Università degli Studi di Padova

Facoltà di Ingegneria

Corso di Laurea in Ingegneria Informatica Magistrale

Progettazione e sviluppo di un framework WebGIS per la simulazione real time e realistica in 3D di ambiti urbani, basata sulle caratteristiche delle feature

Laureando

Riccardo Braga

Relatore

Massimo Rumor

Co-relatore

Eduard Roccatello

Anno Accademico 2014/2015

A mia madre Annalisa e mio padre Raffaele,
a mia nonna Marina, mia zia Gianna e mio zio Nando,
a mia nonna Gabriella, le mie zie Anita e Alessandra, i miei zii Moreno e Stefano, le mie cugine Paola, Sara e Sofia e i miei cugini Marco e Matteo,
a mio nonno Edoardo e i miei bisnonni Rina e Marino,
a mia zia Marisa e mio zio Piero,
alle mie sorelle Federica, Valentina e Giorgia e mio fratello Francesco,
alle Bestie,
ai Lupi del Bronx,
ai Mejores,
al maestro Giuliano e tutti i ragazzi della scuola Miyamoto Musashi,
al mio relatore Massimo Rumor, Eduard e i ragazzi di 3DGIS.srl,
questa opera è dedicata a tutti voi che mi avete e continuate ad accompagnarmi nella vita.
Grazie.

Every day is a journey, and the journey itself is home
Matsuo Basho

Indice

1 Introduzione **3**

2 Cenni preliminari **5**
2.1 WebGIS 5
2.2 Texture 5
2.3 Shader 5
2.3.1 Vertex Shader 6
2.3.2 Fragment Shader 6
2.3.3 Rendering Pipeline 6
2.4 UV Mapping 7
2.5 Texture procedurali 8

3 Librerie utilizzate **9**
3.1 OpenGL 9
3.2 WebGL 9
3.3 Three.js 10

4 Architettura di sistema **11**
4.1 GeoDB 11
4.1.1 PostGIS 11
4.2 GeoServer 12
4.2.1 WMS 12
4.2.2 WFS 13
4.2.3 Servizio costruzione tetti 13

5 Realizzazione edificio **15**
5.1 Modello dati 15
5.2 Shape 17
5.3 ExtrudeGeometry 17
5.3.1 Bevel 18
5.3.2 Calcolo Normali vertici 19

5.4 ShaderMaterial 20
5.5 Costruzione tetto 23
5.5.1 Straight Skeleton 24

6 Geometrie aggiuntive **27**
6.1 Scena e illuminazione 27
6.2 Strade 28
6.2.1 Vertex e Fragment Shader 29
6.3 Alberi 31
6.3.1 Blender 32
6.3.2 Importazione modello 32
6.4 Segnali stradali 34

7 Conclusioni e possibili miglioramenti **37**

Bibliografia **43**

Abstract

L'aumento delle risorse in termini di informazioni e dati nel settore dei GIS, permette di avere sempre più disponibili e a portata di mano caratteristiche e dati riguardanti gli ambienti urbani di ogni area geografica. Grazie all'avanzare della tecnologia informatica è sempre più possibile abbandonare il linguaggio simbolico delle mappe a favore della rappresentazione 3D di queste, anche se sono tuttora presenti problematiche riguardanti le tecniche di modellazione 3D in realtime e di visualizzazione su web di un modello tridimensionale realistico. Obiettivo di questo lavoro di tesi è quello di risolvere le problematiche evidenziate sfruttando le più recenti librerie grafiche. Per realizzare la modellazione e rappresentazione degli oggetti su web, verrà utilizzato Three.js, libreria grafica open-source specializzata per lo sviluppo web. L'architettura dell'applicativo si basa su un geodatabase, contenente le posizioni e i punti che formano lo shape degli oggetti e le caratteristiche che caratterizzano gli oggetti (per un edificio parametri come altezza, numero piani, tipo di edificio, finestre, porte, ecc.), un GeoServer che gestisca il trasferimento delle geometrie dal DB, un servizio apposito che implementi l'algoritmo Straight Skeleton e restituisca per ogni edificio la geometria di ogni tetto e l'applicativo web che realizza la modellazione 3D di tutti gli oggetti in base alle geometrie ricevute e le applica proceduralmente. Le strutture urbane prese in considerazione per questo progetto sono edifici, strade, alberi e segnali stradali. Gli edifici inoltre vengono renderizzati utilizzando gli shader per permettere di mappare proceduralmente e in base ai parametri ricevuti le varie texture. I risultati ottenuti si sono rivelati soddisfacenti e il prodotto ottenuto realizza l'obiettivo prefissato, realizzando tutte le strutture prese in considerazione caratterizzandole con i parametri disponibili per ognuna di essa, il tutto con un esecuzione real-time.

Capitolo 1

Introduzione

L'universo informatico nell'ultimo ventennio ha visto e continua a vedere una continua e sempre più importante crescita in ogni suo ambito, dallo sviluppo web ai sistemi embedded più avanzati e moderni. Tra questi anche lo scambio e l'acquisizione di dati avvengono con frequenza e velocità sempre più crescenti, favorendo la diffusione di informazioni interessanti per ogni possibile ecosistema, privato o pubblico. Di pari passo, anche la mole dei dati è sempre maggiore, più dettagliata, variegata e ricca; basti notare la mole di dati e informazioni all'interno di youtube, google, bing ecc.

Questa mirabolante quantità di informazioni, fenomeno noto come Big Data, copre varie aree di interesse e fra queste sono comprese anche i GIS. Un Geographic Information System (GIS), è un sistema adibito a gestire, memorizzare, rappresentare e computare dati di tipo geografico. Questi dati comprendono planimetrie di edifici, strade e terreni e sono sfruttati per avere una visione dinamica e costruttiva dell'ambiente in esame. Dati come questi possono anche essere usate per realizzare veri e propri modelli 3D, partendo da una qualunque planimetria ed effettuandone il rendering in tempo reale.

Le operazioni di rendering sono generalmente usate in ambito video-ludico e video-editing ma anche in questo frangente, sfruttando le più recenti tecnologie in ambito di realizzazione di texture procedurali e motori grafici, è possibile realizzare un applicativo web che possa essere una rappresentazione virtuale di un determinato territorio.

Sfruttando tutte queste novità, si possono realizzare applicativi web sempre più efficienti e che offrono sempre più informazioni in tempo reale, dando all'utente un senso di realismo e di interazione sempre migliori. Il progetto qui trattato propone lo sviluppo di un sistema che utilizzi i dati geografici di un qualsiasi ambiente urbano (comprensivo di strutture come case, palazzi, ecc.) per poterli rappresentare in tempo reale e tridimensionalmente su web, personalizzandoli e aggiungendo loro varie caratteristiche quali possono es-

sere il tipo di edificio, l'altezza, la posizione delle finestre e altro ancora. Al fine di rendere questa rappresentazione ancora più realistica vengono aggiunti altri dettagli urbani quali strade, segnali stradali e alberi, anch'essi con le proprie caratteristiche da utilizzare al fine di rendere unico e dettagliato ogni oggetto nella scena. Obbiettivo finale di questo progetto è quello di riuscire a ricreare in modo virtuale e in tempo reale un ambiente urbano completo per scopi che possono spaziare dalla semplicemente visualizzazione grafica tridimensionale allo studio e gestione urbanistica di un qualunque complesso urbano.

Capitolo 2

Cenni preliminari

2.1 WebGIS

Con il termine WebGIS si va a indicare un insieme di applicativi inerenti al settore dei Sistemi Informativi Geografici che vengono pubblicati ed eseguite tramite interfaccia Web. Esattamente come un GIS infatti, anche i WebGIS si occupano di gestire dati cartografici e geo-spaziali ma permettono in particolar modo una migliore condivisione di informazioni per altri utenti.

Il sistema che verrà qui trattato è progettato e sviluppato per poter ricevere questa tipologia di dati e per poterli processare in tempo reale su interfaccia web.

2.2 Texture

Nell'ambito della computer grafica, una texture è un'immagine bidimensionale che viene applicata sulle facce di una geometria 3D per aggiungere realismo e dettagli.

2.3 Shader

Lo shader rappresenta una serie di istruzioni legate alla computer grafica 3D che permettono la realizzazione e visualizzazione dell'aspetto finale di ogni oggetto 3D (o anche 2D) sul quale viene applicato, permettendo inoltre di aggiungere effetti speciali o anche effettuare operazioni di post-processing sulle texture. Gli shader sono strumenti messi a disposizione da librerie grafiche quali OpenGL e Direct3D, grazie anche all'evoluzione delle GPU degli ultimi anni.

In questo lavoro l'utilizzo degli shader ha permesso la possibilità di creare e adattare a geometrie molto diverse tra loro e non uniformi, texture ed effetti grafici in base a parametri noti, quali la tipologia dell'edificio o il numero di corsie per una strada, allo stesso tempo semplificando il lavoro per la CPU poichè la componente grafica è interamente gestita dalla GPU. Sono stati utilizzati, tramite la libreria usata, due tipi diversi di shader: Il Vertex Shader e il Fragment Shader.

2.3.1 Vertex Shader

Il Vertex Shader lavora esclusivamente sui vertici della geometria e viene eseguito per ognuno di essi. Il suo scopo è quello di convertire la posizione 3D del vertice nella scena in una coordinata 2D che possa apparire sullo schermo. Di default, ciò avviene attraverso moltiplicazioni matriciali tra la posizione del vertice e matrici di trasformazione generalmente immesse tramite input.

In questo lavoro il Vertex Shader utilizzato non è stato modificato più di tanto dal modello standard, poichè non siamo in un caso in qui la geometria debba essere modificata a livello di vertici o colorata in base alla posizione di questi ultimi. Tuttavia tramite questo shader è stato possibile gestire le normali dei vertici, le quali sono state sfruttate dal Fragment Shader per capire la direzione della faccia da disegnare.

2.3.2 Fragment Shader

Largamente più utilizzato del Vertex Shader, il Fragment Shader ha lo scopo di disegnare per ogni fragment (ovvero ogni pixel) un colore. Essenzialmente il programma per ogni pixel restituisce un colore, che può essere un qualsiasi colore determinato proceduralmente oppure una texture in base alla mappatura uv che anche quest'ultima può subire modifiche in base alle esigenze. Tramite quest'ultimo shader è stato possibile applicare le texture alle varie geometrie in base ai vari parametri di queste ultime quali altezza e dimensioni.

2.3.3 Rendering Pipeline

La serie di passaggi effettuati dal calcolatore per realizzare la visualizzazione su schermo di una scena 3D può essere riassunta in questi passi:

1. La GPU compila lo shader e riceve i dati della geometria a esso associata;

2. Il Vertex Shader effettua la trasformazione dei vertici della geometria;

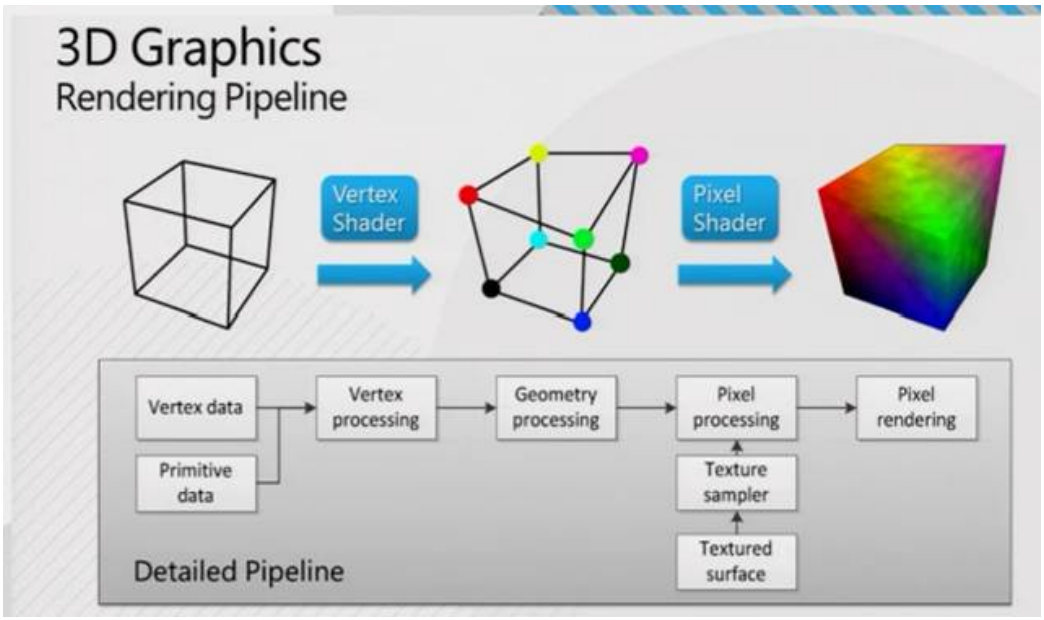

Figura 2.1: Schema di una pipeline

3. La geometria viene triangolata, ovvero suddivisa in triangoli, che rappresentano la forma più semplice per la rappresentazione grafica;
4. I vari triangoli vengono raggruppati in fragment a 2 a 2;
5. I fragment vengono processati dal Fragment Shader;
6. Viene calcolata la profondità, coloro che passano il depth test (ovvero che effettivamente si vedono in base alla posizione della camera) vengono rappresentati sullo schermo.

2.4 UV Mapping

L'operazione di UV Mapping prevede di proiettare una texture in 2D su un oggetto in 3D. Le lettere U e V derivano dal fatto che convenzionalmente sono utilizzate per rappresentare le coordinate della mappatura. I valori delle due coordinate variano da 0 a 1 per effettuare spostamenti e/o ridimensionamenti mentre un

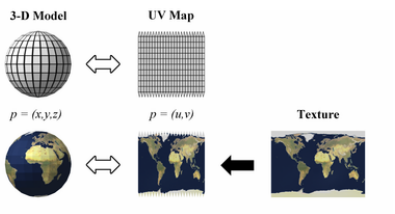

Figura 2.2: Mappatura UV

valore superiore a 1 farebbe ripetere
la texture (con 2 la texture compare 2 volte nello stesso spazio). In fase di rendering verranno poi utilizzate le coordinate UV per determinare la distribuzione dell'immagine sull'oggetto 3D.

2.5 Texture procedurali

Le Texture procedurali sono texture definite matematicamente. Sono generalmente relativamente semplici da usare, perché non necessitano di essere mappate in maniera particolare - il che non significa che le texture procedurali non possano diventare molto complesse.

Questi tipi di texture sono 3D 'reali'. Con questo intendiamo che esse coincidono perfettamente ai contorni e che continuano ad apparire come le si è pensate anche nel caso che vengano tagliate; come se un blocco di legno fosse realmente stato tagliato in due.

Le texture procedurali non sono filtrate o anti-scalettate. E' raro che questo diventi un problema: l'utente può facilmente tenere le frequenze specificate in limiti accettabili. Le texture procedurali possono produrre trame colorate, trame di sola intensità, trame con valori alpha e con i valori delle normali.

Capitolo 3

Librerie utilizzate

3.1 OpenGL

OpenGL è una application programming interface (abbreviato API) che permette di realizzare di gestire e realizzare elementi di grafica, che possono essere oggetti, immagini sfruttando l'hardware, senza legarsi a esso ma rimanendo indipendente dalla piattaforma.

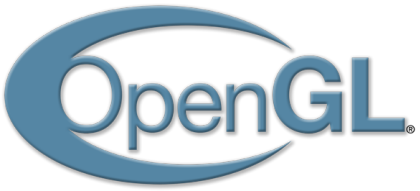

Figura 3.1: Logo OpenGL

Sviluppato nel 1992 dalla Silicon Graphics Inc., a oggi è una delle librerie grafiche più diffuse sia in ambito videoludico che in ambito CAD e realtà virtuale e su ambiente UNIX è considerato lo standard grafico. le funzionalità introdotte da OpenGL sono a oggi alla base delle schede video più moderne destinate al mercato privato, alcune di queste sono:

- Z-buffering;
- Texture mapping;
- Linee, punti e triangoli come primitive grafiche per rappresentare geometrie complesse;
- Alpha blending.

3.2 WebGL

Figura 3.2: Logo WebGL

WebGL (Web-based Graphics Library) è una API multi-piattaforma di grafica 3D per i web browser, realizzata da Khronos Group. Basata su OpenGL ES (2.0), ne sfrutta lo shading language (noto anche come GLSL ES) ed è specializzato nella realizzazione di applicazioni web 3D dinamiche tramite il linguaggio Javascript, le quali si integrano appieno sui più moderni browser.

L'accesso alle interfacce messe a disposizione sono accessibili tramite apposite istruzioni Javascript, le quali saranno poi chiamate dal tag *canvas* presente in *HTML5*, il cui scopo è per l'appunto la rappresentazione di elementi grafici 2D e 3D. Inoltre WebGL coesiste con gli altri contenuti della pagina HTML ponendosi a un livello più basso o più alto rispetto agli altri oggetti presenti.

3.3 Three.js

Figura 3.3: Logo Three.js

Three.js rappresenta una moderna e leggera libreria grafica open-source appositamente creata per lo sviluppo di grafica web. Rilasciata nel 2010 da Ricardo Cabello (meglio noto sul web come Mr.doob), è compatibile con ogni browser che supporti WebGL, HTML5 e SVG. La libreria in questione spicca fra tutte le altre per la completezza e stabilità del proprio codice, permette di realizzare e modificare in ogni suo punto un renderer WebGL e offre una vasta gamma di oggetti 3D sui quali è possibile applicare la maggior parte dei materiali definiti nello studio della grafica 3D oltre ad altri elementi quali luci, ombre, animazioni e interazione.

Capitolo 4

Architettura di sistema

Alla base del progetto sono necessarie una serie di strutture per permettere all'interfaccia web di ricevere correttamente e opportunamente processate i dati di tutte le strutture urbane presenti nel database. Sarà quindi presente un server geo-spaziale i cui servizi forniranno al sistema le informazioni necessarie, prelevate a loro volta da un apposito database di dati geo-spaziali.

4.1 GeoDB

Un database di dati geo-spaziali è un database specializzato nella gestione di informazioni e dati che possono rappresentati come oggetti all'interno di uno spazio geometrico. Questi database permettono la rappresentazione di vari tipi di oggetti come punti, linee e poligoni e offrono funzionalità per processare tali entità in maniera efficiente, permettendo di realizzare strutture più complesse come le geometrie.

4.1.1 PostGIS

Il database viene implementato utilizzando PostgreSQL come DBMS (Data-Base Managment System) con l'estensione PostGIS. L'estensione in questione, rilasciata sotto licenza GPLv2, permette di integrare in Postegre SQL le funzionalità geo-spaziali trattate nella sezione precedente, aggiungendo al DBMS feature quali geometrie e operazioni su di esse.

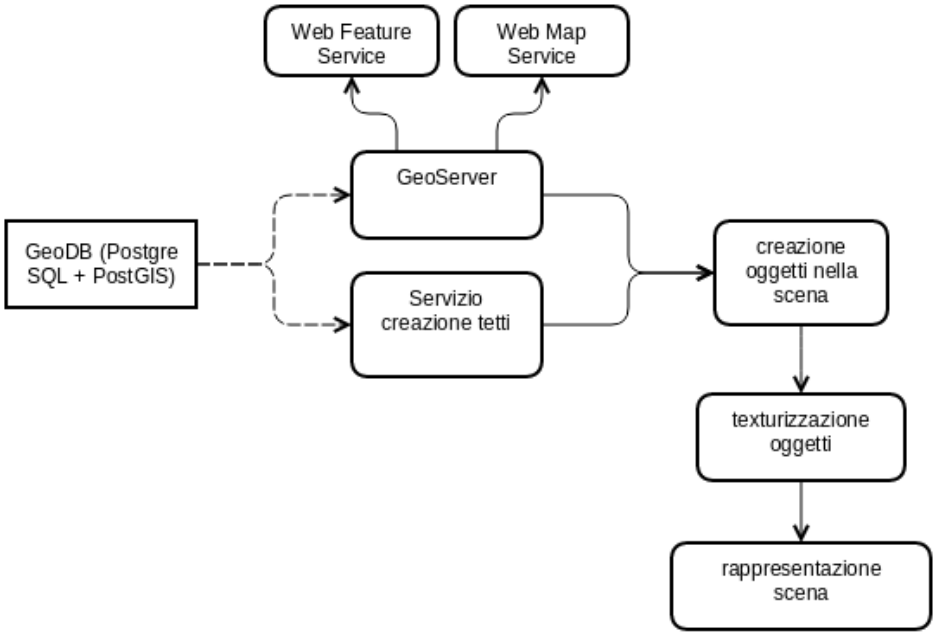

Figura 4.1: UML Activity dell'architettura

4.2 GeoServer

Un server geo-spaziale è un sistema che permette la condivisione e l'elaborazione di dati-geo-spaziali via web. Ne esistono varie implementazioni e quella che verrà considerata per il progetto in questione è GeoServer.

GeoServer è un software open source rilasciato sotto licenza GPL, realizzato in Java che permette di modificare e condividere dati geo-spaziali da ogni insieme di dati geo-spaziali conforme allo standard OGC (Open Geospatial Consortium). GeoServer implementa standard che verranno sfruttati nel progetto come il WMS (Web Map Service) e il WFS (Web Feature Service) per l'invio al sistema delle geometrie e vettori necessari per la creazione degli oggetti tramite Three.js.

4.2.1 WMS

Web Map Service (WMS) è uno standard definito da OGC per la condivisione di informazioni geo-spaziali distribuite via web sotto forma di mappe, le quali vengono visualizzate come immagini in vari formati (JPEG, PNG, GIT, GeoTIFF, ecc.) o come elemento vettoriale (SVG) su qualsiasi browser.

4.2.2 WFS

Web Feature Service (WFS) è uno standard definito da OGC e definisce interfacce per l'accesso e la gestione di feature geo-spaziali utilizzando il protocollo HTTP. Utilizzando queste interfacce un utente può prelevare, modificare e combinare dati geo-spaziali codificati in GML (Geography Markup Language, grammatica definita dall'OGC per esprimere oggetti geografici) da più sorgenti di dati.

4.2.3 Servizio costruzione tetti

Parallelamente ai servizi per la condivisione dei dati geo-spaziali, ne viene utilizzato un altro realizzato apposta e il cui scopo è quello di restituire una geometria per i tetti degli edifici che si andranno a visualizzare, dei quali si tratterà nel capitolo 5. Il servizio riceve la geometria dell'edificio direttamente dal GeoDB e vi applica l'algoritmo di Straight Skeleton (del quale si tratterà nella sezione 5.5.1), inoltre allarga di un certo parametro la geometria, in modo tale da dare al tetto spazio per elementi come le grondaie. La geometria così processata viene passata direttamente all'applicativo sulla quale verranno applicate le tecniche di cui si parlerà nella sezione 5.5.

Capitolo 5

Realizzazione edificio

Il modello principale che si andrà a rappresentare e che sarà caratterizzato dal maggior numero di dettagli grafici in quest'opera sono gli edifici, i quali saranno caratterizzati da vari parametri, i quali determineranno la tipologia di edificio, l'altezza, la posizione e dettagli quali finestre e porte. Nell'immagine 5.1 è possibile visualizzare un grafico UML che mette in evidenzia l'insieme di attività che vengono svolte nell'intero sistema sviluppato.

5.1 Modello dati

Il dato fondamentale con il quale si rappresenta un edificio è la planimetria 2D dello stesso, ovvero l'insieme dei punti che, una volta collegati in ordine, andranno a formare quella che è la superficie che verrà occupata dal modello. In Three.js questo dato viene per l'appunto rappresentato tramite un insieme di punti 2D; ognuno di questi viene rappresentato tramite l'oggetto *Vector2*, una classe di Three.js che implementa un vettore bidimensionale tra cui parametri ed operazioni tra di essi.

Eventuali buchi all'interno della geometria (si pensi al cortile interno di un palazzo, visto dall'alto), si possono gestire allo stesso modo: anch'essi vengono rappresentati come superfici e verranno utilizzati in un'altra classe di cui parleremo più avanti.

Gli altri parametri che comporranno l'edificio sono semplici parametri, con i quali si indicheranno l'altezza dell'edificio (in metri), il tipo dell'edificio (moderno, vecchio ecc.) e i lati dell'edificio in cui verranno rappresentate finestre e porte.

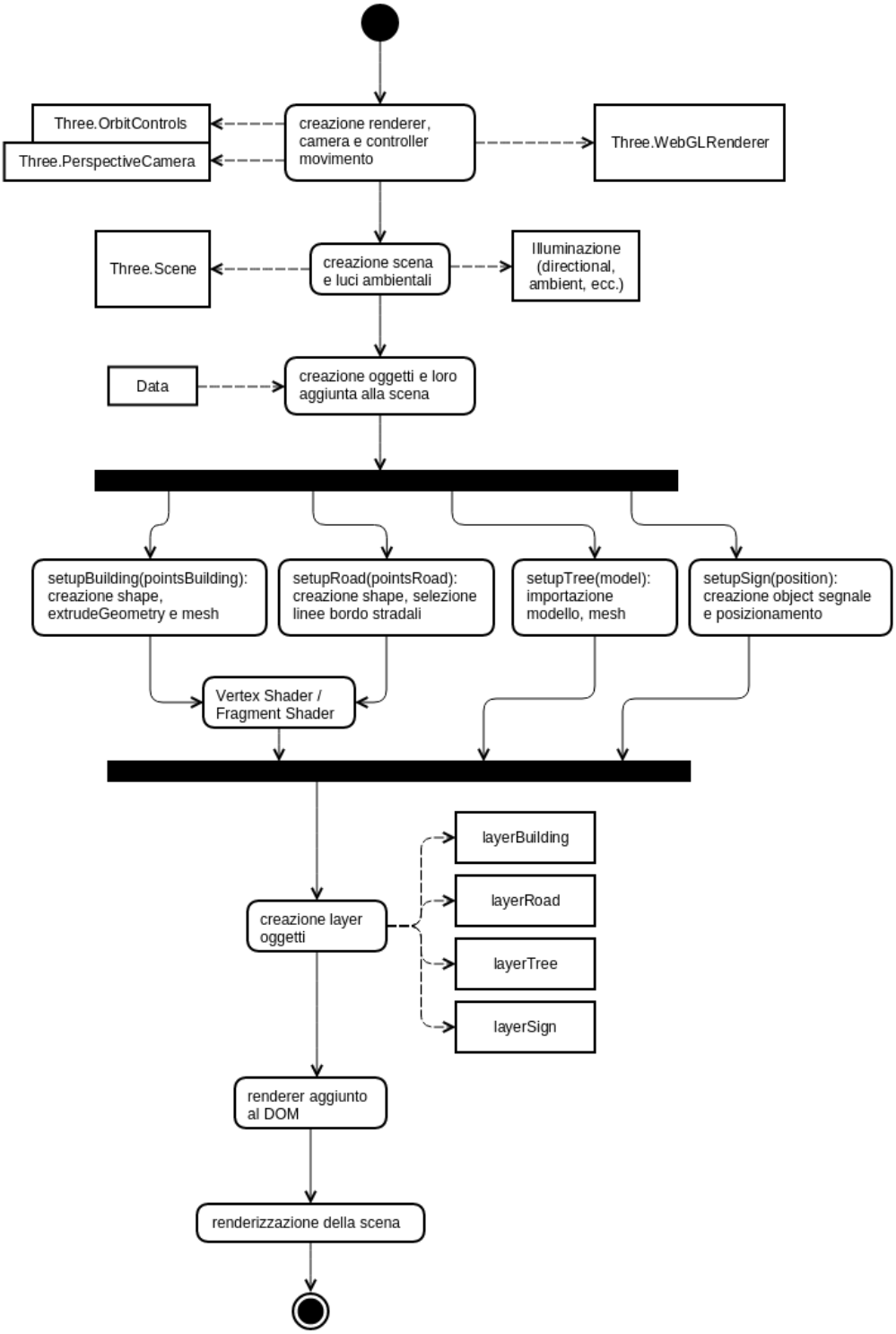

Figura 5.1: UML Activity

5.2 Shape

La geometria precedentemente citata verrà utilizzata come parametro per implementare la classe *Shape*, la base per la realizzazione della struttura. L'oggetto *Shape* unirà tutti i punti presenti nei dati e in un parametro apposta verranno definiti, se presenti, tutti i buchi presenti nella superficie stessa.

L'utilità di questa classe, oltre all'unire tutti i punti rappresentanti la superficie di uno stabile, consiste nel fornire anche una serie di metodi che permettono di ottenere una *Geometry*, oggetto fondamentale in Three.js, poiché grazie a essa si potrà realizzare la *Mesh* che verrà poi renderizzata su schermo.

5.3 ExtrudeGeometry

Implementando la classe *ExtrudeGeometry*, è possibile dare una forma concreta e tridimensionale alla struttura da realizzare: Oltre a ereditare i vari metodi della superclasse *Geometry*, permette di effettuare un extrude (ovvero una estrusione, in questo caso intesa come estensione di una figura 2D in maniera tale da renderla 3D) dello *Shape* creato in precedenza.

La classe in questione dispone di vari parametri impostabili, tali da rendere più dettagliata e varia la geometria; non tutti saranno oggetto di analisi in quanto solo alcuni sono di interesse per il progetto in esame, ma tra i più significativi possiamo notare:

amount: viene utilizzato per definire quanto estendere verso l'alto la superficie e dargli un aspetto tridimensionale; il valore in questione viene preso dal modello ed è espresso da un singolo valore in virgola mobile maggiore di 0;

steps: si usa per dividere la superficie estesa in punti intermedi (in base al valore indicato in amount e il numero di step desiderati, vengono aggiunti vertici in modo da dividere la struttura in vari parallelepipedi uno sopra l'altro invece di crearne uno unico) e ciò semplificherà e migliorerà la mappatura delle texture sulla mesh, evitando che la texture rischi di apparire tirata o di effettuare maggiori calcoli per correggere tale situazione;

bevelEnabled: se abilitato, permette di applicare il bevel alla geometria, se ne parlerà più approfonditamente nella sottosezione 5.3.1;

bevelThickness/bevelSize: se il bevel è abilitato, questi valori permettono di impostare la profondità del bevel e quanto questa si distanzia dalla geometria rispettivamente;

uvGenerator: questo parametro, se utilizzato, riceve una classe il cui scopo è impostare le coordinate uv per le superfici degli edifici. Questa classe è composta da due funzioni:

generateTopUV: questa funzione si occupa di calcolare le uv per quelli che verrebbero rappresentati come il tetto e la base dell'edificio e nel caso di questo progetto non viene sfruttata (verrà utilizzata quella offerta dalla libreria) in quanto entrambe le superfici non si vedranno: al posto del soffitto infatti verrà visualizzato il tetto, del quale si parlerà più nel dettaglio in una sezione successiva;

generateSideWallUV: al contrario, questa funzione riguarda i muri dell'edificio, la cui mappatura uv dovrà essere regolata ad hoc in base ai vari parametri della geometria e in maniera tale che la texture che rappresenterà la superficie del muro si possa ripetere nella maniera più uniforme possibile. Ciò si ottiene calcolando le uv utilizzando come base il lato più corto dello Shape, in modo da regolare il numero di texture da ripetere in base al rapporto tra il lato in esame e quello più corto.

5.3.1 Bevel

Con il termine bevel, si indica un taglio fatto su uno spigolo di una qualsiasi figura tridimensionale tale per cui la superficie di detto taglio non sia perpendicolare alle facce della figura in esame. In parole povere è come se uno spigolo di un solido venisse levigato in maniera tale che al posto della spigolo ci sia una nuova faccia, e di conseguenza al posto del vertice dello spigolo, ce ne sono due per la nuova faccia.

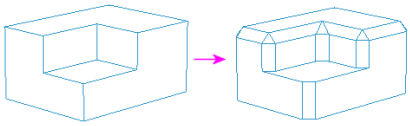

Figura 5.2: Una geometria prima e dopo l'applicazione del bevel

L'utilità di applicare questa tecnica anche nella geometria dell'edificio è legato alle normali dei vertici della geometria. Per normale si intende un vettore direzionale associato a ogni vertice della geometria e si ottiene

tramite la media normalizzata di tutte le normali delle facce che contengono quel vertice. Consideriamo il vertice di un qualsiasi spigolo della struttura in esame: essendo unico in quel punto, la normale che ne risulterà sarà un vettore diagonale, dovuto al fatto che ci sono sia facce verticali che orizzontali che contribuiscono al calcolo della normale del vertice, di conseguenza l'effetto che ne risulterà sarà un effetto non realistico di qualsiasi fonte di illuminazione che colpisca la struttura.

5.3.2 Calcolo Normali vertici

Il problema del calcolo delle normali dei vertici si può ovviare dividendo il vertice in questione in almeno 3 vertici, tutti vicinissimi tra loro (così da sembrare un unico punto) e, in base alla posizione, a ognuno di questi assegnare una normale parallela alla normale della faccia a cui essi sono più vicini. Ciò permette di mettere bene in risalto lo spigolo e il vertice stesso in scenari di luci direzionale o simili, risolvendo il problema sopracitato.

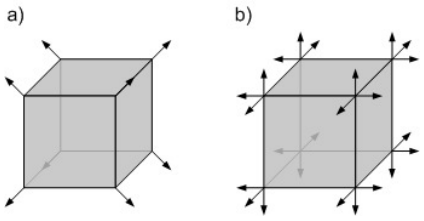

Figura 5.3: a) normali standard b) normali dopo aver applicato il bevel, parallele alla faccia di appartenenza

Per ottenere questa situazione in parte basta permettere alla geometria di utilizzare il bevel; la libreria si occuperà automaticamente di aggiungere vertici e facce necessari per dare l'effetto voluto in base ai parametri desiderati. Con un valore elevato infatti saranno ben visibili le facce e i vertici aggiunti. Non è questo il caso in questione, poiché a vista d'occhio non si deve notare questo stratagemma, per cui si è scelto di usare un bevel molto basso (circa 0.001). C'è da aggiungere che, diversamente da quanto descritto prima, in questo caso il bevel non cambia l'altezza della geometria, semplicemente vertici e facce vengono aggiunti a partire dal vertice più alto, senza modificare quest'ultimo.

Con la tecnica fornita dalla libreria, si ottiene però solo un punto aggiuntivo e per quanto dichiarato prima, ne servono almeno 3. Per ovviare a questo problema è bastato applicare il principio del bevel anche ai punti dello shape, creando lo stesso effetto anche in verticale e con lo stesso valore utilizzato prima. In questo modo otteniamo al posto di un vertice dello spigolo superiore ben 4 vertici, più che sufficienti per realizzare quanto citato prima. Poichè abbiamo diviso tutti i punti dello shape in 2 punti molto vicini, anche il bevel che *ExtrudeGeometry* andrà a realizzare creerà due nuovi

vertici, ecco spiegato il quarto vertice il quale per semplicità viene trattato come l'altro creato dal bevel, di conseguenza entrambi avranno la normale direzionata verso l'alto, ovvero lungo l'asse z.

5.4 ShaderMaterial

Procedura fondamentale nella costruzione di ogni oggetto è quella di definire un *Material*, con il quale andremo a creare la *Mesh*, la quale verrà aggiunta alla scena e sarà visualizzabile su monitor.

Al fine di avere la massima gestione delle texture che andranno a rappresentare l'edificio, verrà utilizzato uno *ShaderMaterial* che come indica il nome sfrutta gli shader e quindi la GPU. Le texture utilizzate vengono inviate allo shader tramite uniform, e di ciò se ne occupa Three.js in fase di creazione del *Material*, insieme a costanti ed attributi.

Il compito del Vertex Shader rimane quello citato nel capitolo 2.3.1, e in più vengono passati al Fragment Shader le normali di ogni vertice, che saranno utili per capire come mappare correttamente le texture su ogni lato dell'edificio, a prescindere della loro posizione e angolo di curvatura rispetto agli assi di riferimento della scena.

Nel Fragment Shader avviene la vera e propria opera di texturizzazione: si procede andando a posizionare su tutti i muri una texture di base che definisca il materiale della struttura (cemento, mattoni, ecc.) dopodiché si vanno ad aggiungere ulteriori texture in base a quanto specificato nel modello dati; per semplicità sono state scelte texture rappresentanti porte e finestre. Le posizioni delle porte e delle finestre vengono decise in base a vari parametri definiti dal modello dati e non solo, quali altezza e larghezza delle texture, distanza tra queste ultime, altezza dell'edificio, ecc.

Tramite questi parametri ricevuti dallo Shader sempre in forma di uniform, è possibile sistemare e correggere la mappatura uv di ogni texture in base a dove quest'ultima verrà rappresentata. Nel caso si voglia variare alcuni parametri, quali dimensioni o numero di piani dell'edificio, il codice dello Shader non verrà toccato, rendendo ogni possibile cambiamento scalabile e privo di rischi per il sistema stesso. Negli algoritmi 5.2 e 5.3 è possibile esaminare il codice implementato per gestire le texture di porte e finestre rispettivamente.

Una volta realizzato anche lo *ShaderMaterial*, unendolo alla *ExtrudeGeometry* è possibile realizzare la *Mesh* e verrà aggiunta alla scena.

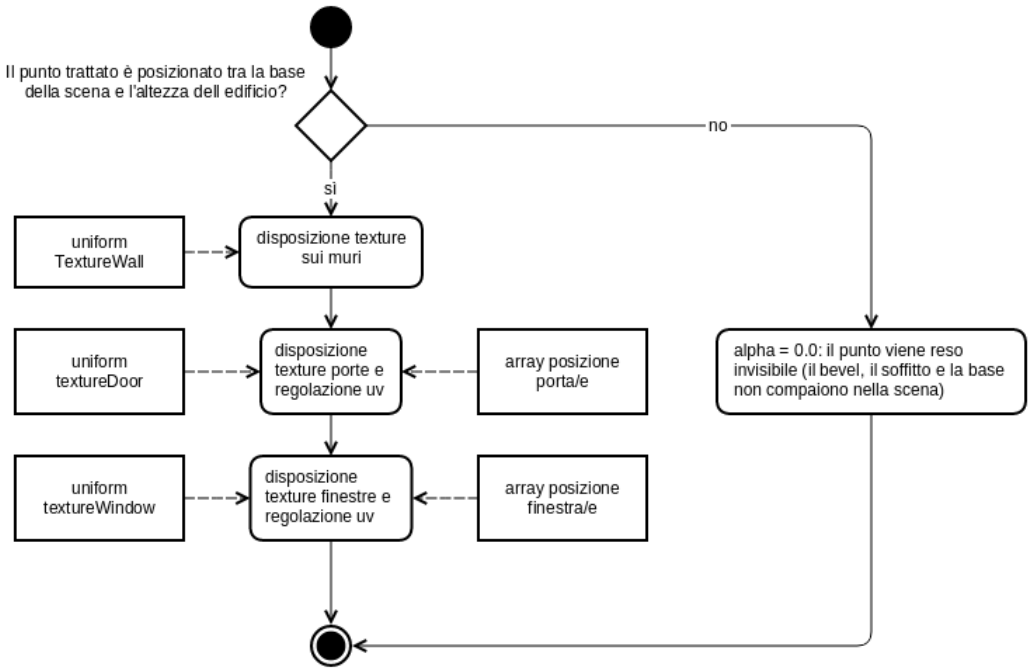

Figura 5.4: UML Activity del Fragment Shader di un edificio

```
varying vec2 vUv;
varying vec3 vPos;
varying vec3 vNormal;

attribute vec3 fNormal;

void main() {
  vPos = position;
  vUv = uv;
  vNormal = fNormal;
  gl_Position = projectionMatrix * modelViewMatrix * vec4(
      position, 1.0 );
}
```

Algoritmo 5.1: Vertex Shader di un edificio

```
for (int i = 0; i < DOOR_ARR_LEN; i += 4) {
  if(within(uDoors[i].z, uDoors[i+1].z, vPos.z)){
    if(within(uDoors[i].y,uDoors[i+1].y, vPos.y) && within(uDoors
       [i].x,uDoors[i+1].x, vPos.x)){

      float uvOffset;
      if(position == 0)
        uvOffset = min(abs(uDoors[i+2].x - uDoors[i+1].x), abs(
           uDoors[i+3].x - uDoors[i].x));
      else
        uvOffset = min(abs(uDoors[i+2].y - uDoors[i+1].y), abs(
           uDoors[i+3].y - uDoors[i].y));

      vec2 vUvD = vUv;
      vUvD.x -= uvOffset / float(SHORTEST_EDGE) / offset;
      vUvD.x /= DOOR_WIDTH / float(SHORTEST_EDGE);
      vUvD.y /= float(DOOR_HEIGHT) / float(FLOOR_HEIGHT);

      gl_FragColor = texture2D(uTexDoor, vUvD);
    }
  }
}
```

Algoritmo 5.2: Disposizione texture porte

```
for (int i = 0; i < WINDOW_ARR_LEN; i += 2) {
  if(within(uWindows[i].z, uWindows[i+1].z, vPos.z)){
    if(within(uWindows[i].x,uWindows[i+1].x, vPos.x) && within(
       uWindows[i].y,uWindows[i+1].y, vPos.y)){
      float uvOffset;
      if(position == 0)
        uvOffset = abs(uVertexWin[i+1].x - uWindows[i].x);
      else
        uvOffset = abs(uVertexWin[i+1].y - uWindows[i].y);

      float uvOffsetZ = abs(uVertexWin[i].z - uWindows[i].z);

      vec2 vUvW = vUv;
      vUvW.x -= uvOffset / float(SHORTEST_EDGE) / offset;
      vUvW.x /= float(WINDOW_WIDTH) / float(SHORTEST_EDGE);
      vUvW.y -= uvOffsetZ / float(abs(uVertexWin[i].z -
          uVertexWin[i+1].z));
      vUvW.y /= float(WINDOW_HEIGHT) / float(abs(uVertexWin[i+1].
          z - uVertexWin[i].z));

      gl_FragColor = texture2D(uTexWindow, vUvW);
    }
  }
}
```

Algoritmo 5.3: Disposizione texture finestre

5.5 Costruzione tetto

Come elemento scenico aggiuntivo verrà aggiunto un tetto che andrà a coprire la parte superiore della *Mesh*, la quale è stata lasciata invariata proprio perché non sarebbe stata visualizzata. Per procedere in questo caso come prima andremo a sfruttare lo Shape della struttura per il perimetro, mentre per realizzare i punti interni (che collegati con i punti dello Shape daranno la forma di un tetto), ci si baserà su un algoritmo detto *Straight Skeleton*.

Per semplicità, non vengono usate texture per definire il Material del tetto, ma viene usato un *LambertMaterial*, presente nella libreria Three.js. Anche il calcolo delle normali non viene approfondito come nel caso dell'*ExtrudeGeometry* in quanto, non avendo a disposizione la stessa classe ma una generica *Geometry*, non è presente la stessa funzionalità di bevel di cui si è parlato in precedenza, per cui ci si accontenta di gestire le normali in maniera tradizionale.

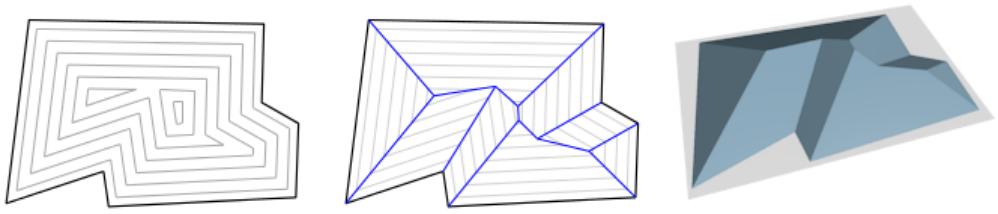

Figura 5.5: Fasi dell'algoritmo Straight Skeleton su una geometria

5.5.1 Straight Skeleton

Lo Straight Skeleton di un qualsiasi poligono consiste in uno o più punti interni alla geometria, spesso rappresentati come una spezzata uniti agli altri vertici della struttura, senza che questi ultimi siano uniti tra di loro e la figura che si ottiene ricorda per l'appunto uno scheletro.

Per ottenere questi punti interni, la tecnica utilizzata è quella di restringere progressivamente e in maniera costante la geometria del poligono, fino a rimpicciolire il poligono il più possibile, arrivando anche a rimuovere eventuali vertici, riducendolo a un triangolo. Una volta effettuata questa operazione basta unire ogni vertice con le sue versioni ridotte, fino ad arrivare al centro del poligono più piccolo generato e una volta fatto ciò, basta unire tutti i punti interni che si sono creati per ottenere lo Straight Skeleton.

Il principio per la realizzazione dei tetti si basa su questa tecnica, della quale sono presenti vari algoritmi i quali però non verranno trattati in questo progetto; semplicemente si prenderanno i punti risultanti dall'algoritmo e verranno sfruttati per realizzare la geometria del tetto, con i punti interni che verranno rialzati di un certo valore rispetto ai punti del perimetro per dare l'idea di un tetto vero e proprio.

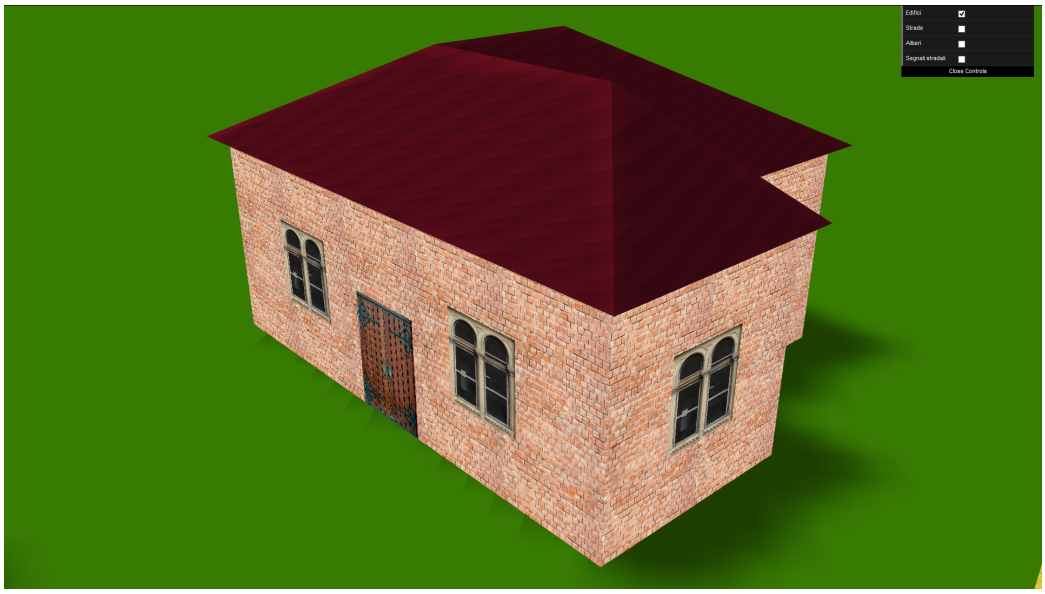

Figura 5.6: Esempio di edificio

Capitolo 6

Geometrie aggiuntive

Per il progetto trattato, gli edifici sono l'elemento principale e maggiormente evidenziato di conseguenza, sia per la loro complessità in fase di sviluppo che per la loro importanza a livello urbanistico. Ciò non toglie comunque che al fine di rendere lo scenario da visualizzare più realistico, si rendono necessari ulteriori sviluppi riguardanti altre strutture nella quasi totalità dei casi presenti in un complesso urbano, oltre a realizzare un ambiente circostante che possa essere più vicino possibile alla realtà. Non sono stati sviluppati in dettaglio altrettanto complessi ma in maniera sufficiente per raggiungere quanto sopra spiegato.

6.1 Scena e illuminazione

La realizzazione di un ambiente naturale in cui andare a creare il complesso urbano trattato finora è la prima cosa che viene realizzata ed è fondamentale per rendere più realistico l'insieme degli elementi. Non essendo comunque un tratto fondamentale del progetto, verranno rappresentati con elementi semplici forniti dalla libreria Three.js.

La scena viene rappresentata su un piano quadrato utilizzando la classe *PlaneGeometry*, la quale viene implementata con l'utilizzo di un semplice *SimpleMaterial* TODO.

Per emulare una illuminazione più simile possibile a quella solare invece, sono state utilizzate due classi:

HemisphereLight: con questa classe, la luce non è generata da un punto in particolare, ma arriva da tutte le direzioni, emulando una luce ambientale. Inoltre è possibile definire il colore della luce in base alla direzione della normale del punto colpito dalla luce: se la normale punta verso

la luce è possibile impostare un colore, mentre se punta nella direzione opposta, se ne punta un altro;

DirectionalLight: rappresenta una luce direzionale, che meglio rappresenta quella che è la luce emessa dal sole: la sorgente della fonte di luce viene considerata estremamente distante dalla scena e la direzione dagli oggetti al sole è la stessa per ogni oggetto presente nella scena.

```
hemiLight = new THREE.HemisphereLight( 0xffffff, 0xffffff, 0.6 );
hemiLight.color.setHSL( 0.6, 1, 0.6 );
hemiLight.groundColor.setHSL( 0.095, 1, 0.75 );
hemiLight.position.set( 0, 1000, 0 );

scene.add( hemiLight );

dirLight = new THREE.DirectionalLight( 0xffffff, 1 );
dirLight.color.setHSL( 0.1, 1, 0.95 );
dirLight.position.set( 200, 1000, 250 );

dirLight.castShadow = true;

dirLight.shadowCameraNear = 3;
dirLight.shadowCameraFar = camera.far;
dirLight.shadowCameraFov = 50;

dirLight.shadowMapBias = -0.001;
dirLight.shadowMapDarkness = 0.5;
dirLight.shadowMapWidth = 2048;
dirLight.shadowMapHeight = 2048;

scene.add( dirLight );
```

Algoritmo 6.1: Illuminazioni scena

6.2 Strade

La realizzazione delle strade è basata su codice e ragionamenti simili a quelli visti finora riguardo gli edifici, ma con qualche variazione.

I dati utilizzati sono composti, oltre che dai punti che formano lo shape, anche dai punti che rappresentano la mezzeria della strada, ovvero la linea che divide la strada lungo la sua lunghezza. Grazie a quest'ultimo parametro è possibile riuscire a determinare i lati esterni della strada, sui quali verranno poi rappresentate le linee stradali, anche in casi di strade non regolari (si pensi a strade con piazzole di sosta o simili). Questi elementi verranno poi passati

allo shader, il quale si occuperà di rappresentare asfalto e linee, senza l'uso di texture ma semplicemente usando le funzionalità del Fragment Shader.

6.2.1 Vertex e Fragment Shader

Per quanto concerne il Vertex Shader non ci sono variazioni e lo schema è esattamente uguale a quello mostrato. Discorso diverso avviene per il Fragment Shader, nel quale la geometria viene così dipinta:

asfalto: lo shader colora tutta la geometria con un colore grigio asfalto. Per non renderlo troppo uniforme e dargli un'idea di realismo un po' maggiore, al colore viene aggiunto del rumore, una procedura usata nell'ambito delle texture procedurali: per ogni punto viene generato un valore casuale entro un certo range che viene moltiplicato per il colore;

linee stradali: per semplicità sono state considerate come linee solo quelle ai bordi della strada e vengono colorate tutti i punti a una certa distanza dai bordi della strada, i quali sono passati in un array al fragment tramite uniform. Ogni punto viene colorato sfruttando la funzione *mix*, presente in GLSL che permette di interpolare due array *vec4* (che vengono usati per definire il colore del punto) in base ai valori del parametro *alpha*.

```
varying vec2 vUv;
varying vec3 vPos;

void main() {

    vPos = position;
    vUv = uv;
    gl_Position = projectionMatrix * modelViewMatrix * vec4(
        position, 1.0 );
}
```

Algoritmo 6.2: Vertex Shader della strada

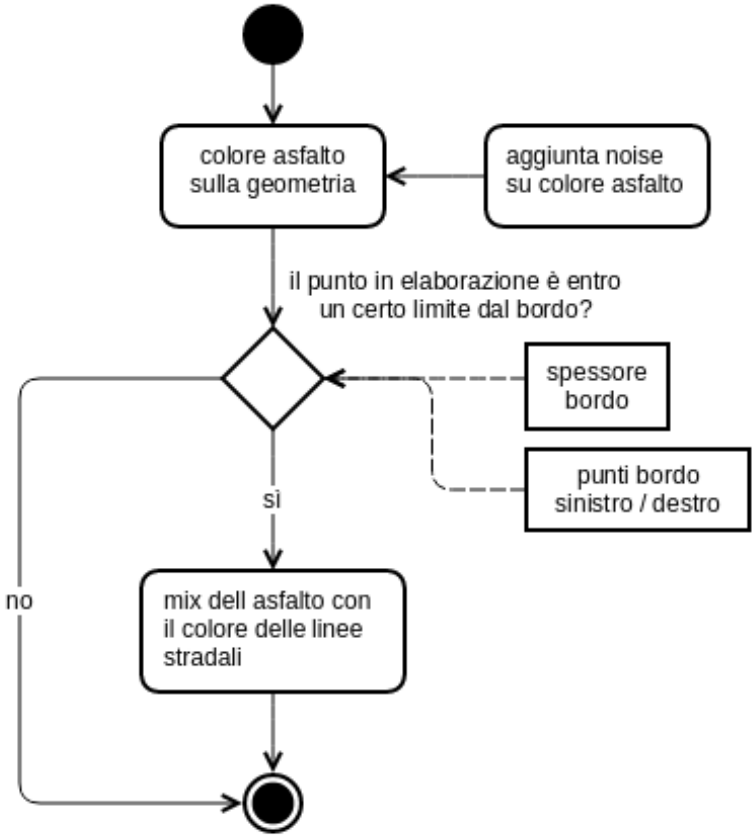

Figura 6.1: UML Activity del Fragment Shader della strada

```
vec4 roadTex = vec4(0.15, 0.15, 0.15, 1.0); // COLORE ASFALTO
vec3 roadN = roadTex.rgb * (noise(vPos.xy));
gl_FragColor.rgb = roadN;
gl_FragColor.a = roadTex.a;

vec4 extLines = vec4(1.0, 1.0, 1.0, 0.6); // COLORE LINEE BORDO
vec3 ce = roadN * (1.0 - extLines.a) + extLines.rgb * extLines.a
    * roadTex.a;

for(int i = 0; i < LEFT_LENGTH-1; i++){

    float angle = abs(cos(degrees(atan(uLeftLine[i+1].y -
        uLeftLine[i].y, uLeftLine[i+1].x - uLeftLine[i].x))) /
        180.0);
    float step;
    if(uLeftLine[i].x == uLeftLine[i+1].x || uLeftLine[i].y ==
        uLeftLine[i+1].y)
        step = LINE_EXT_WIDTH;
    else
        step = LINE_EXT_WIDTH / sqrt(2.0) * angle;

    if((distanceToLine(uLeftLine[i], uLeftLine[i+1], vPos.xy) >=
        float(LINE_DIST))
        && (distanceToLine(uLeftLine[i], uLeftLine[i+1], vPos.xy)
            <= float(LINE_DIST) + step)
        ){
        gl_FragColor = vec4(ce, 1.0);
    }
}
```

Algoritmo 6.3: Fragment Shader della strada: colorazione strada e linee stradali (il ciclo è identico anche per le linee di bordo di destra, per ragioni di spazio si mostra solo il ciclo che disegna la linea sinistra)

6.3 Alberi

Gli alberi sono un'ulteriore struttura dedicata all'abbellimento della scena e i modelli di questi viene rappresentato e creato tramite un applicativo esterno che è Blender. Il modello creato verrà poi importato in formato compatibile con Three.js in modo da avere una geometria complessa già pronta alle quali verrà poi rappresentata con due texture: una per il tronco e una per le foglie.

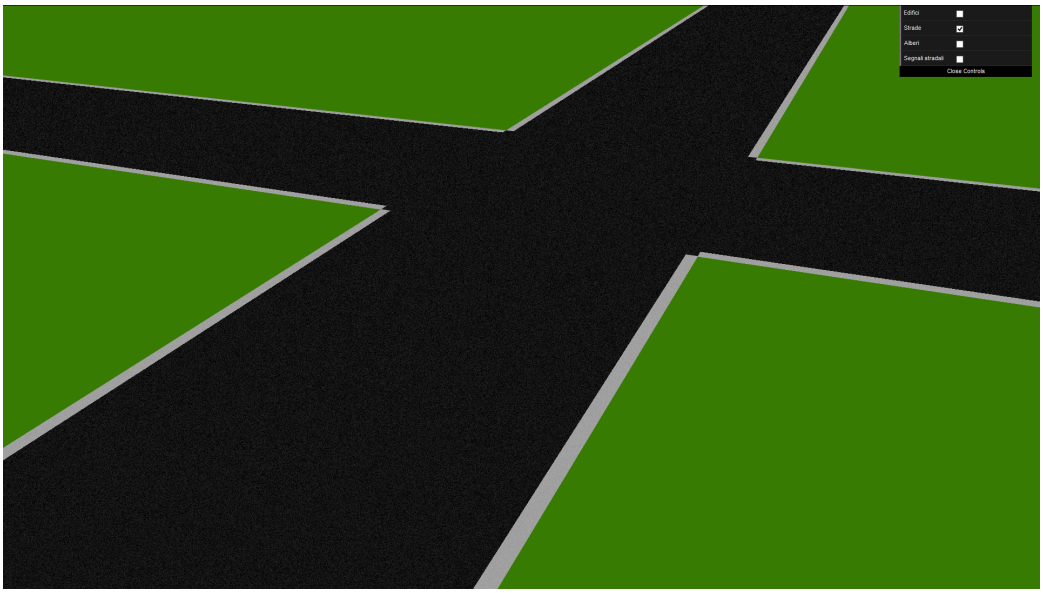

Figura 6.2: Esempio di una strada

6.3.1 Blender

Blender è un applicativo gratuito e open source specializzato nella modellazione e animazione 3D, rilasciato con licenza GPL. Viene usato per effettuare rendering, animazioni e per realizzare modelli anche in ambiti video-ludici e permette uno sviluppo su più piattaforme quali Linux, Windows, OSX, ecc.

Figura 6.3: Logo Blender

Tra le funzionalità di Blender vi è anche l'utilizzo di script in Python e uno di questi, chiamata *Sapling* (in inglese vuol dire alberello), permette di creare modelli di alberi molto complessi e realistici grazie a svariati parametri da regolare a scelta, quali arborescenza, lunghezza rami, ecc.

Una volta realizzato il modello e regolato la mappatura uv delle varie foglie, Three.js offre uno strumento di esportazione apposta per Blender che permette di esportare ogni geometria realizzata in un file *json*, facilmente integrabile nel progetto.

6.3.2 Importazione modello

Con il file *json* creato appositamente, è possibile importare la geometria nell'applicativo in un *Object3D*, un oggetto base della libreria Three.js che

Figura 6.4: Esempio di un albero

permette di legare in unico oggetto più geometrie e di visualizzarle come se fossero un'unica Mesh. Nell'oggetto infatti sono presenti due geometrie: una per il tronco e una per le foglie; in questo modo possiamo, una volta importato correttamente l'oggetto, assegnare a entrambe le geometrie una texture appropriata per poi visualizzare l'oggetto nella scena. Per semplicità è stato realizzato un singolo modello e per renderlo un po' più vario, ogni oggetto che viene aggiunto alla scena viene ruotato e scalato in altezza di un valore casuale. Alla scena verranno realizzati e aggiunti alla scena un numero di alberi basato sui dati disponibili, e per ognuno di essi verrà indicata la posizione in cui collocare l'oggetto nella scena.

```
var loader = new THREE.ObjectLoader();
loader.load('models/tree.json', function ( object ) {
  var o = object.children[0];
    for (var i = 0; i < dataTrees.length; i++) {
      var dt = dataTrees[i];
    var obj = setupTree(o.clone(), dt); //TEXTURE E
        POSIZIONAMENTO
    layerTrees.add(obj);
  };
  scene.add(layerTrees);
});
```

Algoritmo 6.4: Procedura di importazione modello albero

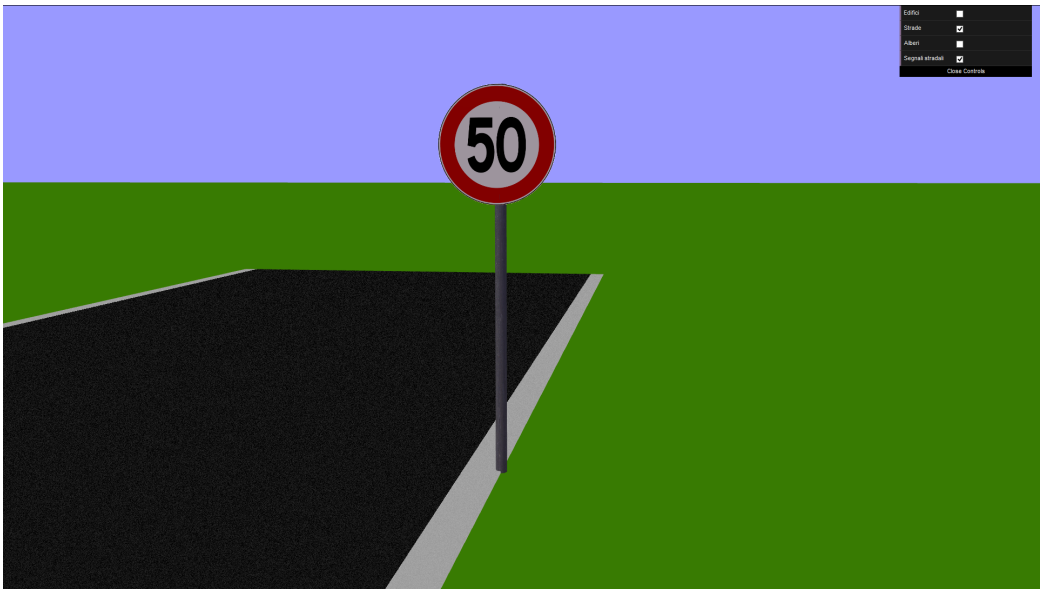

Figura 6.5: Esempio di un segnale stradale

6.4 Segnali stradali

I segnali stradali rappresentano un'ulteriore elemento scenico per rendere più realistica la scena. Dal punto di vista delle classi utilizzate, la procedura utilizzata sarà simile a quella vista prima per gli alberi, con la differenza che non si creerà un modello utilizzando applicativi esterni, ma si utilizzeranno gli strumenti messi a disposizione di Three.js. Un segnale stradale sarà composto da due oggetti:

sostegno: ovvero il palo che sorreggerà il segnale, verrà realizzato usando la classe *CylinderGeometry*, come parametri avrà l'altezza e il diametro del cilindro e avrà una texture metallica;

segnale: verrà rappresentato con *CircleGeometry*, sul quale verrà rappresentato il segnale stradale solo su un lato e verrà posizionato in base all'altezza impostata sul sostegno e il diametro del segnale stesso.

Questi due elementi verranno aggiunti integrati in unico *Object3D* di modo che possano apparire come un'unica Mesh che verrà poi aggiunta alla scena. Il numero di segnali che verranno creati si baserà sui dati disponibili per ognuno che comprenderanno la posizione del segnale e l'angolo di rotazione del segnale.

Figura 6.6: Screenshot della scena con tutti gli elementi presentati (1)

Figura 6.7: Screenshot della scena con tutti gli elementi presentati (2)

Capitolo 7

Conclusioni e possibili miglioramenti

Il progetto che è stato proposto mostra un applicativo web funzionante e operativo con i dati disponibili che, pur nella loro semplicità, hanno rappresentato la base per definire quali possono essere le caratteristiche fondamentali per rappresentare in maniera soddisfacente un ambito urbano. Gli obiettivi prefissati durante lo sviluppo e progettazione del sistema trattato sono stati quindi raggiunti in maniera soddisfacente, realizzando in real-time tutte le strutture previste nel database e trattate nei capitoli precedenti, lasciando al tempo stesso spazio per possibili ulteriori miglioramenti.

Le potenzialità offerte da una libreria grafica 3D come Three.js sono molte e in costante aumento e le funzionalità già presenti vengono sempre più ottimizzate per rendere sempre più fluido e particolareggiato ogni sistema web 3D che utilizzi tale libreria. La quantità di dati è, per ragioni di semplicità bassa e di certo inferiore alla quantità di dati che fornirebbe una complessa sezione urbana. Di conseguenza, in preparazione a una mole di dati che sarà sicuramente maggiore dei dati utilizzati, si renderà necessario effettuare una migliore ottimizzazione del codice di rendering, così da rendere maggiormente scalabile il carico di lavoro dell'applicativo in base all'hardware sul quale viene eseguito e sulla quantità di dati disponibili in base all'area urbana in esame. Oggetto di miglioramento sarebbero sia il codice Javascript per la realizzazione e implementazione di ogni oggetto presente nei dati, che il codice GLSL degli shader per ridurre il carico di lavoro di rendering sulla GPU.

Altro possibile miglioramento potrebbe riguardare la scena, che non essendo di importanza fondamentale per il progetto, è molto semplice; si potrebbe rendere il terreno più realistico, generato con parametri casuali oppure con dati reali legati a una particolare area geografica scelta. Anche l'illumina-

zione e le ombre potrebbero essere oggetto di miglioramenti, sia a livello di dettaglio che da un punto di vista di colore emessa dalla luce.

L'aggiunta di un maggior numero di caratteristiche e texture per oggetto può rendere più vario e personalizzabile ogni modello della scena, in particolar modo per gli edifici: si possono prevedere più tipologie di edificio e in base a quelle variare le texture per quel particolare edificio, oppure in base ad ulteriori parametri è possibile ridimensionare altri elementi quali finestre e porte, distanziarli o avvicinarli tra loro oppure anche ridimensionarli.

Elenco delle figure

2.1 Schema di una pipeline 7
2.2 Mappatura UV 7

3.1 Logo OpenGL 9
3.2 Logo WebGL 10
3.3 Logo Three.js 10

4.1 UML Activity dell'architettura 12

5.1 UML Activity 16
5.2 Una geometria prima e dopo l'applicazione del bevel 18
5.3 a) normali standard b) normali dopo aver applicato il bevel, parallele alla faccia di appartenenza 19
5.4 UML Activity del Fragment Shader di un edificio 21
5.5 Fasi dell'algoritmo Straight Skeleton su una geometria 24
5.6 Esempio di edificio 25

6.1 UML Activity del Fragment Shader della strada 30
6.2 Esempio di una strada 32
6.3 Logo Blender 32
6.4 Esempio di un albero 33
6.5 Esempio di un segnale stradale 34
6.6 Screenshot della scena con tutti gli elementi presentati (1) . . 35
6.7 Screenshot della scena con tutti gli elementi presentati (2) . . 35

Elenco degli Algoritmi

5.1 Vertex Shader di un edificio 21
5.2 Disposizione texture porte 22
5.3 Disposizione texture finestre 23
6.1 Illuminazioni scena 28
6.2 Vertex Shader della strada 29
6.3 Fragment Shader della strada: colorazione strada e linee stradali (il ciclo è identico anche per le linee di bordo di destra, per ragioni di spazio si mostra solo il ciclo che disegna la linea sinistra) 31
6.4 Procedura di importazione modello albero 33

Bibliografia

[1] D. Shreiner, G. Sellers, J. M. Kessenich, and B. M. Licea-Kane, *OpenGL Programming Guide: The Official Guide to Learning OpenGL, Version 4.3*. Paperback, eighth ed., 2013.

[2] T. Parisi, *WebGL: Up and Running*. O'Reilly Media, aug 2012.

[3] J. Dirksen, *Learning Three.js: The JavaScript 3D Library for WebGL*. Packt, oct 2013.

[4] D. S. Ebert, F. K. Musgrave, D. Peachey, K. Perlin, and S. Worley, *Texturing and Modeling, Third Edition: A Procedural Approach (The Morgan Kaufmann Series in Computer Graphics)*. Morgan Kaufmann, third ed., dec 2002.