PEARL

INTRODUZIONE

COS'E' PEARL

Sviluppo da tempo in Unity e durante questi anni ho creato molte classi per semplificare la gestione e la creazione di giochi. Pearl è l'insieme delle mie conoscenze: un Toolkit che facilità lo sviluppo di Unity rendendo l'architettura del gioco molto modulare e quindi meno affetta a errori.
Pearl usa la versione aggiornata di NET (la 4.0), quindi, versione di NET inferiori non faranno funzionare Pear.

STRUTTURA

Pearl è diviso in quattro cartelle principali:

- La cartella "External" consiste in classi che l'utente dovrà modificare durante la creazione del gioco.
- La cartella "Prefabs" contengono prefabs che l'utente potrà mettere direttamente nel gioco-
- Scripts: Il cuore pulsante di Pearl, contengono le classi che sono usate per semplificare il lavoro allo sviluppatore.
- Resources: La cartella che contiene le risorse del gioco (accessibili tramite il metodo Resources.Load<T>(...)) che vengono istanziate dinamicamente.

Dentro la cartella "Scripts" ci sono cartelle che svolgono una specifica funzione. In particolare una cartella contiene numerosi scripts che lavorano insieme che lavorano insieme per risolvere un determinato aspetto del gioco.
Pearl in sostanza è un insieme di funzioni.
Qui sotto si descriverà ogni funzione in modo da poterla usare per ogni progetto.

EVENT SYSTEM

Gli scripts contenuti nella cartella Scripts/Events System sono il cuore di Pearl e consentono di avere tre caratteristiche:

- Estrema modularità.
- Sistema di comunicazione.
- Controllo totale dell'accesso.

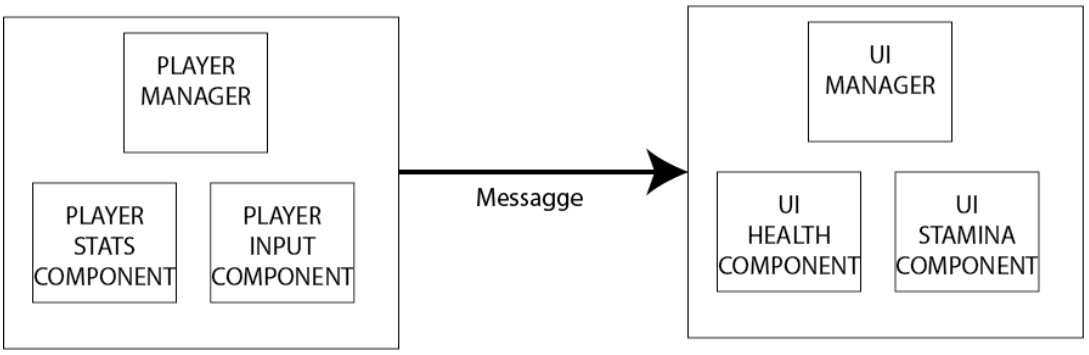
Estrema modularità

Il concetto in cui si basa Pearl è che ci sono diverse entità più o meno grosse che governano un determinato aspetto del gioco (esempio: L'entità nemico governa tutti gli aspetti del nemico).
Queste entità collaborano tra di loro mandando dei messaggi:
per esempio in un gioco ci può essere l'entità UI e l'entità Player. Quando l'entità Player verrà colpita, essa deve mandare un messaggio all'entità UI per aggiornare la barra di salute.

In Pearl questa astrazione è costruita tramite due script:

- *LogicalManager*/*LogicalSimpleManager*
- *LogicalComponent*

Ogni entità è rappresentata da un *LogicalManager*/*LogicalSimpleManager* (alias *manager*) e da diversi *logicalComponent*. Specificatamente un manager può avere zero o più componenti.
I *logicalComponents* rappresentano particolari aspetti dell'entità: per esempio l'entità Player potrebbe avere un componente legato alla gestione delle statistiche e un componente legato al movimento.

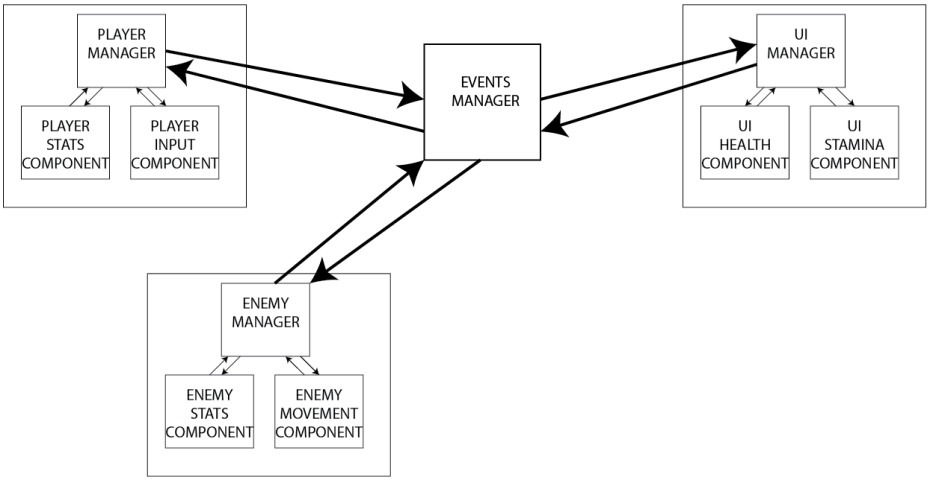


Da questo momento entità e *Manager* rappresenteranno lo stesso concetto. I *managers* sono figli di MonoBehaviour, quindi sono componenti di un GameObject; invece i *logicalComponents* non sono figli di MonoBehaviour.

Sistema di comunicazione

I *managers* comunicano con gli altri manager tramite un gestore eventi centrale (*Events Manager*) che è governato da azioni.
Precisamente un *manager* si registra all'evento associato ad una particolare azione. Quando qualcuno richiama l'azione, si attivano tutti i manager iscritti per l'evento associato a quella azione.
Per esempio l'entità UI registra il suo metodo "*ChangeBarHealth*" all'evento associato all'azione "*ChangeHealth*": quando il player verrà ferito, chiamerà l'evento associato all'azione "*ChangeHealth"*, che richiamerà tutti i metodi associati ad esso, tra cui "*ChangeBarHealth*" dell'entità UI.

Invece i *logicalComponents* di un particolare *manager*, non possono comunicare tra di loro ma possono comunicare solo con il *manager* stesso, tramite interfacce ben precise.



Controllo totale dell'accesso

Tramite il sistema modulare e il sistema di comunicazione, le entità hanno solo due tipi di accesso:

- Esterno (Entità-Entità): tramite la registrazione di un particolare evento in *Events Manager*
- Interno (Entità-componenti): tramite l'interfaccia che i componenti usano per dialogare con il loro Manager.

Si analizza ora gli script che compongono questa astrazione.

EVENT ACTION

Questo enumeratore (che si trova sulla cartella External/Events) contiene le azioni.
L'utente deve inserire un nuovo elemento dell' enumeratore per generare una nuova azione che i manager potranno usare per ascoltare l'evento associato ad essa.
I primi tre elementi sono di default, non si devono eliminare:

- CallPause,
- CallFrameRate
- NewScene
- GetInputEntryMenu
- GetInputReturnUI

EVENT MANAGER

Questa classe consente i vari *managers* di scambiarsi messaggi tramite eventi collegati ad azioni.

AddMethod

Questa funzione consente di aggiungere un metodo all'evento associato ad una particolare azione.

Input

- L'evento da associare il metodo: è rappresentato da un elemento dell'enumeratore EventAction.
- Il metodo stesso, il metodo può avere 0, 1 o 2 parametri di input.

RemoveMethod

Questa funzione consente di eliminare un metodo all'evento associato ad una particolare azione.

Input

- L'evento associato al metodo da eliminare: è rappresentato da un elemento dell'enumeratore EventAction.
- Il metodo stesso, il metodo può avere 0, 1 o 2 parametri di input.

CallEvent

Questa funzione consente di chiamare un'azione e quindi il suo evento corrispondente. L'evento chiamato chiamerà tutti i metodi associati a lui.

Input

- L'evento da chiamare.
- I parametri da associare all'azione, un'azione può avere 0,1 o 2 parametri.

LOGICAL SIMPLE MANAGER

E' una classe astratta ed è il *manager* più semplice: specificatamente il *manager* senza nessun *LogicalComponent*.
Questo *manager* dev'essere il padre dei manager semplici, ossia quelli senza componenti.

Subscrive Events

La funzione consente la creazione dell'interfaccia con *Events Manager*. Infatti, il figlio concreto della classe, nel metodo, dovrà sottoscrivere tutti gli ascoltatori agli eventi di *Events Manager*.
Il metodo è chiamato dal metodo Awake di Unity.

```
protected override void SubscribEvents()
{
    EventsManager.AddMethod<float>(EventAction.SendHealth, ReceiveHealth);
}
```

Remove Events

La funzione consente la rimozione degli ascoltatori ad *Events Manager*. Infatti, il figlio concreto della classe, nel metodo, dovrà descrivere tutti gli ascoltatori agli eventi di *Events Manager*.

```
protected override void RemoveEvents()
{
    EventsManager.RemoveMethod<float>(EventAction.SendHealth, ReceiveHealth);
}
```

OnAwake

Un metodo per serve per mettere all'utente tutte le sue istruzioni che vuole far girare nel metodo Awake di Unity (occupate dal manager)

LOGICAL MANAGER

E’ una classe astratta ed rappresenta un *manager* con diversi componenti.
Questo manager è figlio di *Logical Simple Manager* e dev’essere il padre dei *managers* con uno o più componenti.

CreateComponents

In questo metodo astratto, la classe concreta deve creare i componenti. I componenti si creano tramite un *dictonary <Type, LogicalComponent>* chiamato *listComponents*: la chiave rappresenta il tipo concreto del componente e il valore l’istanza del componente (tutti i componenti sono figli di *LogicalComponent*).

```
protected override void CreateComponents()
{
    listComponents = new Dictionary<Type, LogicalComponent>
    {
        { typeof(PlayerPowerComponent), new PlayerPowerComponent(power1, power2) },
        { typeof(PlayerAnimationComponent), new PlayerAnimationComponent(GetComponent<SpriteRenderer>()
        { typeof(PlayerMovementComponent), new PlayerMovementComponent(gameObject.GetInstanceID(), spee
        { typeof(PlayerStatsComponent), new PlayerStatsComponent(maxHealth) },
        { typeof(InputComponent<PlayerManager>), new InputPlayerComponent(NotifyPower, NotifyMovement,
    };
}
```

GetLogicalComponent<T>

Con questo metodo l’utente riesce a prendere un componente dal manager, al posto del T si deve mettere la classe del componente desiderato.

```
GetLogicalComponent<PlayerStatsComponent>().OnChangeHealth(actualHealth);
```

LOGICAL COMPONENT

E’ una classe astratta ed rappresenta un *Logical Component*. La classe dev’essere il padre che tutti i *Logical Components* devono avere.
La classe component non deriva da Monobehaviour.

STRUTTURA MANAGER

Visto che la classe Manager è una classe molto complessa, si vedano alcune convenzioni per semplificare il lavoro di gestione:

- Ogni metodo dev’essere privato tranne per due eccezioni: i metodi chiamati tramite l’interfaccia UI o se la classe è un Singleton (vedi parte Singleton).
- Suddividere la classe in due regioni: la prima regione, la regione di interfaccia, si devono mettere tutti i metodi che consentono di interfacciarsi con altre classi (l’event

manager, componenti, ecc...); la seconda regione, la regione logica, si devono mettere tutti i metodi che svolgono qualche funzione.

- Tutti i metodi di interfaccia, sono azioni, ossia sono funzioni che restituiscono void.
- I metodi della regione Logica devono iniziare con il prefisso "*Do-*"
- Nei metodi di interfaccia mettere i metodi *SubscribEvents*() e *RemoveEvents*().
- Nei metodi di interfaccia mettere tutti i metodi che sono sottoscritti all'event Manager. In questi metodi mettere il prefisso *"Receive-*". I metodi, al messaggio ricevuto, richiameranno un altro metodo della regione logica che effettivamente risponderà con l'adeguata funzione.
- Nei metodi di interfaccia mettere tutte le funzioni che generano un evento, ossia quei metodi che mandano un messaggio all'event manager: questi metodi hanno come unico scopo chiamare il metodo *CallEvent* di *Events Manager*.
- Nei metodi di interfaccia mettere le funzioni di interfaccia verso i componenti. In questi metodi mettere il prefisso *"Notify-*". Le funzioni, al messaggio ricevuto, richiameranno un altro funzione della regione logica che effettivamente risponderà con l'adeguata funzione.
- I figli di *manager* nel nome della classe devono avere il suffisso "Manager" (ES: PlayerManager).
- Tutti i Monobehaviour del gioco devono essere o figli di *LogicalManager* o figli di *SimpleLogicalManager*.

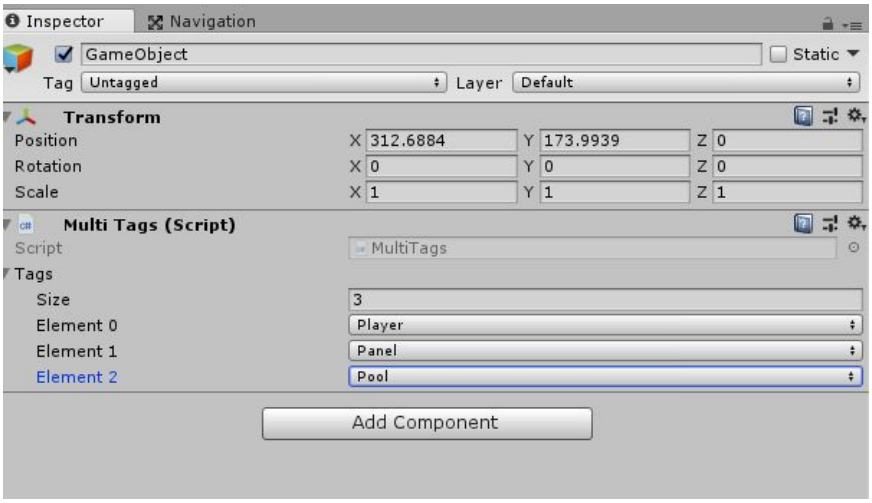
STRUTTURA COMPONENT

Visto che la classe Component è una classe molto complessa, si vedono alcune convenzioni per semplificare il lavoro dei gestione:

- Tutti i metodi di component sono privati ad eccezioni dei metodi che il *manager* dovrà attivare dall'esterno
- Passare nel costruttore i metodi "*Notify-*" (classe *Action<T,F,...>*) del manager e salvarli in delle variabili. Quando il componente dovrà notificare un messaggio al manager, lo notificherà tramite i riferimenti salvati nelle variabili.
- I metodi pubblici sono metodi che il *manager* accede per ordinare al componente di fare qualcosa. In questi metodi mettere il prefisso "Obey-"

MULTI TAGS

Gli scripts contenuti nella cartella Script/Multitags consentono l'uso di più tag in un GameObject.
Per abilitare i tags basta mettere in un GameObject il componente MultiTags.cs.



Dall’editor è possibile aggiungere tags nel componente.

TAGS

Per aggiungere tags si deve modificare lo script Tags.cs nella cartella external/tags. Questo script consiste in un enumeratore che contiene tutti i tags del gioco. Di default nell' enumeratore ci sono tre tags:

- Panel
- Pool
- Obstacle

Questi tags quali non devono essere eliminati perchè fanno funzionare gli altri script del toolkit.

MULTI TAGS MANAGER

In questa classe statica tutti le funzioni sono metodi di estensioni della classe GameObject (quindi l'oggetto GameObject avrà questi nuovi metodi):

FindGameObjectsWithMultiTags

Questo metodo statico consente di trovare i gameobjects con specifici tags.

Input

- Un boolean che specifica che i gameobjects trovati devono avere SOLO quei tag (se il booleano è false, i gameobjects possono avere anche altri tags).
- La lista di tag da ricercare.

Output

- una lista di GameObject che soddisfano i requisiti del metodo.

FindGameObjectWithMultiTags

Questo metodo statico consente di trovare un gameobject con specifici tags.

Input

- Un boolean che specifica che i gameobjects trovati devono avere SOLO quei tag (se il booleano è false, i gameobjects possono avere anche altri tags).
- La lista di tag da ricercare.

Output

- Il primo GameObject che soddisfa i requisiti del metodo.

HasTags

Questo metodo di estensione al GameObject consente di vedere se un particolare GameObject possiede dei tags specifici.

Input

- La lista dei tags da ricercare.

Output

- un booleano che indica se il GameObject possiede i tags specificati.

ReturnTags

Questo metodo di estensione al GameObject consente di avere i tags di un particolare GameObject.

Input

- La lista dei tags da ricercare.

Output

- Una lista di tags che possiede quel GameObject.

AddTags

Questo metodo di estensione al GameObject consente di aggiungere dei tags al suddetto GameObject.

Input

- La lista dei tags da aggiungere.

RemoveTags

Questo metodo di estensione al GameObject consente di eliminare dei tags al suddetto GameObject.

Input

- La lista dei tags da eliminare.

POOL SINGLETON

Il *pool singleton* è una classe statica nella cartella Script/Utility. Lo script ha una lista che contiene tutti i riferimenti ai sigletons dell'applicazione. I singletons devono essere *manager*. Perchè mettere i singleton quando c'è l'*events manager*? Semplicemente è stato deciso di mettere i singletons perchè alcune entità/classi sono molto generali (si pensa alla classe che gestisce l'audio) ed uniche, quindi è naturale *poterle* accedere istantaneamente piuttosto che usare due messaggi dell'*events manager* (il primo messaggio per chiamare l'istanza e il secondo per la risposta dell'istanza).
Nel singleton ci sono i metodi pubblici (si ricorda che il singleton è un *manager*), tali metodi sono metodi di interfaccia e consentono di essere richiamati da altre istanze.

Get<T>

Il metodo Get<T> restituisce l'unica istanza della classe specifica dal parametro generico T. Si ricorda che nella scena ci dev'essere solo un'istanza di quella classe.

Output

- L'istanza della classe specifica dal parametro generico T.

AUDIO SYSTEM

Nella cartella Script/Audio System ci sono script che semplificano l'accesso dell'audio. Per usare queste classi si deve usare un mixer già impostato.

MIXER

Su external/Prefabs si trova il mixer che ha tre canali:

- Music (per la musica)
- Sound effects (per gli effetti sonori)
- Master (che regola l'audio in generale, infatti music e sound sono figli del master).

Il mixer può essere ampliato con altri canali (per esempio si può aggiungere il canale "SoundForest" come figlio di "SoundEffects").

E’ importante che ogni canale ha esposto come parametro il suo volume, il quale deve avere questa nomenclatura (i parametri iniziano con la lettera minuscola):
$nomeCanale$ + ”Volume”.
Per esempio il parametro del volume del canale Master ha come nome “masterVolume”.

AUDIO ENUM

Questo enumeratore, che si trova su External/Audio, contiene gli elementi che rappresentano i canali del mixer.
La nomenclatura di tali nomi dev’essere equivalente al $nomeCanale$ (prima lettera maiuscola).
Se si aggiunge un nuovo canale al mixer, si deve aggiungere l’elemento corrispondente nell’enumeratore AudioEnum.

AUDIO MANAGER

AudioManager è figlio di *LogicalManager* ed è un singleton.
I metodi pubblici sono raggiungibili da chiunque (visto che è un singleton).

GetVolume

Questo metodo consente di avere il volume di un particolare canale del mixer.

Input

- l’enumeratore *AudioEnum* corrispondente al canale del mixer che si vuole ottenere il volume.

Output

- Un float rappresentato in percentuale 0-1, dove 0 equivale ad assenza volume e 1 equivale al volume al massimo.

SetVolume

Questo metodo consente di modificare istantaneamente il volume da un preciso gruppo di mixer.

Input

- l’enumeratore *AudioEnum* corrispondente al canale del mixer che si vuole modificare il volume.
- Un float rappresentato in percentuale 0-1, dove 0 equivale ad assenza volume e 1 equivale al volume al massimo.

SetVolume

Questo consente di modificare il volume da un preciso gruppo di mixer.

Input

- L'enumeratore *AudioEnum* corrispondente al canale del mixer che si vuole modificare il volume.
- Un float rappresentato in percentuale 0-1, dove 0 equivale ad assenza volume e 1 equivale al volume al massimo.
- Un float che rappresenta il tempo di transizione che serve per arrivare al volume indicato (Il float dev'essere maggiore di 0.
- Un parametro opzionale che è una curva (AnimationCurve) che rappresenta il cambiamento del volume nel tempo.
  L'asse delle x rappresenta il tempo: 0 rappresenta il tempo iniziale e 1 rappresenta il tempo iniziale più la durata della transizione.
  L'asse delle y rappresenta il volume: dove 0 rappresenta il volume iniziale e 1 il volume che si vuole raggiungere.
  la curva deve partire nel punto da un punto (0, x) e terminare in un punto (1, y).

CLOCK SYSTEM

Questo componente consente di oggetti che hanno come funzione memorizzare il tempo trascorso.

CLOCK

La classe clock consente di far partire un orologio che inizia a immagazzinare il tempo da quando è stato avviato.

Clock - costruttore -

Input

- un booleano che indica al costruttore se l'orologio deve iniziare spento o acceso.

ResetOn

Il metodo consente di riavviare l'orologio.

Input

- un float che consente di far partire l'orologio con già del tempo immagazzinato.

Pause

Il metodo consente di mettere in pausa il gioco e viceversa.

Input

- un bool che rappresenta se l'orologio deve andare in pausa o no.

ResetOff

Il metodo consente di spegnere l'orologio.
Per riaccendere l'orologio si deve usare il metodo ResetOn.

ActualTime

Il metodo consente di esaminare il tempo trascorso da quando l'orologio si è avviato.

Output

- Un float che rappresenta il tempo trascorso da quando l'orologio è stato avviato.

Timer

La classe timer è simile alla classe clock, con l'unica eccezione che c'è un limite di tempo, facendo diventare l'orologio un timer.

Timer - costruttore -

Il costruttore crea un timer e lo spegne.

Timer - costruttore -

Input

- un float che rappresenta la durata del timer

ResetOn

Il metodo consente di riavviare l'orologio.

Input

- un float che rappresenta la durata del timer.
- un float che consente di far partire l'orologio con già del tempo immagazzinato.

Pause

Il metodo consente di mettere in pausa il gioco e viceversa.

Input

- Un bool che rappresenta se l'orologio deve andare in pausa o no.

ResetOff

Il metodo consente di spegnere l'orologio.
Per riaccendere l'orologio si deve usare il metodo ResetOn.

Limit

Output

- Restituisce il tempo massimo che deve durare il timer.

ActualTime

Il metodo consente di esaminare il tempo trascorso da quando il timer si è avviato,

Output

- un float tra 0 e la durata del timer che rappresenta il tempo trascorso da quando il timer è stato avviato.

ActualTimeReversed

Il metodo consente di esaminare inversamente il tempo trascorso da quando il timer si è avviato (per esempio, la durata del timer è di 5 secondi e sono passati 2 secondi, il metodo mostrerà 3 secondi).

Output

- Un float tra 0 e la durata del timer che rappresenta il tempo trascorso da quando il timer è stato avviato.

IsFinish

Il metodo consente di vedere se il tempo conservato dal timer supera la sua durata.

Output

- un booleano che indica se il tempo è scaduto.

ActualTimeInPercent

Il metodo consente di esaminare il tempo trascorso in percentuale da quando il timer si è avviato.

Output

- Un float tra 0 e 1 (durata del timer) che rappresenta il tempo trascorso da quando il timer è stato avviato.

ActualTimeInPercentReversed

Il metodo consente di esaminare inversamente il tempo trascorso in percentuale da quando il timer si è avviato (per esempio, la durata del timer è di 10 secondi e sono passati 2 secondi, il metodo mostrerà 0.8).

Output

- Un float tra 0 e 1 (durata del timer) che rappresenta il tempo trascorso da quando il timer è stato avviato.

COMPONENTS

In questa cartella ci sono tutti script che sono componenti di Unity: tali script hanno una funzione generale e modulare, quindi si possono attaccare a qualsiasi GameObject.

DON’T DESTROY ON LOAD MANAGER

Questo componente/script che si trova dentro Scripts/Components serve semplicemente a non far distruggere l’oggetto nel cambio delle scene.
Nell’Inspector c’è un booleano “isUnique”, tale booleano se è attivato, controlla se nella scena c’è un’altro GameObject con tale nome, se esiste distrugge se stesso (utile per testare).

DESTRUCTION ELEMENT MANAGER

Questo componente/scripts che si trova dentro Scripts/Components lo si deve mettere aggiungere in un gameobject che si vuole distruggere o disabilitare.
Quando il gameobject verrà disabilitato o distrutto, il componente manda un evento per notificare a tutti i suoi iscritti che l’oggetto è stato distrutto.
Nell’Inspector c’è un float “time”; questo float se attivato, farà distruggere il GameObject dopo il tempo indicato dal parametro.

DEBUG SYSTEM

Questa cartella contiene script dedicati al debug.

DEBUG EXTEND

Questa classe statica che si trova in Scripts/Debug System e consente di debuggare il gioco anche quando è gira in build.

Log

Questo metodo statico consente di debuggare il suo input.
Se il test gira editor, il metodo attiva semplicemente la console. invece, se il test gira in build, il metodo crea una console UI (se non esiste già) e ci mette il suo debug.

Input

- Un oggetto generico che viene stampato o nella console o nella console UI.

INPUT SYSTEM

La cartella Input System consistono in classi che facilitano l'accesso all'input.
L'input System, usa l'e*vents manager* per comunicare a tutti gli oggetti iscritti, la notifica dell'input premuto.
La classe principale è la classe I*nputReaderManager* (un manager).

INPUT READER MANAGER

Il fulcro principale della classe è il suo Update, in esso chiama il suo componente InputReaderManager che controlla effettivamente i tasti premuti.

Infatti una classe che vuole riceve un input deve implementare una di queste interfacce che si differenziano nell'output dell'azione (può essere solo nessun output, un vettore o un float).

INPUT READER COMPONENT

Questo script si trova nella cartella external/input e consiste nella lettura dell'input.
Questo script deve essere scritto dall'utente. Lo script è figlio di una classe statica che obbliga l'utente a implementare due metodi: UpdateKeyboard e UpdateJoystick.
Inoltre esiste una variabile *isPause* che è true se il gioco è in pausa (tramite l'evento CallPause) e false viceversa.

UpdateKeyboard/UpdateJoystick

Un metodo chiamato dall'update dell'inputManager, consiste nella lettura dell'input (se è attivo il joystick, l'input manager chiamerà UpdateJoystick se invece è attiva la tastiera chiamerà UpdateKeyboard)..
Se un preciso di input è attivo si deve chiamare l'eventsManager con l'azione specifica (tramite il metodo statico CallEvent).

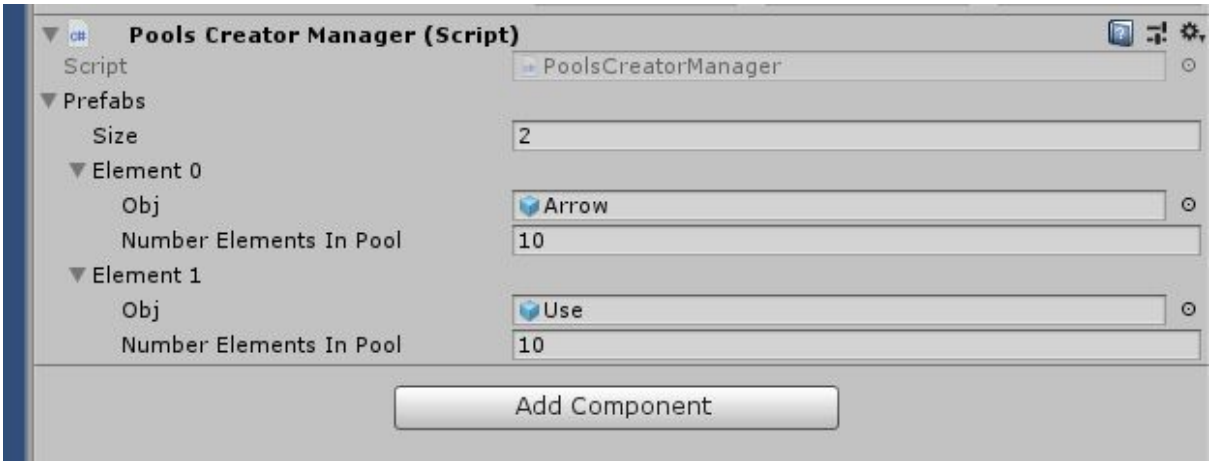
Si vede l'immagine per un esempio:

```
#region Public Methods
public override void UpdateKeyboard()
{
    if (Input.GetButtonDown("Submit"))
        EventsManager.CallEvent(EventAction.GetInputEntryMenu);
    if (Input.GetButtonDown("Cancel"))
        EventsManager.CallEvent(EventAction.GetInputReturnUI);

    if (!isPause)
    {
        EventsManager.CallEvent(EventAction.GetInputMovement, GetMovement());
        if (Input.GetButtonDown("Fire1"))
            EventsManager.CallEvent(EventAction.GetInputUse, EventAction.GetInputUse);

        if (Input.GetButtonDown("Fire2"))
            EventsManager.CallEvent(EventAction.GetInputAttack, EventAction.GetInputAttack);
    }
}
```

POOL SYSTEM

In questa cartella ci sono script che implementano un servizio di pool.

Per creare la pool si deve mettere nella scena il prefab *PoolsCreator* su External/Prefabs.
L'oggetto consiste in una lista di prefabs che, all'inizializzazione della scena, verranno creati e istanziati in delle pools.
Ogni prefab è associato un intero che indica quanti pezzi si devono creare per ciascun prefab.



Per istanziare un prefab della pool si deve usare il metodo:
GameObjectExtend.InstantiatePool
Invece per disabilitare il prefab della pool si deve usare il metodo
GameObjectExtend.Destroy

E' importante che il prefab che si usa per istanziare la pool, in seguito sia lo stesso che si userà nei metodi "*GameObjectExtend.InstantiatePool*" e "*GameObjectExtend.Destroy*"

GAME MANAGER SYSTEM

In questa cartella si trovano gli script che sono legati al concetto di *GameManager*.
Il *GameManager* è uno classe che contiene dati e metodi generali validi per ogni scena del gioco.
Su external/Prefab c'è il prefab GameManager che dev'essere messo nella prima scena del gioco.

SCENE ENUM

Lo script che si trova su External/Scene è un enumeratore dove ogni elemento è il nome di una scena di unity.
Ogni elemento deve avere il nome uguale alla scena di Unity corrispondente.
Di default c'è l'elemento Null, il quale non si deve togliere (l'elemento serve per le scene non registrate nell'enumeratore).

GAME MANAGER

Questo script che si trova in Scripts/Game Manager System è un *manager* ed è singleton.
Nell Inspector ha diverse variabili:

- Il numero di versione del gioco.

- Un elemento di *SceneEnum* che consiste nella scena attuale.

Se si vuole creare un *GameManager* personalizzato basta che il nuovo elemento sia figlio del *GameManager* /La convenzione del nome è $Titolo del gioco$+"Manager"),
Si ricorda di sostituire il *GameManager* personalizzato nel prefab *GameManager*.

NewLevel

Questo metodo consente cambiare scena.

Input

- un elemento di *SceneEnum* che consiste nella scena in cui si vuole arrivare.

EnableMouse

Questo metodo consente abilitare o non abilitare il mouse.

Input

- Un booleano che indica che il mouse dev'essere abilitato o non abilitato.

LEVEL MANAGER SYSTEM

In questa cartella si trovano gli script che sono legati al concetto di *LevelManager*.
Il *LevelManager* è uno classe che contiene dati e metodi generali utili per un livello generico.
Su external/Prefab c'è il prefab *LevelManager* che dev'essere messo in ogni scena di gioco.

LEVEL MANAGER

Questo script che si trova in Scripts/level Manager System è un *manager* ed è singleton.
Nell Inspector ha diverse variabili:

- Il prefabs del Pools Creator (vedi pools system)

Se si vuole creare un *LevelManager* personalizzato per ogni livello è possibile, basta che il nuovo elemento sia figlio del *LevelManager* (La convenzione del nome è $Titolo del livello$+"Manager").
Si ricorda di sostituire il LevelManager personalizzato nell'istanza del prefab di LevelManager.
La classe ha due funzioni:

- Istanziare, se esiste, il pools Creator (se usate il level manager ricordatevi di non mettere nella scena il pools creator).
- Attivare il suo componente *PauseComponent* quando l'evento CallPause è chiamato. In questo componente, quando il gioco è in pausa si attiverà il Time.timeScale = 0 e viceversa.

EXTEND UI

Questa cartella consiste in scripts che consentono di creare UI in maniera più facilmente. L'astrazione che usa Pearl per la creazione dei menu è che una UI è composta da pannelli e in ogni momento c'è solo un pannello aperto mentre tutti gli altri sono chiusi.
Se si vuole creare una UI usando Pearl si devono seguire queste regole.

- I figli diretti del canvas sono un elenco di tutti i pannelli.
- La root di ogni pannello deve avere il componente Multitags e in esso avere il tag Panel.
- I figli del pannello sono gli elementi che appaiono solo dentro il pannello genitore.
- Convenzione: I pannelli devono terminare con suffisso "Panel".

UI MENU MANAGER

La classe che si trova in Script/UI Extend. La classe è un *manager* ed dev'essere il padre di tutte le UI che sono create tramite il sistema di Pearl.
Questa classe ha nell'ispector due variabili:

- La prima variabile indica il primo GameObject che sarà selezionato nella UI.
- La seconda indica se la UI è una UI iniziale (menu) o in game (pause).

Questa classe di default è sottoscritta a due eventi:

- *GetInputEntryMenu*: consente di ritornare all'azione precedente.
- *GetInputReturnUI*: consente di aprire/chiudere il menu, funziona solo con quando la UI è in status pause.

Nel gioco, se si chiama il menù, la classe chiamerà l'evento *CallPause* (mettendo il gioco in pausa).

ChangeButton

Questo metodo pubblico di UI (attivabile, quindi tramite bottoni, ecc...), consente di cambiare l'oggetto selezionato/attivo. Se il bottone si trova in un'altro pannello, cambierà anche il pannello di riferimento.

Input

- Il riferimento al gameobject che si vuole selezionare/attivare.

Quit

Questo metodo consente di chiudere il gioco.

New Game

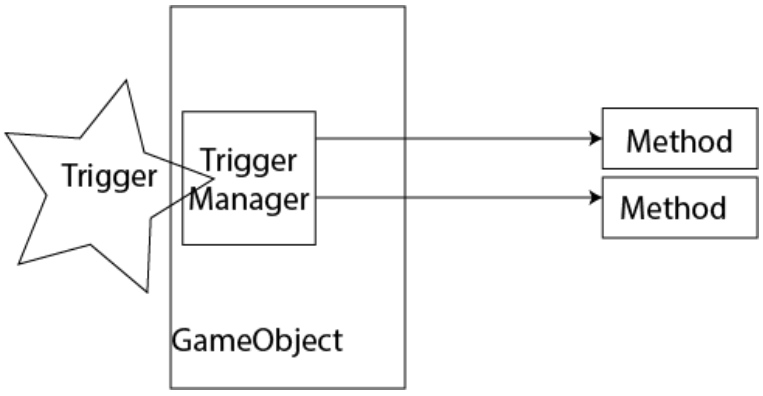
Questo metodo consente di iniziare una nuova partita tramite il *GameManager*.

Input

- l'elemento ScenEnum che rappresenta la scena iniziale

TRIGGER SYSTEM

Premessa: questo elemento è stato testato poco e sicuramente nelle future edizioni verrà corretto e migliorato.
Questa cartella consiste in scripts che consentono di creare Trigger fra gli oggetti in modo più pulito.
Il fulcro consiste di avere *Trigger Manage*r (un gestore di trigger) in ogni gameobject. Il *trigger manager* gestisce tutti i *trigger* (i gameobject triggerati) e smista le informazioni dei *trigger* a gli *eventi* che ne hanno bisogno.



Il *trigger* potrebbe avere informazioni da dare al *Trigger Manager*, per questo deve possedere il componente *ComplexAction* che ha un dizionario di stringhe-object. Quando si attiva il trigger manager (ossia quando il *trigger* colpisce il *trigger manager*), esso prenderà le informazioni del dizionario e le manderà ai vari metodi che aspettano il trigger.

Trigger Manager

La classe principale del Trigger System, si trova in Scripts/Trigger System.
La classe, come detto prima, aspetta qualunque tipo di *trigger* per poi notificare o mandare le informazioni (se ci sono) ai vari metodi che sono iscritti.
Il trigger Manager ha due variabili nell inspector:

- La prima variabile è una lista di gameobject, questi gameobject sono gli ascoltatori del trigger manager. Quando si attiva il *trigger manager,* la classe in elenco notifica (e manda informazioni) a tutti i componenti dei gameobject in elenco che implementano il metodo dell'interfaccia *ITrigger*.
- La seconda variabile è un elenco di tags, se il trigger (il gameobject) non possiede quei tags verrà ignorato e il *trigger manager* non si attiverà.

ITrigger

Questa interfaccia dev'essere implementata per tutte le classi che aspettando una chiamata dal trigger manager.

Trigger

Il metodo si attiva quando viene chiamato dal *trigger manager*.

Input

- il primo parametro è un'istanza di Informations, Informations è una classe che contiene solamente il dizionario di stringhe-oggetti del complex action.
- Il secondo parametro è la lista dei tags del trigger (utile se si vuole fare un doppio controllo specifico per ogni metodo).

```
public void Trigger(Informations informations, List<Tags> tags)
{
    if (tags.Contains(Tags.Attack))
        Debug.Log("I was hit, I took " + informations.Take<byte>("damage") + " of damage");
    if (tags.Contains(Tags.Use))
        EventsManager.CallEvent(EventAction.SendHealth, 0.5f);
}
```