Università degli Studi di Salerno

DIPARTIMENTO DI INFORMATICA

Tesi di Laurea in Informatica

Cinemachine per la simulazione di una visita di un museo

Relatori

Prof. Andrea Francesco Abate

Prof. Ignazio Passero

Candidato

Andrea Sansone

Matr. **0512103436**

Anno Accademico 2020/2021

ABSTRACT

Gestire una camera all'interno di una scena 3D è sempre stato un punto cruciale durante lo sviluppo di esperienze immersive. Alcuni degli approcci più comuni, comportano la programmazione diretta di come l'utente può gestire la sua visuale, generando spesso risultati non del tutto soddisfacenti o errati. Ciò ha portato alla definizione di svariati tools che semplificano la progettazione di un **Camera System** per mezzo di template pre-configurati e un'interfaccia grafica "*user-friendly*". L'obiettivo che si pone la mia tesi è quello di estendere alcune delle funzionalità di questi tools per la simulazione di una visita di un museo. La prima domanda da porsi è: *Quale delle funzionalità è più adatta ad una visita di un museo?* In questo contesto l'utilizzo di un percorso pre-stabilito, su cui la camera si muoverà, risulta la soluzione più efficace, affiancata anche dall'aggiunta di *punti di interesse* all'interno della scena. I risultati ottenuti mostrano come sia semplice adattare queste funzionalità ad un contesto ben specifico, preservando pur sempre la loro flessibilità. Studi futuri potrebbero permettere un controllo della camera più dinamico basato sul singolo punto di interesse aumentando il coinvolgimento e l'impatto visivo sull'utente che vivrà le diverse esperienze.

SOMMARIO

ABSTRACT ..... **I**

CAPITOLO 1 ..... **2**

1. ORIGINE ED EVOLUZIONE DI UN CAMERA SYSTEM 2

1.1. Computer Grafica *2*

1.2. Virtual Camera System *3*

1.3. Cutscene *6*

CAPITOLO 2 ..... **7**

2. UNITY & CINEMACHINE 7

2.1. Unity *7*

2.2. Camera System in Unity *8*

2.3. Cinemachine Package *9*

CAPITOLO 3 ..... **15**

3. IL MUSEO VIRTUALE DELLA SCUOLA MEDICA SALERNITANA 15

3.1. Obiettivi *15*

3.2. Descrizione del progetto *15*

3.3. Modello 3D del Museo Virtuale della Scuola Medica Salernitana *16*

3.4. Punti di interesse – TriggerSpot *18*

3.5. Percorso guidato – Path Trigger List *20*

3.6. Movimento della Camera - Cart Player *23*

3.7. Interfaccia utente – Cart UI *34*

CAPITOLO 4 ..... **38**

BIBLIOGRAFIA **39**

RINGRAZIAMENTI **42**

CAPITOLO 1

FONDAMENTA DI UN MONDO VIRTUALE

1. *Origine ed evoluzione di un Camera System*

1.1. Computer Grafica

La computer grafica è quella disciplina informatica in cui si tratta la generazione e manipolazione di immagini attraverso un computer [1]. Nei suoi primi anni di sviluppo (1955-1960) non era una tecnologia alla portata di tutti, dato l'elevato costo dell'hardware grafico [2]. Le sue prime applicazioni erano disponibili solo su display che utilizzavano la scansione verticale ed erano in bianco e nero (Figura 1).

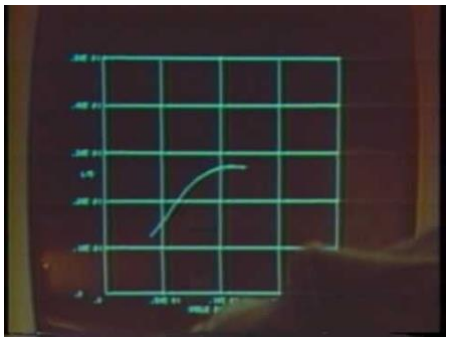

Figura 1 - Grafico generato a computer

Nei decenni successivi, la computer grafica ha subito un'evoluzione radicale grazie all'introduzione di nuove tecniche e hardware più competente. Arrivano i primi cenni di scansione raster, di rendering e dell'uso del termine "pixel" (anni '70) [2]. Prima ancora dell'avvento della *computer grafica 3D* e del **Virtual Camera System**, ciò che veniva generato dal computer era trattato come una matrice bidimensionale le cui dimensioni spesso coincidevano con il display in uso e i cui elementi rappresentavano i pixel dell'immagine. Con lo sviluppo delle tecnologie, questo concetto è stato inglobato nella definizione di *Rendering*, ovvero il processo di *"generazione di una immagine bidimensionale a partire da una descrizione matematica di una scena tridimensionale, interpretata da algoritmi che definiscono il colore di ogni punto dell'immagine"* [3]. In parole più semplici il processo di *Rendering* avviene per mezzo di una Camera che cattura una porzione della scena 3D da cui verrà successivamente generata l'immagine bidimensionale detta *Render*.

1.2. Virtual Camera System

Il *Virtual Camera System* ha lo scopo di gestire una o più camere con l'intento di mostrare a schermo porzioni di una scena 3D [4]. Principalmente è un sistema utilizzato all'interno dei videogiochi che consente di avere la migliore angolazione possibile sull'azione che si sta svolgendo al momento. Al contrario di una camera cinematografica non è possibile pianificare in anticipo le riprese e la regia. Chi sviluppa un sistema di camera dovrà fare i conti con un mondo dinamico e imprevedibile in cui l'utente sarà in costante movimento.
Per risolvere questo problema, il sistema si affida ad un insieme di regole o delle intelligenze artificiali per selezionare lo scatto più pulito e appropriato.
Fondamentalmente si possono identificare tre tipi di camera:

- *Fixed Camera System* – L'utente è inquadrato in una successione di scatti continui in cui la camera non si muove (Figura 2). In fase di progettazione, una Fixed Camera viene definita dalla sua posizione nella scena 3D, l'orientamento in una direzione specifica e il Field of View (FOV – profondità di campo). Uno dei vantaggi di questo sistema è la possibilità di utilizzare un linguaggio cinematografico per creare determinate atmosfere attraverso delle riprese ad hoc.

Figura 2 - Esempio di uso di una Fixed Camera [5]

- *Tracking Cameras* – La camera traccia costantemente l'utente, che non ne ha il controllo, posizionandosi dietro le sue spalle (Figura 3). Questo tipo di camera era popolare nei primi videogiochi 3D data la sua semplicità nell'implementazione ma presentava dei problemi noti. In particolare, se la visuale era occlusa da un qualsiasi oggetto o non mostrava in modo adeguato la posizione dell'utente, poteva rischiare di sobbalzare o ritrovarsi in posizione alquanto scomode [6].

Figura 3 - Esempio di uso di una Tracking Camera [7]

- *Interactive Camera System* – Può considerarsi una variante migliorata della Tracking Camera. È un sistema parzialmente automatizzato che consente all'utente di modificare direttamente la sua visuale. La camera continua con il tracciamento ma alcuni dei suoi parametri gli consentono di modificare autonomamente l'orientamento o la distanza dall'utente (Figura 4).

Figura 4 - Esempio di uso della Interactive Camera, da notare come la visuale si auto-adatti all'ambiente [8]

L'implementazione di un *Camera System* è un'operazione molto complicata e solo per mezzo di tecniche ad hoc è possibile gestirlo in modo pulito e chiaro. Inizialmente il cambio camera veniva attivato per mezzo di script specifici [9] che contenevano le istruzioni necessarie per definire i parametri della camera (Figura 5).

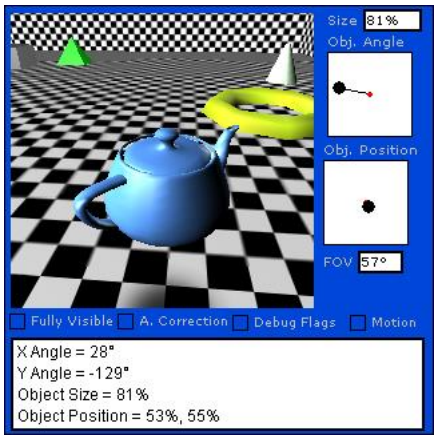

Figura 5 - Interfaccia che mostra la struttura di un Camera System [10]

Le implementazioni più diffuse sono quelle del *Constraint Solver,* dell'*Artificial Intelligence* e dell'*Autonomous Agent*. L'approccio applicato da ogni singola implementazione è diversa l'una dall'altra.

Il **Constraint Solver** [11] genera la miglior ripresa basandosi su un insieme di vincoli visivi, ad esempio "*l'utente deve occupare il 30% dello schermo"* oppure "*l'utente deve essere visualizzato al centro dello schermo".* Nel caso in cui il solver non riesca a soddisfare tutti i vincoli, c'è la possibilità di "alleggerirli" ignorandoli durante la generazione della ripresa [12].

L'**Artificial Intelligence** influenza solo in parte i parametri della camera. Durante il suo sviluppo, l'AI viene informata della conformazione della scena, potendo così anticipare alcune riprese. Un esempio di questo approccio è l'*Interactive Camera*.

L'**Autonomous Agent** [13] prevede che la camera sia un'entità autonoma con una propria personalità, il cui stile delle riprese e il loro ritmo è direttamente influenzato dall'atmosfera della scena.

Ad oggi, l'evoluzione di queste tecniche ha consentito lo sviluppo di nuovi tools che offrono diverse implementazioni e template pre-configurati per poter gestire in maniera professionale un *Camera System*. Ciò ha portato, lo sviluppatore, a spostare il proprio interesse su tutti quei concetti di utilizzo nella fotografia cinematografica: nascono le *Cutscene*.

1.3. Cutscene

Una cutscene (anche detta *Intermezzo*) è una sequenza **non interagibile**, maggiormente sfruttata nei videogiochi, che ha diversi utilizzi in base a ciò che si vuole trasmettere all'utente. Per esempio, per mostrare una nuova meccanica di gameplay, una conversazione tra più personaggi o anticipare eventi futuri [14][15].
Possono essere divise in:

- ***Live-Action***, vengono utilizzate delle sequenze video girate in full-motion e integrate come semplici filmati all'interno dell'esperienza.

- ***Pre-renderizzate***, ovvero una sequenza video pre-registrata, non in tempo reale, di cui è stato effettuato il rendering *frame-by-frame* con l'engine di riferimento. Vantaggiose in termini di prestazioni e qualità, ma svantaggiose poichè risultano mostrare una grafica diversa da quella in game, ad esempio un personaggio creato dall'utente è sostituito nella cutscene da uno di default.

- ***In tempo reale***, anche conosciute con il nome di *Machinima*, sono l'opposto delle cutscene pre-renderizzate. Hanno una qualità di dettaglio molto minore rispetto a quest'ultime ma possono adattarsi ai cambiamenti dinamici dell'esperienza. Ad esempio, sequenze video associate ad eventi randomici (possono essere attivate in qualsiasi momento).

- ***Interattive***, la cutscene coinvolta prende il controllo dell'avatar dell'utente mostrando a schermo delle azioni che quest'ultimo dovrà effettuare per avanzare nella sequenza. Questa meccanica è chiamata *Quick Time Event*.

Le cutscene presentano il cosiddetto "*Rendering On-Fly"* che sfrutta la grafica in engine, insieme ad eventi scriptati, per creare sequenze video che "rompono" il ritmo del dell'esperienza [16]. Sono divenute così comuni nei videogiochi grazie all'avvento del CD-ROM come dispositivo di memorizzazione primario [17]. La maggiore disponibilità di spazio ha consentito agli sviluppatori di utilizzare media di alta qualità portando le cutscene ad un livello superiore come strumento per la *User Experience*.

CAPITOLO 2

DA PROGRAMMATORI … A REGISTI

2. Unity & Cinemachine

2.1. Unity

Per il lavoro di tesi è stato utilizzato Unity [18] (Figura 6), un motore grafico real-time multipiattaforma sviluppato da Unity Technologies, in grado di sviluppare videogiochi e altri contenuti interattivi quali visualizzazioni architettoniche o animazioni 3D. Unity si integra perfettamente con altri software 3D DCCs (Digital Content Creation tools) come Blender, Maya, ZBrush. Una delle sue caratteristiche principali è la facilità con cui gli utenti riescono ad utilizzarlo, grazie alla presenza di un'interfaccia *user-friendly* e una *community* enorme a cui chiedere aiuto.

L'ambiente Unity è disponibile per i sistemi operativi Microsoft Windows, macOS e Linux e consente di esportare gli applicativi sviluppati per piattaforme PC, Console, Mobile, VR, WebGL. Unity Technologies supporta l'engine pubblicando periodicamente nuove versioni al fine di rimanere a passo con lo sviluppo di nuove tecnologie.

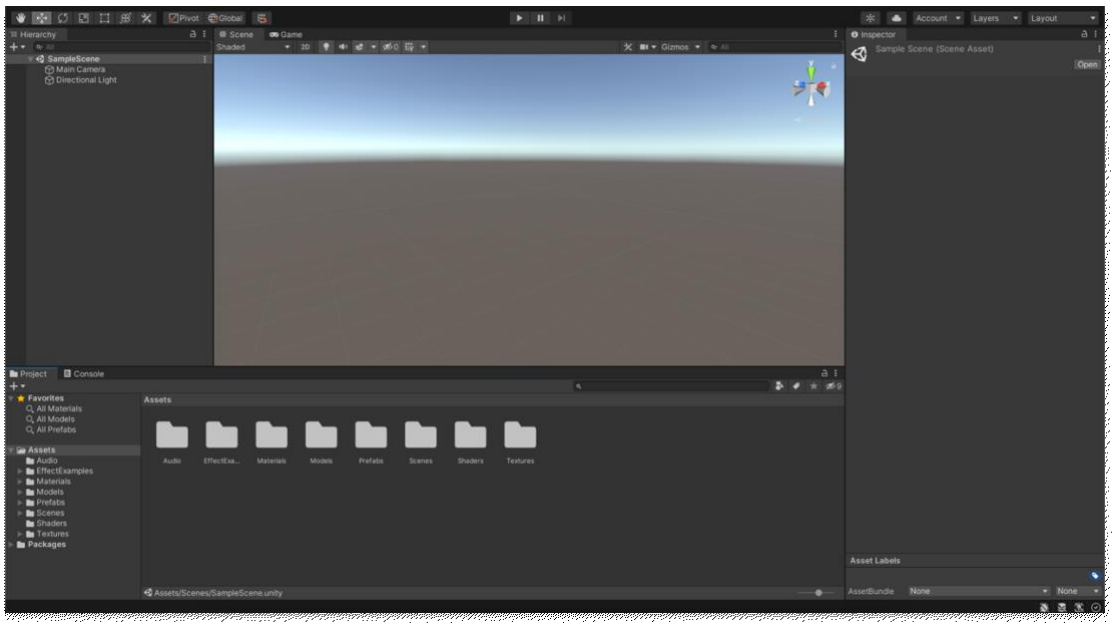

Figura 6 - Schermata iniziale di Unity

2.2. Camera System in Unity

La camera [19] è una componente fondamentale di Unity ed è il primo oggetto che viene istanziato ogni qualvolta creiamo una nuova scena 3D. La camera, come qualsiasi altro oggetto nella scena, è considerato un *GameObject* con due componenti di default (Figura 7): ***Camera Component*** e ***Audio Listener***. La scelta di queste componenti non è casuale, infatti in Unity la camera rappresenta idealmente gli occhi e le orecchie dell'utente.

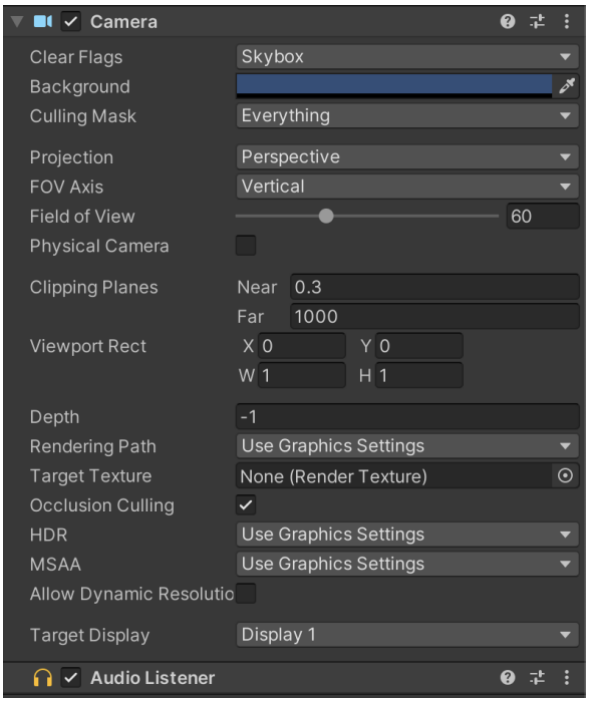

Figura 7 - Componenti attaccate all'oggetto Camera in Unity

Tramite i parametri del Camera Component è possibile configurare il comportamento della camera e il modo con cui mostra la scena all'utente[1]. I più utilizzati sono:

- ***Clear Flags***, indica cosa renderizzare in caso in cui la visuale è vuota. È possibile scegliere tra *Skybox* come simulazione di un cielo, *Solid Color* per un colore a tinta unita e altre opzioni più avanzate.

- ***Culling Mask***, in cui si può scegliere quali *Layer* specifici portare in rendering.

[1] Alcuni di questi parametri cambiano in funzione del tipo di proiezione scelta.

- ***Projection***, modifica il tipo di proiezione da *Prospettica* a *Ortografica*.

- ***Field of View***, cambia la profondità di campo, simile alla funzione di zoom.

- ***Clipping Planes***, definisce la distanza minima e massima di rendering.

Il modo in cui viene renderizzato ciò che l'utente può vedere è dato dal ***Frustum di visione*** (Figura 8), una figura geometrica simile ad una piramide senza punta. È definito dal parametro *Clipping* Planes: solo quello che si trova all'interno del Frustum verrà passato per il rendering.

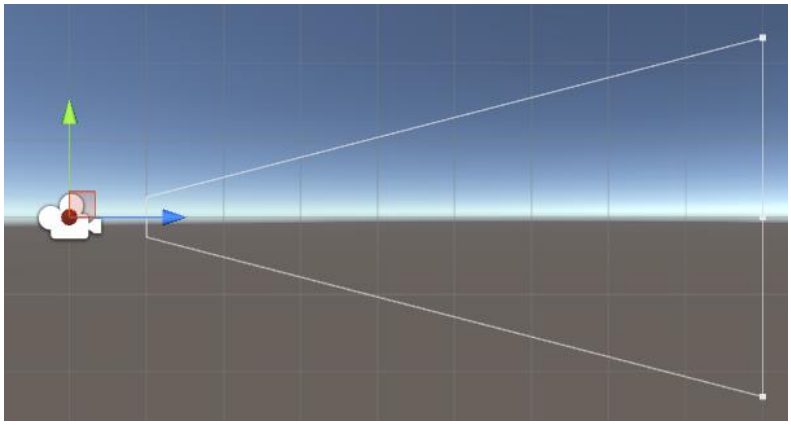

Figura 8 - Frustum di visione

2.3. Cinemachine Package

Il *Cinemachine Package* [20][21] è una suite di moduli che estendono le funzionalità della camera in Unity. Questo package consente di non vincolare lo sviluppo ad un insieme di regole logico-matematiche per il tracciamento di oggetti, il compositing e il cambio tra una ripresa e l'altra. Lo si può includere nel proprio progetto attraverso il *Package Manager* (Figura 9) che automaticamente importerà tutti i file necessari per l'utilizzo. La progettazione ha lo scopo di ridurre significativamente il time-consuming per la manutenzione e revisione degli script associati ad una camera. Infatti, offre la possibilità di modificare i parametri in real-time durante l'esecuzione della cutscene, così da poter testare nuovi punti di ripresa e comportamenti.

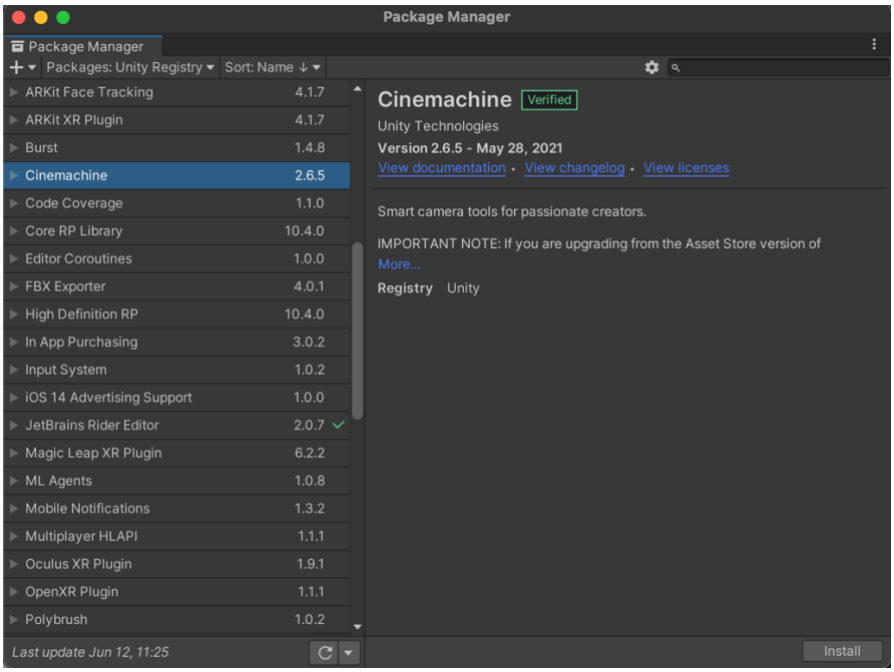

Figura 9 - Package Manager con la Cinemachine pronta per il download

Utilizzare la Cinemachine richiede un nuovo approccio rispetto a quello a cui si era abituati. Prima era necessario investire molto tempo e abilità per poter definire il comportamento scriptato di una camera ma, con l'implementazione della Cinemachine, è possibile ottenere gli stessi risultati in meno tempo e con più facilità grazie ad alcune funzionalità già configurate (Figura 10 e Figura 11). Tra tutte le tipologie di camera offerte, quella utilizzata all'interno del progetto è la ***Cinemachine Dolly Track & Dolly Cart***. Il concetto alla base della Cinemachine è quello di creare un *GameObject* che prende il controllo della camera principale nella scena, e la setti in base ai parametri definiti. Questo oggetto è detto ***Virtual Camera***.

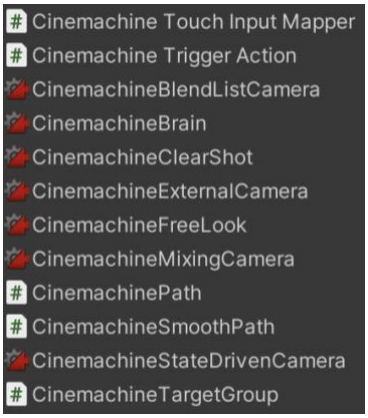

Figura 10 – Alcune componenti del Cinemachine

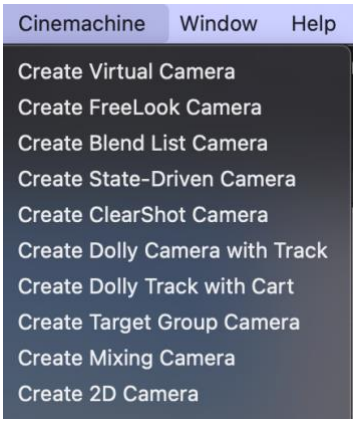

Figura 11 – Funzionalità offerte dal Cinemachine Package

♦ Virtual Camera

Una Virtual Camera [21] è un *GameObject* separato dalla camera di Unity e ha un comportamento indipendente da essa. All'interno della scena possono esistere più di una Virtual Camera, ognuna con il proprio set di parametri. Per questioni di performance sarebbe adeguato avere attiva solo la Virtual Camera in utilizzo in un certo momento, nonostante la Cinemachine invogli la creazione di più camere. Le attività principali di una Virtual Camera sono quelle di posizionare la camera di Unity nella scena, direzionarla verso un punto o un oggetto specifico, farla muovere lungo un percorso o in base al movimento di un altro oggetto, aggiungere dei rumori procedurali come le vibrazioni di un veicolo, e tante altre possibilità. Come per la camera di Unity, la Virtual Camera è caratterizzata dalla componente ***CinemachineVirtualCamera*** (Figura 12) suddivisa in diverse sezioni, ognuna la quale gestisce un aspetto specifico per la camera di Unity.

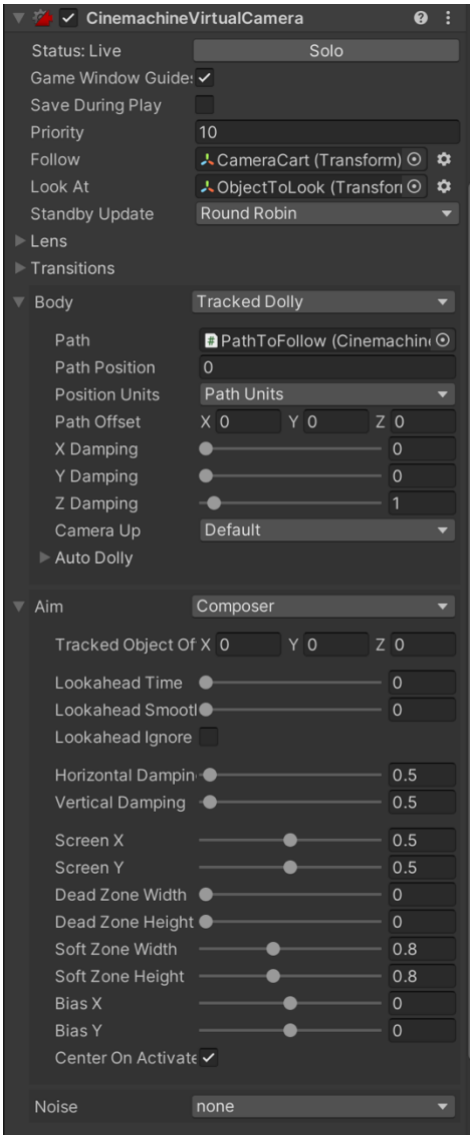

Figura 12 - Componente di uso nella Virtual Camera

- ***Priority***, definisce una priorità per la singola Virtual Camera evitando che durante la cutscene si possano accavallare tra di loro.

- ***Follow / Look At***, prendono come input il *Transform* di un oggetto. Con Follow è possibile far si che la camera segui l'oggetto nella scena, con *Look At* consente di direzionare costantemente la camera verso l'oggetto.

- ***Lens***, sezione in cui è possibile modificare le impostazioni della camera come, ad esempio, l'FOV o la distanza di rendering.

- ***Body***, qui è possibile scegliere il tipo di algoritmo che si occuperà di muovere la camera nella scena. Alcune delle opzioni disponibili sono *Tracked Dolly, 3rd Person View, Orbital*. In funzione di queste opzioni la sezione Body cambierà.

- ***Aim***, qui è possibile scegliere il tipo di algoritmo che si occuperà di direzionare la camera nella scena. Alcune delle opzioni disponibili sono *Composer, POV, Hard Look At*. In funzione di queste opzioni la sezione Aim cambierà. I parametri utilizzati da questi algoritmi sono necessari per far si che la camera riesca a posizionare il suo target sullo schermo in modo corretto.

Nel momento in cui viene creata una Virtual Camera, Unity automaticamente aggiunge una componente alla sua camera principale: ***CinemachineBrain*** [21] (Figura 13). Questa componente monitora costantemente tutte le Virtual Camera presenti in scena[2], decidendo quale di queste potrà avere il controllo della camera principale. Spesso questo passaggio è affiancato dall'uso di alcune *Timeline* che attivano/disattivano le Virtual Camera.

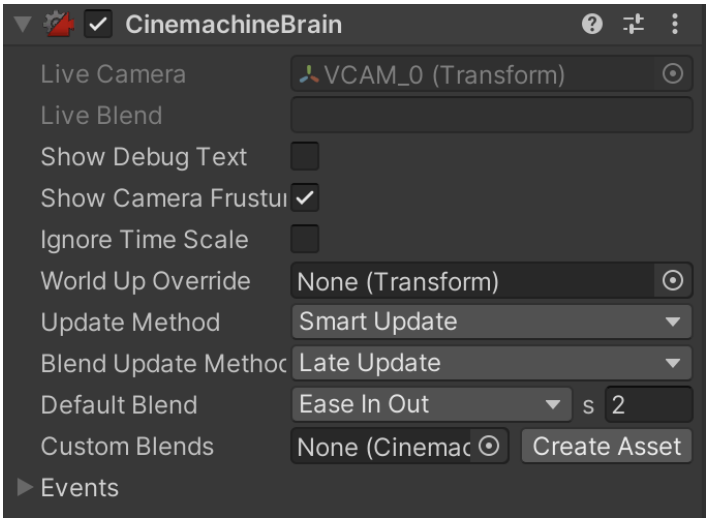

Figura 13 - Componente CinemachineBrain attaccata alla Camera principale

[2] Nel caso in cui ci fossero più Camere di Unity e viene creata una Virtual Camera, è doveroso assicurarsi che la componente Cinemachine Brain sia stata aggiunta solo alla camera di Unity necessaria.

♦ Dolly Track & Dolly Cart

L'idea dietro questa funzionalità è quella di simulare l'utilizzo di un Dolly Cart come quello in uso nelle produzioni cinematografiche (Figura 14). Brevemente, un Dolly è una struttura composta da un binario (*Track*) e un carrello (*Cart*) su cui montare la camera. Successivamente il carrello viene fatto muovere sul binario creando delle riprese controllate e stabili. La sua versione nel *Cinemachine Package* presenta molte similitudini, infatti anch'essa è composta da un *Dolly Cart*, un *Dolly Track* e una *Virtual Camera* (Figura 15).

Figura 14 - Dolly Cart con camera posizionato sopra un Dolly Track

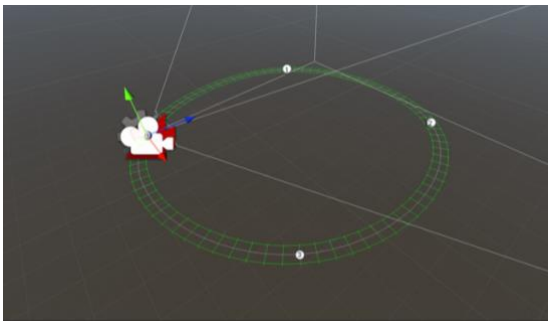

Figura 15 - Dolly Track con Dolly Cart e Virtual Camera

Un ***Dolly Cart*** altro non è che un *GameObject* con una singola componente: ***Cinemachine Dolly Cart*** [21] (Figura 16) con cui è possibile definire la velocità del cart (Speed*)*, la sua posizione lungo il percorso (*Position*), il metodo di aggiornamento della posizione (*Update Method*) e il tipo di unità da utilizzare per definire la posizione del cart (*Position Units*). Quando si andrà a creare la Virtual Camera, il cart dovrà essere passato come riferimento per il parametro *Follow* della componente sulla Virtual Camera e settare l'algoritmo nella sezione *Body* su *Tracked Dolly*.

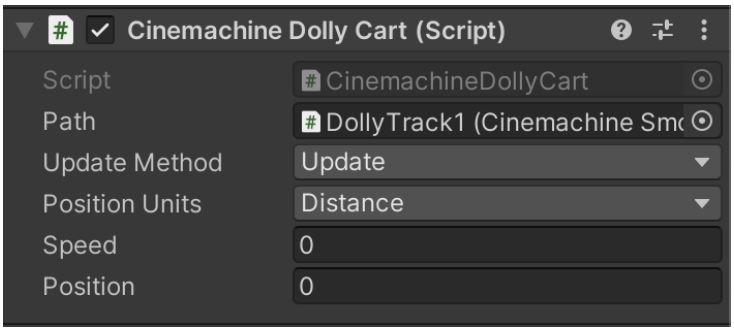

Figura 16 - Componente di un Dolly Cart

Un ***Dolly Track*** è anch'esso un *GameObject* con una singola componente, **Cinemachine Smooth Path** [21] (Figura 17), che definisce un insieme di punti nella scena 3D. I parametri mostrati permettono di definire alcune caratteristiche del percorso:

- *Resolution* è un valore intero che determina la precisione nel calcolare le distanze tra i punti e la posizione del cart.

- *Appearance* permettere di personalizzare la parte grafica del percorso visibile solo in editor, un colore per il percorso attivo, un colore per quello non attivo e la larghezza del binario.

- *Looped* crea un percorso chiuso continuo connettendo l'ultimo punto al primo.

- *Waypoints*, è una lista di punti che definiscono il percorso: i segmenti tra un punto e l'altro vengono curvati per rendere più fluido il movimento di camera[3].

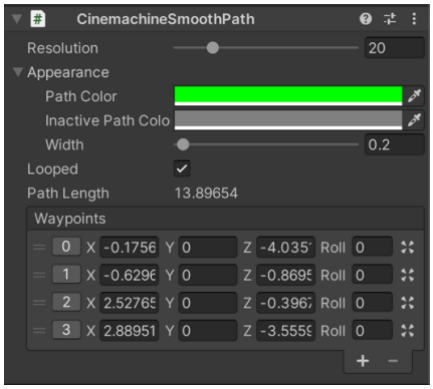

Figura 17 - Componente attaccate al Dolly Track

[3] Esiste una variate: ***Cinemachine Path*** in cui è possibile regolare la curvatura del segmento tra i singoli Waypoint. Animare una camera su di esso potrebbe causare strane vibrazioni alla visuale.

CAPITOLO 3

VISITA GUIDATA DI UN MUSEO

3. Il museo virtuale della Scuola Medica Salernitana

3.1. Obiettivi

Gli obiettivi che ho posto al mio lavoro di tesi sono molteplici.
Il primo è quello di definire uno strumento che faciliti la creazione di un *percorso guidato* per la camera, affiancato da oggetti che rappresentano i *punti di interesse* all'interno della scena 3D. Per il raggiungimento di questo obiettivo mi sono affidato al *Cinemachine Package* e alla sua funzionalità di *Dolly Track & Dolly Cart*.
Il secondo obiettivo è far si che l'utente possa usufruire di questa visita guidata, tramite l'implementazione di una *User Interface* che comprenda dei comandi a schermo per muovere la camera lungo il percorso.

3.2. Descrizione del progetto

Il progetto su cui ho lavorato fa parte di un insieme di diversi lavori, tutti mirati alla ricostruzione virtuale di quei luoghi in cui è nata e si è sviluppata la *Scuola Medica Salernitana* (Figura 18). Nel particolare, il mio progetto si è occupato della chiesa di *San Gregorio* oggi sede del ***Museo Virtuale della Scuola Medica Salernitana***.

Figura 18 - Logo della Scuola Medica Salernitana [22]

Grazie anche al contributo dei miei colleghi, che si sono occupati del posizionamento degli oggetti nella scena, ho potuto facilmente individuare i punti su cui la camera dovrà porre maggiore attenzione, calibrando il percorso per una ripresa fluida.

3.3. Modello 3D del Museo Virtuale della Scuola Medica Salernitana

Il modello è stato realizzato da me personalmente all'interno del software Blender, utilizzando come reference le foto dell'edificio trovate su internet (Figura 19). L'intento principale è stato quello di ottenere un riscontro reale sia in dimensioni, che come resa visiva con le texture. Una volta completata e raffinata la mesh, il modello (Figura 20) è stato importato all'interno di Unity rimappandone i materiali sull'*High Definition Render Pipeline* (HDRP).

Figura 19 - Museo Virtuale, reference

Figura 20 - Museo Virtuale, ricostruzione 3D

L’illuminazione nella sala interna è stata ideata in modo da dare visibilità a tutti gli oggetti posizionati precedentemente e ai dettagli del pavimento a griglia con i relativi riflessi (Figura 21, Figura 22, Figura 23).

Figura 21 - Museo Virtuale, sala interna

Figura 22 - Museo Virtuale, sala interna vuota (ricostruzione 3D)

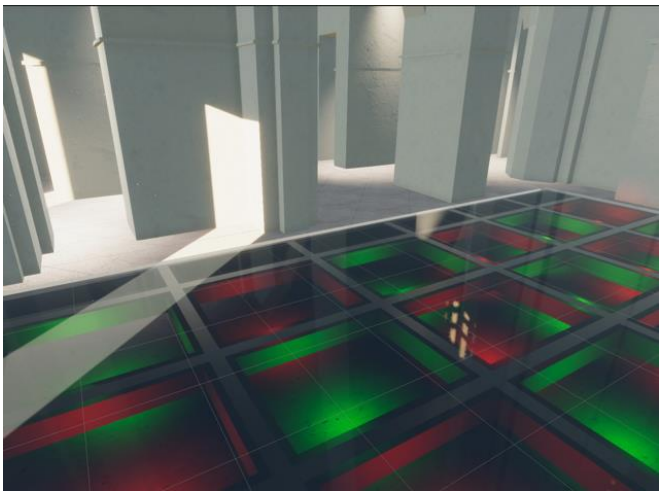

Figura 23 - Museo Virtuale, dettagli pavimento (ricostruzione 3D)

3.4. Punti di interesse – TriggerSpot

I punti di interesse inizialmente, erano stati ideati come dei semplici spot in cui la camera si sarebbe dovuta fermare, orientandosi verso l'oggetto di focus associato. Tra un prototipo e l'altro, ci si è resi conto della possibilità di fornire più informazioni sullo spot, così da renderlo più dinamico: ad esempio la velocità di transizione nell'orientare lo sguardo da un oggetto all'altro, le diramazioni tra più percorsi nella scena, etc. Le informazioni dal punto di interesse vengono estratte durante l'evento di collisione tra lo spot e il cart con la camera. Lo spot è provvisto di uno *Sphere Collider* dichiarato come ***trigger*** e di un tag per poterlo distinguere durante la collisione. Il passo successivo è stato implementare l'uso di queste nuove informazioni (Figura 24) all'interno della visita guidata.

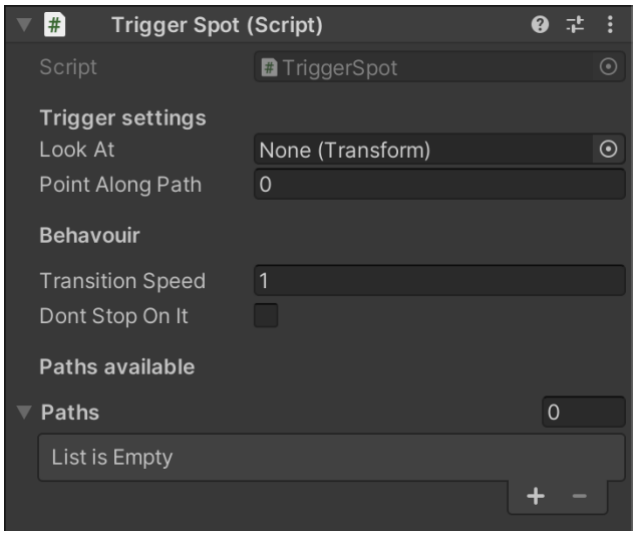

Figura 24 - Componente TriggerSpot

Il *TriggerSpot* è quello script associato allo spot, che ne permette la definizione: non ha nessuna funzione particolare se non i classici Getter/Setter per i parametri esposti. Con questo script (Figura 25) possiamo comunicare allo spot l'oggetto di focus (***Look At***), il punto lungo il percorso in cui si trova lo spot (***Point Along Path***), la velocità di transizione nello spostare lo sguardo (***Transition Speed***), se lo spot è solo di passaggio (***Dont Stop On It***) e per concludere una lista contenente le possibili diramazioni su altri percorsi (***Paths***). In caso di più direzioni alternative, dovrà avere nella lista anche il percorso da cui si dirama.

```
public class TriggerSpot : MonoBehaviour
{
    [Header("Trigger settings")]
    [SerializeField] Transform LookAt;
    [SerializeField] float PointAlongPath;
    [Space]
    [Header("Behavouir")]
    [SerializeField] float transitionSpeed = 1f;
    [SerializeField] bool dontStopOnIt;
    [Space]
    [Header("Paths available")]
    [SerializeField] List<CinemachinePathBase> paths;
    int listPosition = -1;

    // Getter Setter
    public Transform GetLookedObject() { return LookAt; }
    public void SetLookedobject(Transform obj) { LookAt = obj; }
    public float GetPointAlongPath() { return PointAlongPath; }
    public void SetPointAlongPath(float point) { PointAlongPath = point; }
    public bool HaveToStop() { return !dontStopOnIt; }
    public void SetToStop(bool value) { dontStopOnIt = value; }
    public float GetTransitionSpeed() { return transitionSpeed; }
    public void SetTransitionSpeed(float value) { transitionSpeed = value; }
    public int GetListPosition() { return listPosition; }
    public void SetListPosition(int value) { listPosition = value; }

    // Check if are other paths
    public bool HasOtherPaths() { return paths.Count > 1; }

    // Check if a path is in the list
    public int GetPathIndex(CinemachinePathBase pt)
    {
        // Avoid Errors
        if (!HasOtherPaths())
            return -1;

        return paths.FindIndex( x => x.Equals(pt));
    }

    // Return the next path on this spot
    public CinemachinePathBase GetNextPath(int index)
    {
        // Avoid Errors
        if (paths.Count <= 0)
            return null;

        index %= paths.Count;
        return paths[index];
    }
}
```

Figura 25 - Definizione della classe TriggerSpot

Questi parametri rendono il punto di interesse molto più flessibile, consentendone l'utilizzo come punti di passaggio: ad esempio anticipare l'oggetto di focus successivo a vantaggio di un movimento più fluido della camera. D'ora in poi, per una facilità di comprensione, i punti di interesse saranno chiamati *TriggerSpot*.

3.5. Percorso guidato – Path Trigger List

La progettazione dietro al percorso guidato è basata sulla funzionalità di *Dolly Track* del *Cinemachine Package*. La componente ***Cinemachine Smooth Path*** ha permesso di sviluppare un percorso (Figura 26) in modo immediato con la definizione di un insieme di punti detti *Waypoints.* L'interfaccia presente, come visto nel capitolo precedente, offre la possibilità di rappresentare graficamente il percorso e di scegliere con quale unità misurarne la lunghezza (quella utilizzata fa riferimento al sistema metrico di Unity).

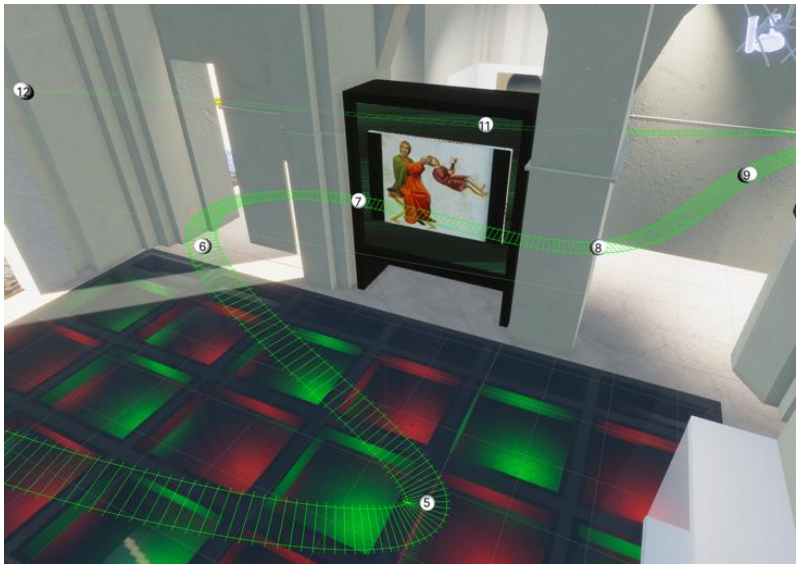

Figura 26 - Percorso guidato con i suoi Waypoints

La gestione dei *Waypoints* mi ha ispirato per il posizionamento dei *TriggerSpot*. Quest'ultimi, devono essere posti in linea con il percorso che seguirà la camera, altrimenti ne rischieremmo l'esclusione durante la visita guidata. Per ovviare a questo problema ho creato uno script ***Path Trigger List*** (Figura 27) che si occupa di esporre un nome per il percorso e, in maniera del tutto autonoma, di riposizionare tutti i *TriggerSpot* disponibili lungo il percorso, memorizzandoli all'interno di una lista. Per far si che lo script ottenga tutti i riferimenti di quest'ultimi, ho creato un *GameObject* (***TriggerSpotList***) (Figura 28) innestato nel percorso, che funge da *root* per tutti i *TriggerSpot*. Successivamente in fase di inizializzazione, lo script scorre tra i *children* di questo oggetto ottenendone il riferimento.

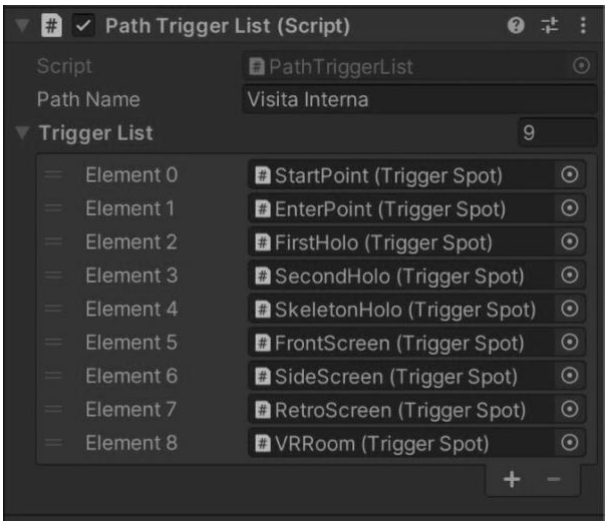

Figura 27 - TriggerList con tutti i TriggerSpot

Figura 28 - Struttura del percorso con i relativi TriggerSpot

Il codice associato allo script ***Path Trigger List*** è diviso in due sezioni: ricerca dei *TriggerSpot* (Figura 29) e la loro inizializzazione (Figura 30). La sezione sulla ricerca è contenuta nel metodo *Start()* di Unity. In breve, la lista dei *TriggerSpot* viene inizializzata e con il riferimento al *root* ***TriggerSpotList***, parte la ricerca di tutti i relativi *children*. Man mano che questi verranno trovati saranno aggiunti alla lista e riordinati in maniera crescente, in base alla loro posizione lungo il percorso. Successivamente con la lista ormai definita, viene comunicato al singolo *TriggerSpot*, il relativo indice di posizionamento nella lista per poter recuperare il riferimento dello spot successivo durante il movimento.

```
private void Start()
{
    // Get the path reference
    path = GetComponent<CinemachinePathBase>();
    pathLength = path.PathLength;

    // Check if the triggerList has been instantiated
    if (triggerList.Count <= 0)
    {
        triggerList = new List<TriggerSpot>();

        // Get the child with the trigger list and add each triggerSpot
        Transform tList = transform.GetChild(0);

        // Check if the trigger root exists
        if (tList == null)
        {
            Debug.LogWarning("No Trigger Root gameobject declared as child of the path");
            return;
        }

        // Add the triggerspot to the list
        for (int i = 0; i < tList.transform.childCount; i++)
        {
            TriggerSpot ts = tList.GetChild(i).GetComponent<TriggerSpot>();
            triggerList.Add(ts);

            // Sort the list based on trigger spot
            triggerList.Sort((t1, t2) => t1.GetPointAlongPath().CompareTo(t2.GetPointAlongPath()));
        }

        for (int i = 0; i < triggerList.Count; i++)
        {
            triggerList[i].SetListPosition(i);
        }

        // Init the triggerSpot position
        InitTriggerSpots();
    }
}
```

Figura 29 - Codice per la ricerca dei TriggerSpot nel percorso

La sezione sull'inizializzazione è più semplice, consiste in poche righe di codice che si occupano di riposizionare i *TriggerSpot* nella scena, sulla base della loro posizione nel percorso (il valore viene modulato sul numero di *Waypoints*) che viene convertito in coordinate relative alla scena.

```
// Initialize each TriggerSpot in list along the path
void InitTriggerSpots()
{
    foreach(TriggerSpot ts in triggerList)
    {
        float pathValue = ts.GetPointAlongPath() % path.PathLength;
        ts.transform.position = path.EvaluatePositionAtUnit( pathValue, CinemachinePathBase.PositionUnits.PathUnits);
    }
    isInit = true;
}
```

Figura 30 - Codice per l'inizializzazione dei TriggerSpot

3.6. Movimento della Camera - Cart Player

Grazie all'utilizzo del *Cinemachine Package* non è stato difficile muovere la camera su di un percorso prestabilito. La sua funzionalità di *Dolly Track & Dolly Cart* è sfruttata a pieno e ha consentito uno sviluppo veloce dell'intera implementazione. Con il suo utilizzo basta passare il riferimento del cart al campo ***Follow***, nella sezione *body* della Virtual Camera, e automaticamente quest'ultima si muoverà lungo il percorso.
Nota Bene: Nella sezione body della Virtual Camera, è presente un sottomenu "Auto Dolly" che consisterà di un flag da spuntare. Questo flag permetterà di far muovere il cart lungo il percorso in autonomia semplicemente variandone la velocità dalla sua componente

Il movimento di camera è stato progetto sullo stesso *GameObject* che funge da cart così da avere un riferimento diretto alle proprietà del *Dolly Cart*. La componente script ***CartPlayer*** (Figura 31) è stata divisa in diverse sezioni per facilitare il settaggio della visita guidata:

- La sezione *Virtual Camera* si occupa del riferimento alla camera (***Vcam***), l'oggetto a cui la camera guarderà (***Vcam Point Interest***), la velocità massima di orientamento della camera (***MaxTransitionSpeed***), il percorso da cui verrà estrapolata la *TriggerList* (***PathTriggerList***) e il riferimento alla *User Interface* (***CartUI***).

- La sezione *Cart Settings* permette di modificare la velocità di movimento del cart (***Cart Speed***) e, tramite un valore booleano, far si che il cart non si fermi su nessuno spot (***Dont Stop On Spot***). Quest'ultimo parametro è presente come funzione di debug per un test veloce del percorso ma può essere implementata facilmente come meccanica futura della visita guidata.

- La sezione *Misc* comprende un valore che definisce il tempo d'attesa prima dell'inizializzazione del cart (***Init Delay***) e un *Vector3* che rappresenta la grandezza del *collider* applicato alla camera (***Camera Box Collider***).

- La sezione *Debugging* è stata utilizzata durante lo sviluppo e il testing. Permette di scegliere un ID per la scrittura in console (***LOG_ID***), un valore booleano per usare o meno i comandi da tastiera (***Use Keyboard***) in assenza della *User Interface*, e un valore booleano per l'attivazione del debug stesso (***Use Debug Console***).

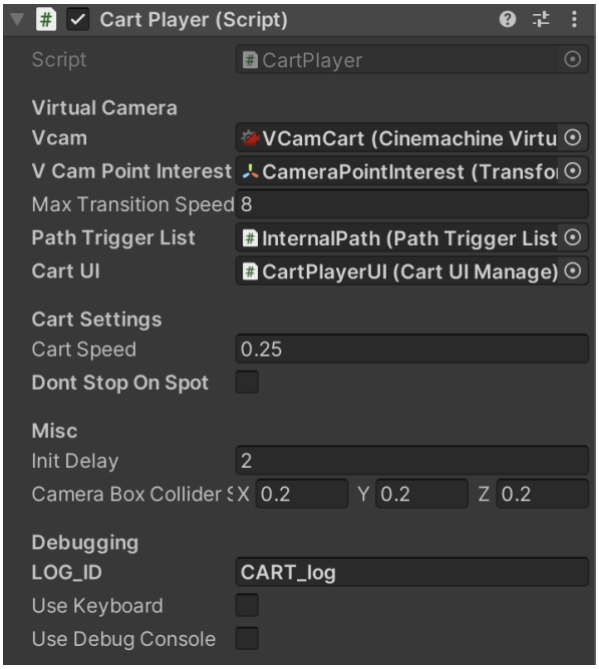

Figura 31 - Script attaccato al Dolly Cart per la gestione del cart

Per poter descrivere dettagliatamente lo script, è necessario approfondire il codice e la logica che caratterizzano l'istanza del cart. L'inizializzazione del cart, che si trova in uno stato di *"non attivo"*, è gestita da una *Coroutine* (Figura 32) che attende un certo tempo *t* dato dal parametro ***InitDelay***. Questa attesa è presente per evitare che le diverse interfacce utente (risultanti dal lavoro precedente dei miei colleghi) si interfoglino con quella che gestisce il movimento della camera. Successivamente vengono raccolte tutte le informazioni necessarie sul percorso di partenza con il metodo *UpdatePath()*. Con l'utilizzo di queste informazioni, la camera viene orientata in direzione dell'oggetto di focus, del primo spot, con il metodo *LookNewTarget()*. Per finire, il cart passa in uno stato di *"attesa"* e vengono invocati sulla *User Interface* quei metodi responsabili di attivare/disattivare gli elementi UI.

```
// Init the cart and path settings
IEnumerator Init()
{
    if (isInit)
    {
        // Already init so is a path change
        yield return new WaitForSeconds(0.01f);
    }
    else
    {
        // Wait to initilize the cart
        yield return new WaitForSeconds(initDelay);
    }

    // Update Path data
    UpdatePath();

    // Look to new target
    LookNewTarget();

    // Show the UI
    CartUI.ShowCommandsUI(true);
    CartUI.ShowPathName(true);

    if (startSpot != null)
    {
        CartUI.ShowPathButton(startSpot.GetComponent<TriggerSpot>().HasOtherPaths());
    }

    // Enable the cart
    cartState = CART_STATUS.WAITING;

    isInit = true;
}
```

Figura 32 - Coroutine per l'inizializzazione del cart

♦ Raccolta di informazioni sul percorso – UpdatePath()

Durante l'inizializzazione lo script prende, come parametro esposto, il riferimento ad un percorso che implementa il *Dolly Track* per poter leggere le informazioni riguardanti la lista dei *TriggerSpot* di quel percorso. Una volta che il cart ha ottenuto il riferimento dello script *PathTriggerList*, può estrapolarne i dati necessari per l'inizializzazione della visita guidata. Il metodo *UpdatePath()* (Figura 33) si occupa proprio di questo: ottiene il riferimento al percorso attuale e la relativa lista di tutti i *TriggerSpot* associati.

```
// Update Data paths
void UpdatePath()
{
    curPath = pathTriggerList.GetPath();

    // Get new Trigger List
    triggerList = pathTriggerList.GetTriggerList();

    // Re Position the cart
    instance.m_Position = triggerList[0].GetComponent<TriggerSpot>().GetPointAlongPath();
    vcamDolly.m_PathPosition = instance.m_Position;

    startSpot = triggerList[0].gameObject;

    // Pass new path to the cinemachine
    instance.m_Path = curPath;
    vcamDolly.m_Path = curPath;

    // Update path name in UI
    CartUI.ChangePath(pathTriggerList.GetPathName());
}
```

Figura 33 - Recupero delle informazioni riguardanti il percorso

Successivamente, comunica al cart il nuovo percorso riposizionandolo alle coordinate che coincidono con il primo spot in lista per poi passare all'interfaccia utente il nome del percorso da visualizzare all'utente.

Da notare che, quando modifichiamo il riferimento al percorso, questo deve essere comunicato anche alla Virtual Camera (***vcamDolly.m_Path***) poiché la sezioni *Body* utilizza l'algoritmo *TrackedDolly* che prende in input un percorso che implementi il *Dolly Track*.

♦ Cambio di inquadratura – LookNewTarget()

Nei primi prototipi della visita guidata, la camera alternava lo sguardo tra l'oggetto di focus dello spot e la direzione del binario del percorso. Ciò creava dei cambi di inquadratura troppo bruschi che confondevano la percezione dell'ambiente attorno alla camera. Questi cambi erano dovuti alla modifica del parametro ***LookAt*** nella Virtual Camera che applicava lo spostamento dello sguardo senza un vincolo temporale. Il passo successivo è stato creare un *GameObject* (***CameraPointInterest***) come riferimento definitivo per il parametro ***LookAt***. Questo oggetto ha lo scopo di muoversi in linea d'aria da un oggetto di focus all'altro. Con il metodo *LookNewTarget()* (Figura 34), possiamo determinare quale sarà il prossimo punto di interesse. Memorizzando il riferimento dell'ultimo *TriggerSpot* incontrato (***lastTriggerVisited***), ne utilizziamo l'indice di posizionamento nella *TriggerList* insieme al ***CartMoveDirection***[4] per estrapolare facilmente il riferimento del prossimo spot e di conseguenza l'oggetto di focus associato. Quest'ultimo verrà a sua volta memorizzato nella variabile ***nextTargetToLook***.

```
// Try to look the a new target
void LookNewTarget()
{
    // Get the next object to look
    if (lastTriggerVisited != null)
    {
        // Current list position of the last trigger visited
        int tsListPos = lastTriggerVisited.GetComponent<TriggerSpot>().GetListPosition();

        // Calculate the next list index
        int newListPos = (int)((tsListPos + cartMoveDirection) % triggerList.Count);
        newListPos = newListPos >= 0 ? newListPos : (int)(triggerList.Count - Mathf.Abs(newListPos));

        // Check if an object to look exists
        nextTargetToLook = triggerList[newListPos].GetLookedObject();
    }
}
```

Figura 34 - Codice per estrapolare le informazioni sul punto di interesse successivo

[4] Rappresenta la direzione di movimento del cart. Verrà approfondito nelle prossime pagine.

Per una transizione fluida è necessario sapere l'oggetto di focus corrente e quello verso il quale si vuole orientare la camera. Definito il segmento tra questi due oggetti, il cambio di inquadratura (Figura 35), computato nel metodo *Update()* di Unity, è vincolato alla velocità di transizione dettata dal *TriggerSpot* di partenza del segmento stesso.

```
    // Move the point of interest
    if ( cartState != CART_STATUS.STOPPED && nextTargetToLook != null)
    {
        VCamPointInterest.position = Vector3.MoveTowards(   VCamPointInterest.position,
                                                            nextTargetToLook.position,
                                                            curTransitionSpeed * Time.deltaTime
                                                        );
    }
}
```

Figura 35 - Codice per gestire la fluidità della transizione

Questo tipo di progettazione fornisce ancora più flessibilità in aggiunta a quella del *Cinemachine Package*. Nel caso in cui si volessero effettuare piccole modifiche sul comportamento della camera, è presente una sezione nella Virtual Camera che consente di definire un certo valore di *damping* da applicare al movimento o alla rotazione di quest'ultima.

♦ Implementazione del Cart a stati

La tecnica con cui il cart è stato implementato è quella degli *stati finiti*. Gli stati sono basati attraverso un'enumerazione **CART_STATUS** che comprende lo stato di *DISABLED, WAITING, RUNNING, STOPPED*. La gestione dei singoli stati avviene all'interno del metodo *Update()* di Unity (Figura 36) con l'uso di uno *switch*.

```
private void Update()
{
    // Compute the cart state
    switch (cartState)
    {
        // Cart is init in the scene, he have to be enabled
        case CART_STATUS.DISABLED:

            Log("Cart is ready!");
            break;

        case CART_STATUS.WAITING:
            // Cart is waiting to start tracking the path
            // Reset the cart speed
            SetCartSpeed(0f);
            cartMoveDirection = 0;

            KEYBOARD TO MOVE

            Log("... Waiting ...");
            break;

        case CART_STATUS.RUNNING:
            // Cart is moving along the path
            // Update the cart speed
            SetCartSpeed(cartSpeed * cartMoveDirection);
            Log("Cart is moving along the path");
            break;

        case CART_STATUS.STOPPED:
            // Cart wait to resume moving
            break;
    }
```

Figura 36 - Gestione degli stati associati al Cart

L'utilizzo degli stati finiti fornisce più controllo sull'esecuzione di codice specifico ma anche nella ricerca di possibili bug.

Lo stato *DISABLED* è utilizzato come stato iniziale del cart, precedente alla sua inizializzazione.

Lo stato *WAITING* rappresenta l'attesa di input da parte dell'utente, il cart ha una velocità nulla.

Lo stato *RUNNING* setta la velocità del cart rispetto alla direzione di movimento.

Lo stato *STOPPED* viene eseguito quando il cart è interrotto durante il movimento e non esce fin quando non riceve il comando di ripresa.

Per far muovere il cart, basta modificarne la velocità. Può accettare anche valori negativi che si traducono in una direzione contraria di movimento. In fase di progettazione è stato deciso di potersi muovere sia verso lo spot successivo che quello precedente, quindi si avrà bisogno di una variabile che memorizzi questa scelta (***CartMoveDirection***) in caso volessimo sospendere il movimento ad un certo istante. Con il metodo *MoveCart()* (Figura 37), la direzione viene passata come parametro (*int direction*) che può assumere 3 valori distinti in base al verso di movimento. Il valore 0 corrisponde al cart fermo, il valore 1 al movimento verso lo spot successivo e il valore -1 al movimento verso lo spot precedente. Vengono recuperate le informazioni sull'oggetto di focus del prossimo spot con *LookNewTarget()* e si porta il cart nello stato di *RUNNING*. L'interfaccia utente verrà avvisata dell'inizio del movimento mostrando solo gli elementi UI necessari. Da notare l'attivazione del flag *"Auto Dolly"* nel caso in cui la direzione sia diversa da zero (*vcamDolly.m_AutoDolly.m_Enabled = direction != 0):* senza questa istruzione la posizione del cart non verrebbe automaticamente aggiornata.

```
// Extract the next point into the path based on direction
// direction == -1 --> take the previous point
// direction == 0  --> stay on the same point
// direction == 1  --> take the next point
public void MoveCart( int direction )
{
    // Return if the cart is not waiting for an input
    if (cartState != CART_STATUS.WAITING || startSpot == null)
        return;

    // Set the direction to move the cart and the LookAt point of the camera
    cartMoveDirection = direction;

    // Update next target to look
    LookNewTarget();

    // The Auto Dolly enable only if the move direction is not 0
    vcamDolly.m_AutoDolly.m_Enabled = direction != 0;
    cartState = CART_STATUS.RUNNING;

    // Disable UI
    CartUI.ShowCommandsUI(false);
    CartUI.ShowPathButton(false);

    Log("... moving ...");
}
```

Figura 37 - Metodo per l'avvio del movimento di camera

Il metodo *MoveCart()* fa parte di quell'insieme di metodi che gestiscono il movimento, la ripartenza, il reset e lo stop del cart. Questi metodi sono stati ideati in modo da poter

essere invocati come effetto dell'evento *OnClick()* della *User Interface*. Analizziamo brevemente i singoli metodi:

- ***StopCart()*** (Figura 38), porta il cart nel suo stato *STOPPED* e ne azzera la velocità. Viene disattivato il flag *"Auto Dolly"* nella Virtual Camera.

```
public void StopCart()
{
    // Return if the cart is already stopped
    if (cartState == CART_STATUS.STOPPED)
        return;

    // Cart stop waiting for next input
    cartState = CART_STATUS.STOPPED;

    // Disable Auto Dolly
    instance.m_Speed = cartSpeed * 0f;
    vcamDolly.m_AutoDolly.m_Enabled = false;

    Log("Cart has been interrupted!");
}
```

Figura 38 - Metodo che ferma il movimento del cart

- ***ResumeCart()*** (Figura 39), riporta il cart nello stato di *RUNNING* settando la sua velocità in base alla direzione di movimenti. Viene attivato il flag *"Auto Dolly"* nella Virtual Camera e il cart riprende la sua corsa.

```
public void ResumeCart()
{
    // Return if the cart is not stopped
    if (cartState != CART_STATUS.STOPPED)
        return;

    // Cart stop waiting for next input
    cartState = CART_STATUS.RUNNING;

    // Disable Auto Dolly
    instance.m_Speed = cartSpeed * cartMoveDirection;
    vcamDolly.m_AutoDolly.m_Enabled = true;

    Log("Cart is resumed");
}
```

Figura 39 - Metodo per riprendere il movimento del cart

- ***ResetCart()*** (Figura 40), il cart passa allo stato di *WAITING*. Il riferimento all'ultimo *TriggerSpot* visitato diventa *null* e si estrae dalla *TriggerList*

l'oggetto di focus del primo spot. Il cart viene riposizionato alla posizione 0 del percorso corrente. La scena si ripresenta come al primo avvio.

```
public void ResetCart()
{
    // Cart stop waiting for next input
    cartState = CART_STATUS.WAITING;

    // Disable Auto Dolly
    vcamDolly.m_AutoDolly.m_Enabled = false;

    lastTriggerVisited = null;
    nextTargetToLook = triggerList[0].GetComponent<TriggerSpot>().GetLookedObject();
    curTransitionSpeed = maxTransitionSpeed*2f;

    // Reset Speed
    cartMoveDirection = 0;
    instance.m_Speed = cartSpeed * cartMoveDirection;
    instance.m_Position = 0f;

    // Reset VCam
    vcam.LookAt = VCamPointInterest;
    vcamDolly.m_PathPosition = 0f;

    Log("Cart has been resetted");
}
```

Figura 40 – Riposizionamento del cart all'inizio del percorso con relativo reset.

♦ Diramazioni e percorsi alternativi - ChangePathUI()

Un'altra funzionalità di cui ancora non si è parlato, è la possibilità di scegliere su quale percorso il cart dovrà spostarsi. Ogni qualvolta il cart collide con il *TriggerSpot*, quest'ultimo gli comunica le informazioni su di sé e nel momento in cui saranno presenti più diramazioni, da quello spot, sull'interfaccia apparirà il pulsante per il *"Cambio Percorso"*. Come per tutti gli altri elementi dell'interfaccia, allo scatenarsi dell'evento *OnClick()* viene invocato il relativo metodo associato a quell'evento. In questo caso il metodo è ***ChangePathUI()*** (Figura 41) che riposiziona il cart sullo spot corrente per arrotondarne il valore e chiama il metodo ***ExtractPath()*** per ottenere il riferimento alla diramazione.

```
// Set a new path from the current triggerSpot
public void ChangePathUI()
{
    // RePosition the cart to the first spot
    instance.m_Position = triggerList[ lastTriggerVisited.GetComponent<TriggerSpot>().GetListPosition() ].
                          GetComponent<TriggerSpot>().GetPointAlongPath();
    vcamDolly.m_PathPosition = instance.m_Position;

    // Extract Path from the triggerSpot
    ExtractPath();
}
```

Figura 41 - Metodo per il cambio percorso invocato dalla UI

Quando l'esecuzione entra nel metodo ***ExtractPath()*** (Figura 42), si "estrae" dall'ultimo *TriggerSpot* visitato la nuova diramazione. Viene disattivato il percorso precedente così da evitare interfogliamenti tra gli spot e viene attivato il percorso nuovo. Come per l'inizializzazione, la stessa *Coroutine* si preoccupa di raccogliere tutte le informazioni necessarie dal nuovo percorso: la lista di *TriggerSpot*, il nome del percorso, etc.

```
// Extract next path from triggerSpot
void ExtractPath()
{
    // Get the new pathTriggerList
    if (lastTriggerVisited != null)
    {
        TriggerSpot lastTriggerSpot = lastTriggerVisited.GetComponent<TriggerSpot>();

        int currentPathIndexInList = lastTriggerSpot.GetPathIndex(curPath);

        if (currentPathIndexInList >= 0)
        {
            // The path exists in the triggerSpot
            // Disable old path
            curPath.gameObject.SetActive(false);

            // Set the new triggerList
            pathTriggerList = lastTriggerSpot.GetNextPath(currentPathIndexInList + 1).gameObject.GetComponent<PathTriggerList>();
            pathTriggerList.gameObject.SetActive(true);

            // Update the path in the cinemachine using the init coroutine to avoid nullExceptions
            StopCoroutine(initRoutine);
            initRoutine = StartCoroutine(Init());
        }
    }
}
```

Figura 42 - Codice per ottenere le informazioni riguardante il nuovo percorso

♦ Collisione con i TriggerSpot – OnTriggerEnter()

Questa sezione rappresenta il cuore dell'intera visita guidata: è qui che il cart e i *TriggerSpot* si scambiano i dati.

Come detto nel paragrafo "*3.4 Punti di Interesse – TriggerSpot"*, ogni spot ha un collider dichiarato come ***trigger***, il che vuol dire che in caso di collisione non funge da oggetto solido ma da "interruttore". Il cart, di conseguenza, è fornito anch'esso di un collider che non è però un ***trigger***, quindi risulta essere un oggetto solido nella scena.

Quando questi due collider si scontrano viene generato un evento[5] di Unity, su entrambi gli oggetti, che si chiama *OnTriggerEnter()* (Figura 43). Questo metodo prende un parametro in input che rappresenta l'altro collider coinvolto nella collisione.

Dato che la collisione avviene per qualsiasi ***trigger***, è necessario definire un modo per poter distinguere gli oggetti con cui si collide: Unity fornisce un sistema di tag come soluzione al problema.

[5] Quando due collider definiti come ***trigger*** si scontrano, non viene generato nessun evento a meno che uno dei due oggetti non abbia la componente *Rigidbody impostata come Kinematic*.

```
private void OnTriggerEnter(Collider other)
{
    if (other.CompareTag("TriggerSpot"))
    {
        // Store the first spot encountered
        /*if (startSpot == null)
        {
            startSpot = other.gameObject;
        }*/

        // Check if the trigger was the same
        if (other.gameObject != lastTriggerVisited)
            lastTriggerVisited = other.gameObject;
        else
        {
            if (other.gameObject != startSpot.gameObject)
                return;
        }

        // If the cart is not ready yet ...
        if (cartState == CART_STATUS.DISABLED)
            return;

        TriggerSpot ts = lastTriggerVisited.GetComponent<TriggerSpot>();

        // Check if the spot is a StaySpot
        if ( (ts.HaveToStop() && !dontStopOnSpot) || (lastTriggerVisited == startSpot) )
        {
            // Cart stop waiting for next input
            cartState = CART_STATUS.WAITING;

            // Disable Auto Dolly
            vcamDolly.m_AutoDolly.m_Enabled = false;

            // Enable UI
            CartUI.ShowCommandsUI(true);

            // Check if have to show the change path too
            CartUI.ShowPathButton(ts.HasOtherPaths());

            // Try to align with TriggerSpot
            Log("TriggerSpot reached. The Cart is stopped!");
        }
        else
        {
            // Update next target to look
            LookNewTarget();
        }

        curTransitionSpeed = Mathf.Clamp(ts.GetTransitionSpeed(),0.1f, maxTransitionSpeed);
    }
}
```

Figura 43 - Gestione della collisione tra il cart e il TriggerSpot

Durante l'esecuzione del metodo, la prima cosa è controllare se il tag del collider coincide con quello associato ai *TriggerSpot*. Come ulteriore controllo ci assicuriamo che il *TriggerSpot* non sia l'ultimo appena incontrato per poi memorizzarne il riferimento che verrà utilizzato dagli altri metodi del cart. Quest'ultimo controlla se il *TriggerSpot* è un punto di interesse o un punto solo di passaggio invocando il metodo sullo spot ***ts.HaveToStop()***. In caso positivo il cart passa in uno stato di *WAITING* disattivando l'*Auto Dolly* e avvisando la *User Interface* di mostrare i comandi a schermo per l'utente (*nell'eventualità ci fossero diramazioni, viene mostrato anche il tasto per il cambio percorso*). In caso negativo il cart estrae dal *TriggerSpot* le informazioni sull'oggetto di focus, orientando la camera sulla base della velocità di transizione dello spot proseguendo la sua corsa.

3.7. Interfaccia utente – Cart UI

Per far si che l'utente possa interagire con la visita guidata, è stata progettata una interfaccia (Figura 44) composta principalmente da soli pulsanti.

Figura 44 - Interfaccia utente con i pulsanti per il movimento della Camera

Nello specifico sono presenti:

- ***Spot precedente / Spot successivo*** che avviano il movimento della camera rispettivamente per raggiungere lo spot precedente o successivo a quello attuale.

- ***Torna all'inizio*** riporta il cart al punto iniziale del percorso attualmente selezionato.

- ***Cambia Percorso*** consente di variare la visita guidata nel caso in cui, sul *TriggerSpot* dove ci troviamo, sono stati definiti più percorsi (condizione necessaria per la visibilità del pulsante). Il nome del percorso viene visualizzato nel testo soprastante.

- ***PAUSA / RIPRENDI*** sono visibili solo durante la transizione tra uno spot e l'altro e consentono di fermare o riprendere il movimento.

Di seguito due esempi di come appare l'interfaccia quando l'utente è fermo (Figura 45) oppure in movimento (Figura 46).

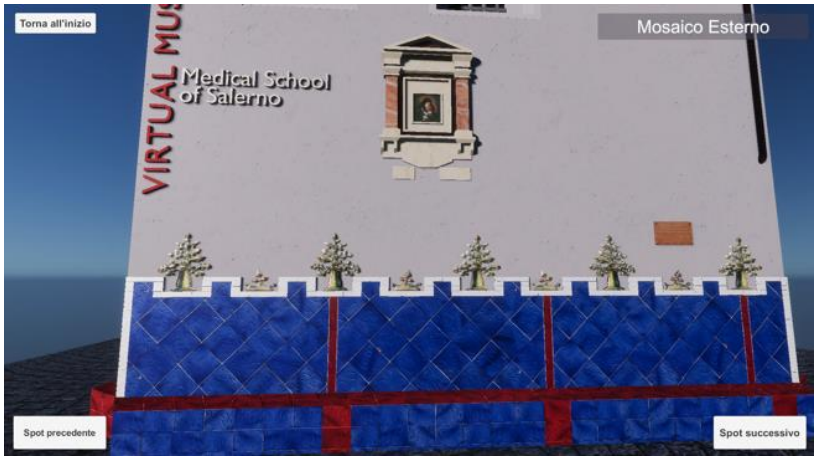

Figura 45 - Interfaccia da fermo

Figura 46 - Interfaccia in movimento

I pulsanti dell'interfaccia sono stati implementati con la componente *Button* (Figura 47) di Unity, appositamente progettata per le *User Interfaces*. L'unica variante tra un pulsante e l'altro è la gestione dell'evento *OnClick()* che prende il riferimento allo script ***CartPlayer***, per poi invocare il metodo definito nell'evento.
Nella figura è visibile la componente *Button* del pulsante per lo spot successivo, il cui metodo che andrà ad invocare, qualora l'utente prema il pulsante, è ***MoveCart()*** con il parametro 1[6].

[6] La descrizione del metodo ***MoveCart()*** dello script CartPlayer è stata effettuata nel paragrafo precedente.

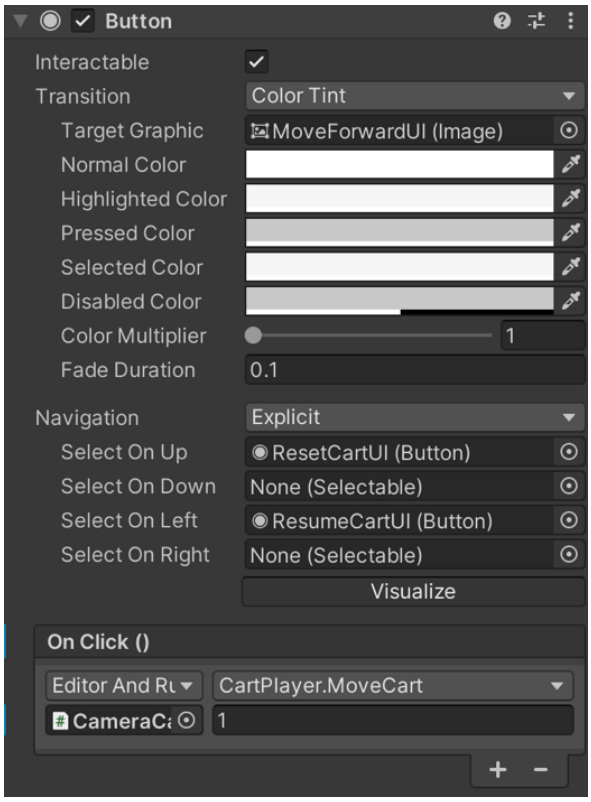

Figura 47 - Componente Button fornita da Unity per la User Interface

L'idea minimalista nel design dell'interfaccia è stata scelta per far concentrare l'utente sul resto della scena. Il posizionamento agli angoli e la presenza solo dei pulsanti veramente necessari, ci consentono di non creare distrazioni durante l'esperienza mantenendo il punto focale al centro dello schermo. L'interfaccia utente è stata strutturata con un *GameObject* (***CartPlayerUI***) che implementa la componente *Canvas* di Unity e funge da root per tutti i pulsanti dell'interfaccia (Figura 48). Questo *GameObject* implementa anche lo script ***CartUIManage*** (Figura 49) per la gestione della visibilità degli elementi in base allo stato del cart.

Figura 48 - Struttura del Canvas e relativi pulsanti

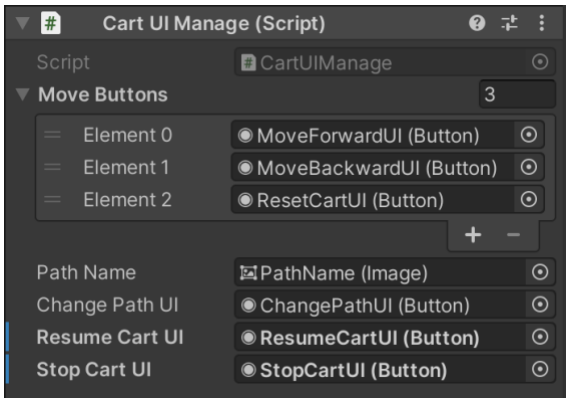

Figura 49 - Script per la gestione dell'interfaccia utente

Lo script prende come parametri i riferimenti di tutti gli elementi dell'interfaccia. Viene definita una lista *MoveButtons* per separare quegli elementi che devono essere resi invisibili durante i movimenti di camera. Ciò viene fatto attraverso dei metodi (Figura 50, Figura 51, Figura 52) che attivano / disattivano gli elementi specifici. Ognuno di questi metodi prende in input un parametro booleano che verrà utilizzato per settare la visibilità dei singoli elementi. Come visto nel paragrafo precedente, l'invocazione di questi metodi è gestita dallo script ***CartPlayer***.

```
// Show or Hide the UI movement based on value
public void ShowCommandsUI(bool value)
{
    foreach(Button bUI in moveButtons)
    {
        bUI.gameObject.SetActive(value);
    }

    ShoWPlayerCommand(!value);
}
```

Figura 50 - Metodo per controllare la visibilità durante il movimento

```
// Show Cart Player Commands
public void ShoWPlayerCommand(bool value)
{
    ResumeCartUI.gameObject.SetActive(value);
    StopCartUI.gameObject.SetActive(value);
}
}
```

Figura 51 - Metodo che gestisce la visibilità solo dei tasti PAUSA / RIPRENDI

```
// Show the path button
public void ShowPathButton(bool value)
{
    changePathUI.gameObject.SetActive(value);
}

// Show or Hide the path name
public void ShowPathName(bool value)
{
    pathName.gameObject.SetActive(value);
}
```

Figura 52 - Codice per mostrare il cambio percorso o il nome del percorso

CAPITOLO 4

CONCLUSIONI E SVILUPPI FUTURI

Durante questo lavoro di tesi, ci si è concentrati principalmente sull'esperienza che l'utente avrebbe vissuto durante la visita guidata. Nel particolare, le attenzioni sono andate al movimento di camera e al suo orientamento verso oggetti di focus. Come strumento, Unity ha fornito una base più che soddisfacente per la fase di progettazione e testing. Il contributo maggiore è arrivato dal *Cinemachine Package* che ha consentito un setup immediato del percorso guidato e delle funzionalità legate alla camera. Una volta implementati i singoli elementi si è passato all'integrazione di questo percorso guidato all'interno della scena 3D, preparata per la visita del museo. Successivamente, fatte le dovute correzioni e modifiche, la visita guidata è stata testata in tutta la sua completezza tenendo nota delle possibili migliorie future. L'intero lavoro è stato effettuato in maniera puramente disciplinare e può essere tranquillamente esteso a più approcci in ambito culturale e non.

Alcuni dei possibili sviluppi futuri sono: gli spot di interesse che possono fornire ancora più informazioni di quelle attuali aumentando la dinamicità del percorso, la *User Interface* che può implementare nuovi comandi per gli utenti, i parametri della camera che possono essere personalizzati e tanti altri.

Il *Cinemachine Package* è un pacchetto in continua evoluzione che offrirà strumenti man mano sempre più vicini a quelli in ambito cinematografico.

BIBLIOGRAFIA

[1] Autori di Wikipedia, *"Computer Grafica"*, Wikipedia, L'Enciclopedia libera, (**19 Giugno 2021**), https://it.wikipedia.org/w/index.php?title=Computer_grafica&oldid=121392758 (controllata il 23 Giugno 2021).

[2] David Salomon, *The Computer Graphics Manual*, Springer Science & Business Media (**18/09/2011**).

[3] Autori di Wikipedia, *"Rendering"*, Wikipedia, L'Enciclopedia libera, (**1 Giugno 2021**), https://it.wikipedia.org/w/index.php?title=Rendering&oldid=121011653 (controllata il 23 Giugno 2021).

[4] Autori di Wikipedia, *"Virtual Camera System"*, Wikipedia, L'Enciclopedia libera, (**28 Febbraio 2021**), https://en.wikipedia.org/w/index.php?title=Virtual_camera_system&oldid=1009357660 (controllata il 27 Giugno 2021).

[5] Resident Evil 2, Capcom, *"A selection of views from Resident Evil 2 to illustrate how Capcom created tension through the use of camerawork"*, Wikipedia, L'Enciclopedia libera, (**23 Maggio 2017**), https://en.wikipedia.org/w/index.php?title=File:Resident-evil-2-camerawork.jpg&oldid=781752430 (controllata il 27 Giugno 2021).

[6] Rollings, Andrew; Ernest Adams (**2006**). *Fundamentals of Game Design*.

[7] Crash Bandicoot 2: Cortex strikes back, Naughty Dogs, *"Screenshot from Crash Bandicoot 2 to illustrate a standard tracking camera system"*, https://www.gamesdatabase.org/game/sony-playstation/crash-bandicoot-2-cortex-strikes-back (controllata il 27 Giugno 2021).

[8] Super Mario 64, Nintendo, *"Three Screenshots from Super Mario 64 to illustrate the camera system artificial intelligence"*, Wikipedia, L'Enciclopedia libera, (**28 Agosto 2020**), https://en.wikipedia.org/w/index.php?title=File:Super-mario-64-camera-system-ai.jpg&oldid=974007922 (controllata il 28 Giugno 2021).

[9] He, Li-wei; Michael F. Cohen; David H. Salesin (**1996**). "The Virtual Cinematographer: A Paradigm for Automatic Real-Time Camera Control and Directing". *International Conference on Computer Graphics and Interactive Techniques*. New York. **23rd**: 217–224

[10] Virtual Camera System, WikiLaurent, *"Virtual Camera System demo showing parameters that can be adjusted, such as the field of view, rotation, distance to the camera, etc."*, Wikipedia, L'Enciclopedia libera, (**3 Luglio 2009**), https://commons.wikimedia.org/w/index.php?title=File:Virtual-camera-system.png&oldid=475851475 (controllata il 27 Giugno 2021).

[11] Bares, William; Scott McDermott; Christina Boudreaux; Somying Thainimit (**2000**). "Virtual 3D camera composition from frame constraints" . *International Multimedia Conference*. California, United States: Marina del Rey: 177–186.

[12] Drucker, Steven M.; David Zeltzer (**1995**). *CamDroid: A System for Implementing Intelligent Camera Control*. *Symposium on Interactive 3D Graphics*.

[13] Tomlinson, Bill; Bruce Blumberg; Delphine Nain (**2000**). *Expressive Autonomous Cinematography for Interactive Virtual Environments*. *Proceedings of the Fourth International Conference on Autonomous Agents*.

[14] Hancock, Hugh (**April 2, 2002**). "Better Game Design Through Cutscenes". Gamasutra.

[15] Aaron, Marcus (**2014**). *Design, User Experience, and Usability. User Experience Design for Diverse Interaction Platforms and Environments*, Springer. p. 662.

[16] Autori di Wikipedia, *"Cutscenes"*, Wikipedia, L'Enciclopedia libera, (**19 Maggio 2021**), https://en.wikipedia.org/w/index.php?title=Cutscene&oldid=1024029980 (controllata il 29 Giugno 2021).

[17] "The Next Generation 1996 Lexicon A to Z: Cut Scene". *Next Generation*. No. 15. **March 1996**. p. 32.

[18] *Sito Ufficiale*, https://unity.com.

[19] *Documentazione del Camera Component di Unity,* https://docs.unity3d.com/Manual/class-Camera.html

[20] *Cinemachine Package Overview,* https://unity.com/unity/features/editor/art-and-design/cinemachine

[21] *Documentazione del Cinemachine Package di Unity,* https://docs.unity3d.com/Packages/com.unity.cinemachine@2.3/manual/index.html

[22] *"Logo del Museo Virtuale della Scuola Medica Salernitana",* http://www.museovirtualescuolamedicasalernitana.beniculturali.it/it/

RINGRAZIAMENTI

Vorrei ringraziare i miei due relatori di tesi

Andrea Francesco Abate & Ignazio Passero,

per avermi guidato durante questo periodo di stesura

e per avermi aiutato a consolidare le mie passioni.

Ringrazio anche la mia famiglia che mi ha sempre

spronato a dare il massimo e raggiungere i miei obiettivi.

Per ultima ma non meno importante ringrazio

la mia fidanzata e compagna di vita Daria:

che la tua luce possa essere per sempre fonte della mia forza.