Unity per lo sviluppo di applicazioni

Matteo Bulgarelli - matricola: 142983

Anno accademico 2022/2023

Dipartimento di ingegneria Enzo Ferrari

Relatore: Professo Nicola Bicocchi

Indice

1 Introduzione

Questa tesi ha lo scopo di illustrare la piattaforma Unity per la creazione di contenuti che spaziano dai videogames, ad ambienti interattivi in ambito architetturale o automotive, fino a creare veri e propri film completamente digitali o semplici applicazioni gestionali per il monitoraggio dell'azienda. L'elaborato cerca quindi di illustrare al meglio le potenzialità di Unity per concretizzare le proprie idee tramite gli strumenti e i supporti che mette a disposizione dell'utente, cercando anche di paragonarlo ad altri due colossi per la creazione di contenuti. La tesi si suddivide in 3 macro-sezioni:

- Una prima sezione dedicata a illustrare l'ambiente in sé, le tecnologie usufruibili al suo interno, le risorse fondamentali per creare un qualunque progetto e come "muoversi" all'interno dell'ambiente per reperirle e utilizzarle al massimo

- Una seconda sezione dedicata in particolare allo sviluppo di videogiochi tramite Unity, con le principali funzionalità che mette a disposizione dell'utente e il modo in cui riesce a farlo per rendere tutto il più facile possibile e permettere a chiunque di imparare velocemente a creare un videogioco in pochi semplici passaggi. In questa parte verrà anche operato un confronto con uno dei principali rivali di Unity nel settore videoludico, ovvero UnrealEngine

- Una terza sezione in cui ci si concentra invece più sull'aspetto di creazione di contenuto artistico di Unity, che è stato per anni un grosso downside del framework, ma che ora dopo vari aggiornamenti e modifiche si è voluto fino ad arrivare a livelli molto alti, che lo hanno reso capace di competere a pieno titolo con altri software per la creazione di scene e modelli 3D o 2D. In questa sezione si farà riferimento ad un altro software, Blender, (che si cercherà di presentare brevemente), che prima degli ultimi update era uno dei principali strumenti di supporto a Unity per creare modelli da importare nell'ambiente per renderli "vivi".

2 Cos'è Unity

Unity è un motore grafico multi piattaforma che consente di realizzare contenuti interattivi in tempo reale, tra cui videogiochi, applicazioni gestionali, ambienti e paesaggi esplorabili, animazioni... L'idea di base dietro a Unity è che l'unico limite sia l'immaginazione del fruitore, per cui qualunque cosa

richieda interazione in tempo reale debba poter essere sviluppato nel minor tempo e il più facilmente possibile, mantenendo un livello di qualità estremamente elevato. Il software è supportato su tutti e 3 i principali sistemi operativi, quindi Windows, iOS e Linux, e nel caso particolare dei videogiochi questi sono progettabili per quasi tutte le piattaforme al momento sul mercato: l'unica assente all'appello, ma si pensa ancora per poco, è Google Stadia. Date le sue funzionalità è diventato abbastanza recentemente anche uno strumento di punta per la creazione di contenuti destinati dispositivi di Realtà Virtuale (Virtual Reality) e di Realtà Aumentata (Augmented Reality), da quando con un recente aggiornamento è stato reso compatibile con il visore per la realtà aumentata.

2.1 Uno sguardo iniziale al framework

Quando si apre Unity viene aperta una finestra del launcher, chiamata UnityHub, che permette sia di creare nuovi progetti e iniziare subito a creare il nostro applicativo, ma presenta anche 3 ulteriori voci: *Learn, Community e Installs*. Le prime due sono di particolare importanza in quanto esprimono al meglio la missione principale del framework, ovvero rendere tutto il più semplice possibile sia per neofiti dell'applicazione, sia per coloro che hanno già iniziato a utilizzare Unity e durante l'utilizzo hanno trovato degli ostacoli che non riescono a risolvere da soli. La sezione "*Learn*" propone una moltitudine di tutorial, sia su funzionalità di base (che permettono a un nuovo utilizzatore di capire i concetti più importanti per iniziare il prima possibile a sviluppare il proprio progetto), sia su funzionalità più avanzate destinate a chi invece ha già confidenza con il software, per rifinire e migliorare il risultato finale della sua applicazione. La sezione "*Community*" invece mostra una lista di canali con cui l'utente può interfacciarsi, sia per risolvere problemi (rivolgendo domande sul forum, o cercando le risposte pubblicate da altri utenti su problemi analoghi), o per pubblicare il proprio prodotto finito in modo da ricevere feedback da altri utenti tramite testing, o per approfondire più nello specifico determinati argomenti grazie a lezioni online in formato video o testuale.

2.2 Come si presenta e cosa propone

Una volta creato il nostro progetto Unity si presenta con un'interfaccia composta da vari pannelli, di cui 4 sono sicuramente i più importanti per lo sviluppo di qualsiasi applicativo, e meritano una breve spiegazione:

- *Scene*: il pannello principale che mostra il contenuto dell'ambiente virtuale su cui si lavora, in cui è possibile posizionare in generale ogni oggetto che si vuole sia osservabile da chi utilizzerà l'applicazione, che sia 2D o 3D. É possibile posizionare oggetti all'interno della scena in due modi principali: o tramite drag-and-drop dal pannello "Project"(qui sotto spiegato), oppure in caso si necessiti di aggiungere oggetti basilari come figure solide semplici, punti di luce o videocamere, semplicemente usando il tasto destro e creando l'oggetto desiderato dal menù a comparsa che compare. Da questa sezione dell'interfaccia è possibile anche avviare una simulazione del progetto in modo da testare se le funzionalità implementate tramite codice o componenti siano funzionanti, quindi per eseguire testing e prototyping.

- *Hierarchy*: è un pannello che permette di vedere tutti gli oggetti posizionati all'interno della scena, in particolare permettendo di vedere anche le possibili correlazioni gerarchiche padre-figlio tra i vari oggetti. In questo pannello è possibile anche creare nuove Scene, che verranno spiegate in maggiore dettaglio nella sezione *Unity per il game-development*

- *Inspector*: selezionando all'interno della scena un oggetto, all'interno di questo pannello compariranno i vari componenti che lo costituiscono e definiscono il suo comportamento. I componenti degli oggetti possono essere sia presi da una vasta gamma di default presente fin dall'installazione di Unity, o scaricati dal web e importati, o anche semplicemente creati dal programmatore stesso in C#, JavaScript e Boo(un linguaggio derivato dal Python)

- *Project*: all'interno di questo pannello si trovano tutti gli oggetti che vengono creati dall'utente all'interno dell'ambiente, sia scripts che modelli 2D/3D, o importati dall'esterno, siano essi scaricati da internet o creati sempre dall'utente tramite software di terze parti. All'apertura di un nuovo progetto, all'interno di questo pannello si troveranno in particolare 2 cartelle: la cartella *Assets*, in cui verranno messi tutti i modelli o scripts creati o importati, e la cartella *Packages*, contenente vari script e componenti aggiuntivi utili per lo sviluppo.

Ci sono tuttavia molti più pannelli che è possibile mostrare o nascondere, come la "*console*" che è utile in fasi di debugging o l'*assets store* per scaricare gratuitamente o a pagamento oggetti creati da altri utenti, ma solitamente questi 4 pannelli sono i più utilizzati per qualsiasi progetto e vengono di conseguenza mostrati di default alla creazione di ogni nuovo progetto a meno che non si siano modificate specificatamente le impostazioni di Unity.

2.3 La mission di Unity: user-friendliness, supporto e documentazione

Unity si pone come primo obiettivo quello di non spaventare i nuovi utilizzatori e facilitare lo sviluppo di qualsiasi idea fornendo strumenti di supporto all'utente che gli permettano di risolvere il più in fretta possibile i problemi che riscontra, o direttamente fornendogli delle risposte prefabbricate da aggiungere semplicemente al progetto per ottimizzare i tempi. Per raggiungere tale obiettivo Unity quindi adotta molteplici soluzioni.

In primo luogo fin da quando ci si trova sul sito ufficiale del framework è possibile accedere a una pagina specifica per imparare ad usare tutti gli strumenti che Unity offre, con video lezioni e tutorial, dai più semplici ai più specifici. É possibile accedere a queste lezioni come già visto anche tramite UnityHub dalla sezione "*Learn*".

Un altro grande supporto che Unity predispone per i suoi utilizzatori è l'*Assets Store*. Si tratta di un sito da cui è possibile scaricare materiale utile creato da altri utenti, in molti casi direttamente dal team di Unity, e reso disponibile gratuitamente a scopo educativo, per semplificare lo sviluppo di progetti di chi non è interessato (o non ha il tempo necessario), ad acquisire le competenze per la creazione di modelli personali, utilizzare e scrivere script, add-ons e così via; o semplicemente perché il materiale di default è sufficiente allo scopo dei suoi progetti. L'asset store permette quindi di scaricare:

- Modelli 3D, per evitare di doverli creare manualmente e custom in caso non si abbiano le competenze per farlo o si voglia risparmiare tempo
- Modelli 2D, in particolare sprites per la creazione di giochi bidimensionali, oppure textures da applicare sui modelli, o anche grafiche in generale per la creazione di menù e interfaccia utente
- Tools e Add-ons, ovvero strumenti utili allo sviluppatore per creare più rapidamente e facilmente il suo applicativo, come strumenti per la creazione di un terreno procedurale e realistico
- Audio e VFX, che sono fondamentali per l'immersione dell'utilizzatore dell'applicazione, ma che sono molto lunghi o complicati da creare
- Templates, utili per partire a creare l'applicativo desiderato con una base specifica alla tipologia del programma che si vuole creare già implementata e testata, che quindi permettono di non dover creare tutto da capo ma semplicemente andare ad aggiungere le funzionalità specifiche che caratterizzano l'idea dello sviluppatore

- Essentials, che è la categoria in cui vengono raccolti vari pacchetti di modelli, strumenti, add-ons e tutto quello elencato precedentemente, ritenuti basilari per la creazione della maggior parte dei progetti

L'ultimo, ma di certo non per importanza, ausilio che Unity fornisce ai suoi utilizzatori, è una documentazione molto ampia e ben strutturata di tutte le funzionalità che implementa. Si divide nello specifico in documentazione sull'editor di Unity, che mostra tutto quello che è possibile realizzare tramite l'ambiente di sviluppo illustrandone tutte le interfacce e le funzionalità che propone; in documentazione sullo scripting, che mostra tutti i dettagli tecnici che Unity implementa di default in ogni progetto tramite la sua libreria, ossia le classi e i loro metodi, che è sempre bene consultare e capire a fondo quando si vuole creare un qualunque progetto.

3 Unity per il game-development

Nonostante Unity sia un framework per lo sviluppo di applicazioni real-time in generale, permettendo quindi di creare molti tipi di applicativi diversi, nasce ed è concepito principalmente per lo sviluppo di videogiochi, e, al giorno d'oggi, è uno dei più famosi ed utilizzati per farlo. E' particolarmente indicato per lo sviluppo di giochi in ambiente mobile, in quanto grazie ai suoi strumenti di base è possibile creare applicazioni computazionalmente poco onerose (adatte quindi a una potenza di calcolo non elevata quanto quella di un computer tradizionale), ma con una qualità elevata, dove ciò che fa davvero la differenza è l'originalità dell'idea dello sviluppatore più che le sue competenze tecniche. In questa sezione andrò quindi a illustrare i fondamentali che Unity offre per lo sviluppo di applicativi real-time, quindi anche videogiochi.

3.1 Nozioni fondamentali per la creazione di videogiochi

Partendo dalle basi, Unity definisce una differenza sostanziale nel sistema di riferimento cartesiano all'interno di ogni scena tra *world space* (ovvero il riferimento di coordinate cartesiane assoluto della scena e dell'ambiente di gioco) e l'*object space* (che è invece un riferimento cartesiano relativo a un certo oggetto, ottenuto tramite relazioni di parentela): prendiamo come esempio quello di voler creare un personaggio giocante, che abbia quindi un modello 3D che si muove all'interno della scena tramite input di tasti, con ad esso connessa una videocamera che permetta al giocatore di vedere cosa lo

circonda. Poniamo ora che il modello 3D venga messo in posizione (30, 40, 0), che sono coordinate del *world space*, quando la videocamera viene definita come figlia di tale modello, le sue coordinate saranno relative alla posizione di quest'ultimo, il quale diventa quindi il punto d'origine delle coordinate della videocamera. Se la videocamera era quindi in posizione (30, 50, 0) all'interno del *world space*, sarà invece dopo la definizione della relazione di parentela in posizione (0, 10, 0).

Le coordinate all'interno di entrambi questi riferimenti sono immagazzinate a livello implementativo all'interno di oggetti *Vector3*, per spazi tridimensionali, o *Vector2* nel caso invece di ambienti 2D. Entrambe queste due classi contengono varie proprietà statiche, proprietà semplici, metodi pubblici e statici, e operatori, che permettono di compiere varie operazioni di trasformazione degli oggetti all'interno dell'ambiente: in particolare vengono quindi usati per modificare la posizione o la direzione/rotazione di un oggetto in scena, ma permettono anche di svolgere operazioni vettoriali.

Ogni scena, ovvero ogni ambiente con cui e in cui è possibile interagire, definisce un *world space* a sé stante. In ogni progetto è possibile avere più scene, il che è molto utile in caso ad esempio si voglia realizzare un gioco con vari livelli: è possibile quindi dedicare una scena per ogni livello, e poi passare da una scena all'altra "attivando" la videocamera che si trova nella scena desiderata, oppure tramite il metodo *Application.LoadLevel()* che definisce la scena attiva. Varie scene sono anche utili in caso si vogliano avere delle *cut-scenes* all'interno del proprio gioco, quindi dedicare una o più scene per realizzare questi brevi filmati.

Definito lo spazio di lavoro di Unity, il pilastro su cui si basa la creazione di videogiochi sono gli *Assets*, che sono qualunque oggetto che venga creato all'interno dell'ambiente o importato dall'esterno: si può notare infatti come alla creazione di un nuovo progetto sia sempre presente nel pannello *Project* una cartella chiamata appunto *Assets* dove viene messo automaticamente tutto quello che vogliamo faccia parte del nostro progetto. Un asset quindi può essere un modello 3D, o una sua animazione, o un file audio, oppure anche un file di script in C#: in generale un asset quindi è un qualsiasi "oggetto" che si può utilizzare all'interno del proprio progetto.

3.1.1 I Gameobject e i Components

Gameobject e relativi componenents sono assets. Ogni asset è trascinabile all'interno della scena che si sta costruendo, e dal momento che viene messo in scena diventa un *GameObject*, ovvero una particolare istanza di un asset all'interno della scena. Quando viene creato un *GameObject*, questo all'interno della scena è univoco, per cui non è possibile avere due *GameObject*

con lo stesso nome, in quanto ognuno di questi è considerato da Unity come una entità indipendente dalle altre. In realtà ogni *GameObject* non è mai completamente indipendente in quanto spesso i suoi comportamenti o azioni dipendono da quelli di altri oggetti all'interno della scena, o sono figli di altri *GameObject*, in quanto Unity permette di definire relazioni di parentela tra oggetti. Un esempio eclatante di questa dipendenza di parentela può essere quello della creazione del personaggio del giocatore, il quale deve avere un modello 3D all'interno della scena che permetta di farlo interagire con gli altri oggetti, ma deve avere anche una videocamera che gli permetta di vedere ciò che lo circonda e che deve seguire il modello del personaggio in base a dove si muove oltre che ruotare in base all'input: per fare questo quindi solitamente si introduce nella scena il modello 3D del personaggio che sarà il padre della videocamera, così che la videocamera sia legata fisicamente a tale modello.

Ogni *GameObject* all'interno della scena è composto e caratterizzato dai suoi *Components*, che definiscono quindi i suoi comportamenti e come deve interagire con l'ambiente che lo circonda. Ogni nuovo oggetto viene creato con 3 *Components* di default:

- *Transform*: è il componente di base di ogni oggetto che esprime la sua posizione e rotazione all'interno del *world space* o del *object space*, e la sua scala, ovvero la sua dimensione rispetto alla dimensione originale del *Asset*. Questo Component ha due tipi di posizioni: *position* e *localPosition* che, quando settate, cambiano la posizione di un oggetto in modi diversi. Il primo definisce la posizione dell'oggetto in termini di coordinate del world-space rispetto all'origine del mondo (coordinate assolute), mentre il secondo esprime la sua posizione in coordinate del object-space (coordinate relative). Poichè l'Inspector del Component Transform visualizza una sola posizione, questa sarà assoluta (position, relativa a *world space*) se l'oggetto a cui il Component è associato è la radice della gerarchia, mentre sarà relativa (localPosition, relativa al *object space*) nel caso in cui l'oggetto a cui è attaccato il Component sia figlio di un altro oggetto.

- *Mesh Filter*: ovvero la mesh dell'oggetto, quindi una rete fatta di triangoli che definiscono la forma dell'oggetto, e lavora poi con il *Mesh Renderer* per appunto rendere visibile tale oggetto all'interno della scena

- *Mesh Renderer*: è il componente che si occupa di disegnare la forma dell'oggetto, quindi renderlo visibile e definire anche come la mesh deve rispondere ai raggi di luce e i materiali di cui è composta la mesh per mostrare un certo colore o texture.

Ci sono però molti *Components* che si possono attribuire ad un oggetto, come ad esempio tutti i vari tipi di *Colliders*, o particolari effetti sonori o visivi, o semplicemente gli script di codice che vengono scritti dallo sviluppatore.

Bisogna chiarire che c'è una differenza importante a livello di codice tra utilizzare il nome di un *Component* "con la lettera maiuscola" e usarlo invece tutto minuscolo. Prendiamo per esempio il caso del *Transform Component*: se viene "invocato" all'interno del codice come "Transform" viene utilizzata la classe che il *Component* definisce, e serve quindi per invocare metodi o accedere a variabili statiche. Nel caso invece venga utilizzato nella forma "transform" si fa riferimento al *Transform Component* specifico dell'oggetto a cui viene attaccato lo script su cui si sta scrivendo il codice, quindi la specifica istanza della classe *Transform* attaccata all'oggetto; non è necessario quindi avere all'interno del codice, la dichiarazione di una variabile di tipo *Transform* per poter accedere a quella specifica dell'oggetto.

3.1.2 Prefabs e differenza con i GameObjects

Non è pensabile tuttavia di utilizzare i *GameObjects* per tutti quegli oggetti che all'interno della scena devono essere creati e distrutti molte volte anche in pochissimo tempo, come ad esempio i proiettili che vengono sparati dal giocatore, o dei nemici che il giocatore deve sconfiggere per proseguire nel livello: per fare questo vengono usati quindi i *Prefabs*. Nei due esempi appena fatti possiamo notare come siano entrambi oggetti che non hanno bisogno di essere modificati ogni volta che vengono creati, ma anzi siano sempre uguali, e inoltre possono dovercene essere anche molteplici all'interno della stessa scena: questa è la regola di base per decidere se per un certo oggetto si debba usare un*Prefab* o un *GameObject*. Il motivo alla base di questa regola è che i *GameObject* possono essere creati solo tramite l'operatore new prima dell'inizio della scena, mentre invece i *Prefab* sono oggetti che possono essere creati a runtime tramite la funzione *instantiate()*, perciò rispettivamente i primi sono quegli oggetti che una volta messi all'interno dell'ambiente di gioco devono rimanerci per sempre, quindi non essere mai distrutti e variare solo in minima parte in base alle interazioni del giocatore, mentre i secondi sono oggetti che vengono istanziati all'interno della scena e una volta che hanno generato un evento o superato un certo tempo, devono essere distrutti: i proiettili ad esempio vengono distrutti quando colpiscono un *Collider* e generano quindi un evento di tipo *Collision*, oppure quando dopo un certo timeout non si scontrano con niente e vengono quindi semplicemente fatti sparire. La funzione *instantiate()* prende 3 parametri: il primo deve indicare quale *Prefab* si vuole istanziare, il secondo la posizione di dove lo si vuole creare, e il terzo la rotazione che l'oggetto deve avere. La differenza

sostanziale tra *new e instantiate()* è che il primo crea un *GameObject* vuoto, che comprende solo *Components* di default, mentre *instantiate()* clona un oggetto così come è stato definito comprendente di tutti i suoi *Components* e caratteristiche già definite dallo sviluppatore, ed è per questo che un oggetto deve essere un *Prefab* se una volta definito con tutte le sue caratteristiche non deve essere più modificato, ma deve sempre essere così come è stato definito ogni volta che viene creato.

3.1.3 Colliders e collision detection

Ogni videogioco all'interno prevede che ci siano delle interazioni tra vari oggetti che generino degli eventi a cui altri oggetti rispondono di conseguenza se interpellati, seguendo quindi la logica degli *Action-Listeners*. I *Colliders*, come suggerisce il nome, sono dei *Components* che servono per gestire le collisioni tra i vari elementi del gioco, e nel caso generino degli eventi a cui gli attori interessati possano rispondere: sono fondamentali ad esempio quando il giocatore viene colpito dal nemico, per cui ci si aspetta che la sua salute diminuisca, oppure quando il giocatore si trova sopra a una pedana che deve far scattare una trappola, o molto più semplicemente per evitare che il giocatore non attraversi i muri o il pavimento. Questi 3 esempi appena citati sono importanti in quanto esprimono i 3 principali utilizzi dei *Colliders*, ovvero:

- *Collider semplice*, ovvero una rete invisibile, quindi che non viene renderizzata, messa attorno a un oggetto per definire il suo perimetro fisico che non può essere attraversato, definendo quindi anche la proprietà fisica dell'oggetto che circonda. Questi tipi di collider possono avere forme semplici, come sfere, cubi, capsule, e così via, o anche mappare la mesh dell'oggetto su cui vengono applicati: in generale si vuole usare un *Collider* semplice quando la precisione per determinare la collisione non è importante e può essere trascurata a vantaggio di una maggiore velocità di calcolo, come ad esempio nel caso di muri e pavimenti in cui è sufficiente definire dei *Collider* cubici, mentre si usa un *Mesh Collider* quando è fondamentale invece la precisione nel determinare dove è avvenuta una collisione, come nel caso di un personaggio a cui passa un proiettile vicino alla testa e la collisione non deve essere registrata.

- *Trigger collider*, che sono una variante dei collider precedenti, ma a differenza di quelli, i *Trigger* possono essere attraversati, e perciò non definiscono un perimetro fisico. Sono particolarmente utili in caso si voglia definire un'area in cui se qualcosa di entra o esce, come il giocatore, venga generato un evento con le conseguenze relative: un semplice

esempio può essere quello di una porta automatica per cui si vuole che si apra quando il giocatore si trova a qualche passo da essa: si definisce quindi un collider più grande dell'effettiva mesh della porta che è l'area in cui, in caso un altro collider entri, faccia scattare l'apertura/chiusura della porta tramite la chiamata del metodo per l'apertura/chiusura

- *Ray casting*, che è invece il metodo utilizzato per prevedere una collisione anche a lunga distanza: l'esempio più comune è quello dei proiettili, per cui per fattori di natura di rendering e di eterogeneità prestazionale dei calcolatori, non si può creare un collider attorno a ogni proiettile, farlo viaggiare ad alta velocità e cercare di catturare la collisione in caso avvenga, É molto più funzionale generare un raggio che parta da dove il proiettile viene generato, e in caso il proiettile venga sparato nel momento opportuno in cui il raggio interseca un altro collider a una certa distanza a gittata, venga generato l'evento di collisione, nonostante questa non sia effettivamente avvenuta: il raggio che viene creato inoltre permette di ottenere anche molte altre informazioni utili, come la distanza tra punto di origine e di interruzione del raggio, o il punto di impatto del raggio. Un esempio più semplice di quello dei proiettili può essere anche quello di un personaggio che deve schiacciare un pulsante, per cui è possibile interagire con esso solo se lo sta direttamente fissando, per cui si fa partire il raggio dagli occhi del giocatore, e solo quando questo interseca il collider del pulsante, viene resa disponibile l'interazione. Per questo tipo di *collision detection* vengono utilizzate variabili di tipo *RaycastHit* per immagazzinare tutte le informazioni del raggio, e il metodo *Physics.Raycast()* per generare il raggio passando determinati parametri. Il metodo ha varie versioni con parametri differenti che è possibile passargli, ma in quella più comune e utilizzata i parametri sono:
  - L'origine, ovvero un un *Vector3*, che è sostanzialmente un set di coordinate del *world space*, che possono essere ottenute anche accedendo al componente *transform* di un oggetto, per indicare l'origine appunto del raggio
  - La direzione, che serve quindi a indicare la direzione del raggio, anche questo di tipo *Vector3*
  - Una struttura dati di tipo *RaycastHit* che serve per immagazzinare tutte le varie informazioni utili relative a cosa il raggio ha colpito
  - Una distanza massima del raggio, espressa tramite un float

É possibile inoltre indicare altri due parametri aggiuntivi che sono un intero che rappresenta una maschera per ignorare selettivamente la collisione con determinati colliders, e una struttura dati di tipo *queryTriggerInteraction* che specifica se tale query dovrebbe interagire anche i triggers.

Nei videogiochi, a livello implementativo, la *Collision detection* è meglio gestirla non andando a definire all'interno dello script del giocatore tutti i casi possibili di interazione con tutti gli oggetti, ma piuttosto invece quindi organizzarla in tutti i piccoli script che gestiscono i singoli oggetti, quindi definire per ognuno di essi il suo comportamento quando il collider del giocatore, o anche altri collider, entrano in contatto con il proprio. Un'eccezione a questa regola tuttavia è però nel caso di *Collision detection* semplice che comprendano il giocatore, per cui è necessario utilizzare la funzione *OnControllerColliderHit()* la quale registra le collisioni tra il *Character controller* e gli altri oggetti, e perciò necessita di essere chiamata in uno script associato ad un oggetto con il *Character Controller Component*, poiché tale funzione prende come argomento un oggetto di tipo *ControllerCollider*, che è lo specifico collider associato al componente *Character Controller*. Questo non è quindi necessario invece nel caso si debbano usare dei trigger o il raycasting per le collisioni, per cui le funzioni di gestione delle collisioni possono essere scritte in appositi script dedicati o anche nello stesso script che definisce l'oggetto.

3.2 Fisica e sua implementazione

La fisica è fondamentale all'interno di ogni videogioco, poiché permette di ottenere comportamenti realistici e far immergere completamente il giocatore all'interno dell'ambiente. In Unity i motori fisici sono 2, uno per i giochi 3D che è un'estensione del motore *Nvidia PhysicX*, e uno per i giochi 2D che è invece una estensione del motore *Box2D engine*: quale di questi viene usato non è definito a priori o in base a una qualche scelta da fare nella fase di creazione del progetto, ma dipende esclusivamente dai tipi di componenti che si assegnano agli oggetti. Se a un modello viene assegnato il componente *RigidBody* questo verrà trattato con il motore 3D, mentre se gli viene assegnato il componente *RigidBody 2D* sarà trattato con l'altro da Unity.

Il fondamentale della fisica in Unity è il componente *RigidBody*, che quando viene applicato a un oggetto permette al motore fisico di prenderlo in considerazione per i suoi calcoli, attivandolo quindi dal punto di vista fisico. Il componente presenta diversi parametri modificabili:

- La massa dell'oggetto, che quindi influisce sulla velocità di caduta e sulla forza necessaria per spostarlo.

- Il *Drag* che gestisce l'attrito con l'aria, ed è di default a 0: in questa condizione quindi se l'oggetto subisce una forza, continuerà a muoversi all'infinito nella direzione di essa, come nel vuoto.

- Il *Angular Drag* che è sempre un attrito, ma applicato alla rotazione dell'oggetto, perciò se lasciato a 0 come di dafault, l'oggetto se inizierà a ruotare non smetterà mai.

- La checkbox *UseGravity* permette di decidere se l'oggetto deve essere o meno soggetto alla forza di gravità, il che significa che l'oggetto cadrà finché il suo collider non entrerà in contatto con un altro collider che lo fermi. Se questa opzione non viene selezionata l'oggetto sarà come nello spazio, quindi fluttuerà e sarà comunque completamente interagibile.

- La checkbox *IsKinematic* permette di definire se l'oggetto può essere o meno influenzato da forze esterne, quindi se selezionata, renderà l'oggetto fermo sul posto in cui è stato messo e non verrà spostato in caso ad esempio il giocatore lo urti con il proprio collider o venga colpito da un proiettile: è comunque possibile spostare tale oggetto tramite codice.

- *Interpolate* è un campo a scelta multipla: l'interpolazione in generale è un metodo usato per rendere i movimenti degli oggetti meno bruschi. In questo campo è possibile specificare quindi se non si vuole nessuna interpolazione, se si vuole una interpolazione di tipo *interpolate* per cui il movimento è reso più dolce in base alla velocità del frame precedente, o se la si vuole di tipo *extrapolate* per cui il movimento è addolcito in base alla velocità stimata del prossimo frame.

- *Collision Detection* è un campo che, a differenza di quanto il nome possa far pensare, non serve a definire il tipo di rilevamento delle collisioni, ma la frequenza con cui l'oggetto deve controllare se il suo collider si è scontrato con altri. Nella maggior parte dei casi la scelta di default "*discrete*" funziona bene, ma in casi in cui l'oggetto si dovesse scontrare con collider che vanno ad alta velocità, questa opzione potrebbe non essere sufficiente in quanto gli oggetti veloci potrebbero attraversarlo e la collisione non verrebbe rilevata: per risolvere questo problema quindi bisognerà modificare questo campo sull'oggetto su cui si scontrano gli altri su "*continuous dynamic*", e quella degli oggetti veloci su "*continuous*".

- *Freeze position/rotation* sono 6 chechboxes, 2 per ogni asse, che se vengono selezionate, permettono di bloccare lo spostamento e/o la rotazione dell'oggetto su quello specifico asse, il che è utile in molte circostanze. Prendiamo come esempio il caso di una leva che il giocatore deve attivare spostandola fisicamente con il suo collider e non interagendoci tramite input: in questo caso vorremo che la leva si sposti o ruoti solo su uno specifico asse e perciò si usano queste checkboxes per definire questi vincoli. Questi campi sono per lo più utilizzati per i giochi bidimensionali piuttosto che per quelli 3D, e resta sempre possibile comunque modificare posizione e rotazione dell'oggetto tramite codice, anche rispetto agli assi che sono bloccati da questi campi.

Tutte queste considerazioni fatte su *RigidBody* sono però inutili se l'oggetto non presenta attaccato a sè anche un collider, in quanto questi sono i componenti che registrano le collisioni e le trasmettono al Component *RigidBody* dell'oggetto a cui sono attaccati. Se ad esempio creiamo una sfera sospesa in aria, con sotto un piano, mettiamo un componente RigidBody a entrambi questi modelli e in più rendiamo Kinematic il piano e attiviamo la gravità sulla sfera, quest'ultima attraverserà il piano durante la caduta, in quanto non è stato assegnato nessun collider né alla sfera né al piano.

Una volta che si è assegnato tale componente a un GameObject o Prefab, se lo si vuole spostare o ruotare è consigliato utilizzare i metodi e le proprietà del componente *RigidBody*, invece che continuare a usare il componente Transform. Uno dei principali metodi usati per fare queste operazioni è il *AddForce()*, che viene chiamato quindi su un oggetto di tipo RigidBody, e accetta come parametro un Vector3 che indichi quindi la direzione e verso della forza, in termini di coordinate del world space. Un altro metodo importante, complementare al precedente, è *addTorque()*, che applica una forza di rotazione sull'oggetto sempre passando come parametro un vettore tridimensionale. Questi solo solo alcuni esempi tra molti metodi pubblici che la classe offre, insieme a tante proprietà, alcune delle quali abbiamo già visto, come "*isKinematic*" o "*useGravity*".

3.3 Scripting e design patterns

Per quanto i *Components* forniti da Unity siano di grande aiuto per strutturare un gioco, ma più in generale un qualunque progetto, sono molto generici e fatti per risolvere delle categorie di problemi, e implementano funzionalità che sono o da modificare in modo che siano specifici per il caso specifico, oppure integrarle con script custom. Ogni script è quindi un asset, e in particolare viene considerato da Unity come un *Component*, che quindi può

essere associato a un qualunque *GameObject o Prefab* per definirne dei comportamenti e caratteristiche specifiche: è quindi importante e utile di norma scrivere codice che sia comunque il più generico possibile in modo che possa essere applicato a più oggetti che rientrano nella stessa categoria, ovvero con lo stesso *Tag*, o che devono comunque comportarsi in modi simili.

3.3.1 La classe capostipite MonoBehavoiur

Si possono notare, aprendo uno script appena creato, 3 elementi importanti:

- *MonoBehaviour*: ogni script è una classe di oggetti, e ogni nuova classe in Unity viene fatta derivare di dafault da questa, la quale implementa molte funzionalità basilari. La classe definisce al suo interno 2 proprietà, alcuni metodi pubblici e statici e molti possibili messaggi utili per gestire vari casi di "basso livello", di cui 2 di questi in particolare sono Start() e Update() che vengono brevemente spiegati qui sotto: i messaggi possono servire sia per svolgere operazioni su tutti gli oggetti presenti in scena, ma anche su singoli *GameObjects* quando si verificano certe condizioni, come ad esempio *OnCollisionEnter()* che viene eseguito quando l'oggetto a cui è attaccato lo script entra in contatto con un altro oggetto in particolare, oppure *OnControllerColliderHit()* che abbiamo appena visto per la collision detection semplice. I metodi pubblici sono anch'essi molto utili e largamente utilizzati, e ne spiegherò alcuni di seguito in questo paragrafo.

- *Start function*, è chiamata una sola volta il primo frame, prima di qualunque altro messaggio di Update, quindi appena la scena viene avviata, ed è utilizzata nella maggior parte dei casi a scopo di inizializzazione di variabili e campi dell'oggetto a cui lo script è associato. Questa funzione è simile ad un'altra che è *Awake()*, la quale anch'essa viene chiamata solo una volta per tutta la durata dell'esecuzione, a inizio frame, però viene chiamata prima ancora di start su tutti gli oggetti: questo è utile in caso si abbia un oggetto A la cui inizializzazione dipende da un oggetto B. Entrambe le funzioni non vengono chiamate se il *GameObject* a cui sono "attaccate" è disattivo in scena, ma un'altra differenza sostanziale tra le due funzioni è che se il *Gameobject* è attivo in scena, e lo script associato contiene entrambe le funzioni, Sart() viene eseguita solo solo se anche lo script è attivo, mentre *Awake()* anche se non lo è.

- *Update function*, è invece la funzione che viene chiamata per ogni frame in cui la scena è attiva, ed è quindi essenziale per controllare lo stato

delle varie parti dell'oggetto o le sue interazioni con gli altri. Viene principalmente usata questa funzione, in quanto chiamata ogni frame prima della renderizzazione, per muovere gli oggetti in una nuova posizione, dando quindi l'illusione del movimento al giocatore, e per controllare gli input. Questa funzione è inoltre indipendente dal Frame-rate, perciò indipendente dalle caratteristiche hardware del calcolatore su cui il gioco viene eseguito, ed essendo che ogni istruzione al suo interno vien eseguita prima di passare al frame successivo, non è possibile neanche creare al suo interno degli effetti pensati per durare più di un frame.

In generale i videogiochi e la loro creazione, quindi anche con Unity, si basano sull'ottica della creazione e gestione di eventi, quindi quando questi vengono generati gli oggetti devono agire da *event-handlers*. Unity applica questi concetti sempre tramite la classe MonoBehaviour che implementa delle particolari funzioni, dette funzioni evento, che vengono eseguite ogni volta che un determinato evento si verifica, come ad esempio il messaggio OnCollisionEnter() che viene chiamato ogni volta che il collider dell'oggetto a cui è associato lo script, entra in contatto o si interseca con il collider di un 'altro oggetto. É quindi poi necessario definire all'interno del corpo della funzione il *handler*, quindi un'altra funzione scritta dal programmatore che definisca come l'oggetto deve comportarsi e reagire a tale evento. Queste particolari funzioni evento non sono tuttavia da inserire all'interno della funzione Update(), in quanto a ogni frame Unity controlla ogni script attivo in scena, e se trova una di queste funzioni predefinite la chiama e quindi controlla se è verificata o meno la condizione specifica della funzione. Queste funzioni sono di default definite private, ma è possibile anche definirle pubbliche per forzarne l'esecuzione anche da altri script anche se una condizione di un evento non si verifica.

Per ogni gioco è inoltre importante definire un modo per rendere tutto Framerate-independent. La funzione Update() definisce quello che si dice un *game-loop*, ovvero un set di istruzioni e condizioni da controllare ciclicamente finchè il gioco è in esecuzione, e che determinano cosa deve accadere in scena per ogni frame. In ogni gioco però il Framerate non è costante, e quindi non spremo mai quanti frame ci sono in un secondo, in quanto per un secondo potrebbero essere 30 e per quello dopo 50, e questo causerebbe delle discrepanze in quello che viene renderizzato in scena rispetto a quello che il programmatore aveva pianificato. Facciamo l'esempio di un oggetto che deve muoversi a una velocità costante: non è possibile assegnargli tale velocità di spostamento effettivamente costante, poiché a causa di questa variabilità dei frame al secondo, per ogni secondo l'oggetto si potrebbe muovere di più o di meno. Per risolvere questo problema quindi, per tutte le velocità o in

generale grandezze che dipendono dal tempo e devono quindi essere indipendenti dal Framrate, che sono da identificare, devono essere moltiplicate per il tempo *Time.deltaTime* che equivale al tempo trascorso dal frame precedente: una grandezza Framrate-dependent moltiplicata quindi per questo valore ci restituirà sempre lo stesso risultato indipendentemente dal numero di frame al secondo.

Un'altra funzione della classe *Monobehaviour* molto utile, è *FixedUpdate*, che viene chiamata a intervalli di tempo regolari, di default ogni 0.2 secondi, e quindi è indipendente dal Framerate. Questa funzione viene chiamata subito prima dei calcoli relativi al motore fisico, e quindi risulta molto utile per tutte le operazioni che riguardano la fisica, come applicare forze su un RigidBody o lavorare sui collider, mentre essendo che viene chiamato molto meno spesso rispetto a quanto Unity renderizza la vista, non è utilizzata per operazioni che invece riguardano la grafica. L'intervallo di tempo ogni quanto la funzione viene chiamata può essere modificato, quindi abbassato per dare più precisione alla simulazione fisica con il contro però di un maggiore carico di lavoro per il processore che su hardware meno potente può risultare anche in calcoli errati: tuttavia ci sono casi in cui la precisione non è essenziale e quindi si può aumentare tale valore per sforzare di meno il processore.

3.3.2 Altre nozioni fondamentali

Fino ad ora abbiamo parlato di Components come entità indipendenti separati gli uni dagli altri, ma è necessario in molti casi che i componenti dello stesso oggetto comunichino tra loro, in particolare accedendo a variabili e metodi di un altro Component e spesso anche per ogni frame, ed è necessario quindi capire come farli comunicare tra loro.

La classe *Monobehaviour* fornisce due metodi pubblici molto utili, ovvero *SendMessage() e BroadcastMessage()* che permettono di eseguire facilmente funzioni chiamandole per nome, su tutti i componenti attaccati a un oggetto: solitamente per chiamare un metodo di una classe è necessario una reference locale di quella classe, quindi un suo oggetto, per accedere ai suoi metodi e chiamarli o per accedere alle sue variabili. Queste due funzioni tuttavia permettono di chiamare funzioni usando delle stringhe che rappresentano il nome di queste ultime, e questo aiuta molto per rendere il codice più leggibile e semplice anche se a costo di efficienza. I metodi quindi accettano come parametro una stringa, che identifica il nome della funzione da chiamare, e delle opzioni che specificano cosa deve fare Unity in caso la funzione non venga trovata su un componente. Prendiamo ad esempio quindi un oggetto con attaccati a se 2 o più script: chiamando *SendMessage()* su uno di questi, che può anche non aver definito al suo interno la funzione passata come para-

metro, questa viene chiamata su tutti gli script, o in generale i Component, del GameObject che hanno definito al loro interno tale funzione. Il metodo *BroadcastMessage()* fa un passo avanti, quindi incorpora lo stesso comportamento di *SendMessage()* ma chiama la funzione anche su tutti i componenti di tutti gli oggetti figli della gerarchia del GameObject: non c'è quindi differenza tra i due metodi in caso di GameObject che non presentano oggetti figli. Bisogna prestare attenzione però che *BroadcastMessage()* funziona solo "verso il basso", quindi nel caso venga chiamato in uno script appartenente a un oggetto figlio di un altro, lo trasmette solo ai suoi figli ma non al padre o agli altri figli del padre. I due metodi sono particolarmente utili per facilitare la comunicazione inter-GameObject e inter-Component, quindi far comunicare componenti o oggetti tra loro in caso sia necessario, per sincronizzare i comportamenti e riciclare la funzionalità del codice senza doverlo copiare. I due metodi tuttavia si basano su una proprietà del C# detta *reflection*, per cui invocando una funzione usando una stringa, l'applicazione deve guardare al suo interno durante il run-time, come se guardasse a un riflesso di sé, e cercare nel suo codice la funzione che deve essere eseguita; questo perciò è un processo computazionalmente complesso rispetto a eseguire una funzione nel modo tradizionale, e perciò bisogna cercare di usare il meno possibile questi due metodi, soprattutto all'interno di eventi *Update()* o altri eventi basati sui frame.

Detto questo perciò abbiamo bisogno di altri modi per poter far comunicare tra loro i Components, soprattutto perché sono metodi che possiamo chiamare ovunque all'interno dello script per chiamare funzioni tramite il loro nome su tutti gli altri componenti connessi allo stesso oggetto, e non prendono in considerazione il tipo del Component su cui chiamano tali funzioni. Questi due metodi hanno però due problemi:

- Ragionano con un'ottica del "tutto o niente", per cui o la funzione viene chiamata su tutti i Component dello stesso oggetto in cui è presente o su nessuno, e quindi non si può scegliere selettivamente su quali Components la funzione deve essere chiamata perché viene chiamata su tutti quelli che la definiscono

- Entrambi si basano sulla *reflection* che come già detto, limita le prestazioni

Nel caso quindi si voglia chiamare una funzione su un singolo componente di cui si conosce il tipo, ci viene in soccorso il metodo *GetComponent()* della classe GameObject: questo metodo prende come parametro il tipo di un componente e ritorna il primo match all'interno dell'oggetto su cui viene chiamato. Una volta ottenuta la reference al Component è quindi possibile

accedere ai suoi metodi e variabili come un normale oggetto. esiste anche un altro metodo simile, che è *GetComponents()*, che permette di ottenere più componenti in una sola volta, ed è quindi utile quando si vogliono ottenere più componenti e inserirli in una lista, oppure avere una lista con tutti i componenti, o una lista di tutti i componenti di un certo tipo. Con questo metodo, chiamandolo una sola volta all'interno della funzione Start() o Awake(), si ottiene lo stesso risultato che si otterrebbe chiamando il *GetComponent()* all'interno della funzione Update(). Unity fornisce inoltre altre due funzioni molto utili per la comunicazione inter-oggetto, quindi non solo tra componenti attaccati allo stesso GameObject, che sono i metodi *GetComponentsInChildren()*, che ritorna una lista dei componenti di tutti i figli, e *GetComponentsInParent()* che ritorna la lista di tutti i componenti del padre.

A questo punto abbiamo un ottimo strumento per la comunicazione inter-componenti che funziona meglio di SendMessage() e BroadcastMessage(), ma ci sono situazioni in cui dato un GameObject serve chiamare un metodo su solo un Component, invece che su tutti come con SendMessage(), senza sapere a priori il tipo del Component con tale metodo. Uno scenario in cui questo sarebbe utile è quello per la creazione di comportamenti riutilizzabili. Il problema quindi è che senza sapere a priori il tipo del Component su cui deve essere chiamata la funzione, con SendMessage() non possiamo chiamare selettivamente solo il metodo di uno specifico Component senza chiamarlo anche su tutti gli altri che lo presentano. Per fare questo si ha a disposizione il metodo *MonoBehaviour.Invoke()*, che accetta due parametri: una stringa che indica il nome del metodo da invocare, e un tempo espresso in float che definisce il lasso di tempo che deve passare prima che il metodo desiderato venga eseguito. Questo metodo risolve il problema in quanto all'interno dello script si potrà dichiarare un oggetto pubblico di tipo MonoBehaviour, e poi dal Inspector fare drag&drop di un qualunque Component all'interno del campo, e poi chiamare il metodo *Invoke()* su tale oggetto passando come parametro il nome di una sua funzione che si deve chiamare.

Abbiamo visto quindi ora i possibili metodi che ci permettono di applicare una comunicazione inter-component all'interno dello stesso GameObject, o una primitiva comunicazione inter-oggetti che funziona però solo tra oggetti legati da una relazione di parentela. In alcune situazioni però è necessario anche dover far comunicare due oggetti completamente distinti e separati, e quindi tramite codice poter trovare un oggetto presente in scena, che non è lo stesso a cui è attaccato lo script su cui si scrive il codice, e che non è legato neanche da relazioni gerarchiche: le funzioni che offre Unity per raggiungere questo obbiettivo sono varie, ma pesanti dal punto di vista dei calcoli, quindi vanno utilizzate solo in caso sia strettamente necessario e in funzioni evento

che vengono eseguite una sola volta, come Start() e Awake(). Queste funzioni statiche sono *GameObject.Find()* e *GameObject.FindObjectWithTag()*, e di queste l'ultima dovrebbe essere sempre preferita rispetto alla prima per performance migliori. La funzione *Find()* prende come parametro una stringa case-sensitive, che rappresenta il nome dell'oggetto da cercare in scena, e ritorna la prima occorrenza che trova di un oggetto con tale nome, cercandolo nel pannello Hierarchy. La funzione fa un confronto di stringhe perciò è un'operazione abbastanza lenta e pesante, e inoltre funziona solo in caso l'oggetto abbia un nome univoco all'interno della scena, il che è una condizione che accade abbastanza raramente: resta comunque un valido strumento in caso gli oggetti siano nominati appropriatamente. La soluzione molto più efficiente in termini di prestazioni invece è la ricerca tramite FindObjectWithTag(), che prende sempre una stringa come parametro, che rappresenta però un Tag, che è unico per ogni oggetto, ma che può essere assegnato anche a più oggetti a formare quindi una collezione: la funzione infatti ritorna sempre un array di tutti i GameObject che fanno match con il Tag passato come parametro. Ci sono casi in cui può fare comodo assegnare più Tag a un singolo oggetto, ma purtroppo Unity non lo permette, perciò per aggirare questo problema la soluzione è di assegnare all'oggetto padre dei GameObject figli vuoti, e assegnare a ognuno di questi un Tag diverso in base alla necessità: bisogna però poi di ricordarsi nel codice di risalire sempre alla reference dell'oggetto padre.

Dopo aver trovato due o più oggetti in scena, si può aver bisogno di confrontare questi oggetti, e in particolare il loro Tag: è possibile fare questo tramite la funzione *GameObject.CompareTag()* che accetta il Tag di un altro oggetto in scena, che quindi deve essere già stato trovato, e ritorna un numero booleano che esprime se i due oggetti hanno lo steso Tag o meno. C'è anche caso che si voglia confrontare due oggetti per capire se sono lo stesso, che è utile ad esempio nel caso di un nemico che deve decidere se affrontare o scappare dal giocatore in base al numero di alleati che ha nelle vicinanze: trovando quindi tutti i nemici tramite il metodo *FindObjectWithTag()*, all'interno della lista viene incluso anche quello che ha fatto la chiamata al metodo per il controllo degli alleati, e perciò va escluso dal conteggio. Si può iterare quindi ogni oggetto della lista restituita e controllare a ogni iterazione se l'ID dell'oggetto correntemente preso in considerazione e l'oggetto stesso in cui si sta eseguendo il ciclo, sono lo stesso tramite il metodo *GetInstanceID()*.

Un altra funzionalità importante da implementare all'interno del proprio gioco è quella di riuscire a determinare se tra due oggetti c'è o meno un ostacolo, il che è molto utile per oggetti che presentano una visuale all'interno dell'ambiente, o più in generale per il *culling* degli oggetti o per le funzionalità del IA. Per culling si intende non renderizzazione di oggetti che

non sono direttamente osservabili dal giocatore, per aumentare la velocità dei calcoli diminuendo il lavoro a carico della CPU e della GPU che altrimenti dovrebbero processare inutilmente degli oggetti all'interno della scena: in Unity di dafault le videocamere attuano un *frustum culling*, ovvero non renderizzano tutti gli oggetti che non sono all'interno del campo visivo della videocamera, ma questo metodo non comprende l'esclusione degli oggetti che sono all'interno del campo, ma nascosti da altri. Un primo metodo che serve a determinare se tra due oggetti ce n'è un altro è il *Physics.LineCast()*, che come suggerisce il nome, genera una linea tra due oggetti e controlla se questa interseca il collider di un altro o meno, prima di arrivare alla destinazione; il metodo accetta due parametri di tipo Vector3 che rappresentano il primo la sorgente della linea, e il secondo la destinazione, e infine una *LayerMask* per specificare quali livelli della scena la linea deve considerare per fare i propri controlli. Il metodo ritorna quindi un booleano che esprime se c'è o meno un ostacolo.

In Unity le scene possono essere viste come universi separati, perciò quando viene inserito un oggetto al loro interno, questo è riferito solo a quella particolare scena in termini di tempo e spazio di essa, e quando questa cambia l'oggetto in questione viene distrutto. Questo è il comportamento che si vuole che la maggior parte degli oggetti seguano, ma ci sono casi particolari in cui l'oggetto deve rimanere "vivo" ed essere quindi immortale per poter essere trasportato da una scena ad un'altra, come nel caso del personaggio del giocatore. Per fare questo si utilizza la funzione *DontDestroyOnLoad()* che però comporta anche alcune conseguenze degne di nota. Questa funzione permette di trasportare quindi tutti i cambiamenti che l'oggetto ha subito e tutti i suoi oggetti figli, in un'altra scena, e per questo motivo solitamente gli oggetti immortali vengono creati leggeri, in particolare vuoti e senza figli, con solo i Components essenziali per poter funzionare, in modo che solo i dati fondamentali sopravvivano, e quelli che non servono più o che possono essere facilmente ricostruiti vengano scartati per rendere tutto più leggero. Questa funzione tuttavia introduce un problema, in quanto quando si passa da una scena dove è presente il nostro oggetto immortale, ad un'altra in cui l'oggetto è già stato in precedenza, Unity crea una nuova versione dell'oggetto, quindi aggiornato, senza però distruggere quello vecchio, creando quindi dei duplicati che non sono nella maggior parte dei casi desiderati: per risolvere questo problema è quindi fondamentale utilizzare un Singleton per la creazione di tale oggetto.

3.3.3 Design pattern principalmente utilizzati

Nel game-development i design pattern sono molto utili, perché rendono il codice più mantenibile e efficiente, semplificando quindi di molto il complesso processo di creazione di un videogioco, o la comprensione del suo codice. Ci sono svariati design pattern utili a diversi scopi, ma qui presento solo quelli che sono più comuni e che attuano delle migliorie ingenti se non essenziali, ovvero: il *Singleton*, lo *Strategy*, l'*Observer*, il *Composite*, il *Model-View-Controller* e il *Template*. Oltre a spiegare brevemente lo scopo e l'implementazione teorica di questi design pattern, mi focalizzerò di più sul loro effettivo utilizzo e i loro benefici nel campo specifico del game-development.

Introduciamo per primo il *Singleton*, in quanto uno dei più semplici da realizzare e perché è stato accennato nel paragrafo precedente. Lo scopo di tale design è assicurare che vi sia sempre una sola istanza di un oggetto di una certa classe, distruggendo tutte quelle che non sono considerate autorevoli. Il design si basa su 3 punti fondamentali:

- La definizione di una variabile privata statica che rappresenta l'istanza unica dell'oggetto
- Un metodo statico getter che ritorni appunto l'istanza, che nel caso specifico di C# viene fatto tramite la definizione di una proprietà statica read-only, quindi con solo il getter.
- Un if-statement all'interno di una funzione, nel caso di Unity Awake() o Start() solitamente, che controlli che l'istanza sia unica, quindi che se la variabile statica non è null distrugga quella che si è cercato di creare, altrimenti assegni a tale variabile statica l'oggetto corrente

Il fatto di avere metodo getter e variabile dell'istanza statici, è fondamentale, poiché è l'unico modo in cui possiamo controllare senza dover avere nessun riferimento a un oggetto della classe, se c'è un solo oggetto di questa o più di uno. Nel game-development in Unity questo design pattern è particolarmente utile, come già accennato, in particolar modo per gli oggetti immortali, che non solo devono essere passati da una scena all'altra, ma per cui è anche necessario averne al massimo uno per scena che non entri in conflitto con altri oggetti della stessa classe ma che contengono informazioni diverse inconsistenti. Un esempio di questi oggetti è il *GameManager*, un oggetto che si vuole unico e immortale per ogni scena, in quanto gestisce tutte le funzionalità di alto livello del gioco, come la messa in pausa o se gli input debbano essere accettati o meno: avere più istanze di questo oggetto in una singola scena non avrebbe senso e porterebbe anche a dei conflitti e possibili

bug del gioco. É importante notare che però non è necessario che tutti gli oggetti Singleton siano anche immortali.

Il design successivo è lo *Strategy*, che è fondamentale per disaccoppiare le interazioni tra i comportamenti e gli input, e la logica di gioco: ogni comportamento è diverso dall'altro, ma ognuno deve restituire lo stesso tipo di valori alla game-logic, e inoltre rimuovere uno di questi comportamenti non dovrebbe risultare nel crash del gioco. Con questo pattern possiamo quindi separare questi due aspetti in modo che sia possibile modificare dei comportamenti dinamicamente, senza dover andare a modificare anche la logica di gioco. Questo pattern sai basa sul separare quello che rimane sempre uguale da quello che invece cambia spesso e che deve quindi essere incapsulato tramite interfacce, in modo che il codice sia aperto all'estensione ma chiuso al cambiamento. Questo design pattern, essendo di tipo comportamentale, si presta bene nella creazione di videogiochi per la definizione di classi di oggetti che sono simili per i loro comportamenti, ma che presentano piccole differenze gli uni dagli altri. Il classico esempio di questo è per la superclasse dei nemici, o ancora più in generale dei personaggi non giocanti di un gioco, in quanto è molto probabile che ci siano diversi tipi di nemici che compiano diversi tipi di attacchi, di difese, e abbiano un loro set specifico di abilità. Semplificando ancora di più ai personaggi non giocanti, in questo caso potremmo avere quindi personaggi ostili, neutrali o amichevoli verso il personaggio, quindi aggiungendo un ulteriore livello di astrazione. Il pattern è particolarmente efficace quindi in quei casi in cui basarsi sulla sola ereditarietà non basta per rendere il codice mantenibile o non risulta funzionale. Prendiamo come esempio la modellazione di una classe Guerriero, e una Arciere, che derivano da una classe padre Personaggio. Entrambi questi personaggi devono poter attaccare, ma i loro attacchi dovranno essere molto differenti in quanto a danni inflitti e portata dell'attacco ad esempio. Se si usasse la sola ereditarietà per implementare queste differenze, implementando quindi uno scheletro dell'attacco all'interno della classe Personaggio, si dovrebbe poi andare a fare l'override dei metodi all'interno delle singole sottoclassi, il che non è un problema finché si hanno 2 classi figlie, ma lo diventa dal momento che si devono implementare molti altri tipi di personaggi o si vuole cambiare radicalmente il metodo generale di attacco. Quello che si fa in questo caso quindi è incapsulare il comportamento di attacco all'interno di una interfaccia, che poi le classi figlie dovranno solo implementare e definire in base alla loro specifica necessità.

Un altro pattern largamente utilizzato è quello *Observer*, che permette di far interagire classi tra loro senza sapere cosa facciano o come siano fatte. Con questo design pattern le classi si iscrivono a una lista tenuta da un oggetto chiamato *publisher*, il quale produce informazioni e le manda solo

agli oggetti che sono iscritti a tale lista, ovvero gli *obeservers*. Questi ultimi possono quindi infine decidere quando iscriversi o disiscriversi a run-time da tale lista in caso le informazioni prodotte dal *publisher* non siano più di loro interesse. Il pattern punta quindi a dare flessibilità al codice, data dallo scarso accoppiamento tra oggetti che interagiscono, rendendo i cambiamenti più semplici. A livello implementativo quindi si ha che il *publisher* sa solo che gli *oberservers* implementano l'omonima interfaccia, e non lo si deve mai modificare per aggiungere *observers*, ed è essenziale sempre definire un metodo di notifica senza il quale questo design pattern non può funzionare, e che deve sempre restare invariato, a differenza di *publisher* e *observers* che invece possono cambiare a loro piacimento. Seguendo questo principio di disaccoppiamento tra classi si ottiene un gioco modulare e aperto all'aggiunta di nuovi comportamenti con una minor probabilità di aggiungere bugs indesiderati. Il *Observer design pattern* si usa quindi appunto quando classi diverse devono comunicare tra loro, ma debbano sapere poco l'una dell'altra, e l'unica cosa che conta è che sappiano che possono comunicare.

Il *Composite design pattern* è usato poichè in ogni gioco solitamente ci sono molteplici viste, quindi non solo la vista principale della videocamera del giocatore dove vengono renderizzati i modelli che può vedere, ma anche molte sotto-viste come tutte quelle del HUD, o nel caso dei giochi mobile anche tutti i pulsanti a schermo. C'è bisogno quindi di poter avere una struttura gerarchica di queste viste, possibilmente ad albero: il *Composite* permette appunto di fare questo, trattando oggetti singoli e composizioni di oggetti allo stesso modo. Si costruisce quindi una struttura ad albero per cui ogni nodo può essere sia un oggetto singolo che una composizione, mentre le foglie rappresentano sempre e solo oggetti singoli. Sfruttando questo pattern si ottiene quindi che ogni scena ha funzioni con lo stesso nome e ha un singolo punto di accesso, quindi la stessa chiamata a funzione può essere in grado di accedere sia alla view principale che a una sub-view.

L'ultimo design pattern che viene largamente utilizzato è il *Model-View-Controller*, che è formato da 3 componenti, ovvero:

- Il *Model* che implementa la logica del gioco
- La *View* che serve per illustrare all'esterno l'output delle azioni svolte
- I *Controller* che sono i componenti a cui l'utente si interfaccia per far svolgere le operazioni all'applicazione.

Questo design pattern si implementa tramite 3 pattern che sono appena stati illustrati, ovvero il *Observer* che implementa il *Model*, il *Composite* che implementa la *View*, e lo *Strategy* che implementa il *Controller*. Con questa

architettura quindi si ha un disaccoppiamento della logica di gioco dalle viste, e si ha che quindi il *Model* può interagire con le altre due parti senza sapere nulla di loro, e si ottiene così un alto disaccoppiamento tra le interazioni di tutte le classi.

Ci sono inoltre altri due design pattern che vengono utilizzati spesso, poiché facili da applicare e rendono il codice sia più mantenibile che intuitivo: sono il *Template* e il *Decorator*, e solitamente vengono utilizzati insieme in quanto le funzionalità di uno sono utili per quelle dell'altro.

Il *Template*, come suggerisce il nome, serve per fare un'astrazione, definendo una classe generica che fa appunto da template per quelle specifiche. La super-classe definisce quindi una categoria di oggetti generici, e va a definire dei metodi che sono comuni a tutti questi, ma lascia quelli che tra i vari oggetti specifici possono differire, astratti, e verranno quindi implementati effettivamente tramite override dala classe dell'oggetto specifico, che penserà a definire le sue caratteristiche particolari. Il *Template* è particolarmente utile nel caso di oggetti che devono eseguire di istruzioni simili, quindi nello stesso ordine ma con alcune di queste istruzioni che differiscono leggermente da oggetto a oggetto: la superclasse implementa quindi un metodo che definisce la scaletta ordinata dei metodi da chiamare, e poi i metodi singoli, che vengono impostati su *final* se sono uguali per tutti gli oggetti appartenenti alla categoria della super-classe, o astratti se invece sono quei metodi per cui in ogni oggetto della categoria differisce da quello degli altri.

Il *Decorator* infine è un pattern che serve per aggiungere nuovi compiti e responsabilità a run-time a un oggetto, senza dover cambiare il codice della classe, rispettando quindi il principio di essere "aperto all'estensione e chiuso alla modifica". Si basa sull'avere un oggetto di base, che si può poi appunto "decorare" durante l'esecuzione, ma è necessario rispettare alcuni vincoli specifici:

- Gli oggetti decoratori devono essere dello stesso super-tipo della classe che decorano
- Un oggetto deve poter avere più decoratori
- Il decoratore aggiunge il suo comportamento prima o dopo aver delegato alla classe che decora, il resto del lavoro
- Gli oggetti devono poter essere decorati in qualunque momento
- Essendo che gli oggetti hanno tutti lo stesso super-tipo, è possibile utilizzare un oggetto decorato al posto dell'originale wrappato

Il pattern quindi serve a implementare una variazione dell'ereditarietà del linguaggio a oggetti, con lo scopo di aggiungere funzionalità. A livello implementativo quindi si ha:

- Una super-classe generica, solitamente astratta, che rappresenta la categoria di oggetti che si vuole decorare con i decoratori
- Un'altra classe generica per i decoratori, anche questa astratta, che deve però derivare, quindi ereditare, dalla classe generica degli oggetti da decorare, e contenere al suo interno un riferimento a un oggetto di tale super-classe
- Diverse classi per ogni tipo di oggetto diverso che appartiene alla super-classe degli oggetti, dove ognuna deve quindi derivare dalla super-classe
- Diversi decoratori applicabili ai vari oggetti, che devono quindi derivare dalla classe generica dei decoratori

Questo design pattern è particolarmente utile quando si hanno giochi con equipaggiamenti customizzabili, per cui il personaggio può voler cambiare a run-time il suo aspetto con particolari estetici, o il vestiario che indossa, o migliorare gli strumenti che usa, e così via.

3.4 Supporto dell'AI

All'interno dei videogiochi è quasi sempre indispensabile avere una intelligenza artificiale per rendere più interessante e avvincente il proprio gioco, permettendo al giocatore di interagire non solo con oggetti statici, ma anche con personaggi non giocanti, detti *NPC* o *Non Playable Character*. Questi agenti devono però essere in grado di comportarsi in certi modi in base al loro ruolo e abilità, e soprattutto devono essere in grado di comportarsi normalmente, ovvero non attraversare muri o avere comportamenti irrealistici.

3.4.1 Come far muovere l'intelligenza artificiale

Il primo ostacolo che si incontra nell'implementazione dell'IA è il suo movimento, per cui deve essere in grado di trovare dei percorsi all'interno dell'ambiente che la facciano muovere per raggiungere il suo scopo, ma che al contempo vengano percepiti come naturali dal giocatore: questo significa non solo che l'intelligenza artificiale deve saper riconoscere gli ostacoli e superarli in un modo che non coinvolga il loro attraversamento, ma deve anche, in versioni sempre più avanzate, saper scegliere diverse velocità di movimento, saltare ostacoli bassi e spazi tra due piattaforme, o abbassarsi in modo da

attraversare buchi nei muri o staccionate troppo alte, ad esempio. Per raggiungere questo obbiettivo Unity permette di creare delle *Navigation Mesh*, ovvero piani non renderizzati, che Unity genera automaticamente su tutte le superfici orizzontali del livello che interpreta come "camminabili", e che vengono classificate come pavimenti: queste mesh contengono quindi i dati necessari poi all'intelligenza artificiale per calcolare e seguire un percorso che eviti gli ostacoli quando richiesto. Dopo averla aggiunta, la *Navigation Mesh* deve essere calcolata, o come si dice "baked", tramite determinati parametri modificabili:

- Il primo è il *Radius*, ovvero il raggio del cerchio con cui gli oggetti che possiedono l'IA vengono approssimati. Serve quindi a indicare lo spazio approssimativo che gli oggetti occupano, e deve essere regolato attentamente. Se troppo largo farà risultare la mesh rotta o fratturata in parti non comunicanti, e si avranno quindi dei passaggi dove teoricamente gli agenti potrebbero passare che invece non sono attraversabili. Se troppo piccolo invece richiederà molto tempo per il calcolo della mesh e inoltre i modelli potrebbero passare per percorsi troppo piccoli per la loro dimensione o fargli compenetrare i muri. Non c'è purtroppo un valore fisso di questo parametro, ma è necessario andare per tentativi e trovare quello più adatto per il proprio progetto.

- Altro parametro importante da modificare è *Height Inaccuracy Percentage* che definisce l'altezza della *Navigation Mesh* rispetto alla mesh vera e propria del pavimento. Settandolo quindi al minimo, la mesh di navigazione sarà quasi coincidente con il pavimento, e l'IA non sembrerà muoversi fluttuando nel vuoto sopra di esso.

É possibile che si voglia avere una *Navigation Mesh* divisa in più parti, definendo quindi delle aree chiuse in cui gli agenti possono muoversi ma da cui non possono uscire, se non tramite dei teletrasporti o dei passaggi particolari attraverso cui devono passare per andare da un'area all'altra. Per ottenere questo è possibile definire dei *off-mesh links*, che sono Component che se attaccati a oggetti permettono di definire particolari passaggi che consentono all'IA di calcolare percorsi validi anche su una mesh separata, definendo quindi dei link tra varie *Navigation Mesh* separate. Una volta attaccato il Component a due oggetti è necessario semplicemente assegnare il Transform dello stesso al parametro Start, e quello dell'altro al parametro End, rendendoli quindi comunicanti. Per aiutare a capire se si è fatto tutto correttamente, all'interno del pannello *Navigation* Unity traccia una linea che mostra gli oggetti che comunicano tra di loro, permettendo quindi di capire dove un agente venga portato se usa uno dei due.

3.4.2 Creare un agente NPC

Tutto ciò che ho mostrato fin'ora risulta inutile però finchè non implementiamo effettivamente una IA da associare a modelli per permettergli di muoversi sulle Navigation Mesh. Per prima cosa quindi ovviamente si dovrà avere a disposizione di una rappresentazione fisica del nostro agente, che sia 2D o 3D, a cui poi dovremo assegnare vari Components che gli permetteranno di muoversi e interagire con l'ambiente e diventare vivo. Quando si svolge questo primo passaggio è importante che in caso il modello sia importato da terze parti, vengano eliminati tutti i possibili Component *Animator*, in quanto è consigliabile costruirne di custom successivamente.

Il passaggio subito successivo è aggiungere al GameObject così creato il *NavMeshAgent Component*, che permette all'oggetto di interagire con la Navigation Mesh e gli permetterà di calcolare percorsi, quando istruito. Anche in questo Component ci sono dei parametri da settare, in particolare *Radius e Height* che devono essere settati sugli stessi valori di quelli della Navigation Mesh. Un altro parametro è la *Stopping Distance* che controlla quanto vicino l'agente può arrivare a una destinazione prima di fermarsi.

É necessario ora aggiungere una fisica al NPC, quindi si deve aggiungere anche un RigidBody Component e renderlo Kinematic in modo da permettergli di entrare in zone di trigger e dandogli modo di generare e ricevere eventi fisici. Se un oggetto è reso "cinematico", Unity non sovrascriverà la sua trasformazione spaziale, e permetterà quindi al component *NavMeshAgent* di occuparsi interamente dello spostamento, rotazione e dimensione dell'agente.

Resta ora solo una cosa da aggiungere prima di implementare la vera e propria intelligenza artificiale, ovvero un BoxCollider Component attorno all'oggetto che serve ad approssimare il suo campo visivo e renderlo un Trigger. Questo servirà all'IA per svolgere i propri calcoli ed eseguire operazioni di risposta quando corpi entreranno al suo interno

3.4.3 Creare la vera e propria intelligenza artificiale

A questo punto non resta quindi che creare un nuovo script che racchiuderà appunto l'intelligenza dell'agente. Prima di iniziare a scrivere codice, è meglio utilizzare un diagramma di automi a stati finiti per determinare i vari stati in cui si vuole che l'agente possa trovarsi, come ad esempio quelli di attacco, fuga o attacco in caso si stia programmando un nemico. Per implementare questa logica si utilizzerà quindi poi un particolare design pattern chiamato appunto *Finite State Machine design pattern*, per cui si programmano i vari stati in cui l'oggetto si può trovare, uno alla volta, e serve poi a controllare i

modi in cui questi sono collegati tra loro, definendo quindi le condizioni per cui l'oggetto passi da uno stato all'altro.

Essendo che l'agente deve però anche avere determinate animazioni per ogni stato, solitamente si utilizza anche un grafo delle animazioni, che controllerà solo quale animazione deve essere riprodotta in base allo stato in cui l'agente si trova. Per creare questo grafo si deve quindi creare un asset detto *Animator Controller*, che definisce tutte le possibili animazioni della mesh, e dovrebbero corrispondere ai possibili stati in cui l'oggetto può trovarsi: tale asset si deve poi aprire all'interno del pannello *Animator*, e una volta aperto aprirà una finestra *Mecanim* in cui si può costruire il grafo in sè. Ogni animazione che verrà utilizzata deve essere settata su *loopable*, così che venga riprodotta ciclicamente se l'oggetto resta nello stesso stato per più tempo, non facendola quindi riprodurre una sola volta. A questo punto si possono aggiungere quindi le varie animazioni alla finestra tramite drag&drop, una per ogni stato, e di norma si aggiunge anche una animazione vuota in più, che sarà quella iniziale dell'oggetto quando si troverà nello stato di default: rappresenta quindi uno stato vuoto in cui l'oggetto si trova all'inizio di ogni scena prima che venga settato un nuovo stato tramite codice. A questo punto si devono definire le connessioni e la logica tra i vari stati, quindi tra le varie animazioni, e per farlo si devono creare tanti *triggers*[1] quanti gli stati, i quali permettono che un certo comportamento sia eseguito una sola volta, come nel caso del cambio di stato, e possono anche essere facilmente referenziati tramite codice. A questo punto è possibile è possibile definire effettivamente i link tra i vari stati in quanto si vuole che l'oggetto passi da uno stato all'altro quando il *trigger* corrispondente al nuovo stato diventa vero: si possono ora creare le connessioni vere e proprie tra i vari stati e la loro direzione. Per attaccare il grafo al modello dell'agente, basterà aggiungergli un *Animator Component* e assegnare il grafo Animator, di default messo nel pannello project, al campo *Controller*.

Finito l'automa a stati finiti che regola le animazioni, si deve passare poi a scrivere il codice che governerà effettivamente i comportamenti e le azioni dell'agente e istanzia i corretti trigger che andranno a inizializzare le animazioni del grafo Mecanim nel momento opportuno.

Per definire l'automa a stati finiti in forma di codice si parte dal definire una struttura dati, detta *enum* che conterrà delle variabili e il loro hash code: questa struttura è simile a un Dictionary, perciò serve a indicare una variabile e inizializzarla con un valore univoco, che è in questo specifico caso

[1]Questi non sono gli stessi trigger definiti in precedenza tramite i collider, ma sono particolari variabili booleane che si possono creare tramite la voce *Parameters* in basso a sinistra della finestra *Mecanim*, che vengono automaticamente reimpostate a false ogni volta che diventano vere

solitamente la traduzione, da stringa ad hash code, del nome dello stato. Definiti tutti gli stati possibili si deve poi creare una variabile che indchi lo stato corrente dell'agente, che cambierà poi dinamicamente quando le varie condizioni di passaggio di stato vengono incontrate, e la si inizializza solitamente a uno stato *Idle*, in cui l'IA non fa nulla. All'interno dello stesso script sarà necessario anche creare reference ad altri Component, in particolare il Animator, il NavMeshAgent, il Transform e il Collider. L'automa a stati finiti è poi da codificare a parte come una Coroutine separata, per cui si dovrà avere una Coroutine per ogni stato: queste verranno riprodotto ciclicamente all'infinito e in maniera esclusiva, almeno finchè non si avrà un cambio di stato in cui verrà cambiata anche la Coroutine da eseguire.

Lo stato *Idle* è particolarmente importante in quando viene solitamente usato non solo in fase iniziale di startup, ma anche come stato intermedio per cui passare prima di passare allo stato successivo realmente desiderato. Di norma per passare da uno stato all'altro o all'inizio della scena in fase di inizializzazione perciò, si passa per lo stato *Idle*, il quale eseguirà la sua animazione, che una volta finità manderà un messaggio al controller per fare la transizione allo stato successivo desiderato. Il codice della Coroutine per questo stato è composta da alcuni elementi ricorrenti: una prima istruzione obbligatoria che imposta lo stato corrente dell'agente su quello *Idle*, una chiamata all'Animator per eseguire l'animazione dello stato tramite l'attivazione del trigger corrispondente, una chiamata al metodo Stop() della NavMeshAgent per far fermare sul posto l'agente, e infine un ciclo infinito per far riprodurre l'animazione con un controllo sulla condizione di transizione di stato che lo interrompa se risulta vera. Definendo così però lo stato *Idle* l'animazione viene riprodotto all'infinito finchè non viene incontrata la condizione richiesta, mentre invece è più probabile che si voglia, come già detto, che sia solo uno stato momentaneo in cui l'agente si trovi solo per il tempo per l'animazione di essere eseguita, ma che una volta finita ci notifichi del suo completamento e faccia passare allo stato successivo desiderato: per fare questo si usano gli *Animation Events*, che si possono aggiungere all'interno di una timeline della finestra Animation, per causare chiamate a funzioni una volta che, ad esempio, un'animazione termina.

L'importanza di usare le Coroutine per programmare gli stati è che essendo, semplificando, delle funzioni asincrone che vengono eseguite in background, in parallelo, come su thread separati, i cicli infiniti non causano il crash del gioco: le Coroutine inoltre ritornato sempre un dato di tipo IEnumerator, che è il tipo specifico che serve per indicare che un determinato metodo all'interno di uno script, non è effettivamente un metodo normale che deve essere eseguito all'interno di un singolo frame, ma che è invece per l'appunto una Coroutine che viene eseguita su più frame, in base a quan-

ti indicati o richiesti per completare le istruzioni. Altra caratteristica delle Coroutine è che possono mettere in pausa l'esecuzione e ridare il controllo a Unity ma continuando da dove si sono interrotte, al frame successivo. All'interno di una Coroutine è inoltre sempre presente uno *yeld statement*, il quale permette di eseguire un'istruzione dopo che in scena trascorre un frame.

Prendiamo a questo punto come esempio il caso di un agente che, una volta uscito dallo stato Idle, debba iniziare a muoversi all'interno della NavigationMesh in modo casuale, e in caso il personaggio giocante sia all'interno della sua area visiva, passare a uno stato di inseguimento per cercare di arrivargli abbastanza vicino da attaccarlo. Lo stato in questione in cui l'agente deve transitare deve essere codificato in maniera ciclica, per cui deve scegliere una posizione randomica, che deve però essere all'interno della NavMesh a cui appartiene, e deve continuamente fare un controllo se il giocatore si trova all'interno della sua area visiva. Il metodo più comune per il primo problema è quello di definire dei punti all'interno della mappa, detti *waypoints*, che saranno GameObject vuoti, che l'agente inserisce all'interno di un array, e ne prende uno a caso al suo interno; il NavMeshAgent si occuperà poi di eseguire gli spostamenti corretti all'interno della mesh di navigazione, tramite metodo *SetDestination()* che prende come parametro un Vector3 che indichi la posizione in cui spostare l'agente. All'interno del loop infinito di stato è poi necessario definire, dopo il controllo della condizione di transizione, ovvero se il giocatore è all'interno del campo visivo, un altro controllo per far scegliere all'agente un altro punto casuale quando raggiunge quello precedentemente scelto. Per controllare invece se l'agente riesce o meno a vedere il giocatore, si crea solitamente una variabile booleana assunta falsa, che viene cambiata in vera quando il Collider del giocatore entra all'interno del Trigger Collider dell'agente. Non basta tuttavia controllare che il giocatore sia all'interno del cubo con cui abbiamo approssimato il Field of View dell'agente, ma è necessario anche verificare che non ci siano ostacoli tra i due modelli: questo si può fare tramite Physics.LineCast() che è stato presentato nel capitolo 3.3.2 "*Scripting e design patterns: altre nozioni fondamentali*".

Nel caso il giocatore sia visibile dall'agente, vogliamo che questo passi allo stato di inseguimento, il quale dovrà cambiare in due circostanze: se NPC raggiunge il giocatore e arriva a portata di attacco, cambiando nello stato corrispondente, oppure in caso il giocatore sparisca dallo spazio visivo e sia passato un determinato periodo di tempo, rinunciare e tornare allo stato di ricognizione. All'interno del ciclo infinito di questa Coroutine si deve per prima cosa impostare come destinazione dell'agente la posizione del giocatore in modo che inizi effettivamente a inseguirlo, e successivamente implementare il controllo se quest'ultimo sia ancora in vista. In caso il giocatore riesca a fuggire dal nemico, l'agente deve iniziare un timeout, che si può facilmente

fare tramite un ciclo infinito e la funzione Time.DeltaTime() per calcolare il tempo trascorso, che se viene raggiunto o superato, fa cambiare di stato l'agente. Un modo per far continuare l'inseguimento anche dopo che il giocatore è uscito dalla visuale è quello di reimpostare la destinazione del nemico sulla posizione del giocatore, ma un metodo più realistico, anche se più complicato, sarebbe quello di impostare come destinazione un *waypoint* casuale all'interno di un array ristretto di waypoint che si trovano a una certa distanza limite dall'agente nella direzione in cui il giocatore si è spostato.

Resta ora solo da codificare lo stato di attacco, in cui l'agente si dovrà trovare quando la sua distanza dal giocatore sarà sufficientemente corta e a gittata, definita dal programmatore, in base sopratutto all'attacco del nemico. Per creare un gioco equilibrato e non impossibile, si programma questo stato in modo che il nemico, una volta sferrato un attacco, abbia bisogno di un certo periodo di tempo per riprendere forze fermandosi o ricaricare l'arma, mentre il giocatore possa continuare a scappare, o comunque muoversi. L'agente deve rimanere all'interno di questo stato per tutto il tempo in cui il giocatore si trovi a gittata e sia però anche visibile, quindi non ci siano oggetti interposti, ed esca per tornare nello stato di inseguimento quando esce dalla gittata dell'attacco.

L'ultimo stato che mostrerò è una ulteriore miglioria che si può apportare al proprio gioco per renderlo più realistico, e che per quanto possa essere facile da implementare, sopratutto per sviluppatori esperti, viene incluso molto raramente. Lo stato in questione è quello di fuga, per cui se il giocatore può difendersi contrattaccando, riducendo il nemico a un certo treshold di punti salute, quest'ultimo passi dallo stato di attacco a quello di fuga, che può essere semplicemente cercare di allontanarsi dal giocatore il più possibile, o muoversi verso destinazioni dove sa ci siano suoi alleati o dove sono presenti oggetti che gli ricaricano la salute. É uno stato che può essere raggiunto da qualunque altro in quanto la transizione non dipende dalla sua relazione con gli altri stati, ma solo dal valore della salute dell'NPC.

4 Conclusioni

In questo breve elaborato ho cercato di mostrare le funzionalità principali e più utili per lo sviluppo di giochi o applicazioni semplici, che possono permettere a neofiti del framework di sviluppare fin da subito alcuni progetti non troppo ambizioni, ma che di certo possono portare a qualche soddisfazione, oltre che farli iniziare a prendere mano con l'ambiente. Ho cercato di spiegare il più esaustivamente possibile i componenti più importanti, provando però anche a tenerli il più concisi possibile per fornire velocemente basi su cui

poter iniziare a lavorare fin da subito, anche man mano la lettura della tesi. Per verificare se i concetti espressi fossero il più chiari possibile ho inoltre creato una applicazione molto elementare che si basa su una buona parte di essi.

La libreria di Unity è tuttavia in continuo sviluppo e vengono rilasciate nuove versioni dell'applicativo molto di frequente, perciò alcuni o molti dei metodi e componenti che ho illustrato, potrebbero diventare deprecati o cambiare nella loro implementazione con il tempo, ma ho volutamente cercato di spiegare quelli che nel corso degli anni dello sviluppo del framework sono rimasti per lo più sempre invariati da una versione all'altra: in particolare tutto ciò riguardante la fisica è rimasto pressochè invariato fin dalla prima release di Unity, in quanto dipendono in gran parte dal motore fisico di Nvidia.

La fortuna di ogni nuovo sviluppatore che decide di interfacciarsi a questo mondo è tuttavia che nonostante i continui aggiornamenti, con Unity cambia costantemente e in maniera molto rigorosa anche la sua documentazione, che resta, oltre alle ricerche sul Web, la migliore fonte di informazione per aggiornarsi e imparare a usare l'applicazione.

Essendo gratuito nella maggior parte dei suoi aspetti, è molto facile trovare librerie o funzionalità create da utenti, ed è sempre più facile utilizzare codice creato da terzi e template per il proprio progetto: lo scopo della tesi era quindi quello di mostrare le basi in modo da essere completamente indipendenti nel momento in cui si voglia sfruttare Unity, e capire cosa c'è alla base del framework. C'è moltissimo codice di terzi che implementa funzionalità nuove o modifica e migliora quelle già presenti, ma Unity è stato pensato, tranne che per la sua parte grafica di modellazione 3D e 2D, per fornire tutto il necessario per creare applicazioni completamente funzionanti e il più ottimizzate possibili, dove il corretto funzionamento del progetto è in mano al solo sviluppatore e alle sue abilità nell'usare le funzionalità corrette per il lavoro.