in the specified directory and returns the resulting object

* "DirectoryExists":

checks if the specified directory exists

* "GetFullPath":

returns the full path of the specified file in the specified directory.

# SaveClassParent Introduction:

# The "SaveClassParent" class extends "MonoBehaviour" and uses T to ensure that T is a class derived from "DataClassParent". It has variables and properties to handle the data to be saved, the name of the file in which the data will be saved, and the directory in which the file will be saved. It has several methods to save and load data, check for the existence of a directory, and retrieve the data.

# Methods:

* "SaveClass":

uses the "SaveSystem" singleton to save the data to a file in the specified directory

* "LoadData":

uses the "SaveSystem" singleton to load data from a file in the specified directory

* "DirectoryExist":

uses the "SaveSystem" singleton to check if the specified directory exists

* "GetData":

returns the data stored in the class

# DataClassParent Introduction:

“DataClassParent” is a class empty that can be derived and by anyone can create custom classes that has methods and variables that i want to save or load using SaveSystem

PoolSystem

# ObjectPoolable Introduction:

The ObjectPoolable script can be attached to a GameObject in Unity, and it contains a series of functions that manage the object's state when it is either enabled or disabled. For example, when the object is disabled, the script will automatically set the object's parent to a specific object that is designated as the "deactivated parent." This script also allows you to control the active state of the object and set a "activated parent"

# MonoBehaviour:

* OnEnable : A virtual function that is called when the object is enabled. It can be overridden in child classes to add custom behaviour when the object is enabled.
* AttachToActivatedParent : A function that attaches the GameObject to the parentWhenActivated object.
* AttachToDeactivatedParent: A function that attaches the GameObject to the parentWhenDeactivated object.
* OnDisable : A virtual function that is called when the object is disabled. It can be overridden in child classes to add custom behaviour when the object is disabled.

# Methods

* SetParents: sets the parentWhenDeactivated and parentWhenActivated values to the passed in objects
* IsActive: Returns whether the object is active in the Unity scene hierarchy
* DeactiveAfterTime: Coroutine that after dieTime seconds, it will deactivate the object and set it to be a child of the parentWhenDeactivated object
* OnSpawn: A virtual function that is called when the object is spawned. It can be overridden in child classes to add custom behavior when the object is spawned

# ObjectPooler Introduction:

The ObjectPooler script is responsible for instantiating and managing a pool of objects. It uses a list of ObjectToPool objects that specify the objects that should be pooled and their properties. This script create the pool of objects, and it can return a pooled object when it's requested.

# MonoBehaviour:

* Awake() : function that is called when the script instance is loaded, it calls InitPoolLists()

# Methods

* InitPoolLists() : function that is used to instantiate the objects that should be pooled and sets them up with the appropriate properties.
* GetPooledObject<T> : function that takes a type and returns an object of that type that is not currently in use
* SpawnFromPool<T>: function that takes a type and a position and rotation, and returns an object of that type from the pool and sets its position and rotation to the passed in values. If all objects of that type are in use and the pool is not expandable, it will return null.