ALMA MATER STUDIORUM · UNIVERSITÀ DI BOLOGNA
CAMPUS DI CESENA

SCUOLA DI SCIENZE

Corso di laurea triennale in Ingegneria e Scienze Informatiche

Progettazione ed implementazione di Shaders Avanzati per rendering realistici in Unity 3D

Tesi di Laurea in Computer Graphics

Relatrice:
Prof.ssa
Damiana Lazzaro

Presentata da:
Marco Amadei

Anno Accademico 2022-2023

Introduzione

L'Informatica è affascinante per diverse ragioni, ma alcune persone si avvicinano a questo mondo in particolare con il desiderio di imparare a realizzare videogiochi o applicazioni con grafica mozzafiato, ed il percorso solitamente inizia con l'apprendimento dei linguaggi di programmazione più tipici, i quali tuttavia solitamente si limitano alla gestione delle istruzioni per la CPU, ma bisogna invece ricordare che la vera potenza di calcolo per la realizzazione di grafica 2D e 3D oggigiorno risiede nella GPU.
L'obiettivo di questo progetto di tesi è mostrare un esempio di approccio allo sviluppo di effetti per grafica digitale utilizzando come base il motore grafico Unity 3D, e come mezzo la programmazione degli *Shaders*.
La scelta del motore (ed in generale di utilizzare un motore grafico) è in realtà una scelta puramente di comodità in quanto questo software permette di inizializzare un progetto e fornisce la possibilità di dedicarsi quasi immediatamente alla programmazione degli *Shader* ed alla gestione della logica dell'applicazione, senza doversi preoccupare di vari aspetti, come la gestione delle finestre e della memoria. Il motore grafico inoltre fornisce l'accesso ad una serie di tool e di esempi, i quali però, in questo progetto di tesi, verranno utilizzati il minimo indispensabile per mostrare la realizzazione di un progetto nel modo più completo possibile.

La realizzazione di grafica al computer può mirare a vari obiettivi, ad esempio la realizzazione di ambienti astratti, o molto colorati, o al contrario, che utilizzano il minor numero possibile di colori. Possono altrimenti mirare a realizzare grafiche in cosiddetta pixel art, o al contrario cercare di imitare la realtà con una grafica cosiddetta fotorealistica. Ognuno di questi stili comporta delle sfide uniche e richiede approcci molto diversi.
In questo progetto ci si focalizzerà sul tentativo di replicare una grafica fotorealistica e dunque che imita la realtà. Nella realizzazione di questo progetto bisogna mettere in conto che la realizzazione di elementi di una scena 3D, il terreno, le strade, gli edifici, il cielo, gli effetti di post produzione e così via richiedono algoritmi la cui complessità aumenta esponenzialmente con l'aumentare della qualità visiva. Quello che verrà mostrato è dunque un compromesso tra una grafica visivamente efficace, ma realizzabile in tempi ragionevoli. Nelle software house di grandi dimensioni, si cerca in realtà di non reinventare la ruota, e di utilizzare sistemi già collaudati, creati da team esperti e che hanno sviluppato nel dettaglio ognuno dei singoli aspetti citati. Spesso questi sistemi sono già incorporati nei motori grafici, tuttavia conoscerne il funzionamento di base è necessario per poter modificare tali sistemi a proprio piacimento, o eventualmente quando strettamente necessario, implementare il sistema grafico di cui si ne-

cessita.
Per mostrare tecnologie grafiche fotorealistiche si è scelta una città Italiana, nella quale si potrà percorrere un cammino che conduce dall'esterno delle mura medievali fino al centro storico, per poi procedere e terminare il tragitto attraverso ad un ponte che collega il cuore della città al borgo. Nel tragitto si potrà interagire con i monumenti più rilevanti per leggerne una breve descrizione. La Città presa come esempio è Faenza, la quale presenta un esempio adatto date le sue ridotte dimensioni ed il suo centro storico piuttosto ben preservato nel corso dei secoli. La recente urbanizzazione ha allargato il perimetro cittadino ben fuori le mura medievali, dunque ci si ispira ad una versione di Faenza non recente e più contenuta, ma nemmeno appartenente ad un periodo storico ben definito, così da poter includere monumenti ed edifici appartenenti ad epoche diverse e che sono tutt'oggi ancora simboli della città.

Si premette inoltre che all'interno del progetto discusso in questo documento, verrà limitato al massimo l'uso di *Shader* "built-in" ma verranno utilizzati quasi esclusivamente *Shader* creati interamente da zero o al più ispirati da progetti disponibili online ai quali sono state eseguite modifiche per poter ottenere l'effetto grafico desiderato per questa applicazione. Il progetto cercherà tuttavia di limitare al massimo l'utilizzo di qualsiasi altra forma di asset come modelli 3d, animazioni e *textures* scaricati da altri portali, cercando di realizzarli sempre ad hoc per mostrare un valido esempio di applicazione 3D interattiva nella sua forma più completa possibile.

I software scelti sono completamente gratuiti, e nello specifico sono:

- **Blender** [1] per la modellazione delle mesh necessarie al funzionamento del progetto.
- **Paint.Net** [13] per la realizzazione e la modifica delle *textures* della scena.
- **Visual Studio** [22] per la scrittura del codice C# per la logica del progetto, e per la scrittura degli *Shaders*.
- **Unity 3D** [21] come ambiente di sviluppo collegato ad un compilatore capace di creare bytecode eseguibile per ottenere un'applicativo a partire dalle risorse e dagli assets generati.

Nella prima sezione di questo documento si fornisce un'**introduzione agli shaders** per chiarire cosa questi siano, accennarne una breve storia e spiegare come questi vengono ge-

stiti su *Unity 3D*. Successivamente nel capitolo relativo alla **modellazione e texturizzazione dell'ambiente** si descrivono i metodi utilizzati per ottenere i modelli 3D per gli ambienti, gli oggetti di scena ed il personaggio. Si mostra inoltre in breve come preparare un nuovo progetto *Unity 3D* e come sfruttare alcune tecniche di ottimizzazione. Si procede poi nella spiegazione del cuore del progetto, ovvero la **creazione degli shaders** e piu' nello specifico si descrivono le tecniche principali utilizzate, presentando alcune porzioni di codice realizzato per implementarle. Si tratta poi l'aspetto della **logica del gioco e del gameplay**, e quindi come si compone la gestione degli input, della navigazione all'interno della scena, delle regole e degli eventi di gioco. Si conclude mostrando alcune catture di schermo che mostrano lo stato finale del progetto completo.

Indice

Capitolo 1

Introduzione agli shaders

Descrivibili come veri e propri programmi indipendenti, gli *Shaders* sono oggi lo standard più utilizzato nella grafica per poter applicare colori, luci, ombre ed effetti specifici all'interno di una *scena*, intesa come collezione di oggetti a 2 o 3 dimensioni visualizzabile su uno schermo. Per quanto esistano tecniche e linguaggi che permettono di eseguire *Shaders* su CPU, i casi più comuni di uso di *shaders* moderni sono per la realizzazione di grafica ad uso cinematografico (come effetti all'interno di clip video o clip video interamente realizzate in grafica al computer) e *Videogames*, i quali sono normalmente eseguiti sulle Schede Video, composte da unità computazionali dedicate specificamente a calcoli matematico-algebrici. Questo è infatti l'ambiente ideale per processare calcoli per riflessi, posizioni, rotazioni, ed inoltre permette di non sovraccaricare la CPU, la quale può venire impiegata per altri scopi concorrenti, quale l'esecuzione di logica, comunicazioni di rete ed altre procedure non connesse al *rendering*.
Per ottenere calcoli più veloci, meno approssimativi, e più complessi volti ad ottenere risultati più realistici con miglior simulazione dell'impatto di luci in una scena nel minor tempo possibile, *Shaders* e Schede Video hanno subito enormi evoluzioni negli anni, tuttavia è importante specificare che la realizzazione di grafica al computer può necessitare di un certo grado di tocco artistico, il quale a volte scarta volutamente la possibilità di ottenere simulazioni complesse e fotorealistiche per puntare invece sull'ottenimento di effetti più unici, innovativi e particolari, facendo venire meno l'uso di tecniche moderne utilizzate per raggiungere una grafica realistica.

1.1 Breve storia

Nel 1972 due fisici ed informatici Ed Catmull e Fred Parke realizzarono un filmato [3] che mostrava una mano ed un volto completamente generati e renderizzati al computer. All'interno

di queste brevi clip, si intravedono metodologie e concetti che sono ancora oggi i pilastri dietro al funzionamento della CGI. È particolarmente interessante inoltre notare come inoltre nei titoli presenti nelle clip si utilizzi una terminologia molto a quella odierna, ed in particolare si fa riferimento al termine *Shading* per indicare il tipo "ombreggiatura" e di "smussamento" applicato alla superficie durante il rendering, e seppure non si hanno molte informazioni su come tale effetto sia stato realizzato, è molto probabile che si fosse ancora lontani dall'utilizzo di ciò che si intende oggi come *Shader*.

Il primo utilizzo del termine *Shader* per definire una tecnologia precisa di sviluppo viene attribuito alla *Pixar Animation Studio* (Della quale Ed Catmull è uno dei fondatori) nel 1988 con la pubblicazione di *Renderman Interface Specification*, un set di protocolli sviluppato per la generazione di immagini fotorealistiche al computer. *RenderMan Interface Specification (RISpec)* utilizzava un linguaggio simile a C chiamato *RenderMan Shading Language (RSI)* [4] per lo sviluppo di codice che definisse il colore finale che le geometrie 3D dovevano avere a schermo, dipendentemente da come luci e riflessi agivano sulle superfici. *RISpec* è stato supportato per oltre 30 anni, ed è stato il cuore del software *Pixar RenderMan* utilizzato dalla Pixar per la realizzazione di gran parte dei loro effetti CG e delle loro opere dal 1987 ad oggi. *Pixar RenderMan* è stato infatti utilizzato in opere come *La bella e la Bestia, Terminator II, Alien III, Jurassic Park, Il Re Leone (1994), Titanic, Il gigante di Ferro, Il Signore degli Anelli (Trilogia completa), Star Wars Episodio III, WALL-E, Avatar, The Avengers, Frozen, Interstellar, Gli Incredibili 2, e Napoleon (2023)* [14]. E questo è solo per citare alcune tra le creazioni più famose che hanno fatto uso di questo software dalla sua creazione ad oggi, infatti *Pixar RenderMan* e dunque la sua tecnologia di base *RISpec* hanno non solo dato un enorme impulso allo sviluppo della grafica 3D, ma ne sono stati anche importanti porta bandiera per decenni, tanto da aver contribuito a opere cinematrografiche conosciute praticamente da chiunque.

Nonostante la *Pixar* sia stata pioniere nel mondo dello sviluppo di tecnologie che hanno portato a quello che oggi definiamo *Shader*, per svariati anni la grafica al computer si è servita di strumenti più statici e meno programmabili per la realizzazione di output a schermo. Nel 1992 *Silicon Graphics* sviluppa il sistema *IRIS (Integrated Raster Imaging System)* [19] per la renderizzazione di grafica raster, considerato semplice da utilizzare ma ancora poco scalabile all'uso pratico e soprattutto limitato ad hardware ben specifici. Tale sistema si evolse nelle API grafiche *OpenGL* pubblicate nel 1992, la cui grande novità iniziale fu la possibilità di adattare le API a moltissimi hardware diversi, anche poco potenti tramite lo sviluppo di driver dedicati. Competitor di *OpenGL* fu molto presto *Direct3D*, rilasciata nel 1996 dalla *Microsoft*. Ci furono tentativi di unificare le API in un unico progetto nel 1997 [12] sotto pressione di alcuni volti

noti nello sviluppo dei videogame [11], ma l'idea venne abbandonata nel giro di pochi mesi. Fu solo nel 2006 con il rilascio di OpenGL 2.0 ed in particolare del *OpenGL Shading Language (GLSL)* che gli *Shaders* si sono evoluti in una forma molto simile a quella che conosciamo oggi, deprecando le *fixed-functions* che sino ad allora costituivano lo strumento principale delle API.
Nel corso degli anni *OpenGL* e *Direct3D* hanno subito evoluzioni importanti che oltre ad ottimizzare la performance delle funzioni rese disponibile dalle API, hanno incluso nuove funzionalità oggi fondamentali nello sviluppo di applicazioni 3D quali il *Geometry Shader*, il supporto all'*Instancing* ed alla *Tessellation*. *OpenGL* cessa di venire supportata nel 2016 con il rilascio delle API *Vulkan*, considerabile come loro diretto successore, sviluppate dalla stessa compagnia *Khronos Group* con il supporto inizialmente di *AMD*.

Oggi *Direct3D* e *Vulkan* sono le API standard sulla quale si sviluppano applicazioni grafiche per *Windows*. *Vulkan* ed *OpenGL* sono invece le API solitamente scelte per quanto riguarda lo sviluppo su *Linux*, mentre *Metal*, un derivato di *OpenGL*, sono le API tipiche utilizzate su *MacOS*.

1.2 Esempi di applicazione

Come si è visto, gli *Shaders* hanno subito evoluzioni risultanti in effetti sempre più fotorealistici, ma esistono anche tecniche che al contrario rinunciano al realismo per ottenere stili grafici specifici. Si presentano quindi alcune delle tecniche e tipologie più comuni di *Shaders*:

- *Physically Based Shader*, utilizzati nell'approccio *Shaders PBR (Physically Based Rendering)* definiscono le tecniche moderne utilizzate per creare un modello che approssimi la realtà, e quindi per ottenere effetti fotorealistici. Queste tecniche normalmente utilizzano un set di *Textures* ampio per applicare ad ogni frammento di uno *shader* molteplici informazioni da considerare durante il *rendering*, come la *Diffuse map* che determina i colori principali della superficie, la *Normal map* che aggiunge informazioni su microimperfezioni della superficie che possono influire sul modo in cui la luce influisce sull'oggetto, la *Metallic map* che influisce nel modo in cui la superficie otterrà un effetto metallico, la *Smoothness map* che determina il valore moltiplicativo dell'effetto di riflessione di Blinn-Phong del frammento e la *Occlusion map* che determina il valore moltiplicativo di quanto una luce può contribuire all'illuminazione del frammento. Questi sono solo alcuni esempi di mappe utilizzabili, e per quanto esistano altre mappe standardizzate nell'approccio *PBR*, è sempre possibile ideare nuovi modelli che includono

ulteriori mappe.

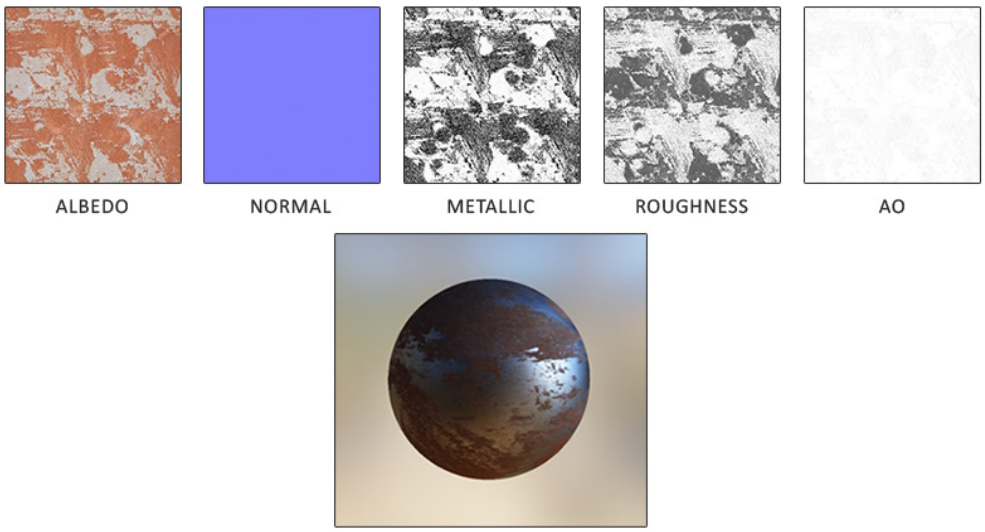


Figura 1.1: PBR Textures example, learnopengl.com

- *Water*, ovvero l'acqua intesa come superficie di oceani, laghi e fiumi, necessità di un modello ad hoc per poter imitare il modo in cui il movimento di una massa liquida modifichi costantemente in modo casuale vari parametri di riflessione e rifrazione della luce.

Figura 1.2: Water Shader example, godotshaders.com

- *Skybox*, ovvero il cielo comprendente nuvole, sole, luna, stelle, ed altri elementi statici o dinamici che fanno parte del *background* di una scena.

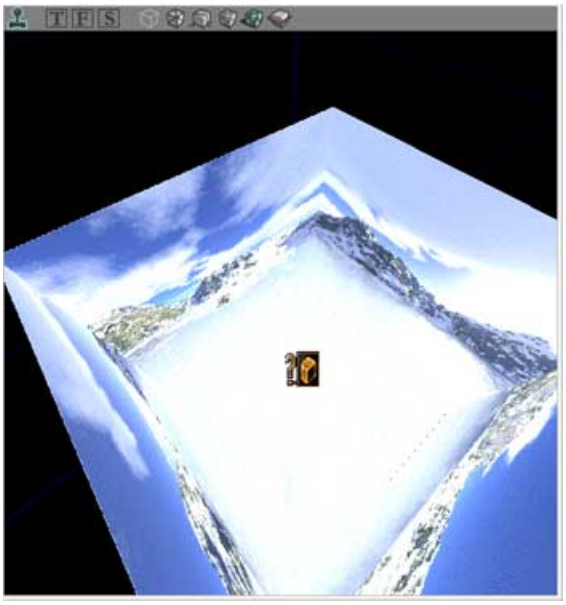


Figura 1.3: Skybox example, creativegames.org.uk

- *Glow & Emission*, ovvero effetti di emissione di luce costante o dinamica, utili per simulare superfici di lampade, fiamme, magma o elementi reali o immaginari.

Figura 1.4: Magma shader example, blendernation.com

- *Outline & Cartoon*, ovvero effetti utilizzati per colorare il bordo di alcune superfici per evidenziare un elemento, ad esempio se lo vogliamo individuare come elemento selezionato in una scena, o per realizzare effetti stile cartone animato.

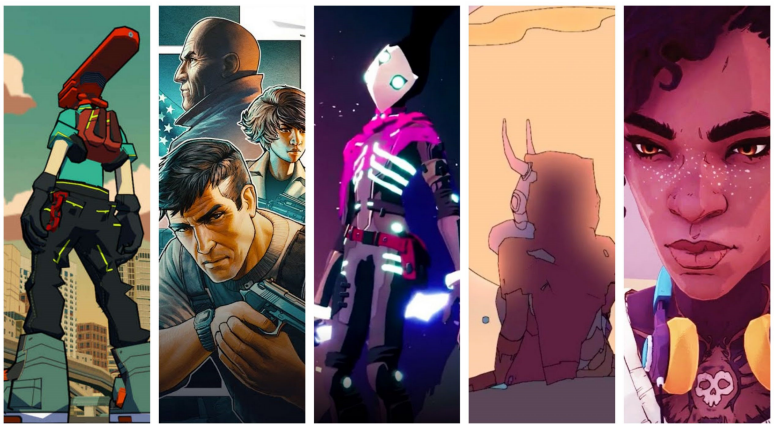


Figura 1.5: Cartoon shader examples in videogames

- *Screen & Post-Processing*, ovvero tecniche per realizzare *Shaders* che manipolano il risultato finale di un'immagine renderizzata prima che venga mostrata a schermo per applicare effetti particolari per migliorare la qualità della immagine, realizzare funzionalità specifiche di un software o come scelta stilistica.

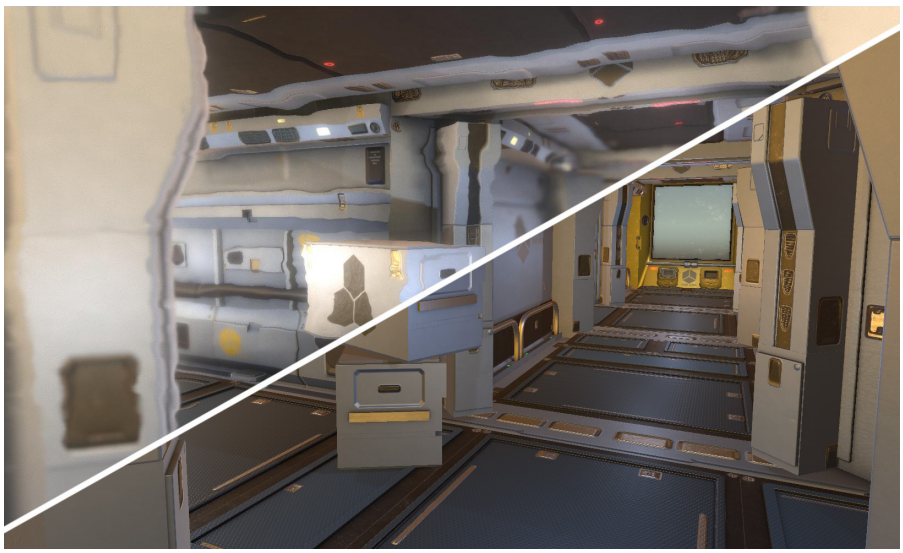


Figura 1.6: Post Processing example, benmandrew.com

1.3 Shaders su Unity

Unity utilizza un linguaggio per la produzione di *Shaders* chiamato *ShaderLab* [20].
È possibile scrivere su Unity shaders su files con estensione *.shader*, o utilizzando lo *Shader Graph*, uno strumento visuale a nodi simile allo *Shader Editor* di *Blender*, il quale però per quanto semplifichi la scrittura di questi effetti è sconsigliato in quanto produce codice che richiede più tempo per essere compilato e permette un minore controllo sulla performance.

Per questa ragione verranno presentati e spiegati nei capitoli successivi shaders unicamente scritti in *ShaderLab* su *VisualStudio*.

Gli *Shader* di *Unity* includono il concetto di *Shader Variant*, ovvero versioni differenti di uno stesso *Shader* che includono diverse funzionalità. Sostanzialmente quando uno *shader* necessita di utilizzare funzionalità che possono venire utilizzate opzionalmente per ragioni stilistiche o di performance, Unity permette di generare delle *Shader Variants* che includono o escludono le suddette *feature*. Quando si vuole utilizzare una di queste funzionalità si va ad utilizzare la *Shader Variant* che la include, o quella che la esclude nel caso si voglia ignorare la funzionalità.

Gli *Shader* vengono applicati attraverso dei *Materiali* che ne contengono e linkano le varie proprietà modificabili, come il colore, le *texture* o parametri specifici. Ogni materiale può venire associato ad un unico *Shader*. Unity fornisce un'interfaccia grafica dove è possibile associare determinate risorse e variabili al materiale, a seconda di quali proprietà dello *Shader* vengono programmate per essere esposte nella suddetta interfaccia.

Ogni volta che un progetto di Unity viene compilato, il motore grafico compilerà automaticamente tutti gli shaders che vengono utilizzati, in ogni *Shader Variant* utilizzata e per ogni *Graphic API* pre determinata. Gli *Shaders* scritti in *ShaderLab* o tramite il *Shader Graph* vengono compilati per le diverse API utilizzando diversi compilatori, quali:

- Microsoft FXC HLSL per le API DirectX.
- Microsoft FXC HLSL con bytecode tradotto in GLSL usando HLSLcc per le API OpenGL (Core & ES).
- Microsoft FXC HLSL con bytecode tradotto in Metal usando HLSLcc per le API Metal.
- Microsoft FXC HLSL con bytecode tradotto in SPIR-V usando HLSLcc per le API Vulkan.
- Le altre piattaforme, come le *Console* di gioco, utilizzano compilatori specifici forniti dagli sviluppatori delle *Console* ai *developers* abilitati.

Capitolo 2

Modellazione e texturizzazione dell'ambientazione

Questa sezione è dedicata alla modellazione ed alla texturizzazione dell'ambientazione dell'applicazione mediante Blender. Si intende quindi preparare le geometrie statiche che si andrà ad utilizzare in tutti i processi successivi.

Per tale scopo ci si serve di un motore grafico, nel nostro caso Unity 3D, per risparmiare del tempo ottenendo svariate funzionalità di base già fornite dal software, e garantendo una alta compatibilità su diverse macchine.
Di Unity si utilizzeranno principalmente i sistemi e le tecnologie che ci permetteranno di:

- Organizzare gli assets (Modelli 3D, animazioni, immagini, codice) in cartelle.
- Astrarre la gestione della scena 3D, con funzioni di istanziazione, cancellazione, traslazione di geometrie.
- Connettere la logica del codice del progetto agli asset, creando codice che permette di interagire con gli assets in modo molto agevole.
- Gestire gli *Shaders* tramite il concetto di materiale con interfacce semplificate per velocizzare il processo di assegnamento degli *Shaders* alle geometrie.

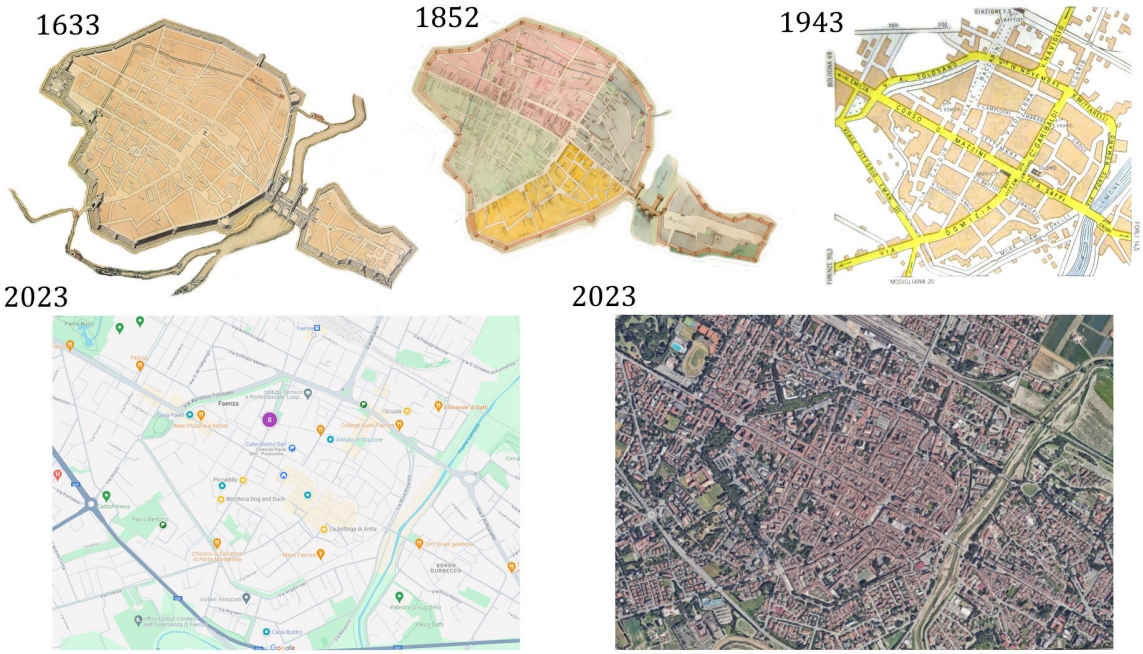
Uno strumento che risulterà utile durante varie fasi sarà Blender, un software libero che si sfrutterà per la modellazione 3D e la texturizzazione.
Con Blender ci si focalizzerà in particolare nella creazione degli edifici e dei *props* per la scena.

Si discuterà infine in questa sezione anche altri strumenti e tecniche che permetteranno di

completare la preparazione della scena per procedere al cuore di questo documento, ovvero la creazione degli Shaders.

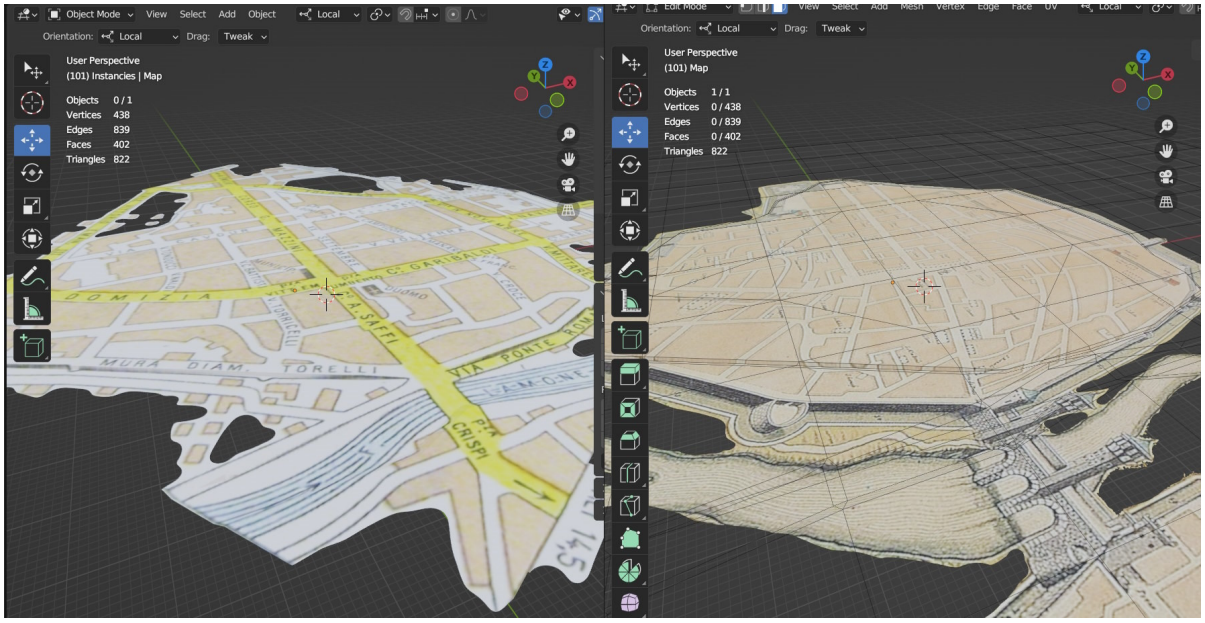
2.1 Creazione dei modelli 3D e delle textures

Blender è un programma complesso e molto sfaccettato, e le tecniche utilizzate per la realizzazione della scena sono varie, dunque è utile un riassunto breve degli elementi principali utilizzati per la generazione degli asset 3D utilizzati.
Si noti che l'obiettivo in questa fase è realizzare una piantina 3D del centro storico di Faenza. Per fare ciò innanzitutto è utile ricercare varie cartine di Faenza risalenti a diverse epoche, successivamente ruotate e ridimensionate per poter essere facilmente sovrapponibili ed intercambiabili.

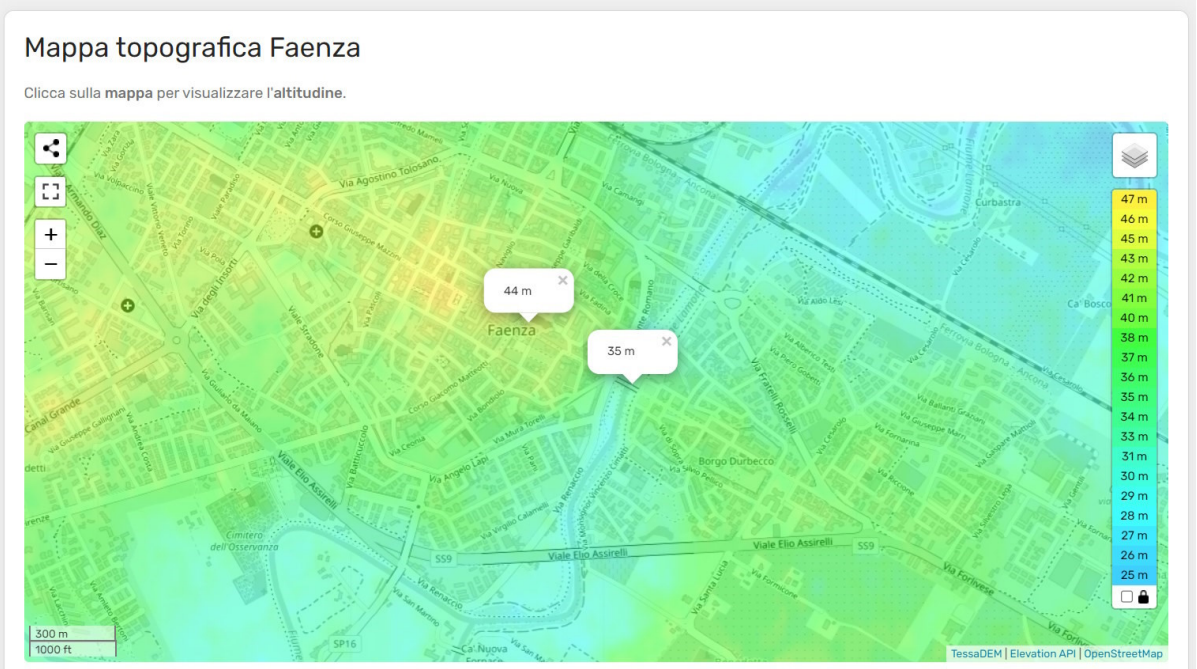


Le cartine storiche si possono ottenere da vari portali, ed in particolare per questo progetto si utilizza una cartina risalente al 1633 [8], una cartina risalente al 1838 [18], una risalente al 1943 [5], e le cartine moderne disponibili su Google Maps [7].
Le cartine provenienti da google maps sono naturalmente le più affidabili in termini di proporzioni e forme e sono dunque utilizzate come scala riferimento per le proporzioni durante la modellazione, mentre le cartine più antiche sono migliori per una rappresentazione più fedele relativamente al popolamento di certe zone e strade in tempi in cui il perimetro cittadino era più contenuto. Bisogna ricordare infatti che per ragioni stilistiche e per avere un ambiente

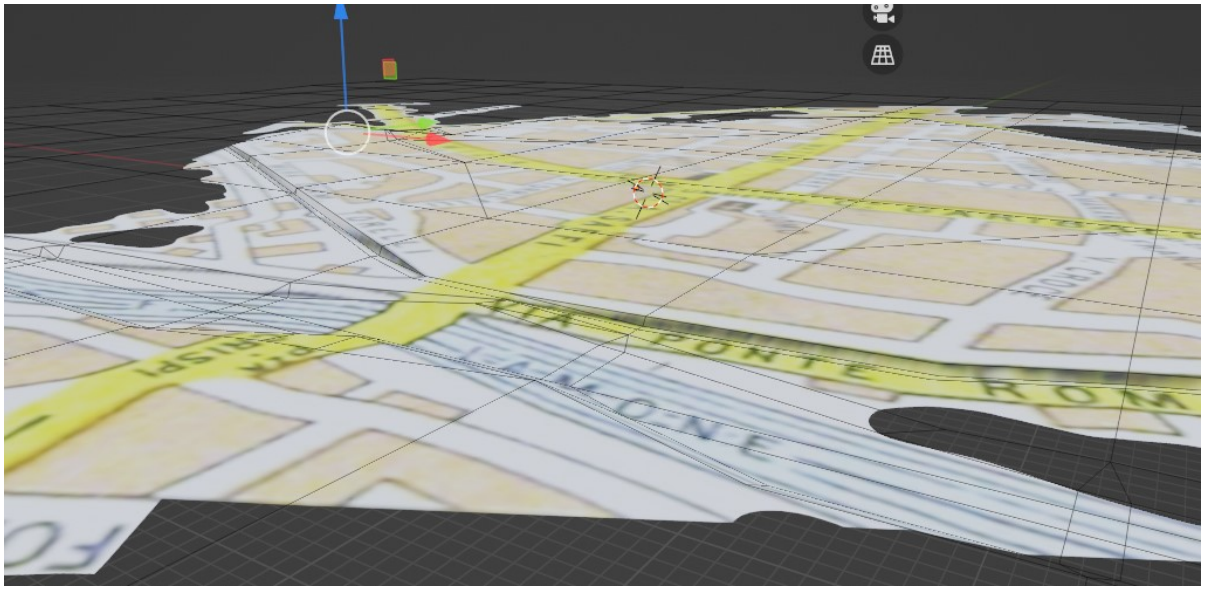
da replicare meno oneroso, si è scelto di puntare alla realizzazione di una versione di Faenza genericamente ispirata all'epoca post-medievale. Ciò permette ad esempio di ignorare l'urbanizzazione esterna alle mura di Faenza avvenuta in epoca contemporanea, focalizzandosi principalmente nella zona del centro storico.
Su Blender si applicano queste cartine come *textures* su un piano, assicurandosi che siano in una scala ben definita, ad esempio 1u = 1m, e che siano orientate con la parte alta dell'immagine rivolta a Nord.
Una volta completato il processo è quindi possibile visualizzare le cartine e sostituirle con pochi click mantenendo le proporzioni di riferimento identiche al cambiare della cartina.



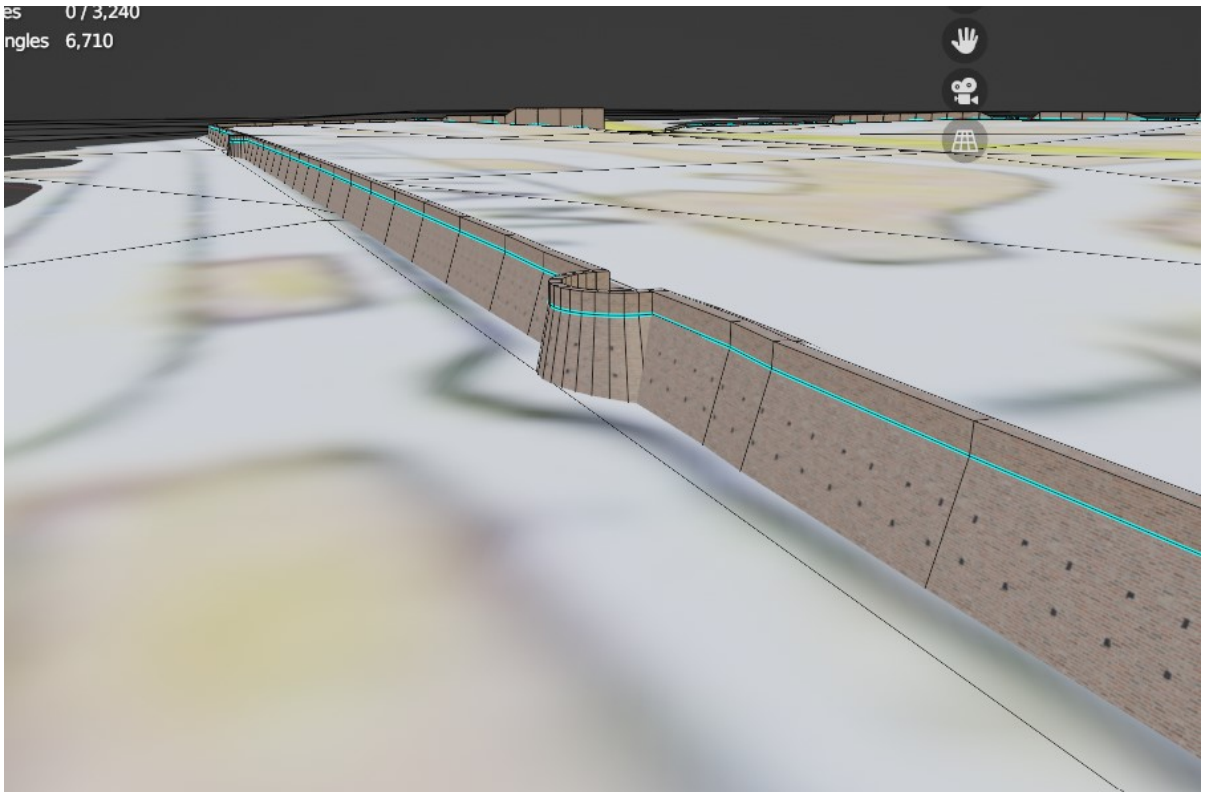
Naturalmente le cartine satellitari moderne sono nettamente più precise rispetto a quelle create con gli strumenti del passato, dunque le si utilizzano come riferimento principale.
Il passo successivo è generare un'approssimazione del terreno e della sua elevazione. Si suddivide il piano texturizzato con la cartina, ed utilizzando topographic-map.com [17] si individua l'altitudine delle varie zone di Faenza, e le si replicano sul piano 3D. Si noti che nei punti più alti Faenza tocca i 46m sul livello del mare, mentre nei punti bassi, ovvero in prossimità del fiume, scende a 33m sul livello del mare.



Lo scopo del progetto non è ricreare una città fedele al 100% e dunque ci si può accontentare di un lavoro eseguito in maniera approssimativa;
Si prendono alcuni punti focali come la piazza, il contorno delle mura, la striscia descritta dal fiume, la zona del borgo ecc, e si modifica il piano in modo da impostare la sua altitudine in prossimità di queste zone chiave con l'altitudine riportata da topographic-map.com utilizzando come scala di altitudine lo spostamento sull'asse Z di blender dove bisogna ricordare che si è scelto 1u = 1m di spostamento.
Si è così creata una bozza della forma del terreno di gioco sulla quale si può iniziare a creare e inserire edifici e *props*.

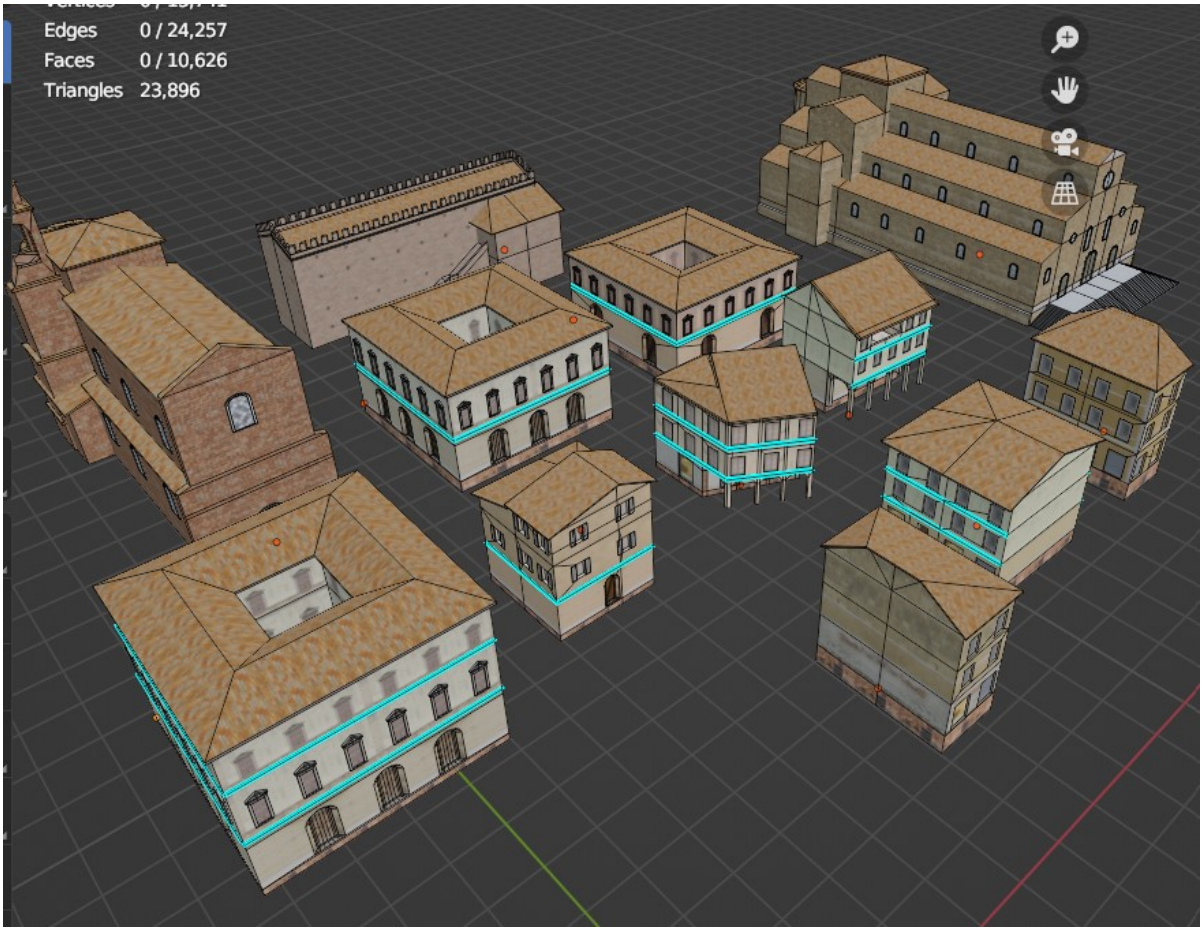


Utilizzando una curva 'path' si segue il contorno delle mura. Si disegna un pezzo di muro dritto, con l'apposito modifier lo si applica alla curva, e lo si ripete molteplici volte con un *array modifier* in modo che segua la curva sino alla fine. Si è in questo modo generata l'intera cinta muraria. Si crea qualche pezzo di muro variante come le postazioni di guardia e la si inserisce in alcuni punti delle mura come indicato dalla cartina di Faenza medievale.



Per popolare la mappa ci si serve principalmente di edifici. Questi vengono modellati sulla

base di veri edifici di Faenza. Partendo da immagini reperibili da internet, in particolar modo da Google Street View, ed eseguendo misurazioni delle dimensioni su Google Maps e Google Earth, si ottengono informazioni sufficienti per per poter replicare le forme delle strutture al meglio. Fare sì che gli edifici abbiano delle proporzioni corrette è in questa fase la priorità, in quanto palazzi ad esempio poco profondi o troppo alti risultano immediatamente poco convincenti all'occhio. Gli edifici che si rappresentano sono tutti basati su palazzi realmente esistenti di Faenza, tuttavia mentre quelli particolarmente iconici possono venire sfruttati una volta sola (come ad esempio il Duomo, il palazzo comunale, o il ponte che porta nella zona del Borgo), altri con un aspetto più generico possono venire riciclati più volte in giro per la scena.
Con un set di strutture sufficientemente alto, è possibile avere abbastanza varietà da ingannare l'occhio e non fare notare la ripetizione degli stessi modelli.
Nell'immagine che segue vengono mostrati alcuni dei modelli realizzati per la scena.



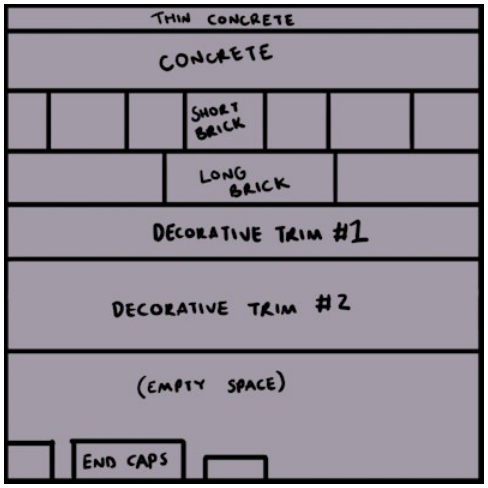
I dettagli su cui ci si focalizza nella creazione di questi modelli sono:

- Come già accennato **le proporzioni** per convincere l'occhio della naturalezza di quello

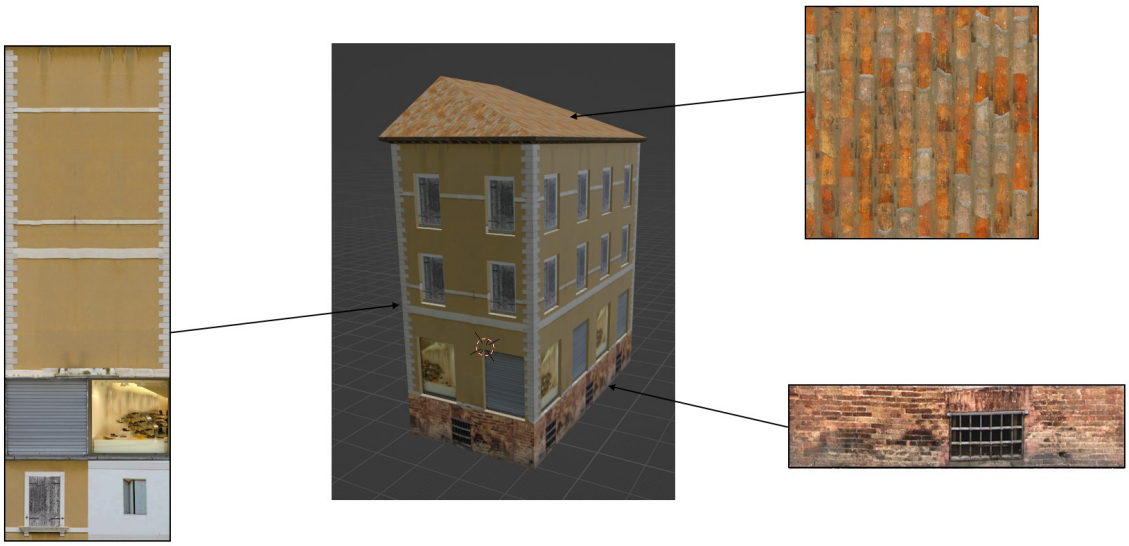
che sta guardando.

- **La sporgenza del tetto** per dare un effetto di profondità agli edifici.
- **Il ricamo che divide i piani degli edifici** altro elemento che aumenta la profondità e la tridimensionalità.
- **Le porte e le finestre** dettagli fondamentali che compongono la forma base di un qualsiasi edificio.

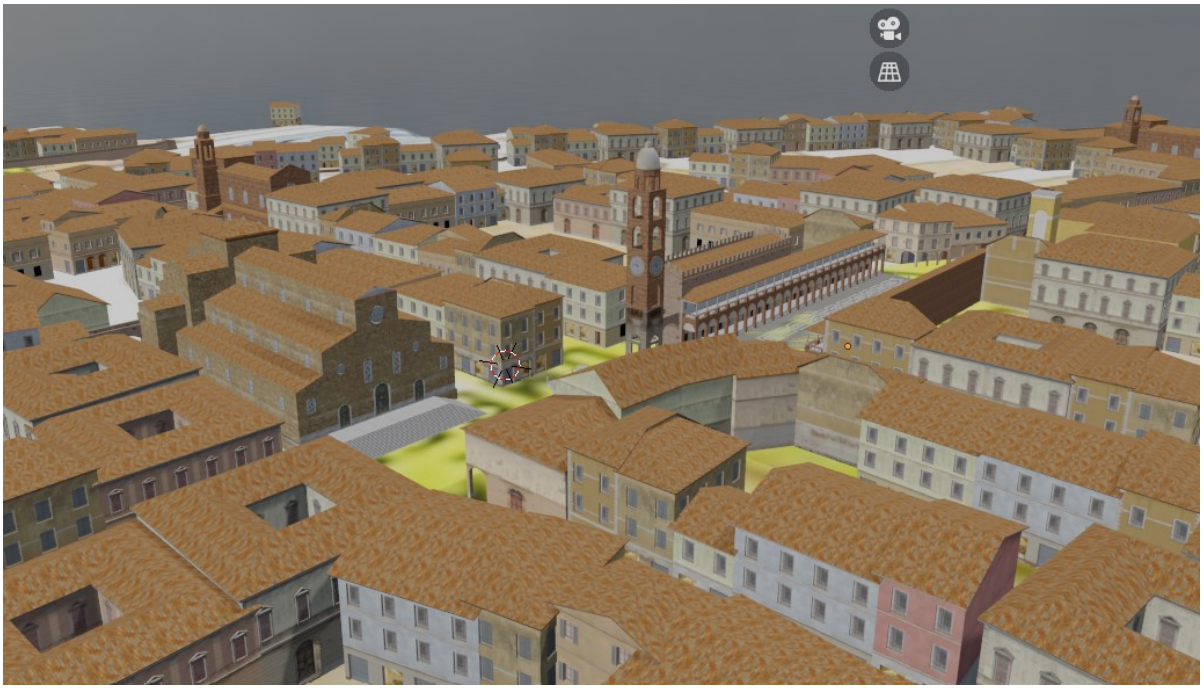
Questi sono elementi minimali ma sufficienti per ottenere modelli efficaci ma senza esagerare con il numero di poligoni, considerazione importante per poter garantire una performance accettabile.
Per la realizzazione delle *textures* invece si utilizza una tecnica chiamata *"Trimsheet"*, simile al concetto di *Texture Atlassing*, che consiste nell'inserire nella stessa *texture* una o più immagini che possono venire riciclate su più edifici.
Di fatto elementi come le tegole, il legno, le finestre, l'intonaco ecc.. di uno specifico colore sono elementi che si possono ripetere su più edifici, di conseguenza si costruisce un set di *textures "Trimsheet"* da riciclare su più edifici differenti. Questa tecnica permette inoltre di sfruttare la proprietà di "tiling" delle *textures*, ovvero la loro ripetizione su uno o più assi. In particolare si sfrutta l'*horizontal tiling* per coprire un'intera parete ripetendo la *texture* tutta attorno ad un edificio, oppure il *horizontal* e *vertical tiling* per materiali come il soffitto, il tetto o il pavimento dove ci serve una ripetizione della *texture* in tutte le direzioni. Questa tecnica ci permette dunque di riciclare le *textures*, ma anche di pianificare un sistema per mantenere una densità di pixel per metro (texel density) più controllata.
Felyx McGarry approfondisce il concetto di *Trimsheet* in una sua guida su Artstation [6].



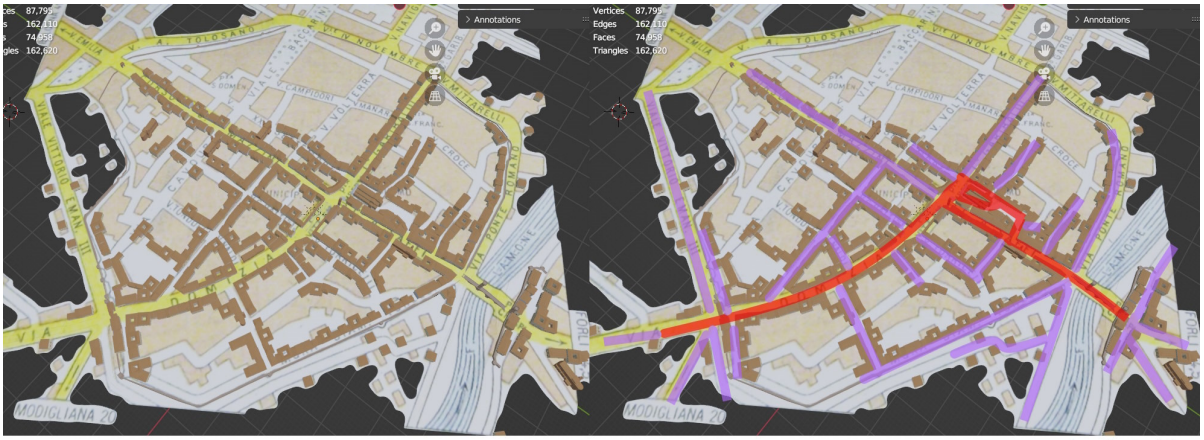
Si compongono le *Trimsheet* usando *textures* ed immagini prelevate da Textures.com [16], ed immagini scattate sul posto con uno smartphone.
Per realizzare le Normal Maps e le altre mappe si è utilizzato Materialize [10], un software molto semplice che permette di generare approssimazioni di Normal maps, Ambient occlusion maps, Height maps e altre *textures* a partire da una diffuse. Per unire le *textures* in *Trimsheet* utilizzabili si è utilizzato Paint.Net [13], un semplice software di modifica immagini molto leggero, versatile ed ampliabile con innumerevoli plugin.



Una volta soddisfatti dal numero di assets creati, si può iniziare a posizionarli opportunamente sul terreno precedentemente creato.



Replicare perfettamente l'intera città sarebbe dispendioso in termini di tempo ed inutile ai fini di questo progetto, dunque l'idea è di popolare con strutture solamente le vie che si intende poter visitare nell'applicazione e le vie che non sono pensate per essere visitabili, ma sono comunque visibili al giocatore, ad esempio le vie che incrociano il percorso previsto.
Si trasmette così l'impressione che la città sia completamente modellata, quando invece non si è semplicemente in grado di vedere le zone non realizzate.



Vista aerea scena completa

Vista aerea scena navigabile

Conclusa la parte di modellazione si procede ad importare la scena su Unity. Si può esportare l'intera scena in .fbx, o semplicemente salvare il file .blend in una cartella del progetto

di Unity, in quanto recenti versioni dell'engine sono capaci di importare direttamente questo formato, molto comodo se si intende fare modifiche al file senza il rischio di perdita di informazione che può avvenire con operazioni di import/export.
Importato il file, lo si può inserire nella Hierarchy e quindi nella scena visualizzabile. Si utilizzano temporaneamente gli *shaders* di default di Unity, assegnado quindi le *textures* ai materiali corrispondenti, e si ottiene la seguente scena:

Blender sarà utilizzato nuovamente più avanti nel progetto per altri piccoli dettagli e modifiche, in particolare per la creazione dell'erba e dei dettagli del terreno, tuttavia la parte di modellazione più consistente termina qui.

2.2 Creazione ed acquisizione degli ulteriori assets necessari

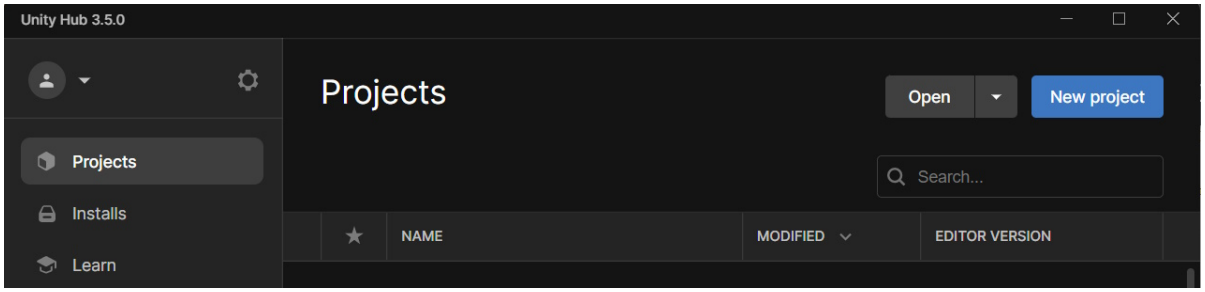
Per finalizzare la scena si acquisiscono alcuni assets di supporto finali.

- **Un personaggio**, per poter dare una rappresentazione più chiara della navigazione all'interno della scena. Modellare un personaggio in 3D richiede tempo e capacità non connesse allo scopo del progetto, dunque si opta per un modello acquisito gratuitamente da CG Trader. Al personaggio viene inoltre aggiunta un'*Armature* di forma umanoide tramite *Blender*.
- **Le animazioni di movimento**, compatibili con l'*Armature* creata. Le animazioni comprendono la posa ferma (in piedi), la camminata frontale, laterale, la camminata all'indietro e la corsa.
- **Gli alberi**, realizzati tramite il software *Speed Tree* [15] con il quale è possibile ottenere dei risultati proceduralmente e relativamente in fretta. Si generano in particolare un Pino domestico ed un albero di forma più generica, ispirato ad una Quercia.
- **Le torce a muro**, che si utilizzeranno assieme ad effetti particellari di un focolare e ad alcune luci nella scena per illuminare la città.
  Si ottiene per semplicità il modello gratuitamente da CG Trader.
- **Barriere in metallo**, che si utilizzeranno per bloccare alcune strade in modo da forzare il percorso lungo alcune vie prestabilite.
  Si ottiene il modello utilizzato gratuitamente da Sketchfab.

2.3 Creazione progetto in Unity

Per iniziare a sviluppare su Unity è necessario installare l'engine.
La versione utilizzata per la realizzazione di questo progetto è Unity 2021.3.28 con le pipelines URP, ma plausibilmente questo dovrebbe funzionare su qualsiasi versione che supporta le pipelines URP.
È possibile ottenere questa o altre versioni dell'engine tramite il portale Unity Download Archives.

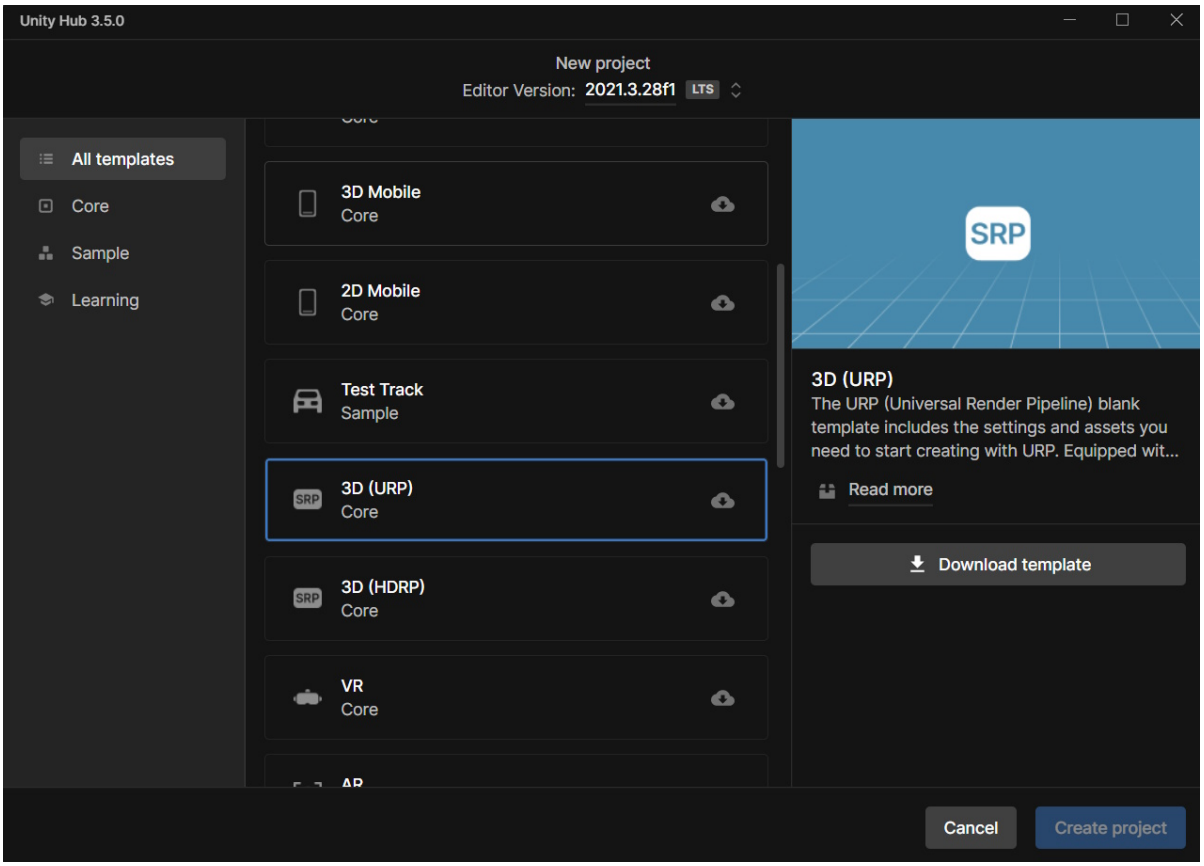
Di recente Unity richiede anche l'installazione di Unity Hub.
Da questo portale è possibile avviare e gestire agevolmente i progetti Unity.
Per creare un nuovo progetto bisogna innanzitutto cliccare su **New project**.



Si sceglie poi un progetto di tipo **3D URP (Core)** il quale inizializza una scena del tipo:

- **3D**, ovvero le impostazioni di base dell'engine verranno inizializzate con valori più consoni allo sviluppo di un'applicazione 3D.
- **URP** indica le render pipelines che si andranno ad utilizzare. Unity fornisce 3 tipi di render pipelines: HDRP, URP e Legacy. La differenza sostanziale sta nel fatto che le render pipelines Legacy utilizzano un approccio diverso (e deprecato) nella scrittura e nella gestione degli *Shaders* e dei buffer. HDRP (High Definition Render Pipeline) e URP (Universal Render Pipeline) sono i nuovi sistemi supportati dall'engine, e si differenziano principalmente per le funzioni fornite dalle librerie *Shaders* di Unity; La prima fornisce funzioni che sfruttano al massimo la GPU e le API grafiche per ottenere un rendering più fotorealistico, ed è pensato principalmente per applicazioni di tipo cinematografico o al più per sviluppo di applicazioni destinate a macchine di nuova generazione e molto potenti. Le render pipeline URP invece sono indicate per una maggior performance e compatibilità con più piattaforme possibili, specialmente nell'ambito console di gioco. Per questo progetto vengono dunque scelte le Universal RP, e gli *Shader* che si scriveranno sfrutteranno funzioni di base contenute nel pacchetto URP.

- **Core** indica che il progetto sarà vuoto e non fornirà esempi o template preconfigurati. L'obiettivo è di mostrare un processo di sviluppo più completo possibile, dunque questa è la scelta utilizzata.



La prima volte che si avvia un progetto di questo tipo è necessario scaricare il template, procedura che prenderà pochi istanti dato che si tratta di un template vuoto.

Una volta all'interno del progetto, si crea una nuova scena all'interno del cui piazzare successivamente i modelli 3D e configurare i Materiali applicando gli *Shaders* che si andrà a scrivere.
Si farà anche uso del sistema di Terrain di Unity per velocizzare la procedura di piazzamento delle patch dell'erba e degli alberi nella scena, tuttavia la modellazione del terreno avverrà su blender, per poi applicare la geometria creata al *terrain* di Unity tramite uno script che si realizzerà successivamente.

2.4 Sistemi ed Ottimizzazioni

Per garantire una performance accettabile e per automatizzare alcune procedure si pianificano ed implementano sistemi ausiliari prima di passare alla scrittura effettiva del codice.

2.4.1 Collisioni con l'ambiente

Unity 3D fornisce un comodo sistema di gestione della fisica. Questo pacchetto consiste in un insieme di componenti e di API. Di esso sono stati utilizzati i componenti *Mesh Collider* per modellare una struttura dati che mantiene le informazioni di collisione di un oggetto a partire da una mesh. Questo sistema non è particolarmente performante dunque è importante ricordare di utilizzarlo solo se strettamente necessario. A questo proposito si implementa un semplice algoritmo *C#*.

```
void Start()
{
    colliders = gameObject.GetComponentsInChildren<MeshCollider>();
}

void Update()
{
    if (id >= colliders.Length) id = 0;
    int scanEnd = Mathf.Min(id + scanPerFrame, colliders.Length - 1);

    while (id <= scanEnd)
    {
        colliders[id].enabled = Vector3.Distance(
          player.transform.position,
          colliders[id].transform.position
          ) < enableDistance;
        id++;
    }
}
```

Questo codice acquisisce innanzitutto tutti i *Mesh Collider* in un array *colliders*, poi ad ogni frame ne esamina una certa quantità *scanPerFrame* e li marca come *enabled* o *disabled* in base alla distanza dalla telecamera. Ci si assicura così che solamente i MeshCollider attorno al giocatore siano abilitati.
In alternativa ai *Mesh Collider* si usano anche i *Box Collider*, sistemi simili ma che modellano le collisioni tramite *Bounding Box 3D*. Questo sistema, molto più performante verrà utilizzato ogni volta in cui è possibile approssimare una collisione ad un cubo.
Ultimo tipo di *Collider* utilizzato sarà il *Terrain Collider*, che viene invece generato a partire

da una *heightmap*, e che sarà utilizzato successivamente per realizzare la mesh che rappresenta il suolo della scena. Questi due sistemi interagiranno con il programma in due modi;

- Tramite il **Character Controller**, il quale verrà usato per muovere il personaggio. Questo componente permetterà di dare un volume al *character* che potrà muoversi attraverso determinate API nello spazio 3D, vincolato però dai *Colliders* nella scena.
- I **Raycast**, ovvero delle API che a partire da un raggio "sparato" da una certa posizione, permettono di ricevere informazioni su quali *Colliders* il raggio interseca. Si utilizzerà questo sistema principalmente per la selezioni di oggetti nello spazio 3D.

2.4.2 LOD System

Un elemento che impatta notevolmente la performance della GPU possono essere il numero di poligoni che vengono renderizzati nella scena, specialmente in casistiche come questa dove si intende avere un intero centro storico a schermo. Si implementa quindi un *LOD (Level of Detail) System* che permetta di scegliere di mostrare una versione ridotta di poligoni di un certo modello quando questo si dovesse trovare distante dalla telecamera. Questo sistema che decide di fare il cambio di modello ad una certa distanza scelta è già presente all'interno dell'engine, quindi è sufficiente creare una o più versioni semplificate di uno stesso modello a cui si intende applicare il sistema di LOD. Questo sistema inoltre fornirà una variabile *unity_LODFade* con cui sarà possibile ottenere lo stato di transizione del modello da uno con più poligoni ad uno con meno poligoni (o viceversa) per creare un effetto più graduale durante il cambio di modello attraverso lo *Shader* del modello. Si mostrerà come implementare questa funzione in un capitolo dedicato. La variabile *unity_LODFade* ed altre variabili di base di Unity utilizzabili negli shaders sono consultabili nella documentazione di Unity [2].
Il *LOD System* viene implementato per i modelli più complessi della scena, tra cui quelli degli alberi, delle transenne e delle lanterne. Si ignora il sistema invece per gli edifici dato che si tratta di mesh già piuttosto semplici e limitate nel numero di poligoni. La scena contiene infatti centinaia di palazzi, ed applicare un sistema del genere per un elevato numero di oggetti potrebbe risultare più costoso del beneficio che si spera di ottenere sulla performance, considerando appunto che i modelli su cui si applica il sistema sarebbero molti e già piuttosto *low-poly*.

2.4.3 Terrain system

Per la rappresentazione del suolo si utilizza il *Terrain Component* di Unity, naturalmente con shaders che verranno programmati e mostrati in capitoli successivi. La comodità della scelta

del *Terrain* di Unity risiede nel fatto che questo offre una serie di vantaggi:

- La possibilità di creare una mesh 3D che rappresenta il suolo a partire da una *heightmap* che può anche essere modificata direttamente in engine tramite strumenti a pennello.
- La possibilità di piazzare velocemente modelli 3D di alberi e cespugli utilizzando uno strumento a pennello.
- La possibilità di disegnare una mappatura su *texture* che determina dove i modelli 3D che rappresentano dettagli del suolo come sassi e prato devono venire renderizzati. Il sistema inoltre si occuperà di inizializzare il rendering dei dettagli del prato usando il sistema di *GPU Instancing*, ovvero renderizzando lo stesso modello 3D più volte in una singola *draw call*.

A partire dal modello realizzato in *Blender* per rappresentare il terreno, viene creata la *heightmap*, che sarà poi passata al *Terrain* di Unity 3D. Sarebbe possibile utilizzare direttamente il modello 3D creato su *Blender*, tuttavia "convertire" la mesh creata in un sistema compatibile con il *Terrain* di Unity permetterà più flessibilità nel caso in cui sia necessario fare delle modifiche all'ultimo al terreno direttamente sull'engine 3D.
Per poter creare la *heightmap* a partire dal modello 3D del terreno si crea un semplice script *C#* che utilizzando dei *Raycast* eseguiti a griglia sul modello 3D del terreno vadano a sondarne l'altitudine, riportando il risultato all'interno di un *color channel* della *heightmap*. Successivamente la prima modifica che si esegue riguarda la creazione delle montagne in prossimità della zona di *Porta Montanara* che rappresentino l'inizio degli appennini. Si creano queste montagne semplicemente usando lo strumento a pennello del *Terrain* di Unity.

2.4.4 Esterno del Terrain

Ultimo sistema che viene presentato, è il sistema che gestisce l'esterno del *Terrain*. Normalmente su Unity il terreno ha una dimensione limitata e ben definita, dove i confini di quest'ultimo corrispondono ai confini della *heightmap* utilizzata per generarlo.
Per dare continuità al terreno all'infuori dei confini si implementa in *C#* un complicato sistema che genera all'infuori dei confini del *Terrain* una mesh che rappresenta la continuità del terreno. Questa mesh viene calcolata solo ad una certa distanza attorno alla telecamera, e quando quest'ultima si sposterà, la mesh dell'esterno del terreno verrà ri-generata costantemente, così facendo si ottiene un terreno che pare essere illimitato in ogni direzione, quando in realtà viene generato proceduralmente sempre e solo attorno alla telecamera. Mentre il *Terrain* di Unity ha una altezza e forma ben predicibile in quanto generato dalla *heightmap* prestabilita,

la parte esterna si implementa tramite un algoritmo procedurale che utilizzando varie funzioni matematiche tra cui *Seno* e *Coseno*, approssima il terreno generato proceduralmente a delle semplici colline. È difficile notare questo sistema all'interno dell'applicazione in quanto il percorso che l'utente potrà seguire è limitato e in gran parte localizzato all'interno del centro storico, tuttavia questo sistema permette di visualizzare Montagne e Pianure all'orizzonte per svariati chilometri quando ci si trova fuori dalle mura storiche, ad esempio all'inizio del percorso.

```
/// <summary>
/// Refresh meshes
/// </summary>
1 reference
private void Refresh()
{
    //patch half size (dx e sx)
    short patchSizeHalfX = (short)(patchesOnAxisX / 2);//range camera raggio e non diametro
    short patchSizeHalfZ = (short)(patchesOnAxisZ / 2);//range camera raggio e non diametro
    int minX = camGridPosX - patchSizeHalfX;//range camera min
    int maxX = camGridPosX + patchSizeHalfX;//range camera max
    int minZ = camGridPosZ - patchSizeHalfZ;//range camera min
    int maxZ = camGridPosZ + patchSizeHalfZ;//range camera max

    //cancella quelle lontane dalla camera (a dire il vero riciclale*)
    List<short[]> toRemove = new List<short[]>();
    foreach (short keyX in patches.Keys)
    {
        foreach (short keyZ in patches[keyX].Keys)
        {
            //if patch is not inside camera range anymore -> delete it (actually just hide it to reuse it later)
            if (!InsideBound(keyX, minX, maxX,keyZ,minZ,maxZ))
            {
                //rimuovi patch uscente dal range camera
                patches[keyX][keyZ].Disable();
                unusued.Push(patches[keyX][keyZ]);//riciclala dopo
                toRemove.Add(new short[] { keyX, keyZ });//metti in lista da rimuovere da dictionary
                //Debug.Log("To remove: " + keyX + "," + keyZ);
            }
            else//se invece ancora in range
            {
                //needs update check (might be collision, detailed mesh or nothing)
                updateList.Push(patches[keyX][keyZ]);
            }
        }
    }
    //rimuovi da dicionary
    for(int i=0; i< toRemove.Count; i++)
        patches[toRemove[i][0]].Remove(toRemove[i][1]);

    //crea patches mancanti
    for (short i = (short)minX; i <= maxX; i++)
    {
        for (short j = (short)minZ; j <= maxZ; j++)
        {
            if (!patches.ContainsKey(i) || !patches[i].ContainsKey(j))//manca quindi generala
                GeneratePatchData(i, j);
        }
    }
}
```

Capitolo 3

Creazione degli shaders

Uno *Shader* per *Unity Universal Render Pipeline (URP)* si scrive tramite il linguaggio *Shaderlab*. Uno *Shader* scritto in *Shaderlab* si compone di varie sezioni e strutture. Vengono di seguito analizzate quelle tipicamente utilizzate all'interno del progetto:

- La sezione *Properties* dove vengono dichiarate quali variabili sono visualizzabili e modificabili dall'*Inspector* di Unity, ovvero l'interfaccia grafica dell'engine. Notare che in questa sezione non si dichiara l'effettiva esistenza di una variabile nello *Shader*, ma l'effettiva dichiarazione avviene in una sezione successiva. Tra le proprietà di una variabile visualizzabile e modificabile nell'*Inspector* è possibile impostare valori di default della variabile e range minimi e massimi di valori che queste possono assumere.

```
Properties
{
  [MainColor] _BaseColor("Base Color", Color) = (1, 1, 1, 1)
  [MainTexture] _BaseMap("Base Texture", 2D) = "white" {}
  [Toggle(_NORMALMAP)] _NmTog("Enable Normal Map", Float) = 0
  [NoScaleOffset] _BumpMap("Normal Map", 2D) = "bump" {}
  _BumpScale("Bump Scale", Range(-3.0, 3.0)) = 1
}
```

- La sezione *SubShader* dove è possibile definire le variabili ed il loro tipo. Queste variabili possono essere settate per *shader* oppure globalmente. In questa sezione si possono anche inserire ulteriori dettagli come il *Queue order*, ovvero il livello di priorità di rendering che determinerà quali shaders vengono renderizzati prima e quali dopo. Si possono utilizzare inoltre dei *Toggle* per controllare delle *Shader Feature*, ovvero delle porzioni di codice che faranno parte esclusivamente di una *Shader Variant* specifica utilizzata nel caso in cui il toggle sia attivo.

```
SubShader
{
  Tags {
    "RenderPipeline" = "UniversalPipeline"
    "RenderType" = "Opaque"
    "Queue" = "Geometry"
  }

  float4 _BaseColor;
  TEXTURE2D(_BaseMap);  SAMPLER(sampler_BaseMap);

  #if _NORMALMAP //Shader Variant relativa al Toggle
    TEXTURE2D(_BumpMap);    SAMPLER(sampler_BumpMap);
    float _BumpScale;
  #endif
  ...
}
```

- il *ForwardLit Pass*, ovvero il codice che viene eseguito per renderizzare il colore della superficie della mesh. All'interno di un qualsiasi *pass* si definiscono prima di tutto parametri di rendering tra cui si notano il *Cull* che determina se lo *Shader* deve venire renderizzato nel lato frontale, posteriore o entrambi i lati di un triangolo, il parametro *ZWrite* che determina se e come lo *shader* deve scrivere nello *Z-Buffer*, ed il *Blend* che determina se e come uno *Shader* trasparente viene renderizzato.
  Successivamente all'interno di un qualsiasi *Pass* si possono importare funzioni, variabili o strutture da file esterni per riciclare porzioni di codice tra più shaders, o importare funzioni e porzioni di codice fornite da Unity.
  Successivamente si possono definire le strutture di input delle funzioni *Vertex* e *Fragment* e le funzioni stesse, nonchè qualsiasi altra variabile, struttura o funzione che venga utilizzata da questi stadi del rendering.
  Come di consueto nel funzionamento di uno *Shader*, il *Vertex stage* si occupa di tradurre la posizione dei vertici dall'*Object Space* al *Clip Space*, nonchè di applicare eventuali effetti di movimento dei vertici.
  Il *Fragment stage* invece si occupa di applicare il colore al frammento secondo la *texture* e/o altri parametri scriptati nella funzione corrispondente.

```
Pass
{
    Name "ForwardLit"
    Tags { "LightMode" = "UniversalForward" }

    #include "SomeFolder/SomeFile.hlsl"
    ...

    struct Attributes {
        float4 positionOS   : POSITION;
        #ifdef _NORMALMAP
            float4 tangentOS    : TANGENT;
        #endif
        ...
    };

    struct Varyings {
        float4 positionCS   : SV_POSITION;
        float2 uv   : TEXCOORD0;
        ...
    }

    Varyings LitPassVertex(Attributes IN) {
        Varyings OUT;
        ...
        return OUT;
    }

    half4 LitPassFragment(Varyings IN) : SV_Target{
        float4 color;
        ...
        return color;
    }

}
```

- il *ShadowCaster Pass* che funziona similmente al *ForwardLit pass*, ma si occupa di manipolare il rendering delle ombre proiettate sempre attraverso un *Vertex* e *Fragment* stage.
  È possibile utilizzare un *ShadowCaster Pass* di base fornito da Unity che non applica alcun effetto particolare importando la libreria apposita.

```
Pass {
    Name "ShadowCaster"
    Tags { "LightMode" = "ShadowCaster" }
    ...
}
```

- il *DepthOnly Pass* che funziona similmente al *ForwardLit pass*, ma si occupa di manipolare il Z-Depth, normalmente per l'applicazione di effetti di post processing, sempre attraverso ad un *Vertex* e *Fragment* stage.

```
Pass {
    Name "ShadowCaster"
    Tags { "LightMode" = "DepthOnly" }
    ...
}
```

È possibile utilizzare un *DepthOnly Pass* di base fornito da Unity che non applica alcun effetto particolare importando la libreria apposita.

- Altri eventuali *Pass* che non tratteremo in quanto non necessari allo scopo del progetto.

3.1 Terreno

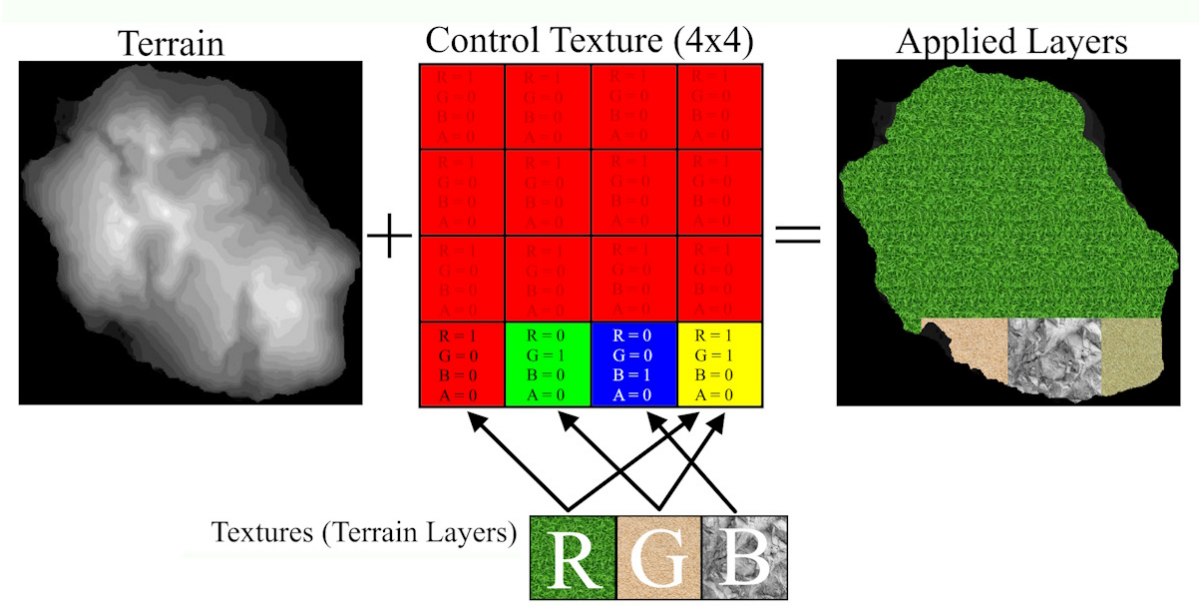
Lo *Shader* del terreno si compone di 3 effetti principali che possono venire chiamati *Texture Mixing*, *Far Textures* e *Snow Effect*.
Questi effetti vengono scritti su file esterni in modo che le funzioni possono venirne riciclate in successivi *Shaders*, come ad esempio *Shaders* di *props* che necessitano di avere un qualche tipo di blending o di fading sul terreno.

3.1.1 Texture Mixing

Con *Texture Mixing* si intende la pratica di prendere un certo set di *textures*, o meglio *Terrain Layers* che si possono usare su un terreno, ad esempio sabbia, erba, fango, roccia, le quali vengono mischiate nel terreno con diversa priorità in modo da poter creare un terreno complesso e vario. Si immagini ad esempio di avere un foglio di carta che rappresenta il nostro terreno, e alcuni pennarelli che rappresentano le possibili *textures* (layers): si può colorare il foglio in modi diversi utilizzando diversi pennarelli per colorare diverse zone (in alcuni casi usando un pennarello solo in un determinato punto, in altri sommando il colore di più pennarelli sovrapposti). Similmente il terreno sarà colorato decidendo di mostrare in zone diverse una o più *textures* mescolate tra loro, dove ogni *texture* applica una percentuale di contributo predefinita.

Per realizzare questa tecnica si utilizza una serie di *texture* aggiuntive chiamate *Control Textures*. Una *Control Texture* rappresenta l'intero terreno, da Nord a Sud e da Est ad Ovest. Ogni pixel della *Control texture* è quindi associata ad una specifica zona quadrata sovrapposta al terreno, e più grande è la risoluzione della *texture*, più piccole saranno le zone quadrate sul terreno corrispondenti ad ogni singolo pixel. Dunque si utilizza ogni pixel della *Control Textu-*

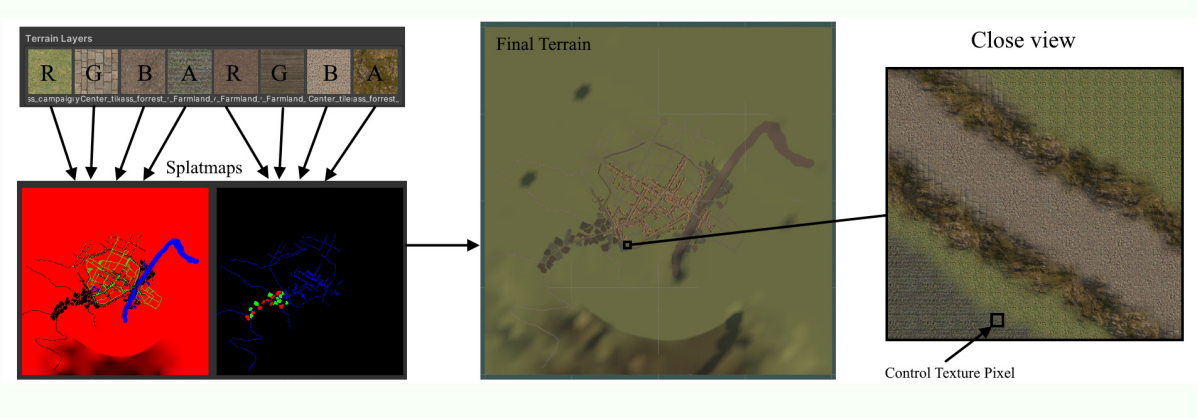
re per decidere quale *texture* applicare al terreno in quella specifica zona quadrata. Per fare ciò ad ogni canale del colore del pixel della *Control Texture* viene associata una diversa *texture* del terreno utilizzabile. Ad esempio al canale R la *texture* "sabbia", al canale G la *texture* "erba", al canale B la *texture* fango ed al canale A la *texture* "roccia". In base al valore di ogni canale (0-256) viene deciso il contributo di ogni singola *texture* in quella zona quadrata del terreno.



Nel caso si vogliano utilizzare più di 4 *texture* è sufficiente aggiungere ulteriori *Control Textures*, ognuna delle quali andrà a mappare altri ulteriori 4 *texture*.
In questo progetto di tesi viene sviluppato uno *Shader* che prende in considerazione due *Control Textures*, e dunque un numero di canali sufficiente a mischiare 8 *textures*. Il mixing a livello di codice dello *Shader* avviene nel seguente modo.

```
half3 LayersCombiner(half3 layer0, half3 layer1, half3 layer2, half3 layer3, float4 splat_control)
{
    half3 col = splat_control.r * layer0;
    col += splat_control.g * layer1;
    col += splat_control.b * layer2;
    col += splat_control.a * layer3;
    return col;
}
```

Similmente si esegue la stessa procedura per Normal Maps ed altre eventuali *textures* che si intende utilizzare all'interno del *Fragment Shader*.
Di seguito viene visualizzato l'effetto finale.



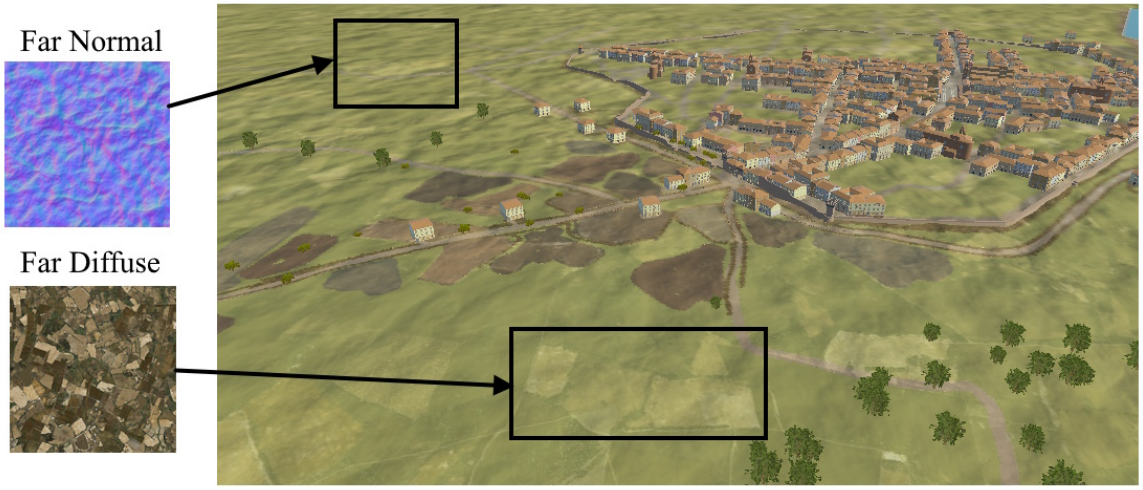
3.1.2 Far Textures

Il texure mixing, quando la telecamera è posta a distanza notevole, rivela il chiaro effetto di *Tiling* delle *textures*, e produce effetti troppo piatti ed uniformi quando osservato da distanze ancora maggiori.
Per fare fronte a questi problemi si implementa una soluzione che si può definire *Far Textures*, la quale consiste nel mixare il risultato precedentemente ottenuto con ulteriori *textures* applicate con una scala molto maggiore in modo che queste siano ben visibili a distanza. Inoltre si sfrutta la distanza tra la telecamera ed il terreno per calcolare l'intensità con cui le *Far Textures* devono essere mescolate, in modo che non si perda la qualità visiva ottenuta quando si è invece vicini al terreno.

Per la realizzazione di tale effetto si mischierà il terreno a distanza con una *Diffuse Texture* e una *Normal Map*.
Il codice per realizzare tale effetto si può riassumere nel seguente:

```
float4 finalTerrainAlbedo = ...
float4 finalTerrainNormal = ...
...
//Implementazione distant noise
//altezza minima per applicare effetto
float heigthMultiplier = clamp(IN.positionWS.y- _AntiTileStartHeight, 0, 1);

//diffuse e normal textures
float4 distanceTex = SAMPLE_TEXTURE2D(_TerrainNoiseTex, sampler_BaseMap, globalUVs_distant * .15);
float4 distanceNm = SAMPLE_TEXTURE2D(_TerrainNoiseNormal, sampler_BaseMap, globalUVs_distant*.3);

//applicazione effetto
finalTerrainAlbedo = lerp(finalTerrainAlbedo, distanceTex, camDistanceLimited * heigthMultiplier);
finalTerrainNormal = lerp(finalTerrainNormal, distanceNm, camDistance * .9);
```

Si ottiene il seguente risultato:



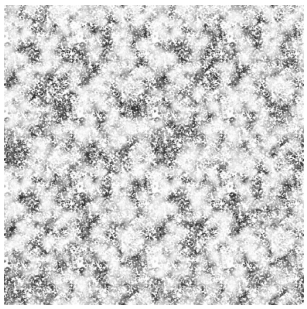
3.1.3 Snow

Altro effetto sul terreno implementato in questo progetto di tesi è la neve procedurale. Similmente al concetto dei *Terrain layers* si acquisisce una *texture* che rappresenta la neve e la si applica secondo una certa maschera. In questo caso non è necessaria una complessa *Control texture*, ma un singolo canale che definisce se la neve deve venire renderizzata in un determinato punto e con quale intensità.
Si intende inoltre realizzare un sistema che preso in input un valore [0-1], venga aggiunto al terreno un certo quantitativo di neve.

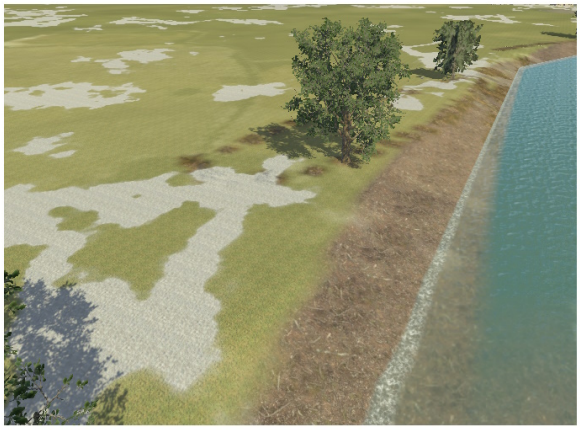
L'implementazione consiste nell'acquisizione di una *Noise Texture* in scala di grigi, alla quale vengono applicate modifiche di luminosità e contrasto in base al parametro di intensità di neve, in modo da far crescere l'intensità e la diffusione della *Noise* che si utilizza per decidere quanta neve mostrare e in quali posizioni.
Viene inoltre implementato l'effetto di riduzione dell'intensità, fino alla completa eliminazione della neve dalla maschera in base ai parametri di inclinazione del terreno (le neve tende a non attaccarsi ed a scivolare via a ad alte inclinazioni) e di altitudine (si può voler mostrare la neve solo a certe altitudini specifiche, e sicuramente non sotto al livello del mare).
Di seguito il codice per realizzare questo effetto:

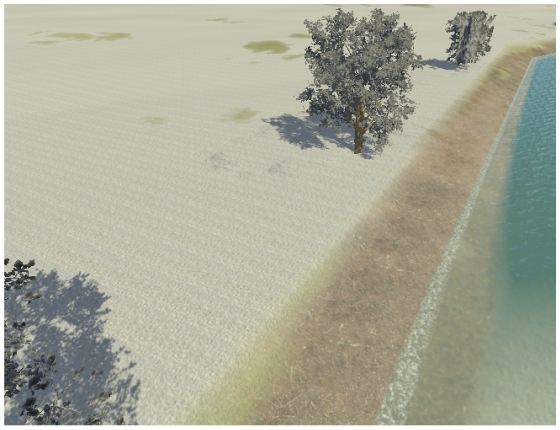
```
half4 ApplySnow(half3 positionWS, half3 normalWS, half4 color, half snowAngle)
{
    if (_GlobalSnowIntensity <= 0)
        return color;

    float _SnowFading = .5;
    float _WaterLevel = 0; //no neve sotto livello del mare
    float _SnowTextureTiling = 4;
    half3 _SnowColor = half3(.65, .65, .65);
    float2 gloabalUVs = GetGlobalUVs(positionWS);

    //float on noise map
    float snowMapValue = GetSnowMask(gloabalUVs);

    //snow on slope
    half snowMultiplierOnSlope =
        pow(
            clamp(dot(normalWS, float3(0, 1, 0)), 0, 1),//multiplier on slope
            (1 - _SnowFading) * snowAngle//snow fading intensity value
        );

    //snow on height
    half snowMultiplierOnHeight =
    clamp((positionWS.y + _WaterLevel) / (1 - (_SnowFading * .9)), 0, 1);

    //final snow mask value
    //moltiplica tutti i valori calcolati precedentemente per
    //sapere dove dipingere la neve
    half finalSnowMaskValue = clamp(snowMultiplierOnSlope * snowMapValue * snowMultiplierOnHeight,0,1);

    //outputs
    float3 snowTex =
    tex2D(_GlobalSnowTexture, gloabalUVs/ _SnowTextureTiling).rgb * _SnowColor;
    return lerp(color, snowTex, finalSnowMaskValue);
}
```

La *Noise texture* utilizzata per applicare l'effetto è la seguente.



Di seguito si mostra l'effetto a due diversi livelli di intensità di neve. Si noti come la neve viene generata similmente alla *Noise Texture* e come essa viene scartata in prossimità dell'argine del fiume in quanto quest'ultimo risulta in un'alta inclinazione del terreno.





Similmente a come si applica la neve, è possibile anche applicare un semplice effetto "bagnato" nel caso in cui si imposti il clima su *Pioggia*. Questo effetto fa risultare la superficie più riflettente aumentando gli effetti *Metallic* e *Smoothness* (Descritti successivamente nel capitolo *PBR Workflow*).

3.2 Superfici generali

Per questo progetto di tesi, è stato implementato un *Shader* generale che viene impiegato su tutte le superfici opache statiche che non necessitano di una ottimizzazione particolare e che possono opzionalmente implementare una serie di effetti utilizzando delle specifiche *Shader Features*. Il caso più tipico dell'utilizzo di questo *Shader* è nelle superfici degli edifici, le superfici dei prop, e i tronchi degli alberi, tuttavia si può considerare questo **lo shader principale del progetto** in quanto è quello che implementa il maggior numero di funzionalità complesse ed è lo *shader* applicato ai casi più generali della scena.
Di seguito vengono descritti gli effetti implementati in questo *Shader*.

3.2.1 Terrain Blendings

Gli effetti discussi per lo *Shader* del terreno vengono richiamati e opzionalmente reimplementati tramite una *Shader Feature*, che si ricorda essere una variante opzionale dello *shader* che implementa un determinato effetto solamente se questo viene effettivamente utilizzato dal materiale.

Per quanto riguarda l'effetto della neve questo può venire abilitato e impostato per essere visibile fino a certe angolazioni specifiche. Questa *Shader Feature* viene usata principalmente nei materiali che rappresentano i tetti degli edifici, dove ci si aspetta che la neve si depositi similmente a come avviene nel terreno.

Per quanto concerne l'uso delle *textures* del terreno (Comprendenti sia i *Terrain Layers* vincolati dalla *Control Texture*, sia le *Far Textures*), questo effetto può venire abilitato in due

varianti:

- **From Flatness**, ovvero l'effetto viene renderizzato in base all'inclinazione della superficie; più questa è piana e più si visualizza il rendering del terreno. Al contrario per una superficie orientata verticalmente le *texture PBR* della mesh saranno visualizzate secondo la tecnica che viene descritta in seguito in questo paragrafo.
- **From Mask**, ovvero l'effetto del terreno verrà visualizzato in prossimità di una maschera in scala di grigi applicata tramite le *UV* della mesh a cui il materiale è applicato. In questo modo è possibile controllare più specificamente dove si vuole renderizzare le *textures* del terreno e dove quelle della mesh.

Questi effetti normalmente possono venire utilizzati per realizzare *props* che mischiano *textures* specifiche della mesh con quelle del terreno come rocce, scogliere, o calanchi.
Si mostra di seguito l'effetto applicato al materiale dei tetti degli edifici, si noti tuttavia che questo non è il caso tipico di utilizzo.

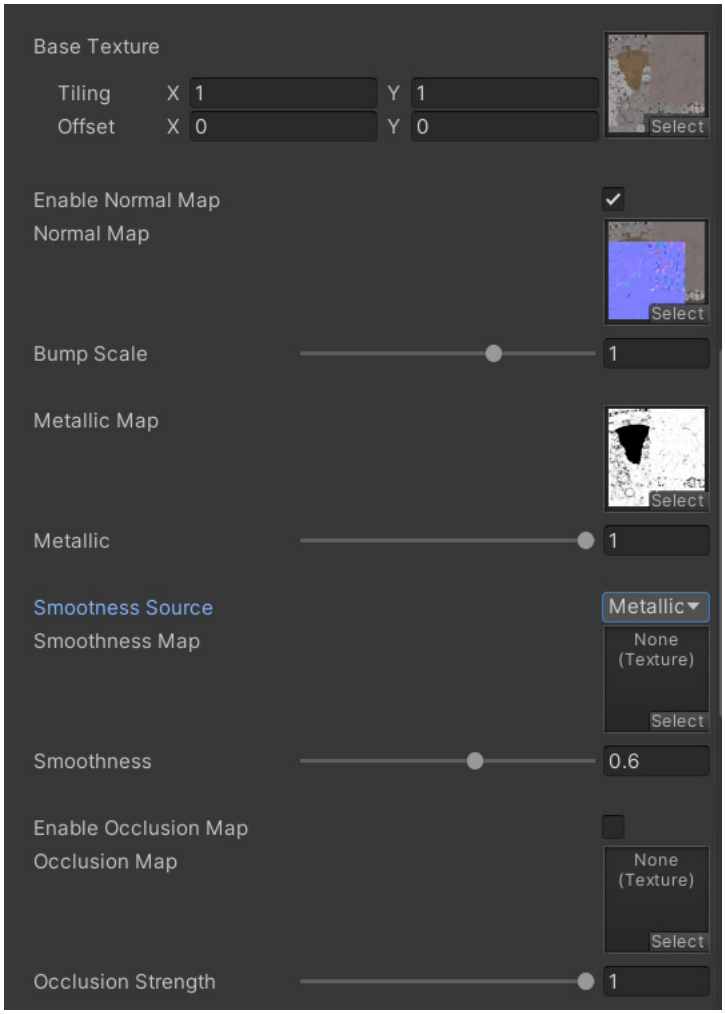
3.2.2 PBR Workflow

Lo *Shader* delle superfici generali, oltre a presentare una serie di effetti quali l'appena descritto *Terrain Blending*, presenta anche un livello di *Shading PBR* che sfrutta le seguenti *Textures Map*:

- **Diffuse Map** utilizzata per determinare il colore base della superficie.
- **Normal Map** (Opzionale) utilizzata per ottenere il vettore normale dei dettagli della superficie durante il calcolo dell'illuminazione.
- **Metallic Map** (Opzionale) utilizzata per calcolare la quantità di riflesso desiderata sulla superficie. Mappata sul canale R.

- **Smoothness Map** (Opzionale) utilizzata per determinare il livello di perfezione delle riflessione, simulando una superficie più liscia o più ruvida. Il livello della smoothness è mappato sul canale R della *texture* dedicata, ma può anche venire acquisito alternativamente dal canale *Alpha* della *Metallic Map* (*Metallic-Smoothness Workflow*) o utilizzare lo stesso canale R della *Metallic Map* se si intende risparmiare memoria.

- **Occlusion Map** (Opzionale) utilizzata per determinare la quantità massima di illuminazione ambientale che una superficie può ricevere, e quindi enfatizzare le ombre in punti che normalmente sarebbero difficili per la luce da raggiungere, simulando un'ombreggiatura più realistica e meno piatta. È mappata sul canale R della *texture* dedicata, ma per risparmiare memoria si può mappare e dunque leggere dal canale G della *Metallic Map*.



Acquisite tutte le mappe ed i parametri necessari si procede all'effettiva parte di shading. Si può sfruttare all'interno del *fragment shader* la funzione *UniversalFragmentPBR* includa nel file *Lighting.hlsl* [9] builtin in *Unity URP* includibile tramite la linea di codice

```
#include "Packages/com.unity.render-pipelines.universal/ShaderLibrary/Lighting.hlsl
```

per ottenere il colore del frammento dopo aver applicato uno shading ispirato alle tecniche di *Blinn-Phong*. Si può visualizzare come tale funzione è implementata direttamente nella *repository GIT di Unity* [9] ed eventualmente si può scaricare e modificare il file per ottenere varianti dello stesso effetto di *Shading*. Naturalmente è anche possibile utilizzare le mappe acquisite per generare uno *shading* completamente personalizzato, cosa che si mostrerà nello specifico nei capitoli dedicati allo *shading* della vegetazione.

Lo *Shader* implementato è di tipologia *Opaque*, ovvero non permette effetti di trasparenza, per una migliore performance. Tuttavia è possibile utilizzare il *clipping* per decidere di non renderizzare un frammento e quindi ottenere dei "buchi" nella superficie che possono in alcuni casi risolvere la necessità di un effetto di trasparenza effettivo.
È possibile ad ogni modo reimplementare lo stesso *Shader* in versione *Transparency* applicando poche modifiche allo shader:

```
SubShader
{
    Tags {
        "RenderPipeline" = "UniversalPipeline"
        "RenderType" = "Transparent"
        "Queue" = "Transparent"
    }
    ...
    Pass
    {
        ...
        Blend SrcAlpha OneMinusSrcAlpha
        ...
    }
    ...
}
```

Bisogna naturalmente ricordarsi in questo caso di manipolare correttamente il valore *Alpha* della funzione *fragment*.

3.2.3 LOD Transitions

L'ultimo effetto notevole implementato nello *Shader* generico delle superfici, è l'effetto per le transizioni *LOD*. Questo effetto intende implementare una breve transizione per il passaggio

da un modello ad alto numero di poligoni verso uno a minor numero di poligoni quando ci si allontana da un oggetto. Sostituire il modello con uno a minore *poly count* senza nessun effetto di transizione risulterebbe infatti in un fastidioso effetto comunemente chiamato "*popping*".

Per implementare una transizione semplice ma efficace, si sfrutta un effetto di *dithering incrementale* dove nel modello che si intende fare "scomparire" si applicano progressivamente dei "buchi" (utilizzando il *diffuse clipping*) fino a fare scomparire il modello completamente. Contemporaneamente, nelle posizioni in cui il dithering smette di renderizzare il primo modello, si inizia invece a renderizzare il secondo, ovvero quello che deve comparire, fino a chè questo non sia del tutto renderizzato. A questo punto si rimuove dalla memoria il primo modello.

Il componente di Unity *LOD group*, che si occupa di effettuare lo swap nella memoria tra il modello *High poly* e quello *Low poly*, fornisce una variabile *unity_LODFade.x* che ritorna un valore compreso tra -1 ed 0 durante il *fade-in* ed un valore compreso tra 0 e 1 durante il *fade-out*. Possiamo dunque utilizzare questa variabile per rappresentare il nostro effetto di *Dithering Cross Fading*.

Si implementa innanzitutto una funzione di *dithering*.

```
float GetDither(float2 ScreenPosition)
{
    float2 uv = (ScreenPosition.xy ) * _ScreenParams.xy;
    float DITHER_THRESHOLDS[16] =
    {
        1.0 / 17.0,  9.0 / 17.0,  3.0 / 17.0, 11.0 / 17.0,
        13.0 / 17.0,  5.0 / 17.0, 15.0 / 17.0,  7.0 / 17.0,
        4.0 / 17.0, 12.0 / 17.0,  2.0 / 17.0, 10.0 / 17.0,
        16.0 / 17.0,  8.0 / 17.0, 14.0 / 17.0,  6.0 / 17.0
    };
    uint index = (uint(uv.x) % 4) * 4 + uint(uv.y) % 4;
    return DITHER_THRESHOLDS[index];
}
```

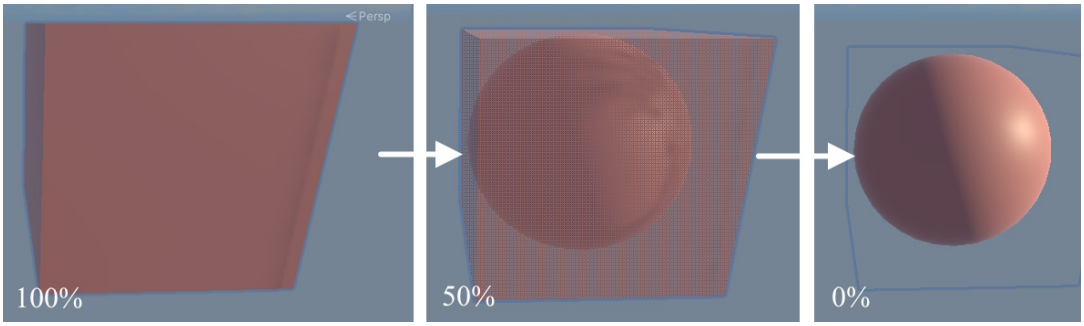
Di seguito l'implementazione di una funzione che dato il valore *Alpha* originale del frammento, il valore massimo di *Clip*, e la posizione a schermo del frammento, restituisce il nuovo valore di *Alpha* per il frammento.

```
//dither crossfade in both directions
float DitherCrossFade(float AlphaChannel, float AlphaClip, float2 ScreenPosition)
{
    if (unity_LODFade.x != 0)//in transizione
    {
        if (AlphaChannel < AlphaClip) //clippa trasparenza
            return 0;
        else if (unity_LODFade.x < 0) //comparsa
            return ((unity_LODFade.x + AlphaClip) + GetDither(ScreenPosition));
        else//scomparsa allontanamento
            return ((unity_LODFade.x + AlphaClip) - GetDither(ScreenPosition));
    }
    else //non in transizione
        return AlphaChannel;
}
```

Un esempio d'uso di questa funzione è il seguente

```
float clipValue = SAMPLE_TEXTURE2D(_BaseMap, sampler_BaseMap, i.uv).a;
clipValue *= DitherCrossFade(1, _Cutoff, WorldToScreenPos(i.positionWS));
clip(clipValue - _Cutoff);
```

Si mostra infine un esempio di risultato atteso in tre fasi dall'allontanamento di un oggetto di forma cubica che si trasforma in un oggetto di forma sferica gradualmente tramite l'applicazione di un effetto *Dithering*.



3.2.4 Fog

Utilizzando ulteiori funzioni messe a disposizioni da *Unity 3D*, è possibile, prima di uscire dalla *fragment function*, applicare un effetto di nebbia sul rendering. Questo effetto è a dir la verità piuttosto semplice da implementare, tuttavia Unity implementa una funzione di facile utilizzo per calcolare il colore finale del frammento dopo aver applicato la *fog*:

```
color.rgb = MixFog(color.rgb, inputData.fogCoord);
```

Il risultato è una semplice interpolazione lineare verso un colore specifico che rappresenta la nebbia, a partire da una distanza specifica.

3.3 Vegetazione

Altri *Shaders* interessanti sono quelli relativi alla vegetazione.
Si differenziano notevolmente da uno *Shader PBR*, ed implementano la gestione di ombreggiature in modo atipico per dare una impressione di traslucenza e di superfici sottili come possono essere le foglie e i fili d'erba, ed implementano un *displacement* sui vertici per creare effetti di movimenti con il vento. Inoltre la vegetazione può risultare talvolta piuttosto densa e quindi è richiesta una particolare attenzione alla performance.

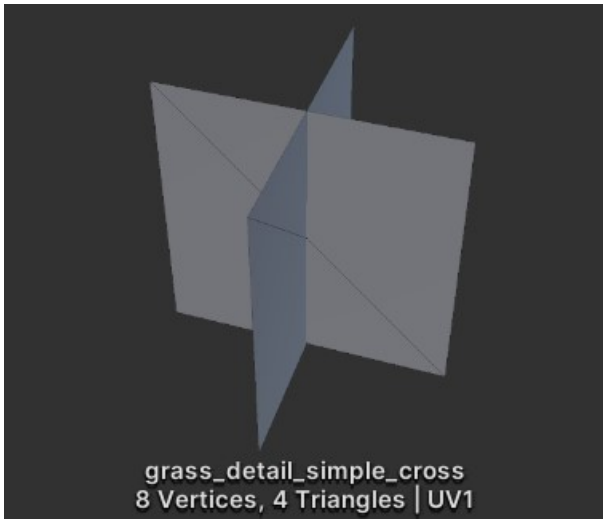
3.3.1 Erba

Lo *shader* dell'erba innanzitutto è basato sul concetto di *GPU Instancing*, ovvero deve essere compatibile con le tecniche di renderizzazione di molteplici copie dello stesso modello sfruttando una sola *draw call*. Per implementare ciò bisogna utilizzare i seguenti comandi nel codice dello *Shader*:

```
// Strutture input vertex / fragent
struct Attributes {
    ...
    // per accedere alle proprietà istanziate
    UNITY_VERTEX_INPUT_INSTANCE_ID
};
struct Varyings {
    ...
    // per accedere alle proprietà istanziate
    UNITY_VERTEX_INPUT_INSTANCE_ID
};
...
Varyings LitPassVertex(Attributes IN) {
    Varyings OUT;
    // prepara i dati per instancing
    UNITY_SETUP_INSTANCE_ID(IN);
    UNITY_TRANSFER_INSTANCE_ID(IN, OUT);
    ...
}
half4 LitPassFragment(Varyings IN) : SV_Target
{
    // prepara i dati per instancing
    UNITY_SETUP_INSTANCE_ID(IN);
    ...
}
```

Il *Terrain system* di Unity si occuperà quindi di renderizzare le *patch* dell'erba con il sistema di *GPU Instancing*. Similmente al sistema della *Control Texture* per lo *shader* del terreno, si utilizza una *texture* per comunicare al *Terrain system* dove si intende renderizzare certi tipi di *patch* d'erba. Non c'è dunque un controllo preciso su dove posizionare i singoli modelli 3D che rappresentano l'erba, ma piuttosto le zone generali dove la si desidera.

L'erba viene renderizzata tramite modelli semplici, come piani su cui vengono applicate *textures* che rappresentano una vista laterale di un gruppo di fili d'erba e rametti. Ciò permette di mantenere una geometria semplice e leggera da renderizzare.



Tuttavia è importante notare che se renderizzata con tecniche tipiche, la faccia del piano che si trova rivolta dal lato opposto alla direzione della luce del sole sarà completamente in ombra. Questo effetto deteriora completamente l'immersività dell'applicazione 3D ed è quindi necessario sfruttare tecniche di *rendering* differenti. Si sfrutta l'occasione per abbandonare l'idea di utilizzare in questo caso un *Rendering PBR* in questo caso in quanto richiede una grande quantità di calcoli a scapito delle performance, per creare uno shading personalizzato che possa minimizzare il costo sulla GPU.
L'obiettivo, quindi, è quello di realizzare uno shading più piatto e traslucente, che non faccia notare particolare differenza di illuminazione della superficie del piano dell'erba se rivolto a diverse angolazioni rispetto alla luce del sole, ma che viene illuminato in base a quanto "il sole è alto nel cielo", ovvero in base a quanto la luce solare è perpendicolare o meno alla superficie del terreno.

```
// fragment shader
...
half4 albedo = SAMPLE_TEXTURE2D(_BaseMap, sampler_BaseMap, IN.uv);
clip(albedo.a - _Cutoff);// Alpha Clipping

// base color
float3 color = albedo.rgb * _BaseColor.rgb * IN.color.rgb * _Lightness;

// translucency
float3 cam_to_grass_vect = normalize(IN.vertexWorldPosAndYMult.xyz - _WorldSpaceCameraPos);
// vettore camera-grass da usare per
// calcolare dot product con direzione mainlight
float translucencyValue = pow(
    clamp(dot(mainLight.direction, cam_to_grass_vect), 0, 1),
    1 + _TranslucencyScattering
  );
float translucencyMult = 1 + translucencyValue * _TranslucencyIntensity;
color.rgb *= translucencyMult;
```

Infine, viene implementato un effetto di movimento dei vertici del piano per simulare il movimento causato dal vento. Di seguito l'implementazione.

```
// vertex shaderlab
...
float yScale= length(float3(
      UNITY_MATRIX_M[0].y,
      UNITY_MATRIX_M[1].y,
      UNITY_MATRIX_M[2].y
    ));
// moltiplicatore Y (ondolamento solo nella parte alta)
OUT.vertexWorldPosAndYMult.w = IN.positionOS.y;
IN.positionOS += float4(sin((IN.positionOS.x + IN.positionOS.y) * 4 + _Time.y * _WindSpeed) * _WindShift* OUT.vertexWorldPosAndYMult.w, 0, 0,0);
```

Si implementa infine un effetto di "neve", che imbianca la scena in base al valore moltiplicativo della neve, applicando cambiamenti di luminosità e contrasto del colore nel *fragment shader*, col fine di simulare una vegetazione più secca o su cui si è posata la neve.

3.3.2 Foglie

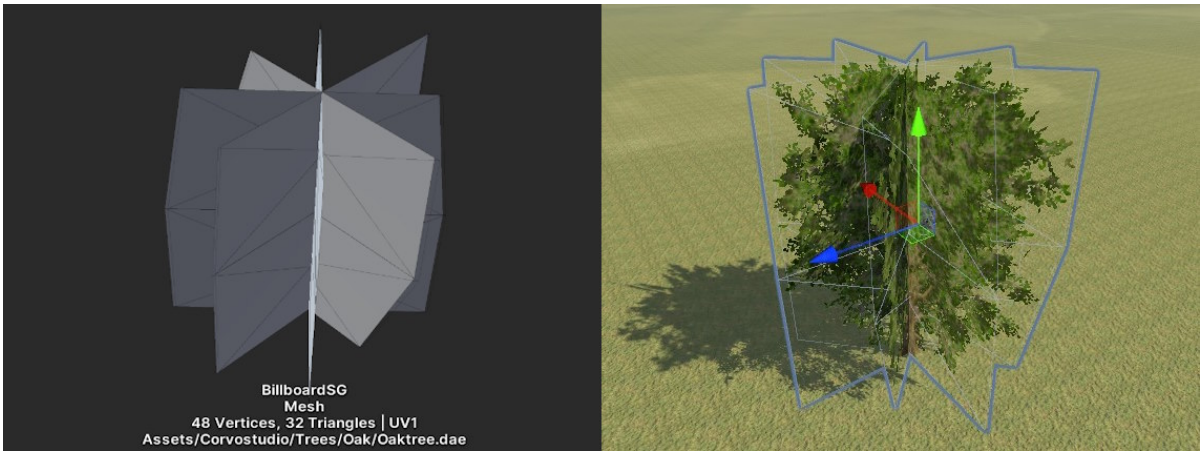
L'idea alla base dello *Shader* delle foglie è simile a quello dell'erba, e funziona sempre tramite modelli 3D di piani contenenti una *texture* che rappresenta una serie di foglie e rami, tuttavia invece che funzionare tramite *GPU Instancing*, questi modelli fanno parte di una mesh di una pianta o un albero posizionata manualmente nella scena 3D, normalmente dispiegati a *clusters* di piani localizzati nella zona della chioma della pianta.

Lo *Shader* delle foglie eredita sia gli effetti di movimento tramite il *vertex shader* per simulare il vento, sia gli effetti di traslucenza e neve applicati tramite il *fragment shader*. Implementa inoltre il sistema di *LOD transition* similmente a come implementato nello *Shader* generico delle superfici.

3.3.3 LOD Alberi e Piante

Il modello finale di *LOD* di una pianta, tipicamente è simile a quello di una *patch* d'erba, ovvero un piano su cui viene applicata una *texture* con vista laterale della pianta per minimizzare l'uso di geometria. Questo modello dunque sfrutta uno *shader* simile a quelli appena presentati

che implementa effetti per traslucenza e neve molto semplici. Dato il modello di un albero può essere molto presente in una scena, specialmente quando un'ambientazione presenta foreste, è molto importante assicurarsi che questo sia uno *shader* particolarmente performante. In esso, infatti, non vengono implementati effetti di nessun altro tipo.



3.4 Decals

Le cosiddette *decals* (traducibili in "decalcomanie") sono effetti normalmente utilizzati per applicare *textures* ed immagini ad altri modelli in modo flessibile e dinamico. Si possono immaginare come adesivi applicati in tempo reale, e costituiscono una sfida sia dal punto di vista della realizzazione che della performance. Si implementano *decals* per realizzare all'interno del progetto strade, pozzanghere ed impronte di passi, tuttavia altri campi di applicazione possono essere manifesti, macchie, effetti di usura, fori di proiettili, effetti di impatto, piccole buche e così via.

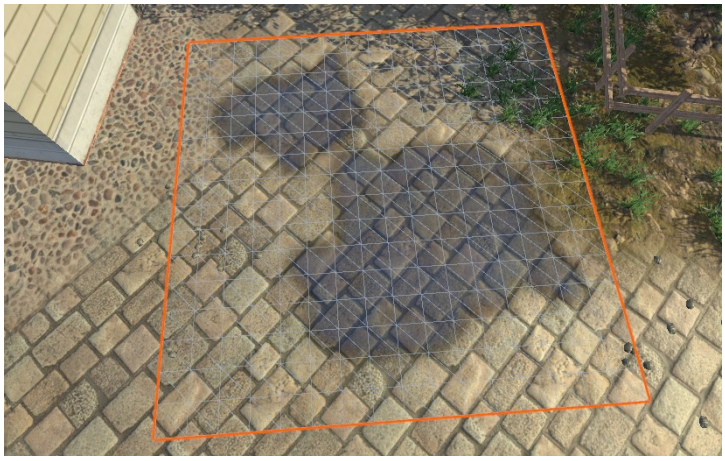
3.4.1 Decals dinamiche

Un primo modo di implementare un effetto *decal* è tramite una geometria che si adatta alle mesh che si trovano sotto di essa. Un sistema del genere è un compromesso con determinati costi e benefici. Questo sistema viene implementato tramite una mesh a forma di griglia che trova il modo di adattarsi ad altre superfici che implementano *Colliders* utilizzando un *Raycast* per ogni vertice della griglia. Questo sistema necessita quindi di una fase di *baking* dove si eseguono calcoli costosi, che vanno rieseguiti nuovamente se si dovesse cambiare forma o posizione della *decal*. Tuttavia una volta applicata, il costo in termini di performance è relativamente basso e dipende unicamente dalla risoluzione della griglia utilizzata e dallo *shader* applicato ad essa, che può essere uno *shader* di qualsiasi tipo, senza particolari limitazioni. Un altra pecca di questo sistema risiede nel fatto che per poter essere applicato perfettamente è necessario selezionare una griglia di risoluzione adeguata, altrimenti si rischia che la *decal* non si adatti in maniera ottimale a superfici non piane. Un'altro beneficio dell'uso di questo sistema è invece la possibilità di modificare la forma della griglia per scopi specifici. Ad esempio, se la *decal* rappresenta un pezzo di strada dritta, è possibile generare la griglia incurvata per poter ottenere una curva della strada senza dover utilizzare ulteriori *textures* o *materials*.

Si mostrano di seguito alcuni esempi di applicazione di questo tipo di *decals*.

Di seguito vengono presentati alcuni estratti di codice con cui si genera la griglia standard utilizzando due semplici cicli che permettono lo spostamento orizzontale e verticale della posizione calcolata per la generazione dei singoli vertici.

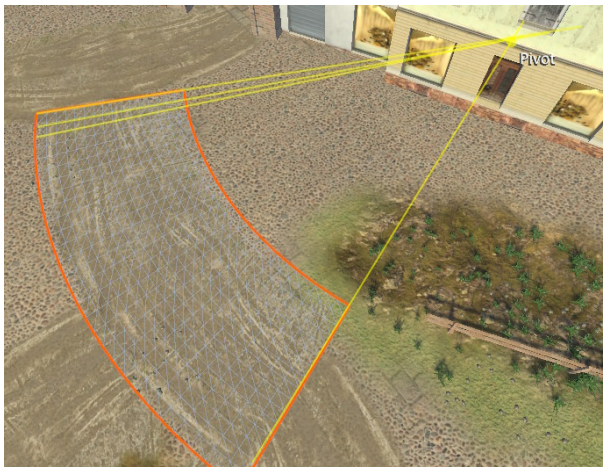
```
for (int i = 0; i < subdivisionsX; i++)
{
  for (int j = 0; j < subdivisionsZ; j++)
  {
    ...
    rayPos = transform.position
    - transform.right * width / 2 + transform.right * width * ((float)i / (subdivisionsX - 1))
    - transform.forward * lenght / 2 + transform.forward * lenght * ((float)j / (subdivisionsZ - 1))
    + transform.up * height / 2;
    ...
    if (Physics.Raycast(rayPos, -transform.up, out RaycastHit rh, height, mask))
    ...
  }
}
```



Di seguito il concetto dietro alla generazione della griglia curvata, il quale sfrutta una po-

sizione "pivot" per generare una serie di raggi su cui posizionare i punti della griglia.

```
for (int i = 0; i < subdivisionsX; i++)
{
  for (int j = 0; j < subdivisionsZ; j++)
  {
    ...
    Vector3 localPivot = Vector3.right * shiftWidth;
    float Zmult = ((float)j / (subdivisionsZ - 1));
    float localAngleOnArch = turnAngle * archLenghtMult * Mathf.Deg2Rad * Zmult;
    ...
    rayPos = globalArchpos - archToCenter * distanceFromCenter
    + transform.up * height / 2
    - transform.right * (width+ shiftWidth*2) //recenter X
    - transform.forward*(lenght/2); //recenter Z
    ...
    if (Physics.Raycast(rayPos, -transform.up, out RaycastHit rh, height, mask))
    ...
  }
}
```



3.4.2 Decals statiche

Un altro metodo tipico di implementazione delle *decals*, è tramite uno *Shaders* che effettua una proiezione sul valore del *depth buffer*, colorando direttamente nella *fragment function* i frammenti traslati nella posizione in cui le superfici dei modelli si intersecano con la normale della mesh della *decal*.

Questa tecnica permette di avere decals relativamente a basso costo che vengono aggiornate ogni frame, e ciò rende possibile che vengano spostate senza compromettere la performance o dover rieseguire calcoli costosi per riadattarle alle superfici posteriori. Queste *decals* tuttavia offrono poche possibilità di modifica e distorsione, ed inoltre richiedono che lo *Shader* sia scritto appositamente per poter ottenere questo effetto. Inoltre queste decal non utilizzando

Colliders per la proiezione possono venire applicate su qualsiasi oggetto, il che può essere un vantaggio o uno svantaggio a seconda della necessità, ad esempio non permettono di venire proiettate su un certo tipo di mesh specifiche.

Si mostra di seguito un esempio di applicazione di questo tipo di *decals*.

Segue un estratto di codice del *fragment shader* con cui si ottiene questo specifico effetto *Decal*, ed in particolare come si trasformano le coordinate UV originali nelle coordinate UV proiettate secondo il valore della *depth texture*.

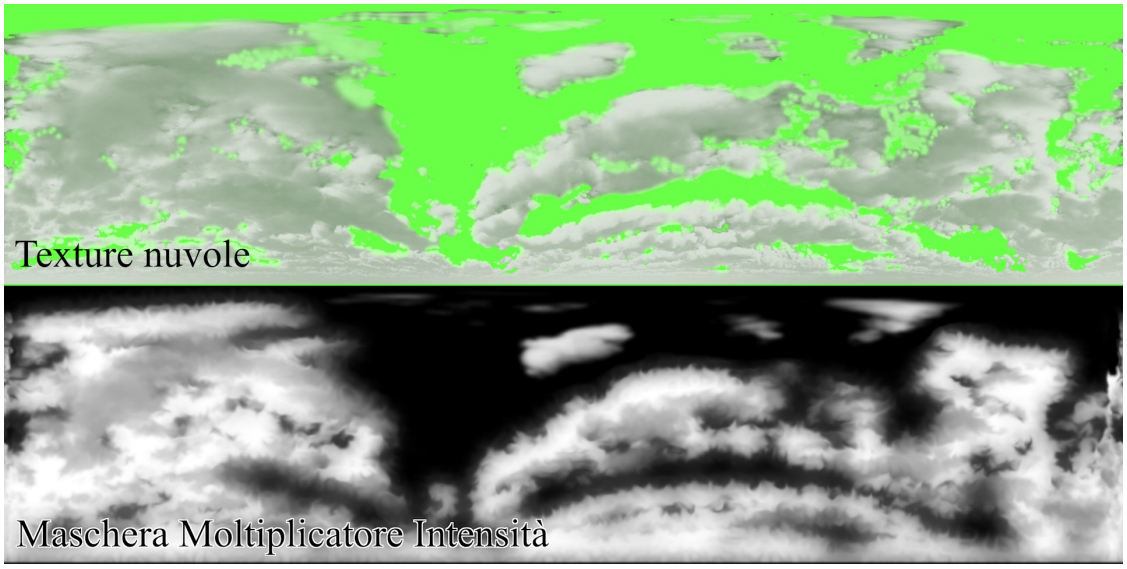
```
float depth = SampleSceneDepth(UV);
float3 wpos = ComputeWorldSpacePosition(UV, depth, UNITY_MATRIX_I_VP);
float3 opos = mul(unity_WorldToObject, float4(wpos, 1)).xyz;

float2 uvMain = (opos.xz + 0.5);
half4 tex = tex2D(_MainTex, uvMain);
```

3.5 Cielo

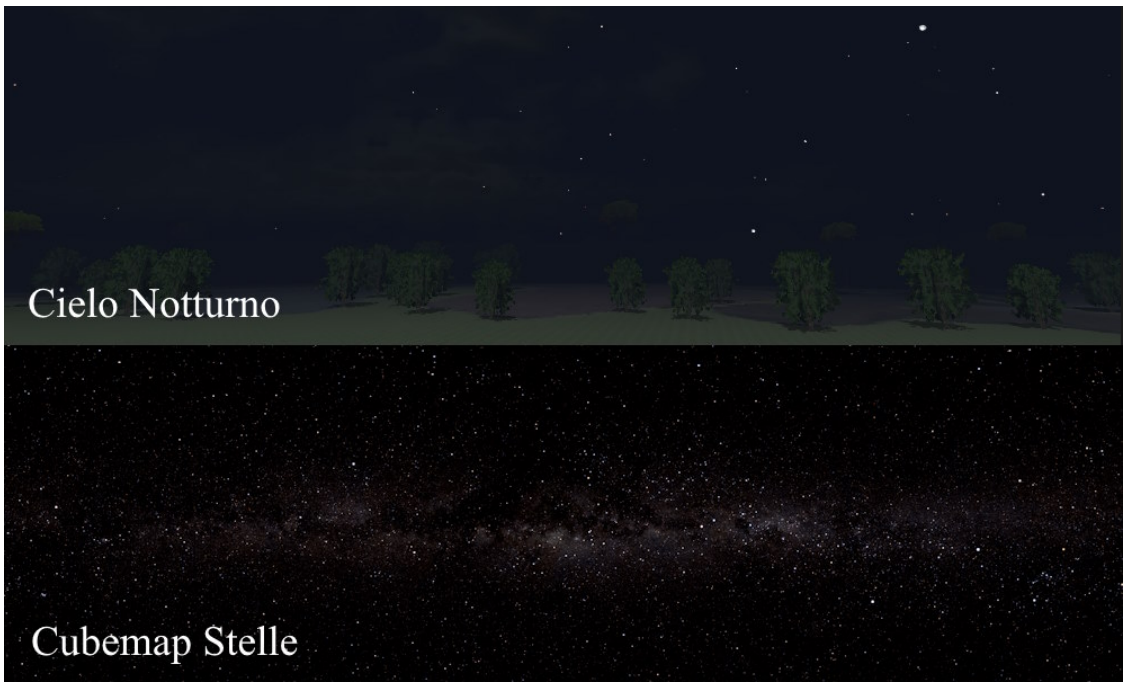
Come è tipico nelle applicazioni di grafica 3D, il cielo è rappresentato tramite uno *Skybox*, ovvero un modello 3D cubico che viene renderizzato prima di ogni altra mesh (così da rimanere nel *background*) e che segue la telecamera in modo da dare l'impressione di essere illimitato. Per realizzare lo *Shader* si mischiano una serie di tecniche che uniscono la tecnica cosiddetta delle *Cubemap* con effetti più dinamici. Il cielo può infatti variare in modo graduale dal giorno alla notte e può essere caratterizzato da nuvole dinamiche, effetti di nebbia ed altro ancora.

Lo *Shader* del cielo progettato per questo progetto di tesi implementa una serie di effetti, quali:

- **Sky color**: Colore di sfondo del cielo, realizzato tramite un colore blu uniforme che si schiarisce leggermente nella parte più bassa del cielo, in prossimità con l'orizzonte.
- **Clouds**: Si implementano le nuvole tramite una *texture Cubemap* con trasparenza che si applica sopra al colore del cielo. Tramite una maschera bianco-nera ed un valore moltiplicatore si determina la presenza o meno dell'effetto delle nuvole. La cubemap delle nuvole viene inoltre shiftata lentamente in una direzione tramite la variabile temporale per dare l'impressione del movimento causato dal vento. A valori alti del moltiplicatore delle nuvole si fa inoltre tendere il colore di sfondo del cielo verso tonalità più grigie.



- **Night**: Tramite un valore moltiplicativo [0-1] si implementa una transizione che scurisce il risultato del colore del cielo e delle nuvole per realizzare un effetto di cielo notturno. Si aggiunge un'ulteriore immagine che rappresenta le stelle al risultato del cielo.



- **Sun**: Si prende un punto sulla UV map calcolato a partire dalla proiezione della posizione della luce nella scena che rappresenta il sole sopra lo skybox. Si calcola la distanza da quel punto: se rientra entro una certa soglia "raggio", descrive un oggetto circolare che colorato di giallo realizza il sole. Similmente si ottiene un aura di luce attorno ad esso per simulare diversi colori dell'atmosfera in prossimità di esso. Ruotando la luce direzionale nella scena (come verrà descritto successivamente nel capitolo dedicato al

Day-Night Cycle), il sole può assumere diverse posizioni in cielo a seconda del momento della giornata.

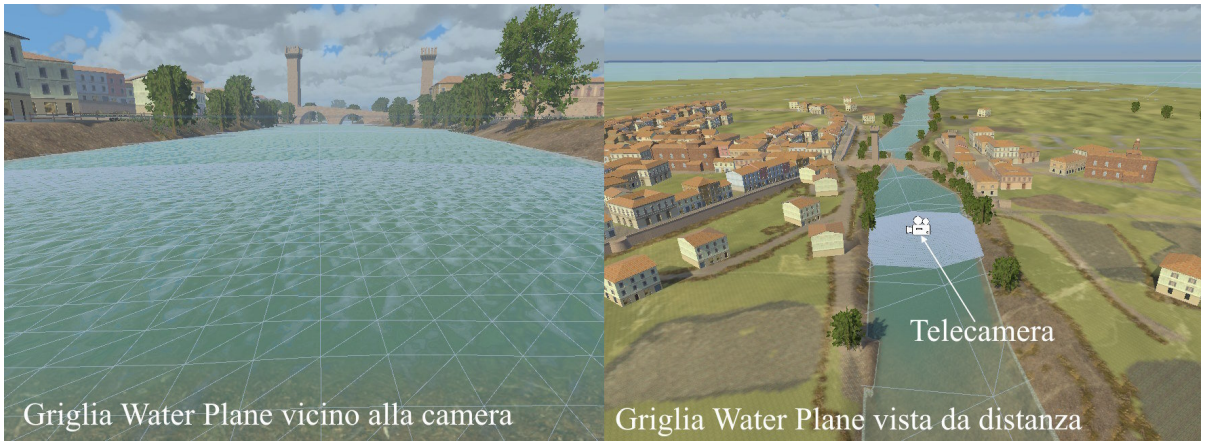
- **Moon**: Analogamente allo shader per il sole, si implementa anche la luna, nella direzione opposta al sole. Per realizzare un effetto di "mezzaluna" si sottrae al moltiplicatore calcolato a partire dalla distanza di un punto sulla UV map (che similmente al sole rappresenterebbe un cerchio) un altro moltiplicatore calcolato a partire dalla distanza di un altro punto, un poco spostato di posizione sulla UV map rispetto al primo. Anche per la luna si implementa un effetto di aura utilizzando ancora la distanza dal punto in cui si proietta la luna.

- **Fog transition**: Si è mostrato in precedenza come si implementa un effetto di nebbia nella scena. Tuttavia la regione nello schermo dove si interrompono il terreno e gli oggetti della scena ed inizia il cielo e lo sfondo è piuttosto evidente. Nella realtà infatti la nebbia è un effetto atmosferico e dunque copre tutto quello che si trova dietro ad essa, il cielo incluso. A questo scopo si realizza un effetto nebbia anche nello *shader* del cielo. Nella parte vicina all'orizzonte si fa tendere il colore del cielo al colore della nebbia. In questo modo si ottiene anche un effetto di continuità tra il terreno ed il cielo, rendendo meno evidente i limiti del mondo e creando un effetto più gradevole e naturale all'occhio.

L'implementazione descritta sfrutta principi relativamente semplici, tuttavia comparata allo *skybox shader* default di Unity 3D, il quale implementa invece un semplice gradiente che interpola il colore del cielo a quello dell'orizzonte, è decisamente più avanzata e permette molta più personalizzazione, nonchè nasconde completamente la linea dell'orizzonte dove la nebbia si interrompe negli shaders degli oggetti della scena, creando un ambiente che risulta molto più continuo e credibile.

3.6 Acqua

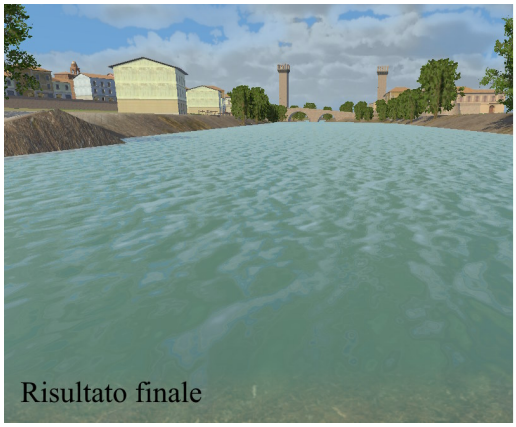
Lo *Shader* dell'acqua, costruito in modo generico per poter realizzare sia fiumi che oceani, sfrutta una serie di tecniche particolari applicate sulla mesh di un piano. Prima di tutto, nel *vertex stage*, utilizzando una combinazione di funzioni di *seno* e *coseno* della variabile temporale, si calcola una distorsione dei vertici del piano in modo da realizzare le onde. Il piano in questione, per poter realizzare grandi oceani o grandi fiumi, dovrebbe essere suddiviso in una griglia molto fitta e densa; tuttavia ciò può far peggiorare la performance, dato che il miglior caso possibile è quello dove si implementa un effettivo sistema di oceani e fiumi che è possibile essere potenzialmente illimitati, o per lo meno sembrare tali. Per questo motivo solo nella parte centrale il piano viene suddiviso in una griglia fine, e via via che si allontana dal centro la griglia diventa più grossolana. Il centro del piano seguirà la telecamera, così che in qualsiasi posizione questa si trovi, sul piano dell'acqua attorno ad essa sarà sempre possibile realizzare un effetto di distorsione del piano per implementare delle onde.

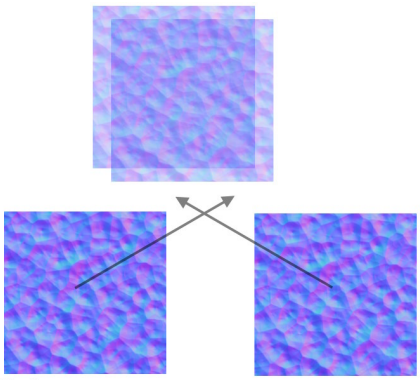


Il *vertex Shader* dell'acqua implementa le seguenti tecniche:

- **Colore di base e riflessi**: Si sfrutta ancora la funzione *UniversalFragmentPBR* da *Lightning.hlsl* [9] per ottenere un *rendering PBR*. Si utilizza un colore di base "blu acqua" e valori di *metallizzazione* e *smoothness* alti per ottenere effetti di riflessione convincenti.

- **Normal Mixing**: Si implementa una *normalmap* che rappresenta l'increspatura dell'acqua. Questa *normalmap* si fa scorrere in una direzione e si miscela ad un'altra *normalmap* identica che scorre in una direzione differente. Il risultato finale è un effetto simile ad increspature che variano nel tempo dando una impressione di fluidità della superficie.


Risultato finale

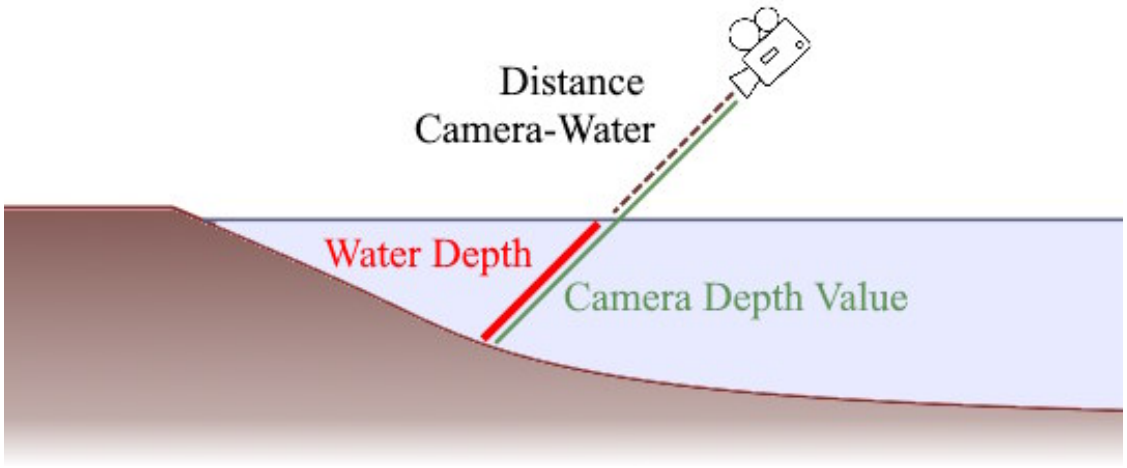

Normal Mixing

- **Depth Value**: Per la realizzazione di alcuni effetti, è comodo ottenere un valore che rappresenta la profondità dell'acqua in un determinato *frammento*. Unity 3D fornisce un buffer chiamato *_CameraDepthTexture* che contiene al suo interno il *Depth Buffer* della camera principale. Sottraendo il valore di questo buffer alla distanza tra la telecamera ed il piano dell'acqua, si ottiene la distanza tra il piano dell'acqua ed il fondale, valore che possiamo interpretare come la profondità del fondale.


Risultato finale

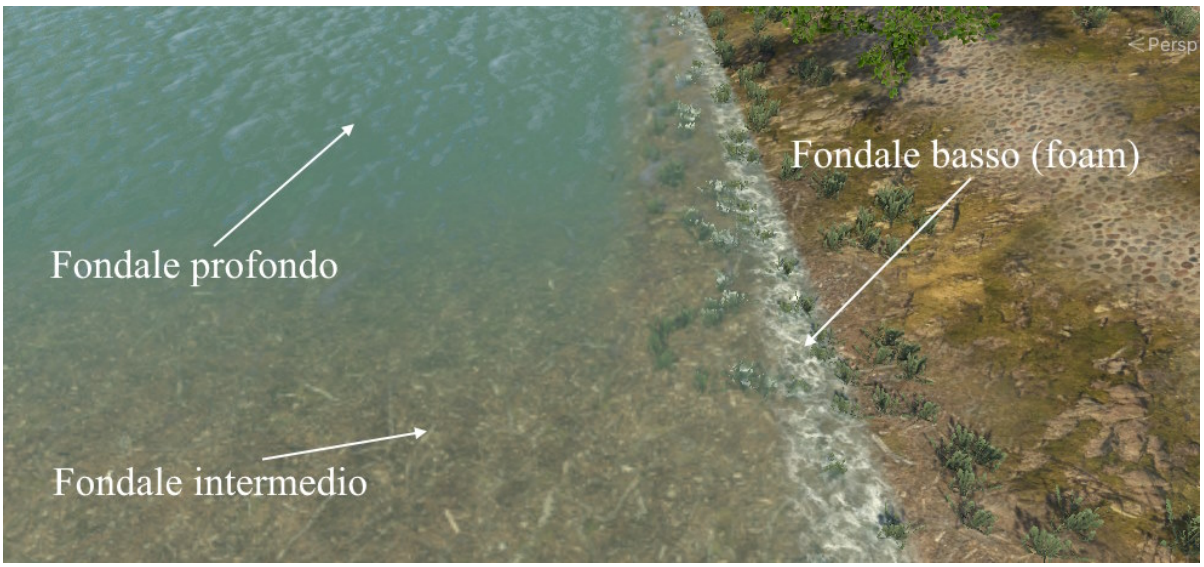

Depth - Distanza Water Plane

- **Foam**: E' possibile utilizzare il *depth-value* per implementare effetti di schiuma o fanghiglia nel caso il suo valore in cui sia sotto una certa soglia. Questo effetto si realizza mescolando, similmente al caso della *normalmap* nelle increspature, due *texture* che scorrono in direzioni opposte.



- **Fondale Distorto** Sempre tramite il valore di *depth* calcolato, è possibile decidere di mostrare il fondale a valori intermedi di profondità dell'acqua. Viene implementato inoltre un effetto di distorsione del fondale, utilizzando il *buffer* di Unity 3D chiamato *_CameraOpaqueTexture* che implementa il risultato del *rendering* opaco della telecamera. Renderizzando questo *buffer* sul piano dell'acqua (trasformando le posizioni UV del piano dell'acqua in quelle UV della camera per poter ottenere il pixel esatto del buffer *_CameraOpaqueTexture*), e applicando una lieve distorsione della UV tramite funzioni di *seno* e *coseno* calcolate sul tempo, si ottiene un effetto di fondale distorto.

- **Fondale Profondo**: Infine a soglie molto alte del valore *depth*, si mostra un colore specifico che rappresenta la torpidità dell'acqua che non permette di intravedere il fondale data l'alta profondità.



3.7 Calore

All'interno della scena modellata vengono posizionati modelli 3D di torce medievali.
Quando il valore del *Day-Night Cycle* indica orari notturni, all'interno della scena vengono applicate alcune luci in prossimità delle torce, assieme a dei *Particle Effects* che rappresentano la fiamma della torcia accesa. Oltre che a *Sprite* di fiamme, dal *Particle* della torcia si sollevano dei piani 2D in modalità *Billboard*, ovvero rivolti verso la telecamera, con sopra applicato uno *Shader* che si può chiamare *Heat Distortion Shader*, il cui scopo è quello di applicare un effetto di distorsione dell'ambiente attorno alla torcia per simulare il calore emanato da essa.
Il funzionamento di questo *Shader* è analogo a quello del fondale nel caso dell'acqua, ovvero il risultato del rendering opaco della telecamera viene ottenuto tramite il buffer *_CameraOpaqueTexture* convertendo le UV del piano *Billboard* in quelle della camera, e la si distorce in prossimità del piano 2D stesso generato dal *Particle Effect*.

```
// Velocità waving dello schermo
float waveSpeed = 3;
//spostamento di un frammento esatto
float2 shift = float2(10 / _ScreenParams.x, (3 / _ScreenParams.y) * .8);
float sinx = sin(_Time.y * waveSpeed + screenUVs.x * 10.0) * shift.x;
float siny = sin(_Time.y * waveSpeed + screenUVs.xy * 15.0) * shift.y;
//uv finali
float2 distorted_uvs = float2(sinx, siny);
```

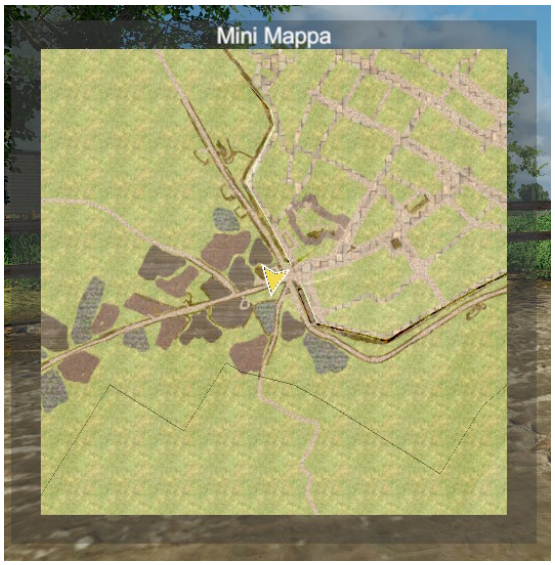
3.8 Mini-Mappa

Le stesse funzioni scritte per lo shader del terreno sono state riciclate per implementare un ulteriore *shader* dedicato ad una Mini Mappa, ovvero una visualizzazione piatta e dall'alto del livello di gioco, che permette di orientarsi facilmente all'interno dell'esperienza.

Sono stati implementati diversi modi di visualizzare il terreno, e la possibilità di sovrapporre sovrapporre le cartine ottenute in precedenza alla Mini Mappa.

- **Mini Mappa Realistica**

  La versione *realistica* della Mini Mappa calcola il colore del pixel in modo completamente analogo a come ciò avviene nel *terrain*, ma con un *tiling* dei *Layer* diverso in modo da ottenerne una visualizzazione proporzionata alla scala della Mini Mappa. Il risultato è un effetto che assomiglia ad una visualizzazione dall'alto del *terrain* della scena.

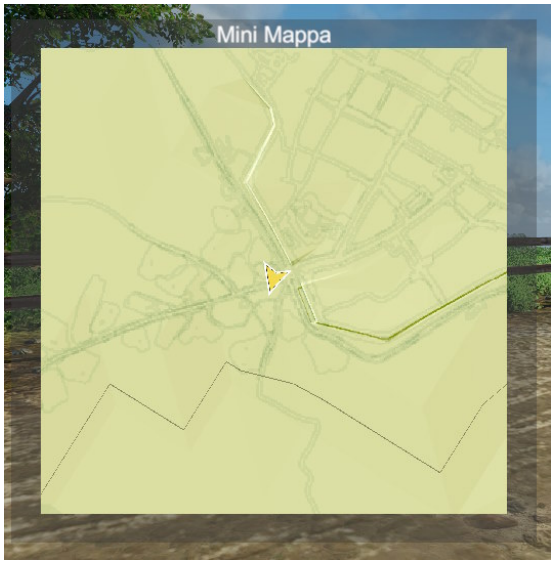


  Tramite la *heightmap* del terreno è possibile inoltre calcolare e mostrare le linee che rappresentano il rilievo del terreno tramite il seguente algoritmo:

```
//linea rilievo topografico
if ((round(height / heightLineDistanceMeters) - round(height_left / heightLineDistanceMeters)) != 0)
  heightLineMult = .35;//blackline  (height)
else if ((round(height / heightLineDistanceMeters) - round(height_up / heightLineDistanceMeters)) != 0)
  heightLineMult = .35;//blackline  (height)
```

- **Mini Mappa Clean** La versione *Clean* della Mini Mappa, invece che utilizzare la *ControlTexture* per mischiare i *Layer* del terreno, utilizza la *ControlTexture* per scegliere di mostrare un colore piatto, che ricorda tipiche tonalità giallognole delle cartine molto datate.

```
float contrast = 300;
float mult = (
      clamp(ct01.r * contrast, 0, 1) +
      clamp(ct01.g * contrast, 0, 1) +
      clamp(ct01.b * contrast, 0, 1) +
      clamp(ct01.a * contrast, 0, 1) +
      clamp(ct02.r * contrast, 0, 1) +
      clamp(ct02.g * contrast, 0, 1) +
      clamp(ct02.b * contrast, 0, 1) +
      clamp(ct02.a * contrast, 0, 1)
      ) / 8;
half3 finalColor = lerp(half3(.8, .8, .4), half3(.1, .3, .1), mult);
```



- **Mini Mappa Flat**

  La versione *Flat* della Mini Mappa utilizza lo stesso principio della versione realistica della Mini Mappa, tuttavia invece che fare il *sampling* della *texture* del *Layer* in modo tipico, si prende sempre il primo pixel dello stesso. In questo modo ogni valore della *ControlTexture* mappa ad un singolo colore, dipendente dal *Layer*, generando un risultato simile ma con colori piatti ed uniformi


```
//always pixel 0
float2 uvs = float2(0, 0);
```

- **Sovrapposizioni di cartine** La modalità di sovrapposizione delle cartine alla Mini Mappa permette di aggiungere una *texture* 'cartina' sovrapposta alla minimappa.
  Le cartine a disposizione per il materiale che usa questo *shader* sono le stesse utilizzate in precedenza per costruire il modello della città su *Blender*.

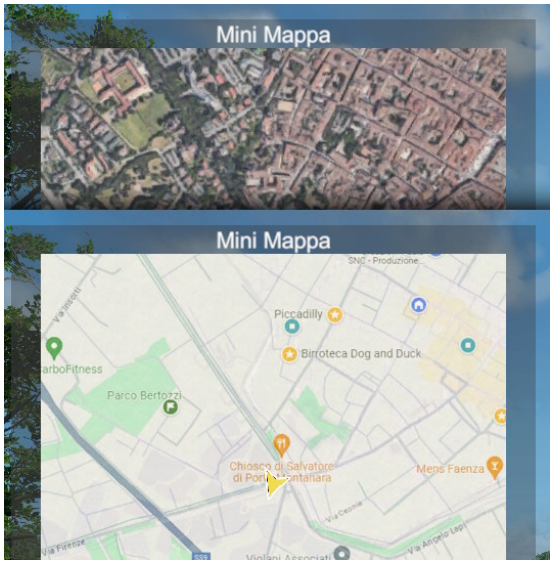


Figura 3.1: Cartina Google Maps / Google Earth

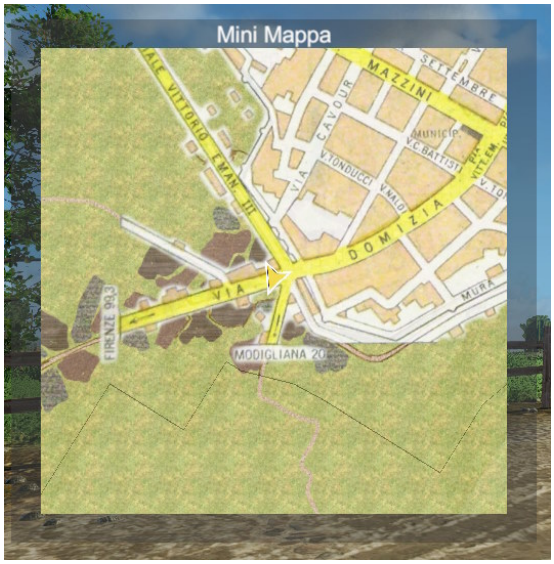


Figura 3.2: Cartina 1943

Figura 3.3: Cartina 1852

Figura 3.4: Cartina 1633

Il sistema della Mini-Mappa presenta inoltre alcune funzionalità dinamiche gestite dal codice *C#*, che permettono lo *zoom* tramite rotella del mouse, il cambio di Mini Mappa tramite le freccette, e la rappresentazione della posizione del giocatore tramite un cursore ruotato nella direzione in cui la telecamera è girata.

Si riportano alcune porzioni rilevanti di codice che gestiscono tale sistema.

```
//player arrow (pos, rot, scale)
playerArrow.anchoredPosition = playerPos;
playerArrow.localEulerAngles = new Vector3(0, 0, ResourcesManager.mainCamera.transform.eulerAngles.y);
playerArrow.localScale = Vector3.one / zoomScale;

//center zoom
miniMap.anchoredPosition = -playerPos;
float zoomXratioLimit = (miniMap.sizeDelta.x - miniMap.sizeDelta.y) / 2;
float limitX = (zoomScale - 1) * (miniMap.sizeDelta.x / 2) + zoomXratioLimit;
float limitY = (zoomScale - 1) * (miniMap.sizeDelta.y / 2);
miniMap.anchoredPosition = new Vector2(
    Mathf.Clamp(-playerPos.x * zoomScale, -limitX, limitX),
    Mathf.Clamp(-playerPos.y * zoomScale, -limitY, limitY)
    );
miniMap.localScale = new Vector3(zoomScale, zoomScale, zoomScale);
```

3.9 Post Processing

Nelle applicazioni di grafica, una volta ottenuto il *rendering* della scena dalla telecamera, talvolta, si utilizzano gli effetti detti di *post-processing* per aumentarne la qualità o applicarne effetti specifici. Si presentano quindi esempi di come ciò può essere realizzato in *Unity 3D*.

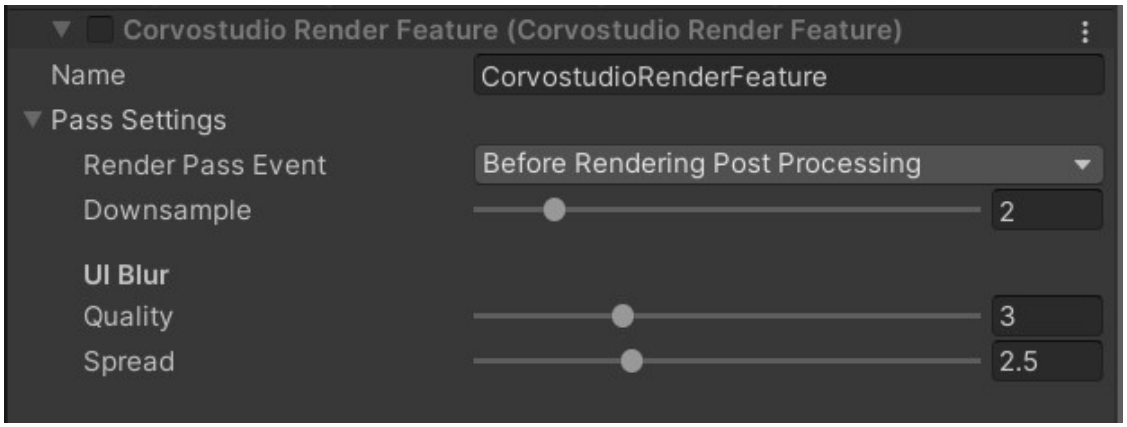
Le 3 versioni di *Unity 3D*, *Universal Render Pipeline (URP)*, *Hight Definition Render Pipeline (HDRP)* e *Legacy*, richiedono metodi diversi per l'implementazione di questa tipologia di effetti.
In questo progetto di tesi viene utilizzato *Unity URP* che implementa due concetti per la realizzazione di effetti di *Post-Processing*:

- **Render Feature**: Si tratta di una funzionalità di *Unity URP* che permette di aggiungere dei *pass* globali per applicare effetti a schermo, normalmente a partire dall'output del rendering della *camera*.
  Si implementano creando una classe estendendo *ScriptableRendererFeature* ed implementando le funzioni
  public override void Create() e
  public override void AddRenderPasses(ScriptableRenderer renderer, ref RenderingData renderingData). È inoltre comune utilizzare questa classe per implementare i parametri visualizzabili tramite l'interfaccia grafica di *Unity*, ad esempio:

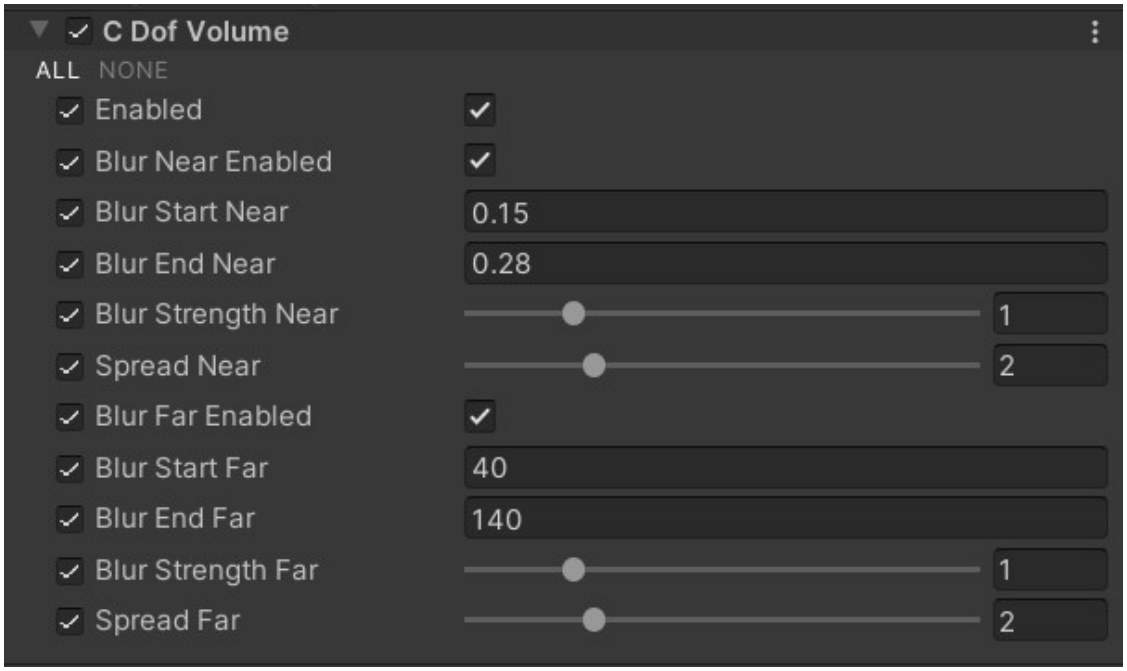
```
// per rendere la classe visibile nell'inspecor
[System.Serializable]
public class PassSettings
{
  // Where/when the render pass should
  // be injected during the rendering process.
  public RenderPassEvent renderPassEvent = RenderPassEvent.AfterRenderingTransparents;

  // Used for any potential down-sampling we will do in the pass.
  [Range(1, 8)] public int downsample = 2;

  //UI Blur
  [Header("UI Blur")]
  [Range(1, 8)] public int quality = 2;
  [Range(.1f, 8)] public float spread = 2f;

  // additional properties ...
}
```

  Si esegue successivamente il *pass* utilizzando una classe derivata dal tipo *ScriptableRenderPass*, che implementa:

- Il costruttore per ricevere le proprietà della render feature
- *public override void OnCameraSetup(CommandBuffer cmd, ref RenderingData renderingData)*
- *public override void Execute(ScriptableRenderContext context, ref RenderingData renderingData)*
- *public override void OnCameraCleanup(CommandBuffer cmd)*



- **Volume Component**: Si tratta di un componente che permette di eseguire dei *pass* per realizzare effetti di *Post-Processing* all'interno di un Volume, ovvero una specifica area, o eventualmente, anche in modo globale. Offre una comoda interfaccia per visualizzare e modificare i parametri degli effetti che controlla. Sebbene questa funzionalità permette di eseguire un *pass* in modo indipendente, negli esempi riportati di seguito si eseguiranno gli effetti solo tramite *Render-Feature* per poter riutilizzare le stesse porzioni di codice per ogni effetto, mantenendo una performance ottimale, e si utilizzerà il *Volume Component* solamente come interfaccia per modificare i parametri.
  Per essere visualizzabile ed utilizzabile nel *Volume Component*, un effetto deve implementare le classi *VolumeComponent* e *IPostProcessComponent*, e successivamente i metodi *public bool IsActive()* e *public bool IsTileCompatible()*.



Tramite questi due metodi si implementano due effetti di *Post-Processing*, quali il *Depth of Field* e il *UI Blur*. Si presentano infine altri effetti *built-in* di *Unity URP* che sfruttano queste tecnologie e che possono venire sfruttati facilmente all'interno del motore grafico.

3.9.1 UI Blur

Con *UI Blur* ci si riferisce ad un *Blur*, ovvero una sfocatura, della *User Interface*, ovvero dell'interfaccia grafica. Questo sistema permette infatti di creare una sfocatura di trasparenza su immagini e bottoni all'interno di un progetto. Questo genere di effetti è molto comune nelle applicazioni grafiche moderne e si implementa in modo piuttosto semplice tramite *shaders* che implementano algoritmi come il *Blur Gaussiano* ed il *Box Blur*.

Questo effetto si implementa tramite una *Render Feature* che processa una *texture* da dare in input allo *shader* che disegna il riquadro.
La *Render Feature* esegue innanzitutto una serie di passi:

Si ottiene un riferimento all'output della telecamera e si prepara un *descriptor* con cui copiare il contenuto del rendering della camera in un buffer apposito. Tramite il parametro *downsample* è possibile impostare il *descriptor* per ottenere una versione in risoluzione ridotta dell'*output* della camera in modo da migliorare la performance dell'operazione. Dovendo creare un effetto di tipo *blur* infatti non ci serve una immagine ad alta risoluzione dato che il risultato finale sarà ad ogni modo completamente sfocato.

```
RenderTextureDescriptor descriptor = renderingData.cameraData.cameraTargetDescriptor;
descriptor.width /= passSettings.downsample;
descriptor.height /= passSettings.downsample;

//color buffer
colorBuffer = renderingData.cameraData.renderer.cameraColorTarget;

// Create a temporary render texture
cmd.GetTemporaryRT(dnwsmplBufferTempID, descriptor, FilterMode.Bilinear);
dwnsmplRtId = new RenderTargetIdentifier(dnwsmplBufferTempID);
```

Si ottiene il *Command Buffer* e si usa il *descriptor* per comparire l'output della camera nel buffer temporaneo. Si invia il buffer copiato come variabile globale per gli *shaders* e si esegue

il *command buffer*.

```
CommandBuffer cmd = CommandBufferPool.Get();

//copy color buffer
Blit(cmd, colorBuffer, dwnsmplRtId);

//send to shader
cmd.SetGlobalTexture(dwnsmplColorBufferShaderPropertyId, dwnsmplRtId);

//execute command buffer
context.ExecuteCommandBuffer(cmd);
cmd.Clear();
CommandBufferPool.Release(cmd);
```

Infine si esegue un cleanup per essere sicuri di non lasciare residui in memoria.

```
cmd.ReleaseTemporaryRT(dnwsmplBufferTempID);
```

Infine si può utilizzare la *texture* copiata nel buffer globale all'interno dell'*UI Blur Shader*. Si implementa un *Box Blur* utilizzando il buffer ottenuto e mescolandolo con i pixel attorno, per un raggio determinato dalla variabile *_BlurQualityUi*.

```
float4 blurColor = float4(0, 0, 0, 0);
for (int x = -_BlurQualityUi; x <= _BlurQualityUi; x++)
    for (int y = -_BlurQualityUi; y <= _BlurQualityUi; y++)
        blurColor += tex2D
        (
            _CDownsampledColorBuffer,
            IN.screenPos + float2(x*shift.x, y*shift.y)
        );
int sampl_in_a_cycle = ((int)_BlurQualityUi * 2 + 1);
blurColor /= (sampl_in_a_cycle* sampl_in_a_cycle);
```

3.9.2 Depth of Field

Questo effetto si riferisce ad una sfocatura di campo simile alla messa a fuoco di una fotocamera, o alla messa a fuoco tipica dell'occhio umano, che permette di visualizzare ciò su cui ci si focalizza, ma sfocando tutto ciò che sta nella visione periferica, ed in questo specifico caso, ciò che sta oltre una certa soglia di distanza minima e massima.

Si implementa questo effetto in modo simile all'*UI Blur*, tuttavia in questo caso si sceglie di eseguire il relativo *shader* come *render pass* dello schermo intero piuttosto che di uno specifico componente della *UI*, dunque il risultato dell'output della camera viene copiato come *pass* dello *shader* e viene eseguito.

```
// shader pass 0
Blit(cmd, dwnsmplRtId, colorBuffer, dof_material, 0);
```

Anche questo *Shader* sfrutta un algoritmo di *Box Blur*, ma in questo caso ottiene la *texture* su cui applicare l'effetto come input del *pass*, seppur il riferimento in memoria sia lo stesso dato che la *texture* deriva dal medesimo codice *Render feature* usato per l'*UI Blur*.

```
for (int x = -strenght; x <= strenght; x++)
for (int y = -strenght; y <= strenght; y++)
    sum += SAMPLE_TEXTURE2D
    (
        _MainTex,
        sampler_MainTex,
        IN.uv + float2(x*shift.x, y*shift.y)
    );
int sampl_in_a_cycle = ((int)strenght * 2 + 1);
sum /= (sampl_in_a_cycle * sampl_in_a_cycle);
```

Per rendere effettiva la sfocatura a partire da una certa distanza, semplicemente viene reso

trasparente il risultato della *fragment function* in modo da visualizzare *l'output standard* della camera dove non intendiamo visualizzare il *DoF*, a seconda della distanza ottenuta tramite il buffer *_CameraDepthTexture*.

```
float remapped_distance = remap(_BlurStartFar, _BlurEndFar, 0, 1, depthInMeters);

float clamped_distance = clamp(remapped_distance, 0, 1);
float final_alpha = pow(clamped_distance, .2);

sum.a = final_alpha;
```

Figura 3.5: Risultato del *_CameraDepthTexture* buffer, utilizzato per decidere dove sfocare l'immagine per ottenere il *DoF*.

3.9.3 Effetti Built-In

Unity URP implementa tramite *Render Feature* e *Volume Component* altri effetti di post processing, messi a disposizione degli utenti come feature di base dell'engine grafico.
Gli effetti che si sfruttano all'interno del progetto includono:

- **Screen Space Ambient Occlusion (SSAO)**: Si tratta di un effetto che simula, senza utilizzare *Ray Tracing* ed altre tecniche computazionalmente costose, il modo in cui la luce viene intrappolata negli angoli e nei punti chiusi nella scena, per realizzare un effetto di penombra.

- **Bloom**: Si tratta di un effetto che imita il modo in cui le telecamere catturano la luce, creando un alone sulla lente. Questo effetto risulta spesso in un'immagine più gradevole dove le luci paiono più complesse e accentuate.

- **Vignette**: Questo effetto scurisce i bordi del rendering finale, rendendo l'output meno piatto e facendo focalizzare il giocatore maggiormente al centro dello schermo.

Tra i vari effetti di *Post-Processing* di base di Unity, quelli descritti rappresentano i più comuni dato il loro basso costo in termini di performance in proporzione all'impatto visivo che forniscono. L'implementazione di questi effetti come è non risulta in concetti molto diversi da quelli rappresentati in precedenza, e quindi si tratta unicamente di algoritmi che sfruttano

in modo molto interessante buffer del colore della camera, il *depth buffer* e altri strumenti di questo tipo.

Capitolo 4

Logica del gioco & Gameplay

Tramite il sistema *Component* di *Unity 3D*, sono stati scritti, mediante classi *C#*, componenti attivi e passivi che possono venire associati a specifici oggetti istanziati nella scena.
I componenti di *Unity*, che estendono la classe *MonoBehaviour*, permettono di implementare una serie di funzionalità:

- **Inspector Variables**: Le variabili definite come pubbliche all'interno della classe possono essere visualizzate e modificate tramite l'*Inspector* di *Unity 3D*, permettendo una agile modifica di esse tramite una *UI* grafica.

- **Start() e Awake()**: Sono funzioni che se implementate verranno richiamate all'avvio della scena, o quando un oggetto che implementa il componente è istanziato in essa; *Awake()* viene chiamata durante l'istanziazione dell'oggetto, mentre *Start()* viene chiamata appena l'istanziazione è completa.

- **Update() e LateUpdate()**: Vengono chiamate prima di disegnare un *frame*, la seconda una volta terminata la prima. Notare come la seconda infatti viene eseguita dopo che ogni animazione ha applicato le trasformazioni nella scena per il *frame* da disegnare successivamente.

- **Destroy() e OnDestroy()**: Si può chiamare la funzione *Destroy()* per rimuovere un oggetto e tutti i componenti ad esso associati dalla scena, e quindi dalla *Hierarchy*. Quando questa funzione è chiamata, tutti i componenti chiameranno automaticamente la funzione *OnDestroy()*, se implementata.

- **GetComponent**$\langle T \rangle$**()**: I componenti possono comunicare tra di loro ottenendo le rispettive *references* come oggetto del tipo della classe che li definisce. Si può ottenere da

un oggetto nella scena *objectReference* uno specifico componente del tipo T tramite la funzione *objectReference.GetComponent⟨T⟩()*.

4.1 Controller del personaggio

Per il controller del personaggio è stato implementato un *Component* chiamato *PlayerController* nel quale si intende gestire l'input da tastiera, il movimento della telecamera, il movimento del del personaggio e le animazioni dello stesso.

Si gestisce l'*Input* dell'utente tramite la classe *Input*, ottenendo lo stato di pressione dei tasti della *keyboard* tramite la funzione *Input.GetKey()*, lo stato della rotella del mouse tramite la variabile *Input.mouseScrollDelta* e il *delta* del movimento del mouse rispetto al *frame* precedente tramite la funzione *Input.GetAxisRaw("Mouse AXIS")*.

Il movimento dell'oggetto che definisce il personaggio e della telecamera all'interno della scena avviene tramite il componente *Transform* che definisce variabili per rappresentare la posizione la rotazione e la dimensione dell'oggetto nello spazio 3D, assieme ad una serie di funzioni per operare traslazioni, rotazioni e ridimensionamenti dello stesso.
Si usa inoltre il componente *CharacterController* per muovere l'oggetto nella scena tramite il *Transform*, vincolando però il movimento secondo le collisioni presenti nella scena e quelle relative all'oggetto stesso (*Unity Colliders*).

Per la gestione delle animazioni si utilizza il componente *Animator*, il quale rappresenta una macchina a stati dove ogni stato è connesso ad una specifica animazione (in formato *.anim*) e la quale transizione avviene secondo lo stato di alcune variabili. Tramite la classe *PlayerController* dunque si calcola il valore da dare alle variabili per raggiungere lo stato e quindi l'animazione desiderata, e si comunica il valore di esse all'*Animator component*.

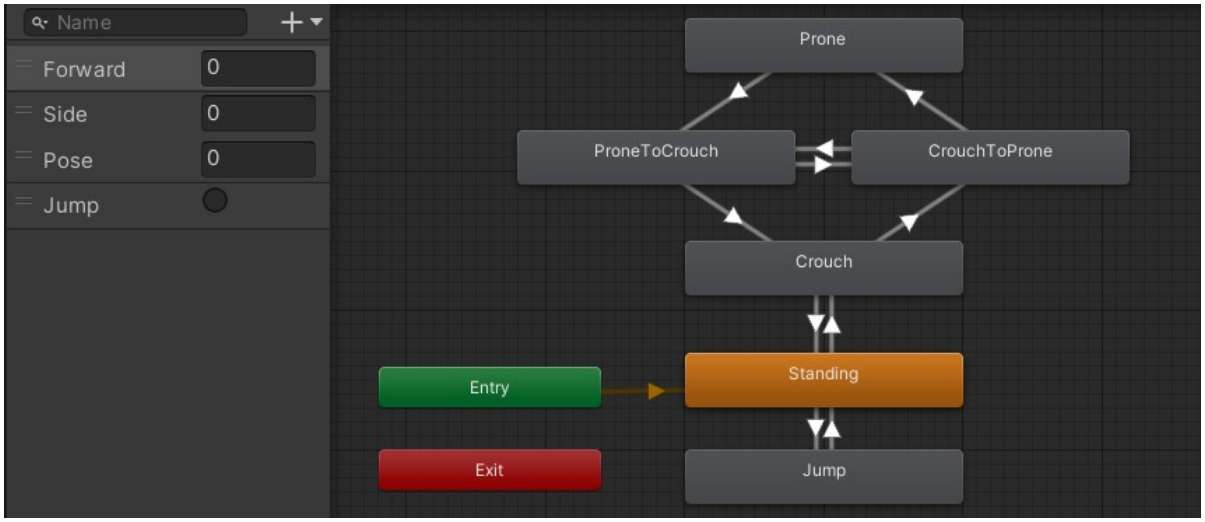


Figura 4.1: Macchina a stati del componente *Animator*

La seguente porzione di codice mostra come si ottengono alcune variabili di *Input*, le si processano, le si utilizzano per muovere l'oggetto che rappresenta il giocatore, ruotare la telecamera ed infine le si inviano alla macchina a stati per la gestione delle animazioni.

```
protected virtual void Update() {
    // ---input---
    float input_x = Input.GetAxisRaw("Horizontal");
    float input_y = Input.GetAxisRaw("Vertical");
    float mouse_x = Input.GetAxisRaw("Mouse X") * LookSensitivity;
    float mouse_y = Input.GetAxisRaw("Mouse Y") * LookSensitivity;
    bool sprinting = Input.GetKey(KeyCode.LeftShift);

    // disable mouse rotate if cursor visible
    if (Cursor.lockState != CursorLockMode.Locked)
      mouse_x = mouse_y = 0;

    // pass scroll wheel input to MiniMapGUI
    // to operate zoom
    MiniMapGUI.ZoomMinimap(Input.mouseScrollDelta.y * .03f);

    if (Input.GetKeyDown(KeyCode.T))
      SetTps(!tps);

    // move direction
    Vector3 direction = Vector3.zero;
    direction += transform.forward * input_y;
    direction += transform.right * input_x;
    ...
    // ---move object in the 3D space---
    // move object
    direction += velocity * Time.deltaTime;
    movementController.Move(direction * Time.deltaTime * currMoveSpeed);

    // rotate camera
    yaw += mouse_x;
    pitch -= mouse_y;
    yaw = ClampAngle(yaw, MinYaw, MaxYaw);
    pitch = ClampAngle(pitch, MinPitch, MaxPitch);
    transform.eulerAngles = new Vector3(0, yaw, 0.0f);
    playerCamera.transform.localEulerAngles = new Vector3(pitch, 0, 0.0f);

    ...
    // ---pass animation variables to state machine---
    m_animator.SetFloat("Forward", sprinting? input_y: input_y/2, .05f, Time.deltaTime);
    m_animator.SetFloat("Side", mouse_x+(sprinting ? input_x: input_x/2), .05f, Time.deltaTime);
    ...
}
```

Le animazioni controllate dal componente *Animator* forzano la posizione delle *bones* del *rig* del personaggio; tuttavia, se si ottiene una *reference* ad una *bone* è possibile modificarne la posizione nella funzione *LateUpdate*, ricordandosi però che nel ciclo di *frame* successivo, le trasformazioni dell'animazione resetteranno la posizione e rotazione di ciò che si manipola in questo punto. Dunque la trasformazione applicata in *LateUpdate* va applicata ogni frame.
Si mostra di seguito un esempio di trasformazione che ruota la *bone* del collo del personaggio per forzarlo a guardare in una determinata direzione.

```
float currentYaw = 0, currentPitch = 0;
private void LateUpdate()
{
  if (focalPoint!=null)// must look somewhere
  {
    Vector3 toFocalPointYaw = (new Vector3(focalPoint.transform.position.x, transform.position.y,focalPoint.transform.position.z)
        - transform.position).normalized;

    Vector3 toFocalPoint = (focalPoint.transform.position - transform.position).normalized;

    float yaw = Mathf.Clamp(Vector3.SignedAngle(transform.forward, toFocalPointYaw, Vector3.up),-50,50);
    float pitch = Mathf.Clamp(Vector3.SignedAngle(toFocalPoint, toFocalPointYaw, transform.right), -35,35);

    currentYaw = Mathf.Lerp(currentYaw, yaw, Time.deltaTime * 5);
    currentPitch = Mathf.Lerp(currentPitch, pitch, Time.deltaTime * 5);

    // rotate neck bone toward calculated angle
    neckBone.transform.localEulerAngles += new Vector3(-currentYaw, 0, -currentPitch);
  }
  else// nothing to look at
  {
    currentYaw = Mathf.Lerp(currentYaw, 0, Time.deltaTime * 5);
    currentPitch = Mathf.Lerp(currentPitch, 0, Time.deltaTime * 5);

    // rotate neck bone toward default angle
    neckBone.transform.localEulerAngles += new Vector3(-currentYaw, 0, -currentPitch);
  }
}
```

Focal Point, come descritto in seguito, rappresenta la reference ad un oggetto, in particolare un monumento, verso il quale la bone del collo, *neckBone*, si volti per dare l'impressione che il nostro personaggio lo stia osservando.

4.2 Menu & Day-Night Cycle

Nei capitoli relativi agli *Shaders* sono stati trattati vari effetti quali la dinamicità dello *skybox*, gli effetti di neve e bagnato sul suolo, e la nebbia. È possibile modificare in tempo reale i suddetti effetti tramite un menù dedicato in gioco.

Figura 4.2: *Day-Night Cycle* menù impostato su clima nevoso.

Per implementare il menu è stato utilizzato il sistema integrato di *Canvas UI* di *Unity 3D*, sfruttando i componenti *Text*, *Slider* e *Dropdown*.
E' stato implementato un componente *UiController* per leggere lo stato dei valori controllati dagli elementi della *UI* per poi processarli ed utilizzarli per modificare il clima in gioco tramite i parametri degli *shaders*.

In particolare è stata implementata la classe *DayNightCycle* come interfaccia la quale interpreta e traduce i valori della *UiController*, tramite interpolazioni lineari ed altre funzioni matematiche, nel valore adatto degli *shaders* per ottenere l'effetto desiderato.
Ad esempio il valore della *UI 'Daytime'*, rappresentato tramite uno *Slider* che ritorna un valore compreso tra 0 e 1, viene innanzitutto interpretato come moltiplicatore dell'ora del giorno come valore che va dalle *00:00* di mattina alle *23:59* di sera. Il valore che rappresenta mezzogiorno è dunque 0.5, e possiamo quindi interpretare il valore *Daytime* attraverso una funzione simile ad una parabola per ottenere a partire da esso il livello di luce ambientale che vogliamo nella scena.

Si passa il valore *Daytime* in una variabile chiamata *currentDayMultiplier* nella classe *Day-*

NightCycle, e si crea la seguente funzione per ottenere il moltiplicatore di luce ambientale descritto:

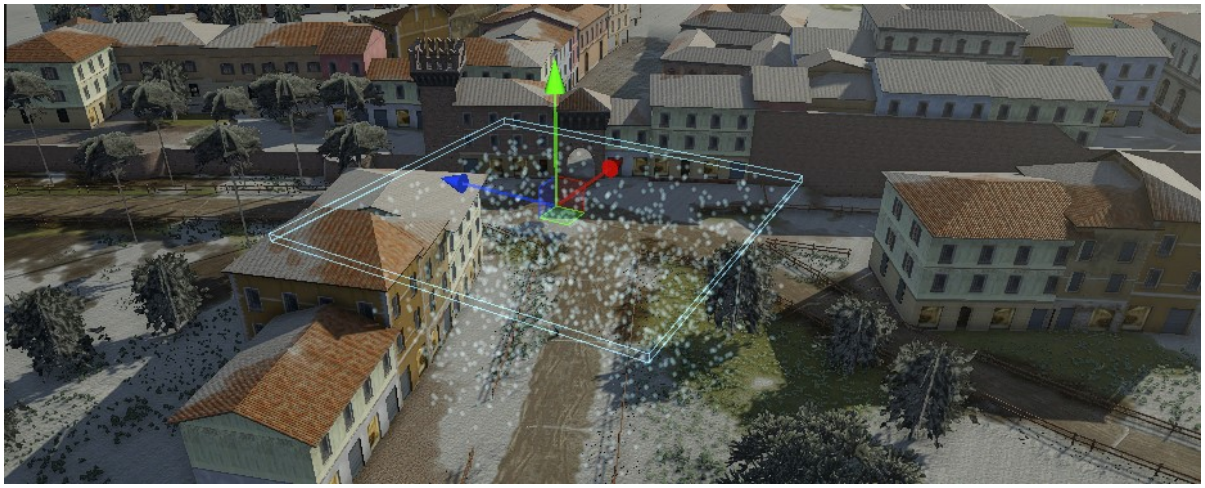
```
public float DayMultiplier {
    get {
      return dayNightPower.Evaluate(currentDayMultiplier);
    }
  }
```

Si interpola il valore di luce ambientale utilizzando il valore ottenuto nel seguente modo.

```
Color ambientColor = Color.Lerp(midNightAmbientColor, midDayAbientColor, DayMultiplier);
RenderSettings.ambientEquatorColor = ambientColor
```

Similmente si creano funzioni e sistemi per applicare dunque il moltiplicatore della neve, l'indice di quantità di nuvole, la rotazione della luce solare e quindi la posizione del sole e della luna, il colore e la quantità di nebbia, il valore aumentato di riflessione degli *shaders* per simulare il terreno bagnato e così via.

Infine, se si dovessero impostare i valori dello *Slider 'Weather'* su *Raining* o *Snowing*, si istanzia nella scena un effetto particellare di *Neve* o *Pioggia*, i quali consistono in piani 2D *Billboard*, ovvero costantemente rivolti verso la telecamera e con applicate *texture* con trasparenza che rappresentano una goccia d'acqua o un fiocco di neve, e ci si assicura che questo effetto segua costantemente la telecamera, in quanto per ottenere una performance accettabile l'area coinvolta dal *Particle Effect* è molto limitata.



L'immagine precedente mostra l'area su cui ha effetto il *Particle Effect* che genera e anima i piani nella scena che contenono le *texture* dei fiochi di neve. L'effetto segue costantemente la camera per assicurarsi che questa sia sempre al centro e nasconda la limitatezza dell'area sulla quale l'effetto si estende.

4.3 Interazione con i monumenti

E' stato implementato un sistema di interazione con i monumenti presenti nella scena. L'interazione avviene rivolgendo la camera verso un determinato monumento nella scena e premendo il *Tasto sinistro del mouse*. L'esecuzione consiste in un effetto particellare che mostra l'inizio della interazione, e la comparsa di uno stendardo contenente un testo che cita il nome del monumento assieme ad una breve descrizione.

L'implementazione del sistema avviene tramite l'uso dei componenti *Box Collider* di *Unity 3D* e tramite la funzione *Physic.Raycast()* per intercettare uno di questi *Collider* a partire dalla posizione e rotazione corrente della camera.

```
if (Input.GetKeyDown(KeyCode.Mouse0))
{
  if (Physics.Raycast(mainCamera.transform.position, mainCamera.transform.forward, out RaycastHit rh, max_focal_dist, LayerMask.GetMask("TransparentFX")))
  {
    focalPoint = rh.collider.GetComponent<FocalPoint>();
    focalPointDistance = Vector3.Distance(ResourcesManager.mainCamera.transform.position, focalPoint.transform.position);

    switch (focalPoint.effect)
    {
      // type of transition: lighting
      case FocalPoint.ParticleEffect.Bolt:
        GameObject boltFX = Instantiate((GameObject)Resources.Load("BoltFX"));
        Destroy(boltFX, 1);// destroy after 1s
        StartCoroutine(SetFocalPointCR(focalPoint, .8f));
        break;

      // type of transition: rock into water
      case FocalPoint.ParticleEffect.Rock:
        GameObject rockFX = Instantiate((GameObject)Resources.Load("RockFX"));
        Destroy(rockFX, 3);// destroy after 3s
        StartCoroutine(SetFocalPointCR(focalPoint, 1.5f));
        break;
    }
  }
}
```

Le classi *Focal Point*, accennate in precedenza, definiscono la struttura dati che dichiara l'esistenza di un dato monumento e ne contengono le proprietà quali la descrizione ed il tipo di effetto particellare da mostrare ad inizio interazione.

4.4 Conclusione del gioco

L'esperienza termina con una schermata conclusiva, una volta raggiunta la fine del percorso. Qui vengono mostrati i crediti finali ed un contatore relativo al numero di monumenti incontrati durante il percorso. Se si è interagito con un monumento, esso viene considerato come monumento incontrato, e quindi il contatore viene incrementato. È anche presente un tasto *"Leave Game"* che ferma l'esecuzione del programma. La schermata finale esegue un *fading* verso una schermata nera e una comparsa del testo graduale, avviata tramite una animazione *.anim* eseguita tramite un *Animation component* chiamando la funzione *end_card_animation.Play*().

```
if (Vector3.Distance(transform.position, quitPos) < quitPosDist)
{
  end_card_animation.Play();
  lock_input = true;
}
```

Screenshots

Si mostrano di seguito una serie di screenshots dalla versione finale del progetto.

Figura 4.3: Porta Montanara, esterno delle mura, giorno.

Figura 4.4: Piazza, vista da Est, giorno.

Figura 4.5: Piazza, vista da Ovest, giorno.

Figura 4.6: Ponte romano, vista aerea, giorno.

Figura 4.7: Ponte romano, giorno, clima nevoso.

Figura 4.8: Piazza, notte, clima piovoso.

Ringraziamenti

Nella chiusura di questo elaborato, intendo dedicare una sezione per mostrare una dovuta riconoscenza nei riguardi di tutti coloro che hanno permesso il mio conseguimento della mia Laurea triennale in questo corso di studi.

Prima di tutto è doveroso ringraziare mia mamma Claudia, il suo compagno Rodolfo, mia zia Flavia e mio nonno Goffredo, i quali mi hanno permesso in primo luogo di poter iniziare questo percorso, sostenendomi in ogni momento.

Un grazie speciale va alla mia docente Damiana Lazzaro, la quale mi ha permesso di affrontare questo tema come conclusione della mia tesi di laurea, il quale è un argomento di cui ho grande passione e che ho trattato con enorme piacere.

Figure chiavi con la quale ho condiviso innumerevoli sessioni di studio, preparazione ad esami e progetti di gruppo sono stati i colleghi conosciuti in questi anni, con i quali non solo ci si è aiutati reciprocamente ma si sono vissute esperienze nuove e costruiti nuovi legami.

Un ultimo ringraziamento va agli amici stretti, che sono stati in grado di sostenermi nei momenti duri e di darmi la forza mentale di arrivare fino alla fine.

Bibliografia

[1] Blender, 3D modeling software. Source Link

[2] Built-in shader variables, Unity 3D documentation. Source Link

[3] Computer Animated Hand 1972, History of Computer Animation (CGI). Source Link

[4] Dominik Susmel, RSL Documentation 0.3. Source Link (Wayback machine)

[5] Esercito Tedesco, Cartina di Faenza (Faenza 1943). Source Link

[6] Felyx McGarry, Jenn's Guide to Trim Sheets (2021). Source Link

[7] Google LLC, Google Maps (2023). Source Link

[8] Joan Blaeu, Faventia vulgo Faenza in Theatrum civitatum et admirandorum Italiæ (Amsterdam 1633). Source Link

[9] Lighting.hlsl, Unity-Technologies URP github repo. Source Link

[10] Materialize, Materials creation tool. Source Link

[11] Next Generation, Top Games Developers Call On Microsoft to Actively Support OpenGL (1997). Source Link

[12] Paul Thurrott, WinInfo article (17/12/1997). Source Link

[13] Paint.net, Photo Editing Software. Source Link

[14] Pixar RenderMan, RenderMan movies. Source Link

[15] Speed Tree, 3D Vegetation Modeling and Middleware. Source Link

[16] Textures.com, Textures & Assets database. Source Link

[17] topographic-map.com (2023). Source Link

[18] Un Faentino anonimo, Cartina di Faenza (Faenza 1838). Source Link

[19] Umips.net, IRIX Info page. Source Link

[20] Unity ShaderLab, Unity Shaders Documentation. Source Link

[21] Unity 3D, Game Engine. Source Link

[22] Visual Studio, IDE. Source Link

[23] Wayne E. Carlson, History of Computer Graphics and Animation. Source Link