UNIVERSITÀ DEGLI STUDI DI NAPOLI FEDERICO II

Dipartimento di Ingegneria Elettronica e delle Tecnologie dell'Informazione

Classe di Laurea in Ingegneria Elettronica, Classe n. L.8

Corso di Laurea in Ingegneria Elettronica

Elaborato di Laurea

Sviluppo di applicazioni in Virtual Reality con Unreal Engine

Relatore:
Ch.mo Prof. Porfirio Tramontana

Candidato:
Francesco Sorrentino
Matr. N43000972

Anno Accademico
2020/2021

"Remember Who You Are"

Indice

Introduzione

L'obiettivo di questa tesi, come possiamo leggere anche dal titolo, è lo sviluppo di applicazioni in Virtual Reality con Unreal Engine. Nel primo capitolo andremo a spiegare cos'è la realtà aumentata, cos'è la realtà virtuale e studieremo le loro differenze. Successivamente, dopo aver fatto questa introduzione, ci immergeremo nel mondo di Unreal Engine. Nel secondo capitolo inizieremo a spiegare cos'è questo motore, a cosa serve, le sue funzionalità e mostreremo, attraverso un percorso storico, lo sviluppo di Unreal Engine fino ad oggi. Nel terzo capitolo creeremo un progetto fatto in Blueprint con Unreal Engine 4. Partiremo dall'inizializzazione della schermata iniziale per poi vedere come si crea un nuovo livello e ambiente di gioco. Chiuderemo il capitolo simulando il nostro progetto grafico. Nel quarto capitolo parleremo della programmazione in C++ di Unreal Engine andando a vedere quelle che sono le nozioni basi e tutta la parte introduttiva alla programmazione. Infine, nel quinto ed ultimo capitolo, realizzeremo un ulteriore progetto con UE4 andando a combinare una parte grafica ed una della programmazione. Studieremo, dunque, delle interazioni tra Blueprint e C++.

Capitolo 1

Realtà Virtuale e Realtà Aumentata

1.1 Realtà Virtuale

Probabilmente tutti abbiamo sentito parlare di Realtà Virtuale, Virtual Reality o di VR. La Realtà Virtuale, per sua stessa definizione, simula la realtà effettiva. L'avanzamento delle tecnologie informatiche permette di navigare in ambientazioni fotorealistiche in tempo reale, interagendo con gli oggetti presenti in esse. Anche se, a livello teorico, la realtà virtuale potrebbe essere costituita attraverso un sistema totalmente immersivo in cui tutti i sensi umani possono essere utilizzati, attualmente il termine è applicato solitamente a qualsiasi tipo di simulazione virtuale creata attraverso l'uso del computer, dai videogiochi che vengono utilizzati su un normale schermo, alle applicazioni che richiedono l'uso degli appositi guani muniti di sensori. La realtà virtuale immersiva (un ambiente costruito intorno all'utente) secondo il livello tecnologico attuale e secondo le previsioni possibili per il prossimo futuro potrà essere utilizzata dalla massa grazie ad alcune periferiche (in parte già utilizzate):

- **Visore**: un casco o dei semplici occhiali in cui schermi vicini agli occhi annullano il mondo reale dalla visuale dell'utente. Il visore può inoltre contenere dei sistemi per la rilevazione dei movimenti, in modo che girando la testa da un lato, ad esempio, si ottenga la stessa azione anche nell'ambiente virtuale.

- **Wire Gloves (guanti)**: questi rimpiazzano mouse, tastiera, joystick, trackball e altri sistemi manuali di input. Possono essere utilizzati per i movimenti, per impartire comandi, digitare su tastiere virtuali
- **Cybertuta**: una tuta che avvolge il corpo e che può simulare il tatto flettendo su sé stessa grazie al tessuto elastico, può realizzare una scansione tridimensionale del corpo dell'utente e trasferirla nell'ambiente virtuale.

La differenza con una realtà virtuale non immersiva consiste nel fatto che in quest'ultimo caso non si fa uso di caschi, ma l'utente si troverà semplicemente dinanzi ad un monitor, il quale fungerà da finestra sul mondo tridimensionale con cui l'utente potrà interagire attraverso joystick appositi.

1.2 Applicazioni della Realtà Virtuale

La VR avrà in futuro impatti dirompenti in numerosi settori: dalla medicina, al retail, dall'architettura all'arte. Vediamo ora nel dettaglio alcuni settori in cui la VR potrebbe trovare ampio spazio in futuro, andando a risolvere problematiche ed esigenze diffuse.

- **VR e il Management Aziendale**: molte aziende stanno iniziando a adottare la realtà virtuale all'interno dei propri ecosistemi in diversi modi. Aziende che producono prodotti pericolosi o prodotti ancora allo stato prototipale usano inoltre la VR per testare la sicurezza e la funzionalità senza mettere a rischio la salute dei proprio dipendenti.
- **Arte**: gli ambiti di utilizzo sono molteplici e vanno dall'accompagnamento alla vista museale vera e propria in cui l'utente può scorgere dettagli e visualizzare opere magari non presenti all'interno della mostra per motivi o esigenze di

trasporto, alla sostituzione della vista vera e propria aiutando quelle persone che, per via di difficoltà motorie, non possono recarsi e godere fisicamente della fruizione museale.

- **Aiuto per i Giovani Chirurghi**: la VR fornisce ai giovani studenti di medicina un ambiente sicuro e comodo per fare pratica con le procedure mediche, permettendogli di fare errori senza avere nessun impatto su pazienti reali e preparandoli ad affrontare anche scenari imprevisti.
- **VR e il settore Automotive**: le aziende utilizzano la VR per fare test di guida, come uno step precedente ai test reali, andando a ridurre i rischi.
- **Aiuto per chi soffre di Autismo**: la tecnica è quella di inserire bambini, adolescenti o giovani affetti da autismo in ambienti in cui sono richieste ampie doti di socializzazione e la capacità di saper stare a contatto e interagire con gli altri. In questo modo possono esercitarsi ed imparare come comportarsi in pubblico, come affrontare determinate situazioni.
- **Edilizia**: la VR fungerà presto da valore aggiunto per architetti e designer nella fase di progettazione di nuove soluzioni. Questo strumento è infatti molto prezioso in quanto permette di camminare e muoversi virtualmente in spazi tridimensionali. Immagini generate al computer andranno infatti a sostituire rendering e disegni fatti a mano, molto meno precisi e realistici, andando a ridurre anche il tempo necessario all'elaborazione dei layout e dei modelli.

1.3 Realtà Aumentata

A differenza della realtà virtuale, che isola totalmente l'utente durante la sua esperienza immersiva proiettandolo in una dimensione del tutto alternativa, la Realtà Aumentata si distingue infatti per proiettare dei contenuti in 3D sulla visione del mondo reale, della realtà che ci circonda. Anziché isolarci per vivere una situazione del tutto alternativa al vero, la realtà aumentata mira piuttosto ad "aumentare" la realtà con una serie di informazioni contestuali, con cui gli utenti possono facilmente operare grazie alla loro esatta corrispondenza spaziale con gli elementi reali con cui sono abituati ad interagire. La realtà aumentata non ci propone un ambiente differente rispetto a quello a noi familiare, ci mantiene a stretto contatto con la realtà che ci circonda, aggiungendo dei layer informativi in 3D per abilitare un range di funzioni potenzialmente infinito. I principali formati tecnologici della realtà aumentata prevedono un output su due tipologie di dispositivi informatici: Mobile (smartphone/tablet) e visori dedicati. Per quanto riguarda i dispositivi Mobile, si tratta del formato di AR ampiamente più diffuso al momento, con risultati sorprendenti, se si considera il successo delle applicazioni più note, la crescente diffusione generale ed il fatto che la AR si una tecnologia a tutti gli effetti giovanissima.

1.3 Differenza tra VR e AR

Dopo aver spiegato cos'è la Realtà Aumentata e cos'è la Realtà Virtuale possiamo, in poche parole, spiegare la differenza tra le due: mentre la realtà virtuale ci porta in un ambiente totalmente artificiale, la realtà aumentata sovrappone all'ambiente reale elementi digitali. In entrambe l'utente può interagire, ma nella prima vengono ingannati i sensi, nella seconda vengono aggiunte informazioni a ciò che i sensi possono percepire.

Capitolo 2

Unreal Engine

2.1 Cos'è Unreal Engine

Unreal Engine è un motore di gioco sviluppato da Epic Games, grande società americana di videogiochi fondata da Tim Sweeney nel 1991, presentato per la prima volta nel 1998 e già da allora compatibile con diversi sistemi operativi quali Microsoft Windows, Linux e MacOs. Nel corso degli anni lo sviluppo del software ha raggiunto alti livelli prestazionali, adattandosi alle potenzialità degli hardware disponibili nel mercato e comunicando con diverse piattaforme. Unreal è il motore grafico del futuro, lo strumento di creazione 3D in tempo reale più aperto ed avanzato al mondo, in continua evoluzione per conseguire non solo il suo scopo originario di motore di gioco all'avanguardia, ma anche per offrire oggi ai creatori di tutti i settori la libertà ed il controllo del proprio progetto offrendo contenuti di nicchia, esperienze interattive e mondi virtuali coinvolgenti.

2.2 Storia

2.2.1 Unreal Engine 1

L'Unreal Engine di prima generazione è stato sviluppato da Tim Sweeney. Egli iniziò a scrivere il motore nel 1995 per la produzione di un gioco che sarebbe poi diventato uno sparatutto in prima persona noto come *irreale*. Dopo anni di sviluppo, ha debuttato con l'uscita del gioco nel 1998. All'inizio, il motore si basava completamente

sul rendering software, il che significa che i calcoli grafici erano gestiti dalla CPU. Tuttavia, nel tempo, è stato in grado di sfruttare le capacità fornite dalle schede grafiche dedicate, concentrandosi sull'API Glide, appositamente progettata per gli acceleratori 3dfx. Tra le sue caratteristiche c'erano il rilevamento delle collisioni, l'illuminazione colorata e una forma limitata di filtraggio delle texture. Il motore integrava anche un editor di livelli, UnrealEd, che supportava le operazioni di geometria solida costruttiva in tempo reale già nel 1996, consentendo ai mappatori di modificare al volo il layout del livello.

2.2.2 Unreal Engine 2

Con lo sviluppo iniziato un anno dopo, la seconda versione ha fatto il suo debutto nel 2002 con *America's Army*, uno sparatutto multiplayer gratuito sviluppato dall'esercito americano come dispositivo di reclutamento. Poco dopo, epica libererebbe *Unreal Championship* sulla Xbox, con essendo uno dei primi giochi ad utilizzare Microsoft Xbox Live. Sebbene basata sul suo predecessore, questa generazione ha visto un notevole progresso in termini di rendering e nuovi miglioramenti al set di strumenti. In grado di eseguire livelli quasi cento volte più dettagliati di quelli trovati in *Unreal*, il motore integrava una varietà di funzionalità.

2.2.3 Unreal Engine 3

Gli screenshot di Unreal Engine 3 sono stati presentati nel luglio 2004. Il motore era basato sulla prima generazione, ma conteneva nuove funzionalità. Tutti i calcoli di illuminazione e ombreggiatura sono stati eseguiti per pixel, anziché per vertice.

Sweeney ha affermato:

"Le decisioni architetturali di base visibili ai programmatori di un design orientato agli oggetti, un approccio di scripting basato sui dati e un approccio abbastanza modulare ai sottosistemi rimangono ancora [da Unreal Engine 1]. Ma le parti del gioco che sono davvero visibili ai giocatori –il renderer, il sistema fisico, il sistema audio e gli strumenti– sono tutti visibilmente nuovi e drammaticamente più potenti".

2.2.4 Unreal Engine 4

Il 19 marzo 2014 Epic Games ha rilasciato Unreal Engine 4. Una delle principali funzionalità pianificate per UE4 era l'illuminazione globale in tempo reale utilizzando la traccia del cono voxel, eliminando l'illuminazione pre-calcolata. Tuttavia, questa funzione, chiamata Sparse Voxel Octree Global Illumination (SVOGI) e mostrata con la demo *Elemental*, è stata sostituita con un algoritmo simile ma meno costoso dal punto di vista computazionale a causa di problemi di prestazioni. UE4 include anche il nuovo sistema di scripting visivo "Blueprints" che consente un rapido sviluppo della logica di gioco senza utilizzare il codice, con conseguente minore divisione tra artisti tecnici, designer e programmatori.

2.3 Approcci e Funzionalità

Le funzionalità di Unreal Engine sono molteplici:

- Serve per creare delle esperienze con le quali si può interagire con il mondo circostante.
- È interattivo: si può interagire con gli oggetti all'interno.
- Può essere utilizzato come intrattenimento: oltre al videogioco si può creare anche un paesaggio o un mondo fantastico.
- È utile per la realizzazione di mondi in 3D.
- Può essere utilizzato per programmare: in un videogioco non ci sono solo le immagini ma un backstage dove vengono programmate le azioni, reazioni, eventi che accadranno.

Inoltre, l'approccio a questo programma può essere fatto in due modi:

- Possiamo lavorare come Designer nella modalità "Blueprint" occupandoci delle creazioni di livelli, della modellazione, della grafica, degli oggetti in 3D, dell'ambientazione. In questa modalità c'è ugualmente una programmazione più visiva anziché sul codice. I pezzetti di programmazione vengono chiamati "Blueprints".
- Possiamo lavorare come Programmatori con C++ occupandoci di tutti gli aspetti tecnici e funzionamenti (punteggio, passaggio dei livelli, ...), del codice, della scrittura degli eventi o azioni.

Capitolo 3

Progetto in Blueprint con UE4

3.1 Inizio Progetto

Quello che facciamo ora è un esempio per realizzare un progetto in Blueprint con UE4. Come prima cosa apriamo Epic Games, clicchiamo su Launch ed aspettiamo che l'Editor si avvii. Possiamo notare che si apre la finestra di benvenuto che utilizzeremo per lavorare. Utilizzando la modalità Blueprint possiamo selezionare un progetto già esistente oppure crearne uno nuovo. Per la creazione di un nuovo progetto bisogna selezionare una categoria con la quale vogliamo lavorare, noi scegliamo "Games".

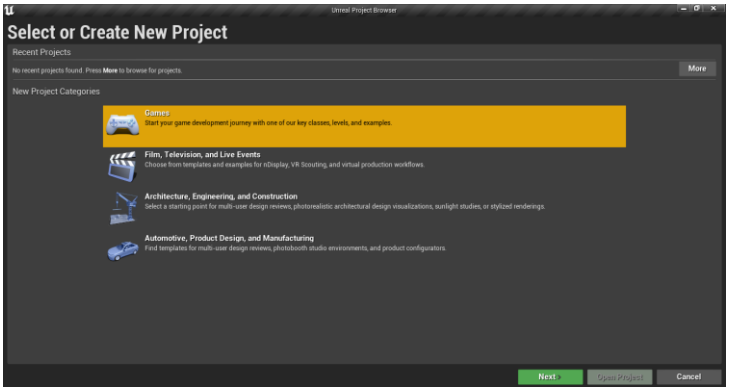


Figura 3.1 - Seleziona o crea un nuovo progetto

Successivamente abbiamo la possibilità di scegliere tra vari Template, ovvero dei livelli di gioco già preimpostati che possiamo utilizzare per non partire da zero. Siccome noi per la realizzazione del progetto vogliamo partire da zero, dobbiamo selezionare il modello vuoto “Blank”.

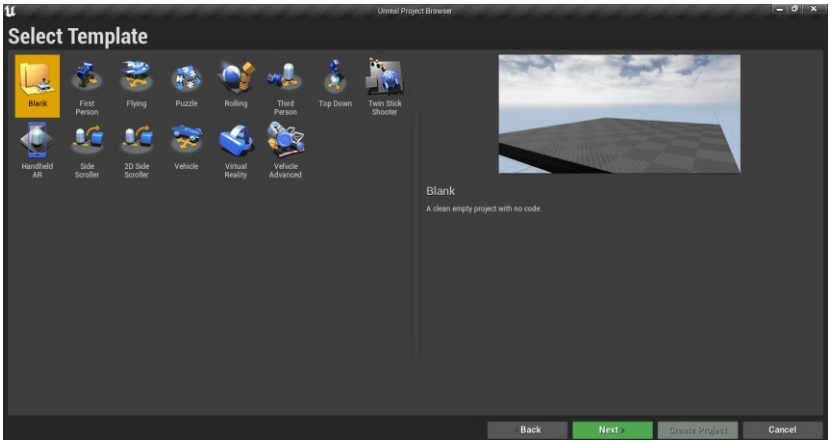


Figura 3.2 - Seleziona Template

Altra cosa importante da fare è scegliere le impostazioni iniziali del progetto. Decidiamo dunque se riprodurre il videogioco o l’applicazione in 3D sul “Desktop” oppure su “Mobile”; quale tipo di qualità possiamo utilizzare all’interno del progetto “Maximum Quality” o “Scalable 3D or 2D”; se partire con pacchetti base oppure senza niente. Infine, dobbiamo scegliere dove il progetto viene salvato, dargli un nome e premere se “Create Project”.

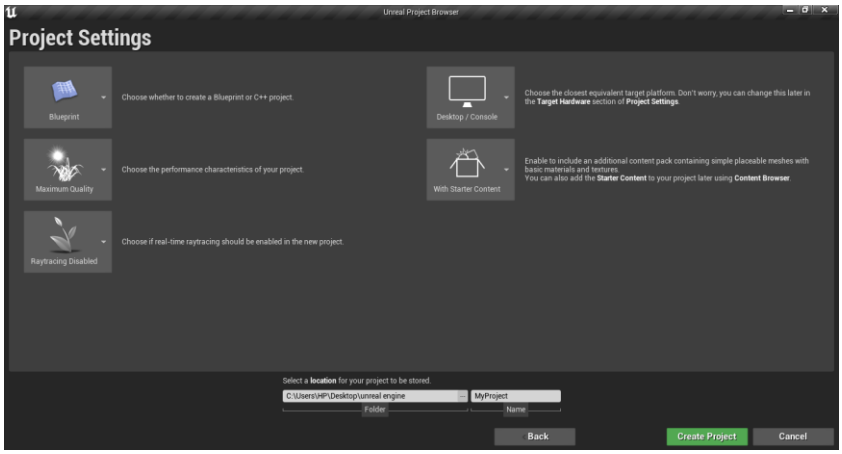


Figura 3.3 - Impostazioni progetto

3.2 Schermata Iniziale e Navigazione nel Viewport

Abbiamo creato il nostro progetto ed ora, aperta l'interfaccia dell'Editor, ci troviamo all'interno del software Unreal Engine. Nella schermata iniziale troveremo un livello base dello starter content di Unreal. La prima cosa che notiamo è il Viewport al centro dell'Unreal Editor. I Viewport sono la finestra nella quale faremo la maggior parte della costruzione del nostro livello. Possono essere navigati proprio come si farebbe in un gioco o possono essere utilizzati nella progettazione più schematica come faremo noi per un Blueprint architettonico.

Figura 3.4 - Schermata iniziale

Il progetto modello selezionato nel passaggio precedente include un piccolo livello di esempio e alcune risorse per iniziare. Usando questa piccola area come punto di riferimento, iniziamo a provare i controlli della telecamera Viewport. Nella Figura (numero) troviamo i controlli standard che rappresentano il comportamento predefinito quando si fa clic e si trascinano nella finestra senza premere altri tasti o pulsanti. Questi sono anche gli unici controlli che possono essere utilizzati per spostarsi tra le Viewport ortografiche. Nella figura (numero) abbiamo i controlli Fly WASD che sono validi solo in un Viewport Perspective e, per impostazione predefinita, è necessario tenere premuto "RMB" per utilizzare i controlli in stile gioco WASD.

Commentato [PT1]: Per WASD si intende gioco guidato dalla tastiera o è un acronimo?

Controllo	Azione
Prospettiva	
LMB + Trascinamento	Sposta la fotocamera avanti e indietro e ruota a sinistra e a destra.
RMB + Trascina	Ruota la telecamera viewport.
LMB + RMB + Trascina	Si muove su e giù.
Ortografica (superiore, anteriore, laterale)	
LMB + Trascinamento	Crea una casella di selezione del perimetro di selezione.
RMB + Trascina	Eseglie la telecamera del viewport.
LMB + RMB + Trascina	Esegue lo zoom avanti e indietro della telecamera viewport.

Figura 3.5 - Controlli standard

Controllo	Azione
W / Numpad8 / Su	Sposta la fotocamera in avanti.
S / Numpad2 / Giù	Sposta la fotocamera all'indietro.
A / Numpad4 / Sinistra	Sposta la fotocamera a sinistra.
D / Numpad6 / Destra	Sposta la fotocamera verso destra.
E / Numpad9 / Pagina su	Sposta la fotocamera verso l'alto.
Q / Numpad7 / Pagina Dn	Sposta la fotocamera verso il basso.
Z / Numpad1	Riduce lo zoom della fotocamera (aumenta il FOV).
C / Numpad3	Ingrandisce la fotocamera (abbassa foV).

Figura 3.6 - Controlli fly wasd

3.3 New Level

Successivamente creiamo un nuovo livello per costruire il nostro ambiente di gioco. All'interno di Unreal Editor, clicchiamo sull'opzione "Menu File" e selezioniamo "New Level". Si aprirà la finestra del Nuovo Livello che include:

- Livello Predefinito: include le risorse più utilizzate per la creazione di livelli
- Livello VR-Basic: include le risorse per la costruzione di livelli con l'Editor VR
- Livello Vuoto: è completamente vuoto, senza risorse

Selezioniamo il livello vuoto e così abbiamo ottenuto un nuovo livello nel quale poter lavorare per il nostro progetto.

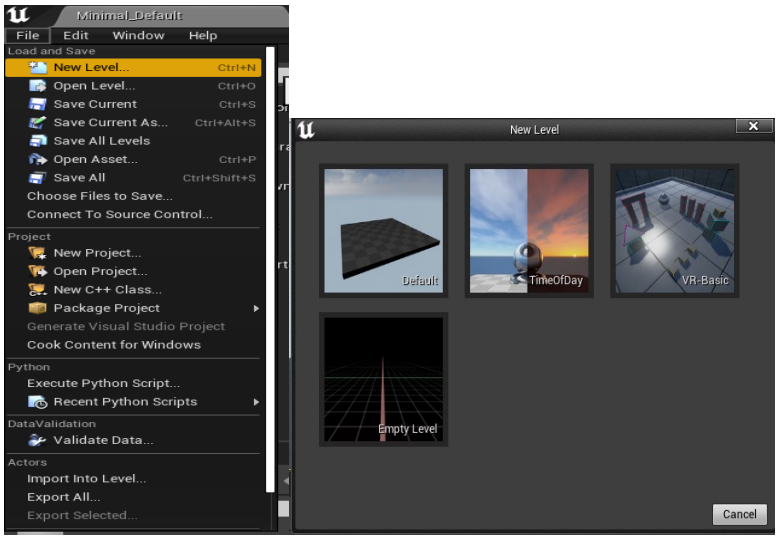


Figura 3.7 - Creazione nuovo livello

3.4 Inserimento Attori e Riproduzione del Livello

Una volta creato un nuovo livello vuoto, possiamo occuparci del posizionamento degli attori con la finalità di dare vita ad un nostro ambiente. A sinistra, nel pannello Posiziona Attori, clicchiamo su “Geometry” e selezioniamo “Box”. Trasciniamo, col il tasto sinistro del mouse, la casella nel riquadro di visualizzazione livello. Impostiamo, nel pannello “Dettagli” (in basso a destra), la scala con la casella ancora selezionata:

1) Posizione e Rotazione su 0
2) Scala su 4×4×0.1

Avremo così un pavimento sul quale il nostro giocatore può volare:

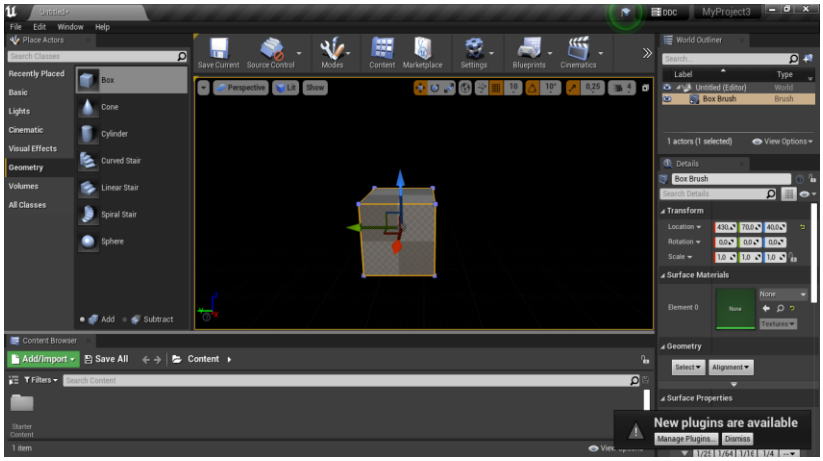


Figura 3.8 - Casella aggiunta al livello

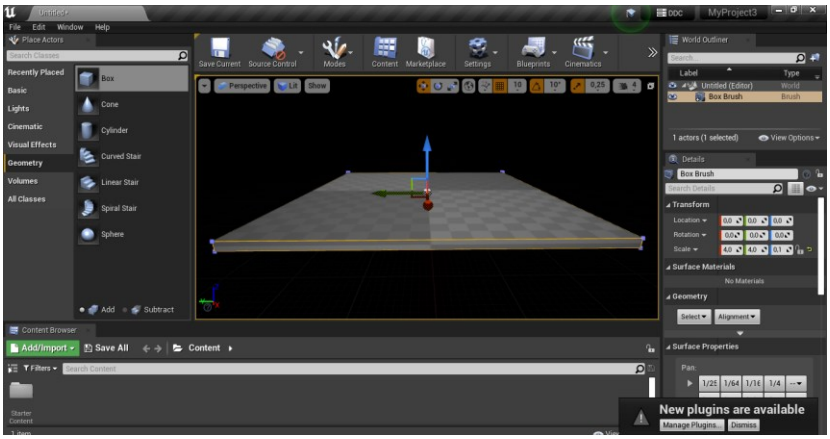


Figura 3.9 - Impostazioni scala

Nel pannello Posiziona Attori, clicchiamo su “Lights”, selezioniamo “Directional Light” e trasciniamola nel livello superiore del pavimento. Successivamente allontaniamo verso l’alto la luce per avere l’illuminazione del pavimento. Sempre nello stesso pannello andiamo nella sezione “Visual Effects” e selezioniamo “Atmospheric Fog” per avere l'attore della nebbia atmosferica, il quale aggiungerà un cielo di base al livello e questo diventerà illuminato anziché scuro. Aggiungiamo poi anche un “PlayerStart”.

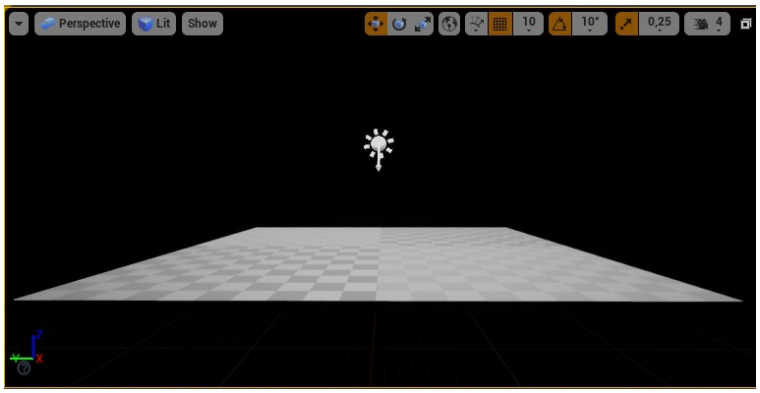


Figura 3.10 - Inserimento luce direzionale

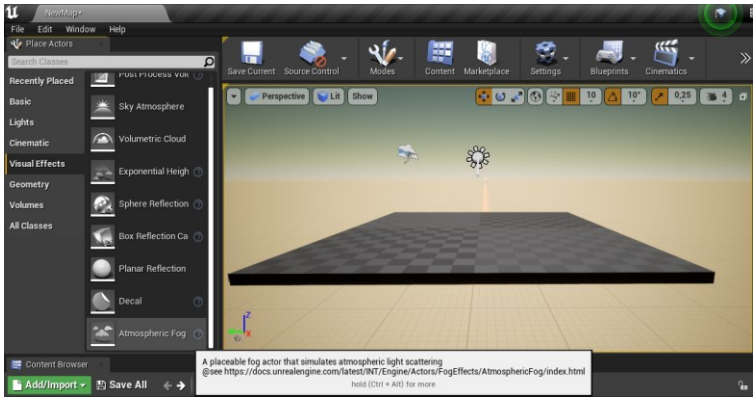


Figura 3.11 - Nebbia atmosferica

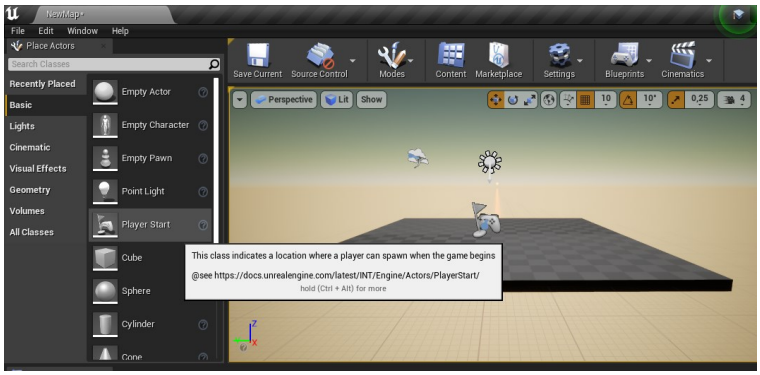


Figura 3.12 - Player start

Selezioniamo, nella scheda “Volumi”, il volume di importanza della massa luminosa che viene utilizzato per controllare e concentrare gli effetti di illuminazione e ombreggiatura all'interno del volume. Quando posizioniamo “Lightmass importance volume” nel livello, la dimensione predefinita del volume non copre la nostra area giocabile, quindi dobbiamo ridimensionarlo. All’interno del viewport di livello premiamo R per passare allo strumento di scala. Facciamo clic e trasciniamo la casella bianca al centro dello strumento scala in modo che il “Lightmass importance volume” incapsula il pavimento.

Figura 3.13 - Lightmass importance volume

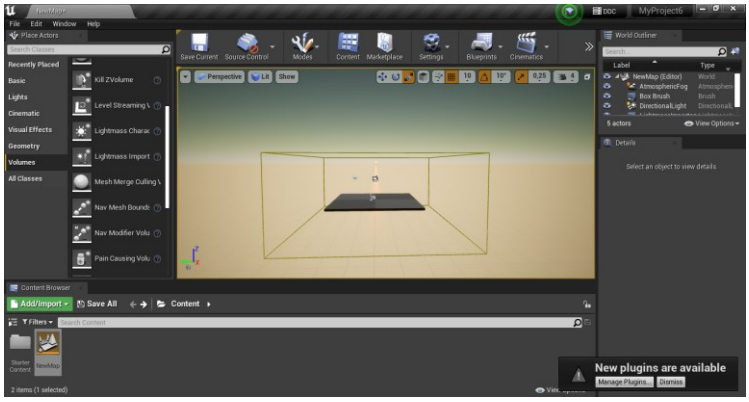


Figura 3.14 - Scala del Lightmass importance volume

All'interno del "Content Browser" in Content > StarterContent > Props, possiamo selezionare e trascinare nel nostro livello gli attori per creare una piccola scena. Inoltre, una volta che abbiamo inserito tutti gli attori, possiamo cambiare il loro materiale e personalizzare la scena. Tutto questo possiamo farlo andando in Materiali nel pannello Dettagli. Infine, dalla barra degli strumenti principale, clicchiamo su "Play" per riprodurre nell'Editor. Utilizzando WASD per muoverci e il mouse per girare con la fotocamera, possiamo volare intorno al nostro livello.

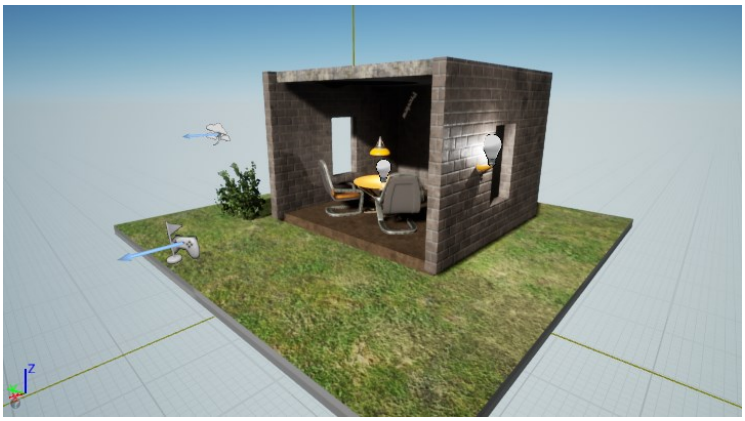


Figura 3.15 - Stanza completata

Capitolo 4

C++ in Unreal Engine 4

UE4 fornisce due metodi, C++ e Blueprint Visual Scripting, per creare nuovi elementi di gioco. Usando C++, i programmatori aggiungono i sistemi di gioco di base sui quali i progettisti possono basarsi o con i quali creare il gameplay personalizzato per un livello o per il gioco. In questi casi, il programmatore C++ lavora in un editor di testo (come Notepad++) o un IDE (di solito Microsoft Visual Studio o XCode di Apple) e il progettista lavora nell'editor Blueprint all'interno di UE4. L'API di gioco e le classi framework sono disponibili per entrambi i sistemi, che possono essere utilizzati separatamente, ma mostrano il loro vero potenziale se usati insieme per completarsi a vicenda. Cosa vuol dire questo? Vuol dire che il motore funziona meglio quando i programmatori creano blocchi di gioco in C++ e i progettisti prendono quei blocchi e creano un gameplay interessante.

Commentato [PT2]: Credo più VSCode

Commentato [PT3]: Rimane qualche dubbio su come sarà possibile fare testing o debugging dal momento che si utilizza uno strumento di sviluppo non integrato con il motore di esecuzione.

4.1 Nozioni di base sulla programmazione

Un gioco individuale è definito da un progetto di gioco che contiene tutto il codice, il contenuto e le impostazioni specifiche di quel particolare gioco. Il codice di gioco è contenuto all'interno di uno o più moduli di gioco ed ogni progetto di gioco deve contenere almeno un modulo. Vengono utilizzati i file di configurazione per impostare i valori e le proprietà che verranno inizializzate al caricamento del progetto. La configurazione è determinata da coppie chiave-valore, disposte in sezioni. Inoltre, uno o

più valori possono essere associati a una determinata chiave. Tutto questo è la base di ogni gioco fatto con Unreal Engine.

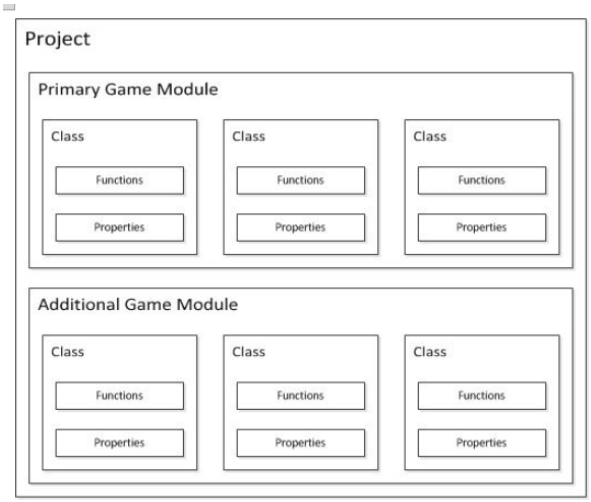


Figura 4.1 - Schema progetto di gioco in C++

4.1.1 Moduli

Come il sistema è composto da una raccolta di moduli, così ogni gioco è composto da uno o più moduli di gioco. Questi moduli sono simili ai packages del motore in quanto sono contenitori per una raccolta di classi correlate. In UE4, poiché il gioco è fatto tutto in C++, i moduli sono in realtà DLL. Inoltre, possiamo dire che le funzioni e le classi a cui è necessario accedere al di fuori del modulo devono essere esposte tramite le macro API (Application Programming Interface). Siccome ogni elemento ha un costo in fase di

compilazione, è necessario accedere solo alla singola funzione in modo tale da risparmiare una quantità significativa di tempo durante la compilazione.

4.1.2 Classi

Le classi di gioco sono definite utilizzando i file di intestazione e sintassi C++ standard. Un Attore è qualsiasi oggetto di gioco che si può posizionare in un livello. Tutti gli Attori si estendono dalla classe AActor, che è la classe base degli oggetti di gioco riproducibili. Gli Attori possono essere visti come contenitori che contengono degli oggetti chiamati componenti. Ad esempio: un oggetto CameraActor contiene un oggetto CameraComponent.

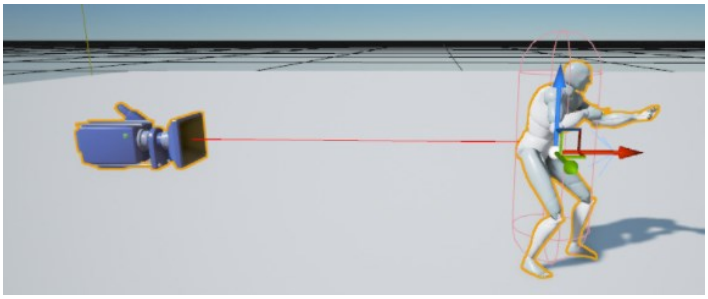


Figura 4.2 - Camera Component

4.2 Assert

Nella programmazione C++, gli Assert consentono di rilevare e diagnosticare condizioni di runtime impreviste o non valide durante lo sviluppo. Queste condizioni sono spesso controlli che un puntatore non è nullo, un divisore è diverso da zero, una funzione non è

Commentato [PT4]: In effetti l'asserzione è il componente fondamentale anche dei casi di test, per verificarne l'esito

in esecuzione in modo ricorsivo. In alcuni casi Assert rileva i bug che causano crash ritardati prima che il crash effettivo avvenga. Ad esempio, la cancellazione di un oggetto che sarà richiesto in un Tick futuro, aiutando lo sviluppatore a scoprire la causa principale di un eventuale crash. Una caratteristica chiave di Assert è che non ha alcun impatto sulle prestazioni di un prodotto spedico, ma non deve nemmeno avere effetti collaterali. Tutto ciò che viene asserito deve essere vero, altrimenti il programma smetterà di funzionare. UE4 fornisce tre famiglie diverse di Assert: Check, Verify, Ensure. Ognuna di queste si comporta in maniera leggermente diversa, ma tutte hanno lo stesso ruolo da utilizzare durante lo sviluppo.

4.2.1 Check

La famiglia Check è la più vicina all'asserzione di base, in quanto i membri di questa famiglia fermano l'esecuzione quando il primo parametro valuta un valore falso.

Macro	Parametri	Comportamento
`check` o `checkSlow`	`Expression`	Interrompe l'esecuzione se è false`Expression`
`checkf` o `checkfSlow`	`Expression, , FormattedText...`	Interrompe l'esecuzione se è false e restituisce al log`ExpressionFormattedText`
`checkCode`	`Code`	Esegue all'interno di una struttura di loop do-while che viene eseguita una sola volta: Utile principalmente come modo per preparare le informazioni che un altro controllo richiede`Code`
`checkNoEntry`	(nessuna)	Interrompe l'esecuzione se la riga viene mai colpita, in modo simile a , ma destinata a percorsi di codice che dovrebbero essere irraggiungibili`check(false)`
`checkNoReentry`	(nessuna)	Interrompe l'esecuzione se la linea viene colpita più di una volta
`checkNoRecursion`	(nessuna)	Interrompe l'esecuzione se la linea viene colpita più di una volta senza uscire dall'ambito
`unimplemented`	(nessuna)	Interrompe l'esecuzione se la linea viene mai colpita, simile a , ma destinata a funzioni virtuali che devono essere ignorate e non chiamate`check(false)`

Figura 4.3 - Macro Check

4.2.2 Verify

La famiglia Verify si comporta in modo identico alla famiglia Check nella maggior parte delle build. Tuttavia, le macro Verify valutano le loro espressioni anche nelle build in cui le macro Check sono disabilitate. Questo significa che dovremmo utilizzare le macro Verify solo quando l'espressione deve essere eseguita indipendentemente dai controlli diagnostici. Ad esempio, se abbiamo una funzione che esegue un'azione e poi restituisce un valore che indica se l'azione è riuscita o fallita, dovremmo utilizzare Verify anziché Check per assicurarci che l'azione abbia avuto successo. Tutto questo perché nelle build di spedizione, Verify ignora il valore di ritorno eseguendo ugualmente l'azione; Check, invece, non chiama proprio la funzione nelle build di spedizione.

Commentato [PT5]: Probabilmente si intendono le versioni destinate alla pubblicazione

Macro	Parametri	Comportamento
`verify` o `verifySlow`	`Expression`	Interrompe l'esecuzione se è false`Expression`
`verifyf` o `verifyfSlow`	`Expression, , FormattedText...`	Interrompe l'esecuzione se è false e restituisce al log`ExpressionFormattedText`

Figura 4.4 - Macro Verify

4.2.3 Ensure

La famiglia Ensure è simile alla famiglia Verify, una funzione con errori non irreversibili. Questo significa che se l'espressione di una macro Ensure viene valutata come falsa, l'Engine informerà il crash reporter, ma continuerà a funzionare. Per evitare di inondare

il crash reporter, dobbiamo assicurare che le macro Ensure vengano riportate solo una volta per sessione. Se invece abbiamo bisogno che la macro Ensure venga segnalata ogni volta che viene valutata come falsa, dobbiamo utilizzare la versione “Always” della macro.

Commentato [PT6]: Non ho capito la differenza tra Check e Ensure: entrambe all’occorrenza di un valore false nella condizione di controllo non terminano l’esecuzione e rilasciano nel log qualche indicazione relative a quest’evento

Macro	Parametri	Comportamento
`ensure`	`Expression`	Notifica al segnalatore dell'incidente la prima volta che è falso`Expression`
`ensureMsgf`	`Expression, , FormattedText...`	Notifica al segnalatore di arresto anomalo e l'output nel registro la prima volta è false`FormattedTextExpression`
`ensureAlways`	`Expression`	Notifica al segnalatore dell'arresto anomalo se è falso`Expression`
`ensureAlwaysMsgf`	`Expression, , FormattedText...`	Notifica al segnalatore di arresto anomalo e restituisce al registro se è false`FormattedTextExpression`

Figura 4.5 - Macro Ensure

4.3 Introduzione alla programmazione C++ in UE4

4.3.1 Creazione guidata di Classi

La prima cosa che facciamo è utilizzare Class Wizard all'interno dell'Editor per generare la classe C++ di base che verrà estesa da Blueprint in seguito.

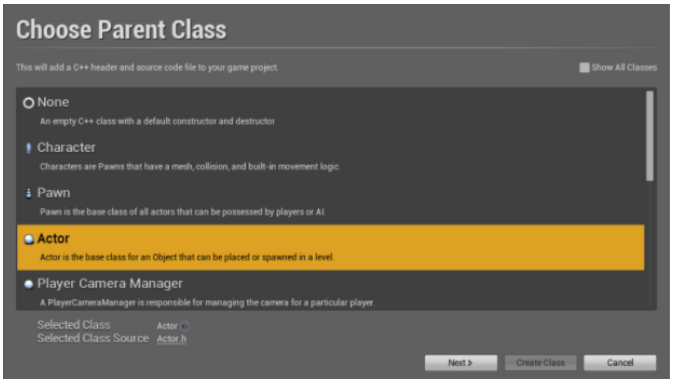


Figura 4.6 - Creazione di un nuovo Attore

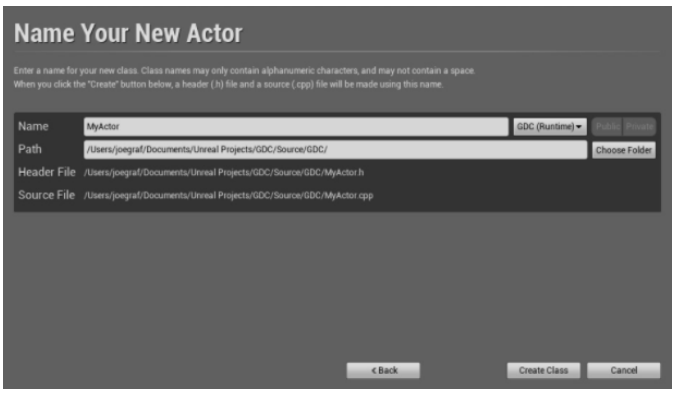


Figura 4.7 - Nome della Classe

Una volta che abbiamo creato la classe, verrà generato un file e si aprirà l'ambiente di sviluppo in modo tale da poterlo modificare. Class Wizard genera la nostra classe con BeginPlay e Tick:

- Begin Play è un evento che ci permette di sapere che l'Attore è entrato nel gioco in uno stato giocabile.
- Tick è chiamato una volta per frame con la quantità di tempo trascorso dall'ultima chiamata passata. Possiamo eseguire qualsiasi logica ricorrente qui. Se non abbiamo bisogno di questa funzionalità, possiamo rimuoverla per risparmiare una piccola quantità di prestazioni.

```
#include "GameFramework/Actor.h"
#include "MyActor.generated.h"

UCLASS()
class AMyActor : public AActor
{
    GENERATED_BODY()

public:
    // Sets default values for this actor's properties
    AMyActor();

    // Called every frame
    virtual void Tick( float DeltaSeconds ) override;

protected:
    // Called when the game starts or when spawned
    virtual void BeginPlay() override;
};
```

Figura 4.8 - Generazione del file

4.3.2 Creazione di una proprietà nell'Editor

Dopo aver creato la classe, possiamo impostare alcune proprietà che servono ai designer ad effettuare una configurazione nell'Editor. Tutto quello che bisogna fare è mettere UPROPERTY(EditAnywhere) sulla riga sopra la nostra dichiarazione di proprietà.

```
UCLASS()
class AMyActor : public AActor
{
    GENERATED_BODY()
public:

    UPROPERTY(EditAnywhere)
    int32 TotalDamage;

    ...
};
```

Figura 4.9 - Proprietà nell'Editor

4.3.3 Ricarica a caldo

Una caratteristica importante ed interessante di Unreal è quella di compilare le modifiche C++ senza arrestare l'Editor. Ci sono due modi per farlo:

- Con l'Editor ancora in esecuzione, andiamo avanti e compiliamo da Visual Studio o Xcode. L'Editor rileverà le DLL appena compilate e ricaricherà immediatamente le modifiche.

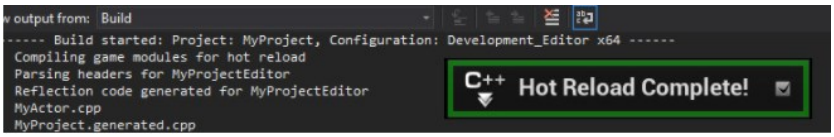


Figura 4.10 - Compilazione in Visual Studio

- In alternativa, clicchiamo sul pulsate “Compile” sulla barra degli strumenti principale dell’Editor.

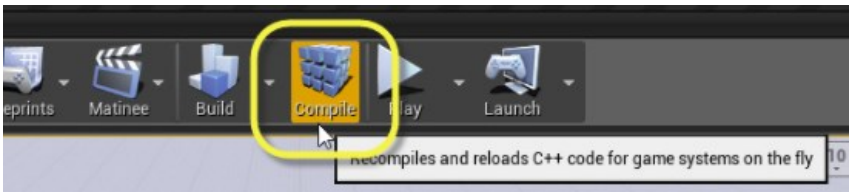


Figura 4.11 - Compilazione nell'Editor

4.3.4 Classi di gioco: Oggetti, Attori e Componenti

Ci sono quattro tipi di classi principali da cui derivano la maggior parte delle classi di gioco: UObject, AActor, UActorComponent e UStruct.

1. **UObject:** questa classe, insieme ad UClass, ci permette di fare un certo numero di funzioni importanti. UObject e UClass insieme sono alla base di tutto ciò che

un oggetto di gioco fa durante la sua vita. Il modo migliore per pensare alla differenza tra una UClass e un UObject è che la UClass descrive come sarà un'istanza di un UObject, quali proprietà sono disponibili per la serializzazione, la messa in rete e così via.

2. **AActor:** Un AActor è un UObject destinato a far parte dell'esperienza di gioco. Gli attori sono collocati in un livello da un designer o creati a runtime tramite sistemi di gioco. Gli attori possono essere distrutti esplicitamente attraverso il codice di gioco (C++ o Blueprints) o dal meccanismo standard di garbage collection quando il livello proprietario viene scaricato dalla memoria.

3. **UActorComponent:** Gli Actor Components sono di solito responsabili di funzionalità che sono condivise tra molti tipi di Actor, come fornire mesh visive, effetti particellari, prospettive della telecamera e interazioni fisiche. Un componente può essere collegato ad un solo componente genitore o Attore, ma può avere molti componenti figli collegati a sé stesso. Per illustrare la relazione tra un AActor e i suoi UActorComponents, analizziamo nel Blueprint che viene creato quando si genera un nuovo progetto basato sul First Person Template. Con le Figure (i due numeri delle figure) mostriamo l'albero dei componenti per l'Attore FirstPersonCharacter; il RootComponent è il CapsuleComponent;

collegato al CapsuleComponent c'è l'ArrowComponent, il Mesh component, e il FirstPersonCameraComponent. Il componente "Mesh1P" è parente del FirstPersonCameraComponent, il che significa che il mesh in prima persona è relativo alla telecamera in prima persona.

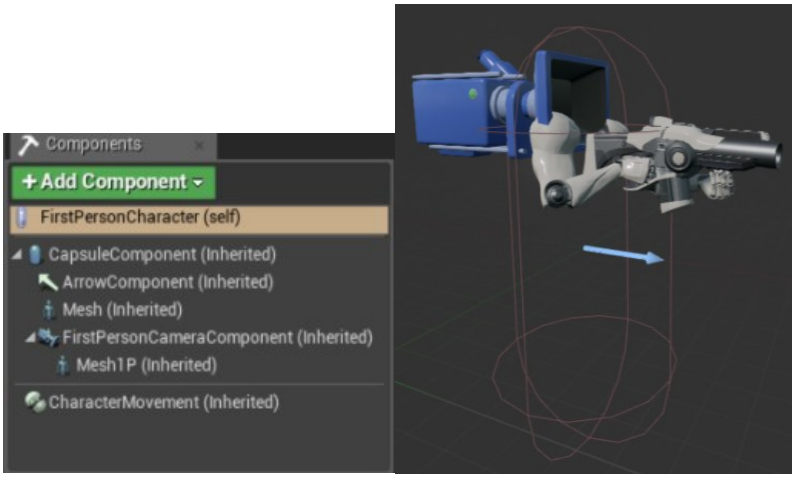


Figura 4.12 - Albero di componenti nello spazio 3D

4. **UStruct:** Per utilizzare un UStruct dobbiamo indicare la struct con UStruct() e i nostri strumenti di compilazione faranno il lavoro di base per noi. A differenza di un UObject, le istanze non vengono raccolte nella spazzatura. Se creiamo le istanze dinamiche, dobbiamo gestirne il ciclo di vita. Un UStruct è un semplice tipo di dati con supporto di riflessione UObject per la modifica all'interno dell'Unreal Editor, gestione di Blueprint, serializzazione e networking.

4.3.5 Prefissi di denominazione delle Classi

Unreal Engine fornisce strumenti che generano codice durante il processo di compilazione. Questi strumenti hanno alcune aspettative sui nomi delle classi e attiveranno avvisi o errori se i nomi non corrispondono alle aspettative. L'elenco dei prefissi di classe è il seguente:

- Le classi derivate da Actor hanno il prefisso A, come AController.
- Le classi derivate da Object hanno il prefisso U, come UComponent.
- Gli Enum hanno il prefisso E, come EFortificationType.
- Le classi interfaccia sono di solito precedute dal prefisso I, come IAbilitySystemInterface.
- Le classi template hanno il prefisso T, come TArray.
- Le classi che derivano da SWidget (Slate UI) hanno il prefisso S, come SButton.
- Tutto il resto è preceduto dalla lettera F, come FVector.

Capitolo 5

Telecamere Controllate

5.1 Posizionamento delle Telecamere

Presentiamo ora un progetto nel quale interagiamo con la parte Blueprint tramite la programmazione C++. Impariamo ad attivare e passare da una prospettiva di visualizzazione all'altra delle nostre due telecamere. Iniziamo creando un nuovo progetto di codice di base, con contenuto iniziale, chiamato "HowTo_AutoCamera". La prima cosa che dobbiamo fare è creare due telecamere nel nostro mondo. Poiché ci sono diversi modi per configurare le telecamere, noi mostreremo i due più comuni. Per la prima telecamera andiamo nel pannello "Posiziona Attori", selezioniamo "All Classes" poi "Camera" e trasciniamo il nostro attore della fotocamera nell'Editor livelli in modo che abbia una buona visione della nostra scena.

Figura 5.1 - Inserimento Camera One

Per la seconda telecamera utilizziamo un metodo che va un po' più in profondità e ci dà un po' più di controllo. Sempre nel pannello “Posiziona Attori”, selezioniamo “Basic” poi “Cube” e trasciniamo il cubo nella finestra Editor livelli. Una volta che il nostro attore “Cube” è posizionato, aggiungiamo un “Componente Fotocamera” facendo clic sul pulsante “+ Add Component” nel pannello Dettagli per il cubo. Impostiamo una posizione e una rotazione del “CameraComponent” per darci una visione diversa della scena rispetto al “CameraActor” posizionato in precedenza.

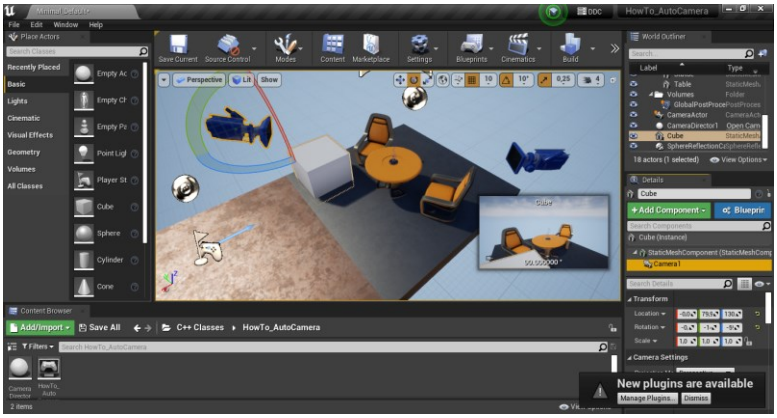


Figura 5.2 - Inserimento Camera Two

5.2 C++: Controllo della Vista

Siamo pronti ora a creare una classe C++ per controllare la vista della telecamera. Per questo progetto, possiamo estendere “Actor” in una nuova classe che chiamiamo “CameraDirector”.

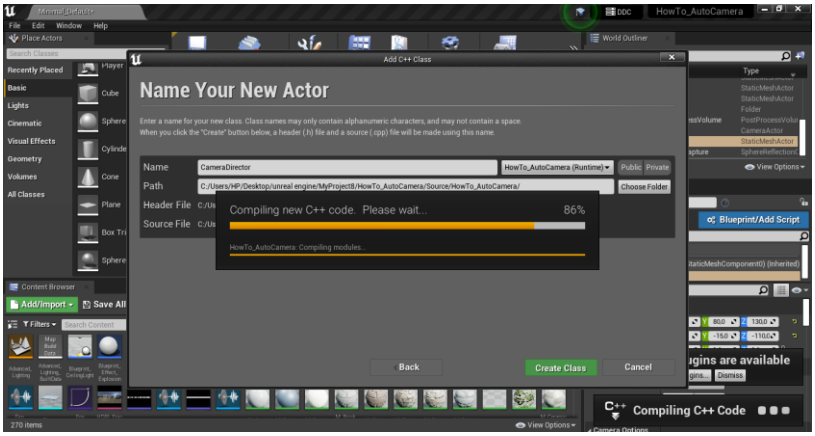


Figura 5.3 - Nuova classe in C++

Una volta che il nuovo codice C++ è stato compilato, apriamo Visual Studio e facciamo delle modifiche al codice. In “CameraDirector.h” aggiungiamo le macro UPROPERTY in modo tale da rendere le nostre variabili visibili ad Unreal Engine. Così facendo, i valori impostati in queste variabili non verranno ripristinati quando avviamo il gioco oppure ricarichiamo il nostro livello o progetto in una sessione di lavoro futura. Aggiungiamo anche la parola chiave “EditAnywhere” che ci permette di impostare la CameraOne e la CameraTwo. Successivamente, in “CameraDirector.cpp”, aggiungiamo il file di intestazione “GameplayStatics” che ci dà accesso ad alcune funzioni gerarchiche. Infine,

nella parte inferiore di “ACameraDirector::Tick” inseriamo il codice che ci permetterà di cambiare la vista predefinita del giocatore tra due diverse telecamere ogni tre secondi. Il nostro codice ora è pronto per essere compilato, quindi ritorniamo all’Unreal Editor e premiamo il pulsante “Compile”. Questo è il codice finito:

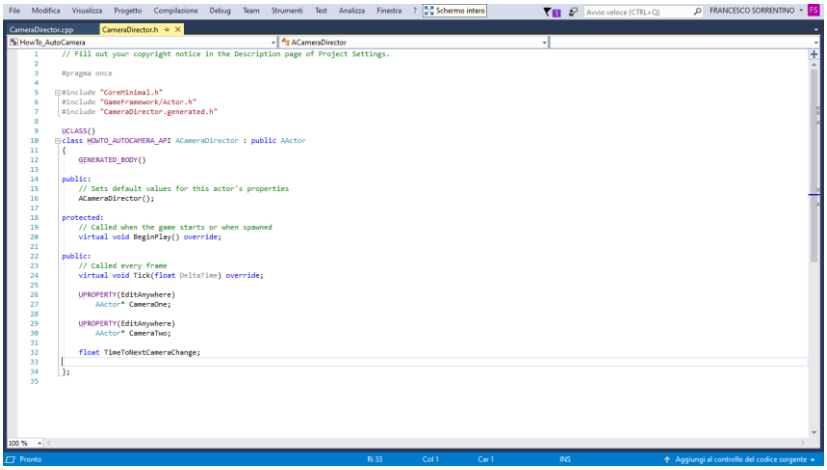
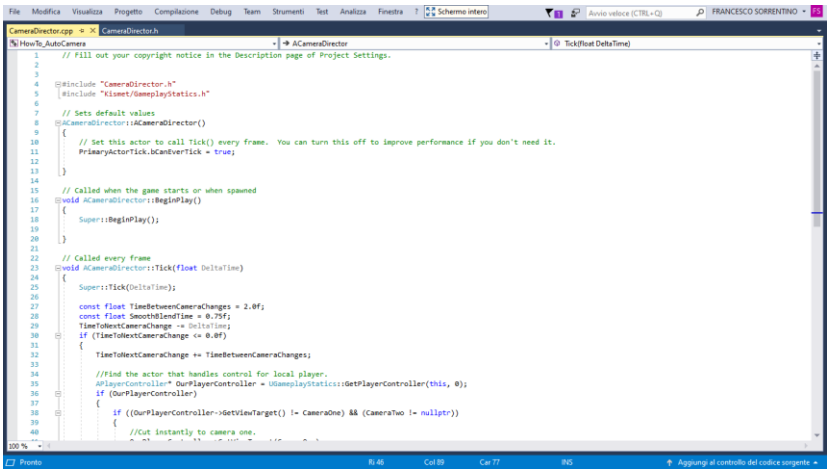
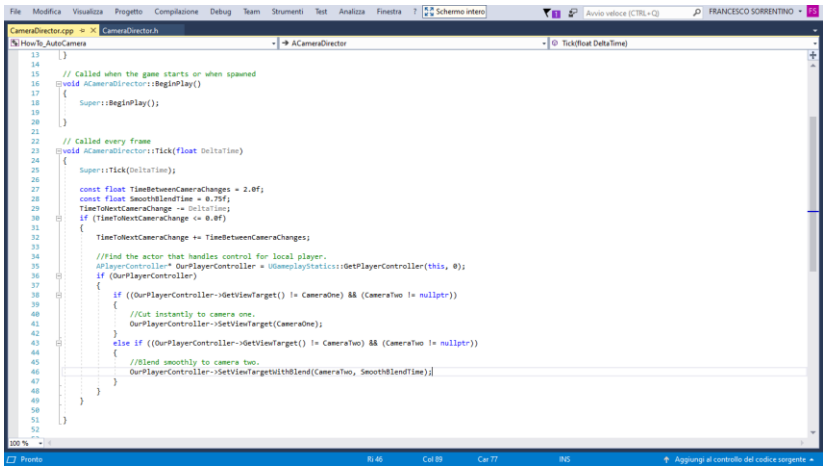


Figura 5.4 - CameraDirector.h



```
// Fill out your copyright notice in the Description page of Project Settings.


#include "CameraDirector.h"
#include "Kismet/GameplayStatics.h"

// Sets default values
ACameraDirector::ACameraDirector()
{
    // Set this actor to call Tick() every frame.  You can turn this off to improve performance if you don't need it.
    PrimaryActorTick.bCanEverTick = true;

}

// Called when the game starts or when spawned
void ACameraDirector::BeginPlay()
{
    Super::BeginPlay();

}

// Called every frame
void ACameraDirector::Tick(float DeltaTime)
{
    Super::Tick(DeltaTime);

    const float TimeBetweenCameraChanges = 2.0f;
    const float SmoothBlendTime = 0.75f;
    TimeToNextCameraChange -= DeltaTime;
    if (TimeToNextCameraChange <= 0.0f)
    {
        TimeToNextCameraChange += TimeBetweenCameraChanges;

        //Find the actor that handles control for local player.
        APlayerController* OurPlayerController = UGameplayStatics::GetPlayerController(this, 0);
        if (OurPlayerController)
        {
            if ((OurPlayerController->GetViewTarget() != CameraOne) && (CameraTwo != nullptr))
            {
                //Cut instantly to camera one.
```

Figura 5.5 - CameraDirector.cpp



```
}

// Called when the game starts or when spawned
void ACameraDirector::BeginPlay()
{
    Super::BeginPlay();

}

// Called every frame
void ACameraDirector::Tick(float DeltaTime)
{
    Super::Tick(DeltaTime);

    const float TimeBetweenCameraChanges = 2.0f;
    const float SmoothBlendTime = 0.75f;
    TimeToNextCameraChange -= DeltaTime;
    if (TimeToNextCameraChange <= 0.0f)
    {
        TimeToNextCameraChange += TimeBetweenCameraChanges;

        //Find the actor that handles control for local player.
        APlayerController* OurPlayerController = UGameplayStatics::GetPlayerController(this, 0);
        if (OurPlayerController)
        {
            if ((OurPlayerController->GetViewTarget() != CameraOne) && (CameraTwo != nullptr))
            {
                //Cut instantly to camera one.
                OurPlayerController->SetViewTarget(CameraOne);
            }
            else if ((OurPlayerController->GetViewTarget() != CameraTwo) && (CameraTwo != nullptr))
            {
                //Blend smoothly to camera two.
                OurPlayerController->SetViewTargetWithBlend(CameraTwo, SmoothBlendTime);
            }
        }
    }

}
```

Figura 5.6 - CameraDirector.cpp

5.3 Inserimento di un Camera Director

Una volta compilato il nostro codice, possiamo configurare il nostro “CameraDirector” all’interno del progetto. Trasciniamo un’istanza della nostra nuova classe dal “Content Browser” nell’Editor di livelli. Successivamente impostiamo le variabili CameraOne e CameraTwo e nel pannello Dettagli modifichiamo il CameraDirector nel Word Outliner.

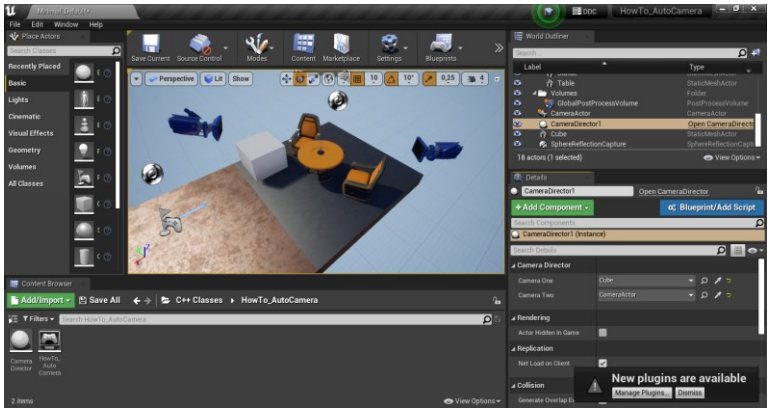


Figura 5.7 - Configurazione del Camera Director

Ora abbiamo un sistema che muove la telecamera del giocatore basandosi esclusivamente sulla logica del gioco. Cliccando “Play” vediamo la fotocamera scattare da una vista all’altra dove aspetterà qualche secondo prima di tornare indietro.

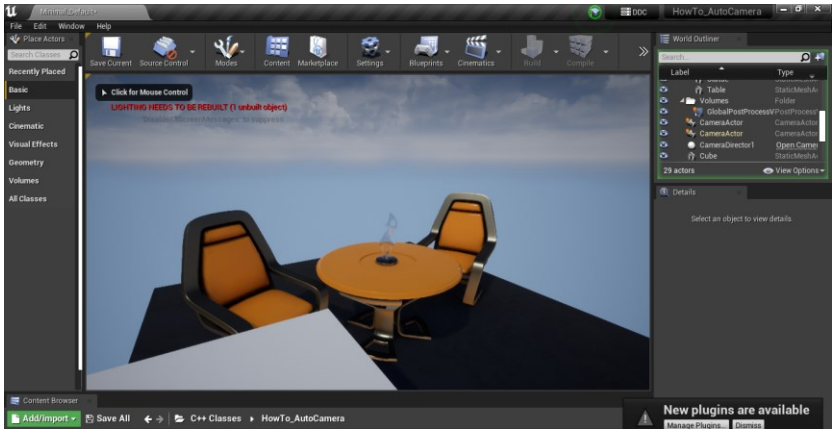


Figura 5.8 - Prima vista

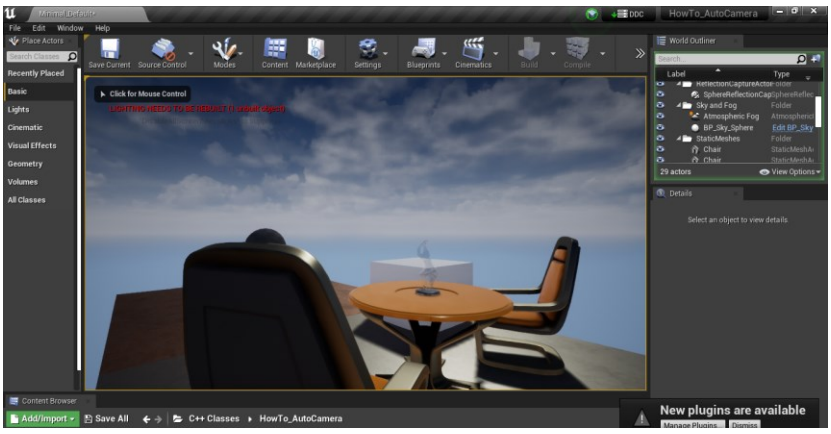


Figura 5.9 - Seconda vista

Conclusione

Quello che abbiamo visto in questa tesi è una piccola parte di quello che si può realizzare con Unreal Engine. Devo dire che ho riscontrato diverse difficoltà durante lo studio e la progettazione con UE4 poiché era la prima volta che utilizzassi questo motore. Grazie allo studio fatto sul manuale del sito ufficiale di Unreal Engine, alle numerose ricerche fatte sul web ed i tanti video tutorial visti su YouTube, sono riuscito a risolvere i vari problemi avuti durante la trattazione ed a realizzare i progetti presentati nella tesi. Devo inoltre dire che, dopo lo studio fatto a riguardo, mi sono appassionato di questo mondo e sicuramente lo approfondirò privatamente.

Commentato [PT7]: Manca la bibliografia che elenca appunto le fonti che hai utilizzato