

*Frontespizio*

*A.A. 2020/2021*

*Leo Marzoli, 298777*

## *Specifica del software:*

Lo scopo del progetto è la realizzazione di un videogioco attraverso Unity, un game engine che utilizza il framework .NET 4.x e linguaggio C#.

Il gioco consiste in un'avventura progressiva, dove il giocatore si impersona in un cagnolino di nome Oliver, che dovrà affrontare diversi livelli per completare il gioco. Per riuscirci dovrà rispondere a dei quiz prettamente a tema C#, se le domande sono superate correttamente avrà la possibilità di passare al livello successivo, altrimenti ha l'obbligo di ritentare fino al suo successo. Il giocatore prima di ogni livello riceverà un aiuto, questo sarà casuale, perciò superato l'ultimo livello con fortuna, il gioco termina.

## *Studio del problema*

### ***Punti Critici***

Dal progetto possono insorgere diverse problematiche, alcune di queste sono:

- 1) L'adattamento dei design pattern allo sviluppo del videogioco.
- 2) Il corretto funzionamento delle classi, rispetto ciò che si vede a video.
- 3) L'utilizzo di file esterni per il caricamento dei dati nel gioco.
- 4) Lo studio di un sistema di progressione del gioco.
- 5) La ricerca di intrattenimento del giocatore, che non deve annoiarsi.

### ***Fronteggiamento***

L'implementazione dei design pattern può avvenire tramite una loro "rivisitazione" adattata alle classi del game engine che permettono di creare il gioco.

Il corretto funzionamento delle classi è implementabile grazie al sistema di debug che ci permette di verificarne l'esecuzione, anche di quelle classi che sono "isolate" poiché invocate unicamente da degli eventi associati a delle funzionalità del game engine.

Il caricamento dei dati a tempo d'esecuzione per interagire correttamente alle domande del quiz, viene effettuato attraverso delle classi che entrano in comunicazione con dei file JSON.

Il sistema di progressione del gioco è stato implementato grazie all'utilizzo di variabili opportunamente aggiornate all'interno di alcune classi che si occupano di gestire le meccaniche principali.

Riguardo l'aspetto di game design per l'intrattenimento del giocatore si è cercato di rendere il più interattivo possibile il gioco in modo da creare un'esperienza immersiva che coinvolgesse, perciò il nemico o il giocatore vengono gradualmente "uccisi" dalla risposta data alle domande, il tutto è amplificato dalla difficoltà anche questa progressiva dei quiz e velocizzato da un timer che se scade detiene il Game Over.

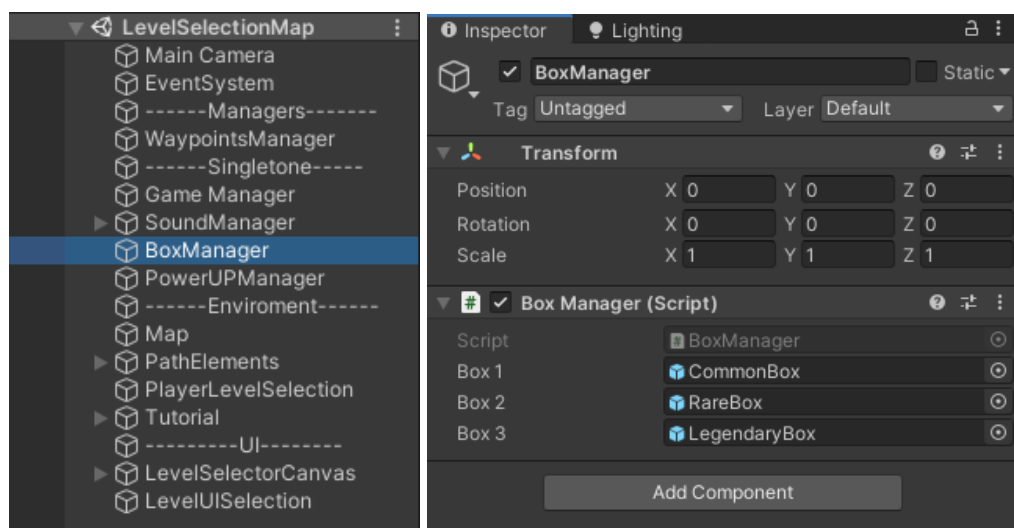
## Scelte Architettureali

### Diagramma delle classi

<https://drive.google.com/file/d/1bK8Cr4M47SJdqZqfZiOVJPlGYTBXHKP3/view?usp=sharing>

### Descrizione dell'architettura

L'architettura del software è incentrata su classi che si concentrano su specifiche funzionalità del gioco. Ognuna di queste classi per poter comunicare con Unity deve utilizzare la libreria UnityEngine ed implementare la classe MonoBehaviour che al suo interno definisce una serie di metodi che se utilizzati opportunamente permettono di interagire con la sua logica. Se una classe implementa MonoBehaviour deve necessariamente essere “attaccata”, diventando così una componente, ad un oggetto di scena, detto GameObject. Quindi un GameObject è formato da componenti. (Questo può essere visibile o non visibile a seconda dello scopo che ha)



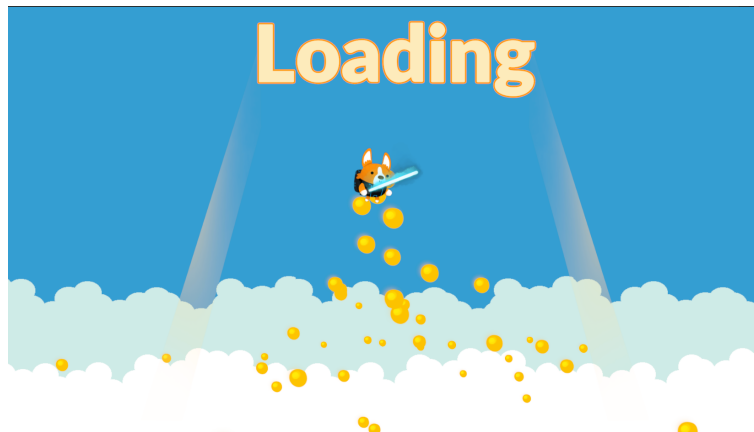
Il gioco è diviso in scene, ogni sezione del gioco viene associato ad una particolare scena, si può passare da una scena all'altra in un qualsiasi momento tramite codice, implementando la libreria UnityEngine.SceneManagement che al suo interno definisce tutti i metodi per poter operare con esse.

Scenes In Build	
✓ ProgettoEsame2021/Scenes/LevelSelectionMap	0
✓ ProgettoEsame2021/Scenes/Level1	1
✓ ProgettoEsame2021/Scenes/Level2	2
✓ ProgettoEsame2021/Scenes/Level3	3
✓ ProgettoEsame2021/Scenes/Level4	4
✓ ProgettoEsame2021/Scenes/Level5	5
✓ ProgettoEsame2021/Scenes/LoadingTransition	6
✓ ProgettoEsame2021/Scenes/Powerup	7
✓ ProgettoEsame2021/Scenes/EndGame	8

Il gioco parte presentando una mappa con dei puntini rossi detti altresì Waypoint, questa mappa viene gestita principalmente dalla classe `LevelSelectionMap` la quale tramite logiche booleane applicate al metodo `Update()`, definito da `MonoBehaviour` la quale viene invocato ad ogni frame del gioco, permette di controllare lo spostamento di un cagnolino che visualizza a quale livello ci troviamo, quest'ultimo in particolare sarà mostrato a video tramite la classe `LevelUISelection`.

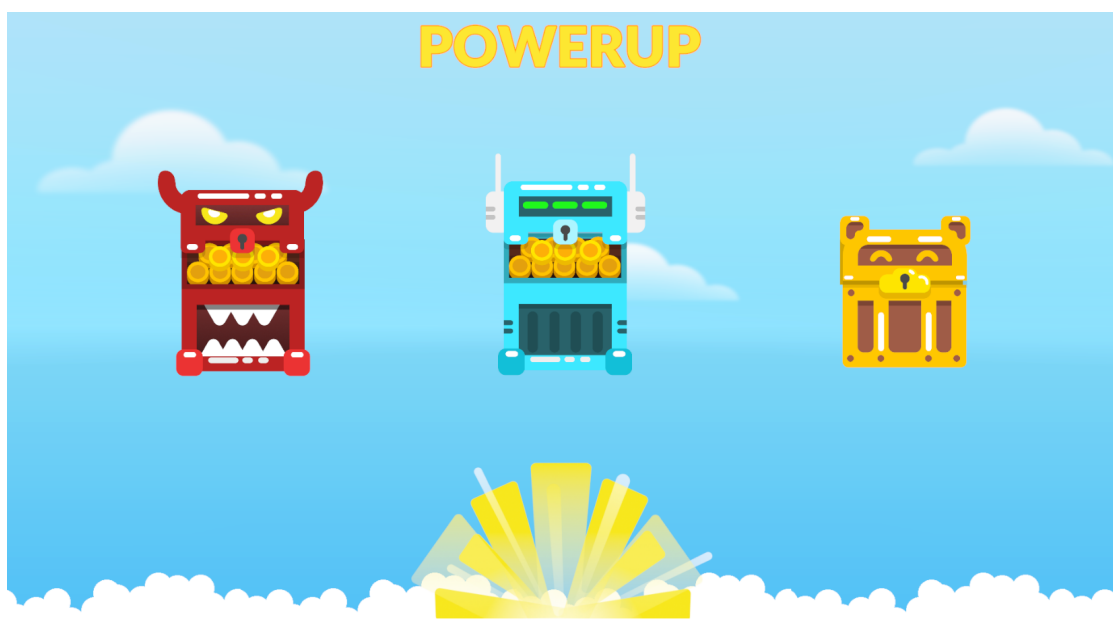


Una volta deciso di iniziare a giocare in un livello e perciò aver premuto il bottone `Start`, viene invocato da una funzionalità di eventi del Game Engine un metodo `LoadTransition()` definito in `StartLevel` che passa alla scena di transizione per dare tempo ai `GameObject` di caricarsi. In questa scena sono definiti ulteriori componenti utilizzati da altri `GameObject`, come `Transition` che definisce un intero `secondsToWait` inizializzato a 5 per la durata della transizione e tramite l'utilizzo del metodo `Start()` definito da `MonoBehaviour` che ha la particolare proprietà di essere eseguito non appena una classe viene caricata con successo ( in particolare è invocato il primissimo frame prima dell'`Update()` ), utilizza un'altra funzionalità di `MonoBehaviour` ovvero `Invoke(methodName : String, time : Float)` che invoca un metodo (in questo caso invocherà una transizione di scena) dopo un certo ammontare di tempo, creando così un effetto di delay, tra il passaggio da una scena all'altra.



La scena successiva a quella di transizione è quella di gestione dei power up da assegnare prima di ogni livello. Qui si è scelto di adottare un design pattern di tipo composite, ma le funzionalità principali vengono gestite da BoxManager e PowerUpManager. Rispettivamente si occupano di istanziare a tempo d'esecuzione dei gameobject di tipo box ai quali poi sono associati delle animazioni di apertura o chiusura a seconda di quale di questi si sia dovuto aprire o meno e in base al numero di box aperti si aumenta la probabilità di ottenere dei power up che poi andranno ad influenzare le dinamiche del livello.

Mentre PowerUpManager si occuperà dell'apparizione anch'essa a tempo d'esecuzione dei gameobject di tipo power up che anch'essi tramite animazioni permetteranno di apparire una volta che la sequenza per mostrare i box sarà terminata. Finito l'intero procedimento si passerà alla scena in cui vi si trova il rispettivo livello. Entrambe le classi implementano il Singleton la cui proprietà gli permette di essere uniche e in particolare all'interno di Unity di diventare un oggetto "indistruttibile" che perdura con le sue funzionalità per tutte le scene.



Ogni livello si presenta con un UI a forma di riquadro al centro della schermata contenente una domanda e quattro possibili risposte, queste domande vengono gestite dalla classe `QuestionManager` che legge un file JSON attraverso il metodo `QuestionDeserialize()` e tramite `DisplayQuestion()` ne assegna il valore ogni volta. La caratteristica di queste domande è quella di essere selezionate in maniera randomica per evitare il ripetersi dell'ordine in cui vengono mostrate nel caso in cui un giocatore effettuasse più volte un `Game Over`, perciò dispone anche del metodo `CountQuestions()`. Ogni bottone di risposte ad una domanda a sua volta dispone di una componente `AnswerController` che ha lo scopo di verificare che la risposta premuta alla domanda sia quella corretta.

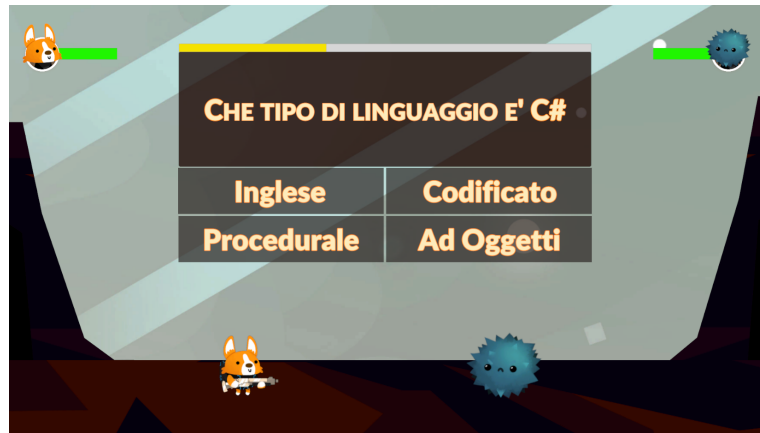
Perciò utilizza un evento collegato ad un delegato, per “catturare” il segnale di risposta. A questo evento è collegato il metodo `OnAnswerEvaluation(value : Bool)` definito nel `GameManager`, la quale si occuperà di eseguire determinate azioni in base al valore della risposta associato ad un booleano passato come parametro.

Tutte le meccaniche principali del gioco che riguardano il suo aspetto interattivo con il giocatore e che ne danno l'esperienza sono gestite dalla classe `GameManager`. Questa è infatti collegata alla maggiorparte delle classi all'interno del progetto, poiché si occupa di tenere quasi tutto sotto controllo. Adotta il pattern Singleton che ne assicura l'unicità all'interno di tutto il progetto, ma in particolare tutte le classi che derivano dalla classe Singleton perdurano per tutte le scene, senza mai essere “distrutte”, perciò `GameManager` sarà capace di accedere a qualsiasi script in una qualsiasi scena. Tramite il metodo `SetHealthGame()` imposta correttamente la vita del `Player` e dell' `Enemy` in base al livello in cui si trovano e valutando eventuali power up. Tramite il metodo `LevelVictory()` e `LevelGameOver()` gestisce la vittoria o la sconfitta di un normale livello o l'eventuale pareggio, che però risulterà in un `Game Over`, mentre `FinalLevelVictory()` serve per la gestione della eventuale vittoria del livello finale.

Ogni livello dispone anche di un timer che effettua un countdown, questo corrisponde esattamente al tempo che si ha prima di effettuare `Game Over` senza aver dato una risposta alla domanda. Questa funzionalità è gestita dalla classe `TimerCountdown` che nel suo metodo `Update()` ad ogni frame scala un valore della barra altresì chiamata slider, avvertendo visivamente il giocatore attraverso il cambio opportuno di colore in base a `sliderToBecomeYellow` e `sliderToBecomeRed` due variabili che ne definiscono lo stato.

In ogni livello si può trovare un `Player` ed un `Enemy`, questi sono dei `gameobject` ognuno con le rispettive componenti che ne definisce il comportamento. Sono caratterizzati dal metodo `Shoot()` che gestisce il momento di attaccare ( in questo caso sparare ) all'avversario e `OnTriggerEnter2D( other : Collider2D )` un evento classificato da `MonoBehavior` che permette di eseguire blocchi di istruzione nella manifestazione di una collisione con un altro `gameobject`. Entrambi ereditano da una classe `CharacterBase` che ne definisce i membri in comune evitando così ridondanza

di codice. In particolare definisce un booleano `isDead` che permette di valutare l'eventuale vittoria o perdita di un livello. Qui si è deciso di applicare un altro design pattern: lo State, modificato leggermente per poter essere adattato alla logica di Unity.



Oltre alle meccaniche principali, sono state definite componenti attaccati a `gameObject` non visibili nel gioco poiché non ritenuto necessario per il loro scopo. Ne è un esempio la classe `EffectsManager` il cui unico scopo è quello di fruire alle altre classi dei metodi pubblici per la generazione di effetti nel gioco in particolari momenti, come: appena un livello inizia attiva la visione del riquadro delle domande e il countdown tramite `ShowBoxQuestionAndTimer()`, mentre se il livello è terminato tramite `HideBoxQuestionAndTimer()` lo nasconde.

Un'altra classe non visibile, ma implementata è `SoundManager`. Ha il compito di gestire i suoni all'interno delle scene. Implementa il Singleton perciò in un qualsiasi momento tramite `PlaySound(index : int)` oppure `StopSound()` può manipolare i suoni, poiché perdura.

Una volta terminato un livello, da vincente o sconfitto, si ritornerà alla scelta dei livelli con i Waypoints perciò l'intero processo sopra descritto si ripeterà fino a che il gioco non sarà terminato.

## ***Descrizione dei pattern utilizzati***

All'interno del progetto ho deciso di utilizzare un pattern per ogni tipo di quelli visti a lezione: creazionale, strutturale, comportamentale.

### **1. Singleton**

Questo pattern all'interno del gioco è presente come una classe a se stante che ha la particolare proprietà di rendere qualunque classe che decida di utilizzarla "indistruttibile" tramite l'invocazione di un metodo `DontDestroyOnLoad()`, normalmente dal passaggio fra una scena e l'altra di un gioco, i `gameObject` creati in una vengono distrutti nel passaggio dell'altra.



Tramite questo metodo invece possiamo conservare un particolare oggetto di scena, che può essere acceduto da qualunque altra classe in un qualsiasi momento, poiché normalmente per accedere ad una classe che è attaccata come componente ad un gameobject dovremmo assicurarci che questo gameobject a tempo d'esecuzione si trovi nella nostra scena e non dia un riferimento null, generando così un errore. Questa metodologia ci fornisce un ulteriore vantaggio, ovvero quello di assicurarci che esista una sola copia di qualunque classe decida di implementare questo pattern, perché ogni altro riferimento se presente all'interno di una scena, viene distrutto tramite metodo Destroy(gameobject).

## 2. State Pattern

Si è deciso di utilizzare questo pattern comportamentale per rappresentare lo stato del player o dell' enemy che a tempo d'esecuzione può cambiare da uno stato di idle (fermo) ad uno di attacco ad un eventuale stato di morte. Questo è stato rivisitato alla logica e ai metodi che Unity disponeva, perciò si è fatto uso del metodo Update(), che appunto opera ogni frame.

La classe Player e Enemy ereditano dalla classe base CharacterBase, dunque per fare riferimento ad uno dei due useremo un'istanza della classe base. Quest'ultima utilizza l'interfaccia ICharacterState che definisce al suo interno un metodo DoState(character : CharacterBase ) e ritorna uno stato (di tipo ICharacterState ). Ogni stato che uno dei due soggetti può assumere è rappresentato come classe isolata che generalizza ICharacterState e ridefinisce le istruzioni di DoState() a seconda di cosa vuole che si faccia, eventualmente all'interno troveremo anche il passaggio da uno stato all'altro (per esempio, dentro IdleState si passerà ad AttackState o a DeadState ).

All'interno del metodo Update() di CharacterBase invochiamo continuamente il DoState dello stato in cui ci troviamo, così da poter passare da uno stato all'altro in rapidissimo tempo.

## 3. Composite Pattern

Come ultimo pattern è stato utilizzato quello di tipo strutturale. L'utilizzo di questo pattern lo si può trovare all'interno della gestione dei power up e di conseguenza dei box che permettono di farli apparire. Anche qui è stato riadattato alle funzionalità di Unity perciò si presenta in una maniera leggermente modificata. La classe BoxManager inizializza 3 Box : common, rare, legendary. Si occupa poi di riempire i box: il common avrà al suo interno 3 possibili power up di cui soltanto 1 però verrà selezionato, il rare sarà composto da 2 common e il legendary sarà formato da 1 rare e 1 common, cioè l'equivalente di avere 3 common. Questi vengono aggiunti ad una lista di Box chiamata boxList, composta quindi di 3 elementi e che permetterà di accedere

in base alla posizione ad uno specifico Box. L'intento è quello di prendere un Box casuale all'interno di questa lista e "prelevare" tutto quello che vi è all'interno (che possono essere anche altri Box, aperti come una matrioska) per poi mostrarlo ed applicare i power up. Per fare ciò usiamo un metodo `Pick()` definito dall'interfaccia `IComponent`, questo è un metodo ricorsivo, che permette di entrare ricorsivamente all'interno di un Box e inserire con opportune istruzioni dentro una lista di `IComponent` chiamata `itemsInABox` tutto quello che c'è dentro il Box su cui abbiamo invocato il metodo ricorsivo `Pick()`. Successivamente con delle istruzioni di selezione per concentrarci su esclusivi elementi della lista, ne preleveremo e li mostreremo a tempo d'esecuzione nella scena dedicata ai power up del gioco.

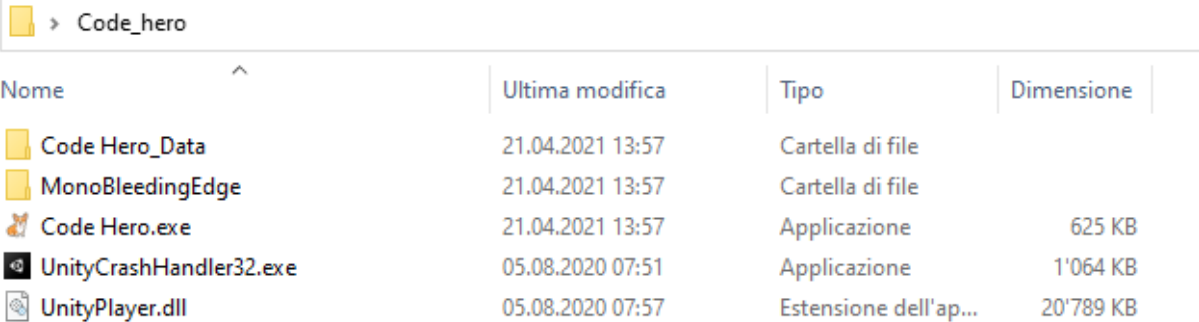
Invece la classe `PowerUpManager` si occupa dell'inizializzazione dei power up e dell'effettiva applicazione degli stessi avendo accesso ad alcune variabili per effettuare il tuning di gioco. Infatti, nel momento in cui con il metodo ricorsivo `Pick()` che abbiamo precedentemente descritto si entra all'interno della classe `PowerUp` questa ha all'interno della sua definizione di metodo `Pick()` delle istruzioni di selezione che permettono di impostare a true dei booleani per valutare quale powerup è o non è stato attivato. La classe `PowerUpManager` si occuperà di controllare con altrettante istruzioni questo booleano ed applicare il powerup al gioco in base al loro valore.






## *Documentazione sull'utilizzo*

Il gioco si presenta come uno standalone perciò una cartella contenente i file per il funzionamento e l'eseguibile che terminerà con una estensione .exe, avviando quest'ultimo si aprirà la schermata di gioco e questo sarà giocabile.

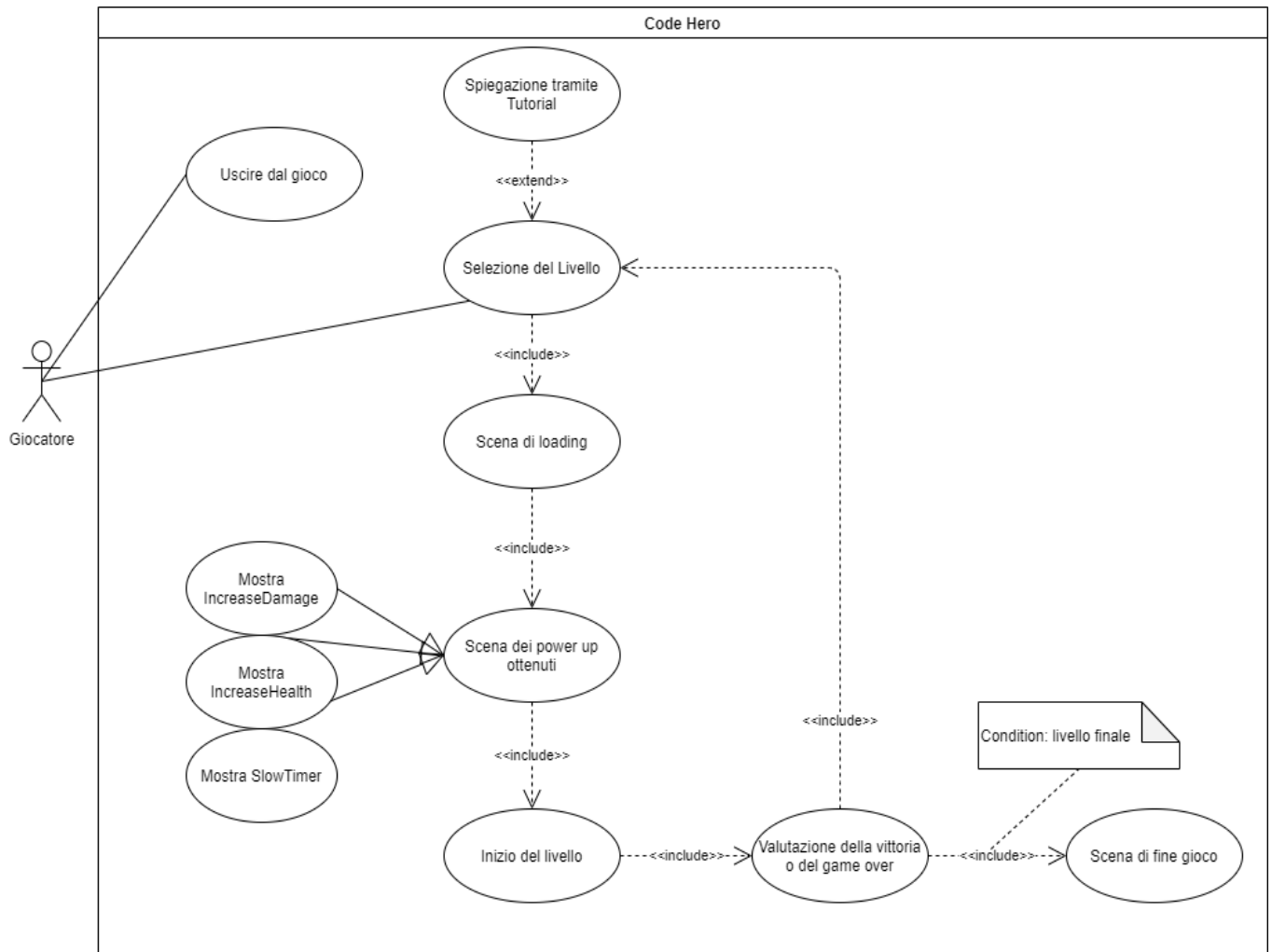
Non vi sono particolari parametri con cui questo debba essere eseguito.

Per poter uscire dal gioco bisogna usare la sequenza di comandi da tastiera ALT + F4.



Nome	Ultima modifica	Tipo	Dimensione
 Code Hero_Data	21.04.2021 13:57	Cartella di file	
 MonoBleedingEdge	21.04.2021 13:57	Cartella di file	
 Code Hero.exe	21.04.2021 13:57	Applicazione	625 KB
 UnityCrashHandler32.exe	05.08.2020 07:51	Applicazione	1'064 KB
 UnityPlayer.dll	05.08.2020 07:57	Estensione dell'ap...	20'789 KB

## Use Cases con relativo schema UML



Nel diagramma sopra descritto si nota che il giocatore ha la capacità di uscire dal gioco oppure decidere di iniziare a giocare selezionando il livello. Nel caso in cui si verifichi quest'ultima vi è inizialmente un tutorial che spiegherà le meccaniche di gioco, per poi passare al passaggio di tutte le scene che precludono l'inizio di un livello. Successivamente in base all'esito di questo si tornerà alla selezione del livello, ripetendo il ciclo, fintanto che non ci troveremo al livello finale dove si passerà alla scena ultima di fine gioco.