Alma Mater Studiorum – Università di Bologna

DIPARTIMENTO DI INFORMATICA - SCIENZA E INGEGNERIA (DISI)
Corso di Laurea Magistrale in Ingegneria Informatica

Tesi di Laurea Magistrale
in
Fondamenti di Computer Graphics M

Sviluppo ed implementazione di un Digital Twin della città di Bologna

Candidato:
Federico Andrucci

Relatore:
Chiar.ma Prof.ssa Serena Morigi

Correlatore:
Ing. Antonella Guidazzoli

Anno Accademico 2022-2023
Sessione VI

Introduzione

La continua crescita delle tecnologie digitali sta trasformando radicalmente il modo in cui interagiamo con il mondo che ci circonda, offrendo nuove prospettive nel modo in cui possiamo progettare, costruire, gestire e vivere le nostre città. In questo contesto, il concetto di Digital Twin, ovvero un modello digitale dinamico che replica fedelmente un sistema fisico, emerge come una soluzione promettente per affrontare le complesse sfide urbane del XXI secolo.

La presente tesi, svolta in uno stage curricolare presso Cineca a Bologna, uno dei più importanti centri di supercalcolo e HPC in Europa, è incentrata sullo sviluppo di un proof of concept di un Digital Twin della città di Bologna. *Bologna Digital Twin* è un importante progetto finanziato e promosso dal Comune di Bologna, Università degli Studi di Bologna, Cineca e Fondazione Bruno Kessler. Ha come obiettivo la costruzione di un sofisticato Digital Twin dell'intera città metropolitana di Bologna, con lo scopo di fornire un potente strumento che possa facilitare il planning urbano, la gestione delle infrastrutture, la partecipazione dei cittadini, ma soprattutto analizzare e prevedere il comportamento della città in risposta a eventi naturali o antropici.

Questo progetto di tesi è stato sviluppato in parallelo ad un altro collega, quindi i due elaborati sono strettamente correlati e si completano a vicenda. Per questo motivo la trattazione è stata suddivisa nei due progetti di tesi. Per evitare ripetizioni tra i due elaborati e per mantenere una coerenza tra i due, saranno in comune solo i capitoli introduttivi del progetto, mentre i capitoli successivi, ovvero l'implementazione, saranno trattati in maniera differente e separata.

All'interno del progetto Bologna Digital Twin nostro obiettivo è stato quello di sviluppare un proof of concept di Digital Twin della città di Bologna,

partendo dal processing di dati grezzi fino alla visualizzazione dell'ambiente 3D, in particolare facendo focus sull'ottenimento di modelli 3D accurati e completi, partendo da dati eterogenei e un successivo sviluppo di un Digital Twin utilizzando questi dati. Inoltre un altro obiettivo era quello di rendere questo Digital Twin un ambiente dinamico attraverso lo sviluppo di servizi tramite il quale il Digital Twin possa essere costantemente aggiornato con i dati in tempo reale.

Nel primo capitolo verrà analizzato lo stato dell'arte dei Digital Twin Urbani gia esistenti. Per poi successivamente spostare l'attenzione sull'analisi dei dati grezzi di partenza, forniti da Cineca, e sulle criticità e le sfide che si sarebbero dovute affrontare.

Dal terzo capitolo in poi, i due progetti di tesi si separano e trattano parallelamente il processing dei dati di partenza con l'obiettivo di ottenere modelli 3D concreti ed utilizzabili per costruire il Digital Twin della città di Bologna. Nel dettaglio, il mio elaborato tratta l'ottenimento delle mesh degli edifici della città, il texturing degli stessi e l'elaborazione dei dati Comunali sul verde pubblico.

Infine, negli ultimi due capitoli il focus si sposta sullo sviluppo di un proof of concept di Digital Twin. Nel quarto capitolo viene costruita la scena 3D statica, mentre nel quinto ed ultimo capitolo si sviluppa la dinamicità del Digital Twin, ovvero la navigazione all'interno della città e l'interfacciamento con i dati in tempo reale.

Indice

Introduzione **i**

1 Digital Twin **1**
1.1 Che cos'è un Digital Twin 1
1.2 Digital Twin Urbani 2
1.3 Stato dell'arte dei Digital Twin 3
1.3.1 Architettura 3
1.3.2 Esempi di Digital Twin Urbani 6

2 Bologna Digital Twin **11**
2.1 Obiettivi del progetto 11
2.2 Analisi di Progetto 12
2.2.1 Raccolta Dati 13
2.2.1.1 OpenData Comune di Bologna 13
2.2.1.2 Rilievi aerei 18
2.2.2 Modellazione geometrica 3D del paesaggio 25
2.2.3 Il problema del texturing 28
2.2.4 Motore grafico di visualizzazione 29
2.2.5 Integrazione di dati real time 31
2.3 Workflow di elaborazione dati 32
2.4 Tool e Software utilizzati 33

3 Elaborazione di dati grezzi **37**
3.1 Elaborazione dati per ottenere mesh degli edifici 38
3.1.1 Processing csv carta tecnica comunale 39
3.1.2 Preparazione shapefile da csv 43
3.1.2.1 Organizzazione degli edifici in tiles 43
3.1.2.2 Creazione shapefiles edifici 45

3.1.3 Texturing buildings 46
3.1.3.1 Sviluppo del Texturer 49
3.1.3.2 Risultati texturing e fixing script 54
3.1.4 Export mesh degli edifici 59
3.2 Digital Terrain Model: DTM 59
3.3 Processing di dati sul Verde Urbano 61
3.3.1 Preparazione e organizzazione dati 62
3.3.2 Sviluppo ed implementazione della RestAPI 65
3.4 Elaborazione dati LiDAR per ottenere i tetti 71

4 Sviluppo del Digital Twin statico 75
4.1 Importazione dei modelli 3D della città 75
4.2 Modellazione degli alberi 79
4.3 Creazione della scena statica del Digital Twin 82

5 Sviluppo della dinamicità del Digital Twin 85
5.1 Struttura di un progetto Unreal Engine 85
5.2 Introduzione all'intera infrastruttura progettata 86
5.3 Sviluppo del core di Bologna Digital Twin 87
5.3.1 Sviluppo della classe GameMode 87
5.3.2 Implementazione della navigazione 89
5.3.3 Implementazione della Ciclo Giorno/Notte 92
5.4 Collegamento con dati esterni 95
5.4.1 Alberi 97
5.4.2 Precipitazioni 102
5.4.3 Traffico 103
5.5 Profiling prestazioni 104
5.5.1 Profiling Navigazione e Layer di Visualizzazione 107
5.5.2 Profiling Alberi 110

A Codice addizionale 113

Conclusioni 119

Elenco delle figure

1.1 Architettura semplificata di un DT 4
1.2 Digital Twin della città di Shangai 6
1.3 Digital Twin della città di Wellington 7
1.4 Digital Twin della città di Herrenberg 7
1.5 Digital Twin della città di Perugia 8

2.1 Open Data: edifici volumetrici 15
2.2 Open Data: distribuzione alberi 18
2.3 Comune di Bologna: suddivisione in tile 20
2.4 Visualizzazione Las su CloudCompare 21
2.5 Esempio foto obliqua 23
2.6 Esempio ortofoto 24
2.7 Esempio disposizione ortofoto 25
2.8 Esempio ortofoto (versione CIR) 26
2.9 Nuvola di punti LiDAR visualizzata in CloudCompare 27
2.10 Diagramma del worflow completo 32
2.11 Blender 34
2.12 Meshlab 34
2.13 Unreal Engine 5 35

3.1 Sezione wowflow: edifici 38
3.2 Numero di edifici con quota s.l.m. 0 41
3.3 Rimozione di edifici con descrizioni non utili 42
3.4 Dataframe di tutti i LLC dei tiles 44
3.5 Stampa dello shape di un tile preso come esempio 46
3.6 Mesh di un tile su Blender 47
3.7 Piazza Maggiore su Google Earth 48
3.8 Mesh Google in Blender 49

3.9 Diagramma UML delle classi dell'estensione Blender 51
3.10 Texturing tramite estensione Blender 54
3.11 Atlas risultante 55
3.12 Errore nel texturing dovuto a elementi esterni 56
3.13 Esempio view 1 57
3.14 Esempio view 2 58
3.15 Sezione worflow: DTM 59
3.16 Visualizzazione LOD mesh DTM 60
3.17 Sezione worflow: DTM 60
3.18 Sezione worflow: verde urbano 61
3.19 Distribuzione delle specie arboree 63
3.20 Distribuzione delle specie arboree nell'area del centro storico . 64
3.21 Mappa degli alberi del centro storico 66
3.22 Schema parziale del database degli alberi 68
3.23 Diagramma uml della REST API 69
3.24 Sezione workflow: tetti 72
3.25 Step by step del processo di ottenimento dei tetti 73
3.26 Visualizzazione finale dei tetti sul Digital Twin 74

4.1 LOD visualizzati in wireframe 76
4.2 Diverse tipologie di collisioni 77
4.3 Visualizzazione poligoni di collisione 78
4.4 Materiale di un DTM 78
4.5 Visualizzazione delle collisioni dei buildings 79
4.6 Mesh dei tetti del centro 80
4.7 Tree It 80
4.8 Texture del tronco 81
4.9 Texture della foglia 82
4.10 LOD degli alberi 83
4.11 Previsualizzazione centro Bologna in Unreal 84

5.1 Diagramma dell'architettura del Digital Twin 87
5.2 Blueprint della GameMode 89
5.3 Visualizzazione degli edifici senza texture 90
5.4 Blueprint del Controller 91
5.5 Navigazione in prima persona 92
5.6 Navigazione in top down 92

5.7 Parametri per il calcolo della posizione del sole 94
5.8 Timeline per il calcolo della posizione del sole 94
5.9 Blueprint di gestione del ciclo giorno/notte 95
5.10 Ciclo giorno notte 96
5.11 Gerarchia di componenti dell'attore TreeSpawner 99
5.12 Evento BeginPlay dell'attore TreeSpawner 100
5.13 Alberi posizionati in scena tramite TreeSpawner, zona Giardini Margherita 100
5.14 Evento di click con il tasto destro del mouse dell'attore TopDownCamera 101
5.15 Ottenimento albero selezionato e popolazione del widget . . . 101
5.16 Sistema particellare per la pioggia 102
5.17 Visualizzazione della pioggia 103
5.18 Dati sul traffico: Via Rizzoli 105
5.19 Dati sul traffico: San Felice 105
5.20 Profiler di Unreal Engine 106
5.21 Profiling navigazione e layer di visualizzazione 107
5.22 Profiling ToggleVisibility 108
5.23 Profiling SwapAllMaterials 108
5.24 Nuovo materiale parametrizzabile 109
5.25 Profiling con nuovo sistema di materiali 110
5.26 Profiling 5k alberi 111
5.27 Profiling con numero di alberi superiori al normale 112

Capitolo 1

Digital Twin

Contents

1.1 Che cos'è un Digital Twin 1

1.2 Digital Twin Urbani 2

1.3 Stato dell'arte dei Digital Twin 3

1.3.1 Architettura 3

1.3.2 Esempi di Digital Twin Urbani 6

1.1 Che cos'è un Digital Twin

Il Digital Twin, rappresenta una delle innovazioni più significative ed impattanti nell'ambito della digitalizzazione e dell'industria 4.0. Definirlo semplicemente come una replica digitale di un oggetto fisico sarebbe riduttivo; il Digital Twin è, infatti, un modello dinamico che simula con precisione le caratteristiche, i processi e il comportamento di sistemi fisici nel mondo reale. Questo tipo di modello è diventato sempre più rilevante con l'avvento dell'Internet delle cose (IoT) [43] e del Cloud Computing [20]. Permettendo l'integrazione e l'analisi di vasti volumi di dati raccolti da sensori IoT dislocati sul sistema fisico, consentendo di monitorare lo stato, di prevedere le prestazioni future attraverso simulazioni e di ottimizzare i processi decisionali in tempo reale [30].

La caratteristica distintiva dei Digital Twin risiede nella loro capacità di creare un ponte tra il mondo fisico e quello virtuale, offrendo un ambiente

simulato in cui testare ipotesi, simulare, prevedere esiti, e guidare le decisioni senza interferire direttamente con l'oggetto fisicoi [21]. Questo aspetto si rivela particolarmente vantaggioso per la gestione del ciclo di vita dei prodotti, dalla progettazione alla manutenzione, fino al ritiro dal servizio [22], ma anche per l'ottimizzazione di sistemi complessi come le reti energetiche e idriche, i sistemi di trasporto, e le infrastrutture urbane [56].

Il concetto di Digital Twin è stato formalizzato nei primi anni del 2000 da Michael Grieves, anche se trova la sue radici in precedenti lavori ed idee legate alla modellazione e simulazione di sistemi complessi. Ad esempio la NASA già negli anni '60 per le missioni spaziali, ha sviluppato modelli simulati di navicelle spaziali per testare e valutare le performance in scenari critici senza rischiare la sicurezza delle missioni e degli astronauti.

Ai giorni d'oggi, con la crescita delle tecnologie di IoT, analisi di big data e cloud computing, la capacità di sviluppare e utilizzare Digital Twins è cresciuta esponenzialmente, trasformando questo strumento in una componente fondamentale per l'innovazione in numerosi settori. Infatti dalle semplici repliche di oggetti fisici, i Digital Twins sono diventati sistemi sempre più complessi capaci di simulare intere catene di produzione, infrastrutture urbane e addirittura città intere, offrendo un potenziale inesplorato per la gestione ottimale delle risorse, la pianificazione urbana e la sostenibilità ambientale.

1.2 Digital Twin Urbani

Tra le tante, una delle applicazioni più promettenti e rivoluzionarie del concetto di Digital Twin riguarda la creazione di Digital Twin Urbani. Un Digital Twin Urbano è una replica digitale e dinamica di una città o parti di essa, che integra dati real time e non provenienti da una fitta rete di sensori distribuiti nell'ambiente urbano. Questi dati includono, ma non si limitano a, informazioni su traffico, utilizzo dell'energia, qualità dell'aria, infrastrutture e servizi pubblici. Un Digital Twin Urbano fa ingestione di questi dati per fornire agli urbanisti, ai decisori politici e ai cittadini strumenti avanzati per la pianificazione e la gestione ottimale della città.

La specificità degli UDT risiede nella loro capacità di rappresentare in modo accurato e multidimensionale l'ecosistema urbano, consentendo analisi dettagliate su come vari fattori possono interagire tra di loro e influenzano la

vita cittadina. Ad esempio i UDT, attraverso tecnologie avanzate come l'IA per l'elaborazione e l'analisi dei dati, possono prevedere l'impatto di cambiamenti infrastrutturali, politiche urbane, eventi eccezionali (come pandemie o disastri naturali) e tendenze di sviluppo a lungo termine.

Nell'ambito dell'ambiente un UDT gioca un ruolo cruciale nella promozione della sostenibilità ambientale e nella mitigazione dei cambiamenti climatici. Attraverso simulazioni dettagliate è possibile ottimizzare l'uso delle risorse quindi di ridurre le emissioni di carbonio migliorando la resilienza urbana agli impatti climatici, guidando così verso una transizione ecologica della città.

Un aspetto degli UDT, non sempre scontato, è la possibilità di facilitare la partecipazione cittadina e la governance collaborativa. Fornendo una piattaforma visuale ed interattiva, i cittadini possono essere coinvolti attivamente nella pianificazione e nella gestione della città, contribuendo con feedback e proposte. Questo non solo migliora la trasparenza e l'efficacia delle decisioni urbane ma promuove anche un maggiore senso di appartenenza e responsabilità tra gli abitanti.

In sintesi, il UDT rappresenta una potente leva per la trasformazione delle città in ambienti più vivibili, sostenibili ed inclusivi. Esso simboleggia l'incarnazione della visione di una "smart city", dove la tecnologia e dati vengolo utilizzati strategicamente per migliorare la qualità della vita urbana, preservare le risorse naturali e costruire comunità resilienti ed adattabili ai cambiamenti futuri [3].

1.3 Stato dell'arte dei Digital Twin

Negli ultimi 10 anni, l'adozione di Urban Digital Twin è cresciuta in modo esponenziale, con numerose città che hanno abbracciato questa tecnologia per migliorare la pianificazione urbana. Alcune delle implementazioni più avanzate sono esempi lampati di come queste soluzioni possono trasformare la vivibilità e la sostenibilità delle città.

1.3.1 Architettura

La tecnologia dei digital twin urbani richiede una convergenza di varie fonti di dati, strumenti analitici e tecniche di visualizzazione per creare una replica

digitale in tempo reale degli ambienti urbani. L'architettura di un digital twin urbano può essere strutturata in diversi strati e componenti chiave ognuno dei quali svolge funzioni distinte operando al contempo in modo integrato.

In Fig. 1.1 è mostrata una rappresentazione semplificata dell'architettura di un Digital Twin Urbano.

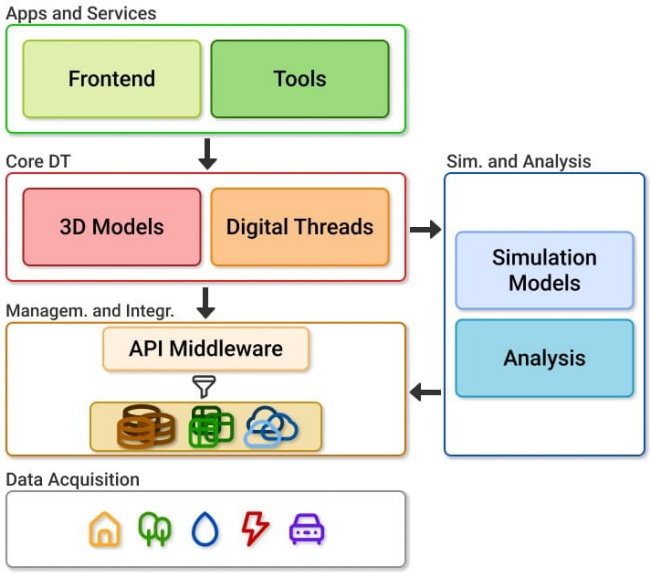


Figura 1.1: Architettura semplificata di un DT

Layer 1: Acquisizione Dati

La base di qualsiasi digital twin urbano sono i dati utilizzati per replicare l'ambiente reale. Questo strato coinvolge la raccolta di dati da varie fonti, tra cui:

- **Sensori e dispositivi IoT**: dislocati in tutta la città per raccogliere dati su traffico, qualità dell'aria, condizioni meteorologiche, consumo energetico, e altro;
- **Dati Geospaziali**: comprendono mappe, immagini satellitari, dati LiDAR, e altre forme di dati spaziali che forniscono il layout fisico e le caratteristiche dell'ambiente urbano;
- **Dati Amministrativi**: comprendono dati demografici, dati sulle infrastrutture, dati sulla mobilità, e altri dati.

Layer 2: Gestione ed Integrazione Dati

Questo strato gestisce l'archiviazione, l'elaborazione e l'integrazione dei dati raccolti. Garantisce che il digital twin abbia accesso a set di dati tempestivi, accurati e completi. Componenti chiave includono:

- **Data Lakes/Data Warehouses**: permettono l'archiviazione dell'enorme quantità di dati strutturati e non strutturati;
- **Framewrok di Elaborazione**: strumenti ed algoritmi che si occupano di pulire, trasformare, analizzare ed integrare i dati provenienti da fonti eterogenee;
- **API e Middleware**: facilitano lo scambio di dati tra sistemi e componenti diversi del DT.

Layer 3: Simulazione e Analisi

Un insieme di modelli computazionali ed algoritmi che si occupano di simulare i processi urbani ed analizzare i dati, includendo:

- **Modelli di Simulazione**: modelli matematici e fisici che simulano il comportamento di vari aspetti della città;
- **Analisi di Dati**: strumenti e tecniche per analizzare i dati, indentificare i modelli e generare osservazioni. Basandosi su modelli di machine learning si possono prevedere tendenze future.

Layer 4: Nucleo del DT

Uno strato centrale che rappresenta il cuore del Digital Twin, integrando dati e osservazioni dei livelli precedenti per costruire ed aggiornare la replica digitale dell'ambiente urbano. Questo strato include:

- **Modelli e Visualizzazione 3D**: modelli 3D ad alta fedeltà che rappresentano l'ambiente urbano e le sue caratteristiche;
- **Digital Threads**: flussi continui di dati che collegano ambienti digitali e fisici, aggiornando il Digital Twin in tempo reale.

Layer 5: Applicazioni e Servizi

Posto al di sopra del nucleo, fornisce applicazioni e servizi specifici per gli utenti finali. Tradice i complessi dati e i modelli derivanti in osservazioni e interfacce utente di semplice comprensione, includendo:

- **Strumenti di Pianificazione Urbana**: per la pianificazione di scenari, zonizzazione e sviluppo infrastrutturale;
- **Sistemi di Risposta alle Emergenze**: utilizzando i dati in tempo reale permettono di ottimizzare sia i tempi che la qualità delle risposte a disastri ed emergenze;
- **Piattaforme per i Cittadini**: permettono di coinvolgere i residenti facendoli interfacciare con il Digital Twin.

1.3.2 Esempi di Digital Twin Urbani

Di seguito alcuni esempi di Digital Twin Urbani sviluppati.

- **Shangai** (Fig. 1.2): rappresenta il Digital Twin più evoluto al momento, è stato sviluppato dall'azienda cinese 51World [1], azienda leader nella costruzione di gemelli digitali. Include un'area di circa 3750 km quadrati ed utilizza Unreal Engine 5 come motore di visualizzazione. Per la modellazione della città sono stati utilizzati satelliti, droni, sensori. Per quanto riguarda i dati sensoristici IoT ed altro sono presenti poche informazioni, molto probabilmente per gli edifici sono stati utilizzati dati di tipo BIM.

Figura 1.2: Digital Twin della città di Shangai

- **Wellington**[9] (Fig. 1.3): sviluppato dall'azienda Buildmedia. Anch'esso è stato sviluppato in Unreal Engine 5 e attraverso dati real time fornisce:
  - statistiche in tempo reale sul traffico, nel dettaglio bus, treni, traghetti, auto e biciclette;

- visualizzazione del traffico aereo
- disponibilità di parcheggi per automobili

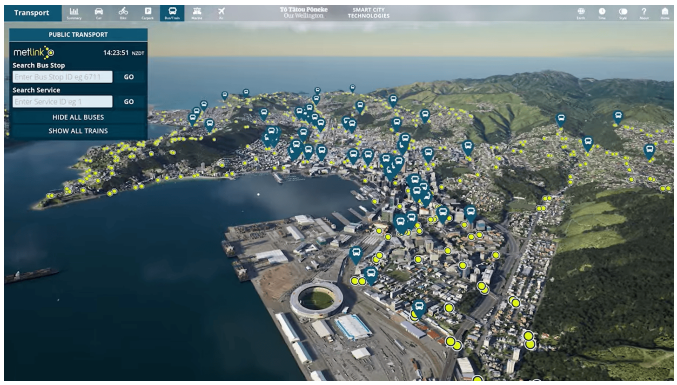


Figura 1.3: Digital Twin della città di Wellington

- **Herrenberg** (Fig. 1.4): piccola città Tedesca di circa 30000 abitanti. Hanno sviluppato un Digital Twin [55] molto avanzato utilizzando dati molto eterogenei, per la modellazione 3D sono stati utilizzati i LiDAR e i dati BIM. Inoltre i loro dataset contengono dati sul modello stradale, pattern di movimento dei cittadini, qualità dell'aria e urban emotions. Per le simulazioni si basano sulla computazione su cluster HPC per avere delle predizioni accurate sul traffico e il flusso del vento.

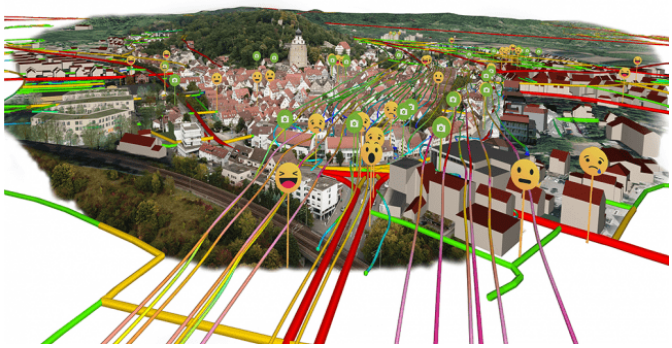


Figura 1.4: Digital Twin della città di Herrenberg

- **Perugia** (Fig. 1.5): presentato nel 2023 e sviluppato dall'azienda Wisetown [54], sempre in Unreal Engine 5. Rispetto agli altri digital twin è meno incentrato sulla resa visiva in quanto non vi è alcuna resa

fotorealistica degli edifici ma si concentra di più sui dati sul verde e facilita l'interazione tra il comune e il cittadino [27]. Permette di:

- visualizzazione della distribuzione demografica della città, anche a livello di edificio;
- analisi sul verde urbano per monitorare la salute della vegetazione
- issue tracking tramite il quale i cittadini possono aprire ticket sul malfunzionamento delle infrastrutture cittadine

Figura 1.5: Digital Twin della città di Perugia

L'esplorazione dello stato dell'arte dei Digital Twin urbani attraverso gli esempi di Shangai, Wellington, Herremberg e Perugia offre una panoramica sul potenziale di questa tecnologia per le città di tutto il mondo. Ogni caso di studio presenta una visione unica di come i Digital Twin possono essere utilizzati per affrontare sfide urbane specifiche, dalla gestione della mobilità e della pianificazione urbana alla mitigazione dei cambiamenti climatici e all'incremento della patercipazione civica.

Dai casi esaminati emergono alcune lezioni fondamentali che saranno poi ampiamente discusse nei capitoli successivi:

- **L'importanza dei dati**: la qualità, la quantità e l'accessibilità dei dati sono cruciali per lo sviluppo e l'efficacia dei Digital Twins. La sfida di integrare dati provenienti da sorgenti diverse richiede soluzioni innovative in termini di standardizzazione e protezione della privacy;
- **Adattabilità e scalabilità**: i Digital Twins devono essere progettati per essere adattabili e scalabili, in modo da poter evolvere in risposta

ai cambiamenti delle città e delle loro necessità. La flessibilità è fondamentale per garantire che questi strumenti rimangano pertinenti nel tempo.

Capitolo 2

Bologna Digital Twin

Contents

2.1 Obiettivi del progetto 11
2.2 Analisi di Progetto 12
2.2.1 Raccolta Dati 13
2.2.1.1 OpenData Comune di Bologna 13
2.2.1.2 Rilievi aerei 18
2.2.2 Modellazione geometrica 3D del paesaggio 25
2.2.3 Il problema del texturing 28
2.2.4 Motore grafico di visualizzazione 29
2.2.5 Integrazione di dati real time 31
2.3 Workflow di elaborazione dati 32
2.4 Tool e Software utilizzati 33

2.1 Obiettivi del progetto

L'obiettivo principale di questo progetto riguarda la costruzioen di un Digital Twin della cittò, che possa essere utile come strumento per il supporto alla pianificazione urbana, l'analisi ambientale (effettuando simulazioni per prevedere comportamenti sul cambiamento climatico), il monitoraggio del traffico e l'analisi del verde urbano.

Il ruolo che ci è stato affidato in questo progetto è la realizzazione dell'ambiente 3D di Bolgona, partendo dai dati forniti, con un'attenzione particolare su alcuni aspetti chiave. Questi includono l'estrazione di informazioni 3D dai

dati LiDAR, la creazione di layer di visualizzazione, l'analisi del verde urbano e l'integrazione di fonti di dati real-time.

Il nostro approccio sarà strettamente di tipo data-driven in quanto è necessario fondare le nostre decisioni progettuali sui dati reali della città.

Durante la realizzazione di questo progetto ci siamo potuti confrontare con altri dipartimenti di sviluppo e abbiamo tratto da queste conversazioni gli obiettivi principali del nostro progetto.

2.2 Analisi di Progetto

La costruzione di un Digital Twin 3D della città di Bologna presenta una serie di sfide tecniche e logistiche. Queste sfide derivano principalmente dalla natura dei dati disponibili e dalla mancanza di altri tipi di dati.

Quello che si vorrebbe ottenere, anche in base all'analisi dello stato dell'arte dei Digital Twins già esistenti, sarebbe un ambiente 3D completamente navigabile dell'intera città, nel quale devono essere presenti diversi layer di visualizzazione dipendenti dal tipo di dettaglio che si desidera. Inoltre un ruolo importante lo svolge la possibilità di interagire con i vari elementi presenti nel DT. Questo permetterebbe agli utenti di ottenere informazioni dettagliate ad esempio su: edifici, alberi, strade, e altri elementi urbani semplicemente cliccando su di essi nel modello 3D. Un altro aspetto cruciale è la capacità di aggiornare e mantenere il DT nel tempo. Poiché la città è un organismo in continua evoluzione, è fondamentale che il DT possa riflettere queste modifiche in modo accurato e tempestivo. Questo richiederà l'implementazione di procedure efficienti per l'aggiornamento dei dati e la loro integrazione del modello 3D. Infine, la creazione di un DT 3D dovrebbe tenere in considerazione le esigenze dei vari stakeholder, tra cui urbanisti, ingegneri, amministratori pubblici e cittadini. Pertanto è fondamentale sviluppare un'interfaccia utente intuitiva e accessibile che permetta a tutti di esplorare ed interagire con il DT 3D.

Quindi riassumendo, i requisiti fondamentali nella costruzione del Digital Twin della città di Bologna sono i seguenti:

- **Immersività**: il DT deve essere quanto più possibile fedele alla città, questo requisito è raggiungibile utilizzando mesh e texture ad alta risoluzione e ovviamente un modello d'illuminazione realistico che rispecchi quello della città.

- **Performance**: è molto importante, ai fini della navigabilità e l'utilizzo che le performance del prodotto siamo sempre alte, mantenendo alto e stabile il numero di *frames al secondo*.
- **Data-Driven**: questo è il requisito forse più importante, dato che l'obiettivo è costruire un Urban Digital Twin è di primaria importanza che la costruzione dello stesso modello 3D e tutte le visualizzazioni che vi ruotano intorno si basino sui dati reali della città.

Qui di seguito verrà fatta una breve analisi del problema individuando e organizzando i problemi base alla loro natura.

Alla luce dell'esplicazione di tutti i tipi di dato in nostro possesso, la prima sfida che si presenta è proprio la costruzione del modello 3D dell'intera città. Innanzitutto come si è potuto intuire gli unici elementi che possono essere intese come mesh sono i modelli del DTM. Invece tutti gli altri tipi di dati hanno bisogno di processing per ottenere modelli tridimensionale.

2.2.1 Raccolta Dati

I dati fornitici provengono dal **Comune di Bologna** in due distinte sorgenti:

- **Bologna Open Data** [6]
- **Rilievi aerei**

Bologna Open Data è un progetto del Comune di Bologna che mira a rendere accessibili al pubblico una serie di dati in formato aperto (Open). L'obiettivo è quello di permettere a cittadini, aziende e associazioni di utilizzare e valorizzare questi dati. Il catalogo presente sul portale è in costante aggiornamento.

Inoltre ci sono stati forniti dei dataset risultanti dall'ultima serie di **rilievi aerei** commissionati dal Comune di Bologna. Questi dati sono stati forniti a Cineca in seguito all'accordo per lo sviluppo del Digital Twin, sono tutt'ora caricati sul supercomputer Leonardo e contengono dati **LiDAR** e **fotografici**.

2.2.1.1 OpenData Comune di Bologna

Alcuni esempi dei dati disponibili includono la rilevazione del flusso di veicoli, le posizioni delle fermate dei mezzi pubblici, i flussi di biciclette in vari punti

della città e persino il registro dei defibrillatori. Inoltre, sono disponibili dati climatici come temperature, precipitazioni e le misurazioni della qualità dell'aria.

Tra di essi troviamo anche alcuni dati per noi essenziali ai fini di questo progetto, essi sono **edifici volumetrici** e **alberi in manutenzione**.

Edifici volumetrici

Gli edifici volumetrici fanno parte della **Carta Tecnica Comunale** (CTC) (Fig. 2.1), una cartografia tecnica in scala 1:200 che il Sistema Informativo Territoriale (SIT) del Comune di Bologna ha fatto realizzare nel 2001 tramite il metodo fotogrammetrico diretto. In questa restituzione sono state inserite anche la **Quota di Gronda** e la **Quota al Piede** degli edifici, con una tolleranza di 54 cm.

L'aggiornamento continuo della CTC avviene utilizzando fonti primarie come progetti edilizi e fonti secondarie come ortofoto di precisione, librerie di immagini oblique e dati LiDAR aerei come quelli fornitici.

Il **Sistema di Riferimento** utilizzato è il ***WGS84*** [53], in formato *Gradi Decimali*, e sono tutt'ora presenti 66095 record in totale.

Nello specifico, abbiamo un dataset tabellare con i seguenti dati:

- **CODICE_OGGETTO** (int): un codice univoco per ciascun edificio.
- **DATA_ISTITUZIONE** (date): data dell'inserimento nel CTC.
- **DATA_VARIAZIONE** (date): data dell'ultima modifica.
- **NOTEOGG** (string): testo con eventuali note.
- **PERIMETRO** (int): lunghezza del perimetro.
- **DESCRIZIONE** (string): descrizione/classe tipologia dell'edificio.
- **ORIGINE** (string): quale tipo di dato è stato consultato per l'inserimento.
- **QUOTA_GRONDA** (double): quota media della gronda sul livello del mare.
- **QUOTA_PIEDE** (double): quota media del piede sul livello del mare.

- **ALTEZZA_GRONDA** (double): differenza tra le quote di gronda e di piede.
- **AREA_OGG** (double): area dell'edificio.
- **VOLUME** (double): volume dell'edificio.
- **Geo Point** (geo_point_2d): coordinate geografiche (latitudine e longitudine) del centroide dell'edificio.
- **Geo Shape** (geo_shape): coordinate geografiche (latitudine e longitudine) che definiscono la forma poligonale dell'edificio.

Si nota che anche se viene utilizzato un double per i valori di quota la precisione massima al momento apprezzabile sul dataset è al decimetro.

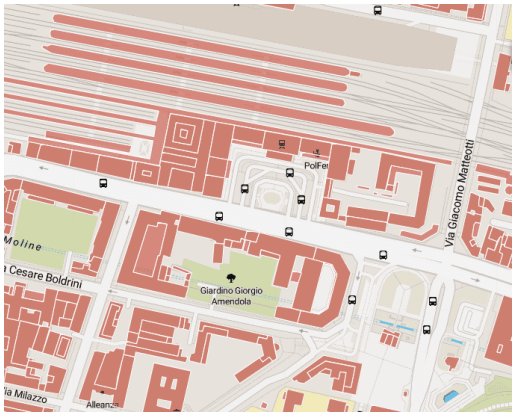


Figura 2.1: Open Data: edifici volumetrici (zona stazione)

Alberi in manutenzione

Il dataset degli Alberi in manutenzione del Comune di Bologna contiene i dati sugli alberi soggetti a manutenzione all'interno del territorio comunale di Bologna. Sono quindi presenti i soli esemplari arborei rilevati attraverso il sistema di gestione della relativa manutenzione. Sono tutt'ora presenti 85043 record in totale. Le coordinate delle loro posizioni rispettano lo stesso formato precedentemente descritto (WGS84 in Gradi Decimali)

Nello specifico, questo dataset tabellare presenta i seguenti dati:

- **Progressivo pianta** (string): un codice univoco per ciascuna pianta.

- **Codice albero** (string): un codice non univoco che può rappresentare gruppi di piante.
- **Pianta isolata** (string): una stringa binaria (S/N) che indica se l'albero è isolato o meno.
- **Specie arborea** (string): specie della pianta.
- **Classe di altezza** (string): classe di altezza della pianta, divisa nelle seguenti categorie:
  - Cl1: <6mt
  - Cl2: 6mt - 12mt
  - Cl3: 12mt - 16mt
  - Cl4: 16mt - 23mt
  - Cl5: >23mt
- **Classe circonferenza (diametro)** (string): classe di circonferenza della pianta, divisa nelle seguenti categorie:
  - Cl1: <15 (<5 cm)
  - Cl2: 15 - 30 (5-10 cm)
  - Cl3: 30 - 45 (10-15 cm)
  - Cl4: 45 - 60 (15-19 cm)
  - Cl5: 60 - 90 (19-28 cm)
  - Cl6: 90 - 110 (28-35 cm)
  - Cl7: 110 - 140 (35-45 cm)
  - Cl8: 140 - 170 (45-54 cm)
  - Cl9: 170 - 200 (54-64 cm)
  - Cl10: 200 - 230 (64-73 cm)
  - Cl11: 230 - 260 (73-80 cm)
  - Cl12: >260 (>80 cm)
- **Dimora** (string): indica la tipologia di dimora nella quale è inserita la pianta, questa può essere delle seguenti categorie:
  - Prato
  - Terra
  - Fioriera
  - Formella (e sotto tipologie)
- **Irrigazione** (string): una stringa binaria (S/N) che indica se l'albero è irrigato.
- **Albero di pregio** (string): una stringa binaria (S/N) che indica se l'albero è di pregio o meno.

- **Data inserimento inventario** (date): data dell'inserimento nell'inventario.
- **Data ultimo aggiornamento** (date): data dell'ultima modifica.
- **Distanza dai fabbricati** (string): indica la distanza dai fabbricati della pianta, può ricadere nelle seguenti categorie:
  - Da 0 a 3mt
  - Da 3mt a 6mt
  - Oltre 6mt
- **Data impianto** (date): data in cui è stato piantato l'albero.
- **Geo Point** (geo_point_2d): coordinate geografiche (latitudine e longitudine) del centro del tronco dell'albero.
- **Geo Shape** (geo_shape): coordinate geografiche (latitudine e longitudine) dello shape dell'albero (punto al centro del tronco).
- **Campagna** (string): indica in quale campagna di rimboscamento appartiene.
- **IN_PATRIM** (string): una stringa binaria (0/1) che indica se l'albero è in patrimonio o meno.
- **Zone di prossimità** (string): indica la zona alla quale appartiene l'albero.
- **Area statistica** (string): indica una zona più specifica alla "Zona di prossimità".

Le date di piantumazione sono presenti solo in maniera parziale e sono state inserite in inventario dal 2014 in poi.

Attraverso il sito degli OpenData è possibile vedere la distribuzione degli alberi in manutenzione all'interno del comune di Bologna, come si può vedere in Fig. 2.2.

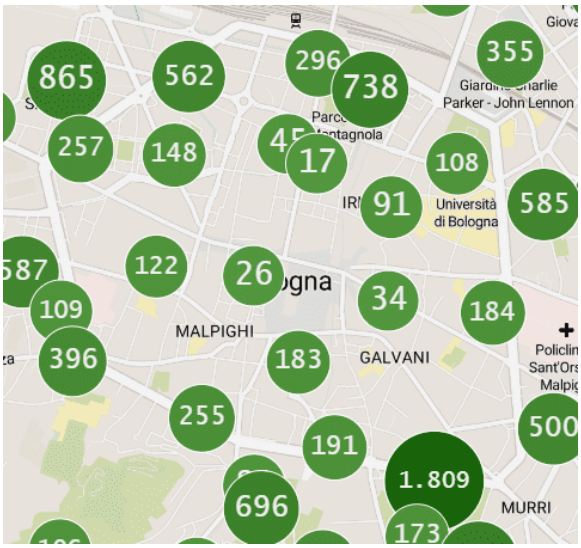


Figura 2.2: Open Data: visualizzazione distribuzione alberi

2.2.1.2 Rilievi aerei

I dati risultanti da questi rilievi sono di vario tipo, principalmente si dividono in derivati da pointcloud LiDAR e fotografie.

Dati LiDAR

La tecnologia **LiDAR** (Light Detection and Ranging) è un metodo di rilevamento remoto che misura le distanze tra il sensore e gli oggetti tramite un laser pulsato. Questa tecnologia è ampiamente utilizzata in numerose applicazioni come la mappatura topografica, la forestazione, l'urbanistica e recentemente la creazione di modelli 3D per creare i gemelli digitali di città.

Spesso i dati ottenuti durante la rilevazione richiedono una fase di elaborazione prima di essere effettivamente utilizzati. Questi processi includono principalmente processi come la georeferenziazione, la triangolazione aerea e la classificazione dei punti.

Nel nostro caso questi dati sono archiviati all'interno del supercomputer **Leonardo** [46], al momento 2° supercomputer più potente in Europa e 6° al mondo, ospitato al tecnopolo di Bologna e gestito dal consorzio interuniversitario CINECA. Abbiamo quindi fatto richiesta di accesso ad esso per il nostro progetto e, dopo la avvenuta approvazione dei nostri username, siamo stati aggiunti al gruppo risorse contenente i dati. Tramite connessione **SSH**

abbiamo inizialmente trasferito i nostri dati alle nostre directory remote e quindi li abbiamo ottenuti utilizzando **FTP**.

I dati presenti nel nostro dataset sono i seguenti:

	Dimensione	**Num. tot. file**	**Formato**	**Tipo di dato**
Raw Data	947 G	1738	.las	Dati LiDAR senza classificazione
Nuvola Classificata	198 G	654	.las	Dati LiDAR con classificazione
DSM	4.1 G	654	.asc	Digital Surface Model
DTM	4.1 G	654	.asc	Digital Terrain Model

Tabella 2.1: Tabella dati LiDAR

Con **Raw Data** si intendono i dati raccolti direttamente da strumentazione. Essi vengono memorizzati in un formato file denominato **LAS** (LASer) [45], che è un open standard specificato dalla American Society for Photogrammetry and Remote Sensing (ASPRS). Questo formato binario è progettato per l'archiviazione delle nuvole di punti ottenute durante i rilievi e supporta la rappresentazione di dati 3D con l'aggiunta di altri attributi associati ad ogni punto.

Purtroppo l'unico documento in nostro possesso riguardante questi dati illustra unicamente gli strumenti utilizzati e le metodologie con i quali i dati di partenza (Raw Data) sono stati rilevati, tralasciando ogni informazione sul processing utilizzato per ottenere i dati finali, suddivisi in:

- **Nuvola Classificata**
- **Digital Surface Model** (DSM) [13]
- **Digital Terrain Model** (DTM)

Questi file sono stati organizzati in modo da dividere il comune di Bologna in una griglia, suddividendolo in **Tile** di dimensione 500x500m. Come si può capire dalla tabella, questa suddivisione genera in totale 654 tile organizzati nel seguente modo.

Come si può notare dalla Fig. 2.3 le nuvole di punti seguono i confini del Comune di Bologna, perciò sono presenti unicamente i punti con coordinate all'interno di esso. Di conseguenza, i tile di confine non sono tutti di forma quadrata, anche se il nome rimane in riferimento all'LLC come se il tile fosse riempito interamente.

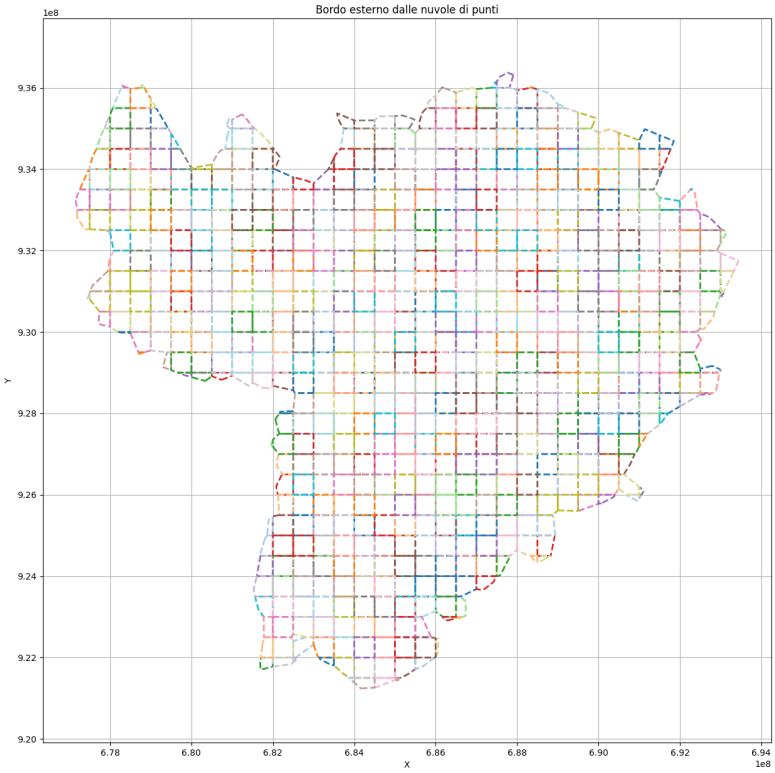


Figura 2.3: Comune di Bologna: suddivisione in tile 500x500

La convenzione di denominazione dei nomi di questi file deriva dalle coordinate dell'angolo in basso a sinistra, ovvero l'**LLC** (Lower Left Corner), in formato **UTM (WGS84)**. Questo sistema divide la terra in 60 bande longitudinali, rappresentate con numeri, ognuna di esse suddivisa a sua volta in 20 bande latitudinali, rappresentate con lettere. All'interno di queste zone le coordinate dei punti vengono rappresentate utilizzando valori metrici suddivisi in Est (E) e Nord (N).

Nel nostro caso, Bologna ricade nel quadrante 32T, perciò i nostri file sono nominati con la stringa **"32_E_N"** (la T viene omessa) dove **E** e **T** sono appunto le coordinate dell'LLC del nostro tile. Per esempio, il file contenente i dati su Piazza Maggiore avrà il nome *32_686000_4929000*.

Andando nel dettaglio, i file della **Nuvola Classificata** sono nuvole di punti non organizzate contenute in file di tipo LAS (Fig. 2.4). Oltre ai campi base possiamo trovare anche diversi campi calcolati in un momento successivo rispetto al rilievo. In particolare possiamo trovare il campo di

Classificazione (*classification*) nel quale si determina se quel punto appartiene o meno al ground. Anche in questo caso non sappiamo come questa classificazione è avvenuta, tuttavia sembra essere una stima con una buona attendibilità.

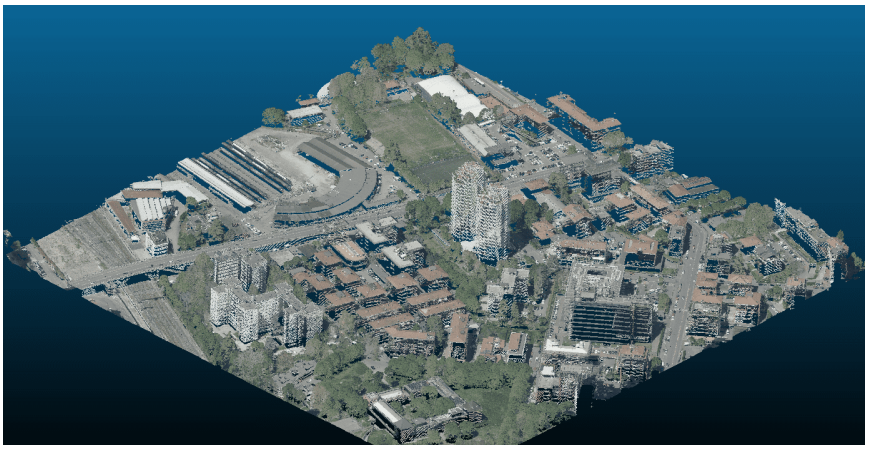


Figura 2.4: File LAS visualizzato su CloudCompare (32_687000_4930500)

Ogni file di questo tipo presenta **dai 9 ai 10 milioni** di punti, avendo quindi una densità che si aggira intorno ai **38 punti al m^2**.

Per ogni punto abbiamo le seguenti informazioni:

Successivamente sono stati creati anche i file del **Digital Surface Model (DSM)** e del **Digital Terrain Model (DTM)**. Il primo di essi contiene punti appartenenti a qualsiasi classificazione, mentre il secondo si basa solamente sui punti classificati come terreno (Ground).

Questi file sono di tipo binario rappresentati nel formato ASCII Grid (.asc) e contengono nuvole di punti molto più sparse ma organizzate in una griglia 1000x1000, avendo quindi un punto ogni 0.5m. In questo formato vengono mantenuti solo i valori di posizione, andando a specificare solo la coordinata z per ogni punto, permettendo di avere file di piccole dimensioni.

Dati fotografici

I dati fotografici che ci sono stati passati sono foto aeree scattate durante i rilievi LiDAR (**Oblique**) e **Ortofoto** aeree geometricamente corrette e georeferenziate.

Le foto *oblique* (Fig. 2.5) sono state ottenute da 4 camere inclinate a 45° nelle quattro direzioni rispetto all'andamento dell'aeromobile (.jpg) e sono ad altissima risoluzione (14192x10640, 240 dpi).

Dimensione	Descrizione
x, y, z	Coordinate (UTC WGS84)
intensity	Misura dell'intensità di ritorno dell'impulso laser
return_number	Numero di ritorno dell'impulso per un dato impulso
number_of_returns	Numero totale di ritorni per un dato impulso
synthetic	Se un punto è stato creato con un metodo diverso dalla raccolta lidar, ad esempio digitalizzato da un modello stereo fotogrammetrico
key_point	Se un punto è considerato un punto-chiave del modello e non deve essere traslaciato in nessun algoritmo di assestamento
withheld	Se il punto non deve essere incluso nell'elaborazione
overlap	Se un punto si trova all'interno della regione di sovrapposizione di due o più linee di volo o strisciate
scanner_channel	Il canale (testa dello scanner) di un sistema multicanale
scan_direction_flag	La direzione in cui viaggiava lo specchio dello scanner al momento dell'impulso di uscita
edge_of_flight_line	Ha valore 1 solo quando il punto si trova alla fine di una scansione
classification	Può essere 1 (Unassigned) o 2 (Ground)
user_data	Può essere utilizzata a discrezione dell'utente per dati extra
scan_angle	Angolo di uscita del punto laser dal sistema laser (compreso il rullo dell'aeromobile)
point_source_id	Il file da cui proviene questo punto
gps_time	Il valore del time tag a virgola mobile doppia in cui il punto è stato acquisito
red, green, blue	Colore del punto
nir	La regione del vicino infrarosso dello spettro

Tabella 2.2: Descrizione dei dati nel file LAS.

Ad esse si aggiunge l'immagine nadir (.tif) che inquadra il terreno al momento dello scatto, presentano la stessa dimensione delle loro compagne ma hanno una risoluzione diversa (72x72).

Per quanto riguarda la convenzione di denominazione di queste immagini, esse presentano nomi particolari in cui è inclusa la data di acquisizione, alcuni numeri che indicano in modo vago la posizione di acquisizione e il tipo di orientamento della camera.

Basandoci sul numero totale dei file, possiamo dedurre che abbiamo all'incirca 7 quintuple di immagini per ogni nostro tile.

Le *Ortofoto* (Fig. 2.6) sono invece immagini di formato .tif ad altissima risoluzione (20802x20802) scattate in modo ortogonale al terreno. Ognuna

	Dimensione	Num. tot. file	Formato	Tipo di dato
Oblique	3.6 T	4749	.jpg	right
		4788	.jpg	left
		4826	.jpg	back
		4811	.jpg	forward
		4651*	.tif	nadir**
Ortofoto	658 G	199***	.tif	Tavole CIR
		199***	.tif	Tavole RGBN

* per ogni file .tif ci sono i corrispondenti file .dbf, .prj, .shp, .shx, .tfw, .tif
** punto del terreno che si trova verticalmente sotto il centro prospettico dell'obiettivo della fotocamera o dei rilevatori dello scanner
*** per ogni file .tif ci sono i corrispondenti file .tfw e .prj

Tabella 2.3: Tabella dati fotografici

Figura 2.5: Esempio foto obliqua

di queste foto ci fornisce una visualizzazione di vasta area di territorio, poco più di $\mathbf{2km^2}$. Nel seguente esempio possiamo vedere parte del quartiere Saffi.

Per capire la convenzione di denominazione di queste immagini dobbiamo prima capire come esse sono organizzate. Ognuna di esse appartiene a una quadrupla disposta come in Fig. 2.7.

Come si può notare che c'è una leggera sovrapposizione tra i bordi delle immagini. Oltre a questo il punto di riferimento (Ref. point) non è all'esatto

Figura 2.6: Esempio ortofoto

angolo ma all'LLC del rettangolo interno che arriva a metà del bordo di sovrapposizione.

Quindi, la convenzione di denominazione utilizzata utilizza la zona (sempre 32), le prime 4 cifre delle coordinate Est e Nord del punto di riferimento e il la posizione dell'immagine all'interno del gruppo per denominarla. Nel nostro caso la ortofoto visualizzata in figura 2.6 avrà il nome *32_68404930_4.tif*.

Queste ortofoto sono quindi suddivise in due cartelle in base alla tipologia di dati rappresentati:

- **RGBN**: a quattro canali (8 bit ognuno), rispettivamente Red-Green-Blue-Near Infrared.
- **CIR**: a tre canali (8 bit ognuno), rispettivamente Near Infrared-Red-Green (Fig. 2.8).

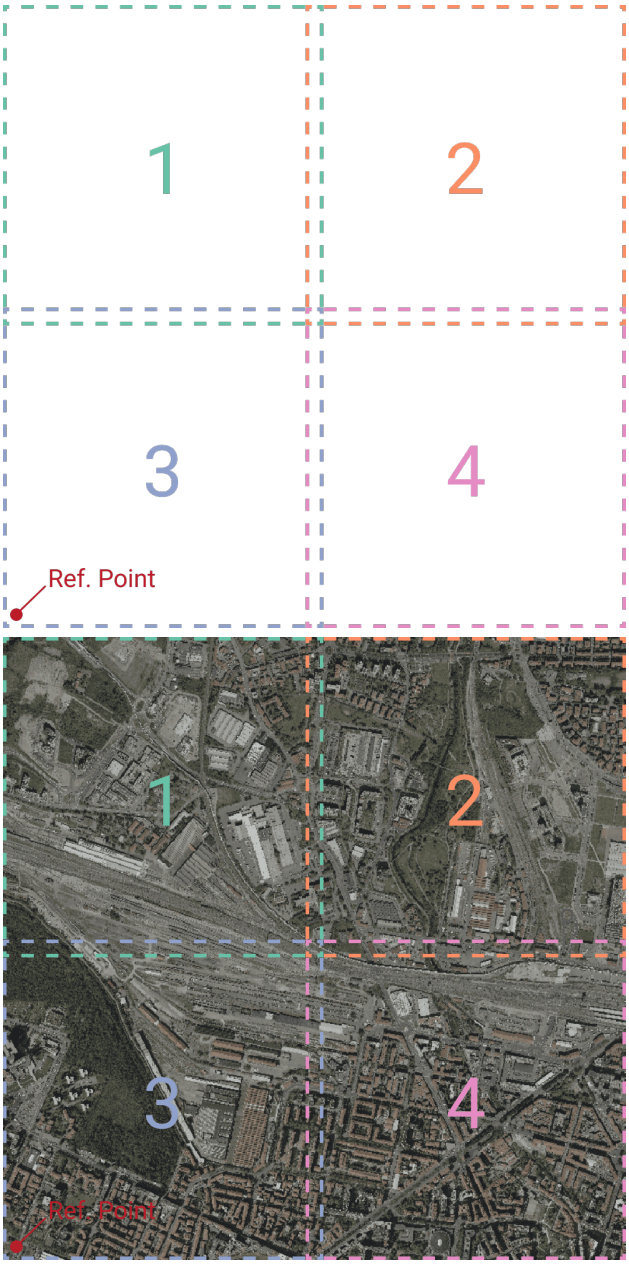


Figura 2.7: Esempio disposizione ortofoto

2.2.2 Modellazione geometrica 3D del paesaggio

La sfida più grossa risulta appunto l'ottenimento del modello 3D degli edifici della città.

Innanzitutto data la natura delle rilevazioni dei dati LiDAR, ovvero cat-

Figura 2.8: Esempio ortofoto (versione CIR)

turate dall'alto tramite aereo, vi è la completa mancanza delle facciate verticali nella nuvola di punti. Questo significa che, dalla nuvola di punti, si hanno solo informazioni inerenti alle superfici orizzontali con qualche sporadica e mal distribuita informazione sulla verticalità. La necessità di effettuare una serie di elaborazioni dei dati LiDAR emerge quindi come un passo critico per ottenere le mesh degli edifici (Fig. 2.9).

Integrazione con altre fonti di dati

Per superare la limitazione dei dati LiDAR, l'integrazione con altre fonti di dati geospaziali emerge come soluzione chiave. L'uso combinato ad esempio di immagini satellitari, ortofoto o file con informazione sugli edifici potrebbe compensare la mancanza di informazioni, offrendo una visione più completa e dettagliata degli edifici urbani. Questo approccio multidimensionale potrebbe permettere di arricchire le informazioni in nostro possesso facilitando la ricostruzione di modelli 3D accurati.

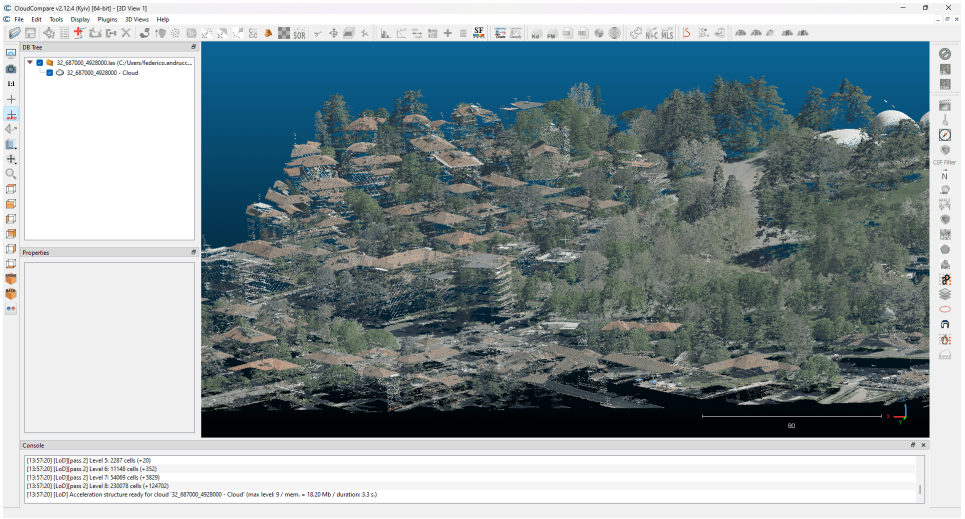


Figura 2.9: Nuvola di punti LiDAR visualizzata in CloudCompare

Utilizzo di tecniche di Computer Vision

L'utilizzo di tecniche avanzate di Image Processing e Computer Vision potrebbe essere un'altra strategia fondamentale con cui poter trattare le nuvole di punti.

Gestione della complessità computazionale

La gestione efficace della complessità computazionale è essenziale per garantire che i modelli 3D siano non solo dettagliati ma anche ottimizzati per il rendering e la simulazione in tempo reale. Tecniche come la riduzione poligonale e l'implementazione di livelli di dettaglio (LOD) permettono di equilibrare il dettaglio visivo con le prestazioni, assicurando che il modello 3D sia fruibile su dispositivi con differenti capacità di calcolo.

Standard e formati di modellazione

L'aderenza a standard e formati di modellazione riconosciuti internazionalmente garantisce che il modello 3D possa essere facilmente integrato in applicazioni esistenti, promuovendo l'interoperabilità e la collaborazione tra differenti piattaforme e strumenti.

2.2.3 Il problema del texturing

Uno dei problemi più importanti risiede nelle informazioni visive delle mesh degli edifici della città, quindi nel Texturing. Questo problema ha la stessa natura del problema precedentemente discusso, ovvero la mancanza di informazioni sulla verticalità degli edifici. Se la nuvola di punti LiDAR avesse avuto anche le superfici verticali questo problema sarebbe stato in parte mitigato in quanto ogni punto della nuvola, come precedentemente spiegato, oltre alle informazioni di locazione (x, y, z) ha anche informazioni sul colore del punto (R, G, B). Quindi sarebbe relativamente immediato ricavare una texture a partire da queste informazioni.

Inoltre questo problema è ulteriormente accentuato anche andando ad analizzare gli altri tipi di dato forniti. Le ortofoto, come dice anche il nome sono foto ortogonali al terreno, quindi non sono presenti le facciate degli edifici. Invece per quanto riguarda le altre foto aeree fornite, ovvero le fotografie Oblique; queste in alcuni casi presentano informazioni sulle facciate degli edifici ma non sono costanti e richiederebbero un copioso processing.

Strategie di texturing in assenza di dati specifici sulle facce

L'assenza di informazioni dettagliate sulle facciate degli edifici richiede un approccio creativo al texturing.

Le soluzioni possibili potrebbero essere:

- utilizzo di texture generiche che rappresentino materiali da costruzione comuni (mattoni, intonaco, vetro, ecc), applicate in modo procedurale utilizzando degli algoritmi in modo da adattare la texture al tipi di facciata, anche in base allo stile architettonico desiderato. Questa soluzione sarebbe ottimale in qualsiasi altro contesto, ma all'interno di un Digtial Twin non è la soluzione migliore in quanto si perderebbe la corrispondenza fedele con la città fisica.

- utilizzare database di texture presenti in rete come ad esempio Google Maps, Google Earth o OpenStreetMap. Questa soluzione

potrebbe essere l'ideale qualora fossero presenti i dati in quanto permetterebbe di applicare le texture corrette agli edifici.

Texturing del modello del terreno

D'altro canto, per il texturing del DTM le fotografie aree rappresentano una risorsa preziosa. Elaborando queste immagini per adattarle alla mesh del terreno, è possibile ottenere un livello di dettaglio e realismo elevato. Il processo include l'allineamento delle immagini con le caratteristiche topografiche dei modelli 3D e l'adattamento delle texture per riflettere accuratamente strade, vegetazione, corsi d'acqua e altre caratteristiche naturali ed artificiali del paesaggio urbano.

Dato l'ampio territorio da coprire e la potenziale complessità delle texture del terreno, l'ottimizzazione diventa cruciale. Tecniche come il mipmapping e il tiling delle texture possono aiutare a mantenere elevate le prestazioni del sistema di visualizzazione garantendo allo stesso tempo un elevato grado di dettaglio visivo dove necessario.

2.2.4 Motore grafico di visualizzazione

L'implementazione efficace di un Digital Twin della città di Bologna, arricchito da modelli 3D dettagliati e informazioni visive avanzate, richiede la scelta di un motore o framework di visualizzazione adeguato. Questa decisone è cruciale non solo per garantire la fedeltà grafica e l'accuratezza del modello virtuale, ma anche per supportare un'interazione fluida e immersiva da parte degli utenti. Con una varietà di opzioni disponibili, ciascuna con i propri punti di forza e limitazioni, diventa essenziale valutare attentamente quale piattaforma meglio si allinea agli obietti del progetto, ai requisiti tecnici e alle aspettative dell'esperienza utente.

Andando ad analizzare gli strumenti presenti, spiccano senza dubbio i seguenti motori grafici:

- **Unreal Engine 5** [17]:
  - **Pro**: elevata qualità grafica, supporto avanzato per l'illuminazione dinamica e dettagli architettonici, ampie possi-

bilità di interazione e immersività, plugin già presenti per simulazioni, robusto supporto per VR/AR,

- **Contro**: curva di apprendimento più ripida rispetto ad altri concorrenti, requisiti di sistema elevati per sfruttare appieno le funzionalità avanzate, mancato export dell'ambiente 3D per il web.

- **Unity** [35]:
  - **Pro**: ampio supporto alla community, flessibilità e compatibilità con una vasta gamma di piattaforme (da dispositivi mobili a desktop a web), forte interazione con strumenti di AR e VR, relativamente accessibile ai principianti
  - **Contro**: per alcuni aspetti specifici di rendering e dettaglio grafico è inferiore rispetto ad Unreal Engine 5

- **Godot** [24]:
  - **Pro**: open source e gratuito, leggero e versatile, adatto per progetti di varie dimensioni, supporto sia per la grafica 3D che 2D
  - **Contro**: la community più piccola si traduce in meno risorse disponibili e supporto, molte limitazioni in termini di funzionalità avanzate di rendering rispetto sia ad Unity che Unreal Engine 5

- **WebGL e Three.js** [52][51]:
  - **Pro**: compatibilità web nativa, facilità di accesso senza necessità di installare software aggiuntivi, Three.js offre API di alto livello per semplificare lo sviluppo
  - **Contro**: prestazioni legate alle capacità del browser e del dispositivo, molte limitazioni nel dettaglio grafico e nelle funzionalità avanzate rispetto ai motori desktop.

La scelta del motore di visualizzazione ideale implica la considerazione di diversi fattori, inclusi la qualità grafica desiderata, la complessità del modello, le funzionalità interattive, e la facilità di accesso

per gli utenti finali. Inoltre, aspetti come il supporto alla community, la documentazione disponibile, e la sostenibilità del progetto nel lungo termine giocano un ruolo fondamentale nella determinazione della piattaforma più consona alla realizzazione del Digital Twin.

2.2.5 Integrazione di dati real time

Una volta che la città sarà completamente modellata l'attenzione si sposterà sull'interattività del DT e all'integrazione di dati real time. Questo è un aspetto fondamentale in quanto senza dati in tempo reale che modificano l'ambiente 3D dinamicamente a quello che succede all'esterno non può essere considerato un vero e proprio DT.

Risulta fondamentale in primo luogo analizzare ed esaminare le diferse fonti di dati che possono alimentare il DT, come ad esempio sensori IoT dislocati in tutta la città, feed di dati da sistemi di gestione del traffico, dati ambientali e così via. La selezione delle fonti di dato dipenderà dagli obiettivi specifici del DT e dalle aree di interesse.

Una volta identificate le fonti dati, emerge la sfida tecnologica della loro integrazione nel modello 3D. Tecnologie come RestAPI per lo scambio di dati diventano strumenti essenziali in questo progetto.

Per quanto riguarda la visualizzazione di questi dati all'interno del DT dipende dal dominio dei dati stessi. Se si sta parlando ad esempio di dati parametrici come l'intesità di traffico di una via sono sufficienti indicatori cromatici; invece se si vuole integrare nuovi elementi all'interno del DT, ad esempio l'impianto di un nuovo albero un indicatore cromatico non è sufficiente e quindi bisognerebbe istanziare un nuovo modello 3D all'interno del DT.

L'architettura del sistema di visualizzazione richiede una progettazione oculata per garantire flessibilità e scalabilità, soprattutto nell'integrazione e visualizzazione di dati dinamici. Un approccio modulare e l'adozione di interfacce comuni per le diverse sorgenti e tipologie di dati diventano così elementi chiave per il successo del sistema. Questa strategia permette non solo di facilitare l'integrazione di nuovi flussi di dati ma anche di assicurare che il digital twin possa evolvere e adattarsi a

nuove esigenze senza richiedere rivisitazioni radicali dell'infrastruttura esistente.

2.3 Workflow di elaborazione dati

Dopo aver fatto un'analisi dei problemi, in questa sezione verrà presentato brevemente il *workflow* seguito nello sviluppo di questo progetto di tesi.

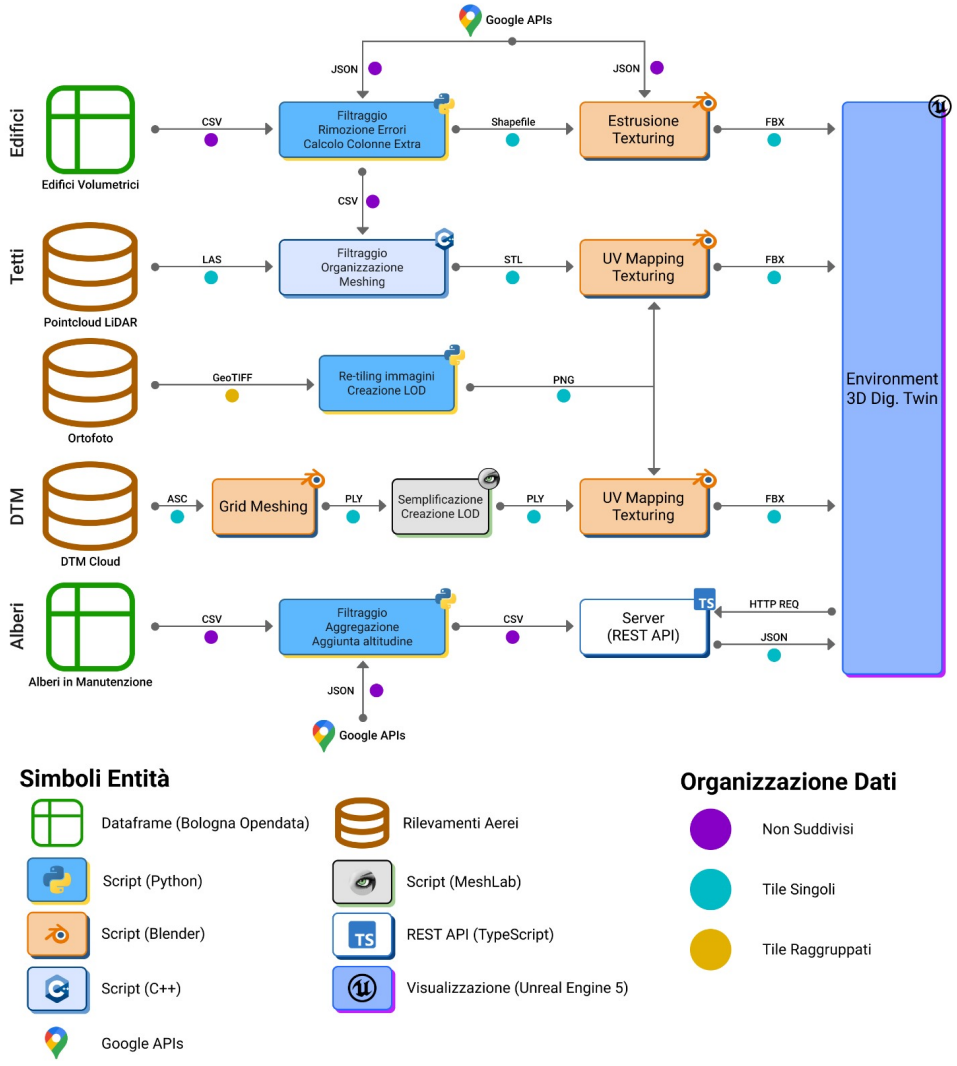


Figura 2.10: Diagramma del worflow completo

Come mostrato in Fig. 2.10, per ogni tipo di dato è presente una pipeline differente, completamente isolata dalle altre. Ogni pipeline parte dai dati grezzi, i quali vengono processati e terminano in un formato dati comune ovvero il *FBX* ed entrano nell'ambiente 3D del DT.

Facendo una veloce panoramica delle varie pipeline abbiamo:

- **Edifici**: si parte dai dati catastali forniti dal Comune di Bologna, vengono costruite le mesh e viene effettuato il texturing tramite Blender [29].
- **Nuvola di punti LiDAR**: verrà utilizzata per estrarre informazioni sui tetti e costruire il modello 3D, successivamente verrà fatto il texturing in Blender.
- **Ortofoto**: tramite codice scritto in Python [49] verranno separate in modo da rispettare lo standard dei tile 500x500 e verranno utilizzate per fare texturing dei tetti e delle mesh del terreno.
- **Modello del terreno**: attraverso Meshlab [12] vengono semplificate le mesh, e in Blender verranno costruiti i Level of Details, quindi infine applicate le textures.
- **Alberi**: dai dati del comune verrà costruito un servizio cloud che fornisce informazioni real time sugli alberi al DT.

Questo grafico fornisce un'ottima panoramica di tutti i task che sono stati svolti al fine di costruire una proof of concept del Digital Twin della città di Bologna.

2.4 Tool e Software utilizzati

Di seguito verrà fatto un elenco e una breve descrizione degli strumenti e software utilizzati:

- **Blender**: Software open source per la creazione di grafica 3D, supporta l'intera pipeline di produzione 3D tra cui: modellazione, animazione, simulazione, rendering, compositing e motion tracking (Fig. 2.11).

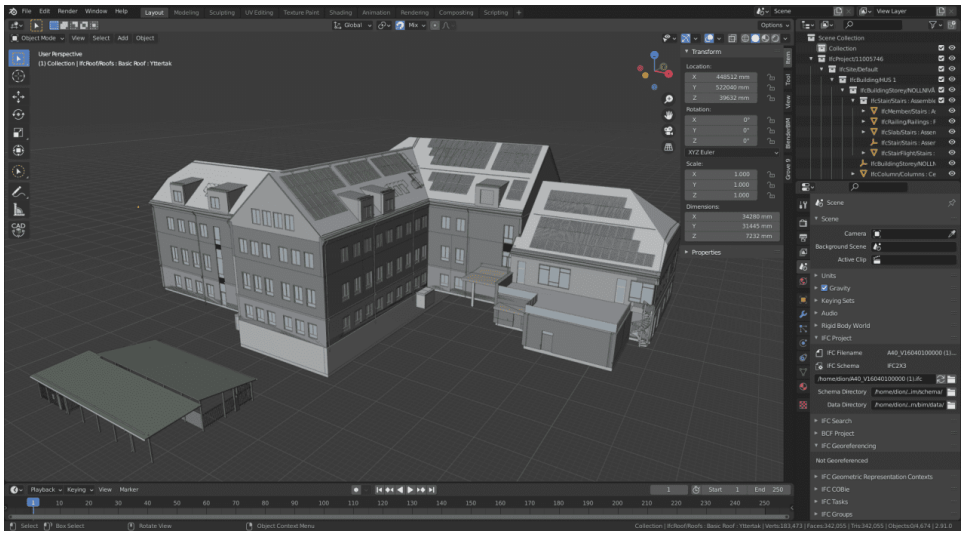


Figura 2.11: Blender

- **Meshlab**: Software open source per l'elaborazione e l'editing di mesh 3D, fondamentale per compiti quali la pulizia delle mesh, la semplificazione e la ricostruzione (Fig. 2.12).

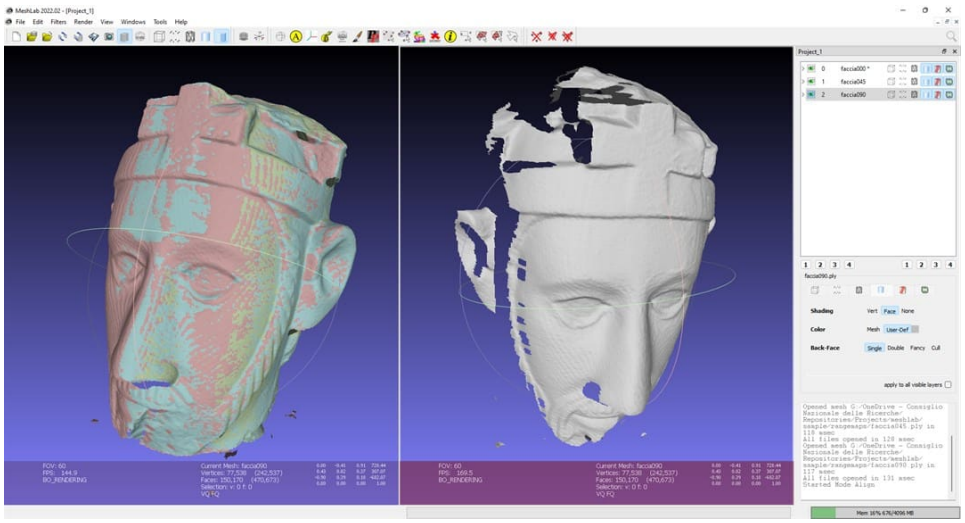


Figura 2.12: Meshlab

- **Unreal Engine**: Motore di gioco avanzato, utilizzato per lo sviluppo di videogiochi, simulazioni, effetti speciali e visualizzazioni 3D in tempo reale (Fig. 2.13).

La scelta di questo Engine grafico è stata guidata da diverse considerazioni chiave. Innanzitutto, Unreal Engine è noto per la sua potente piattaforma di rendering in tempo reale che offre immagini di alta qualità e dettagliate, essenziali per rappresentare accuratamente la complessità visiva di un ambiente urbano. Inoltre, la sua architettura flessibile e la capacità di supportare vasti scenari open-world lo rendono particolarmente adatto per modellare grandi estensioni urbane con un elevato grado di dettaglio.

Inoltre, Unreal Engine fornisce un ampio set di strumenti tra cui il linguaggio di scripting visuale *Blueprint* e la programmazione C++, che permettono di creare logiche complesse e interazioni dinamiche all'interno del Digital Twin.

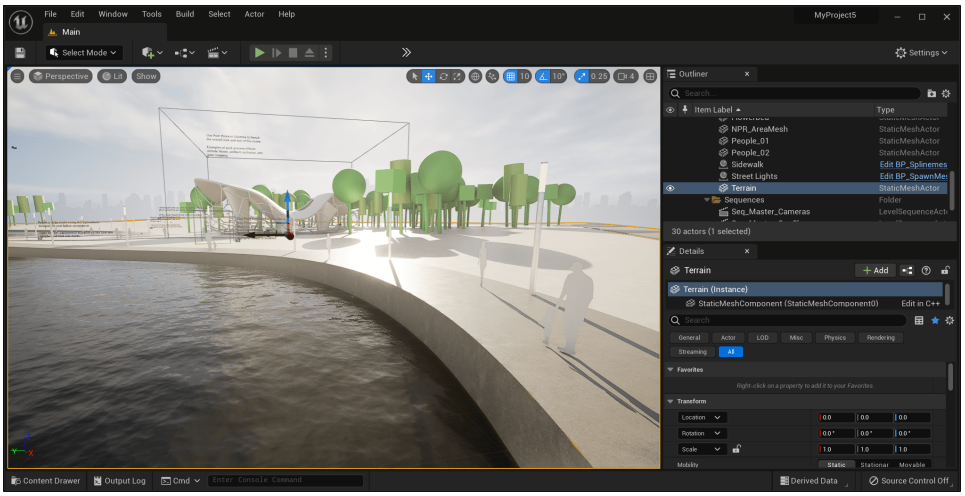


Figura 2.13: Unreal Engine 5

Capitolo 3

Elaborazione di dati grezzi per la creazione di modelli 3D

Contents

3.1 Elaborazione dati per ottenere mesh degli edifici 38

3.1.1 Processing csv carta tecnica comunale 39

3.1.2 Preparazione shapefile da csv 43

3.1.2.1 Organizzazione degli edifici in tiles 43

3.1.2.2 Creazione shapefiles edifici 45

3.1.3 Texturing buildings 46

3.1.3.1 Sviluppo del Texturer 49

3.1.3.2 Risultati texturing e fixing script 54

3.1.4 Export mesh degli edifici 59

3.2 Digital Terrain Model: DTM 59

3.3 Processing di dati sul Verde Urbano 61

3.3.1 Preparazione e organizzazione dati 62

3.3.2 Sviluppo ed implementazione della RestAPI 65

3.4 Elaborazione dati LiDAR per ottenere i tetti . 71

3.1 Elaborazione dati per ottenere mesh degli edifici

In questa sezione verranno analizzati tutti i metodi e le procedure adottate per la generazione delle mesh 3D degli edifici della città di Bologna.

Come anticipato precedentemente le nuvole punti LiDAR, sebbene siano uno strumento prezioso per la mappatura di precisione delle caratteristiche geografiche e per la rilevazione di elementi naturali ed artificiali in 3D, presentano alcune limitazioni significative quando utilizzate per la ricostruzione dettagliata degli edifici. La principale di queste limitazioni è l'assenza di dati sulle superfici verticali degli edifici. Poiché le rilevazioni LiDAR vengono effettuate prevalentemente dall'alto, i dati raccolti forniscono una rappresentazione accurata delle superfici orizzontali, come i tetti, ma lasciano uno spazio vuoto dove ci si aspetterebbe di trovare le facciate.

Quindi data la difficoltà nel ricostruire le mesh partendo dalla nuvola di punti LiDAR è stata adottata una strategia differente. Per la costruzione delle sole mesh degli edifici ci appoggiamo al file csv della *carta tecnica comunale*, mentre la nuvola di punti LiDAR sarà utilizzata solo per ottenere le mesh dei tetti degli edifici.

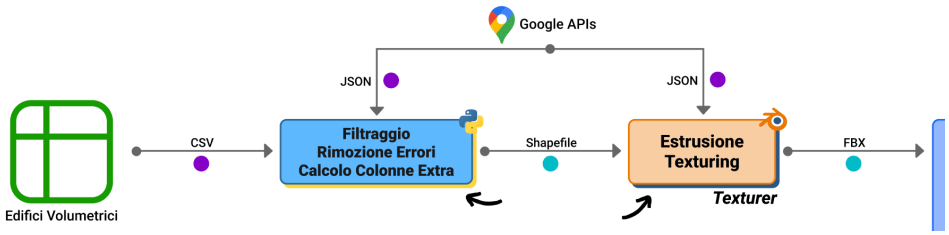


Figura 3.1: Sezione wowflow: edifici

Come è possibile vedere in Fig. 3.1, la pipeline di questa sezione è organizzata in 2 step principali: in primo luogo la tutto ciò che riguarda la pulizia, la preparazione dati e l'ottenimento di shapefile degli edifici, in secondo luogo il texturing delle mesh degli edifici con le texture

ottenute da Google Maps e lo sviluppo di un'estensione ad hoc per Blender.

3.1.1 Processing csv carta tecnica comunale

Come formalizzato nel capitolo di analisi dei dati forniti, il csv della CTC è ottimale per poter costruire le mesh degli edifici in quanto, per ogni edificio (quindi per ogni riga del csv) fornisce preziose informazioni riguardanti:

- **Geo Point**: il punto del baricentro della pianta dell'edificio;
- **Geo Shape**: ovvero una lista di coordinate che costituiscono il perimetro dell'edificio;
- **Quota piede**: la quota dell'edificio sul livello del mare;
- **Quota gronda**: la quota della gronda dell'edificio sul livello del mare, per quota di gronda s'intende la quota dell'edificio prima del tetto.

Quindi l'obiettivo è quello di costruire uno shapefile degli edifici partendo dalla CTC. Uno shapefile è un particolare tipo di formato GIS che permmette di rappresentare entità geometriche come ad esempio punti, linee, polilinee ed associare ad esse informazioni. Nel nostro caso appunto la quota di piede e la quota di gronda. Questo tipo di file è comodo in quanto è interoperabile con Blender tramite l'estensione *Blender-GIS* permettendo di importarlo direttamente come mesh. Tutto ciò evita di doversi costruire a mano la mesh tramite triangolazione.

Andando ad analizzare nel dettaglio il csv si nota subito che c'è bisogno di fare pulizia sui dati. Per compiere questi task è stato scritto un **Jupyter Notebook** usando il linguaggio di programmazione **Python** sfruttando le seguenti librerie:

- **Pandas**: per lavorare sui tabulati
- **Matplotlib**: per visualizzare grafici

- **Json**: per poter trattare alcune stringhe come oggetti Json

La prima cosa che si può vedere facendo delle print sul dataset riguarda proprio il sistema di coordinate utilizzato per localizzare gli edifici. Il sistema di coordinate utilizzato in questo dataset è il **WGS84** rappresentato in gradi decimali, quindi ad esempio un geo point sample preso dal dataframe è rappresentato dalla doppia (44.53593, 11.361562). Invece tutti gli altri file messi a disposizione utilizzano un altro sistema di coordinate, sempre in WGS84 ma in formato UTM. Quindi la doppia (44.53593, 11.361562) in UTM è rappresentata dalla tripla (32, 687624.47, 4934112.89).

Perciò il primo task effettuato sul dataframe è proprio questa conversione di coordinate. A tale scopo è stata utilizzata la libreria Python UTM.

Dato che ogni entry, quindi ogni edificio ha due colonne con informazioni di coordinate (Geo point e Geo shape) è importante che questa conversione venga fatta per entrambi i valori. Per i Geo point è immediato in quanto è un singolo punto sotto forma di stringa con i due valori separati da virgola. Più complesso è la conversione del Geo shape in quanto è in formato Json. Per questo motivo è stata utilizzata la libreria Python omonima che permette di leggere una stringa in formato json come se fosse un oggetto.

Per ogni riga del tabulato vengono convertite le coordinate GeoPoint e memorizzate in una nuova colonna e per le coordinate del GeoShape si cicla su tutte le coordinate presenti nella shape, vengono convertite e memorizzate come array di triple in una nuova colonna del dataframe.

Successivamente effettuando una serie di visualizzazioni di valori del dataset si presentano diversi problemi sulla natura dei dati, ad esempio che molti edifici presentano una quota sul livello del mare di 0, cosa impossibile nella città di Bologna in quanto mediamente la città è collocata circa 50 metri s.l.m.

Facendo ispezioni molto più approfondite sulle informazioni di questi 11490 edifici come mostrato in Fig. 3.2, si notano i primi pattern, ovvero che la maggior parte di queste entry appartengono a categorie

```
# Edifici con quota piede 0 (sul livello del mare)
print(df[df['QUOTA_PIEDE'] == 0].shape[0])
```

✓ 0.0s

```
11490
```

Figura 3.2: Numero di edifici con quota s.l.m. 0

di edifici che effettivamente non sono significative, nel dettaglio queste categorie sono:

- Cabina Enel;
- Stazione di Rifornimento
- Chiosco gelati
- Chiosco fiori
- Chiosco giornali
- In costruzione

Qui c'è da fare un piccolo appunto; tra le descrizioni degli edifici appena citate se si pensa in ottica di un Digital Twin potrebbero essere edifici molto importanti soprattutto se si pensa di mappare tutte le Cabine Enel per un questione di simulazioni energetiche, allo stesso modo anche le stazioni di rifornimento. Per come sono stati inseriti in questo tabulato, non hanno informazioni in alcun modo interessanti se si ha come scopo la sola costruzione dei buildings in quanto tutte queste entry, oltre che a presentare quota piede 0 hanno anche il valore dell'area della superficie pari a 0. Tutto ciò dimostra come questi file non siano esenti da errori o problemi che richiedono una attenta analisi e processing.

Ai fini di sola visualizzazione, e dato che l'energia non è presente tra i main goal di questo progetto di tesi ma solamente un obiettivo futuro del Digital Twin, tutti questi edifici che appartengono alle categorie sopracitate verrano scartati dal dataset. In quanto quello che interessa da queste fasi di processing è ottenere le superfici degli edifici per la

ricostruzione 3D quindi -ovviamente- edifici con area pari a 0 non hanno alcuna importanza nella ricostruzione del modello.

```
useless_desciriptions = ['Cabina ENEL', 'Stazione di rifornimento', 'Chiosco gelati',
                         'Chiosco fiori', 'Chiosco giornali', 'In costruzione']

# removing useless descriptions
df = df[~df['DESCRIZIONE'].isin(useless_desciriptions)]

# removing entries with height == 0.0
df = df[df['ALTEZZA_GRONDA'] != 0.0]

print(df[df['QUOTA_PIEDE'] == 0].shape[0])
```

✓ 0.0s

```
403
```

Figura 3.3: Rimozione di edifici con descrizioni non utili

Una volta rimossi tutti gli edifici con area 0, cioè tutti gli edifici con quelle descrizioni, si nota che rimangono circa 400 edifici che hanno ancora quota sul livello del mare pari a 0 (Fig. 3.3). Ciò è un problema in quanto tutte queste entry sono valide, cioè hanno area maggiore a 0 quindi con una GeoShape valida. Questo molto probabilmente rappresenta un errore umano nella rilevazione, quindi ai fini di ottenere le mesh è fondamentale correggere le quote di tali edifici.

In assenza di informazioni sulle quote degli edifici l'unica tecnica attuabile per ottenere tali quote è attraverso le API di Google Maps. Tramite queste API è possibile avere accesso ad un vasto database geospaziale, permettendo di recuperare informazioni precise sull'altitudine in qualsiasi punto del globo terrestre. Attraverso l'invio di una richiesta HTTP che include le coordinate geografiche dell'edificio in esame, consegnando il valore di altitudine corretto per quelle coordinate.

Un'altra tecnica potrebbe essere sfruttare i DTM forniti ma ciò richiederebbe di implementare una logica di matching tra i punti del perimetro dell'edificio sulla superficie del DTM.

A tale scopo è stata scritta una funzione che prende in ingresso un array di coordinate e restituisce un oggetto json contenente la quota di ogni coordinata presente nell'array di input.

Questa funzione viene chiamata una sola volta per tutti i 409 edifici

in quanto le API di Google permettono di specificare più coordinate in ingresso. Questo garantisce un guadagno in tempi di esecuzione in quanto è molto più efficiente effettuare una sola chiamata HTTP per tutti gli edifici che una chiamata HTTP per ogni edificio.

Una volta ottenute queste informazioni è bastato sovrascriverle nel dataframe in corrispondenza agli edifici con quota 0.

3.1.2 Preparazione shapefile da csv

Una volta che i dati sono stati puliti da tutti gli errori, malformattazioni, ed impurità è possibile proseguire e costruire lo shapefile.

Prima di spiegare i passi operativi è opportuno spiegare nel dettaglio cosa sia uno shapefile e perché può essere molto utile nel nostro caso.

Lo shapefile è un formato di file vettoriale per software di sistema informativo geografico (GIS) sviluoppato da ESRI. Consiste in una collezione di files che descrivono forme geometriche come punti, linee e poligoni. Ogni shapefile è accompagnato da file aggiuntivi che possono contenere attributi dati per ogni forma, come altezza, materiale, o uso dell'edificio, rendendolo estremamente versatile e informativo per analisi spaziali e modellazione 3D. La comodità dello shapefile è proprio questa perché è possibile combinare alle informazioni geometriche precise informazioni di ogni tipo, nel nostro caso l'altezza dell'edificio.

Per quanto riguarda la modellazione 3D, l'integrazione di dati GIS come shapefile è possibile attraverso l'estensione Blender-GIS che permette di importare direttamente nella scena 3D shapefile correttamente geolocalizzati. Il vero punto forte di questa estensione sta nel fatto che, durante l'import di uno shapefile chiede se se superfici rappresentate dai poligoni scritti nel file debbano essere estruse. E questo è proprio ciò che fa al nostro caso perchè possiamo estrudere automaticamente l'edificio, lungo l'asse z, di un quantitativo pari alla quota di gronda del singolo edificio.

3.1.2.1 Organizzazione degli edifici in tiles

Prima di mostrare la logica tramite il quale sono stati ottenuti gli shapefile a partire dal csv degli edifici è fondamentale dire con che

organizzazione desideriamo ottenere i dati. Tutti i dati LiDAR, quindi nuvole di punti e DTM sono organizzati in tile che coprono un'area di 500x500 metri ciascuno, è nostro obiettivo organizzare secondo la stessa logica anche gli shapefile degli edifici, in modo tale che una volta che le mesh saranno completamente pronte saranno automaticamente allineate.

Dato che questa informazione non è presente nel nostro dataframe contentente gli edifici è fondamentale aggiungere una nuova colonna che associ un edificio ad un tile (o più tile nel caso l'edificio si trovi al confine tra due tiles).

A tale scopo prima è stato costruito, molto velocemente, un nuovo dataframe contenente 3 colonne, la prima contiene il nome del file secondo lo standard dei file LiDAR forniti dall'azienda, ovvero le tre coordinate WGS84 UTM separate da '_' indicanti l'angolo in basso a sinistra del tile. E le altre due colonne contengono il valore di latitudine e di longitudine dell'angolo.

É possibile vedere un estratto di questo dataframe in Fig. 3.4.

	A	B	C
1	FileName	Lat	Lon
2	32_677000_4930500.las	677000	4930500
3	32_677000_4932500.las	677000	4932500
4	32_677000_4933000.las	677000	4933000
5	32_677000_4933500.las	677000	4933500
6	32_677000_4934000.las	677000	4934000
7	32_677500_4930000.las	677500	4930000
8	32_677500_4930500.las	677500	4930500

Figura 3.4: Dataframe di tutti i LLC dei tiles

Successivamente, nel Jupyter Notebook utilizzato per fare pulizia dati sul dataframe è stata aggiunta anche la logica nel quale sia associa un edificio ad uno o più tiles.

L'algoritmo applicato ad ogni edificio segue la seguente logica:

- crea un oggetto *Poligono* dato dalla lista dei punti del GeoShape dell'edificio, usando la libreria **Polygon**

- si cicla su tutti i tiles del csv contenente la lista di tutti i tiles di Bologna
- per ogni tile si costruiscono 4 linee che rappresentano i 4 lati del quadrato del tile, e si controlla se il poligono interseca una di queste 4 linee, se si si aggiunge il tile in una lista;
- successivamente, per ogni coordinata dello shape, per ogni tile presente nel csv dei tiles, si controlla se quella coordinata appartiene al tile, se si si aggiunge alla lista.

Risulta molto importante prima ciclare controllando se il poligono interseca le linee di confine del tile perchè velocizza il processo di scoperta del tile successivamente, se non è stato trovato alcun tile per tutte le coordinate di ogni shape si controllano tutti i punti. Il codice è visualizzabile nell'appendice A.

Al termine di tale processing quindi si ottiene una nuova colonna nel dataframe contenente una lista di tiles a cui gli edifici appartengono.

3.1.2.2 Creazione shapefiles edifici

Per creare gli shapefiles è stata utilizzata la libreria Python **GeoPandas**, libreria perfettamente interoperabile con Pandas che da la possibilità di trattare dati georeferenziati e permette la lettura e scrittura di file come *GeoJson*, *GeoPackage* e *Shapefile*.

Attraverso GeoPandas è molto semplice costruire uno shapefile, tutto ciò che serve è costruire un GeoPandas dataframe assicurandosi di creare una colonna *geometry* nel quale si inseriscono tutti i punti del GeoShape dell'edificio. Invece per aggiungere informazioni personalizzate, come nel nostro caso la quota piede e la quota gronda basta aggiungere una colonna nuova per tipo di informazione desiderata. Infine per renderlo geolocalizzato correttamente bisogna assegnare il giusto codice al campo *crs (Coordinate Reference System)* dell'oggetto Geopandas dataframe, nel nostro caso il codice crs corretto è il *EPSG:32632*.

L'algoritmo è immediato, si cicla su tutti i tile del csv dei tiles, si prendono dal csv degli edifici tutti gli edifici che appartengono a quel tile, si crea il dataframe geopandas a cui gli si associa la geometria

di tutti gli edifici e le informazioni di quota piede e quota gronda, si imposta il crs corretto infine si esporta lo shapefile. Una prima visualizzazione del risultato della creazione di uno shapefile è consultabile in Fig. 3.5. Infine, importando lo shapefile in Blender si ottiene la mesh 3D degli edifici di un tile, come mostrato in Fig. 3.6. Il codice invece è visualizzabile nell'appendice A.

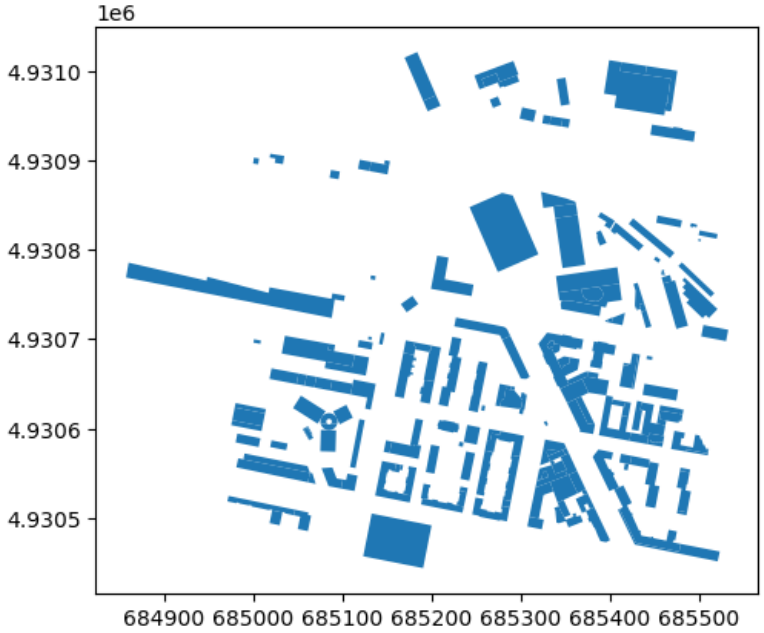


Figura 3.5: Stampa dello shape di un tile preso come esempio

Ora che gli shapefile sono stati scritti è possibile visualizzare la mesh 3D all'intero di Blender attraverso la sua estensione Blender-GIS. Prendendo uno shapefile come esempio tra i tiles del centro storico di Bologna ed importandolo in Blender si ottiene ciò:

3.1.3 Texturing buildings

Una volta ottenute le mesh degli edifici, per soddisfare il requisito dell'immersività è fondamentale riuscire a costruire delle texure da porre sulle facce delle mesh degli edifici. Come anticipato anche in precedenza, la creazione di texture dettagliate e fedeli alla realtà si scontra con la mancanza di informazioni sulle superfici verticali degli edifici, sia per quanto riguarda i dati LiDAR che i dati fotografici. Questa

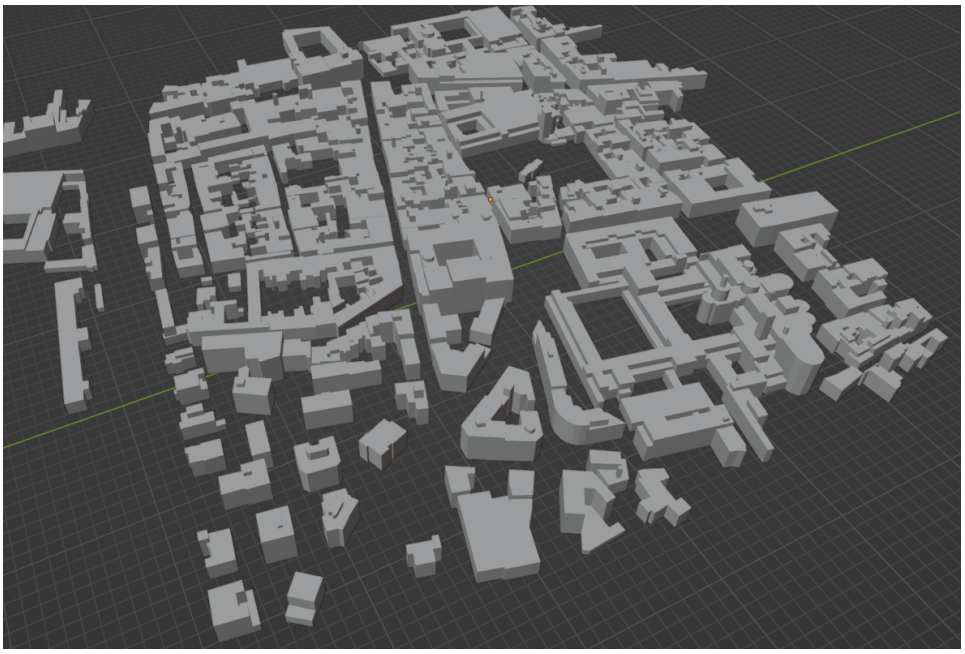


Figura 3.6: Mesh di un tile su Blender

limitazione implica la necessità di adottare strategie alternative per la gestione delle texture.

Dopo un'accurata valutazione delle opzioni disponibili, elencate nell'analisi del problema alla sezione 2.2.3, è stata presa la decisione di sfruttare le potenzialità offerte dalle **Google API** per acquisire le texture delle facciate degli edifici. La scelta di questo approccio su basa soprattutto sul fatto che in rete Google Maps è l'unica banca di dati che mette a disposizine texture di qualsiasi tipo di edificio ad altissima risoluzione.

I modelli 3D che Google mette a disposizione per il download sono i modelli visualizzabili su Google Earth (Fig. 3.7), quindi per quanto riguarda le texture degli edifici rappresentano lo stato dell'arte.

La prima domanda che ci poniamo è la seguente:

Visto che le API di Google permettono di ottenere mesh e texture ad altissima risoluzione, perché non utilizzare direttamente quelle mesh all'interno del Digital Twin?

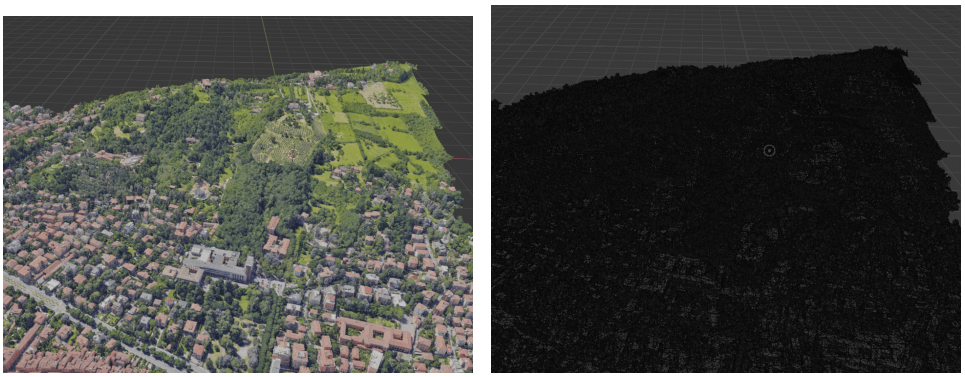
Figura 3.7: Piazza Maggiore su Google Earth

La risposta in questo caso è più semplice del previsto e si basa principalmente su due punti fondamentali:

- Le mesh di Google in quanto sono ad altissima risoluzione presentano una densità poligonale altissima, mentre le nostre mesh costruite dagli shapefile hanno il minimo numero di triangoli possibili e quindi sono molto più leggere. All'interno di Google Earth le mesh hanno una sistema di Level of Detail (LOD) ma una volta che si effettua la HTTP request viene consegnata la mesh ad un'unica alta risoluzione, quindi anche se si volesse fare dei LODs bisognerebbe richiedere la stessa mesh a risoluzioni differenti.

- Le mesh costruite dai dati catastali del Comune di Bologna sono perfette in ottica di un Digital Twin perché innanzitutto rappresentano le posizione corrette degli edifici (con Google non si ha questa certezza) e inoltre, dato che sono state costruite tramite un csv contenente altre informazioni quali area, perimetro, ecc, queste informazioni possono essere utilizzate all'interno del Digital Twin per aggiungere funzionalità di visualizzazione.

Analizzando la documentazione delle GoogleAPI si nota che effet-



(a) Mesh con texture (b) Mesh in wireframe

Figura 3.8: Mesh Google in Blender

tuando una richiesta HTTP all'url non viene scaricata alcuna mesh, ma bensì queste API devono essere utilizzate all'interno di un motore di rendering, nella documentazione viene fatto l'esempio utilizzando il motore di rendering web di Cesium per la visualizzazione delle mappe. Non è proprio quello che si desidererebbe in ottica di ottenere solo le texture dal modello 3D.

Per ottenere la mesh vera e propria è possibile utilizzare una estensione Blender chiamata **BLOSM**, la quale permette di scaricare le mesh da Google specificando un'area di interesse e visualizzarle nella scena Blender. In questo modo si ottiene il modello 3D non solo per la visualizzazione ma anche per l'editing (Fig. 3.8).

Dando una rapida occhiata alla mesh ottenuta da Google si nota come il material e quindi anche la texture non sono direttamente accessibili, quindi c'è bisogno di un processo che permetta di trasferire le texture dalla mesh di Google alla nostra mesh. La soluzione più immediata ma al contempo la più efficace è quella di allineare perfettamente le due mesh e attraverso una estensione scritta ad hoc per Blender automatizzare il processo di copia della texture dalla mesh di Google alla nostra mesh.

3.1.3.1 Sviluppo del Texturer

Per affrontare il trasferimento delle texture dalla mesh di Google alle nostre mesh della città di Bologna, è stata sviluppata un'estensione ad

hoc per Blender, sfruttando le potenzialità offerte dal modulo **Bpy**. Questo modula rappresenta il cuore della programmazione in Blender, fornendo un'interfaccia di scripting potente e flessibile che consente agli sviluppatori di automatizzare le operazioni di modellazione, animazione, rendering e texturing.

Prima di spiegare nel dettaglio l'implementazione dell'estensione Blender sviluppata è fondamentale spiegare la logica di base dietro all'algoritmo per il texturing. L'idea di base sta nel fatto che le nostre mesh e le mesh di Google sono perfettamente sovrapponibili, quindi una volta sovrapposte è possibile identificare tutte le facce verticali delle nostre mesh, e per ogni faccia scattare un render della faccia sovrapposta di Google e applicare l'immagine risultante dal render alla faccia della nostra mesh.

Inoltre, c'è da fare una breve precisazione su un paio di scelte progettuali che sono state prese durante lo sviluppo di questa estensione. Innanzitutto si è deciso di effettuare l'operazione di texturing tile per tile esportando direttamente un tile texturato con un unico materiale ed un'unica texture, e non esportando una mesh per edificio con un materiale per edificio. Questa scelta è dovuta soprattuto per alleggerire il motore di rendering in quanto è molto più efficiente renderizzare un unico oggetto con un materiale rispetto a N oggetti con N materiali. Questo significa che è necessaria costruire una texture *Atlas* ovvero una immagine molto grande contenente una griglia tutte le texture degli edifici del tile. L'altra scelta progettuale sta nel fatto che questa estensione per lavorare ha bisogno che nella scena sia presente la mesh di Google e tutte le mesh degli edifici che vogliamo texturare con una specifica naming syntax. Le mesh degli edifici devono avere il nome del tile a cui appartengono. Questa ultima scelta non è molto importante nel texturing ma più che altro è stata fatta in ottica di operazioni successive di allineamento con le mesh dei DTM aventi lo stesso nome.

Implementazione codice:

La struttura del codice di questa estensione è stata suddivisa in 6 classi in modo da favorire la modularità ed estensibilità, essenziale per mantenere l'organizzazione e la chiarezza del codice, facilitando

al contempo future espansioni. Ogni classe è progettata con un focus su una specifica area di responsabilità all'interno dell'algoritmo. In Fig. 3.9 è possibile vedere il diagramma UML delle classi sviluppate. Andando nel dettaglio:

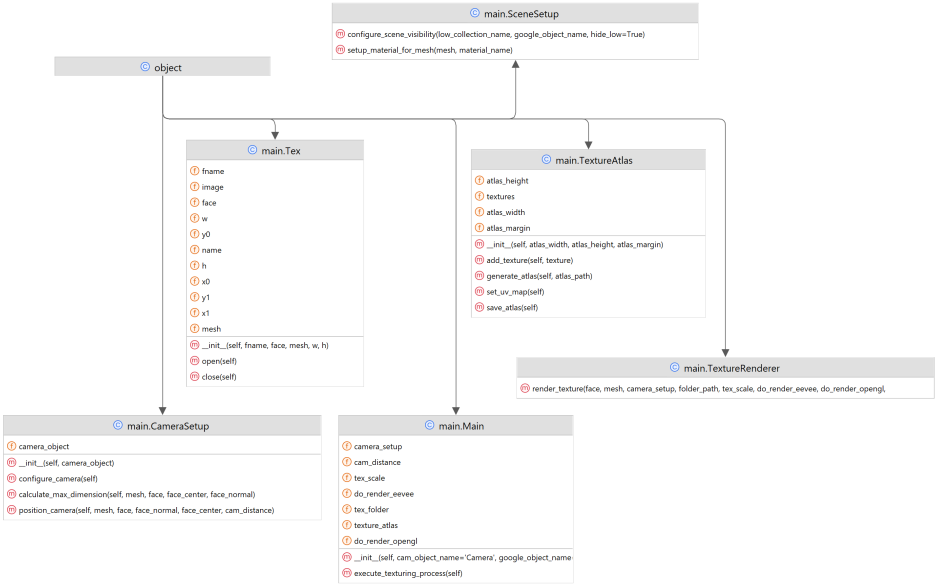


Figura 3.9: Diagramma UML delle classi dell'estensione Blender

- **Main**: classe che si occupa di coordinare l'intero processo di texturing, dalla configurazione iniziale della scena alla generazione dell'atlas finale, fungendo da punti di ingresso per l'esecuzione dell'estensione.

- **SceneSetup**: prepara la scena di Blender configurando la visibilità degli oggetti, assicurando che solo gli elementi rilevanti siano visibili durante il processo di renderizzazione delle textures.

- **CameraSetup**: configura la camera virtuale di Blender per catturare correttamente le facciate degli edifici, impostando parametri come il tipo di camera.

- **Tex**: gestisce le informazioni relative allte texture individuali, incluse le dimensioni, posizione e il file immagine associato, fonda-

mentale per la preparazione e l'organizzazione delle texture prima della loro applicazione sulle mesh.

- **TextureAtlas**: si occupa della creazione dell'atlas di texture, organizzando e combinando tutte le texture individuali in un'unica immagine compatta. Questo processo ottimizza l'uso delle risorse grafiche e facilita la gestione delle texture all'interno di Blender.
- **TextureRenderer**: automatizza il processo di rendering delle texture per ogni faccia delle mesh degli edifici, utilizzando la configurazione della camera per catturare le immagini necessarie.

Andando nel dettaglio dell'algoritmo la logica è molto semplice, di seguito verrà spiegato brevemente step per step:

1. si istanzia la classe Main la quale inizializza un oggetto CameraSetup e TextureAtlas, successivamente viene settata la visibilità della scena tramite l'oggetto CameraSetup.
2. il metodo *execute_texturing_process* di Main inizia il processo di texturing.
3. per ogni edificio, prima si setta il materiale nuovo, successivamente si cicla su tutte le facce e si individuano le facce verticali.
4. per ogni faccia verticale il metodo *TextureRenderer.render_texture()* posiziona la camera ortogonale alla faccia, crea l'immagine di output ed effettua il render.
5. si crea un oggetto Tex con l'immagine renderizzata da TextureRenderer.
6. si aggiunge la texture nella lista di texture contenuta nella classe TextureAtlas.
7. una volta ciclato su tutti gli edifici si costruisce l'atlas tramite il metoto *generate_atlas*.
8. si ordinano le texture in base all'altezza in ordine decrescente e inizia il loop di posizionamento delle texture dell'atlas il quale per ogni riga dell'atlas si calcola quanto spazio occupa una riga e aggiunge texture fino a che c'è spazio

9. per ogni texture nella riga si calcolano le coordinate dell'angolo in alto a sinistra e in basso a destra.

10. si posizionano le immagini vere nella texture in base alle coordinate appena calcolate.

11. infine si calcolano gli uv mapping di tutte le texture in modo da associare ad ogni faccia le coordinate uv in base alla posizione della texture nell'atlas.

Di seguito verranno spiegati più esaustivamente alcuni dei passi più importanti dell'algoritmo.

4. Metodo TextureRenderer.render_texture()

La prima cosa che viene fatta all'interno di questo metodo è settare la posizione della camera in modo che guardi esattamente la faccia da renderizzare, attraverso il metodo *set_camera_position()*.

Il metodo *calculate_max_dimension()* si occupa di calcolare la scala ortografica della camera per assicurasi che l'intera faccia sia visibile dentro l'inquadratura. Il calcolo viene effettutato in base ai vertici della faccia e la normale, si considera la proiezione della faccia lungo la sua normale per determinare le dimensioni effettive.

Successivamente in base alle dimensioni calcolate della camera, viene settata la risoluzione del render e attraverso bpy si scatta il render.

7./10. Metodo TextureAtlas.generate_atlas()

La prima cosa che viene è ordinare le texture in base alla loro altezza in ordine decrescente. Questo aiuta ad ottimizzare il packing delle texture nell'atlas, mettendo prima le texture più alte e potenzialmente riducendo lo spazio vuoto.

Successivamente successivamente si organizzano le immagini all'interno dell'atlas. Ad ogni immagine viene calcolato un margine in modo da essere perfettamente separata dalle altre immagini vicine.

Un ciclo viene eseguito fino a che sono presenti texture da posizionare. All'interno di questo loop si calcola per ogni "riga" dell'atlas quanto spazio occupa la stessa in termini di larghezza (w). Quindi, per ogni texture nella riga corrente vengono calcolate le coordinate *(x0,*

y0) dell'angolo in basso a sinistra e *(x1, y1)* dell'angolo in alto a destra che rappresentano la posizione che deve avere la texture all'interno dell'atlas.

Infine, per dopo aver calcolato le posizioni in cui le texture dovranno essere posizionate vengono effettivamente incollate nella texture atlas e viene salvato il file immagine.

11. Calcolo UVMapping

La funzione *TextureAtlas.set_uv_map()* si occupa di assegnare le coordinate UV di ogni vertice delle facce di una mesh basandosi sulle posizioni delle relative texture all'intero dell'atlas.

3.1.3.2 Risultati texturing e fixing script

Quindi facendo partire l'estensione con la scena preparata, ovvero con mesh di Google e mesh ottenute da shapefile, un esempio di texturing che si ottiene è visualizzabile in Fig. 3.10, e l'atlas risultante in Fig. 3.11.

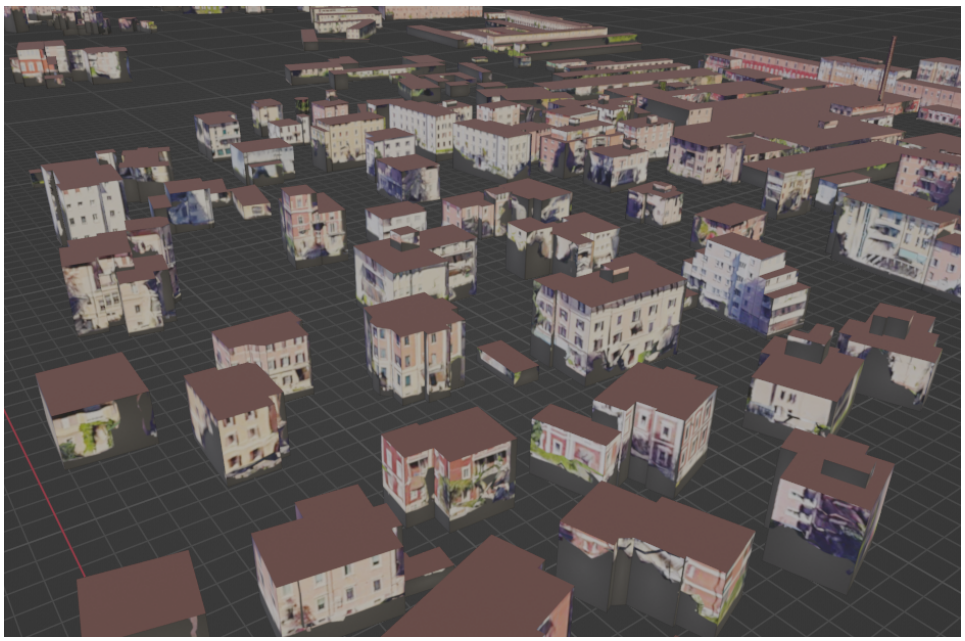


Figura 3.10: Texturing tramite estensione Blender

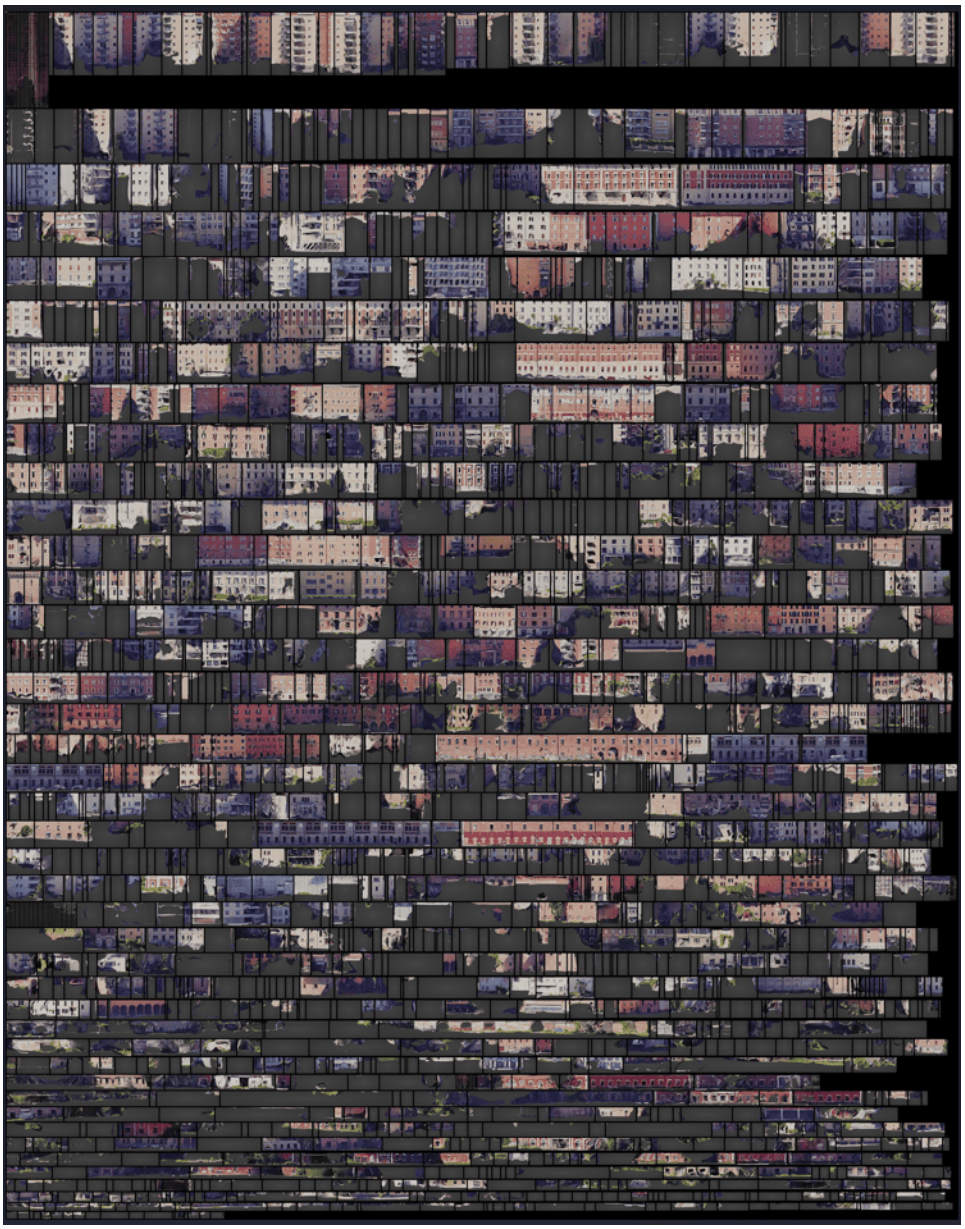
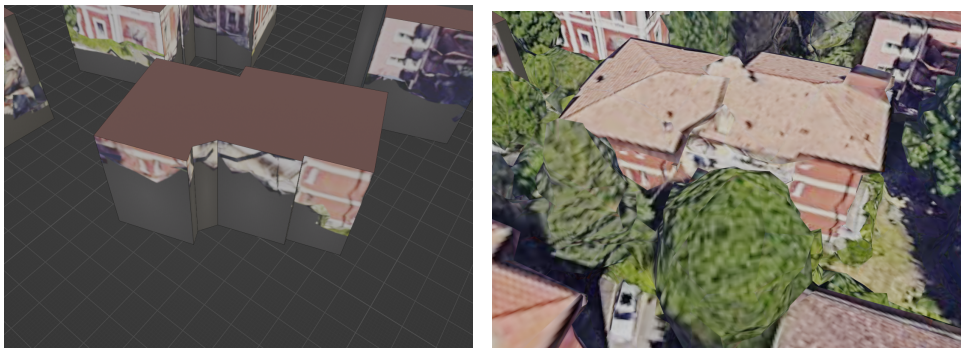


Figura 3.11: Atlas risultante

Come risultato è molto soddisfacente, però non è esente da problemi. Innanzitutto il problema più grosso si può vedere in alcuni edifici in particolare, ad esempio il building in basso a destra in Fig. 3.10 e Fig. 3.12. Si può notare che per metà facciata la texture è molto "sporca". Questo è dovuto al fatto che la mesh di Google non contiene solamente



(a) Mesh con texture parziale (b) Mesh di Google

Figura 3.12: Errore nel texturing dovuto a elementi esterni

gli edifici, ma ache tutti gli altri elementi architettonici e in questo caso alberi. Quindi in corrispondenza di quella faccia era presente un albero, e la camera posizionandosi ortogonalmente alla faccia per metà si trovava all'interno dell'albero e quindi risulta in un render non completo dell'intera faccia.

Data la natura delle mesh di Google e all'impossibilità di applicare dei filtri all'API oppure nascondere determinati elementi questo problema risulta difficilmente risolvibile se non utilizzando tecniche di ricostruzione di facciate tramite Intelligenza Artificiale.

Il pc utilizzato per effettuare il texturing è un pc con una scheda video *Nvidia GeForce 4070* e un processore *Intel Core i7 13th gen*, quindi un hardware molto potente. Nonostante ciò sorge il problema del tempo di esecuzione. Come esempio è stato preso il tile con coordinate *(32, 685000, 4928000)* in quanto è il tile con meno edifici e quindi quello che si presta meglio ad essere sottoposto a testing. La fase di rendering rappresenta un vero e proprio collo di bottiglia in quanto impiega circa 15 secondi per ogni faccia da renderizzare, rendendo impossibile in tempi brevi il texturing in tile molto più complicati. In questo tile d'esempio sono presenti 311 facce da texturizzare e impiega 71 minuti. Ci sono altri tile che contengono circa 10/15mila facce da renderizzare e quindi impiegherebbe giorni interi ad effettuare il texturing.

La soluzione più immediata consiste nel cambiare il motore di rendering all'interno di blender. In questo modo si ha uno speedup di 10 volte impiegando circa 30 secondi a renderizzare tutte le facce del tile

e costruire l'intero atlas.

Questa differenza di tempo tra i due motori di rendering sta nel fatto che Eevee, nonostante sia un motore che sfrutta la rasterizzazione, permette di produrre effetti di illuminazione avanzati come l'illuminazione globale e una gestione accurata delle ombre, e questo in render molto semplici come quelli in questione aggiungono un overhead. D'altra parte il rendering in OpenGL è molto più veloce in quanto non permette di renderizzare questi effetti luminosi, lavora quasi come uno screenshot della scena. E dato che non sono affatto necessari nel texturing, il motore di rendering di OpenGL è quello più adatto al task.

Alcuni esempi di texturing effettuati su tile complessi appartenenti al centro storico di Bologna sono visualizzabili in Fig. 3.13 e Fig. 3.14.

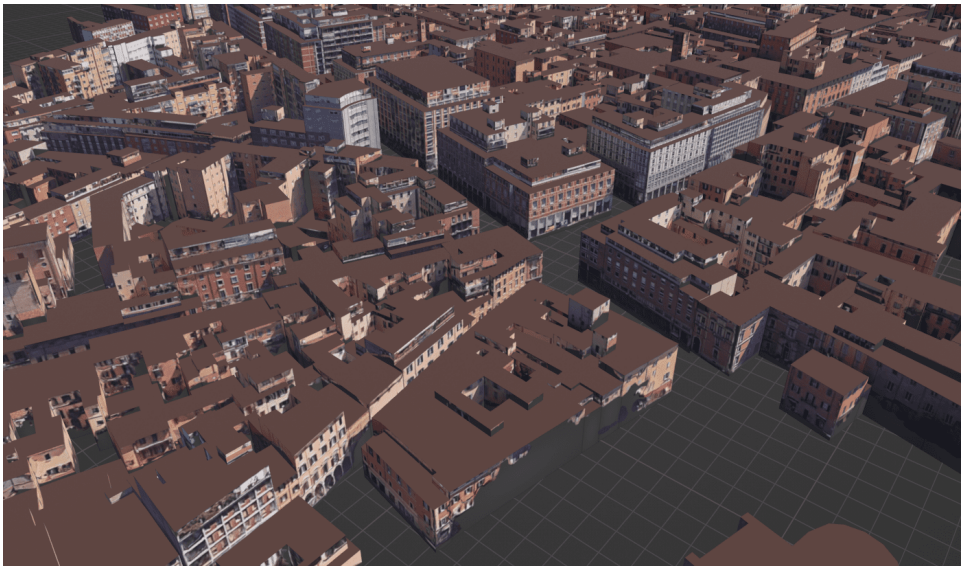


Figura 3.13: Esempio view 1

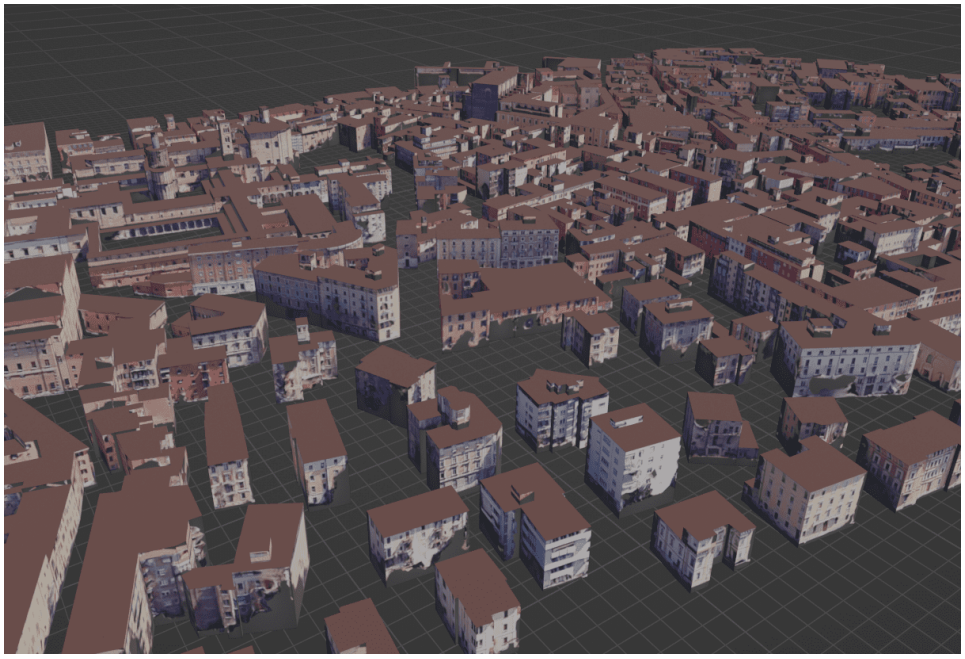


Figura 3.14: Esempio view 2

3.1.4 Export mesh degli edifici

Una volta che le mesh sono state create e texturizzate sono pronte per essere esportate in modo da poterle usare all'interno dell'engine di visualizzazione su cui verrà costruito il Digital Twin. Generalmente questo è un task non molto interessante e parlarne potrebbe essere superfluo. Ma dato che, come presentato anche all'inizio, ogni singolo pezzo di software sviluppato all'interno di questo progetto è stato pensato come un pezzo di un'unica pipeline, quindi anche la parametrizzazione e l'export delle mesh è stato automatizzato in modo da poter processare e di conseguenza esportare in maniera automatica senza bisogno di operare manualmente.

Come definito nel capitolo precedente, si è deciso di utilizzare come formato standard per le mesh il formato **FBX**. A tale scopo è stata scritta una funzione Python che, passato in ingresso un oggetto, lo esporta in fbx.

Una volta effettuato il texturing su tutte le mesh, ciclando su tutte gli oggetti presenti nella scena Blender, questo script permette di ottenere i file FXB pronti per essere utilizzati all'interno del Digital Twin.

3.2 Digital Terrain Model: DTM

In questa sezione verranno mostrati brevemente i risultati conseguiti dal mio collega per ciò che riguarda l'ottenimento, la pulizia e la texturizzazione delle mesh dei DTM.

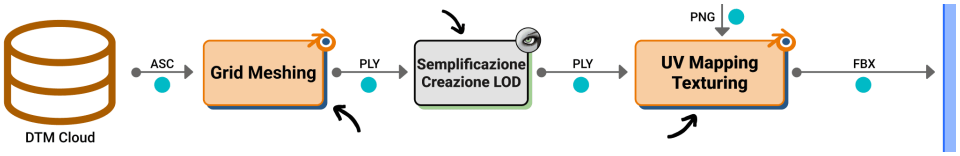
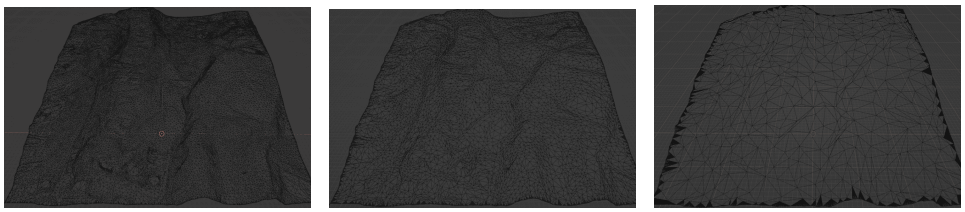


Figura 3.15: Sezione worflow: DTM

La pipeline in Fig. 3.15 mostra molto chiaramente le operazioni che sono state eseguite. I DTM sono stati forniti, come spiegato anche nel capitolo di analisi dati (sez. 2.2.1.1) in formato ASCII grid, e attraverso un approccio basato sulla *4-Connectivity* in Blender sono state



(a) LOD0 (b) LOD1 (c) LOD2

Figura 3.16: Visualizzazione LOD mesh DTM

ottenute le mesh. Successivamente, dato l'altissima densità poligonale di queste mesh, è stato necessario effettuare un processo di decimazione per ridurre il numero di poligoni e quindi alleggerire il peso della mesh e costruire i Level of Details. Questo processo è stato conseguito sfruttando Meshlab e la tecnica di *Quadric Edge Collapse Decimation*.

Infine, è stato effettutato il texturing delle mesh. A tale scopo sono state utilizzate le Ortofoto aeree, le quali sono state proiettate sulle mesh utilizzando Blender. In Fig. 3.16 è possibile vedere i LODs ottenuti e in Fig. 3.17 è possibile vedere la mesh texturizzata.

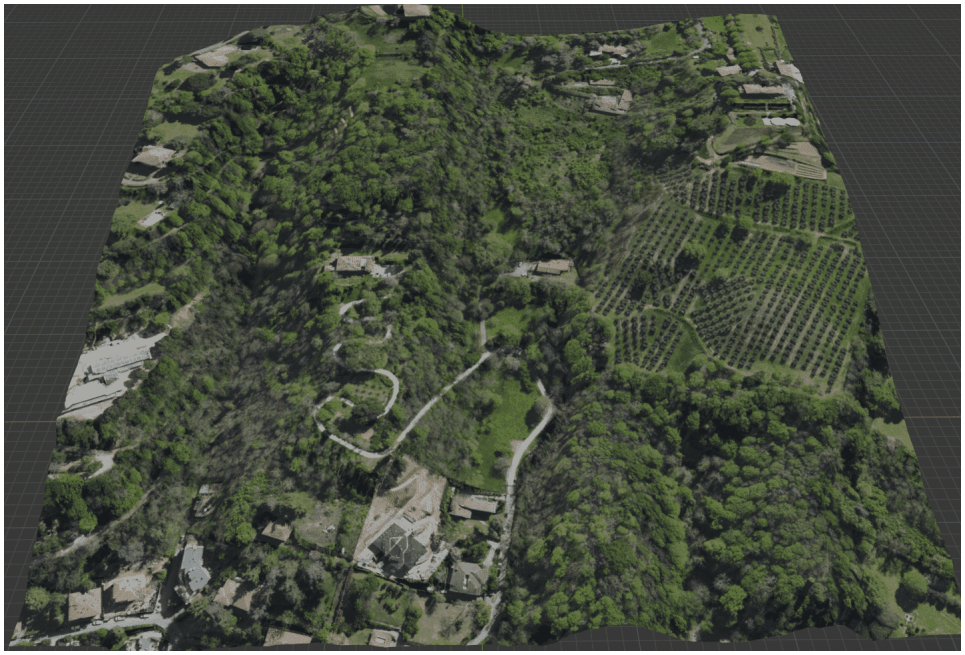


Figura 3.17: Sezione worflow: DTM

3.3 Processing di dati sul Verde Urbano

Uno dei primi task del progetto del Digital Twin è l'analisi del verde urbano presente nella zona metropolitana di Bologna.

All'interno del progetto questo task non riguarda solo la visualizzazione ma bensì anche l'estrazione di informazioni sul verde dalla nuvola di punti in modo tale da poter costruire dei modelli predittivi sulla crescita del verde e l'impatto sul clima. Ma dato che questa tesi è incentrata sulla computer grafica, verrà trattata solamente la parte di estrazione di dati sugli alberi, costruire mesh per ogni tipo di alberi e preparare un servizio che sarà collegato al Digital Twin che permetterà di avere informazioni sempre aggiornate sul verde della città.

Questo è il primo e vero task in ottica di Digital Twin in quanto permette all'ambiente di essere realmente "vivo", cioè aggiornato sia nei dati che nella visualizzazione man mano che viene fatto data ingestion sul database del DT.

In Fig. 3.18 è possibile vedere la pipeline che è stata seguita per portare a termine questo task.

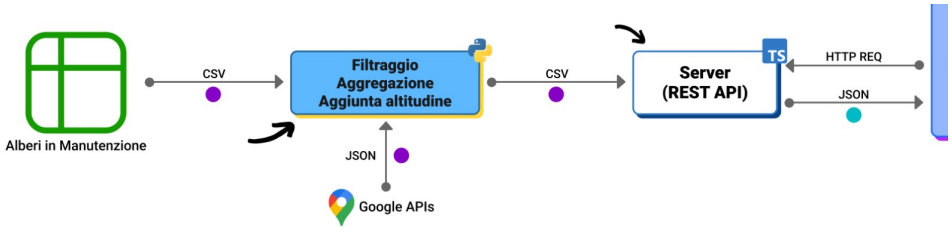


Figura 3.18: Sezione worflow: verde urbano

Il punto di partenza di questo task sta nel trovare dei dati aggiornati sul verde urbano. Dato che purtroppo siamo entrati nel progetto Bologna Digital Twin ai suoi albori ancora non sono state effettuate alcune operazioni di estrazione dati dalla nuvola di punti LiDAR, quindi non disponiamo di alcun dato elaborato. Quindi, come anticipato anche nella sezione di analisi di dati, per portare a termine questo compito è stato utilizzato il databse sul verde urbano degli OpenData del Comune di Bologna. Successivamente, attorno a quel database dopo una fase di attenta pulizia e aggregazione di dati verrà costruito un servizio web

tramite il quale l'ambiente 3D del Digital Twin si potrà interfacciare per essere costantemente aggiornato.

3.3.1 Preparazione e organizzazione dati

Allo stesso modo del *CTC degli edifici*, il dataset contenente le informazioni sugli alberi è uno strumento prezioso e un ottimo punto di partenza per effettuare le prime visualizzazioni in real-time sul Digital Twin ed eventualmente, una volta che saranno pronti gli strumenti necessari, utilizzarlo per effettuare le prime simulazioni. Tuttavia, nonostante le utili informazioni, proprio come il CTC degli edifici, non è esente da errori e problemi.

Analizzando nel dettaglio il dataset possiamo vedere che sono presenti molti dati utili, come ad esempio:

- **Specie Arborea**: che indica la specie a cui la pianta appartiene;
- **Classe di altezza**;
- **Classe di circonferenza**;
- **Geo Point**;

Nonostante ciò sono mancanti molte informazioni essenziali, come ad esempio la quota sul livello del mare dell'albero (il Geo Point indica solamente la coordinata in latitudine e longitudine, senza specificare l'altitudine) ed è presente una colonna intitolata **Data Impianto** ma è vuota per ogni albero.

Proprio come per il csv dei buildings, per fare pulizia dati e processing, anche per questo dataset è stato scritto un **Python Notebook** in modo tale da visualizzare attraverso dei grafici le operazioni di pulizia che man mano venivano effettuate.

Prima di andare nel dettaglio delle operazioni effettuate, è necessario fare un breve riassunto di tutto il processing sul dataset:

1. Conversione di coordinate di ogni albero;
2. Costruzione di grafici per visualizzare la distribuzione delle specie arboree;

3. Aggregazione di specie arboree simili;

4. Calcolo della quota sul livello del mare di ogni albero tramite le Google API.

Processing dataset

Andando nel dettaglio dell'elaborazione, la primissima operazione che è stata effettuata sul dataset coincide con quello che è stato fatto anche nel csv degli edifici, ovvero la conversione di coordinate. Anche questo dataset contiene coordinate rappresentate in WGS84 in gradi decimali, e quindi per mantenere uno standard comune con tutti i dati a nostra disposizione, tutti i Geo Point sono stati convertiti in WGS64 UTM sfruttando il pacchetto Python *utm* e il codice A.

Successivamente sono stati costruiti una serie di grafici per vedere la distribuzione degli alberi e delle specie arboree.

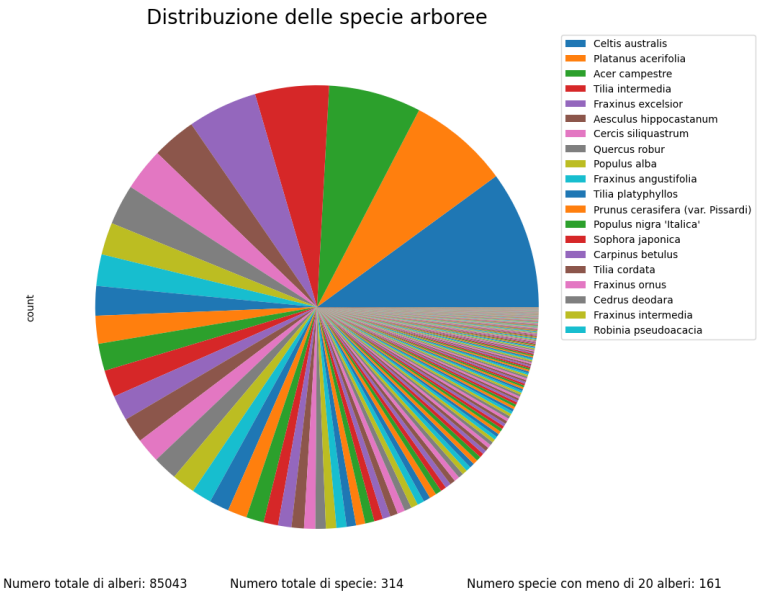


Figura 3.19: Distribuzione delle specie arboree

Si può notare in Fig. 3.19 che su circa 85000 alberi sono presenti 320 specie arboree e la distribuzione di tali specie non è uniforme. 320 specie arboree mettono in evidenza la grande biodiversità di verde presente nella città metropolitana di Bologna. Inoltre è stato conteggiato anche il numero di specie arboree che hanno meno di 20 alberi,

e sono circa 160. Ciò significa che più della metà delle specie sono rappresentate da piccole popolazioni all'interno del dataset.

Questo dataset contiene tutti gli alberi che sono presenti nella città metropolitana di Bologna, però dato che questo Digital Twin, in quanto proof of concept, verrà inizialmente sviluppato solo dell'area che comprende il centro storico della città, è più interessante costruire dei grafici di distribuzione che riguardano solo quell'area per avere un'idea più chiara.

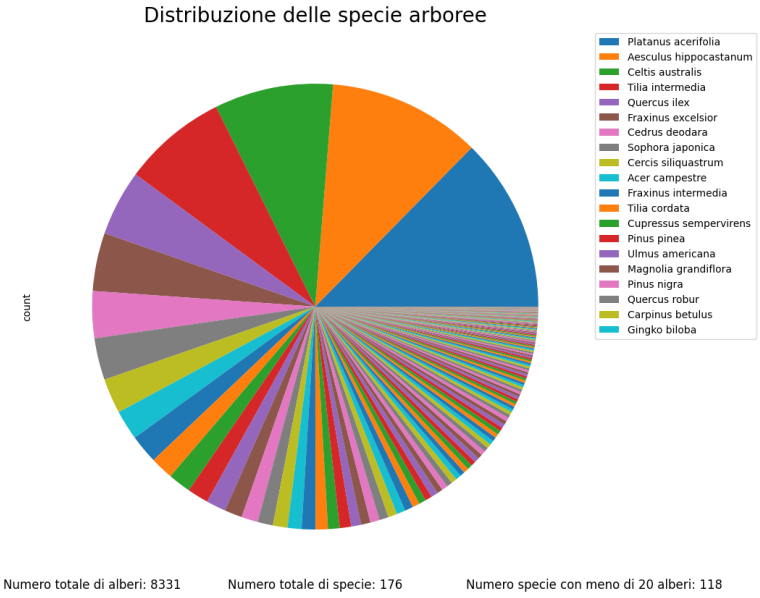


Figura 3.20: Distribuzione delle specie arboree nell'area del centro storico

Anche nel centro la tendenza è circa la stessa, sono presenti più della metà delle specie con una popolazione molto piccola, come dimostra la Fig. 3.20.

Quindi, dato che è impossibile nel tempo a disposizione modellare 176 mesh 3D di alberi differenti, uno per ogni specie arborea, è necessario effettuare una classificazione sulle specie.

Come si può notare anche nella legenda dei grafici, sono presenti specie arboree molto simili tra di loro. Nel senso che generalmente la specie arborea in questo dataset è costituita da 2 parole, una per indicare il *genere* dell'albero e l'altra per indicare la *specie*. Ad esempio in *"Celtis australis"*, "Celtis" è il genere e "australis" è la specie. Sa-

pendo ciò è possibile raggruppare tutte le specie in base al loro genere. Ad esempio *"Quercus ilex"* e *"Quercus robur"* si può raggruppare in *"Quercus"*. Successivamente, per ridurre ulteriormente il numero di specie arboree si può raggruppare sotto la categoria *"Other"* tutte le specie che hanno un numero di alberi inferiore a 50.

In questo modo otteniamo una distribuzione più uniforme e un numero totale di specie arboree pari a 26.

C'è da fare un piccolo appunto sullo scopo di queste operazioni. Questa aggregazione di dati viene fatta solo per facilitare la visualizzazione in quanto non sarebbe pensabile modellare tutte quelle mesh. Anche i Digital Twin molto più avanzati utilizzano mesh generalizzate. Comunque non vi è alcuna perdita di informazioni sulla specie arborea originale in quanto il nuovo valore non sovrascrive il precedente ma bensì viene inserito in una nuova colonna del dataset. Quindi nel Digital Twin sarà presente una mesh per aggregato di specie arborea, però se si interrogherà ogni singolo albero verranno mostrate tutte le informazioni originali.

Quindi stampando un grafo xy nel quale vengono posizionati gli alberi con un colore per ogni classe arborea, si ottiene il risultato mostrato in Fig. 3.21.

L'ultima operazione da effettuare sul dataset riguarda l'ottenimento della quota su cui giace ogni singolo albero in quanto questo dato è mancante del tutto.

Il procedimento è analogo a quanto fatto per la correzione della quota degli edifici. Ovvero utilizzando le **Google Elevations API** passando in ingresso la coordinata di ogni singolo albero.

3.3.2 Sviluppo ed implementazione della RestAPI

Una volta che i dati sugli alberi sono stati processati è necessario sviluppare un sistema affidabile e scalabile per accedere e manipolare questi dati. A tale scopo è stata sviluppata una **REST API** dedicata, ovvero un servizio web che aderisce ai principi dell'architettura REST (Representational State Transfer). Questo non è un lavoro di Computer Grafica, però è comunque un task fondamentale nella realizzazione di un proof of concept di un Digital Twin.

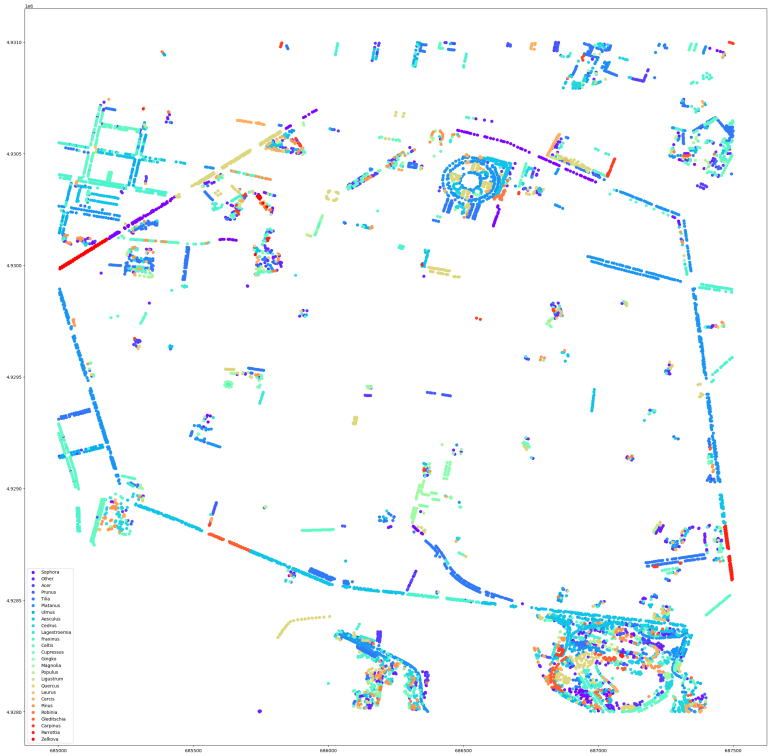


Figura 3.21: Mappa degli alberi del centro storico

Le REST API giocano un ruolo chiave nell'ecosistema di un Digital Twin, consentendo l'integrazione flessibile e l'interoperabilità tra diversi sistemi ed applicazioni. La loro importanza si può manifestare in vari modi:

- **Accesso Standardizzato**: forniscono un metodo standardizzato per l'accesso ai dati, consentendo agli sviluppatori e ai sistemi automatizzati di interagire con il Digital Twin senza la necessità di interfacce utente complesse.

- **Aggiornamenti in tempo reale**: permetto l'aggiornamento e la sincronizzazione in tempo reale dei dati tra il digital twin e le fonti dati esterne, assicurando che il modello digirale rifletta accuratamente lo stato attuale dell'ambiente urbano.

- **Integrazione di sistemi diversificati**: grazie alla loro natura agnostica rispetto alla piattaforma, le REST API possono essere

utilizzate per collegare una varietà di sistemi e dispositivi, inclusi ad esempio sensori IoT, database, applicazioni mobile, ecc.

- **Scalabilità**: le REST API supportano la scalabilità delle applicazioni, consentendo di gestire un numero crescente di richieste e di funzionalità senza compromettere le prestazioni o la stabilità del sistema.

Al fine di poter utilizzare e sviluppare una REST API è necessario prima avere un databse sul quale la REST API può essere utilizzata per effettuare query e ottenere i dati. Quindi, dato che l'Università di Bologna mette a disposizione crediti gratuiti da utilizzare nella piattaforma cloud **Microsoft Azure**, si è deciso di sfruttare questa opportunità per allestire un proof of concept dell'infrastruttura necessaria al Digital Twin, in quanto quella definitiva è ancora in via di sviluppo da parte dei partner del progetto *Bologna Digital Twin*.

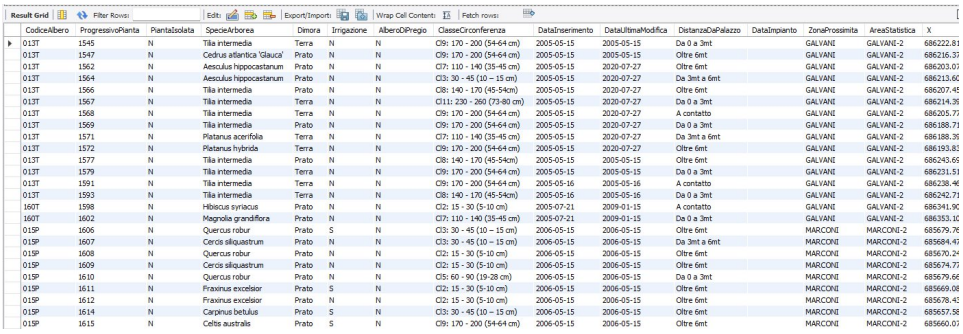
La scelta di Azure come piattaforma cloud mette a disposizione una vasta gamma di servizi per la gestione dati, come ad esempio **Azure MySQL Database**. Il quale permette di immagazzinare e gestire un grande volume di dati. Inoltre, la capacità di scalare dinamicamente le risorse a seconda delle necessità assicura che la piattaforma possa adattarsi all'aumento del carico di lavoro.

Andando nel dettaglio operativo, la prima cosa è stata creare un gruppo di risorse cloud all'interno del portale Azure. Un gruppo di risorse non è altro che un contenitore che tiene insieme risorse correlate tra di loro per una applicazione cloud.

La prima risorsa creata è il database MySQL. IL quale è stato impostato per garantire la scalabilità automatica e la ridondanza geografica. Questo è molto importante per garantire una affidabilità pari al 99.99%. All'interno del database è stata costruita la tabella che dovrà immagazzinare tute le informazioni sugli alberi del comune di Bologna. La tabella ha la stessa forma del dataset degli OpenData del comune, con l'unica differenza delle colonne nuove che sono state aggiunte, quindi: *Classe Arborea* e *Quota*.

Successivamente questo database è stato popolato attraverso uno script scritto in Python il quale, per ogni entry del dataframe degli alberi effettua una *INSERT* nel database.

In Fig. 3.22 è possibile vedere la tabella del database popolata con gli alberi ottenuti.



CodiceAlbero	ProgressivoPianta	PiantaIsolata	SpecieArborea	Dimora	Irrigazione	AlberoDiPregio	ClasseCirconferenza	DataInserimento	DataUltimaModifica	DistanzaDaPalazzo	DataImpianto	ZonaProssimita	AreaStatistica	X
013T	1545	N	Tilia intermedia	Terra	N	N	C9: 170 - 200 (54-64 cm)	2005-05-15	2005-05-15	Da 0 a 3mt		GALVANI	GALVANI-2	686222.81
013T	1547	N	Cedrus atlantica 'Glauca'	Prato	N	N	C9: 170 - 200 (54-64 cm)	2005-05-15	2005-05-15	Oltre 6mt		GALVANI	GALVANI-2	686216.37
013T	1562	N	Aesculus hippocastanum	Prato	N	N	C7: 110 - 140 (35-45 cm)	2005-05-15	2020-07-27	Oltre 6mt		GALVANI	GALVANI-2	686203.01
013T	1564	N	Aesculus hippocastanum	Prato	N	N	C3: 30 - 45 (10 – 15 cm)	2005-05-15	2020-07-27	Da 3mt a 6mt		GALVANI	GALVANI-2	686213.60
013T	1566	N	Tilia intermedia	Prato	N	N	C8: 140 - 170 (45-54cm)	2005-05-15	2020-07-27	Oltre 6mt		GALVANI	GALVANI-2	686207.45
013T	1567	N	Tilia intermedia	Terra	N	N	C11: 230 - 260 (73-80 cm)	2005-05-15	2020-07-27	Da 0 a 3mt		GALVANI	GALVANI-2	686214.39
013T	1568	N	Tilia intermedia	Terra	N	N	C9: 170 - 200 (54-64 cm)	2005-05-15	2020-07-27	A contatto		GALVANI	GALVANI-2	686205.77
013T	1569	N	Tilia intermedia	Prato	N	N	C9: 170 - 200 (54-64 cm)	2005-05-15	2020-07-27	Da 0 a 3mt		GALVANI	GALVANI-2	686188.71
013T	1571	N	Platanus acerifolia	Terra	N	N	C7: 110 - 140 (35-45 cm)	2005-05-15	2020-07-27	Da 3mt a 6mt		GALVANI	GALVANI-2	686188.39
013T	1572	N	Platanus hybrida	Terra	N	N	C9: 170 - 200 (54-64 cm)	2005-05-15	2020-07-27	Oltre 6mt		GALVANI	GALVANI-2	686193.81
013T	1577	N	Tilia intermedia	Prato	N	N	C8: 140 - 170 (45-54cm)	2005-05-15	2005-05-15	Oltre 6mt		GALVANI	GALVANI-2	686243.65
013T	1579	N	Tilia intermedia	Terra	N	N	C9: 170 - 200 (54-64 cm)	2005-05-15	2005-05-15	Da 0 a 3mt		GALVANI	GALVANI-2	686231.51
013T	1591	N	Tilia intermedia	Terra	N	N	C9: 170 - 200 (54-64 cm)	2005-05-16	2005-05-16	A contatto		GALVANI	GALVANI-2	686238.46
013T	1593	N	Tilia intermedia	Terra	N	N	C8: 140 - 170 (45-54cm)	2005-05-16	2005-05-16	Da 0 a 3mt		GALVANI	GALVANI-2	686242.71
160T	1598	N	Hibiscus syriacus	Prato	N	N	C2: 15 - 30 (5-10 cm)	2005-07-21	2009-01-15	A contatto		GALVANI	GALVANI-2	686341.90
160T	1602	N	Magnolia grandiflora	Prato	N	N	C7: 110 - 140 (35-45 cm)	2005-07-21	2009-01-15	Da 0 a 3mt		GALVANI	GALVANI-2	686353.10
015P	1606	N	Quercus robur	Prato	S	N	C3: 30 - 45 (10 – 15 cm)	2006-05-15	2006-05-15	Oltre 6mt		MARCONI	MARCONI-2	685679.76
015P	1607	N	Cercis siliquastrum	Prato	N	N	C3: 30 - 45 (10 – 15 cm)	2006-05-15	2006-05-15	Da 3mt a 6mt		MARCONI	MARCONI-2	685684.47
015P	1608	N	Quercus robur	Prato	N	N	C2: 15 - 30 (5-10 cm)	2006-05-15	2006-05-15	Oltre 6mt		MARCONI	MARCONI-2	685670.24
015P	1609	N	Cercis siliquastrum	Prato	N	N	C2: 15 - 30 (5-10 cm)	2006-05-15	2006-05-15	Oltre 6mt		MARCONI	MARCONI-2	685674.77
015P	1610	N	Quercus robur	Prato	N	N	C5: 60 - 90 (19-28 cm)	2006-05-15	2006-05-15	Da 0 a 3mt		MARCONI	MARCONI-2	685679.66
015P	1611	N	Fraxinus excelsior	Prato	S	N	C2: 15 - 30 (5-10 cm)	2006-05-15	2006-05-15	Oltre 6mt		MARCONI	MARCONI-2	685669.08
015P	1612	N	Fraxinus excelsior	Prato	N	N	C2: 15 - 30 (5-10 cm)	2006-05-15	2006-05-15	Oltre 6mt		MARCONI	MARCONI-2	685678.43
015P	1614	N	Carpinus betulus	Prato	S	N	C3: 30 - 45 (10 – 15 cm)	2006-05-15	2006-05-15	Oltre 6mt		MARCONI	MARCONI-2	685657.58
015P	1615	N	Celtis australis	Prato	S	N	C9: 170 - 200 (54-64 cm)	2006-05-15	2006-05-15	Oltre 6mt		MARCONI	MARCONI-2	685660.07

Figura 3.22: Schema parziale del database degli alberi

Una volta messo online il database MySQL si è sviluppata la REST API che permette tramite richieste http di interagire direttamente con il database. La REST API è stata sviluppata utilizzando le **Azure Functions**. Le Azure Functions sono un servizio messo a disposizione da Azure che permette di sviluppare dei veri e proprio microservizi che vengono invocati ed eseguiti solo alla necessità. Esse sono un servizio serverless, ovvero permettono di costruire e distribuire API senza doversi preoccupare della gestione dell'infrastruttura sottostante. Ciò significa che l'unico focus da avere è sulla logica di business e sul codice, mentre Azure gestisce automaticamente la scalabilità, la manutenzione e l'high availability del servizio.

La Azure Function è stata sviluppata in **Typescript**, sopra al framework **Node.js**. La scelta di Typescript è dovuta al fatto che innanzitutto è un superset di Javascript il quale lo rende molto più affidabile soprattutto dal punto di vista della tipizzazione statica. Inoltre Typescript è sviluppato e manutenuto direttamente da Microsoft e quindi ha una perfetta integrazione con tutti i servizi Azure. Infine, rispetto ad altri framework web come ad esempio **.NET**, Typescript permette una molto più rapida prototipazione della API, garantendo così di andare in produzione molto più rapidamente.

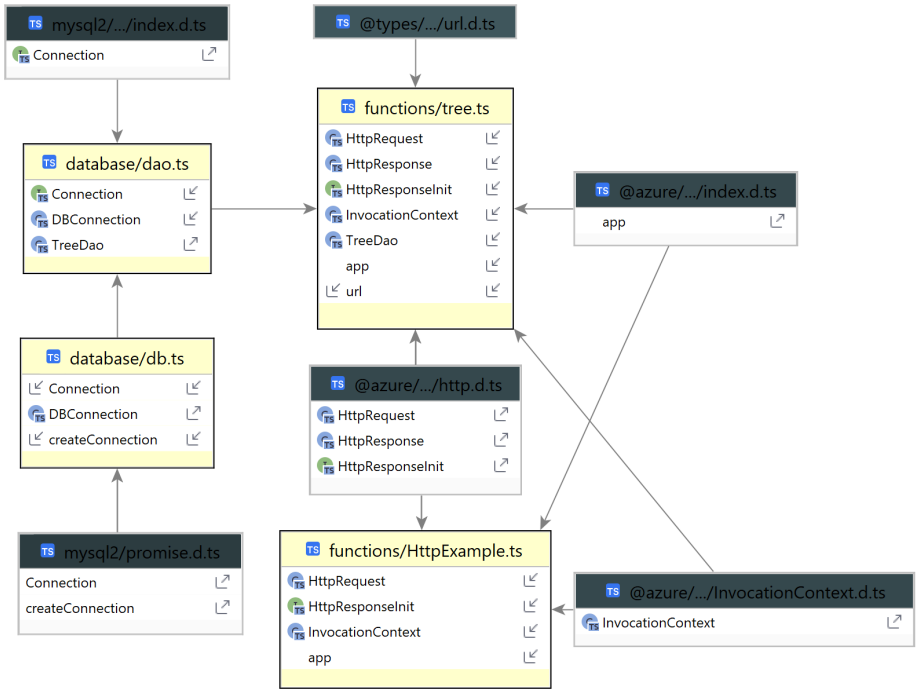


Figura 3.23: Diagramma uml della REST API

Andando nel dettaglio dell'implementazione della API, sono stati sviluppati 4 componenti (Fig. 3.23):

Modello degli Alberi ('model/Tree.ts')

Questo file definisce il tipo *Tree*, rapresentando la struttura dati di un albero nella città. Ogni attributo corrisponde ad una colonna del database precedentemente creato.

La definizione di *Tree* funge da contratto per i dati che verranno gestiti attraverso la REST API, assicurando che tutte le operazioni sul database manipolino oggetti che rispecchiano questa struttura.

Connettore Database ('database/db.ts')

Il file *db.ts* definisce la classe **DBConnection**, che gestisce la connessione al database MySQL. Utilizza il modulo *mysql2/promise* per creare una connessione asincrona che supporta promesse, facilitando la gestione delle operazioni del database in modo asincrono.

- **Metodo 'open'**: verifica se esiste già una connessione attiva; in caso contrario, ne apre una nuova utilizzando le credenziali e le impostazioni del database specificate nelle variabili d'ambiente di Azure.

- **Metodo 'close'**: chiude la connessione al database quando non è più necessaria, liberando risorse e prevedendo potenziali perdite di memoria.

Data Access Object (DAO) ('database/dao.ts')

Il file *dao.ts* introdice la classe **TreeDao**, che incapsula la logica per interagire con i dati degli alberi nel database. DAO è un modello architetturale che permette di separare la logica di accesso ai dati dal resto dell'applicazione. Quindi consente di incapsulare l'accesso ai dati in classi specifiche riducendo la complessità del codice e migliorando la manutenibilità.

La classe TreeDao utilizza la classe DBConnection per stabilire la connessione al database e fornisce metodi specifici per eseguire le operazioni CRUD sugli alberi.

- **Metodi *'getTrees'*, *'getTree'*, *'deleteTree'*, *'createTree'***: ogni metodo implementa una specifica operazione del database, come recuperare tutti gli alberi, ottenere un singolo albero per ID, eliminare un albero, o aggiungere uno nuovo. Questi metodi utilizzano la connessione al database per eseguire query SQL e restituire i risultati in formato JSON, aderendo alla struttura del tipo *Tree*.

Azure Function ('functions/tree.ts')

Il file *tree.ts* definisce la Azure function HTTP che gestisce le richieste estere alla REST API. Utilizza il modello e i DAO definiti precedentemente per eseguire operazioni sui dati degli alberi in base al tipo di richiesta ricevuta.

- **Routing delle richieste**: la funzione analizza il metodo HTTP e i parametri della richiesta per determinare l'azione da eseguire.

Per le richieste *GET*, può recuperare tutti gli alberi o un singolo albero specificato da un ID. Per le richieste *POST* è possibile aggiungere un albero al database passando nel body i campi dell'oggetto Tree. E le richieste *DELETE* può eliminare l'albero specificato dall'ID.

- **Gestione delle riposte**: ogni operazione restituisce un oggetto **HttpResponseInit**, che include lo stato della risposta, un body contenente i dati richiesti (in formato JSON), e gli eventuali header necessari.

- **Registrazione della funzione**: infine, la funzione è registrata per gestire le richieste HTTP con un determinato percorso e metodo, configurando anche il livello di autenticazione.

Una volta sviluppata, la Azure Function è pronta per essere compilata ed venire *"deployata"* sul cloud di Azure, in modo che venga gestita automaticamente la replicazione e la scalabilità. Non appena la funzione sarà online basterà fare una HTTP request all'url della azure function, specificando il tipo (GET, POST, DELETE), ed essa ritornerà il risultato.

Concludendo, l'implementazione di questa Azure Function per interagire con gli alberi della città di Bologna rappresenta un altro tassello del puzzle nella realizzazione di una proof of concept di un Digital Twin della città di Bologna. Il prossimo passo, sarà proprio lo sviluppo del vero e proprio ambiente 3D del Digital Twin.

3.4 Elaborazione dati LiDAR per ottenere i tetti

Verrà qui presentato brevemente il lavoro svolto dal mio collega per l'ottenimento delle mesh tetti dei tetti sfruttando la nuvola di punti LiDAR. Questo è un task estremamente complesso in quanto sono state utilizzate tecniche appartenenti a campi differenti, ad esempio la computer vision, la computer grafica e la geometria computazionale.

In Fig. 3.24 è possibile vedere la pipeline che è stata seguita per ottenere le mesh dei tetti.

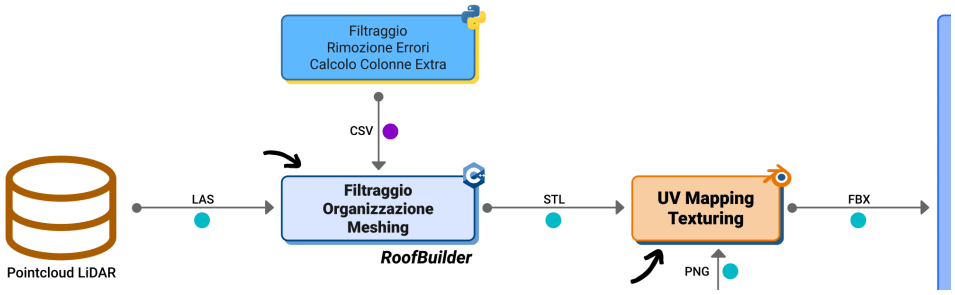


Figura 3.24: Sezione workflow: tetti

Per compiere questo task è stato sviluppato un software in C++ che, prendendo in ingresso la nuvola di punti LiDAR e il csv degli edifici 2.2.1.1, è in grado di ottenere le mesh dei tetti. Andando leggermente nel dettaglio, questo software esegue i seguenti passaggi:

1. filtraggio dei punti LiDAR per ottenere solamente quelli che appartengono ad ogni singolo tetto;
2. clustering per eliminare eventuali punti che non appartengono al tetto;
3. organizzazione di punti in griglia e conversione in immagine;
4. ottenimento punti del bordo esterno tramite tecniche di Computer Vision;
5. triangolazione tramite Delaunay;
6. ottenimento dei punti di gronda;
7. triangolazione finale aggiungendo i punti di gronda;

Una volta ottenute le mesh dei tetti, queste sono state texturizzate con le ortofoto aeree attraverso la costruzione di un Atlas, infine sono state esportate in formato FBX per essere utilizzate all'interno del Digital Twin.

Dato che il software sviluppato e la logica dietro agli algoritmi utilizzati è molto complessa, si consiglia la lettura della tesi del mio collega per avere una spiegazione molto più dettagliata [19].

Qui di seguito sono presentati i risultati da lui ottenuti. In Fig. 3.25 è possibile vedere il processo passo passo per ottenere le mesh dei tetti, mentre in Fig. 3.26 è possibile vedere il risultato finale, insieme anche a DTM ed edifici.

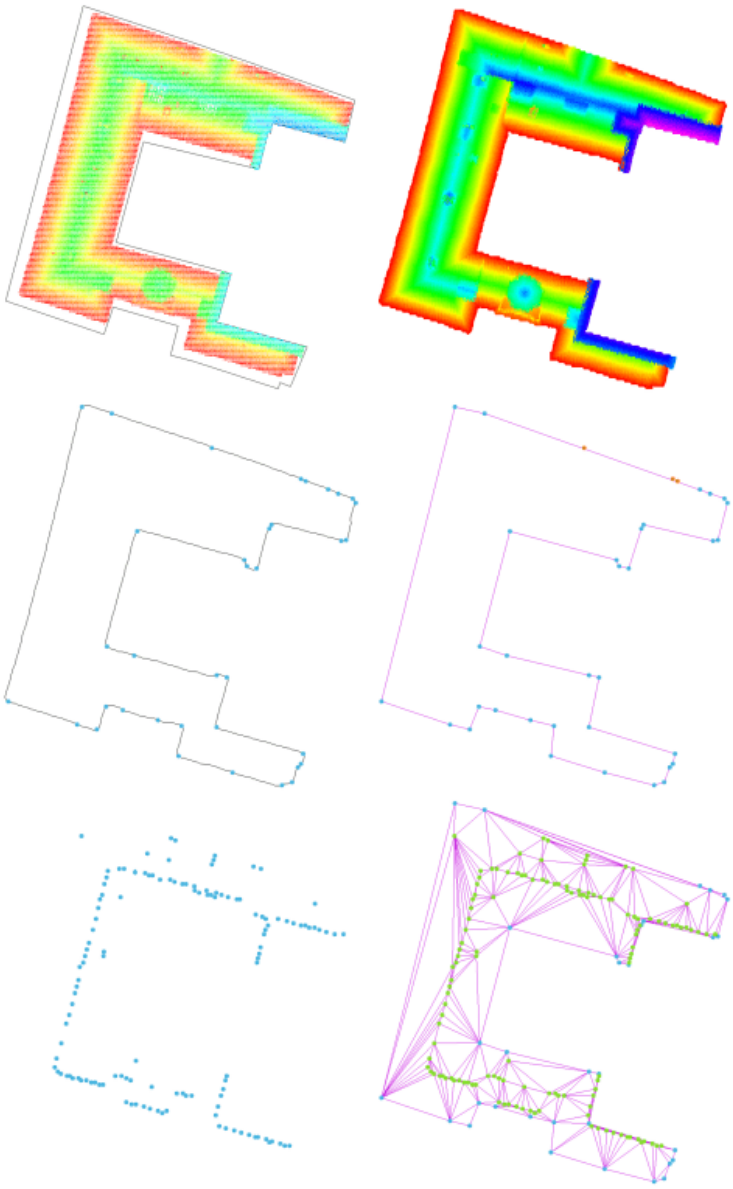


Figura 3.25: Step by step del processo di ottenimento dei tetti

Figura 3.26: Visualizzazione finale dei tetti sul Digital Twin

Capitolo 4

Sviluppo del Digital Twin statico

Contents

4.1	**Importazione dei modelli 3D della città**	**75**
4.2	**Modellazione degli alberi**	**79**
4.3	**Creazione della scena statica del Digital Twin** .	**82**

4.1 Importazione dei modelli 3D della città

Una volta creato un progetto vuoto di Unreal Engine, il primo passo è stato importare le mesh 3D di ogni elemento da visualizzare all'interno del Digital Twin. Questa fase coninvolge l'integrazione degli elementi ottenuti nel capitolo precedente, come il modello del terreno (DTM), gli edifici, e i tetti. Inoltre verranno aggiunti anche i modelli 3D degli alberi da posizionare utilizzando la API precedentemente sviluppata.

Di seguito verranno descritti i tipi di dati importati e i passaggi per ciascuna categoria di mesh.

Mesh del terreno (DTM)

Le mesh del terreno, semplificate e con i LOD costruite sono state esportate in FBX, quindi direttamente importabili all'interno del progetto di Unreal Engine.

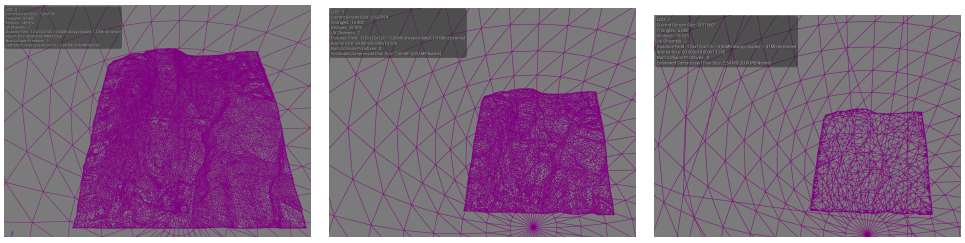
Su queste mesh sono state effettuate una serie di operazioni e tuning.

Innanzitutto, dato che le mesh sono state esportate direttamente con i LOD, essi devono essere impostati all'interno dell'editor, altrimenti risulterebbero inutili. I LOD, o Level of Detail, sono diverse versioni della stessa mesh ma con il numero di poligoni differente, vengono utilizzati per alleggerire il carico computazionale. In base alla distanza della camera dalla mesh si renderizza un LOD differente, più si è vicini più il modello è ad alta risoluzione, e viceversa.

Per ogni mesh importata si imposta il valore di *Screen Size* di ogni LOD. Questo parametro rappresenta una misura di quanto grande deve apparire una mesh sullo schermo. É un valore compreso tra 0 e 1, dove 1 significa che la mesh occupa l'intera altezza dello schermo, e 0 indica che è infinitamente lontana. Nel nostro caso sono stati impostati i seguenti valori:

- **LOD0**: 1
- **LOD1**: 0.75
- **LOD2**: 0.5

In Fig. 4.1 è possibile vedere i LOD impostati visualizzati in base alla distanza della camera.



(a) LOD0 (b) LOD1 (c) LOD2

Figura 4.1: LOD visualizzati in wireframe

Dopo aver impostato il valore di LOD è stato necessario sistemare i poligoni di collisione della mesh. Le collisioni sono fondamentali in quanto è stata sviluppata una modalità di navigazione immersiva in prima persona all'interno del DT, quindi avere delle collisioni corrette con la mesh è stato di primaria importanza.

Generalmente per le collisioni si utilizzano delle strutture che possono essere semplici come cubi, cilindri, oppure strutture più complesse che possono coincidere con i poligoni della mesh (Fig. 4.2).

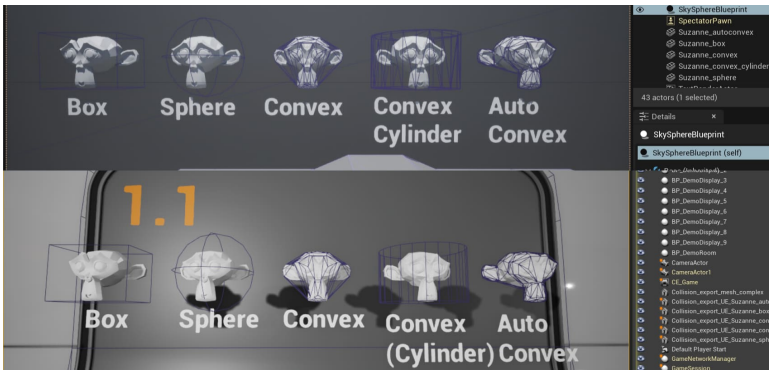


Figura 4.2: Diverse tipologie di collisioni

Con queste mesh era impossibile utilizzare le collisioni semplici in quanto il modello del terreno è un oggetto discontinuo ed irregolare. Quindi si è obbligati ad utilizzare il modello poligonale della mesh come modello di collisione. Ciò però aumenta esponenzialmente i calcoli che l'engine deve fare per capire se qualche oggetto sta collidendo. Per alleggerire questa computazione, Unreal mette a disposizione di scegliere quale LOD della mesh utilizzare per approssimare le collisioni, quindi -ovviamente- è stata scelta la LOD0 (Fig. 4.3). Questa scelta, per ovvie ragioni, è possibile solo su modelli in cui sono presenti LOD.

Infine, l'ultima operazione che è stata fatta ai modelli del terreno riguarda il materiale. La mesh è stata esportata da Blender con il parametro di *Roughness* (ruvidità) del materiale a 1, ciò significa un materiale ruvido e non riflessivo. Ma non vi è una conversione 1 a 1 da materiale Blender a materiale Unreal e quindi questo parametro non risultava impostato, risultando così in un modello con un materiale riflettente. Per rendere questo materiale più realistico possibile è stata impostata la roughness a 0.6 e in aggiunta è stata inserita una normal map che da l'impressione di rugosità sulla geometria, ma senza alterare la geometria. Il materiale è visibile in Fig. 4.4.

Mesh degli edifici

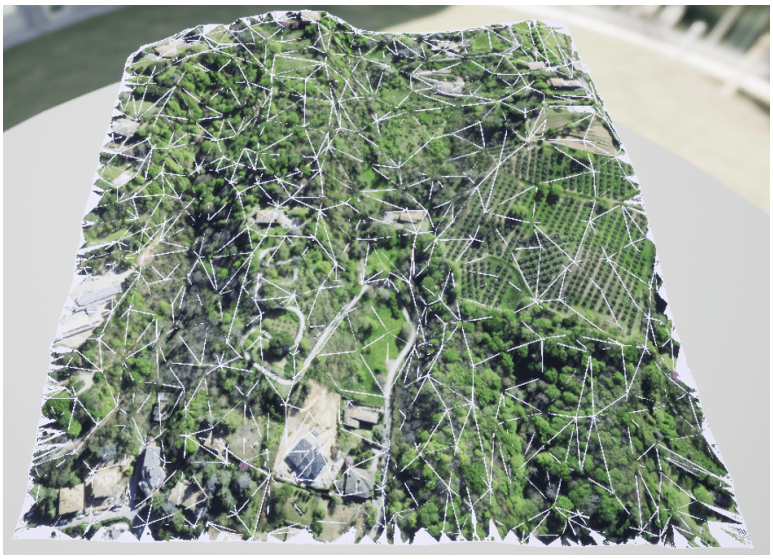


Figura 4.3: Visualizzazione poligoni di collisione

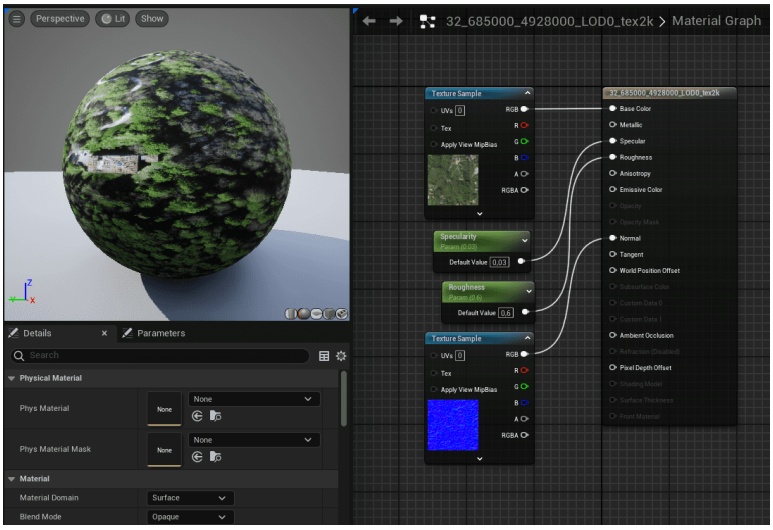


Figura 4.4: Materiale di un DTM

Anche sulle mesh degli edifici sono state effettuate le stesse operazioni, all'infuori della gestione dei LOD. Questo in quanto le mesh degli edifici, essendo dei parallelepipedi estrusi, hanno il minor numero possibile di triangoli e quindi tecniche di Level of Details sono superflue.

Invece, per quanto riguarda la gestione delle collisioni si è utilizzato la stessa tecnica. La topologia dei poligoni della mesh funge anche da

topologia per le collisioni, come si può vedere in Fig. 4.5.

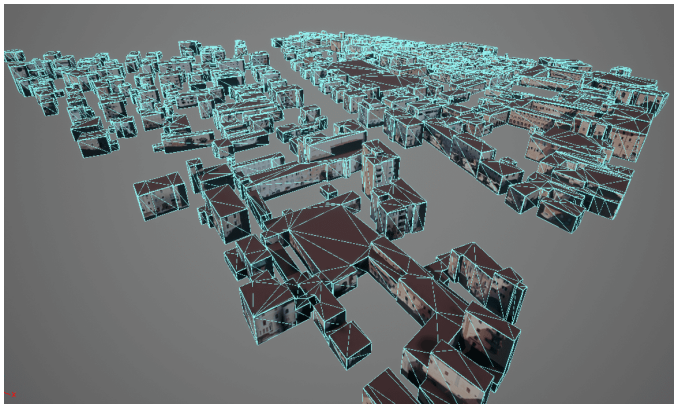


Figura 4.5: Visualizzazione delle collisioni dei buildings

Mesh dei tetti

Le mesh dei tetti (Fig. 4.6), invece richiedono meno lavoro rispetto alle precedenti. Innanzitutto non sono presenti LOD, come le mesh degli edifici. Ma soprattutto non è neccessaria -per ora- nemmeno una gestione delle collisioni, dato che sarà impossibile collidere con queste mesh in navigazione immersiva in prima persona. Quindi per evitare di fare dei calcoli inutili e di conseguenza appesantire la computazione si è optato di eliminare completamente la collisione della mesh.

Però c'è da dire che le mesh dei tetti, rispetto a tutte le altre mesh differiscono leggermente. Tutte le altre mesh (rappresentanti un tile 500x500), ogni mesh è associata al proprio materiale. Invece per i tetti è presente un materiale ogni 5 mesh contigue (5 tile vicini). Questo è dovuto al fatto che in fase di texturing è risultato più comodo proseguire per questa strada, soprattutto per alleggerire le texture atlas contente i tetti dato che sono ad altissima risoluzione.

4.2 Modellazione degli alberi

Nel capitolo precendente è stata sviluppata una REST API che permette di accedere ad informazioni dettagliate sugli alberi della città. Tuttavia non disponiamo di alcun modello 3D di alberi da poter inse-

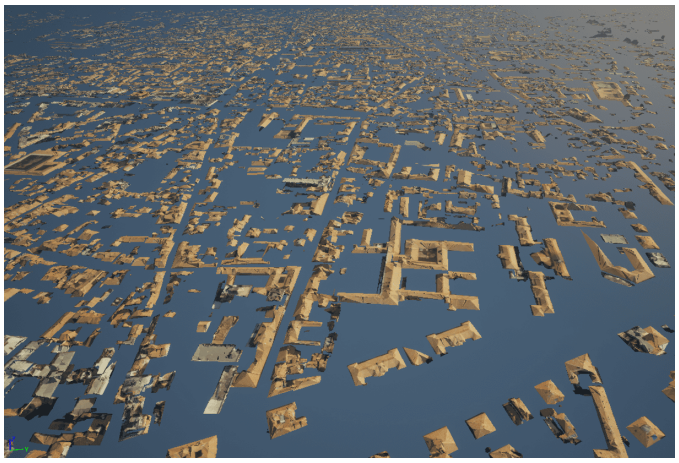


Figura 4.6: Mesh dei tetti del centro

rire all'interno del Digital Twin per sfruttare al meglio questa REST API.

Quindi si è reso necessario procedere con la modellazione degli alberi. Per svolgere questo compito è stato utilizzato **Tree It** (Fig. 4.7), un software gratuito che permette di modellare facilmente e velocemente modelli 3D di alberi in maniera realistica e dettagliata.

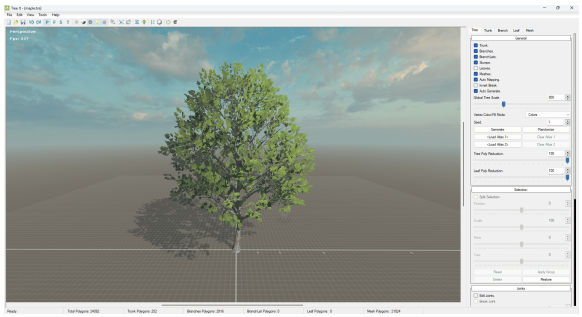


Figura 4.7: Tree It

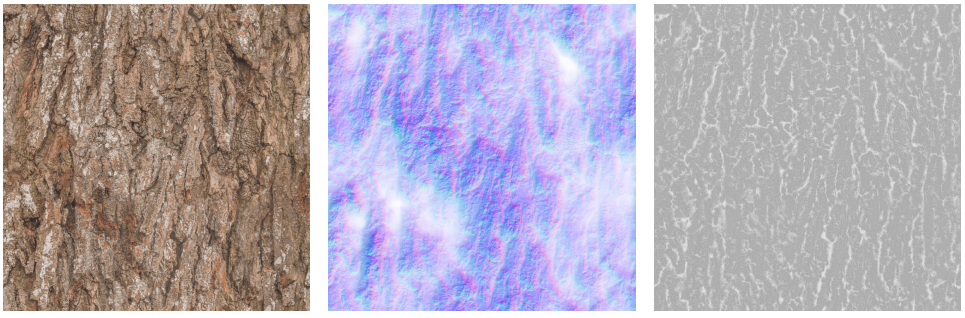
Nel nostro database sono presenti 19 tipologie di alberi, molti dei quali però hanno un numero relativamente basso di esemplari. Quindi, per semplicità sono stati modellati solamente gli alberi con il più elevato numero di esemplatri. Questo ha portato ad avere 6 modelli di alberi, tra cui:

- **Acero**

- **Quercia**
- **Pino**
- **Olmo**
- **Platano**
- **Betulla**

Attraverso TreeIt è quindi possibile costruire molto velocemente modelli 3D di alberi. Esso permette di modellare come entità separate il tronco, i rami e le foglie. Questi tre elementi possono essere modellati attraverso dei parametri modificabili da interfaccia grafica. Ad esempio per il tronco è possibile impostare: numero di segmenti (quindi di poligoni), lunghezza, larghezza, raggio, curvatura, ecc. Per i rami invece sono presenti molti più parametri da impostare in quanto è possibile modellare ricorsivamente ramificazioni di ramificazioni. Per le foglie invece è possibile impostare la dimensione, la densità, la distribuzione sui rami, ecc.

Ovviamente sono necessarie delle texture per rendere realistici i modelli degli alberi. Fortunatamente TreeIt mette a disposizione un ampio set di texture da utilizzare per modellare molte tipologie di alberi. Nel dettaglio fornisce texture per il tronco, per le foglie. In Fig. 4.8 e 4.9 è possibile vedere un esempio di texture di un acero.



(a) Texture (b) Normal map (c) Roughness

Figura 4.8: Texture del tronco

Tutti questi alberi sono stati generati a 4 diverse risoluzioni poligonali, in modo da poter costruire i LOD. I Level of Details nelle mesh

(a) Texture (b) Normal map (c) Roughness

Figura 4.9: Texture della foglia

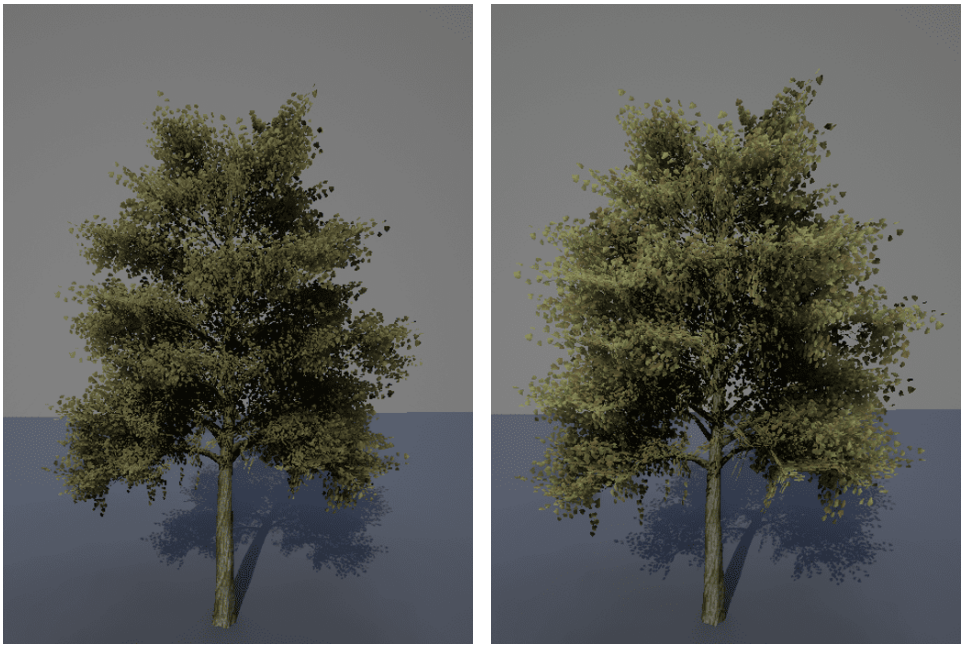
degli alberi sono fondamentali in quanto, per loro natura, hanno molta geometria. Quindi per alleggerire il carico di rendering è di fondamentale importanza avere dei LOD ben fatti. Essi sono stati costruiti andando a modificare il parametro *number of segments* direttamente da TreeIt, mentre per l'ultimo LOD è stata utilizzata direttamente una texture 2D che rappresenta l'albero. In Fig. 4.10 è possibile vedere i LOD di un albero.

4.3 Creazione della scena statica del Digital Twin

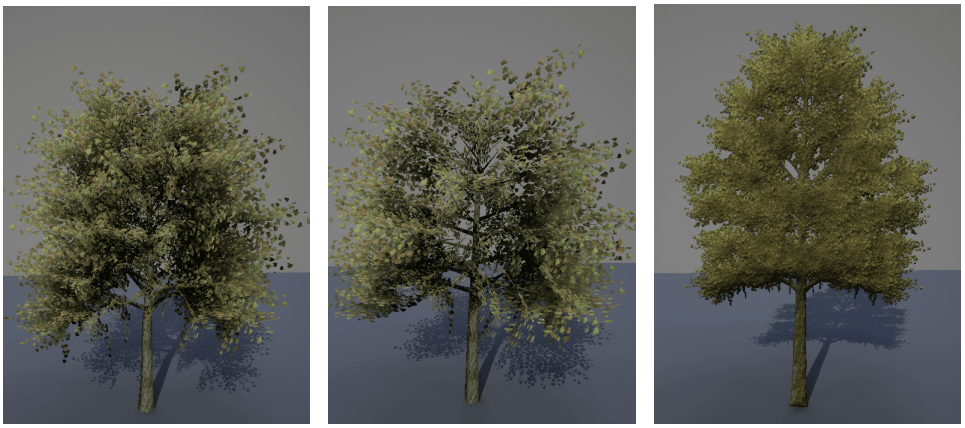
Non appena tutte le mesh 3D sono state importate all'interno del progetto di Unreal Engine, è stato possibile procedere con la creazione della vera e propria scena del Digital Twin. Ovvero l'ambiente 3D che servirà da base per tutto ciò che riguarda la visualizzazione del Digital Twin.

Dato che nei capitoli precedenti avevamo come partenza dati georeferenziati, di conseguenza anche le mesh che abbiamo ottenuto sono a loro volta georeferenziate. E dato che è stato utilizzato come standard all'interno di tutte le fasi del progetto la suddivisione di ogni modello in tile 500x500 georeferenziato, è stato molto semplice importare le mesh all'interno della scena. Non è stato necessario alcun tipo di trasformazione, in quanto le mesh sono già allineati di loro natura.

In riferimento alla Fig. 4.11, si può vedere cosa si ottiene importando tutte le mesh 3D della città.



(a) LOD0 (b) LOD1



(c) LOD2 (d) LOD3 (e) LOD4

Figura 4.10: LOD degli alberi

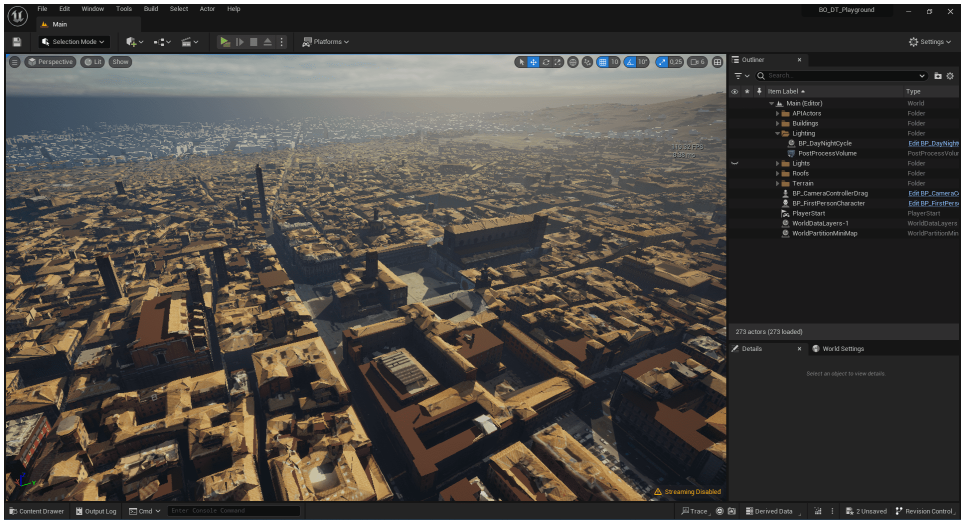


Figura 4.11: Previsualizzazione centro Bologna in Unreal

Capitolo 5

Sviluppo della dinamicità del Digital Twin

Contents

5.1 Struttura di un progetto Unreal Engine 85
5.2 Introduzione all'intera infrastruttura progettata 86
5.3 Sviluppo del core di Bologna Digital Twin . . . 87
5.3.1 Sviluppo della classe GameMode 87
5.3.2 Implementazione della navigazione 89
5.3.3 Implementazione della Ciclo Giorno/Notte 92
5.4 Collegamento con dati esterni 95
5.4.1 Alberi 97
5.4.2 Precipitazioni 102
5.4.3 Traffico 103
5.5 Profiling prestazioni 104
5.5.1 Profiling Navigazione e Layer di Visualizzazione . 107
5.5.2 Profiling Alberi 110

5.1 Struttura di un progetto Unreal Engine

Prima di andare nel dettaglio dell'implementazione è fondamentale spiegare brevemente la struttura di un progetto Unreal Engine. Un progetto è costituito da una serie di elementi fondamentali che permettono di costruire scene ed interazioni. Tra questi vi sono:

- **GameMode**: classe che definisce la logica complessiva di gioco, e il flusso di esecuzione. Si occupa di specificare la classe *Pawn*, ovvero "l'avatar" controllabile dall'utente, e configura tutti i parametri e variabili utili durante l'esecuzione.
- **Controller**: classe che gestisce tutti gli input da parte dell'utente, traducendoli in azioni all'interno del gioco. Il Controller fa da intermediario tra l'utente e la classe Pawn.
- **Game State**: mantiene tutte le informazioni globali sullo stato di gioco, visibili a tutti gli attori presenti in scena.

5.2 Introduzione all'intera infrastruttura progettata

Prima di immergersi nella spiegazione di tutte le funzionalità implementate all'interno del Digital Twin risulta fondamentale prima presentarle una ad una molto brevemente. Tutta l'implementazione si basa su 3 categorie di elementi sviluppati, visibili in Fig. 5.1, i quali sono:

- **Core**: il core contiene il cuore dell'implementazione del Digital Twin. É costituito da due elementi chiave: la **GameMode** e il **Controller**. Rispettivamente la GameMode funge da punto d'incontro tra tutti gli attori presenti in scena, mettendo a disposizione funzioni di comune utilità. Invece il Controller è l'attore che permette di raccogliere gli input da parte dell'utilizzatore e tradurli in movimento all'interno della scena. É l'elemento che permette di navigare all'interno della città. Inoltre in questa categoria rientra anche l'attore che gestice il ciclo giorno notte all'interno del Digital Twin.
- **DataBridge**: contiene tutti quegli attori che prendono le informazioni dall'esterno e popolano ed aggiornano il Digital Twin. Sono coloro che rendono veramente "vivo" il DT.
- **UI**: insieme di attori e **Widget** che permettono di costruire e definire l'interfaccia grafica dell'applicativo.

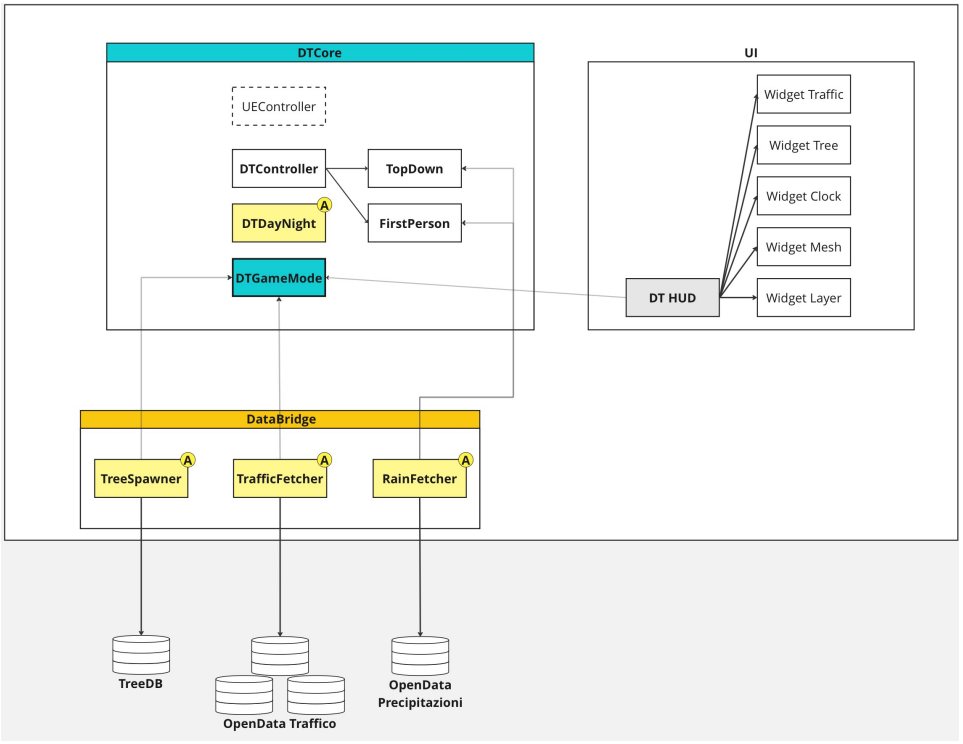


Figura 5.1: Diagramma dell'architettura del Digital Twin

5.3 Sviluppo del core di Bologna Digital Twin

In questa sezione si parlerà della vera e propria implementazione del Digital Twin, fino ad ora si è discusso solamente di elementi 3D statici. Ora questi elementi serviranno per rendere il Digital Twin un ambiente "vivo" e fedele alla realtà.

5.3.1 Sviluppo della classe GameMode

Il pilastro centrale su cui si poggia tutta l'implementazione del Digital Twin è la *GameMode*. Per questo elemento è stata sviluppata la classe base **DTGameMode** in C++. Questa classe funge da cuore pulsante del Digital Twin, mantenendo i riferimenti a tutti gli elementi chiave dell scena, quali

- Array contenente le mesh degli edifici,
- Array contenente le mesh dei tetti,

- Array contenente i materiali di tutti gli edifici,
- Array contenente i materiali di tutti i tetti.

La centralizzazione di questi riferimenti nella GameMode offre un accesso globale agevole, rendendoli disponibili a qualiasi attore o componente all'interno dell'ambiente di gioco. In questo modo, le funzionalità essenziali possono essere eseguite in maniera efficiente, senza la necessità di duplicare il codice o di riferimenti incrociati complessi.

- **PopulateMeshArray()**: metodo che viene chiamato dalla GameMode stessa non appena viene premuto "play", si occupa di popolare gli array di riferimenti alle mesh sopra citati.
- **SwapAllMaterials()**: permette di cambiare il materiale a tutte le mesh presenti in scena. Questo è utile in quanto permette di definire dei layer di visualizzazione differenti. Ad esempio non tutti gli utenti del Digital Twin potrebbero essere interessati a vedere tutti gli edifici e tetti con texture, soprattutto in caso in futuro si vogliano fare delle simulazioni.
- **ToggleActorsVisibility()**: metodo che da la possibilità di rendere visibile o invisibile una lista di mesh. Ad esempio se si vogliono nascondere tutti gli edifici ma non il terreno.

Il Blueprint che estende questa classe (Fig. 5.2) invece definisce dei *Custom Event*, ovvero eventi chiamabili da chiunque conosca la GameMode (quindi tutti). Sono stati definiti 4 custom event che permettono di chiamare le funzioni C++ scritte nella classe base, e sono stati predisposti in modo da essere chiamati dal Layer della UI, in modo tale da rendere separata la logica di business dalla logica di UI. Nel dettaglio, i 4 eventi sono:

- **Toggle Roofs**: chiama la funzione C++ *toggleActorsVisibility* passando in ingresso l'array di riferimenti alle mesh dei tetti e si occupa di rendere visibili/invisibili appunto questi elementi.
- **Toggle Buildings**: identico a ToggleRoofs ma con la differenza che viene passato in ingresso l'array delle mesh degli edifici.

- **Toggle Trees**: non chiama alcuna funzione C++, e si occupa di rendere visibili/invisibili gli alberi.
- **Swap Materials**: chiama la funzione C++ *SwapAllMaterials*.

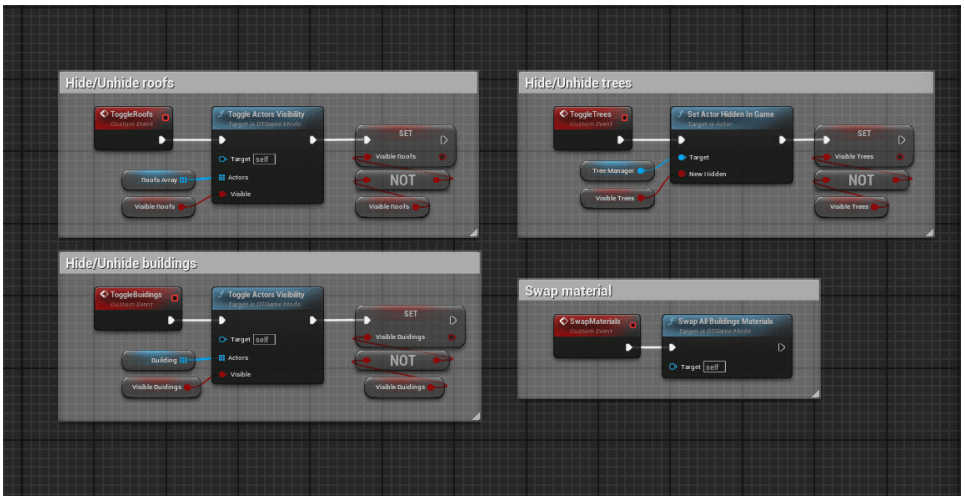


Figura 5.2: Blueprint della GameMode

Tutte queste funzioni vengono chiamate dalla UI, è infatti stato sviluppato un **HUD** contenente una serie di **Widget** (ovvero elementi riutilizzabili per l'interfaccia grafica) che permettono di interagire con il Digital Twin. Questa parte di UI permette solamente di scegliere dei layer di visualizzazione differenti, utilizzando i metodi della GameMode sopra citati. Ad esempio se un utente è interessato a vedere solamente gli alberi senza avere edifici e tetti, o viceversa. Inoltre è possibile visualizzare gli edifici e i tetti con o senza texture (Fig. 5.3).

5.3.2 Implementazione della navigazione

In questa sezione verrà spiegata l'implementazione del **Controller**. L'idea di base è quella di implementare la navigazione tradizionale dei software di mappe, ad esempio *Google Maps* o *Google Earth*, nel quale per navigare in scena è possibile utilizzare solamente il mouse semplicemente cliccando e trascinando. Inoltre, dato che uno dei requisiti definiti nelle fasi iniziali del progetto è l'immersività, allora è stato implementato anche una modalità di navigazione della scena in prima

Figura 5.3: Visualizzazione degli edifici senza texture

persona. Quest'ultima modalità di navigazione é stata pensata sopratutto per gli utenti normali del Digital Twin, ovvero i cittadini, i quali, proprio come in Google Earth possono navigare per le strade di Bologna in prima persona. Inoltre è stata anche pensata per una futura implementazione della navigazione utilizzando la realtà virtuale.

Andando nel dettaglio dell'implementazione, per prima cosa è stato sviluppato il **Controller**, successivamente un **Character Controller** per la navigazione in prima persona, e un **Pawn** per la navigazione dall'alto.

Un Controller non è altro che un intermediario tra l'input dell'utente e l'agente che esegue l'azione di navigazione. É il responsabile del *possesso* di uno specifico Pawn o Character, che sono gli enti effettivi che si muovono all'interno dell'ambiente, e si occupa di trasmettere a loro gli input. Per possesso si intende la capacità del controller di trasferire gli input dell'utente ad un'entità specifica, in modo tale che l'entità possa eseguire azioni in base a questi input.

Esso mette a disposizione 2 eventi, consultabili in Fig. 5.4:

- **PossessFirstPerson**: setta come character di default il **FirstPersonCharacter**, quindi abilita la navigazione in prima persona;

- **PossessTopDown**: setta di default il **TopDownCamera**, ovvero il pawn per la navigazione dall'alto.

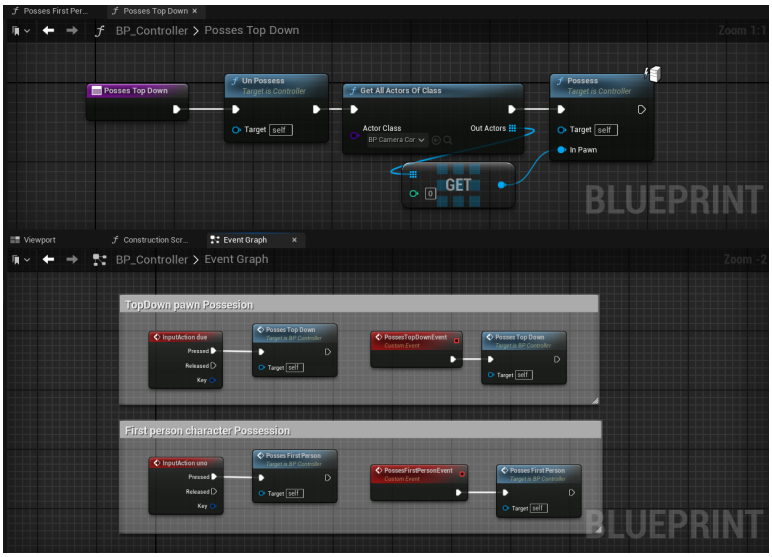


Figura 5.4: Blueprint del Controller

Questi eventi sono stati collegati alla pressione dei tasti della tastiera **1** e **2**, in modo tale da passare velocemente dall'uno all'altro. Inoltre sono stati predisposti due eventi generici che svolgono lo stesso compito in modo da essere chiamati da qualsiasi altro componente, ad esempio dalla UI.

FirstPersonCharacter

Questo componente appartiene ai characters precostruiti di Unreal, quindi non è stato sviluppato da noi, è solo stato integrato all'interno del progetto ed è stato impostato un diverso input mapping. I tasti **WASD** permettono di muoversi come se fossero le frecce direzionali, e con il movimento del mouse si può spostare la visuale (Fig. 5.5).

TopDownCamera

Questo componente invece è stato sviluppato da zero, interamente utilizzando i Blueprint. Attraverso questa pawn è possibile navigare la scena utilizzando solamente il mouse (Fig. 5.6):

- prendendo e trascinando con il tasto sinistro del mouse, ci si sposta nella scena;

Figura 5.5: Navigazione in prima persona

- premendo e trascinando con il tasto destro del mouse, si può ruotare la visuale attorno all'asse Z e X;
- con la rotella del mouse si può fare zoom in e zoom out.

Figura 5.6: Navigazione in top down

5.3.3 Implementazione della Ciclo Giorno/Notte

L'implementazione di un ciclo giorno/notte in un Digital Twin è importante per diversi motivi che vanno oltre l'aspetto estetico. Innanzitutto,

il realismo e l'immersione è uno dei primi requisiti di questo progetto, quindi avere un ciclo giorno/notte è molto importante.

Invece in ottica futura, avere un ciclo giorno/notte che simula accuratamente la luce solare e le condizioni di illuminazione permette di valutare l'impatto dell'illuminazione naturale sugli spazi urbani, ma soprattutto può essere utilizzato per condurre studi e simulazioni energetiche, come ad esempio l'analisi dell'efficienza di pannelli solari in diverse ore del giorno e stagioni. Quest'ultimo punto è anche uno tra i primi obiettivi del progetto *Bologna Digital Twin*. Quindi avere un ciclo giorno/notte simulato correttamente è fondamentale.

Il ciclo giorno e notte è stato implementato interamente in Blueprint partendo dall'attore già presente in Unreal Engine **Sun Position Calculator**. Questo attore permette di calcolare correttamente la posizione del sole all'interno della scena in base alle seguenti informazioni di input:

- **Latitudine e Longitudine**: coordinate geografiche del luogo;
- **Time Zone**: fuso orario del luogo;
- **North Offset**: offset in gradi rispetto al nord.
- **Ora, Giorno, mese e anno**: data corrente.

Così come si presenta questo attore non è dinamico, ovvero fornita una data posiziona staticamente il sole (Fig. 5.7). Per renderlo dinamico si è dovuto modificare abbastanza pesantemente il Blueprint di questo attore, consultabile in Fig. 5.9. In particolare è stato aggiunto un *Timeline*. Una timeline essenzialmente è una sequenza di valori o *keyframes* che definiscono come una specifica proprietà o parametro deve cambiare nel tempo. In questo caso la timeline è stata utilizzata per aggiornare costantemente la data e l'ora, in modo tale da far cambiare la posizione del sole in base all'ora corrente.

Come si può vedere dalla Fig. 5.8, sono presenti due timeline, la prima si occupa di aggiornare l'ora e varia da 0 a 24, mentre la seconda è una Event Timeline, e scatena l'evento *DayEnd* quando l'ora raggiunge 24. Questo evento è stato utilizzato per far avanzare di un giorno la data.

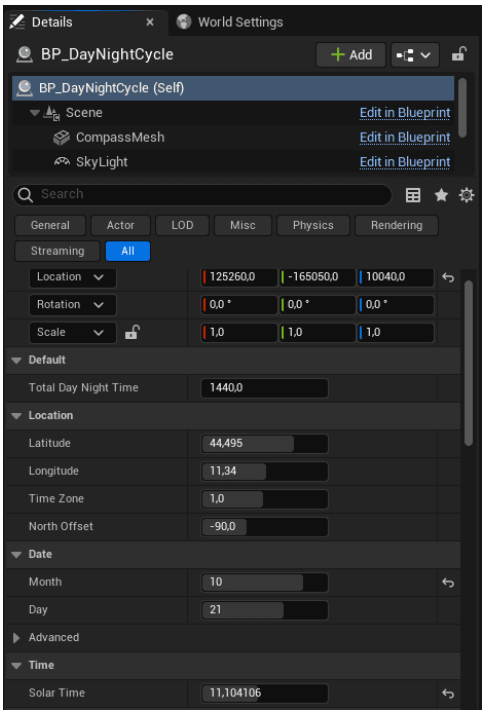


Figura 5.7: Parametri per il calcolo della posizione del sole

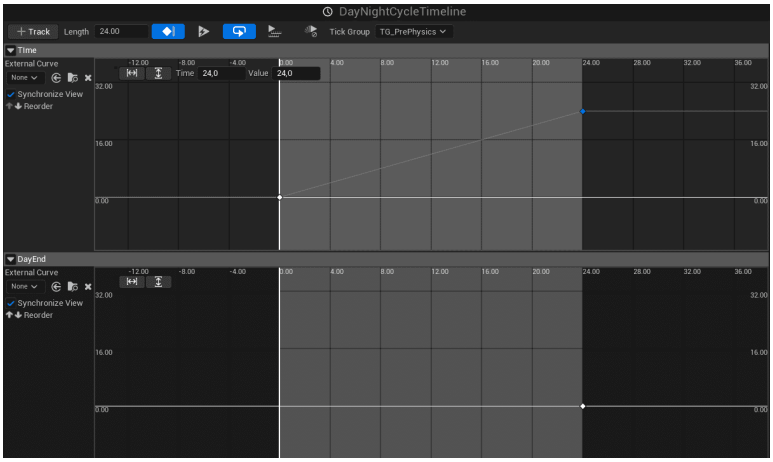


Figura 5.8: Timeline per il calcolo della posizione del sole

Inoltre attraverso una variabile *TotalDayNightTime* è stata data la possibilità di regolare la velocità del ciclo giorno/notte.

Andando a parlare più dal punto di vista grafico, questo attore inizialmente conteneva solamente la *DirectionalLight*, ovvero la luce sola-

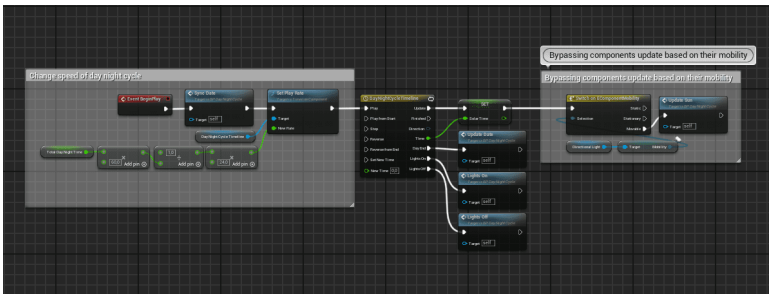


Figura 5.9: Blueprint di gestione del ciclo giorno/notte

re. Per rendere l'ambiente più realistico, sono stati aggiunti i seguenti elementi:

- **SkyLight**: luce che simula la luce diffusa proveniente dal cielo. Eśtata impostata in modo tale da avere un colore bluastro durante il giorno.
- **SkyAtmosphere**: componente che simula l'atmosfera, e permette di ottenere effetti di scattering e di colorazione del cielo.
- **ExponentialHeightFog**: componente che simula la nebbia in lontananza.
- **VolumetricCloud**: componente che simula le nuvole.
- **DirectionalLight**: un'altra luce posizionata a 180 gradi rispetto alla luce solare, in modo tale da ottenere la luce della luna durante la notte.

In Fig. 5.10 sono presenti 4 immagini che mostrano il ciclo giorno/-notte all'interno del Digital Twin.

5.4 Collegamento con dati esterni

Una volta che il Digital Twin era pronto e navigabile è iniziato lo sviluppo di tutti quegli attori che rendono questo applicativo un vero e proprio Digital Twin. Ovvero gli elementi che fanno richiesta di informazioni all'esterno e dinamicamente popolano l'ambiente 3D in modo da poter essere sempre aggiornato con l'esterno.

(a) Alba (b) Ore 12:00

(c) Tramonto (d) Ore 21:00

Figura 5.10: Ciclo giorno notte

Come specificato anche in precedenza, dato che siamo entrati in questo progetto nella fase iniziale di prototipazione, ancora non è presente alcuna infrastruttura cloud che fa da unico entry point per tutti i dati e che mette a disposizione servizi per effettuare simulazioni di vario genere, Cineca ci ha indirizzati verso una struttura più generale possibile in modo che poi in futuro possa essere facilmente integrato nella infrastruttura definitiva. Quindi non avendo nemmeno flussi di dati real time che provengono dall'esterno ci siamo basati, per sviluppare questo proof of concept, oltre che alla RestAPI sviluppata nel capitolo precedente, sugli OpenData del Comune di Bologna i quali sono comunque sempre aggiornati.

Quindi riassumento, le informazioni che vengono prese dinamicamente dall'esterno per aggiornare il DT sono:

- dati sugli alberi, presenti sul nostro database;
- dati sulle precipitazioni atmosferiche, da Bologna OpenData;
- dati sul traffico in determinati punti della città.

Sviluppo di un componente riusabile per fare richieste HTTP

Dato che questi attori si interfacciano all'esterno effettuando HTTP request, e che Unreal adotta il principio di *composition over inheritance*, è stato sviluppato un componente in C++ che permette di incapsulare richieste HTTP. La classe **HttpComponent** permette quindi di essere aggiunto a qualsiasi attore che necessita di recuperare dati esterni, riducendo la duplicazione di codice, rendendo gli attori indipendenti dai moduli di rete di Unreal quindi si devono occupare solo di lanciare la richiesta tramite il componente e gestire la logica una volta ottenuta una risposta.

HttpComponent agisce quindi come un ponte tra l'attore e il web: invia richieste a un URL specificato e gestisce la risposta. Quando una risposta viene ricevuta, il componente notifica l'attore della ricezione della risposta. Per fare ciò si utilizzano i *delegati*, ovvero dei puntatori a funzione. Questo delegato permette ad altri attori di "ascoltare" quando il componente ha ricevuto una risposta. Se qualcuno è interessato, si può registrare presso il componente e venire notificato non appena arriverà la risposta.

Il Componente HttpComponet definisce il delegato **OnHttpResponseReceived**, presso i quali gli attori si registreranno passando in ingresso una funzione di callback, e implementa due metodi:

- **GetRequest(URL)**: funzione che permette di effettuare una richiesta HTTP specificando un URL;
- **OnResponseReceived()**: funzione di callback che viene chiamata da GetRequest() una volta che sarà arrivata la risposta, nel quale viene "spacchettata" e attraverso i delegati vengono notificati tutti gli attori interessati a quella HttpRequest passandogli in ingresso la risposta.

5.4.1 Alberi

Il primo, e anche più corposo, elemento di collegamento con i dati esterni è l'attore che permette di fare retrieving dei dati dal database degli alberi costruito nel capitolo precedente.

L'obiettivo principale di questo attore è di popolare dinamicamente la scena della città con le mesh degli alberi discusse in precedenza, in modo che la loro posizione e dimensione rispecchi quella originale. Inoltre, una volta popolata la mappa permette di ottenere informazioni sui singoli alberi senza dover effettuare nuovamente la chiamata HTTP.

L'attore **TreeSpawner** è stato implementato in parte in C++ e in parte in Blueprint. In C++ è stata implementata la parte più critica, ovvero quella che si occupa di lanciare la richiesta HTTP e gestire il risultato. Mentre in Blueprint è stata sviluppata la parte più di integrazione con il resto del sistema.

Quindi, TreeSpawner deve gestire tutta la logica di instancing di nuove mesh. Generalmente per collocare nel mondo nuove mesh si utilizza il metodo di Unreal *SpawnActor()*, il quale alloca nuovo spazio in memoria per ogni mesh instanziata. Ciò non è l'ideale vista la quantità di alberi da posizionare. Per questo motivo si utilizza il componente **InstancedStaticMeshComponent**, il quale permette di instanziare molteplici copie di una mesh static con un impatto molto ridotto sulle prestazioni, poichè tutte le istanze condividono la stessa mesh e materiali, riducendo così la memoria utilizzata e il carico di elaborazione sulla GPU. Inoltre, la renderizzazione di mesh istanziate è fortemente ottimizzata a livello di game engine da Unreal, nel quale le istanze vengono batchate e renderizzate insieme dove possibile.

L'attore TreeSpawner ha un numero di *InstancedStaticMeshComponent* pari al numero di mesh diverse di alberi (Fig. 5.11).

Analizzando prima l'implementazione C++, di seguito verranno descritti i metodi principali:

- **Costruttore**: crea ed aggiunge il componente *HttpComponent* all'attore, in modo tale da poter effettuare richieste HTTP;

- **BeginPlay()**: metodo chiamato appena l'attore viene instanziato, inizializza e popola tutti i *InstancedStaticMeshComponent* e inoltre si registra al delegato *OnHttpResponseReceived* del componente HttpComponent, in modo da venire notificato non appena arriverà la risposta;

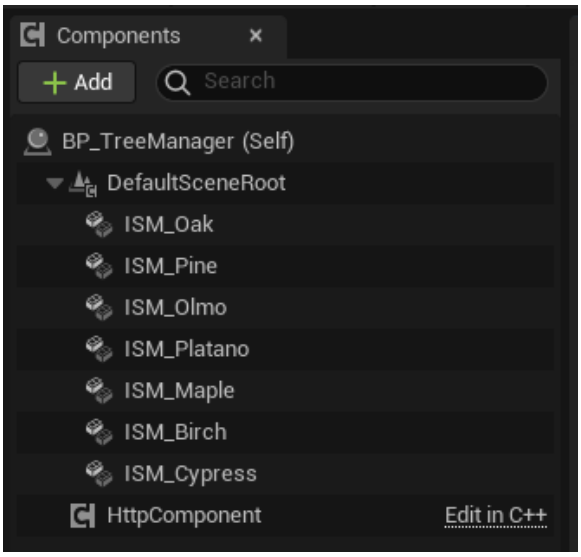


Figura 5.11: Gerarchia di componenti dell'attore TreeSpawner

- **FetchTrees()**: metodo che effettua la richiesta HTTP al database degli alberi, utilizzando il HttpComponent tramite il metodo *GetRequest()*;

- **HandleHttpResponse()**: metodo di callback che viene chiamato non appena arriva la risposta, in questo metodo vengono "spacchettati" i dati e viene popolato un array di strutture dati che contengono tutte le informazioni sugli alberi, successivamente chiama il metodo che si occupa di popolare la scena con le mesh degli alberi;

- **SpawnTrees()**: metodo che si occupa di popolare la scena con le mesh degli alberi, utilizzando i dati ottenuti dalla risposta;

- **CalculateCorrectPosition()**: metodo che effettua la conversione da coordinate geografiche a coordinate locali di Unreal Engine;

- **AddInstance()**: metodo che aggiunge una nuova istanza di mesh all'interno di un *InstancedStaticMeshComponent*.

Lato Blueprint invece, all'evento *Event BeginPlay* (Fig. 5.12) è stato collegato il metodo C++ *FetchTrees()* dopo un delay di 1 secondo,

in modo da dare il tempo all'attore di inizializzarsi completamente e registrarsi al delegato del componente HttpComponent.

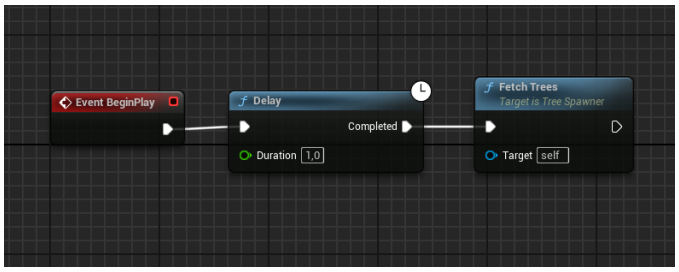


Figura 5.12: Evento BeginPlay dell'attore TreeSpawner

Quindi, non appena si fa partire il Digital Twin, l'attore TreeSpawner effettua una richiesta HTTP al database degli alberi, e non appena arriva la risposta popola la scena con le mesh degli alberi, ottenendo il risultato mostrato in Fig. 5.13.

Figura 5.13: Alberi posizionati in scena tramite TreeSpawner, zona Giardini Margherita

Infine, l'ultima feature implementata di questo attore è la possibilità di ottenere informazioni su un albero specifico semplicemente cliccandoci sopra. Questa funzionalità è stata implementata direttamente utilizzando i Blueprint in quanto sono ottimali per gestire eventi di input da mouse e tastiera.

Per poter visualizzare a schermo le informazioni degli alberi (Fig. 5.15) è stato sviluppato un **Widget**, ovvero un componente riutilizzabile della UI, contenente tutte le informazioni da mostrare.

Nell'attore *TopDownCamera* è stato aggiunto un evento di click con il tasto centrale del mouse. Questo evento, quando scatenato, lancia un raycast dalla posizione del mouse verso la scena, e se colpisce un albero allora notifica l'attore TreeSpawner, il quale si occupa di rendere visibile il widget popolato con le informazioni dell'albero selezionato. In Fig. 5.14 è presente il Blueprint dell'evento di click.

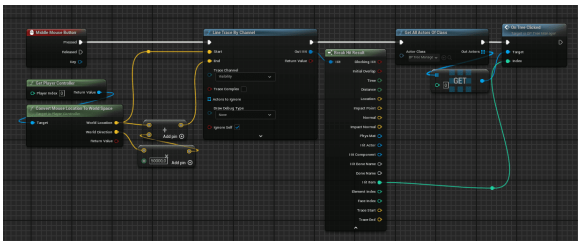


Figura 5.14: Evento di click con il tasto destro del mouse dell'attore TopDownCamera

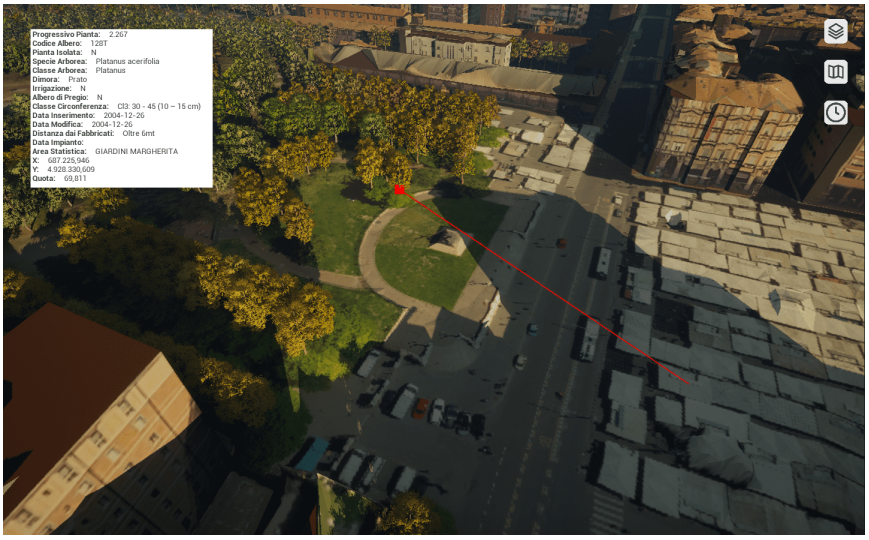


Figura 5.15: Ottenimento albero selezionato e popolazione del widget

5.4.2 Precipitazioni

L'altro elemento che è stato implementato per adattare dinamicamente il Digital Twin è l'attore che si occupa di recuperare i dati sulle precipitazioni. Questo attore al momento permette solo di visualizzare le precipitazioni attuali, ma in futuro potrebbe essere esteso per visualizzare anche le previsioni meteo, uno storico delle precipitazioni, e potrebbe essere utilizzato per effettuare simulazioni di qualsiasi genere.

I dati sulle precipitazioni sono stati presi da Bologna OpenData, sono aggiornati ogni 30 minuti e ritornano un valore numerico che rappresenta l'intensità delle precipitazioni in mm/h. Bologna OpenData mette a disposizione un endpoint attraverso il quale, facendo richieste HTTP, è possibile effettuare query per ottenere informazioni personalizzate sulle precipitazioni.

Prima di parlare dell'implementazione di questo attore, è necessario spiegare come si è deciso di visualizzare le precipitazioni. La pioggia è stata implementata utilizzando **Niagara**, ovvero il nuovo sistema di particelle di Unreal Engine.

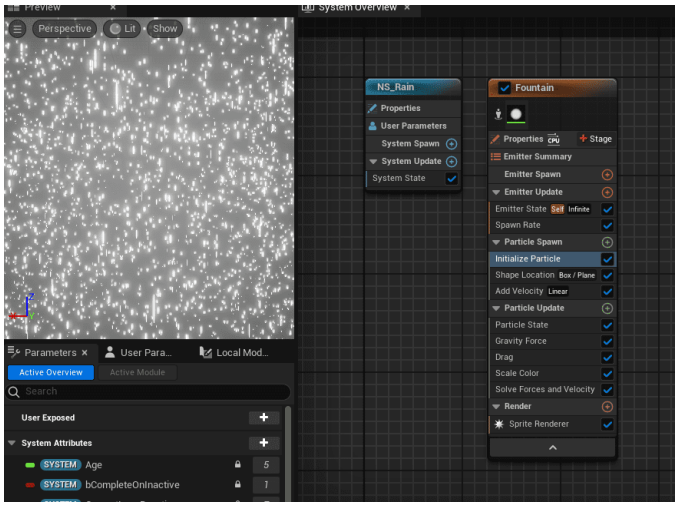


Figura 5.16: Sistema particellare per la pioggia

Nel quale è stata impostata la dimensione di ogni singola particella, la velocità di caduta, la quantità di particelle, e il rate di emissione (Fig. 5.16). L'area di spawn delle particelle è limitata ad un'area molto piccola. Questo perchè avere un'area grande come tutta la città com-

porterebbe un impatto sulle prestazioni molto elevato. Quindi questo problema è stato aggirato mettendo il particle system (con area di spawn molto piccola) come figlio dell'attore *TopDownCamera*, in modo tale che il sistema di particelle segua la camera e quindi sembri che la pioggia cada su tutta la città.

Lo sviluppo dell'attore che si occupa di ottenere dati sulla pioggio, chiamato **RainFetcher**, è analogo a quanto fatto per l'attore *TreeSpawner*, quindi in C++ e Blueprint. Ovviamente questo attore utilizza **HttpComponent** per effettuare richieste HTTP, e attraverso la funzione di callback processa i dati ottenuti e li utilizza per visualizzare il sistema particellare della pioggia (Fig. 5.17).

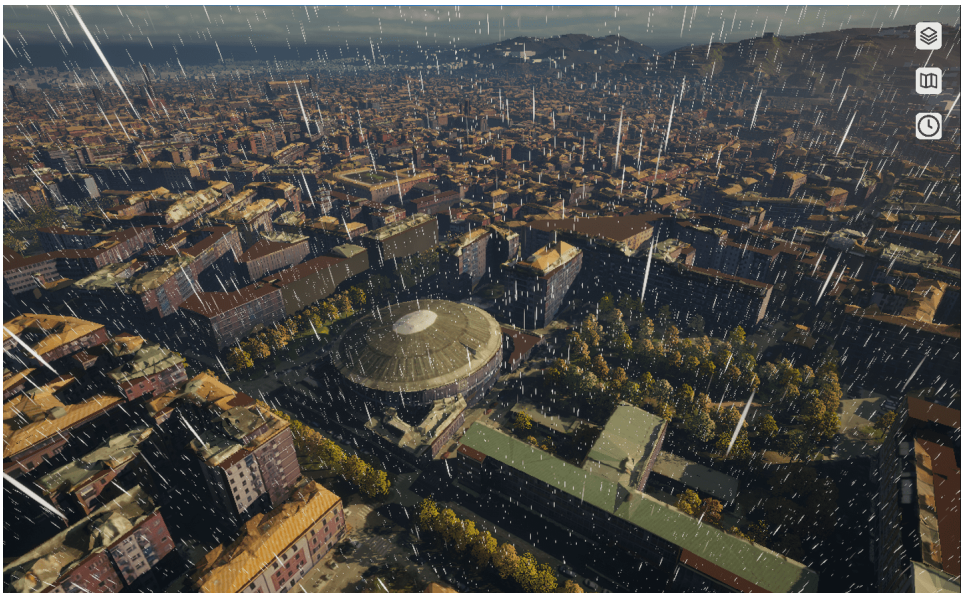


Figura 5.17: Visualizzazione della pioggia

5.4.3 Traffico

L'ultimo attore che è stato implementato per adattare dinamicamente il Digital Twin è l'attore che si occupa di recuperare i dati sul traffico. Come per l'attore *RainFetcher*, anche questo attore attualmente permette solo di visualizzare lo stato attuale del traffico, ma in futu-

ro potrebbe essere esteso per effettuare previsioni sul traffico e fare simulazioni.

Anche i dati sul traffico sono presi da Bologna OpenData, e sono aggiornati ogni 15 minuti. E contengono il numero di veicoli, suddivisi per categoria, che hanno transitato nell'arco di 15 minuti. Solo per scopo dimostrativo verranno visualizzati i passaggi totali nell'arco dell'intera giornata. A differenza dei dati sulle precipitazioni, i quali erano all'interno di un unico endpoint, i dati sul traffico hanno endpoint differenti per ogni località della città. In tutta la città sono presenti 79 località di rilevamento del traffico. Quindi sono presenti 79 url differenti a cui fare richieste per ottenere le informazioni sul traffico.

Proprio come per l'attore *TreeSpawner* e *RainFetcher*, anche l'attore *TrafficFetcher* è stato implementato in C++ e Blueprint, e utilizza **HttpComponent** per effettuare richieste HTTP. L'unica differenza rispetto agli attori precedenti è che l'url a cui fare richiesta non è più embeddato all'interno del codice, ma è stato aggiunto un campo all'interno dell'attore accessibile direttamente dall'editor di Unreal Engine, in modo tale da poter instanziare più volte lo stesso attore settando ad ogni istanza un url differente.

Ogni attore *TrafficFetcher* è stato posizionato in corrispondenza di una delle 79 località di rilevamento del traffico, ed è possibile ottenere le informazioni sul traffico semplicemente cliccandoci sopra, proprio come per l'attore *TreeSpawner*, utilizzando la tecnica di raycasting.

In Fig. 5.18 e Fig. 5.19 sono presenti due immagini che mostrano i dati sul traffico ottenuti dagli attori *TrafficFetcher* posizionati in corrispondenza di Via Rizzoli e San Felice.

5.5 Profiling prestazioni

Per garantire che il Digital Twin della città di Bologna operi con la massima efficienza, è cruciale analizzare le prestazioni del sistema effettuando *Profiling* e *Benchmarking* con l'obiettivo di indentificare i punti critici che influenzano negativamente le prestazioni.

L'utilizzo del Profile di Unreal Engine (Fig. 5.20) si possono raccogliere informazioni prezione sul framerate, sulle risorse di CPU e GPU,

Figura 5.18: Dati sul traffico: Via Rizzoli

Figura 5.19: Dati sul traffico: San Felice

l'uso della memoria e le draw calls. Inoltre attraverso il profiling è possibilie identificare in tempo reale l'impatto di ogni componente all'interno dell'applicativo, dalle mesh agli script, dall'illuminazione alle texture.

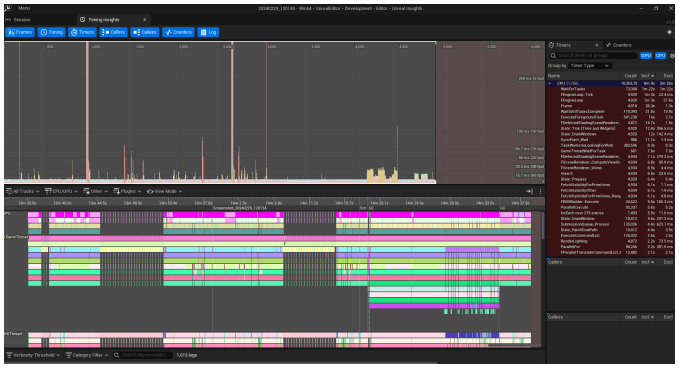


Figura 5.20: Profiler di Unreal Engine

Il Profile mostra informazioni riguardanti ogni singolo frame, e permette di visualizzare il tempo di esecuzione dello stesso. Queste informazioni sono visibili in due grafici differenti, nel grafo sopra si possono vedere una serie di rettagolini che rappresentano il tempo di esecuzione di ogni singolo frame, più il rettangolo è alto più il frame è stato eseguito in un tempo lungo. Nel grafico sottostante invece si può vedere un *flame graph*, ovvvero un grafico a barre orizzontali che rappresenta il tempo di esecuzione di ogni singola funzione. Il flame graph mostra una pila di chiamate sotto forma di blocchi, dove ogni blocco rappresenta una funzione, e la lunghezza del blocco rappresenta il tempo di esecuzione della funzione. Inoltre è possibile vedere tutte le chiamate di funzione nelle barre sottostanti.

Di seguito verranno effettuate una serie di misurazioni in condizioni di esecuzioni differenti. Nel dettaglio:

- **Navigazione nella scena e layer di visualizzazione**: si vuole misurare l'impatto sulle prestazioni della navigazione nella scena e del cambio di modalità di visualizzazione;
- **Numero di alberi**: si vuole misurare l'impatto sulle prestazioni in base al numero di alberi presenti in scena;

5.5.1 Profiling Navigazione e Layer di Visualizzazione

In questa sezione verrà analizzato l'impatto sulle prestazioni della navigazione nella scena, sia in prima persona che dall'alto, e del cambio di layer di visualizzazione. Per quest'ultima misurazione si è deciso di analizzare i metodi della GameMode:

- **ToggleActorsVisibility()**: per capire se ha impatto sulle performance rendere visibile/invisibile molte mesh contemporaneamente;
- **SwapAllMaterials()**: per capire che impatto sulle performance ha cambiare il materiale di tutte le mesh a runtime.

Per identificare l'inizio e la fine delle funzioni sono stati settati da codice C++ dei *Bookmark*, in modo tale da poterli visualizzare nel profiler.

All'interno di questa simulazione sono state effettuate le seguenti azioni, in ordine: navigazione top down (con zoom anche molto veloci), navigazione in prima persona, ritorno in navigazione top down, hide/unhide di tutte le mesh, cambio di materiale di tutte le mesh.

Il risultato del primo profiling è visibile in Fig. 5.21.

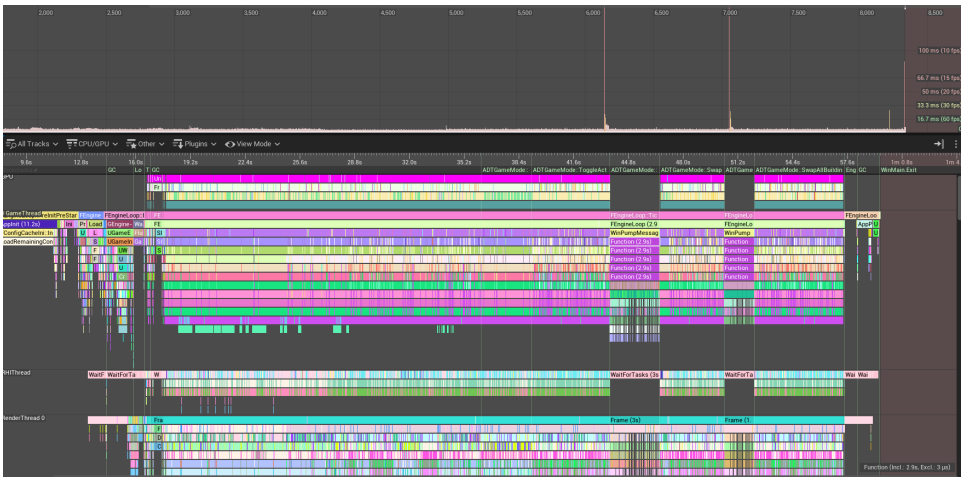


Figura 5.21: Profiling navigazione e layer di visualizzazione

Come si può vedere dal grafico, stando all'ordine di esecuzione delle azioni sopracitate, si può notare che durante la navigazione sia topdown che in prima persona non vi è alcun cambio di performance.

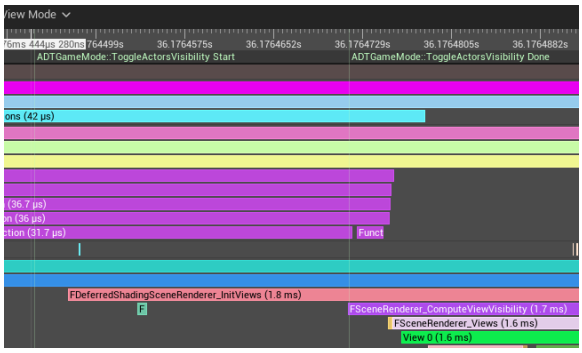


Figura 5.22: Profiling ToggleVisibility

Andando avanti nella timeline si nota (Fig. 5.22) che l'operazione di hide/unhide ha impatto completamente nullo sulle performance in quanto tutta l'operazione dura meno di 1ms.

Il problema maggiore di performance si presenta quando si cambia il materiale di tutte le mesh. Questo è ben visibile (Fig. 5.23) sia dal grafico dei frame nel quale sono presenti 2 picchi molto alti, sia dal flame graph, nel quale si può notare che il rendering in GPU viene stoppato per ben 3 secondi lasciando alla CPU il compito di cambiare i materiali di tutte le mesh.

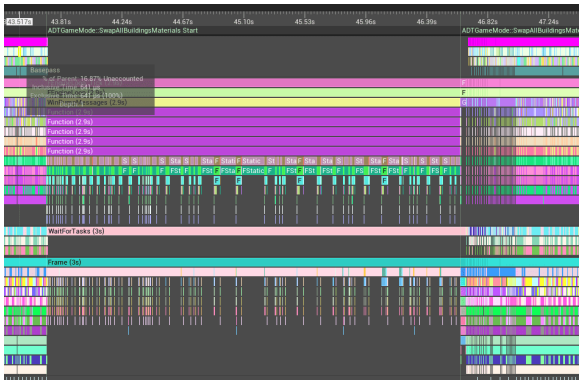


Figura 5.23: Profiling SwapAllMaterials

Andando più nel dettaglio, questo collo di bottiglia è generato dalla funzione *SetMaterial*, sia quando si passa da materiale flat a materiale con texture. Il cambio di materiale è un'operazione molto onerosa per diverse ragioni:

- **Caricamento delle risorse**: i materiali, dato che contengono risorse come shader, texture, ecc, c'è la necessità di caricare su GPU tutte queste risorse; e ciò è molto dispendioso in quanto le texture degli atlas degli edifici sono molto pesanti;
- **Compilazione degli shader**: ogni volta che si applica un materiale, il motore di rendering deve ricompilare gli shader di ogni materiale, e questa è una operazione onerosa.

Quindi, ora avendo individuato il problema è possibile pensare a delle soluzioni per risolverlo. La soluzione più immediata è quella di cambiare la logica con cui viene effettuato il cambio di visualizzazione, se il collo di bottiglia è proprio il cambio di materiale, allora si potrebbe pensare di utilizzare i *MaterialInstanceDynamic*, ovvero delle istanze di materiali che permettono di cambiare le proprietà del materiale a runtime senza dover ricompilare. A tale scopo quindi è stato creato un nuovo materiale base, uguale per tutti, parametrizzabile, nel senso che possono essere passati a runtime dei valori per modificare la visualizzazione. Quindi si crea un'istanza di questo materiale per ogni mesh, e quando si vuole cambiare la visualizzazione si cambiano i valori dei parametri dell'istanza (Fig. 5.24).

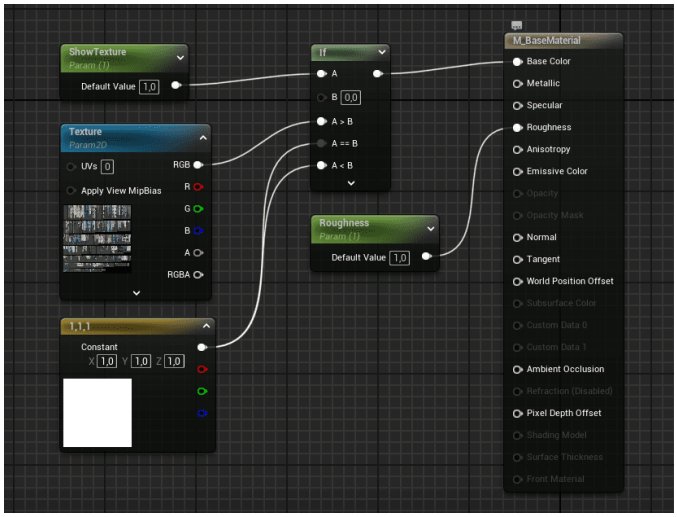


Figura 5.24: Nuovo materiale parametrizzabile

Con questo nuovo sistema di materiali si può effettuare nuovamete il profiling per vedere se il problema è stato risolto (Fig. 5.25).

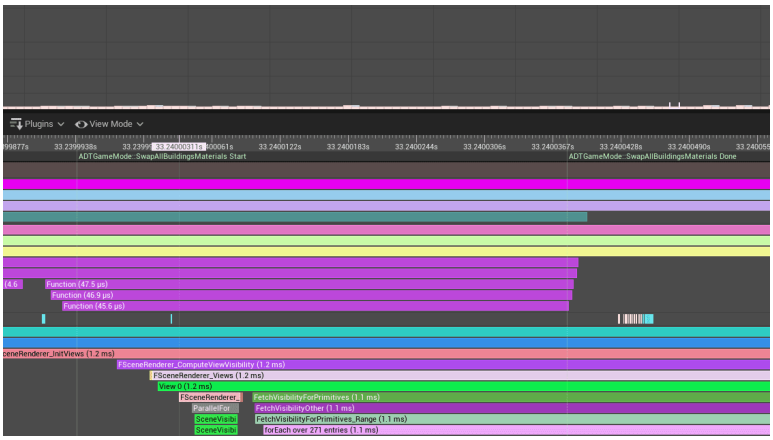


Figura 5.25: Profiling con nuovo sistema di materiali

Come si può vedere dal grafico, il problema è stato completamente risolto, ottenendo un aumento di performance notevole, e un'operazione che prima durava 3 secondi ora dura meno di 1ms.

5.5.2 Profiling Alberi

Un altro elemento cruciale da analizzare con il profiler è analizzare che impatto hanno sulle prestazioni gli alberi. Facendo anche test con numeri di alberi molto superiori a quelli attuali, in modo tale da capire se il sistema di istanziazione delle mesh è efficiente e scalabile.

Quindi, saranno effettuati test con 3 diversi numeri di alberi: il numero effettivo quindi circa 5000, poi si proverà con 50000 e infine con 100000.

Come si può vedere dal grafico in Fig. 5.26, l'operazione di istanziazione delle mesh, nonstante il numero di alberi, è molto efficiente ed ha un impatto minimo sulle performance, impiegando circa 80ms per istanziare 5000 alberi.

Analizzando invece le due situazioni successive (Fig. 5.27) si può intuire un comportamento lineare, ovviamente più mesh vengono istanziate più tempo impiega il motore di rendering ad allocarle. Nel caso delle 50k mesh impiega circa 600ms, mentre nel caso delle 200k mesh impiega circa 4s. La cosa più interessante da notare però è il grafo del framerate, in entrambi i casi dopo aver istanziato le mesh il framera-

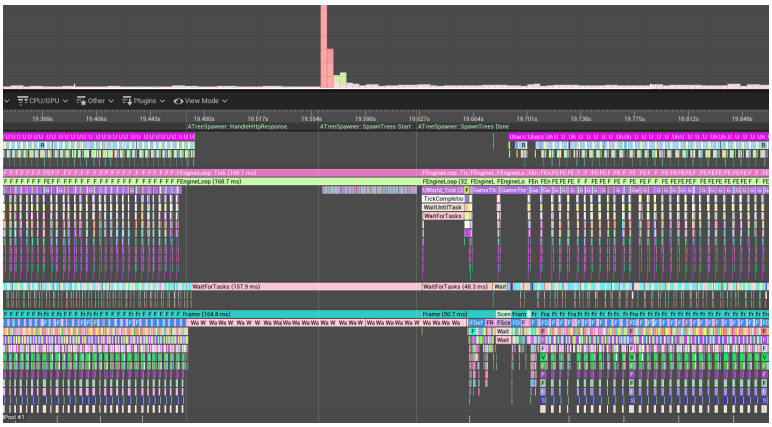
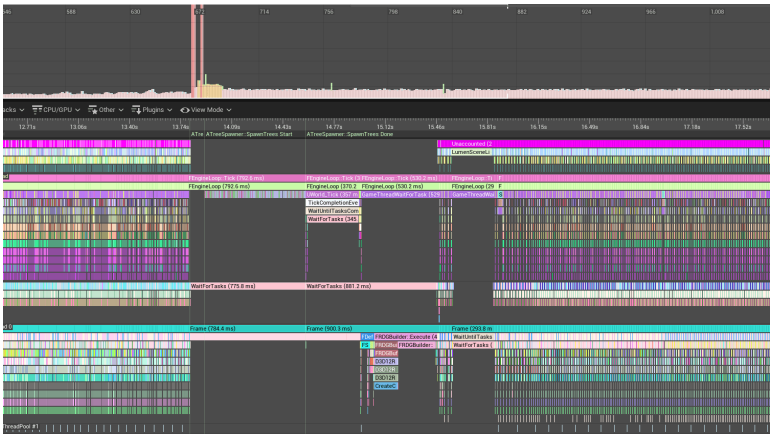
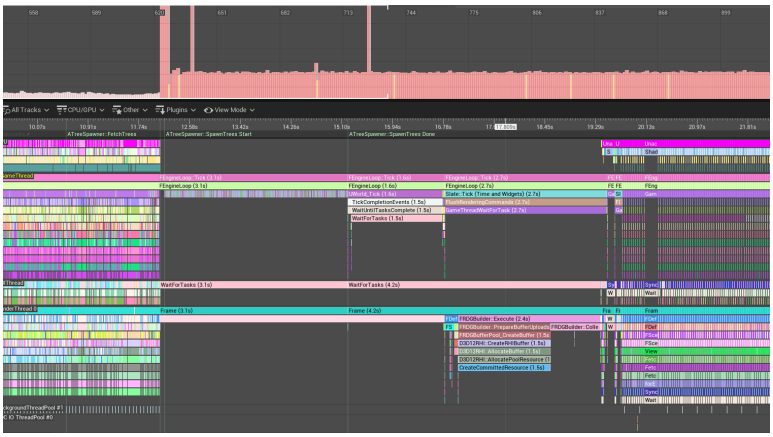


Figura 5.26: Profiling 5k alberi

te diminuisce in quanto il motore di rendering deve renderizzare più elementi nonostante esse siano InstancedStaticMesh.



(a) 50k Alberi



(b) 200k Alberi

Figura 5.27: Profiling con numero di alberi superiori al normale

Appendice A

Codice addizionale

Elaborazione di dati grezzi

```
def decimal_to_utm(self, lat, lon):
    utm_coords = utm.from.latlon(lat, lon)
    return [utm_coords[0], utm_coords[1]
```

Listing A.1: Conversione di coordinate

```
df = pd.read_csv('filtered_buildings_extra.csv', delimiter=';')
las_df = pd.read_csv('LASdataframe.csv', delimiter=';')

tiles_list = []
las_data = list(split_columns(las_df))

for s in shapes:
    shape_set = set()
    polyg = Polygon(s)

    for fileName, lat, lon in las_data:
        if len(shape_set) == 4:
            break
        min_x = lat
        min_y = lon
        max_x = min_x + 500
        max_y = min_y + 500

        line1 = LineString([(min_x, min_y), (min_x, max_y)])
        line2 = LineString([(min_x, max_y), (max_x, max_y)])
```

```
        line3 = LineString([(max_x, max_y), (max_x, min_y)])
        line4 = LineString([(max_x, min_y), (min_x, min_y)])

        if line1.intersects(polyg) or line2.intersects(polyg)
    or line3.intersects(polyg) or line4.intersects(polyg):
            shape_set.add(fileName)

    for x, y in s:
        for fileName, lat, lon in las_data:
            if fileName in shape_set:
                continue
            elif len(shape_set) == 4:
                break

            min_x = lat
            min_y = lon
            max_x = min_x + 500
            max_y = min_y + 500

            if min_x <= x <= max_x and min_y <= y <= max_y:
                shape_set.add(fileName)

        if len(shape_set) == 4:
                break

    tiles_list.append(list(shape_set))


df['Tiles'] = tiles_list
```

Listing A.2: Associazione di ogni edificio ai tile di appartenenza

```
for fileName, lat, lon in zip(las_df['FileName'], las_df['Lat'
    ], las_df['Lon']):
    filter = df['Tiles'].str.contains(fileName, case=False)
    df_f = df[filter]

    if df_f.empty:
        ## tile doesn't contain buildings, skip
        continue

    shapeName = fileName.split(".")[0]
    dirPath = output_folder + "/" + shapeName
```

```
    os.makedirs(dirPath, exist_ok=True)

    polys = [Polygon(shape) for shape in df_f['Geo Shape UTM']]
    df_f = df_f.assign(GEOM=polys)

    gdf = gpd.GeoDataFrame({'geometry': df_f['GEOM']})
    gdf['Q_PIEDE'] = df_f['QUOTA_PIEDE']
    gdf['ALT_GRONDA'] = df_f['ALTEZZA_GRONDA']
    gdf['COD_OGG'] = df_f['CODICE_OGGETTO']
    gdf.crs = "EPSG:32632"

    try:
        gdf.to_file(dirPath + "/" + shapeName + ".shp")
    except UserWarning as e:
        print(f"Si e verificato un errore durante la scrittura
    del file shapefile: {shapeName}")
```

Listing A.3: Costruzione degli shapefiles

```
def position_camera(self, mesh, face, face_normal, face_center,
     cam_distance):
    co = self.camera_object
    co.location.x = face_center.x + face_normal.x *
    cam_distance
    co.location.y = face_center.y + face_normal.y *
    cam_distance
    co.location.z = face_center.z
    co.rotation_mode = 'XYZ'
    co.rotation_euler.x = pi / 2
    co.rotation_euler.y = 0
    co.rotation_euler.z = atan2(face_normal.y, face_normal.x) +
     pi/2
    fw, fh = self.calculate_max_dimension(mesh, face,
    face_center, face_normal)
    co.data.ortho_scale = max(fw, fh)
```

Listing A.4: Posizionamento della Camera

```
def generate_atlas(self, atlas_path):
    # Ordina le texture per altezza
    self.textures.sort(key=lambda tex: tex.h, reverse=True)
```

```
4
5       m = self.atlas_margin
6       y = m
7       i = 0
8       ni = len(self.textures)
9       row = 0
10
11      while i<ni:
12          w = self.textures[i].w + m
13          h = self.textures[i].h + m
14          j=i+1
15          while j<ni and ((w + self.textures[j].w + 2*m ) < self.
    atlas_width):
16              w += self.textures[j].w + m
17              j += 1
18          x = m
19          for k in range(i,j):
20              t = self.textures[k]
21              t.x0 = x
22              t.y0 = y
23              t.x1 = x + t.w
24              t.y1 = y + t.h
25              x += t.w
26              x += m
27          y = y+h
28          i=j
29          row +=1
30
31      self.atlas_height = y
32
33      atlas = Image.new('RGB', (self.atlas_width, self.
    atlas_height))
34      atlas_draw = ImageDraw.Draw(atlas)
35      atlas_draw.rectangle( [(0,0),(m,m)], fill=(170,113,94) ) #
    roof color
36
37      for tex in self.textures:
38          # Apri l'immagine della texture
39          tex.open()
40          # Posiziona la texture nell'atlas
41          atlas.paste(tex.image, (tex.x0, tex.y0))
42          # Chiude l'immagine della texture per liberare risorse
43          tex.close()
44
45      # Salva l'immagine atlas finale
```

```
46      atlas.save(atlas_path)
```

Listing A.5: Costruzione dell'atlas

```
1  def set_uv_map(self):
2      for tex in self.textures:
3          face = tex.face
4          mesh = tex.mesh
5          face.material_index = 0
6          n = face.normal
7          c = face.center
8
9          for vert_idx, loop_idx in zip(face.vertices, face.
       loop_indices):
10             uv_coords = mesh.uv_layers.active.data[loop_idx].uv
11             v = mesh.vertices[vert_idx].co
12             dz = v.z-c.z
13             if dz > 0:
14                 uv_coords.y = 1-float(tex.y0)/self.atlas_height
15             else:
16                 uv_coords.y = 1-float(tex.y1)/self.atlas_height
17
18             u = Vector((0,0,1)) # up vector
19             s = n.cross(u)
20             cv = Vector(( v.x-c.x, v.y-c.y, v.z-c.z ))
21             if cv.dot( s ) < 0:
22                 uv_coords.x = float(tex.x1)/self.atlas_width
23             else:
24                 uv_coords.x = float(tex.x0)/self.atlas_width
```

Listing A.6: UV mapping

Conclusioni

Il progetto di tesi Bologna Digital Twin, ha mirato alla creazione di un Digital Twin della città di Bologna, dalla fase iniziale di elaborazione dei dati grezzi fino alla realizzazione di un ambiente 3D interattivo e dinamico. L'indagine preliminare ha evidenziato l'importanza fondamentale dei dati, i quali devono essere accurati, completi e aggiornati. Proprio grazie questa approfondita analisi dei dati di partenza è stato possibile, anche con l'ausilio di strumenti e tecniche avanzate della Computer Graphics, la costruzione di un Digital Twin della città di Bologna. Il quale non solo è una replica virtuale della città, ma è un ambiente dinamico che permette di simulare ed analizzare diversi aspetti della città.

Oltre ad un'attenta analisi dati, è risultata vincente una organizzazione della pipeline e del workflow di lavoro prima di procedere con la costruzione del Digital Twin. Questo ha permesso di individuare sin da subito le maggiori criticità e le maggiori sfide che si sarebbero dovute affrontare. Tra cui la ricostruzione degli edifici, il texturing degli stessi in assenza di informazioni e il processing di dati sul verde pubblico.

Nonostante le sfide poste dai dati di partenza e dalla costruzione di un Digital Twin di una città in assenza di esempi o standard su cui basarsi, il progetto pone solide basi per miglioramenti, svilupppi e richerche future. Infatti, in un futuro prossimo, con molti più dati a disposizione e con infrastrutture atte a produrre dati in tempo reale, simulazioni e predizioni, si potrebbe elevare al suo massimo il Digital Twin di Bologna.

Concludendo, il Digital Twin sviluppato in questo elaborato di tesi e tutti i workflow e le tecniche utilizzate, si potrebbero porre come

riferimento per la costruzione del vero e proprio Digital Twin della città di Bologna.

Bibliografia

[1] 51World. *51World*. URL: https://www.51vr.com.au/.

[2] B Danette Allen. *Digital Twins and Living Models at NASA*. 2021. URL: https://ntrs.nasa.gov/citations/20210023699.

[3] M. Angelidou. «The role of smart city characteristics in the plans of fifteen cities». In: *Journal of Urban Technology* 24.4 (2017), pp. 3–28. DOI: 10.1080/10630732.2017.1348880.

[4] The B1M. *How China Cloned Shanghai*. 2020. URL: https://www.youtube.com/watch?v=hOJZhsNtB6g.

[5] Comune di Bologna. *Bologna avrà un Gemello digitale*. 2023. URL: https://www.comune.bologna.it/notizie/gemello-digitale.

[6] Comune di Bologna. *Bologna Open Data*. URL: https://opendata.comune.bologna.it/pages/home/.

[7] Comune di Bologna. *Comune di Bologna*. URL: https://www.comune.bologna.it/home.

[8] Università di Bologna. *Università di Bologna*. URL: https://www.unibo.it/it.

[9] Buildmedia. *Buildmedia's Digital Twin of Wellington | Build: Architecture 2021*. 2021. URL: https://www.youtube.com/watch?v=Y-0m9GHl86I.

[10] CINECA. *CINECA*. URL: https://www.cineca.it/.

[11] Google Cloud. *Elevation API*. URL: https://developers.google.com/maps/documentation/elevation/overview.

[12] ISTI - CNR. *MeshLab*. Ver. 2023.12. 2023. URL: https://www.meshlab.net/.

[13] Paolo Corradeghini. *DTM VS DSM VS DEM*. URL: https://3dmetrica.it/dtm-dsm-dem/.

[14] Domlysz. *Blender GIS*. URL: https://github.com/domlysz/BlenderGIS.

[15] Unreal Engine. *Sun position calculator*. URL: https://docs.unrealengine.com/4.27/en-US/BuildingWorlds/LightingAndShadows/SunPositioner/.

[16] Blender Foundation. *bpy*. URL: https://docs.blender.org/api/current/info_overview.html.

[17] Epic Games. *Unreal Engine*. Ver. 5.3. 2024. URL: https://www.unrealengine.com.

[18] GeoPandas. *GeoPandas*. URL: https://geopandas.org/.

[19] Alex Gianelli. *Bologna Digital Twin: Modellazione Urbana Data-Driven*. Tesi Magistrale in Ingegnaria Informatica, Università di Bologna. 2024.

[20] Daniel Girardeau-Montaut. *Cloud Compare*. Ver. 2.12.4. 2022. URL: https://github.com/CloudCompare/CloudCompare.

[21] Edward Glaessgen e D. Stargel. «The Digital Twin Paradigm for Future NASA and U.S. Air Force Vehicles». In: *53rd AIAA/ASME/ASCE/AHS/ASC Structures, Structural Dynamics and Materials Conference*. 2012. DOI: 10.2514/6.2012-1818.

[22] Michael Grieves. *Digital Twin: Manufacturing Excellence through Virtual Factory Replication*. 2002.

[23] Fondazione Bruno Kessler. *Fondazione Bruno Kessler*. URL: https://www.fbk.eu/it/.

[24] Juan Linietsky; Ariel Manzur. *Godot*. Ver. 4.2.1. 2023. URL: https://godotengine.org/.

[25] Microsoft. *Azure Functions*. URL: https://azure.microsoft.com/it-it/services/functions/.

[26] Microsoft. *Azure MySQL Database*. URL: https://azure.microsoft.com/it-it/products/mysql.

[27] Comune di Perugia. *Digital Twin Comune di Perugia*. 2023. URL: https://digitaltwin.comune.perugia.it/.

[28] Comune di Perugia. *Digital Twin Comune di Perugia*. 2023. URL: https://www.youtube.com/watch?v=2LA_rqaQbb8.

[29] Ton Roosendaal. *Blender*. Ver. 4.1.0. 2023. URL: https://www.blender.org/.

[30] Benjamin Schleich et al. «Shaping the digital twin for design and production engineering». In: *CIRP Annals* 66.1 (2017), pp. 141–144. DOI: 10.1016/j.cirp.2017.04.040.

[31] Evolved software. *TreeIt*. URL: http://www.evolved-software.com/treeit/treeit.

[32] P. S. Solanki. *Polygon*. URL: https://pypi.org/project/polygon/.

[33] Fei Tao et al. «Digital Twin-driven Product Design, Manufacturing and Service with Big Data». In: *The International Journal of Advanced Manufacturing Technology* 94.9-12 (2019), pp. 3563–3576. DOI: 10.1007/s00170-017-0233-1.

[34] Fei Tao et al. «Digital Twin-driven product design, manufacturing and service with big data». In: *The International Journal of Advanced Manufacturing Technology* 94.9-12 (2018), pp. 3563–3576. DOI: 10.1007/s00170-017-0233-1.

[35] Unity Technology. *Unity*. Ver. 2023.1.5. 2023. URL: https://unity.com/.

[36] vvoovv. *BLOSM*. URL: https://github.com/vvoovv/blosm.

[37] Wikipedia. *Bing Maps*. URL: https://en.wikipedia.org/wiki/Bing_Maps.

[38] Wikipedia. *Esri grid*. URL: https://en.wikipedia.org/wiki/Esri_grid.

[39] Wikipedia. *FBX*. URL: https://en.wikipedia.org/wiki/FBX.

[40] Wikipedia. *GeoTIFF*. URL: https://en.wikipedia.org/wiki/GeoTIFF.

[41] Wikipedia. *Google Earth*. URL: https://en.wikipedia.org/wiki/Google_Earth.

[42] Wikipedia. *Google Maps*. URL: https://en.wikipedia.org/wiki/Google_Maps.

[43] Wikipedia. *Internet of things*. URL: https://en.wikipedia.org/wiki/Internet_of_things.

[44] Wikipedia. *JPEG*. URL: https://en.wikipedia.org/wiki/JPEG.

[45] Wikipedia. *LAS file format*. URL: https://en.wikipedia.org/wiki/LAS_file_format.

[46] Wikipedia. *Leonardo (supercomputer)*. URL: https://it.wikipedia.org/wiki/Leonardo_(supercomputer).

[47] Wikipedia. *Level of detail (computer graphics)*. URL: https://en.wikipedia.org/wiki/Level_of_detail_(computer_graphics).

[48] Wikipedia. *Lidar*. URL: https://en.wikipedia.org/wiki/Lidar.

[49] Wikipedia. *Python (programming language)*. URL: https://en.wikipedia.org/wiki/Python_(programming_language).

[50] Wikipedia. *Shapefile*. URL: https://en.wikipedia.org/wiki/Shapefile.

[51] Wikipedia. *Three.js*. URL: https://en.wikipedia.org/wiki/Three.js.

[52] Wikipedia. *WebGL*. URL: https://en.wikipedia.org/wiki/WebGL.

[53] Wikipedia. *World Geodetic System*. URL: https://en.wikipedia.org/wiki/World_Geodetic_System.

[54] WiseTown. *WiseTown*. URL: https://wise.town/.

[55] Sarah Wray. *How a small German town is using an advanced digital twin*. 2020. URL: https://cities-today.com/how-a-small-german-town-is-using-an-advanced-digital-twin/.

[56] Y. Zheng, S. Yang e J. C. P. Cheng. «Digital Twin-based Smart Production Management and Control Framework for the Complex Product Assembly Plant». In: *Advanced Engineering Informatics* 43 (2020), p. 101043. DOI: 10.1016/j.aei.2019.101043.

Ringraziamenti

Il mio primo, enorme, ringraziamento va alla Professoressa Serena Morigi, relatrice di questa tesi, per avermi dato la possibilità di lavorare a questo stimolante progetto e per il suo prezioso supporto e la sua guida durante tutto il lavoro di tesi.

Un altro sentito ringraziamento va allo staff di Cineca, in particolare alla tutor di questo progetto, Ing. Antonella Guidazzoli, per averci accolti, supportati, guidati e motivati durante tutto il progetto. Sempre di Cineca un rigraziamento va a Silvano e Daniele per il loro supporto tecnico; senza i loro consigli e suggerimenti, portare a termine il progetto sarebbe stato molto più difficile.

Un rigraziamento molto speciale va alla mia famiglia, mia madre Cristina, mio padre Lorenzo e mio fratello Andrea. Grazie per il vostro supporto, la vostra pazienza e per avermi sempre incoraggiato a dare il meglio di me stesso permettendomi e dandomi la possibilità di raggiungere questo importante traguardo. Ringrazio infinitamente anche la mia ragazza, Martina, per avere sempre creduto in me, per il suo supporto incondizionato e per essere stata affianco a me in tutti questi anni.

Ringrazio inoltre Alex, collega all'interno del progetto, ma soprattutto amico. Condividere questa esperienza, le difficoltà e le soddisfazioni, è stato bellissimo. Ringrazio ovviamente anche tutti i miei amici dell'Università, tra cui Gabriele, Michele e Lorenzo. Non avrei potuto chiedere migliori compagni di viaggio in questi anni. Un ringraziamente speciale va anche ad Andrea, mio coinquilino e amico, per il suo supporto, e il divertimento passato in questi anni Universitari.

Infine, il mio ultimo ringraziamento va ai miei amici di sempre, dai tempi dell'asilo, Michele, Giacomo, Gianluca e Marco. Grazie per essere sempre stati al mio fianco, e per avermi insegnato cos'è la vera amicizia.