POLITECNICO DI TORINO

Corso di Laurea Magistrale in Ingegneria del Cinema e dei Mezzi di Comunicazione

Tesi di Laurea Magistrale

Workflow di controllo real-time di parametri audio e video nell'ambito di una installazione data-driven

Relatore

Prof. Riccardo Antonino

Candidato

Mattia Gravina

Anno Accademico 2023/2024

Sommario

Negli ultimi anni, la produzione in larga scala e il facile accesso a grandi quantità di dati hanno fatto sì che l'utilizzo di questi ultimi acquisisse un valore sempre maggiore. Uno degli aspetti in cui l'uso di dati è diventato essenziale è la ricerca, dove si è osservato un aumento delle pubblicazioni di articoli che adottano un approccio basato sui dati, anche nelle discipline umanistiche e sociali. Questo fenomeno ha reso la *data visualization* sempre più importante, e la ricerca di nuovi metodi per visualizzare e rendere comprensibili dati complessi di diverso tipo rappresenta una sfida sempre più comune per chi si occupa di design o comunicazione. In aggiunta, lo sviluppo delle tecnologie digitali ha reso popolare il concetto di arte generativa, ovvero la pratica di definire degli algoritmi che hanno il compito di generare un'opera d'arte. Il presente lavoro di tesi si inserisce all'interno dello sviluppo di Egitto Immersivo, un progetto del Museo Egizio di Torino che prevede la creazione di due sale immersive. Parte del contenuto di queste sale verrà renderizzato in real-time, e fornirà una rappresentazione aggiornata in tempo reale di dati riguardanti gli studi in Egittologia.

La tesi presenta un workflow per controllare i parametri che definiscono il comportamento e le caratteristiche di sistemi particellari e sistemi di rendering audio in tempo reale tramite dati online. Nello specifico, il metodo definito prevede l'utilizzo del motore grafico Unreal Engine 5, scelto grazie alla capacità di eseguire render di alta qualità di scene anche molto complesse in tempo reale. Nel presente lavoro di tesi il metodo è stato applicato sul sistema di Visual Effects Niagara e sul sistema audio MetaSounds, presenti in Unreal Engine 5. La programmazione è stata eseguita in Blueprint Visual Scripting, ovvero il sistema di scripting a nodi interno al motore. Per quanto riguarda i dati, è stato usato il software di fogli elettronici Google Sheets, sia per via della sua semplicità d'uso, sia per via

della presenza di API che permettono di eseguire l'accesso ai dati presenti in uno spreadsheet.

Nella prima parte dell'elaborato, sono presentati i lavori preliminari, svolti prima della fase operativa del progetto. Per prima cosa, sono analizzate alcune tecniche che sono state poi utilizzate nello sviluppo del workflow, ad esempio il controllo dei parametri tramite Blueprint, e tecniche usate nei progetti dimostrativi, come il sampling delle mesh tramite Niagara. In seguito, si analizzano le tre scene realizzate per le proiezioni della Cena di Natale 2023 del Museo Egizio. Queste proiezioni hanno avuto come scopo quello di mostrare allo staff del Museo delle proposte stilistiche in vista del progetto Egitto Immersivo.

Nei capitoli successivi, viene presentato il workflow sviluppato per il lavoro di tesi. In primo luogo, sono studiate le Structures e le Data Table, che sono gli strumenti attraverso cui creare tabelle di dati in Unreal Engine. Poi, si analizzano i metodi attraverso cui usare tali asset per controllare i parametri di sistemi particellari da script Blueprint. Successivamente, vengono presentati i metodi per leggere i dati da un foglio di calcolo Google Sheets e usare questi dati per controllare in real-time sia sistemi particellari, sia sistemi di rendering audio. Vengono presi in considerazione e analizzati due tipi di design possibile. Il primo caso prevede la lettura continuativa dei dati e un conseguente aggiornamento in tempo reale dei parametri dei sistemi controllati. Il secondo caso prevede un'organizzazione sequenziale dei dati, la quale fa sì che i sistemi si evolvano secondo la sequenza scritta nella tabella. Nell'elaborato sono anche esposti alcuni esempi applicati di questi metodi e anche alcune buone pratiche, come l'uso di interpolazioni per evitare di creare discontinuità. Inoltre, vengono mostrati i metodi con cui sono stati svolti alcuni test di performance svolti sugli script creati in questa fase, i quali hanno evidenziato quali sono i limiti in termini di latenza e quantità di dati.

L'ultimo capitolo rappresenta l'analisi un progetto realizzato a scopo dimostrativo, nel quale vengono controllati contemporaneamente sia sistemi di particelle, sia sistemi di generazione audio. Il soggetto della demo è una tempesta di sabbia, realizzata tramite Niagara, e nella scena è possibile sentire sia il sound effect del vento, creato in maniera procedurale, sia elementi musicali di sintesi. Inoltre, viene presentato un esempio di design più complesso, in cui i dati presenti nella tabella online sono in numero molto inferiore rispetto ai parametri controllati nel progetto,

per cui un dato influenza contemporaneamente più di un parametro e un parametro può essere influenzato da più di un dato. Questo design è pensato per replicare una situazione nella quale all'utente finale venga nascosta la complessità dei sistemi e possa controllarli più facilmente attraverso l'aggiunta di un livello di astrazione. Inoltre, vengono usati diversi metodi di interpolazione, il cui uso è reso necessario dalla presenza di parametri che controllano proprietà di natura diversa.

Infine, è consigliato che eventuali sviluppi futuri si concentrino, per quanto riguarda il lato tecnico, sull'ottimizzazione delle soluzioni illustrate nella tesi, e sulla sperimentazione di altri tipi di basi di dati. Per quanto riguarda il design, si raccomanda di ricercare metodi più astratti di visualizzazione dei dati, e di integrare una componente interattiva ai metodi descritti.

Indice

Elenco delle figure

Capitolo 1

Introduzione

Negli ultimi anni, i dati hanno assunto un'importanza sempre maggiore in ambiti molto diversi tra loro, ad esempio nel settore delle analisi predittive o per la personalizzazione dell'esperienza utente. Uno degli aspetti in cui l'uso di dati è diventato essenziale è la ricerca, dove anche le discipline umanistiche hanno visto una importante crescita nella pubblicazione di articoli che adottano un approccio basato sui dati[1]. Questo fenomeno ha fatto sì che anche la *data visualization* abbia assunto importanza, per cui trovare nuovi metodi per visualizzare e rendere comprensibili le informazioni estratte da dataset complessi rappresenta una sfida sempre più comune. Un altro effetto dello sviluppo delle tecnologie digitali è il crescente uso di arte generativa, ovvero la pratica di definire degli algoritmi che hanno il compito di generare un'opera d'arte. Nel caso di questo lavoro di tesi, il campo di applicazione è quello museale, e si inserisce all'interno di un progetto che ha lo scopo di realizzare un'installazione immersiva, che abbia al suo interno degli elementi generativi e data-driven.

Questa tesi fornisce un metodo per permettere a dei dati di controllare in tempo reale dei parametri che definiscono le caratteristiche e i comportamenti di sistemi audio e video. Il metodo descritto prevede l'uso del motore di gioco Unreal Engine, che racchiude al suo interno sia i sistemi di creazione di Visual Effects e audio, sia il sistema di scripting per poterli controllare. Per l'accesso ai dati è stato usato il software di fogli di calcolo online Google Sheets. Il workflow descritto in questa tesi descrive i metodi per:

- Creare dei Visual Effects o sistemi audio i cui parametri possano essere modificati in tempo reale.

- Definire internamente all'engine la struttura dei dati, così che possano essere interpretati correttamente.

- Definire degli script che leggano tali dati da un foglio online e li applichino secondo una certa mappatura ai sistemi audio e video, che saranno renderizzati in tempo reale.

Vengono inoltre forniti degli esempi di uso di questi metodi, e delle buone pratiche da seguire per avere dei risultati di alta qualità.

L'uso di questo workflow, unito a un design curato e a una sufficiente potenza di calcolo, permette di creare delle esperienze multimediali che reagiscono in tempo reale con dei dati presenti online, con possibili applicazioni in ambito museale e artistico.

1.1 Il progetto Egitto Immersivo

Il presente lavoro di tesi si colloca all'interno del progetto Egitto Immersivo del Museo Egizio di Torino. Il progetto prevede la realizzazione di due sale immersive che verranno aperte al pubblico nel 2025. La realizzazione del progetto è stata affidata a Robin Studio, uno studio di produzione video con sede a Torino. Le due sale previste hanno due concept distinti e seguiranno due flussi di storytelling distinti. Parte dei contenuti dell'installazione verranno renderizzati in tempo reale tramite Unreal Engine e proiettati sulle pareti delle due sale.

1.1.1 Sala 1

La Sala 1 ha come concept principale quello di esplorare il processo di ricerca attuato dal Museo sugli oggetti che fanno parte della sua collezione. In particolare, verrà dato un focus sulla ricostruzione del rapporto che tali oggetti hanno avuto con il paesaggio da cui essi provengono. In tale senso, la ricerca archeologica attua una decontestualizzazione e interrompe la relazione fra oggetto e paesaggio. Il

compito del museo può quindi essere visto come quello di ri-contestualizzare tali oggetti anche attraverso la creazione di sale immersive.

Nella sala sarà presente un'unica vetrina, all'interno di una colonna posta al centro della sala, e la narrazione si concentrerà su un singolo oggetto, ovvero un vaso di terracotta con scene di ambientazione sul Nilo[1].

La narrazione sarà strutturata come un "flusso di coscienza" da parte del Museo, e sarà divisa in tre segmenti:

- Il primo segmento riguarderà l'evoluzione degli strumenti di analisi degli oggetti
- Il secondo segmento riguarderà il territorio egiziano, sottolineando il modo in cui le sue caratteristiche naturali hanno condizionato le comunità che vi risiedevano. Verrà poi esplorato il modo in cui il territorio si è evoluto, attraverso i casi studio di Hammamija e Gebelein.
- Il terzo segmento estenderà il problema analizzato nel precedente segmento all'intero processo di conoscenza, attraverso la generazione di contenuti a partire da dataset sulle pubblicazioni scientifiche sull'Egitto.

1.1.2 Sala 2

Il soggetto principale della Sala 2 sarà il ciclo delle stagioni in un anno, che coincide con il ciclo di inondazioni del Nilo.

Nella sala saranno presenti delle vetrine disposte su tre pareti, ognuna delle quali corrisponderà a una delle tre stagioni dell'antico Egitto (akhet, peret e shemu). Le vetrine conterranno degli oggetti relativi alla vita agricola.

Per ognuna delle tre stagioni verranno illustrati gli eventi naturali che la caratterizzano, e come questi abbiano influenzato la vita degli abitanti di quei territori. Tali eventi saranno messi in relazione con gli oggetti esposti nelle vetrine.

[1] `https://collezioni.museoegizio.it/it-IT/material/S_4705/`

Capitolo 2

Unreal Engine

Per lo sviluppo del progetto Egitto Immersivo, è stato scelto come software principale il game engine *Unreal Engine 5*. I motivi di questa scelta sono molteplici:

- Unreal Engine offre la possibilità di eseguire rendering real-time di scene anche molto complesse. Inoltre, il nuovo sistema di geometria Nanite permette di renderizzare in tempo reale mesh con un numero di poligoni molto alto.
- Unreal Engine ha una licenza gratuita per progetti indipendenti, poiché impone una royalty del 5% solo per progetti i cui guadagni superano la soglia di incassi di $1 milione.
- La documentazione su Unreal Engine è molto estesa e la comunità online condivide una grande quantità di risorse.
- Unreal Engine permette di creare degli script tramite cui automatizzare processi o creare comportamenti più o meno complessi.
- Il sistema di shading di Unreal Engine è molto versatile e permette di personalizzare lo stile visivo.
- Unreal Engine permette di accedere a Quixel, una libreria molto ampia di asset di fotogrammetria.
- Unreal Engine ha al suo interno diversi sistemi e tool, che possono comunicare fra loro. Questo permette di avere un workflow che preveda un uso limitato degli altri software.

Unreal Engine è un motore di gioco 3D la cui prima versione è stata sviluppata nel 1998 da Tim Sweeney, fondatore della casa di sviluppo Epic Games. Attualmente, Unreal Engine è alla sua quinta versione, rilasciata nel 2022. Nello specifico, il lavoro di tesi è stato realizzato nella versione 5.3, rilasciata a settembre 2023. Le due maggiori innovazioni portate dal rilascio di Unreal Engine 5 sono Lumen e Nanite. Lumen è un sistema di illuminazione globale che permette di definire un'illuminazione dinamica nella scena, che esegue il calcolo dei *light bounces* (ovvero le interazioni luminose fra gli oggetti della scena) in real-time. Questo si traduce in un'illuminazione fotorealistica senza bisogno di effettuare operazioni di *baking*. Nanite è un sistema di geometria virtualizzata che sfrutta un formato di mesh interno per renderizzare dettagli fino alla scala del pixel e gestire in maniera automatica i *Level of Detail* (LOD). In questo modo, è possibile renderizzare in real-time scene che contengono mesh con un numero di poligoni nell'ordine delle decine di milioni, come ad esempio mesh provenienti da software di *sculpting* o di fotogrammetria che non sono state sottoposte a un processo di *retopology*. La combinazione di queste due innovazioni rende possibile effettuare rendering real-time fotorealistici, e questo è uno dei principali motivi per cui negli ultimi anni Unreal Engine ha espanso il suo mercato e ha cominciato a diffondersi anche nell'industria cinematografica, nell'animazione, nell'industria dell'automotive, nell'architettura e nell'Extended Reality. Nel corso di questo capitolo verrà illustrata la struttura di Unreal Engine, e poi si approfondiranno gli aspetti più rilevanti per la realizzazione di questo progetto di tesi, ovvero i Materiali, Niagara, Blueprint e MetaSounds.

2.1 Terminologia

In questo paragrafo si stilerà una lista dei termini appartenenti al lessico di Unreal Engine presenti in questa tesi. Parte delle informazioni riportate sono state raccolte dall'Unreal Engine Glossary, presente nella documentazione online[1]:

- **Actor**: qualsiasi oggetto che possa essere posizionato nel livello e sottoposto a trasformazioni 3D. Alcuni esempi di tipi di *Actor* sono *Static Mesh*, *Skeletal Mesh*, *Cine Camera Actor*, *Directional Light* o *Post Process Volume*.

[1] `https://docs.unrealengine.com/5.3/en-US/unreal-engine-glossary/`

- **Asset**: termine generico che indica i contenuti usati per realizzare un progetto. In Unreal Engine, quando si crea o si importa un qualsiasi tipo di asset, questo viene salvato o convertito nel formato interno *.uasset*, che gestisce i dati in modo diverso dai formati comuni. Ad esempio, le Texture vengono salvate sotto forma di *bitmap*. Per questo motivo, l'importazione e l'esportazione di asset deve essere effettuata da Unreal Engine, e non può essere fatta dal *file explorer*. Alcuni esempi di tipi di asset sono i Materiali, le classi Blueprint, tutti i tipi di *Actor* o i *Niagara System*.

- **Blueprint**: indica il Blueprint Visual Scripting, ovvero il sistema di scripting di Unreal Engine basato sui nodi. Solitamente con "Blueprint" si intende anche una Blueprint Class, ovvero asset che contiene una classe di Blueprint Visual Scripting e che definisce un certo comportamento.

- **Livello**: area definita dall'utente contenente tutti gli oggetti che si possono vedere e/o con cui si può interagire. I livelli vengono salvati con estensione *.umap* e per questo vengono chiamati anche "mappe"

- **Materiale**: asset che definisce il modo in cui la superficie di un oggetto interagisce con la luce.

- **Niagara**: sistema di creazione di Visual Effects in Unreal Engine che permette di creare sistemi particellari visualizzabili in real-time.

- **Pawn**: sottoclasse di Actor che indica un attore che può essere controllato dal giocatore o da una AI.

2.2 Interfaccia di Unreal Engine 5

In questo paragrafo verranno illustrati gli elementi più comuni dell'interfaccia di Unreal Engine 5. Si farà riferimento alla documentazione ufficiale online [2].

Segue una breve analisi delle sue componenti, numerati come nella Figura 2.1:

[2] `https://docs.unrealengine.com/5.3/en-US/unreal-editor-interface/`

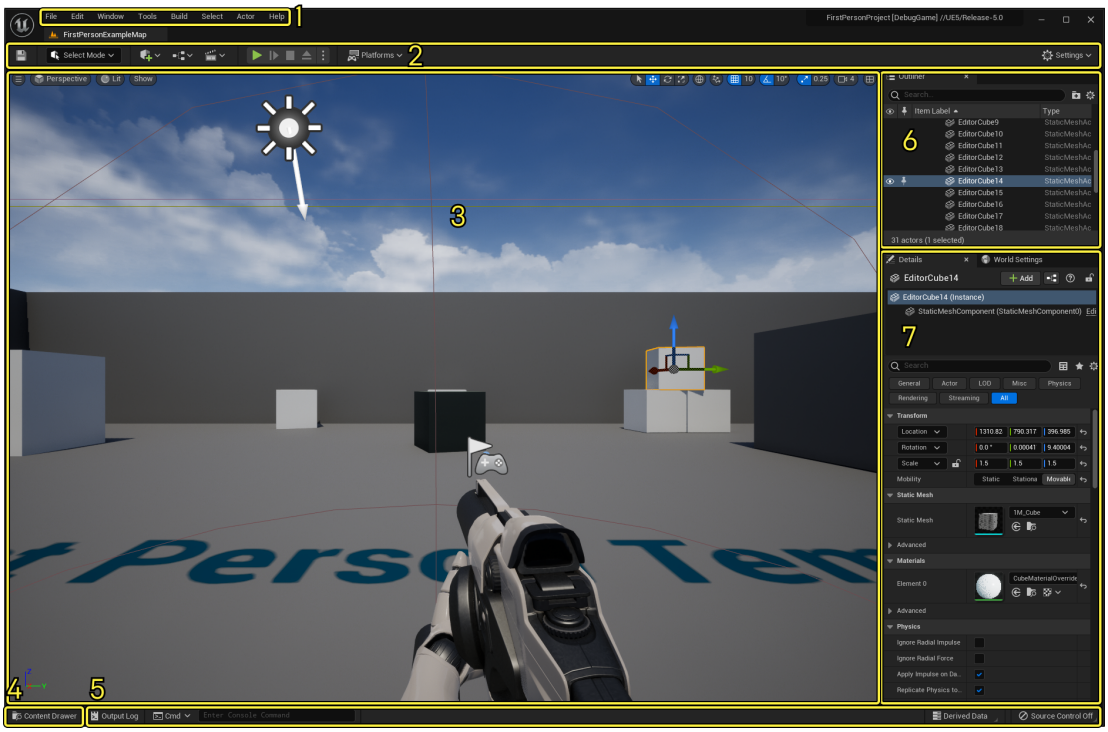


Figura 2.1: Interfaccia di default dell'Editor di Unreal Engine 5

1. **Barra dei Menu**: contiene i menu specifici dell'editor, come *Window*, che permette di mostrare o nascondere finestre, *Edit*, che permette di accedere alle preferenze, *Build*, che contiene gli strumenti per effettuare build delle luci o delle texture o *Actor*, che permette di eseguire operazioni sull'oggetto selezionato.

2. **Toolbar Principale**: contiene *shortcut* per gli strumenti più utilizzati in Unreal Engine e la possibilità di entrare in *Play mode*, ovvero la simulazione real-time del progetto. Sono inoltre presenti strumenti di gestione delle diverse piattaforme di destinazione. Gli strumenti presenti nella Toolbar sono:

   - Pulsante di salvataggio del livello corrente.
   - Selezione della modalità di interazione: permette di scegliere tra *Select Editing*, *Landscape Editing*, *Foliage Editing*, *Mesh Painting*, *Modeling*, *Fracture Editing*, *Brush Editing* e *Animation*.

- Actor: permette di aggiungere rapidamente i principali tipi di Actor nel livello o di importarli dalla memoria del computer, dall'Unreal Marketplace, o da Quixel Bridge.
- Blueprint: permette di creare nuove classi di Blueprint, aprire la Blueprint del livello o convertire uno o più attori in Blueprint.
- Level Sequence: permette di creare sequenze video all'interno del livello.

3. **Viewport del Livello**: mostra i contenuti presenti nel livello attualmente aperto. Sono inoltre presenti strumenti per il monitoraggio delle prestazioni, scelta fra visualizzazione prospettica e ortografica, scelta della View Mode (ovvero il modo in cui si visualizza il livello, principalmente a fini di debug), scelta dei tipi di oggetti da visualizzare e strumenti di trasformazione e posizionamento degli Actors.

4. **Content Drawer**: permette di aprire il Content Drawer, ovvero la raccolta di tutti gli asset presenti nella cartella Content del progetto. Il Content Drawer è organizzato gerarchicamente, in cartelle e sottocartelle.

5. **Toolbar inferiore**: contiene gli *shortcuts* per strumenti di controllo per sviluppatori, come la *Command Console*, l'*Output Log* e la *Derived Data Cache*.

6. **Outliner**: visualizzazione degli Actor presenti nel livello. È possibile nascondere determinati Actor o raggrupparli.

7. **Details**: mostra le proprietà dell'Actor selezionato. Le proprietà mostrate variano a seconda del tipo di Actor selezionato.

2.3 Materiali

In questo paragrafo verrà fornita un'introduzione alla creazione e gestione dei Materiali in Unreal Engine. Verrà fatta una panoramica sull'interfaccia del Material Editor, poi si vedranno dei cenni sul funzionamento di base dello shading, basato sull'approccio del Physically Based Rendering, in seguito si vedranno alcuni dei

nodi più importanti e largamente usati e infine verranno date le basi su come creare e gestire le Material Instance.

Come detto in precedenza, i Materiali definiscono il modo in cui la superficie di un oggetto interagisce con la luce. Questo viene fatto tramite shader, ovvero algoritmi che simulano in maniera più o meno realistica il comportamento di un materiale reale. Unreal Engine 5 gestisce la creazione di shader personalizzati mediante un sistema a nodi. Nel Material Editor, uno shader è rappresentato come un grafo in cui l'utente può aggiungere nodi e collegarli l'uno con l'altro. Ogni nodo del Material Editor contiene una porzione di codice di **High-Level Shader Language** (**HLSL**). Ogni nodo ha almeno un *pin* di ingresso e/o almeno un *pin* di uscita. Tali *pin* corrispondono agli input e output del codice HLSL presente nel nodo, e permettono di far comunicare porzioni di codice e creare shader più complessi. I nodi possono essere sia **Material Expression**, sia **Material Functions**. Le Material Expression sono set di istruzioni statici, e solitamente svolgono operazioni semplici. Le Material Function sono funzioni contenenti Material Expression e possono essere create e modificate dall'utente.

Nella figura 2.2 si possono osservare alcuni esempi di nodi presenti nel Material Editor.

2.3.1 Interfaccia del Material Editor

In questo sottoparagrafo verranno illustrati gli elementi principali dell'interfaccia del Material Editor di Unreal Engine 5. Si farà riferimento alla documentazione ufficiale online[3].

La Figura 2.3 illustra i principali elementi dell'interfaccia dell'Editor di Unreal Engine 5. Segue una breve analisi delle sue componenti, numerati come nella Figura 2.3:

1. **Barra dei Menu**: contiene i menu dell'Editor.

2. **Toolbar**: contiene *shortcut* per gli strumenti più utilizzati nel Material Editor. Permette di salvare il Materiale, trovarlo all'interno del Content Browser,

[3] `https://docs.unrealengine.com/5.3/en-US/unreal-engine-material-editor-ui/`

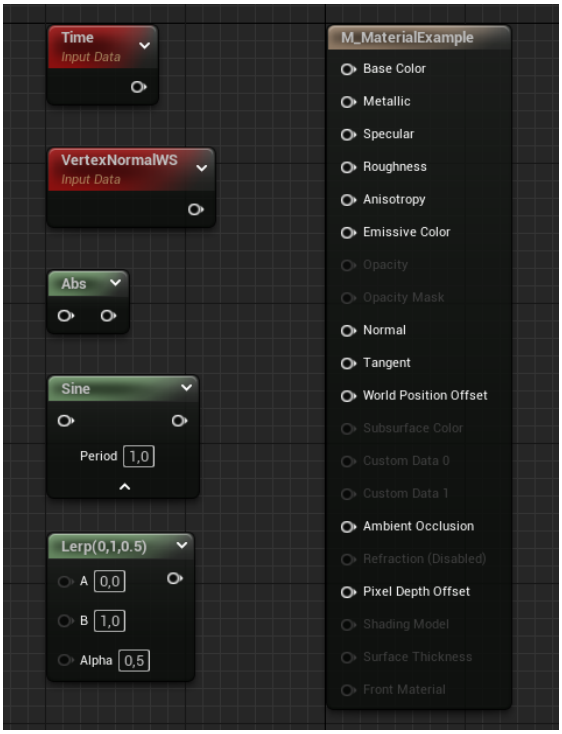


Figura 2.2: Esempi di nodi nel Material Editor di Unreal Engine 5

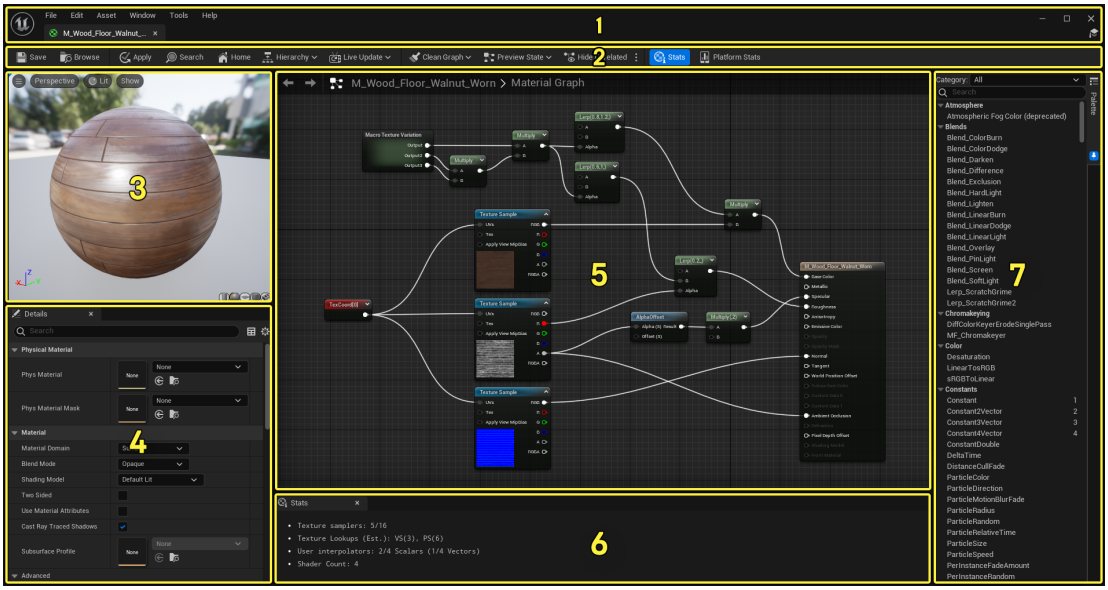


Figura 2.3: Interfaccia del Material Editor di Unreal Engine 5

compilare il codice, organizzare il grafo, visualizzare la gerarchia del Materiale e accedere ad alcune statistiche.

3. **Viewport**: visualizzazione della preview del Materiale che si sta creando applicata a una mesh. Da qui è possibile cambiare le opzioni di visualizzazione e modificare la mesh su cui si sta applicando la preview del Materiale.

4. **Details**: mostra le proprietà del nodo selezionato o le proprietà di base del Materiale.

5. **Grafo del Materiale**: finestra di visualizzazione e modifica del grafo.

6. **Stats**: contiene statistiche di base sul Materiale, ad esempio il numero di istruzioni del codice HLSL, ed eventuali errori di compilazione.

7. **Palette**: lista organizzata in categorie dei nodi che si possono aggiungere al Materiale.

2.3.2 Cenni sul Physically Based Rendering

Unreal Engine, per la creazione di Materiali, adotta l'approccio del Physically Based Rendering (PBR), ovvero una metodologia di shading basata sulla simulazione del comportamento fisico di interazione fra luce e superfici. Questo approccio sostituisce i precedenti modelli di shading, che venivano creati *ad hoc* e con parametri più o meno arbitrari e ha il vantaggio di dare risultati realistici con qualsiasi tipo di illuminazione. Il modello di shading PBR usato maggiormente oggi deriva da quello sviluppato da Disney per il film d'animazione Wreck-It Ralph (titolo italiano Ralph Spaccatutto), dove ha portato miglioramenti sia in termini qualitativi, sia in termini di workflow[2]. I tre tipi di interazione principali simulate dal Physically Based Rendering sono[3]:

- **Assorbimento**: fenomeno per cui il mezzo assorbe una certa quantità della luce che transita attraverso esso. Riguarda i mezzi omogenei, ovvero con indice di rifrazione costante nella scala della lunghezza d'onda della luce visibile, e con alto indice assorbimento, ovvero la parte immaginaria dell'indice di rifrazione.

- **Scattering**: fenomeno che ha luogo nei punti in cui l'indice di rifrazione ha una variazione improvvisa e su distanze brevi in un mezzo eterogeneo. In questo caso la luce si "spezza" in componenti lungo molteplici direzioni.

- **Emissione**: fenomeno per cui la luce viene creata a partire da altre forme di energia.

Inoltre, si deve tenere conto del *subsurface scattering*, ovvero il fenomeno per cui la luce, dopo essere stata rifratta in un mezzo, va incontro a scattering all'interno di esso, e una porzione di questa luce viene nuovamente diffusa verso l'esterno del mezzo. Tale effetto riguarda esclusivamente materiali non metallici.

Per simulare la ruvidità dei materiali, viene utilizzata la *microfacet theory*, che descrive le superfici con irregolarità di dimensioni maggiori della lunghezza d'onda della luce visibile ma minori della grandezza di un pixel, quindi impossibili da rappresentare, ad esempio, con geometria o *normal mapping*. Si assume che le irregolarità siano localmente piatte dal punto di vista ottico, per cui per ognuna di esse sarà possibile ricavare una soluzione analitica tramite le Equazioni di Fresnel, che descrivono le direzioni di raggio riflesso e raggio rifratto. Ognuna di queste piccole superfici piatte è detta "*microfacet*". Sulla base di queste assunzioni, è possibile ricavare una funzione $f_{\mu facet}$ che dipende dal prodotto della riflettività del materiale (chiamata riflettività di Fresnel), della porzione delle *microfacets* visibili nel punto di vista della camera e dalla distribuzione statistica delle direzioni delle normali alle *microfacet*. Nei modelli di shading che adottano i principi del PBR, la presenza di *microfacet* è regolata con il parametro di **roughness**

La riflessione di questo tipo è molto più marcata nei metalli, che riflettono dal 50% al quasi 100% della luce incidente, rispetto ai non metalli, che solitamente riflettono fra il 2% e il 5% e sono maggiormente soggetti a scattering. Per questo motivo, quando si parla di colore del materiale, si parla di Specular Color per i metalli, Diffuse Color per i non metalli.

Inoltre, la riflettività di Fresnel dipende fortemente dall'angolo compreso fra la direzione del raggio di luce incidente e la normale della superficie: essa rimane circa costante fra gli 0° e i 45° a un valore $F(0°)$, chiamato Riflettanza a 0°, e poi ha una crescita esponenziale fino a raggiungere il valore di 1 a 90°. Questo andamento è detto Effetto di Fresnel ed è valido per tutti i tipi di materiale. La Riflettanza a

0° è dipendente dalla lunghezza d'onda, quindi determina il colore della superficie metallica.

Da questi punti di partenza, si costruisce una funzione di distribuzione che descrive la riflettanza, chiamata BRDF (Bidirectional Reflectance Distribution Function), che, per essere fisicamente accurata, deve soddisfare il principio di conservazione dell'energia (ovvero la luce riflessa non può essere più intensa di quella incidente) e di reciprocità (ovvero scambiando i raggi incidente e riflesso il risultato rimane invariato).

In conclusione, per creare un modello di shading fisicamente realistico, occorre tenere conto di tutti i fenomeni sopra elencati. Deve quindi seguire il principio di conservazione dell'energia, gestire la differenza di comportamento fra superfici metalliche e non metalliche, simulare l'effetto di Fresnel e calcolare l'intensità delle riflessioni a partire dalla BRDF, dalla *roughness* del materiale e dal valore di $F(0\check{r})$. Oltre alla complessità del modello fisico, occorre anche fare in modo che gli utenti abbiano la possibilità di creare tutta la varietà possibile di materiali a partire da un numero limitato di parametri.

2.3.3 Il Main Material Node

Il Main Material Node rappresenta il nodo più importante in un qualsiasi Materiale in Unreal Engine. Esso raccoglie tutti gli input provenienti dagli altri nodi del grafo e li elabora per definire l'aspetto finale del Materiale. L'aspetto del Materiale dipende anche dalle opzioni impostate nei dettagli del nodo. Unreal Engine adotta l'approccio del Physically Based Rendering a partire dalla sua quarta versione e basa il suo design su una versione semplificata di quello proposto da Disney e ottimizzata per il rendering real time[4].

Gli input principali del Main Material Node che si rifanno ai metodi PBR sono:

- **Base Color**: colore della luce diffusa riflessa dal materiale, escludendo le riflessioni speculari.

- **Metallic**: definisce se la superficie deve avere un comportamento metallico o no. Nella maggior parte dei casi ha un valore di 0 (puramente non metallica) o 1 (puramente metallica).

- **Specular**: regola la quantità di luce riflessa dal materiale.
- **Roughness**: regola la levigatezza del materiale. Più è alta, più le riflessioni saranno sfocate a causa dello scattering.

Nel Main Material Node sono presenti altri nodi, come **Emissive Color**, che regola il colore e l'intensità della luce emessa dal Materiale, **Normal**, che simula dettaglio sulla superficie e **World Position Offset**, che permette di impostare una traslazione dei vertici della mesh.

Nei dettagli del Main Material Node, è inoltre possibile definire tre proprietà che determinano il comportamento di base del Materiale:

- **Blend Mode**: definisce il modo in cui il Materiale si mescola ai pixel dietro di esso. Alcuni esempi sono:
  - Opaque: descrive una superficie totalmente opaca, per cui i colori sullo sfondo non sono visibili
  - Translucent: descrive una superficie più o meno trasparente. Ha un costo computazionale abbastanza elevato. Ad esempio, può essere usato per simulare una superficie di vetro.
  - Masked: permette di specificare una maschera che descrive in maniera binaria le parti completamente opache e quelle completamente trasparenti. Ha un costo computazionale minore di Translucent.
  - Additive: effettua un'operazione di somma tra i pixel del Materiale e quelli dello sfondo. Per questo motivo, non può dare scurimento e il nero viene interpretato come trasparente. Ad esempio, può essere usato in effetti particellari che simulano il fuoco.
- **Shading Model**: definisce il modello di interazione fra la luce e la superficie. In altri termini, definisce il modo in cui processare gli input per determinare il colore finale del Materiale. Alcuni tra gli Shading Model usati più di frequente sono:
  - Default Lit: modello di default, e usato per la maggior parte delle superfici. Interagisce con l'illuminazione diretta e indiretta e supporta le riflessioni

speculari. Si specifica sia il colore della superficie (Base Color), sia, eventualmente, il colore di emissione (Emissive Color).

  - Unlit: descrive superfici puramente emissive, la cui superficie non proietta ombre. Viene specificato solo l'Emissive Color. È un modello molto usato nei VFX.
  - Subsurface: simula il fenomeno di *subsurface scattering*, per cui la luce, dopo essere penetrata in un oggetto, diffonde all'interno di esso. Viene specificato sia il colore della superficie, sia il Subsurface Color, ovvero il colore di tale diffusione. Viene usato per superfici quali la pelle, le candele di cera e il ghiaccio.

- **Material Domain**: indica il tipo di asset su cui si userà il Materiale. Alcuni esempi sono Surface (Materiali applicati sulla superficie di una mesh), Post Process (Materiali applicati in un Post Process Material, usati per dare look personalizzati al mondo) o Light Function (Materiali applicati alle luci)

La scelta di queste tre proprietà determinano anche quali tra i *pin* del Main Material Node vengono resi accessibili.

2.3.4 Tipi di dato nei Materiali

Nella creazione di uno shader, è fondamentale comprendere il tipo di dato che si sta trattando e il modo in cui lo si sta manipolando. Bisogna quindi tenere sempre in considerazione quali siano gli output di un certo nodo e gli input richiesti da un altro. Nel Material Editor di Unreal Engine, i quattro tipi di dato più frequentemente usati sono:

- **Float**: rappresenta un singolo numero in rappresentazione *floating point*. Il nodo **Constant** permette di definire una costante di tipo float. I float sono usati, ad esempio, per descrivere proprietà come roughness o metallic o per eseguire operazioni aritmetiche.

- **Float2**: un array di due float. Il nodo **Constant2Vector** definisce una costante di tipo float2. Sono usati principalmente per la manipolazione delle coordinate UV.

- **Float3**: un array di tre float. Il nodo **Constant3Vector** definisce una costante di tipo float3. Sono usati principalmente per contenere informazioni di colore (in formato RGB) o di coordinate tridimensionali (interpretate come XYZ).
- **Float4**: un array di quattro float. Il nodo **Constant4Vector** definisce una costante di tipo float4. Sono principalmente usati in relazione a texture o colori che contengono il canale alfa.

Di default, Unreal Engine applica una nomenclatura di tipo RGBA ai dati float, per cui, ad esempio, un float2 sarà interpretato come un dato contenente i due canali R e G. In alcuni casi può essere utile selezionare solo alcuni degli elementi dei vettori, o combinare due o più dati per ottenere un vettore di dimensioni maggiori. Per tali scopi, sono presenti i nodi **AppendVector** e **AppendMany**, che eseguono un'operazione di Append sugli input, e il nodo **ComponentMask**, che permette di selezionare uno o più elementi di un vettore.

2.3.5 Gestione delle Texture e delle UV

Nella creazione di un Materiale, risulta quasi sempre essenziale utilizzare delle texture che descrivano le variazioni delle sue proprietà. È inoltre fondamentale avere controllo sulle coordinate UV, ovvero la proiezione della superficie dell'oggetto 3D su una superficie bidimensionale.

Per importare una texture nel Material Editor, è necessario utilizzare il nodo **Texture Sample**, il quale riceve in input una texture e restituisce in output i valori di colore di essa. La texture su cui effettuare l'operazione di *sampling* può sia essere specificata nei Dettagli del nodo, sia essere data in input tramite il *pin* Tex, ad esempio connettendovi un nodo **Texture Object**, che non effettua nessuna operazione di *sampling*.

Per quanto riguarda la manipolazione delle coordinate UV, è necessario usare il nodo **TextureCoordinate**, che dà in output le coordinate UV sotto forma di float2. Le operazioni più comuni di manipolazione delle coordinate sono le operazioni di *tiling* e *offsetting*, ovvero la modifica della scala e della posizione delle coordinate. Per effettuare tali operazioni, è necessario applicare a ognuna delle due coordinate UV un'operazione di moltiplicazione e una di addizione. L'operazione di

moltiplicazione controllerà il *tiling*, ovvero la scala della proiezione nello spazio UV. Valori di tiling maggiori di 1 faranno apparire la texture più piccola. L'operazione di addizione controllerà l'*offset*, ovvero la traslazione dell'origine dello spazio UV. Per trattare le due coordinate separatamente è necessario l'utilizzo di due nodi Component Mask, e per riportare i dati nel formato corretto è necessario l'utilizzo di un nodo AppendVector. Nella figura 2.4 è possibile osservare un esempio di porzione di network per il controllo delle coordinate UV.

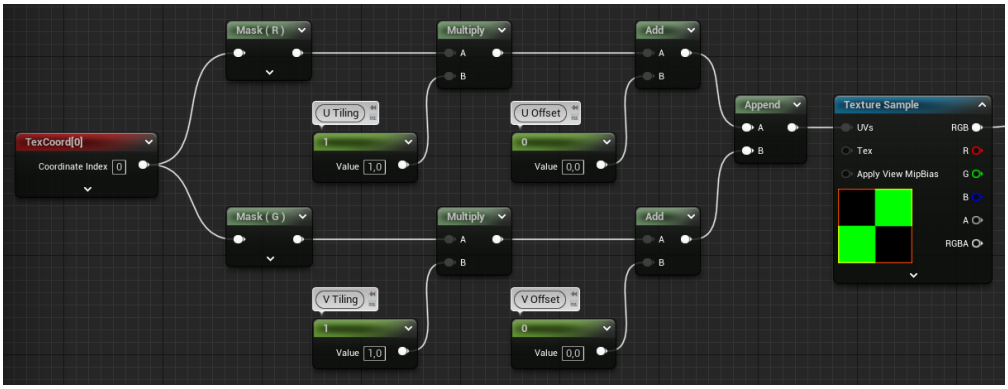


Figura 2.4: Esempio di controllo delle coordinate UV

2.3.6 Nodi di Matematica

Nella creazione di uno shader, è frequente avere la necessità di eseguire operazioni matematiche più o meno complesse sui dati. Il Material Editor offre una serie di nodi che hanno lo scopo di eseguire tali operazioni. Di seguito sono elencati i principali di essi:

- **Operazioni aritmetiche**: ne fanno parte i nodi **Add**, **Subtract**, **Multiply** e **Divide**, che eseguono l'operazione corrispondente fra i due input. Sono inoltre presenti i nodi **Power**, che esegue l'operazione di elevamento a potenza fra gli input di base ed esponente, e **SquareRoot**, che calcola la radice quadrata dell'input. Il nodo **Abs** calcola il valore assoluto dell'input. Il nodo **Fmod** esegue l'operazione *modulo* (resto della divisione) fra i due input. Un caso particolare è il nodo **OneMinus** calcola la differenza fra 1 e l'input, e viene usato soprattutto per invertire l'effetto di maschere o in generale qualsiasi quantità che abbia come range 0-1.

- **Funzioni goniometriche**: i nodi più comuni sono **Sine** e **Cosine**, usati principalmente per dare periodicità a un certo comportamento dello shader. Sono inoltre presenti il nodo **Tangent** e le funzioni inverse **Arcsine**, **Arccosine** e **Arctangent**, di cui esistono anche le versioni approssimate e con minore costo computazionale, e hanno suffisso "Fast".

- **Funzioni di Minimo e Massimo**: accessibili tramite i nodi **Min** e **Max**, selezionano il minimo o il massimo tra i due input. Sono usati, ad esempio, per effettuare operazioni di *Lighten* o *Darken* tra due colori in input.

- **Funzioni di *remapping***: il nodo **Normalize** normalizza il vettore in input, il nodo **Clamp** limita il valore dell'input fra due valori di minimo e massimo definiti dall'utente e il nodo **Saturate** esegue un'operazione simile a Clamp, ma ha i valori di minimo e massimo sono fissati a 0 e 1 e ha minore costo computazionale.

- **Operazioni Vettoriali**: i nodi **DotProduct** e **CrossProduct** eseguono, rispettivamente, il prodotto scalare e vettoriale dei vettori di input.

- **Funzioni di parte intera e frazionaria**: i nodi **Floor** e **Ceil** calcolano rispettivamente la parte intera e la parte intera superiore dell'input, il nodo **Frac** ne calcola la parte frazionaria.

2.3.7 Il nodo Lerp

Il nodo **LinearInterpolate**, solitamente abbreviato con **Lerp**, esegue un'interpolazione lineare fra due valori A e B secondo una maschera Alpha. Quando Alpha vale 0, l'output vale A, quando Alpha vale 1, l'output vale B. Per valori intermedi, l'output è calcolato come un'interpolazione lineare tra A e B secondo una pesatura proporzionale al valore di Alpha.

Un esempio di utilizzo è l'interpolazione fra due valori di roughness secondo una texture di rumore. Un altro esempio è il *blend* di due texture secondo l'orientamento in coordinate globali delle *vertex normal* della mesh, così da avere una texture che dipende dall'orientamento delle facce. Quest'ultimo esempio è mostrato nella figura 2.5.

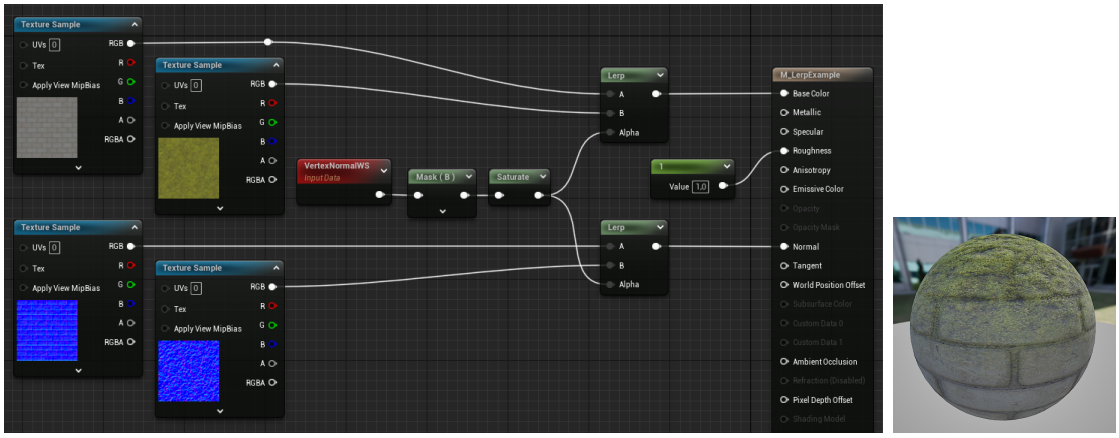


Figura 2.5: Esempio di utilizzo del nodo Lerp

2.3.8 Material Instance e Parametri

Per applicare una modifica di qualsiasi tipo a un Materiale in Unreal Engine, è necessario compilare lo shader. Talvolta, è necessario fare modifiche iterative per raggiungere il look desiderato, e questo processo può occupare molto tempo, specie nel caso di Materiali complessi, i cui tempi di compilazione sono più lunghi. Inoltre, la creazione di più Materiali molto simili tra loro non è efficiente dal punto di vista della memoria e delle performance, poiché ognuno di essi viene renderizzato separatamente.

Per risolvere questo problema, in Unreal Engine è possibile usare le Material Instance, che sfruttano il concetto di ereditarietà, per il quale le proprietà di una classe genitore vengono trasmesse ai figli. Una Material Instance, quindi, esiste solo in relazione al Materiale da cui eredita le proprietà, chiamato **Parent**. Una volta compilato il Materiale Parent e applicata la Material Instance sull'oggetto, sarà possibile modificare i parametri di quest'ultima e tali modifiche verranno applicate in real-time, senza che sia necessario effettuare alcuna compilazione. Questo presenta vantaggi sia nel workflow di artisti che si occupano del look, poiché i tempi di modifica delle proprietà del Materiale si accorciano, sia per quanto riguarda l'ottimizzazione della memoria e delle performance.

In una Material Instance, è possibile modificare solo le proprietà che l'utente ha scelto di esporre. Tali proprietà esposte alle Instance sono dette Parametri. Non è invece possibile modificare la struttura del network di nodi. Per introdurre un

Parametro in un Materiale, è possibile sia promuovere una costante, sia creare un nodo Parametro *ex novo*. Inoltre, è possibile creare delle Material Instance dinamiche, i cui parametri possono essere modificati runtime, ad esempio tramite l'uso di Blueprint. Nel Material Editor sono presenti diversi tipi di Parametri:

- **Scalar Parameters**: Parametri che contengono un singolo valore floating-point.
- **Vector Parameters**: Parametri che contengono dati di tipo float2, float3 o float4.
- **Texture Parameters**: Parametri che contengono informazioni di texture. Il più comune è **TextureSampleParameter2D**, ovvero l'equivalente parametrico del nodo Texture Sample.
- **Static Parameters**: Parametri che non possono essere cambiati *runtime*, ma solo in Editor. Un esempio è lo **Static Switch Parameter**, che permette di scegliere tra due input in base a un valore booleano.

Il workflow tipicamente usato è quello di creare un Master Material, ovvero un Materiale genitore, per ogni macrocategoria di Materiale presente nel progetto. In questo Materiale verranno inseriti dei Parametri che consentiranno di modificare le proprietà delle Instance. La tipologia e il numero di Master Material da realizzare dipende dal progetto. Solitamente, seguendo questo tipo di workflow, il Master Material non viene applicato su nessun oggetto, poiché ha il solo scopo di generare delle Instance. Nella figura 2.6 possiamo vedere un esempio di Materiale parametrico.

2.4 Niagara

In questo paragrafo verrà fornita una panoramica del Niagara VFX System e un'introduzione alla creazione di effetti particellari. Si partirà da una breve analisi della struttura di un VFX realizzato con Niagara, poi verranno presentate l'interfaccia del Niagara Editor, la struttura di un Emitter e alcuni dei moduli più usati. Infine si mostreranno alcune pratiche per la creazione di Materiali usati in effetti particellari.

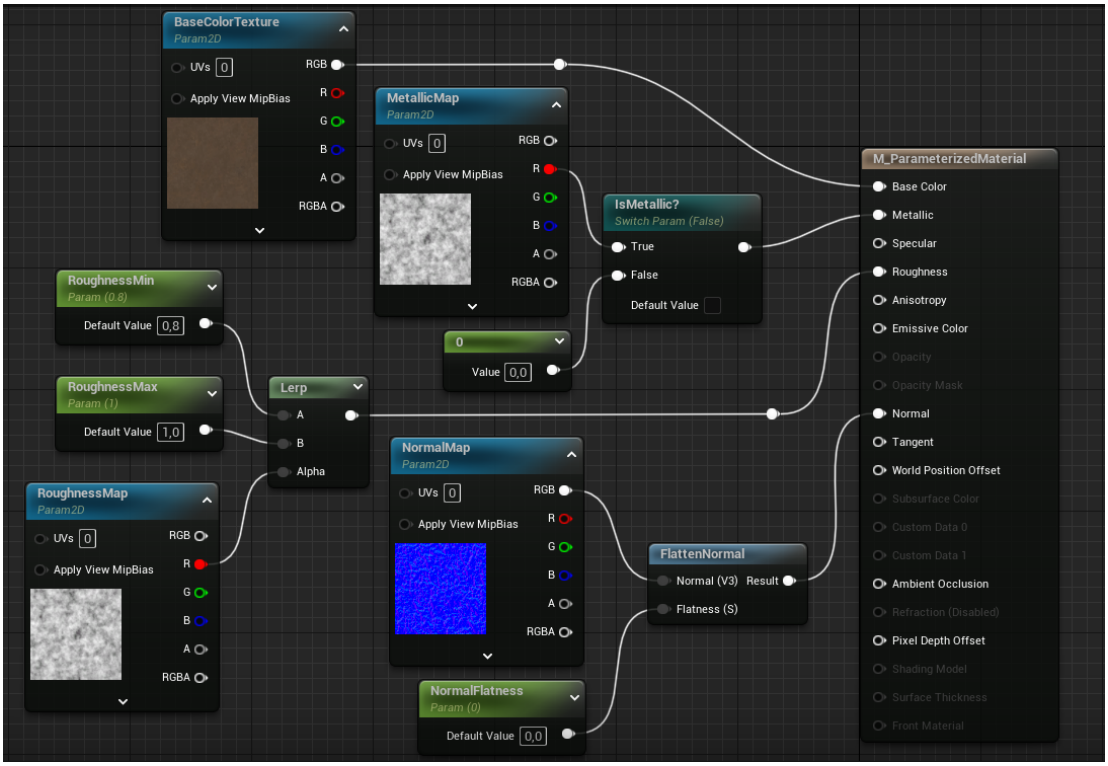


Figura 2.6: Esempio di uso dei Parametri per la realizzazione di un Master Material.

Il Niagara VFX System è lo strumento principale di creazione di VFX in Unreal Engine. È stato introdotto nel 2018 in versione beta tramite la release 4.20 di Unreal Engine ed è pensato per sostituire gradualmente il vecchio sistema di creazione di VFX, Cascade.

Niagara segue il paradigma di programmazione *stack-oriented*, ovvero l'utente definisce una serie di funzionalità, posti uno sopra l'altro in una sorta di pila (*stack*), alla quale passa dei parametri che determineranno il comportamento finale del VFX. In Niagara, i pezzi che costruiscono questa pila sono detti **Moduli** e, in modo analogo ai nodi del Material Editor, contengono codice HLSL. Niagara è organizzato secondo una struttura gerarchica costituita da quattro elementi, ognuno dei quali è il contenitore del successivo:

- **System**: è il contenitore di uno o più Emitter e definisce alcuni comportamenti globali del VFX. È l'unico elemento della gerarchia a poter essere posizionato

in un livello.

- **Emitter**: si occupa di generare le particelle e di controllarne i comportamenti. Rappresenta lo *stack* che viene eseguito.
- **Module**: descrive un singolo comportamento delle particelle. Contiene codice HLSL e rappresenta il singolo elemento dello *stack*.
- **Parameter**: dato che viene passato al modulo per definirne quantitativamente il comportamento.

Inoltre, Niagara supporta la definizione Moduli personalizzati (**Custom Modules**), che permettono di descrivere comportamenti più o meno complessi non previsti dai Moduli presenti di default in Unreal Engine. Lo scripting di tali Moduli segue un paradigma basato sui nodi, simile a quello adottato in Blueprint.

2.4.1 Interfaccia del Niagara Editor

In questo sottoparagrafo verranno illustrati gli elementi principali dell'interfaccia del Niagara Editor di Unreal Engine 5. Si farà riferimento alla documentazione ufficiale online[4].

La Figura 2.7 illustra i principali elementi dell'interfaccia del Niagara Editor di Unreal Engine 5. Segue una breve analisi delle sue componenti, numerati come nella Figura 2.7:

1. **Barra dei Menu**: contiene i menu dell'Editor.
2. **Toolbar**: contiene *shortcut* per gli strumenti più utilizzati del Niagara Editor. Consente di salvare e compilare i Moduli del Sistema, generare una *thumbnail* del Sistema, visualizzare statistiche di performance, mostrare la *bounding box* del Sistema, accedere a strumenti di debug e cambiare alcune impostazioni della simulazione.
3. **Preview**: visualizzazione di un'anteprima del Sistema. Permette inoltre di cambiare le opzioni di visualizzazione e mostrare statistiche di performance.

[4] `https://docs.unrealengine.com/5.3/en-US/editor-ui-reference-for-niagara-effects-in-unreal-engine/`

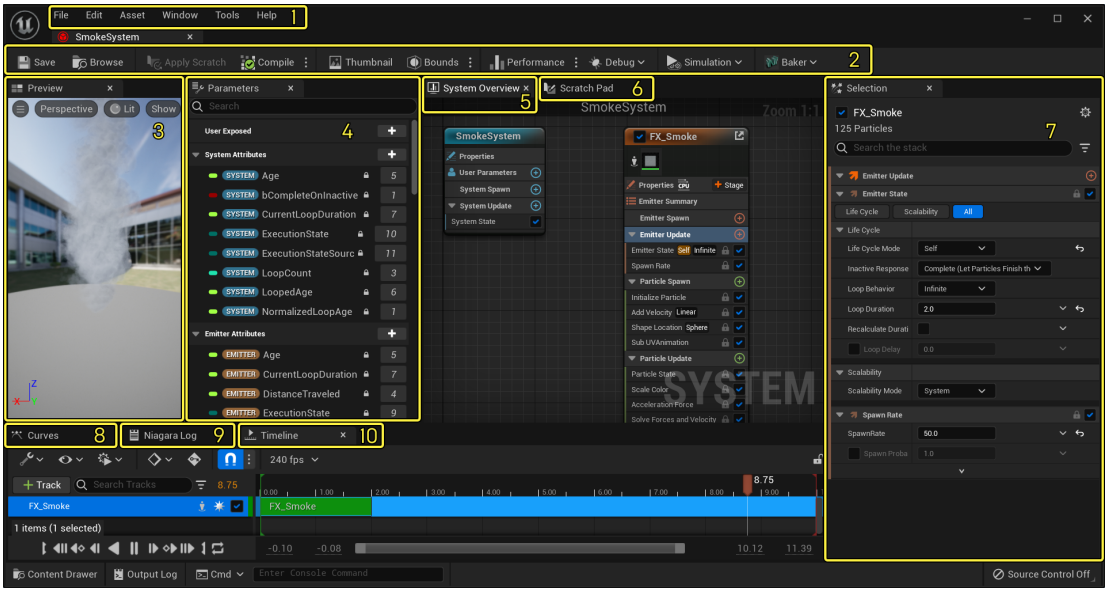


Figura 2.7: Interfaccia del Niagara Editor di Unreal Engine 5

4. **Parameters**: mostra i parametri usati dall'Emitter Selezionato e il numero di riferimenti associati a ognuno di essi. Inoltre, permette di creare riferimenti e di definire nuovi parametri.

5. **System Overview**: finestra di visualizzazione ad alto livello e modifica degli Emitter in forma di *stack*.

6. **Scratch Pad**: finestra di scripting di moduli personalizzati o di input dinamici.

7. **Selection**: mostra i dettagli dell'Emitter o del Modulo selezionato. Selezionando un Emitter, permette di aggiungervi nuovi moduli. Selezionando un Modulo, permette di modificarne i parametri o di aggiungervi input dinamici.

8. **Curves**: permette di definire curve che descrivono l'evoluzione del valore di un parametro in funzione di una certa variabile. Un esempio è la modifica della luminosità di una particella al variare della sua età.

9. **Niagara Log**: mostra eventuali errori e avvisi relativi alla compilazione dello script, sia a livello Emitter che a livello di Sistema.

10. **Timeline**: permette di definire il comportamento dei singoli Emitter attivi nel sistema al variare del tempo.

2.4.2 Struttura di un Niagara Emitter

In questo sottoparagrafo verrà illustrata la struttura e il funzionamento a fasi di un Niagara Emitter.

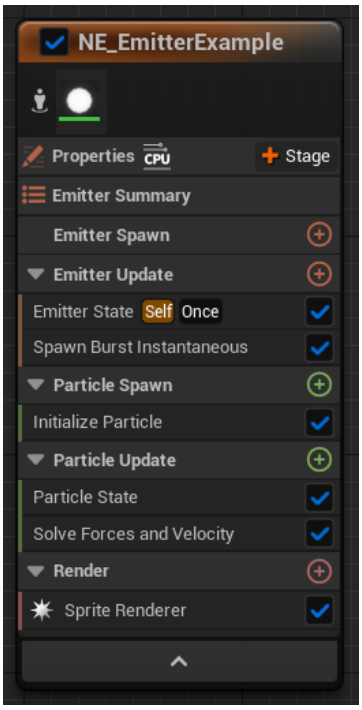


Figura 2.8: Struttura di un Niagara Emitter secondo la rappresentazione presente nel System Overview del Niagara Editor

Nel System Overview del Niagara Editor, un Emitter è rappresentato come nella figura 2.8. Un Emitter contiene al suo interno più fasi (o *stage*), che vengono eseguite dall'alto verso il basso. Le fasi che contengono "Spawn" nel nome vengono eseguite solo una volta, alla creazione dell'Emitter o della particella; le fasi che contengono "Update" nel nome vengono eseguite ogni frame. Le cinque fasi presenti in tutti gli Emitter sono:

- **Emitter Spawn**: permette di modificare i parametri con cui viene creato l'Emitter. Raramente vengono inseriti Moduli al suo interno.

- **Emitter Update**: permette di impostare il comportamento dell'Emitter nel tempo. I Moduli più comunemente posti al suo interno sono **Emitter State**,

tramite il quale si definiscono eventuali comportamenti a *loop* dell'Emitter, e i Moduli di **Spawn**, che sono i responsabili della creazione delle particelle.

- **Particle Spawn**: permette di impostare le proprietà iniziali delle particelle. È sempre presente un modulo di inizializzazione, solitamente **Initialize Particles**, che permette di impostare tempo di vita, posizione, massa e colore delle particelle, oltre ad altri attributi dipendenti dal tipo di renderer usato. Altri moduli comunemente usati in questa fase sono **Shape Location**, che permette di creare le particelle in punti casuali appartenenti al volume di una forma geometrica di base (ad esempio sfera, cubo, toro) e **Add Velocity**, che aggiunge una velocità iniziale alle particelle.

- **Particle Update**: permette di modificare le proprietà delle particelle durante la loro vita. È sempre presente un modulo **Particle State**, che ha il compito di eliminare le particelle che hanno terminato la loro vita. Nella maggior parte dei Sistemi è anche presente il modulo **Solve Forces and Velocity**, che raccoglie le informazioni provenienti dai precedenti moduli di forza e velocità convertendole in proprietà della particella, facendo in modo che tali attributi siano indipendenti dal frame rate. Inoltre, sono comunemente usati i moduli di forza (**Force**), che applicano forze di vario tipo, il modulo **SubUVAnimation**, che permette di creare animazioni con le *flipbook texture*, il modulo **Collision** che permette di simulare la fisica delle collisioni, e altri moduli che permettono di aggiornare attributi della particella come colore, orientamento e scala.

- **Render**: permette di scegliere il modo in cui visualizzare le particelle di cui si sono calcolate le proprietà. I tipi di renderer più comuni sono lo **Sprite** e il **Mesh Renderer**.

Oltre a quelle elencate, è inoltre possibile aggiungere un'ulteriore fase, chiamata **Event Handler**, che ha il compito di raccogliere informazioni da Moduli evento presenti in altri Emitter e definire comportamenti delle particelle in base a tali eventi.

Nella parte superiore dell'Emitter è infine presente il pannello **Properties**, che permette di definire alcune proprietà tecniche dell'Emitter. Alcune delle proprietà principali sono **Sim Target**, che permette di scegliere se i calcoli verranno effettuati

su CPU o GPU, e **Requires Persistent ID**, che associa un identificativo univoco a ogni particella generata dall'Emitter.

2.4.3 Simulazioni CPU e GPU

Nella creazione di un Emitter all'interno di un Niagara System, la scelta della destinazione di calcolo fra CPU e GPU (Sim Target) ricopre un ruolo fondamentale, poiché ognuna delle due opzioni presenta dei vantaggi e dei limiti. Di seguito sono elencate alcune delle caratteristiche più importanti associate a ognuna delle due opzioni:

- **CPUSim**: nel caso di calcoli eseguiti da CPU sono disponibili i moduli di invio e gestione degli eventi. È inoltre possibile impostare dei *bound* dinamici alla simulazione, che cioè vengono calcolati a ogni frame. Lo svantaggio di questo tipo di calcolo è che il numero massimo di particelle simulabili in tempo reale è abbastanza basso.
- **GPUComputeSim**: i calcoli eseguiti da GPU riescono a supportare un numero di particelle molto più elevato delle simulazioni CPU. Sono inoltre presenti alcuni moduli che sono disponibili solo per Emitter con calcolo GPU, ad esempio **Sample Texture** o i moduli che eseguono calcoli basati sui **Distance Fields** e sul **Depth Buffer**. Tuttavia, non sono attualmente supportati i moduli relativi agli eventi e i *bound* degli Emitter devono essere statici.

2.4.4 Input Dinamici

Il risultato finale del sistema particellare dipende pesantemente dal valore assegnato alle variabili all'interno di ogni modulo. In Niagara, è possibile passare alle variabili non solo dei valori statici, ma anche delle chiamate a funzioni che permettono di effettuare delle operazioni dinamiche. Questi sono detti Input Dinamici (**Dynamic Input**). Alcune delle funzionalità degli input dinamici sono l'introduzione di valori casuali (ad esempio **Random Range Float**), la scomposizione di vettori nelle singole componenti (ad esempio **Make Vector**), la definizione di relazioni di una variabile in base a un'altra quantità (ad esempio **Float From Curve** o **Color**

From Curve) o operazioni e funzioni matematiche (ad esempio **Multiply Vector**, **Sine** o **Lerp Float**).

2.4.5 Tipi di Renderer in Niagara

La selezione del renderer in un Niagara System determina il tipo di particella renderizzata. Si tratta di una delle scelte più importanti nella creazione di un sistema, poiché ne definisce il look generale e i costi computazionali. Inoltre, alcuni Moduli e variabili sono specifici per alcuni renderer, ad esempio **Scale Sprite Size**, **Initial Mesh Orientation** o **Scale Ribbon Width**. Di seguito i più importanti renderer in Niagara:

- **Sprite Renderer**: le particelle vengono renderizzate come *sprite*, ovvero dei piani bidimensionali a cui è applicato un Materiale. Di default, questi piani sono orientati in direzione della camera. Solitamente, sulle sprite viene applicata una maschera, spesso circolare. È un renderer a basso costo computazionale, per cui è possibile renderizzare numeri di particelle molto elevati.
- **Mesh Renderer**: le particelle vengono renderizzate come mesh, sulle cui trasformate 3D è possibile agire tramite i Moduli. È possibile ottenere una varietà di effetti molto ampia, ma il costo in performance potrebbe essere molto alto dipendentemente dal numero di particelle e dalla complessità della mesh.
- **Ribbon Renderer**: le particelle vengono renderizzate come un nastro continuo. È utile, ad esempio, per creare effetti di scia dietro altre particelle o oggetti in movimento.
- **Component Renderer**: permette di renderizzare le particelle come qualsiasi tipo di componente, ad esempio tutti i tipi luci, asset audio o Skeletal Mesh. Al momento, è ancora una funzionalità sperimentale ed è controindicato usarlo per progetti real-time, a causa di una mancata ottimizzazione per le performance.

È possibile combinare più Moduli Renderer all'interno dello stesso Emitter. Tuttavia, è da tenere in considerazione che, salvo per proprietà applicabili a un solo tipo di particella, le proprietà calcolate dai precedenti moduli saranno comuni a tutte le particelle, indipendentemente dal tipo di Renderer.

2.4.6 Tipi di Spawner in Niagara

La scelta del tipo di *spawner* determina il modo in cui le particelle verranno create e il tipo di evoluzione che il Sistema avrà nel tempo. Di seguito sono descritti i quattro tipi di moduli di Spawn:

- **Spawn Rate**: viene emesso un certo numero di particelle al secondo. È utile nel caso in cui il sistema debba andare in loop, poiché non c'è soluzione di continuità fra le diverse ripetizioni. Inoltre, è molto importante bilanciare i parametri di Spawn Rate e durata della vita delle particelle (**Lifetime**), poiché con vite troppo lunghe, il numero totale delle particelle in vita può raggiungere numeri molto alti.

- **Spawn Burst Instantaneous**: viene emesso un certo numero di particelle una sola volta per tutta la durata del loop. È usato, ad esempio, in effetti come esplosioni o altri effetti che hanno una forte emissione istantanea e poi un decadimento.

- **Spawn Per Unit**: viene emesso un certo numero di particelle per ogni unità di gioco (che in Unreal Engine corrisponde a un centimetro) per la quale il sistema viene spostato all'interno del livello. È possibile impostare una soglia di velocità oltre la quale il sistema smette di emettere per impedire di avere un numero di particelle troppo alto. È anche presente una soglia minima per impedire al sistema di emettere particelle nel caso di movimenti molto piccoli. Questo tipo di Spawner può essere utile, ad esempio, nel caso di effetti magici.

- **Spawn Per Frame**: viene emesso un certo numero di particelle ad ogni frame. Ha un comportamento molto simile al Modulo Spawn Rate, e può essere usato nel caso si voglia avere un'emissione di particelle molto abbondante dall'inizio del ciclo di vita dell'Emettitore o nel caso si stia usando un Ribbon Renderer che abbia elevata velocità.

I Moduli di *spawning* possono essere usati anche in maniera combinata: ad esempio, si possono combinare Spawn Burst Instantaneous e Spawn Rate per un VFX che abbia una forte emissione di particelle all'inizio del ciclo e poi si stabilizzi su un'emissione continua.

2.4.7 Moduli di Velocità e Forza in Niagara

I moduli di velocità e forza hanno lo scopo di descrivere il moto delle particelle durante il loro ciclo di vita. Essi permettono quindi di definire la dinamica e la "forma" del VFX. I moduli di velocità (**Velocity**) sono solitamente usati per aggiungere una velocità iniziale alla particella, perciò vengono usati principalmente nella fase Particle Spawn. I moduli di forza (**Force**) sono usati per applicare una forza alla particella, che può essere sia applicata alla sua creazione (nel caso il Modulo venga aggiunto nella fase di Particle Spawn), sia applicata in maniera continua durante il suo ciclo di vita (nel caso il modulo venga aggiunto nella fase di Particle Update).

I moduli di velocità e forza, per funzionare, necessitano la presenza del modulo **Solve Forces and Velocity**, che deve essere posto al di sotto di essi e ha il compito di leggere il parametro delle forze totali calcolate e sovrascrivere i parametri di posizione e tempo delle particelle. Inoltre, se si aggiungono moduli di forza nella fase di Particle Spawn, è necessario aggiungere il modulo di **Apply Initial Forces** nella stessa fase.

Un altro fattore da tenere in considerazione quando si crea un Emitter contenente moduli di velocità e forza è il loro costo in termini di performance: i Moduli applicati nella fase di Particle Spawn vengono eseguiti solo una volta per particella, mentre i Moduli applicati nella fase di Particle Update vengono eseguiti ad ogni frame, per cui hanno costi computazionali molto maggiori. Per questo motivo, occorre limitare il numero di Moduli di forza nella fase Particle Update se si vogliono risparmiare risorse di calcolo.

Di seguito sono elencati alcuni dei Moduli di velocità e forza più importanti:

- **Add Velocity**: aggiunge una velocità iniziale alle particelle. Ha tre modalità di funzionamento: **Linear**, in cui la velocità aggiunta è rappresentata da un vettore indicato dall'utente; **From Point**, in cui la velocità viene aggiunta radialmente a partire da un determinato punto dello spazio; **In Cone**, in cui la velocità viene aggiunta seguendo la forma di un cono.

- **Curl Noise Force**: aggiunge una forza alle particelle seguendo un pattern di *curl noise*, che simula le turbolenze presenti in fluidi incompressibili a partire

da rumore di Perlin[5]. Viene usato per aggiungere una componente casuale e variazione al moto delle particelle.

- **Drag/Aerodynamic Drag**: aggiunge una forza che simula la resistenza fluidodinamica, che si oppone alla risultante di tutte le altre forze. Viene usato per limitare la velocità delle particelle. Aerodynamic Drag rappresenta una versione più avanzata che modifica anche l'orientamento delle particelle e risulta utile, ad esempio, per simulare l'effetto di oggetti sottili in aria, come fogli di carta o foglie.

- **Wind Force**: aggiunge una forza alle particelle in una certa direzione, più un contributo opzionale di turbolenza. Inoltre, quando le particelle raggiungono la velocità del vento, esse non vengono più sottoposte a tale forza. Richiede la presenza di un modulo di Drag o Aerodynamic Drag.

- **Gravity Force**: aggiunge una forza gravitazionale alle particelle lungo una direzione, e misurata in cm/s^2.

- **Vortex Velocity/Force**: aggiunge alle particelle una velocità o forza angolare intorno a un asse.

- **Point Attraction Force**: applica alle particelle una forza attrattiva verso un punto.

- **Point Force**: applica alle particelle una forza repulsiva da un punto.

La combinazione di queste forze, la modifica dei parametri e l'uso di input dinamici permettono di ottenere effetti molto diversi tra loro.

2.4.8 Materiali e Particelle

Nella creazione di un VFX, il Materiale applicato alle particelle ricopre un ruolo molto importante per definirne l'aspetto. In questo sottoparagrafo verranno presentate alcune pratiche da adottare nella creazione di un Materiale da applicare a un sistema particellare. Verrà approfondito il caso di uso di Sprite Renderer, ma molti degli strumenti che saranno presentati possono essere applicati anche ad altri tipi di

Renderer. In Niagara, la selezione del Materiale da applicare alle particelle avviene nel Renderer e qualunque Materiale presente nel progetto può esservi applicato.

Uno degli aspetti principali da considerare è la scelta dello Shading Model e della Blend Mode. Riguardo lo Shading Model, solitamente per Sprite e Ribbon si usa il modello **Unlit**, poiché le superfici su cui esso è applicato risultano puramente emissive, non risentono dell'illuminazione della scena e non proiettano ombre. Il sistema di illuminazione Lumen, inoltre, permette ai Materiali emissivi di illuminare anche gli altri oggetti nella scena, incrementandone il realismo. Per quanto riguarda la Blend Mode, le tre opzioni più comuni sono:

- **Additive**: permette di impostare maschere di opacità, ma i colori scuri sono mostrati come trasparenti, quindi non possono essere applicati alle particelle. Viene usato per effetti puramente emissivi o con colori chiari, come fuoco, scintille o schizzi d'acqua. Ha una complessità computazionale abbastanza alta, ma minore di Translucent.

- **Translucent**: permette di impostare maschere di opacità e prevede l'utilizzo di colori scuri. Viene usato per effetti in cui servono anche colori scuri, ad esempio fumo, gocce di sangue o qualsiasi particella rappresenti un essere vivente, come uccelli o farfalle. Permette inoltre di impostare un indice di rifrazione, quindi è usato anche per simulare distorsioni dovute al calore o effetti contenenti liquidi, ad esempio la pioggia o una cascata. Tuttavia, questa Blend Mode ha una complessità computazionale molto elevata, quindi va evitata se non necessaria.

- **Masked**: permette di impostare maschere di opacità, che verranno interpretate in maniera binaria, ovvero completamente opaco/completamente trasparente. Questo impedisce l'uso di effetti di *fade in/fade out*, per cui le particelle da invisibili diventano immediatamente visibili, senza transizioni. Questo problema può essere parzialmente risolto tramite l'uso del nodo **DitherTemporalAA**, ovvero un pattern di rumore binario che ha una media pari al suo input Alpha Threshold. Tale pattern riesce a simulare un effetto di dissolvenza semplicemente animando il suo input di soglia. Ha un costo computazionale basso.

Per quanto riguarda il Material Graph, sono disponibili dei nodi specifici per l'uso accoppiato con Niagara. Essi raccolgono informazioni sulle particelle e le restituiscono in output. Tutti questi nodi hanno il prefisso "**Particle**". Essi possono essere usati per definire la logica dello shader in dipendenza dalle proprietà delle particelle. È importante notare che i dati trasmessi da questi nodi sono relativi a ogni singola particella, quindi con la giusta logica è possibile dare proprietà estetiche molto diverse a particelle facenti parte dello stesso Emitter, ma che hanno proprietà diverse. La documentazione online di Unreal Engine presenta una lista completa di questi nodi[5]. Di seguito sono elencati alcuni di essi:

- **ParticleColor**: dà in output il colore della particella indicato nel Niagara Editor. È necessario usare questo nodo per combinare le informazioni di colore provenienti dall'Emitter con quelle dello shader. Ad esempio, bisogna combinare i canali alfa di Emitter e shader per fare sì che gli effetti di *fade* impostati in Niagara abbiano effetto.

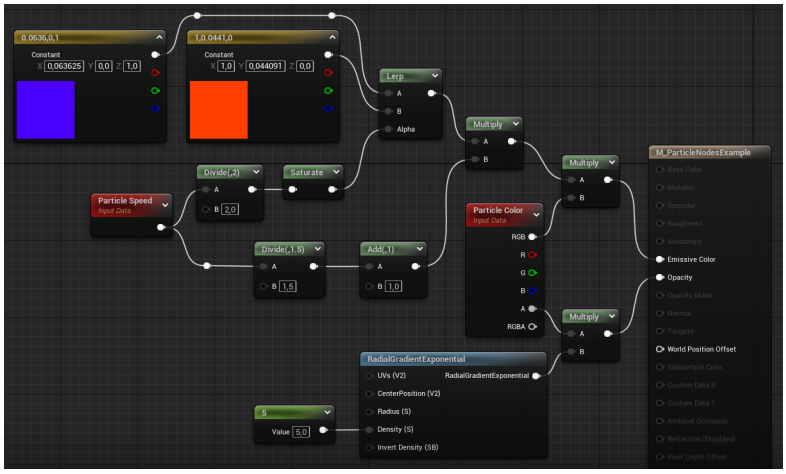
- **ParticleSpeed**: dà in output la velocità della particella in cm/s.

- **ParticleRadius**: dà in output il raggio della particella in cm.

- **ParticleRelativeTime**: dà in output l'età attuale della particella normalizzata tra 0 e 1.

La figura 2.9a presenta un esempio di utilizzo di tali nodi: i dati provenienti dal nodo ParticleSpeed fanno sì che le particelle più veloci abbiano un colore più vicino al rosso e una potenza di emissione maggiore, mentre le particelle più lente avranno un colore più vicino al blu e saranno più debolmente emissive. La figura 2.9b presenta la preview dell'applicazione di tale Materiale a un Emitter con un Modulo di Add Velocity in modalità From Point per spingere le particelle in modo radiale a partire dal punto di spawn, un Modulo di Drag per frenarle, e un modulo di Scale Color per dar loro un effetto di *fade out*, possibile grazie al nodo Particle Color presente nel Materiale.

[5] `https://docs.unrealengine.com/5.3/en-US/particle-material-expressions-in-unreal-engine/`



(a) Material Graph

(b) Preview del Niagara System

Figura 2.9: Esempio di utilizzo dei nodi Particle nel Material Graph

Infine, tramite Niagara è possibile scrivere dati di qualsiasi tipo sul Materiale tramite i parametri dinamici (**Dynamic Parameter**). Ogni set di Dynamic Parameter contiene quattro variabili float, e ogni Materiale può avere fino a quattro set di Dynamic Parameter, ognuno con un proprio **Parameter Index**. Quindi, un Materiale può ricevere fino a 16 valori scalari da un Niagara System. Perché un Materiale possa leggere dei parametri dinamici da Niagara, bisogna aggiungere un nodo **DynamicParameter** nel Material Graph. Ognuno dei suoi *pin* rappresenterà un parametro scalare. In Niagara, per scrivere il valore dei parametri dinamici, occorre aggiungere il Modulo **Dynamic Material Parameters** nella fase di Particle Update. Come per tutti i parametri di Niagara, è possibile usare valori statici, input dinamici o anche passare un qualsiasi parametro del Sistema. Dal punto di vista artistico, l'uso di parametri dinamici permette un livello di personalizzazione molto maggiore. Inoltre, nella creazione di Sistemi più complessi, spesso occorre assegnare lo stesso Materiale o Material Instance a diversi Emitter, ma ognuno con dei parametri diversi. Grazie all'uso di parametri dinamici è possibile avere un set di parametri unico per ogni Emitter, senza creare duplicati, che altrimenti avrebbero un costo sia in termini di memoria che di performance.

2.4.9 Parametri e Namespace

In Niagara, tutte le funzionalità di un sistema sono controllate dal valore dei parametri al suo interno. Ogni modulo prevede dei **Parameter Reads** e dei **Parameter Writes**, ovvero, rispettivamente, la lettura e scrittura di valori in dei parametri, visualizzabili dai dettagli del modulo. Nel caso vengano usati moduli avanzati, leggere quali siano i Parameter Reads e Writes può aiutare la comprensione del funzionamento e i modi con cui questi possano interagire con altri moduli. I parametri sono organizzati in **Namespace**, ovvero delle collezioni che servono a disambiguare parametri con nomi simili o uguali e a definire quali porzioni del Niagara System possono accedervi. Inoltre, ogni membro della gerarchia del Niagara System può scrivere i propri parametri solo nel proprio Namespace di riferimento. I Namespace presenti in Niagara sono **System**, **Emitter**, **Particle**, **Transient**, **StackContext**, **Engine**, **Input**, **Output** e **User**. Ad esempio, un Emitter ha accesso in lettura ai parametri presenti nei Namespace System, Emitter, Engine e User, ma scriverà solo nel Namespace Emitter. I parametri definiti dall'utente vengono inseriti nel Namespace User. È anche possibile usare i parametri come input nelle proprietà dei moduli che ne hanno accesso.

2.5 Blueprint Visual Scripting

Il **Blueprint Visual Scripting** è un sistema di scripting basato sui nodi e orientato agli oggetti. Ognuno dei nodi contiene del codice C++, che è anche il linguaggio usato per lo scripting più avanzato. Lo scripting in Blueprint è usato sia per definire la logica del gioco o, in generale, dell'eseguibile, sia per definire dei parametri per attori o gruppi di attori che possono essere modificati nell'Editor, in fase di *world building*, risultando quindi utile sia ai programmatori, sia agli artisti. In generale, Blueprint offre un controllo molto ampio su tutti i tipi di asset.

In Blueprint, si creano delle classi (che in Unreal Engine si chiamano (**UClass**) per definire un comportamento. Nella creazione di una classe Blueprint, occorre per prima cosa scegliere la classe da cui erediterà. In Unreal Engine, sono presenti delle classi *built-in* che è possibile scegliere come classi genitore (**Parent Class**). La classe Blueprint definita dall'utente eredita le funzionalità della classe genitore, per cui lo scripting che si effettuerà sarà molto legato a questa prima scelta. Nel

sistema gerarchico delle classi definite in Unreal Engine, la classe da cui tutte le altre ereditano è **Object**, che rappresenta un oggetto generico di Unreal Engine. La classe che si utilizza maggiormente è **Actor**, che rappresenta un oggetto che può essere posizionato nel mondo, ed è una classe direttamente figlia di Object.

Una volta scelta la classe genitore, nella classe Blueprint si possono definire uno o più dei seguenti elementi:

- **Components**: uno o più oggetti, organizzati in maniera gerarchica, che compongono la classe Blueprint e che sono necessari al suo funzionamento. Alcuni esempi di Component sono Static Mesh, Camera, tutti i tipi di luci, audio, diversi tipi di movimento, forze fisiche o volumi di collisione. Dal punto di vista dell'interfaccia, l'Editor dei componenti è simile all'Editor principale di Unreal Engine.

- **Construction Script**: contiene la logica da eseguire prima dell'avvio. Solitamente contiene dei parametri che permettono di modificare l'aspetto dei componenti contenuti nella classe Blueprint direttamente dall'Editor, per cui risulta utile maggiormente ai *level artist*.

- **Event Graph**: contiene la logica che viene eseguita runtime. È il luogo in cui si esegue solitamente la maggior parte dello scripting.

In una classe Blueprint è inoltre possibile definire elementi comuni della programmazione ad oggetti, ovvero **funzioni** e **interfacce**. Con Blueprint è inoltre possibile definire la logica delle animazioni e della UI, ma questi argomenti sono al di fuori dell'interesse di questa tesi.

2.5.1 Interfaccia del Blueprint Editor

In questo sottoparagrafo verranno illustrati gli elementi principali dell'interfaccia del Blueprint Editor di Unreal Engine 5, specificamente l'editor di una Blueprint Class. Si farà riferimento alla documentazione ufficiale online[6].

[6] `https://docs.unrealengine.com/5.3/en-US/blueprints-visual-scripting-user-interface-for-blueprint-classes-in-unreal-engine/`

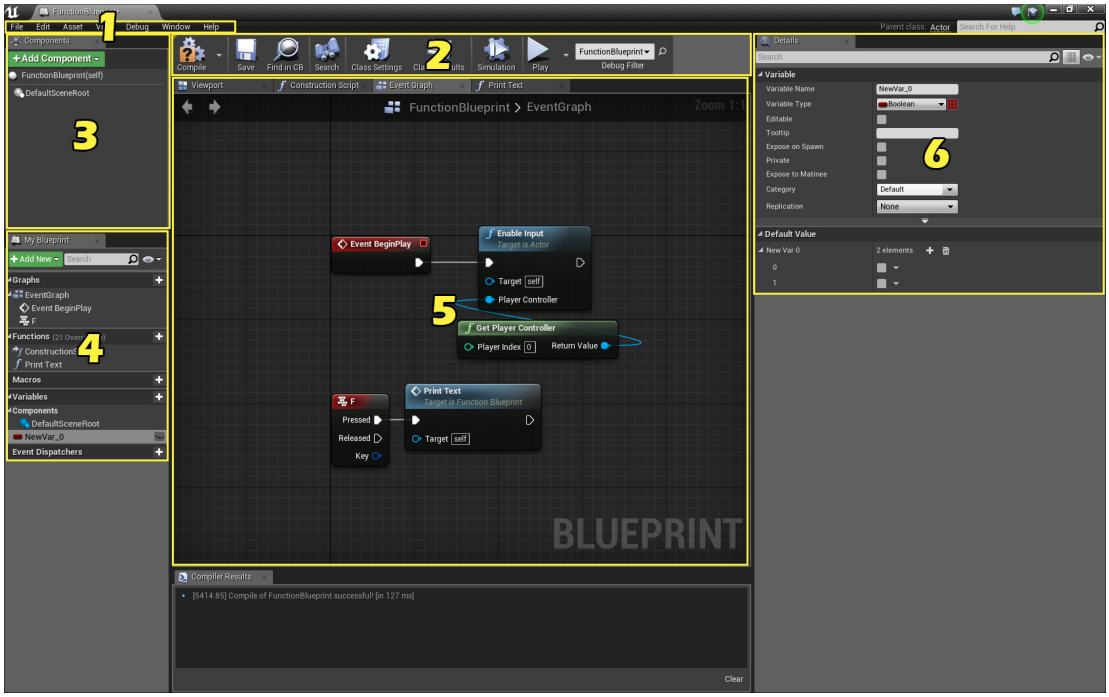


Figura 2.10: Interfaccia del Blueprint Class Editor di Unreal Engine

La Figura 2.10 illustra i principali elementi dell'interfaccia del Niagara Editor di Unreal Engine. La figura mostra l'interfaccia di Unreal Engine 4, a causa di un mancato aggiornamento della documentazione, ma gli elementi presenti sono rimasti sostanzialmente invariati. Segue una breve analisi delle componenti della UI, numerati come nella Figura 2.10:

1. **Menu**: contiene i menu dell'Editor, che danno accesso a operazioni di salvataggio, modifica, visualizzazione, version control e debug.

2. **Toolbar**: dà accesso diretto alle funzionalità più comunemente usate, ovvero il salvataggio, la compilazione, l'operazione di *diff* nel version control, l'accesso ai default della classe o alle impostazioni della classe, avviare la simulazione, strumenti di esecuzione della *play mode* e opzioni di debug.

3. **Componenti**: presenta la lista dei componenti e la loro organizzazione gerarchica, permette di imparentarli e di aggiungerne di nuovi.

4. **My Blueprint**: presenta una vista ad albero dei grafici presenti nella classe Blueprint e gli eventi contenuti al loro interno e la lista di funzioni, macro, variabili ed *Event Dispatchers*. Permette inoltre di creare nuovi elementi di questi tipi e di creare dei riferimenti ad essi da inserire nel grafico.

5. **Graph Editor**: permette di visualizzare e modificare la struttura a nodi del grafico.

6. **Details**: permette di visualizzare e modificare le proprietà del nodo selezionato.

2.5.2 Struttura di un nodo

Un nodo, nel Material Editor è rappresentato come diviso in due parti: la parte sinistra contiene i *pin* di input, la parte di destra i *pin* di output. Alcuni nodi, ad esempio i nodi evento, non contengono *pin* di input, ma solo di output.

In ognuna delle due parti è presente almeno uno fra i due tipi di *pin*:

- **Pin di esecuzione**: definisce il flusso dell'esecuzione, quindi l'ordine sequenziale in cui eseguire i nodi del grafo.

- **Pin di dati**: trasferiscono dati di un determinato tipo in input o output. È possibile convertire tali dati in variabili.

Nel caso si colleghino due *pin* di dato con tipi di dato diversi fra loro, dove possibile, verrà effettuata una operazione di conversione di tipo o *auto-casting*. Un esempio di conversione possibile è da int a float o da float a stringa.

2.5.3 Tipi di dato in Blueprint

In Blueprint, oltre ai tipi comuni alla maggior parte dei linguaggi di programmazione (bool, int, float e string), sono presenti dei tipi di dato specifici per le funzionalità di Unreal Engine. Di seguito sono elencati alcuni di questi tipi di dato:

- **Vector**: set di tre valori float. Usato per descrivere trasformazioni di posizione o scala o per rappresentare valori RGB.

- **Rotator**: set di tre valori float che descrive una rotazione 3D.

- **Transform**: set di tre dati vettoriali che descrivono la posizione, la rotazione e la scala di un oggetto nello spazio 3D.

- **Name**: testo che identifica un oggetto. È più efficiente delle stringhe quando si effettuano operazioni di confronto.

- **Text**: testo esposto all'utente. È il tipo di dato che viene maggiormente sottoposto a localizzazione.

- **Oggetto**: riferimento a un qualsiasi tipo di oggetto nel progetto.

- **Enum**: variabili che possono assumere solo i valori definiti dall'utente.

2.5.4 Nodi Evento

Poiché Blueprint è progettato principalmente per lo scripting in prodotti interattivi, l'uso di eventi risulta centrale. In Blueprint, gli eventi sono dei nodi che vengono chiamati quando avvengono determinate azioni, che possono essere prodotti o no dall'utente. Quando gli eventi vengono chiamati, avviano un flusso di esecuzione.

Gli eventi possono sia essere legati allo stato dell'eseguibile, sia ad azioni relative ai componenti di un Blueprint. Alcuni esempi della prima categoria sono il nodo **Event Begin Play**, che viene chiamato quando l'eseguibile viene avviato o quando un *Actor* viene posizionato nel mondo, e il nodo **Event Tick**, che viene chiamato a ogni frame o ogni gruppo di frame. Alcuni eventi relativi ai componenti sono **On Component Begin Overlap**, che viene chiamato quando un oggetto si sovrappone al componente, che solitamente è un volume di collisione, o **On Component Hit**, che viene chiamato quando il componente colpisce o viene colpito da un oggetto solido.

Infine, è possibile definire dei **Custom Event**, ovvero degli eventi che vengono definiti dall'utente e che possono essere chiamati in qualsiasi punto dello stesso Event Graph o in altre classi Blueprint. Quando viene definito l'evento, si possono aggiungere degli input, che saranno passati alle chiamate a tale evento, così che si possano trasmettere dati tra diverse Blueprint.

2.5.5 Il nodo Timeline

Il nodo **Timeline** permette di definire delle curve che descrivono l'animazione nel tempo di un valore float, vettoriale, di colore o relativo a un evento. Il nodo Timeline è pensato per essere utilizzato in animazioni semplici, come apertura di porte o modifica dell'intensità di una luce. Un singolo nodo Timeline può contenere un numero qualsiasi di tracce (**Track**) che descrivono in modo indipendente diversi tipi di dato, ma tutte esse condivideranno le informazioni temporali. Attraverso i *pin* corrispondenti alle tracce è poi possibile dare in output il valore animato o avviare un nuovo flusso d'esecuzione in caso di traccia evento. Sono presenti diversi *pin* di esecuzione, tramite i quali si può riprodurre la traccia sia in avanti che al contrario, fermarla, sovrascrivere il tempo e avviare un nuovo ramo di esecuzione quando il tempo raggiunge la fine.

La figura 2.11 mostra un esempio di nodo Timeline, nel quale sono state definite quattro tracce, una per ogni tipo.

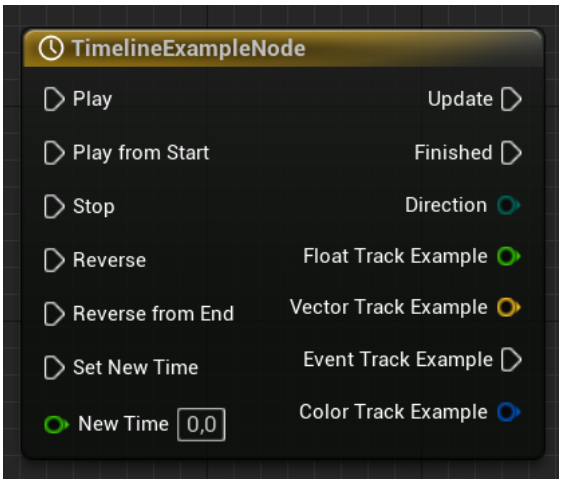


Figura 2.11: Esempio di nodo Timeline

2.6 MetaSounds

MetaSounds è il sistema audio di Unreal Engine, introdotto nella versione 5, che rende possibile una gestione completa delle sorgenti audio tramite un grafo di **Digital Signal Processing** (**DSP**). Attualmente, il sistema si trova ancora

in fase Beta, quindi non possiede ancora tutte le feature previste e potrebbe dare problemi in fase di *shipping*, ovvero la creazione della build a partire dal progetto. MetaSounds è pensato come l'equivalente audio del Material Graph, e ha un approccio totalmente procedurale. Esso offre la possibilità agli Audio Programmer di eseguire un'implementazione audio esclusivamente all'interno di Unreal Engine, senza il supporto di *middleware* come Wwise o FMOD, con benefici sul workflow e sui costi. Poiché MetaSounds è un sistema destinato *in primis* all'uso nei videogiochi, esso presenta una compatibilità molto alta con Blueprint, tramite il quale è possibile far interagire lo scripting a eventi con il rendering audio procedurale. Esso presenta inoltre delle prestazioni che fanno sì che il consumo di risorse computazionali sulla CPU sia ridotto, poiché tutto deve poter essere eseguito in tempo reale. Tramite MetaSounds è possibile sia riprodurre degli asset audio presenti nel progetto, sia generare segnali audio a partire da oscillatori o *wavetable*, con un approccio basato sulla sintesi.

Ci sono due tipi di asset relativi a MetaSounds:

- **MetaSound Source**: asset all'interno del quale viene definito il grafo DSP. Può essere posizionato e riprodotto direttamente nel mondo.
- **MetaSound Patch**: permette di definire dei nodi personalizzati da riutilizzare nelle MetaSound Source.

2.6.1 Tipi di dato in MetaSound

In MetaSounds, oltre ai tipi di dato comuni come float, int (nello specifico int32), string, bool e enum, sono presenti dei tipi specifici per la gestione dell'audio e della sua implementazione:

- **Audio**: buffer contenente un segnale audio, che può essere processato.
- **Trigger**: dato che serve a eseguire altri nodi, in maniera simili agli eventi in Blueprint.
- **Time**: valore di tempo espresso in secondi.
- **UObject**: asset presente nel progetto, solitamente un file audio.

2.6.2 Input, Output e Variabili in una MetaSound Source

Per fare in modo che una MetaSound Source possa comunicare con altri asset presenti nel progetto e che il segnale audio generato sia riprodotto, è necessario definire degli input e degli output di diverso tipo.

Di default, alla creazione di una MetaSound Source sono presenti un input e due output:

- **On Play**: input di tipo trigger che viene eseguito quando la sorgente viene riprodotta. Serve quindi a dare inizio alla riproduzione del segnale. Può essere eliminato dal grafo e sostituito da altri input personalizzati.
- **On Finished**: output di tipo trigger che viene eseguito quando la riproduzione della sorgente è conclusa e si occupa di liberare le risorse computazionali occupate dalla MetaSound Source. Può essere eliminato solo se si elimina l'interfaccia UE.Source.OneShot, specifica per i suoni che vengono riprodotti una volta sola.
- **Out Mono**: output di tipo audio che rappresenta il risultato del grafo in termini di segnale audio. Non può essere eliminato, ma può essere modificato dalle impostazioni della sorgente, le quali permettono di selezionare anche i tipi di output a più canali, ovvero Stereo, Quad, 5.1 e 7.1, per i quali saranno creati tanti nodi quanti sono i canali.

In aggiunta, è possibile definire altri input e output, di qualunque tipo supportato da MetaSounds. Ad esempio, si può definire un output di tipo trigger che segnali l'inizio della fase di rilascio di un inviluppo, oppure un input di tipo enum che definisca la modalità di funzionamento di un filtro. Per input di tipo float, è inoltre possibile impostare il **Value Type**, ovvero il tipo di valore controllato: lineare, logaritmico per le frequenze (espresso in Hz) o logaritmico per il volume (espresso in dB).

Infine, è possibile aggiungere delle variabili in una MetaSound Source. A differenza di input e output, le variabili non sono accessibili da Blueprint, quindi servono per una gestione interna dei dati. Inoltre, a differenza di Blueprint è anche possibile eseguire un **Get Delayed**, ovvero la lettura della variabile con un certo ritardo, utile per la gestione dei loop. Nello specifico, il nodo Get Delayed Variable

legge la variabile con un ritardo pari a un **block**, ovvero un'unità di tempo definita dal **block rate** impostato dall'utente. Il valore di default del block rate è 100 (inteso come blocchi letti al secondo), per cui la lettura è ritardata di 10 ms. Valori maggiori di block rate diminuiscono la latenza ma aumentano l'uso della CPU.

2.6.3 Il nodo Wave Player

Il nodo **Wave Player** permette di riprodurre asset audio (chiamati Sound Wave asset) presenti nel progetto. Esistono diverse versioni di questo nodo, a seconda del numero di canali della traccia (mono, stereo, quad, 5.1 e 7.1). Per riprodurre un suono, bisogna specificare l'asset corrispondente nell'input Wave Asset e dare un trigger nell'input Play. È possibile interrompere la riproduzione dando un trigger nell'input Stop. I *pin* output "Out X" contengono l'audio vero e proprio.

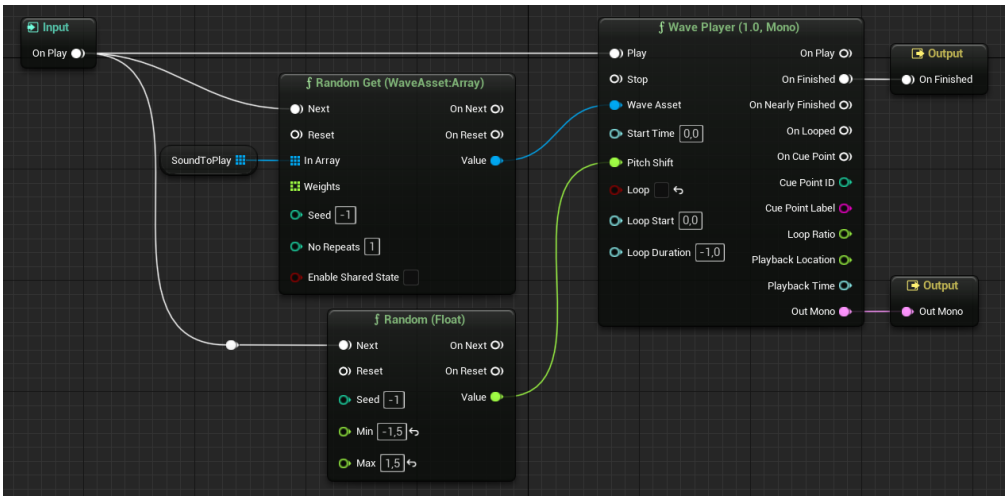


Figura 2.12: Esempio di randomizzazione del sample e del pitch

Inoltre, tramite i **pin** di input è possibile impostare delle opzioni di riproduzione aggiuntive, ovvero il tempo di inizio della riproduzione, la trasposizione in frequenza (Pitch Shift), un eventuale riproduzione in loop, tempo di inizio del loop e durata del loop.

Tramite i *pin* di output è invece possibile ricevere dei trigger che riguardano lo stato di riproduzione dell'asset, eventuali marker (Cue Point) inseriti all'interno del file audio e la percentuale di completamento di un loop.

Questo nodo permette di aggiungere casualità alla riproduzione di file audio, risultando utile nel caso lo stesso suono debba essere riprodotto. La figura 2.12 mostra un esempio di randomizzazione dell'asset audio da riprodurre a partire da un array e della quantità di trasposizione in frequenza tramite un valore float casuale.

2.6.4 I nodi generatori

In MetaSounds, oltre alla riproduzione di file audio già campionati, è possibile sintetizzare nuovi suoni a partire da forme d'onda base. Per generare delle forme d'onda si usano i nodi generatori (**Generators**). Di seguito sono elencati i nodi generatori attualmente presenti in MetaSounds:

- **Sine**, **Saw**, **Triangle**, **Square**: generano rispettivamente un'onda sinusoidale, a dente di sega, triangolare e quadra alla frequenza indicata. Tramite il *pin* di input Modulation è possibile modulare la frequenza con un altro segnale audio. Rappresentano la base per la maggior parte dei suoni sintetizzati.

- **Additive Synth**: genera l'onda risultante dalla somma di diverse onde sinusoidali a partire dalla frequenza di base in base alle frequenze delle armoniche indicate e alla loro ampiezza relativa.

- **Super Oscillator**: genera un segnale a partire da un numero di oscillatori interni compreso fra 1 e 16. Se presenti, gli oscillatori secondari possono avere una frequenza leggermente diversa da quello principale.

- **LFO** (**Low-Frequency Oscillator**): oscillatore a bassa frequenza (con frequenza massima pari al block rate), solitamente usato a frequenze al di sotto dei 20 Hz (frequenze subsoniche). Sono usati per creare effetti di modulazione o per dare variazione ai parametri.

- **Noise**, **Perlin Noise**: rispettivamente, generano rumore bianco o rosa e rumore di Perlin.

- **Low-Frequency Noise**: genera dei valori casuali alla frequenza specificata e dà in output una curva che interpola tra tali valori.

- **WaveTable Oscillator**: legge una *wavetable*, ovvero una *lookup table* contenente il periodo di un'onda, alla frequenza indicata.
- **WaveTable Player**: legge una wavetable a partire da un insieme di esse (**WaveTable Bank**) in base all'indice indicato.

Questi nodi consentono di definire all'interno di MetaSounds delle funzionalità affini a quelle dei sintetizzatori, con tutti i principali tipi di sintesi: additiva, sottrattiva, FM e WaveTable. Oltre che per motivi estetici, la sintesi risulta utile in quanto permette di risparmiare memoria, in quanto non richiede campioni e il calcolo viene fatto in tempo reale. È tuttavia da tenere in considerazione che un approccio basato sulla sintesi può avere un costo computazionale abbastanza elevato, per cui occorre regolare la complessità del grafo in base alla complessità complessiva del progetto.

2.6.5 I nodi inviluppo

Nel design di un suono, una delle caratteristiche da tenere in considerazione è la sua evoluzione nel tempo. Gli oscillatori visti nel sottoparagrafo precedente generano un'onda la cui ampiezza rimane costante nel tempo, che è una caratteristica che non trova riscontro con i suoni in natura. I nodi inviluppo (**Envelope**) permettono di definire una curva che descrive l'evoluzione della maggior parte dei suoni. In MetaSound, esistono due nodi inviluppo:

- **ADSR Envelope**: descrive una curva parametrizzabile che ha un iniziale tempo di attacco (Attack) fino al picco massimo, un certo tempo di decadimento (Decay) fino a un valore stabile di sostegno (Sustain) e un tempo di rilascio (Release). Permette di emulare il comportamento della maggior parte degli strumenti musicali reali.
- **AD Envelope**: versione più semplice dell'inviluppo ADSR, prevede solo un tempo di attacco e uno di decadimento, senza un valore di sostegno. È utile per modellare suoni impulsivi.

La figura 2.13 mostra un esempio di utilizzo dei nodi inviluppo per la sintesi di un effetto di *whoosh* (usato nel sound design per sonorizzare movimenti rapidi). In

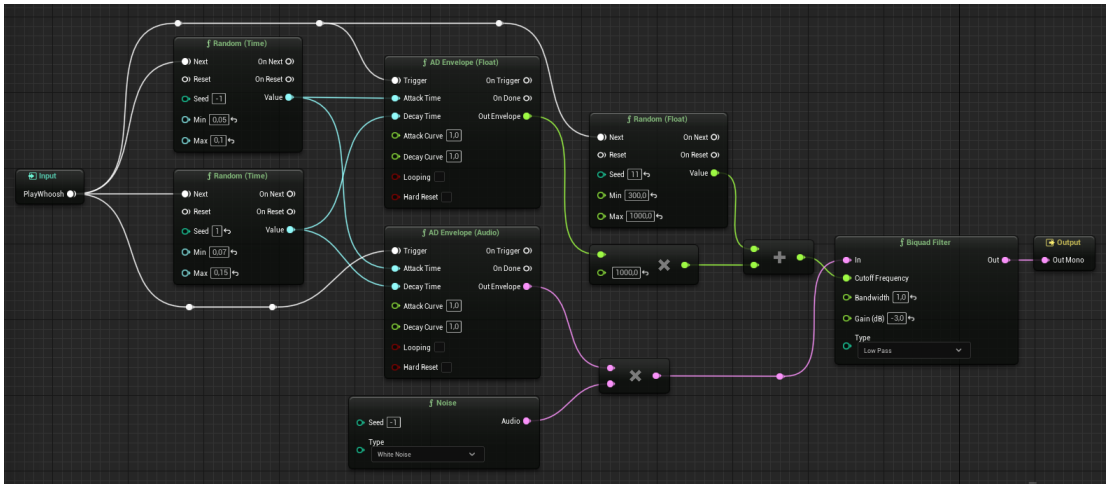


Figura 2.13: Esempio di sintesi di un whoosh tramite l'uso di rumore bianco e inviluppi AD

questo il suono è stato sintetizzato applicando un filtro passa basso al rumore bianco. È stato poi applicato un inviluppo AD con gli stessi tempi di attacco e decadimento sia al rumore che alla frequenza di taglio del filtro, così che il punto a massima ampiezza sia anche quello con le componenti spettrali a frequenza maggiore. Inoltre, sono stati usati valori di input randomici per l'attacco e il decadimento, così che il suono risulti leggermente diverso ogni volta che viene eseguito il trigger iniziale, dando maggiore naturalezza all'effetto.

Capitolo 3

Lavori Preliminari

In questo capitolo verranno analizzati i lavori svolti prima dell'inizio della fase operativa del progetto Egitto Immersivo. Questa fase preliminare ha avuto una duplice funzione: da un lato, ha reso possibile una conoscenza più approfondita delle possibilità offerte da Unreal Engine e ha permesso di sperimentare con gli strumenti disponibili; dall'altro, ha permesso di mostrare direttamente al cliente, ovvero il Museo Egizio, quale fosse la direzione artistica che Robin Studio avesse intenzione di prendere per la fase operativa di lavoro sul progetto.

Il capitolo sarà diviso secondo le due funzioni appena descritte. La prima parte presenterà alcune delle sperimentazioni ed esplorazioni degli strumenti realizzate durante il periodo di formazione. I risultati prodotti in questa fase non sono stati presentati al cliente, ma sono serviti internamente a Robin Studio per valutare le possibilità e i limiti offerti dagli strumenti utilizzati, in particolare Unreal Engine. La seconda parte sarà un'analisi della produzione delle proiezioni realizzate per la Cena di Natale del Museo Egizio, che, oltre alla funzione decorativa per l'occasione, ha avuto lo scopo di mostrare allo staff del Museo alcune proposte di direzione artistica per il progetto Egitto Immersivo.

3.1 Sperimentazioni iniziali

In questo paragrafo verranno illustrati alcuni risultati delle sperimentazioni realizzate durante la fase di formazione e di preparazione alla fase operativa sul progetto

Egitto Immersivo. In particolare, si vedranno alcune tecniche avanzate di creazione di VFX in Niagara, come il sampling di Static Mesh o l'uso di Moduli basati sui Distance Fields, delle tecniche per controllare i parametri di un Niagara System tramite una classe Blueprint, alcuni dei Materiali creati come proposta stilistica prima della Cena di Natale e l'analisi di una scena destinata alla proiezione durante la Cena di Natale scartata per incompatibilità stilistiche con le altre scene.

Come già detto, nessuno fra questi lavori è stato presentato al Museo Egizio, poiché sono stati usati internamente per ricerca di strumenti e stili. Molti degli esempi presenti in questo paragrafo saranno applicati a una mesh della statua della divinità egizia Sekhmet, facente parte della collezione del Museo Egizio[1]. La mesh proviene da un lavoro di fotogrammetria svolto in precedenza da Robin Studio.

3.1.1 Sampling di Static Mesh in Niagara

Il progetto Egitto Immersivo prevede l'uso di sistemi particellari renderizzati in tempo reale. Inoltre, il Museo Egizio è in possesso di scansioni 3D di molti degli oggetti presenti al suo interno. Per questo motivo la fase iniziale di sperimentazione si è basata sulla ricerca di un modo di sfruttare gli asset già disponibili per creare dei VFX real-time.

In Niagara, è presente il modulo **Static Mesh Location**, che effettua un'operazione di sampling della superficie della Static Mesh indicata e sovrascrive la posizione iniziale delle particelle (attributo **Particles.Position**) in modo che queste nascano in punti appartenenti alla superficie della mesh. Il Modulo Static Mesh Location va inserito nella fase di Particle Spawn dell'Emitter. Nei dettagli di Static Mesh Location occorre indicare la Mesh su cui effettuare il sampling nelle proprietà Preview Mesh e Default Mesh: la prima serve solo per la visualizzazione nell'anteprima dell'effetto nel Niagara Editor, la seconda è la mesh su cui verrà effettuato il sampling a meno di operazioni di *override*. Inoltre, è possibile scegliere il **Mesh Sampling Type**, ovvero il modo in cui si effettua il sampling della mesh. Sono disponibili tre opzioni:

[1] `https://collezioni.museoegizio.it/it-IT/material/Cat_252/`

- **Triangles**: viene effettuato il sampling delle facce della mesh. Una volta selezionata la faccia, la particella viene creata in un punto casuale di essa. È l'opzione che permette di avere la distribuzione più uniforme sulla superficie della mesh.

- **Sockets**: viene effettuato il sampling esclusivamente delle Socket presenti nella Static Mesh. Le Socket sono dei punti della Mesh (sia Static che Skeletal) indicati dall'utente, che possono anche non fare parte della sua superficie, e vengono solitamente usati per delle operazioni di *attaching*, ovvero l'imparentamento di un oggetto alla mesh in un determinato punto di quest'ultima. Le Socket possono essere usate, ad esempio, per fare l'*attaching* di un'arma a un avatar o di una porta a uno stipite. Di default, una mesh non ha nessuna Socket.

- **Vertices**: viene effettuato il sampling dei vertici della mesh. Le particelle, quindi, nasceranno solo in una serie limitata di punti. Può essere utile per avere un effetto "a griglia"

In tutti e tre i tipi di sampling è poi possibile impostare la **Sampling Mode**: il sampling può essere effettuato in maniera casuale o diretta (ovvero specificando l'ID di triangolo, socket o vertice), e può essere fatto o meno un filtraggio in base al Materiale applicato. La figura 3.1 mette a confronto i tipi di sampling Triangles e Vertices.

Quando si effettua il sampling di una Static Mesh, è importante tenere in considerazione la sua topologia: le zone con densità maggiore di vertici e facce avranno anche una densità maggiore di particelle. Questo è dovuto al fatto che il sampling viene effettuato in modo casuale sull'ID, e non tiene conto della densità. Inoltre, nel caso si effettui un sampling sui triangoli, bisogna valutare l'uniformità della loro area: poiché la selezione è più o meno uniforme su tutte le facce, quelle che avranno area minore, avranno una densità maggiore di particelle. Per avere quindi una creazione di particelle uniforme sulla superficie della mesh, occorre avere una topologia il più uniforme possibile.

Una volta effettuato il sampling, potrebbe essere necessario far muovere le particelle. In casi in cui la mesh campionata abbia una forma complessa, il movimento può rendere poco chiara la forma iniziale. Dopo alcuni tentativi,

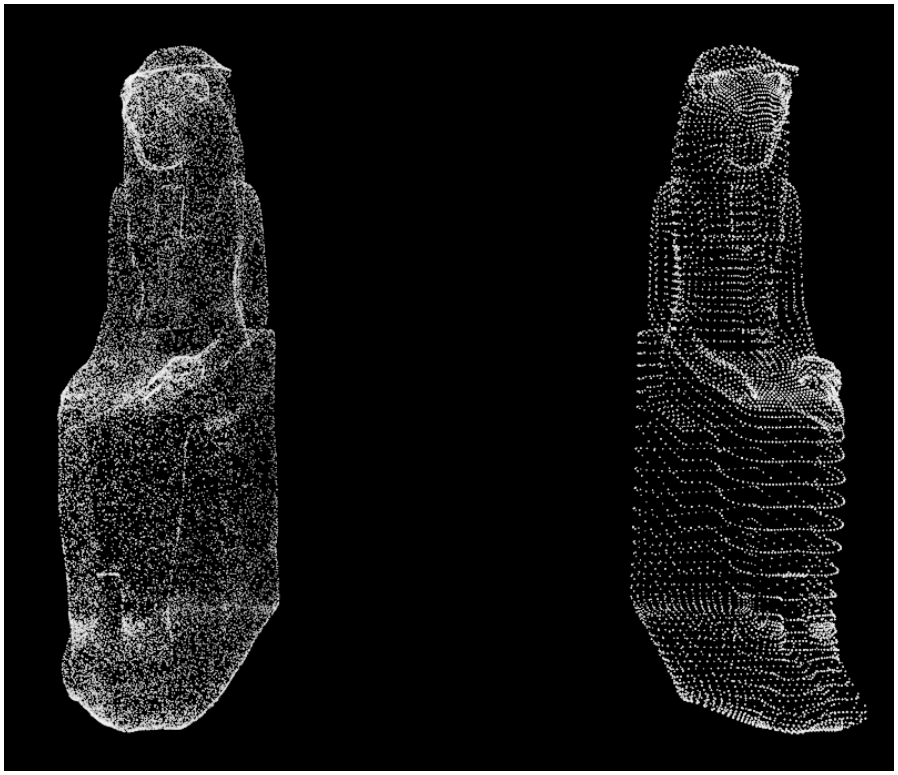


Figura 3.1: Confronto dei Mesh Sampling Type Triangles (a sinistra) e Vertices (a destra) del Modulo Static Mesh Location in Niagara

sono stati individuati due metodi per rendere più chiara la forma della mesh pur applicando il movimento:

- La prima opzione è quella di usare due Emitter: in uno le particelle che campionano la mesh rimangono ferme, nell'altro si applicano le velocità o le forze.

- La seconda opzione è quella di applicare le forze solo nella parte finale della vita delle particelle, ad esempio usando l'input dinamico Float From Curve e usando l'età normalizzata delle particelle come indice.

La figura 3.2 mette a confronto i tre approcci: a sinistra la forza viene applicata su tutte le particelle a partire dalla loro creazione, al centro la forza viene applicata solo a una porzione di particelle (nello specifico, l'80%), a destra la forza viene applicata solo a partire da un certo punto della vita delle particelle. È possibile

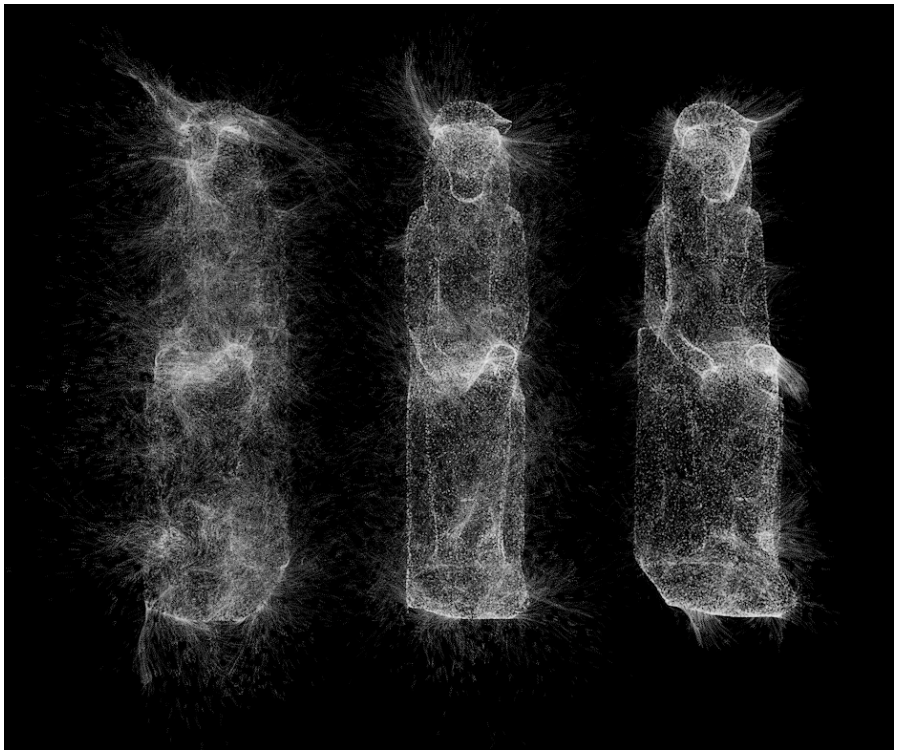


Figura 3.2: Diversi approcci all'uso di forze su un sistema di particelle: applicazione della forza su tutte le particelle (sinistra), su una porzione di particelle (centro), sulle particelle oltre una certo tempo di vita (destra)

notare come, nel primo caso, la forma non venga conservata, mentre negli altri due casi è possibile riconoscere la mesh di partenza. In tutti e tre i casi vengono visualizzate circa 50 000 particelle, e la forza applicata è di tipo curl noise, con intensità 50 e frequenza 5.

3.1.2 Controllo di Niagara System da Blueprint

È possibile avere un controllo ancora maggiore dei Niagara System tramite scripting in Blueprint. Combinando gli strumenti di programmazione offerti da Blueprint con le funzionalità dei Moduli offerti da Niagara, si creano nuove possibilità creative. Ad esempio, è possibile modulare la forza applicata alle particelle in base a un dato esterno, o il numero di particelle generate in base a una certa interazione dell'utente.

In Blueprint, esistono diversi nodi per il controllo di Niagara System. Una categoria di nodi utile in questo senso è **Set Parameter**, presenti nella categoria Effects. Essi permettono di sovrascrivere il valore di un parametro di un Niagara System definito dall'utente **User Parameter**. Questo parametro può poi essere usato come input in una proprietà qualsiasi in un qualsiasi Modulo del sistema. È possibile scrivere un tipo qualsiasi di parametro, ad esempio float, int, bool, Actor o Material.

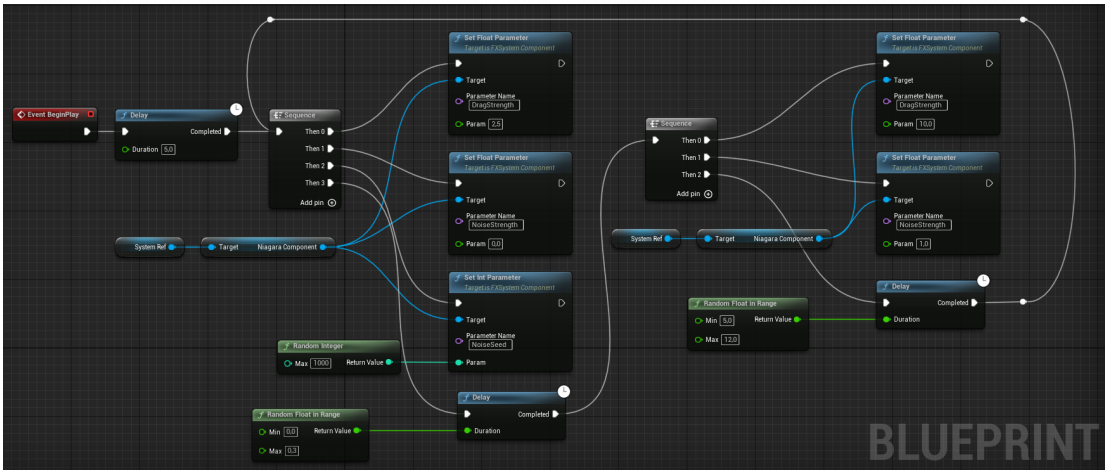


Figura 3.3: Esempio di utilizzo del nodo Set Float Parameter per randomizzare l'applicazione di una forza di tipo curl noise su un Niagara System

La figura 3.3 mostra un esempio di utilizzo di questi nodi, nella fattispecie **Set Float Parameter** e **Set Int Parameter**. Nel Niagara System sono stati definiti tre User Parameter: due parametri float chiamati DragStrength e NoiseStrength e un parametro int chiamato NoiseSeed. Il sistema contiene un solo Emitter, nel quale è presente un Modulo Static Mesh Location per il campionamento della mesh e i Moduli Curl Noise Force e Drag per definire la dinamica delle particelle. Questi ultimi sono stati impostati nel seguente modo:

- Nel Modulo Curl Noise Force la Noise Strength è stata impostata sull'input dinamico Float From Curve. In questo input è stata impostata una curva crescente, con indice NormalizedAge, così che la forza venga applicata in maniera proporzionale all'età; come Scale Curve è stato assegnato un altro input dinamico Multiply Float, e in uno dei due campi è stato assegnato il

parametro Noise Strength, nell'altro il numero 1000, un valore arbitrario che funge semplicemente da moltiplicatore. Inoltre alla proprietà Random Seed è stato assegnato il parametro NoiseSeed.

- Nel Modulo di Drag alla proprietà Drag è stato assegnato il parametro DragStrength.

È stata poi creata una classe Blueprint di tipo Actor, e al suo interno è stata definita la variabile SystemRef, di tipo Niagara Actor Object Reference, e sono state attivate le proprietà Instance Editable e Expose on Spawn, che permettono di assegnare alla variabile un Niagara Actor diverso per ogni istanza del Blueprint direttamente dall'Editor principale. Nel Graph Editor è stato inserito prima di tutto un nodo Event BeginPlay, così che lo script venga eseguito non appena avviata la play mode; ad esso è stato collegato un nodo di **Delay**, che ritarda l'azione successiva, a cui è stato assegnato il valore di 5 secondi per permettere al sistema di raggiungere un numero di particelle abbastanza elevato per rendere riconoscibile la forma. In seguito, è presente un nodo di **Sequence**, che permette di aggiungere in output un numero qualsiasi di *pin* di esecuzione e di eseguirli nell'ordine indicato. In questa prima sequenza di azioni vengono aggiornati i valori dei User Parameter del Niagara System attraverso i nodi Set Parameter: il valore di DragStrength a 2.5 per impedire alle particelle di allontanarsi dal loro punto di creazione, il valore di NoiseStrength a 0 come valore di riposo e il valore di NoiseSeed a un intero compreso fra 0 e 1000 per randomizzare la forma del pattern di rumore. Come ultimo comando, il nodo Sequence chiama un nodo Delay, con un ritardo casuale compreso fra 0.0 e 0.5 secondi. Di seguito, è inserito un altro nodo Sequence, che ha lo scopo di impostare i valori dei parametri di DragStrength a 10 e NoiseStrength a 2, così che alle particelle venga applicata la forza secondo il seed impostato nella sequenza precedente. L'ultimo nodo della sequenza è un altro nodo Delay, con un ritardo casuale compreso tra i 5 e i 12 secondi. Esso è poi collegato al primo nodo Sequence, creando così un loop.

In questo modo, è stato creato un sistema particellare sottoposto a un loop infinito, ma di durata variabile grazie ai nodi Delay con input casuale. Il sistema, ad ogni iterazione del loop, ha dei brevi momenti di quiete seguiti da un alcuni

momenti di forte presenza della forza di tipo curl noise, la quale ad ogni iterazione presenterà un pattern diverso grazie alla randomizzazione del seed del rumore.

Questo esempio mostra come, con un sistema particellare con pochi Moduli e uno script Blueprint abbastanza semplice, è possibile creare dei comportamenti abbastanza strutturati e non raggiungibili attraverso il solo uso di Niagara. La figura 3.4 mostra tre momenti di questo sistema: la prima è la fase iniziale, durante il primo delay, le altre raffigurano due iterazioni successive del loop. È possibile notare un cambio di pattern del curl noise.

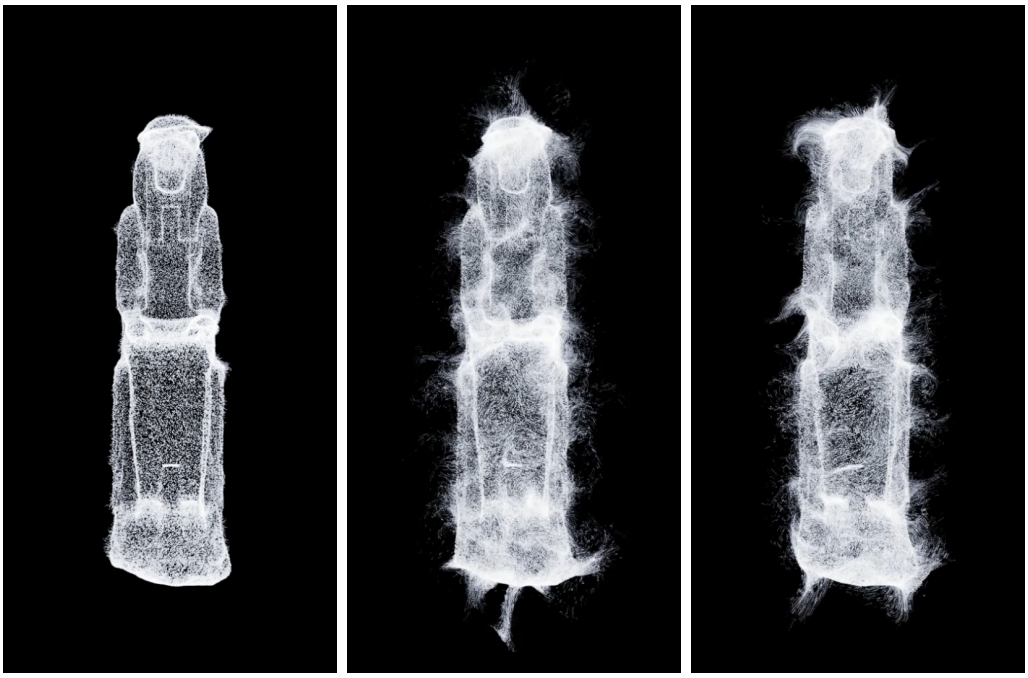


Figura 3.4: Esempi di risultati ottenuti tramite il controllo di un Niagara System tramite Blueprint

3.1.3 Creazione di Materiali complessi

In questo sottoparagrafo verrà analizzato un Materiale realizzato come proposta stilistica per le proiezioni della Cena di Natale del Museo Egizio. Lo scopo era quello di creare alcuni shader che non fossero fotorealistici e che avessero un'identità

Figura 3.5: Reference presa in considerazione per la creazione dello shader analizzato

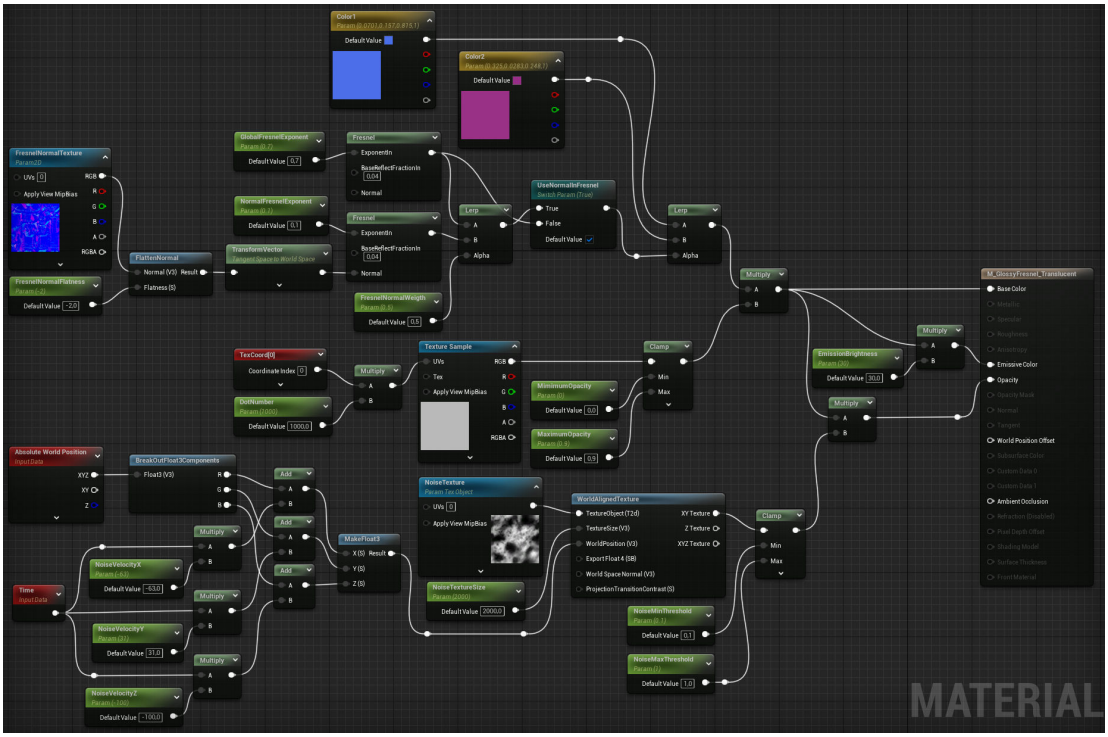


Figura 3.6: Esempio di Material Graph per la creazione di shader parametrici non fotorealistici

abbastanza forte. È stata selezionata una reference da Pinterest[2], proveniente dal social network Reddit, all'interno di una comunità dedicata al software Cinema 4D e pubblicata dall'utente qerplonk. La figura 3.5 mostra l'immagine usata come reference.

Per prima cosa, è stata creata una texture per regolare l'opacità del materiale. La texture è formata semplicemente da un cerchio bianco su sfondo nero. In questo modo, il pattern di opacità della texture sarà di tipo circolare. È stato poi creato il Materiale, e sono stati selezionati lo Shading Model Default Lit e la Blend Mode Translucent. È stato poi aggiunto un nodo Texture Sample, al quale è stato assegnata la texture appena descritta. Sono state poi manipolate le UV della texture per fare in modo che essa si ripeta lungo ognuna delle due dimensioni un numero di volte pari a quello indicato nel parametro DotNumber. Al risultato è stato poi applicato il nodo Clamp, che ha il ruolo di limitare i valori secondo i due estremi indicati, che sono stati parametrizzati.

In seguito, per fare in modo che il colore del Materiale cambiasse in base all'angolo di vista, è stato usato il nodo **Fresnel**, che simula l'effetto di Fresnel e dà in output il valore 0 quando la normale della superficie forma un angolo nullo con la direzione della camera, e valore 1 quando le due direzioni formano un angolo di 90°; i valori intermedi sono calcolati con una curva esponenziale, regolabile tramite il *pin* di input ExponentIn. Nel nodo Fresnel è inoltre presente il *pin* Normal, che permette di applicare l'effetto di Fresnel anche tenendo conto della *normal map* di un Materiale. Per far sì che la *normal map* venga letta in modo corretto è necessario usare un nodo **Transform**, che converte il vettore in input da uno spazio di coordinate a un altro. In questo caso, bisogna selezionare Tangent Space come Source e World Space come Destination. Tuttavia, quando a un nodo Fresnel si applica una *normal map*, specie se molto dettagliata, l'effetto sulla forma globale della mesh può risultare meno chiaro. Per questo motivo, sono stati usati due nodi di Fresnel, uno con in input la *normal map* della mesh, l'altro senza, in modo che i risultati prodotti da ciascuno di essi potessero essere combinati attraverso un nodo Lerp, secondo un parametro in input al *pin* Alpha. A ognuno dei due nodi è

[2] `https://www.reddit.com/r/Cinema4D/comments/ge3j3o/light_through_the_veins_c4d_octane/`

stato assegnato un parametro di regolazione dell'input ExponentIn, così da avere la massima personalizzazione. Alla *normal map* usata è stato collegato il nodo **FlattenNormal**, che permette di attenuarla o accentuarla. Inoltre, è stato usato un nodo StaticSwitchParameter, che permette di scegliere di non usare il nodo di Fresnel con l'input Normal. Il risultato di queste operazioni è stato poi collegato al *pin* di input Alpha di un ulteriore nodo Lerp, che interpola tra due colori, anche questi convertiti in parametri per aumentare il grado di personalizzazione delle istanze. Il risultato di questa interpolazione è stato usato sia nel colore di base che in quello di emissione.

In ultimo, è stata aggiunta la logica per fare sì che un pattern presente in una texture controllasse l'emissività del Materiale. È stato aggiunto un Texture Parameter per controllare il pattern di emissione, e poterlo modificare dalle Instance del Materiale. Esso è stato collegato al nodo **WorldAlignedTexture**, che effettua una proiezione della texture sull'oggetto in base alle coordinate del mondo, e non in base alle coordinate UV dell'oggetto stesso: questo permette di avere una texture che non cambia in base alle trasformate dell'oggetto. In questo caso, il nodo WorldAlignedTexture è stato usato perché l'UV mapping della mesh presentava dei *seam* visibili, quindi una normale proiezione di una texture non creata appositamente per la mesh avrebbe messo in risalto delle discontinuità, creando un effetto non piacevole. Per fare in modo che la texture si muovesse, è stato usato il nodo **Absolute World Position**, che restituisce la posizione del pixel in coordinate globali. A questo, viene aggiunto in maniera costante un offset, creato tramite il nodo **Time**, che restituisce il tempo in secondi a partire dall'avvio dell'editor o della Play Mode, moltiplicato per un valore diverso per le tre coordinate spaziali: questo crea un vettore velocità che descrive lo spostamento della texture. Per trattare separatamente le tre coordinate, è stato usato un nodo **BreakOutFloat3Components**, che restituisce in output gli elementi all'interno di un float3; queste componenti vengono sommate al vettore velocità e poi vengono ricomposte in un float3 tramite il nodo **MakeFloat3**. Il risultato è connesso al *pin* di input WorldPosition del nodo WorldAlignedTexture, così che la texture possa muoversi. È inoltre presente un parametro che controlla la dimensione della texture, ed è stato usato un float perché non era necessario modificarne la scala in maniera non uniforme. Il risultato è stato poi connesso a un nodo Clamp per limitare il

valore della texture. Infine, il risultato è combinato con le informazioni di colore della logica dell'effetto Fresnel e connesso all'input dell'emissività del Materiale, dopo essere stato moltiplicato per un parametro che ne controlla la luminosità. La figura 3.6 presenta il Material Graph del Materiale analizzato.

La figura 3.7 presenta il risultato di quanto detto: alla statua a sinistra è stato assegnato il Materiale precedentemente analizzato, alla statua a destra è stato assegnato un Materiale con un funzionamento simile, ma che applica il pattern di rumore all'opacità invece che all'emissività.

Figura 3.7: Esempi di Materiali non fotorealistici

3.1.4 Moduli Distance Fields

Un altro argomento oggetto di ricerca in questa fase preliminare è stato un metodo per fare sì che le particelle di un Niagara System, dopo essere generate sulla superficie, si muovano su di essa. È quindi necessario vincolare la posizione delle particelle nel loro ciclo di vita sulla superficie di una mesh. Una soluzione trovata è quella di usare i Moduli che basano il loro funzionamento sui **Distance Fields**. I Distance Fields sono campi scalari in cui ogni punto ha come valore la distanza dalla superficie più vicina. La figura 3.8 presenta una visualizzazione dei Distance Fields delle mesh presenti in una scena; l'immagine proviene dalla documentazione

online[3]. In Unreal Engine, i Distance Fields sono usati principalmente in fase di rendering per la creazione di Ambient Occlusion e ombre dinamici, ma sono resi accessibili anche ad altre parti del motore, così che gli utenti possano sfruttarli per delle funzionalità personalizzate. Ad esempio, i Distance Fields possono essere usati nei Materiali per renderli sensibili alla presenza di altre mesh nelle vicinanze o in Blueprint per controllare una funzionalità in base alla vicinanza di una mesh.

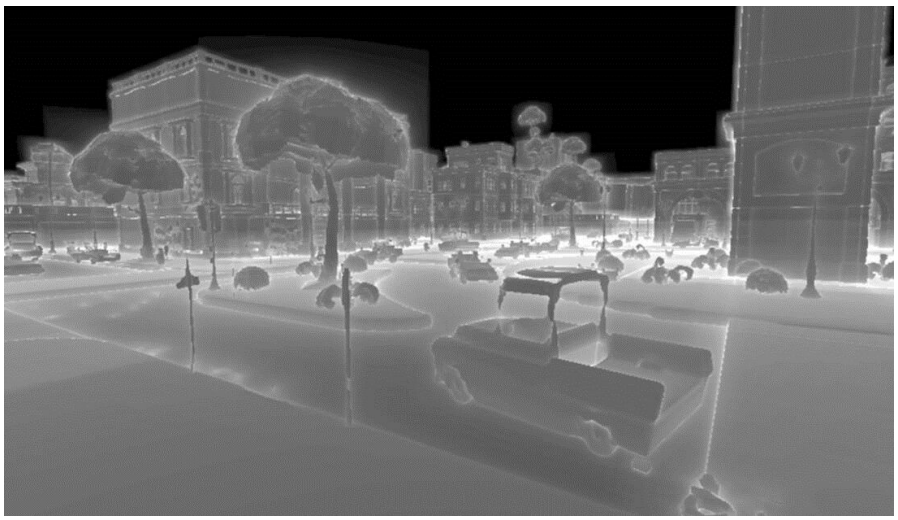


Figura 3.8: Visualizzazione dei Distance Fields delle mesh in una scena

In Niagara, sono presenti due moduli il cui funzionamento è basato sui Distance Fields:

- **Move to Nearest Distance Field Surface GPU**: pone le particelle sulla superficie con Distance Field più vicina.
- **Avoid Distance Field Surface GPU**: crea una forza che spinge via le particelle dalle superfici con Distance Field.

Entrambi questi moduli richiedono che l'Emitter che li contiene abbia come Sim Target la GPU, il che impedisce, ad esempio, di usarli combinatamente con moduli che mandano o ricevono eventi, e impone che i bound della simulazione siano fissati.

[3] `https://docs.unrealengine.com/5.0/en-US/mesh-distance-fields-in-unreal-engine/`

Figura 3.9: Esempio di utilizzo combinato dei Moduli di Distance Field e Curl Noise Force in un Niagara System

Figura 3.10: Esempio di utilizzo combinato dei Moduli di Distance Field e Gravity Force in un Niagara System

Per vincolare il movimento delle particelle sulla superficie di una mesh è quindi necessario usare il modulo Move to Nearest Distance Field Surface GPU. È necessario però capire anche quali moduli possano essere combinati con questo modulo per dare risultati soddisfacenti. Uno dei moduli che può risultare più utile è Static Mesh Location: se si specifica la stessa mesh su cui si muoveranno le particelle e si pone il Niagara Actor nelle stesse trasformate di essa, si può fare in modo che le particelle nascano sulla superficie della mesh e si muovano lungo essa, senza il rischio che si muovano sulla superficie di una mesh vicina.

Sono stati poi sperimentati moduli per il movimento delle particelle. Il modulo Curl Noise Force può essere utile, poiché il pattern di rumore lo rende adatto ad essere utilizzato su forme complesse. Bisogna regolare la Noise Frequency per far sì che il pattern si adatti a forma e dimensioni della mesh su cui si muovono le particelle. In generale, i valori di frequenza saranno bassi, per evitare di creare un movimento troppo caotico. Anche la Noise Strength va impostata su un valore basso per evitare troppo caos nel movimento. Inoltre, un eventuale *panning* del pattern (ovvero uno spostamento costante lungo un vettore) deve essere lento, poiché un movimento troppo veloce impedisce alle particelle di stabilizzarsi. Un'applicazione così "cauta" di questa forza è dovuta al fatto che il pattern del curl noise è tridimensionale, mentre il modulo Move to Nearest Distance Field Surface GPU vincola le particelle a un movimento bidimensionale. La figura 3.9 mostra un esempio di Niagara System in cui la Curl Noise Force mette in moto le particelle e il modulo Move to Nearest Distance Field Surface GPU le vincola alla superficie di una mesh.

Un altro tipo di forza che è possibile applicare insieme al modulo Move to Nearest Distance Field Surface GPU è Gravity Force. Vincolando il movimento verso il basso alla superficie di una mesh si possono ottenere effetti che simulano il movimento di acqua o sabbia sulla superficie. Anche in questo caso è stato riscontrato che valori troppo alti di forza (ad esempio il default di -981 lungo z) creano problemi, poiché seguono traiettorie che non corrispondono alla curva della mesh. Questo è probabilmente dovuto a una differenza sulle frequenze di campionamento della posizione della mesh e del Distance Field. La figura 3.10 presenta un esempio di Niagara System che combina questi due moduli. In questo caso, è stato usato il modulo **Kill Particles in Volume**, impostato in modalità Plane, e il piano è stato posto in corrispondenza della parte bassa della mesh, così

da evitare che le particelle arrivate in fondo alla mesh spinte dalla Gravity Force si accumulassero. La dimensione delle sprite è stata impostata sulla modalità Random Non-Uniform per dar loro una forma allungata (in questo caso la dimensione lungo Y è molto maggiore di quella lungo X) e l'allineamento è stato impostato su Velocity Aligned, che orienta la rotazione della sprite lungo il suo vettore velocità. Infine, per dare un'evoluzione nel tempo al sistema, l'offset lungo l'asse Z del modulo Kill Particles in Volume è stato animato, spostandolo dall'alto verso il basso.

3.1.5 Bozza della Scena del Nilo

In questo sottoparagrafo verrà analizzata una scena che è stata preparata per essere proiettata durante la Cena di Natale del Museo Egizio, ma in seguito scartata per incompatibilità stilistiche con le altre scene. Tuttavia, essa potrebbe essere riutilizzata per dei progetti futuri, ed è stata usata come base per il progetto dimostrativo di tesi, presentato nel capitolo 5.

La scena ha come soggetto principale il fiume Nilo, usato come espediente per mostrare alcune attrazioni legate al mondo dell'Antico Egitto e alcune scene di vita quotidiana del mondo antico. È da specificare che, trattandosi solo di una proposta di scena, ciò che viene presentato non è accurato né dal punto di vista storico, né da quello geografico, poiché i risultati ottenuti sono serviti solo come proposta alla direzione artistica di Robin Studio.

Come base per la scena è stato utilizzato un Landscape che riproduce il deserto, realizzato da un membro di Robin Studio in Unreal Engine per un precedente progetto. Per creare il Nilo è stato usato un Actor di tipo River Water Body, il quale simula il corso di un fiume a partire da una spline, che "scava" il Landscape, creando delle transizioni naturali fra acqua e terreno. Per ognuno dei punti della spline è possibile definire proprietà come la velocità dell'acqua, la larghezza e la profondità del fiume. Per ottenere una simulazione fotorealistica della luce diurna e del cielo, sono stati aggiunti degli Actor di tipo Directional Light, Sky Atmosphere, Sky Light, Volumetric Cloud e Exponential Height Fog. Questi tipi di Actor sono analizzati più nel dettaglio nel sottoparagrafo 3.2.3. In particolare, nell'Actor Exponential Height Fog è stato impostato un valore di densità molto alto per ottenere l'effetto "*god rays*", che in natura si verifica grazie allo scattering fra la luce solare e particelle presenti nell'aria, come pulviscolo o nebbia.

Figura 3.11: Due frame provenienti dalla bozza della scena del Nilo

Successivamente, sono state aggiunte le camere per fare in modo che la scena venisse costruita in funzione di esse. Poiché la proiezione della scena sarebbe stata fatta sulle quattro pareti di una sala, si è scelto di creare un *rig* di quattro Cine Camera Actor, ruotati di 90° l'uno rispetto all'altro lungo l'asse z. Questo permette di avere quattro rendering eseguiti in parallelo, con la possibilità di proiettarli sulle quattro pareti di una stanza. Il *rig* è stato posizionato al centro del fiume.

In seguito, è stata costruita la scena. Sono stati aggiunti modelli di rocce provenienti da Quixel Bridge per arricchire il paesaggio e per nascondere alcune

imperfezioni create dal River Water Body Actor. Poi, è stata aggiunta della flora in Foliage Mode. Per gli elementi artificiali, sono stati utilizzati dei modelli 3D con licenza CC Attribution: la Piramide di Menkaure[4], un obelisco[5] e un tempio di Horus di fantasia[6]. Infine, sono stati aggiunti delle Skeletal Mesh animate, provenienti da Mixamo[7], con delle animazioni che hanno lo scopo di riprodurre scene di vita quotidiana, come conversare, pregare o pescare. La figura 3.11 presenta due render provenienti dalla scena.

Inoltre, sono stati creati dei Niagara System per fornire una proposta stilistica alternativa al fotorealismo. Lo scopo era quello di creare un effetto di dissolvenza che ricordasse la sabbia al vento. Come prima cosa, è stata modificata la topologia delle mesh che sono state campionate nei Niagara System, per fare in modo che la dimensione delle facce fosse il più possibile uniforme, così che anche la distribuzione delle particelle fosse uniforme. In seguito, sono stati aggiunti due Emitter nel sistema per fare in modo che la mesh campionata fosse ancora distinguibile, come visto nel sottoparagrafo 3.1.1. Il primo Emitter esegue solo il sampling della mesh con il modulo Static Mesh Location, mentre al secondo Emitter sono stati aggiunti anche i moduli di forza Wind Force e Drag. Nel modulo di Wind Force è stata inserita una Turbulence per fare in modo che l'effetto del vento fosse credibile. In entrambi è inoltre presente il modulo Scale Color, tramite il quale è ottenuto un effetto di *fade out*.

Alle particelle è stato assegnato un Materiale creato appositamente per questo Niagara System. Il Materiale è stato creato a partire dal DefaultSpriteMaterial, applicato di default alle sprite nel modulo Sprite Renderer. In esso sono stati modificati la Blend Mode e lo Shading Model, che sono stati impostati rispettivamente su Masked e Default Lit. La Blend Mode Masked consuma meno risorse computazionali rispetto a Additive o Translucent, e in questo caso l'effetto desiderato era quello di una dissolvenza "rumorosa" o "granulosa", che è stata ottenuta tramite il nodo DitherTemporalAA. Lo Shading Model Default Lit è stato scelto

[4] `https://skfb.ly/oLnUu`

[5] `https://skfb.ly/6BtAz`

[6] `https://skfb.ly/oHWvH`

[7] `https://www.mixamo.com`

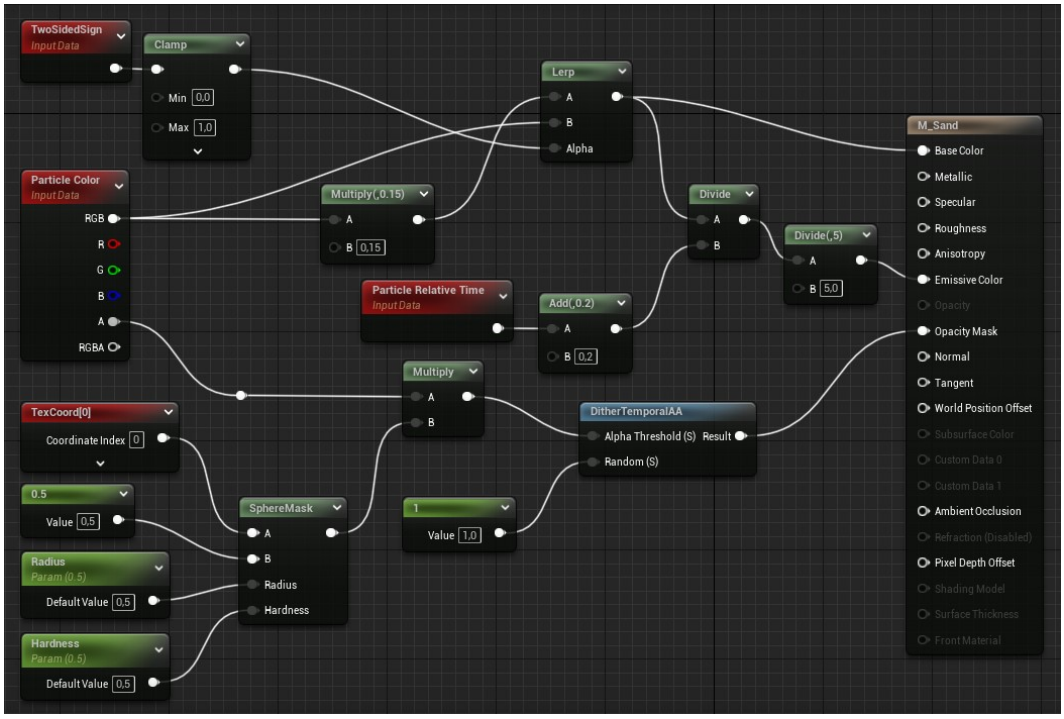


Figura 3.12: Materiale applicato alle particelle di sabbia del Niagara System creato per la scena del Nilo

Figura 3.13: Esempio di Niagara System creato per la scena del Nilo

per fare in modo che le particelle del sistema non fossero puramente emissive, ma avessero anche una componente di colore di base che interagisse con la luce della

scena. Inoltre, è stato usato il nodo Particle Relative Time per fare in modo che l'emissività delle particelle fosse più elevata all'inizio della vita delle particelle e poi diminuisse col tempo. La figura 3.12 mostra il grafo del Materiale appena descritto.

Il Niagara System appena descritto è pensato per essere applicato a una grande varietà di oggetti. Tuttavia, al variare delle dimensioni delle mesh, occorre modificare i parametri del sistema per fare in modo che l'effetto sia lo stesso. Un possibile approccio è quello di creare un sistema per ogni mesh sul quale si vuole applicare l'effetto, e in ogni sistema modificare il valore dei parametri. Un altro approccio, più efficiente e scalabile, è quello di impostare come input dei vari parametri degli User Parameter, i quali permettono di essere modificati per ogni istanza del Niagara System direttamente dall'Editor. Inoltre, questo approccio permette di modificare contemporaneamente il comportamento globale di tutte le istanze. La figura 3.13 mostra un esempio del Niagara System descritto applicato a una mesh (con licenza CC Attribution) che raffigura un vaso antico[8].

3.2 Proiezioni per la Cena di Natale del Museo Egizio

In questo paragrafo verranno analizzate le tre scene realizzate come proiezioni per la Cena di Natale del Museo Egizio. L'evento è avvenuto in data 21 dicembre 2023 presso la Sala Conferenze del Museo. Si trattava di un evento interno allo staff del Museo, quindi, oltre alla funzione estetica e decorativa, le proiezioni hanno anche avuto lo scopo di presentare allo staff le possibilità offerte dai mezzi e dalle competenze di Robin Studio e di avanzare alcune proposte stilistiche in vista della realizzazione del progetto Egitto Immersivo.

Dal punto di vista dei requisiti forniti dalla direzione artistica di Robin Studio, la durata indicativa delle proiezioni doveva essere di circa 10 minuti, da dividere in tre scene. Le scene dovevano avere soggetti diversi: una riguardante la civiltà dell'Antico Egitto, una riguardante la natura in Egitto e una con soggetto a scelta.

[8] `https://skfb.ly/oysKK`

La scena con soggetto a scelta aveva lo scopo di dare una forte identità stilistica alle proiezioni e dare una dimostrazione di quello che è possibile fare.

Durante la fase di pre-produzione è stato importante tenere in considerazione due aspetti, ovvero il contesto e il pubblico. Dal punto di vista del contesto, le proiezioni sono state realizzate per un evento natalizio, quindi, per quanto possibile, era opportuno adottare uno stile che si rifacesse all'estetica natalizia. Dal punto di vista del pubblico, poiché alcuni dei partecipanti alla cena erano studiosi o esperti di Egittologia, occorreva evitare di creare le scene in maniera approssimativa.

3.2.1 Scena 1: Tomba di Horemheb

La prima delle tre scene proiettate alla Cena di Natale del Museo Egizio aveva come vincolo quello di avere come soggetto qualcosa riguardante la civiltà dell'Antico Egitto. Il soggetto scelto è stata la tomba del faraone Horemheb, situata a Menfi. Il Dipartimento di Architettura, Ingegneria delle Costruzioni e Ambiente Costruito del Politecnico di Milano ha messo a disposizione di Robin Studio il modello 3D della tomba, proveniente da una fotogrammetria. Dall'intera struttura della tomba è stato scelto come soggetto il colonnato, e l'idea di base è stata quella di creare dei sistemi particellari che effettuassero il campionamento delle colonne.

Il modello inviato contava 180 000 facce, tutte triangolari, e una topologia irregolare. Come prima cosa, il modello è stato ridotto, poiché solo la sala del colonnato era utile per la proiezione. Questa operazione, eseguita nel software di computer grafica 3D **Blender**, ha portato il numero di facce a circa 30 000. La figura 3.14 mostra un confronto fra la pianta iniziale della tomba e il risultato della riduzione.

In secondo luogo, le colonne sono state separate individualmente dalla mesh principale, per far sì che queste fossero campionate individualmente. In seguito, è stata effettuata un'operazione di retopology del modello per far sì che il campionamento fosse eseguito nella maniera ottimale. La retopology è stata eseguita in maniera automatica, tramite il tool QuadriFlow Remesh di Blender, che genera una mesh con una topologia basata su facce quadrate (quads) che riproduce la mesh iniziale. Per fare in modo che la nuova mesh riproducesse in maniera corretta tutte le feature della mesh iniziale, il numero di facce è stato aumentato, portandolo a

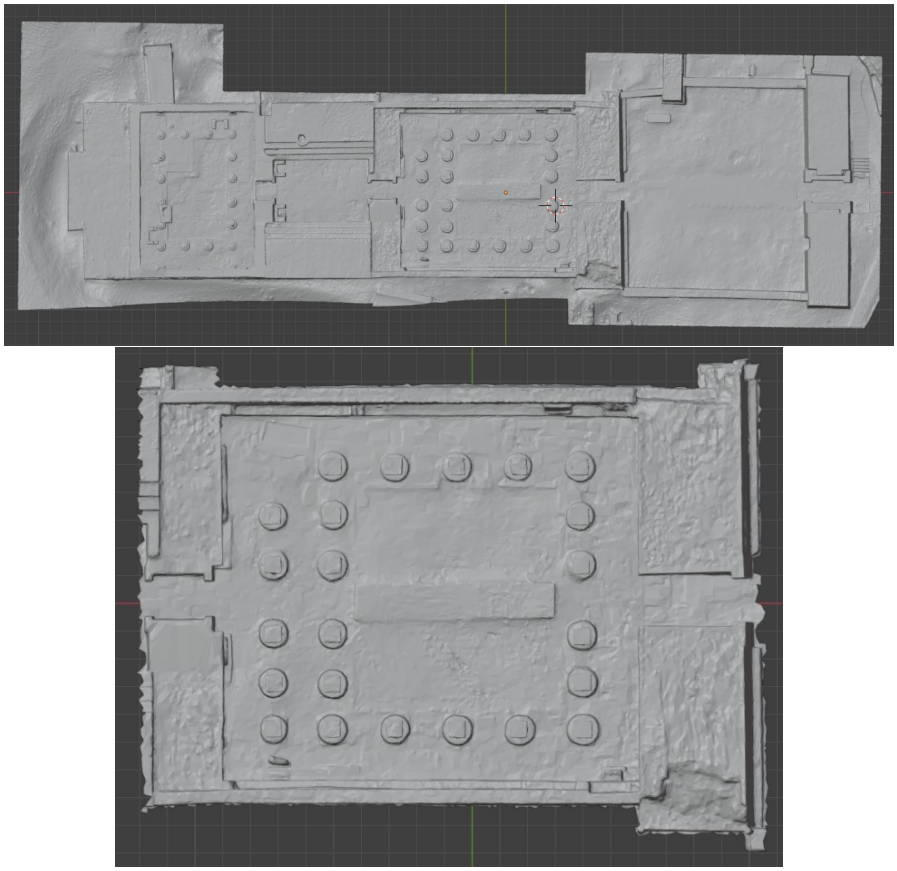


Figura 3.14: Riduzione della dimensione della mesh della Tomba di Horemheb

circa 100 000. La figura 3.15 mostra un confronto fra la topologia iniziale e quella risultante da questa operazione.

Il modello così ottenuto è stato poi passato ad altri dipartimenti di Robin Studio, che si sono occupati dei sistemi particellari e del compositing.

3.2.2 Scena 2: Papireto

La seconda scena proiettata alla Cena di Natale del Museo Egizio aveva come vincolo quello di raffigurare un soggetto che avesse a che fare con la natura in Egitto. È stato quindi deciso di raffigurare una pianta di papiro egiziano (*Cyperus papyrus*), che cresce nel delta del Nilo e veniva usato nell'Antico Egitto per la produzione di papiri, usati per la scrittura. L'idea era quella di campionare la mesh di un papiro e ricrearla tramite particelle in Niagara. La mesh di base è stata

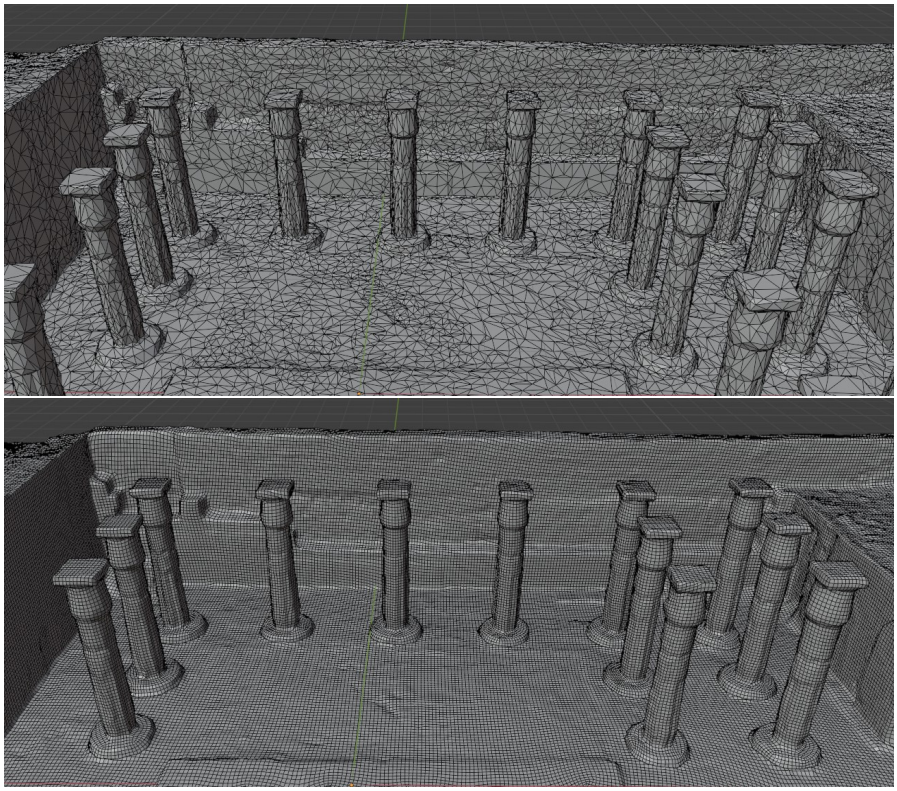


Figura 3.15: Confronto della topologia iniziale della mesh della Tomba di Horemheb con il risultato dell'operazione di retopology

scaricata dal sito Zeel Project[9]. Dalla mesh di base, contenente circa 30 piante, è stata estratta una mesh tramite Blender, contenente una sola pianta. Questo è dovuto al fatto che un campionamento di molte piante tutte vicine potesse dare un risultato caotico. La mesh è stata poi importata in Unreal Engine affinché fosse campionata.

In seguito, è stato creato un Niagara System in Unreal Engine. Nel primo Emitter creato al suo interno è stato inserito un modulo Static Mesh Location per effettuare il campionamento della mesh e fare in modo che le particelle nascano in corrispondenza della sua superficie. Come modulo di generazione delle particelle è stato usato Spawn Rate, e il numero di particelle generate al secondo è stato impostato su 50 000, mentre il tempo di vita delle particelle è stato impostato

[9] `https://zeelproject.com/42447-cyperus-papyrus-4.html`

come numero casuale compreso tra 0.5 e 1, risultando in un numero di particelle vive in un qualsiasi istante pari a circa 37 500. La Sprite Size, ovvero la dimensione delle particelle, è stata impostata su un numero casuale compreso tra 0.1 e 0.2. Il colore delle particelle è stato impostato su un colore simile all'oro, ma con una saturazione più bassa. Il secondo Emitter aggiunto è sostanzialmente uguale al primo, con l'unica differenza nel colore assegnato alle particelle, che in questo caso è stato impostato su rosso. Questa palette è stata scelta in quanto spesso associata al Natale.

Sono poi stati creati altri due Emitter sostanzialmente identici ai primi due, ai quali è stato aggiunto il modulo Curl Noise Force, con una Noise Strenght di 1, una Noise Frequency di 2, e un Pan Noise Field con valori intorno a 0.1 lungo le tre dimensioni. Questo Modulo permette di dare dinamicità al sistema, se pure con valori bassi di forza applicati alle particelle. Il Pan è stato aggiunto per far variare lentamente il pattern di rumore applicato alle particelle.

Infine, è stato aggiunto un quinto e ultimo Emitter, che ha come scopo quello di aggiungere delle particelle che diano l'effetto di luci natalizie. Per questo motivo le particelle di questo Emitter sono più grandi, luminose e con un maggior tempo di vita. Per la creazione delle particelle, è stato aggiunto un modulo Spawn Rate con un valore di 5. La dimensione è stata impostata su un valore casuale compreso tra 5 e 10, il tempo di vita su un valore casuale compreso tra i 3 e i 7 secondi e il colore su bianco, nel quale è stato impostato il Value a 10 per rendere le particelle più emissive. È stato poi creato un Materiale apposito, nel quale è stata inserita una maschera a forma di stella, proveniente dal pack "Infinity Blade: Effects", presente nel Marketplace di Unreal Engine[10]. Nell'Emitter è poi stato aggiunto un modulo **Sprite Rotation Rate**, che permette di far ruotare la sprite intorno al suo asse, e in questo caso è stato impostato un input dinamico di tipo Float From Curve per far sì che la rotazione acceleri e deceleri durante la vita della particella, raggiungendo il massimo della rotazione a metà vita.

Il sistema particellare è stato poi posizionato all'interno di una scena vuota. Sono stati effettuati quattro render diversi, ognuno dalla durata di 3 minuti (5400

[10] `https://www.unrealengine.com/marketplace/en-US/product/infinity-blade-effects`

frame, a 30 frame al secondo). Il primo render ha una risoluzione di 2160×3840 (Full HD 9:16) e ritrae il sistema particellare per intero. Il secondo render ha una risoluzione di 3840×2160 (Full HD 16:9) e ritrae il dettaglio della parte superiore del sistema, corrispondente alle foglie della pianta di papiro. Per gli altri due render è stato aggiunto un componente **Rotating Movement** al Niagara System in scena, che permette di dare una rotazione costante nel tempo all'Actor a cui è aggiunto. È stata impostata un Rotation Rate di 2 lungo l'asse z, così che il sistema facesse un giro completo in 3 minuti. Per questi due render sono state usate le stesse impostazioni di camera dei precedenti due. È stato anche usato un Actor di tipo Post Process Volume, che permette di regolare delle impostazioni di post-produzione del render, nel quale è stato aggiunto un effetto di Bloom per dare più uniformità al risultato. La figura 3.16 mostra due frame estratti dai primi due render.

Figura 3.16: Due frame estratti dai render del particellare usato per creare la scena del papiro

I render sono poi stati utilizzati in fase di compositing per ottenere il risultato finale.

3.2.3 Scena 3: Lanterne

La terza e ultima scena proiettata alla Cena di Natale del Museo Egizio non aveva nessun vincolo riguardo il soggetto, se non quello di non avere a che fare con

l'Egitto, e serviva come ulteriore proposta stilistica in vista del progetto Egitto Immersivo. È stato quindi deciso di creare una scena immersiva, con video diversi proiettati sulle due pareti. Come soggetto, sono state scelte delle lanterne cinesi che partono da uno specchio d'acqua e salgono verso l'alto. Inoltre, è stato scelto di ambientare la scena durante un tramonto.

Per realizzare la scena, sono stati prima di tutto modellati in Blender due tipi diversi di lanterne, così da avere variazione interna. In seguito, è stato creato un nuovo livello in Unreal Engine, nel quale sono stati aggiunti i seguenti tipi di Actor per dare realismo alla scena:

- **Directional Light**: simula la luce emessa da un oggetto infinitamente distante, per cui i raggi sono paralleli tra loro. È usato per simulare il sole.

- **Sky Atmosphere**: simula fisicamente il cielo e l'atmosfera, rispondendo in tempo reale con variazioni di colore alla direzione di una Directional Light.

- **Sky Light**: aggiunge all'illuminazione globale anche una componente di luce dipendente dal colore del cielo.

- **Volumetric Cloud**: simula fisicamente le nuvole, reagendo ai cambiamenti di colore di Sky Atmosphere.

- **Exponential Height Fog**: simula la nebbia, con una densità più alta nelle zone a bassa altitudine e che decresce via via all'aumentare dell'altitudine.

Per realizzare l'acqua è stato adottato un approccio non basato sul realismo. Per prima cosa è stato creato un Materiale e applicato su un Landscape. Nel Materiale il parametro di Roughness è stato impostato a 0, quello di Metallic a 1 e il Base Color è stato impostato su bianco. Questo crea una superficie perfettamente riflettente. Inoltre, è stata inserita una normal map nel pin Normal, usata per simulare le increspature della superficie dell'acqua. Ad essa è stato collegato un nodo **Panner**, che permette di muovere le coordinate UV di una texture a velocità costante. Come velocità del Panner sono stati usati valori dell'ordine di 0.01, così da creare un movimento molto leggero, ma percepibile.

In seguito, sono stati creati due Cine Camera Actor, posizionati circa al centro della scena nello stesso punto e con una rotazione relativa di 180°. Questo permette di avere due video complementari sulle due pareti della sala, aumentando l'immersività.

Figura 3.17: Due frame estratti dai due diversi video realizzati per la scena delle lanterne

Per simulare un gran numero di lanterne che si muovono in modo più o meno casuale, è stato creato un Niagara System. Nel sistema, sono presenti 6 diversi Emitter al fine di restituire un certo grado di variabilità all'interno della scena.

Verranno prima di tutto analizzati gli elementi comuni presenti in ognuno degli Emitter, e in seguito le differenze che li contraddistinguono. Prima di tutto, è presente un modulo di Spawn Rate, che garantisce una generazione costante di nuove particelle. È stato poi inserito un modulo Initial Mesh Orientation, che serve a impostare la rotazione con cui vengono generate le particelle di tipo mesh. In tutti gli Emitter è presente un modulo Shape Location, che genere le particelle secondo una certa forma primitiva, ma nei diversi Emitter tale primitiva cambia. Nella fase di Particle Update, è presente il modulo Scale Mesh Size, nel quale è stata inserita una curva che permette di far ingrandire le mesh da 0 alla loro scala all'inizio della loro vita e di farle sparire rimpicciolendole alla fine della vita; questo è utile per evitare che le particelle compaiano dal nulla e scompaiano nel nulla istantaneamente, poiché risulterebbe fastidioso. Per far muovere le particelle verso l'alto, viene usato un modulo Wind Force, con una Wind Speed casuale compresa fra i valori (-30, -30, 50) e (30, 30, 100), quindi sempre verso l'alto, ma con le componenti orizzontali variabili; è poi presente una componente di Turbulence per per simulare l'effetto reale del vento, nel quale c'è una componente di rumore. Per mantenere fisso l'orientamento è presente un modulo Aerodynamic Drag, nel quale è stato inserito un valore di Aerodynamic Rotational Drag molto alto per fare in modo che la rotazione delle particelle rimanesse sempre la stessa. Infine, nella fase di rendering sono presenti due moduli: un Mesh Renderer per le lanterne e un Point Light Component Renderer per la luce all'interno di esse. Sebbene i moduli Component Renderer siano sperimentali e non ottimizzati per il rendering in real-time, in questo caso sono stati usati perché il rendering è stato effettuato offline. È da notare che, per fare in modo che la luce fosse visibile attraverso le lanterne, nel Materiale applicato a queste ultime lo Shading Model è stato impostato su Subsurface, che permette alla luce di propagare attraverso le facce di una mesh.

Saranno ora analizzate le differenze fra gli Emitter presenti nel sistema. I 6 Emitter possono essere raggruppati in tre coppie. Le prime due di esse contengono due Emitter che si differenziano solo dal punto di vista estetico: nel Mesh Renderer del primo Emitter è stato assegnato il primo tipo di mesh di lanterna, dalla forma cilindrica, e nel Component Renderer è stato assegnato un colore della luce tendente al giallo; il secondo Emitter della coppia genera il secondo tipo di mesh di lanterna, dalla forma più tondeggiante, e la luce ha un colore tendente al rosso. Le tre coppie

si differenziano per tipo di forma primitiva presente nel modulo di Shape Location e moduli di forza applicati:

- Nella prima coppia è presente uno Shape Location di tipo Ring/Disc, che genera le particelle su una circonferenza. Questo tipo di forma fa sì che le camere siano "circondate" dalle particelle generate dal sistema. Non sono presenti moduli di forza aggiuntivi.

- Anche la seconda coppia ha come forma primitiva Ring/Disc. È anche presente un modulo Vortex Force di intensità casuale compresa fra -2000 e 2000, che ha lo scopo di creare più variabilità nel movimento delle particelle e aumentare la componente orizzontale del moto.

- La terza coppia emette solo mesh di lanterna del primo tipo e con luce gialla. In ognuno dei due è presente un modulo Shape Location di tipo Plane, che genera le particelle su un'area rettangolare. In questo caso l'area è un quadrato, e i due Emitter si differenziano solo per l'offset applicato ad esso, così da porre i due quadrati di fronte alle due camere. Questo tipo di forma fa sì che le particelle non vengano generate solo a una distanza fissa dalle camere, ma che ci siano anche particelle più vicine ad esse.

Nel livello è poi stato aggiunto un Actor di tipo Post Process Volume, tramite il quale è stata aggiunta una vignettatura. Il rendering è poi stato effettuato tramite un **Level Sequencer**, ovvero un Actor che permette di creare sequenze all'interno del livello, dando accesso a strumenti di editing. Nel Level Sequence l'orientamento della Directional Light è stato animato, in modo da restituire l'effetto di un tramonto. La figura 3.17 presenta due frame estratti dai due video renderizzati.

Capitolo 4

Workflow per Installazioni Data-Driven

In questo capitolo verranno analizzati i metodi di creazione e gestione delle strutture di dati in Unreal Engine, e di come questi possano essere utilizzati per controllare altri elementi del progetto, in questo caso i parametri di un Niagara System o una MetaSound Source. Nello specifico, verranno analizzati i concetti di struttura (struct) e Data Table, e un possibile workflow che comprende anche l'esportazione dei dati in formato CSV e la gestione di essi in software di fogli di calcolo. In seguito, verrà introdotto un esempio di workflow che prevede che i dati siano presenti in rete e che la modifica di essi abbia effetti in tempo reale sui dati all'interno del progetto, con dei vantaggi sia in termini di usabilità, sia in termini di reattività. Come tipo di struttura dei dati online è stato scelto il foglio di calcolo, grazie alla sua facilità d'uso e alla compatibilità con la gestione dei dati in Unreal Engine tramite le Data Table. Come software di fogli di calcolo è stato scelto Google Sheets, poiché gratuito e compatibile con un plugin di Unreal Engine, Runtime DataTable, che permette di leggere i dati da un foglio di calcolo e importarli nel progetto runtime, dando accesso a una risposta in tempo reale degli elementi del progetto alle modifiche a tali dati. La descrizione di tali metodi sarà accompagnata a degli esempi che mostreranno come creare un sistema del genere sia dal punto di vista dei sistemi audio e video, sia dal punto di vista dello scripting in Blueprint.

4.1 Gestione di Database in Unreal Engine

In questo paragrafo verrano analizzati i metodi di gestione di database in Unreal Engine, specificamente attraverso le Data Table, ovvero degli asset che consistono in tabelle che possono contenere tutti i tipi di dato di Unreal Engine e che sono compatibili con il formato CSV, di interesse per questo progetto di tesi. Verranno poi visti alcuni metodi di lettura dei dati da una Data Table tramite Blueprint e un esempio di uso di tali dati per il controllo dei parametri in un Niagara System. Il workflow presentato prevede un aggiornamento degli asset solo in "*design time*", quindi tramite l'Editor.

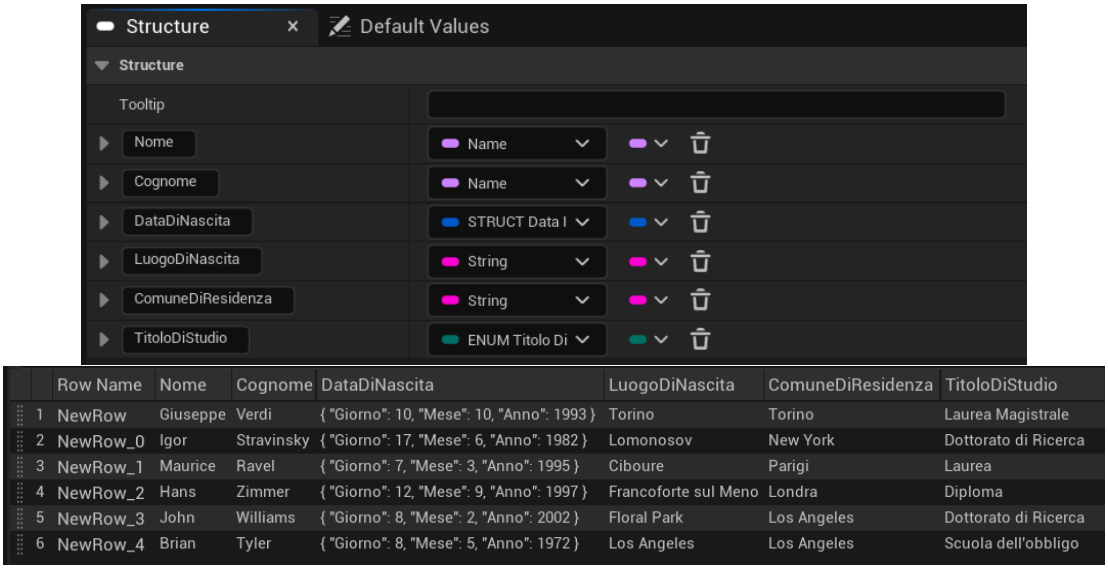
4.1.1 Structure e Data Table

In Unreal Engine è possibile creare delle strutture (**structure** o **struct**), ovvero dei contenitori di tipi di dato diversi tra loro che vengono trattati insieme. Il vantaggio delle struct è quello di poter usare un unico riferimento per accedere a tutti i dati contenuti in esse. Le struct sono a tutti gli effetti delle classi, e quindi come prima cosa vanno definite. Nella definizione di una struct è possibile aggiungere un numero indefinito di variabili, per ognuna delle quali si specifica il tipo e il nome.

Dal punto di vista dello scripting, in Blueprint sono presenti due categorie di nodi per trattare individualmente le variabili all'interno di una struct:

- **Break Struct**: prendono in input la struct e danno in output i valori delle singole variabili al suo interno.
- **Make Struct**: prendono in input tutti i tipi di variabili presenti all'interno di una struct e danno in output la struct stessa.

Una volta definita una struct, è poi possibile creare delle **Data Tables**, ovvero tabelle di dati basate su una struct. In una Data Table, le colonne rappresentano le variabili che formano la struct su cui è basata; è inoltre sempre presente la colonna **Row Name**, che indica in maniera univoca il nome di una riga, e serve per accedere al contenuto della riga stessa. Le righe sono possono essere interpretate come istanze della struct su cui è basata la Data Table. La figura 4.1 presenta un esempio di struct e di Data Table basata su essa; è da notare come fra le variabili nella struct sia possibile inserire tipi di dato più avenzati, come enum o altre struct.



	Row Name	Nome	Cognome	DataDiNascita	LuogoDiNascita	ComuneDiResidenza	TitoloDiStudio
1	NewRow	Giuseppe	Verdi	{ "Giorno": 10, "Mese": 10, "Anno": 1993 }	Torino	Torino	Laurea Magistrale
2	NewRow_0	Igor	Stravinsky	{ "Giorno": 17, "Mese": 6, "Anno": 1982 }	Lomonosov	New York	Dottorato di Ricerca
3	NewRow_1	Maurice	Ravel	{ "Giorno": 7, "Mese": 3, "Anno": 1995 }	Ciboure	Parigi	Laurea
4	NewRow_2	Hans	Zimmer	{ "Giorno": 12, "Mese": 9, "Anno": 1997 }	Francoforte sul Meno	Londra	Diploma
5	NewRow_3	John	Williams	{ "Giorno": 8, "Mese": 2, "Anno": 2002 }	Floral Park	Los Angeles	Dottorato di Ricerca
6	NewRow_4	Brian	Tyler	{ "Giorno": 8, "Mese": 5, "Anno": 1972 }	Los Angeles	Los Angeles	Scuola dell'obbligo

Figura 4.1: Esempio di struct e relativa Data Table

Uno dei vantaggi nell'uso di Data Table è la sua compatibilità con i formati **CSV** e **JSON**, entrambi molto comuni nell'ambito di database. In questa sede, sarà preso in esame solo il formato CSV. Il formato CSV consente una compatibilità completa con strumenti quali Microsoft Excel e Google Sheets, i quali permettono di esportare o importare file in formato CSV e interpretarli come fogli di calcolo. Unreal Engine, inoltre, permette di esportare le Data Table in formato CSV e di importare i file CSV e convertirli in Data Table. Questa compatibilità permette di integrare strumenti di gestione di fogli elettronici in Unreal Engine.

4.1.2 Uso di Blueprint per la lettura di una Data Table

Una volta creata o importata la Data Table contenente i dati necessari per il controllo di un elemento del progetto, è necessario poter accedere ai dati in essa presenti. In Blueprint sono presenti alcuni nodi per la lettura di Data Table. Di seguito sono elencati tre dei più importanti fra essi:

- **Get Data Table Row Names**: prende in input il riferimento a una Data Table e restituisce un array di variabili di tipo Name contenenti il Row Name di tutte le righe della tabella.

- **Get Data Table Row**: prende in input un dato di tipo Name e restituisce il contenuto della riga con tale nome (se presente) sotto forma di struct.
- **Get Data Table Column as String**: prende in input un dato di tipo Name e restituisce il contenuto della colonna con tale nome (se presente) sotto forma di array di stringhe.

La figura 4.2 mostra un possibile approccio per la lettura e il salvataggio di una Data Table. In questo caso l'idea è quella di salvare il contenuto di ognuna delle colonne presenti nella tabella sotto forma di array, così che sia possibile iterare facilmente su essi. Per prima cosa si aggiunge un nodo Get Data Table Row Names, e si specifica su quale Data Table effettuare la lettura. Questo darà in output un array di stringhe. Per iterare su un array si può usare un nodo **For Each Loop**, il quale permette di eseguire delle operazioni per ogni elemento dell'array in input e dà in output sia l'elemento corrispondente all'iterazione, sia l'indice di tale elemento nell'array. In seguito, si usa un nodo Get Data Table Row, a cui si dà in input l'elemento dell'array di nomi, e che dà in output la riga corrispondente sotto forma di struct. A questo punto bisogna usare un nodo Break per avere accesso alle singole variabili. Infine, si effettua un'operazione di **Add** sugli array, che permette di aggiungere l'elemento della cella alla fine degli array. Questo loop viene quindi ripetuto fino alla fine dell'array di nomi, quindi per ognuna delle righe della Data Table.

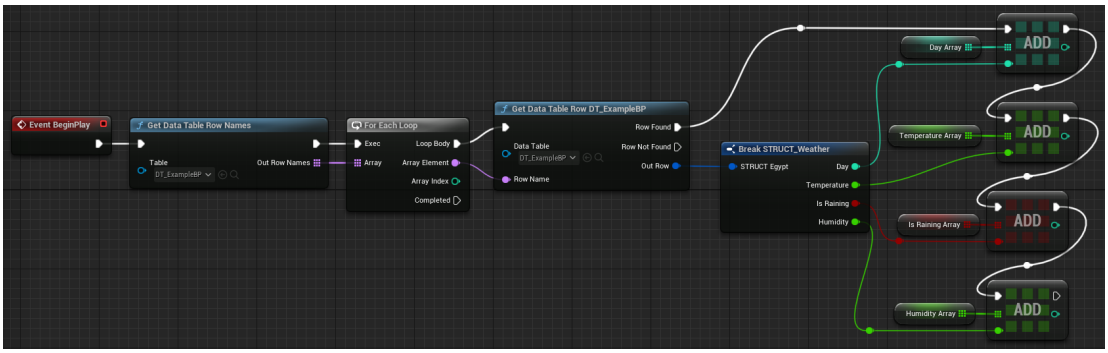


Figura 4.2: Metodo di lettura di dati da Data Table e scrittura su Array

È da sottolineare che questo approccio non è consigliabile per dataset di grandi dimensioni, poiché richiede una fase di lettura iniziale prima di poter procedere

con l'uso dei dati salvati. Nel caso la tabella sia di grandi dimensioni, un approccio migliore è quello di svolgere le operazioni direttamente sui dati nella struct, senza salvare i dati su array. Nel caso in cui i nomi delle righe siano già noti (ad esempio grazie a una formattazione coerente), allora è anche possibile evitare di usare il nodo Get Data Table Row name, e iterare in base al formato dei nomi, così da evitare il processo di scrittura dell'array di nomi.

4.1.3 Controllo di un Niagara System tramite Data Table

Il passo successivo è quello di usare le informazioni presenti all'interno della Data Table per controllare i parametri all'interno di un Niagara System, e creare così un VFX che reagisce a un database. Per fare ciò, bisogna mettere insieme quanto visto nel sottoparagrafo precendente con il controllo di Niagara System visto nel sottoparagrafo 3.1.2: una volta letta la Data Table, ed eventualmente salvato il suo contenuto in array, bisogna iterare su essi e usare dei nodi di tipo Set Parameter per scrivere il contenuto di essi nel Niagara System.

Viene ora presentato un esempio, nel quale è stato usato un Niagara System che simula la pioggia, creato in precedenza, durante la fase di formazione. In questo Niagara System sono stati aggiunti due User Parameter: Spawn Rate e WindIntensity. Il primo controlla il numero di particelle di pioggia vengono create, quindi simula l'intensità della pioggia; il secondo controlla il parametro di velocità della Wind Force applicata alle particelle, quindi simula l'intensità del vento; questo parametro controlla inoltre la quantità di Turbulence nella Wind Force, così da creare un vento dinamico. Sono stati usati due soli parametri per semplicità, ma il metodo può essere esteso a un numero qualsiasi di parametri di qualsiasi tipo.

In seguito, è stata creata una classe Blueprint di tipo Actor, nel quale sono state aggiunte una variabile di tipo Niagara Actor Object Reference, chiamata System Ref, una di tipo int, chiamata Index (che servirà nel loop principale), e due array di float per memorizzare i dati dalla Data Table. Per prima cosa, è stato implementato lo stesso metodo di lettura e salvataggio visto nel sottoparagrafo precedente. Una volta completato il loop, l'esecuzione passa a un nodo Sequence, il quale esegue un secondo loop, che ha lo scopo di scrivere i parametri nel Niagara System, con un delay fra un'iterazione e un'altra. In questo caso, il loop non viene creato con un nodo For Each Loop, poiché non è compatibile con il nodo Delay,

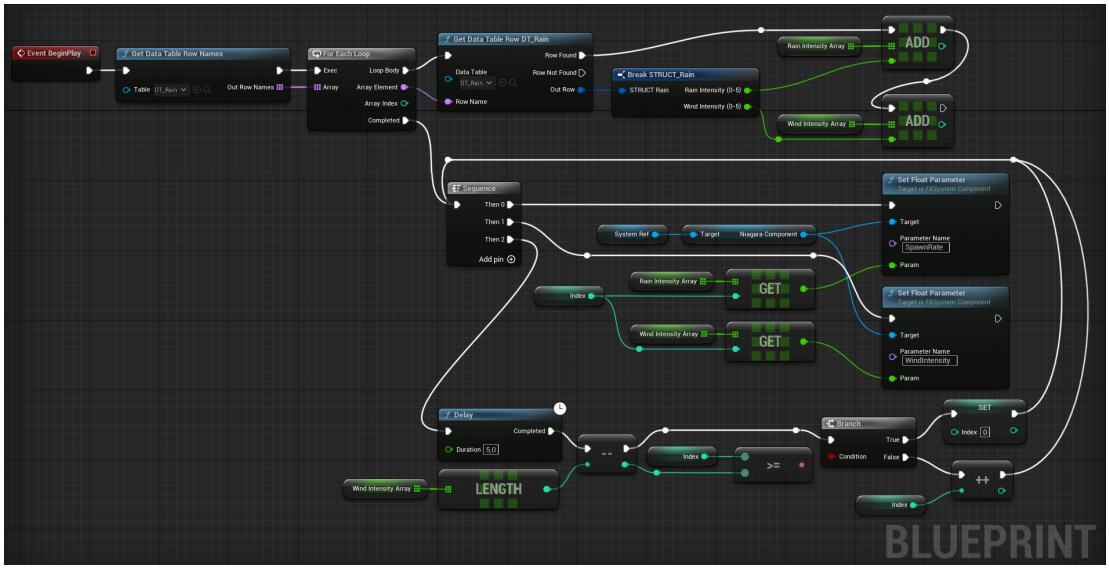


Figura 4.3: Esempio di uso di Blueprint per il controllo di un Niagara System con dati provenienti da Data Table

quindi viene implementato manualmente un loop tramite indice. Per questo motivo, la variabile Index viene inizializzata a 0. I primi due *pin* di esecuzione del nodo Sequence scrivono i parametri nel Niagara System. Vengono usati dei nodi **Get**, che danno in input l'elemento dell'array con l'indice indicato. Questi elementi vengono poi usati nei nodi Set Parameter per effettuare la scrittura vera e propria. Il terzo *pin* di esecuzione del nodo Sequence prima usa un Delay di 5 secondi per ritardare l'iterazione successiva, poi confronta l'indice con la lunghezza dell'array e usa un nodo **Branch** (corrispondente al costrutto if) per aggiornare l'indice: se si trova ancora nell'array, allora viene incrementato, mentre una volta raggiunta l'ultimo elemento dell'array, viene impostato a 0 per far ricominciare il loop. La figura 4.3 mostra il grafo appena analizzato. La figura 4.4 presenta due diverse fasi dell'evoluzione del sistema, da un'intensità di pioggia e vento bassa a una più elevata.

4.2 Lettura di dati online

In questo paragrafo verrà analizzato il metodo elaborato per la realizzazione di un Niagara System controllabile tramite dati online. Per prima cosa verrà presentato

Figura 4.4: Due diverse iterazioni del loop che controlla intensità di pioggia e vento di un Niagara System da Blueprint tramite dati presenti in una Data Table

il plugin Runtime DataTable, che è lo strumento che permette la comunicazione fra i dati presenti nel foglio di calcolo online e asset nel progetto di Unreal Engine. In secondo luogo verranno analizzati alcuni esempi di utilizzo di questo strumento per creare dei sistemi particellari controllati da dati online.

Il primo passo per l'implementazione del sistema presentato è stato scegliere il

tipo di struttura dati e gli strumenti online atti a contenere tali dati. Dopo una ricerca sugli strumenti compatibili con Unreal Engine, le opzioni erano sostanzialmente tre: Firebase Realtime Database, database SQL online (ad esempio PlanetScale) e lo strumento di fogli di calcolo Google Sheets. Dal punto di vista dei costi, tutti e tre sono strumenti gratuiti, ma con delle limitazioni: ad esempio, PlanetScale ha un limite di letture di righe mensile, Firebase ha un limite di dati salvati e scaricati mensilmente e Google Sheets ha solo un limite di dati salvati imposto dalla capienza di Google Drive. Un altro aspetto di fondamentale importanza riguarda la facilità d'uso e la manutenzione: l'aggiornamento dei dati online potrebbe essere la mansione di una persona senza competenze tecniche avanzate, per cui potrebbe non avere familiarità con l'uso database di tipo SQL o chiave/valore (come nel caso di Firebase). Per questo motivo, è stato scelto di utilizzare Google Sheets.

4.2.1 Il plugin Runtime DataTable

Nel workflow offline illustrato nel precedente paragrafo, il caricamento della struttura dati nel progetto avviene a *design time*: i dati vengono salvati in formato CSV e poi, tramite l'Editor, vengono convertiti in asset di tipo Data Table, che possono essere letti da Blueprint. Il plugin **Runtime DataTable** semplifica questo processo, poiché permette di leggere i dati online, da Google Sheets, quindi senza il bisogno di aggiornare i dati tramite l'Editor di Unreal Engine. Inoltre, la lettura dei dati viene fatta runtime, ovvero durante l'esecuzione. Questo permette di creare un progetto che legge i dati online in qualsiasi momento e cambia il suo comportamento in base ad essi, tutto durante l'esecuzione.

La caratteristica centrale di Runtime DataTable è la possibilità di creare asset di tipo struct a partire da dati in formato CSV a *runtime*. I dati possono essere letti da qualunque foglio di calcolo Google Sheets, ma nel caso di fogli con limitazioni di lettura è necessario creare una chiave di accesso e inserirla nel progetto. Le funzionalità del plugin sono offerte sotto forma di funzioni in Blueprint.

4.2.2 Blueprint per la lettura di dati da Google Sheets

In questo sottoparagrafo verranno analizzati i metodi forniti dal plugin Runtime DataTable per leggere dati da un foglio di calcolo Google Sheet e come integrare

questi metodi con il workflow di controllo di sistemi particellari presentato nel precedente paragrafo. Per continuità, verrà usato lo stesso esempio del VFX della pioggia usato nel paragrafo precedente.

Per poter scaricare i dati da Google Sheets, il primo nodo da usare è **BuildGoogleSheetDownloadLinkAndGetAsCsv**, che prende in input il link del foglio di calcolo e ne legge i suoi dati in formato CSV. A operazione conclusa, permette di chiamare una funzione da cui è poi possibile proseguire con le funzionalità dello script Blueprint. È sconsigliato usare il *pin* d'esecuzione in output, poiché c'è il rischio che le operazioni successive vengano eseguite quando l'accesso ai dati non è ancora stato completato, quindi è preferibile usare l'esecuzione basata sull'evento. L'input In Token Info è necessario solo nel caso il foglio di calcolo abbia restrizioni di accesso. L'input Operation Params permette di impostare delle opzioni avanzate, ovvero il nome dell'operazione da usare in eventuali callback e il tempo di timeout, superato il quale l'operazione di accesso deve considerarsi fallita. L'evento definito a partire dal pin Call On Complete dà in output una struct chiamata Callback Info, che contiene delle informazioni sull'operazione, ovvero il nome, l'esito e gli effettivi dati in formato CSV contenuti nel foglio di calcolo, passati come dati di tipo string. La struct Callback info viene poi passata al nodo **ValidateGoogleSheetsDownloadAndLoadBackupIfNeeded**, che prende in input anche il percorso della cartella in cui salvare il file contenente le informazioni provenienti dal foglio di calcolo online e il percorso della cartella da cui prendere le informazioni nel caso in cui l'operazione precedente non sia andata a buon fine. In alternativa, è anche possibile usare la funzione **MakeCsvInfoFromString**, che però non esegue salvataggi o letture di backup. In seguito, bisogna usare il nodo **UpdateArrayFromCsvInfo**, che aggiorna l'array di struct in input sulla base dei dati in formato CSV in input. Il nodo dà in output un dato booleano, che indica se l'operazione di aggiornamento dell'array è andato a buon fine. Questa informazione, ad esempio, può essere usata per un nodo Branch per continuare con le operazioni solo se l'aggiornamento è stato eseguito con successo.

A questo punto il procedimento è analogo a quello visto nel paragrafo precedente, con l'unica differenza che i dati sono salvati in un unico array di struct, invece che in array di variabili singole, quindi va usato un nodo Break per accedere alle singole variabili. Un'ulteriore differenza è data dal fatto che, quando si arriva a fine

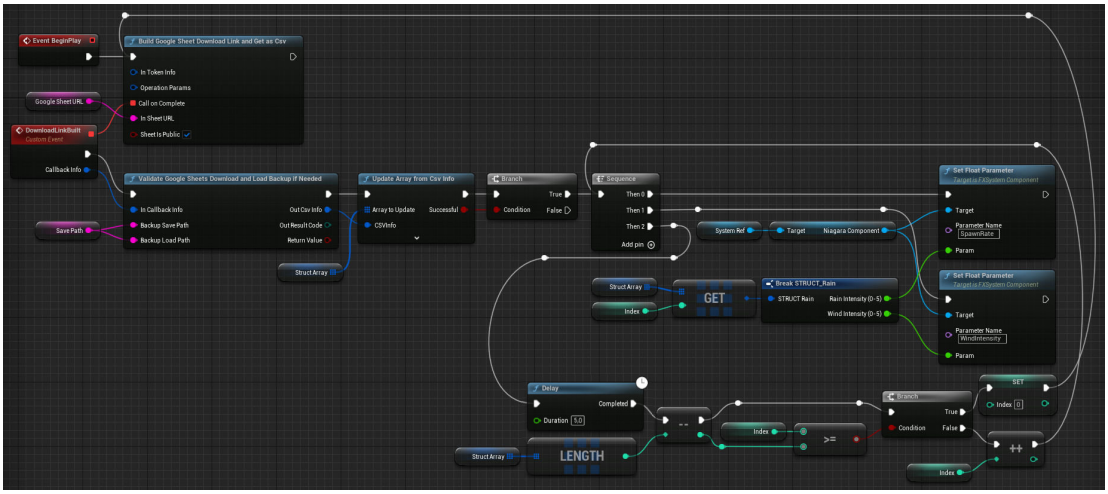


Figura 4.5: Esempio di uso di Blueprint per il controllo di un Niagara System con dati provenienti da foglio di calcolo online Google Sheets tramite plugin Runtime DataTable

ciclo e l'indice viene azzerato, la prima operazione che viene eseguita non è quella di modifica del primo parametro, ma viene ripetuta la lettura dei dati online, in modo che il comportamento del Niagara System possa variare anche fra due cicli successivi, se i dati vengono modificati. Questo, tuttavia, introduce alcune frazioni di secondo di latenza, ma il problema può essere risolto calcolando il tempo passato dall'inizio dell'operazione alla fine e sottraendolo al Delay successivo. In questo caso, tale metodo non è stato applicato perché la latenza è trascurabile rispetto alla durata di un ciclo. La figura 4.5 presenta il grafo Blueprint che implementa questo metodo.

4.2.3 Controllo in real-time di un Niagara System tramite Google Sheets

Gli esempi di controllo di Niagara System tramite dati in una tabella visti finora usano i dati nelle diverse righe in maniera sequenziale, regolando quindi l'evoluzione del sistema. In questo caso, quando i dati all'interno della tabella vengono modificati, bisogna riavviare l'esecuzione o aspettare la fine di un ciclo per vedere la modifica applicata al sistema. Un altro approccio possibile è quello di usare una tabella di una sola riga ed eseguire la lettura ad intervalli di tempo molto più limitati. In

questo modo, quando si modificano i dati all'interno della tabella, le modifiche dei parametri all'interno del Niagara System vengono eseguite con una latenza circa pari al periodo con cui vengono effettuate le letture del foglio di calcolo. Al momento, non è possibile fare in modo che le letture vengano fatte solo quando si verifica una modifica nel foglio di calcolo, quindi le letture vanno necessariamente fatte a cadenza regolare.

Per realizzare una dimostrazione di questo approccio, è stato creato un Niagara System *ad hoc*, pensato per rendere evidenti le variazioni di parametro. È stata scelta la forma astratta di un vortice, con il colore delle particelle che cambia dalla loro creazione alla fine della loro vita. Per questo tipo di effetto, è stato scelto uno Sprite Renderer con il Materiale di default. Le particelle vengono create tramite uno modulo Spawn Rate, che genera 10 000 particelle al secondo, la cui vita è un numero casuale compreso fra i 3 e i 5 secondi, perciò a regime sono presenti circa 40 000 particelle vive in ogni istante. È inoltre presente un modulo Shape Location, che fa in modo che le particelle siano generate in maniera casuale in un volume sferico, in questo caso di raggio 50; questo viene fatto per creare un effetto più naturale rispetto a un sistema le cui particelle si generano tutte nello stesso punto. Per quanto riguarda la fase di Particle Update, il modulo più importante per ottenere la dinamica del sistema è Vortex Force, che definisce il moto vorticoso delle particelle. È stato poi aggiunto un modulo Curl Noise Force per dare alle particelle un moto più caotico e quindi naturale. Per impedire alle particelle di avere un moto troppo caotico dovuto a una forza eccessiva, è stato aggiunto un modulo Drag, con un valore di Drag di 5. In seguito, per fare in modo che le particelle cambiassero colore, è stato aggiunto un modulo Interpolate Over Time, con il tipo di variabile impostato su Linear Color. Alla proprietà Rate of Change è stato assegnato un input dinamico che lo fa passare da 0 a 10 a circa metà della vita della particella, per fare in modo che il cambiamento di colore sia visibile e che la particella abbia il tempo di cambiare colore prima di morire. Per fare in modo che il colore cambi, è stato aggiunto il modulo Scale Color, e come input RGB è stato inserito il parametro Moving Average, scritto dal modulo Interpolate Over Time, mentre come input Alpha è stata inserita una curva discendente, per fare in modo che le particelle abbiano un *fade out*. Infine, sono stati creati 9 User Parameter per poter controllare il sistema da Blueprint:

- VortexForce, di tipo float, è applicato alla proprietà VortexForceAmount del modulo Vortex Force.

- NoiseStrength, di tipo float, è applicato alla proprietà Noise Strength del modulo Curl Noise Force.

- NoiseSeed, di tipo int, è applicato alla proprietà Random Seed del modulo Curl Noise Force.

- Color1R, Color1G e Color1B, di tipo float, sono applicati ai canali RGB della proprietà Initial Color Value all'interno del modulo Interpolate Over Time.

- Color2R, Color2G e Color2B, di tipo float, sono applicati ai canali RGB della proprietà Target Color Value all'interno del modulo Interpolate Over Time.

È stato poi creata una struct contenente tutti i parametri definiti nel Niagara System, e a partire da essa è stata creata una Data Table con una singola riga. La Data Table è stata poi esportata in formato CSV e caricata su un foglio di calcolo Google Sheets.

È stata poi creata una classe Blueprint di tipo Actor. Dal punto di vista dello scripting, il procedimento è quasi uguale a quello visto nel sottoparagrafo precedente, ma con la differenza che non viene eseguito nessun ciclo sulle righe della tabella, poiché la riga è solo una. Quindi, dopo aver eseguito la lettura dall'elemento di indice 0 dell'array di struct e applicato il contenuto ai vari parametri del Niagara System, viene applicato un breve ritardo con un nodo Delay (in questo caso 200 millisecondi), e dopo questo viene ripetuta la lettura del foglio di lavoro online. La figura 4.6 presenta il grafo Blueprint appena analizzato.

Questo procedimento permette di aggiornare il comportamento del Niagara System quasi istantaneamente quando viene applicata una modifica ai dati nel foglio di Google Sheets. È presente un piccolo ritardo dovuto sia al nodo Delay, la cui durata è modificabile, sia al processo di download e conversione dei dati dal foglio di calcolo. La figura 4.7 mostra due esempi di risultati ottenuti.

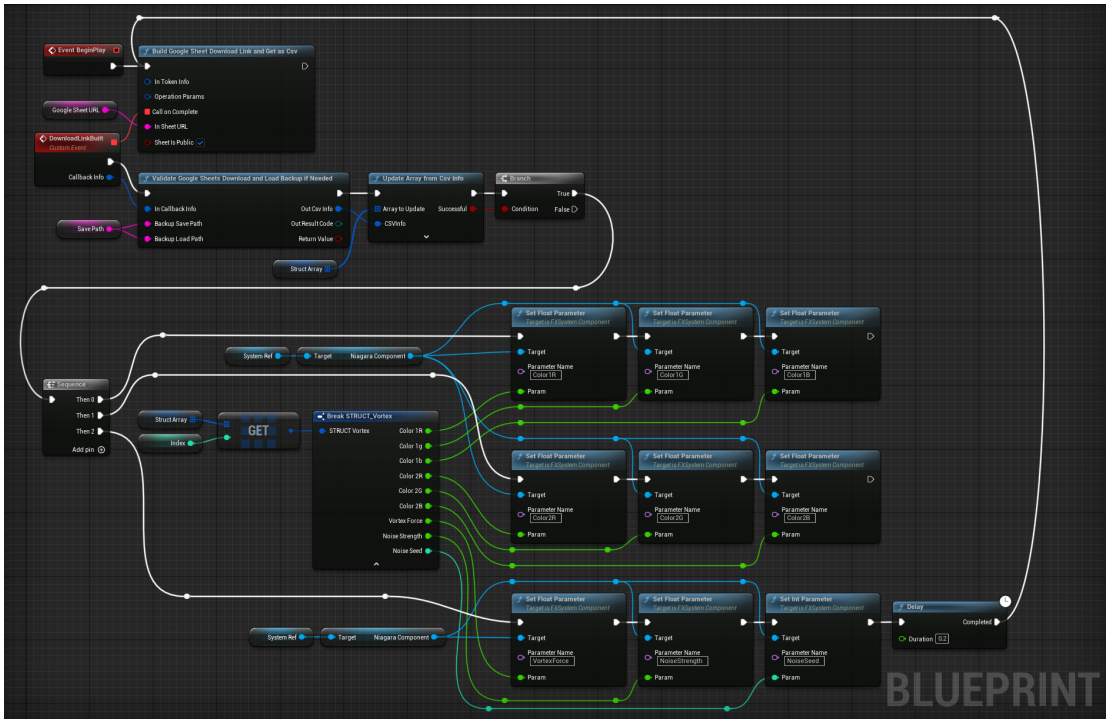


Figura 4.6: Uso di Blueprint per il controllo di un Niagara System in real-time tramite Google Sheets

4.2.4 Transizione graduale tra due valori

Negli script Blueprint per il controllo di parametri Niagara che sono stati analizzati fino a questo punto, i parametri vengono sovrascritti con i nuovi valori e il cambiamento avviene quindi in maniera istantanea. È possibile adoperare l'uso di alcuni nodi in Blueprint per far avvenire il cambiamento del valore del parametro senza soluzione di continuità, ovvero eseguendo un'interpolazione fra il valore corrente e il valore assegnato.

La figura 4.8 mostra un possibile approccio per risolvere questo problema. In questo caso, l'interpolazione viene fatta sul colore delle particelle del sistema, e il cambio di colore avviene ogni volta che si preme il tasto "P" della tastiera, scelto in maniera arbitraria. Il sistema a cui è stato applicato questo metodo è una versione semplificata del VFX del vortice visto nell'esempio precedente, poiché al suo interno è presente un solo valore, chiamato Color e di tipo Linear Color. L'interpolazione viene fatta fra il colore di partenza, chiamato Color Start, e il colore

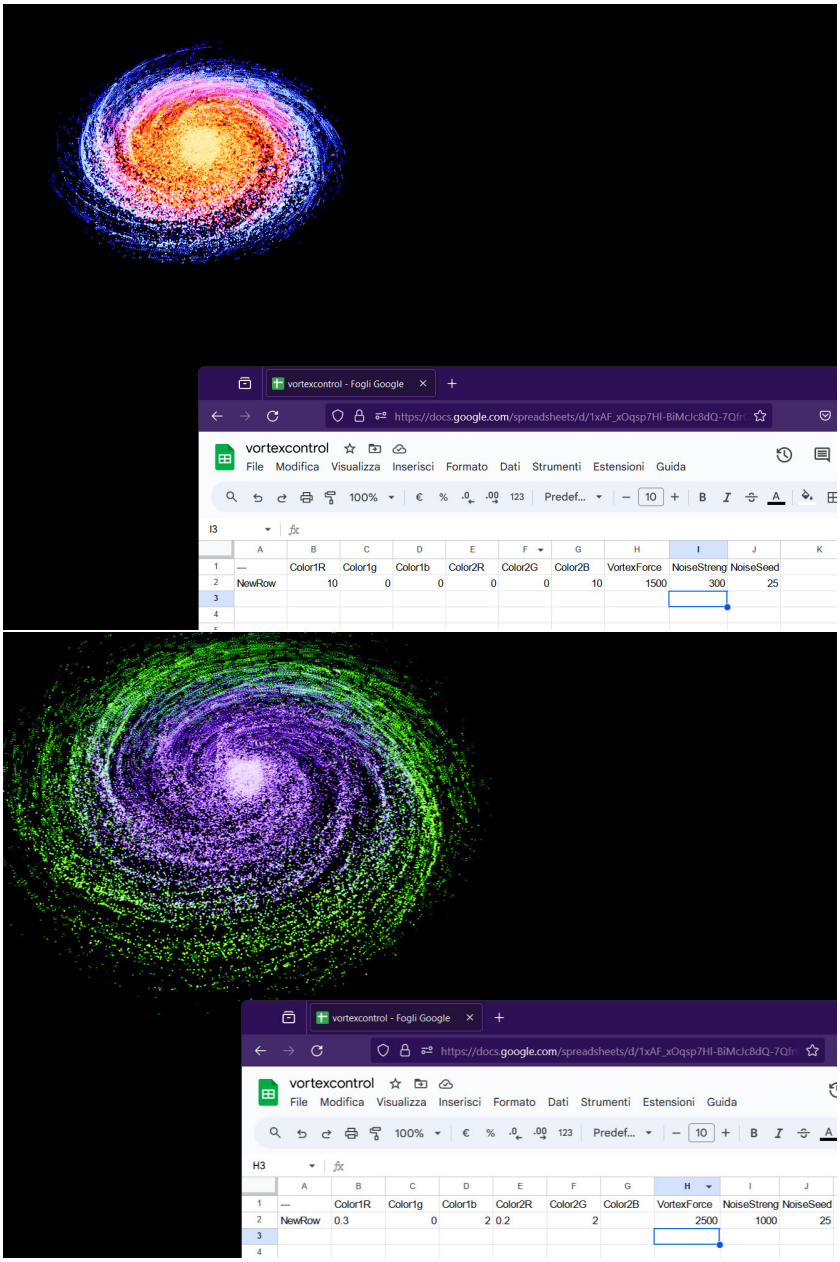


Figura 4.7: Due esempi di set di dati dati in input al Niagara System e relativi risultati, ottenuti in tempo reale

di arrivo, chiamato Color Target, a una velocità definita dalla variabile Play Rate, che rappresenta un moltiplicatore della velocità di riproduzione dell'animazione. Tutte e tre queste variabili sono modificabili tramite l'Editor.

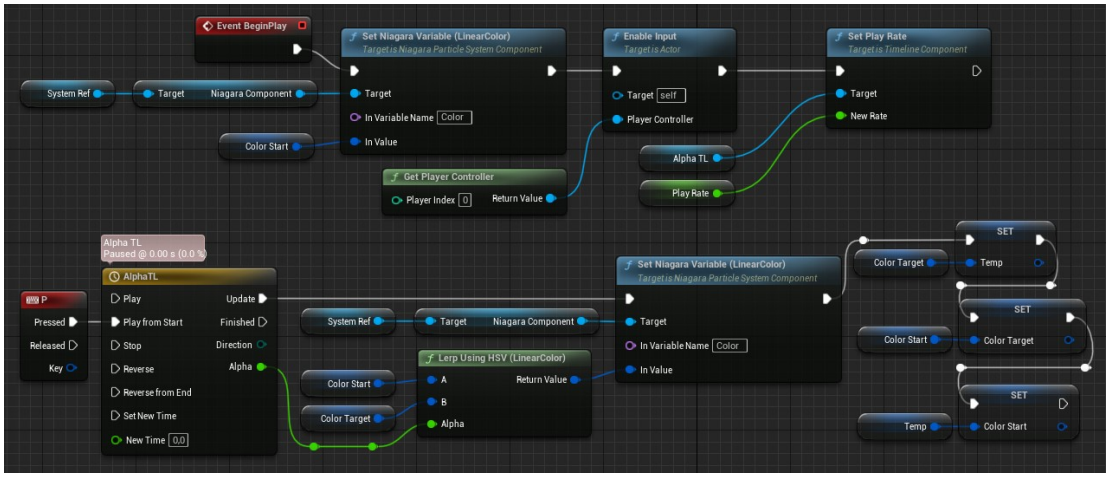


Figura 4.8: Possibile approccio per eseguire l'interpolazione fra due valori usati per controllare un Niagara System

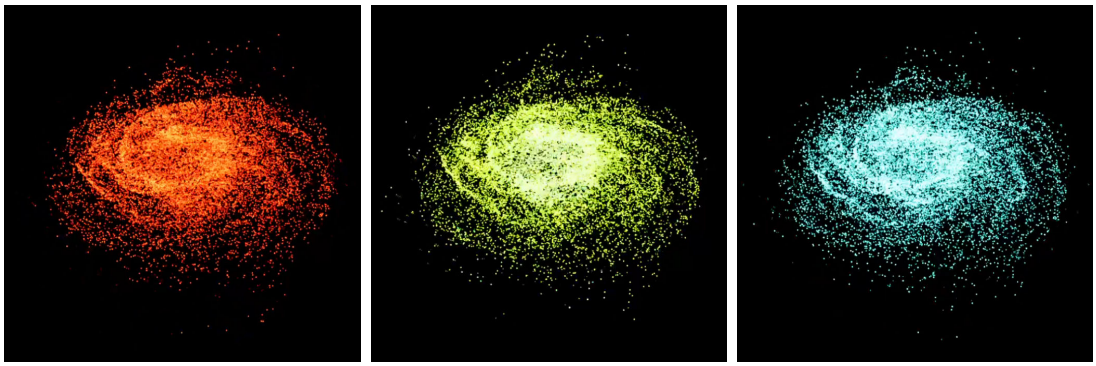


Figura 4.9: Esempio di risultato di interpolazione del colore delle particelle di un Niagara System tramite Blueprint

Il nodo Event BeginPlay è collegato a un nodo Set Niagara Variable, che serve a impostare il colore contenuto nella variabile Color Start come colore delle particelle del sistema. In seguito, è connesso un nodo **Enable Input**, che è necessario per far sì che gli input da tastiera vengano letti dal Blueprint Actor. Ad esso è connesso il nodo **Set Play Rate**, che permette di modificare la velocità di riproduzione di una Timeline.

È poi presente il network che descrive l'interpolazione fra i due colori. Per prima cosa è presente il **Keyboard Event** relativo al tasto P, che viene chiamato ogni volta che questo viene premuto. In seguito, è presente un nodo Timeline, all'interno

del quale è stata definita solo una traccia di tipo float, chiamata Alpha, che contiene semplicemente una rampa che va da 0 a 1 linearmente in un secondo. Questo permette di avere una Timeline con un tempo unitario, modificabile attraverso la variabile Play Rate. L'output della traccia Alpha è collegato al pin di input Alpha del nodo **Lerp Using HSV (Linear Color)**, che, analogamente al nodo Lerp del Material Graph (descritto nel sottoparagrafo 2.3.7), esegue un'interpolazione lineare tra i due valori in input A e B, che in questo caso corrispondono alle variabili Color Start e Color Target, secondo il valore di Alpha. In pratica, la Timeline in questo caso viene usata per animare il valore del peso relativo assegnato ai due membri dell'interpolazione. Il nodo Lerp usato in questo caso esegue l'interpolazione nello spazio colore HSV, che dà risultati migliori per quanto riguarda i valori intermedi, ma con un costo computazionale maggiore. Il risultato di questa interpolazione viene passato al nodo **Set Niagara Variable (Linear Color)**, che imposta il valore passatogli come valore dell'User Parameter all'interno del Niagara System. Infine, i valori di Color Start e Color Target vengono scambiati per far sì che l'interpolazione successiva avvenga fra i valori corretti. La figura 4.9 mostra un esempio di risultato di questa tecnica, ovvero un'interpolazione fra due colori con tinta (hue) opposta, rispettivamente 0 e 180, con un Play Rate di 0.2, che si traduce in una durata dell'animazione di 5 secondi.

Una possibile criticità di questo metodo è l'impossibilità di inserire una Timeline all'interno di una funzione in Blueprint, il che può portare ad avere un numero molto alto di nodi in un grafo se i parametri su cui si sta effettuando l'interpolazione sono molti, facendo sorgere problemi di scalabilità e leggibilità.

4.3 Estensione del workflow a MetaSounds

In questo paragrafo i metodi descritti nei due paragrafi precedenti verranno applicati su asset di tipo MetaSound Source. Questo permette di controllare contemporaneamente sia parametri video che audio all'interno dello stesso progetto, e, con un design adatto, di usare gli stessi dati per entrambi i tipi di asset.

Il controllo dei parametri audio delle MetaSound Source richiede l'uso di metodi diversi rispetto a quelli visti finora, sia per via del funzionamento diverso rispetto ai sistemi particellari, sia per via della natura diversa del fenomeno, poiché eventuali

difetti nel segnale audio possono generare degli artefatti sonori chiaramente udibili. Verranno mostrati metodi per generare dei segnali privi di artefatti e che evolvono in maniera naturale al variare dei parametri.

Nel paragrafo verranno prima esposti i metodi di controllo dei parametri audio tramite Blueprint, in un approccio non basato su set di dati. In un secondo momento verranno analizzati i metodi per eseguire aggiornamenti in real-time delle MetaSound Source a partire da dati online.

4.3.1 Controllo di MetaSound Sources tramite Blueprint

In questo sottoparagrafo si analizzeranno i metodi per creare una MetaSound Source che sia controllabile tramite script esterni e per creare degli script Blueprint che ne modifichino i parametri. In particolare, sarà necessario fare in modo da evitare di creare, all'interno del segnale acustico generato, delle discontinuità digitali, le quali possono dare origine a fenomeni acustici chiamati come **click**[6], che possono risultare fastidiosi. Per evitare questi fenomeni, sarà usato il metodo visto nel sottoparagrafo 4.2.4, ovvero l'uso di nodi Lerp e Timeline.

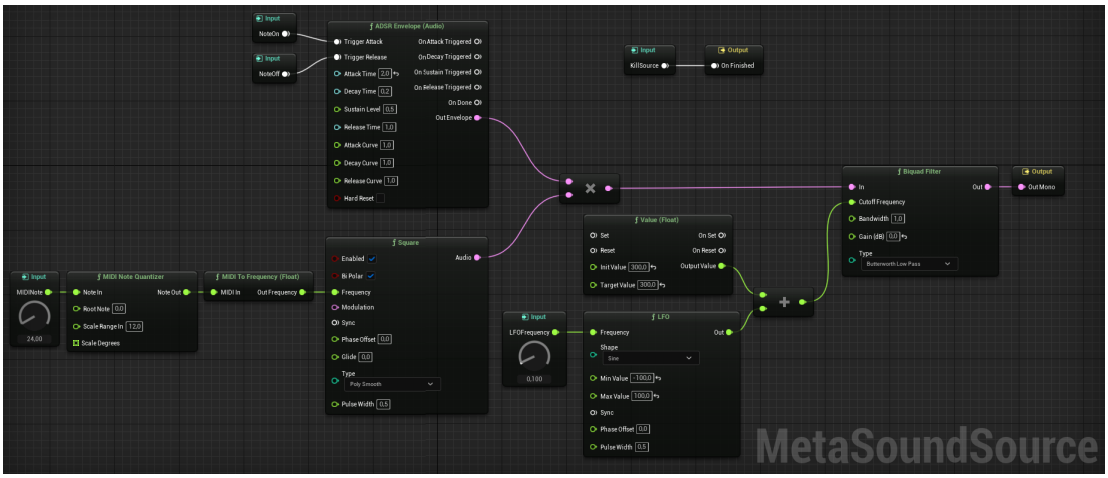


Figura 4.10: Grafo di una MetaSound Source con input parametrici

Verrà ora analizzato un esempio di MetaSound Source controllata da Blueprint. Per fare in modo che una MetaSound Source sia controllabile da asset esterni, è necessario inserire degli input, e impostare il loro range in maniera adatta. Nell'esempio analizzato, la MetaSound Source creata ha una struttura abbastanza

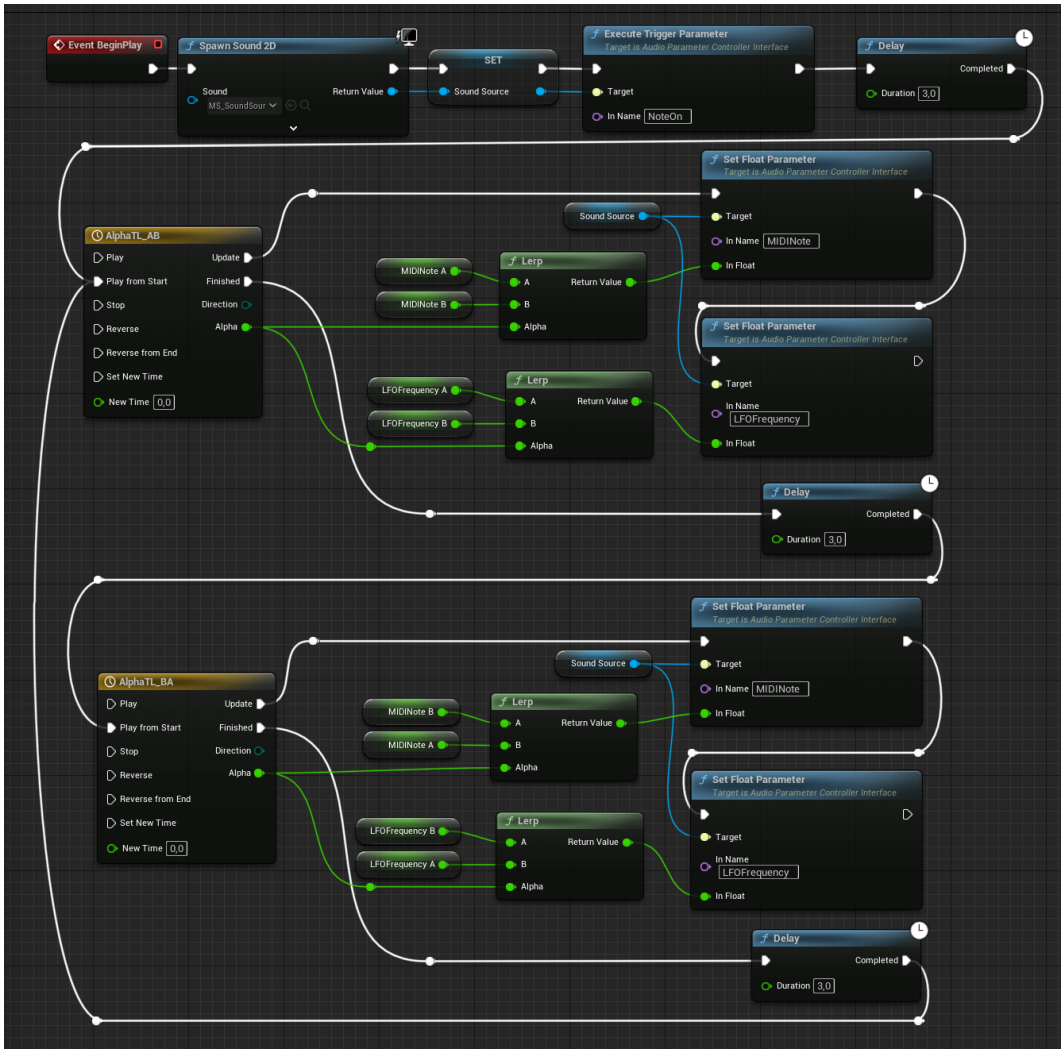


Figura 4.11: Esempio di controllo di parametri di una MetaSound Source tramite Blueprint

semplice: viene generata un'onda quadra di frequenza impostata tramite input, un inviluppo ADSR controlla l'evoluzione nel tempo dell'ampiezza di tale onda, e il risultato viene passato in un filtro passa basso la cui frequenza di taglio viene controllata da un oscillatore a bassa frequenza (LFO) di tipo sinusoidale; i parametri da controllare sono la frequenza dell'onda quadra e quella dell'LFO. Per generare un'onda quadra si usa il nodo **Square**, e per fare in modo che la sua ampiezza sia controllata da un inviluppo, si moltiplica il suo output audio con quello di un nodo **ADSR Envelope (Audio)**. Per controllare la frequenza dell'onda quadra è stato

aggiunto un input di tipo float, chiamato MIDINote. Esso è stato poi collegato a un nodo **MIDI Note Quantizer**, che serve a quantizzare il valore float in ingresso in uno che rappresenti il numero corrispondente a una nota nel protocollo MIDI. Il risultato è poi collegato al nodo **MIDI To Frequency**, che converte il numero MIDI della nota nota nella frequenza corrispondente alla stessa nota. Collegando il valore in uscita da tale nodo al *pin* Frequency dell'oscillatore si fa in modo di riprodurre esclusivamente le frequenze corrispondenti alle note presenti nella scala cromatica (ovvero le note presenti su un pianoforte, ad esempio). Per fare in modo che si seguisse l'inviluppo definito nel nodo ADSR Envelope, sono stati aggiunti due input di tipo trigger, chiamati NoteOn e NoteOff, e collegati rispettivamente agli input Trigger Attack e Trigger Release del nodo. Il risultato della moltiplicazione fra oscillatore e inviluppo è stato poi collegato a un nodo **Biquad Filter**, ovvero un filtro che modifica lo spettro del segnale audio in ingresso, impostato nella modalità Low Pass, che attenua o elimina le frequenze al di sopra della frequenza di taglio del filtro (Cutoff Frequency). Per fare in modo che la frequenza di taglio oscilli intorno a un valore centrale, sono stati sommati gli output di un nodo **LFO** in modalità seno e un valore float costante. È stato poi creato un input float, chiamato LFOFrequency, per controllare la frequenza dell'LFO. L'output del filtro è stato collegato all'output audio Out Mono. È inoltre presente un ulteriore input di tipo trigger, chiamato KillSource, e collegato all'output OnFinished, per fare in modo che si possa fermare l'esecuzione della MetaSound Source da comando in Blueprint. La figura 4.10 mostra il grafo MetaSound appena analizzato.

In seguito, è stata creata una classe Blueprint di tipo Actor per controllare gli input della MetaSound Source. Al nodo Event BeginPlay è stato collegato il nodo **Spawn Sound 2D**, che permette di istanziare un componente audio che non deve essere spazializzato, ed è usato per elementi audio extradiegetici. Dopo aver salvato il componente istanziato, viene usato il nodo **Execute Trigger Parameter**, che esegue l'input trigger con il nome corrispondente a quello specificato. Viene poi eseguito un Delay di 3.0 secondi per permettere all'inviluppo di arrivare alla sua fase di sostegno. In seguito, viene avviata una Timeline contenente una curva che va da 0 a 1 in 2 secondi. L'output della Timeline viene usato per controllare due nodi Lerp di tipo float che interpolano tra il valore attuale dei parametri MIDINote e LFOFrequency (salvati rispettivamente nelle variabili

MIDINoteA e LFOFrequencyA) e il loro valore obiettivo (salvati rispettivamente nelle variabili MIDINoteB e LFOFrequencyB). Per applicare i valori risultanti dalle interpolazioni vengono usati due nodi **Set Float Parameter**, che scrivono sugli input corrispondenti della MetaSound Source. Alla fine della Timeline viene eseguito un Delay di tre secondi, dopo il quale viene eseguita una seconda Timeline, che esegue l'interpolazione fra i valori di MIDINoteB e LFOFrequencyB e i valori MIDINoteA e LFOFrequencyA, scrivendo i risultati negli input della MetaSound Source. Alla fine di questa seconda Timeline, viene eseguita nuovamente la prima Timeline.

Il comportamento risultante è quello di un loop fra due diversi valori di parametri, che in questo caso corrispondono alle note MIDI 45 e 48 (a loro volta corrispondenti alle note A2 e C3 in notazione inglese) e alle frequenze di LFO 0.1 e 10. Il metodo di interpolazione tramite Timeline viene usato per evitare di creare delle discontinuità nel segnale audio generato, le quali risulterebbero in degli artefatti chiamati click, chiaramente udibili e potenzialmente fastidiosi. La figura 4.10 mostra lo script Blueprint appena analizzato.

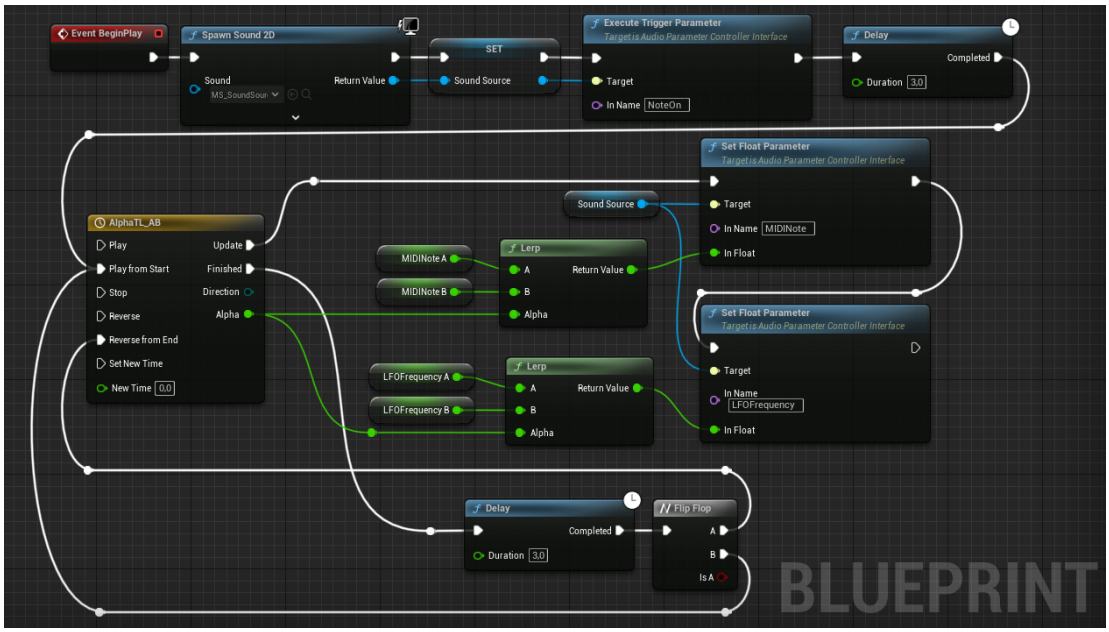


Figura 4.12: Approccio alternativo per la scrittura di parametri nel caso si esegua un loop fra due set di valori

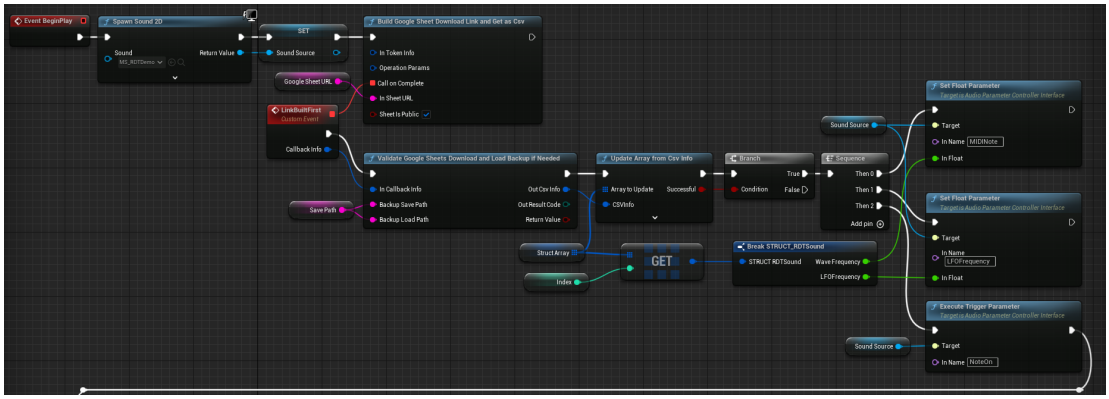
Infine, è possibile notare che, poiché in questo caso specifico si esegue sempre l'interpolazione fra due soli set di valori in loop, si può adottare una struttura leggermente diversa di script per usare meno risorse. Al posto del secondo nodo set di nodi usati per l'aggiornamento dei parametri dal set di valori B al set di valori A, si usa l'input Reverse From End del nodo Timeline, che legge le tracce al suo interno dalla fine verso l'inizio. Questo input permette al primo nodo Timeline (ovvero quello che guida l'interpolazione dal set di valori A al set di valori B) eseguire un'interpolazione inversa rispetto a quella descritta, quindi equivalente a quella svolta dal nodo Timeline precedentemente eliminato. Per fare in modo che i due metodi di esecuzione della Timeline si alternino, viene usato un nodo **Flip Flop**, che esegue in maniera alternata i due percorsi A e B in output. La figura 4.12 mostra lo script appena descritto.

4.3.2 Controllo in real-time di una MetaSound Source tramite Google Sheets

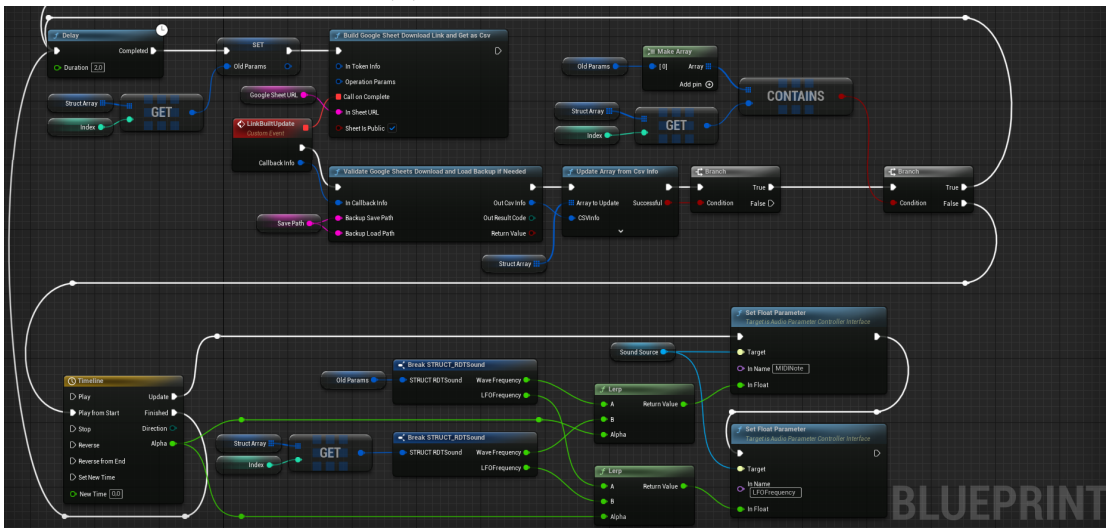
In questo sottoparagrafo verrà analizzato un metodo per controllare i parametri di una MetaSound Source tramite dati online, mediante l'uso del plugin Runtime DataTable. In questo caso l'aggiornamento dei parametri viene eseguito in real-time, grazie a una lettura continua dei dati online, analogamente a quanto visto nel sottoparagrafo 4.2.3. Il download dei dati viene eseguito come visto nei sottoparagrafi 4.2.2 e 4.2.3, ma nel caso di controllo di parametri audio la struttura dello script risulta diversa rispetto al controllo di Niagara System a causa del funzionamento a trigger delle sorgenti audio e del rischio di creare discontinuità nel segnale che possano risultare udibili. Per questo motivo, lo script si divide in due fasi, una di inizializzazione e una di aggiornamento.

- Nella fase di inizializzazione la MetaSound Source viene istanziata, ma non siene eseguito il trigger di inizio, poi viene eseguito il download dei dati online, i quali vengono scritti come parametri nella MetaSound Source; solo dopo questa scrittura viene eseguito il trigger per riprodurre l'audio.
- Nella fase di aggiornamento i parametri applicati vengono salvati in una variabile, poi viene eseguito il download dei dati online. I parametri precedentemente salvati vengono quindi confrontati con quelli appena scaricati: se

sono uguali, allora non viene eseguita nessuna scrittura sui parametri della sorgente; se sono diversi, allora si esegue una modifica dei parametri audio tramite interpolazione, al fine di non creare discontinuità.



(a) Fase di inizializzazione



(b) Fase di aggiornamento

Figura 4.13: Esempio di uso di Blueprint per il controllo in real-time di una MetaSound Source tramite Google Sheets

Di seguito viene analizzato nel dettaglio un esempio di script che adotta il metodo appena descritto. In questo caso, per semplicità, è stata usata la stessa MetaSound Source descritta nel sottoparagrafo precedente. All'evento Event BeginPlay viene collegato il nodo Spawn Sound 2D per istanziare la MetaSound Source, la quale viene poi salvata nella variabile SoundSource. In seguito, sono collegati i nodi

forniti dal plugin Runtime DataTable, come già analizzato nel sottoparagrafo 4.2.2. Se l'operazione di download dei dati e aggiornamento dell'array di struct è andata a buon fine, allora viene eseguita una Sequence che esegue prima due comandi di Set Parameter per sovrascrivere i parametri della MetaSound Source, ovvero MIDINote e LFOFrequency, poi esegue il trigger NoteOn tramite il nodo Execute Trigger Parameter, che dà inizio alla riproduzione dell'audio. A questo punto, la fase di inizializzazione termina.

La seconda fase, ovvero quella di aggiornamento, inizia con un nodo Delay, che ritarda di 2 secondi l'aggiornamento dei dati ad ogni esecuzione dello script. In seguito, i parametri contenuti nell'array di struct vengono salvati nella variabile OldParams. Vengono poi nuovamente scaricati i dati online tramite i nodi del plugin Runtime DataTable, e l'array di struct viene aggiornato. Se l'operazione è andata a buon fine, si esegue un confronto fra i dati contenuti nell'array di struct e quelli salvati nella variabile OldParams sono uguali. Unreal Engine non permette di eseguire il confronto tra due variabili di tipo struct direttamente tramite il nodo operatore Equal, quindi occorre utilizzare il nodo operatore **Contains**, che prende in input un array e un altro elemento restituisce un valore booleano vero se l'elemento fa parte dell'array. In questo caso, perché lo script sia logicamente corretto occorre costruire un array di un solo elemento a partire dalla variabile OldParams tramite il nodo **Make Array**, e confrontarlo con l'unico elemento dell'array di struct contenente i nuovi parametri; tuttavia, poiché si stanno confrontando due soli elementi, usare l'array con i nuovi parametri come primo input e la variabile struct con i vecchi parametri è totalmente equivalente e permette di usare meno nodi, e quindi eseguire meno operazioni. Il risultato del nodo Contains viene usato come input di un branch: se il risultato è vero, allora i parametri vecchi e quelli nuovi sono uguali, quindi ricomincia il ciclo di aggiornamento a partire dal nodo Delay iniziale; se il risultato è falso, si procede con l'aggiornamento dei parametri nella MetaSound Source. La scrittura dei parametri avviene come visto nei precedenti esempi: viene eseguito un nodo Timeline che guida l'interpolazione lineare tra i valori contenuti nella variabile OldParams e i nuovi valori scaricati da Google Sheets e la scrittura avviene tramite nodi Set Parameter. Alla fine della Timeline, viene eseguito il Delay iniziale, e ricomincia il ciclo di aggiornamento. La figura 4.13 mostra il grafo Blueprint dello script appena analizzato.

4.4 Test di performance

In questo paragrafo verranno illustrati i metodi utilizzati per misurare la performance dei metodi illustrati nel presente capitolo. Per prima cosa, si analizzano i metodi per misurare la latenza, poiché essa può avere un ruolo cruciale nella buona riuscita di un aggiornamento in tempo reale. È da specificare che, nel contesto per il quale è stato svolto il lavoro di tesi, ovvero l'ambito di un'installazione museale, avere tempi di latenza molto bassi non è un requisito fondamentale. Il metodo di misurazione verrà applicato sullo script Blueprint analizzato nel sottoparagrafo 4.2.3.

Il metodo di misurazione implementato consiste nel misurare il tempo che intercorre tra la costruzione del link di download da Google Sheets alla fine dei controlli sul successo dell'operazione, e quindi l'inizio della scrittura sui parametri. Per farlo, viene usato il nodo **Get Accurate Real Time**, che restituisce il tempo passato dall'avvio dell'applicazione, e viene fatta una differenza tra i due tempi, misurati rispettivamente prima del nodo BuildGoogleSheetDownloadLinkAndGetAsCsv e prima del nodo Branch che dà inizio alla scrituttra dei parametri. Tale differenza viene poi stampata a schermo tramite nodo **Print String**. Inoltre, viene calcolata e stampata anche la media aritmetica dei tempi misurati.

La figura 4.14 mostra le modifiche fatte al Blueprint di partenza. Per un'immagine completa del grafo Blueprint, si faccia riferimento alla figura 4.6. Il nodo Get Accurate Real Time restituisce il tempo in parte intera, misurata in secondi, e parte frazionaria, misurata in millisecondi, quindi per ottenere una misura unica viene applicato un nodo Add. Il tempo così misurato viene poi salvato in una variabile: in questo caso Time1 è il tempo misurato prima della costruzione del link di download, e il tempo Time2 è il tempo misurato prima della scrittura dei parametri. Inoltre, è stata definita una terza variabile, corrispondente a un array di tipo float e chiamata TimeArray, usata per salvare tutte le misure effettuate, nel caso si vogliano svolgere altre operazioni su esse.

Per il calcolo della differenza e della media delle differenze e per la stampa a schermo, è stata definita una funzione, utile per rendere il grafico leggibile e per poter essere riutilizzata. La figura 4.15 mostra il grafo della definizione della funzione. La funzione prende in input i riferimenti alle variabili Time1, Time2 e

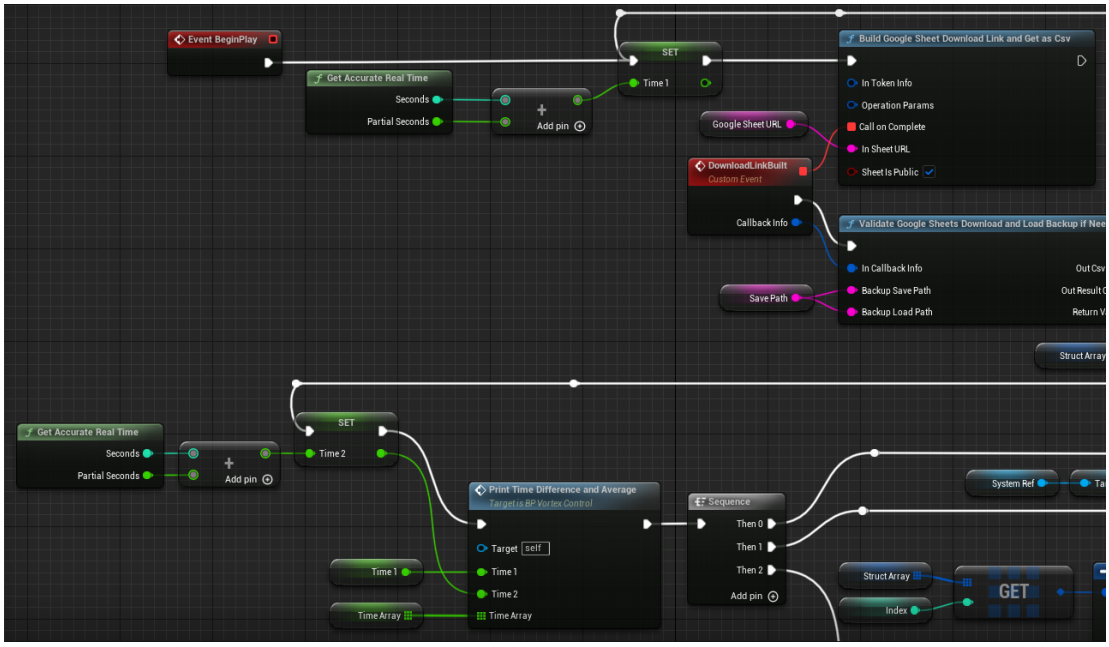


Figura 4.14: Integrazione dello script Blueprint per effettuare misurazioni di latenza

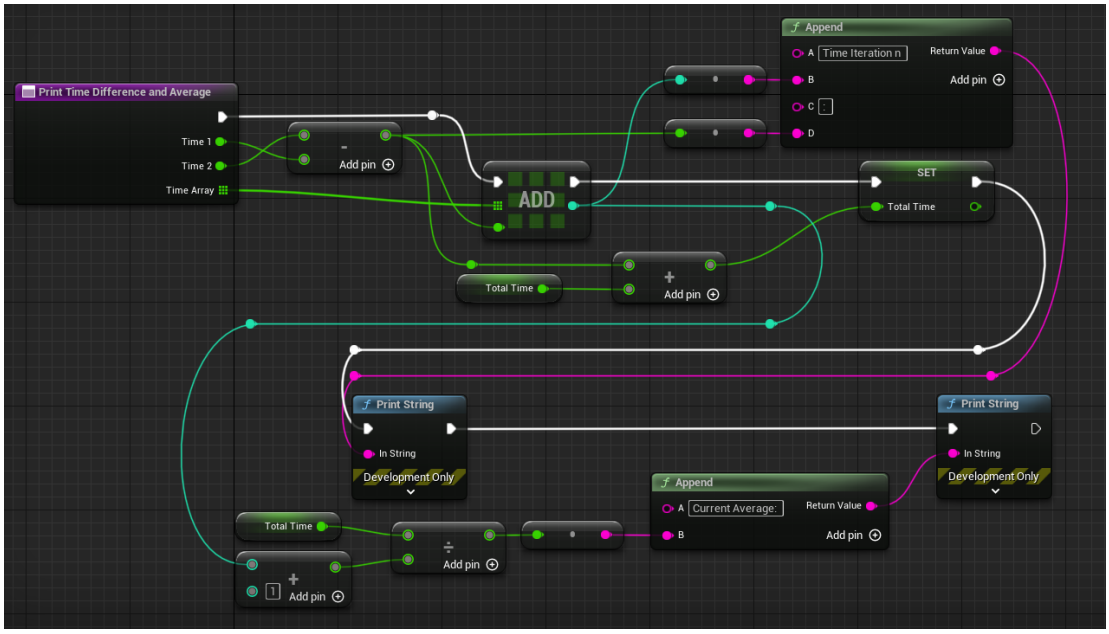


Figura 4.15: Funzione Blueprint per calcolare e stampare la latenza e la media delle latenze

TimeArray, e non restituisce output. La prima operazione è quella di differenza fra Time2 e Time1, che restituisce la latenza vera e propria. Questo valore viene sia aggiunto come elemento in TimeArray, sia aggiunto alla variabile della funzione TotalTime, che conserva il valore totale del tempo. Il valore di latenza viene poi stampato tramite il nodo PrintString, ma la stringa viene prima formattata tramite il nodo **Append**, che permette di mettere in sequenza due o più stringhe: in questo caso, viene stampato anche il numero di iterazione, corrispondente all'indice dell'elemento aggiunto in TimeArray. Poi, viene calcolata la media aritmetica semplicemente dividendo la variabile TotalTime per l'indice dell'elemento aggiunto all'array sommato a 1. Il valore della media così calcolata viene poi stampato tramite nodo Print String.

Di seguito vengono presentati i risultati ottenuti sulla tabella usata nel sottoparagrafo 4.2.3: sono state eseguite 100 iterazioni del ciclo, ed è stata misurata una media di 637 millisecondi di latenza. È da notare che la prima iterazione ha richiesto un tempo molto maggiore della media, ovvero 1272 millisecondi, mentre le altre iterazioni hanno richiesto un tempo compreso circa fra i 570 e i 640 millisecondi. È opportuno specificare che, in questo caso, la tabella è di piccole dimensioni, poiché ha solo 10 colonne e 2 righe, quindi si può supporre che il tempo misurato sia dovuto in larga parte a tutte le operazioni che non siano il download dei dati e la conversione.

Numero di elementi nella tabella	Latenza media	Latenza alla prima iterazione
20	637 ms	1272 ms
100	667 ms	695 ms
1000	649 ms	665 ms
10000	692 ms	1161 ms
100000	1181 ms	1408 ms

Tabella 4.1: Risultati delle misurazioni di latenza rapportate alle dimensioni delle tabelle

Lo stesso test è stato poi ripetuto variando arbitrariamente i valori della tabella. Nella tabella 4.1 sono presentati i risultati di queste misurazioni. È possibile notare

come la latenza media abbia delle variazioni di poche decine di millisecondi dai 20 ai 10 000 elementi, mentre nella misurazione dei 100 000 elementi il valore abbia un incremento maggiore. Per quanto riguarda la latenza alla prima iterazione, essa varia in maniera indipendente dal numero di elementi nella tabella fino ai 10 000 elementi, per poi avere un incremento significativo nella misurazione sulla tabella con 100 000 elementi.

Inoltre, durante la misurazione sulla tabella da 100 000 elementi, era possibile osservare dei cali di frame rate. Ripetendo il test, è stato possibile analizzare questi cali di performance grazie allo strumento di analisi Unreal Insights. La figura 4.16 mostra il grafico dell'evoluzione del frame rate durante il test. Inoltre, dalle statistiche presenti in Unreal Insights è stato possibile osservare come il collo di bottiglia, durante questi cali di frame rate, sia la CPU.

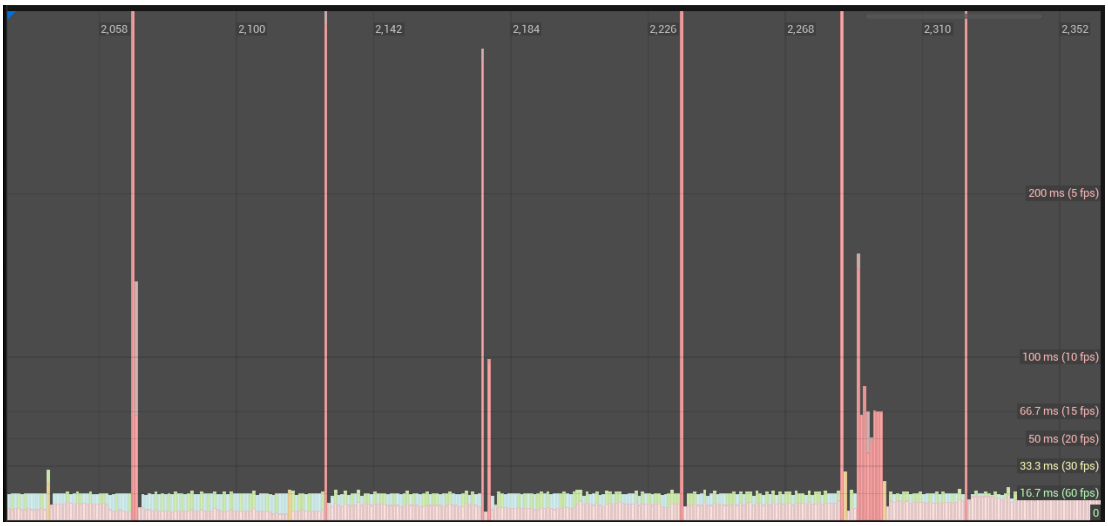


Figura 4.16: Grafico dei cali di frame rate dovuto a dimensioni elevate delle tabelle

Dalle misurazioni effettuate, è possibile concludere che il metodo descritto nella tesi ha una latenza media compresa fra i 600 e i 700 millisecondi per tabelle con un numero di elementi fino all'ordine delle decine di migliaia, mentre per tabelle con dimensioni maggiori la latenza può cominciare a crescere. Inoltre, è stato osservato che per dimensioni molto elevate, si possono osservare dei cali di performance per via della grande quantità di calcoli dovuta alle dimensioni delle tabelle. È quindi sconsigliato, dove possibile, l'uso di tabelle di dimensioni elevate per evitare di

osservare cali di performance. Tuttavia, è opportuno notare che i test sono stati effettuati su un calcolatore di livello consumer, mentre in ambito museale è possibile avere accesso ad hardware di fascia più alta, quindi è si consiglia la ripetizione degli stessi test sull'hardware su cui verrà eseguito il rendering.

Capitolo 5

Progetto Demo

In questo capitolo verrà analizzato il progetto demo realizzato per presentare il lavoro di tesi. Il fine di questo progetto è di dimostrare le potenzialità offerte dal workflow descritto nel precedente capitolo, applicandolo in un progetto con un design che prevede a monte l'utilizzo di tali metodi. È stato scelto come soggetto di tale dimostrazione una tempesta di sabbia, sia per l'aderenza ai temi trattati nel progetto Egitto Immersivo, sia per l'esistenza di un environment di partenza, ovvero quello della scena del Nilo, descritta nel sottoparagrafo 3.1.5. Questo progetto contiene, dal punto di vista dei metodi descritti nel presente lavoro di tesi, tre elementi creati per essere controllati tramite dati online:

- Un Niagara System per la creazione del Visual Effect della tempesta di sabbia.
- Una MetaSound Source per la sintesi del sound effect del vento.
- Una MetaSound Source per la sintesi di audio extradiegedico, nello specifico un *pad*.

Inoltre, viene controllato come parametro anche il colore della Directional Light che simula il sole. In tutto, i parametri usati per controllare i sistemi sopra elencati sono 10. Tuttavia, a differenza degli esempi presenti nel capitolo precedente, non vi è una corrispondenza uno a uno fra i dati nella tabella e i parametri presenti nel progetto, ma sono stati creati 4 parametri nella foglio Google Sheets, ognuno dei quali controlla contemporaneamente più parametri nel progetto. Per semplicità, da questo punto in avanti, i parametri presenti nella tabella online verranno chiamati

"dati di input", mentre i parametri che controllano il comportamento degli effetti visivi e sonori verranno chiamati "parametri dei sistemi". Questa scelta di design è stata fatta per fornire un esempio di mapping fra i dati nella tabella e i valori inseriti nei parametri dei sistemi. In questo caso, i dati di input possono avere esclusivamente valori compresi tra 0 e 1, e controllano tramite diverse funzioni di interpolazione diversi parametri contemporaneamente, per ognuno dei quali è stato definito un range specifico. Inoltre, sono stati definiti alcuni parametri che dipendono da più di un dato di input, creando così funzioni di mappatura complesse.

È opportuno specificare che, oltre agli elementi controllati da dati di input, alla scena è stato aggiunto anche un sound effect per il rumore del fiume Nilo. In questo caso, la MetaSound Source creata contiene solo un nodo Wave Player. L'asset proviene dal sito Pixabay[1], ma è stato modificato in un software DAW per poter essere riprodotto in loop senza discontinuità, diminuendone la durata. Sono state inoltre eseguite operazioni di pitch shift e di distorsione per avvicinare il suono di partenza a quello che può essere il suono del Nilo. La MetaSound Source è stata poi posizionata al centro del fiume, vicino la posizione di partenza della camera (specificata tramite un Actor di tipo **Player Start**). Per fare in modo che il suono si attenui quando la camera si allontana dal punto in cui è stato posizionato, è stato creato un asset di tipo **Sound Attenuation**, che permette di modificare le proprietà del suono a cui è applicato in base alla distanza. La forma dell'attenuazione è stata impostata su una sfera. Per poter avere un risultato corretto, si sarebbe dovuto legare la posizione dell'Actor alla spline che definisce il fiume, ma in questo progetto non è stato fatto per semplicità, ma potrebbe essere implementato in eventuali versioni future.

5.1 Effetto visivo: tempesta di sabbia in Niagara

Il Niagara System creato per la simulazione ha tre tipi di Emitter al suo interno: uno che genera dei ciottoli che rotolano sul terreno, e due che generano due tipi

[1] `https://pixabay.com/sound-effects/river-in-icelandic-nature-ambience-5-minutes-18951/`

diversi di sabbia.

Nell'Emitter che genera i ciottoli è stato usato un Mesh Renderer. La mesh del ciottolo è stata creata in Blender deformando la forma di base "Icosphere". È stato poi creato un Materiale nel quale sono state inserite le texture di Base Color, Roughness e Normal dal pacchetto "Rock 034" dal sito 3D Textures[2]. Nel Materiale è stato anche aggiunto un parametro di tipo float3 che è stato moltiplicato per il Base Color, così da variarne la tinta e renderla più omogenea con la scena. All'interno dell'Emitter sono poi stati aggiunti i moduli Spawn Rate per generare continuamente particelle, Shape Location in modalità Plane per fare in modo che nascano su un piano, Add Velocity per dare loro una velocità iniziale verso il basso, così che finiscano sul terreno nel caso vengano generate troppo in alto, e **Initial Mesh Orientation** in modalità Random per dare un orientamento casuale alle particelle alla loro nascita. Nel modulo Initialize Particle la vita è stata impostata in un range da 10 a 15 secondi e la Mesh Scale Mode su Random Non-Uniform per fare in modo che tutte le particelle generate abbiano una forma leggermente diversa. Successivamente, nella fase Particle Update sono stati inseriti i moduli Collision per fare in modo che le particelle rimangano sul terreno e non lo oltrepassino, Gravity Force per dare loro un movimento accurato fisicamente, Wind Force per fare in modo che vengano spinte lungo la direzione del vento e Drag per limitare il moto.

Successivamente sono stati creati gli Emitter che simulano la sabbia spostata dal vento. La struttura usata in questi due tipi Emitter è identica, e a variare sono solo il Materiale applicato e il valore di alcuni parametri. Inoltre, nel Niagara System sono stati aggiunti 5 Emitter uguali di ciascuno di questi due tipi. Questo è stato fatto perché, per valori troppo alti di Spawn Rate e dimensioni delle sprite troppo elevate, ci possono essere dei problemi di *clipping* durante il rendering real-time. Questo problema viene risolto duplicando gli Emitter e far generare ad ognuno di essi un numero minore di particelle.

Per le particelle di sabbia, è stato creato un Materiale da applicare alle particelle di questi due tipi di Emitter. Lo Shading Model è stato impostato su Default Lit e la Blend Mode su Translucent, così che le particelle reagiscano alla luce e possano avere un certo grado di trasparenza. Sono poi stati aggiunti un nodo Texture

[2] `https://3dtextures.me/2019/07/10/rock-034/`

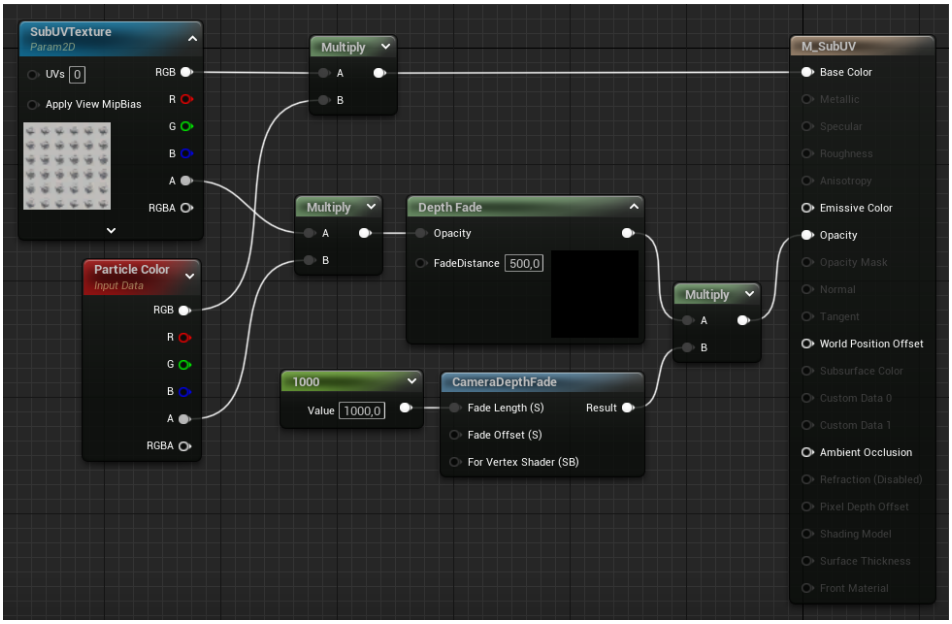


Figura 5.1: Grafo del Materiale applicato alle sprite del Niagara System della tempesta di sabbia

Sample e un nodo Particle Color, i cui rispettivi output RGB e Alpha sono stati moltiplicati. Il risultato della moltiplicazione RGB è stato collegato al pin Base Color del Main Material Node, mentre il risultato della moltiplicazione dei canali Alpha è stata prima collegata a un nodo **Depth Fade**. Il nodo Depth Fade crea un gradiente di trasparenza intorno all'intersezione fra un Materiale opaco e uno traslucido, per fare in modo che il punto di contatto venga nascosto. L'output di tale nodo è stato poi moltiplicato con quello di un nodo **CameraDepthFade**, che esegue la stessa operazione di gradiente di trasparenza, ma in base alla distanza dalla camera. Infine, il nodo Texture Sample è stato promosso a parametro per fare in modo che alle Material Instance del Materiale potessero essere applicate texture diverse. La figura 5.1 mostra il grafo descritto. In seguito, sono state create due Material Instance, in cui sono state inserite le texture "T_Smoke_A" del pacchetto "Realistic Starter VFX Pack Vol 2"[3] e "T_SmokeBall_8_8" del pacchetto "Infinity

[3] `https://www.unrealengine.com/marketplace/en-US/product/realistic-starter-vfx-pack-vol`

Blade: Effects"[4].

Figura 5.2: Visual Effect della tempesta di sabbia visto da una prospettiva esterna

Negli Emitter delle particelle di sabbia è stato usato uno Sprite Renderer, all'interno del quale sono state inserite le due Material Instance sopra descritte. Inoltre, nelle impostazioni del modulo è stata impostata una Sub Image Size sul 8.0 sia in direzione X che in direzione Y. Questa impostazione viene usata per le *flipbook texture*, ovvero texture che contengono più immagini al loro interno, ognuna delle quali rappresenta un frame di un'animazione. Per fare in modo che le immagini della *flipbook texture* si susseguano, è stato inserito il modulo **SubUV Animation**, impostato nella modalità Infinite Loop e con un Play Rate impostato su un valore casuale compreso tra 0.25 e 0.4 per fare in modo che tutte le animazioni avvengano con ritmi leggermente diversi. Nell'Emitter sono poi stati inseriti un modulo di Spawn Rate per generare continuamente particelle e un modulo di Shape Location in modalità Box per generarle all'interno di un volume di forma parallelepipeda. Nel modulo Initialize Particles è stata impostata un tempo di vita casuale compreso tra i 10 e i 15 secondi, e un colore simile al colore della sabbia, che viene applicato alle particelle grazie al nodo Particle Color nel Materiale. Nella fase di Particle

[4] `https://www.unrealengine.com/marketplace/en-US/product/infinity-blade-effects`

Update sono poi stati inseriti i moduli **Sprite Rotation Rate**, con un valore compreso fra -50 e 50 per dare una rotazione alle particelle durante il loro tempo di vita, un modulo Scale Sprite Size per far ingrandire le particelle nel tempo, simulando il comportamento reale della sabbia, un modulo Scale Color per dare un effetto di *fade in* e *fade out*, e i moduli Gravity Force, Wind Force e Drag per definire il moto delle particelle, in modo simile a quanto fatto con l'Emitter dei ciottoli. La figura 5.2 mostra il risultato finale del Niagara System.

Infine, sono stati definiti degli User Parameter per rendere possibile il controllo del comportamento del sistema tramite script Blueprint. I parametri creati sono tre, contrassegnati dal prefisso "BPP_" per distinguerli dai parametri usati per il posizionamento e la modifica del sistema nel livello:

- BPP_SpawnRate: parametro applicato ai moduli di Spawn Rate di tutti gli Emitter.
- BPP_WindSpeed: parametro applicato alla proprietà Wind Speed dei moduli Wind Force. Nel caso degli Emitter dei ciottoli il parametro viene moltiplicato per il valore 0.5 per limitare l'effetto del vento su essi.
- BPP_Opacity: parametro applicato come moltiplicatore del canale Alpha nei moduli Scale Color, presenti solo negli Emitter della sabbia.

5.2 Audio non diegetico: pad in MetaSound

Il secondo elemento controllato tramite i dati di input controlla una MetaSound Source che sintetizza audio extradiegetico. Come reference, è stato usato quello che in musica viene chiamato un *pad*, ovvero un suono a lungo sostegno prodotto da un sintetizzatore, che spesso ha movimenti molto lenti e copre una vasta gamma delle frequenze audio. Il pad viene solitamente usato per far risaltare l'armonia di un brano, dare maggiore omogeneità ai vari elementi frequenziali o per definire l'atmosfera di una scena. In questo caso, è stato scelto di usarlo come strumento per dare un corrispettivo uditivo di quello che viene mostrato in video: il pad avrà un suono più "dolce" per valori di input che definiscono una situazione atmosferica più sicura, mentre avrà suoni più "duri" quando la tempesta sarà più intensa, quindi più pericolosa.

Per prima cosa, nella MetaSound Source vengono calcolate le frequenze da dare in input agli oscillatori, i quali produrranno le onde vere e proprie. Per calcolare la frequenza, si è scelto di partire dal numero della nota nel protocollo MIDI, perciò è stato creato il primo input di tipo float, chiamato "NoteIn". L'input è collegato al nodo MIDI Note Quantizer per quantizzare il valore in ingresso e produrre una frequenza corrispondente a una nota del sistema musicale standard. A partire dal valore prodotto vengono calcolate due frequenze, corrispondenti alle frequenze riprodotte dai due oscillatori del sistema: la prima nota rimane invariata, mentre alla seconda vengono aggiunti 7 semitoni, corrispondenti a un intervallo di quinta giusta. Questo è stato fatto perché tale intervallo è consonante e arricchisce le frequenze prodotte dal primo oscillatore. In seguito, ai numeri delle note MIDI così calcolati viene aggiunta una variazione grazie a un nodo di Low Frequency Noise, che produce, con una frequenza di 0.125 Hz (ovvero ogni 8 secondi), un valore compreso tra -0.15 e 0.15 e calcola l'interpolazione tra tali valori. I valori in uscita vengono sommati ai numeri MIDI calcolati, in modo che la frequenza data in input agli oscillatori non sia mai statica, ma abbia un movimento lento. Gli output di tale operazione vengono collegati ai nodi MIDI To Frequency, che convertono i numeri MIDI in frequenze in Hertz. Questi valori sono salvati come variabili, chiamate rispettivamente "OSC1 Frequency" e "OSC2 Frequency". La figura 5.3 mostra la porzione di grafo appena analizzata.

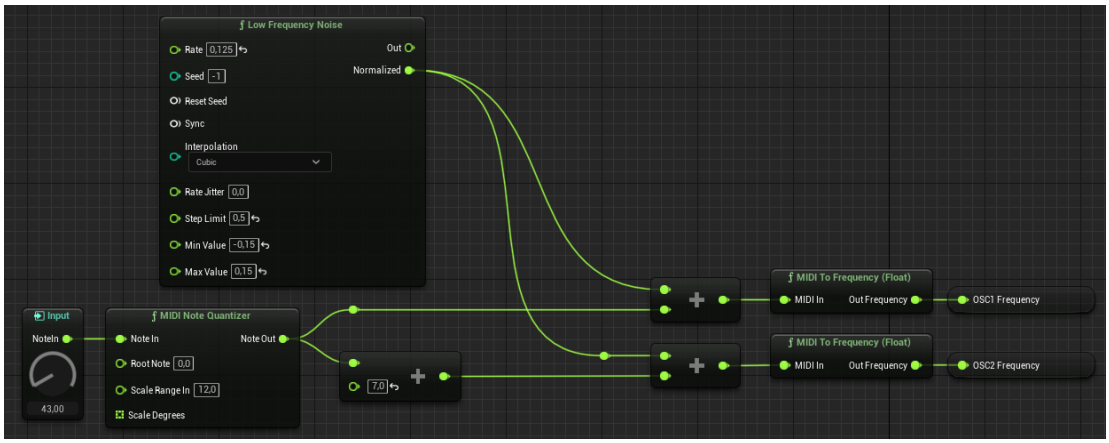


Figura 5.3: Porzione di grafo della MetaSound Source del pad per il calcolo delle frequenze

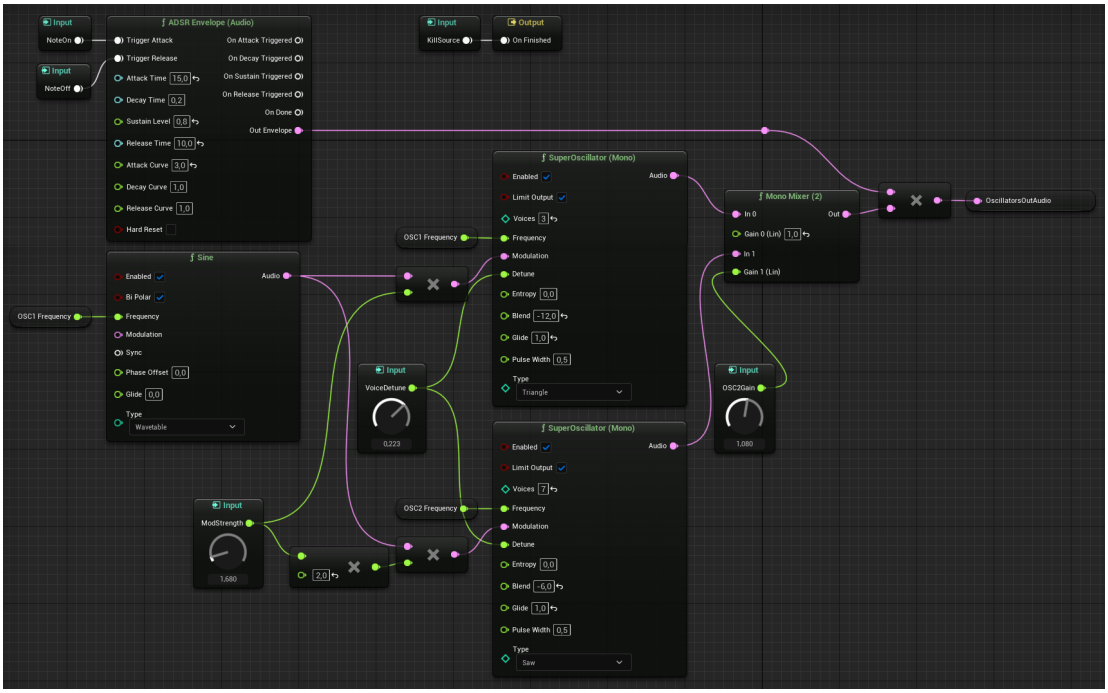


Figura 5.4: Porzione di grafo della MetaSound Source del pad contenente gli oscillatori e l'inviluppo

La fase successiva è quella di generazione delle onde a partire da dei nodi di tipo oscillatore. Come oscillatori, sono stati usati dei SuperOscillator, che producono un numero compreso tra 1 e 16 onde, a frequenze leggermente diverse, permettendo di avere un suono più ricco. Al primo oscillatore è stata assegnata come frequenza la variabile OSC1 Frequency, è stata scelta come forma l'onda triangolare ed è stato impostato 3 come numero di voci. Al secondo oscillatore è stata assegnata come frequenza la variabile OSC2 Frequency, è stata impostata come forma d'onda l'onda a dente di sega e come numero di voci 7. Per la proprietà Detune, ovvero la distanza in frequenza delle onde secondarie rispetto a quella dell'onda principale, è stato creato un input chiamato Voice Detune, assegnato a entrambi gli oscillatori. Inoltre, è stato aggiunto un terzo oscillatore, di tipo sinusoidale (nodo Sine) per operare una modulazione in frequenza sulle onde generate dai due oscillatori sopra descritti. Per quest'onda modulante, è stata usata la variabile OSC1 Frequency come frequenza. L'output di questo nodo è poi stato collegato a dei nodi di moltiplicazione prima di essere collegati ai *pin* Modulation dei due oscillatori. Il secondo termine di tali

moltiplicazioni è l'input float ModStrength, che regola l'intensità di modulazione delle onde. Nel caso del secondo oscillatore, tale input viene moltiplicato per due, per avere un effetto ancora maggiore. La modulazione in frequenza, in questo caso, viene usata per arricchire ulteriormente il suono aggiungendo frequenze, e per dare la possibilità di avere un cambio di timbro, poiché per valori alti di ModStrength il suono prodotto diventa molto ricco di armoniche, a livelli quasi di rumore, così che quando il VFX della tempesta mostra situazioni di pericolo, la modulazione in frequenza permetta di variare con continuità da un timbro più dolce a uno più minaccioso.

In seguito, gli output dei due oscillarori sopra descritti vengono collegati a un nodo Mono Mixer, che combina i due input audio in entrata, regolandone il volume. Il volume del secondo oscillatore è regolato da un input float, chiamato OSC2Gain. Infine, il risultato del mixer viene moltiplicato per l'output di un nodo ADSR Envelope, a cui sono stati collegati gli input di tipo trigger NoteOn e NoteOff, che permettono rispettivamente di attivare la fase di attacco e rilascio dell'inviluppo. È stato impostato un tempo di attacco molto lungo, che permette di far crescere molto lentamente il volume del suono generato. La figura 5.4 mostra la porzione di grafo MetaSound appena analizzata.

L'ultima fase del grafo di sintesi del *pad* riguarda il trattamento del suono generato dagli oscillatori. L'audio generato dai due oscillatori viene passato attraverso due nodi **Biquad Filter**, che effettuano un filtraggio delle frequenze. Il primo filtro è di tipo passa basso, e quindi attenua le frequenze maggiori di quella di taglio. È stato creato un input, chiamato "CutoffFrequency" per regolare la frequenza di taglio di tale filtro. A questa frequenza viene prima sommato il risultato di un LFO (ovvero un oscillatore a bassa frequenza), di forma d'onda triangolare. Il valore massimo e quello minimo di quest'onda vengono presi dividendo il valore dell'input CutoffFrequency per 5, quindi l'LFO fa oscillare il valore della frequenza di taglio di $\pm 20\%$ rispetto al valore specificato nell'input. La frequenza specificata nel nodo LFO è 0.1 Hz, quindi tale movimento della frequenza di taglio ha un periodo di 10 secondi, il che dà ulteriore movimento al suono. L'input CutoffFrequency viene moltiplicato per l'output di un nodo ADSR Envelope, che fa in modo che la frequenza di taglio del filtro passi da 0 al valore indicato nel tempo di attacco impostato nel nodo quando viene eseguito l'input trigger NoteOn. Il secondo nodo

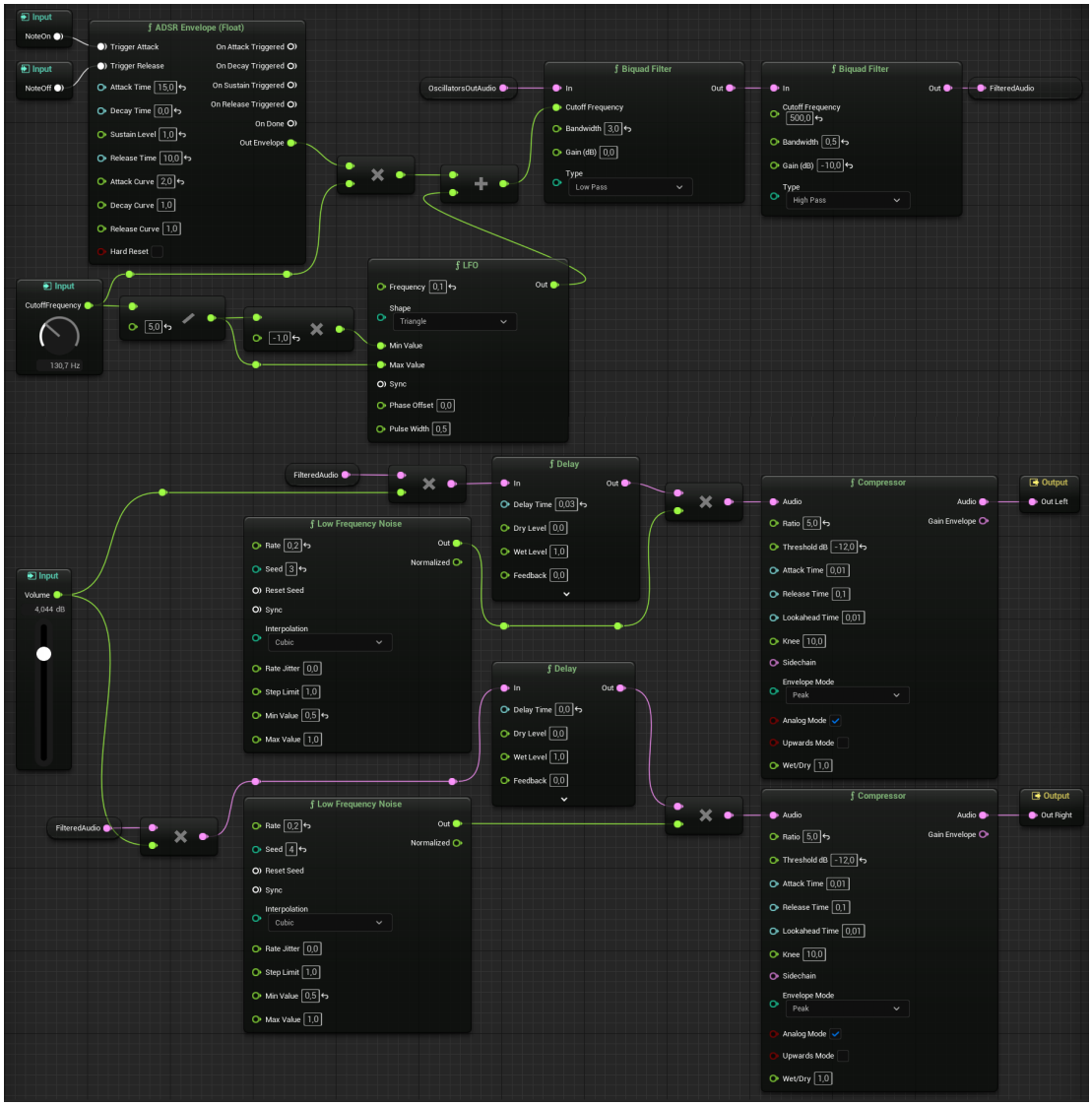


Figura 5.5: Porzione di grafo della MetaSound Source del pad per l'applicazione di effetti

Biquad Filter è invece impostato in modalità passa alto, con una frequenza di taglio di 500 Hz, e ha il ruolo di limitare le basse frequenze, che possono mascherare quelle presenti nel sound effect del vento presente nella scena.

L'output di tali filtri viene poi collegato a due nodi **Delay**, che hanno lo scopo di creare un ritardo di 30 millisecondi nel canale stereo sinistro (L) rispetto a quello destro (R), creando così un effetto di stereofonia a partire da degli oscillatori mono.

Il nodo Delay del canale destro è stato inserito nel grafico nonostante il ritardo nullo per permettere in futuro di modificare il suono senza cambiare la struttura del grafo.

I due output dei nodi Delay vengono poi moltiplicati per l'output di due nodi Low Frequency Noise, con un Rate di 0.2 e due seed diversi, e con valori compresi tra 0.5 e 1. Questo permette di creare ulteriormente differenza tra i due canali stereo, poiché avranno sempre volumi diversi.

Infine, i due segnali audio provenienti dai due nodi Delay vengono mandati su due nodi **Compressor**, che limitano la dinamica del segnale, impedendo di avere picchi di intensità troppo alti. Il risultato di questa ultima operazione viene mandato nei due output stereo. La figura 5.5 mostra la porzione di grafico appena descritta.

Per riassumere, è stato creato un pad basato su due oscillatori principali, ognuno dei quali contenente al suo interno più oscillatori di frequenza simile, che generano onde le cui frequenze formano un intervallo musicale di quinta giusta. Le frequenze mandate a tali oscillatori hanno delle variazioni casuali, con ampiezza massima di 15 centesimi di semitono. Agli oscillatori è applicata una modulazione in frequenza di livello variabile. Il suono prodotto viene filtrato con un passa alto e un passa basso, quest'ultimo con una frequenza che oscilla periodicamente intorno a un centro. Viene poi creato un effetto di stereofonia grazie a un ritardo di 30 millisecondi fra i due canali e con delle variazioni casuali indipendenti di volume. Gli input creati per il controllo di questa MetaSound Source tramite Blueprint sono cinque:

- NoteIn: regola la nota riprodotta dagli oscillatori e coincide con il numero MIDI della nota riprodotta dall'oscillatore 1.
- VoiceDetune: regola la distanza massima in frequenza delle voci secondarie degli oscillatori rispetto alla voce principale.
- ModStrength: indice di modulazione in frequenza applicato agli oscillatori.
- OSC2Gain: regola il volume dell'oscillatore 2.
- CutoffFrequency: regola la frequenza di taglio centrale applicata al filtro passa basso.

È inoltre presente un sesto input float, chiamato Volume, il cui valore non è controllato da Blueprint, ma è stato usato solo per una regolazione dei volumi in fase di design.

5.3 Audio diegetico: sound effect del vento in MetaSound

Il terzo elemento della scena controllato tramite Blueprint è la MetaSound Source che riproduce il rumore del vento. Questo elemento è stato inserito per conferire maggiore realismo alla scena, e fare in modo che ciò che viene mostrato in video abbia una corrispondenza con i suoni riprodotti. Anche in questo caso è stato adottato un approccio basato sulla sintesi piuttosto che su tracce audio preregistrate: questo consente di avere pieno controllo sull'audio riprodotto, ma c'è il rischio che il risultato finale non sia realistico. Inoltre, la sintesi può avere un costo computazionale elevato per la CPU, sottraendo risorse per altri tipi di calcolo. In questo progetto, viene usato un approccio puramente generativo a scopo dimostrativo, ma è possibile adottare un approccio ibrido, combinando suoni di sintesi con suoni campionati. Nel caso di questa MetaSound Source, è stato scelto per semplicità di avere un solo input, chiamato "WindIntensity", allo scopo di limitare il numero di parametri da gestire tramite scripting. Poiché questo input controlla diversi aspetti del suono, è stato scelto di limitarlo a un intervallo compreso tra 0 e 1, per poi modificare questo intervallo tramite operazioni matematiche nelle sue diverse applicazioni nel grafo.

Il punto di partenza della generazione del rumore del vento è un nodo Noise, impostato sulla modalità Pink Noise, un rumore in cui le componenti a bassa frequenza hanno potenza maggiore di quelle ad alta frequenza. Ad esso è collegato un nodo **One-Pole High Pass Filter**, ovvero un filtro passa alto, la cui frequenza è impostata su 100 Hz, che ha lo scopo di tagliare le frequenze molto basse. L'audio in uscita viene poi mandato in un Biquad Filter, in modalità Parametric EQ, che ha lo scopo di simulare l'effetto di risonanza sul filtro precedente. Il risultato è poi collegato a un terzo e ultimo nodo filtro, di tipo **Ladder Filter**, che è un filtro passa basso che permette di dare risonanza alla frequenza di taglio e alle

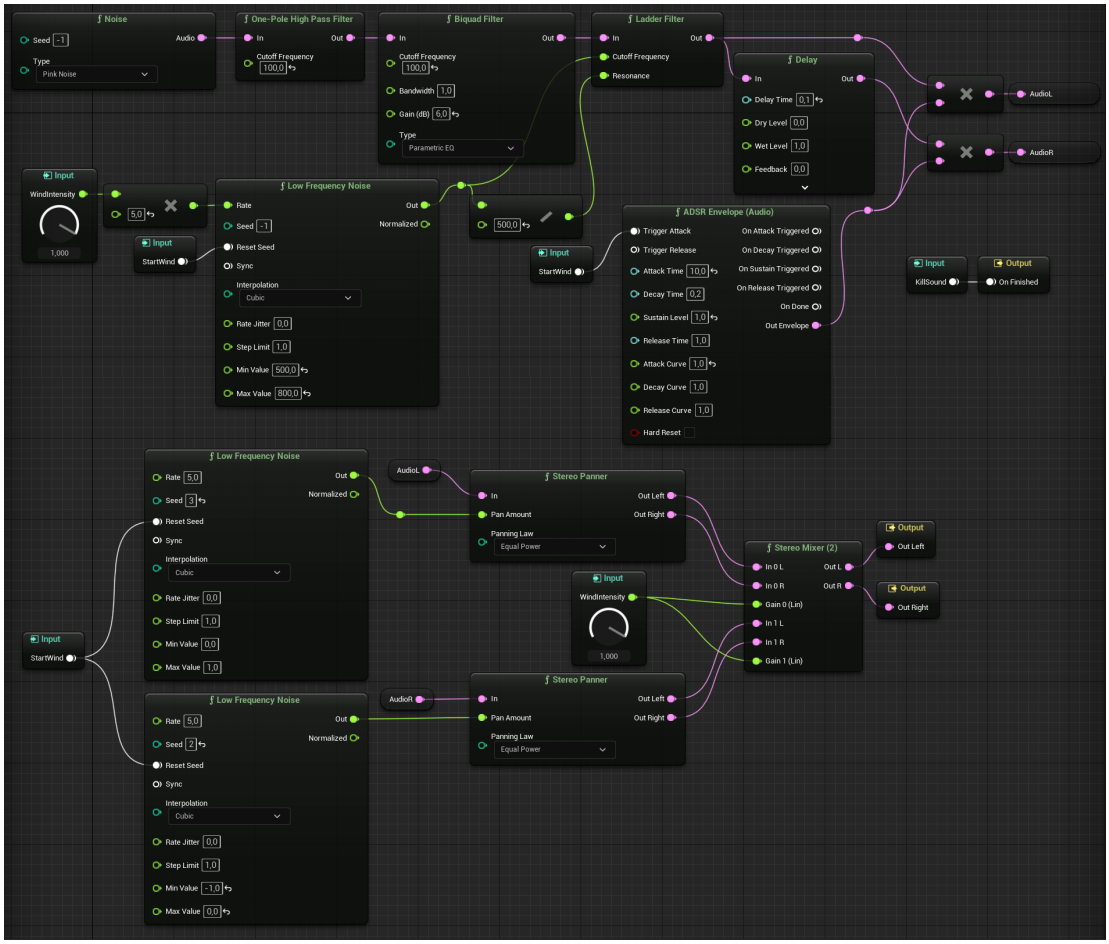


Figura 5.6: Grafo MetaSound per la generazione dell'effetto sonoro del vento

sue armoniche. Entrambi gli input di questo filtro, ovvero la frequenza di taglio e la risonanza, vengono generati da un nodo Low Frequency Noise, il cui valore di frequenza (Rate) corrisponde al valore dall'input WindIntensity moltiplicato per 5. Il nodo genera valori compresi tra 500 e 800, che vengono usati come valori di frequenza di taglio del Ladder Filter. Gli stessi valori vengono anche divisi per 500 e usati come input del parametro Resonance del filtro. Questo fa sì che a frequenze di taglio più alte corrispondano anche risonanze di potenza maggiore.

In seguito, viene implementata la stessa tecnica usata nel sottoparagrafo precedente per ottenere un suono stereo, ovvero tramite un nodo Delay viene applicato al canale audio R un ritardo di 100 millisecondi. Il risultato viene poi moltiplicato per l'output di un canale ADSR Envelope, che permette di dare inizio allla riproduzione

del suono tramite l'input trigger "StartWind".

I due audio risultanti vengono poi collegati a due nodi Stereo Panner, che permettono di mandare un audio mono nei due canali stereo con una proporzione stabilita dall'input Pan Amount. Quest'ultimo è definito da due nodi Low Frequency Noise, che hanno range tali da spostare la traccia L da completamente a sinistra al centro e la traccia R da completamente a destra al centro. Così facendo, la potenza relativa fra le due tracce cambia continuamente, dando dinamicità al sound effect. Le due tracce così ottenute vengono combinate tramite un nodo **Stereo Mixer**, analogo stereo del Mono Mixer. Il volume dato a entrambe le tracce viene definito a partire dall'input WindIntensity. Le due tracce L e R vengono infine mandate nei rispettivi output audio.

5.4 Design della mappatura fra dati di input e parametri

In questo paragrafo verranno analizzati i metodi usati per collegare il valore dei dati di input presenti online col valore dei parametri che controllano i sistemi descritti nei paragrafi precedenti. Per questo progetto demo, si è scelto di usare un numero di dati di input inferiore a quello dei parametri, per simulare una situazione in cui all'utente finale viene parzialmente nascosta la complessità del sistema tramite l'aggiunta di un livello di astrazione, che permette di interfacciarsi con un numero limitato di controlli, che possano risultare familiari anche ad utenti senza competenze tecniche.

Per prima cosa, occorre scegliere il numero e il nome di dati di input tramite cui controllare globalmente la scena. Sono stati individuati quattro parametri adatti per questo scopo. Per comodità, è stato scelto di usare il range da 0 a 1 per tutti e quattro tali dati, così da rendere le operazioni di mappatura più facili. I quattro dati di input individuati sono elencati di seguito:

- Storm Intensity: rappresenta la quantità di sabbia alzata a causa della tempesta.

- Wind Speed: rappresenta la velocità delle raffiche di vento.

- Visibility: controlla la visibilità all'interno della tempesta, e quindi l'opacità delle sprite di sabbia generate dal Niagara System.

- Danger Level: rappresenta il grado di pericolo che l'utente deve percepire dalla scena.

In aggiunta ai parametri presenti nei sistemi descritti nei paragrafi precedenti, è stato aggiunto un ulteriore parametro, chiamato SunColor, che regola il colore assegnato alla Directional Light presente nella scena. Nel caso di valori elevati di Storm Intensity e Danger Level, il colore passerà da bianco a arancione, per aumentare la drammaticità della scena.

La fase successiva è stata quella di individuare le relazioni tra i dati di input e i parametri dei sistemi presenti nella scena. Il tipo di relazione più semplice è l'interpolazione lineare, per cui il dato di input rappresenta l'input Alpha che controlla il peso assegnato ai due estremi dell'interpolazione. Tuttavia, per alcuni tipi di dato controllati dai parametri, l'interpolazione lineare non è il tipo di mappatura più adatta. Ad esempio, quando si controllano dei parametri associati alle frequenze, occorre usare un'interpolazione di tipo esponenziale, mentre per quanto riguarda il volume di un suono occorre verificare se la grandezza è espressa in lineare o in decibel, e quindi decidere se usare un'interpolazione lineare o logaritmica. In generale, tipi di interpolazione diversa da quella lineare possono essere utili per personalizzare l'effetto finale.

Una volta scelte le funzioni di interpolazione, sono stati individuati il valore minimo e il valore massimo entro i quali limitare il valore di ciascuno dei parametri. Questo processo può risultare particolarmente complesso e richiedere numerose iterazioni a causa dell'alto numero di parametri.

Infine, ad ogni parametro sono stati assegnati uno o più dati di input che controllano l'interpolazione fra valore minimo e valore massimo. Nella maggior parte dei casi la relazione è diretta, ovvero all'aumentare del valore del dato di input aumenta anche il valore del parametro. In alcuni casi, invece, la relazione è inversa, ovvero all'aumentare del valore del dato di input il valore del parametro diminuisce. Nel caso di relazione diretta, l'interpolazione viene quindi fatta usando assegnando all'input A dell'interpolazione il valore minimo del range e all'input B il valore massimo; in caso di relazione inversa, come input A si usa il valore

Nome del Parametro	Valore minimo	Valore massimo	Dati di input di controllo	Tipo di interpolazione
SpawnRate	0	30	Storm Intensity	Lineare
WindSpeed	2000	5000	Wind Speed	Lineare
Opacity	0.1	1.0	Visibility (I)	Lineare
OSC2Gain	0.2	2	Storm Intensity (I)	Lineare
ModStrength	0	30	Storm Intensity, Danger Level	Lineare
VoiceDetune	0.05	0.3	Storm Intensity, Danger Level	Lineare
Cutoff Frequency	100	450	Storm Intensity, Wind Speed	Expo In
NoteIn	43	46	Wind Speed	Lineare (con arrotondamento)
Wind Sound Intensity	0.02	1	Wind Speed	Circular Out
SunColor	FFFFFFFF	FF2B00FF	Storm Intensity, Danger Level	Lineare (Linear Color)

Tabella 5.1: Mappatura fra dati di input e parametri. Le relazioni inverse sono contrassegnate dalla lettera "I".

massimo e come input B si usa il valore minimo. Inoltre, nel caso più dati di input controllino lo stesso parametro, si calcola la media aritmetica dei valori dei dati di input e la si usa per controllare l'interpolazione. La mappatura eseguita fra i dati di input e parametri del progetto demo è illustrata nel dettaglio nella tabella 5.1. Nel caso del parametro NoteIn, dopo aver eseguito l'interpolazione, viene eseguito

un arrotondamento a intero, poiché solo i numeri interi restituiscono delle frequenze corrispondenti a delle note, mentre valori floating point non interi restituiscono delle frequenze intermedie fra due note. Per quanto riguarda il parametro SunColor, è stata scelta un'interpolazione sui valori RGB piuttosto che HSV perché ha minore costo computazionale e perché in questo caso non ci sono variazioni nella tinta, ma solo nella saturazione, il che crea meno problemi nel caso di interpolazione RGB.

5.5 Blueprint per il controllo di sistemi audio e video

L'ultimo passo per la realizzazione del progetto demo è stata la creazione dello script Blueprint per l'inizializzazione dei sistemi, il download dei dati di input e l'aggiornamento dei parametri. L'approccio è sostanzialmente identico a quello visto nei sottoparagrafi 4.2.3 e 4.3.2, ovvero l'aggiornamento del sistema avviene in tempo reale, per cui i dati vengono scaricati ad ogni iterazione del loop. La presenza di sistemi sia visivi che uditivi ha reso necessario l'uso di interpolazione al fine di non creare discontinuità, seguendo l'approccio descritto nel sottoparagrafo 4.2.4. Lo script realizzato presenta inoltre delle funzionalità aggiuntive rispetto a quelli già analizzati, le quali saranno analizzate nel corso di questo paragrafo.

5.5.1 Ottimizzazione della scena tramite operazione di attaching

La prima funzionalità aggiuntiva integrata nello script Blueprint è l'operazione di *parenting* del Niagara System alla camera. In questo modo, la tempesta di sabbia "seguirà" la camera dell'utente nel caso in cui la sposti, dando così l'illusione che le particelle siano presenti in tutta la mappa. Questo approccio permette quindi di ridurre il numero di particelle da renderizzare, riducendo quindi le risorse usate per il rendering. Per eseguire questa operazione, è stato usato il nodo **Attach Actor to Actor**, collegando il riferimento del sistema al pin Target e l'output del nodo **Get Player Pawn** al pin Parent Actor. Il nodo Get Player Pawn restituisce il riferimento al Pawn controllato dal giocatore, che nella Game Mode di default

coincide con la camera. Sono state impostate come Location Rule Snap to Target, come Rotation Rule Keep World e come Scale Rule Keep Relative.

5.5.2 Applicazione di riverberi a convoluzione

La seconda funzionalità aggiuntiva riguarda l'aggiunta di effetti ai componenti audio della classe Blueprint. In questo caso, è stato aggiunto un effetto di riverbero a convoluzione all'audio prodotto dal pad. L'aggiunta di un effetto di riverbero a uno strumento pad è una pratica diffusa, che permette di dare un maggiore "senso di grandezza" al suono. Un riverbero a convoluzione permette di replicare il riverbero di un ambiente a partire dalla registrazione della sua risposta all'impulso. In Unreal Engine, è possibile creare delle risposte all'impulso a partire da un qualsiasi asset audio. Per questo progetto, sono state create le risposte all'impulso da audio provenienti dal dataset C4DM RIR[7]. Per applicare un effetto a un componente audio, occorre applicare ad esso una **Source Effect Chain**, ovvero un asset che contiene un array di effetti che verranno applicati al componente audio. Ogni elemento dell'array sarà un asset di tipo **Source Effect Preset**, che definisce il tipo di effetto e i suoi parametri. Nello specifico, nel caso di un riverbero a convoluzione l'asset è della classe **SourceEffectConvolutionReverbPreset**, all'interno del quale è possibile inserire l'asset di risposta all'impulso. Inoltre, in esso è possibile regolare come parametri i volumi dei segnali *wet* e *dry*, ovvero rispettivamente il segnale risultante dalla convoluzione e il segnale originale.

Un uso meno convenzionale del riverbero a convoluzione è la creazione di risposte all'impulso a partire da effetti digitali invece che da registrazioni di ambienti reali. In questo caso, basta creare una traccia di rumore bianco e tagliarla in modo che sia approssimabile a un impulso e applicarvi l'effetto da campionare in una workstation di audio digitale (DAW). Nel caso del progetto demo è stato campionato un effetto di *shimmer*, un effetto simile al riverbero ma con la creazione di armoniche di frequenza maggiore, che permettono di estendere la banda di frequenze del suono a cui è applicato. La traccia è poi stata tagliata e importata in Unreal Engine, tramite il quale è stato creato un asset di risposta all'impulso, applicato nell'effetto di riverbero a convoluzione.

5.5.3 Funzione di mappatura tra dati di input e parametri

La mappatura vista nel paragrafo 5.4 è abbastanza complessa e, dal punto di vista dello scripting Blueprint, richiede l'uso di molti nodi, che potrebbero rendere difficile la leggibilità del grafico. Inoltre, come si vedrà nel prossimo sottoparagrafo, il calcolo dei valori dei parametri a partire dai dati di input viene fatto sia in fase di inizializzazione che in fase di aggiornamento. Per questi motivi, è stato scelto di creare una funzione Blueprint, così da rendere il grafico più leggibile e la manutenzione dello script più semplice.

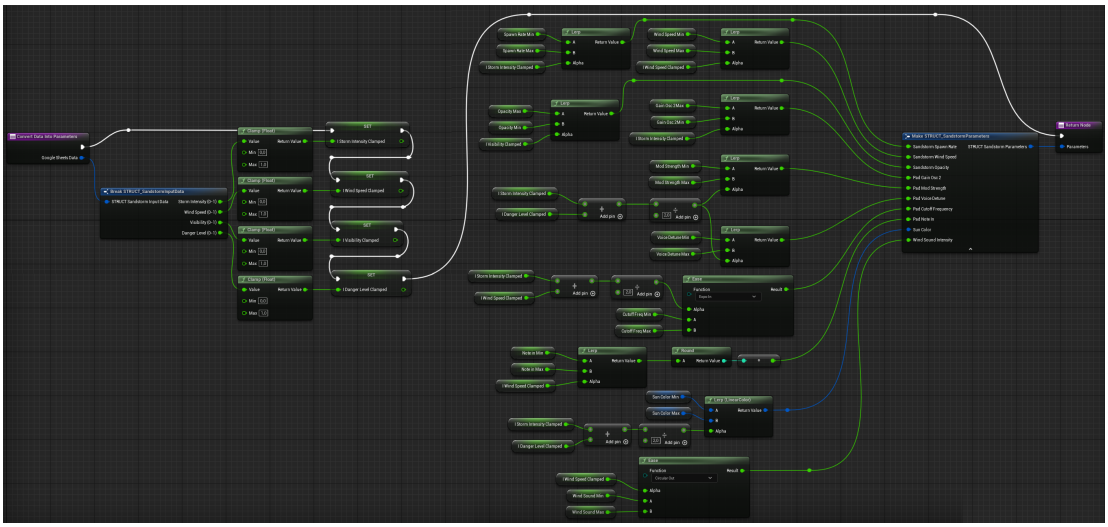


Figura 5.7: Grafo Blueprint della funzione Blueprint per la mappatura dei dati di input in parametri da scrivere nei sistemi del progetto demo.

La funzione creata, chiamata ConvertDataIntoParameters, prende in input i dati provenienti da Google Sheets e calcola i valori dei parametri da applicare nei diversi sistemi presenti nella scena. Per comodità, sono state create due struct, chiamate "STRUCT_SandstormInputData" e "STRUCT_SandstormParameters", che contengono le variabili corrispondenti rispettivamente ai dati di input e ai parametri. Nella funzione sono quindi stati impostati come input una variabile di tipo STRUCT_SandstormInputData chiamata "GoogleSheetsData" e come output una variabile di tipo STRUCT_SandstormParameters chiamata "Parameters". All'interno della funzione, come prima cosa viene applicato un nodo di Break all'input, che permette di trattare le variabili all'interno della struct separatamente.

A ciascuna di tali variabili è poi applicato un nodo **Clamp**, tramite il quale si limita il valore di input tra 0 e 1. I valori risultanti vengono poi salvati come variabili locali. Per ognuno dei parametri, sono state create due variabili locali, che contengono rispettivamente il valore minimo e il valore massimo applicabile al parametro. I valori inseriti in queste variabili sono visualizzabili nella tabella 5.1.

In seguito, per ognuno dei parametri viene eseguita un'operazione di interpolazione. Nel caso di interpolazioni lineari viene usato un nodo Lerp, nel caso di interpolazioni non lineari viene usato un nodo **Ease**, nel quale è specificato il tipo di funzione utilizzata. Nel caso di relazioni dirette viene usato come input A il valore minimo del parametro e come input B il valore massimo; nel caso di relazioni inverse viene usato come input A il valore massimo del parametro e come input B il valore minimo. Nel caso di parametri che dipendono da più dati di input, viene usato un nodo Add per sommarli e un nodo Divide per calcolare la media. I valori dei parametri risultanti vengono infine collegati a un nodo Make che permette di definire i valori della struct di output. La figura 5.7 mostra il grafo Blueprint della funzione analizzata.

5.5.4 Script Blueprint per il controllo di sistemi audio e video

La fase finale di realizzazione del progetto demo è stato lo scripting vero e proprio nella classe Blueprint, che ha lo scopo di collegare in modo corretto i dati di input presenti nel foglio di calcolo Google Sheets con i parametri dei diversi sistemi presenti nella scena. In modo analogo allo script analizzato nel sottoparagrafo 4.3.2, lo script creato del progetto demo può essere diviso in due parti principali, ovvero la fase di inizializzazione e quella di aggiornamento.

Nella fase di inizializzazione, oltre a eseguire una prima lettura dei dati e una scrittura dei parametri nei sistemi presenti nella scena, si esegue l'operazione di *attaching* del Niagara System al Player Pawn e imposta i parametri in modo da non avere discontinuità nelle fasi iniziali dell'esecuzione. Inoltre, in questa fase vengono eseguiti i parametri trigger delle MetaSound Source per iniziare la riproduzione dell'audio.

Nella fase di aggiornamento, viene eseguita la lettura dei dati di input, poi il nuovo set di dati viene confrontato con quello precedente, e se essi sono diversi, viene eseguita la conversione dei dati di input in parametri dei sistemi, i quali saranno scritti seguendo il metodo dell'interpolazione visto nel sottoparagrafo 4.2.4.

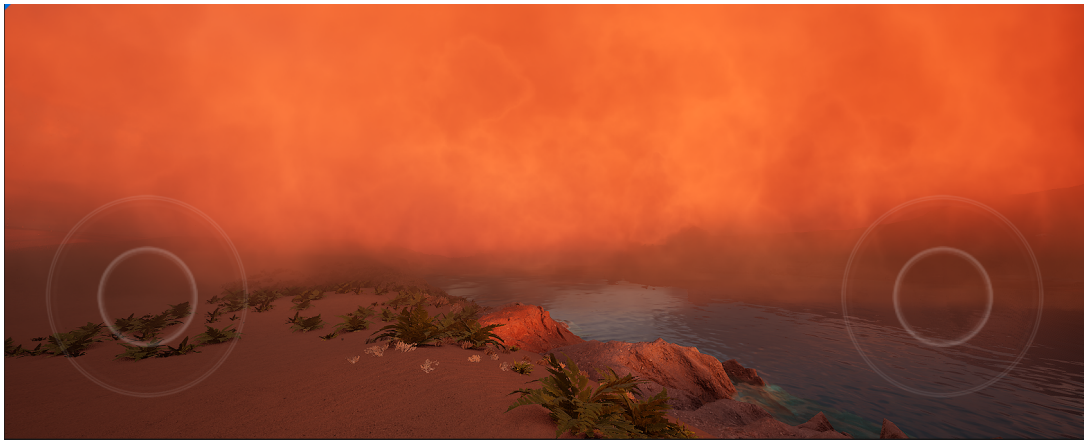
Di seguito verrà analizzato in dettaglio lo script realizzato. Al nodo Event Begin Play è collegato il nodo Set Float Parameter con riferimento al Niagara System. Tramite questo nodo si imposta il valore del parametro "BPP_SpawnRate" a 0. Questo fa sì che la condizione di inizio dell'esecuzione sia l'assenza della tempesta di sabbia, il che rende la scena più naturale. In seguito, viene eseguito il nodo Attach Actor to Actor, che imposta il Player Pawn (ovvero la camera controllata dal giocatore) come genitore del Niagara System. In seguito, vengono eseguiti i nodi del plugin Runtime DataTable per eseguire il download dei dati da Google Sheets e convertirli in formato CSV. Viene poi usato un nodo Branch, che permette di rieseguire tali nodi nel caso in cui le operazioni non siano andate a buon fine. Nel caso le operazioni abbiano avuto successo, si esegue la conversione dei dati di input in parametri, tramite la funzione ConvertDataIntoParameters, descritta nel sottoparagrafo precedente. In seguito i parametri vengono scritti nei rispettivi sistemi tramite nodi Set Parameter e si eseguono i parametri trigger delle MetaSound Source del pad e del vento.

La fase di aggiornamento inizia con un nodo Delay, che serve a separare temporalmente le esecuzioni del loop. Le struct contenenti i dati di input e i valori corrispondenti dei parametri vengono salvate in due variabili, chiamate rispettivamente "OldData" e "OldParams". In seguito, vengono nuovamente eseguiti i nodi per la lettura dei dati da foglio di calcolo Google Sheets. A questo punto viene eseguito un confronto fra i dati di input letti nell'esecuzione corrente e quelli salvati nella variabile tramite il nodo Contains: se il contenuto delle due struct è uguale, allora si ripete il loop, se è diverso, allora si procede con l'aggiornamento. Si convertono i dati di input in parametri tramite la funzione ConvertDataIntoParameters, e si salva il nuovo set di parametri in una variabile chiamata "CurrentParams". Infine, si esegue l'aggiornamento vero e proprio: un nodo Timeline contenente una rampa con ease in e ease out dalla durata di 5 secondi controlla il parametro Alpha dei nodi Lerp. Le interpolazioni vengono eseguite fra i dati presenti nelle variabili OldParams e CurrentParams. e i dati vengono scritti tramite nodi Set

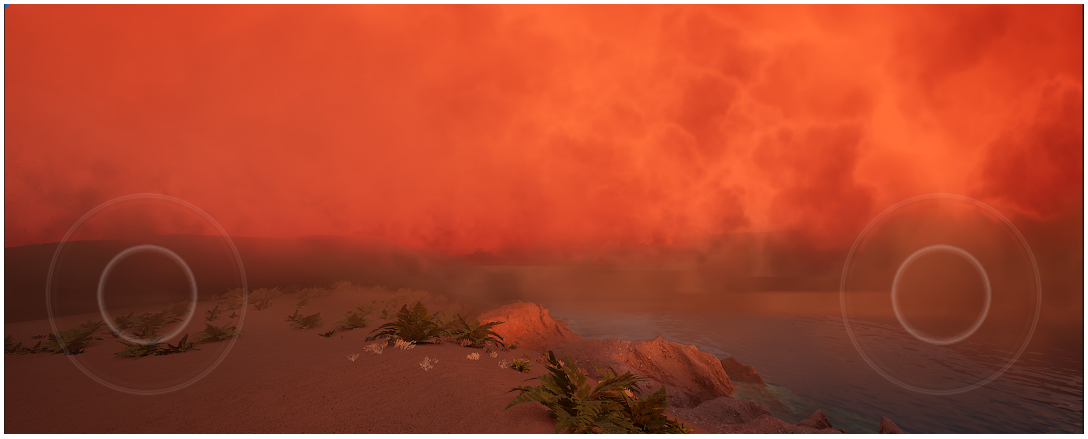
Parameter. Una volta terminata la riproduzione della Timeline, viene rieseguito il loop di aggiornamento.

Infine, l'Actor Blueprint è stato posizionato nel mondo. La figura 5.8 mostra tre stati del sistema creato: uno di quiete, uno con valori intermedi, e uno con tutti i parametri al massimo.

(a) Storm Intensity: 0, Wind Speed: 0, Visibility: 0, Danger Level: 0



(b) Storm Intensity: 0.5, Wind Speed: 0.7, Visibility: 0.7, Danger Level: 0.6



(c) Storm Intensity: 1, Wind Speed: 1, Visibility: 0, Danger Level: 1

Figura 5.8: Schermate del progetto demo con tre diversi set di dati di input

Capitolo 6

Conclusioni

In questa tesi sono stati illustrati i metodi per creare dei sistemi audio e video parametrici che possano essere controllati da dati online tramite script Blueprint in Unreal Engine 5. È stato elaborato un workflow che prevede, in ordine, la creazione di tali sistemi, la definizione dei parametri, la definizione della struttura dei dati online, la creazione di tabelle a partire da asset di tipo DataTable e lo scripting Blueprint per il controllo. Sono stati presi in considerazione due possibili scenari, ovvero sistemi reattivi i cui parametri vengono modificati in tempo reale grazie a una lettura continua o sistemi la cui evoluzione è regolata da dati definiti in modo sequenziale. Per entrambi questi casi sono stati forniti degli esempi di script per il controllo. Inoltre, sono state elaborate alcune buone pratiche da seguire per creare sistemi simili, come l'uso di interpolazioni per evitare di creare discontinuità, l'uso di Timeline per avere un controllo preciso sull'evoluzione temporale del sistema o l'uso di funzioni di interpolazione non lineari per alcuni tipi di parametri. È stato presentato e analizzato un progetto demo, con lo scopo di applicare i metodi elaborati su un progetto più complesso, che comprendesse sia sistemi di Visual Effects, sia sistemi audio. Sono state elaborate anche delle soluzioni di design che permettono di nascondere parte della complessità all'utente finale, tramite una corrispondenza non univoca tra dati online e parametri.

Inoltre, sono stati svolti dei test su alcuni degli script creati durante il lavoro di tesi, con un focus sulle misurazioni di latenza e dimensioni delle tabelle. I risultati mostrano tempi di latenza dovuti al download e conversione dei dati di

circa 600 ms, che rappresenta un tempo trascurabile per l'applicazione in ambito di un'installazione museale. Per quanto riguarda la quantità di dati, si osserva come le prestazioni rimangano sostanzialmente invariate per tabelle con un numero di elementi inferiore ai 10000, mentre tabelle di dimensioni maggiori causano cali di frame rate a causa del sovraccarico delle operazioni su CPU, oltre a un aumento della latenza.

Dal punto di vista dello scripting, si consiglia che gli sviluppi futuri si concentrino sull'ottimizzazione delle soluzioni illustrate nella tesi, con lo scopo di ridurre i tempi di latenza e limitare gli effetti di tabelle di grandi dimensioni. Potrebbe essere utile sperimentare altri metodi di memorizzazione e lettura dei dati online, ad esempio testando l'uso di basi di dati di diverso tipo. Una delle criticità dei metodi illustrati nell'elaborato è la lettura continua del contenuto delle tabelle, per cui un possibile sviluppo sarebbe quello di creare script reattivi che eseguano la lettura solo in seguito alla modifica dei dati.

Dal punto di vista del design, si consiglia che gli sviluppi futuri si concentrino su metodi di mapping fra dati online e parametri e su metodi più astratti di visualizzazione dei dati. Altri sviluppi possibili riguardano l'uso dei metodi descritti in ambiti diversi da quello museale o l'integrazione di una componente interattiva alle esperienze realizzate con il workflow presentato.

Bibliografia

[1] Barbara McGillivray, Paola Marongiu, Nilo Pedrazzini, Marton Ribary, Mandy Wigdorowitz e Eleonora Zordan. «Deep Impact: a study on the impact of data papers and datasets in the humanities and social sciences». In: *Publications* 10.4 (2022), p. 39 (cit. a p. 1).

[2] Brent Burley e Walt Disney Animation Studios. «Physically-Based Shading at Disney». In: *Acm Siggraph*. Vol. 2012. vol. 2012. 2012, pp. 1–7 (cit. a p. 11).

[3] Yoshiharu Gotanda, Adam Martinez e Ben Snow. «Physically Based Shading Models in Film and Game Production: Practical Implementation at tri-Ace». In: *SIGGRAPH 2010* (2010) (cit. a p. 11).

[4] Brian Karis e Epic Games. «Real Shading in Unreal Engine 4». In: *Proc. Physically Based Shading Theory Practice* 4.3 (2013), p. 1 (cit. a p. 13).

[5] Robert Bridson, Jim Houriham e Marcus Nordenstam. «Curl-Noise for Procedural Fluid Flow». In: *ACM Transactions on Graphics (ToG)* 26.3 (2007), 46–es (cit. a p. 30).

[6] Simon Godsill, Peter Rayner e Olivier Cappé. *Digital audio restoration*. Springer, 2002 (cit. a p. 91).

[7] Rebecca Stewart e Mark Sandler. «Database of omnidirectional and B-format room impulse responses». In: *2010 IEEE International Conference on Acoustics, Speech and Signal Processing*. 2010, pp. 165–168. DOI: 10.1109/ICASSP.2010.5496083 (cit. a p. 120).