Politecnico di Torino

POLITECNICO DI TORINO

Corso di Laurea Magistrale in Software Engineering

Tesi di Laurea Magistrale

Motore di Gioco basato su Unity 3D per la creazione di videogiochi 2D open-ended

Sviluppo di dinamiche open-ened con l'obiettivo di creare un sistema di interazione tra oggetti di gioco per la creazione di applied game che mirano all'educazione STEM per i bambini.

Relatore
Prof. Marco MAZZAGLIA

Relatore esterno / Tutor Aziendale
Dott. Andrea BOZZI
Dott. Ing. Francesco CAPOZZI

Candidato
Matteo DI FABIO
Matricola: s**257563**

Anno accademico 2020/2021

Sessione di Laurea Aprile 2022

Per una lettura più chiara, adatta ed efficace è caldamente suggerita la visione del documento su due pagine [1] in modo che i documenti, i riferimenti e le immagini, necessari a chiarire il contenuto dell'argomento di cui si sta parlando, siano più agevoli.

[1] **Per visualizare il documento su due pagine è possibile selezionare l'opzione dal programma usato per visuallizare PDF. Può trovarsi sotto la voce "Duale" oppure "Visualizzazione a due pagine". È necessario poi selezionare l'opzione per avere il frontespizio separato. Alcuni esempi:**

- **Adore Acrobat Reader: Visualizza -> Visualizzazione Pagina -> Visuale a due pagine & Visualizza frontespizio nella visualizzazione a due pagine.**
- **Microsoft Edge: Visualizzazione Pagina -> Visuale a due pagine & Visualizza frontespizio separatamente.**
- **Firefox: "»" (simbolo in alto a destra) -> Raggruppamento pari**
- **Evince: "☰" -> Duale**

Ringraziamenti

Ringrazio innanzitutto il prof. Marco Mazzaglia che ha avuto un ruolo molto importante: ha creduto in me e mi ha permesso di realizzare la tesi nel settore videoludico e, mediante la sua mentorship, mi ha guidato nella stesura tecnica del lavoro e nella ricerca di un'azienda dove sostenere questo progetto.

A seguire ringrazio tutta Marshmallow Games per l'opportunità che mi ha concesso accogliendomi come uno della famiglia. In particolare, ringrazio Andrea Bozzi, mio supervisor all'interno dell'azienda, che mi ha supportato e sopportato in maniera egregia in questo mio percorso.

Altra menzione importante mia Zia Rosita che mi ha aiutato nella revisione linguistica e stilistica della Tesi rendendola più "elegante" e fruibile.

In ultimo ma non per minor importanza, ringrazio i miei genitori e tutti i miei parenti che mi hanno sostenuto sia economicamente che emotivamente e che mi hanno permesso di fare questo percorso in un ambiente familiare tranquillo e di scrivere serenamente questo elaborato, spingendomi sempre a raggiungere i miei traguardi.

Ringrazio anche tutte le persone che con la loro amicizia hanno fatto parte di questo lungo e faticoso cammino universitario rendendolo più prezioso e più leggero da affrontare.

Title of the thesis: Motore di Gioco basato su Unity 3D per la creazione di videogiochi 2D open-ended

Autore: Matteo Di Fabio

Relatore: Marco Mazzaglia

Tutor Aziendale: Andrea Bozzi & Francesco Capozzi

Abstract: Il seguente elaborato e gli argomenti in esso trattati sono il risultato di un lavoro progettuale sviluppato con il Gruppo italiano Marshmallow Games, azienda innovativa che, attraverso una tecnologia proprietaria, sviluppa e distribuisce giochi educativi *mobile* per bambini con lo scopo di insegnare in modo divertente attraverso la fusione del gioco e della narrazione.

Il presente lavoro svolto ha l'obiettivo di definire un prototipo di motore grafico che permetta di costruire, in modo agevole e scalabile, scene open-ended, che diano la possibilità di far interagire liberamente tra loro oggetti, protagonisti e ambienti, mediante N (idealmente infinite) dinamiche di interazione diverse e la fantasia del giocatore. Così come accade per "La casa delle bambole", in questo mondo virtuale tutto è gestito dall'immaginazione del bambino, il quale, usando e spostando a suo piacimento gli oggetti e i personaggi, crea una propria realtà, delle sue avventure, senza vincoli ma liberamente guidato solo dalla sua creatività. Nonostante in commercio siano già presenti app con questa tipologia di gameplay, l'obiettivo dell'Azienda è di sfruttare eventualmente, in futuro, il prototipo sviluppato, per creare una app di gioco che stimoli non solo la creatività dei bambini o la capacità di narrare storie ma che possa anche incentivare l'apprendimento delle STEM in età prescolare.

La tesi, dunque, si propone di illustrare le prime fasi di ideazione e realizzazione di questo motore grafico, che sarà in grado di creare un gioco, pensato per bambini di età 4+, utilizzabile su dispositivi *mobile*, smartphone e tablet ed i cui contenuti siano accessibili attraverso un'App.

Saranno messe in evidenza le fasi di studio e di ricerca fatte sugli ambienti da creare focalizzando l'attenzione sull'architettura del Sistema di interazione necessario per pianificare le infrastrutture utili ed essenziali a generare le interconnessioni che servono a dar vita ad una storia. Si analizzeranno, infine, le possibili situazioni che, attraverso la progettazione di un ambiente idoneo alle narrazioni, permetta al bambino, di volta in volta, di far prendere vita alle storie.

Indice

Elenco delle figure

Elenco dei Codici

Parte I

Introduzione

Capitolo 1

Il Contesto

Questo capitolo ha lo scopo di introdurre il lavoro svolto, spiegando l'obiettivo della tesi e di descrivere la struttura del documento.

1.1 Motivazioni

Ogni tecnologia inizia ad essere conosciuta quando c'è un interesse nel mercato che ne fa aumentare la richiesta e dunque l'acquisto. Quando questo avviene, i prezzi cominciano a scendere e sempre più persone sono invogliate ad acquistare il prodotto facendo aumentare così il numero degli utenti che lo utilizzano. Negli ultimi anni ciò è avvenuto anche per le applicazioni *mobile* per bambini. Infatti, con il progredire delle tecnologie e l'ormai diffusissima distribuzione di device portatili quali smartphone e tablet, è impossibile trovare bambini, anche in età prescolare, che non facciano mai uso di questi strumenti. Dato il diffondersi di questa tendenza e la preoccupazione che i bambini possano incorrere in contenuti a loro non adatti, Marshmallow Games, azienda che supporta questo lavoro di tesi, da sempre ha cercato di dare valore al tempo che i bambini trascorrono con smartphone e tablet realizzando esperienze digitali sicure che coinvolgano non solo i bambini ma tutta la famiglia. Tale traguardo è raggiunto con lo sviluppo e la distribuzione di giochi educativi *mobile* per bambini che permettano loro di imparare in modo divertente attraverso la fusione del gioco e della narrazione. Tutta l'esperienza maturata da Marshmallow Games in questi anni è convogliata nella realizzazione di "Smart Tales - Impara le STEM", un'app che contiene una raccolta di racconti animati ed interattivi e giochi utili ad avvicinare i bambini attraverso lo storytelling alle STEM (Scienza, Tecnologia, Ingegneria e Matematica) in modo divertente. Nell'ultimo periodo, il settore di videogiochi *mobile* per bambini ha mostrato particolare attenzione verso i giochi open-ended, ossia giochi con finale aperto e libera interpretazione che diano la possibilità di far interagire liberamente tra loro oggetti, protagonisti e ambienti, mediante N (idealmente infinite) dinamiche di interazione diverse e la fantasia del giocatore. Anche Marshmallow Games ha puntato i suoi occhi su questo tipo di prodotto, per cui l'azienda ha deciso di affidare la definizione e lo sviluppo di un prototipo di gioco che estenda il framework di Unity e che permetta di costruire, in modo agevole e scalabile, scene open-ended.

1.2 Obiettivi

Il motore di gioco, quale obiettivo finale di questo lavoro, dovrà fornire la possibilità di creare ambientazioni in cui ci sarà uno sfondo navigabile in orizzontale e verticale che a sua volta sarà popolato da alcuni personaggi e dai loro oggetti. Ogni oggetto o personaggio sarà toccabile per suscitare una sua reazione, come un personaggio che saluta o una palla che rimbalza e trascinabile con il dito in giro per la schermata per poi essere posato dove si vuole. Alcuni oggetti avranno delle interazioni "speciali" e specifiche come una bottiglia che può essere agitata per spruzzare acqua o una lavagna su cui si può disegnare. Si vuole ricreare fedelmente l'esperienza di un bimbo quando gioca con i suoi giocattoli preferiti, per questo si potranno aggiungere altri oggetti o personaggi ad ogni luogo. La tesi, dunque, si propone di illustrare le prime fasi di ideazione e realizzazione di questo motore di gioco che possa permettere un giorno di creare un'ipotetica app che si distinguerà dai competitor grazie ai contenuti educativi fortemente mirati all'apprendimento delle materie STEM. Saranno messe in evidenza le fasi di studio e di ricerca fatte sugli ambienti da creare focalizzando l'attenzione sull'architettura del Sistema di interazione necessario per pianificare le infrastrutture utili ed essenziali a generare le interconnessioni che servono a dar vita ad una storia.

1.3 Struttura della Tesi

Questo elaborato è suddiviso in 4 macro sezioni, ognuna delle quali è composta da uno o più capitoli che descrivono i seguenti argomenti:

- parte I - INTRODUZIONE: Presentazione della situazione generale sui giochi educativi e open-ended attraverso i capitoli:
  - capitolo 1 : motivazione e obiettivi della Tesi
  - capitolo 2 : evoluzione dei giochi educativi dalle origini a quelli open-ended
  - capitolo 3 : descrizione dell'ambiente lavorativo in cui è stato svolto il lavoro di Tesi
  - capitolo 4 : motivazione sulla selta di un motore di gioco basato su Unity
- parte II - IDEAZIONE E REALIZZAZIONE : scelte tecniche per la realizzazione del motore di gioco, divisa in:
  - capitolo 5 : descrizione della situazione tecnologica all'interno dell'azienda e delle ricerche necessarie effettuate ex ante ossia prima di iniziare lo sviluppo del motore
  - capitolo 6 : funzionamento del Drag&Drop
  - capitolo 7 : spiegazione delle scelte tecniche usate per implementare il sistema d'interazione tra oggetti
- parte III - LA DEMO : descrizione dell'ambientazione e funzioni all'interno di una demo basata sul prototipo sviluppato
  - capitolo 8, capitolo dedicato a descrivere e comprendere com'è composta la demo e quali sono le caratteristiche più interessanti
- parte IV CONCLUSIONE :
  - capitolo 9 : descrizione dei risultati ottenuti,
  - capitolo 10 : riflessioni su eventuali miglioramenti che potrebbero essere apportati e l'obbiettivo generale del futuro.

Capitolo 2

State Of Art

In questo capitolo verrà mostrata tutta l'evoluzione dei giochi educativi passando per quelli open-ended e facendo riferimento, inoltre, ai giochi attualmente sul mercato.

2.1 Evoluzione dei videogiochi educativi

Se a partire dagli anni '40/'50 i videogiochi si sono evoluti tantissimo fino a rappresentare il fotorealismo [1], la stessa evoluzione è toccata ai videogiochi educativi, che ormai fanno sempre più parte dell'apprendimento dei più piccoli.

> *I videogiochi sono stati definiti "palestre di apprendimento", ma dobbiamo chiarire a quali tipi di apprendimento ci riferiamo.*
>
> *Giocando impariamo di noi stessi (Come reagisco alle difficoltà, alle sconfitte, alle sfide? Quanto persevero e mi impegno per raggiungere un obiettivo? Quanto riesco a pianificare e controllare le mie azioni?) e anche dell'ambiente di gioco (i generi, le regole, il linguaggio, le azioni da mettere in atto, le meccaniche).*
>
> *Attraverso il gioco impariamo a compiere azioni fisiche (esplorare l'ambiente, cogliere più informazioni contemporaneamente, coordinare i movimenti) e mentali (selezionare, decidere, risolvere, pianificare, anticipare, collaborare, comunicare, adattare una strategia non utile, riprendere una strategia già utilizzata in passato, creare, ecc.). La ripetizione delle azioni e la possibilità di avere feedback sulla loro efficacia aiutano a consolidare l'apprendimento, ma è solo quando ragioniamo su quello che stiamo facendo che riusciamo a far davvero tesoro di quanto appreso e a metterlo in pratica anche fuori dal videogioco.* [2]

Per questo motivo potremmo considerare Pac-Man come uno dei primissimi giochi educativi ma in realtà non è così poichè mentre Pac-man stimolava la logica, il problem solving e il suo obiettivo era divertire e intrattenere il giocatore, oggi per videogioco educativo si intende un gioco che mira all'insegnamento di uno o più argomenti come possono essere la storia, la geografia, la matematica, oppure il leggere e lo scrivere.

---

[1] Il fotorealismo significa semplicemente che una scena simulata è indistinguibile da una fotografia, o per estensione dalla vita di tutti i giorni. [1]

> *Un videogioco educativo (chiamati anche edugame dall'inglese) è un videogioco che tratta intenzionalmente temi educativi. Non si tratta di un genere in senso stretto, in quanto videogiochi di ogni genere potrebbero avere contenuti educativi. Videogiochi di questo tipo vengono talvolta utilizzati per scopi didattici e per le moderne metodologie dell'insegnamento* [3]

Proprio per le considerazioni appena fatte, come primissimo videogioco educativo consideriamo *Logo Programming* [4], sviluppato nel 1967 presso Bolt, Beranek e Newman (BBN), una società di ricerca di Cambridge, nel Massachusetts, che aveva come scopo l'insegnamento di un linguaggio di programmazione. *L'obiettivo era creare un ambiente in cui i bambini potessero giocare con parole e frasi.* [5]. Questo gioco riscosse talmente tanto successo che venne rilasciato anche in italiano e negli anni 90 ne vennero realizzate versioni per personal computer utilizzate a scopi didattici. Appena dopo *Logo Programming*, trova successo *Lemonade Stand*[6] creato nel 1973 e portato sulla piattaforma Apple II nel 1979. Il gioco è apparentemente semplice: i giocatori gestiscono un chiosco che vende limonata. Le loro azioni sono finalizzate alla scelta o alla quantità di ingredienti da acquistare, a come pubblicizzare il prodotto, scegliere il prezzo da imporre per la vendita della limonata ecc. per poi valutare a fine giornata il profitto ottenuto. Il gioco così, in realtà insegnava ai giocatori lezioni di gestione complesse su affari ed economia tanto che è stato uno dei primi a utilizzare una piattaforma di gioco per farlo.
Successivamente, durante gli anni '80 vengono rilasciati diversi videogiochi educativi, tra cui *Operation FROG* che aveva lo scopo di insegnare come vivisezionare una rana, insegnamento tipico delle scuole americane, oppure *The Body Transparent* progettato per insegnare il funzionamento del corpo umano, le caratteristiche principali degli organi nonché le malattie degli stessi mediante due metodi di gioco: l'assemblaggio che prevedeva il mettere insieme le parti del corpo e l'associazione, ossia la possibilità di collegare nomi agli organi o alle ossa ecc. Nello stesso periodo si afferma anche *The Learning Company (TLC)* una società di software educativo fondata nel 1980 a Palo Alto, in California. L'azienda ha prodotto una linea di software di apprendimento, giochi di *edutainment*[2] e strumenti di produttività basati sui voti. I suoi titoli includevano la serie di punta Reader Rabbit, per bambini in età prescolare fino alla seconda elementare, e The ClueFinders, per studenti più avanzati.[8]

[2]Il termine edutainment è un neologismo coniato, negli anni 90 da Bob Heyman, documentarista del National Geographic, per indicare la possibilità e la necessità di insegnare e imparare divertendosi in gruppo. [7]

I più famosi di questo decennio, sono però, *The Oregon Trail* [9] e *Where in the World Is Carmen Sandiego?* [10] .
The Oregon Trail è una serie di giochi educativi per computer, il primo di questa serie è stato pubblicato nel 1971, ma è con il rilascio su Apple II nel 1980 che acquisisce la sua fama. Il gioco originale è stato progettato per insegnare agli scolari la realtà della vita dei pionieri del 19° secolo sull'Oregon Trail [3]; mentre il secondo è un videogioco educativo sulla geografia, a tema investigativo, pubblicato da Brøderbund nel 1985 divenuto talmente famoso da avere diverse trasposizioni animate (tra cui anche la più recente su Netflix) ma soprattutto ha avuto uno show dedicato negli USA che ha riscosso tantissimo successo. Il gioco nei decenni successivi ha vantato la pubblicazione di diversi sequel; l'ultima release risale al 2015 con *Carmen Sandiego Returns* [11].
Negli anni sono stati prodotti sempre più videogiochi educativi in particolare negli anni '90 dove la diffusione degli Home computer ha contribuito a questo trend, ricordiamo:
Museum Madness (1994) [12] dove i giocatori apprendono nuovi fatti e informazioni su una vasta gamma di argomenti educativi, tra cui storia, geologia, evoluzione, spazio e tecnologia.
Ready Robot Club (1994) [13] proposto con un servizio di abbonamento mensile. Ogni mese, i bambini fortunati ricevevano un disco caricato con giochi divertenti e contenuti educativi. Inoltre musica digitale, giochi di memoria, esperimenti scientifici, aggiornamenti spaziali, esercizi di matematica e informazioni su personaggi storici avrebbero accompagnato ogni numero.
Al giorno d'oggi, grandi aziende come Nintendo e Sony, ma non solo, stanno creando giochi educativi rivolti sia ai bambini che agli adulti: giochi come Brain Age, Little Big Planet e Minecraft sono incredibilmente popolari. Ma il numero di argomenti esplorati dai moderni giochi educativi non si limita solo a matematica, scienze e storia. L'arte, la musica, la letteratura, la dattilografia, la risoluzione dei problemi e altro ancora trovano casa nell'edutainment. Inoltre, questi giochi si estendono su una varietà di piattaforme diverse tra cui console, PC (browser, download e CD), iPad, tablet e smartphone.
Vista la quantità e la varietà di giochi presenti, spesso ci si domanda se i videogiochi possano essere visti come strumento educativo e dunque portare vantaggi anche all'interno delle scuole. Svariati studi e ricerche effettuate sull'argomento mettono in luce i benefici derivanti dall'uso di giochi digitali.

[3] La Pista dell'Oregon (Oregon Trail) era una delle principali strade di migrazione via terra nel continente nordamericano, che portava da luoghi lungo il fiume Missouri all'Oregon Country. [9]

Tra i videogiochi educativi di grandi compagnie del settore, trovano spesso posto videogiochi il cui obiettivo primario non è insegnare, ma rendere possibile l'apprendimento al giocatore, che vede all'interno del gioco stesso ambientazioni storiche realmente accadute. Giochi tra loro differenti che vanno dallo strategico allo sparatutto in prima persona, ma accomunati da un'ambientazione storica fedele.
Harry J. Brown, nel suo volume "Videogames and Education", parla di come le nuove tecnologie consentano a chiunque di approcciare in maniera sempre più agevole il mondo videoludico, per riuscire ad utilizzarlo come un mezzo per imparare, educando ed educandosi [14].
Dunque, con il progredire dell'importanza dei videogiochi nel mercato multimediale, aumenta il diffondersi di vari generi che mirano ad identificarsi con uno in particolare, per essere il must have di quella tipologia di gioco. A questa "specializzazione" fa parte anche il settore del videogioco educativo che ha visto nascere tante realtà che mirano allo sviluppo di solo giochi educativi. Tra queste aziende:

- Spin Master

  Azienda leader mondiale di intrattenimento per bambini che crea esperienze di gioco eccezionali attraverso un portafoglio diversificato di giocattoli innovativi, franchise di intrattenimento e giochi digitali. Fondata nel 1994 a Toronto, in Canada, da due amici d'infanzia, Ronnen Harary e Anton Rabie e successivamente raggiunta dal loro compagno di classe Ben Varadi, Spin Master è cresciuta da un prodotto iniziale fino a diventare un leader globale nell'intrattenimento per bambini con 28 uffici in tutto il mondo e quasi 2.000 dipendenti.

- Toca Boca

  Studio svedese di sviluppo di app focalizzato su applicazioni a misura di bambino per tablet e smartphone. Secondo il sito Web dell'azienda, "Produciamo giocattoli e giochi digitali che aiutano a stimolare l'immaginazione". L'azienda è di proprietà di Spin Master (dal 21 aprile 2016) e ha sede a Stoccolma, in Svezia. Toca Boca è il brand numero uno di app per bambini nell'App Store

- Marshmallow Games

  Società italiana specializzata in videogiochi per bambini in cui è stato svolto il presente lavoro di tesi. Per una descrizione dettagliata, si rimanda alla sezione 3.1

- Tinybop

  Studio con sede a Brooklyn che crea prodotti educativi che hanno l'obiettivo di essere giocati a casa e in classe, per stimolare la curiosità dei bambini di tutto il mondo.

- Dr. Panda Limited

  Studio cinese che crea applicazioni educative per bambini su smartphone e tablet. Nato nel 2011 a Chengdu sede del Chengdu Research Base of Giant Panda Breeding, un centro di conservazione dove i visitatori possono osservare nel loro habitat naturale i panda giganti a rischio di estinzione. Dal rilascio del suo primo gioco nel 2012 Dr. Panda ha riscosso molto successo tanto da ampliare le tipologie di giochi proposti, aprire diversi nuovi studi di sviluppo e lanciare una serie TV con la loro mascotte (Dr. Panda) come protagonista.

- Ahoiii Entertainment

  Nata dall'esigenza di offrire videogiochi educativi ai propri bambini, l'azienda che ha riscosso moltissimo successo grazie alle numerose app educative sviluppate. Successo riconosciuto dai diversi premi vinti tra cui Best of 2016 su Appple Store o Best Kids Games per la German Developer Award 2013.

- Sago Sago Toys Inc

  Azienda canadese, Sago Mini è una filiale di Toca Boca, che ha sviluppato moltissime app per bambini, ottenendo molteplici premi per la categoria app per bambini in età prescolare.

Tutte queste Aziende, nonostante siano grandi competitor nel settore, sono accomunate da uno stesso denominatore ossia lo scopo ludico che vogliono perseguire con i loro prodotti tuttavia ciascuna di esse, con la propria specificità, cerca quell'app che la distingua dalle altre e la inserisca in una posizione di privilegio nella creazione di videogiochi per bambini.

2.2 Giochi Open-Ended

Per capire cosa siano i videogiochi Open-Ended è utile partire dai giochi tradizionali per bambini che vengono considerati tali. Prendiamo come esempio una macchina giocattolo e dei blocchi geometrici, tipo lego. Mentre la prima ha una funzione precisa, rappresentare la macchina e trasportare qualcuno, i blocchi di lego lasciano libero spazio alla fantasia del bambino che può comporre qualsiasi tipo di cosa e giocarci in modo sempre differente.
Nel primo caso il giocattolo e dunque il gioco ha una funzione pre-determinata mentre nel secondo, il gioco può essere definito Open-Ended poiché è gestito liberamente dal bambino e il suo estro, la sua creatività, l'immaginazione lo rendono giocabile.

> *L'apprendimento che deriva da molti giocattoli con funzione pre-determinata è più di tipo "causa-effetto". Ad esempio: se spingi piano l'automobilina questa va piano, se spingi forte va forte, spingi troppo forte vedi che si ribalta.*
>
> *Un giocattolo con funzione pre-determinata richiede delle azioni specifiche (anche semplici ma ben precise) da compiere per essere giocato. Se non fai quelle azioni il gioco non è giocabile, o comunque non fa ciò per cui è stato progettato. I giocattoli open-ended , invece, non hanno una funzione associata ma fanno leva sull'immaginazione del bambino, lo spingono a creare in continuazione nuove modalità di gioco e quindi promuovono la capacità del bambino di risolvere problemi sempre nuovi che variano ogni volta che si cambia il tipo di gioco.*
>
> [...] *Investire in un gioco "open-ended" di qualità può essere (in alcuni casi) inizialmente più costoso, tuttavia il giocattolo durerà di più. Non tanto per la qualità del giocattolo, ma proprio per il fatto che il divertimento non diminuisce negli anni perché le modalità di gioco saranno continuamente reinventate dal bambino.*[15]

Anche quando si parla di videogioco open-ended, dunque, sono presenti le stesse caratteristiche di un gioco classico con la possibilità di creare autonomamente l'attività, senza regole o istruzioni, ossia un videogioco dove non si è guidati da una storia preconfezionata per ottenere un fine, ma la vicenda viene creata dal giocatore stesso, grazie alla sua fantasia, divenendo esso stesso artefice del gioco, decidendo secondo la sua creatività come giocarci.
Nel mercato dei videogiochi esistono già molti prodotti che entrano nella fascia di giochi open-ended, pur tuttavia presentando delle differenze dal gioco open-ended "puro"; spesso sono conosciuti con il termine sandbox oppure open world.
Un gioco sandbox è un videogame con un elemento di gioco che offre all'utente l'uso di un alto grado di creatività e libertà per completare le attività verso l'obiettivo

intrinseco al gioco. All'interno del game il player può disattivare o ignorare gli obiettivi di gioco poiché dispone di strumenti per modificare il mondo, se stesso e le modalità del gioco stesso, aprendo nuove possibilità che non erano previste dal game designer. Dunque, ciò che distingue un gioco sandbox da un gioco open-ended è che quest'ultimo è completamente privo di obiettivi, diversamente dal primo che dispone di traguardi da raggiungere per progredire nella storia.

> *Con il termine open world si intende un videogioco in cui il giocatore può muoversi liberamente all'interno di un mondo virtuale; infatti è data ampia libertà al giocatore il quale può scegliere come e quando affrontare obiettivi o dedicarsi alla semplice interazione con l'ambientazione e ciò che la popola.*
>
> *Un gioco a mondo aperto non implica necessariamente un vero e proprio sandbox, in genere fanno rispettare alcune restrizioni per l'ambiente di gioco a causa di limitazioni tecniche assolute o limitazioni imposte dalla linearità di un gioco.* [16]

Prese in considerazione queste differenze potremmo definire open-ended il videogioco con il livello di libertà massimo e a scalare Sandbox e Open World saranno suoi sott'insiemi.
Siccome il gioco open-ended è il massimo strumento ludico utile a stimolare la fantasia, a ragionare, alla comprensione causa-effetto e al problem solving e che ha un fattore di rigiocabilità molto alto, anche il settore educativo si è approcciato a questo "mondo".
Nell'ambito delle app per bambini il gioco open-ended di maggior successo si chiama Toca World ma non è l'unico ad avere un buon seguito, ci sono altre aziende videoludiche per bambini che hanno preso ispirazione e portato il proprio stile nei videogiochi open-ended.
A seguire, una breve carrellata di alcune tra le più importanti app di giochi Open-Ended.

2.2.1 Toca Life World

Toca Life World è la nuova app sviluppata da Toca Boca, con cui ciascuno può creare il suo mondo virtuale e giocare con la storia che vuole. Questa mega-app mette assieme tutte le app di Toca Life (City, Vacation, Ufficio, Hospital e altre ancora) collegando tutto in un mondo sconfinato in cui giocare. Il mondo di Toca Life World è enorme, si possono creare storie con i tuoi personaggi preferiti, nel luogo che si sceglie e che stimola di più la creatività: ad es. si può portare a scuola un animale o andare in skate con un bradipo. In Toca Life World il giocatore "crea", porta avanti il gioco, comanda e dà vita alla storia che vuole. [17]

2.2.2 Fiete World

Sviluppato da Ahoiii Entertainment, Fiete World è un gioco open ended in cui il giocatore potrà scoprire e conoscere attraverso i personaggi di tutte le etnie, le peculiarità, le differenze e le somiglianze dei diversi paesi del Mondo. Anche qui l'utente può ogni volta scegliere autonomamente un'ambientazione diversa e viaggiare, esplorare e conoscere la parte del Mondo che vuole attraverso gli oggetti più disparati.[18]

2.2.3 Sagomini World

Sviluppato da Sago Sago Toys Inc, in Sagomini World i bambini viaggeranno in un mondo stravagante. Con Harvey, Jinja, Robin e Jack, i protagonisti del gioco, si fanno esperienze istruttive e formative. Si può esplorare lo spazio, costruire un robot, progettare un mostro, diventare supereroi e altro ancora, tutto portando a termine delle semplici missioni ma divertenti. [19]

2.2.4 Dr. Panda Città

Questa fantastica app permette di far volare la fantasia dei bambini utilizzando gli ambienti e le località presenti. Si può scoprire ogni angolo della città e di volta in volta si può raccontare la personale storia. In qualunque caso si può scegliere il personaggio che si vuole interpretare e dar sfogo a creatività e fantasia: si può essere uno scienziato, uno chef, un poliziotto, vestire i panni di chi si decide di essere e fare avventure senza fine. [20]

2.2.5 Smart Tales World

Smart Tales World è un gioco in fase embrionale che l'Azienda Marshmallow Games potrebbe sviluppare grazie agli eventuali risultati positivi ottenuti 9 da questo lavoro di tesi. Smart Tales World è un gioco sandbox pensato per bambini di età 4+ per dispositivi mobile, smartphone e tablet. È ambientato nel mondo di "Smart Tales - Impara le STEM", un'app di racconti e giochi educativi creati dall'azienda già presente sul mercato e sviluppato dall'Azienda, basato sul vasto assortimento di personaggi e scenari nel corso degli anni. I personaggi di Smart Tales sono simpatici animali parlanti, ognuno con il proprio carattere e le proprie abilità, che vivono tra loro condividendo le esperienze più disparate. Questo nuovo gioco però sarebbe piuttosto differente da Smart Tales - Impara le STEM in quanto permetterebbe al bambino di creare le proprie storie grazie a un ambiente ludico libero da obiettivi (open-ended). I due prodotti sono tra loro completamente indipendenti, condividono esclusivamente i personaggi con le loro caratteristiche e le ambientazioni. Scopo di questo progetto è partire da una proprietà intellettuale ormai nota per creare un'esperienza completamente nuova. Nella schermata principale di Smart Tales

World sarà raffigurato l'intero mondo di Smart Tales dal quale si potrà accedere a diversi luoghi dove si svolgerà l'azione di gioco vera e propria. Ogni luogo avrà uno sfondo navigabile in orizzontale e verticale e sarà popolato da alcuni animali e dai loro oggetti. Ogni oggetto o personaggio sarà toccabile per suscitare una sua reazione e trascinabile con il dito in giro per la schermata per poi essere posato dove si vuole. Alcuni oggetti, inoltre, avranno delle interazioni speciali specifiche. Si vuole ricreare fedelmente l'esperienza di un bimbo quando gioca con i suoi giocattoli preferiti, per cui potrà aggiungere, a suo piacimento, altri oggetti o personaggi ad ogni luogo estraendoli da un personale inventario per accendere la sua fantasia e inventare ed ideare sempre nuove storie. Inoltre, sarà possibile creare il proprio avatar personalizzato, oltre ad altri personaggi di propria invenzione, con il quale poter poi giocare all'interno di Smart Tales World. L'obiettivo è progettare un'esperienza di gioco per bambini che sia sicura, senza limiti di tempo e capace di stimolare la loro creatività rimanendo fedeli al nostro impegno nella educazione delle materie STEM (Scienza, Tecnologia, Ingegneria e Matematica) e in una sana ed equilibrata crescita emotiva. Come altri prodotti competitor già presenti sul mercato, questa app si fonda su meccaniche semplici come drag&drop, swipe, draw e tap che usiamo per implementare un mondo composto di singole scene in cui il giocatore può muovere personaggi e oggetti posizionandoli a piacimento. Smart Tales World si differenzia principalmente in 4 punti dai suoi competitor:

- Contenuti educativi fortemente mirati all'apprendimento delle materie STEM
- Meccanica di interazione con i singoli personaggi
- Photo mode
- Piccoli Filmati non interattivi come gratification

Capitolo 3
L'Azienda

Nel presente capitolo sarà descritta dettagliatamente l'Impresa , avendo cura di indicare, inoltre, i vari progetti portati avanti nello stesso ambito.

3.1 Marshmallow Games S.R.L.

Marshmallow Games SRL è un'Azienda italiana leader nel settore dei videogiochi educativi per bambini; è una PMI innovativa che attraverso una tecnologia proprietaria, sviluppa e distribuisce giochi educativi *mobile* per bambini offrendogli la possibilità di imparare in modo divertente attraverso la fusione del gioco e della narrazione.
L'obiettivo di Marshmallow Games è dare valore al tempo che i bambini trascorrono con smartphone e tablet e nel contempo realizzare esperienze digitali sicure che coinvolgano tutta la famiglia. Dalla sua fondazione nel 2014 ha realizzato e distribuito 20 educational game raggiungendo oltre 2 milioni di bambini in tutto il mondo. Quasi tutti i prodotti realizzati sono stati promossi da Apple sugli App Store di 150 nazioni raggiungendo in esse i primi posti delle classifiche della categoria "Kids". I contenuti sono tradotti in più di 5 lingue e questo ha permesso di raggiungere i mercati internazionali fin da subito.

Fra i prodotti che hanno riscosso maggior successo troviamo:

- Cosmolander: disponibile per smartphone e tablet iOS e Android e su AppleTV, ha superato i 100.000 download in tutto il mondo. Attraverso giochi e quiz, questa App consente di imparare a conoscere il sistema solare per bambini dai 6 ai 9 anni.
- Whiskey il Ragnetto: realizzato in partnership con Coccole Sonore ha ottenuto 280.000 download su App Store e Google Play. Contiene giochi di logica e creatività per bambini dai 3 ai 6 anni basati sul personaggio Whiskey il Ragnetto protagonista di video Youtube e di una serie animata distribuita su uno dei principali network televisivo nazionali.
- Hexaparty: disponibile su App Store e Google Play ha superato i 350.000 download ed è stata "App of the Day" sugli App Store in 150 nazioni. Si tratta del primo gioco di pixel art per bambini dai 6 agli 11 anni con un catalogo di oltre 800 disegni e funzionalità di realtà aumentata che permettono una modalità di gioco immersiva in cui i giocatori possono "sparare" proiettili colorati per completare disegni in hexel art posizionati sulle pareti di casa propria.

Tutti gli educational game sviluppati da Marshmallow Games per la creazione di contenuti digitali interattivi si basano su un Tool, una tecnologia proprietaria sviluppata internamente che consente di ridurre di oltre il 50% i tempi di produzione di un nuovo contenuto permettendo di focalizzarsi esclusivamente sul vero cuore dei prodotti, ovvero i giochi.
A riprova della capacità di design e sviluppo e grazie alla tecnologia proprietaria sviluppata, in questi anni Marshmallow Games ha anche avviato diverse collaborazioni con importanti aziende come TIM, Clementoni, ActionAid, DNA, Mukako, Museo del Risparmio, affiancandole nella realizzazione di giochi educativi per bambini e ragazzi. A cavallo fra il 2020 ed il 2021 sono stati chiusi importanti accordi per la distribuzione dei contenuti proprietari con Telco in oltre 50 paesi.

3.2 Smart Tales

Tutta l'esperienza maturata in questi anni è convogliata nella realizzazione di Smart Tales, una raccolta di racconti animati ed interattivi e di giochi utili ad avvicinare i bambini in età prescolare alle STEM (Scienza, Tecnologia, Ingegneria e Matematica) in modo divertente attraverso lo storytelling. Obiettivo prioritario è quello di creare un brand di riferimento per l'intrattenimento videoludico educativo per bambini dai 3 ai 6 anni I contenuti di Smart Tales sono accessibili attraverso applicazione mobile presente sulle principali piattaforme e sono localizzati in 5 lingue (Italiano, Inglese, Spagnolo, Francese e Tedesco). In Smart Tales i racconti prendono vita e tramite una voce narrante il bambino viene immerso nella storia; pagine di racconto e pagine di gioco si alternano per rendere l'esperienza ancora più coinvolgente e interattiva. I giochi accompagnano all'apprendimento di materie scientifiche. Al momento, all'interno della piattaforma sono contenuti oltre 60 racconti interattivi e più di 400 giochi educativi. Inoltre ogni settimana vengono aggiunti nuovi contenuti per consentire ai bambini di continuare ad imparare e a divertirsi. L'applicazione è scaricabile gratuitamente ma per accedere a tutto il catalogo di giochi e contenuti in costante aggiornamento è necessario sottoscrivere un abbonamento mensile o annuale. L'app ha superato i 200.000 download senza investimenti in marketing, è stata scelta da Apple come "App del giorno " in più di 50 nazioni, raggiungendo i primi posti della classifica nella categoria "kids" e ottenendo più di 1000 recensioni con una media di 4,6 stelle su 5. L'applicazione inoltre è oggetto di una partnership con l'UNICEF Italia [21] con l'obiettivo di promuovere i diritti dei bambini. In aggiunta, grazie ad accordi di co-marketing, l'App è distribuita in collaborazione con grandi brand come Intesa San Paolo, Mediaworld, Plasmon, Eni, Vodafone, in Italia e all'estero.

3.3 Principali Milestone Societarie

Dopo aver vinto il bando Valore Assoluto 2.0 promosso da Camera di Commercio di Bari nel 2014 ed essere entrati nel percorso di accelerazione TIM #WCAP Milano 2015, ad Aprile 2016 Marshmallow Games ha ottenuto un primo investimento pre-seed da parte di Boost Heroes SpA. Nel 2018 Marshmallow Games, dopo la partecipazione al percorso di accelerazione "BHeroes", al quale hanno partecipato business angels e imprenditori digital italiani, ha chiuso un round da 300.000€. Contestualmente all'aumento di capitale è stato formato un advisory board composto da imprenditori e professionisti del panorama startup italiano. Ad inizio 2019 Marshmallow Games ha co-fondato Tabi Labs in joint venture con IdeaSolution SRL, che con 60 milioni di download, è uno dei leader italiani del settore mobile. Tabi Labs si occupa della realizzazione e distribuzione di Tabi Learning, un educational game multidisciplinare incentrato sul programma scolastico della scuola primaria.

3.4 Le Tecnologie Utilizzate

3.4.1 Unity3D

> *Unity è un motore grafico multipiattaforma sviluppato da Unity Technologies che consente lo sviluppo di videogiochi e altri contenuti interattivi* [22]

Unity viene usato per moltissimi lavori oltre che per i videogame. Il suo impiego può essere sfruttato anche in altri ambiti come le visualizzazioni architettoniche, ambientazioni tridimensionali, shorts films e piccoli video tridimensionali con costruzioni in tempo reale e animazioni 3D. È diventato sempre più popolare, soprattutto fra gli sviluppatori mobile, fino a potersi definire l'engine per videogame più usato al mondo. L'azienda in cui è stato svolto questo lavoro di tesi, utilizza un motore grafico proprietario basato proprio su Unity 3D. Per questo motivo è stato necessario sviluppare il progetto in Unity, così da poter successivamente integrare il motore di gioco open-ended ad esso.

> *Unity è definito "multipiattaforma" perché il suo motore permette di "scrivere il gioco" una sola volta e realizzarlo o trasformarlo per ambienti o circuiti diversi: parliamo della creazione di uno stesso gioco per PC (Windows, Mac), Play Station, Xbox, console Nintendo comprese le piattaforme per dispositivi mobili ossia Android, iOS , Windows Phone ecc.*
>
> *L'ambiente di sviluppo di Unity è composto da un motore grafico, un motore fisico molto potente e un live game preview. Quest'ultimo permette di visualizzare in real-time le modifiche apportate al gioco durante le operazioni di programmazione.*[23]

3.4.2 C#

Sebbene il runtime di Unity sia scritto in C++, le API di scripting sono scritte in C#. Attualmente, C# è il linguaggio di scripting dominante per la piattaforma Unity (alias Unity3D) poiché entrambe le sue alternative, Boo e UnityScript, sono state deprecate nel 2017. UnityScript, una volta versione popolare di JavaScript modificata per Unity, è stata deprecata, solo il 3,6% dei progetti utilizzava pesantemente il linguaggio, sebbene il tempo dedicato al supporto fosse notevole.
Infatti, Unity espone un'API .NET in modo che lo sviluppatore non debba affrontare la complessità di scrivere il suo gioco in C++, ma può scriverlo in C#.
L'interfaccia utente dell'applicazione editor è scritta in C#, utilizzando principalmente la stessa API che è esposta agli sviluppatori di giochi.
C# è un linguaggio di programmazione generico orientato ai componenti basato su classi di alto livello creato come estensione di C. C# è stato sviluppato in Microsoft intorno al 2000 dall'ingegnere informatico danese Anders Hejlsberg e dal suo team come parte dell'architettura .NET.
Oltre a Unity, C# viene spesso utilizzato per sviluppare applicazioni Windows desktop e server.

3.4.3 Spine

Per lo sviluppo della demo, sono state utilizzate diverse sprite[1] e animazioni realizzate dagli artisti 2D in Spine. [25]
Spine è uno strumento di animazione che si utilizza specificatamente per i giochi 2D e mira ad avere un flusso di lavoro efficiente e semplificato, sia per la creazione di animazioni impiegando l'editor sia per l'utilizzo di tali animazioni nei giochi grazie ai runtime Spine.
Per l'implementazione del motore di gioco, però, non si usa direttamente il software di Spine, che come già accennato viene utilizzato dai grafici, ma si è usufruito di un pulg-in per Unity fornito dalla stessa software house produttrice di Spine che si chiama Spine-Unity [26]. Questo plug-in permette di espandere la libreria di unity ed aggiungere API utili alla gestione delle animazioni prodotte in Spine.

[1] Con sprite, in informatica, si indica un'immagine in grafica raster, generalmente bidimensionale (2D), che fa parte di una scena più grande (lo "sfondo") e che può essere spostata in maniera indipendente rispetto ad essa.Può essere sia statica che dinamica. Esempi di sprite sono gli oggetti 2D inclusi nei giochi, le icone delle interfacce grafiche delle applicazioni e le piccole immagini pubblicate sui siti web. [24]

Capitolo 4

Motore di Gioco o Motore grafico?

Questo capitolo spiegherà essenzialmente la ragione per cui si è deciso e scelto di sviluppare un framework di gioco su un motore grafico già esistente e la sua utilità. Si chiariranno le motivazioni che hanno guidato questo lavoro verso la costruzione e realizzazione del motore di gioco ed infine il rapporto e la relazione con un framework.

4.1 Cos'è un Motore Grafico

Per spiegare le motivazioni per cui si è deciso di costruire un motore di gioco basato su un motore grafico pre-esistente, c'è bisogno prima di tutto di spiegare cosa sia un motore grafico e di quanto sia dispendioso e difficile da realizzare in tempo, investimenti e creatività.
Un motore grafico è un software che consente lo sviluppo di videogiochi e altri contenuti interattivi tra cui visualizzazioni architettoniche, ambientazioni tridimensionali, shorts films e piccoli video tridimensionali con costruzioni in tempo reale e animazioni 3D. Esso è formato da componenti multipli necessari ad esempio per la gestione degli spazi/grafica, per le collisioni, la fisica, l'intelligenza artificiale, il suono, le animazioni ecc. e ognuno di questi componenti a sua volta può gestire la sua funzione corrispondente. Dunque ciascuna di queste parti rappresenta una complessissima entità che gestisce qualcosa di preciso e che deve interagire con le altre per rendere possibile la realizzazione di un videogioco. E' evidente quindi che un motore grafico non è uno strumento facile da realizzare, ci vogliono esperti specializzati in campi diversi per crearne uno. Potremmo dire che un motore grafico presenta due livelli: il primo composto da chi crea il motore e cioè esperti programmatori che danno vita a tutte le sue funzionalità; il secondo, composto da chi usa il motore, dal level designer all'artista, all'animatore. Chi lavora al basso livello impacchetta tutte le funzioni in una forma facilmente fruibile per chi poi va a costruire il gioco. Chi invece lavora ad alto livello, può anche non curarsi di come funziona il motore ma deve solo saperlo usare.
Siccome costruire un motore grafico da zero richiede molte risorse e denaro, possiamo catalogare a loro volta tre tipi di aziende. Quelle che costruiscono un motore grafico da rendere disponibile alle altre aziende, quelle che usano le componenti principali del codice per creare il "game content" (o "game assets") cioè ambientazione, armi e livelli ed infine quelle che usano ambedue le situazioni. Dunque

le prime aziende concentrano tutta la propria forza lavoro solo sul motore grafico così hanno la possibilità di mettere in licenza il proprio motore per consentire a chi lo usa di commercializzare il prodotto finale ricevendo compensi da moltissime aziende di videogiochi che sfruttano questa modalità di sviluppo. Mentre le seconde possono concentrarsi sulla qualità del gioco risparmiando sulla forza lavoro in quanto la licenza costa meno rispetto agli investimenti necessari alla creazione di un motore grafico. Infine ci sono quelle software house che hanno le risorse economiche e umane tali da potersi permettere di sviluppare un proprio motore grafico che sia su misura per i "suoi" videogiochi sfruttando il vantaggio di ottimizzare il proprio motore e riadattarlo per altri scopi migliorandone lo sviluppo senza dover aspettare nuove release di motori di terze parti.

4.2 Cos'è un Framework

Il motore di gioco che è al centro del presente lavoro di tesi, è un framework ossia un sistema già pronto all'uso che permette un utilizzo semplice e veloce degli strumenti per realizzare alcune specifiche funzioni. In particolare, la scelta è dettata essenzialmente da esigenze di praticità e anche di costi che questo sistema riesce a compensare. Inoltre in questo caso specifico, il framework è considerato lo strumento più idoneo per dar vita al gioco open-ended poiché grazie a questo sistema e alla sua flessibilità è possibile effettuare azioni e comandi in modo semplice e veloce sui quali poi poggerà la struttura e un ambiente "ludico" libero da obiettivi (open-ended) organizzato dallo sviluppatore su cui l'utente/giocatore potrà creare le proprie storie liberamente.

> *In informatica, un framework è un sistema che consente di estendere le funzionalità del linguaggio di programmazione su cui è basato, fornendo allo sviluppatore una struttura coerente ed efficace al fine di effettuare azioni e comandi in modo semplice e veloce.*[27]

> *Un framework, parola inglese traducibile con il termine struttura, ha lo scopo di fornire strumenti per facilitare e velocizzare il lavoro di programmazione di chi lo adotta. Un framework dunque semplifica l'utilizzo di uno o più linguaggi di programmazione e può gestire strumenti informatici diversi, come altri framework o software.*[28]

> un*framework ti dà la certezza che stai sviluppando un'applicazione che è nel pieno rispetto delle regole di business, che è strutturata e che è sia manutenibile che aggiornabile.*[29]

Dunque, nel nostro caso il framework rappresenta il motore di gioco utilizzato per automatizzare e facilitare la creazione di un gioco open-ended con scene modificabili e adattabili al fine di inserire nuovi ambienti, personaggi, strumenti e situazioni.

4.3 La Scelta e l'Utilizzo

Come già esposto nella sezione 3.1, l'azienda in cui è stato svolto il lavoro di tesi, è una PMI innovativa, che è nata meno di 10 anni fa, per cui piuttosto giovane e non dispone di tutte le risorse necessarie per creare un proprio motore grafico e i propri prodotti. Per questo motivo ha deciso di utilizzare un motore grafico "già pronto" come Unity3D, che porta con sè molti vantaggi tra cui è multi-piattaforma, cioè permette di sviluppare una volta sola per più device con architetture diverse (console, pc e smartphone) e l'ampio utilizzo, da parte di aziende e sviluppatori indipendenti, che ha dato vita ad un'enorme community da cui ottenere ulteriore supporto. Marshmallow Games per far fronte alle difficoltà di chi non è esperto di programmazione o grafica 3D nel poter costruire i giochi, ha creato un framework, basato su unity che potremmo definire motore di gioco detto anche tool di authoring, che semplifichi la configurazione di ambientazioni e dinamiche di gioco senza interventi sul codice. Sebbene l'utilizzo di un motore grafico come unity già permettesse la costruzione di videogiochi, lo scopo del motore di gioco basato su Unity, è automatizzare e facilitare la costruzione di scene di gioco in maniera da semplificare le scelte e l'inserimento di nuovi livelli, strumenti, personaggi e interazioni all'interno del gioco da parte di level designer, artisti o animatori. Per questo motivo è stato necessario sviluppare il progetto di tesi in Unity, così da poter successivamente integrare il motore di gioco open-ended a quello usato in azienda che non gestisce la costruzione di questa tipologia di videogame. Quindi si creerà un framework per Unity, rappresentato dal nostro motore di gioco, come se si andasse a creare un altro livello ancora più alto rispetto a quello descritto precedentemente nella sezione 4.1. Mentre Unity richiede conoscenze più tecniche per costruire una scena di gioco, il tool di authoring fornisce nuove opzioni all'editor di Unity che permettano in maniera semplice di aggiungere tutto il set di personaggi, oggetti e interazioni disponibili. Qualora per una nuova scena, il creatore ritenga opportuno aggiungere nuove cose, quali personaggi, strumenti di gioco o interazioni, allora è necessario l'intervento del programmatore, che creerà quel che serve seguendo le specifiche del game designer e rendendolo disponibile all'interno del motore, cosicché poi il "costruttore" possa trovare la voce corrispondente nell'editor e inserire in un click tutti gli elementi che aveva richiesto. Si può concludere affermando che il motore di gioco realizzato con il lavoro svolto presso Marshmallow Games, sarà utilizzato per automatizzare all'interno dell'azienda il processo di creazione di scene open-ended, evitando così, di usare direttamente Unity che richiede un livello di competenze più tecnico e di contro permettere a quante più figure lavorative di generare nuove ambientazioni di gioco per integrarsi con il tool di authoring già in utilizzo presso la compagnia e permettere di estendere il framework alla composizione di giochi di tipologia differente a quelli già rilasciati.

Parte II

La fase creativa: dall'ideazione alla realizzazione

Capitolo 5

Prima Di Iniziare

Questo capitolo ha l'obiettivo di delineare le fasi salienti percorse per sviluppare il motore di gioco. Nella fase ideativa si è scelto innanzitutto cosa voler fare, sono state fatte ricerche e studi sull'argomento, si sono analizzati i diversi lavori dei competitor e fatte scelte tecniche per rendere il lavoro attuabile per poi passare ad impostare l'attività preliminare allo sviluppo del motore di gioco. A seguire si è passati alla realizzazione dell'idea, risolvendo problematiche e imprevisti incontrati per poi arrivare alla chiusura e realizzazione del lavoro.

5.1 Scelte tecniche

All'interno dell'azienda in cui è stato svolto questo progetto, è stato sviluppato un motore basato su Unity, una sorta di tool di authoring che permette di creare facilmente scene di gioco. In particolar modo, questo motore permette a chi costruisce il gioco di non dover conoscere necessariamente la programmazione in Unity e in più permette di scaricare i giochi in App in tempo reale mediante una connessione ai server dell'azienda. Infatti, questo framework basato su Unity, sfrutta una struttura ben dettagliata di Game Object all'interno dell'inspector di Unity, che permette di costruire un JSON da consultare in fase di caricamento della scena che indica tutte le risorse da scaricare dai server mettendoli nel preciso ordine con cui è stata creata la scena. Il file JSON contiene una lista dei componenti della scena di gioco sistemata come la struttura interna del gioco stesso, comune per tutti i prodotti di Marshmallow Games, per cui l'applicazione sa dove prendere le risorse e come posizionarle nella scena di gioco. Sebbene possa sembrare lo stesso lavoro che è stato svolto per questa tesi, tale tool di authoring è usato per un'altra tipologia di giochi, non open-ended, dunque con caratteristiche diverse. Inoltre, questo motore proprietario viene usato per prodotti che sfruttano il Canvas di Unity, per cui è "limitato" per questa tipologia di tecnologia. Partendo da questo presupposto, si capisce che per i giochi open-ended serve qualcosa di simile da sviluppare in Unity, magari per adattare in un futuro la logica di caricamento dei giochi usati nel motore già esistente. Per questo motivo come prima cosa si è studiato se il Canvas fosse adatto all'open-ended e dopo svariati esperimenti, sebbene sembrasse una scelta adatta si è optato per sviluppare il tool di authoring open-ended in 2D, cioè con il motore rendering 3D di Unity al completo sfruttando una camera ortografica per la visualizzazione in 2D. Questa scelta è stata effettuata perchè in questa maniera si è potuto sfruttare il motore grafico al massimo delle sue potenzialità, ottenendo flessibilità, maggior performance e varietà di componenti(se volessimo, in futuro, aggiungere la fisica, le collisioni o il particle system per aggiungere nuove interazioni, per esempio, potremmo farlo).

Capitolo 6

La struttura del Drag&Drop

In questa sezione verrà spiegato com'è stato implementato il drag & drop per la camera e per gli oggetti, ponendo particolare attenzione alle problematiche incontrate.

6.1 Drag & Drop in Generale

Prioritario sarà illustrare le scelte tecniche comuni sia per gli oggetti che per la camera, necessarie per implementare il Drag & Drop .

6.1.1 OnMouse... Unity Callback

Partendo dalla scelta di sviluppare il motore in 2D lo step successivo è stato trovare una soluzione per individuare il drag e spostare di conseguenza l'oggetto selezionato. Il primo passaggio sperimentato è stato quello di rilevare questa interazione nella funzione Update che ci fornisce Unity, quindi, per ogni oggetto trascinabile, ad ogni frame del gioco si interroga questa funzione che mediante un "if" rilevava se è stato premuto il tasto adibito per il trascinamento (in questo caso lo schermo). Questo però ha evidenziato che un gioco basato su questa strategia prevede che ad ogni frame è necessario lanciare la funzione Update di tutti gli oggetti trascinabili, dunque non è una strada efficiente. A questo punto, a seguito di ulteriori studi di fattibilità, sono stati scelti come metodo per rilevare il Drag & Drop, gli event handler forniti da MonoBehaviour, la classe base da cui deriva ogni script Unity, cioè: OnMouseDown() , OnMouseDrag() e OnMouseUp(). Queste callback vengono chiamate rispettivamente quando l'utente ha fatto clic su un *Collider* [1] (Down), tenendo premuto il mouse (Drag) e rilasciando infine il clic (Up). Grazie a questi metodi non è più necessario entrare in tutti gli Update di tutti gli oggetti ma verrà chiamato solo il metodo associato al singolo oggetto a cui è attaccato lo script che controlla se è avvenuto un clic, riducendo così le chiamate per controllare se un oggetto è stato trascinato o meno. A seguire è stato necessario ideare algoritmi adeguati alla tipologia dell'oggetto di gioco che si voleva utilizzare in quel momento, ad esempio, la camera e un personaggio, che avranno un movimento differente per cui si è dovuto operare in maniera diversa a seconda se si agisce sulla camera o sugli oggetti di gioco.

[1] Un Collider è il componente base per le collisioni in Unity e definiscono la forma fisica di un oggetto. [30]

6.1.2 La scelta del Box Collider

Come detto nella sezione precedente, le callback usate per gestire questa dinamica di gioco, vengono chiamate quando l'utente fa click su un collider, ragion per cui ogni oggetto trascinabile, che verrà inserito nella scena di gioco usando il framework oggetto di questo lavoro di tesi, dovrà avere allegato un Collider. Nello specifico, è stato scelto di usare i BoxCollider che sono collider a forma di parallelepipedo quindi di forma tridimensionale, che vengono creati automaticamente intorno all'immagine usata per rappresentare l'oggetto di gioco. Seppure il motore di gioco in sviluppo sia destinato per videogiochi in 2D, è stata effettuata questa scelta poichè tutti i Collider2D rendono impossibile captare quale sia l'oggetto da trascinare. Infatti, quando si clicca un Collider2D e dietro di esso ce ne sono altri, i metodi "OnMouse..." vengono scatenati da entrambi, e durante il trascinamento avviene su tutti i Collider2D che si trovano sotto il punto in cui viene cliccato lo schermo. Un'alternativa ai BoxCollider è stata quella di scegliere i PolygonCollider2D, in quanto quest'ultimi creano un collider composto da molti triangoli che seguono la forma dell'immagine visibile fornita come sprite dell'oggetto di gioco. Per utilizzare questo tipo di collider è stato necessario pensare ad una strategia centralizzata, cioè uno script che potesse gestire il rilevamento del clic per tutti gli oggetti e poi reindirizzare il movimento al singolo oggetto che è stato soggetto dell'interazione dell'utente. Seppure sia stata raggiunta una versione che permettesse di valersi di questo vantaggio, impiegare i PolygonCollider non è stato del tutto conveniente perchè la creazione automatica di questi collider, talvolta portava ad avere dei "buchi", cioè delle parti di oggetto che non venivano ricoperte dal collider e quindi a dover aggiustare manualmente questa mancanza, allontanandoci così dall'obiettivo del tool di authoring oggetto di studio, cioè quello di automatizzare e facilitare la creazione di scene open-ended al level designer o artista che componga la scena di gioco. Per questi ed altri motivi, si è optato per sfruttare i BoxCollider come mezzo per rilevare le interazioni di Drag&Drop.

6.2 La Camera

In questa sezione verrà spiegato come viene gestito il drag della camera per scorrere la visualizzazione all'interno dell'ambiente di gioco.

6.2.1 DragCamera

Il drag della camera, come per qualsiasi oggetto di gioco è gestito tramite gli event handler sopracitati. A differenza degli oggetti però il corpo dei metodi "OnMouse..." è molto più semplice. Infatti, per il trascinamento della visuale, basta registrare la posizione della camera nel momento del tocco dello schermo 1 e poi, durante il drag, spostarla nella direzione opposta rispetto a dove è stato spostato il dito dello stesso spazio di trascinamento moltiplicato per la velocità di trascinamento della camera desiderato 2, facilmente configurabile da un campo apposito dove poter inserire tale valore.

6.2.2 Limiti di movimento della Camera

La camera, però, non può scorrere ovunque si voglia; bisogna mantenere la visuale di gioco all'interno dello spazio giocabile previsto. Questo spazio è rappresentato da un'immagine di sfondo a partire dalla quale è possibile calcolare le coordinate globali massime entro cui è possibile spostare la camera e gli oggetti di gioco. È quindi necessario allegare al background uno script che permetta di calcolare questi limiti. Com'è possibile osservare nel seguente metodo 3 , si sfrutta lo SpriteRenderer necessario per visualizzare l'immagine di background e da esso si estrapolano i valori massimi e minimi raggiunti dallo sfondo sugli assi x e y che poi verranno restituiti a chiunque abbia necessità.

Codice 1: primo passso per trascinamento camera

```
1  Vector3 lastPosView;
2
3  void OnMouseDown()
4  {
5    setLastPosView();
6  }
7
8  public void setLastPosView()
9  {
10   lastPosView = Camera.main.ScreenToViewportPoint(Input.mousePosition);
11 }
```

Codice 2: secondo passso per trascinamento camera

```
void OnMouseDrag()
{
  DragCamera();
}

public void DragCamera()
{
  Vector3 newPosView = Camera.main.ScreenToViewportPoint(Input.mousePosition);
  Vector3 cameraMovment = (lastPosView - newPosView) * speedFactor;

  if ((cameraMovment) != Vector3.zero)
  {
    lastPosView = newPosView;

    cameraMovment = gm.Limit2Bound(cameraMovment);

    if (HorizentalDrag)
      Camera.main.transform.Translate(new Vector3(cameraMovment.x, 0, 0));
    if (VerticalDrag)
      Camera.main.transform.Translate(new Vector3(0, cameraMovment.y, 0));
  }
}
```

Codice 3: metodo per ottenere le coordinate limite della scena

```
public Vector2[] CalculateBoundWorlds()
{
  SpriteRenderer m_SpriteRenderer = gameObject.GetComponent<SpriteRenderer>();
  Sprite sprite = m_SpriteRenderer.sprite;

  Bounds bounds = sprite.bounds;
  Vector3 max, min;
  max = bounds.max;
  min = bounds.min;

  Vector3 xBound = transform.TransformPoint(min.x, max.x, 0);
  Vector3 yBound = transform.TransformPoint(min.y, max.y, 0);

  //v1
  Vector2[] boundWorlds = new Vector2[2];
  boundWorlds[0] = new Vector2(xBound.x, xBound.y);
  boundWorlds[1] = new Vector2(yBound.x, yBound.y);

  return boundWorlds;
}
```

Il metodo appena riportato, svolge il compito di restituire gli estremi dello sfondo sotto forma di coordinate globali e siccome queste rimangono invariate, a meno che si cambi scenario con uno di dimensioni differenti, viene chiamato principalmente dal Game Manager, argomento di cui parleremo più avanti nella sezione 6.6. In Particolare, all'interno del Game Manager è contenuto un metodo che utilizza questi limiti per evitare di posizionare la camera fuori dallo spazio di gioco, cioè il metodo **Limit2Bound**. È possibile osservare il suo utilizzo nel codice 4 .

Il metodo di cui parliamo si presenta così:

Codice 4: metodo chiamato ad ogni drag per impedire che la camera esca fuori dallo sfondo

```
public Vector3 Limit2Bound(Vector3 distanceView)
{
  if (distanceView.x < 0) // Check left limit
  {
    if (Camera.main.transform.position.x + distanceView.x < xBoundWorld.x)
    {
      distanceView.x = xBoundWorld.x - Camera.main.transform.position.x;
    }
  }
  else // Check right limit
  {
    if (Camera.main.transform.position.x + distanceView.x > xBoundWorld.y)
    {
      distanceView.x = xBoundWorld.y - Camera.main.transform.position.x;
    }
  }

  if (distanceView.y < 0) // Check down limit
  {
    if (Camera.main.transform.position.y + distanceView.y < yBoundWorld.x)
    {
      distanceView.y = yBoundWorld.x - Camera.main.transform.position.y;
    }
  }
  else // Check top limit
  {
    if (Camera.main.transform.position.y + distanceView.y > yBoundWorld.y)
    {
    distanceView.y = yBoundWorld.y - Camera.main.transform.position.y;
    }
  }

  return distanceView;
}
```

Geometria vs Fisica

Per limitare i movimenti di gioco e farli rimanere all'interno dell'immagine di sfondo, sono state valutate anche altre soluzioni, come quelle di usare il motore fisico messo a disposizione da Unity. La strategia era quella di inserire degli edge collider lungo i lati del background e attivare le collisioni tra Collider, in maniera tale che la fisica potesse gestire automaticamente questa dinamica. Questa maniera di contenere il trascinamento però si è rivelata fallace in quanto per bloccare tutte le collisioni è stato necessario usare un rilevamento continuo anziché discreto in quanto quest'ultimo, di tanto in tanto, permetteva ai collider degli oggetti trascinati di uscire al di fuori dello sfondo. Tuttavia usare il rilevamento continuo è molto dispendioso in prestazioni quindi, sebbene eviti di dover creare un sistema di formule geometriche da consultare ad ogni drag, è stato ritenuto inutile continuare con esso perché, essendo la tipologia di gioco per cui è destinato l'utilizzo del tool di authoring privo di collisioni tra elementi di gioco, si sarebbe dovuta anche gestire l'assenza di tali collisioni, a parte quella con i bordi del background. Dunque, per evitare questa complicazione e risparmiare risorse è stato deciso di continuare con un calcolo geometrico che restituisca i limiti di gioco e contestualmente controlli che tali limiti non siano superati.

6.2.3 Reindirizzamento drag camera

Durante lo sviluppo del motore è stato ritenuto necessario costruire uno script che reindirizzasse il drag di un oggetto di scena al drag della camera. Se il trascinamento viene effettuato a partire da qualche oggetto particolare, esso non si muove ma si effettua un normale movimento della camera e questo perché potrebbero essere presenti degli oggetti che non sono mobili o trascinabili all'interno del gioco ma che hanno un BoxCollider usato per la rilevazione delle interazioni tra oggetti. È probabile infatti, che quando si sta navigando all'interno della scena di gioco, draggando la camera, si clicchi proprio su uno di questi oggetti immobili e che la visuale non si sposti, proprio perchè non associata ad essi nessuna logica di trascinamento. Per questo motivo è stato creato uno script da aggiungere a questa tipologia di elementi cosicchè i loro metodi OnMouseUp e OnMouseDrag richiamino le stesse funzioni usate all'interno delle corrispettive callback della camera.

6.3 Oggetti di gioco

6.3.1 DragOggetto

Come ormai appare chiaro, gli oggetti vengono spostati mediante trascinamento del dito sullo schermo grazie alle funzioni OnMouse della libreria MonoBehaviour. Rispetto alla camera, il drag relativo agli oggetti di gioco è più complesso poiché le azioni e le operazioni da gestire sono maggiori e complicate. Innanzitutto, gli oggetti dovranno seguire il drag e quindi avranno come direzione e verso quella del trascinamento, con l'aggiunta di tutta una serie di controlli che facciano in modo che durante il trascinamento ci siano delle animazioni adeguate a far capire che è disponibile un'interazione e che al rilascio dell'elemento draggato si genera un effetto. Per esempio, se rilasciassimo un colore (es. penna o pastello) su un foglio bianco, la logica di drag & drop riconoscerà l'interazione e colorerà il foglio, oppure se rilascio un oggetto su un pavimento, verrà posizionato adeguatamente alle coordinate corrispettive. Quindi, nella funzione di drag di un oggetto, oltre al calcolo dello spostamento dalla coordinata di inizio drag (OnMouseDown) a quella attuale del dito, la sua funzione esegue una serie di check, che permettano poi alla fine (OnMouseUp) di realizzare un'azione corrispondente a dove si è fermato il trascinamento.
Nel codice riportato 5, si può osservare come venga calcolato lo spostamento da effettuare. Dunque si può controllare che quello che si genera non è uno spostamento nullo e, nel caso in cui non lo fosse, si può rilasciare un metodo per spostare la camera; qualora l'oggetto si avvicinasse ai bordi di essa, viene traslato, limitandone il movimento all'interno della camera. Il controllo sullo spostamento nullo è utile non solo per non sprecare risorse andando avanti nel codice inutilmente, ma anche perchè l'oggetto appena selezionato potrebbe trovarsi parzialmente già al di là della camera per qualche motivo, come un precedente trascinamento di quest'ultima. Mentre si fa tutto ciò l'oggetto viene tenuto in sovraimpressione grazie al metodo setPosition della classe SetPositionOnZ destinata alla gestione della coordinata z di ogni oggetto; ma questo argomento verrà affrontato successivamente nella sezione 6.5.2. Infine, viene lanciato un altro metodo, che verifica tramite la logica su cui è basato il sistema d'interazioni, se si sta posizionando l'oggetto su un altro oggetto con cui può interagire (Vedi capitolo 7).

Codice 5: metodo chiamato durante il drag dell'oggetto

```
1  private void DraggingObject()
2  {
3    Vector3 objectDragPos_debug = GetMouseWorldPos();
4    Vector3 objectMovment = objectDragPos_debug - objectDragOrigin;
5
6    if ((objectMovment) != new Vector3(0f, 0f, 10f))
7    {
8      objectDragOrigin = objectDragPos_debug;
9
10     MoveCamera(objectMovment);
11     objectMovment = LimitObjectBound(objectMovment);
12     transform.Translate(objectMovment);
13
14     sPoZ.Pt = positionType.draggingPos;
15     sPoZ.setPosition(); //set Z Position to dragging
16
17     ic.checkPulse(); //check if a interaction is available and animate a
           feedback for this
18   }
19 }
```

6.3.2 Limiti di movimento

Come è possibile verificare nel codice della precedente sezione 6.3.1 lo spazio di trascinamento dell'oggetto prima di essere applicato viene modificato dalla funzione **LimitObjectBound**. Osservando il metodo LimitObjectBound già dal nome si può notare la somiglianza con Limit2Bound. Infatti, entrambi controllano se sui 4 lati, rispettivamente camera e oggetto rientrano nelle coordinate massime e minime che ci aspettiamo. In particolar modo, nel codice di seguito riportato 6, si è usata la dimensione del Collider dell'oggetto; per capire se l'oggetto stesso rimane all'interno dello spazio di trascinamento al fine di evitare che fuoriesca dalla visuale di gioco, e qualora fosse così, si farà in modo che l'oggetto venga riposizionato cosicché coincida col bordo della camera. Ciò avviene modificando il valore dello spazio di movimento dell'oggetto effettuato, passato come argomento della funzione, calcolando la distanza dal bordo della camera alla coordinata dell'elemento trascinato e restituendo questo valore che poi successivamente verrà usato nella funzione Translate (vedi 5).

6.3.3 Offset di inizio drag

Quando il trascinamento viene effettuato, non è detto che l'oggetto venga trascinato dal suo centro, quindi dal punto delle coordinate di posizione nello spazio, ma potrebbe essere che il giocatore piuttosto tocchi la parte superiore o laterale del centro. Quando ciò avviene, l'oggetto viene centrato sotto il punto in cui il player ha cliccato; ancora prima di iniziare a draggare, diventa perciò necessario considerare questo fattore nei calcoli dello spostamento dell'oggetto di gioco, per cui quando si inizia un drag, calcoliamo questo offset che ci aiuta a non far spostare inutilmente l'oggetto fino a quando non viene mosso il dito. Questo offset è fondamentale, nel caso in cui il trascinamento sia fatto aggiornando di volta in volta il vettore posizione dell'oggetto draggato, se invece, come in questo caso, l'oggetto viene spostato usando la funzione di traslazione, si calcola la singola distanza dal punto di partenza a quello attuale del trascinamento, ignorando il centro dell'oggetto e aggiornando la posizione solo quando questa distanza è maggiore di zero, per cui al tocco non ci sarà movimento finchè il dito, seppur premuto sullo schermo, non si sposta di qualche millimetro.

Codice 6: funzione che non permette all'oggetto draggato di fuoriuscire dai bordi della camera

```
private Vector3 LimitObjectBound(Vector3 dragMovment)
{
  CalculateObjectBounds(out float xMax, out float yMax, out float xMin, out
      float yMin);
  //localSize doesn't give the right dimension of the object but only its
      scale, not the unit
  Vector3 size = coll.bounds.size;
  Vector3 halfSize = size / 2;

  if (dragMovment.x < 0) // Check left limit
  {
    if (transform.position.x + dragMovment.x - halfSize.x < xMin)
    {
      dragMovment.x = xMin - transform.position.x + halfSize.x;
    }
  }
  else if (dragMovment.x > 0) // Check right limit
  {
    if (transform.position.x + dragMovment.x + halfSize.x > xMax)
    {
      dragMovment.x = xMax - transform.position.x - halfSize.x;
    }
  }
  if (dragMovment.y < 0) // Check down limit
  {
    if (transform.position.y + dragMovment.y < yMin)
    {
      dragMovment.y = yMin - transform.position.y;
    }
  }
  else if (dragMovment.y > 0) // Check top limit
  {
    if (transform.position.y + dragMovment.y + size.y > yMax)
    {
      dragMovment.y = yMax - transform.position.y - size.y;
    }
  }

  return dragMovment;
}
```

6.4 Drop Oggetto

In questa sezione verrà trattato cosa accade una volta che si rilascia l'oggetto e qual è la logica che si cela dietro al posizionamento di oggetti che possono essere trasportati in qualsiasi posto nella scena.

6.4.1 Y-Sorting Layer

Dal momento che si è in un ambiente 2D che permette libero movimento agli oggetti di gioco, è possibile che questi si sovrappongano l'un l'altro generando inevitabilmente dei dubbi: quale oggetto viene visualizzato in primo piano rispetto all'altro? Ovvero, quale Sprite viene renderizzata per prima? In quale ordine? Per dissipare queste incertezze ci viene in aiuto un video tutorial [31] che ci mostra come affrontarle poiché questo è un problema classico dei giochi *Beat 'em up* [2]. Prima di tutto bisogna utilizzare dei layer, cioè degli strati che l'inspector window di Unity permette di impostare, dopodichè, li si può ordinare con il sorting layer [33], in modo che ogni layer venga ordinato e renderizzato per prima rispetto ad un altro. In questa maniera, assegnando un determinato tipo di layer ad un oggetto, questo, come altri con lo stesso layer, verrà sempre renderizzato avanti ai layer con minor priorità e dietro a quelli con maggior priorità. Per raggiungere lo scopo di questo lavoro, cioè visualizzare un oggetto avanti rispetto ad un altro, come suggerito dal video citato, assegniamo un valore al sorting layer in base alla posizione sull'asse y occupata dagli oggetti. Tuttavia ciò non basta, perchè come possiamo notare nella immagine a seguire 1, estrapolata dal video, anche se un oggetto, a rigor di prospettiva dovrebbe essere posizionato dietro rispetto all'altro, esso non viene comunque visualizzato in primo piano; questo perchè le posizioni degli oggetti sono assegnati rispetto al *pivot* [3] di una sprite e siccome il sorting layer in questo caso viene assegnato in base alla posizione su Y, se abbiamo delle immagini con pivot centrale, finchè il centro di una sprite, non sorpassa quello dell'altra, il sorting layer basato sulla posizione y, non avrà effetto. In tal caso, potremmo incorrere in situazioni come quelle della fig. 1 dove seppure i piedi di un personaggio sono più in alto di quelli di un altro, non viene visualizzato dietro. Per far fronte a questa problematica, basta utilizzare lo sprite editor messo a disposizione da Unity per spostare il pivot in basso.

[2] Beat 'em up è un genere videoludico tipico dei vecchi cabinati da sala giochi, in cui un personaggio protagonista combatte con molti antagonisti in uno spazio 2D. [32]

[3] Il pivot è un punto immaginario che viene usato come punto ideale per leggere la posizione di un oggetto nella scena 3D. Quindi quando posizioniamo a determinate coordinate un oggetto, stiamo posizionando il pivot della sua sprite

Partendo da in alto a sinistra è possibile vedere come spostando il personaggio piccolo verso l'alto esso non viene visualizzato dietro finchè il suo centro non sorpassa quello del personaggio grande. Al contrario nell'ultima immagine, in basso a destra questo non avviene in quanto i pivot sono stati sposati alla base delle immagini dei personaggi

Figura 1: Comparazione tra gli Y-Sorting Layer render della stessa Sprite con posizione del Pivot differente.

La formula usata per calcolare il valore del sorting Layer in base alla Y dell'oggetto di scena è:

$$sortingOrder_{\text{oggetto}} = \min\left(32763 \cdot \frac{y_{\text{object}}}{Y_{\max}} + alwaysInFront * 2, 32765\right) \qquad (6.1)$$

Il Sorting Order è un intero a 16 bit in Complemento a 2 [34], per cui ha un range di valori possibili compreso tra [-32768, 32767] ma nella formula è tutto riproporzionato per avere come intervallo di valori possibili [-32763,32763] in maniera tale di non incorrere in overflow e riservare dei valori a utilizzi speciali come quello della variabile alwaysInFront che, quando è settata, permette all'oggetto di rimanere sempre in sovraimpressione rispetto agli altri oggetti che si trovano sulla stessa y. Un altro utilizzo dei valori riservati è quello dedicato al trascinamento: come già detto nei paragrafi precedenti, durante questa azione l'oggetto è in sovraimpressione, ciò è reso possibile perchè durante il drag gli si attribuisce il valore massimo disponibile, cioè 32767. Dunque, se un oggetto è posizionato/rilasciato alla Y massima possibile, cioè quella rappresentata dal bordo superiore dello sfondo, avrà un sorting order pari a 32763, se invece sarà posizionato sul bordo inferiore il valore del suo ordinamento sarà -32763 mentre per le posizioni intermedie sarà proporzionato.

6.5 Il problema dei boxCollider e la profondità

Sebbene usare i sorting layer e più in particolare il valore di sorting order permetta di visualizzare in modo corretto oggetti sovrapposti, tuttavia non ne permette ancora il corretto funzionamento. Infatti, come documentato in [35] il sorting order permette di decidere solo l'ordine di visualizzazione degli oggetti, ma questi non sono effettivamente posizionati così, tant'è che una sprite, seppure è visualizzata dietro di un'altra, se ha una z più vicina alla camera, essa verrà catturata dal nostro OnMouseDown e non quella che vediamo davanti. Inoltre, Unity non ha un ordine preciso per quelle situazioni in cui la z coincide, tant'è che di solito quando si va ad usare un OnMouseDown o un rayCastHit, l'oggetto che è stato creato per primo verrà rilevato come più vicino e quindi, come nel nostro caso, anche draggato e dunque potrebbe trovarsi renderizzato dietro quello che volevamo effettivamente trascinare, risultando un funzionamento anomalo. L'unica soluzione trovata per aggirare questo problema, come anche suggerito nell'articolo citato, è quella di posizionare gli oggetti ad una Z ben precisa, riproporzionata in base alla posizione sull'asse y dell'oggetto, come fatto per il sorting layer e come verrà mostrato più avanti.

6.5.1 Set Position On Y

Set Position On Y è il nome di uno script che controlla il comportamento di un oggetto al suo rilascio. Infatti, quando un oggetto viene rilasciato dopo un trascinamento è necessario capire dove questo andrà a finire, se si trova su una parte destinata ad ospitare oggetti, come il suolo ad esempio, così esso verrà rilasciato in quella posizione, mentre invece se venisse rilasciato a mezz'aria, questo cadrà fin quando non incontra una superficie su cui può poggiare, che può essere sempre il suolo o qualcos'altro come un tavolo o uno scaffale. Questo componente gestisce tale funzionamento modificando la Y dell'oggetto rilasciato in base alla tipologia di posizione che esso richiede e a dove si trova. Infatti, un oggetto potrebbe avere un funzionamento speciale o semplicemente trovarsi già in una situazione in cui non è necessario modificare la sua Y, per cui mediante uno switch case, Set Position On Y, capisce quale posizione è stata atttribuita all'oggetto in questione e agisce di conseguenza. Solitamente, il funzionamento di default è quello di ricorrere in una funzione in cui per prima cosa si controlla se ci troviamo sul terreno di gioco o al di sopra di esso, quindi a mezz'aria, dopodichè si calcola la nuova Y che dovrebbe assumere l'oggetto droppato e si ricorre finchè non trova una posizione adatta a dove può stare. Ad esempio, se dovesse essere rilasciato a mezz'aria e sotto di lui dovesse esserci un elemento di gioco molto più grande di esso che comprometterebbe il funzionamento del gioco perchè coprirebbe l'oggetto draggato rendendolo inutilizzabile, l'oggetto non cadrebbe dove era destinato a cadere ma verrebbe fatto cadere appena sotto l'elemento di grandi dimensioni, cosicchè la sua y sia minore e, successivamente l'algoritmo di posizionamento sulla Z possa settare il sorting layer e la sua z in maniera da visualizzarlo davanti e non farlo coprire(si veda sezione 6.5.2). Questo funzionamento rappresenta la situazione che si verifica più frequentemente per gli oggetti, infatti, un'altra casistica potrebbe essere quella di incontrare un oggetto su cui è possibile poggiare delle cose prima che si tocchi il terreno, per cui, a livello logico e visivo, l'oggetto si fermerà su di esso e non sul terreno; oppure si potrebbe verificare che l'oggetto sia già posizionato su una superfice e quando essa verrà spostata, l'oggetto stesso seguirà il suo movimento, ragion per cui non è necessario che venga calcolata anche la sua Y di destinazione in quanto questo avverrà già per la superficie su cui poggia e a cui è legato tramite una "parentela", grazie all'utilizzo del metodo SetParent [36] fornitoci dalla libreria MonoBehaviour.

6.5.2 Set Position On Z

Sviluppando in ambiente 2D con dinamiche potenzialmente infinite di interazioni, bisogna gestire anche la sovrapposizione di oggetti posizionati in luoghi corrispondenti o molto vicini. Come già anticipato nella sezione 6.4.1 si gestisce questa cosa assegnando un valore di sorting order della sprite ben preciso in base alla Y in cui si trova l'oggetto, stessa cosa accade per la coordinata Z. Per gestire questi due valori si è creato questo componente rinominato "SetPositionOnZ.cs" che tendenzialmente viene chiamato dopo il posizionamento sulla Y di "SetPosizionOnY". Ogni oggetto che ha questo componente può avere uno o più stati, come Default, Children, ChildrenBehind o AlwaysInFront, che indicano come devono essere posizionati. Infatti, chiamando il metodo principale, setPosition, si accederà ad uno switch case che in base al tipo di posizionamento selezionato chiamerà il metodo corrispondente che eseguirà i giusti calcoli per quell'oggetto con un determinato stato. Per esempio, un banalissimo oggetto che ha un posizionamento standard avrà semplicemente il suo sorting layer e la sua z proporzionali alla y in cui si trova, mentre un oggetto identificato come figlio, avrà in base alla tipologia di posizionamento che gli è stata attribuita un valore leggermente superiore o inferiore rispetto all'oggetto padre. In generale, il posizionamento sulla z esegue una proporzione ben precisa e cioè:

$$z_{\text{oggettoRilasciato}} = \frac{z_{\text{farClipPlane}} \cdot (y_{\text{object}} + Y_{\text{max}})}{Y_{\text{max}} \cdot 2} + z_{\text{camera}} \tag{6.2}$$

Tuttavia tenendo anche conto che potremmo forzare la posizione avanzata di un oggetto rispetto agli altri con cui si sovrappone, in linguaggio C# si avrà:

Codice 7: calcolo sorting layer e z dell'oggetto

```
private void defaultPositioning()
{
  ...

  sprite.sortingOrder = Mathf.Min((-Mathf.CeilToInt(32763 * transform.position
      .y / gm.YMax) + System.Convert.ToInt32(alwaysInFront) * 2), 32765);

  var zNew = (Camera.main.farClipPlane * (transform.position.y + gm.YMax) / (
      gm.YMax * 2)) + Camera.main.transform.position.z; //inside the camera
      frustrum -> range value [zCam,farClippingPlane]
  transform.position = new Vector3(transform.position.x, transform.position.y,
       zNew + 1); //positioning in front of camera (+1)

  ...

}
```

6.6 Game Manager

In un videogioco, tutti i componenti di una scena, come personaggi, oggetti, mobili, terreno e camera, condividono delle informazioni essenziali e comuni a tutti: parliamo in generale del livello di vita dei personaggi e dei nemici, le loro posizioni, ma anche delle coordinate massime dello spazio di gioco che, nel caso specifico di questo lavoro rappresentano le dimensioni dell'immagine di background che comporranno la scena di gioco. Per non sprecare risorse e salvare tutte queste informazioni, per ciascun GameObject che compone il mondo di gioco, si crea uno script generalmente chiamato GameManager, che gestisce tutte le variabili e/o metodi di gioco che servono a tutti i componenti che lo costituiscono. Il GameManager è dunque un GameObject/Script che rimane in funzione per tutta la durata del gioco, dal suo avvio alla sua chiusura e segue la struttura del design pattern *Singleton* [4]. Una classe che segue la struttura di un Singleton, è una classe che permette di creare una sola istanza di quell'oggetto e quindi, nel caso del GameManager è molto utile per non sprecare risorse, esisterà una e una sola istanza del Game Manager a cui tutti gli altri script attingeranno per reperire le informazioni o le funzioni di cui avevano bisogno (ad esempio, LimitObjectBound). Nel lavoro, il Game Manager ha a disposizione tutte le informazioni relative al background, in maniera tale che in qualsiasi momento un altro oggetto di gioco faccia richiesta di sapere quali siano i limiti massimi su cui potersi spostare (ciò avviene durante il drag, ad esempio), basta che acceda alla sua istanza di Game Manager che si è salvato e reperire tali valori da esso. Inoltre, il Game Manager è stato progettato anche per gestire le dimensioni della camera di gioco, che si deve riproporzionare in base alle risoluzioni dei device su cui andrà ad essere eseguito un ipotetico videogioco creato con il framework sviluppato in questo lavoro di tesi. Questo speciale GameObject è dunque il fulcro su cui si basa tutto il motore, in quanto a partire dai parametri che gli vengono assegnati tramite l'editor panel di unity, crea e ridimensiona l'ambiente di gioco, adattandolo in base alle esigenze della persona che sta usando il tool di authoring per creare il suo gioco 2D. Tra i vari parametri disponibili, tramite il GameManager è possibile modificare le velocità di drag della camera e degli oggetti. In conclusione GameManager, grazie ai numerosi parametri che contiene, è uno script molto utile in quanto la sua funzione principale è centralizzare le parti comuni del gioco.

[4] Singleton è un modello di progettazione creativo che consente di assicurare che una classe abbia solo un'istanza, fornendo al contempo un punto di accesso globale all'istanza stessa. [37]

Capitolo 7

Libertà d'Interagire

Il capitolo in questione illustrerà come è stata ideata la dinamica d'interazione tra oggetti. Si passerà dalla strategia generale su come si è impostato il riconoscimento tra oggetti che possano interagire tra loro, ad un'analisi più dettagliata di come si è riusciti a captare la coppia di oggetti interagibili, capire se questi siano compatibili e lanciare il codice per effettuare l'interazione. Si partirà dall'architettura di interazione quale struttura riutilizzabile e flessibile tra dinamiche diverse che permette di poter modificare e/o implementare nuove tipologie di interazioni.

7.1 L'architettura d'Interazione

7.1.1 I "Ruoli" delle Interazioni

Ancora prima di poter sviluppare il codice per creare delle interazioni tra oggetti, è stato necessario studiare a lungo quale fosse la metodologia più adatta a permettere infinite tipologie di interazioni tra infiniti tipi di oggetti. Infatti, come spesso accaduto durante il progresso di questo progetto, la componente open-ended vincola lo sviluppo a pensare sempre a come rendere il codice e quindi il motore grafico adattabile a modifiche future, in quanto un'interazione che si è pensata oggi, potrebbe essere ideata più avanti nel tempo da qualcun altro e aggiungerla a quelle che il tool di authoring mette a disposizione. C'è stato bisogno, dunque, di trovare una strategia utile che potesse permettere di aggiungere oggetti e interazioni di vario genere senza modificare il codice già esistente, essendo il framework sviluppato in questo lavoro, uno strumento che deve facilitare la costruzione di giochi 2D open-ended, limitando al minimo lo sviluppo di codice da parte dei creator. Per far ciò, le interazioni sviluppate e gli oggetti interessati da queste non dovevano dipendere l'uno dall'altro ed evitare il problema dell'accoppiamento (o *coupling* [1]). La soluzione adoperata per raggiungere questo obbiettivo è stata quella di dividere i protagonisti di un'interazione in due ruoli: Interactor, cioè colui che esegue l'azione e Interactable, colui che subisce l'azione. A loro volta ciascuna interazione è divisa in due tipologie: una attiva e una passiva. L'interazione attiva è quella che si scatena trascinando un Interactor sull'Interactable corrispondente a quell'interazione,

[1] L'Accoppiamento fa riferimento ai legami esistenti tra unità (classi) separate di un programma. In generale, diremo che se due classi dipendono strettamente l'una dall'altra (ovvero hanno molti dettagli che sono legati vicendevolmente) allora esse sono strettamente accoppiate (si parla anche di strong coupling) [38]. Per approfondire [39]

mentre quella passiva è un'azione che si scatena quando un oggetto di tipo Interactable viene trascinato sul suo equivalente Interactor che ha una interazione passiva da svolgere su di esso, subendone l'azione. Ciò rende possibile creare facilmente, senza fare delle modifiche al codice, gli oggetti che possono essere Interactor con un certo set di altri oggetti e Interactable con un altro set di oggetti. Per rendere le idee più chiare un esempio semplice di interazione attiva è il seguente: in gioco abbiamo un tubicino di tempera rossa e una tela bianca, trascino la tempera rossa sulla tela e questa si colora di rosso, in questo caso la tempera è un interactor, cioè un oggetto che colora altri oggetti di tipo "colorabile", mentre la tela è un Interactable della tipologia dei "colorabili" dunque la tempera fa l'azione, la tela la subisce. Un'interazione passiva, invece, potrebbe essere rappresentata da una bilancia sopra la quale, se si poggiano degli oggetti ne mostra il peso. Nel caso di questa interazione passiva, la bilancia è l'interactor che pesa gli oggetti mediante un'azione passiva, mentre un qualsiasi oggetto che ci poggiamo sopra che ha il componente Interactable adeguato, subirà l'azione scatenata dalla bilancia, cioè quella di essere pesato. Dunque nel primo caso chi viene trascinato esegue l'azione sull'oggetto su cui viene rilasciato mentre nel secondo esempio l'oggetto che è fermo fa l'azione sull'oggetto che gli viene trascinato sopra.

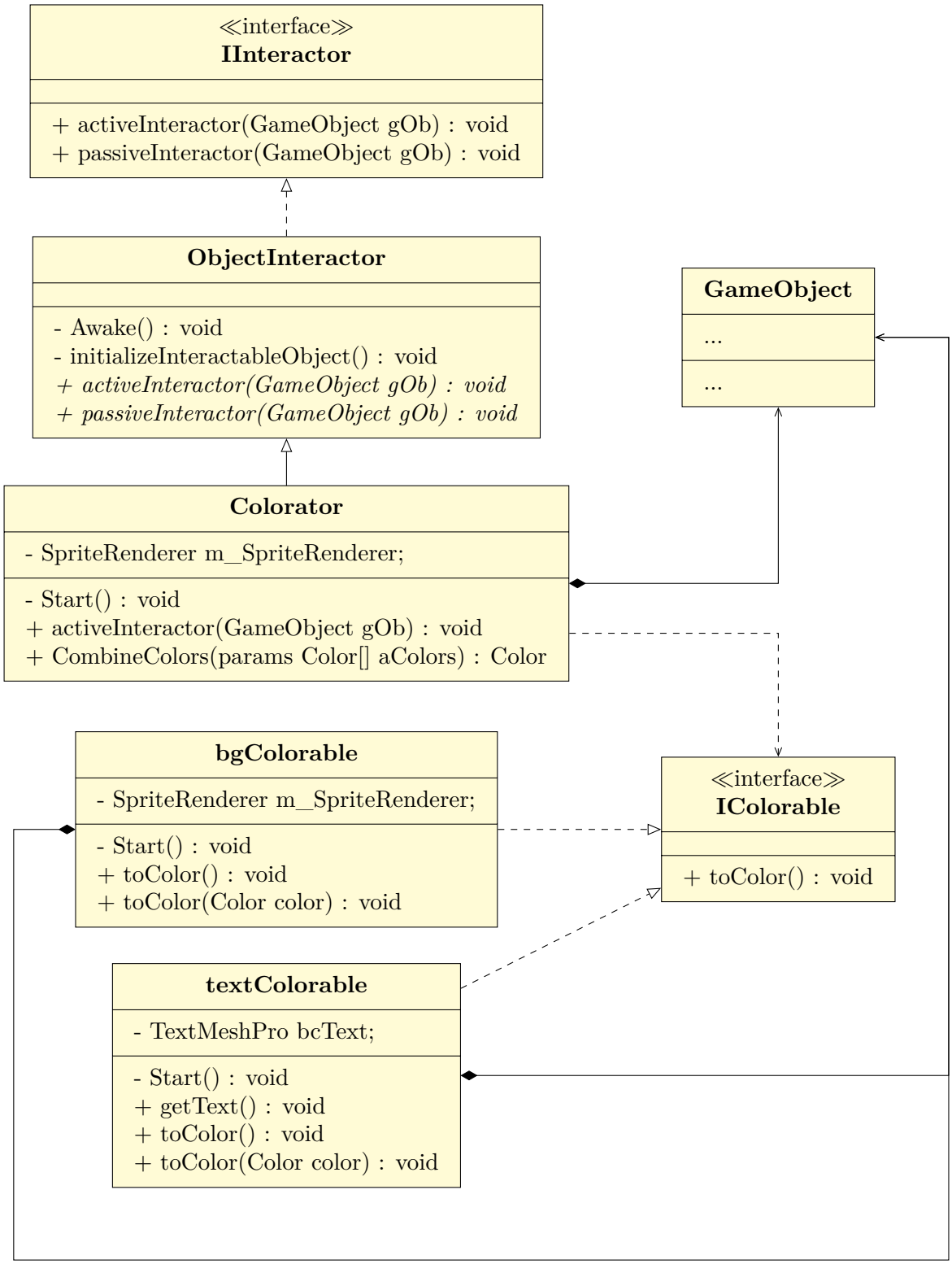
7.1.2 Interactor ed Interactable, i componenti d'interazione

Nella pratica Interactor e Interactable sono delle classi che devono ereditare un qualsiasi componente che rappresenta una nuova interazione. Interactor sa che può interagire con un tipo di interfaccia e sa cosa fare perché richiama la funzione che si aspetta di trovare dentro l'interfaccia stessa mentre Interactable eredita l'interfaccia e ne implementa le funzioni. Seguendo gli esempi fatti nella sezione precedente, il tubicino di tempera rossa avrà un interactor chiamato ad esempio Colorator, mentre la tela bianca avrà un interactable chiamato Colorable. Essi erediteranno rispettivamente la classe Objectinteractor e l'interfaccia IInteractable(caso generico, nell'esempio IColorable). Di conseguenza in questa maniera si vincolano tutti gli interactor e interactable ad avere sempre la stessa struttura base per le interazioni, nello specifico due funzioni per l'interactor chiamate activeInteractor e passiveInteractor, mentre per l'interactable le funzioni definite nell'interfaccia per poter interagire con l'interactor corrispondente, in questo esempio con Colorator. In particolare, Colorator avrà al suo interno un override della funzione activeInteractor che a sua volta contiene il codice per colorare l'oggetto con cui ha interagito, qualora questo fosse un interactable predisposto ad essere colorato e che quindi, in questo caso, ha un componente di tipo IColorable. Questa soluzione è efficace perchè, oltre a non dover modificare il codice di nessuno dei componenti nel caso di introduzione di nuovi oggetti da colorare, l'interactor non deve sapere cosa va a colorare permettendo, senza modificare l'interactor, la gestione dei colorabili in maniere differenti in base alle esigenze del creator. Nel momento in cui un Interactor

sa che l'altro oggetto corrisponde all'IInteractable con cui vuole interagire, chiama semplicemente il metodo che si aspetta di trovare all'interno di quell'interfaccia poi, l'oggetto che è entrato in interazione con lui, avrà il suo codice personalizzato. Riprendendo l'esempio della tela e della tempera si può specificare che una volta che la tempera ha verificato che l'oggetto con cui è andato ad interagire è di tipo IColorable, chiamerà la funzione toColor passandogli il colore rosso come parametro,senza fare altro, senza sapere cosa poi effettivamente accadrà, in quanto ogni componente che implementa l'interfaccia di colorable può avere una sua definizione di ToColor; la tela all'interno di toColor colorerà se stessa o una parte di essa. Invece se si usasse una tela su sfondo bianco con scritta nera, potrebbe essere che il toColor in questione sia programmato per colorare solo il testo o solo lo sfondo ad insaputa del Colorator che avrà eseguito il suo ruolo semplicemente riconoscendo l'interazione e avviandola. Le stesse identiche cose valgono anche nel caso di un'azione passiva, sarà compito dell'oggetto trascinato capire se l'oggetto con cui entra in collisione sia predisposto ad interagire o meno e con quale modalità. Il rilevamento delle interazioni sarà trattato nella prossima sezione 7.2.
Riassumendo: ogni interazione è composta da un oggetto Interactor che esegue l'interazione e un oggetto Interactable che la subisce, il primo segue la struttura di Object Interactor e quindi ha al suo interno metodi per azioni attive e passive, mentre il secondo ha un componente Interactable che cambia in base alla tipologia di interazione che eredita l'interfaccia predisposta ad interagire con l'Interactor. Per una maggiore comprensione prendere visione dei seguenti esempi di codice e del grafico UML.[2] [3]

[2]In ingegneria del software, UML (Unified Modeling Language, "linguaggio di modellizzazione unificato") è un linguaggio di modellazione e di specifica basato sul paradigma orientato agli oggetti [40]. In questo caso particolare è mostrato un Class Diagram, ovvero un UML che consente di descrivere tipi di entità, con le loro caratteristiche e le eventuali relazioni fra questi tipi [41].

[3]Se si osserva bene, l'UML non coincide precisamente con i tipi dei metodi di Colorator perché altrimenti sarebbero usciti riquadri troppo grandi.


≪interface≫
IInteractor
+ activeInteractor(GameObject gOb) : void
+ passiveInteractor(GameObject gOb) : void
ObjectInteractor
- Awake() : void
- initializeInteractableObject() : void
+ activeInteractor(GameObject gOb) : void
+ passiveInteractor(GameObject gOb) : void
GameObject
...
...
Colorator
- SpriteRenderer m_SpriteRenderer;
- Start() : void
+ activeInteractor(GameObject gOb) : void
+ CombineColors(params Color[] aColors) : Color
bgColorable
- SpriteRenderer m_SpriteRenderer;
- Start() : void
+ toColor() : void
+ toColor(Color color) : void
≪interface≫
IColorable
+ toColor() : void
textColorable
- TextMeshPro bcText;
- Start() : void
+ getText() : void
+ toColor() : void
+ toColor(Color color) : void

Codice 8: Componente Colorator da associare ad un oggetto che colora

```
1  public class Colorator : ObjectInteractor
2  {
3    SpriteRenderer m_SpriteRenderer;
4
5    void Start()
6    {
7      m_SpriteRenderer = GetComponent<SpriteRenderer>();
8    }
9
10   public override interactionResult activeInteractor(GameObject
         a_OtherInteractable)
11   {
12     IColorable colorable = a_OtherInteractable.GetComponent<IColorable>();
13     Colorator colorator = a_OtherInteractable.GetComponent<Colorator>();
14     if (colorable != null)
15     {
16       colorable.toColor(m_SpriteRenderer.color);
17       return interactionResult.occurred;
18     }
19     else if (colorator != null)
20     {
21       SpriteRenderer a_OtherSpriteRenderer = a_OtherInteractable.
             GetComponent<SpriteRenderer>();
22       Color newColor = CombineColors(m_SpriteRenderer.color,
             a_OtherSpriteRenderer.color);
23       m_SpriteRenderer.color = newColor;
24       Destroy(a_OtherInteractable);
25       return interactionResult.occurred;
26     }
27     else
28     {
29       Debug.Log("No active Interaction present for this object");
30       return interactionResult.notOccurred;
31     }
32   }
33
34   public static Color CombineColors(params Color[] aColors)
35   {
36     Color result = new Color(0, 0, 0, 0);
37     foreach (Color c in aColors)
38     {
39       result += c;
40     }
41     result /= aColors.Length;
42     return result;
43   }
44 }
```

Codice 9: Componente Colorator da associare ad un oggetto che può essere colorato

```
1  public class bgColorable : MonoBehaviour, IColorable
2  {
3    SpriteRenderer m_SpriteRenderer;
4
5    void Start()
6    {
7      m_SpriteRenderer = GetComponent<SpriteRenderer>();
8    }
9
10   public void toColor(Color color)
11   {
12     m_SpriteRenderer.color = color;
13   }
14
15   public void toColor()
16   {
17     m_SpriteRenderer.color = Color.blue;
18   }
19 }
```

L'esempio presentato è solo un caso base tra quelli possibili che si possono avere per poter spiegare in maniera chiara il funzionamento di questo sistema. Essendo la struttura d'interazione molto personalizzabile senza dover modificare niente del codice originale ma al massimo aggiungendo script che seguono la struttura di Interactor e Interactable, è possibile realizzare interazioni molto più complesse come quelle degli oggetti inseriti nella Demo; tramite questo tool di authoring è possibile aggiungere nuove tipologie di interazione arricchendo così il gioco. Qualora fosse necessario sapere se due oggetti sono compatibili a livello d'interazione, anche se non devono necessariamente svolgere l'azione, si hanno a disposizione da parte di objectInteractor, in aggiunta ad activeInteractor e passiveInteractor, altre 2 funzioni virtuali, canActiveInteract e canPassiveInteract. Non sono state illustrate queste due funzioni in quanto molto specifiche e non necessariamente saranno impiegate, di solito servono per gestire un'animazione di feedback durante la rilevazione di interazioni quando un oggetto viene draggato, ma questo verrà spiegato nel dettaglio nella prossima sezione 7.2. Tuttavia è utile specificare che ogni interazione che segue questa struttura si basa sull'architettura d'interazione creata e gli esempi utilizzati sono semplicemente serviti a spiegarne il funzionamento.

7.2 Rilevare le interazioni

Finora è stato spiegato come costruire degli oggetti interattivi mediante l'utilizzo di Interactor e Interactable, adesso però è necessario captare durante la fase di gioco quando e se due oggetti entrati a contatto possono interagire. Per fare ciò è stato creato un componente apposito chiamato Interactable Checker, il quale, durante il drag&drop di un oggetto verifica se quest'ultimo viene trascinato sopra altri oggetti e se questi sono compatibili per un'interazione. In tal caso, se l'oggetto è ancora in fase di drag, l'altro oggetto verrà animato con una pulsazione per dare un feedback all'utente che qualcosa potrebbe avvenire se rilasciasse l'oggetto di gioco in quel momento: se venisse rilasciato l'interazione verrebbe eseguita. L'animazione di feedback può essere cambiata qualora se ne prediligesse un'altra, per questa fase di sviluppo è stata scelta una pulsazione dell'oggetto fermo nell'ambiente di gioco. Siccome gli oggetti di scena sono tutti posizionati su coordinate z differenti, come spiegato in 6.5, per captare se l'oggetto trascinato sia sopra un altro oggetto, è stato necessario trovare una strategia che potesse permettere di rilevare le collisioni su tutto l'asse z. Infatti, durante il drag dell'oggetto, questo viene sempre messo in sovraimpressione sullo schermo rispetto agli altri oggetti di scena cosicchè durante il trascinamento sia sempre visibile. Se eventualmente, usando i più comuni metodi per le collisioni come OnTriggerEnter oppure OnCollisionEnter, passasse davanti un altro oggetto posto ad una z differente,gli oggetti non verrebbero scatenati perchè a tutti gli effetti non colliderebbero, seppure a livello visivo, in 2D, parrebbe il contrario. È stato usato per rilevare le collisioni, il metodo degli ovelapBox, ovvero una funzione fornita sempre da MonoBehaviour la quale prende in input le dimensioni e il centro di un BoxCollider e restituisce tutti i Collider che si trovano al suo interno. Visto che questo metodo restituisce tutti i collider che tocca, è necessario usare un layerMask per poter indicare i layer che non devono essere considerati durante il calcolo dei Collider che si trovano all'interno dell'overlap box. Quando si inizia a trascinare un oggetto gli si assegnerà il layer indicato nel layerMask in modo che non venga considerato se stesso nell'overlapbox evitando così la collisione. Ciò fa sì che il metodo usato ci restituisca i collider diversi rispetto a quello usato.

Inoltre, la lista di oggetti restituiti da parte di OverlapBox viene riordinata inserendo gli oggetti in ordine crescente considerando il proprio valore della coordinata z, cosicchè poi venga utilizzato l'oggetto più vicino per l'interazione, dato che potrebbero esserci casi in cui più elementi di gioco si trovano nella stessa zona ma a profondità differenti. Come già anticipato in 7.1.2, durante il drag lo script Interactable Checker farà una semplice scansione attraverso CanActiveInteract e CanPassiveInteract per controllare se è possibile eseguire l'azione ossia costatare e vedere se è possibile far partire l'animazione e dimostrare la compatibilità degli oggetti per interagire tra loro. Dunque i metodi CanActiveInteract e CanPassiveInteract degli oggetti interessati restituiscono i booleani e in base ai risulati riportati, viene lanciata l'animazione di pulsazione tramite la libreria DoTween[4]. Entrando nel dettaglio si può vedere nel codice mostrato in 10, che sono state usate misure approssimative di un dito e posizionare il centro dell'OverlapBox a metà della distanza tra la camera e il piano più lontano dello spazio visibile da essa, per poi essere successivamente riordinati secondo la loro posizione sull'asse z.

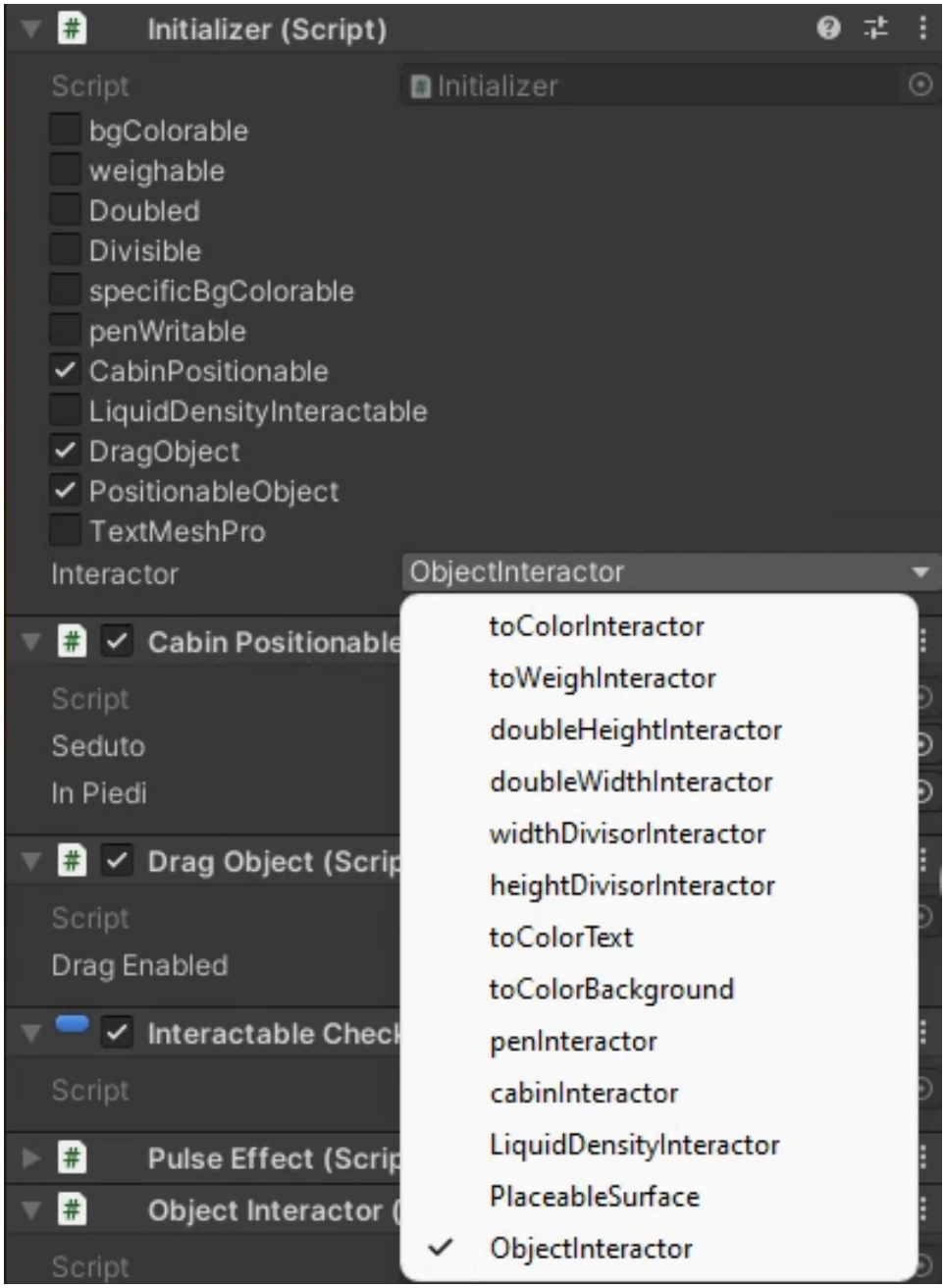
Codice 10: Overlapbox usato per rilevare interazioni

```
public void checkInteraction()
{
  ...

  hitColliders = Physics.OverlapBox(overlapBoxCenDito, overlapBoxDimDito / 2,
      Quaternion.identity, m_LayerMask).OrderBy(c => c.transform.position.z).
      Where(c => c.transform.position.z > transform.position.z)

  ...

}
```

[4]DOTween è un motore di animazione orientato agli oggetti veloce, efficiente e completamente indipendente dai tipi per Unity, ottimizzato per utenti C#, gratuito e open source, con tonnellate di funzionalità avanzate [42]

7.3 Aggiungere interazioni agli oggetti direttamente dall'editor

Introdotto un sistema completo per creare interazioni tra oggetti e rilevazione ed esecuzione di esse, è stato creato uno script per poter automatizzare l'aggiunta di componenti d'interazione con un click, qualora nuovi o vecchi oggetti necessitino di fare una nuova interazione. Questo tipo di script è generalmente chiamato Editor Script, cioè una classe che eredita la classe Editor di MonoBehaviour. Da questa, infatti, si ereditano metodi che modificano l'editor di Unity ampliandone le funzionalità. Nel caso specifico di questo motore di gioco, sono stati creati uno script chiamato Initialiazer, che è lo script base che aggiunge delle funzionalità essenziali al gameobject ed un altro script chiamato InitializerEditor, più complesso, che modifica l'editor per estendere le funzionalità di Initializer per essere direttamente modificate dall'editor. Infatti, tramite questi due script, in particolare con l'editor script, è possibile aggiungere mediante un click nell'editor tutte le funzioni messe a disposizione del framework e cioè tutti gli interactable sviluppati e altre funzionalità speciali quali l'aggiunta di testo o il drag degli oggetti ma anche semplicemente per assegnargli il ruolo di Interactor. Nella seguente immagine 2 c'è il risultato ottenuto nel caso di Initializer.



In questo screen è possibile visualizzare alcuni campi "toggle", cioè campi che si possono selezionare e deselezionare e che quindi aggiungono o tolgono il componente corrispondente, o un campo "popup" da cui è posibile scegliere se si vuole rendere l'oggetto in questione un Interactor aggiungendo il componente corrispondente

Figura 2: Sezione dell'editor che mostra i campi inseriti ad Initializer per aggiungere componenti al game object

Probabilmente, la parte più macchinosa di questo progetto è aggiungere nuove opzioni all'editor in questione. Infatti, non esiste un processo automatico che permetta a nuove funzioni ed interazioni appena create di essere aggiunte al dataset disponibile in questo script editor, però non è così difficile aggiungerle a mano, serve solo un po' di conoscenza base di programmazione in Unity e C#. Se si volesse aggiungere qualcosa a questo editor, basterebbe inserire una nuova voce alla lista di tipologie di funzionalità ed una chiamata ad un metodo che è stato sviluppato per accettare qualsiasi tipo di componente. Per una migliore comprensione visualizzare il codice 11 nella pagina accanto.
Com'è possibile notare nel codice ci sono array per Interactable, Interactors e funzionalità speciali che non rientrano né nel primo né nel secondo array. Quindi per aggiungere nuovi campi all'editor innazitutto bisogna aggiungerli nell'array corrispondente, dopodichè nel metodo OnInspectorGUI va invocato il metodo DrawToggleComponent se si volessero aggiungere Interactable o caratteristiche speciali (in questo caso è stato usato l'esempio di bgColorable). Mentre, per gli Interactor basta lasciare l'unica chiamata a DrawComponentsPopup che gestisce l'array di Interactor e permette di selezionarne solo uno alla volta, rimuovendo il precedente se già presente.

Codice 11: Pezzo di codice dell'Editor Script InitializerEditor

```
[CustomEditor(typeof(Initializer))]
public class InitializerEditor : Editor
{
  private static readonly Type[] interactorsType = new Type[]
  {
    typeof(Colorator),
    typeof(toWeighInteractor),
    typeof(doubleHeightInteractor),
    typeof(doubleWidthInteractor),
    typeof(widthDivisorInteractor),
    typeof(heightDivisorInteractor),
    typeof(toColorText),
    typeof(toColorBackground),
    typeof(penInteractor),
    typeof(cabinInteractor),
    typeof(LiquidDensityInteractor),
    typeof(PlaceableSurface),
    typeof(ObjectInteractor)
  };
  private static string[] interactors;

  private static readonly Type[] specialInteractable = new Type[]
  {
    typeof(TextMeshPro),
    typeof(FerrisWheelManager),
    typeof(PlaceableSurface)
  };

  ...

  public override void OnInspectorGUI()
  {
    DrawDefaultInspector();
    Initializer script = (Initializer)target;

    ...

    DrawToggleComponent(script.gameObject, out bgColorable bgc, onAdd: mr =>
        Debug.Log("bgColorable added"), onRemove: ma => Debug.Log("bgColorable
         removed"));

    DrawComponentsPopup(script.gameObject, interactors, interactorsType, "
        Interactor");
  }

  ...
}
```

Parte III

LA DEMO

Capitolo 8

Demo di Gioco

Per mostrare i risultati ottenuti con lo sviluppo del motore di gioco è stata ideata una Demo dimostrativa. Questo capitolo ha il compito di descrivere come è stata ideata e spiegarne le funzionalità salienti.

8.1 Ambientazione

Per unire gioco e STEM è stata pensata un'ambientazione che potesse unire le due cose. Per questo motivo si è pensato ad una fiera americana di campagna che di solito include attrazioni varie, tra cui giostre, tendoni con attività differenti e animali. Nel caso particolare di questa demo sono state inserite una ruota panoramica, un tendone scientifico e personaggi di animali con caratteristiche umane. Le dinamiche STEM, di cui parleremo in sezione 8.2.1, sono state inserite nell'interazione con la ruota panoramica ed un contenitore di liquidi.

8.2 Funzionalità

La Demo mostra quello che è un'ipotetica scena di gioco, cioè un luna park all'interno del quale il bambino può giocare liberamente usando la sua fantasia. Può trascinare la camera per muoversi tra il tendone di scienze, il cielo e il terreno, oppure può trascinare i personaggi in giro per la scena e farli interagire con altri elementi di gioco. Tra questi elementi osserviamo una ruota panoramica, un contenitore con dei liquidi per fare esperimenti sulla densità di quest'ultimi e delle superfici su cui è possibile poggiare gli oggetti e i personaggi. Nel tendone di scienze si troveranno scaffali con ampolle, un orso scienziato e un contenitore di liquidi con cui potersi divertire. Con la ruota panoramica invece si potrà cambiare la forma delle cabine, inserirci dei personaggi e la si potrà far girare; ci sono anche superfici d'appoggio, come il tetto del tendone o le balle di fieno sulle quali è possibile posizionare gli elementi di gioco. Sebbene sembri un gioco molto semplice, quello che c'è dietro è più complicato e rappresenta il vero cuore del motore di gioco. Difatti, partendo da ciò che si è sviluppato mediante la ricerca di un sistema adattabile all'introduzione di nuovi oggetti open-ended, si sono creati oggetti specifici che da questo momento in poi sono inseribili facilmente nella scena (o in altre creazioni future). Ad esempio, chi un giorno andrà a costruire un'area di gioco dove inserire la ruota panoramica, trascinerà semplicemente il prefab all'interno della scena di unity personalizzandola a piacere dall'editor di unity, oppure potrà creare da zero con un click utilizzando

lo script editor descritto in precedenza e personalizzarla direttamente dall'editor. Ciò che di davvero importante si cela dietro la Demo, dunque, è la possibilità di introdurre questi oggetti di gioco e personalizzarli come si vuole.
Il designer dell'ambientazione potrà scegliere quante cabine dovrà avere la ruota panoramica, quanto dovrà essere lunga la sequenza da far indovinare al giocatore per farla girare (la sequenza sarà sempre lunga tanto quanto un divisore del numero di cabine), potrà personalizzare la velocità con cui la ruota conclude un giro e altro ancora. Allo stesso modo, per il gioco sulla densità dei fluidi, basterà inserire in scena un contenitore e dei liquidi e definire quanti ne potrà contenere, la sprite del liquido e la sua densità, in maniera tale che poi quando il player userà questa dinamica di gioco, riceverà un feedback veritiero e cioè che il liquido a densità maggiore si poserà sul fondo e quelli a densità minore pian piano si ritroveranno uno sopra l'altro. Poi ci sarà una funzionalità per svuotare il contenitore e da lì si potrà ripetere di nuovo "l'esperimento"

8.2.1 Dinamica STEM

Questa breve sotto sezione ha il compito di indicare dove è stata inserita questa dinamica STEM all'interno della Demo. In particolare, le STEM (acronimo di Science, Technology, Engineering and Mathematics ossia Scienza, Tecnologia, Ingegneria. e Matematica) possono essere individuate nella ruota panoramica come la presenza del contenitore dei liquidi e la forza di gravità (non realistica ma funzionale al gioco). Con la ruota panoramica il giocatore dovrà stimolare la logica in quanto al centro di essa, ci sarà un tabellone che indica una sequenza di cabine che deve essere riprodotta esattamente per far girare la ruota. Infatti, le cabine della ruota panoramica avranno la forma di un ortaggio, altro elemento simbolo dei luna park tipici delle fiere di paese americane e l'utente potrà cliccarle per fargli cambiare forma da un ortaggio all'altro. Sarà l'intuizione del bambino a fargli capire che dovrà riprodurre la sequenza centrale della ruota per ricevere un feedback di successo identificata con la rotazione della ruota. Successivamente la ruota panoramica potrà generare una nuova sequenza da riprodurre. La gravità, invece, contribuisce al senso di peso degli oggetti, sebbene non sia una gravità realistica, comunque gli oggetti non possono fluttuare nell'aria e si poggiano sul terreno o altri oggetti predisposti. Il contenitore, invece, permette di sperimentare la densità dei liquidi che vengono inseriti al suo interno, difatti un esperimento tipico di scienze a scuola è quello di inserire in un bicchiere diversi liquidi di densità differente per dimostrare che non si mescolano ma si posizionano a profondità diverse.

8.3 Le interazioni nella demo: i componenti di gioco

In questa sezione verranno descritte alcune delle interazioni più interessanti inserite appositamente per la realizzazione della demo e che rimarranno nel dataset di interazioni disponibili nel motore di gioco. Infatti,(vedi 7.1) essendo la struttura d'interazione molto flessibile all'aggiunta di nuove tipologie di azioni tra oggetti di gioco, gli script che si inseriscono per raggiungere una nuova dinamica possono essere salvati nel framework sviluppato cosicchè poi possa essere riutilizzato per altri progetti.

8.3.1 Ruota Panoramica

La ruota panoramica è probabilmente il componente di gioco più complesso che è stato implementato, difatti, non è un singolo elemento ma è una composizione di più gameobject. La base su cui poggia, la ruota stessa, il tabellone e le cabine sono elementi a sé stanti, uniti tra loro mediante un rapporto di parentela tipico di Unity che li lega e gli permette di essere considerati o come un'unica identità o identità diverse. Bisogna riuscire a coordinare perfettamente tutti gli elementi in quanto ognuno ha funzionalità differenti: ad esempio le cabine sono a sé stanti e possono cambiare forma ad ogni click ma se poi la sequenza centrale viene riprodotta, la ruota deve girare e le cabine girano con essa senza tuttavia che quest'ultime perdino il loro orientamento, infatti, se facessimo girare la ruota senza tener conto delle cabine, queste ruoterebbero con essa ma si ritroverebbero a testa in giù, invece per riprodurre l'effetto di una cabina mobile agganciata alla ruota solo all'estremità superiore, bisogna che ogni cabina sia orientata con la propria base verso il basso. Tutto questo non è riproducibile facilmente, ma c'è una struttura di classi, event&delegates ed elementi da coordinare precisamente. Per esempio, ognuno di questi elementi che compone la ruota, avrà degli script specifici per gestirli: la cabina avrà un manager che controlla se essa viene cliccata, cambierà forma, controllerà se la sequenza voluta è ottenuta e avvierà così la rotazione; anche la circonferenza ha un suo manager che gestisce il numero delle cabine, dove posizionarle e con quale velocità deve ruotare. La cabina, inoltre, avrà un interactor passivo chiamato cabin interactor, il quale si attiva qualora si trascinasse sulla cabina stessa un oggetto di gioco a cui è stato aggiunto il componente cabin interactable. Inoltre, una volta che un oggetto viene inserito nella cabina, questo viene legato ad essa, rendendola suo parent, facendo sì che quando la ruota gira, con sé girino anche le cabine e l'oggetto inserito in esse. La parte più ostica in questo caso, invece, è gestire il trascinamento dell'oggetto che si trova in cabina al di fuori di essa; infatti si deve far in modo che quando si dragga l'oggetto, esso si svincoli dalla cabina. Tuttavia non è necessario che ad ogni drag un oggetto si separi da un altro, ma sarà fondamentale creare un evento tale per verificare alcune condizioni, cosicché esso si sleghi dal suo parent

che in questo caso è la cabina.
Inoltre, per rendere facile e veloce la creazione della ruota panoramica nella sua interezza, essa è stata aggiunta al set di funzionalità disponibili dell'editor di Initializer cosicchè partendo anche da un oggetto vuoto, aggiungendo initiliazer e cliccando sull'opzione della ruota, essa viene creata in automatico. Il creatore del gioco che usa il motore grafico sviluppato in questo lavoro di tesi, non deve far altro che aggiungere le sprite con cui vuole rappresentare graficamente la ruota. A sua volta è stato creato uno script editor per poterla personalizzare in maniera rapida, infatti lo script editor in questione permette di scegliere quante cabine la ruota deve avere, quale sia la velocità di rotazione, creare la sequenza e quanto lunga questa sequenza deve essere al centro.Tutte le parti della ruota così come finora descritte, con grande dispendio di energie, sono state implementate volutamente nella demo, aggiungendo la ruota così "costruita" nella sua totalità come componente interno del framework per far sì che chiunque andrà ad usarlo successivamente, potrà crearla facilmente in pochi click.

8.3.2 Esperimento densità liquidi

Nella Demo sono stati inseriti dei componenti necessari alla dinamica STEM, infatti sono disponibili delle ampolle con dei liquidi al loro interno e un contenitore. Questi elementi sono necessari e vengono usati per riprodurre un semplice esperimento di scienze sulla densità dei liquidi. Infatti, le ampolle contengono acqua, sapone liquido, miele e olio, tutti elementi di densità differenti. Per questo motivo, quando inseriti nello stesso contenitore non si mescolano ma si posano uno sopra l'altro, precisamente dal basso verso l'alto: miele, sapone, acqua, olio. Ampolle e contenitore interagiscono seguendo la struttura descritta nella sezione 7.1.2 e cioè mediante LiquidDensityInteractor e LiquidDensityInteractable. Il primo script, attraverso una mappa, più tecnicamente un sorted dictionary personalizzato, gestisce i liquidi che interagiscono con il contenitore e li ordina e posiziona in base al valore della densità associata a ciascun liquido. Oltre a questo, il componente associato al contenitore controlla se gli oggetti con cui entra a contatto siano dei ILiquidDensityInteractable (l'interfaccia ereditata da LiquidDensityInteractable) e li inserisce al suo interno se c'è ancora spazio. Inoltre LiquidDensityInteractor attiva il bottone per svuotare il contenitore non appena gli venga inserito il primo liquido all'interno. Per quanto riguarda l'interactable associato ai liquidi, invece, non fa altro che restituire la densità al contenitore per fargli capire quale sia la sua posizione all'interno di esso. Anche in questo caso, come quello della ruota panoramica, una volta che un'ampolla viene inserita all'interno del contenitore, questi elementi si legano tra loro con un legame di parentela che verrà poi gestito con un evento lanciato dal bottone di svuotamento.

Visto che i liquidi non possono essere rimossi a mano dall'interno del contenitore per svuotarlo se non utilizzando il bottone, ne viene disabilitato il drag&drop. Il contentore è ideato per non essere spostato dal giocatore, per questo il drag su di esso è reindirizzato alla camera ma, qualora si volesse renderlo trascinabile, basterebbe cliccare sulla voce corrispondente a DragObject nell'editor di Intializer per aggiungerlo. In questo caso è già implementata una logica tra Interactor e Interactable che gestisce lo spostamento dei liquidi interni al contenitore qualora esso venisse spostato.

8.3.3 Superfici d'Appoggio

Altro elemento che è stato inserito durante la costruzione della Demo è quello di superfici sopra cui gli oggetti possano essere poggiati o possano cadere. La particolarità principale di questi elementi è quella che già è stata descritta nella sezione 8.3.2, cioè del legame che intercorre tra gli elementi poggiati su una superficie e la superficie stessa. In particolare, potendo trascinare alcuni oggetti designati come superficie d'appoggio è stato necessario utilizzare una dinamica che permettesse di draggarlo anche con oggetti soprastanti. Per ottenere il funzionamento di questi arnesi sono stati creati PositionableObject (Interactable) e SurfacePositionable (Interactor); il primo script (Interactable) si iscrive all'evento di gestione della "collocazione" da parte della superficie e cambia la sua posizione da default a children (vedi sezione 6.5.1 e sezione 6.5.2), e verfica se l'altro oggetto è posizionabile, e se cade interamente sopra di esso e gestisce il movimento degli oggetti soprastanti, di cui parlavamo nelle righe precedenti. Presente anche in questo caso l'evento che permette di scindere gli oggetti posizionati sulla superficie qualora vengano trascinati via. L'utilizzo della struttura per l'aggiunta di interazioni permette di rendere alcuni oggetti posizionabili e altri no, ad esempio è possibile scegliere se far poggiare una sopra l'altra, due o più superfici. Per creare oggetti e superfici poggiabili è sempre sufficiente cliccare il campo corrispondente nell'editor di Unity.

Parte IV

Conclusioni

Capitolo 9

Risultati ottenuti

Dopo un'attenta attività di ricerca, di ideazione, progettualità e creazione, il risultato ottenuto con il lavoro svolto presso Marshmallow Games è la produzione di un framework da aggiungere ad Unity che renda facilmente fruibile un motore di gioco per la creazione semplice ed immediata di giochi 2D open-ended. Difatti con questo framework sarà possibile riprodurre giochi simili alla Demo descritta nel capitolo 8 senza l'utilizzo di linguaggi di programmazione ma usando solo le funzionalità implementate e le immagini necessarie per creare personaggi, sfondo e oggetti. Inoltre il framework permette facilmente di essere aggiornato da futuri sviluppatori che lavoreranno su questo progetto grazie alla semplicità nell'estensione della struttura su cui è basato. Infatti, lo sviluppo di questo motore è stato pensato proprio per essere il più flessibile possibile e poter essere, in futuro, utile a tutti e adoperato a lungo.

Capitolo 10

Miglioramenti Futuri

Sebbene si sia ottenuto il risultato sperato e funzioni tutto come ci si aspettava senza problemi, ci sono un'infinità di miglioramenti che sarà possibile apportare a questo progetto. D'altronde dovendo permettere di creare giochi open-ended che forniscano infinite possibilità di gameplay al giocatore, sono illimitate anche le funzionalità da aggiungere al motore di gioco. Tra le cose da poter aggiungere alla demo possiamo nominarne alcune, come la possibilità di vestire i personaggi con capi d'abbigliamento e cappelli, poterne cambiare la capigliatura, dal taglio al colore, poter arredare la stanza di gioco e molto altro. A livello più tecnico si potrebbe pensare di migliorare la camera cosicchè possa seguire l'oggetto draggato a velocità sempre più alte man mano che l'oggetto trascinato si avvicina al bordo dello schermo oppure introdurre un Object Manager che possa gestire tutte le risorse condivise tra più componenti di uno stesso oggetto, come per esempio lo Sprite Renderer che spesso è utilizzato da più script associati agli elementi di gioco. Ma si potrebbe anche migliorare la ruota panoramica per essere ancora più personalizzabile, come ad esempio modificare il numero di cabine e/o la lunghezza della sequenza dopo ogni reset di essa e ancora, si potrebbe pensare ad utilizzare un file JSON per salvare i componenti di una scena appena creata. Infine, la cosa più utile in assoluto sarebbe adattare questo framework al tool di authoring interno di Marshmallow Games, quello usato dall'azienda per creare attualmente i propri giochi. Infatti, poter ampliare le potenzialità del motore propietario di casa Marshmallow sarebbe la conclusione perfetta di un progetto di questa fattura.

Bibliografia

[14] *Videogame and Education*. Link (cit. a p. 9).

[43] *e-Learning and the Science of Instruction*. Link.

[44] «American Psychological Association: The Benefits of Playing Video Game». In: (). Link.

Sitografia

[1] *Cos'è il fotorealismo*. Link (cit. a p. 6).

[2] *Videogioco come palestra d'apprendimento*. Link (cit. a p. 6).

[3] *Videogioco Educativo*. Link (cit. a p. 7).

[4] *LogoProgramming, il primo videogioco educativo*. Link (cit. a p. 7).

[5] *Cynthia Solomon, su logothings.wikispaces.com*. Link (cit. a p. 7).

[6] *Lemonade Stand*. Link (cit. a p. 7).

[7] *Edutainment Wiki*. Link (cit. a p. 7).

[8] *The Learning Company (TLC)*. Link (cit. a p. 7).

[9] *The Oregon Trail*. Link (cit. a p. 8).

[10] *Where in the World Is Carmen Sandiego?* Link (cit. a p. 8).

[11] *Carmen Sandiego Returns*. Link (cit. a p. 8).

[12] *Museum Madness (1994)*. Link (cit. a p. 8).

[13] *Ready Robot Club (1994)*. Link (cit. a p. 8).

[15] *Cosa sono i giocattoli "open-ended" e perché noi adulti non li capiamo (...ma i bambini li adorano!)* Link (cit. a p. 11).

[16] *Videogiochi Open World*. Link (cit. a p. 12).

[17] *Toca Life World*. Link (cit. a p. 12).

[18] *Fiete World*. Link (cit. a p. 13).

[19] *Sagomini World*. Link (cit. a p. 13).

[20] *Dr. Panda Città*. Link (cit. a p. 13).

[21] *Collaborazione Unicef e Smart Tales*. Link (cit. a p. 16).

[22] *Unity3D Wikipedia*. Link (cit. a p. 17).

[23] *Unity3D*. Link (cit. a p. 17).

[24] *Definizione di Sprite Wikipedia*. Link (cit. a p. 18).

[25] *Spine*. Link (cit. a p. 18).

[26] *Spine-Unity*. Link (cit. a p. 18).

[27] *Framework cosa sono e quali dominano le classifiche*. Link (cit. a p. 20).

[28] *Che cos'è un framework*. Link (cit. a p. 20).

[29] *Why use a framework*. Link (cit. a p. 20).

[30] *Manuale di Unity - Pagina sui Collider*. Link (cit. a p. 25).

[31] *Y-Axis Sorting in 2D Beat em up style games*. Link (cit. a p. 35).

[32] *Beat 'em up wiki*. Link (cit. a p. 35).

[33] *Sorting Layer Scripting Documentation*. Link (cit. a p. 35).

[34] *Numero Binario con notazione Complemento a 2*. Link (cit. a p. 37).

[35] *Numero Binario con notazione Complemento a 2*. Link (cit. a p. 37).

[36] *Set Parent Scripting Documentation*. Link (cit. a p. 38).

[37] *Singleton Deisgn Pattern*. Link (cit. a p. 40).

[38] *Coesione e Accoppiamento*. Link (cit. a p. 41).

[39] *Coupling (computer programming)*. Link (cit. a p. 41).

[40] *UML: Unified Modeling Language*. Link (cit. a p. 43).

[41] *UML Class Diagram*. Link (cit. a p. 43).

[42] *DoTween*. Link (cit. a p. 48).

[45] *Game based learning, gamification e didattica: cosa sono*. Link.