Università di Genova | DIPARTIMENTO DI INFORMATICA, BIOINGEGNERIA, ROBOTICA E INGEGNERIA DEI SISTEMI

SCUOLA DI SCIENZE MATEMATICHE, FISICHE E NATURALI

Corso di laurea in INFORMATICA (L-31)

Anno accademico 2024/2025

Sviluppo semi procedurale di rampicanti tramite Unreal Engine

Candidata: Sofia Nocco

Relatore

Prof. Enrico Puppo

Indice

Abstract

La tesi descrive l'attività di tirocinio svolta presso l'azienda *Untold Games*, incentrata sullo studio e lo sviluppo di tecniche procedurali per la generazione di elementi naturali tridimensionali all'interno di ambienti virtuali.

L'obiettivo del progetto è stato la realizzazione di un algoritmo per la creazione di rampicanti in maniera semi-procedurale, destinato a un videogioco in produzione presso l'azienda, mediante l'utilizzo del motore grafico *Unreal Engine*.

In seguito a una fase iniziale di apprendimento degli strumenti e di familiarizzazione con l'ambiente di sviluppo, la scelta metodologica è ricaduta sull'*algoritmo di Space Colonisation* come base di riferimento e sono stati sperimentati diversi approcci per adattarlo alle specifiche esigenze del progetto.

I risultati finali mostrano un'elevata capacità di adattamento alle superfici delle mesh e un effetto visivo realistico dalla prospettiva di gioco, contribuendo così a migliorare l'immersione e la qualità estetica dell'ambiente virtuale.

Il lavoro ha permesso di approfondire la conoscenza del linguaggio *C++* e di acquisire competenze pratiche nell'utilizzo di *Unreal Engine*, fornendo una solida esperienza nello sviluppo di contenuti procedurali per videogiochi.

Capitolo I – Introduzione e definizione del problema

Il presente lavoro di tesi si propone di sviluppare un *algoritmo per la generazione semi-procedurale di vegetazione rampicante*, concepito come *plugin integrabile* all'interno del videogioco *City 20*, un titolo ambientato in un contesto semi-apocalittico. L'obiettivo principale è fornire un sistema in grado di creare dinamicamente elementi vegetali realistici, capaci di adattarsi in modo coerente alle superfici e alle geometrie dell'ambiente virtuale, contribuendo così a migliorare il livello di immersività visiva e la verosimiglianza complessiva della scena.

Lo sviluppo di questo progetto nasce dall'esigenza di combinare realismo grafico, efficienza computazionale e flessibilità di utilizzo. In ambienti di gioco complessi e fortemente dettagliati come quelli di City 20, la generazione manuale di elementi naturali può risultare estremamente onerosa in termini di tempo e risorse. L'approccio semi-procedurale consente invece di automatizzare parte del processo creativo, lasciando al designer la possibilità di intervenire sulle variabili principali per ottenere risultati coerenti con lo stile e le esigenze estetiche del progetto. Ciò permette di raggiungere un equilibrio tra controllo artistico e automazione, ottimizzando al contempo le prestazioni in fase di esecuzione.

Per la realizzazione del plugin si è scelto di utilizzare *Unreal Engine 5.4* [1], il motore grafico adottato per lo sviluppo del gioco. Tale piattaforma offre un ampio insieme di strumenti dedicati alla generazione procedurale di contenuti, oltre a un'architettura modulare che facilita l'integrazione di componenti personalizzati. In particolare, il progetto fa uso delle *classi C++*, che consentono un controllo fine sulle logiche di programmazione e sull'ottimizzazione delle performance, nonché del *plugin di Procedural Content Generation (PCG)* [2] per la gestione dello spawning semi-procedurale delle mesh. Sono stati inoltre impiegati *Blueprints* e *volumi di attivazione*, che permettono di definire in modo visivo e parametrico le aree di generazione e le condizioni di crescita dei rampicanti.

Nei paragrafi successivi verranno analizzati in dettaglio il contesto in cui si inserisce il progetto, la definizione del problema tecnico affrontato e la metodologia adottata per la sua implementazione, con particolare attenzione alle soluzioni algoritmiche e alle scelte architetturali adottate per garantire un risultato ottimale sia dal punto di vista qualitativo sia prestazionale.

II.1– Glossario dei Termini Tecnici

Dato l'uso estensivo di terminologie specifiche del game development e della computer grafica, si definiscono di seguito i concetti chiave essenziali per la comprensione del progetto.

- Spawn Volume (Volume di Generazione): Un'entità spaziale tridimensionale (spesso un box o una sfera) che delimita l'area nella scena in cui è consentita la crescita e la generazione degli elementi procedurali (i rampicanti). Nel progetto è stato sviluppato attraverso l'utilizzo di una classe Attore in C++ (paradigma di programmazione trattato in seguito).
- Exclusion Volume (Volume di Esclusione): Un'entità spaziale tridimensionale che delimita un'area in cui la crescita e la generazione degli elementi procedurali è vietata (utilizzabile per escludere finestre, porte o scritte che vogliamo lasciare perfettamente visibili). Lavora in modo complementare allo Spawn Volume.
- Raycast: Una tecnica comune di computer grafica che consiste nel proiettare un raggio da un punto nello spazio in una data direzione per determinare se interseca una geometria nella scena e, in caso, restituirne informazioni come la posizione del punto di impatto e la normale della superficie colpita. Nel contesto di questo progetto, si parla di Raycast Multipli in quanto vengono lanciati più raggi da un singolo punto, in diverse direzioni, per determinare l'aderenza su superfici complesse.
- Polyline: Un'entità geometrica di tipo bidimensionale o tridimensionale, definita da una sequenza ordinata di vertici (punti) connessi da segmenti di linea retta.
- Spline: Una curva parametrica definita da un insieme di punti di controllo (nel nostro caso saranno i GrowPoints) e dalle loro tangenti. È essenziale per rappresentare il percorso del rampicante in modo continuo e fluido, a differenza di una semplice polyline.

Capitolo II – Framework di Sviluppo: Unreal Engine e Strumenti

Il seguente capitolo illustra il contesto tecnologico, descrivendo il motore di sviluppo Unreal Engine 5.4 e gli strumenti fondamentali utilizzati nel progetto, focalizzandosi sulle potenzialità offerte e su come queste siano state sfruttate. Comprendere l'architettura del motore e le sue logiche di programmazione è cruciale per valutare le scelte implementative.

II.1 – Introduzione a Unreal Engine 5.4

Unreal Engine (UE) è un game engine robusto e versatile, ampiamente utilizzato per lo sviluppo di videogiochi, simulazioni e contenuti cinematografici.

Il motore supporta due principali paradigmi di programmazione:

- C++: Utilizzato per il nucleo dell'algoritmo di Space Colonisation (la classe USpaceColonisationAlgorithm) e per la definizione della logica dei volumi di generazione ed esclusione (classi ASpawnVolume e AExcludeVolume), offre un controllo fine sulla memoria e sulle performance, essenziale per l'efficienza computazionale.
- Blueprint: Un sistema di *scripting visuale* basato su nodi logici. È stato utilizzato nella fase preliminare per l'analisi di fattibilità e per la definizione rapida dei volumi di spawning e interazione. Consente ai designer di intervenire sui parametri senza modificare il codice.

L'algoritmo principale è implementato come *UActorComponent*, che è una classe fondamentale fornita dalla libreria di base di Unreal Engine, utilizzata per incapsulare funzionalità riutilizzabili che possono essere aggiunte a qualsiasi Actor (oggetto di gioco) della scena. Derivare il nostro algoritmo da essa ci permette di trattarlo come un "comportamento" che può essere attaccato, staccato e configurato su diversi oggetti della scena, mantenendo separata la logica di calcolo (l'algoritmo) dall'oggetto fisico (Actor) su cui cresce il rampicante.

II.1.1 – Classe Base AActor e i Volumi

I volumi spaziali utilizzati per definire le aree di crescita e di esclusione (ASpawnVolume e AExclusionVolume) sono derivati dalla classe base AActor di Unreal Engine.

Si tratta di una classe fondamentale in UE che rappresenta qualsiasi oggetto che possa essere posizionato, ruotato e scalato nella scena 3D (il "mondo" di gioco). Un Actor può contenere uno o più componenti (UComponent) che definiscono le sue proprietà e il suo comportamento.

Grazie alle sue caratteristiche di manipolabilità e riconoscibilità risulta un componente comodo da posizionare e modificare, nonché facilmente individuabile dall'algoritmo principale (USpaceColonisation).

II.2 – Procedural Content Generation (PCG)

Il Procedural Content Generation (PCG) è un sistema nativo di Unreal Engine 5.4, introdotto come plugin per generare asset e dettagli della scena in modo algoritmico, ottimizzando il level design.

PCG opera tramite la definizione di un grafo a nodi (PCGGraph). Ogni nodo rappresenta un'operazione (ad esempio, campionare superfici, filtrare, trasformare, generazione di mesh).

Nell'ambito del progetto, il PCG riceve come input i dati geometrici generati dal nostro algoritmo (le spline) e li trasforma in mesh renderizzabili (foglie, fusti, rami). Esso gestisce la variazione di scala, rotazione, materiali e la densità delle mesh lungo il percorso della spline, automatizzando la fase finale di asset spawning.

Capitolo III – Metodologia

In questo capitolo verrà illustrato nel dettaglio il processo metodologico seguito per la realizzazione del sistema di generazione procedurale di rampicanti, basato sull'*algoritmo di Space Colonisation* e integrato in Unreal Engine 5.4 tramite codice **C++** e il modulo Procedural Content Generation (PCG). L'obiettivo è ottenere un sistema flessibile, efficiente e adattabile a differenti geometrie ambientali, capace di simulare in maniera realistica la crescita naturale di un rampicante su superfici tridimensionali complesse.

III.1 – L'algoritmo di Space Colonisation e adattamento ai rampicanti

L'algoritmo di Space Colonisation, introdotto originariamente da Runions, Lane e Prusinkiewicz (2007) [3], è un metodo di crescita procedurale utilizzato per simulare la distribuzione naturale di rami e foglie negli alberi. L'idea principale è quella di far evolvere una rete di segmenti (rami) che si espandono progressivamente verso un insieme di punti di attrazione che rappresentano lo spazio "colonizzabile" dalla pianta.

III.1.1 – Funzionamento generale

Il funzionamento dell'algoritmo può essere riassunto nei seguenti passaggi:

1. Distribuzione dei punti di attrazione: vengono generati casualmente nello spazio (o sulla superficie di una mesh) i punti che indicano le direzioni di crescita desiderate.
2. Ricerca del ramo più vicino: per ogni punto di attrazione viene individuato il ramo più vicino, purché la distanza sia inferiore a una soglia di influenza $D_{attrazione}$.
3. Calcolo della direzione di crescita: i rami vengono attratti dai punti circostanti e la loro direzione media di crescita viene calcolata come somma normalizzata dei vettori di attrazione.
4. Generazione di nuovi segmenti: nuovi nodi (GrowPoints) vengono creati lungo la direzione risultante, a distanza fissa S_{step}.
5. Rimozione dei punti consumati: i punti di attrazione troppo vicini a un ramo vengono eliminati.
6. Iterazione: il processo viene ripetuto fino a quando non restano punti attivi o viene raggiunta una condizione di terminazione.

III.1.2 –Adattamento all'ambiente dei rampicanti

Nel contesto di questo progetto, l'algoritmo è stato adattato per la simulazione della crescita di rampicanti aderenti a superfici tridimensionali.

Gli adattamenti principali riguardano:

- Crescita aderente alla superficie: i punti di attrazione vengono proiettati sulla superficie dei volumi di spawn tramite raycast multipli, permettendo al rampicante di seguire le forme dell'oggetto.
- Gestione dei volumi di esclusione: sono stati introdotti volumi di esclusione per impedire la crescita in aree indesiderate (finestre, porte, passaggi, ecc.), attraverso controlli spaziali durante ogni iterazione di crescita.
- Creazione della spline: la crescita non viene rappresentata solo come una sequenza di punti discreti, ma come curve spline, che consentono una rappresentazione continua e fluida delle ramificazioni e di conseguenza della forma finale del rampicante.
- Integrazione con PCG: le spline vengono poi convertite in geometrie procedurali utilizzando il sistema PCG di Unreal Engine, che genera automaticamente mesh e materiali seguendo la traiettoria delle spline.

Un esempio di calcolo vettoriale impiegato per l'espansione dei rami è mostrato di seguito:

```
if (MinDistance < AttractioThreshold)
{
    // 1. Calculate the theoretical position (which will be detached from the mesh)
    FVector FinalGrowPoint = (((CurrentAttractionPoint - NearestBranch).GetSafeNormal()) * GrowStep) + NearestBranch;
    FinalGrowPoint = FinalGrowPoint + HitNormal * 1.0f;
```

Figura II.1. Frammento di codice riguardante il calcolo vettoriale dei punti di crescita.

III.1.3 – Calcoli alla base della crescita dei rampicanti

La parte numerica e vettoriale dell'algoritmo si fonda su operazioni geometriche nello spazio tridimensionale. Di seguito sono riassunti i principali calcoli.

1. Distanza tra punti:

Per valutare l'influenza dei punti di attrazione (A_{ix}) sui rami esistenti (B_j) si calcola la distanza euclidea:

$$d(A_i, B_j) = \| A_i - B_j \| = \sqrt{\left(A_i^x - B_j^x\right)^2 \left(A_i^y - B_j^y\right)^2 + \left(A_i^z - B_j^z\right)^2}$$

Solo se ,$d(A_i, B_j) < D_{attrazione}$, il punto viene considerato vicino abbastanza da influenzare la crescita del ramo. $D_{attrazione}$ è la soglia di influenza.

2. Direzione di crescita:

La direzione vettoriale normalizzata ($\vec{G}$) del nuovo segmento è data dalla formula:

$$\vec{G} = \frac{A_i - B_j}{\|A_i - B_j\|}$$

Dove A_i rappresenta il punto di attrazione con le sue coordinate e B_j è la posizione del nodo di partenza del ramo. Questa direzione viene poi corretta per aderire alla superficie attraverso la normale:

$$\vec{G}_{finale} = \vec{G} + \epsilon \cdot \vec{\mathrm{N}}$$

dove $\vec{G}_{finale}$ è la direzione corretta di crescita, $\vec{\mathrm{N}}$ è la normale della superficie (ottenuta tramite raycast) nel punto B_j, e ϵ è un piccolo offset per evitare penetrazioni nella geometria.

3. Aggiornamento del punto di crescita:

Ogni nuovo nodo del rampicante (B_{next}) viene aggiunto in base alla direzione di crescita finale ($\vec{G}_{finale}$) e alla lunghezza dello step (S_{step}):

$$B_{next} = B_j + S_{step} \cdot \vec{G}_{finale}$$

dove B_{next} è la nuova posizione del nodo e S_{step} è il passo di crescita che in questa implementazione è un valore predefinito costante.

4. Calcolo delle tangenti spline:

Al fine di garantire una transizione fluida e omogenea del tracciato e consentire l'uso di spline cubiche (Ordine 3), la direzione tangenziale in ciascun punto (T_j) è definita in modo più dettagliato. Una tangente locale che approssima la curvatura viene calcolata attraverso una media ponderata tra la direzione del segmento secante adiacente e la tangente locale derivante dall'approssimazione circolare.

La formula implementata per determinare la direzione tangenziale normalizzata (D_T) è un'interpolazione lineare (Lerp) tra il vettore della direzione desiderata e il vettore di vincolo superficiale.

$$D_T = \text{Normalize}(\text{D}_{\text{sec}} \cdot (1 - \alpha) + \text{D}_{\text{Surf}} \cdot \alpha)$$

La tangente finale (T_j) viene ottenuta moltiplicando la direzione normalizzata per la lunghezza del segmento tangente (L_T):

$$T_j = \text{D}_\text{T} \cdot \text{L}_\text{T}$$

Dove D_{Sec} è la direzione di crescita organica, D_{Surf} è la direzione della tangente ideale che impone il vincolo di aderenza geometrica radiale, α è il fattore di influenza che pondera il contributo di direzione superficiale sul vettore finale, D_T è la direzione tangenziale finale normalizzata e infine L_T è la lunghezza della tangente che definisce l'intensità della curvatura nel punto j.

L'uso di questa interpolazione Lerp consente di ottenere la transizione morbida richiesta dalle spline cubiche, bilanciando il movimento rettilineo con l'aderenza.

5. Controllo dei volumi di esclusione:

Per garantire che i rampicanti non penetrino in volumi di esclusione, si effettua una verifica spaziale in coordinate locali del volume di esclusione:

$$P_{locale} = T_{volume}^{-1} \cdot \text{P}_\text{mondo}$$

$$P_{clamped\ locale} = clamp(P_{locale}, -Extent, +Extent)$$

$$P_{finale\ mondo} = T_{volume} \cdot \text{P}_\text{clamped locale}$$

Dove T_{volume} è la matrice di trasformazione del volume di esclusione e Extent è un vettore tridimensionale che rappresenta la semi dimensione del volume, ossia la distanza dal centro (noto come Origin) della Bounding Box fino al suo punto più lontano. Se P_{locale} (il punto in coordinate locali) ricade all'interno di $P_{clamped\ locale}$ (punto limitato alle coordinate del volume di esclusione) senza essere modificato, il punto è dentro il volume di esclusione e viene ricalcolato dall'algoritmo.

6. Iterazione dell'algoritmo:

Il processo continua iterativamente finché tutti i punti di attrazione vengono "consumati", generando una struttura ramificata che rispetta la geometria del volume e i vincoli spaziali.

III.2 – Programmazione preliminare tramite Visual Scripting

La fase iniziale di sviluppo, caratterizzata da una limitata familiarità con il *framework* e le sue funzionalità, ha previsto l'implementazione preliminare dell'algoritmo adattato mediante il sistema di programmazione visuale *Blueprint*, nativo di Unreal Engine. Tale approccio di *scripting* visuale

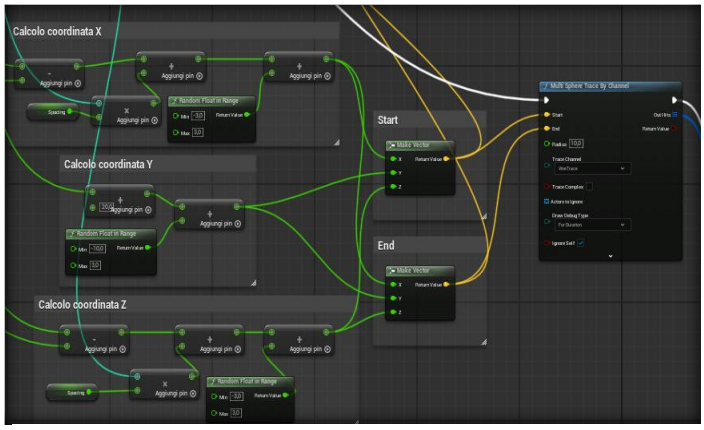

Figura II.2. Frammento di codice a nodi (campionamento punti di attrazione).

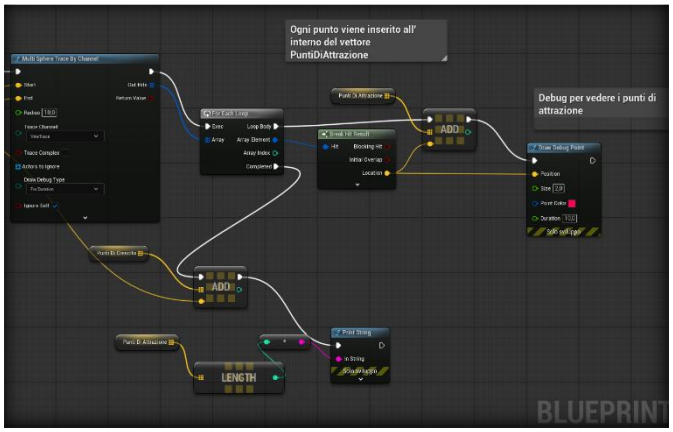

Figura II.3. Frammento di codice a nodi (gestione array dei punti)

consente la creazione della logica di gioco e delle interazioni attraverso la connessione di nodi grafici, in sostituzione della tradizionale scrittura di codice.

Per illustrare il funzionamento del visual scripting Blueprint nella fase preliminare, si considera un esempio fondamentale: la determinazione dei punti di attrazione (Attraction Points) che guidano la crescita. Le Figure II.2 e II.3 mostrano solo una porzione del codice Blueprint complessivo, ma evidenziano i passaggi logici chiave.

III.2.1 – Flusso di Lavoro a Nodi

L'approccio utilizzato si basa sull'esecuzione di un multi-sphere tracing all'interno di un'area definita, simulando una scansione tridimensionale dello spazio colonizzabile.

1. Definizione del volume di scansione (Figura II.2): il processo inizia definendo i vettori Start e End della scansione, i quali delimitano un volume di ricerca. I nodi "Calcolo coordinate X/Y/Z (Figura II.2, sinistra) ricevono i parametri parametrici dai volumi di spawn recuperati tramite un nodo Get All Actor Of Class (che si occupa per l'appunto di recuperare tutti gli attori di una certa classe, in questo caso AActor) e li convertono nelle coordinate spaziali necessarie a impostare i punti di inizio e fine del raggio di tracciamento.
2. Esecuzione del Tracing (Multi Sphere Tracing): il core del campionamento è affidato al nodo Multi Sphere Trace By Channel. Partendo dal punto Start e terminando in End, questo nodo

esegue una serie di tracciamenti sferici lungo il percorso, generando un array di risultati per ogni elemento che interseca.

3. Filtraggio e archiviazione (Figura II.3): una volta ottenuto l'array di risultati, la logica passa al filtraggio. Come si osserva nella figura, l'array viene iterato per estrarre le posizioni dei punti validi. Tali posizioni vengono quindi inserite in un array di punti di attrazione (PuntiAttrazione in figura).
4. Debug visivo: per la verifica immediata durante la fase di sviluppo, è stato aggiunto il nodo Draw Debug Point. Questo nodo, attivo solo in fase di test, visualizza direttamente nello spazio 3D la posizione di ogni punto individuato, confermando visivamente che il campionamento sta avvenendo correttamente.

III.2.2 – Blueprint come Analisi di Fattibilità

Questo esempio dimostra l'efficacia del visual scripting Blueprint nel tradurre in nodi e connessioni una complessa operazione vettoriale e di campionamento dello spazio, rivestendo un ruolo cruciale come analisi di fattibilità ed efficienza dell'algoritmo adattato. L'analisi inoltre ha permesso di evidenziare le principali criticità logistiche, fra cui spicca il malfunzionamento dell'algoritmo in presenza di rotazioni delle mesh. Le problematiche riscontrate e le relative soluzioni adottate saranno oggetto di trattazione approfondita nei capitoli successivi.

III.3 – Implementazione avanzata tramite C++ e PCG

L'intero sistema è stato implementato in **C++** come componente modulare integrato in Unreal Engine 5.4, con l'obiettivo di mantenere un elevato controllo sul flusso di calcolo e sulla generazione procedurale.

Il nucleo principale è la classe *USpaceColonisationAlgorithm*, derivata da *UActorComponent*, in grado di interagire con gli attori della scena per generare automaticamente i rampicanti.

III.3.1 – Struttura generale del sistema

Il sistema si compone di tre elementi principali:

1. Space Colonisation Algorithm (USpaceColonisationAlgorithm): gestisce la logica di calcolo e la generazione dei GrowPoints.
2. Spawn Volumes (ASpawnVolume): rappresentano le aree in cui è consentita la crescita dei rampicanti.
3. Exclusion Volumes (AExclusionVolume): aree in cui la generazione è vietata.

Durante l'inizializzazione, il componente effettua la scansione della scena per individuare tutti gli attori che implementano questi volumi, costruendo così una mappa spaziale di riferimento per la simulazione.

III.3.2 – Flusso di esecuzione

Il flusso operativo del sistema è articolato nei seguenti passaggi:

1. Rilevamento dei volumi:

   Viene eseguita una ricerca globale nella scena per individuare tutti gli ASpawnVolume e AExclusionVolume.
   Ogni volume di spawn definisce la propria area di generazione tramite un *UBoxComponent* e un *UArrowComponent* (elemento freccia fornito dalle librerie di UE) che indica la direzione di crescita principale.

2. Campionamento dei punti di attrazione:

   All'interno di ogni volume di spawn viene eseguito un semi-campionamento della facciata frontale del volume di spawn creando un insieme di punti di attrazione con la sicurezza che siano entro i limiti del volume.

   Con semi-campionamento ci si riferisce ad una distribuzione dei punti di attrazione che non è puramente casuale ma segue un pattern quasi-casuale lungo o una distribuzione mirata lungo la superficie. In questo progetto il semi-campionamento è controllato tramite dei parametri modificabili che gestiscono la distanza tra ogni punto seguita da un piccolo offset per aggiungere una leggera variazione di posizione. Questo per garantire che i punti siano sufficientemente vicini per influenzare la crescita, ma non troppo vicini per risultare ridondanti, ottimizzando l'efficienza.

   Successivamente, tramite raycast multipli lungo le direzioni ±X, ±Y, ±Z, i punti vengono proiettati sulla superficie della geometria sottostante. La proiezione è infatti necessaria a posteriori perché i punti sono inizialmente generati all'interno del volume di spawn per stabilire un obiettivo di crescita, ma devono essere poi per l'appunto "ancorati" alla mesh sottostante per garantire che il rampicante aderisca perfettamente alla topologia (muri, colonne, ecc.) anziché fluttuare nello spazio.

3. Esecuzione dell'algoritmo di Space Colonisation:

L'algoritmo viene iterato in un ciclo controllato, aggiornando la lista dei GrowPoints fino al completamento della crescita. Ogni nuova iterazione valuta i punti di attrazione, calcola le direzioni di crescita e aggiorna la lista dei rami.

La decisione di generare un *branching point* (ramificazione) è gestita da una logica algoritmica all'interno del ciclo iterativo, basata su parametri come la lunghezza del ramo corrente e la densità dei punti di attrazione non consumati nelle vicinanze.

Se un nuovo GrowPoint risulta interno a un volume di esclusione, il punto viene ricalcolato.

4. Generazione delle spline:

   Una volta terminata la fase di crescita, i punti generati vengono convertiti in una spline (USplineComponent). Ogni punto della spline memorizza posizione, tangente (direzione del percorso) e roll (angolo di rotazione applicato attorno all'asse del percorso in quel punto specifico, che controlla l'inclinazione laterale), garantendo continuità geometrica.

5. Generazione procedurale tramite PCG:

   La spline generata viene fornita al sistema PCG di Unreal Engine, attraverso il metodo PCGGeneration(). In questa fase, un UPCGComponent viene istanziato dinamicamente e collegato alla spline, che funge da input per la generazione della mesh del rampicante. Il sistema PCG utilizza un grafico dedicato (PCGGraph_Vine) che definisce i nodi di costruzione: mesh di base, materiali, rumore, variazioni di scala e rotazione.

```
// -- PCG GENERATION LOGIC --
void USpaceColonisationAlgorithm::PCGGeneration(AActor* Owner, USplineComponent* SplineComponent)
{
    // Crea un nuovo UPCGComponent sull'Owner.
    UPCGComponent* PCGComp = NewObject<UPCGComponent>(Owner);

    if (PCGComp)
    {
        // Registra il componente nel sistema di componenti di Unreal.
        PCGComp->RegisterComponent();

        // Imposta il PCG Graph che deve essere eseguito (assumendo che PCGGraph_Vine esista).
        PCGComp->SetGraph(PCGGraph_Vine);

        // Aggiunge il PCG Component come componente di istanza all'Owner.
        Owner->AddInstanceComponent(PCGComp);

        // Attiva la generazione (il parametro 'true' forza la rigenerazione immediata).
        PCGComp->Activate(true);

        // Imposta il parametro chiamato "SplineInput" nel PCG Graph per usare la nostra Spline.
        // Questo collega la Spline che abbiamo calcolato al PCG Graph per la generazione.
        UPCGGraphParametersHelpers::SetObjectParameter(PCGComp->GetGraphInstance(), FName("SplineInput"), SplineComponent);

        // Esegue la generazione PCG.
        PCGComp->Generate();
    }
}
```

Figura II.4. Frammento di codice per il setting della generazione procedurale.

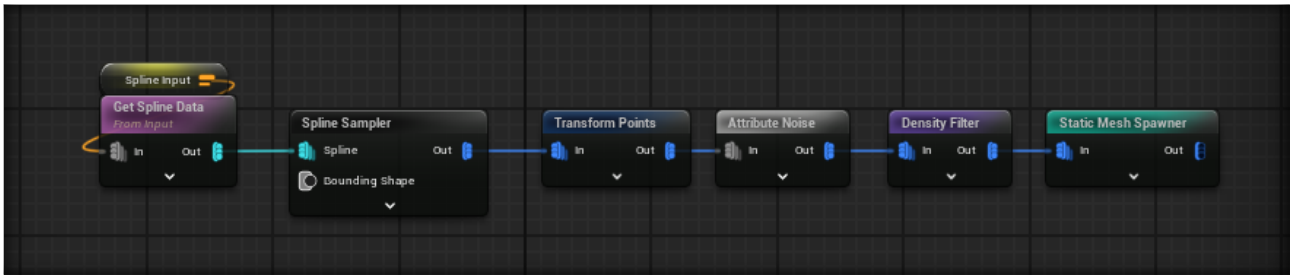

Figura II.5. Grafico PCG (PCGGraph_Vine)

6. Aggiornamento e rigenerazione: L'intero processo può essere richiamato in runtime o in editor mode, consentendo di rigenerare dinamicamente i rampicanti in risposta a modifiche della scena.

III.3.3 – Ottimizzazioni e scelte progettuali

Durante la fase di implementazione sono state adottate varie strategie per garantire prestazioni elevate e flessibilità:

- Ottimizzazione dei raycast: per ridurre il numero di calcoli, i raycast per la proiezione sulla superficie non vengono lanciati in modo puramente casuale, ma sono distribuiti secondo un pattern pre-computato e non casuale.

  Questo pattern consiste in un insieme fisso di 6 direzioni precalcolate (come una sfera a raggi) che vengono testate in sequenza, garantendo una copertura superficiale efficace con un numero di calcoli noto e limitato, a differenza di una ricerca puramente casuale che potrebbe richiedere molte più iterazioni per trovare un hit valido.

- Gestione asincrona delle generazioni: la creazione delle spline e la generazione PCG sono gestite separatamente per ogni volume, questo per poter effettuare modifiche in piena libertà sulla singola superfice evitando il ricalcolo dell'algoritmo nell'intero livello.

- Uso di strutture dati efficienti: i GrowPoints vengono memorizzati in TArray e gestiti tramite riferimenti leggeri per minimizzare overhead di memoria.

- Sistema modulare: l'algoritmo è stato progettato per essere riutilizzabile come componente generico, indipendente dalla tipologia di pianta o superficie.

- Supporto editor: sono stati aggiunti parametri editabili (GrowStep, AttractionDistance,) per consentire sperimentazioni visive.

III.4 – Conclusione della sezione

La rielaborazione proposta mantiene intatta la natura locale e iterativa dell'algoritmo di *Space Colonisation*, fornendo allo stesso tempo le regole formali per integrare controlli volumetrici, proiezione su mesh e vincoli di esclusione necessari per generare rampicanti aderenti e controllabili.

Inoltre, l'implementazione dell'algoritmo ottimizzata per il flusso di lavoro di content creation tramite l'utilizzo di una funzione marcata come CallInEditor permette di eseguire l'intera logica di generazione procedurale direttamente nell'ambiente di sviluppo di Unreal Engine, garantendo la creazione di rampicanti realistici, adattabili a qualsiasi topologia superficiale, e pienamente controllabili tramite la manipolazione dei parametri in tempo reale.

Il vantaggio cruciale di questa scelta risiede nell'efficienza:

- Generazione Statico/Off-line: l'esecuzione tramite una funzione CallInEditor esegue i calcoli complessi una volta sola durante la fase di design, e successivamente salva i risultati (le mesh generate) come elementi statici all'interno del livello.
- Vantaggio su Event Begin Play: questo approccio constrasta nettamente con l'attivazione dell'algoritmo tramite l'Event Begin Play (una funzione che esegue la sua logica ad ogni avvio del gioco). Evitando di ripetere i calcoli iterativi complessi durante l'esecuzione runtime, si ottiene un significativo incremento dell'efficienza, riducendo drasticamente il carico computazionale e i requisiti di memoria durante il gioco effettivo. Tale ottimizzazione è fondamentale per mantenere elevate frame rate e prestazioni stabili, poiché sposta il costo computazionale della fase di gameplay a quella di editing.

In conclusione, il sistema realizzato unisce fondamenti algoritmici ispirati alla crescita naturale con le potenzialità tecnologiche di Unreal Engine 5.4, integrando in modo coerente logiche procedurali, calcoli geometrici e pipeline grafica.

Capitolo IV – Problemi incontrati e soluzioni adottate

Durante lo sviluppo dell'Algoritmo di Space Colonisation sono state riscontrate diverse problematiche legate a lapsus logici o implementativi. In questo capitolo si analizzano i macro-problemi che hanno richiesto la fase più critica di debugging e ottimizzazione: Rotazione delle mesh, Mesh Complesse e Superfici cilindriche.

IV.1 – Rotazioni delle mesh

Sebbene l'esecuzione iniziale apparisse corretta, dopo i primi test è sorta la criticità legata alla rotazione degli attori.

L'algoritmo, se eseguito su superfici nella loro configurazione predefinita, funzionava correttamente; tuttavia, una qualsiasi rotazione della mesh causava un malfunzionamento, a partire dalle coordinate utilizzate per i campionamenti.

Le cause alla base di questa problematica risiedevano in 2 punti: Incoerenza del sistema di Coordinate e Bounding Box e Errato Campionamento.

IV.1.1 – Incoerenza del sistema di Coordinate e Bounding Box

La logica iniziale si basava sull'utilizzo delle coordinate globali per il riconoscimento della bounding box della mesh (definita tramite -Box Extent, +Box Extent e Box Origin che corrispondono alle coordinate più esterne e quella centrale dell'attore), un approccio che si adatta solo all'*Axis-Aligned Bounding Box (AABB).* Con la rotazione, invece di seguire la rotazione della mesh, la buonding box si ingrandiva semplicemente sugli assi globali per contenere la geometria ruotata.

Per superare questa criticità è stata adottata *la Oriented Bounding Box (OBB)*. Nello specifico è stata implementata la funzione ausiliaria *GetLocalWallBoundings*, la quale converte i punti nello spazio locale dell'attore ruotato per ricavare i corretti valori per la creazione di una bounding box che segua correttamente le rotazioni della mesh ad essa collegata.

Figura III.1. Errore di campionamento su superficie ruotata

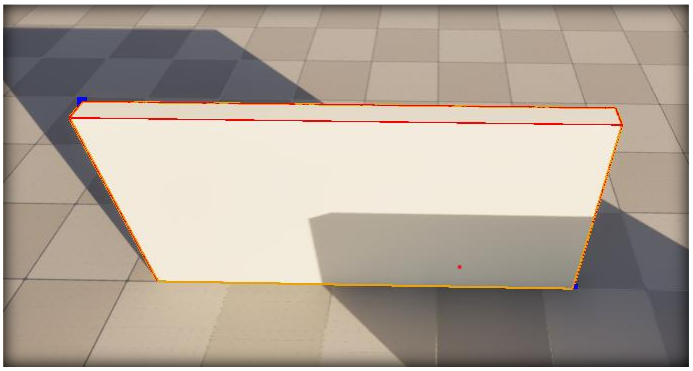

Figura III.2. Binding box standard senza rotazioni.

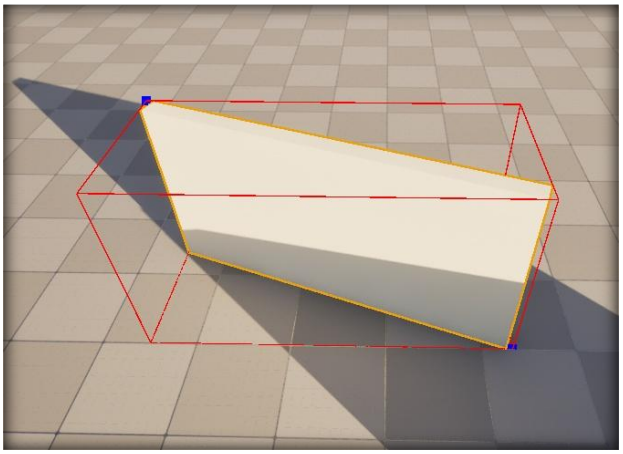

Figura III.3. Binding box ruotata pre-modifica

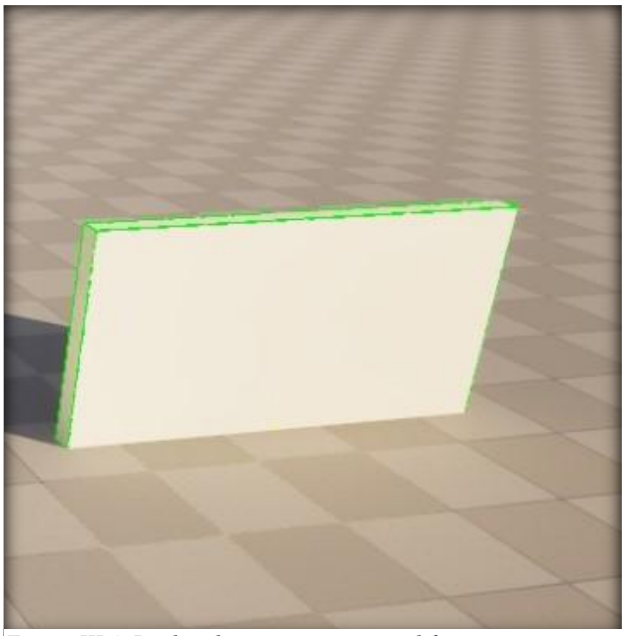

Figura III.4. Binding box ruotata post-modifica

La corretta creazione della bounding box ha permesso di creare la base per il campionamento, che ci porta alla seconda causa della problematica di base.

IV.1.2 – Errato Campionamento:

È stata eseguita una migrazione integrale della logica di crescita allo Spazio Locale del muro. I punti di attrazione ottenuti tramite raycasting vengono immediatamente convertiti dalle coordinate del mondo a coordinate locali e, in seguito, è stata applicata la *FTransform* completa (posizione e rotazione) del muro per proiettare i punti generati nello spazio del mondo.

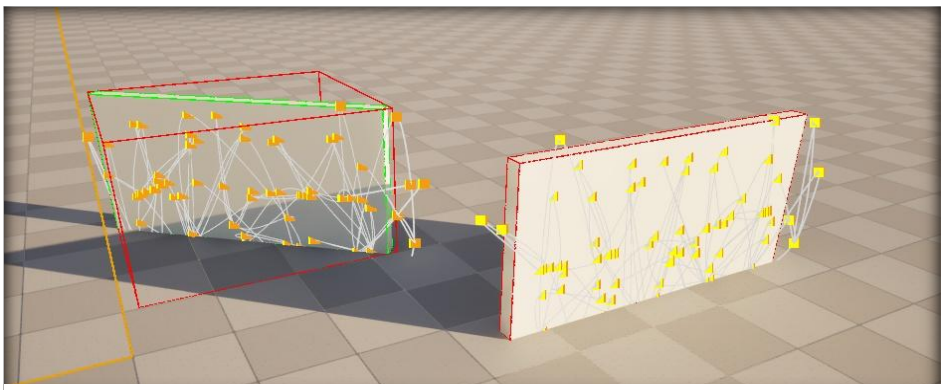

Figura III.5. Campionamento post-correzione

IV.2 – Mesh Complesse

La seconda macro-problematica è emersa come diretta conseguenza di una discordanza informativa con il team di sviluppo. Contrariamente alle assunzioni iniziali che prevedevano l'interazione con mesh singole e semplici (es. pareti), l'ambiente di gioco presentava geometrie complesse, costituite da un set di mesh separate e successivamente fuse in fase di importazione nel motore grafico. Tali asset rappresentavano interi edifici, veicoli o contenitori.

Nella sua configurazione iniziale, l'algoritmo operava recuperando a livello globale tutti gli attori associati a un tag specifico e avviando l'esecuzione sull'intera superficie di ciascuno. Questo approccio si è rilevato inefficace e insostenibile dal punto di vista logico per la gestione di mesh complesse che necessitavano di una generazione localizzata e selettiva.

Per ovviare a tale limitazione, è stata attuata una riorganizzazione della logica decisionale di spawning. Inizialmente si è sostituito l'utilizzo dei tag come meccanismo di ricerca delle superfici operative, con l'identificazione esplicita di tutte le istanze della classe ASpawnVolume.

Questa scelta strategica è stata adottata per acquisire il pieno controllo sulle aree di spawning, confinando l'applicazione dell'algoritmo unicamente alle zone delimitate da un volume di spawn. Ciò ha risolto la problematica della crescita incontrollata sull'intera superficie degli attori. Inoltre, tale modifica ha eliminato la necessità di assegnare un tag ad ogni singola superficie, ottimizzando i così i tempi operativi dei level designer. L'implementazione di questo nuovo approccio ha richiesto una modifica mirata del ciclo di iterazione principale.

Per la gestione del ciclo principale, la cui funzione è definire il numero totale di esecuzioni dell'algoritmo, è stata utilizzata la lunghezza dell'array contenente tutti gli attori della classe ASpawnVolume. Questa modifica, insieme all'utilizzo della faccia frontale dei volumi per il campionamento dei punti di attrazione ha permesso il restringimento dell'area di campionamento alla superficie diretta del volume che, come anticipato nel paragrafo precedente, garantisce che l'algoritmo operi in modo coerente e mirato solo sulle superfici intersecate da un volume di spawn, lasciando inalterate le aree non definite e circoscrivendo l'azione all'interno della zona specificata dal volume stesso.

Infine, è stato implementato un raycast direzionale proiettato verso il centro del volume. Questa tecnica è stata introdotta per garantire l'adesione superficiale alla mesh di gioco, poiché i punti campionati si trovano originariamente al livello della faccia frontale del volume e non ancorati alla geometria di gioco sottostante.

Durante la fase di testing della nuova logica, è emerso un artefatto geometrico critico: alcuni dei punti campionati si manifestavano sul lato opposto della superficie rispetto a quello desiderato.

La causa di tale anomalia risiedeva nella logica di generazione dei punti di crescita sul volume di spawn. In determinate condizioni, i punti generati attraversavano la mesh. Il raycast successivo, proiettato verso il centro del volume, intercettava come primo hit la superficie più distante (il lato opposto), invalidando la coerenza del campionamento desiderato.

Per risolvere questa situazione indesiderata e ripristinare la coerenza del campionamento, è stata implementata una sequenza logica di tre passaggi:

1. Campionamento e Logica Iniziale: esecuzione del campionamento iniziale e della logica di generazione dei punti di crescita.
2. Proiezione: I punti generati sono stati riposizionati forzatamente sulla faccia frontale del volume di spawn, garantendo che l'origine del raycast fosse esterna rispetto alla superfice di destinazione.
3. Raycast Correttivo Finale: È stato eseguito un raycast conclusivo da questa posizione corretta, garantendo l'adesione di tutti i punti al lato desiderato della superficie.

Tale affinamento ha stabilizzato la generazione dei punti, assicurando che la logica di spawning rispettasse l'orientamento geometrico previsto.

Questa modifica ha drasticamente migliorato la versatilità e la scalabilità dell'algoritmo, consentendo una generazione precisa su un'ampia gamma di superfici (pareti, pavimenti, elementi scenici complessi, veicoli) e offrendo al *level designer* il pieno controllo sulla localizzazione, inclusa la possibilità di discriminare tra l'applicazione interna o esterna sulla struttura di un edificio.

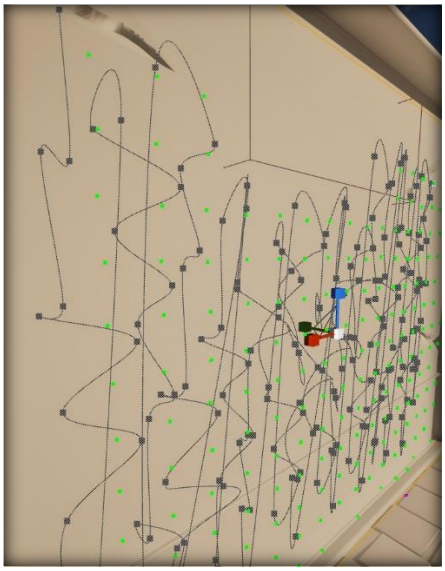

Figura III.6. Punti di attrazione corretti con spline posizionata sul lato errato

IV.3 – Superfici cilindriche

Questo problema, riguardante la compenetrazione della spline all'interno delle mesh, ha portato un cambiamento della logica di adesione superficiale.

IV.3.1 – Correzione dell'Orientamento e Adesione

Questo cambiamento è consistito nell'ottenere la normale della superfice su cui opera l'algoritmo. La funzione *CreatingSpline* è stata corretta per eseguire un raycast per ogni punto di crescita, ottenendo la normale superficiale e posizionando il punto finale a un piccolo offset di sicurezza dalla superficie.

IV.3.2 – Malfunzionamento su superfici Cilindriche:

Questo approccio, sebbene efficace per le superfici planari, ha causato un malfunzionamento su superfici cilindriche, con il risultato di una penetrazione eccessiva della spline all'interno della mesh. La motivazione alla base del problema risiede nel fatto che la spline, nel collegare due punti distanti, sceglie il percorso più breve, intersecando il volume interno del cilindro. In questi casi, la normale superficiale, pur corretta, è poco influente se la distanza tra due punti è elevata.

Per quanto a livello visivo non creasse nessun disagio, si trattava comunque di un problema di efficienza dell'algoritmo, che andava pertanto risolto.

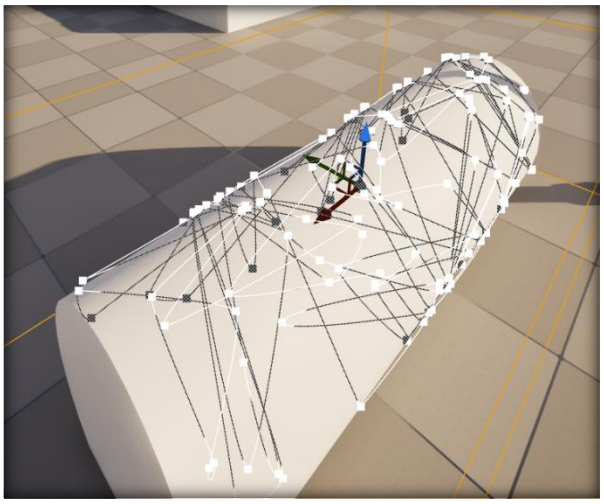

Figura III.7. Evidente compenetrazione su superficie cilindrica

IV.3.3 – Soluzione Parziale

L'approccio più robusto ha previsto l'adozione di un modello di calibrazione basato sulla distanza tra due punti di crescita.

Le operazioni sono state inizialmente suddivise in tre classi di distanza (breve, media, lunga), al fine di modulare l'influenza del vettore normale superficiale sulla traiettoria della spline.

Si è stabilito che l'incidenza del vettore normale dovesse essere direttamente proporzionale all'incremento della distanza tra due punti, scalata con una funzione esponenziale per massimizzare la correzione tangenziale dei segmenti più lunghi.

Successivamente, questa classificazione è stata affinata introducendo un'ulteriore *clusterizzazione* basata sul criterio di coerenza del vettore normale. Attraverso il campionamento della normale lungo il percorso, è stato possibile discriminare i punti che giacevano sul medesimo lato del cilindro.

Tale livello di segmentazione si è reso indispensabile per mitigare l'eccessivo gradiente di curvatura della spline tra punti a distanza medio-lunga posizionati sullo stesso lato del cilindro.

L'implementazione descritta costituisce una soluzione parziale o un *workaround* migliorativo. Sebbene abbia significativamente ridotto la frequenza e l'entità dei fenomeni di compenetrazione della spline all'interno delle mesh cilindriche, la loro occorrenza non è stata azzerata. Ritroveremo questa problematica nel Capitolo 4 – Risultati e Sviluppi Futuri.

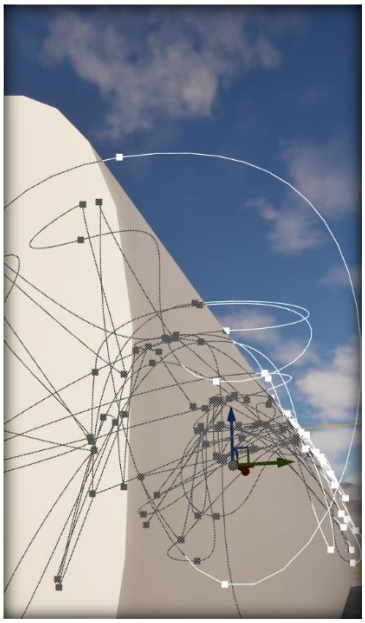

Figura III.8. Eccessiva curvatura pre-creazione classi di distanza

Capitolo V – Risultati e Sviluppi Futuri

V.1 – Risultati

Il presente sottocapitolo è dedicato all'illustrazione e analisi dei risultati ottenuti dall'applicazione dell'algoritmo sviluppato su specifiche mesh di gioco, all'interno dell'ambiente del motore Unreal Engine. Per ogni oggetto di gioco sottoposto a test, verranno presentate tre distinte visualizzazioni, al fine di documentare l'intero processo di generazione e il suo esito finale:

- Visione della spline: rappresentazione della geometria della spline generata, priva dell'applicazione delle risorse visive finali (asset delle foglie). Questa vista dimostra la correttezza strutturale del percorso generato dall'algoritmo.
- Visione in editor: questa modalità di visualizzazione fa riferimento all'ambiente di sviluppo di Unreal Engine (Editor Mode), che permette di ispezionare gli asset e i componenti del livello. In questa fase, le mesh delle foglie sono applicate alla spline, consentendo una valutazione del risultato visivo intermedio, ma senza l'esecuzione della logica completa di gioco, mostrando la visuale di un level designer all'opera.
- Visione simulazione: questa rappresentazione costituisce il risultato finale e definitivo, ovvero l'output visualizzato dal videogiocatore durante la fase di gameplay. Tale prospettiva è cruciale per valutare le prestazioni, l'integrazione visiva e la coerenza del sistema nel contesto dinamico del gioco.

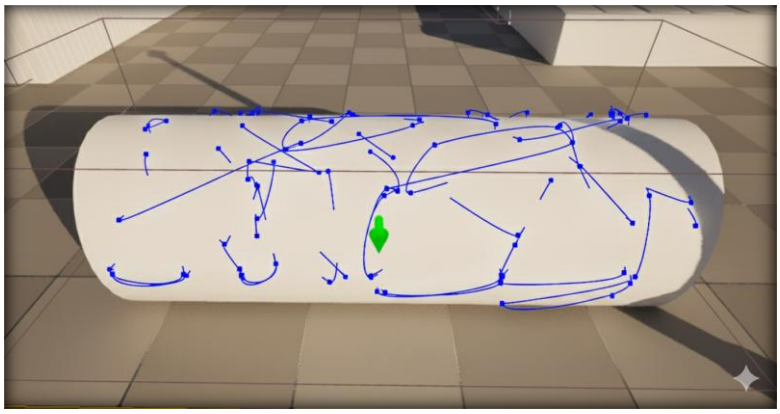

Figura IV.1. Visione della spline su cilindro

Figura IV.2. Visione in editor su cilindro

Figura IV.3. Visione simulazione su cilindro

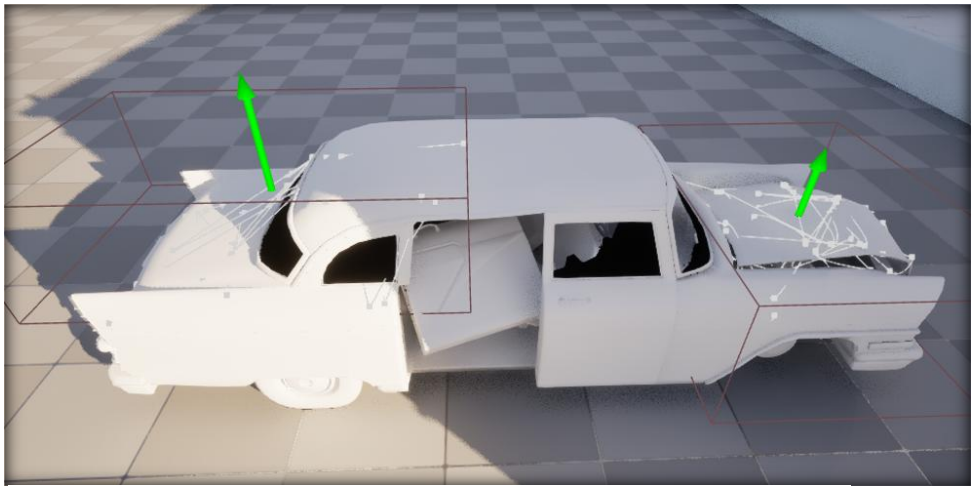

Figura IV.4. Visione della spline su mesh complessa.

Figura IV.5. Visione in editor su mesh complessa

Figura IV.6. Visione simulazione su mesh complesse

Figura IV.7. Visione della spline su muro

Figura IV.8. Visione in editor su muro

Figura IV.9. Visione simulazione su muro

V.2 – Conclusioni

V.2.1 – Analisi delle Prestazioni e Limitazioni Attuali

Sulla base del tempo di sviluppo concesso e delle conoscenze possedute, l'algoritmo sviluppato dimostra una notevole efficienza di esecuzione su superfici sia di tipologia semplice che complessa. Questa efficacia ne attesta la validità come strumento di base per la generazione procedurale di elementi ambientali.

È tuttavia necessario sottolineare che, sebbene funzionale, l'algoritmo non è esente da limitazioni e presenta un sostanziale margine di miglioramento. Tali miglioramenti possono essere perseguiti sia sul piano dell'efficacia algoritmica, mirata all'accuratezza e al realismo del risultato finale, sia su quello dell'usabilità per i professionisti del design di livello.

V.2.2 – Sviluppi Futuri: Aspetto Visivo, Adattamento Geometrie e Usabilità

Uno degli sviluppi di primaria importanza riguarda il miglioramento dell'aspetto fotorealistico dell'elemento generato, in particolare la creazione della mesh definitiva del fogliame. Attualmente, la rappresentazione visibile si basa su una mesh surrogata; la sua sostituzione con la risorsa finale in-game renderà indispensabile una ricalibrazione dei parametri algoritmici.

Questa modifica sarà cruciale per garantire che la distribuzione, la densità e l'orientamento del rampicante si adattino in modo ottimale alle dimensioni e alla struttura geometrica del nuovo asset del fogliame, preservando l'armonia visiva e la coerenza ambientale.

Un altro cruciale sviluppo riguarda la completa robustezza e affidabilità dell'algoritmo nell'interazione con superfici non planari, in particolare le geometrie cilindriche.

Attualmente, si riscontra una criticità legata alla compenetrazione della spline di base all'interno della geometria degli asset, un fenomeno che compromette l'efficienza dell'algoritmo. La risoluzione di questa problematica deve essere ancora esplorata, ma un primo approccio per mitigare il problema consiste nell'implementare una strategia di campionamento adattivo durante la fase di generazione della spline. Nello specifico, è possibile aggiungere un meccanismo di rifinitura che valuti la distanza tra due punti consecutivi della spline e, contemporaneamente, analizzi la discontinuità vettoriale delle normali della superficie sottostante in corrispondenza di tali punti.

Se la distanza tra i punti supera una soglia predefinita e l'angolo formato dai vettori normali si avvicina ai 180 gradi (indicando che i due punti si trovano su facce o curvature opposte del cilindro), l'algoritmo dovrebbe inserire iterativamente nuovi punti di controllo lungo l'arco.

Un'ulteriore strategia da testare consiste nella parametrizzazione adattata a *UV*. Nel caso di geometrie semplici come i cilindri, si potrebbe esplorare la possibilità di mappare l'estensione del rampicante direttamente nello spazio UV della mesh, assicurando che la generazione avvenga sulla superficie esterna e segua la curvatura parametrica, per poi re-importare le coordinate 3D finali.

Queste soluzioni seppur mirate a stabilizzare il comportamento dell'algoritmo su superfici curve, sono puramente ipotetiche in quanto sviluppate a posteriori del periodo di implementazione principale, basandosi su ricerche mirate effettuate in fase di post-tirocinio.

Di conseguenza, non è stato possibile convalidare sperimentalmente l'efficacia di tali approcci nella risoluzione definitiva della problematica di compenetrazione geometrica e la loro validità rimane pertanto un elemento da verificare in un contesto applicativo futuro.

Per quanto concerne l'aspetto della facilità d'uso e applicazione da parte dei level designer, la strategia di sviluppo futuro prevede un approccio iterativo e *user-centric*.

Si prevede l'implementazione di un processo di feedback strutturato con i professionisti del design di livello. La raccolta e analisi di questi riscontri saranno essenziali per identificare le criticità nell'interfaccia utente e nell'esperienza d'uso dell'algoritmo.

Le informazioni così acquisite saranno la base fondamentale per l'introduzione di migliorie mirate al workflow e alla parametrizzazione del sistema, assicurando che l'algoritmo non solo sia efficace dal punto di vista computazionale, ma anche efficiente e comodo da integrare nei processi di creazione di ambienti di gioco.

Bibliografia

[1] Epic Games, “Unreal Engine 5.4 Documentation”, 2024 [Online]. Available: https://dev.epicgames.com/documentation/en-us/unreal-engine/unreal-engine-5.4-release-notes?application_version=5.4. [Accessed: 24 Settembre 2025]

[2] Epic Games, “Unreal Engine 5.2 Release Notes” 2023 [Online]. Avaible: https://dev.epicgames.com/documentation/en-us/unreal-engine/unreal-engine-5.2-release-notes?application_version=5.2. [Accessed: 24 Settembre 2025]

[3] A. Runions, B. Lane, and P. Prusinkiewicz, “Modeling Trees with a Space Colonisation algorithms”, in ACM SIGGRAPH 2007 Papers, 2007, pp. 1-9

Ringraziamenti

Desidero dedicare questo spazio, a tutte le persone che hanno reso possibile il completamento di questo percorso formativo. I ringraziamenti che seguono sono dettati da un profondo senso di gratitudine per il supporto ricevuto in questi anni accademici.

Un ringraziamento particolare e doveroso è rivolto al mio Relatore, il chiarissimo Professor Enrico Puppo. La sua guida non solo mi ha permesso la realizzazione di questo elaborato finale, ma mi ha soprattutto aperto le porte verso un nuovo e affascinante mondo professionale di studio. Sono profondamente grata per l'opportunità di tirocinio curriculare che mi ha consentito di conseguire presso l'azienda Untold Games. La sua costante disponibilità e la rapidità nel dirimere ogni mio dubbio e richiesta sono state fondamentali per la fluidità del lavoro e per il raggiungimento di questo obiettivo.

Un ringraziamento speciale è rivolto ai miei Genitori. La loro lungimiranza e il loro costante incoraggiamento sono stati fondamentali nell'indirizzarmi verso questa disciplina. Li ringrazio per aver celebrato con entusiasmo ogni traguardo conseguito e, cosa ancora più cruciale, per avermi offerto incondizionato sostegno e conforto nei momenti difficoltà e di fronte ai fallimenti lungo il percorso. La loro fiducia è stata la mia risorsa più preziosa.

Esprimo la mia gratitudine ai miei Amici più cari. Il loro ascolto paziente di ogni preoccupazione, lamentela o momento di stress, in particolare durante le intense sessioni d'esame, è stato un pilastro emotivo. Sono stati un punto di riferimento insostituibile, risollevandomi da terra ogni volta che la determinazione vacillava.

Infine, un ringraziamento sentito ai miei Compagni di Corso. Il tempo trascorso insieme, dalle proficue sessioni di studio condivise ai momenti di leggerezza, come i nostri pranzi al sushi, ha reso l'esperienza universitaria non solo sostenibile, ma memorabile. Sono la ragione principale per cui, se mi venisse posta la domanda: "Tornando indietro, rifaresti lo stesso percorso di studi?", la mia risposta sarebbe indubbiamente affermativa.